KỸ THUẬT LẬP TRÌNH HỆ CƠ ĐIỆN TỬ

Programming Engineering in Mechatronics

Giảng viên: TS. Nguyễn Thành Hùng

Đơn vị: Bộ môn Cơ điện tử, Viện Cơ khí

Email: hung.nguyenthanh@hust.edu.vn

thanhhung.hust@gmail.com

Hà Nội, 2020

Tổng quan về ngôn ngữ lập trình

1. Giới thiệu học phần
2. Giới thiệu về ngôn ngữ lập trình

Giới thiệu học phần

- ❖ Kỹ thuật lập trình trong Cơ điện tử ME3213
- ❖ Khối lượng: 3(2-2-0-6)
  - ➢ Giờ giảng lý thuyết: 45 tiết.
  - ➢ Giờ bài tập, thảo luận: 15 tiết và bài tập lớn.

Mục tiêu học phần

- Môn học cung cấp cho sinh viên những kiến thức tổng quát về lập trình và kỹ thuật lập trình, đồng thời có các kĩ thuật cơ bản về ngôn ngữ lập trình C và C++ và phong cách lập trình hướng đối tượng.
- Phương pháp thiết kế giao diện đồ họa người dùng (GUI: Graphical User Interface)
- Lập trình giao tiếp với các thiết bị ngoại vi như camera, vi điều khiển, PLC, …
- Sinh viên có khả năng triển khai các chương trình cụ thể để giải quyết các bài toán kĩ thuật
- Sinh viên cũng có thể viết các chương trình chuyên dụng kết nối và điều khiển các thiết bị ngoại vi.

Nội dung môn học

- Giới thiệu môn học và Tổng quan về ngôn ngữ lập trình
- Chương 1: Cơ sở của C++
- Chương 2: Lập trình cấu trúc trong C++
- Chương 3: Lập trình hướng đối tượng trong C++
- Chương 4: Lập trình giao diện đồ họa người dùng
- Chương 5: Lập trình giao diện phần cứng

Sách và tài liệu tham khảo

❖ **Tài liệu học tập**:

1. Sách giáo trình: Péter Tamás, Antal Huba, József Gräff: Mechatronic Systems Programming in C++, BME MOGI, 2014
2. Bài giảng: Bài giảng Kỹ thuật lập trình trong Cơ điện tử (dạng slide bài giảng).
3. Công cụ: Visual Studio, QT, Dev C++, …

Sách và tài liệu tham khảo

Tài liệu tham khảo

1. Brain W.Kernighan, Dennis M.Ritchie: *The C Programming Language,* Second Edition, Prentice Hall, 1988.
2. Bjarne Stroustrup: *The C++ Programming Language*, Third Edition, AT&T, 1997.
3. Stephan C. Dewhurst: C++ Gotchas: Avoiding Common Problems in Coding and Design, Addison Wesley, 2002.
4. H. M. Deitel: *C++ How to program*, Fifth Edition, Prentice Hall, 2005.
5. Ivor Horton: *Beginning Visual C++ 2005*, Wiley Publishing, Inc, 2006.
6. Shaharuddin Salleh, Albert Y. Zomaya, Sakhinah Abu Bakar: *Computing For Numerical Methods Using Visual C++*, John Wiley and Sons, Inc, 2008.

Đánh giá kết quả

❖ **Điểm quá trình: hệ số 0,3**

- Chuyên cần: tham gia đầy đủ các buổi học
- Bài tập: nộp đầy đủ các bài tập
- Kiểm tra giữa kỳ

❖ **Điểm cuối kỳ: hệ số 0,7**

- Hoàn thành bài tập lớn: Sinh viên cần viết một ứng dụng có thể kết nối với một thiết bị phần cứng trong thời gian thực.

Tổng quan về ngôn ngữ lập trình

1. Giới thiệu học phần
2. Giới thiệu về ngôn ngữ lập trình

Tổng quan về ngôn ngữ lập trình

1. Giới thiệu

- Máy tính, phần cứng, phần mềm
- Các mức của ngôn ngữ lập trình
- Ngôn ngữ bậc cao và C++
- Các thành phần cơ bản của chương trình C++
- Các kiểu dữ liệu cơ bản trong C++
- Các bước giải bài toán
- Các loại lỗi và xử lý lỗi

Tổng quan về ngôn ngữ lập trình

1.1. Máy tính:

- Máy tính (máy vi tính hay máy điện toán) là thiết bị hay hệ thống được dùng để tính toán hay kiểm soát các hoạt động mà có thể biểu diễn dưới dạng số hay quy luật lôgic.

1.2. Thiết bị phần cứng:

- Phần cứng (hardware) là các thành phần cụ thể của máy tính có thể chạm vào được như màn hình, chuột, bàn phím, máy in, máy quét, vỏ máy tính, đơn vị vi xử lý CPU, bo mạch chủ, các loại dây nối, loa, ổ mềm, ổ cứng, ổ CDROM, ...

Tổng quan về ngôn ngữ lập trình

1.2. Thiết bị phần cứng:

- Phân loại:

  - Thiết bị nhập (Input): Các bộ phận thu nhập dữ liệu hay mệnh lệnh như là bàn phím, chuột...

  - Thiết bị xuất (Output): Các bộ phận trả lời, phát tín hiệu, hay thực thi lệnh ra bên ngoài như là màn hình, máy in, loa, ...

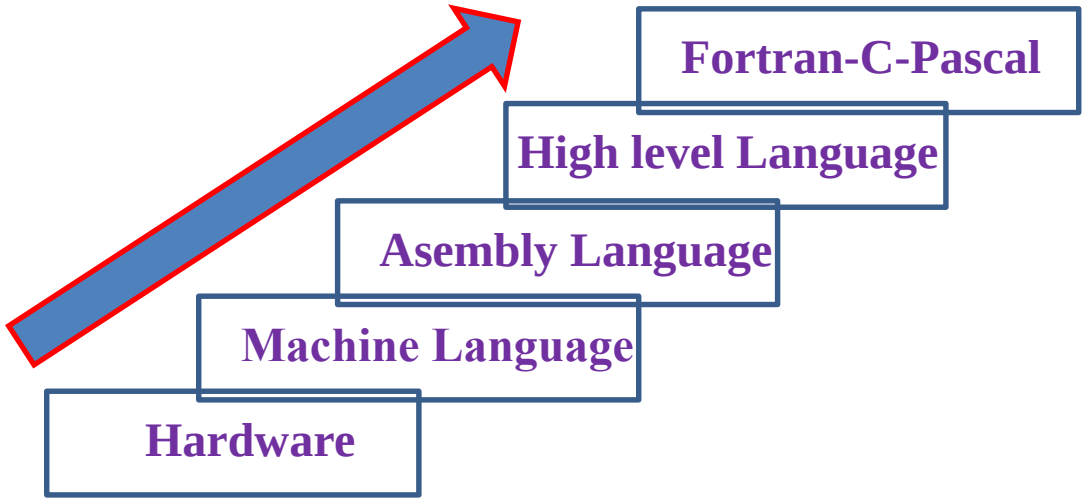
Tổng quan về ngôn ngữ lập trình

1.3. Phần mềm máy tính:

- Phần mềm (Software) là một tập hợp những câu lệnh được viết bằng một hoặc nhiều ngôn ngữ lập trình theo một trật tự xác định nhằm tự động thực hiện một số chức năng hoặc giải quyết một bài toán nào đó.
- Ngôn ngữ: Ngôn ngữ trong máy tính là một công cụ để thực hiện việc giao tiếp giữa người và máy.
- Lệnh: Lệnh là tập hợp một nhóm các ký hiệu của một ngôn ngữ nào đó nhằm giúp cho người lập trình có thể xây dựng chương trình trên ngôn ngữ đó.

Tổng quan về ngôn ngữ lập trình

- Các mức của ngôn ngữ lập trình:



Tổng quan về ngôn ngữ lập trình

- Ngôn ngữ được thiết kế và chuẩn hóa (từ khóa và cú pháp) để truyền các chỉ thị cho máy tính.

- Dùng để tạo ra các chương trình điều khiển máy tính hoặc mô tả các thuật toán.

- Ngôn ngữ máy: là ngôn ngữ duy nhất máy trực tiếp hiểu được và thực hiện. Dựa trên đại số Boolean với 2 giá trị mức logic 0, 1. Chương trình viết bằng ngôn ngữ máy có thể nạp trực tiếp vào bộ nhớ thi hành ngay.

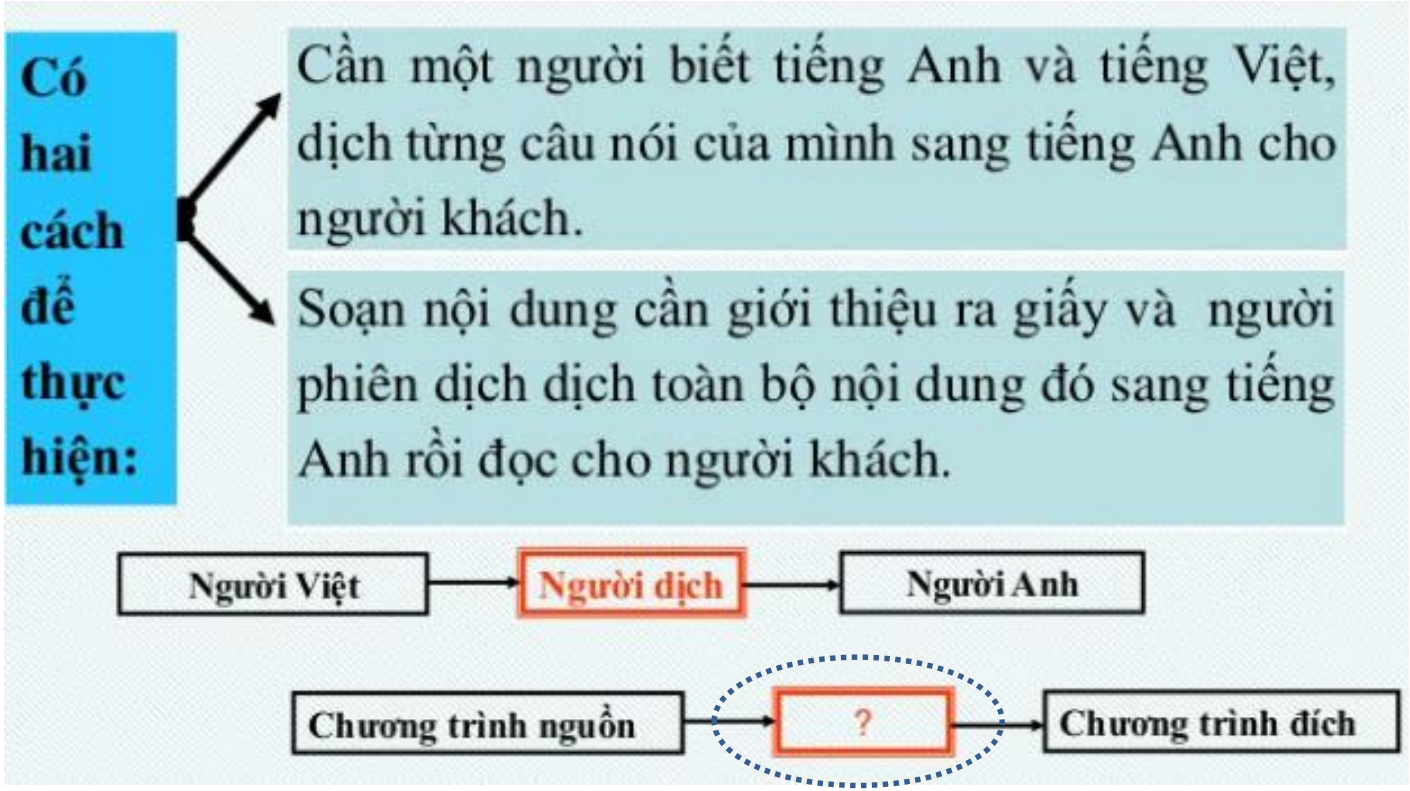
Tổng quan về ngôn ngữ lập trình

- Hợp ngữ: Là ngôn ngữ rất gần với ngôn ngữ máy, nhưng mã lệnh được thay bằng tên viết tắt của thao tác (Tiếng Anh). Hợp ngữ cần chương trình

- Ngôn ngữ bậc cao: Là ngôn ngữ gần với ngôn ngữ tự nhiên. Chương trình viết bằng ngôn ngữ bậc cao không phụ thuộc máy, muốn thi hành được cần chuyển sang ngôn ngữ máy (trình biên dịch)

Tổng quan về ngôn ngữ lập trình

- Giao tiếp người - máy: Các phần mềm thiết kế, chương trình gia công, điều khiển được mã hóa thông qua phần mềm biên dịch tương ứng với máy, để máy có thể hiểu và thực hiện.
- Ví dụ:



Trình biên dịch

Tổng quan về ngôn ngữ lập trình

- **Chương trình dịch** là chương trình đặc biệt có chức năng chuyển đổi chương trình được viết bằng ngôn ngữ lập trình bậc cao thành chương trình thực hiện được trên máy tính.

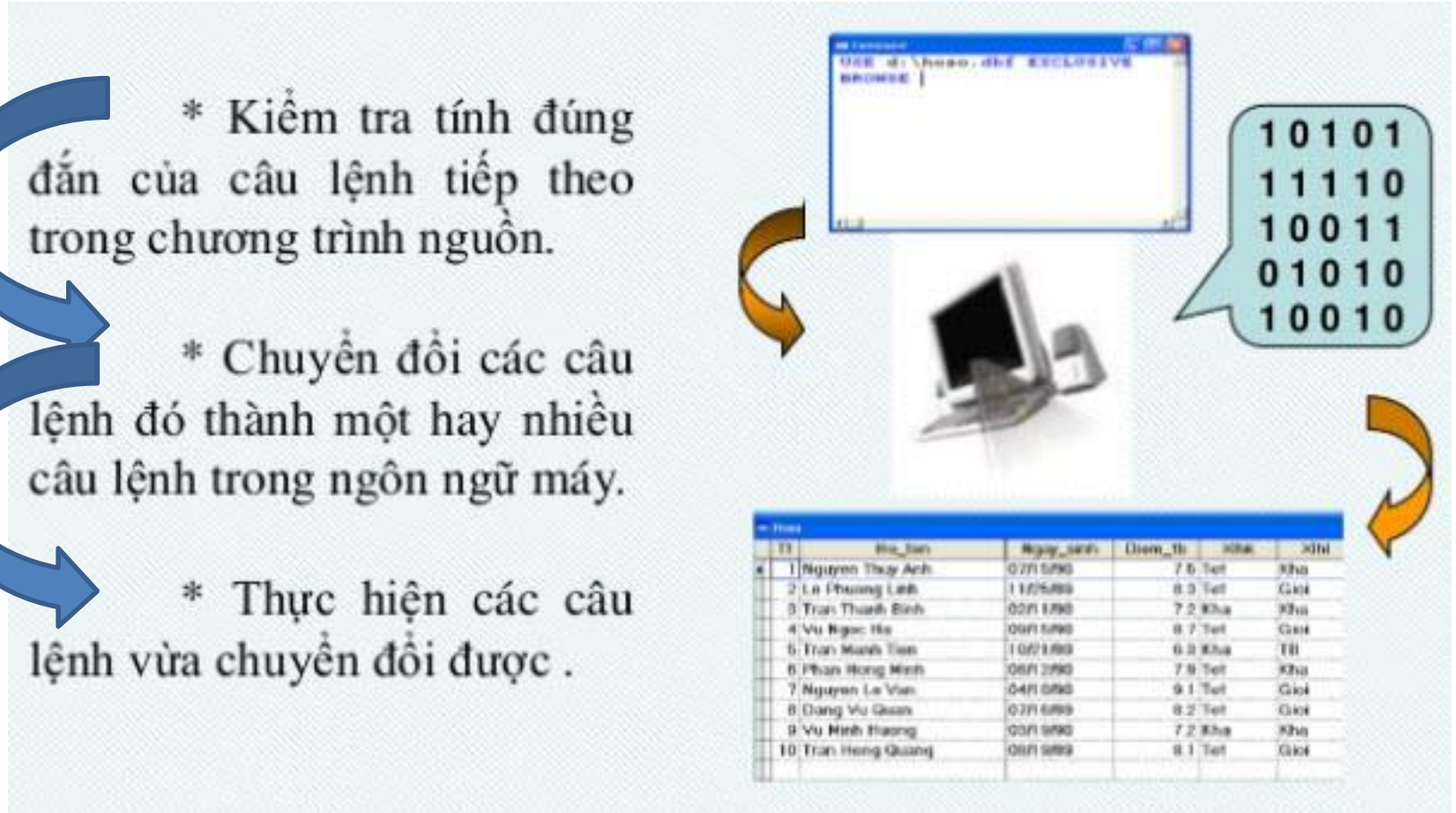
Chương trình dịch có mấy loại?

Answer

Chương trình dịch có 2 loại là: *thông dịch* và *biên dịch*

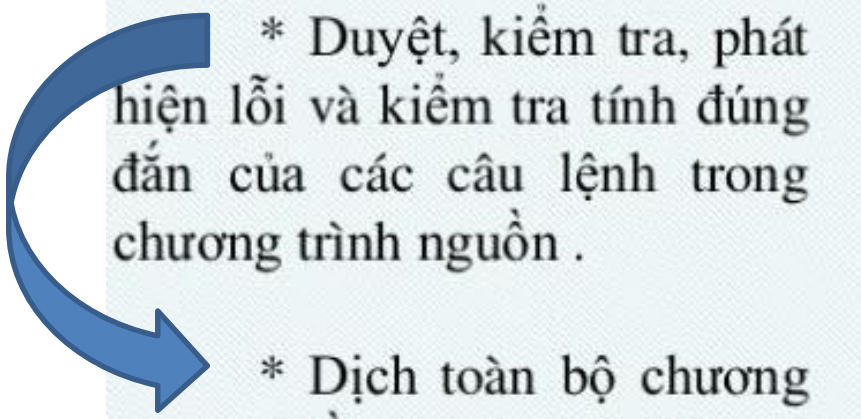
Tổng quan về ngôn ngữ lập trình

- Thông dịch (Interpreter): Thông dịch được thực hiện theo các bước sau



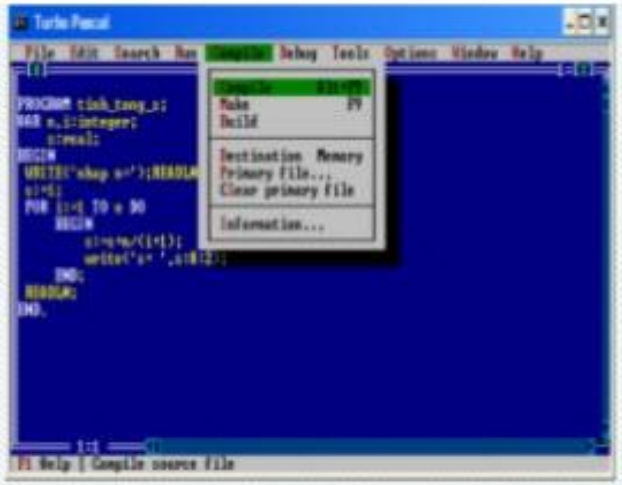
Tổng quan về ngôn ngữ lập trình

- Biên dịch (Complier): Thực hiện các bước sau:



* Duyệt, kiểm tra, phát hiện lỗi và kiểm tra tính đúng đắn của các câu lệnh trong chương trình nguồn .

* Dịch toàn bộ chương trình nguồn thành một chương trình đích (ngôn ngữ máy) để có thể thực hiện trên máy và có thể lưu trữ để sử dụng lại khi cần.



- Biên dịch: Kiểm tra và dịch toàn bộ, chuyển thành ngôn ngữ máy và có thể lưu trữ lại trong khi thông dịch là sử dụng trực tiếp cho máy

Tổng quan về ngôn ngữ lập trình

1.4. Thành phần cơ bản của ngôn ngữ lập trình:

Tiếng Việt có các thành phần cơ bản nào?

Các thành phần cơ bản của ngôn ngữ tiếng Việt là:

- Bảng chữ cái tiếng Việt, số, dấu..
- Cách ghép các ký tự thành từ, ghép từ thành câu.
- Ngữ nghĩa của từ và câu.

Trong ngôn ngữ lập trình thì sao?

Trong ngôn ngữ lập trình cũng tương tự như vậy, nó gồm có các thành phần cơ bản: Bảng chữ cái, cú pháp và ngữ nghĩa.

Tổng quan về ngôn ngữ lập trình

- **Bảng chữ cái:** là tập các kí tự được dùng để viết chương trình.

Bảng chữ cái tiếng Anh:

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

a b c d e f g h i j k l m n o p q r s t u v w x y z

Hệ đếm: 0 1 2 3 4 5 6 7 8 9

Ký hiệu đặc biệt: + - * / = < > [ ] . , _ ; # ^ $ & ( ) { } : ‘

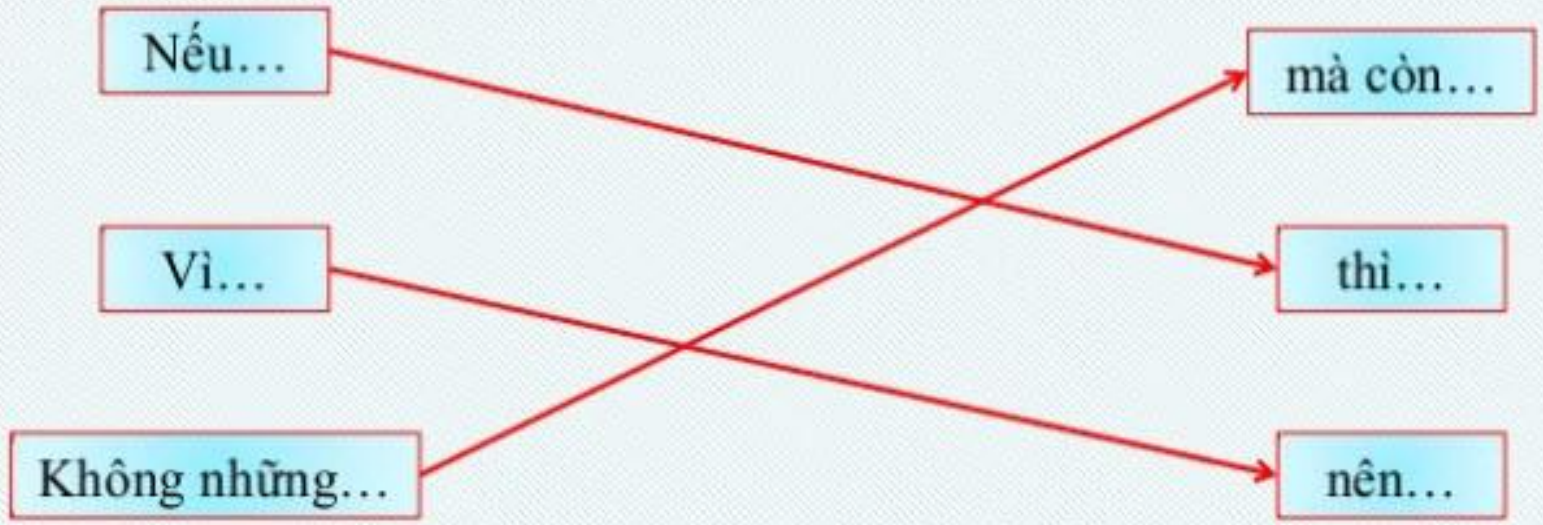
Các ngôn ngữ lập trình khác nhau cũng có sự *khác nhau* về bảng chữ cái.

Ví dụ: Bảng chữ cái trong ngôn ngữ lập trình *C/C++* so với *Pascal* có bổ sung thêm một số kí tự như: “ \ ! ? % |

Tổng quan về ngôn ngữ lập trình

- **Cú pháp:** là bộ quy tắc để viết chương trình.

Ghép các cặp từ sau đây sao cho phù hợp với quy tắc sử dụng trong tiếng Việt:



Tổng quan về ngôn ngữ lập trình

- **Ngữ nghĩa:** xác định ý nghĩa thao tác cần phải thực hiện ứng với tổ hợp kí tự dựa vào ngữ cảnh của nó.

Xác định ý nghĩa của kí tự + trong 2 trường hợp sau:

A + B	A + B
Với A, B là 2 đại lượng nhận giá trị số nguyên.	Với A, B là 2 đại lượng nhận giá trị số thực.
Kí tự + là phép cộng 2 số nguyên.	Kí tự + là phép cộng 2 số thực.

Tổng quan về ngôn ngữ lập trình

1.5. Phương pháp lập trình

- **Lập trình tuyến tính (tuần tự từ trên xuống)**

  - Chương trình chỉ gồm hàm chính và dữ liệu

Lập trình tuyến tính :

- Còn gọi là lập trình phi cấu trúc
- Giải quyết các bài toán tương nhỏ, đối đơn giản

Đặc điểm:

- Chỉ gồm một chương trình chính
- Gồm một dãy tuần tự các câu lệnh
- Chương trình ngắn, ít hơn 100 dòng

Main function
+
Data

Tổng quan về ngôn ngữ lập trình

Nhược điểm:

- Không sử dụng lại được các đoạn mã
- Không có khả năng kiểm soát phạm vi truy xuất dữ liệu
- Mọi dữ liệu trong chương trình là toàn cục
- Dữ liệu có thể bị sửa đổi ở bất cứ vị trí nào trong chương trình

Không đáp ứng được việc triển khai phần mềm

Tổng quan về ngôn ngữ lập trình

- **Lập trình cấu trúc**

  Ra đời vào những năm 70:

  Chương trình được chia nhỏ thành chương trình con:

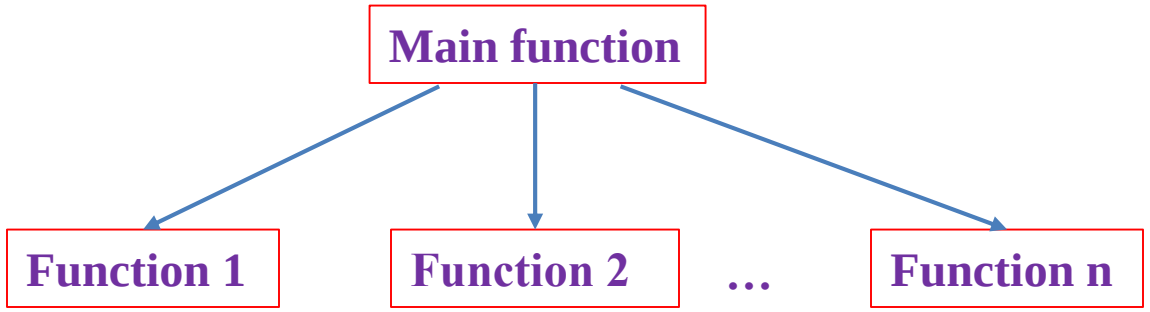
  - Thủ tục (Procedure)
  - Hàm (Function)

  - Chương trình chỉ gồm hàm chính (main function) và hàm con (sub-functions)

Tổng quan về ngôn ngữ lập trình

- Các chương trình con:
  - Độc lập với nhau và có dữ liệu riêng
  - Trao đổi qua: tham số và biến toàn cục

- Hàm con sau khi hoàn tất khai báo, có thể truy xuất, và gọi ra nhiều lần (tránh việc trùng lặp mã nguồn)
- Trao đổi dữ liệu giữa các hàm nhờ các tham số



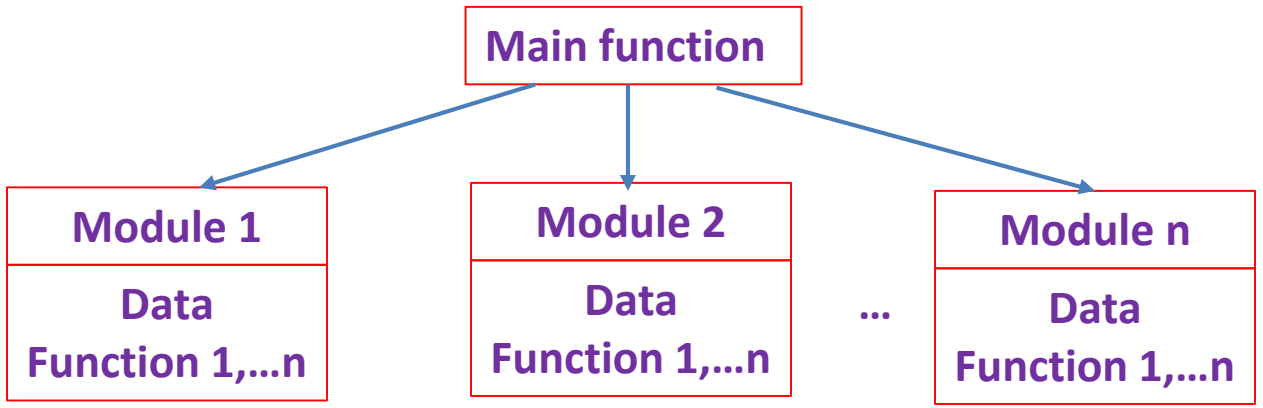
Tổng quan về ngôn ngữ lập trình

- **Lập trình module**

  - Các hàm được xây dựng và đóng gói trong các thư viện độc lập - module (dll, lib)

  - Các chương trình khác nhau có thể đồng thời cùng truy xuất được hàm trong thư viện

  - Ẩn và đóng gói dữ liệu cũng như triển khai bên trong



Tổng quan về ngôn ngữ lập trình

- Nhược điểm:
  - Chương trình khó kiểm soát
  - Khó khăn trong việc bổ sung, nâng cấp chương trình
  - Khi thay đổi, bổ sung dữ liệu dùng chung thì phải thay đổi gần như tất cả thủ tục/hàm liên quan
  - Khả năng sử dụng lại các đoạn mã chưa nhiều
  - Không mô tả đầy đủ, trung thực hệ thống trong thực tế

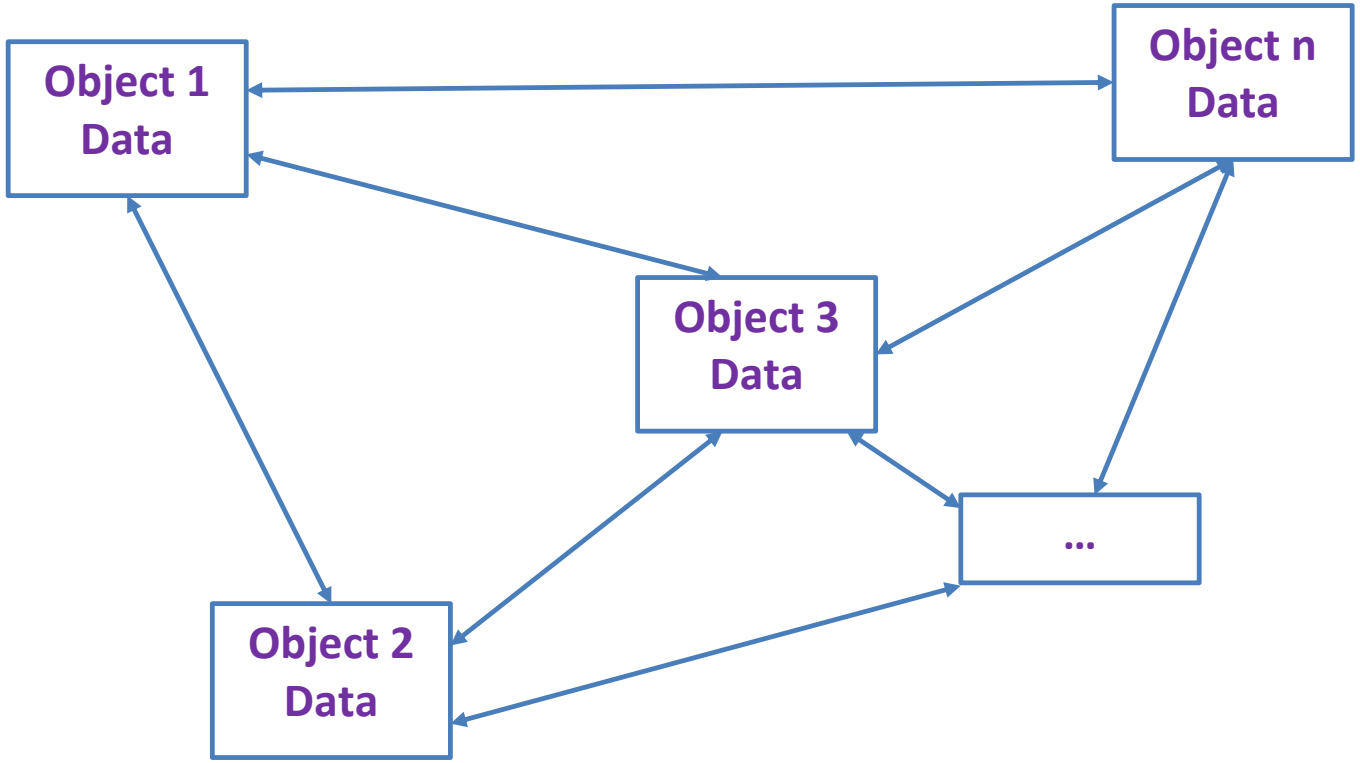
Tổng quan về ngôn ngữ lập trình

- **Lập trình hướng đối tượng**

Là phương pháp lập trình:

- Mô tả chính xác các đối tượng trong thế giới
- Lấy đối tượng làm nền tảng xây dựng thuật toán
- Thiết kế xoay quanh dữ liệu của hệ thống
- Chương trình được chia thành các lớp đối tượng
- Dữ liệu được đóng gói, che dấu và bảo vệ
- Đối tượng làm việc với nhau qua thông báo
- Chương trình được thiết kết theo cách từ dưới lên (bottom-up)

Tổng quan về ngôn ngữ lập trình



- Dữ liệu được trừu tượng hóa và triển khai thành lớp.

- Sử dụng lớp để tạo ra các đối tượng.

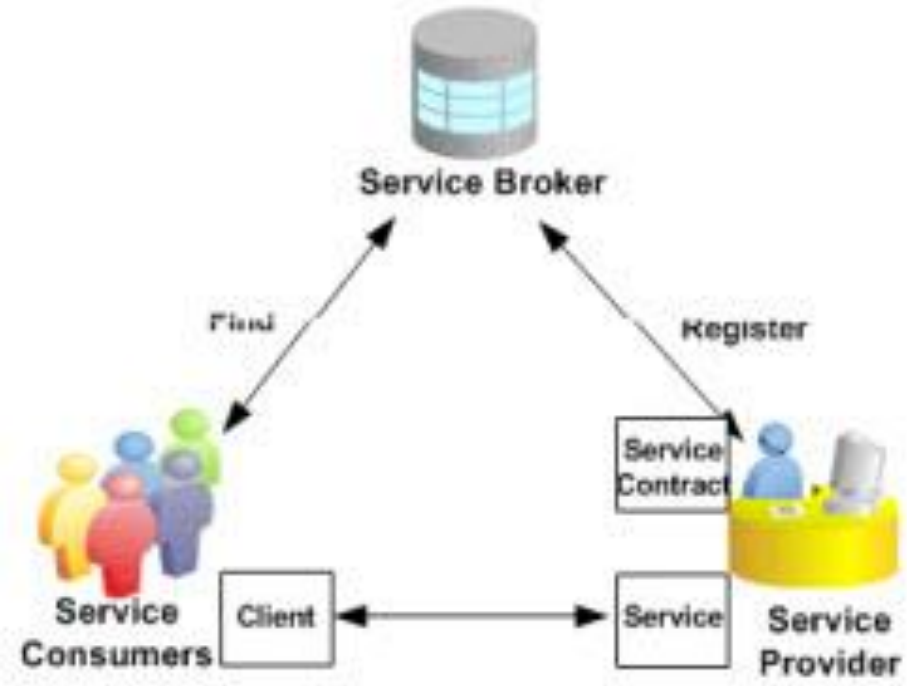
- Các đối tượng sử dụng thông điệp để trao đổi với nhau.

Tổng quan về ngôn ngữ lập trình

- **Lập trình hướng dịch vụ (SOA - Service Oriented Architecture)**

- Dịch vụ được cung cấp cho các ứng dụng khác qua giao thức truyền thông, chủ yếu là qua mạng.

- Quy tắc hướng dịch vụ là độc lập với bất cứ nhà cung cấp, sản phẩm hay công nghệ.



TRƯỜNG ĐẠI HỌC BÁCH KHOA HÀ NỘI

KỸ THUẬT LẬP TRÌNH HỆ CƠ ĐIỆN TỬ

Programming for Mechatronic Systems

Giảng viên: TS. Nguyễn Thành Hùng

Đơn vị: Bộ môn Cơ điện tử, Viện Cơ khí

Hà Nội, 2020

Chapter I. Basics and data management of C++

- 1. Creation of C++ programs
- 2. Basic data types, variables and constants
- 3. Basic operations and expressions
- 4. Control program structures
- 5. Exception handling
- 6. Pointers, references and dynamic memory management
- 7. Arrays and strings
- 8. User-defined data types

1. Creation of C++ programs

❖ Some important rules

- The basic elements of the program: the characters of the 7 bit ASCII code table
- Character and text constants, as well as remarks may contain characters of any coding

```
/* Value is given for an integer, a character and a text
   (string)  variable (multiline remark) */
int variable = 12.23;  // value giving (remark until the
                       // end of the line)
char sign = 'Á';
string header  = "Programming is fun";
```

1. Creation of C++ programs

❖ Some important rules

- C++ compiler differentiates small and capital letters in the words (names) used in the program.
- Certain (English) words cannot be used as own names since these are keywords of the compiler.
- In case of creating own names please note that they have to start with a letter (or underscore sign), and should contain letters, numbers or underscore signs in their other positions.

1. Creation of C++ programs

❖ The first C++ program in two versions

C

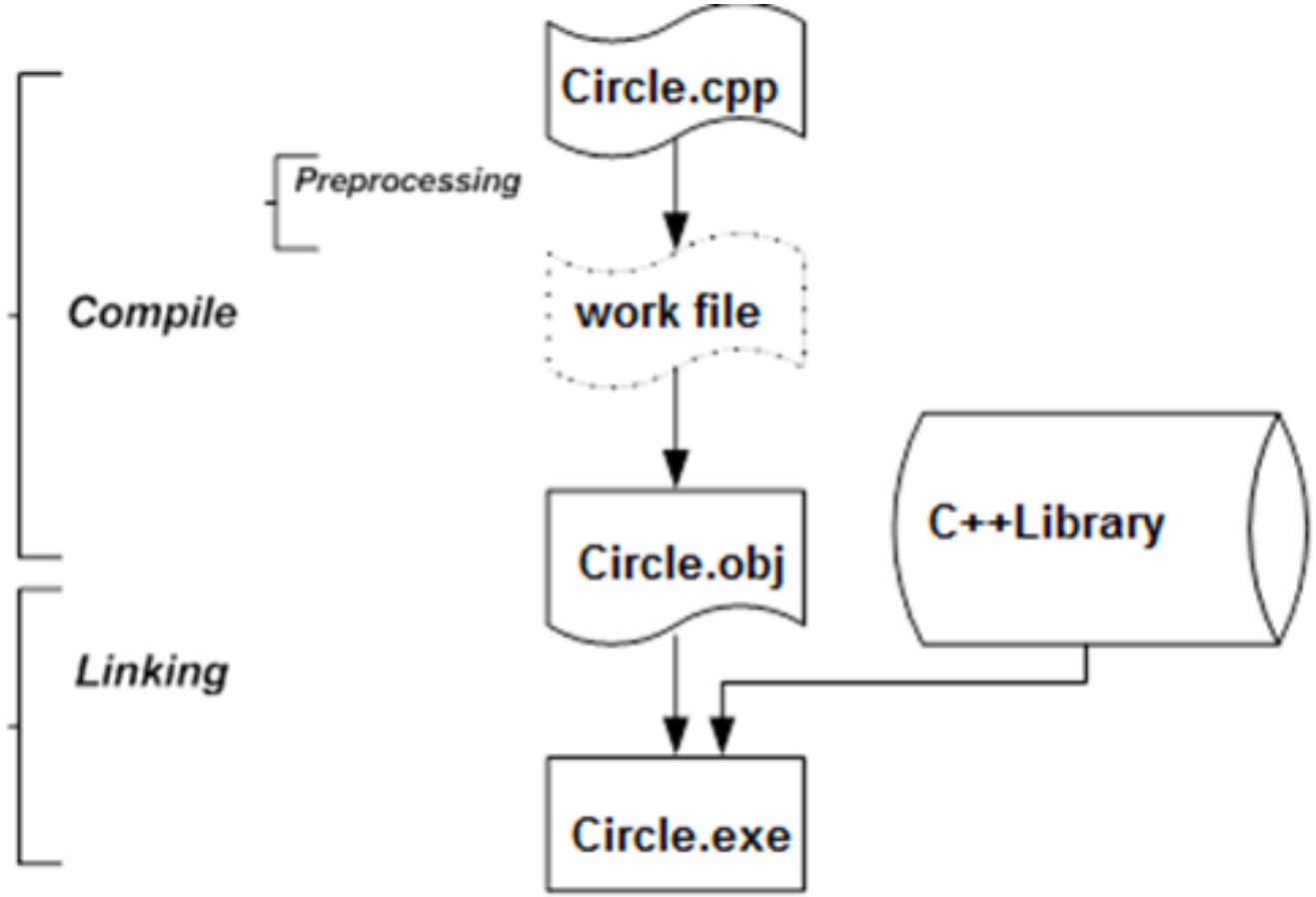
```c
// Circle1.cpp
#include "cstdio"
#include "cmath"
using namespace std;

int main()
{
     const double pi = 3.14159265359;
     double radius, area, perimeter;
     // Reading radius
     printf("Radius = ");
     scanf("%lf", &radius);
     // Calculations
     perimeter = 2 * radius*pi;
     area = pow(radius, 2)*pi;
     printf("Perimeter: %7.3f\n",
     perimeter);
     printf("Area: %7.3f\n", area);
     // Waiting for pressing Enter
     getchar();
     getchar();
     return 0;
}
```

C++

```cpp
// Circle2.cpp
#include "iostream"
#include "cmath"
using namespace std;

int main()
{
     const double pi = 3.14159265359;
     // Reading radius
     double radius;
     cout << "Radius = ";
     cin >> radius;
     // Calculations
     double perimeter = 2 * radius*pi;
     double area = pow(radius, 2)*pi;
     cout << "Perimeter: " << perimeter <<
     endl;
     cout << "Area: " << area << endl;
     // Waiting for pressing Enter
     cin.get();
     cin.get();
     return 0;
}
```

1. Creation of C++ programs

❖ Compilation and running of C++ programs



Steps of C++ program compilation

1. Creation of C++ programs

❖ Structure of C++ programs

```
// C++ preprocessor directives
#include <iostream>
#define MAX 2012

// in order to reach the standard library names
using namespace std;

// global declarations and definitions
double fv1(int, long);              // function prototype
const double pi = 3.14159265;       // definition

  // the main() function
int main()
{
/* local declarations and definitions
statements                                        */
return 0;  // exit the program
}
```

```
// function definition
double fv1(int a, long b)
{
/* local declarations and definitions
statements                                        */
return a + b; // return from the functions
}
```

1. Creation of C++ programs

❖ Structure of C++ programs

C++ object-oriented (OO) approach

```
/// Circle3.cpp
#include "iostream"
#include "cmath"
using namespace std;

// Class definition
class Circle
{
double radius;
static const double pi;
public:
Circle(double r) { radius = r; }
double Perimeter() { return 2 * radius*pi; }
double Area() { return pow(radius, 2)*pi; }
};
const double Circle::pi = 3.14159265359;
```

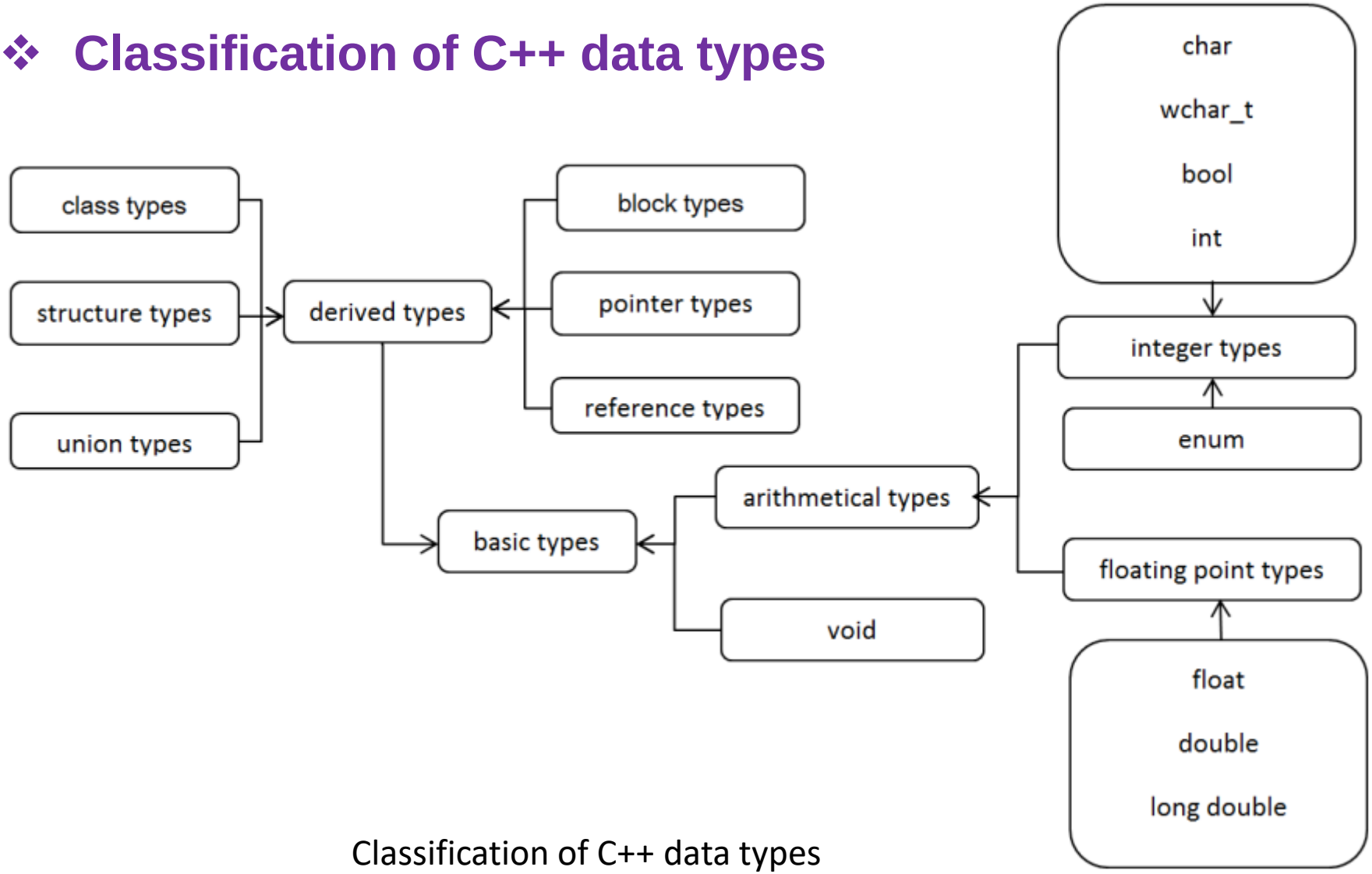
```
int main()
{
// Reading radius
double radius;
cout << "Radius = ";
cin >> radius;
// Creation and usage of object Circle
Circle circle(radius);
cout << "Perimeter: " <<
circle.Perimeter() << endl;
cout << "Area: " << circle.Area() <<
endl;
// Waiting for pressing Enter
cin.get();
cin.get();
return 0;
}
```

Chapter I. Basics and data management of C++

- 1. Creation of C++ programs
- 2. Basic data types, variables and constants
- 3. Basic operations and expressions
- 4. Control program structures
- 5. Exception handling
- 6. Pointers, references and dynamic memory management
- 7. Arrays and strings
- 8. User-defined data types

2. Basic data types, variables and constants

❖ Classification of C++ data types



Classification of C++ data types

2. Basic data types, variables and constants

❖ Type modifiers

- The **signed/unsigned** modifier pair: negative numbers or not.
- The **short/long** pair: size of the storage can be fixed to 16 or 32 bits.
- The **long long** modifier: 64bits
- Type modifiers can also be used as type definitions alone.

2. Basic data types, variables and constants

❖ Type modifiers

char	**signed** char		
short int	**short**	**signed short** int	**signed short**
int	**signed**	**signed** int	
long int	**long**	**signed long** int	**signed long**
long long int	**long long**	**signed long long** int	**signed long long**
unsigned char			
unsigned short int	**unsigned short**		
unsigned int	**unsigned**		
unsigned long int	**unsigned long**		
unsigned long long int	**unsigned long long**		

Elements in each row designate the same data type.

2. Basic data types, variables and constants

❖ Type modifiers

Data type	Range of values	Size (bytes)	Precision (digits)
bool	false, true	1	
char	-128..127	1	
signed char	-128..127	1	
unsigned char	0..255	1	
wchar_t	0..65535	2	
int	-2147483648..2147483647	4	
unsigned int	0..4294967295	4	
short	-32768..32767	2	
unsigned short	0..65535	2	

Data type	Range of values	Size (bytes)	Precision (digits)
long	-2147483648..2147483647	4	
unsigned long	0..4294967295	4	
long long	-9223372036854775808 .. 9223372036854775807	8	
unsigned long long	0..18446744073709551615	8	
float	3.4E-38..3.8E+38	4	6
double	1.7E-308..1.7E+308	8	15
long double	3.4E-4932..3.4E+4932	10	19

2. Basic data types, variables and constants

❖ Defining variables

- Generalized forms

⟨*storage class*⟩ ⟨*type qualifier*⟩ ⟨*type modifier ...* ⟩ *typevariable name* ⟨= *initial value*⟩ ⟨, … ⟩;

⟨*storage class*⟩ ⟨*type qualifier*⟩ ⟨*type modifier ...* ⟩ *typevariable name* ⟨(*initial value*)⟩ ⟨, … ⟩;

- the ⟨ ⟩ signs indicate optional elements while the three points show that a definition element can be repeated.
- The ***storage classes*** – **auto**, **register**, **static and extern** – of C++ determine the lifetime and visibility of variables.

2. Basic data types, variables and constants

❖ Defining variables

- With ***type qualifiers*** further information can be assigned to variables.

➢ Variables with **const** keyword cannot be modified (they are read-only, i.e. constants).

➢ The **volatile** type qualifier indicates that the value of the variable can be modified by a code independent of our program (e.g. by another running process or thread).

int const
const double
volatile char
float volatile
const volatile
bool

2. Basic data types, variables and constants

❖ Defining variables

- **Initial values of variables**

```
using namespace std;

int sum, product(1);

int main()
{
    int a, b=2012, c(2004);
    double d=12.23, e(b);
}
```

```
#include <cmath>
#include <cstdlib>
using namespace std;

double pi = 4.0*atan(1.0);   // π
int randomnumber(rand() % 1000);

int main()
{
    double alimit = sin(pi/2);
}
```

2. Basic data types, variables and constants

❖ Basic data types

- **Character types**

```
char lettera = 'A';
cout << lettera << endl;
char response;
cout << "Yes or No? ";
cin>>response;
// or
response = cin.get();
```

➢ Example character C: ’C’ 67 0103 0x43

2. Basic data types, variables and constants

A.3. Escape characters

Description	ASCII character	Escape sequence
audible bell (alert)	*BEL*	'\a'
backspace	*BS*	'\b'
form feed – new page	*FF*	'\f'
line feed - new line	*NL (LF)*	'\n'
carriage return	*CR*	'\r'
horizontal tab	*HT*	'\t'
vertical tab	*VT*	'\v'
single quote	'	'\''
double quote	"	'\"'
backslash	\	'\\'
question mark	?	'\?'
ANSI character with an octal code	*ooo*	'\ooo'
Null character	*NUL*	'\0'
ANSI character with a hexadecimal code	*hh*	'\xhh'
16-bit Unicode character	*hhhh*	'\uhhhh'
32-bit Unicode character	*hhhhhhhh*	'\Uhhhhhhhh'

2. Basic data types, variables and constants

❖ Basic data types

▪ Character types

- **unsigned char** type: the 8-bit ***ANSI*** code table or a one-byte integer value
- the two-byte **wchar_t** type: a character of the ***Unicode*** table
- Constant character values should be preceded by capital letter *L*.

```
wchar_t uch1 = L'\u221E';
wchar_t uch2 = L'K';
wcout << uch1;
wcin >> uch1;
uch1 = wcin.get();
```

2. Basic data types, variables and constants

❖ Basic data types

▪ Logical Boolean type

➢ **Bool** type variables can have two values: logical **false** is 0, while logical **true** is 1.

```
bool start = true, end(false);
cout << start;
cin >> end;

This default operation can be overridden by boolalpha and noboolalpha I/O manipulators:

bool start = true, end(false);
cout << boolalpha << start << noboolalpha; // true
cout << start;                             // 1
cin >> boolalpha >> end;                   // false
cout << end;                               // 0
```

2. Basic data types, variables and constants

❖ Basic data types

▪ Integer types

➢ The type of constant integer values can be provided by the **U** and **L** postfixes. *U* means **unsigned**, *L* means **long**:

2012	**int**
2012**U**	**unsigned int**
2012**L**	**long int**
2012**UL**	**unsigned long int**
2012**LL**	**long long int**
2012**ULL**	**unsigned long long int**

```
#include <iostream>
using namespace std;

int main()
{
    int x=20121004;
    cout << hex << x << endl;
    cout << oct << x << endl;
    cout << dec << x << endl;
    cin>> hex >> x;
}
```

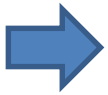
2. Basic data types, variables and constants

❖ Basic data types

▪ Integer types

- ***setw*** (): set the width of the field to be used in printing operations
- ***left*****:** aligned to the (***left***)
- ***right***: aligned to the right (***right***), which is the default value.

```
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
    unsigned int number = 123456;
    cout<<'|' << setw(10) << number << '|' << endl;
    cout<<'|' << right << setw(10) << number << '|' << endl;
    cout<<'|' << left << setw(10) << number << '|' << endl;
    cout<<'|' << setw(10) << number << '|' << endl;
}
```



```
|    123456|
|    123456|
|123456    |
|123456    |
```

2. Basic data types, variables and constants

❖ Basic data types

▪ Floating point types

➢ floating point types: **float**, **double**, **long double** (Visual C++ treats the **long double** type as **double**.)

```
double d =0.01;
float f = d;
cout << setprecision(12) << d*d << endl;   // 0.0001
cout << setprecision(12) << f*f << endl;   // 9.99999974738e-005
```

2. Basic data types, variables and constants

❖ Basic data types

▪ Floating point types

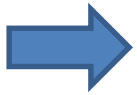
- There is only one value the value of which is surely exact: 0.
- Floating point constant values are **double** type by default. Postfix ***F*** designates a **float** type variable, whereas ***L*** designates a **long double** variable: 12.3F, 1.2345E-10L.
- ***setw()*** - the field width, - ***setprecision()*** - the number of digits after the decimal point, ***fixed*** - decimal representation, ***scientific*** - scientific representation.

2. Basic data types, variables and constants

❖ Basic data types

- **Floating point types**

```
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
    double a = 2E2, b=12.345, c=1.;
    cout << fixed;
    cout << setw(10)<< setprecision(4) << a << endl;
    cout << setw(10)<< setprecision(4) << b << endl;
    cout << setw(10)<< setprecision(4) << c << endl;
}
```



```
200.0000
12.3450
1.0000
```

2. Basic data types, variables and constants

❖ Basic data types

- **Floating point types**
- A type with a smaller value range can be converted into a type with a wider range without data loss.
- However, in the reverse direction, the conversion generally provokes data loss.

```
short int s;
double d;
float f;
unsigned char b;

s = 0x1234;
b = s; // 0x34
// ------------------------
f = 1234567.0F;
b = f; // 135
s = f; // -10617
// ------------------------
d = 123456789012345.0;
b = d; // 0
s = d; // 0
f = d; // f=1.23457e+014 - precision loss
```

2. Basic data types, variables and constants

❖ Basic data types

▪ enum type

- The readability of our programs is much better if these values are replaced by names.

enum ⟨type identifier⟩ { enumeration };

- If *type identifier* is not given, the type is not created only the constants.

enum workdays {Monday, Tuesday, Wednesday, Thursday, Friday};

- By default, the value of the first element (*Monday*) is *0*, that of the next one (*Tuesday*) is *1*, and so on (the value of *Friday* is *4*).

2. Basic data types, variables and constants

❖ Basic data types

▪ enum type

➢ In enumerations, we can directly assign values to their elements.

enum consolecolours {black,blue,green,red=4,yellow=14,white};

➢ In the enumeration named *consolecolours* the value of *white* is *15*.

```
#include <iostream>
using namespace std;

int main()
{
    enum card { clubs, diamonds, hearts, spades };

    enum card cardcolour1 = diamonds;
    card cardcolour2 = spades;
    cout << cardcolour2 << endl;

    int colour = spades;
    cin >> colour;
    cardcolour1 = card(colour);
}
```

2. Basic data types, variables and constants

❖ Basic data types

- **sizeof operation**

➢ the size of any type or any variable and expression type in bytes.

sizeof(typename)

sizeof(variable/expression)

```
cout << sizeof('A' + 'B') <<endl;       // 4 - int
cout << sizeof(10 + 5) << endl;         // 4 - int
cout << sizeof(10 + 5.0) << endl;       // 8 - double
cout << sizeof(10 + 5.0F) << endl;      // 4 - float
```

2. Basic data types, variables and constants

❖ Creation of alias type names

- **volatile unsigned short int** sign;

→ **typedef volatile unsigned short int** uint16;

uint16 sign;

- **typedef** can also be useful in case of enumerations:

typedef enum {falsevalue = -1, unknown, truevalue} bool3;

bool3 start = unknown;

- It is particularly useful to use **typedef** in case of complex types, where type definition is not always simple.

typedef unsigned char byte, uint8;

typedef unsigned short word, uint16;

typedef long long int int64;

2. Basic data types, variables and constants

❖ Constants in language C++

- Constants (macros) ***#define*** should be avoided in C++ language.
- The big advantage and disadvantage of this solution is untypedness.

```
#define ON 1
#define OFF 0
#define PI 3.14159265

int main()
{
    int switched = ON;
    double rad90 = 90*PI/180;
    switched = OFF;
}
```

2. Basic data types, variables and constants

❖ Constants in language C++

- Constant solutions supported by C++ language are based on **const** type qualifiers and the **enum** type.

```
const int on = 1;
const int off = 0;
const double pi = 3.14159265;

int main()
{
    int switched = on;
    double rad90 = 90*pi/180;
    switch = off;
}
```

2. Basic data types, variables and constants

❖ Constants in language C++

- The third possibility is to use an **enum** type, which can only be applied in case of integer (**int**) type constants.

```
enum onoff { off, on };
int switched = on;
switch = off;
```

- **enum** and **const** constants are real constants since they are not stored in the memory by compilers. While ***#define*** constants have their effects from the place of their definition until the end of the file, **enum** and **const** constants observe the traditional C++ visibility and lifetime rules.

Chapter I. Basics and data management of C++

- 1. Creation of C++ programs
- 2. Basic data types, variables and constants
- 3. Basic operations and expressions
- 4. Control program structures
- 5. Exception handling
- 6. Pointers, references and dynamic memory management
- 7. Arrays and strings
- 8. User-defined data types

3. Basic operations and expressions

❖ Classification of operators based on the number of operands

- In case of operators with one operand (unary) the general form of the expression is:

op operand	or	operand op
prefix form		*postfix* form

Examples:

-n	sign change,
n++	incrementing the value of n (postfix),
--n	decrementing the value of n (prefix),
double(n)	transformation of the value of n to real.

3. Basic operations and expressions

❖ Classification of operators based on the number of operands

- Most operations have two operands – these are called two operand (binary) operators:

operand1 op operand2

Examples:

n & 0xFF	obtaining the low byte of n,
n + 2	calculation of n + 2,
n << 3	shift the bits of n to the left with 3 positions,
n += 5	increasing the value of n with 5.

- The C++ language has one three operand operation, this is the conditional operator:

```
operand1 ? operand2 : operand3
```

3. Basic operations and expressions

❖ Precedence and grouping rules

- Rule of precedence: If different precedence operations are found in one expression, then always the part that contains an operator of higher precedence is evaluated first.

Example: The evaluation sequence of expressions a+b*c-d*e

```
int a = 6, b = 5, c = 4, d = 2, e = 3;
b * c   ⇒ 20
d * e   ⇒  6
a + b * c ⇒ a + 20  ⇒ 26
a + b * c - d * e ⇒ 26 - 6 ⇒ 20
```

The steps of processing expression (a+b)*(c-d)*e are:

```
int a = 6, b = 5, c = 4, d = 2, e = 3;
(a + b)  ⇒  11
(c - d)  ⇒   2
(a + b) * (c - d) ⇒ 11 * 2  ⇒ 22
22 * e   ⇒ 22 * 3   ⇒ 66
```

3. Basic operations and expressions

❖ Precedence and grouping rules

- Rule of associativity: Associativity determines whether the operation of the same precedence level is carried out form left to right or from right to left.

Example: In the group of assignment statements evaluation is carried out from the right to the left

a = b = c = 0; identical with a = (b = (c = 0));

In case operations of the same precedence level can be found in one arithmetic expression, the rule from left to right is applied.

```
int a = 6, b = 5, c = 4, d = 2, e = 3;
b * c    ⇒ 20
b * c / d ⇒ 20 / d ⇒ 10
b * c / d * e ⇒ 10 * e ⇒ 30
a + b * c / d * e ⇒ a + 30 ⇒ 36
```

3. Basic operations and expressions

❖ Precedence and grouping rules

Precedence	Operator	Name or meaning	Associativity
1.	**::**	scope resolution	*none*
2.	**( )**	function call, member initialization	*from left to right*
	[ ]	array indexing	
	->	indirect member selection (pointer)	
	.	direct member selection (object)	
	++	(postfix) increment	
	--	(postfix) decrement	
	type **()**	type-cast (conversion)	
	dynamic_cast	checked type-cast at runtime (conversion)	
	static_cast	checked type-cast during compilation time (conversion)	
	reinterpret_cast	unchecked type-cast (conversion)	
	const_cast	constant type-cast (conversion)	
	typeid	type identification	

3. Basic operations and expressions

❖ Precedence and grouping rules

3.	**!**	logical negation (NOT)	*from right to left*
	~	bitwise negation	
	+	+ sign (numbers)	
	-	- sign (numbers)	
	++	(prefix) increment	
	--	(prefix) decrement	
	&	address-of operator	
	*****	indirection operator	
	(*type* **)**	type-cast (conversion)	
	sizeof	size of an object/type in bytes	
	new	allocating dynamic memory space	
	delete	deallocating dynamic memory space	
4.	**.***	direct reference to a class member	*from left to right*
	->*	indirect reference to a member of the object the pointer points to	

3. Basic operations and expressions

❖ Precedence and grouping rules

5.	*******	multiplication	*from left to right*
	/	division	
	%	modulo	
6.	**+**	addition	*from left to right*
	–	subtraction	
7.	**<<**	bitwise left shift	*from left to right*
	>>	bitwise right shift	
8.	**<**	less than	*from left to right*
	<=	less than or equals	
	>	greater than	
	>=	greater than or equals	
9.	**==**	equal to	*from left to right*
	!=	not equal to	

3. Basic operations and expressions

❖ Precedence and grouping rules

10.	**&**	bitwise AND	*from left to right*
11.	**|**	bitwise inclusive OR	*from left to right*
12.	**^**	bitwise exclusive OR (XOR)	*from left to right*
13.	**&&**	logical AND	*from left to right*
14.	**||**	logical OR	*from left to right*
15.	*expr* **?** *expr* **:** *expr*	conditional expression	*from right to left*

3. Basic operations and expressions

❖ Precedence and grouping rules

16.	**=**	simple value assignment	*from right to left*
	***=**	multiplication assignment	
	/=	division assignment	
	%=	modulo assignment	
	+=	addition assignment	
	-=	subtraction assignment	
	<<=	bitwise left shift assignment	
	>>=	bitwise right shift t assignment	
	&=	bitwise AND assignment	
	^=	bitwise XOR assignment	
	|=	bitwise OR assignment	
17.	**throw** *expr*	throwing an expression	*from right to left*
18.	*expr* **,** *expr*	operation sequence (comma operator)	*from left to right*

3. Basic operations and expressions

❖ Mathematical expressions

- **Arithmetical operators**

- the operator of modulo (%), Addition (+), subtraction (-), multiplication (*) and division (/).

  Example: 29 / 7 = 4; 29 % 7 = 1

- **Mathematical functions**

Usage	Type	Function	Include file
calculation of absolute value	real	fabs(real x)	cmath
calculation of absolute value	integer	abs(integer x)	cstdlib
cosine of an angle (in radians)	real	cos(real x)	cmath
sine of an angle (in radians)	real	sin(real x)	cmath
tangent of an angle (in radians)	real	tan(real x)	cmath

3. Basic operations and expressions

❖ Mathematical expressions

▪ Mathematical functions

Usage	Type	Function	Include file
the inverse cosine of the argument (in radians)	real	acos(real x)	cmath
the inverse sine of the argument (in radians)	real	asin(real x)	cmath
the inverse tangent of the argument (in radians)	real	atan(real x)	cmath
the inverse tangent of y/x (in radians)	real	atan(real x, real y)	cmath
natural logarithm	real	log(real x)	cmath
base 10 logarithm	real	log10(real x)	cmath
e^x	real	exp(real x)	cmath
power (x^y)	real	pow(real x, real y)	cmath
square root	real	sqrt(real x)	cmath
random number between 0 and RAND_MAX	real	int rand(void)	cstdlib
the value of π	real	4.0*atan(1.0)	cmath

Where the type *real* designates one of the following types: **float**, **double** or **long double**. The type *integer* designates one of the **int** or **long** types

3. Basic operations and expressions

❖ Mathematical expressions

- **Mathematical functions**

```
#include <iostream>
#include <cmath>
using namespace std;

int main()
{
    double a = 1, b = -5, c =6, x1, x2;
    x1 = (-b + sqrt(b*b-4*a*c))/(2*a);
    x2 = (-b - sqrt(b*b-4*a*c))/(2*a);
    cout << x1 << endl;
    cout << x2 << endl;
}
```

3. Basic operations and expressions

❖ Assignment

```
variable = value;
```

- **Left value and right value**

- The value of the expression on the left side of the equation sign is called ***left value*** (*lvalue*), while the expression on the right side is called ***right value*** (*rvalue*).

- **Side effects in evaluation**

- During processing certain operations – assignment, function call and increment, decrement (++, --), the value of operands may also change besides the value of the expression. This phenomenon is called side effect.

```
a[i] = i++;                          //should be avoided
y = y++ + ++y;                       //should be avoided
cout << ++n << pow(2,n) << endl;     //should be avoided
```

3. Basic operations and expressions

❖ Assignment

- **Assignment operators**

a = 4; b = (a+10)*4;	b = ((a = 4) + 10)* 4;
a = 10; b = 10;	a = b = 10;
a = a + 2;	a += 2;
`expression1 = expression 1 op expression 2`	`expression 1 op= expression 2`

The compound assignment usually results in a faster code, and therefore the source program can be interpreted easier.

3. Basic operations and expressions

❖ Increment and decrement operations

- ++ (increment), -- (decrement)
- The operators can be used only with left value operands, however both prefix and postfix forms can be applied:

```
int a;
// prefix forms:
      ++a;          --a;
// postfix forms:
      a++;          a--;
```

3. Basic operations and expressions

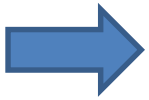
❖ Increment and decrement operations

```
int n, m = 5; m = ++n; // m ⇒ 6, n ⇒ 6

double x, y = 5.0; x = y++; // x ⇒ 5.0, y ⇒ 6.0
```

```
int a = 2, b = 3, c; c = ++a + b--; // a will be 3, b 2 and c 6
```

<==> a++, c=a+b, b--; <==> a++; c=a+b; b--;

```
a = a + 1; a += 1;
a = a - 1; a -= 1;
```



```
++a; or a++;
--a; or a--;
```

```
a += a++ * ++a; //should be avoided
```

3. Basic operations and expressions

❖ Phrasing of conditions

▪ Relational and equality operations

- Two operand, relational operators are available for carrying out comparisons, according to the table below:

Mathematical form	C++ expression	Meaning
a < b	a < b	*a* is less than *b*
a ≤ b	a <= b	*a* is less than or equal to *b*
a > b	a > b	*a* is greater than *b*
a ≥ b	a >= b	*a* is greater than or equal to *b*
a = b	a == b	*a* is equal to *b*
a ≠ b	a != b	*a* is not equal to *b*

All C++ expressions above are **int** type. The value of expressions is **true** (1) if the examined relation is true and **false** (0) if not.

3. Basic operations and expressions

❖ Phrasing of conditions

- **Relational and equality operations**

- Let's take the example of some true expressions that contain different type operands:

```
int i = 3, k = 2, n = -3;
i > k          n <= 0          i+k > n          i != k

char first = 'A', last = 'Z';
first <= last          first == 65          'N' > first

double x = 1.2, y = -1.23E-7;
-1.0 < y          3 * x >= (2 + y)          fabs(x-y)>1E-5
```

3. Basic operations and expressions

❖ Phrasing of conditions

- **Relational and equality operations**

- It is to be noted that due to the computational and representation inaccuracy the identity of two floating point variables cannot be checked with operator ==.

```
double x = log(sin(3.1415926/2));
double y = exp(x);
cout << setprecision(15) << scientific << x << endl;
// x ⇒ -3.30669073875470e-016
cout << setprecision(15) << scientific << y << endl;
// y ⇒ .99999999999997e-001
cout << (x == 0) << endl;            // false
cout << (y == 1) << endl;            // false
cout << (fabs(x)<1e-6) << endl;      // true
cout << (fabs(y-1.0)<1e-6)<< endl;   // true
```

3. Basic operations and expressions

❖ Phrasing of conditions

- **Relational and equality operations**

- Frequent program error is to confuse the operations of assignment (=) and identity testing (==).
- Comparison of a variable with a constant can be made safer if the left side operand is a constant, since the compiler expects a left value during assignment in this case:

```
2004 == dt    instead of    dt == 2004
```

3. Basic operations and expressions

❖ Phrasing of conditions

▪ Logical operations

- The operation of logical operators can be described with a so called truth table:

a	!a
false	true
true	false

logical negation

a	b	a&&b
false	false	false
false	true	false
true	false	false
true	true	true

logical AND operation

a	b	a||b
false	false	false
false	true	true
true	false	true
true	true	true

logical OR operation

3. Basic operations and expressions

❖ Phrasing of conditions

- **Logical operations**

- In C++ programs numerical variable *ok* is frequently used in expressions:

`!ok`	instead of	`ok == 0`
`ok`	instead of	`ok != 0`

- Right side expressions are recommended to be used mainly with **bool** type variable *ok*.

3. Basic operations and expressions

❖ Phrasing of conditions

▪ Conditional operator

- Conditional operator (*?:*) has three operands:

```
condition ? true_expression : false_expression
```

- If the condition is **true**, the value of *true_expression* provides the value of the conditional expression, otherwise the *false_expression* after the colon (:).

```
(n > 0) ? 3.141534 : 54321L;

ch = n >= 0 && n <= 9 ? '0' + n : 'A' + n - 10;

c = 1 > 2 ? 4 : 7 * 2 < 3 ? 4 : 7 ; // 7 should be avoided
c = (1 > 2 ? 4 : (7 * 2)) < 3 ? 4 : 7 ; // 7 should be avoided

c = (1 > 2 ? 4 : 7) * (2 < 3 ? 4 : 7) ; //28
```

3. Basic operations and expressions

❖ Bit operations

- The C++ language contains six operators with the help of which different bitwise operations can be carried out on signed and unsigned integer data.

▪ Bitwise logical operations

- The bitwise logical operations make it possible to test, delete or set bits:

Operator	Operation
~	Unary complement, bitwise negation
&	bitwise *AND*
|	bitwise *OR*
^	bitwise exclusive *OR*

3. Basic operations and expressions

❖ Bit operations

- **Bitwise logical operations**

a	b	a & b	a | b	a ^ b	~a
0	0	0	0	0	1
0	1	0	1	1	1
1	0	0	1	1	0
1	1	1	1	0	0

3. Basic operations and expressions

❖ Bit operations

▪ Bitwise logical operations

```
unsigned short int x = 2525; // 0x09dd
```

Operation	Mask	C++ instruction	Result
Bit setting	*0010 0000 0001 0000*	x = x | 0x2010;	*0x29dd*
Bit deletion	*1101 1111 1110 1111*	x = x & 0xdfef;	*0x09cd*
Bit negation (switching)	*0010 0000 0001 0000*	x = x ^ 0x2010; x = x ^ 0x2010;	*0x29cd (10701)* *0x09dd (2525)*
Negation of all bits	*1111 1111 1111 1111*	x = x ^ 0xFFFF;	*0xf622*
Negation of all bits		x = ~x;	*0xf622*

3. Basic operations and expressions

❖ Bit operations

▪ Bit shift operations

- Shift can be carried out either to the left (<<) or to the right (>>). During shifting the bits of the left side operand move to the left (right) as many times as the value of the right side operand shows.
- In case of shifting to the left bit 0 is placed into the free bit positions, while the exiting bits are lost.
- Shift to the right takes into consideration whether the number is signed or not. In case of **unsigned** types bit 0 enters from the left, while in case of **signed** numbers bit 1 comes in. This means that bit shift to the right keeps the sign.

3. Basic operations and expressions

❖ Bit operations

▪ Bit shift operations

- If the bits of an integer number is shifted to the left by *n* steps, the result is the multiplication of that number with 2^n.
- Shift to the right by *m* bits means integer division by 2^m.

```
short int num;
unsigned char lo, hi;

// Reading the number
cout << "\nPlease enter an integer number [-32768,32767] : ";
cin >> num;

// Determination of the lower byte by masking
lo = num & 0x00FFU;

// Determination of the upper byte by bit shift
hi = num >> 8;

//In the last example the byte sequence of the 4 byte int type variable is reversed:
int n =0x12345678U;
n = (n >> 24) |        // first byte is moved to the end,
    ((n << 8) & 0x00FF0000U) |    // 2nd byte into the 3rd byte,
    ((n >> 8) & 0x0000FF00U) |    // 3rd byte into the 2nd byte,
    (n << 24);        // the last byte to the beginning.
cout << hex << n <<endl;          // 78563412
```

3. Basic operations and expressions

❖ Bit operations

▪ Bit operations in compound assignment

Operator	Relation sign	Usage	Operation
Assignmnet by shift left	<<=	x <<= y	shift of bits of x to the left with y bits,
Assignmnet by shift right	>>=	x >>= y	shift of bits of x to the right with y bits,
Assignmnet by bitwise OR	|=	x |= y	new value of x: x | y,
Assignmnet by bitwise AND	&=	x &= y	new value of x: x & y,
Assignmnet by bitwise exclusive OR	^=	x ^= y	new value of x: x ^ y,

3. Basic operations and expressions

❖ Bit operations

- **Bit operations in compound assignment**

```
unsigned z;

z = 0xFFFFFFFF, z <<= 31; // z ⇒ 80000000
z = 0xFFFFFFFF, z <<= 32; // z ⇒ ffffffff
z = 0xFFFFFFFF, z <<= 33; // z ⇒ fffffffe
```

3. Basic operations and expressions

❖ Comma operator

- x = (y = 4 , y + 3); → y = 4 → y = *4+3=7* → *x = 7.*
- Comma operator is frequently used when setting different initial values for variables in one single statement (expression):

```
x = 2, y = 7, z = 1.2345 ;
```

- Comma operator should be used also when the values of two variables should be changed within one statement (using a third variable):

```
c = a, a = b, b = c;
```

3. Basic operations and expressions

❖ Type conversions

- **Implicit type conversions**

- **char**, **wchar_t**, **short**, **bool**, **enum** type data are automatically converted to **int** (**unsigned int**) type → "integer conversion" (integral promotion).
- During type conversion the "smaller" type operand is converted to the "larger" type → "common arithmetical conversions".

int < **unsigned** < **long** < **unsigned long** < **long long** < **unsigned long long** < **float** < **double** < **long double**

3. Basic operations and expressions

❖ Type conversions

▪ Explicit type conversions

- The (static) type conversions below are all carried out during the compilation of the C++ program. A possible grouping of type conversions:

type conversion (C/C++)	(*type name*) *expression*	(long)p
function-like form	*type name* (*expression*)	int(a)
checked type conversions	**static_cast**< *type name* >(*expression*)	static_cast<double>(x)

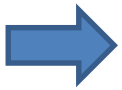
3. Basic operations and expressions

❖ Type conversions

▪ Explicit type conversions

- In case of writing any expression implicit and the maybe necessary explicit conversions have to be considered always.

```
long a =12, b=7;
double d = (a+b)/2;
cout << d << endl;     // 9
```



```
d = (a+b)/2.0;
d = (double)(a+b)/2;
d = double(a+b)/2;
d = static_cast<double>(a+b)/2;
```

Chapter I. Basics and data management of C++

- 1. Creation of C++ programs
- 2. Basic data types, variables and constants
- 3. Basic operations and expressions
- 4. Control program structures
- 5. Exception handling
- 6. Pointers, references and dynamic memory management
- 7. Arrays and strings
- 8. User-defined data types

4. Control program structures

Category	C++ statements
Declaration/definition statements	*types* (**class, struct, union, enum, typedef**), *functions*, *objects*
Expression statements	*expression*;
Empty statement	;
Compound statement	{ *statements* }
Selection statements	**if**, **else**,**switch**, **case**
Iteration statements	**do**, **for**, **while**
Flow control statements	**break**, **continue**, **default**, **goto**, **return**
Exception handling statements	**throw**, **try-catch**

4. Control program structures

❖ Empty statements and statement blocks

- ***Empty statements*** consist only of a semicolon (;). They should be used if no activity has to be performed logically.
- Curly brace brackets ({ and }) enclose declarations and statements that make up a coherent unit together within a ***compound statement*** or ***block***.

```
{
   local definitions, declarations
   statements
}
```

4. Control program structures

❖ Empty statements and statement blocks

To be used in the following three cases:

- when more statements forming together a logical unit should be treated as one (in these cases, blocks only contain statements in general),
- in the body of functions,
- to localize the validity of definitions and declarations.

```
double a, b, c;
cout << "a = "; cin >> a;
cout << "b = "; cin >> b;
cout << "c = "; cin >> c;

if (b*b-4*a*c>=0) {
    double x1, x2;
    x1 = (-b + sqrt(b*b-4*a*c))/(2*a);
    x2 = (-b - sqrt(b*b-4*a*c))/(2*a);
    cout << x1 << endl;
    cout << x2 << endl;
}
```

4. Control program structures

❖ Selective structures

▪ if statements:

- In the case of an **if** statement, the execution of an activity (*statement*) depends on the value of an expression (*condition*). **if** statements have three forms:

One-way branch

- In the following form of **if**, the *statement* is only executed if the value of *condition* is not zero (i.e. **true**).

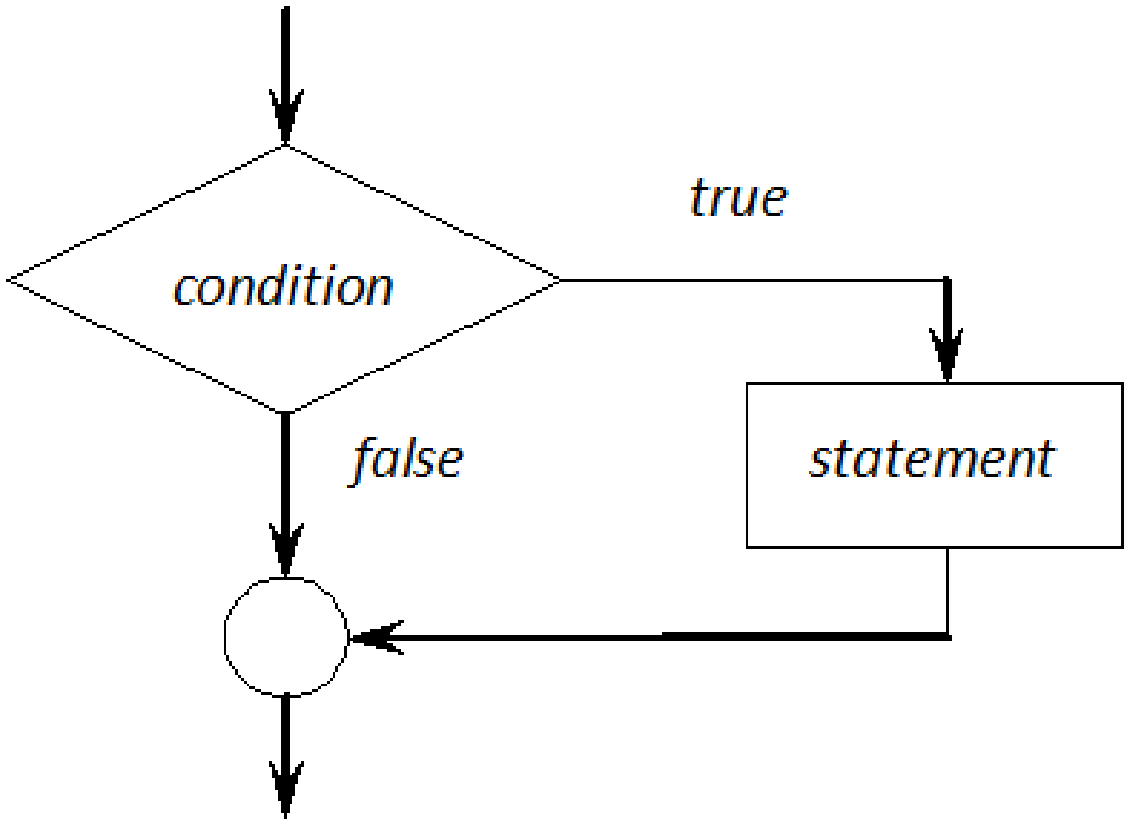
if (*condition*)
statement

4. Control program structures

❖ Selective structures

- **if statements:**

One-way branch



Functioning of a simple if statement

4. Control program structures

❖ Selective structures

- **if statements:**

One-way branch

```
#include <iostream>
#include <cmath>
using namespace std;

int main()
{
    double x = 0;
    cout << "x = "; cin >> x;
    if (x >= 0) {
        cout << sqrt(x) << endl;
    }

    return 0;
}
```

4. Control program structures

❖ Selective structures

▪ if statements:

Two-way branches

- In the complete version of an **if** statement, an activity can be provided (*statement2*) when the value of the *condition* is zero (i.e. **false**).

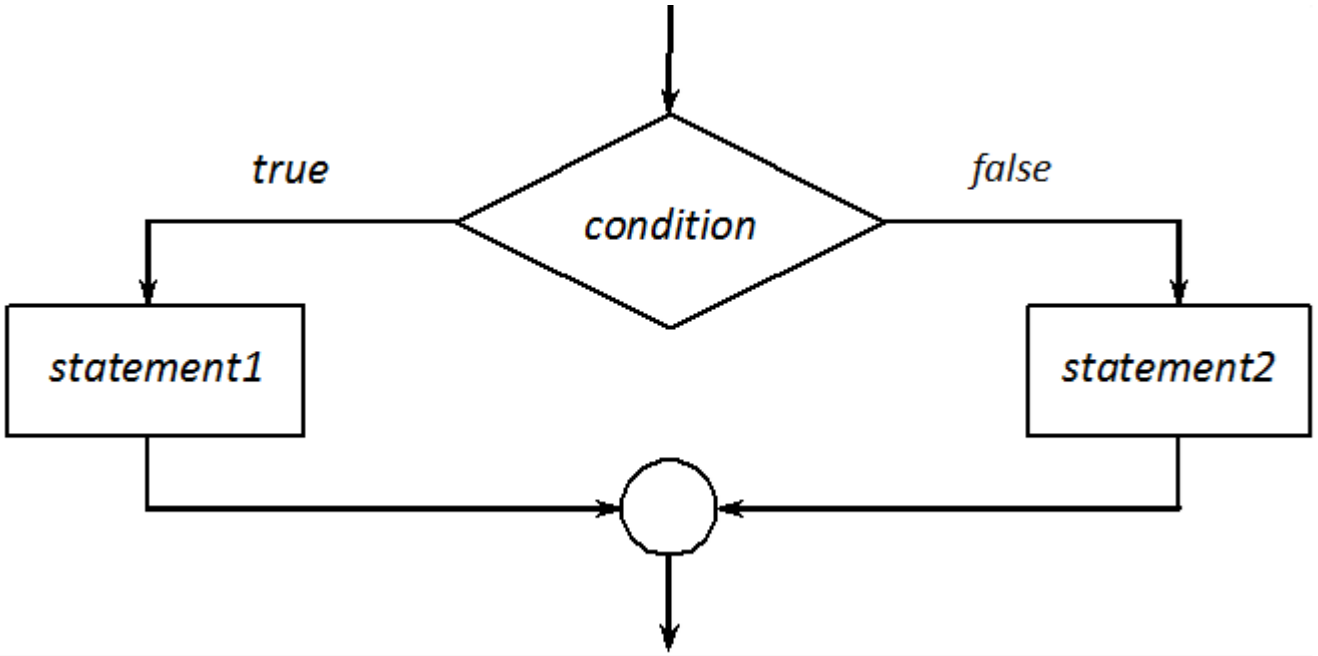
```
if (condition)
        statement1
else
        statement2
```

4. Control program structures

❖ Selective structures

- **if statements:**

Two-way branches



Logical representation of if-else structures

4. Control program structures

❖ Selective structures

- **if statements:**

Two-way branches

```
#include <iostream>
#include <cmath>
using namespace std;

int main()
{
    int n;
    cout << "Type an integer number: "; cin >> n;
    if (n % 2 == 0)
        cout << "The number is even!" << endl;
    else
        cout << "The number is odd!" << endl;

    return 0;
}
```

4. Control program structures

❖ Selective structures

- **if statements:**

Two-way branches

- **if** statements can be nested in one another.
- Compilers connect **else** branches to the closest preceding **if** statement.

```
if (n > 0)
    if (n % 2 == 1)
        cout<<"Positive odd number."<< endl;
    else ;
else
    cout<<"Not a positive number."<<endl;
```

4. Control program structures

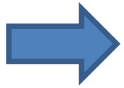
❖ Selective structures

- **if statements:**

Two-way branches

```
if (n > 0) {
    if (n % 2 == 1)
        cout << "Positive odd number." << endl;
} else
    cout << "Not a positive number." << endl;
```

```
if (n > 0) {
    if (n % 2 == 1) {
        cout << "Positive odd number." << endl;
    }
}
else {
    cout << "Not a positive number." << endl;
}
```



best

4. Control program structures

❖ Selective structures

- **if statements:**

Multi-way branches

- A frequent case of nested **if** statements is to use further **if** statements in **else** branches.

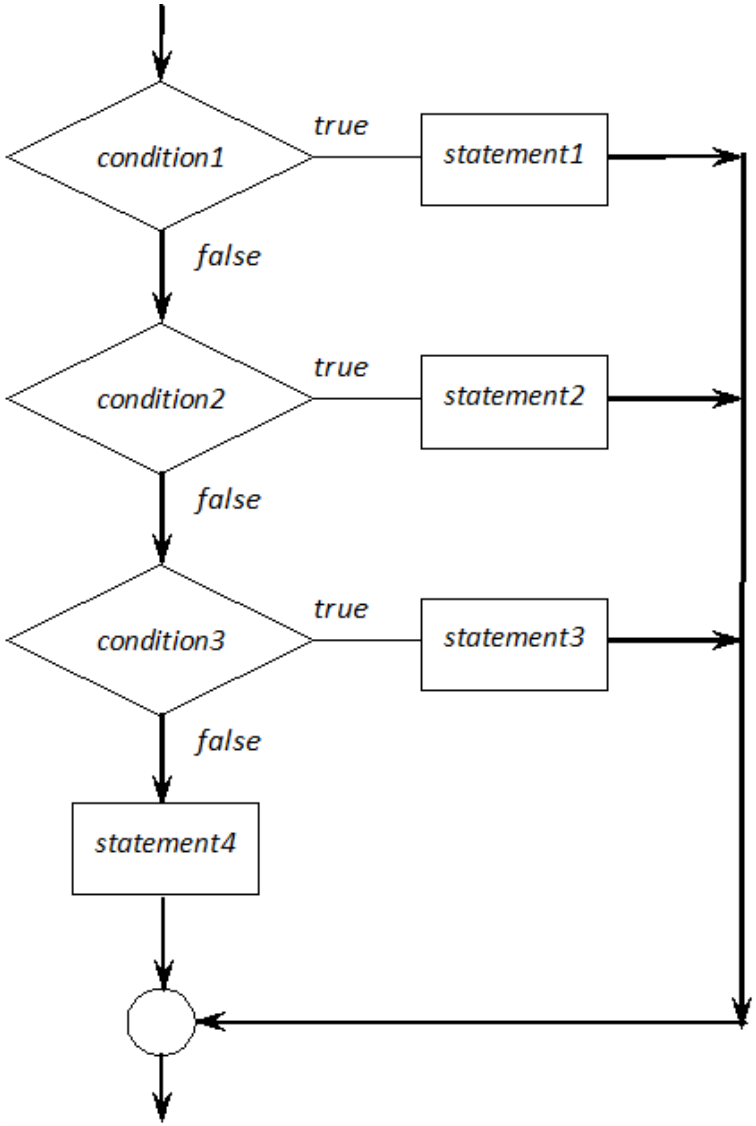
```
if (condition1)
        statement1
else if (condition2)
        statement2
else if (condition3)
        statement3
else
        statement4
```

4. Control program structures

❖ Selective structures

- **if statements:**

Multi-way branches



Logical representation of multi-way branches

4. Control program structures

❖ Selective structures

- **if statements:**

Multi-way branches

```
if (n > 0)
    cout << "Positive number" << endl;
else if (n==0)
    cout << "0" << endl;
else
    cout << "Negative number" << endl;
```

4. Control program structures

❖ Selective structures

- **if statements:**

Multi-way branches

```
char op;
double a, b, c;
cout << "expression : ";
cin >> a >> op >> b;// reading from keyboard: 4+10 <Enter>

if (op == '+')
    c = a + b;
else if (op == '-')
    c = a - b;
else {
    cout << "Not a valid operator: " << op <<endl;
    return -1;
}
cout << a << op << b << '=' << c << endl;
```

4. Control program structures

❖ Selective structures

- **if statements:**

Multi-way branches

```
int points;
char grade = 0;
cout << "Points: ";  cin >> points;

if (points >= 0 && points <= 100)
{
    if (points < 40)
        grade = 'A';
    else if (points >= 40 && points < 55)
        grade = 'B';
    else if (points >= 55 && points < 70)
        grade = 'C';
    else if (points >= 70 && points < 85)
        grade = 'D';
    else if (points >= 86)
        grade = 'F';
    cout << "Grade: " << grade << endl;
}
else
    cout <<"Not a valid number!" << endl;
```

4. Control program structures

❖ Selective structures

▪ switch statements:

- In fact, **switch** statements are statement blocks that we can enter into depending on the value of a given integer *expression*.

```
switch (expression)
{
case      constant_expression1 :
          statements1
case      constant_expression2 :
          statements2
case      constant_expression3 :
          statements3
default :
          statements4
}
```

4. Control program structures

❖ Selective structures

- **switch statements:**

```
int n = 4, f(1);
switch (n) {
        case 5: f *= 5;
        case 4: f *= 4;
        case 3: f *= 3;
        case 2: f *= 2;
        case 1: f *= 1;
        case 0: f *= 1;
}
```

4. Control program structures

❖ Selective structures

- **switch statements:**

- In most cases **switch** statements are used, similarly to **else-if** structures, to realize multi-way branches.
- For that purpose, all statement blocks that correspond to a case have to end with a jump statement (**break**, **goto** or **return**).
- **break** statements transfer control to the statement immediately following the **switch** block, **goto** to the statement with the specified label within the function block and finally **return** exits the function.

4. Control program structures

❖ Selective structures

- **switch statements:**

```
#include <iostream>
#include <cmath>
using namespace std;

int main()
{
    char op;
    double a, b, c;
    cout << "expression :";
    cin >> a >> op >> b;

    switch (op) {
        case '+':
            c = a + b;
            break;
        case '-':
            c = a - b;
            break;
        default:
            cout << "Not a valid operator: " << op << endl;
            return -1;
    }
    cout << a << op << b << '=' << c << endl;

    return 0;
}
```

4. Control program structures

❖ Selective structures

- **switch statements:**

```
#include <iostream>
#include <cmath>
using namespace std;

int main()
{
    cout << "The response [Y/N]?";
    char response = cin.get();

    switch (response) {
        case 'y':
        case 'Y':
            cout << "The answer is YES." << endl;
            break;
        case 'n':
        case 'N':
            cout << "The answer is NO." << endl;
            break;
        default:
            cout << "Wrong response!" << endl;
            break;
    }

    return 0;
}
```

4. Control program structures

❖ Iteration structures (loops)

▪ while loops:

- **while** loops repeat *statements* belonging to them (the body of the loop), *while* the value of the examined *condition* is **true** (not *0*). Evaluation of the *condition* always precedes the execution of the statement.

while (*condition*)
statement

4. Control program structures

❖ Iteration structures (loops)

- **while loops:**

```
#include <iostream>
#include <cmath>

using namespace std;

int main()
{
    int n = 2012;
    cout << "The sum of the first " << n << " natural number ";
    unsigned long sum = 0;
    while (n>0) {
        sum += n;
        n--;
    }
    cout << "is: " << sum << endl;

    return 0;
}
```

4. Control program structures

❖ Iteration structures (loops)

- **while loops:**

```
#include <iostream>
#include <cstdlib>
#include <ctime>

using namespace std;

int main()
{
    srand((unsigned)time(NULL));
    while (int n = rand()%10)
        cout<< n << endl;

    return 0;
}
```

4. Control program structures

❖ Iteration structures (loops)

▪ for loops:

- In the general form of **for** statements, the role of each expression is also mentioned:

```
for (initialization; condition; increment)
        statement
```

- In reality, **for** statements are the specialized versions of **while** statements, so the above **for** loop can perfectly be transformed into a **while** loop:

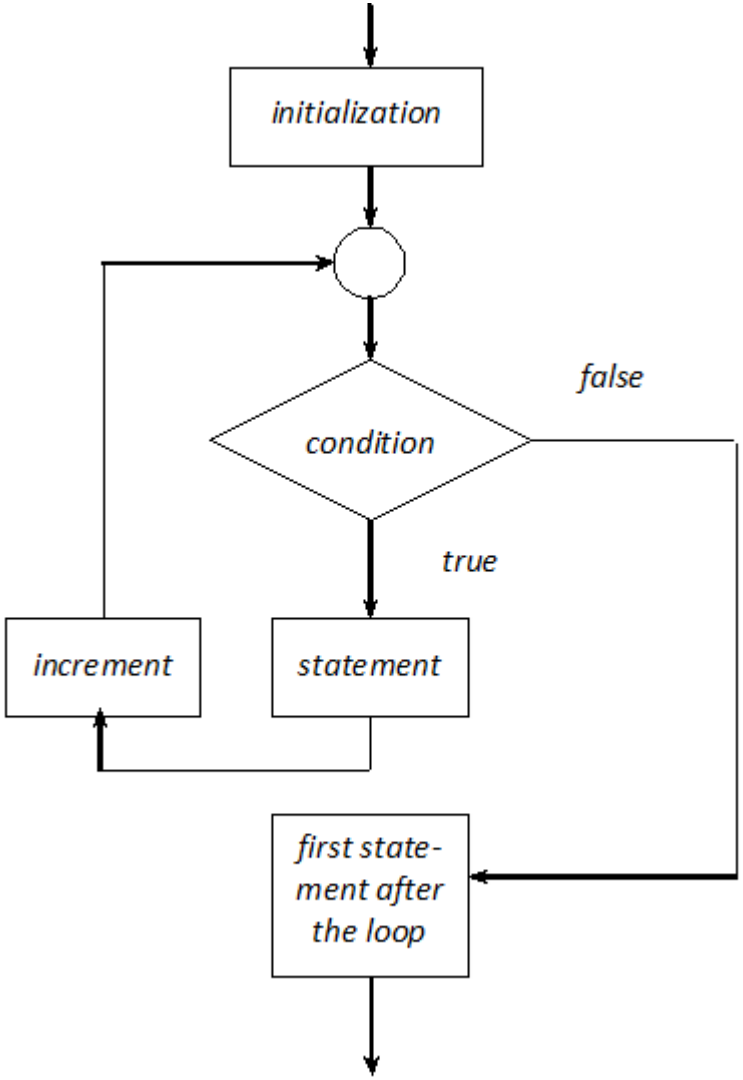
```
initialization;
while (condition) {
        statement;
        increment;
}
```

4. Control program structures

❖ Iteration structures (loops)

- **for loops:**

Logical representation of for loops



4. Control program structures

❖ Iteration structures (loops)

- **for loops:**

```
#include <iostream>

using namespace std;

int main()
{
    unsigned long sum;
    int i, n = 2012;
    cout << "The sum of the first " << n << " natural number ";
    for (i = 1, sum = 0; i <= n; i++)
        sum += i;
    cout << "is: " << sum << endl;

    return 0;
}
```

4. Control program structures

❖ Iteration structures (loops)

- **for loops:**

```
*
* *
* * *
* * * *
* * * * *
* * * * * *
* * * * * * *
* * * * * * * *
* * * * * * * * *
* * * * * * * * * *
* * * * * * * * * * *
```

```
#include <iostream>

using namespace std;

int main()
{
    const int maxn = 12;
    for (int i = 0; i < maxn; i++) {
        for (int j = 0; j < maxn-i; j++) {
            cout << " ";
        }
        for (int j = 0; j < i; j++) {
            cout << "* ";
        }
        cout << endl;
    }

    return 0;
}
```

4. Control program structures

❖ Iteration structures (loops)

▪ do-while loops:

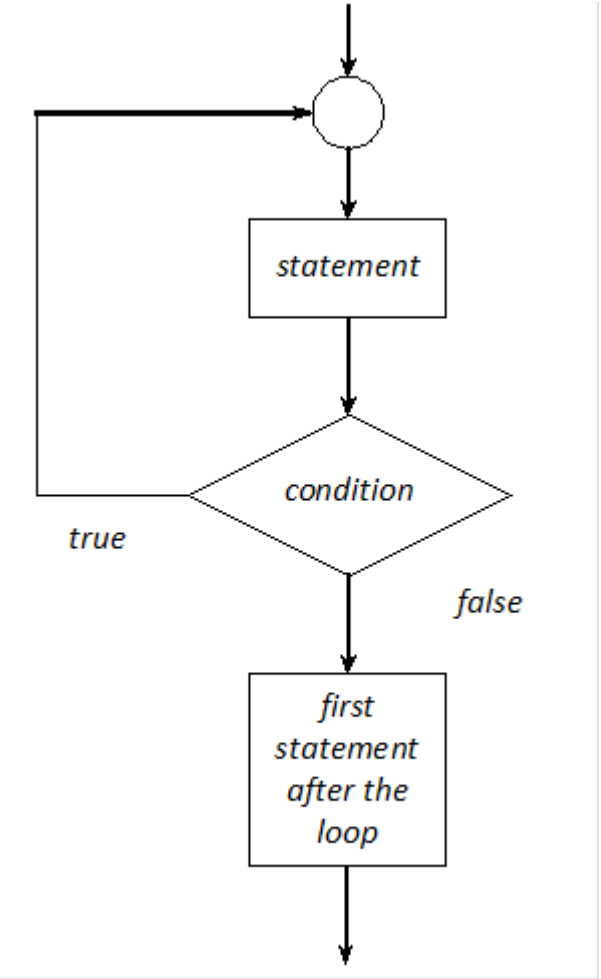
- In **do-while** loops, evaluation of the *condition* takes place after the first execution of the body of the loop, so the loop body is always executed at least once:

```
do
        statement
while (condition);
```

4. Control program structures

❖ Iteration structures (loops)

- **do-while loops:**



Logical representation of do-while loops

4. Control program structures

❖ Iteration structures (loops)

- **do-while loops:**

```
#include <iostream>

using namespace std;

int main()
{
    int n = 2012;
    cout << "The sum of the first " << n << " natural number ";
    unsigned long sum = 0;
    do {
        sum += n;
        n--;
    } while (n > 0);
    cout << "is: " << sum << endl;

    return 0;
}
```

4. Control program structures

❖ Iteration structures (loops)

- **do-while loops:**

```
#include <iostream>

using namespace std;

int main()
{
    int m = 0;
    do {
        cout << "Enter an integer number betwwen 2 and 100: ";
        cin >> m;
    } while (m < 2 || m > 100);

    return 0;
}
```

4. Control program structures

❖ Iteration structures (loops)

- **do-while loops:**

```
#include <iostream>
#include <cstdlib>

using namespace std;

int main()
{
    double base;
    int exponent;
    cout << "base:  "; cin >> base;
    cout << "exponent:  "; cin >> exponent;
    double power = 1;
    if (exponent != 0)
    {
        int i = 1;
        do
        {
            power *= base;
            i++;
        } while (i <= abs(exponent));
        power = exponent < 0 ? 1.0 / power : power;
    }
    cout << "The power is :  " << power << endl;

    return 0;
}
```

4. Control program structures

❖ Iteration structures (loops)

- **break statements in loops:**

- The **break** statement interrupts the execution of the nearest **while**, **for** and **do-while** statements and control passes to the first statement following the interrupted loop:

```
#include <iostream>
#include <cstdlib>

using namespace std;

int main()
{
    int a, b, lcm;
    cout << "a = "; cin >> a;
    cout << "b = "; cin >> b;
    lcm = min(a,b);
    while (lcm <= a*b) {
        if (lcm % a == 0 && lcm % b == 0)
            break;
        lcm++;
    }
    cout << "The lowest common multiple is: " << lcm << endl;
}
```

4. Control program structures

❖ Iteration structures (loops)

- **break statements in loops:**

```
#include <iostream>

using namespace std;

int main () {
    const int maxn =2012;
    int divisor;
    bool prime;
    for(int num = 2; num <= maxn; num++) {
        prime = true;
        for(divisor = 2; divisor <= (num/divisor); divisor++) {
            if (num % divisor == 0) {
                prime = false;
                break; // has a divisor, not a prime
            }
        }
        if (prime)
            cout << num << " is a prime number" << endl;
    }

    return 0;
}
```

4. Control program structures

❖ Iteration structures (loops)

- **break statements in loops:**

```
#include <iostream>
#include<cmath>

using namespace std;

int main () {
    int left, right;
    cout << "left  = "; cin >> left;
    cout << "right = "; cin >> right;
    bool found = false;
    for(int a = left, c, c2; a <=right && !found; a++) {
        for(int b = left; b <= right && !found; b++) {
            c2 = a*a + b*b;
            c = static_cast<int>(sqrt(float(c2)));
            if (c*c == c2) {
                found = true;
                cout << a << ", " << b  << ", " << c << endl;
            } // if
        } // for
    } // for

    return 0;
} // main()
```

4. Control program structures

❖ Iteration structures (loops)

- **continue statements:**

- **continue** statements start the next iteration of **while**, **for** and **do-while** loops. In the body of these loops, the statements placed after **continue** are not executed.
- In the case of **while** and **do-while** loops, the next iteration begins with evaluating again the *condition*. However, in **for** loops, the processing of the *condition* is preceded by the *increment*.

4. Control program structures

❖ Iteration structures (loops)

- **continue statements:**

```
#include <iostream>

using namespace std;

int main(){
    const int maxn = 123;
    for (int i = 1; i <= maxn; i++) {
        if ((i % 7 != 0) && (i % 12 != 0))
            continue;
        cout << i << endl;
    }

    return 0;
}
```

Chapter I. Basics and data management of C++

- 1. Creation of C++ programs
- 2. Basic data types, variables and constants
- 3. Basic operations and expressions
- 4. Control program structures
- 5. Exception handling
- 6. Pointers, references and dynamic memory management
- 7. Arrays and strings
- 8. User-defined data types

5. Exception handling

- An anomalous state or event that hinders the normal flow of the execution of a program is called an *exception*.
- Three elements that are needed for the type-oriented exception handling of C++ are the following:

➢ selecting the code part under exception inspection (**try**-block),

➢ transferring exceptions (**throw**),

➢ catching and handling exceptions (**catch**).

5. Exception handling

- The try – catch program structure
- Provoking exceptions - the throw statement
- Filtering exceptions
- Nested exceptions

5. Exception handling

❖ The try – catch program structure

- In general, a **try-catch** program structure contains only one **try** block and any number of **catch** blocks:

```
try
{
    // statements under exception inspection
    statements
}
catch (exception1 declaration) {
    // the handler of exception1
    statements1
}
catch (exception2 declaration) {
    // the handler of exception2
    statements2
}
catch (...) {
    // the handler of any other exception
    statements3
}
// statements that are executed after successful handling
statements4
```

5. Exception handling

❖ The try – catch program structure

```
#include <iostream>
#include <cstdlib>

using namespace std;

int main()
{
    try {
        double a, b, c;
        cout << " coefficients (separated from each"
             "other by a space) :";
        cin >> a >> b >> c;
        if (0 == a) throw false;
        cout << a << "x^2+" << b << "x+" << c << "=0" << endl;
    }
    catch (bool) {
        cerr << "The equation is not quadratic!" << endl;
        exit(-1);
    }
    catch (...) {
        cerr << "An error occurred..." << endl;
        exit(-1);
    }

    return 0;
}
```

5. Exception handling

❖ Provoking exceptions - the throw statement

- Exceptions have to be handled locally within the given part of a code with the keywords **try**, **throw** and **catch**. Exception handling only takes place in case the code in the catch block (and the code called from there) is executed. From the selected code portion, a **throw** statement

  throw *expression*;

  passes control to the handler that corresponds to the type of the expression, which should be provided after the keyword **catch**.

- Exception classes make it possible to pass "smart" objects as exceptions instead of simple values.

5. Exception handling

❖ Provoking exceptions - the throw statement

Exception classes			Header file
exception			<exception>
	bad_alloc		<new>
	bad_cast		<typeinfo>
	bad_typeid		<typeinfo>
	logic_failure		<stdexcept>
		domain_error	<stdexcept>
		invalid_argument	<stdexcept>
		length_error	<stdexcept>
		out_of_range	<stdexcept>
	runtime_error		<stdexcept>
		range_error	<stdexcept>
		overflow_error	<stdexcept>
		underflow_error	<stdexcept>
	ios_base::failure		<ios>
	bad_exception		<exception>

5. Exception handling

❖ Provoking exceptions - the throw statement

```
#include <iostream>
#include <cstdlib>
#include <ctime>

using namespace std;

int main() {
    int number;
    srand(unsigned(time(NULL)));
    try {
        while (true) {
            number = rand();
            if (number > 1000)
                throw "The number is too big";
            else if (number < 10)
                throw "The number is too small";
            cout << 10*number << endl;
        } // while

    } // try
    catch (const char * s) {
        cerr << s << endl;
    } // catch

    return 0;
} // main()
```

5. Exception handling

❖ Filtering exceptions

- Functions in C++ have an important role in exception handling.
- In the header of a function definition, the keyword **throw** can be used in a specific way, so the type of the exception to be thrown to the handler can also be defined. By default, all exceptions are transferred.

5. Exception handling

❖ Filtering exceptions

```
// all exceptions are transferred.
int funct1();
// only exceptions of type char and bool are transferred
int funct2() throw(char, bool);
// only exceptions of type bool are transferred
int funct3() throw(bool);
// no exceptions are transferred
int funct4() throw();
```

5. Exception handling

❖ Nested exceptions

- A **try-catch** exception handling structure can be placed within another try-block, either directly or indirectly (i.e. in the function called from that try-block).

```
#include <iostream>

using namespace std;

int main()
{
    try {
        try {
            throw "exception";
        }
        catch (bool) {
            cerr << "bool error" << endl;
        }
        catch(const char * s) {
            cout << "inner " << s << endl;
            throw;
        }
    }
    catch(const char * s) {
        cout << "outer " << s << endl;
    }
    catch(...) {
        cout << "unknown exception" << endl;
    }

    return 0;
}
```

5. Exception handling

❖ Nested exceptions

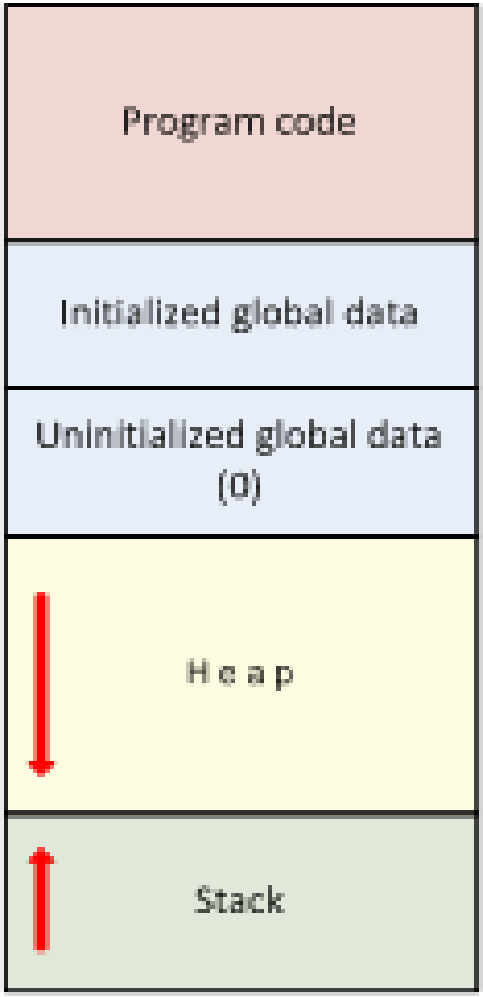
```
#include <iostream>

using namespace std;

enum zerodiv {DividingWithZero};
enum wrongoperation {WrongOperation};

int main()
{
    double a, b, e=0;
    char op;
    while (true)  // exiting if op is x or X
    {
        cout << ':';
        cin >> a >> op >> b;
        try {
            switch (op) {
                case '+':    e = a + b;
                             break;
                case '-':    e = a - b;
                             break;
                case '*':    e = a * b;
                             break;
                case '/':    if (b == 0)
                             throw DividingWithZero;
                             else
                                 e = a / b;
                             break;
                case 'x': case 'X' :
                             throw true;
                default: throw WrongOperation;
            }
            cout << e << endl;
        } // try
        catch (zerodiv) {
            cout << " Dividing with zero" << endl;
        }
        catch (wrongoperation) {
            cout << " Wrong operation" << endl;
        }
        catch (bool) {
            cout << "Off" << endl;
            break;   // while
        }
        catch(...) {
            cout << " Something is bad " << endl;
        }
    } // while

    return 0;
}
```

Chapter I. Basics and data management of C++

- 1. Creation of C++ programs
- 2. Basic data types, variables and constants
- 3. Basic operations and expressions
- 4. Control program structures
- 5. Exception handling
- 6. Pointers, references and dynamic memory management
- 7. Arrays and strings
- 8. User-defined data types

6. Pointers, references and dynamic memory management

- Based on the storage class, compiler places global (**extern**) data grouped together on a memory place which is always accessible while the program is running.
- Local (**auto**) data are stored in a *stack* memory block when the corresponding functions are entered into and they are deleted when these functions are exited.



C++ program memory usage

6. Pointers, references and dynamic memory management

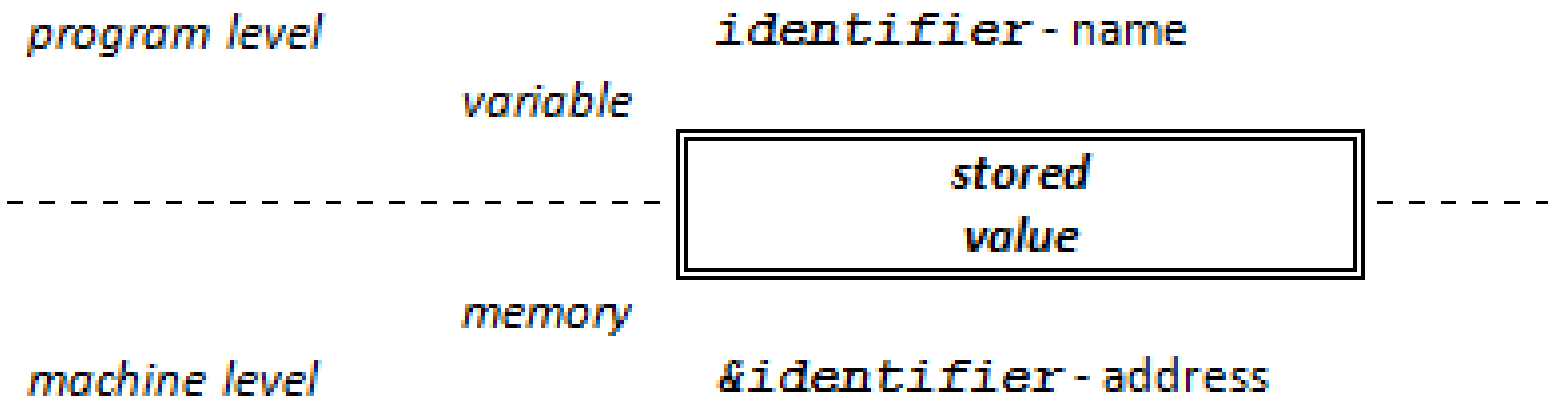
- However, there is a space named *heap*, where programmers can place variables and delete those that are not needed anymore.
- They can be referenced by variables storing their address (pointers). C++ has a couple of operators that make dynamic memory management possible: *, *&*, *new*, *delete*.
- There are many domains where pointers are used in C/C++ programs: function parameter passing, linked data structure management, etc.

6. Pointers, references and dynamic memory management

- Pointers
- References
- Dynamic memory management

❖ Pointers

- In most cases, we assign values to variables and read their content by using their names.
- There are cases where this approach is insufficient and the address of a variable in the memory should be used directly (for example when calling the Standard Library function ***scanf*** *()*).



❖ Pointers

- With the help of pointers, the address of variables (data stored in memory) and functions can be stored and managed.
- A pointer does not only store address but also have information how to interpret how many bytes from that given address.
- And this is the type of the referenced data, which is provided in the definition of pointers (variables).

❖ Pointers

▪ Single indirection pointers:

- First, let's start with the most frequently used and the simplest form of pointers, that is with single indirection pointers the general definition of which is:

  *type *identifier;*

- It is a safe solution to initialize always pointers after they are created with ***NULL*** value, which can be found in most header files:

  *type *identifier = NULL;*

- If more pointers are created within a single statement, all identifier names should be preceded by an asterisk:

  *type *identifier1, *identifier2;*

❖ Pointers

▪ Single indirection pointers:

- Many operations can be carried out for pointers; however, there are three operators that can exclusively be used with pointers:

**ptr*	Accessing the object *ptr* points to.
ptr->member	The access of the given *member* of the structure *ptr* points to
&leftvalue	Getting the address of *leftvalue*.

- An alternative pointer type can also be created by using the keyword **typedef**:

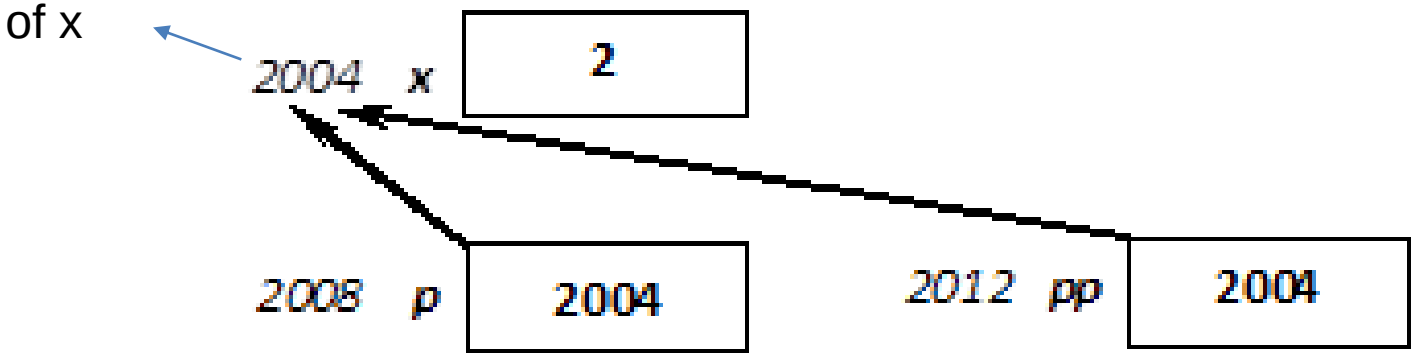
```
int x = 2;
typedef int *tpi;
tpi p = &x;
```

❖ Pointers

- **Single indirection pointers:**

- A single variable can be referenced by many pointers and its value can be modified by any of them:

```
int x = 2;
int * p, *pp;
p     = &x;
pp    = p;
*pp += 10; // the value of x is 12 again
```



❖ Pointers

- **Single indirection pointers:**

- If a pointer is initialized with an address of a variable of a different type, a compiler error message is sent:

```
long y = 0;
char *p = &y;// error! --> should be avoided
```

- If it is not an error and the **long** type data is to be accessed by bytes, compilers can be asked to do value assignment by type cast:

```
long y = 0;
char *p = (char *)&y;
```

or

```
long y = 0;
char *p = reinterpret_cast<char *>(&y);
```

❖ Pointers

▪ Pointer arithmetic:

- The allowed pointer arithmetic operations are summarised in a table where *q* and *p* are pointers (not of **void*** type), *n* is an integer number (**int** or **long**):

Operation	Expression	Result
two pointers of the same type can be subtracted from each other	q - p	*integer number*
an integer number can be added to a pointer	p + n, p++,++p, p += n,	*pointer*
an integer number can be subtracted from a pointer	p – n, p--, --p, p -= n	*pointer*
two pointers can be compared	p == q, p > q, etc.	*bool (false or true)*

❖ Pointers

- **Pointer arithmetic:**

- Accordingly, increment and decrement operators can also be used for pointers and not only for arithmetic types, but in the former case it means going to the neighbouring element and not an increment or decrement by one byte.

```
int *p, *q;
p  = p + 1;
p += 1;
p++;
++p;
```

```
p  = p - 1;
p -= 1;
p--;
--p;
```

❖ Pointers

▪ void * type general pointers:

- C++ also allows for using general pointers without a type (**void** types),

```
int x;
void * ptr = &x;
```

which only store addresses, so they are not associated to any variable.

- C++ ensures two implicit conversions for pointers. A pointer with any type can be transformed into a general (**void**) type pointer, and all pointers can be initialized by zero (0). For a conversion in the other direction, explicit type cast should be used.

❖ Pointers

- **void * type general pointers:**

```
int x;
void *ptr = &x;

*(int *)ptr = 1002;

typedef int * iptr;
*iptr(ptr) = 1002;

*static_cast<int *>(ptr) = 1002;

*reinterpret_cast<int *>(ptr) = 1002;
```

After any of these indirect value assignments, the value of *x* becomes 1002.

❖ Pointers

▪ Multiple indirection pointers:

- Pointers can also be used in the case of multiple indirection relations. In these cases, the definition of pointers contains more asterisks (*):

  type **pointer;

- Let's have a look at some definitions and let's find out what the created variable is.

```
int x;     x is an int type variable,

int *p;    p is an int type pointer (that can point to an int variable),

int **q;   q is an int* type pointer (that may point to an int* variable,
 that is to a pointer pointing to an integer).
```

❖ Pointers

▪ Multiple indirection pointers:

- The above detailed definitions can be rewritten in a more understandable way by alternative (**typedef**) type names:

```
// iptr - pointer type pointing to an integer
typedef int *iptr;
iptr p, *q;
```

Or:

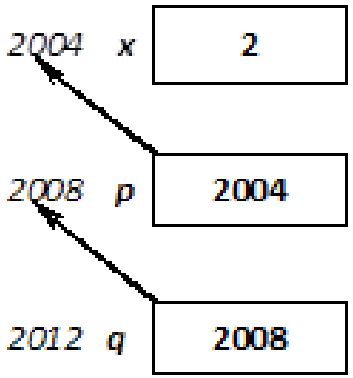
```
// iptr - pointer type pointing to an integer
typedef int *iptr;

// type of a pointer pointing to an iptr type variable
typedef iptr *ipptr;

iptr p;
ipptr q;
```

❖ Pointers

- **Multiple indirection pointers:**

- When these definitions are provided, and when the statements

```
x = 2;
p = &x;
q = &p;
x = x + *p + **q;
```



are executed, the value of x will be 6.

❖ Pointers

▪ Constant pointers

- C++ compilers strictly verify the usage of **const** type constants, for example a constant can be referenced by an appropriate pointer:

```
const double pi = 3.141592565;
// pointer pointing to a double type data
double *ppi = &pi;// error! --> should be avoided
```

- The assignment can be carried out by a pointer pointing to the constant:

```
// pointer pointing to a double constant
const double *pdc;
const double dc = 10.2;
double d = 2012;
pdc  = &dc;// pdc pointer points to dc
cout << *pdc << endl;// 10.2
pdc  = &d;// pdc pointer is set to d
cout << *pdc << endl;// 2012
```

❖ Pointers

▪ Constant pointers

- By using the *pdc* pointer, we cannot modify the value of the *d* variable:

```
*pdc = 7.29;// error! --> should be avoided
```

- It is a pointer with a constant value, so the value of the pointer cannot be changed:

```
int month;
// constant pointer pointing to an int type data
int *const actmonth = &month; //
```

- The value of the *actmonth* pointer cannot be changed only that of **actmonth*.

```
*actmonth = 9;
cout << month << endl; // 9
actmonth = &month; // error! --> should be avoided
```

❖ Pointers

▪ Constant pointers

- Pointer with a constant value pointing to a constant:

```
const int month = 10;
const int amonth = 8;
// constant pointer pointing to an int type constant
const int *const actmonth = &month;
cout << *actmonth << endl;    // 10
```

- Neither the pointer, nor the referenced data can be changed.

```
actmonth = &amonth;// error! --> should be avoided
*actmonth = 12;// error! --> should be avoided
```

❖ References

- Reference types make it possible to reference already existing variables while defining an alternative name. The general form of their definition:

  type &identifier = variable;

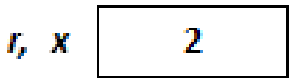
- When defining many references with the same type, the sign & should be typed before all references:

  type &identifier1 = variable1, &identifier2 = variable2 ;

- When a reference is defined, initialisation has to be done with a left value. Let's make a reference to the **int** type *x* variable as an example.

```
int x = 2;
int &r = x;
```

- Contrary to pointers, no variable is created to store references in general. Compilers just give a new name as a second name to the variable *x* (*r*).



❖ References

- As a consequence, the value of *x* becomes 12 when the following statement is evaluated:

  r = x + 10;

- While the value of a pointer, and the referenced storage place as a consequence, can be modified at any time, the reference *r* is bound to a variable.

```
int x = 2, y = 4;
int &r = x;
r = y;              // normal value assignment
cout << x << endl;// 4
```

❖ References

- If a reference is initialized by a constant value or a variable of a different type, a compiler error message is sent.

```
const char &lf = '\n';
```

```
unsigned int b = 2004;
const int &r = b;
b = 2012;
cout << r << endl; // 2004
```

- Synonymous reference types can also be created by the keyword **typedef**:

```
typedef int &rint;
int x = 2;
rint r = x;
```

❖ References

- References can be created to pointers just like for other variable types:

```
int n = 10;
int *p = &n;        // p points to the variable n
int* &rp = p;       // rp is a reference to the pointer p
*rp = 4;
cout << n << endl;// 4
```

- The same can be done with **typedef**:

```
typedef int *tpi; // type of a pointer pointing to an integer
typedef tpi &rtpi;// reference to a pointer pointing to an integer
int n = 10;
tpi p = &n;
rtpi rp = p;
```

❖ References

- It should be noted that a reference or a pointer cannot be defined for a reference.

```
int n = 10;
int &r = n;
int& *pr = &r;// error! --> should be avoided
int& &rr =r;// error! --> should be avoided
int *p = &r;
// pointer p points to n via the reference r
```

❖ Dynamic memory management

- The dynamic management of free memory (*heap*) is a vital part of all programs. C ensures Standard Library functions for the needed memory allocation (***malloc*** *()*,...) and deallocation (***free*** *()*) operations. In C++, the operators **new** and **delete** replace the above mentioned library functions (although the latter are also available).
- Dynamic memory management consists of the following three steps:

1. allocating a free memory block while verifying the success of the allocation,
2. accessing the memory space with a pointer,
3. freeing (deallocating) the previously allocated memory space.

❖ Dynamic memory management

- **Allocating and accessing heap memory**

- The operator **new** allocates a memory space in the heap of the size corresponding to the *type* provided in its operand and returns a pointer pointing to the beginning of that memory space.

```
pointer = new type;

pointer = new type(initial value);
```

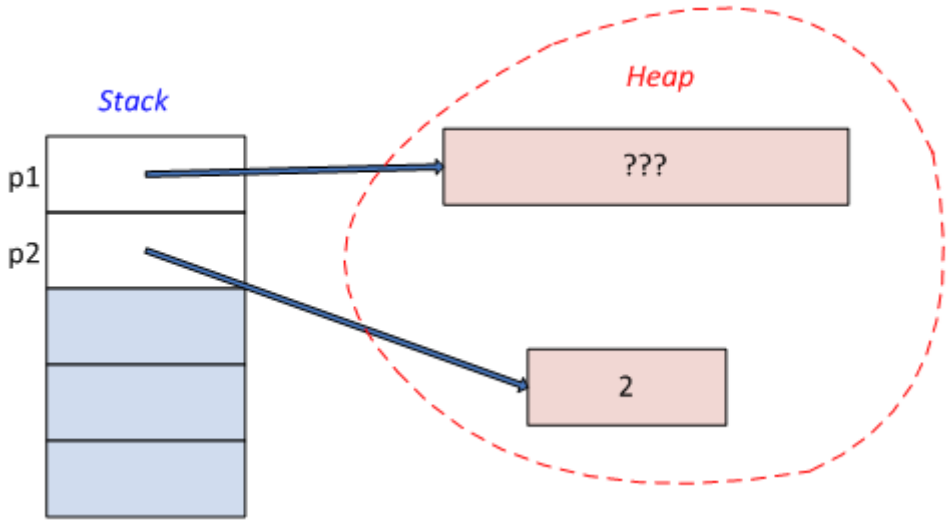
- A dynamic array.

```
pointer = new type[number_of_elements];
```

❖ Dynamic memory management

- **Allocating and accessing heap memory**

```
int main() {
    int *p1;
    double *p2;
    p1 = new int(2);
    p2 = new double;
}
```



Dynamic memory allocation

❖ Dynamic memory management

- **Allocating and accessing heap memory**

```
#include <iostream>
#include <exception>

using namespace std;

int main() {
    long * pdata;
    // Memory allocation
    try {
      pdata = new long;
    }
    catch (bad_alloc) {
      // Unsuccessful allocation
      cerr << "\nThere is no enough memory!" << endl;
      return -1;  // Program is exited
    }
    // ...
    // Deallocating (freeing) allocated memory space
    delete pdata;
    return 0;
}
```

❖ Dynamic memory management

- **Allocating and accessing heap memory**

```
#include <iostream>
#include <new>

using namespace std;

int main() {
    long * pdata;
    // Memory allocation
    pdata = new (nothrow)long;
    if (0 == pdata) {
        // Unsuccessful allocation
        cerr << "\nThere is no enough memory!" << endl;
        return -1;  // Program is exited
    }
    // ...
    // Deallocating (freeing) allocated memory space
    delete pdata;
    return 0;
}
```

❖ Dynamic memory management

- **Allocating and accessing heap memory**

- **new** can also be followed directly by a pointer in parenthesis, which makes the operator return the value of the pointer (thus it does not allocate memory):

```
int *p = new int(10);
int *q = new(p) int(2);
cout << *p << endl;    // 2
```

- In the examples above, the pointer *q* reference the memory space *p* points to. Pointers can be of a different type:

```
long a = 0x20042012;
short *p = new(&a) short;
cout << hex << *p << endl;  // 2012
```

❖ Dynamic memory management

- **Deallocating allocated memory**

- Memory blocks allocated by the operator **new** can be deallocated by the operator **delete**:

  delete *pointer*;
  delete[] *pointer*;

- The first form of the operation is used to deallocate one single dynamic variable, whereas the second one is used in the case of dynamic arrays.

- **delete** operation also works correctly with pointers of *0* value. In every other case where the value was not assigned by **new**, the result of **delete** is unpredictable.

❖ Dynamic memory management

- **Deallocating allocated memory**

```
int main() {
    int *p1;
    double *p2;
    p1 = new int(2);
    p2 = new double;
    delete p1;
    delete p2;
    p1 = 0;
    p2 = 0;
}
```

Chapter I. Basics and data management of C++

- 1. Creation of C++ programs
- 2. Basic data types, variables and constants
- 3. Basic operations and expressions
- 4. Control program structures
- 5. Exception handling
- 6. Pointers, references and dynamic memory management
- 7. Arrays and strings
- 8. User-defined data types

7. Arrays and strings

- C++ array types
- Dynamically allocated arrays
- The usage of the vector type
- Handling C-style strings

7. Arrays and strings

❖ C++ array types

- An *array* is a set of data of the same type (*elements*) that are placed in memory in a linear sequence.
- The most frequently used array type has only one dimension: ***one-dimensional*** array (*vector*).
- If the elements of an array are intended to be identified by more integer numbers, storage should be realised by ***multi-dimensional*** arrays.
- From among these, we only detail the second most frequent array type, the ***two-dimensional*** array, i.e. (*matrix*), the elements of which are stored in a linear sequence by rows.

7. Arrays and strings

❖ C++ array types

- The definition of n-dimensional arrays:

$$element\_type\ array\_name[size_1][size_2][size_3]\ldots[size_{n-1}][size_n]$$

where $size_i$ determines the size of the i^{th} dimension.

- In order to access the array elements, an index should be provided for every dimension in the closed interval between $0, size_i - 1$: :

$$array\_name[index_1][index_2][index_3]\ldots[index_{n-1}][index_n]$$

7. Arrays and strings

❖ C++ array types

▪ One-dimensional arrays

- Definition of one-dimensional arrays:

element_type array_name[*size*];

	square	*element in the array*
0^{th}	0	square[0]
1^{st}	1	square[1]
2^{nd}	4	square[2]
3^{rd}	9	square[3]
4^{th}	16	square[4]
5^{th}	25	square[5]
6^{th}	36	square[6]

Graphical representation of an one-dimensional array

```
const int maxn =7;
int square[maxn];
```

7. Arrays and strings

❖ C++ array types

▪ One-dimensional arrays

- Access the elements of the array using a **for** loop

```
for (int i = 0; i< maxn; i++)
    square[i] = i * i;
```

- The size of memory in bytes allocated for the array *square* is returned by the expression ***sizeof*** *(square)*, whereas the expression ***sizeof****(square[0])* returns the size of one element.
- The number of elements of an array :

```
int number_of_elements = sizeof(square) / sizeof(square[0]);
```

7. Arrays and strings

❖ C++ array types

▪ One-dimensional arrays

- It should be noted that C++ carry out ***no check*** on array indexing. Trying to access an element at an index that is outside the array bounds can lead to runtime errors, and tracing back these errors can take too much time.

```
double october [31];
october [-1] = 0;// error --> should be avoided
october [31] = 0;// error --> should be avoided
```

7. Arrays and strings

❖ C++ array types

▪ One-dimensional arrays

```
numbers[0] = 12.23
numbers[1] = 10.2
numbers[2] = 7.29
numbers[3] = 11.3
numbers[4] = 12.7

The average is 10.744
0.      12.23   -1.486
1.      10.2    0.544001
2.      7.29    3.454
3.      11.3    -0.556
4.      12.7    -1.956
```

```
#include <iostream>

using namespace std;

const int maxn = 5 ;
int main()
{
    float numbers[maxn], average = 0.0;
    for (int i = 0; i < maxn; i++)
    {
        cout << "numbers[" << i << "] = ";
        cin >> numbers[i];          // reading from keyboard
        average += numbers[i];      // sum of all elements
    }
    cin.get();
    average /= maxn;                // counting the average
    cout << endl << "The average is " << average << endl;

    // printing out differences
    for (int i = 0; i < maxn; i++) {
        cout << i << ".\t" << numbers[i];
        cout << '\t' << average-numbers[i] << endl;
    }
}
```

7. Arrays and strings

❖ C++ array types

▪ One-dimensional arrays

- Initializing and assigning values to one-dimensional arrays

element_typearray_name[*size*] = { *initialization list delimited by commas* };

```
int primes[10] = { 2, 3, 5, 7, 11, 13, 17, 19, 23, 27 };

char name[8] = { 'I', 'v', 'a', 'n'};

double numbers[] = { 1.23, 2.34, 3.45, 4.56, 5.67 };

int big[2012] = {0};
```

```
double numbers[] = { 1.23, 2.34, 3.45, 4.56, 5.67 };
const int number = sizeof(numbers) / sizeof(numbers[0]);
```

```
double eh[3]= { sqrt(2.3), exp(1.2), sin(3.14159265/4) };
```

7. Arrays and strings

❖ C++ array types

▪ One-dimensional arrays

- Initializing and assigning values to one-dimensional arrays

element_typearray_name[*size*] = { *initialization list delimited by commas* };

```
char line[80];
memset( line, '=', 80 );
double balance[365];
memset( balance, 0, 365*sizeof(double) );
//or
memset( balance, 0, sizeof(balance) );
```

7. Arrays and strings

❖ C++ array types

▪ One-dimensional arrays

There are two methods for value assignment between two arrays of the same type and size: **for** loop and ***memcpy()***.

```
#include <iostream>
#include <cstring>

using namespace std;

int main()
{
    const int maxn = 8 ;
    int source[maxn]= { 2, 10, 29, 7, 30, 11, 7, 12 };
    int destination[maxn];
    for (int i=0; i<maxn; i++) {
        destination[i] = source[i];// copying elements
    }
    // or
    memcpy(destination, source, sizeof(destination));
}
```

7. Arrays and strings

❖ C++ array types

▪ One-dimensional arrays

In the following example, a new element is intended to be inserted in the array *ordered* at the position with index *1*:

```
#include <iostream>
#include <cstring>

using namespace std;

int main()
{
    const int maxn = 10 ;
    int ordered[maxn]= { 2, 7, 12, 23, 29 };
    for (int i=5; i>1; i--) {
        ordered[i] = ordered[i-1];  // copying elements
    }
    ordered[1] = 3;
    // or
    memmove(ordered+2, ordered+1, 4*sizeof(int));
    ordered[1] = 3;
}
```

7. Arrays and strings

❖ C++ array types

▪ One-dimensional arrays

- One-dimensional arrays and the typedef

$$a = a_0 i + a_1 j + a_2 k$$

$$b = b_0 i + b_1 j + b_2 k$$

$$a \times b = \begin{vmatrix} i & j & k \\ a_0 & a_1 & a_2 \\ b_0 & b_1 & b_2 \end{vmatrix} = (a_1 \cdot b_2 - a_2 \cdot b_1) i - (a_0 \cdot b_2 - a_2 \cdot b_0) j + (a_0 \cdot b_1 - a_1 \cdot b_0) k$$

```
int a[3], b[3], c[3];
//or
typedef int vector3[3];
vector3 a, b, c;
//or
typedef int vector3[3];
vector3 a = {1, 0, 0}, b = {0, 1, 0}, c;
c[0] =   a[1]*b[2] - a[2]*b[1];
c[1] = -(a[0]*b[2] - a[2]*b[0]);
c[2] =   a[0]*b[1] - a[1]*b[0];
```

7. Arrays and strings

❖ C++ array types

▪ One-dimensional arrays

- One-dimensional arrays and the typedef

```
double *xp[12];//xp is an array of 12 elements and the array name is
preceded by the type of its elements
double (*xp)[12];//xp is a pointer and the type of the referenced data
is double[12], that is a double array of 12 elements
//it is more rapid and safe to use the keyword typedef:
typedef double dvect12[12];
dvect12 *xp;
double a[12];
xp = &a;
(*xp)[0]=12.3;
cout << a[0];    // 12.3
```

7. Arrays and strings

❖ C++ array types

▪ Two-dimensional arrays

element_type array_name[*size1*][*size2*];

$$\begin{bmatrix} 12 & 23 & 7 & 29 \\ 11 & 30 & 12 & 7 \\ 10 & 2 & 20 & 12 \end{bmatrix}$$

```
int matrix[3][4] = { { 12, 23,  7, 29 },
                     { 11, 30, 12,  7 },
                     { 10,  2, 20, 12 } };
```

r/c	0	1	2	3
0	12	23	7	29
1	11	30	12	7
2	10	2	20	12

matrix[1][2]

Graphical representation of a two-dimensional array

7. Arrays and strings

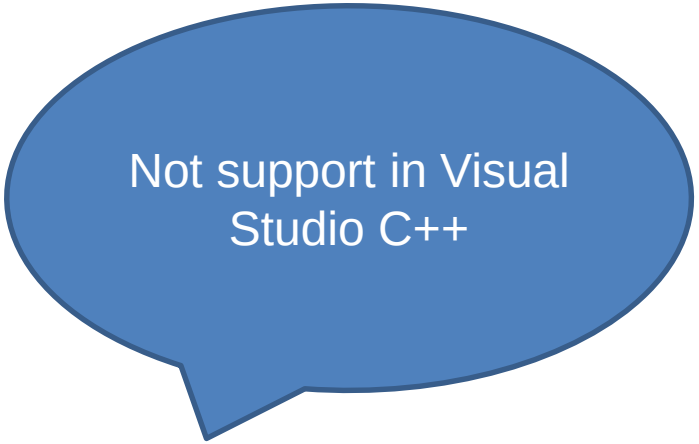
❖ C++ array types

- **Two-dimensional arrays**

```
int maxe, mine;
maxe = mine = matrix[0][0];
for (int i = 0; i < 3; i++)
{
    for (int j = 0; j < 4; j++)
    {
        if (matrix[i][j] > maxe )
            maxe = matrix[i][j];
        if (matrix[i][j] < mine )
            mine = matrix[i][j];
    }
}

for (int i = 0; i < 3; i++)
{
    for (int j = 0; j < 4; j++)
        cout << '\t' << matrix[i][j];
    cout << endl;
}
```

7. Arrays and strings

- **C++ array types**
  - **Variable-length arrays**



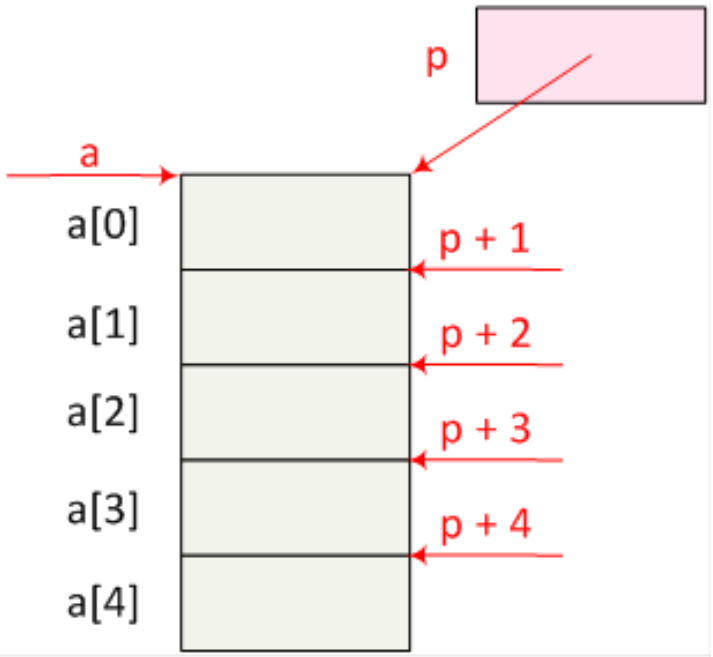

```
#include <iostream>

using namespace std;

int main() {
    int size;
    cout << "The number of elements of the vector: ";
    cin >> size;
    int vector[size];
    for (int i=0; i<size; i++)
    {
        vector[i] = i*i;
    }
}
```

7. Arrays and strings

❖ C++ array types

- **The relationship between pointers and arrays**

```
int a[5];

int *p;
p = &a[0];  or  p = a;
```



The relationship between pointers and arrays

7. Arrays and strings

❖ C++ array types

▪ The relationship between pointers and arrays

- There is an important difference between the two pointers: pointer *p* is a variable (its value can therefore be modified any time), while *a* is a constant value pointer that the compiler fixes in memory.
- The address of the *i*th element:

```
&a[i]    &p[i]    a+i p+i
```

- The *0*th element of the array:

```
a[0]    p[0]    *a    *p    *(a+0)    *(p+0)
```

- The *i*th element of the array:

```
a[i]    p[i]    *(a+i)    *(p+i)
```

7. Arrays and strings

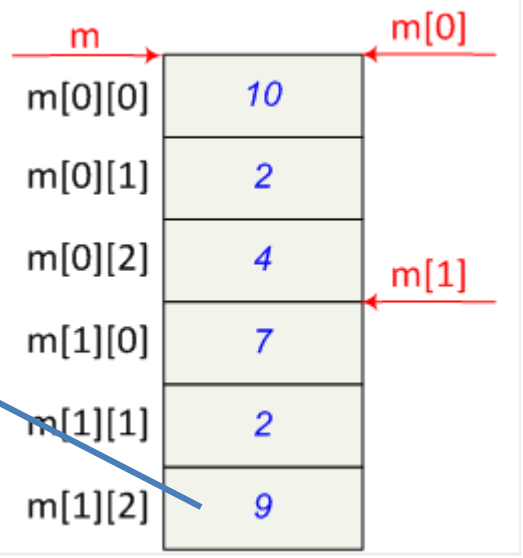
❖ C++ array types

▪ The relationship between pointers and arrays

- In the case of multi-dimensions, analogy is only formal; however, it can often help using correctly more complex data structures. Let's see the following **double** type matrix:

```
double m[2][3] = { { 10, 2, 4 },
                   { 7,  2, 9 } };
```

```
m[1][2]

*(m[1] + 2)

*(*(m+1)+2)
```



Two-dimensional arrays in memory

7. Arrays and strings

❖ Dynamically allocated arrays

▪ One-dimensional dynamic arrays

- It is one-dimensional dynamic arrays that are the most frequently used in programming.

```
type *pointer;

pointer = new type [number_of_elements];

or

pointer = new (nothrow) type [number_of_elements];
```

- In the first case, the compiler throws an exception (***bad_alloc***) if there is not enough available contiguous memory
- In the second case, a pointer with the value of *0* is returned.

7. Arrays and strings

❖ Dynamically allocated arrays

▪ One-dimensional dynamic arrays

- In case of arrays, it is extremely important not to forget about deleting allocated memory:

```
delete [] pointer;
```

```
#include <iostream>
#include <exception>
#include <ctime>
#include <cstdlib>

using namespace std;

int main() {
   long *data, size;
   cout << "\nEnter the array size: ";
   cin >> size;
   cin.get();

   // Memory allocation
   try {
     data = new long [size];
   }
   catch (bad_alloc) {
     // Unsuccessful allocation
     cerr << "\nThere is not enough memory!\n" << endl;
     return -1;  // Program is exited
   }
   // Filling up the array with random numbers
   srand(unsigned(time(0)));
   for (int i=0; i<size; i++) {
      data[i] = rand() % 2012;
      // or  *(data+i) = rand() %2012;
   }
   // Deleting (freeing) allocated memory space
   delete[] data;
   return 0;
}
```

7. Arrays and strings

❖ Dynamically allocated arrays

▪ Two-dimensional dynamic arrays

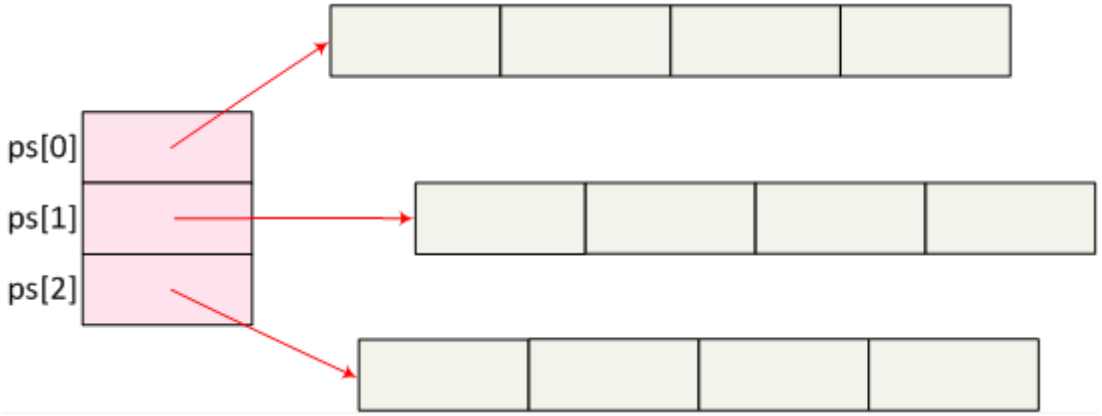
```
const int rowlength = 4;
int number_of_rows;
cout<<"Number of rows: "; cin >> number_of_rows;
int (*mp)[rowlength] = new int [number_of_rows][rowlength];

//or

const int rowlength = 4;
int number_of_rows;
cout << "Number of rows: "; cin >> number_of_rows;
typedef int rowtype[rowlength];
rowtype *mp = new rowtype [number_of_rows];
```

❖ Dynamically allocated arrays

▪ Two-dimensional dynamic arrays

- For both solutions, setting all elements to zero can be carried out by the following loops:

```
for (int i=0; i<number_of_rows; i++)
    for (int j=0; j<rowlength; j++)
        mp[i][j]=0;
```



Dynamically allocated row vectors

7. Arrays and strings

❖ Dynamically allocated arrays

▪ Two-dimensional dynamic arrays

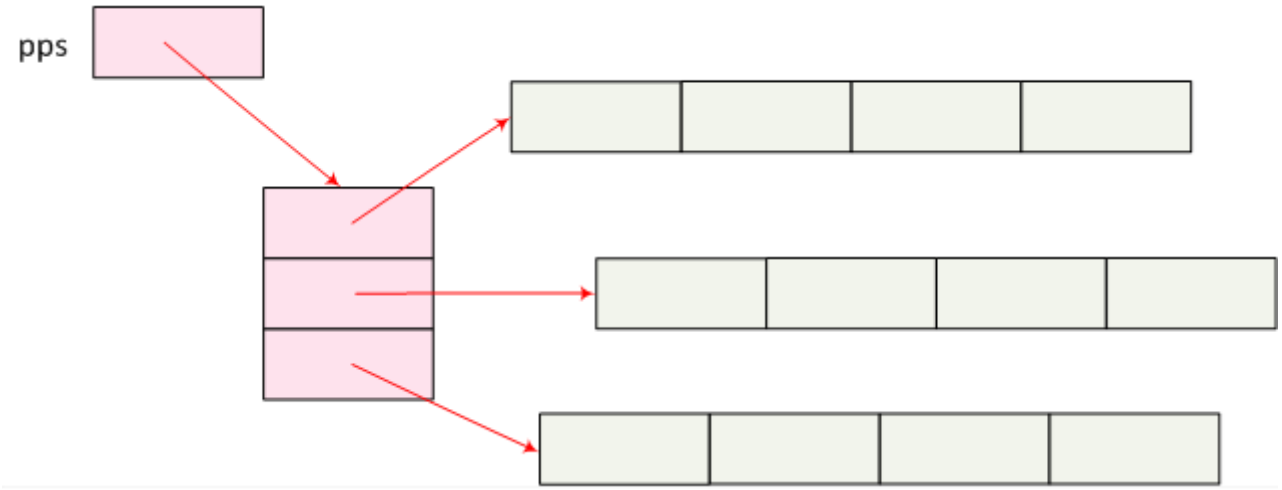
- Memory allocation, access and deallocation can easily be traced in the following example code:

```
#include <iostream>

using namespace std;

int main() {
    int number_of_rows, rowlength;
    cout<<"Number of rows: "; cin >> number_of_rows;
    cout<<"The length of rows: "; cin >> rowlength;

    // Memory allocation
    int* *pps;
    // Defining the pointer vector
    pps = new int* [number_of_rows];

    // Allocating rows
    for (int i=0; i< number_of_rows; i++)
        pps[i] = new int [rowlength];

    // Accessing the array
    for (int i=0; i<number_of_rows; i++)
        for (int j=0; j<rowlength; j++)
            pps[i][j]=0;

    // Deleting (freeing) allocated memory space
    for (int i=0; i<number_of_rows; i++)
        delete pps[i];
    delete[] pps;
}
```

7. Arrays and strings

❖ Dynamically allocated arrays

▪ Two-dimensional dynamic arrays



Dynamically allocated pointer vector and row vectors

7. Arrays and strings

❖ The usage of the vector type

▪ One-dimensional arrays in vectors

- As its name suggests, vector type replaces one-dimensional arrays.

```
vector<int> ivector;
ivector.resize(10);

vector<long> lvector(12);
vector<float> fvector(7, 1.0);
```

ivector is an empty vector, the size of which is set by the function ***resize()*** to *10*.

lvector contains 12 elements of type **long**. In both cases, all elements are initialized to 0.

fvector is created with 7 elements of type **float**, the values of which are all initialized to *1.0*.

❖ The usage of the vector type

▪ One-dimensional arrays in vectors

- The actual number of elements can be obtained by the function ***size*** *()*. An important feature of ***vector*** type is the function ***push_back*** *()* that adds an element to the vector.

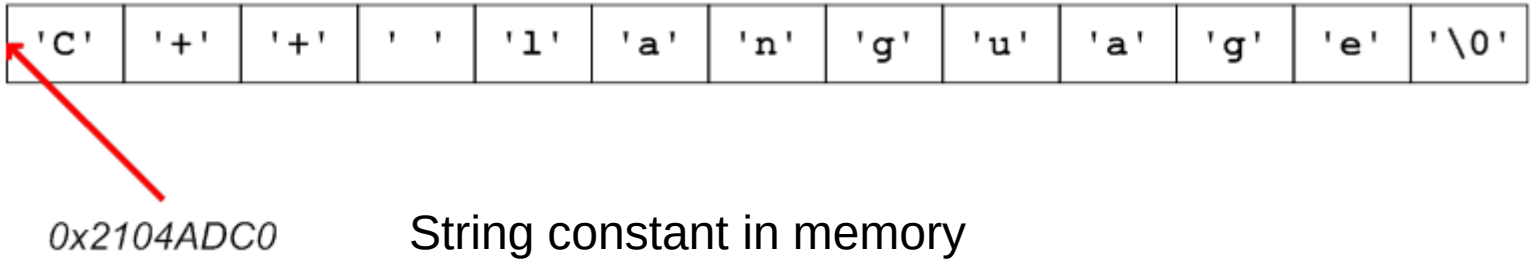
```
#include <iostream>
#include <vector>

using namespace std;

int main() {
    vector<int> ivector(10, 5);

    ivector.push_back(10);
    ivector.push_back(2);

    for (unsigned index=0; index<ivector.size(); index++) {
        cout << ivector[index] << endl;
    }
}
```

7. Arrays and strings

❖ Handling C-style strings

- Two-dimensional, dynamic arrays can be created by nesting vector types, and it is much easier than the methods mentioned before.

```
cout << "C++ language";
```



String constant in memory

❖ Handling C-style strings

- Strings composed of wide characters are also stored in that way but in this case the type of the elements of the array is **wchar_t**.

```
wcout << L"C++ language";
```

- In C++, the types ***string*** and ***wstring*** can also be used to process texts, so we give an overview of these types, too.

7. Arrays and strings

❖ Handling C-style strings

▪ Strings in one-dimensional arrays

- If a text of at most *80* characters is intended to be stored in the array named *str* then its size should be *80+1=81*:

```
char line[81];
```

- In programming tasks, we often use strings having an initial value.

```
char st1[10] = { 'O', 'm', 'e', 'g', 'a', '\0' };

wchar_t wst1[10] = { L'O', L'm', L'e', L'g', L'a', L'\0' };

char st2[] = { 'O', 'm', 'e', 'g', 'a', '\0' };

wchar_t wst2[] = { L'O', L'm', L'e', L'g', L'a', L'\0' };
```

7. Arrays and strings

❖ Handling C-style strings

▪ Strings in one-dimensional arrays

- Initializing character arrays is much safer by using string literals (string constants):

```
char st1[10] = "Omega";     wchar_t wst1[10] = L"Omega";

char st2[] = "Omega";         wchar_t wst2[] = L"Omega";
```

Operation	Function (char)	Function (wchar_t)
reading text from the keyboard	***cin*** >>, ***cin*** . ***get*** (), ***cin*** . ***getline*** ()	***wcin*** >>, ***wcin*** . ***get*** (), ***wcin*** . ***getline*** ()
printing out a text	***cout*** <<	***wcout*** <<
value assignment	***strcpy*** (), ***strncpy*** ()	wcscpy(), wcsncpy()
concatenation	***strcat*** (), ***strncat*** ()	wcscat(), wcsncat()
getting the length of a string	***strlen*** ()	wcslen()
comparison of strings	***strcmp*** (), ***strcnmp*** ()	wcscmp(), wcsncmp()
searching for a character in a string	***strchr*** ()	wcschr()

7. Arrays and strings

❖ Handling C-style strings

▪ Strings in one-dimensional arrays

```
#include <iostream>
#include <cstring>
#include <cctype>

using namespace std;

int main() {
    char s[80];
    cout << "Type in a text: ";
    cin.get(s, 80);
    cout << "The read text: " << s << endl;
    for (int i = strlen(s)-1; i >= 0; i--)
        cout << (char)toupper(s[i]);
    cout << endl;
}
```

```
#include <iostream>
#include <cwchar>
#include <cwctype>

using namespace std;

int main() {
    wchar_t s[80];
    wcout << L"Type a text: ";
    wcin.get(s, 80);
    wcout << L"The read text: " << s << endl;
    for (int i = wcslen(s)-1; i >= 0; i--)
        wcout << (wchar_t)towupper(s[i]);
    wcout << endl;

    return 0;
}
```

❖ Handling C-style strings

▪ Strings and pointers

- Character arrays and character pointers can both be used to manage strings but pointers should be used more carefully. Let's see the following frequent definitions.

```
char str[16] = "alfa";

char *pstr = "gamma";
```

- The value of the pointer *pstr* can be modified later (which causes the loss of the string "*gamma*"):

```
pstr = "iota";
```

- A pointer value assignment takes place here since *pstr* now points to the address of the new string literal. On the contrary, if it is the name of the array *str* to which a value is assigned, an error message is obtained:

```
str = "iota";    // error! --> should be avoided
```

7. Arrays and strings

❖ Handling C-style strings

▪ Strings and pointers

```
#include <iostream>

using namespace std;

const unsigned char key = 0xCD;
int main() {
    char s[80], *p;
    cout <<"Type in a text: ";
    cin.get(s, 80);

    for (int i = 0; s[i]; i++)// encryption
        s[i] ^= key;
    cout << "The encrypted text: "<< s << endl;

    p = s;
    while (*p)// decryption
        *p++ ^= key;
    cout << "The original text: "<< s << endl;

    return 0;
}
```

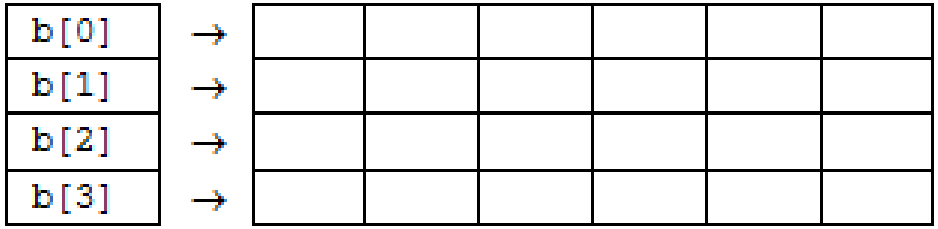
```
char *sp = new char [33];
strcpy(sp, "C++");
cout << ++*sp << endl;      // D
cout << sp << endl;         // D++
cout << *sp++ << endl;      // D
cout << sp << endl;         // ++
```

7. Arrays and strings

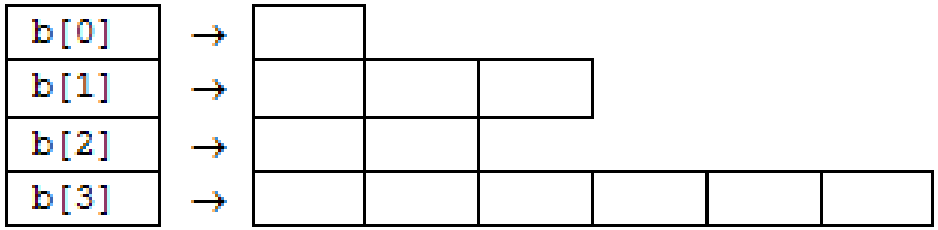
❖ Handling C-style strings

- **Using string arrays**

```
int s1[6], s2[6], s3[6], s4[6];
int* b[4] = { s1, s2, s3, s4 };
```




```
int s1[1], s2[3], s3[2], s4[6];
int* b[4] = { s1, s2, s3, s4 };
```

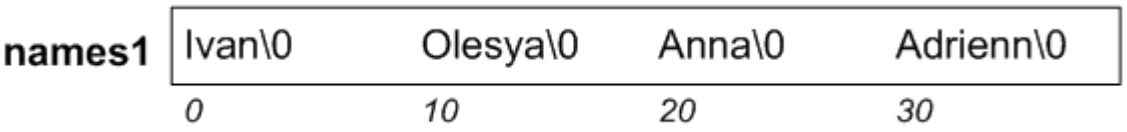


7. Arrays and strings

❖ Handling C-style strings

▪ Using string arrays

```
static char names1[][10] = { "Ivan",
                             "Olesya",
                             "Anna",
                             "Adrienn" };
```



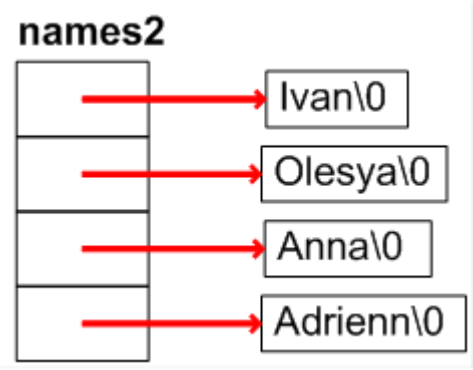
String array stored in a two-dimensional array

7. Arrays and strings

❖ Handling C-style strings

▪ Using string arrays

```
static char* names2[] = { "Ivan",
                          "Olesya",
                          "Anna",
                          "Adrienn" };
```



Optimally stored string array

```
cout << names1[0] << endl;        // Ivan
cout << names1[1][4] << endl;     // y
cout << names2[0] << endl;        // Ivan
cout << names2[1][4] << endl;     // y
```

7. Arrays and strings

❖ Handling C-style strings

▪ The string type

Operation	C++ solution - string	C++ solution - wstring
reading text from the keyboard	***cin>>*** , ***getline*** *()*	***wcin>>*** , ***getline*** *()*
printing out a text	cout<<	***wcout<<***
value assignment	**=** , ***.assign*** *()*	**=** , ***.assign*** *()*
concatenation	**+** , **+=**	**+** , **+=**
accessing the characters of a string	[]	***[]***
getting the length of a string	***.size*** *()*	***.size*** *()*
comparison of strings	***.compare*** *()*, **==** , **!=** , **<** , **<=** , **>** , **>=**	***.compare*** *()*, **==** , **!=** , **<** , **<=** , **>** , **>=**
conversion into C-style character sequence	***.c_str*** *()*, ***.data*** *()*	***.c_str*** *()*, ***.data*** *()*

7. Arrays and strings

❖ Handling C-style strings

▪ The string type

```
#include <string>
#include <iostream>

using namespace std;

const unsigned char key = 0xCD;
int main() {
    string s;
    char *p;
    cout << "Type in a text    : ";
    getline(cin, s);

    for (int i = 0; s[i]; i++)      // encryption
        s[i] ^= key;
    cout << "The encrypted text: " << s << endl;

    p=(char *)s.c_str();
    while (*p)                      // decryption
        *p++ ^= key;
    cout << "The original text: " << s << endl;

    return 0;
}
```

```
#include <string>
#include <iostream>
#include <cwchar>

using namespace std;

const unsigned wchar_t key = 0xCD;
int main()
{
    wstring s;
    wchar_t *p;
    wcout << L"Type a text  : ";
    getline(wcin, s);

    for (int i = 0; s[i]; i++)      // encryption
        s[i] ^= key;
    wcout << L"The encrypted text: " << s << endl;

    p =(wchar_t *)s.c_str();
    while (*p)                      // decryption
        *p++ ^= key;
    wcout << L"The original text: " << s << endl;

    return 0;
}
```

Chapter I. Basics and data management of C++

- 1. Creation of C++ programs
- 2. Basic data types, variables and constants
- 3. Basic operations and expressions
- 4. Control program structures
- 5. Exception handling
- 6. Pointers, references and dynamic memory management
- 7. Arrays and strings
- 8. User-defined data types

8. User-defined data types

- The structure type
- The class type
- The union type
- Bit fields

8. User-defined data types

❖ The structure type

▪ Structure type and structure variables

- The general declaration of structures is as follows:

```
struct  structure_type {
    type 1 member 1 ;// without an initial value!

    type 2 member 2 ;

    . . .

    type n member n ;
};
```

A structure variable (a structure) of the type above can be created by the already known method:

```
struct  structure_type structure_variable;// C/C++

structure_type structure_variable;        // only in C++
```

8. User-defined data types

❖ The structure type

▪ Structure type and structure variables

- In C++, the name standing after the keywords **struct**, **union** and **class** can be used as type names without using the keywords. When **typedef** is used, the difference between the two programming languages disappears:

```
typedef struct {

    type 1 member 1 ;

    type 2 member 2 ;

    . . .

    type n member n ;

} structure_type;

structure_type structure_variable;// C/C++
```

8. User-defined data types

❖ The structure type

▪ Structure type and structure variables

- The lists of the expressions, separated from one another by commas, initializing the data members should be enclosed within curly brackets.

structure_type structure_variable= {*initial_value_list*}; // C/C++

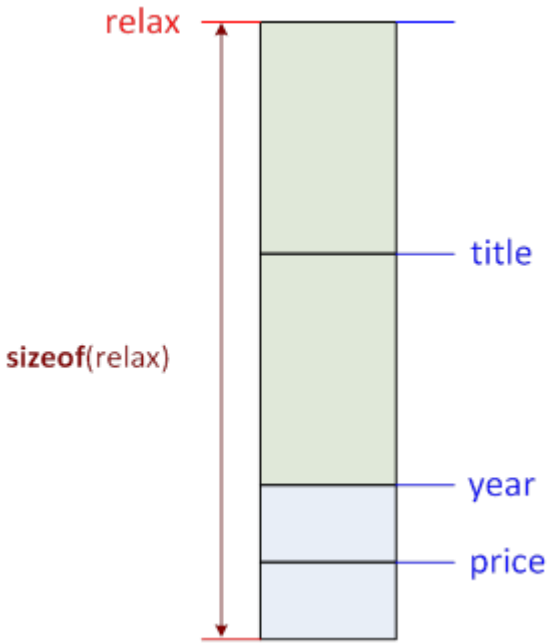
```
struct musicCD {
    // the performer and the title of the CD
    string performer, title;
    int year;              // year of publication
    int price;          // price of the CD
};
```

8. User-defined data types

❖ The structure type

▪ Structure type and structure variables

```
musicCD classic = {"Vivaldi","The Four Seasons", 2009, 2590};
musicCD *pmusic = 0, rock, tale = {};
musicCD relax;
```



Structure in memory

8. User-defined data types

❖ The structure type

▪ Accessing the data members of structures

- Let's define some variables by using the type *musicCD* declared before.

```
musicCD s1, s2, *ps;

ps = &s1;
//or
ps = new (nothrow) musicCD;
if (!ps) exit(-1);
// ...
delete ps;
```

- How to give values to data members

```
s1.performer = "Vivaldi";
s1.title     = "The Four Seasons";
s1.year      = 2005;
s1.price     = 2560;
```

```
(*ps).performer = "Vivaldi";
(*ps).title     = "The Four Seasons";
(*ps).year      = 2005;
(*ps).price     = 2560;
```

```
ps->performer = "Vivaldi";
ps->title     = "The Four Seasons";
ps->year      = 2005;
ps->price     = 2560;
```

8. User-defined data types

❖ The structure type

▪ Accessing the data members of structures

- Accordingly, the meaning of the expression *ps->price* is: "*the data member named price within the structure to which the pointer ps points*".

```
s2.performer = s1.performer;
s2.title     = s1.title;
s2.year      = s1.year;
s2.price     = s1.price;
```

```
s2  = s1 ;// this corresponds to
         //the 4 assignments above
*ps = s2 ;
s1  = *ps = s2 ;
```

8. User-defined data types

```
#include <iostream>
#include <string>

using namespace std;

struct musicCD {
        string performer, title;
        int year, price;
};

int main() {

   musicCD cd;

   // reading the data from the keyboard
   cout<<"Please type in the data of the music CD." << endl;
   cout<<"Performer           : ";    getline(cin, cd.performer);
   cout<<"Title               : ";    getline(cin, cd.title);
   cout<<"Year of publication : ";    cin>>cd.year;
   cout<<"Price               : ";    cin>>cd.price;
   cin.get();

   // printing out the data
   cout<<"\nData of the music CD:" << endl;
   cout<<"Performer           : ";    cout << cd.performer << endl;
   cout<<"Title               : ";    cout << cd.title << endl;
   cout<<"Year of publication : ";    cout << cd.year << endl;
   cout<<"Price               : ";    cout << cd.price << endl;
}
```

8. User-defined data types

❖ The structure type

▪ Nested structures

```
struct date {
    int year, month, day;
};

struct person {
    string name;
    date birthday;
};

person brother = { "Ivan", {2004, 10, 2} };
person student;

student.name = "Bill King";
student.birthday.year = 1990;
student.birthday.month = 10;
student.birthday.day = 20;
```

8. User-defined data types

❖ The structure type

▪ Nested structures

- If the structure of type *date* is not used anywhere else then it can be integrated directly as an anonymous structure in the structure *person*:

```
struct person {
    string name;
    struct {
        int year, month day;
    } birthday;
};
```

- Self-referential structures

```
struct list_element {
    double data_member;
    list_element *link;
};
```

8. User-defined data types

❖ The structure type

▪ Structures and arrays

- Arrays as data members of a structure

```
const int maxn = 10;
struct svector {
    int v[maxn];
    int n;
};

svector a = {{23, 7, 12}, 3};
svector b = {{0}, maxn};
svector c = {};

int sum=0;
for (int i=0; i<a.n; i++) {
    sum += a.v[i];
}

c = a;
```

```
svector *p = new svector;
p->v[0] =  2;
p->v[1] = 10;
p->n    =  2;
delete p;
```

8. User-defined data types

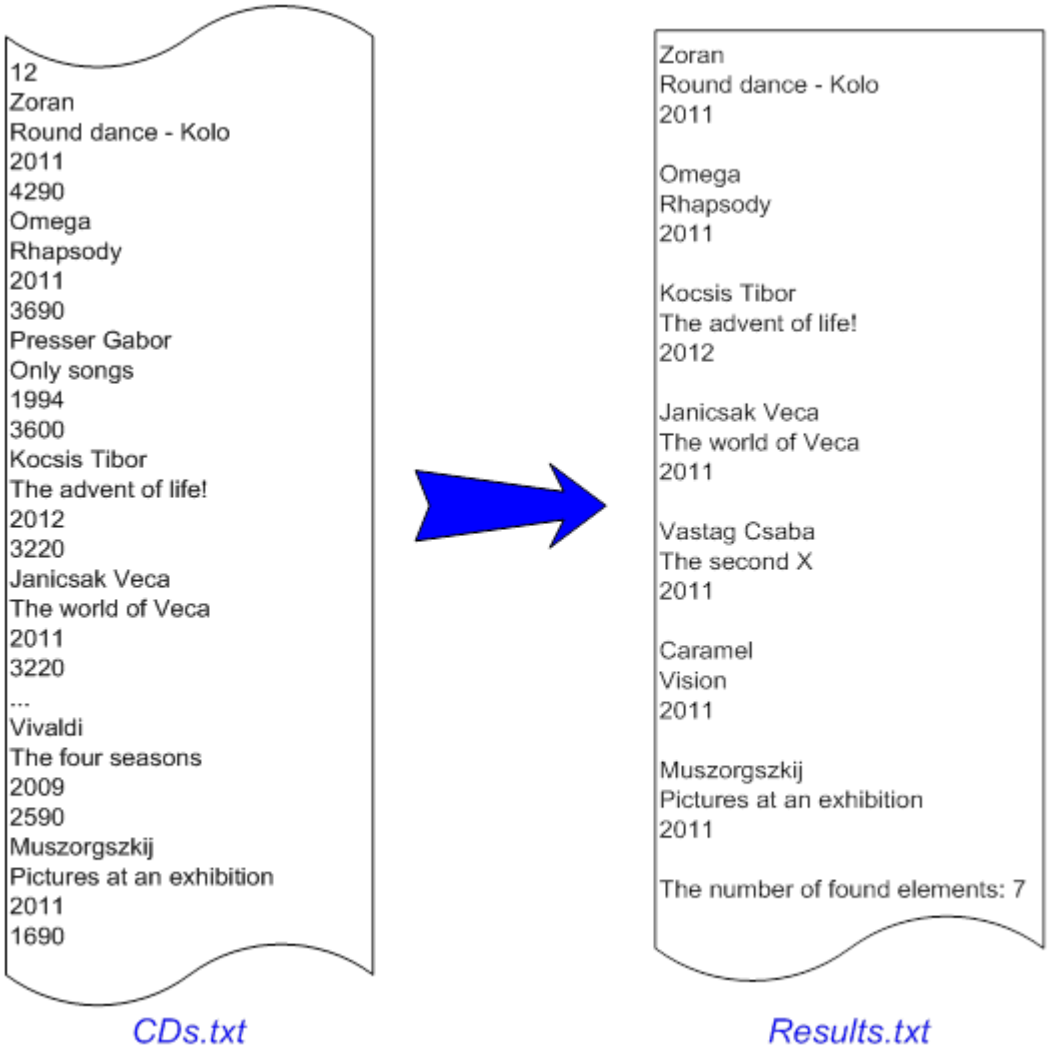
❖ The structure type

▪ Structures and arrays

- Structures as array elements

```
#include <iostream>
#include <string>
using namespace std;

struct musicCD {
    string performer, title;
    int year, price;
};

int main() {
    cout<<"The number of CDs:";
    int num;
    cin>>num;
    cin.ignore(80, '\n');

    // Memory allocation with checking
    musicCD *pCDcatalog = new (nothrow) musicCD[num];
    if (!pCDcatalog) {
        cerr<<"\a\nThere is not enough memory!\n";
        return -1;
    }

    // Reading the data of CDs from the keyboard
    for (int i=0; i<num; i++) {
        cout<<endl<<"The data of the "<<i<<"th CD:"<<endl;
        cout<<"Performer: ";
                getline(cin, pCDcatalog[i].performer);
        cout<<"Title:     "; getline(cin, pCDcatalog[i].title);
        cout<<"Year:      "; cin>>pCDcatalog[i].year;
        cout<<"Price:      "; cin>>pCDcatalog[i].price;
        cin.ignore(80, '\n');
    }

    // Searching for the requested CDs
    int found = 0;
    for (int i = 0; i < num; i++) {
        if (pCDcatalog[i].year >= 2010 &&
                pCDcatalog[i].year <= 2012) {
            cout<<endl<<pCDcatalog[i].performer<<endl;
            cout<<pCDcatalog[i].title<<endl;
cout<<pCDcatalog[i].year<<endl;
            found++;
        }
    }

    // Printing out the results
    if (found)
        cout<<"\nThe number of found elements: "<<found<<endl;
    else
        cout<<"There is no CD that matches the criteria!"<<endl;

    // Deallocating memory space
    delete [] pCDcatalog;
}
```

8. User-defined data types



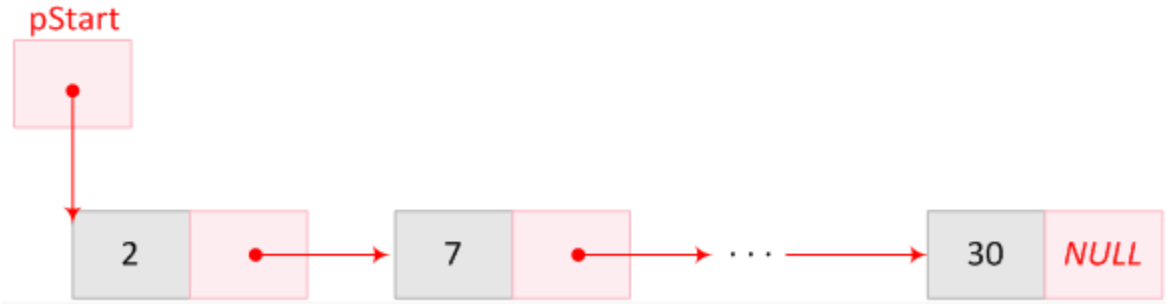
Processing data in the program CDCatalogue

8. User-defined data types

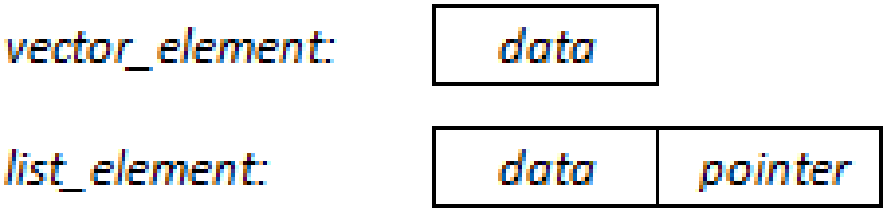
❖ The structure type

▪ Creating singly linked lists

- The simplest forms of linked lists are singly linked lists in which all elements possess a reference to the next list element. The reference in the last element has the value null.



A singly linked list



8. User-defined data types

❖ The structure type

▪ Creating singly linked lists

- C++ list elements can be created by the already presented self-referential structure.

```
struct list_element {
    int data;
    list_element *pnext;
};
```

- Since we allocate memory for each new list element in this example, this operation is carried out by a function to be presented in the next chapter of the present book:

```
list_element *NewElement(int data) {
    list_element *p = new (nothrow) list_element;
    assert(p);
    p->data = data;
    p->pnext = NULL;
    return p;
}
```

8. User-defined data types

❖ The structure type

▪ Creating singly linked lists

- *Creating the list* and filling up it from the array *data*.

```
int data [] = {2, 7, 10, 12, 23, 29, 30};
const int num_of_elements = sizeof(data)/sizeof(data[0]);

list_element *pStart = NULL, *pActual, *pPrev, *pNext;

for (int index = 0; index<num_of_elements; index++) {
    pNext = NewElement(data[index]);
    if (pStart==NULL)
       pActual = pStart = pNext ;          // first element
    else
       pActual = pActual->pnext = pNext;    // not a first element
}
pActual->pnext = NULL;     // closing the list
// list: pStart → 2 → 7 → 10 → 12 → 23 → 29 → 30
```

8. User-defined data types

❖ The structure type

▪ Creating singly linked lists

- *printing out the elements of the list*.

```
pActual = pStart;
while (pActual != NULL) {
    cout<< pActual->data << endl;
    // stepping to the next element
    pActual = pActual->pnext;
}
```

- *remove an element from a list*.

```
// identifying the place of the element having the index 4 (23)
pActual = pStart;
for (int index = 0; index<4; index++) {
    pPrev    = pActual;
    pActual = pActual->pnext;
}
// removing the element of the linked list
pPrev->pnext = pActual->pnext;
// deallocating memory space
delete pActual;
// the list: pStart → 2 → 7 → 10 → 12 → 29 → 30
```

8. User-defined data types

❖ The structure type

▪ Creating singly linked lists

- *inserting a new element to a list*.

```
// determining the place of the preceding element of index 3 (12)
pActual = pStart;
for (int index = 0; index<3; index++)
    pActual = pActual->pnext;
// allocating memory for the new element
pNext = NewElement(23);
// inserting the new element in the linked list
pNext->pnext = pActual->pnext;
pActual->pnext = pNext;
// list: pStart → 2 → 7 → 10 → 12 → 23 → 29 → 30
```

8. User-defined data types

❖ The structure type

▪ Creating singly linked lists

- *add a new element* to (the end of) the list.

```
// searching for the last element
pActual = pStart;
while (pActual->pnext!=NULL && (pActual = pActual->pnext));
// allocating memory for the new element
pNext = NewElement(80);
// adding the new element to the end of the list
pActual->pnext = pNext;
// the list: pStart → 2 → 7 → 10 → 12 → 23 → 29 → 30 → 80
```

8. User-defined data types

❖ The structure type

▪ Creating singly linked lists

- *search for an element* of a given value (*sdata*) in the list.

```
int sdata = 29;
pActual = pStart;
while (pActual->data!=sdata && (pActual = pActual->pnext));
if (pActual!=NULL)
    cout<<"Found: "<<pActual->data<< endl;
else
    cout<<" Not found!"<<endl;
```

- *delete* the elements of a *list*.

```
pActual = pStart;
while (pActual != NULL) {
    pNext = pActual->pnext;
    delete pActual;
    pActual = pNext;
}
pStart = NULL;  // there is no list element!
```

8. User-defined data types

❖ The class type

- A class can contain member functions besides its data members.

```
struct time {
    int hour;
    int minute;
    int second;
};

struct time {
 private:
    int hour;
    int minute;
    int second;
};
```

```
class time {
 public:
    int hour;
    int minute;
    int second;
};

class time {
    int hour;
    int minute;
    int second;
};
```

8. User-defined data types

❖ The class type

- The definitions of variables of a **struct** or **class** type can only contain initial values, if the given class type only has public data members.

```
class time {
 public:
     int hour;
     int minute;
     int second;
};

int main() {
     time beginning ={7, 10, 2};
}
```

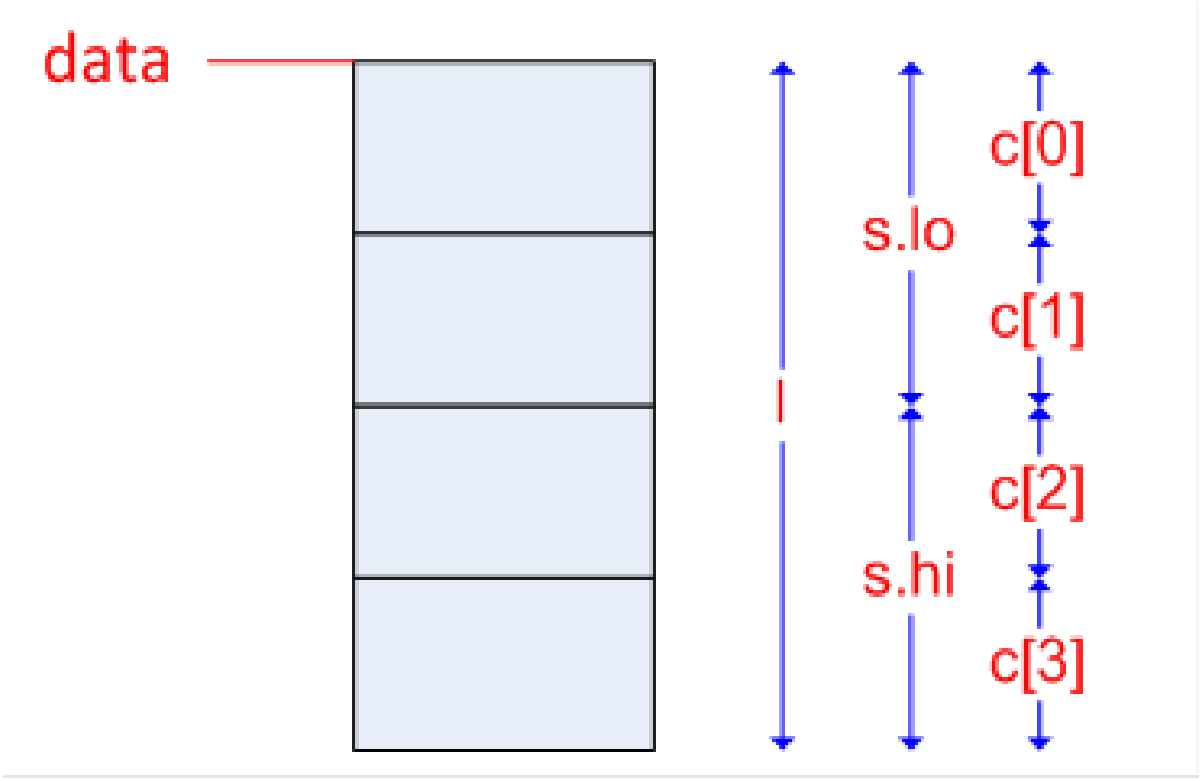
8. User-defined data types

❖ The union type

```
#include <iostream>
using namespace std;

union conversion {
  unsigned long l;
  struct {
    unsigned short lo;
    unsigned short hi;
    } s;
  unsigned char c[4];
};

int main()
{
    conversion data = { 0xABCD1234 };
    cout<<hex<<data.s.lo<<endl;          // 1234
    cout<<data.s.hi<<endl;              // ABCD
    for (int i=0; i<4; i++)
        cout<<(int)data.c[i]<<endl;     // 34 12 CD AB

    data.c[0]++;
    data.s.hi+=2;
    cout <<data.l<<endl;               // ABCF1235
}
```

8. User-defined data types

❖ The union type



Union in memory

8. User-defined data types

❖ The structure type

- **Anonymous unions**

```
static union {
    long a;
    double b;
};

int main() {
    union {
        char c[4];
        float f;
    };

    a = 2012;
    b = 1.2;    // the value of a changed!
    f = 0;
    c[0] = 1;    // the value of f changed!
}
```

8. User-defined data types

- ❖ **The structure type**
- ■ **Anonymous unions**

```
#include <iostream>
using namespace std;

struct vrecord  {
    char type;
    char name[25];
    union {
        char address[50];
        unsigned long ID;
    };                  // <---- there is no member name!
};

int main() {
    vrecord vr1={0,"BME","Budapest, Muegyetem rkpt 3-11."};
    vrecord vr2={1, "National Bank"};
    vr2.ID=3751564U;
    for (int i=0; i<2; i++) {
        cout<<"Name    : "<<vr1.name<<endl;
        switch (vr1.type) {
            case 1 :
                cout<<"ID      : "<<vr1.ID<<endl;
                break;
            case 0 :
                cout<<"Address : "<<vr1.address<<endl;
                break;
            default  :
                cout<<"Not a valid data type!"<<endl;
        }
        vr1 = vr2;
    }
}
```

Name   : BME
Address : Budapest, Muegyetem rkpt 3-11.
Name   : National Bank
ID   : 3751564

8. User-defined data types

❖ Bit fields

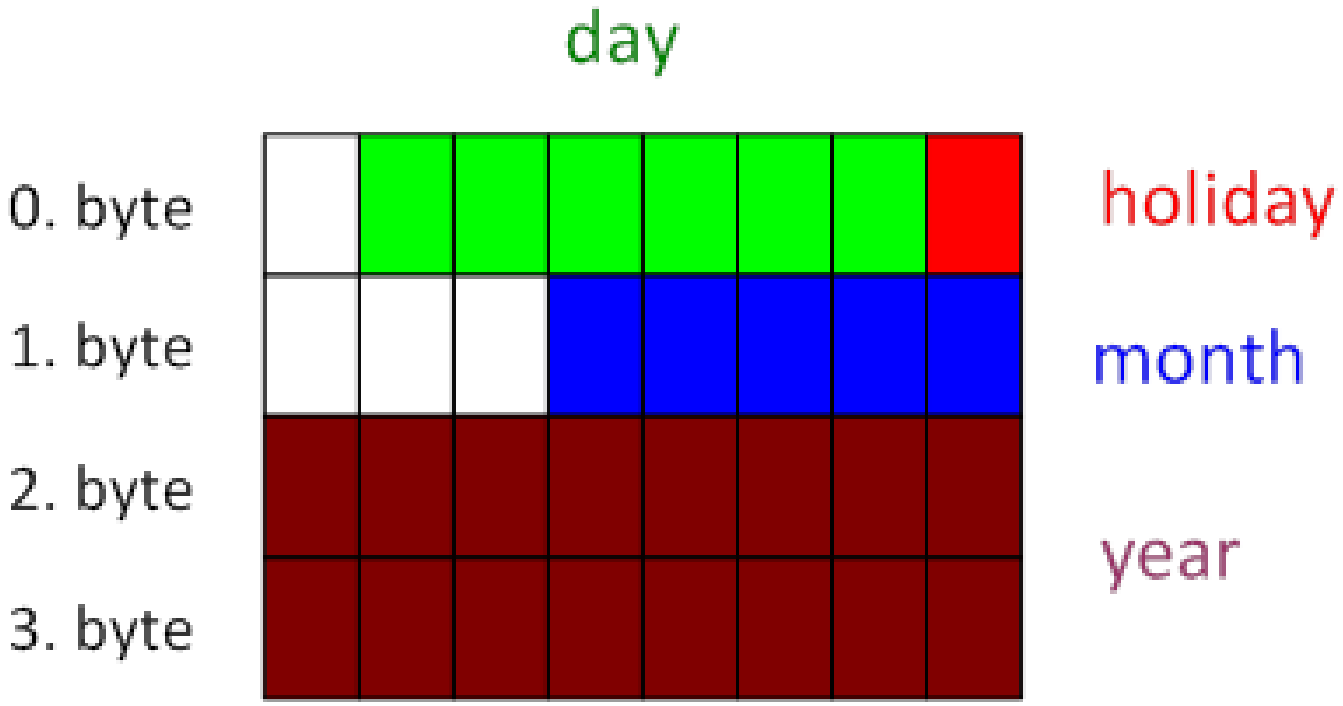
- Classes and structures may contain members for which compilers use a space less than for integer types. Since the storage space is determined by number of bits for these members, they are called bit fields. The general declaration of bit fields:

type name_of_bitfield : *bitlength*;

```
#include <iostream>
using namespace std;

#pragma pack(1)
struct date {
   unsigned char holiday    : 1;    // 0..1
   unsigned char day        : 6;    // 0..31
   unsigned char month        : 5;    // 0..16
   unsigned short year;
};

int main() {
    date today = { 0, 2, 10, 2012 };
    date holiday = {1};
    holiday.year = 2012;
    holiday.month = 12;
    holiday .day = 25;
}
```

8. User-defined data types

❖ Bit fields



The layout of the structure date in memory

8. User-defined data types

- **Bit fields**

```
#include <iostream>
#include <conio.h>
using namespace std;

union LCR {
    struct {
        unsigned char datalength : 2;
        unsigned char stopbits   : 1;
        unsigned char parity     : 3;
        unsigned char            : 2;
    } bsLCR;
    unsigned char byLCR;
};

enum RS232port {eCOM1=0x3f8, eCOM2=0x2f8 };

int main() {
    LCR reg = {};
    reg.bsLCR.datalength = 3; // 8 data bits
    reg.bsLCR.stopbits = 0;   // 1 stopbit
    reg.bsLCR.parity   = 0;   // no parity
    outport(eCOM1+3, reg.byLCR);
}
```

KỸ THUẬT LẬP TRÌNH HỆ CƠ ĐIỆN TỬ

Programming Engineering in Mechatronics

Giảng viên: TS. Nguyễn Thành Hùng

Đơn vị: Bộ môn Cơ điện tử, Viện Cơ khí

Hà Nội, 2020

Chapter II. Modular programming in C++

- Structured programming relies on *top-down* design.
- In C and C++ languages, the smallest structural unit having independent functionality is called ***function***.
- If functions or a group of functions belonging together are put in a separate module (source file), ***modular programming*** is realised.
- Structural programming also contributes to creating new programs from achieved modules (components) by *bottom-up* design.
- This chapter aims to introduce the modular and ***procedural programming*** in C++.

Chapter II. Modular programming in C++

- The basics of functions
- How to use functions on a more professional level?
- Namespaces and storage classes
- Preprocessor directives of C++

Chapter II. Modular programming in C++

- The basics of functions
- How to use functions on a more professional level?
- Namespaces and storage classes
- Preprocessor directives of C++

The basics of functions

- Defining, calling and declaring functions
- The return value of functions
- Parametrizing functions
- Programming with functions

The basics of functions

- In C++, a function is a unit (a subprogram) that has a name and that can be called from the other parts of a program as many times as it is needed.
- In order to use a function efficiently, some of its inner variables (***parameters***) are assigned a value when the function is called.
- When a function is called (activated), the values (**arguments**) to be assigned to each parameter have to be passed in a similar way.
- The called function passes control back to the place where it was called by a **return** statement.
- The value of the expression in the **return** statement is the ***return value*** returned back by the function, which is the result of the function call expression.

The basics of functions

❖ Defining, calling and declaring functions

- C++ Standard Library provides us many useful predefined functions.

function	header file
sqrt()	cmath
isalpha()	cctype
atoi()	cstdlib
rand()	cstdlib
strlen()	cstring
wcslen()	cwchar

The basics of functions

❖ Defining, calling and declaring functions

- The general form of a ***function definition*** is the following (the signs ⟨ ⟩ indicate optional parts):

function header

return_type function_name (⟨*type*$_1$ *param*$_1$, *type*$_2$ *param*$_2$, ... *type*$_n$ *param*$_n$⟩)

function body

{

⟨*local definitions and declarations*⟩

⟨*statements*⟩

return ⟨*expression*⟩;

}

Function definition

The basics of functions

❖ Defining, calling and declaring functions

```
int isum(int n)
{
    int s = 0;
    for (int i=1; i<=n; i++)
        s += i;
    return s;
}

int main()
{
    cout << isum(10) << endl;
    int s = isum(7) * isum(10) + 2;
}
```

The basics of functions

❖ Defining, calling and declaring functions

- The steps of calling a function

function_name (⟨*argument* $_1$, *argument* $_2$, … *argument* $_n$ ⟩)

```
double VCuboid(double, double, double);          // prototype

int main()
{
        double v = VCuboid(10, 7.23, 10.2);      // function call
}
                737.46

double VCuboid(double a, double b, double c)     // function definition
{
        return a*b*c;
}
```

Steps of calling a function

The basics of functions

❖ Defining, calling and declaring functions

- C++ standards require that functions have to be declared before they are called.

Function declaration

Function definition

```
int isum(int); // prototype

int main() {
 cout << isum(10) << endl;
 int s = isum(7) * isum(10) + 2;
}

int isum(int n)
{
    int s = 0;
    for (int i=1; i<=n; i++)
        s += i;
    return s;
}
```

The basics of functions

❖ Defining, calling and declaring functions

- The ***complete declaration of a function*** (its ***prototype***):

return_value function_name(⟨parameter declaration list⟩);

return_value function_name(⟨type_list⟩);

```
int isum(int);
int isum(int n);
int isum(int lots);
```

declaration	**C interpretation**	**C++ interpretation**
type funct();	*type* funct(...);	*type* funct(**void**);
type funct(...);	*type* funct(...);	*type* funct(...);
type funct(**void**);	*type* funct(**void**);	*type* funct(**void**);

The basics of functions

❖ Defining, calling and declaring functions

- C++ makes it possible that a parameter list containing at least one parameter should end with three dots (...).

```
int sscanf ( const char * str, const char * format, ...);
```

- The transferring (throw) of exceptions to the caller function can be enabled or disabled in function header

return_type function_name (⟨*parameterlist*⟩) ⟨**throw**(⟨*type_list*⟩)⟩

{

⟨*local definitions and declarations*⟩

⟨*statements*⟩

return ⟨*expression*⟩;

}

```
int funct() throw(int, const char*); // int and const char*
int funct();                         // all
int funct() throw();                 // not any
```

The basics of functions

❖ The return value of functions

- The *return_type* figuring in the definition/declaration of a function determines the return type of the function, which can be of any C++ type with the exception of arrays and functions.

```
bool IsPrime(unsigned n)
{
    if (n<2) // 0, 1
        return false;
    else {
        unsigned limit = (unsigned)sqrt(1.0*n);
        for (unsigned d=2; d<=limit; d++)
            if ((n % d) == 0)
                return false;
    }
    return true;
}
```

```
bool IsPrime(unsigned n)
{
    bool result = true;
    if (n<2) // 0, 1
        result = false;
    else {
        unsigned limit = (unsigned)sqrt(1.0*n);
        for (unsigned d=2; d<=limit && result; d++)
            if ((n % d) == 0)
                result = false;
    }
    return result;
}
```

The basics of functions

❖ The return value of functions

- By using the type **void**, we can create functions that do not return any value.

```
void PerfectNumbers(int from, int to)
{
        int sum = 0;
        for(int i=from; i<=to; i++) {
            sum = 0;
            for(int j=1; j<i; j++) {
                if(i%j == 0)
                    sum += j;
            }
            if(sum == i)
                cout<< i <<endl;
        }
}
```

The basics of functions

❖ The return value of functions

- Functions can return pointers or references

```
#include <cassert>
#include <new>
using namespace std;

double * Allocate(int size) {
    double *p = new (nothrow) double[size];
    assert(p);
    return p;
}

int  & DinInt() {
    int *p = new (nothrow) int;
    assert(p);
    return *p;
}
```

```
int main() {
    double *pd = Allocate(2012);
    pd[2] = 8;
    delete []pd;

    int &x = DinInt();
    x = 10;
    delete &x;
}
```

The basics of functions

❖ Parametrizing functions

- A parameter can be scalar (**bool**, **char**, **wchar_t**, **short**, **int**, **long**, **long long**, **float**, **double**, enumeration, reference and pointer) or structure, union, class or array.

The general form of polynomials: $a_n x^n + a_{n-1} x^{n-1} + \ldots + a_2 x^2 + a_1 x + a_0$

Horner's scheme $((\ldots((a_n x + a_{n-1})x + a_{n-2})x + \ldots + a_3)x + a_2)x + a_1)x + a_0$

```
double Polynomial(double x, int n, const double c[]) {
    double y = 0;
    for (int i = n; i > 0; i--)
        y = (y + c[i]) * x;
    return y + c[0];
}

int main(){
    const int degree = 3;
    double coefficients[degree + 1] = { 5, 2, 3, 1};
    cout << Polynomial(2,degree,coefficients)<< endl; // 29
}
```

The basics of functions

❖ Parametrizing functions

▪ Parameter passing methods

- Passing parameters by value

$$1+\frac{1}{1!}+\frac{1}{2!}+\frac{1}{3!}+\frac{1}{4!}+\frac{1}{5!}+\ldots+\frac{1}{n!}$$

```
double enumber(int n) {
    double f = 1;
    double eseq = 1;
    for (int i=2; i<=n; i++) {
        eseq += 1.0 / f;
        f *= i;
    }
    return eseq;
}
```

```
int main(){
    long x =1000;
    cout << enumber(x)<< endl;
    cout << enumber(123) << endl;
    cout << enumber(x + 12.34) << endl;
    cout << enumber(&x) << endl;
}
```

The basics of functions

❖ Parametrizing functions

▪ Parameter passing methods

- Passing parameters by value

```
void pswap(double *p, double *q) {
    double c = *p;
    *p = *q;
    *q = c;
}

int main(){
    double a = 12, b =23;
    pswap(&a, &b);
    cout << a << ", " << b<< endl; // 23, 12
}
```

The basics of functions

❖ Parametrizing functions

▪ Parameter passing methods

- Passing parameters by reference

```
void rswap(double & a, double & b) {
    double c = a;
    a = b;
    b = c;
}

int main(){
    double x = 12, y =23;
    rswap(x, y);
    cout << x << ", " << y << endl; // 23, 12
}
```

The basics of functions

❖ Parametrizing functions

▪ Parameter passing methods

- Passing parameters by reference

```
#include <iostream>
#include <cstdlib>
#include <ctime>
using namespace std;

struct svector {
    int size;
    int a[1000];
};

void MinMax(const svector & sv,
int & mi, int & ma) {
    mi = ma = sv.a[0];
    for (int i=1; i<sv.size; i++) {
        if (sv.a[i]>ma)
            ma = sv.a[i];
        if (sv.a[i]<mi)
            mi = sv.a[i];
    }
}
```

```
int main() {
    const int maxn = 1000;
    srand(unsigned(time(0)));
    svector v;
    v.size=maxn;
    for (int i=0; i<maxn; i++) {
        v.a[i]=rand() % 102 + (rand() % 2012);
    }
    int min, max;
    MinMax(v, min, max);
    cout << min << endl;
    cout << max << endl;
}
```

The basics of functions

❖ Parametrizing functions

▪ Parameter passing methods

- Passing parameters by reference

```
int Gcd(const int & a, const int & b ) {
    int min = a<b ? a : b, gcd = 0;
    for (int n=min; n>0; n--)
        if ( (a % n == 0) && (b % n) == 0) {
            gcd = n;
            break;
        }
    return gcd;
}

int main() {
    cout << Gcd(24, 32) <<endl; // 8
}
```

The basics of functions

❖ Parametrizing functions

▪ Using parameters of different types

- Arithmetic type parameters

```
void PrintOutF(double data, int field, int precision) {
    cout.width(field);
    cout.precision(precision);
    cout << fixed << data << endl;
}
...
PrintOutF(123.456789, 10, 4);  //  123.4568
```

```
long Product(int a, int b) {
   return long(a) * b;
}
...
cout << Product(12, 23) <<endl; // 276
```

The basics of functions

❖ Parametrizing functions

▪ Using parameters of different types

- Arithmetic type parameters

```
void Cube(double a, double & surface_area, double & volume,
          double & diagonal) {
    surface_area = 6 * a * a;
    volume = a * a * a;
    diagonal = a * sqrt(3.0);
}
...
double f, v, d;
Cube(10, f, v, d);
```

The basics of functions

❖ Parametrizing functions

- **Using parameters of different types**
- User-defined type parameters

```
struct complex {
    double re, im;
};

void CSum1(const complex*pa, const complex*pb, complex *pc){
  pc->re = pa->re + pb->re;
  pc->im = pa->im + pb->im;
}

complex CSum2(complex a, complex b) {
  complex c;
  c.re = a.re + b.re;
  c.im = a.im + b.im;
  return c;
}
```

The basics of functions

❖ Parametrizing functions

- **Using parameters of different types**
- User-defined type parameters

```
complex Csum3(const complex & a, const complex & b) {
  complex c;
  c.re = a.re + b.re;
  c.im = a.im + b.im;
  return c;
}

int main() {
    complex c1 = {10, 2}, c2 = {7, 12}, c3;

    // all the three arguments are pointers
    CSum1(&c1, &c2, &c3);

    c3 = CSum2(c1, c2);   // arguments are structures

    // arguments are structure references
    c3 = CSum3(c1, c2);
}
```

The basics of functions

❖ Parametrizing functions

- **Using parameters of different types**
- Passing arrays to functions

```
long VectorSum1 (const int vector[], int n) {
  long sum = 0;
  for (int i = 0; i < n; i++) {
     sum+=vector[i];   // or sum+=*(vector+i);
  }
  return sum;
}
...
int v[7] = { 10, 2, 11, 30, 12, 7, 23};
cout <<VectorSum1(v, 7)<< endl;  // 95
cout <<VectorSum1(v, 3)<< endl;  // 23

long VectorSum2 (const int * const vector, int n) {
  long sum = 0;
  for (int i = 0; i < n; i++) {
     sum+=vector[i];   // or sum+=*(vector+i);
  }
  return sum;
}
```

The basics of functions

❖ Parametrizing functions

- **Using parameters of different types**
- Passing arrays to functions

```
void Sort(double v[], int n) {
    double temp;
    for (int i = 0; i < n-1; i++)
        for (int j=i+1; j<n; j++)
            if (v[i]>v[j]) {
                temp = v[i];
                v[i] = v[j];
                v[j] = temp;
            }
}

int main() {
    const int size=7;
    double v[size]={10.2, 2.10, 11, 30, 12.23, 7.29, 23.};
    Sort(v, size);
    for (int i = 0; i < size; i++)
        cout << v[i]<< '\t';
    cout << endl;
}
```

The basics of functions

❖ Parametrizing functions

▪ Using parameters of different types

- Passing arrays to functions

```
long VectorSum3 (const int (&vector)[6]) {
  long sum = 0;
  for (int i = 0; i < 6; i++) {
     sum+=vector[i];   // or sum+=*(vector+i);
  }
  return sum;
}
...
int v[6] = { 10, 2, 11, 30, 12, 7};
cout << VectorSum3(v) << endl;

void PrintMatrix23(const int matrix[2][3]) {
  for (int i=0; i<2; i++) {
     for (int j=0; j<3; j++)
       cout <<matrix[i][j] <<'\t';
     cout<<endl;
  }
}
...
int m[2][3] = { {10, 2, 12}, {23, 7, 29} };
PrintMatrix23(m);
```

```
10      2       12
23      7       29
```

The basics of functions

❖ Parametrizing functions

- **Using parameters of different types**
- Passing arrays to functions

```
void PrintMatrixN3(const int matrix[][3], int n) {
  for (int i=0; i<n; i++) {
     for (int j=0; j<3; j++)
       cout <<matrix[i][j] <<'\t';
     cout<<endl;
  }
}
...
int m[2][3] = { {10, 2, 12}, {23, 7, 29} };
PrintMatrixN3(m, 2);
cout << endl;
int m2[3][3] = { {1}, {0, 1}, {0, 0, 1} };
PrintMatrixN3(m2, 3);
```

```
10      2       12
23      7       29

1       0       0
0       1       0
0       0       1
```

The basics of functions

❖ Parametrizing functions

- **Using parameters of different types**
- Passing arrays to functions

```
void PrintMatrixNM(const void *pm, int n, int m) {
  for (int i=0; i<n; i++) {
     for (int j=0; j<m; j++)
        cout <<*((int *)pm+i*m+j) <<'\t';
     cout<<endl;
  }
}
...
int m[2][3] = { {10, 2, 12}, {23, 7, 29} };
PrintMatrixNM(m, 2, 3);
cout << endl;
int m2[4][4] = { {0,0,0,1}, {0,0,1}, {0,1}, {1} };
PrintMatrixNM(m2, 4, 4);
```

```
10      2       12
23      7       29

0       0       0       1
0       0       1       0
0       1       0       0
1       0       0       0
```

The basics of functions

❖ Parametrizing functions

▪ Using parameters of different types

- String arguments

```
unsigned CountChC1(const char s[], char ch) {
    unsigned cnt = 0;
    for (int i=0; s[i]; i++) {
        if (s[i] == ch)
            cnt++;
    }
    return cnt;
}

unsigned CountChC2(const char *s, char ch) {
    unsigned cnt = 0;
    while (*s) {
        if (*s++ == ch)
            cnt++;
    }
    return cnt;
}
```

The basics of functions

❖ Parametrizing functions

▪ Using parameters of different types

- String arguments

```
unsigned CountChCpp(const string & s, char ch) {
    int cnt = -1, position = -1;
    do {
        cnt++;
        position = s.find(ch, position+1);
    } while (position != string::npos);
    return cnt;
}

char s1[] = "C, C++, Java, C++/CLI / C#";
string s2 = s1;

cout<<CountChC1("C, C++, Java, C++/CLI / C#", 'C')<<endl;
cout<<CountChC2(s1, 'C')<<endl;

cout << CountChC2(s2.c_str(), 'C') << endl;
cout << CountChCpp(s2, 'C') << endl;
```

The basics of functions

❖ Parametrizing functions

▪ Using parameters of different types

- String arguments

```
char * StrReverseC1(char s[]) {
    char ch;
    int length = -1;
    while(s[++length]); // determining the string length
    for (int i = 0; i < length / 2; i++) {
        ch = s[i];
        s[i] = s[length-i-1];
        s[length-i-1] = ch;
    }
    return s;
}

char * StrReverseC2(char *s) {
    char *q, *p, ch;
    p = q = s;
    while (*q) q++;      // going through the elements until byte 0
    closing the string
    p--;              // p points to the first character of the string
    while (++p <= --q) {
        ch = *p;
        *p = *q;
        *q = ch;
    }
    return s;
}
```

The basics of functions

❖ Parametrizing functions

▪ Using parameters of different types

- String arguments

```
string& StrReverseCpp(string &s) {
    char ch;
    int length = s.size();
    for (int i = 0; i < length / 2; i++) {
        ch = s[i];
        s[i] = s[length-i-1];
        s[length-i-1] = ch;
    }
    return s;
}

int main() {
    char s1[] = "C++ programming";
    cout << StrReverseC1(s1) << endl;     // gnimmargorp ++C
    cout << StrReverseC2(s1) << endl;     // C++ programming
    string s2 = s1;
    cout << StrReverseCpp(s2) << endl;     // gnimmargorp ++C
    cout << StrReverseCpp(string(s1));     // gnimmargorp ++C
}
```

The basics of functions

❖ Parametrizing functions

▪ Using parameters of different types

- Functions as arguments

```
#include <iostream>
#include <cstdlib>
#include <cstring>
using namespace std;

int icmp(const void *p, const void *q) {
 return *(int *)p-*(int *)q;
}

int scmp(const void *p, const void *q) {
 return strcmp((char *)p,(char *)q);
}

int main() {
    int m[8]={2, 10, 7, 12, 23, 29, 11, 30};
    char names[6][20]={"Dennis Ritchie", "Bjarne Stroustrup",
                       "Anders Hejlsberg","Patrick Naughton",
                       "James Gosling", "Mike Sheridan"};

    qsort(m, 8, sizeof(int), icmp);
    for (int i=0; i<8; i++)
      cout<<m[i]<<endl;

    qsort(names, 6, 20, scmp);
    for (int i=0; i<6; i++)
      cout<<names[i]<<endl;
}
```

The basics of functions

❖ Parametrizing functions

▪ Using parameters of different types

- Functions as arguments

```
The values of the function sqr() ([-2,2] dx=0.5)
     -2.0000       4.0000000
     -1.5000       2.2500000
     -1.0000       1.0000000
     -0.5000       0.2500000
      0.0000       0.0000000
      0.5000       0.2500000
      1.0000       1.0000000
      1.5000       2.2500000
      2.0000       4.0000000

The values of the function sqrt() ([0,2] dx=0.2)
      0.0000       0.0000000
      0.2000       0.4472136
      0.4000       0.6324555
      0.6000       0.7745967
      0.8000       0.8944272
      1.0000       1.0000000
      1.2000       1.0954451
      1.4000       1.1832160
      1.6000       1.2649111
      1.8000       1.3416408
      2.0000       1.4142136
```

```cpp
#include <iostream>
#include <iomanip>
#include <cmath>
using namespace std;

// Prototypes
void tabulate(double (*)(double), double, double, double);
double sqr(double);

int main() {
    cout<<"\n\nThe values of the function sqr() ([-2,2] dx=0.5)"<<endl;
    tabulate(sqr, -2, 2, 0.5);
    cout<<"\n\nThe values of the function sqrt() ([0,2] dx=0.2)"<<endl;
    tabulate(sqrt, 0, 2, 0.2);
}

// The definition of the function tabulate()
void tabulate(double (*fp)(double), double a, double b,
              double step){
    for (double x=a; x<=b; x+=step) {
        cout.precision(4);
        cout<<setw(12)<<fixed<<x<< '\t';
        cout.precision(7);
        cout<<setw(12)<<fixed<<(*fp)(x)<< endl;
    }
}

// The definition of the function sqr()
double sqr(double x) {
    return x * x;
}
```

The basics of functions

❖ Parametrizing functions

- **Using parameters of different types**
- Default arguments

```
// prototype
long SeqSum(int n = 10, int d = 1, int a0 = 1);

long SeqSum(int n, int d, int a0) {  // definition
    long sum = 0, ai;
    for(int i = 0; i < n; i++) {
        ai = a0 + d * i;
        cout << setw(5) << ai;
        sum += ai;
    }
    return sum;
}
```

Call	Parameters		
	n	d	a0
SeqSum()	10	1	1
SeqSum(12)	12	1	1
SeqSum(12,3)	12	3	1
SeqSum(12, 3, 7)	12	3	*7*

The basics of functions

❖ Parametrizing functions

- **Using parameters of different types**
- Variable length argument list

```
int printf(const char * format, ... );

char name[] = "Bjarne Stroustrup";
double a=12.3, b=23.4, c=a+b;
printf("C++ language\n");
printf("Name: %s \n", name);
printf("Result: %5.3f + %5.3f = %8.4f\n", a, b, c);
```

The basics of functions

❖ Parametrizing functions

- **Using parameters of different types**
- Variable length argument list

```
#include<iostream>
#include<cstdarg>
using namespace std;

double Average(int num, ... ) {
    va_list numbers;
    // passing through the first (num) arguments
    va_start(numbers, num);
    double sum = 0;
    for(int i = 0; i < num; ++i ) {
        // accessing the double arguments
        sum += va_arg(numbers, double);
    }
    va_end(numbers);
    return (sum/num);
}

int main() {
    double avg = Average(7,1.2,2.3,3.4,4.5,5.6,6.7,7.8);
    cout << avg << endl;
}
```

The basics of functions

❖ Parametrizing functions

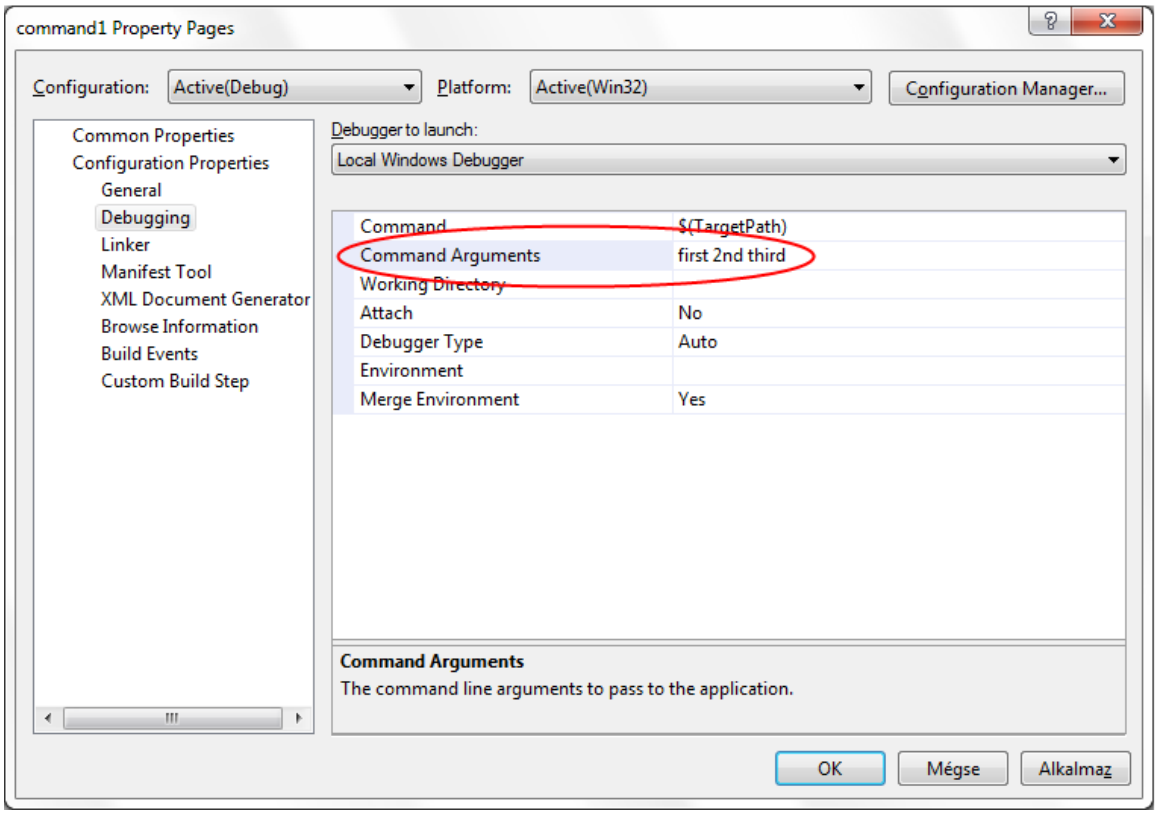
- **Using parameters of different types**
- Parameters and return value of the main() function

```
#include <iostream>
using namespace std;

int main(int argc, char *argv[]) {
    cout << "number of arguments: " << argc << endl;
    for (int narg=0; narg < argc; narg++)
        cout << narg << " " << argv[narg] << endl;
    return 0;
}
```

The basics of functions

❖ Parametrizing functions

- **Using parameters of different types**
- Parameters and return value of the main() function



Providing command line arguments

The basics of functions

❖ Parametrizing functions

- **Using parameters of different types**
- Parameters and return value of the main() function

In a console window

```
C:\C++Book>command1 first 2nd third
number of arguments: 4
0: command1 1: first 2: 2nd 3: third
C:\C++Book>
```

In a development environment

```
C:\C++Book>command1 first 2nd third
number of arguments: 4
0: C:\C++Book\command1.exe 1: first 2: 2nd
3: third
C:\C++Book>
```

The basics of functions

❖ Parametrizing functions

- **Using parameters of different types**
- Parameters and return value of the main() function

```
#include <iostream>
#include <cstdlib>
using namespace std;

int main(int argc, char *argv[]) {
    if (argc !=3 ) {
        cerr<<"Wrong number of parameters!"<<endl;
        cerr<<"Usage: command2 arg1 arg2"<<endl;
        return EXIT_FAILURE;
    }
    cout<<"Correct number of parameters:"<<endl;
    cout<<"1. argument: "<<argv[1]<<endl;
    cout<<"2. argument: "<<argv[2]<<endl;
    return EXIT_SUCCESS;
}
```

The basics of functions

❖ Parametrizing functions

- **Using parameters of different types**
- Parameters and return value of the main() function

Wrong number of parameters: *command2*	**Correct number of parameters**: *command2 alfa beta*
Wrong number of parameters! Usage: command2 arg1 arg2	Correct number of parameters: 1. argument: alfa 2. argument: beta

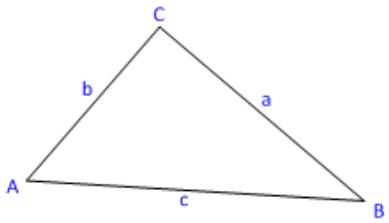
The basics of functions

- Defining, calling and declaring functions
- The return value of functions
- Parametrizing functions
- Programming with functions

The basics of functions

❖ Programming with functions

- **Exchanging data between functions using global variables**



Heron's formula:

$$s = \frac{a+b+c}{2}$$

$$t = \sqrt{s(s-a)(s-b)(s-c)}$$

```
#include <iostream>
#include <cmath>
using namespace std;

int main() {
    double a, b, c;
    do {
        cout <<"a side: "; cin>>a;
        cout <<"b side: "; cin>>b;
        cout <<"c side: "; cin>>c;
    } while(!((a<b+c) && (b<a+c) && (c<a+b)));
    double p = a + b + c;
    double s = p/2;
    double area = sqrt(s*(s-a)*(s-b)*(s-c));
    cout << "perimeter: " << p <<endl;
    cout << "area: " << area <<endl;
}
```

The basics of functions

❖ Programming with functions

▪ Exchanging data between functions using global variables

- Solution for above example

```
#include <iostream>
#include <cmath>
using namespace std;

// global variables
double a, b, c;

// prototypes
void ReadData();
bool TriangleInequality();
double Perimeter();
double Area();

int main() {
    ReadData();
    cout << "perimeter: " << Perimeter()<<endl;
    cout << "area: " << Area()<<endl;
}
```

```
void ReadData() {
    do {
        cout <<"a side: "; cin>>a;
        cout <<"b side: "; cin>>b;
        cout <<"c side: "; cin>>c;
    } while(!TriangleInequality() );
}

bool TriangleInequality() {
    if ((a<b+c) && (b<a+c) && (c<a+b))
        return true;
    else
        return false;
}

double Perimeter() {
    return a + b + c;
}

double Area() {
    double s = Perimeter()/2;
    return sqrt(s*(s-a)*(s-b)*(s-c));
}
```

The basics of functions

❖ Programming with functions

- **Exchanging data between functions using global variables**
- Solution for above example

```
int main() {
    double a1, b1, c1, a2, b2, c2;
    ReadData();
    a1 = a, b1 = b, c1 = c;
    ReadData();
    a2 = a, b2 = b, c2 = c;
    a = a1, b = b1, c = c1;
    cout << "perimeter: " << Perimeter()<<endl;
    cout << "area: " << Area()<<endl;
    a = a2, b = b2, c = c2;
    cout << "perimeter: " << Perimeter()<<endl;
    cout << "area: " << Area()<<endl;
}
```

The basics of functions

❖ Programming with functions

▪ Exchanging data between functions using parameters

```
#include <iostream>
#include <cmath>
using namespace std;

void ReadData(double &a, double &b, double &c);
bool TriangleInequality(double a, double b, double c);
double Perimeter(double a, double b, double c);
double Area(double a, double b, double c);

int main() {
    double x, y, z;
    ReadData(x , y , z);
    cout << "perimeter: " << Perimeter(x, y, z)<<endl;
    cout << "area: " << Area(x, y, z) <<endl;
}

void ReadData(double &a, double &b, double &c) {
    do {
        cout <<"a side: "; cin>>a;
        cout <<"b side: "; cin>>b;
        cout <<"c side: "; cin>>c;
    } while(!TriangleInequality(a, b, c) );
}
```

```
bool TriangleInequality(double a, double b, double c) {
    if ((a<b+c) && (b<a+c) && (c<a+b))
        return true;
    else
        return false;
}

double Perimeter(double a, double b, double c) {
    return a + b + c;
}

double Area(double a, double b, double c) {
    double s = Perimeter(a, b, c)/2;
    return sqrt(s*(s-a)*(s-b)*(s-c));
}
```

The basics of functions

❖ Programming with functions

- **Exchanging data between functions using parameters**

```
int main() {
    double a1, b1, c1, a2, b2, c2;
    ReadData(a1, b1, c1);
    ReadData(a2, b2, c2);
    cout << "perimeter: " << Perimeter(a1, b1, c1)<<endl;
    cout << "area: " << Area(a1, b1, c1)<<endl;
    cout << "perimeter: " << Perimeter(a2, b2, c2)<<endl;
    cout << "area: " << Area(a2, b2, c2)<<endl;
}
```

The basics of functions

❖ Programming with functions

▪ Exchanging data between functions using parameters

```
#include <iostream>
#include <cmath>
using namespace std;

struct triangle {
    double a, b, c;
};

void ReadData(triangle &h);
bool TriangleInequality(const triangle &h);
double Perimeter(const triangle &h);
double Area(const triangle &h);

int main() {
    triangle h1, h2;
    ReadData(h1);
    ReadData(h2);
    cout << "perimeter: " << Perimeter(h1)<<endl;
    cout << "area: " << Area(h1)<<endl;
    cout << "perimeter: " << Perimeter(h2)<<endl;
    cout << "area: " << Area(h2)<<endl;
}
```

```
void ReadData(triangle &h) {
    do {
        cout <<"a side: "; cin>>h.a;
        cout <<"b side: "; cin>>h.b;
        cout <<"c side: "; cin>>h.c;
    } while(!TriangleInequality(h) );
}

bool TriangleInequality(const triangle &h) {
    if ((h.a<h.b+h.c) && (h.b<h.a+h.c) && (h.c<h.a+h.b))
        return true;
    else
        return false;
}

double Perimeter(const triangle &h) {
    return h.a + h.b + h.c;
}

double Area(const triangle &h) {
    double s = Perimeter(h)/2;
    return sqrt(s*(s-h.a)*(s-h.b)*(s-h.c));
}
```

The basics of functions

❖ Programming with functions

- **Implementing a simple menu driven program structure**

```
// Defintion of the menu
char * menu[] = {"\n1. Sides",
                "2. Perimeter",
                "3. Area",
                "----------",
                "0. Exit" , NULL };
```

```
1. Sides
2. Perimeter
3. Area
----------
0. Exit
1
a side: 3
b side: 4
c side: 5

1. Sides
2. Perimeter
3. Area
----------
0. Exit
2
perimeter: 12

1. Sides
2. Perimeter
3. Area
----------
0. Exit
3
area: 6
```

```
int main() {
    triangle h = {3,4,5};
    char ch;
    do {
      // printing out the menu
      for (int i=0; menu[i]; i++)
          cout<<menu[i]<< endl;

      // Processing the answer
      cin>>ch; cin.get();
      switch (ch) {
        case '0': break;
        case '1': ReadData(h);
                  break;
        case '2': cout << "perimeter: ";
                   cout << Perimeter(h)<<endl;
                  break;
        case '3': cout << "area: " << Area(h)<<endl;
                  break;
        default:  cout<<'\a';
      }
    } while (ch != '0');
}
```

The basics of functions

❖ Programming with functions

▪ Recursive functions

- Factorial: $$n! = \begin{cases} 1, & if \ \ n = 0 \\ n \cdot (n-1)!, & if \ \ n > 0 \end{cases}$$

```
unsigned long long Factorial(int n)
{
    if (n == 0)
        return 1;
    else
        return n * Factorial(n-1);
}
```

more efficient

```
unsigned long long Factorial(int n)
{
    unsigned long long f=1;
    while (n != 0)
        f *= n--;
    return f;
}
```

The basics of functions

❖ Programming with functions

- **Recursive functions**
- *Fibonacci numbers:*

$$Fibo(n) = \begin{cases} 0, if\ n = 0 \\ 1, if\ n = 1 \\ Fibo(n-1) + Fibo(n-2), \quad if\ n > 1 \end{cases}$$

```
unsigned long Fibonacci( int n ) {
    if (n<2)
        return n;
    else
        return Fibonacci(n-1) + Fibonacci(n-2);
}

unsigned long Fibonacci( int n ) {
    unsigned long f0 = 0, f1 = 1, f2 = n;
    while (n-- > 1) {
        f2 = f0 + f1;
        f0 = f1;
        f1 = f2;
    }
    return f2;
}
```

more efficient

The basics of functions

- **Programming with functions**
- **Recursive functions**
- *greatest common divisor (gcd):*

$$\gcd(p,q) = \begin{cases} p, if \;\; q = 0 \\ \gcd(q, p\%q), \;\; else \end{cases}$$

```
int Gcd(int p, int q) {
    if (q == 0)
        return p;
    else
        return Gcd(q, p % q);
}
```

The basics of functions

❖ Programming with functions

▪ Recursive functions

- *binomial numbers:*

$$B(n,k)=\begin{cases}1, if\ \ k=0\\ B(n,k-1)+B(n-1,k-1),\ \ if\ \ 0<k<n\\ 1, if\ \ k=n\end{cases}$$

```
int Binom1(int n, int k)
{
    if (k == 0 || k == n)
        return 1;
    else
        return Binom1(n-1,k-1) + Binom1(n-1,k);
}
```

$$B(n,k)=\begin{cases}1, if\ \ k=0\\ B(n,k-1)\cdot\dfrac{n-k+1}{k},\ \ if\ \ 0<k\le n\end{cases}$$

```
int Binom2(int n, int k)
{
    if (k==0)
        return 1;
    else
        return Binom2(n, k-1) * (n-k+1) / k;
}
```

The basics of functions

❖ Programming with functions

- **Recursive functions**

```
#include <iostream>
#include <iomanip>
using namespace std;

typedef double Matrix[12][12];

double Determinant(Matrix m, int n);
void PrintOutMatrix(Matrix m, int n);

int main() {
  Matrix m2 = {{1, 2},
               {2, 3}};

  Matrix m3 = {{1, 2, 3},
               {2, 1, 1},
               {1, 0, 1}};

  Matrix m4 = {{ 2, 0, 4, 3},
               {-1, 2, 6, 1},
               {10, 3, 4,-2},
               { 2, 1, 4, 0}};

  PrintOutMatrix(m2,2);
  cout << "Determinant(m2) = " << Determinant(m2,2) << endl;

  PrintOutMatrix(m3,3);
  cout << "Determinant(m3) = " << Determinant(m3,3) << endl;

  PrintOutMatrix(m4,4);
  cout << "Determinant(m4) = " << Determinant(m4,4) << endl;

  cin.get();
  return 0;
}
```

The basics of functions

❖ Programming with functions

- **Recursive functions**

```
     1     2
     2     3
Determinant(m2) = -1

     1     2     3
     2     1     1
     1     0     1
Determinant(m3) = -4

     2     0     4     3
    -1     2     6     1
    10     3     4    -2
     2     1     4     0
Determinant(m4) = 144
```

```
void PrintOutMatrix(Matrix m, int n)
{
   for (int i=0; i<n; i++) {
      for (int j=0; j<n; j++) {
        cout << setw(6) << setprecision(0) << fixed;
        cout << m[i][j];
      }
      cout << endl;
   }
}

double Determinant(Matrix m, int n)
{
  int q;
  Matrix x;

  if (n==2)
    return m[0][0]*m[1][1]-m[1][0]*m[0][1];

  double s = 0;
  for (int k=0; k<n; k++)  // n subdeterminants
  {
      // creating submatrices
      for (int i=1; i<n; i++)    // rows
      {
         q = -1;
         for (int j=0; j<n; j++) // columns
          if (j!=k)
            x[i-1][++q] = m[i][j];
      }
      s+=(k % 2 ? -1 : 1) * m[0][k] * Determinant(x, n-1);
  }
  return s;
}
```

Chapter II. Modular programming in C++

- The basics of functions
- How to use functions on a more professional level?
- Namespaces and storage classes
- Preprocessor directives of C++

How to use functions on a more professional level?

- Inline functions
- Overloading (redefining) function names
- Function templates

How to use functions on a more professional level?

❖ Inline functions

- C++ compilers decrease the time spent on calling the functions marked with the keyword **inline**.
- This solution is recommended to be used for small-sized and frequently called functions.

```
inline double Max(double a, double b) {
    return a > b ? a : b;
}

inline char LowerCase( char ch ) {
   return ((ch >= 'A' && ch <= 'Z') ? ch + ('a'-'A') : ch );
}
```

How to use functions on a more professional level?

❖ Inline functions

```
#include <iostream>
using namespace std;

inline int BinarySearch(int vector[], int size, int key) {
    int result = -1; // not found
    int lower = 0;
    int upper = size - 1, middle;
    while (upper >= lower) {
        middle = (lower + upper) / 2;
        if (key < vector[middle])
            upper = middle - 1;
        else if (key == vector[middle]) {
            result = middle;
            break;
        }
        else
            lower = middle + 1;
    } // while
    return result;
}

int main() {
    int v[8] = {2, 7, 10, 11, 12, 23, 29, 30};
    cout << BinarySearch(v, 8, 23)<< endl;  // 5
    cout << BinarySearch(v, 8,  4)<< endl;  // -1
}
```

How to use functions on a more professional level?

- Inline functions
- Overloading (redefining) function names
- Function templates

How to use functions on a more professional level?

❖ Overloading (redefining) function names

- **Different functions** can be defined with the **same name** and within the **same scope** but with a **different parameter list**.

```
#include <iostream>
using namespace std;

inline int Absolute(int x) {
    return x < 0 ? -x : x;
}

inline double Absolute(double x) {
    return x < 0 ? -x : x;
}

int main() {
    cout << Absolute(-7) << endl;          // int
    cout << Absolute(-1.2) << endl;     // double
    cout << Absolute(2.3F) << endl;     // double
    cout << Absolute('I') << endl;         // int
    cout << Absolute(L'\x807F') << endl;    // int
}
```

How to use functions on a more professional level?

❖ Overloading (redefining) function names

```
#include <iostream>
using namespace std;

unsigned int VectorSum(unsigned int a[], int n) {
   int sum=0;
   for (int i=0; i<n; i++)
      sum += a[i];
   return sum;
}

double VectorSum(double a[], int n) {
   double sum = 0;
   for (int i=0; i<n; i++)
     sum += a[i];
   return sum;
}

int main() {
   unsigned int vu[]={1,1,2,3,5,8,13};
   const int nu=sizeof(vu) / sizeof(vu[0]);
   cout <<"\nThe sum of the elements of the unsigned array: "
        <<VectorSum(vu, nu);

   double    vd[]={1.2,2.3,3.4,4.5,5.6};
   const int nd=sizeof(vd) / sizeof(vd[0]);
   cout << "\nThe sum of the elements of the double array: "
        << VectorSum(vd, nd);
}
```

How to use functions on a more professional level?

- Inline functions
- Overloading (redefining) function names
- Function templates

How to use functions on a more professional level?

❖ Function templates

- **Creating and using function templates**

- A template declaration starts with the keyword **template**, followed by the parameters of the template enclosed within the signs **<** and **>**.

```
template <class TYPE>
inline TYPE Absolute(TYPE x) {
    return x < 0 ? -x : x;
}

template <typename TYPE>
inline TYPE Absolute(TYPE x) {
    return x < 0 ? -x : x;
}
```

```
cout << Absolute(123)<< endl;    // TYPE = int
cout << Absolute(123.45)<< endl;// TYPE = double

cout << Absolute((float)123.45)<< endl;// TYPE = float
cout << Absolute((int)123.45)<< endl;  // TYPE = int

cout << Absolute<float>(123.45)<< endl;// TYPE = float
cout << Absolute<int>(123.45)<< endl;  // TYPE = int
```

How to use functions on a more professional level?

❖ Function templates

▪ Creating and using function templates

```
template <typename T>
inline T Maximum(T a, T b) {
   return (a>b ? a : b);
}

int main() {
    int a = 12, b=23;
    float c = 7.29, d = 10.2;

    cout<<Maximum(a, b); // Maximum(int, int)
    cout<<Maximum(c, d); // Maximum(float, float)
    cout<<Maximum(a, c); // there is no Maximum(int, float)--> should
    be avoided
    cout<<Maximum<int>(a,c);
    // it is the function Maximum(int, int) that is called with type
    conversion
}
```

How to use functions on a more professional level?

❖ Function templates

- **Creating and using function templates**

```
template <typename T1, typename T2>
inline T1 Maximum(T1 a, T2 b) {
   return (a>b ? a : b);
}

int main() {
    cout<<Maximum(5,4);          // Maximum(int, int)
    cout<<Maximum(5.6,4);        // Maximum(double, int)
    cout<<Maximum('A',66L);      // Maximum(char, long)
    cout<<Maximum<float, double>(5.6,4);// Maximum(float, double)
    cout<<Maximum<int, char>('A',66L);  // Maximum(int, char)
}
```

How to use functions on a more professional level?

❖ Function templates

▪ Function template instantiation

- A function template can be instantiated in an *explicit* way as well if concrete types are provided in the **template** line containing the header of the function:

```
template inline int Absolute<int>(int);
template inline float Absolute(float);
template inline double Maximum(double, long);
template inline char Maximum<char, float>(char, float);
```

How to use functions on a more professional level?

❖ Function templates

▪ Function template specialization

```
#include <iostream>
#include <cstring>
using namespace std;

// The basic template
template< typename T > void Swap(T& a, T& b) {
   T c(a);
   a = b;
   b = c;
}

// Template specialised for pointers
template<typename T> void Swap(T* a, T* b) {
   T c(*a);
   *a = *b;
   *b = c;
}

// Template specialised for C strings
template<> void Swap(char *a, char *b) {
   char buffer[123];
   strcpy(buffer, a);
   strcpy(a, b);
   strcpy(b, buffer);
}
```

```
int main() {
    int a=12, b=23;
    Swap(a, b);
    cout << a << "," << b << endl;      // 23,12

    Swap(&a, &b);
    cout << a << "," << b << endl;        // 12,23

    char *px = new char[32]; strcpy(px, "First");
    char *py = new char[32]; strcpy(py, "Second");
    Swap(px, py);
    cout << px << ", " << py << endl;    // Second, First
    delete px;
    delete py;
}
```

How to use functions on a more professional level?

❖ Function templates

- **Some further function template examples**

```
#include <iostream>
using namespace std;

template<class TYPE>
TYPE VectorSum(TYPE a[], int n) {
    TYPE sum = 0;
    for (int i=0; i<n; i++)
        sum += a[i];
    return sum;
}
```

```
int main() {
    int         ai[]={1,1,2,3,5,8,13};
    const int ni=sizeof(ai) / sizeof(ai[0]);
    cout << "\nThe sum of the elements of the int array: "
         <<VectorSum(ai,ni);

    double      ad[]={1.2,2.3,3.4,4.5,5.6};
    const int nd=sizeof(ad) / sizeof(ad[0]);
    cout << "\nThe sum of the elements of the double array: "
         <<VectorSum(ad,nd);

    float       af[]={3, 2, 4, 5};
    const int nf=sizeof(af) / sizeof(af[0]);
    cout << "\nThe sum of the elements of the  float  array: "
         <<VectorSum(af,nf);

    long        al[]={1223, 19800729, 2004102};
    const int nl=sizeof(al) / sizeof(al[0]);
    cout << "\nThe sum of the elements of the  long   array: "
         <<VectorSum(al,nl);
}
```

How to use functions on a more professional level?

❖ Function templates

- **Some further function template examples**

```
#include <iostream>
using namespace std;

template< typename T > void Swap(T& a, T& b) {
    T c(a);
    a = b;
    b = c;
}

template<class Typ, int Size> void Sort(Typ vector[]) {
    int j;
    for(int i = 1; i < Size; i++){
        j = i;
        while(0 < j && vector[j] < vector[j-1]){
            Swap<Typ>(vector[j], vector[j-1]);
            j--;
        }
    }
}

int main() {
    const int size = 6;
    int array[size] = {12, 29, 23, 2, 10, 7};
    Sort<int, size>(array);
    for (int i = 0; i < size; i++)
        cout << array[i] << " ";
    cout << endl;
}
```

How to use functions on a more professional level?

❖ Function templates

▪ Some further function template examples

```
#include <iostream>
#include <sstream>
#include <string>
#include <typeinfo>
#include <cctype>
using namespace std;

template <typename T> T Character(T a) {
    return a;
}

string Character(char c) {
    stringstream ss;
    if (isprint((unsigned char)c))
        ss << "'" << c << "'";
    else
        ss << (int)c;
    return ss.str();
}

template <typename TA, typename TB>
void PrintOut(TA a, TB b){
    cout << "Function<" << typeid(TA).name() << ","
         << typeid(TB).name();
    cout << ">(" << Character(a) << ", " << Character(b)
         << ")\n";
}
```

```
template <typename TA, typename TB>
void Function(TA a = 0, TB b = 0){
    PrintOut<TA, TB>(a,b);
}

template <typename TA>
void Function(TA a = 0, double b = 0) {
    Function<TA, double>(a, b);
}

void Function(int a = 12, int b = 23) {
    Function<int, int>(a, b);
}

int main() {
        Function(1, 'c');
        Function(1.5);
        Function<int>();
        Function<int, char>();
        Function('A', 'X');
        Function();
}
```

Chapter II. Modular programming in C++

- The basics of functions
- How to use functions on a more professional level?
- Namespaces and storage classes
- Preprocessor directives of C++

Namespaces and storage classes

- Storage classes of variables
- Storage classes of functions
- Modular programs in C++
- Namespaces

Namespaces and storage classes

❖ Storage classes of variables

- A storage class:
  - defines the lifetime or storage duration of a variable,
  - determines the place from where the name of a variable can be accessed directly – visibility, scope – and also determines which name designates which variable – linkage.

- A storage class (**auto**, **register**, **static**, **extern**) can be assigned to variables when they are defined.

Namespaces and storage classes

❖ Storage classes of variables

```
#include <iostream>
using namespace std;

static int bufferpointer;  // module level definition

// n and first are block level definitions
void IntToHex(register unsigned n, auto bool first = true) {
    extern char buffer[];  // program level declaration
auto int digit;     // block level definition
// hex is block level definition
    static const char hex[] = "0123456789ABCDEF";
    if (first)
        bufferpointer=0;
    if ((digit = n / 16) != 0)
        IntToHex(digit, false);   // recursive call
    buffer[bufferpointer++] = hex[n % 16];
 }

extern char buffer[];  // program level declaration

int main() {
    auto unsigned n;   // block level definition
    for (n=0; n<123456; n+=9876) {
        IntToHex(n);
        buffer[bufferpointer]=0;  // closing the string
        cout << buffer << endl;
    }
}

extern char buffer [123] = {0};   // program level definition
```

Namespaces and storage classes

❖ Storage classes of variables

▪ Accessibility (scope) and linkage of variables

- In a C++ source code, variables can have one of the following scopes:

block level

A variable of this type is ***only visible in the block*** (function block) where it has been defined so ***its accessibility is local***.
If a variable is defined on a block level without the storage classes **extern** and **static**, it **does not have any linkage**.

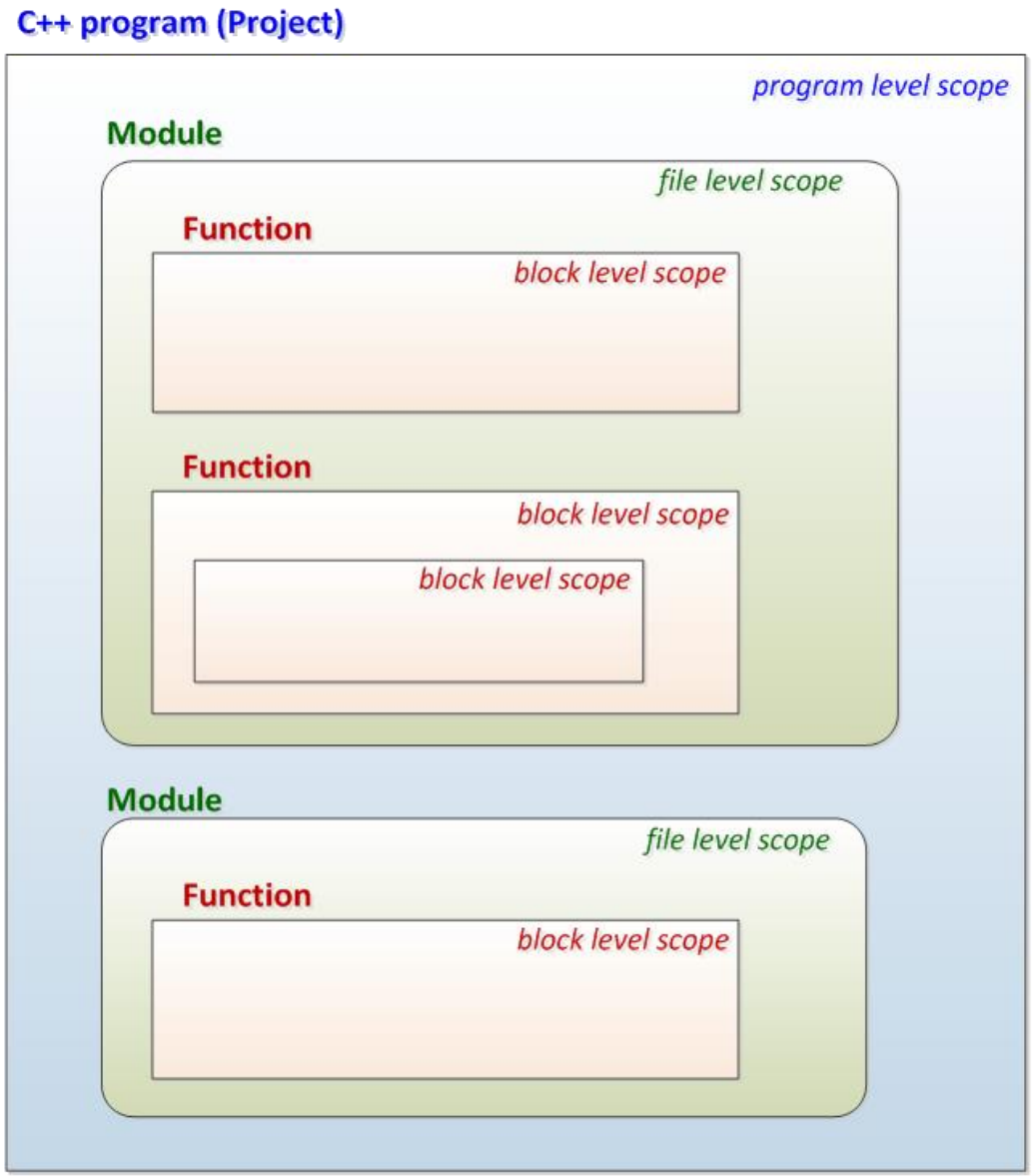
file level

A file level variable is ***only visible in the module containing its declaration***.
Identifiers having file level scope are those that are declared outside the functions of the module and that are declared with **internal linkage** using the **static** storage class.

program level

A program level variable is ***accessible from the functions of all the modules*** (all compilation units) of a program.
Global variables that are declared outside the functions of a module (that is with **external linkage**) have program level scope.

Namespaces and storage classes



Variable scopes

Namespaces and storage classes

❖ Storage classes of variables

▪ Lifetime of variables

- A *lifetime* is a period of program execution where the given variable exists.

static lifetime	Identifiers having a **static** or **extern** storage class have a static lifetime.
automatic lifetime	Within blocks, variables defined without the **static** storage class and the parameters of functions have automatic (local) lifetime.
dynamic lifetime	Independently of their storage classes, memory blocks that are allocated by the operator **new** and deallocated by the operator **delete** have dynamic lifetime.

Namespaces and storage classes

❖ Storage classes of variables

▪ Storage classes of block level variables

- Automatic variables: Automatic variables are created when control is passed to their block and they are deleted when that block is exited.

```
{
    definitions, declarations and
    statements
}
```

```
int main()
{
    double limit = 123.729;
    // the value of the double type variable limit is 123.729
    {
        long limit = 41002L;
        // the value of the long type variable limit is 41002L
    }
    // the value of the double type variable limit is 123.729
}
```

Namespaces and storage classes

❖ Storage classes of variables

▪ Storage classes of block level variables

- The register storage class: The **register** storage class can only be used for automatic local variables and function parameters.

```
void Swap(register int &a, register int &b) {
   register int c = a;
                a = b;
                b = c;
}
```

```
long StrToLong(const string& str)
{
    register int index = 0;
    register long sign = 1, num = 0;
    // leaving out leading blanks
    for(index=0; index < str.size()
                && isspace(str[index]); ++index);
    // sign?
    if( index < str.size()) {
        if( str[index] == '+' ) { sign =  1; ++index; }
        if( str[index] == '-' ) { sign = -1; ++index; }
    }
    // digits
    for(; index < str.size() && isdigit(str[index]); ++index)
        num = num * 10 + (str[index] - '0');
    return sign * num;
}
```

Namespaces and storage classes

❖ Storage classes of variables

▪ Storage classes of block level variables

- Local variables with static lifetime.

```
#include <iostream>
using namespace std;

unsigned Fibo() {
    static unsigned a0 = 0, a1 = 1;
    unsigned a2 = a0 + a1;
    a0 = a1;
    a1 = a2;
    return a2;
}

int main() {
    for (register int i=2; i<10; i++)
        cout << Fibo() << endl;
}
```

Namespaces and storage classes

❖ Storage classes of variables

▪ Storage classes of file level variables

```
#include <iostream>
using namespace std;

const unsigned first = 0, second = 1;
static unsigned a0 = first, a1 = second;

void FiboInit() {
    a0 = first;
    a1 = second;
}

unsigned Fibo() {
    unsigned register a2 = a0 + a1;
    a0 = a1;
    a1 = a2;
    return a2;
}

int main() {
    for (register int i=2; i<5; i++)
        cout << Fibo() << endl;
    FiboInit();
    for (register int i=2; i<8; i++)
        cout << Fibo() << endl;
}
```

Namespaces and storage classes

❖ Storage classes of variables

- **Storage classes of program level variables**

Same definitions (only one of them can be used)	Declarations
double sum;	**extern double** sum;
double sum = 0;	
extern double sum = 0;	
int vector[12];	**extern int** vector[];
extern int vector[12] = {0};	**extern int** vector[12];
extern const int size = 7;	**extern const int** size;

Namespaces and storage classes

- Storage classes of variables
- Storage classes of functions
- Modular programs in C++
- Namespaces

Namespaces and storage classes

❖ Storage classes of functions

Definitions (only one of them can be used)	Prototypes
double GeomMean(**double** a, **double** b) { return sqrt(a*b); }	**double** GeomMean(**double**, **double**);
extern double GeomMean(double a, double b) { return sqrt(a*b); }	**extern** double GeomMean(double, double);
static double GeomMean(double a, double b) { return sqrt(a*b); }	**static** double GeomMean(double, double);

Namespaces and storage classes

❖ Storage classes of functions

```
#include <iostream>
using namespace std;

extern int Random(int);
double Sin(double);

int main() {
    for (int i=0; i<5; i++)
        cout << Random(12)<< endl;
    cout << Sin(45) << endl;
    extern const double e;
    cout << e << endl;
}
```

```
#include <ctime>
#include <cstdlib>
#include <cmath>
using namespace std;

extern const double pi = asin(1)*2;
extern const double e = exp(1);

static double Degree2Radian(double);
static bool firstrnd = true;

int Random(int n) {
    if (firstrnd) {
        srand((unsigned)time(0));
        firstrnd = false;
    }
    return rand() % n;
}

double Sin(double degree) {
    return sin(Degree2Radian(degree));
}

static double Degree2Radian(double degree) {
   return degree/180*pi;
}
```

Namespaces and storage classes

❖ Storage classes of functions

- Accessing the compiled C functions from within C++ source: use the *extern "C"* declaration.

```
extern "C" double sqrt(double a);

extern "C" {
    double sin(double a);
    double cos(double a);
}

extern "C" {
    #include <rs232.h>
}
```

Namespaces and storage classes

- Storage classes of variables
- Storage classes of functions
- Modular programs in C++
- Namespaces

Namespaces and storage classes

❖ Modular programs in C++

```
// MatModule.h
#ifndef _MATMODULE_H_
#define _MATMODULE_H_

    #include <cmath>
    using namespace std;

    // file level definitions/declarations
    const double pi = asin(1)*2;
    const double e = exp(1);
    static double Degree2Radian(double);

    // program level declarations
    int Random(int n);
    double Sin(double degree);

#endif
```

```
// MatModule.cpp
#include <ctime>
#include <cstdlib>
using namespace std;
#include "MatModul.h"

static bool firstrnd = true;

int Random(int n) {
    if (firstrnd) {
        srand((unsigned)time(0));
        firstrnd = false;
    }
    return rand() % n;
}

double Sin(double degree) {
    return sin(Degree2Radian(degree));
}

static double Degree2Radian(double degree) {
   return degree/180*pi;
}
```

Namespaces and storage classes

❖ Modular programs in C++

```
// MainModule.cpp
#include <iostream>
using namespace std;
#include "MatModul.h"

int main() {
    for (int i=0; i<5; i++)
        cout << Random(12)<< endl;
    cout << Sin(45) << endl;
    cout << e << endl;
}
```

Namespaces and storage classes

- Storage classes of variables
- Storage classes of functions
- Modular programs in C++
- Namespaces

Namespaces and storage classes

❖ Namespaces

- **The default namespaces of C++ and the scope operator**

- C++ enclose Standard library elements in the namespace called ***std***.
- In order to be able to refer the elements of a namespace, we have to know how to use the scope operator (::).
- With the help of ***::*** operator, we can refer names having file and program level scope (that is identifiers of the ***global*** namespace) from any block of a program.

Namespaces and storage classes

❖ Namespaces

- **The default namespaces of C++ and the scope operator**

```
#include <iostream>
using namespace std;

long a = 12;
static int b = 23;

int main() {
    double a = 3.14159265;
    double b = 2.71828182;
    {
        long a = 7, b;
        b = a * (::a + ::b);
        // the value of b is 7*(12+23) = 245
        ::a = 7;
        ::b = 29;
    }
}
```

```
#include <iostream>
int main() {
    std::cout<<"C++ language"<<std::endl;
    std::cin.get();
}
```

Namespaces and storage classes

❖ Namespaces

- **Creating and using user-defined namespaces**
- Creating namespaces

```
namespace myOwnNamespace {
    definitions and declarations
}
```

```
#include <iostream>
#include <string>

// declarations
namespace nsexample {
    extern int value;
    extern std::string name;
    void ReadData(int & a, std::string & s);
    void PrintData();
}

// definitions
namespace nsexample {
    int value;
    std::string name;
    void PrintData() {
        std::cout << name << " = " << value << std::endl;
    }
}

void nsexample::ReadData(int & a, std::string & s) {
    std::cout << "name:   "; getline(std::cin, s);
    std::cout << "value: "; std::cin >> a;
}
```

Namespaces and storage classes

❖ Namespaces

- **Creating and using user-defined namespaces**
- Accessing the identifiers of a namespace

Directly by using the scope operator:

```
int main() {
    nsexample::name = "Sum";
    nsexample::value = 123;
    nsexample::PrintData();
    nsexample::ReadData(nsexample::value,
                        nsexample::name);
    nsexample::PrintData();
    std::cin.get();
}
```

*Or by using the directive **using namespace**:*

```
using namespace nsexample;
int main() {
    name = "Sum";
    value = 123;
    PrintData();
    ReadData(value, name);
    PrintData();
    std::cin.get();
}
```

Namespaces and storage classes

❖ Namespaces

- **Creating and using user-defined namespaces**
- Accessing the identifiers of a namespace

*Or by providing **using** -declarations:*

```
int PrintData() {
    std::cout << "Name conflict" << std::endl;
    return 1;
}

int main() {
    using nsexample::name;
    using nsexample::value;
    using nsexample::ReadData;
    name = "Sum";
    value = 123;
    nsexample::PrintData();
    ReadData(value, name);
    PrintData();
    std::cin.get();
}
```

Namespaces and storage classes

❖ Namespaces

- **Creating and using user-defined namespaces**
- Nested namespaces, namespace aliases:

```
namespace Project {
    typedef unsigned char byte;
    typedef unsigned short word;

    namespace Intel {
        word ToWord(byte lo, byte hi) {
            return lo + (hi<<8);
        }
    }

    namespace Motorola {
        word ToWord(byte lo, byte hi) {
            return hi + (lo<<8);
        }
    }
}
```

Namespaces and storage classes

❖ Namespaces

- **Creating and using user-defined namespaces**
- Nested namespaces, namespace aliases:

```
using namespace Project::Intel;
int main() {
    cout << hex;
    cout << ToWord(0xab,0x12)<< endl;              // 12ab
}
// ------------------------------------------------------
int main() {
    using Project::Motorola::ToWord;
    cout << hex;
    cout << ToWord(0xab,0x12)<< endl;             // ab12
}
// ------------------------------------------------------
using namespace Project;
int main() {
    cout << hex;
    cout << Intel::ToWord(0xab,0x12)<< endl;      // 12ab
    cout << Motorola::ToWord(0xab,0x12)<< endl;   // ab12
}
// ------------------------------------------------------
int main() {
    cout<<hex;
    cout<<Project::Intel::ToWord(0xab,0x12)<< endl;    // 12ab
    cout<<Project::Motorola::ToWord(0xab,0x12)<<endl; // ab12
}
```

Namespaces and storage classes

❖ Namespaces

- **Creating and using user-defined namespaces**
- Nested namespaces, namespace aliases:

```
namespace alias =externalNS::internalNS … ::most_internal_NS;

int main() {
    namespace ProcI = Project::Intel;
    namespace ProcM = Project::Motorola;
    cout << hex;
    cout << ProcI::ToWord(0xab,0x12)<< endl;      // 12ab
    cout << ProcM::ToWord(0xab,0x12)<< endl;      // ab12
}
```

Namespaces and storage classes

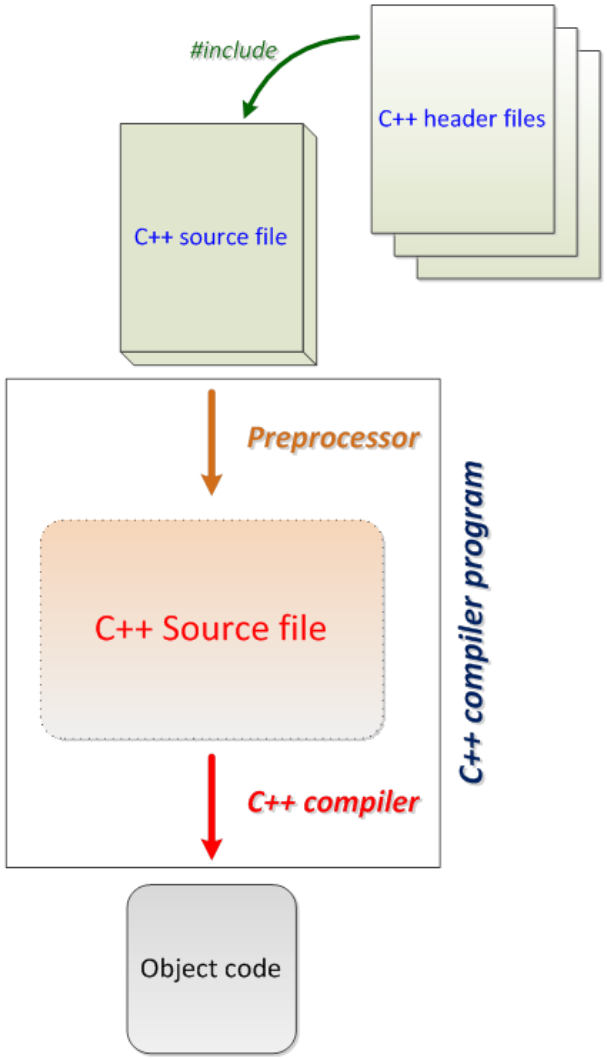
- Anonymous namespaces:

```
#include <iostream>
#include <string>
using namespace std;

namespace {
    string password;
}

namespace {
    bool ChangePassword() {
        bool success = false;
        string old, new1, new2;
        cout << "Previous password: "; getline(cin, old);
        if (password == old) {
            cout << "New Password: "; getline(cin, new1);
            cout << "New Password: "; getline(cin, new2);
            if (new1==new2) {
                password = new1;
                success = true;
            }
        }
        return success;
    } // ChangePassword()
}

int main() {
    password = "qwerty";
    if (ChangePassword())
        cout << "Successful password change!" << endl;
    else
        cout << "Not a successful password change!" << endl;
}
```

Chapter II. Modular programming in C++

- The basics of functions
- How to use functions on a more professional level?
- Namespaces and storage classes
- Preprocessor directives of C++

Preprocessor directives of C++



The compilation process in C++

Preprocessor directives of C++

- Including files
- Conditional compilation
- Using macros

Preprocessor directives of C++

- **Including files**

```
#include <header>

#include "filepath"
```

The following table sums up the keywords and language elements that can be used in header files:

C++ elements	Example
Comments	// comment
Conditional directives	#ifdef MOTOROLA
Macro definitions	#define INTEL
#include directives	#include <string>
Enumerations	enum response {no, yes, maybe};
Constant definitions	const double pi=3.1415265;

Preprocessor directives of C++

❖ Including files

C++ elements	Example
Namespaces having an identifier	namespace nsrand { }
Name declarations	struct vector3D;
Type definitions	struct complex {double re, im;};
Variable declarations	extern double point[];
Function prototypes	double Average(double, double);
inline function definitions	inline int Sqr(int a) {return a*a;}
template declarations	template <class T> T Sqr(T a);
template definitions	template <class T> T Sqr(T a) {return a*a;}

Preprocessor directives of C++

❖ Including files

The following elements should never be placed in *include* files:

- definitions of non-*inline* functions,
- variable definitions,
- definitions of anonymous namespaces.

Preprocessor directives of C++

❖ Including files

- One part of the elements that can be placed in a header file have to be included only once in a code.
- That is why, all *header* files have to have a special preprocessing structure based on conditional directives:

```
// Module.h
#ifndef _MODULE_H_
#define _MODULE_H_

    the content of the header file

#endif
```

Preprocessor directives of C++

- Including files
- Conditional compilation
- Using macros

Preprocessor directives of C++

❖ Conditional compilation

- The code parts to be compiled under conditions can be selected in many ways, by using the following preprocessor directives: *#if*, *#ifdef*, *#ifndef*, *#elif*, *#else* and *#endif*.

```
#define TEST 1

int main() {
    const int size = 5;
    int array[size] = { 12, 23, 34, 45, 56 };
    int i;

    for (int i = 0; i < size; i++) {
        array[i] *= 2;
        #if TEST != 0
            cout << "i = " << i << endl;
            cout << "\tarray[i] = " << array[i] << endl;
        #endif
    }
}
```

Preprocessor directives of C++

❖ Conditional compilation

- Instead of the structure above, it is better to use a solution that examines the definition of the symbol *TEST*.

```
#define TEST

int main() {
    const int size = 5;
    int array[size] = { 12, 23, 34, 45, 56 };
    int i;
    for (int i = 0; i < size; i++) {
        array[i] *= 2;
        #if defined(TEST)
            cout << "i = " << i << endl;
            cout << "\tarray[i] = " << array[i] << endl;
        #endif
    }
}
```

Preprocessor directives of C++

❖ Conditional compilation

- Each pair of the following checkings return the same results:

```
#if defined(TEST)
 ... // defined
#endif
```

```
#ifdef TEST
 ... // defined
#endif
```

```
#if !defined(TEST)
... // not defined
#endif
```

```
#ifndef TEST
... // not defined
#endif
```

Preprocessor directives of C++

❖ Conditional compilation

- If the following structure is used, we can choose between two code parts:

```
#if constant_expression
      codepart1
#else
      codepart2
#endif
```

Preprocessor directives of C++

❖ Conditional compilation

- The following example creates the two-dimensional array named *array* depending on the value of the symbol *SIZE*:

```
#define SIZE 5
int main() {
    #if SIZE <= 2
        int array[2][2];
        const int size = 2;
    #else
        int array[SIZE][SIZE];
        const int size = SIZE;
    #endif
}
```

Preprocessor directives of C++

❖ Conditional compilation

- For more complex structures, it is recommended to use multi-way branches.

```
#if constant_expression1
      codepart1
#elif constant_expression2
      codepart2
#else
     codepart3
#endif
```

Preprocessor directives of C++

❖ Conditional compilation

- The following example integrates the declaration file on the basis of the manufacturer:

```
#define IBM 1
#define HP 2
#define DEVICE IBM

#if DEVICE == IBM
    #define DEVICE_H "ibm.h"
#elif DEVICE == HP
    #define DEVICE_H "hp.h"
#else
    #define DEVICE_H "other.h"
#endif
#include DEVICE_H
```

Preprocessor directives of C++

- Including files
- Conditional compilation
- Using macros

Preprocessor directives of C++

❖ Using macros

▪ Symbolic constants

- Symbolic constants can be created by using the simple form of the ***#define*** directive:

```
#define  identifier

#define  identifier  replacement_text
```

```
#define SIZE 4
#define DEBUG
#define AND &&
#define VERSION "v1.2.3"
#define BYTE unsigned char
#define EOS '\0'

int main() {
    int v[SIZE];
    BYTE m = SIZE;
    if (m > 2 AND m < 5)
        cout << "2 < m < 5" << endl;
    #ifdef DEBUG
        cout << VERSION << endl;
    #endif
}
```

Preprocessor directives of C++

❖ Using macros

- **Symbolic constants**

```
#define DEBUG
#define AND &&
const int size = 4;
const string version = "v1.2.3";
const char eos = '\0';
typedef unsigned char byte;

int main() {
    int v[size];
    byte m = size;
    if (m > 2 AND m < 5)
        cout << "2 < m < 5" << endl;
    #ifdef DEBUG
        cout << version << endl;
    #endif
}
```

Preprocessor directives of C++

❖ Using macros

▪ Parameterized macros

- The macros can be efficiently used in much more cases if they are parameterized. The general form of a function-like parameterized macro:

```
#define  identifier(parameter_list)  replacement_text
```

- Using (calling) a macro:

```
identifier(argument_list)
```

Preprocessor directives of C++

❖ Using macros

▪ Parameterized macros

```
// determining the absolute value of x
#define ABS(x)    ( (x) < 0 ? (-(x)) : (x) )

// calculating the maximum value of a and b
#define MAX(a,b) ( (a) > (b) ? (a) : (b) )

// calculating the square of X
#define SQR(X)    ( (X) * (X) )

// generating a random number within a given interval
#define RANDOM(min, max) \
    ((rand()%(int)(((max) + 1)-(min)))+ (min))

int main() {
    int x = -5;
    x = ABS(x);
    cout << SQR(ABS(x));
    int a = RANDOM(0,100);
    cout << MAX(a, RANDOM(12,23));
}
```

Preprocessor directives of C++

❖ Using macros

▪ Parameterized macros

- All advantages of macros (type-independence, faster code) can be achieved by using **inline** functions/function templates (instead of macros) while keeping the C++ code legible:

```
template <class tip> inline tip Abs(tip x) {
  return  x < 0 ? -x : x;
}

template <class tip> inline tip Max(tip a, tip b) {
  return a > b ? a : b;
}

template <class tip> inline tip Sqr(tip x) {
  return x * x;
}
```

```
inline int Random(int min, int max) {
  return rand() % (max + 1 - min) + min;
}

int main() {
    int x = -5;
    x = Abs(x);
    cout << Sqr(Abs(x));
    int a = Random(0,100);
    cout << Max(a, Random(12,23));
}
```

Preprocessor directives of C++

❖ Using macros

▪ Undefining a macro

- A macro can be undefined anytime and can be redefined again, even with a different content. It can be undefined by using the ***#undef*** directive.

Original source code

```
int main() {
    #define MACRO(x) (x) + 7
    int a = MACRO(12);
    #undef MACRO
    a = MACRO(12);
    #define MACRO 123
    a = MACRO
}
```

Substituted code

```
int main() {
    int a = (12) + 7;
    a = MACRO(12);
    a = 123
}
```

Preprocessor directives of C++

❖ Using macros

▪ Macro operators

- If the **#** character is placed before the parameter in a macro, the value of the parameter is replaced as enclosed within quotation marks (i.e. as a string).

```
#include <iostream>
using namespace std;

#define INFO(variable) \
        cout << #variable " = " << variable <<endl;

int main() {
   unsigned  x = 23;
   double   pi = 3.14259265;
   string    s = "C++ language";
   INFO(x);
   INFO(s);
   INFO(pi);
}
```

```
x = 23
s = C++ language
pi = 3.14259
```

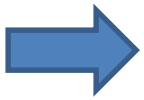
Preprocessor directives of C++

❖ Using macros

▪ Macro operators

- By using the **##** operator, two syntactic units (*tokens*) can be concatenated.

```
#include <string>
using namespace std;

struct person {
    string name;
    string info;
};

string Ivan_Info ="K. Ivan, Budapest,  9";
string Alice_Info ="O. Alice, Budapest, 33";

#define PERSON(NAME) { #NAME, NAME ## _Info }

int main(){
    person people[] = {
        PERSON (Ivan),
        PERSON (Alice)
    };
}
```



```
int main(){
    person people[] = {
        { "Ivan", Ivan_Info },
        { "Alice", Alice_Info }
    };
}
```

Preprocessor directives of C++

❖ Using macros

▪ Predefined macros

Macro	*Description*	*Example*
__DATE__	String constant containing the date of the compilation.	"Oct 02 2013"
__TIME__	String constant containing the time of the compilation.	"10:02:04"
__TIMESTAMP__	The date and time of the last modification of the source file in a string constant"	"Mon Jul 29 07:33:29 2013"
__FILE__	String constant containing the name of the source file.	"c:\\preproc.cpp"
__LINE__	A numeric constant, containing the number of the actual line of the source file (numbering starts from 1).	1223
__STDC__	Its value is 1 if the compiler works as an ANSI C++, otherwise it is not defined.	
__cplusplus	Its value is 1, if its value is tested in a C++ source file, otherwise it is not defined.	

Preprocessor directives of C++

❖ Using macros

- **#line, #error and #pragma directives**

- The ***#line*** directive forces C++ compilers not to signal the error code in the C++ source text but in the original source file written in another special language.

```
#line beginning_number

#line beginning_number "filename"
```

Preprocessor directives of C++

❖ Using macros

- **#line, #error and #pragma directives**

- An ***#error*** directive can be used to print out a compilation error message which contains the text provided in the statement:

#error error_message

```
#if !defined(__cplusplus)
    #error Compilation can only be carried out in C++ mode!
#endif
```

Preprocessor directives of C++

❖ Using macros

- **#line, #error and #pragma directives**

- ***#pragma*** directives are used to control the compilation process in an implementation-dependent way.

```
#pragma pack(8)
// aligned to 8 byte boundary
#pragma pack(push, 1)
// aligned to 1 byte boundary
#pragma pack(pop)
// aligned again to 8 byte boundary

#pragma warning( disable : 2312 79 )
```

- The empty directive (#) can also be used, but it does not affect preprocessing.

KỸ THUẬT LẬP TRÌNH HỆ CƠ ĐIỆN TỬ

Programming Engineering in Mechatronics

Giảng viên: TS. Nguyễn Thành Hùng

Đơn vị: Bộ môn Cơ điện tử, Viện Cơ khí

Hà Nội, 2020

Chapter III. Object-oriented programming in C++

- Introduction to the object-oriented world
- Classes and objects
- Inheritance (derivation)
- Polymorphism
- Class templates

Chapter III. Object-oriented programming in C++

- Introduction to the object-oriented world
- Classes and objects
- Inheritance (derivation)
- Polymorphism
- Class templates

Introduction to the object-oriented world

- Basics
- Basic principles
- An object-oriented example code

Introduction to the object-oriented world

❖ Basics

▪ Class

- A *class* determines the abstract features of a object, including its features (attributes, fields, properties) and its behaviour (what the thing can do, methods, operations and functions).
- We can say that a class is a scheme describing the nature of something.
- Both the integrated properties and the methods of a class are called class members.

Introduction to the object-oriented world

❖ Basics

- **Object**

- An object is a scheme (an example) of a class.

- **Instance**

- Instance means an actual object created at runtime: *myCar* is an instance of the class *Truck.*
- The set of the property values of the actual object is called the *state* of that object.

Introduction to the object-oriented world

❖ Basics

▪ Method

- Methods are responsible for the capabilities of objects: the methods of *myCar*: *Brake()*, *Ignition()*, …
- In C++, methods are rather called member functions.

▪ Message passing

- Message passing is the process during which an object sends data to another object or "asks" another object to execute one of its methods.
- On the code level, message passing is realised by calling a method in C++.

Introduction to the object-oriented world

- Basics
- Basic principles
- An object-oriented example code

Introduction to the object-oriented world

❖ Basic principles

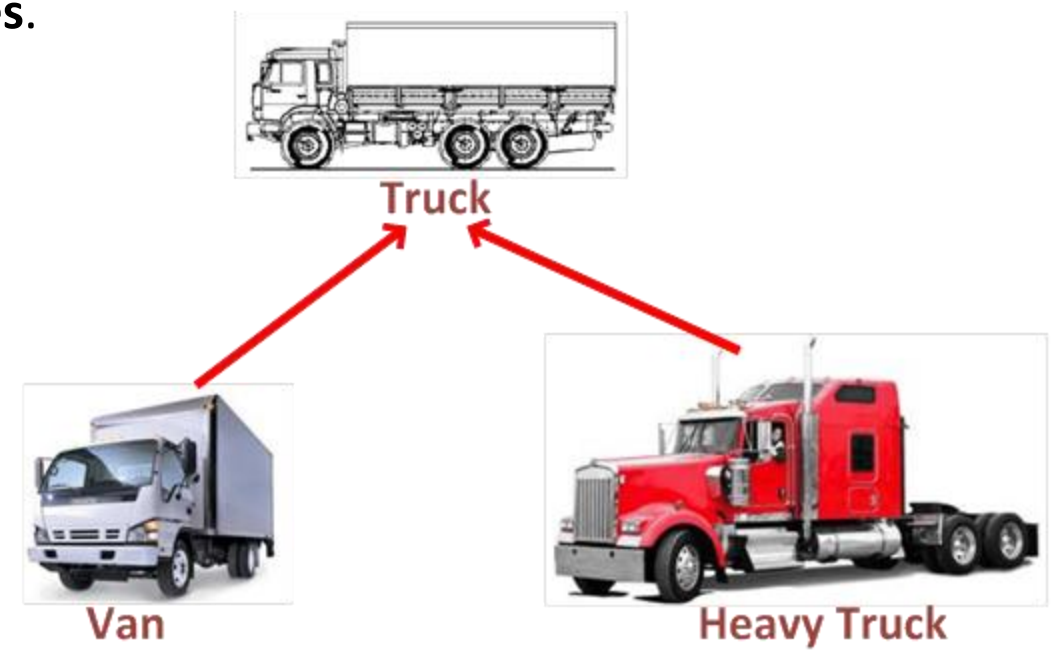
- **Encapsulation, data hiding**

- Classes principally consist of **features** (**state**) and **methods** (**behaviour**).
- There are some **features** and **methods** that we **hide from other objects**. These are internal (***private*** or ***protected***) states and behaviour.
- However, the others are made ***public***.
- According to the basic principles of OOP, the **state features have to be private** while **most of the methods may be public**.

❖ Basic principles

▪ Inheritance

- Inheritance means creating specific versions of a class that inherit the features and behaviour of their parent class (base class) and use them as if they were of their own. The classes created in this way are called ***subclasses*** or ***derived classes***.



Inheritance

❖ Basic principles

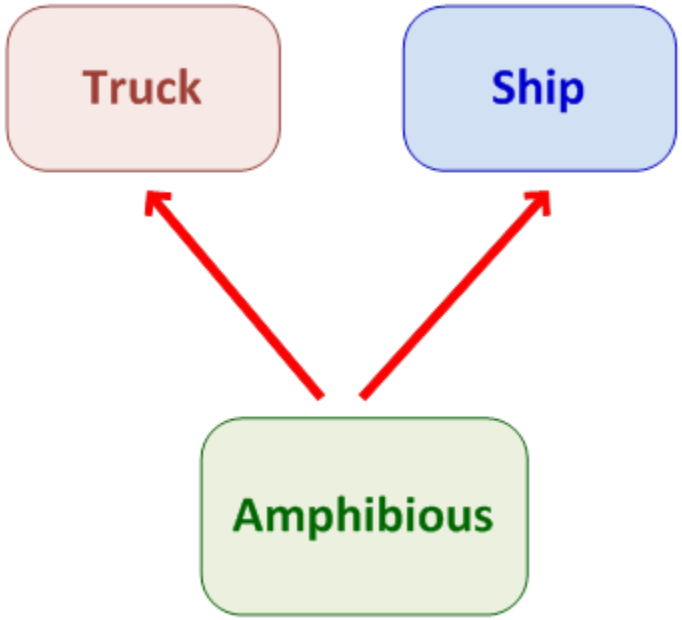
▪ Inheritance

- Actually, inheritance is an *is-a* relation: *myCar* is a *HeavyTruck*, a *HeavyTruck* is a *Truck*. So *myCar* has the methods of both *HeavyTruck* and *Truck*.
- Both derived classes have one direct parent class, namely *Truck*. This inheritance method is called ***single inheritance***.
- ***Multiple inheritance*** means that a derived class inherits the members of more direct parent classes.

Introduction to the object-oriented world

❖ Basic principles

- **Inheritance**



Multiple inheritance

Introduction to the object-oriented world

❖ Basic principles

▪ Abstraction

- Abstraction simplifies complex reality by modelling problems with their corresponding classes and it has its effects on the level of inheritance appropriate for these problems.
- Abstraction can be achieved through composition.
- An interface determines how to send element or receive from element messages and it gives information about the interaction between the components of the class.

Introduction to the object-oriented world

❖ Basic principles

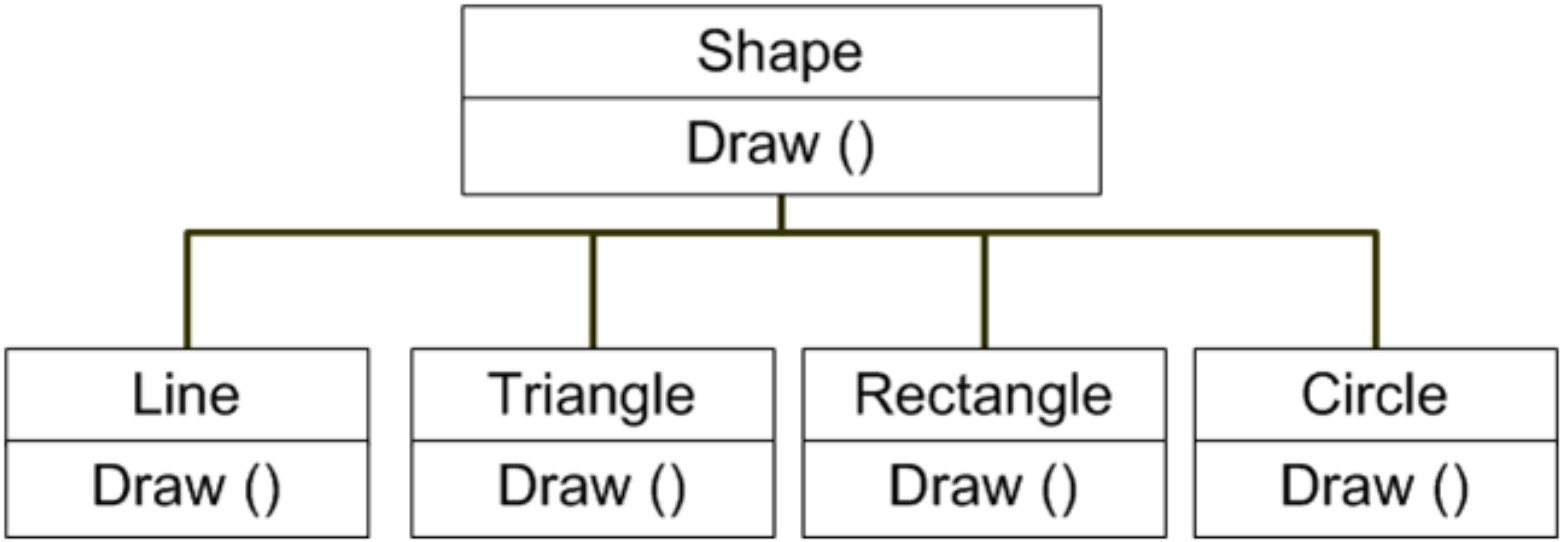
- **Abstraction**

Introduction to the object-oriented world

❖ Basic principles

▪ Polymorphism

- Polymorphism makes it possible to replace the content of some inherited (deprecated) behaviour forms (methods) with a new one in the derived class and to treat the new, replaced methods as the members of the parent class.



Introduction to the object-oriented world

- Basics
- Basic principles
- An object-oriented example code

Introduction to the object-oriented world

❖ An object-oriented example code

```
#include <iostream>
#include <string>
using namespace std;

class Truck {
   protected:
      string manufacturer;
      string engine;
      string brake_system;
      string maximum_load;
   public:
      Truck(string ma, string en, string brake,
                  double load) {
         manufacturer = ma;
         engine = en;
         brake_system = brake;
         maximum_load = load;
      }
      void StartUp() { }
      void StepOnTheGas() { }
      virtual void Brake() {
          cout<<"Classic braking."<< endl;
      }
      void TurnLeft() { }
      void TurnRight() { }
};
```

```
class Van : public Truck {
    public:
      Van(string ma, string en, string brake)
            : Truck(ma, en, brake, 20) {  }
};

class HeavyTruck : public Truck {
    public:
      HeavyTruck(string ma, string en, string brake, double load)
            : Truck(ma, en, brake, load) {  }
      void Brake() { cout<<"Brake with EBS."<< endl; }
      void Navigate() {}
};

int main() {
    Van post("ZIL", "Diesel", "airbrake");
    post.Brake();      // Classic braking.
    HeavyTruck myCar("Kamaz", "gas engine", "EBS", 40);
    myCar.Brake();    // Brake with EBS.
}
```

Chapter III. Object-oriented programming in C++

- Introduction to the object-oriented world
- Classes and objects
- Inheritance (derivation)
- Polymorphism
- Class templates

Classes and objects

❖ **A class declaration has two parts:**

- The header of the class contains the keyword **class**/**struct**, followed by the name of the class.
- The class body is enclosed within curly brackets followed by a semi-colon ➔ contain the data members, member functions, and the keywords regulating access to the members and followed by a colon: **public**, **private** (hidden) and **protected**.

Classes and objects

❖ **General form of a class:**

```
class ClassName {
  public:
    type4 Function1(parameterlist1) {   }
    type5 Function2(parameterlist2) {   }
  protected:
     type3 data3;
  private:
     type1 data11, data12;
     type2 data2;
};
```

Classes and objects

- From structures to classes
- More about classes
- Operator overloading

Classes and objects

From structures to classes

- **A little revision**

```
struct Employee{
    int employeeID;
    string name;
    float salary;
};

void IncreaseSalary(Employee& a, float percent) {
    a.salary *= (1 + percent/100);
}

int main() {
    Employee engineer;
    engineer.employeeID = 1234;
    engineer.name = "Tony Clever";
    engineer.salary = 2e5;
    IncreaseSalary(engineer,12);
    cout << engineer.salary << endl;

    Employee *pAccountant = new Employee;
    pAccountant->employeeID = 1235;
    pAccountant->name = "Sarah Rich";
    pAccountant->salary = 3e5;
    IncreaseSalary(*pAccountant,10);
    cout << pAccountant->salary << endl;
    delete pAccountant;
}
```

Classes and objects

❖ From structures to classes

- **Grouping together data and operations**

```
struct Employee {
    int employeeID;
    string name;
    float salary;
    void IncreaseSalary(float percent) {
        salary *= (1 + percent/100);
    }
};

int main() {
    Employee engineer;
    engineer.employeeID = 1234;
    engineer.name = "Tony Clever";
    engineer.salary = 2e5;
    engineer.IncreaseSalary(12);
    cout << engineer.salary << endl;

    Employee *pAccountant = new Employee;
    pAccountant->employeeID = 1235;
    pAccountant->name = "Sarah Rich";
    pAccountant->salary = 3e5;
    pAccountant->IncreaseSalary(10);
    cout << pAccountant->salary << endl;
    delete pAccountant;
}
```

Classes and objects

❖ From structures to classes

▪ Data hiding

- In object-oriented programming it is required that the data members of classes could not be accessed directly from the outside.
- The type **struct** offers complete access to its members by default, whereas the **class** type completely hides its members from the outside.
- The access of class elements can be defined by programmers as well with the keywords **private**, **protected** and **public**.

Classes and objects

From structures to classes

- **Data hiding**

```
class Employee{
  private:
    int employeeID;
    string name;
    float salary;
  public:
    void IncreaseSalary(float percent) {
        salary *= (1 + percent/100);
    }
    void SetData(int code, string n, float s) {
        employeeID = code;
        name = n;
        salary = s;
    }
    float GetSalary() const {
        return salary;
    }
};
```

```
int main() {
    Employee engineer;
    engineer.SetData(1234, "Tony Clever", 2e5);
    engineer.IncreaseSalary(12);
    cout << engineer.GetSalary() << endl;

    Employee *pAccountant = new Employee;
    pAccountant->SetData(1235, "Sarah Rich", 3e5);
    pAccountant->IncreaseSalary(10);
    cout << pAccountant->GetSalary() << endl;
    delete pAccountant;
}
```

❖ From structures to classes

▪ Constructors

- A ***constructor*** is a ***member function*** the name of which corresponds to ***the name of the class*** and ***has no return type***.
- A constructor only has to initialise the memory space already allocated for the object.
- A class has two constructors by default: a constructor without parameters (*default*) and *a copy constructor*.

Classes and objects

From structures to classes

Constructors

```
class Employee{
  private:
    int employeeID;
    string name;
    float salary;
  public:
    // default
    Employee() {
        employeeID = 0;
        name = "";
        salary = 0;
    }
    // by parameters
    Employee(int code, string n, float s) {
        employeeID = code;
        name = n;
        salary = s;
    }
    // by copying values
    Employee(const Employee & a) {
        employeeID = a.employeeID;
        name = a.name;
        salary = a.salary;
    }
    void IncreaseSalary(float percent) {
        salary *= (1 + percent/100);
    }
    void SetName(string n) {
        name = n;
    }
    float GetSalary() const {
        return salary;
    }
};
```

Classes and objects

- From structures to classes
  - Constructors

```
int main() {
    Employee employee;
    employee.SetName("Stephen Smith");

    Employee engineer(1234, "Tony Clever", 2e5);
    engineer.IncreaseSalary(12);
    cout << engineer.GetSalary() << endl;

    Employee firstEngineer = engineer;
    // or: Employee firstEngineer(engineer);
    firstEngineer.IncreaseSalary(50);
    cout << firstEngineer.GetSalary() << endl;

    Employee *pEmployee = new Employee;
    pEmployee->SetName("Stephen Smith");
    delete pEmployee;

    Employee *pAccountant;
    pAccountant = new Employee(1235, "Sarah Rich", 3e5);
    pAccountant->IncreaseSalary(10);
    cout << pAccountant->GetSalary() << endl;
    delete pAccountant;

    Employee *pFirstEngineer=new Employee(engineer);
    pFirstEngineer->IncreaseSalary(50);
    cout << pFirstEngineer->GetSalary() << endl;
    delete pFirstEngineer;
}
```

Classes and objects

❖ From structures to classes

▪ Constructors

- Constructors with and without parameters are often contracted by introducing default arguments:

```
class Employee{
  private:
    int employeeID;
    string name;
    float salary;
  public:
    Employee(int code = 0, string n ="", float s=0) {
        employeeID = code;
        name = n;
        salary = s;
    }
    …
}
```

Classes and objects

- From structures to classes
- **Constructors**
- Using member initialisation lists

```
class Employee{
  private:
    int employeeID;
    string name;
    float salary;
  public:
    Employee(int code=0, string n="", float s=0)
        : employeeID(code), name(n), salary(s) { }
    Employee(const Employee & a)
        : employeeID(a.employeeID), name (a.name),
          salary(a.salary) { }
    …
}
```

Classes and objects

- From structures to classes
- **Constructors**
- Explicit initialisation of objects

```
class Number
{
  private:
    int n;
  public:
    explicit Number( int x) {
        n = x;
        cout << "int: " << n << endl;
    }
    Number( float x) {
        n = x < 0 ? int(x-0.5) : int(x+0.5);
        cout << "float: " << n << endl;
    }
};

int main() {
  Number a(123);     // explicit call
  Number b = 123;    // implicit (not explicit) call
}
```

❖ From structures to classes

▪ Destructor

- C++ offers a special member function, the destructor, in which we can free the allocated resources.
- The name of a destructor has to be provided as a class name with the tidle character (~).
- A destructor, just like constructors, does not return any value.

Classes and objects

From structures to classes

Destructor

```
class Employee{
  private:
    int employeeID;
    string name;
    float salary;
    int *pWorkhours;
  public:
    Employee(int code = 0, string n ="", float s=0) {
        employeeID = code;
        name = n;
        salary = s;
        pWorkhours = new int[12];
        for (int i=0; i<12; i++) pWorkhours[i]=0;
    }
    Employee(const Employee & a) {
        employeeID = a.employeeID;
        name = a.name;
        salary = a.salary;
        pWorkhours = new int[12];
        for (int i=0; i<12; i++)
            pWorkhours[i]=a.pWorkhours[i];
    }
    ~Employee() {
        delete[] pWorkhours;
        cout << name << " deleted" << endl;
    }
    void IncreaseSalary(float percent) {
        salary *= (1 + percent/100);
    }
    void SetWorkhours(int month, int hours) {
        if (month >= 1 && month <=12) {
            pWorkhours[month-1]=hours;
        }
    }
    float GetSalary() const {
        return salary;
    }
};
```

Classes and objects

❖ From structures to classes

▪ Destructor

- If a destructor is not written for a class, the compiler automatically adds an empty destructor for that class.

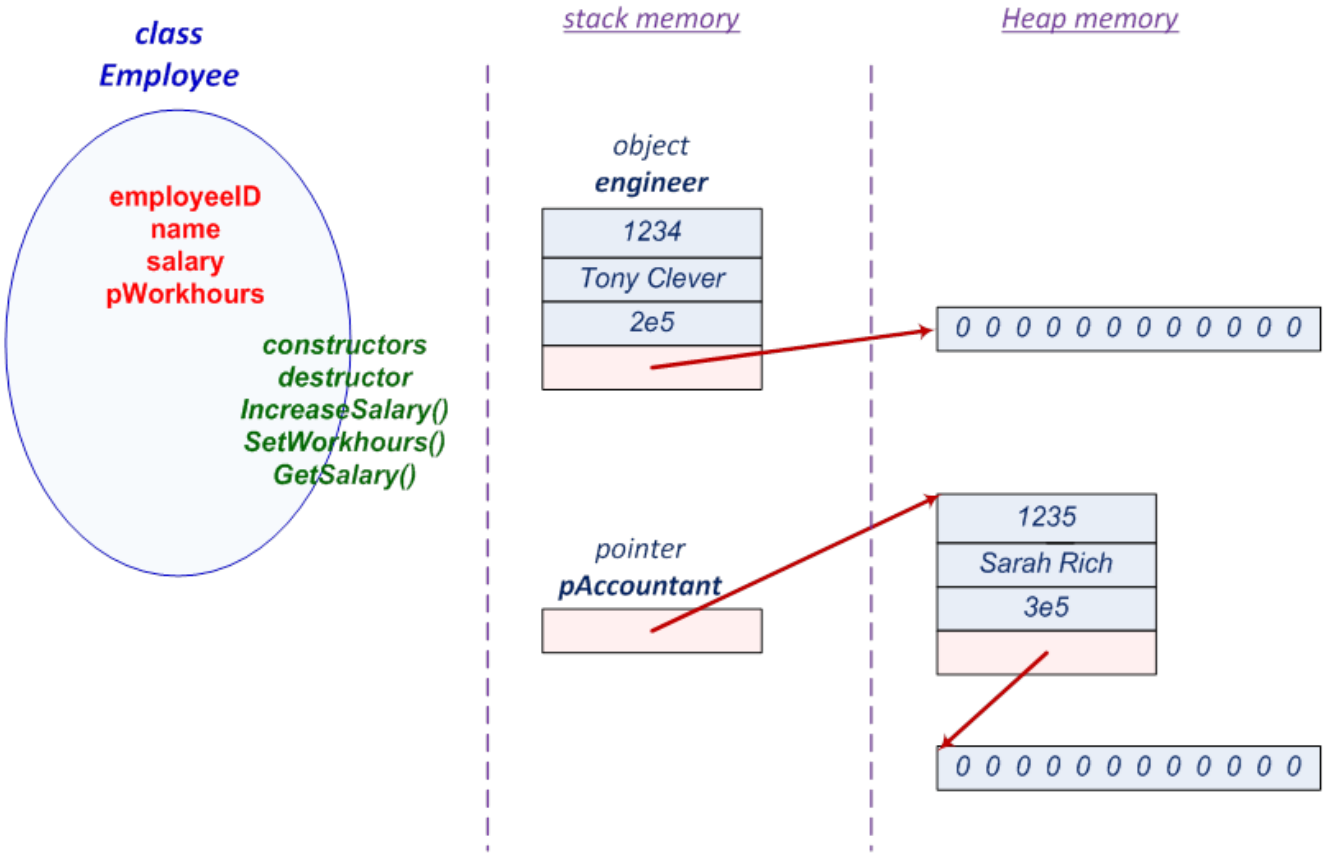
```
int main() {
    Employee engineer(1234, "Tony Clever", 2e5);
    engineer.IncreaseSalary(12);
    engineer.SetWorkhours(3,192);
    cout << engineer.GetSalary() << endl;

    Employee *pAccountant;
    pAccountant = new Employee(1235, "Sarah Rich", 3e5);
    pAccountant->IncreaseSalary(10);
    pAccountant->SetWorkhours(1,160);
    pAccountant->SetWorkhours(12,140);
    cout << pAccountant->GetSalary() << endl;
    delete pAccountant;
}
```

```
224000
330000
Sarah Rich deleted
Tony Clever deleted
```

Classes and objects

❖ From structures to classes

- **Objects of a class, the pointer this**



The class *Employee* and its objects

Classes and objects

❖ From structures to classes

▪ Objects of a class, the pointer this

- Each member function has an invisible parameter (**this**) in which a pointer to the actual object is passed to the function when it is called.
- All references to data members are inserted in a program code automatically in the following way:

```
this->datamember
```

Classes and objects

❖ From structures to classes

▪ Objects of a class, the pointer this

- Programmers may also use the pointer **this** within member functions.

```
class Employee{
  private:
    int employeeID;
    string name;
    float salary;
  public:
    Employee(int employeeID=0, string name="", float salary=0){
        this->employeeID = employeeID;
        this->name = name;
        this->salary = salary;
    }
};
```

Classes and objects

- From structures to classes
- More about classes
- Operator overloading

More about classes

Static class members

- A static data member that is created in only one instance belongs directly to the class; therefore it is available for it even if there are no objects for that class.
- A static data member should not be initialised within its class (independently of its access restriction).
- If a static data member is public, then it can be used anywhere in the program code by the name of the class and the scope operator (**::**).

Classes and objects

More about classes

- **Static class members**

```
#include <iostream>
#include <iomanip>
#include <cmath>
using namespace std;

class Math {
  public:
    enum Unit {degree, radian};
  private:
    static double dDegree2Radian;
    static Unit eMode;
  public:
    static const double Pi;

    static double Sin(double x)
           {return sin(eMode == radian ? x : dDegree2Radian*x);}
    static double Cos(double x)
           {return cos(eMode == radian ? x : dDegree2Radian*x);}
static void MeasureUnit(Unit mode = radian) {
                                    eMode = mode; }
    void PrintOutPI() { cout.precision(18); cout<<Pi<<endl;}
};

// creating and initialising static members
const double Math::Pi = 3.141592653589793238462;
double Math::dDegree2Radian = Math::Pi/180;
Math::Unit Math::eMode = Math::radian;
```

Classes and objects

More about classes

- **Static class members**

```
int main() {
    double y = Math::Sin(Math::Pi/6);     // counts in radian
    Math::MeasureUnit(Math::degree);      // counts in degrees
    y = Math::Sin(30);
    Math::MeasureUnit(Math::radian);  // counts in radian
    y = Math::Sin(Math::Pi/6);

    Math m;                                // class instance
    m.MeasureUnit(Math::degree);           // or
    m.MeasureUnit(m.degree);
    y = m.Sin(30);
    m.MeasureUnit(m.radian);               // or
    m.MeasureUnit(Math::radian);
    y = m.Sin(Math::Pi/6);
    m.PrintOutPI();
}
```

Classes and objects

More about classes

- **How to structure classes**
- Implicit inline member functions

```
class Point {
    private:
        int x,y;
    public:
        Point(int a = 0, int b = 0) { x = a; y = b; }
        int GetX() const { return x; }
        int GetY() const { return y; }
        void SetX(int a) { x = a; }
        void SetY(int a) { y = a; }
        void Move(int a, int b) { x = a; y = b; }
        void Move(const Point& p) { x = p.x; y = p.y; }
        void PrintOut() const { cout<<"("<<x<<","<<y<<")\n"; }
};
```

Classes and objects

More about classes

- **How to structure classes**
- Class structures in C++/CLI applications

```
class Point {
   private: int x,y;
   public: Point(int a = 0, int b = 0) { x = a; y = b; }
   public: int GetX() const { return x; }
   public: int GetY() const { return y; }
   public: void SetX(int a) { x = a; }
   public: void SetY(int a) { y = a; }
   public: void Move(int a, int b) { x = a; y = b; }
   public: void Move(const Point& p) { x = p.x; y = p.y; }
   public: void PrintOut() const { cout<<"("<<x<<","<<y<<")\n"; }
};
```

Classes and objects

More about classes

How to structure classes

- Storing member functions in separate modules

```
//Point.h
#ifndef __PONT_H__
#define __PONT_H__
 class Point {
   private:
      int x,y;
   public:
      Point(int a = 0, int b = 0);
      int GetX() const;
      int GetY() const;
      void SetX(int a);
      void SetY(int a);
      void Move(int a, int b);
      void Move(const Point& p);
      void PrintOut() const;
 };
#endif
```

```
//Point.cpp
#include <iostream>
using namespace std;
#include "Point.h"

Point::Point(int a, int b) {
   x = a; y = b;
}

int Point::GetX() const {
   return x;
}

int Point::GetY() const {
   return y;
}

void Point::SetX(int a) {
   x = a;
}

void Point::SetY(int a) {
   y = a;
}

void Point::Move(int a, int b) {
   x = a; y = b;
}

void Point::Move(const Pont& p) {
   x = p.x; y = p.y;
}

void Point::PrintOut() const {
   cout<<"("<<x<<","<<y<<")\n";
}
```

Classes and objects

More about classes

Friend functions and classes

- The **friend** mechanism makes it possible for us to access the *private* and *protected* members of a class from a function outside the class.

```
class AClass;

class BClass {
 public:
  int Count(int x) { return x++; }
};

class CClass {
  friend long Sum(int a, int b);
  friend int BClass::Count(int x);
  friend class AClass;
  // ...
};

long Sum(int a, int b) {
  return long(a) + b;
}
```

Classes and objects

More about classes

- **Friend functions and classes**

```
#include <iostream>
#include <cmath>
using namespace std;

class Point {
      friend double Distance(const Point & p1,
                             const Point & p2);
   private:
      int x,y;
   public:
      Point(int a = 0, int b = 0) { x = a; y = b; }
      int GetX() const { return x; }
      int GetY() const { return y; }
      void SetX(int a) { x = a; }
      void SetY(int a) { y = a; }
      void Move(int a, int b) { x = a; y = b; }
      void Move(const Point& p) { x = p.x; y = p.y; }
      void PrintOut() const { cout<<"("<<x<<","<<y<<")\n"; }
};

double Distance(const Point & p1, const Point & p2) {
  return sqrt(pow(p1.x-p2.x,2.0)+pow(p1.y-p2.y,2.0));
}

int main() {
  Point p, q;
  p.Move(1,2);
  q.Move(4,6);
  cout<<Distance(p,q)<<endl;
}
```

Classes and objects

More about classes

- **What can we also add to classes?**
- Constant data members of objects

```
class User {
    string password;
 public:
    const string name;
    User(string user, string psw="") : name(user) {
        password=psw;
    }
    void SetPassword(string newpsw) { password = newpsw;}
};

int main() {
   User nata("Lafenita");
   User gardener("Liza");
   nata.SetPassword("Atinefal1223");
   gardener.SetPassword("Azil729");
   cout<<nata.name<<endl;
   cout<<gardener.name<<endl;
   User alias = nata;
   // alias = gardener;   // error!
}
```

Classes and objects

More about classes

- **What can we also add to classes?**
- Reference type data members

```
class Sensor {
   private:
      int data;
   public:
      Sensor(int x) { data = x; }
      int Read() { return data; }
};

class Controller {
   private:
      Sensor& sensor;
   public:
      Controller(Sensor& s) : sensor(s) {}
      void ReceiveData() { cout<<sensor.Read(); }
};

int main() {
   Sensor speed(0x17);
   Controller ABS(speed);
   ABS.ReceiveData();
}
```

Classes and objects

More about classes

- **What can we also add to classes?**
- Data members as objects

```
class Sensor {
   private:
      int data;
   public:
      Sensor(int x) { data = x; }
      int Read() { return data; }
};

class Controller {
   private:
      Sensor sensor;
   public:
      Controller() : sensor(0x17) {}
      void ReceiveData() { cout<<sensor.Read(); }
};

int main() {
   Controller ABS;
   ABS.ReceiveData();
}
```

Classes and objects

More about classes

- **What can we also add to classes?**

- Pointers to class members

  In order to define pointers correctly, we have to use the name of the class and the scope operator:

class Class;	forward class declaration,
int Class::*p;	*p* is a pointer to a data member of type **int**,
void (Class::*pf unct)(int);	*pfunct* may point to a member function that is called with an argument of type **int** and that returns no value.

Classes and objects

More about classes

- **What can we also add to classes?**
- Pointers to class members

```
#include <iostream>
using namespace std;

class Class {
  public:
    int a;
    void f(int b) { a += b;}
};

int main() {
   // pointer to a data member of Class of type int
   int Class::*intptr = &Class::a;

   // pointer to the member function of Class
   // of type void and with a parameter of type int
   void (Class::* functptr)(int) = &Class::f;

   // creating object instances
   Class object;
   Class * pobject = new Class();

   // accessing the data member with a pointer
   object.*intptr = 10;
   pobject->*intptr = 100;

   // calling the member function f() with a pointer
   (object.*functptr)(20);
   (pobject->*functptr)(200);

   cout << object.a << endl;     // 30
   cout << pobject->a << endl;    // 300
   delete pobject;
}
```

Classes and objects

❖ More about classes

- **What can we also add to classes?**

- Pointers to class members

  By using **typedef**, expressions containing pointers are easier to be handled:

```
typedef int Class::*pointer_int;

typedef void (Class::*pointer_funct)(int);

…
pointer_int intptr = &Class::a;

pointer_funct functptr = &Class::f;
```

Classes and objects

- From structures to classes
- More about classes
- Operator overloading

❖ Operator overloading

- An operator function can be used if one of its parameters is a class of type **class** or **struct**. General declaration of operator functions:

```
type operator op(parameterlist);
```

Where the sequence *op* can be replaced by any of the following operators:

[]	()	.	->	++	--	&	**new**
*	+	-	~	!	/	%	**new**[]
<<	>>	<	>	<=	>=	==	**delete**
!=	^	|	&&	||	=	*=	**delete**[]
/=	%=	+=	-=	<<=	>>=	&=	
^=	|=	,	->*				

❖ Operator overloading

- **The following operators cannot be overloaded :** member selection (.), indirect member selection (.*), scope (*::*), conditional (*?:*) and the operators **sizeof** and **typeid** since their overloading would result in undesired side effects.
- The assignment (=), the "address of" (&) and the comma (,) operations can be **applied** to objects without overloading.
- Overloading operators **does not result in modifying** the operator precedence and associativity, and it is **not possible** to introduce new operations.

Classes and objects

❖ Operator overloading

▪ Creating operator functions

Expression	Operator(♣)	Member function	External function
♣ a	+ - * & ! ~ ++ --	A::operator ♣ ()	operator ♣ (A)
a ♣	++ --	A::operator ♣ (int)	operator ♣ (A, int)
a ♣ b	+ - * / % ^ & | < > == != <= >= << >> && || ,	A::operator ♣ (B)	operator ♣ (A, B)
a ♣ b	= += -= *= /= %= ^= &= |= <<= >>= []	A::operator ♣ (B)	-
a(b, c...)	()	A::operator()(B, C...)	-
a->b	->	A::operator->()	-

❖ Operator overloading

▪ Creating operator functions

- The operators *=*, *()*, *[]* and *->* can only be overloaded by non-static member functions.
- The operators ***new*** and ***delete*** are overloaded with static member functions.
- All other operator functions can be created as member functions or external (in general *friend*) functions.

❖ Operator overloading

▪ Creating operator functions

- Binary operands:

Realisation	Syntax	Actual call
member function	X op Y	X.operator op(Y)
external function	X op Y	operator op(X,Y)

❖ Operator overloading

▪ Creating operator functions

- Unary operands:

Realisation	Syntax	Actual call
member function	op X	X.operator op()
member function	X op	X.operator op(0)
external function	op X	operator op(X)
external function	X op	operator op(X,0)

Classes and objects

❖ Operator overloading

▪ Creating operator functions

```
//Vector.h
#ifndef __VectorH__
#define __VectorH__

class Vector {
inline friend Vector  operator+ (const Vector& v1,
                                 const Vector& v2);
  private:
    int size, *p;
  public:
    // ----- Constructors -----
    // initialising a vector of a given size
    Vector(int n=10) {
        p = new int[size=n];
        for (int i = 0; i < size; ++i)
            p[i] = 0;    // initialising all elements to zero
    }
    // Initialising by another vector - a copy constructor
    Vector(const Vector& v) {
         p = new int[size=v.size];
         for (int i = 0; i < size; ++i)
            p[i] = v.p[i];   // copying all elements
    }
    // Init.by a traditional vector with n number of elements
    Vector(const int a[], int n) {
        p = new int[size=n];
        for (int i = 0; i < size; ++i)
           p[i] = a[i];
    }
    // ----- Destructor -----
    ~Vector() {delete[] p; }

    // ----- Member function returning the size -----
    int GetSize() const { return size; }

    // ----- Operator functions -----
    int& operator [] (int i) {
        // checking whether index is within bounds
        if (i < 0 || i > size-1)
            throw i;
        return p[i];
    }

    const int& operator [] (int i) const {
        return p[i];
    }

    Vector operator = (const Vector& v) {
        delete[] p;
        p=new int [size=v.size];
        for (int i = 0; i < size; ++i)
           p[i] = v.p[i];
        return *this;
    }

    Vector operator += (const Vector& v) {
        int m = (size < v.size) ? size: v.size;
        for (int i = 0; i < m; ++i)
           p[i] += v.p[i];
        return *this;
    }
};
```

Classes and objects

❖ Operator overloading

- **Creating operator functions**

```
// ----- External function  -----
inline Vector operator+(const Vector& v1, const Vector& v2) {
    Vector sum(v1);
    sum+=v2;
    return sum;
}
#endif
```

Classes and objects

❖ Operator overloading

▪ Creating operator functions

```
#include <iostream>
using namespace std;
#include "Vector.h"

void show(const Vector& v) {
  for (int i=0; i<v.GetSize(); i++)
    cout<<v[i]<<'\t';
  cout<<endl;
}

int main() {
    int a[5]={7, 12}, b[7]={2, 7, 12, 23, 29};
    Vector x(a,5);      // x:  7  12   0   0  0
    Vector y(b,7);      // y:  2   7  12  23 29  0  0
    Vector z;           // z:  0   0   0   0  0  0  0  0  0  0
    try {
       x = y;           // x:  2   7  12  23 29  0  0
       x = Vector(a,5);// x:  7  12   0   0  0
       x += y;          // x:  9  19  12  23 29
       z = x + y;       // z: 11  26  24  46 58
       z[0] = 102;
       show(z);         // z:102  26  24  46 58
    }
    catch (int n) {
        cout<<"Not a valid array index: "<<n<<endl;
    }
    const Vector v(z);
    show(v);             // v:102  26  24  46 58
    // v[0] = v[1]+5;    // error: assignment of read-only…
}
```

❖ Operator overloading

- **Using type conversion operator functions**

```
#include <cmath>
#include <iostream>
using namespace std;

class Complex {
  public:
    Complex () { re=im=0; }
    Complex(double a) : re(a), im(0) { }
    // conversion constructor
    Complex(double a, double b) : re(a), im(b) { }
    // conversion operator
    operator double() {return sqrt(re*re+im*im);}

    Complex operator *= (const Complex & k) {
       Complex t;
       t.re=(k.re*re)-(k.im*im);
       t.im=(k.re*im)+(re*k.im);
       return *this = t;
    }
    void Print() const { cout << re << "+" << im << "i"; }
  private:
    double re, im;
  friend Complex operator*(const Complex&, const Complex&);
};
```

Classes and objects

❖ Operator overloading

- **Using type conversion operator functions**

```
Complex operator*(const Complex& k1, const Complex& k2) {
    Complex k=k1;
    k*= k2;
    return k;
}

int main() {
    Complex k1(7), k2(3,4), k3(k2);
    cout << double(k3)<< endl;              // printed value:  5
    cout <<double(Complex(10))<< endl;      // printed value: 10
    Complex x(2,-1), y(3,4);
    x*=y;
    x.Print();                              // 10+5i
}
```

Classes and objects

❖ Operator overloading

- **Extending classes with input/output operations**
- To "teach" I/O data streams based on classes to handle the objects of user-defined classes.

```
#include <cmath>
#include <sstream>
#include <iostream>
using namespace std;

class Complex {
  public:
    Complex(double a=0, double b=0) : re(a), im(b) {}
    operator double() {return sqrt(re*re+im*im);}

    Complex operator *= (const Complex & k) {
       Complex t;
       t.re=(k.re*re)-(k.im*im);
       t.im=(k.re*im)+(re*k.im);
       return *this = t;
    }
  private:
    double re, im;
  friend Complex operator*(const Complex&, const Complex&);
  friend istream & operator>>(istream &, Complex &);
  friend ostream & operator<<(ostream &, const Complex &);
};
```

Classes and objects

❖ Operator overloading

- **Extending classes with input/output operations**

```
Complex operator*(const Complex& k1, const Complex& k2) {
    Complex k=k1;
    k*= k2;
    return k;
}

// The format of data input: 12.23+7.29i and 12.23-7.29i
istream & operator>>(istream & is, Complex & c) {
    string s;
    getline(is, s);
    stringstream ss(s);
    if (!(ss>>c.re>>c.im))
        c=Complex(0);
    return is;
}
```

Classes and objects

❖ Operator overloading

- **Extending classes with input/output operations**

```
// The format of data output: 12.23+7.29i and 12.23-7.29i
ostream & operator<<(ostream & os, const Complex & c) {
    os<<c.re<<(c.im<0? '-' : '+')<<fabs(c.im)<<'i';
    return os;
}

int main() {
    Complex a, b;
    cout<<"Please enter a complex number: ";  cin >> a;
    cout<<"Please enter a complex number: ";  cin >> b;
    cout<<"The product of the complex numbers: " << a*b;
    cout<<endl;
}
```

Chapter III. Object-oriented programming in C++

- Introduction to the object-oriented world
- Classes and objects
- Inheritance (derivation)
- Polymorphism
- Class templates

Inheritance (derivation)

- **Inheritance** makes it possible to create (to derive) new classes from already existing ones.
- Derivation means that a new class inherits the **public** and **protected** properties (data members) and behaviour (member functions) of already existing classes and it then uses them as its own.

Inheritance (derivation)

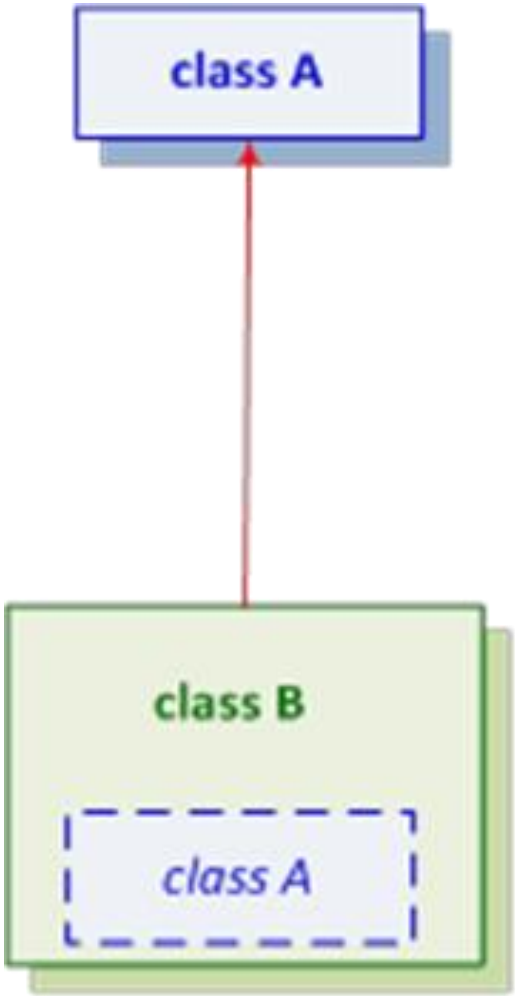
- However:

➢ already existing classes may be extended with a new class,

➢ new data members and member functions may be defined

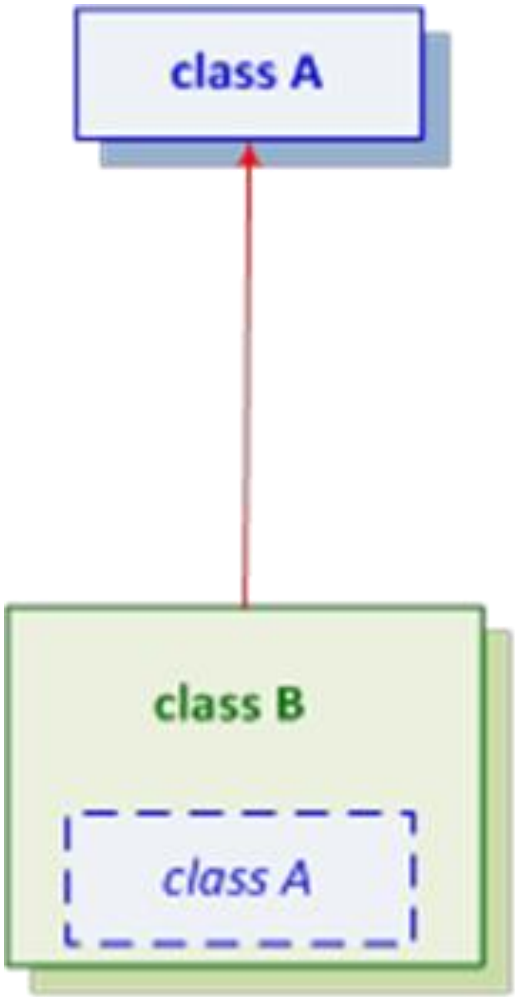
➢ or inherited member functions may be reinterpreted (replaced) if they become deprecated concerning their functioning (*polymorphism*).

Inheritance (derivation)



- *class A*, that is derived or from which members are inherited: *base class* , *ancestor class* , parent class, superclass
- *the operation: inheritance* , *derivation* , extending, subclassing
- *class B*, the result of derivation: *descendant class*, *derived class*, extended class, child class, subclass

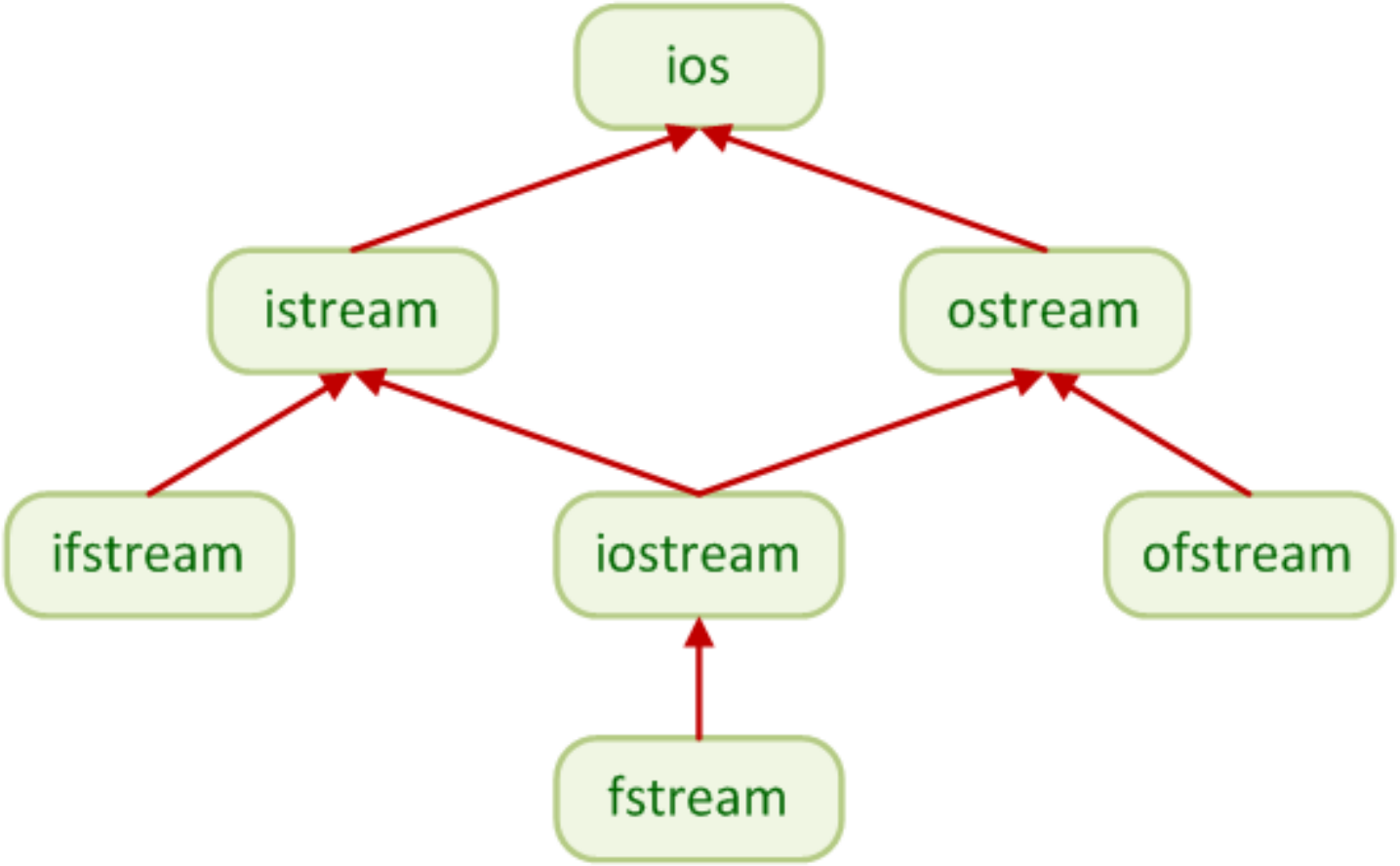
Inheritance (derivation)



The C++ program code that realises the relation above:

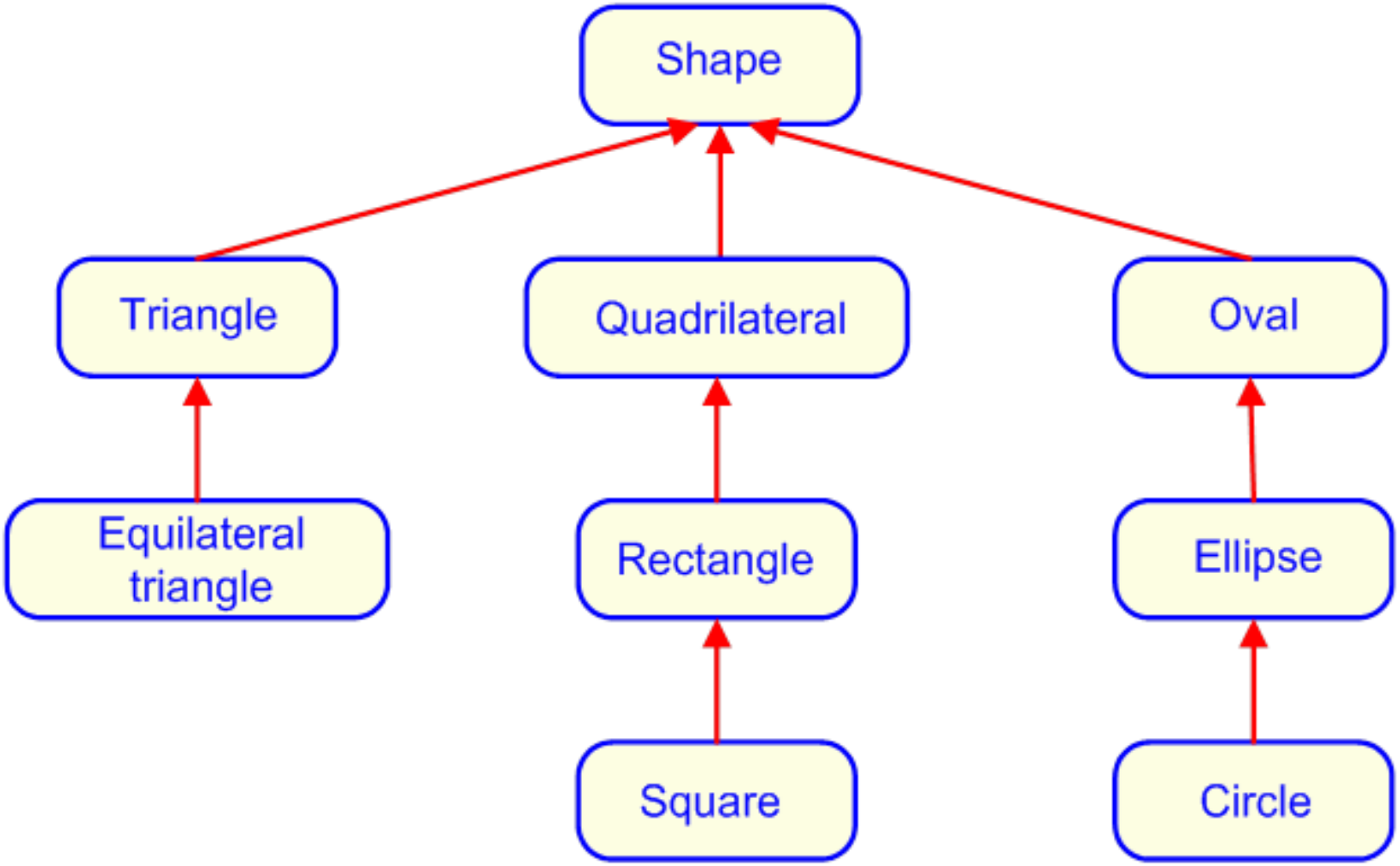
```
class ClassA {
   // ...
};

class ClassB : public  ClassA {
   // ...
};
```

Inheritance (derivation)



The multiple inheritance of I/O classes in C++

Inheritance (derivation)



Hierarchy of geometrical classes

Inheritance (derivation)

- Derivation of classes
- Initialising base class(es)
- Accessing class members in case of inheritance
- Virtual base classes in case of multiple inheritance
- Inheritance and/or composition?

Inheritance (derivation)

❖ Derivation of classes

- A **derived** (descendant) class is a class that **inherits** its data members and member functions from one or more already **defined class(es)**.
- The class from which a derived class inherits is called base class (ancestor class).
- A derived class inherits all the members of its base class; however, it only have access to the **public** and **protected** members of its base class as its own.

Inheritance (derivation)

❖ Derivation of classes

- The place where a derivation is indicated in a program code is the class header where the mode of derivation (**public**, **protected**, **private**) is indicated before the names of base classes:

```
class Derived : public Base1, ...private BaseN
{
 // the class body
};
```

Inheritance (derivation)

❖ Derivation of classes

```
class Point2D {
    protected:
        int x,y;
    public:
        Point2D(int a = 0, int b = 0) { x = a; y = b; }
        void GetPoint2D(int& a, int& b) const { a=x; b=y;}
        void Move(int a=0, int b=0) { x = a; y = b; }
        void Move(const Point2D& p) { x = p.x; y = p.y; }
        void PrintOut() const {
                cout<<'('<<x<<','<<y<<')'<<endl; }
};

class Point3D : public Point2D {
    protected:
        int z;
    public:
        Point3D(int a=0, int b=0, int c=0):Point2D(a,b),z(c) {}
        Point3D(Point3D & p):Point2D(p.x, p.y),z(p.z) {}
        void GetPoint3D(int& a, int& b, int& c) const {
            a = x; b = y; c = z; }
        void Move(int a=0, int b=0, int c=0) {
            x = a; y = b; z = c; }
        void Move(const Point3D& p) {
            Point2D::x = p.x; y = p.y; z = p.z;}
        void PrintOut() const {
            cout<<'('<<x<<','<<y<<','<<z<<')'<<endl;}
};
```

Inheritance (derivation)

❖ Derivation of classes

```
int main() {
   Point2D p1(12,23), p2(p1), p3;
   Point3D q1(7,29,80), q2(q1), q3;
   p1.PrintOut();                  // (12,23)
   q1.PrintOut();                  // (7,29,80)
   // q1 = p1;                  // error! --> should be avoided
   q2.Move(10,2,4);
   p2 = q2;
   p2.PrintOut();                  // (10,2)
   q2.PrintOut();                  // (10,2,4)
   q2.Point2D::PrintOut();          // (10,2)
   Visualise(p2);                    // (10,2)
   Visualise(q2);                    // (10,2)
}
```

Inheritance (derivation)

❖ Derivation of classes

- The keywords **public**, **protected** and **private** used in a derivation list restrict the access of inherited (public and protected) members in their new classes:

Mode of inheritance	Access in the base class	Access in the derived class
public	public protected	public protected
protected	public protected	protected protected
private	public protected	private *private*

Inheritance (derivation)

❖ Derivation of classes

- The access of any member (the access type of which is **protected** or **public** in the **base class**) can be manually set directly.

```
class Point3D : private Point2D {
   protected:
      int z;
      Point2D::x;
      Point2D::y;
   public:
      Point3D(int a=0, int b=0, int c=0):Point2D(a,b),z(c) {}
      Point3D(Point3D & p):Point2D(p.x, p.y),z(p.z) {}
      void GetPoint3D(int& a, int& b, int& c) const {
           a = x; b = y; c = z; }
      void Move(int a=0, int b=0, int c=0) {
           x = a; y = b; z = c; }
      void Move(const Point3D& p) {
           x = p.x; y = p.y; z = p.z;}
      void PrintOut() const {
           cout<<'('<<x<<','<<y<<','<<z<<')'<<endl;}
      Point2D:: GetPoint2D;
};
```

Inheritance (derivation)

- Derivation of classes
- Initialising base class(es)
- Accessing class members in case of inheritance
- Virtual base classes in case of multiple inheritance
- Inheritance and/or composition?

Inheritance (derivation)

❖ Initialising base class(es)

- In order that base class(es) be initialised, it is the extended version of **member initialisation list**s that is used.

```
class Point3D : public Point2D {
   protected:
      int z;
   public:
      Point3D(int a=0, int b=0, int c=0):Point2D(a,b),z(c) {}
      Point3D(Point3D & p):Point2D(p.x, p.y),z(p.z) {}
      // …
};
```

Inheritance (derivation)

- Derivation of classes
- Initialising base class(es)
- Accessing class members in case of inheritance
- Virtual base classes in case of multiple inheritance
- Inheritance and/or composition?

Inheritance (derivation)

❖ Accessing class members in case of inheritance

- **Accessing inherited members**

```
class Point3D : public Point2D {
   protected:
      int z;
   public:
      Point3D(int a=0, int b=0, int c=0):Point2D(a,b),z(c) {}
      Point3D(Point3D & p):Point2D(p.x, p.y),z(p.z) {}
      // …
      void Move(const Point3D& p) {
         Point2D::x = p.x; y = p.y; z = p.z;}
      // …
};
int main() {
  Point2D p1(12,23), p2(p1), p3;
  Point3D q1(7,29,80), q2(q1), q3;
  q2.Point2D::PrintOut();
  q2.Point2D::Move(1,2); // Moving in the x-y plane
  q2.Point2D::Move(p1);
  // …
}
```

Inheritance (derivation)

❖ Accessing class members in case of inheritance

- **Accessing inherited members**

The members of the base class Point2D:	**The members of the derived class Point3D**
protected: x, y	protected: x, y, z
public: Point2D(),	public: Point3D(int…), Point3D(Point3D&…),
GetPoint2D(), Move(int…), Move(const…), PrintOut()	GetPoint2D(),Point2D()::Move(int…), Point2D()::Move(const…), Point2D()::PrintOut(), GetPoint3D(), Move(int…), Move(const…), PrintOut()

Inheritance (derivation)

❖ Accessing class members in case of inheritance

▪ The friend relationship in inheritance

- In a ***derived class***, a ***friend*** of the ***base class*** can ***only access*** the members inherited from the base class.
- A "***friend***" of a ***derived class*** can ***only access*** public and protected members from the base class.

Inheritance (derivation)

- Derivation of classes
- Initialising base class(es)
- Accessing class members in case of inheritance
- Virtual base classes in case of multiple inheritance
- Inheritance and/or composition?

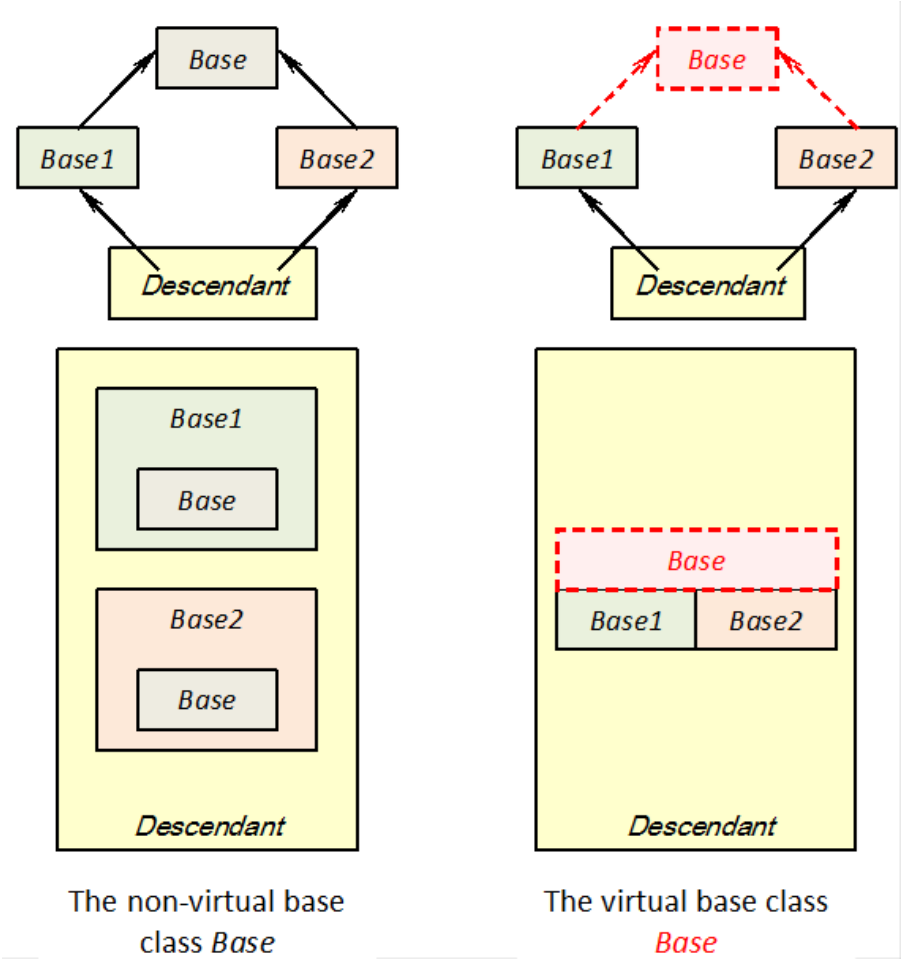
Inheritance (derivation)

❖ Virtual base classes in case of multiple inheritance

- In case of ***multiple inheritance***, it may be a ***problem*** if the ***same base class*** appears as many ***instances*** in the ***derived class***.
- If ***virtual base classes*** are used, problems of that type can be avoided.

Inheritance (derivation)

❖ Virtual base classes in case of multiple inheritance



Using virtual base classes

Inheritance (derivation)

❖ Virtual base classes in case of multiple inheritance

```
class Base {
    int q;
  public:
    Base(int v=0) : q(v) {};
    int GetQ() { return q;}
    void SetQ(int q) { this->q = q;}
};

// the virtual base class named Base
class Base1 : virtual public Base {
    int x;
 public:
    Base1(int i): x(i) {}
};

// the virtual base class named Base
class Base2: public virtual Base {
    int y;
  public:
    Base2(int i): y(i) {}
};
```

Inheritance (derivation)

❖ Virtual base classes in case of multiple inheritance

```
class Descendant:  public Base1,  public Base2 {
    int a,b;
  public:
    Descendant(int i=0,int j=0): Base1(i+j),Base2(j*i),a(i),b(j)
    {}
};
int main() {
   Descendant descendant;
   descendant.Base1::SetQ(100);
   cout << descendant.GetQ()<<endl;          // 100
   cout << descendant.Base1::GetQ()<<endl;   // 100
   cout << descendant.Base2::GetQ()<<endl;   // 100
   descendant.Base1::SetQ(200);
   cout << descendant.GetQ()<<endl;          // 200
   cout << descendant.Base1::GetQ()<<endl;   // 200
   cout << descendant.Base2::GetQ()<<endl;   // 200
}
```

Inheritance (derivation)

- Derivation of classes
- Initialising base class(es)
- Accessing class members in case of inheritance
- Virtual base classes in case of multiple inheritance
- Inheritance and/or composition?

Inheritance (derivation)

❖ Inheritance and/or composition?

- A big ***advantage*** of C++ programming language is that it supports the ***reusability*** of program code.
- ***Reusing*** means that a new program code is made without modifying the original one.

Inheritance (derivation)

❖ Inheritance and/or composition?

- If the object-oriented tools of C++ are used, there are three approaches to choose from:

➢ The most simple and frequent reuse of a code stored in a given class is when an **object instance** is created or when already existing objects (***cin*** , ***cout*** , ***string*** , *STL* etc.) are used in a program.

```
class X {
    // …
};

int main() {
    X a, *pb;
    pb = new X();
    cout<<"C++"<<endl;
    // …
    delete pb;
}
```

Inheritance (derivation)

❖ Inheritance and/or composition?

- Another possibility is to place objects of other classes in our own codes as member objects ➔ This method is called **composition**. If the new object will only contain a pointer or a reference to other objects, it is called an **aggregation**.

```
class X {
   // …
};

class Y {
  X x;     // composition
};

class Z {
  X& x;  // aggregation
  X *px;
};
```

Inheritance (derivation)

❖ Inheritance and/or composition?

- The third solution: when a new class is created by **public derivation** from other classes, then the relationship is of an **is-a** type ➔ a derived object behaves exactly the same way as its ancestor class.

```
// base class
class X {
    // …
};

// derived class
class Y : public X {
    // …
};
```

Inheritance (derivation)

❖ Inheritance and/or composition?

```
#ifndef __POINT_H__
#define __POINT_H__
class Point {
   protected:
      int x,y;
   public:
      Point(int a = 0, int b = 0) : x(a), y(b) {}
      Point(const Point& p) : x(p.x), y(p.y) {}
      int GetX() const { return x; }
      int GetY() const { return y; }
      void SetX(int a) { x = a; }
      void SetY(int a) { y = a; }
      void Move(int a, int b) { x = a; y = b; }
      void Move(const Point& p) { x = p.x; y = p.y; }
      void PrintOut() const { cout<<'('<<x<<','<<y<<')'<< endl; }
};
#endif
```

Inheritance (derivation)

❖ Inheritance and/or composition?

- **Reuse with composition**

```
#include "Point.h"

class Circle {
  protected:
    int r;
  public:
    Point p; // composition
    Circle(int x=0, int y=0, int r=0)
        : p(x, y), r(r) {}
    Circle(const Point& p, int r=0)
        : p(p), r(r) {}
    int GetR() {return r;}
    void SetR(int a) { r = a; }
};

int main() {
    Circle c1(100, 200, 10);
    c1.p.PrintOut();
    cout<<c1.p.GetX()<<endl;
    cout<<c1.GetR()<<endl;
}
```

Inheritance (derivation)

❖ Inheritance and/or composition?

- **Reuse by public inheritance**

```
#include "Point.h"

class Circle : public Point { // inheritance
  protected:
    int r;
  public:
    Circle(int x=0, int y=0, int r=0)
        : Point(x, y), r(r) {}
    Circle(const Point & p, int r=0)
        : Point(p), r(r) {}
    int GetR() {return r;}
    void SetR(int a) { r = a; }
};

int main() {
    Circle c1(100, 200, 10);
    c1.PrintOut();
    cout<<c1.GetX()<<endl;
    cout<<c1.GetR()<<endl;
}
```

Chapter III. Object-oriented programming in C++

- Introduction to the object-oriented world
- Classes and objects
- Inheritance (derivation)
- Polymorphism
- Class templates

Polymorphism

In C++, polymorphism rather means that the object of a derived class is accessed by a pointer or a reference in the base class.

- *Coercion* polymorphism means implicit and explicit type casts.

➢ In that case, the polymorphism of a given operation is made possible by different types that may be converted if needed.

- As an opposite to coercion, the so-called *ad-hoc* (“for that purpose”) polymorphism is better known by the name of function *overloading*.

➢ In that case, a compiler chooses the appropriate function from the variants on the basis of parameter types.

Polymorphism

- The extended version of this polymorphism is called *parametrical* or *compile-time* polymorphism, which makes it possible to execute the same code with any type.

➢ In C++, parametric polymorphism is realised by function and class *templates*. Using templates actually means reusing a C++ source code.

Polymorphism

- Virtual member functions
- Redefining virtual functions
- Early and late binding
- Virtual destructors
- Abstract classes and interfaces
- Run-time type informations in case of classes

Polymorphism

❖ Virtual member functions

- A virtual function is a **public** or **protected** member function of the base class.
- It can be redefined in the derived class in order that the behavior of the class would change.
- A virtual function is generally called by a reference or a pointer of a public base class, the actual value of which is determined at run-time (dynamic binding, late binding).
- Declaration of the virtual function:

```
class Example {
  public:
    virtual int vf();
};
```

Polymorphism

❖ Virtual member functions

- It is not necessary that a virtual function in the base class have a definition as well.
- Instead, the prototype of the function should be ended with the expression **= 0;** .
- In that case, it is a so-called **pure virtual function**:

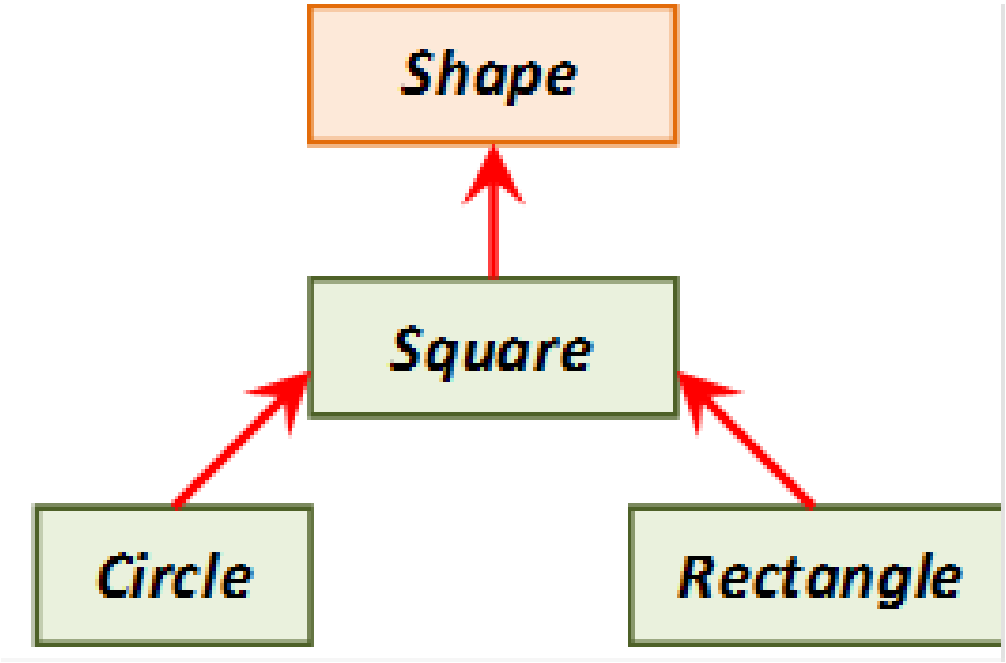
```
class Example {
  public:
    virtual int pvf() = 0;
};
```

Polymorphism

- Virtual member functions
- Redefining virtual functions
- Early and late binding
- Virtual destructors
- Abstract classes and interfaces
- Run-time type informations in case of classes

Polymorphism

- Redefining virtual functions



Polymorphism

❖ Redefining virtual functions

```
class Base {
    int q;
  public:
    Base(int v=0) : q(v) {};
    int GetQ() { return q;}
    void SetQ(int q) { this->q = q;}
};

// Abstract base class
class Shape {
  protected:
    int x, y;
  public:
    Shape(int x=0, int y=0) : x(x), y(y) {}
    virtual double Area()=0;
    virtual double Perimeter()=0;
    void Visualise() {
        cout<<'('<<x<<','<<y<<")\t";
        cout<<"\tArea: "<< Area();
        cout<<"\tPerimeter: "<< Perimeter() <<endl;
    }
};
```

```
class Square : public Shape {
  protected:
    double a;
  public:
    Square(int x=0, int y=0, double a=0)
          : Shape(x,y), a(a) {}
    double Area() {return a*a;}
    double Perimeter() {return 4*a;}
};

class Rectangle : public Square {
  protected:
    double b;
  public:
    Rectangle(int x=0, int y=0, double a=0,  double b=0)
          : Square(x,y,a), b(b) {}
    double Area() {return a*b;}
    double Perimeter() {return 2*(a+b);}
};

class Circle : public Square {
  const double pi;
  public:
    Circle(int x=0, int y=0, double r=0)
          : Square(x,y,r), pi(3.14159265) {}
    double Area() {return a*a*pi;}
    double Perimeter() {return 2*a*pi;}
};
```

Polymorphism

❖ Redefining virtual functions

```
int main() {
    Square s(12,23,10);
    cout<<"Square: ";
    s.Visualise();

    Circle c(23,12,10);
    cout<<"Circle: ";
    c.Visualise();

    Rectangle r(12,7,10,20);
    cout<<"Rectangle: ";
    r.Visualise();

    Shape* shapes[3] = {&s, &c, &r} ;
    for (int i=0; i<3; i++)
      shapes[i]->Visualise();
}
```

Polymorphism

❖ Redefining virtual functions

- Virtual functions and public inheritance make it possible to create external functions that can be called by every object in the class hierarchy:

```
void VisualiseAll(Shape& a) {
    cout<<"Area: "<<a.Area()<<endl;
    cout<<"Perimeter: "<<a.Perimeter()<<endl;
}
```

Polymorphism

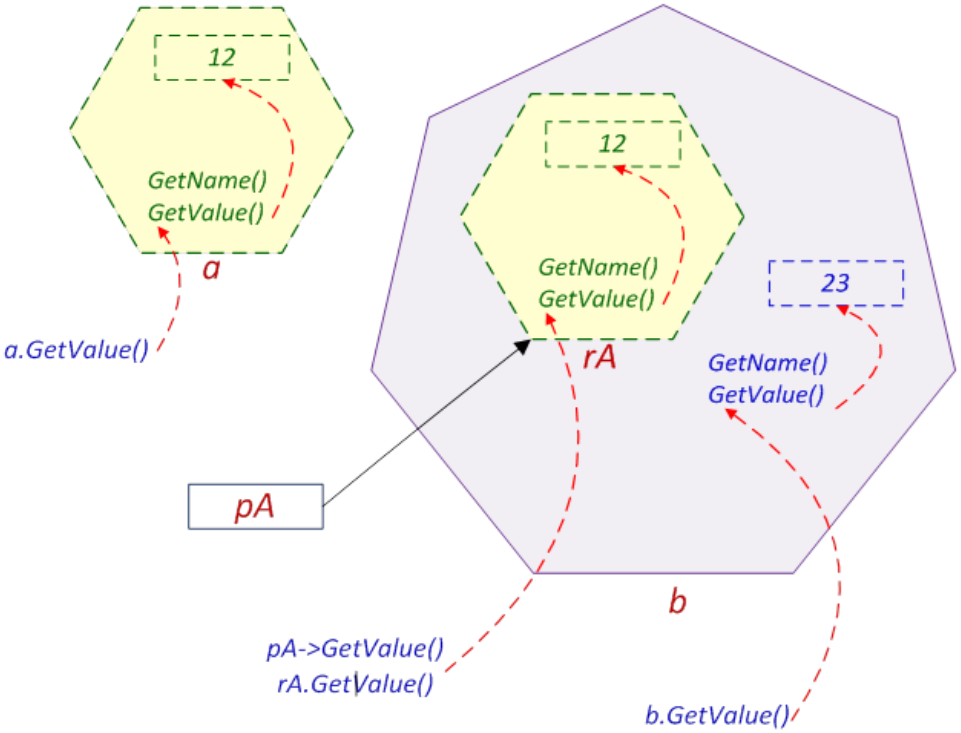
- Virtual member functions
- Redefining virtual functions
- Early and late binding
- Virtual destructors
- Abstract classes and interfaces
- Run-time type informations in case of classes

Polymorphism

❖ Early and late binding

▪ Static early binding

- During early binding, compilers integrate statically direct member function calls into the code.



Early binding example

Polymorphism

❖ Early and late binding

▪ Static early binding

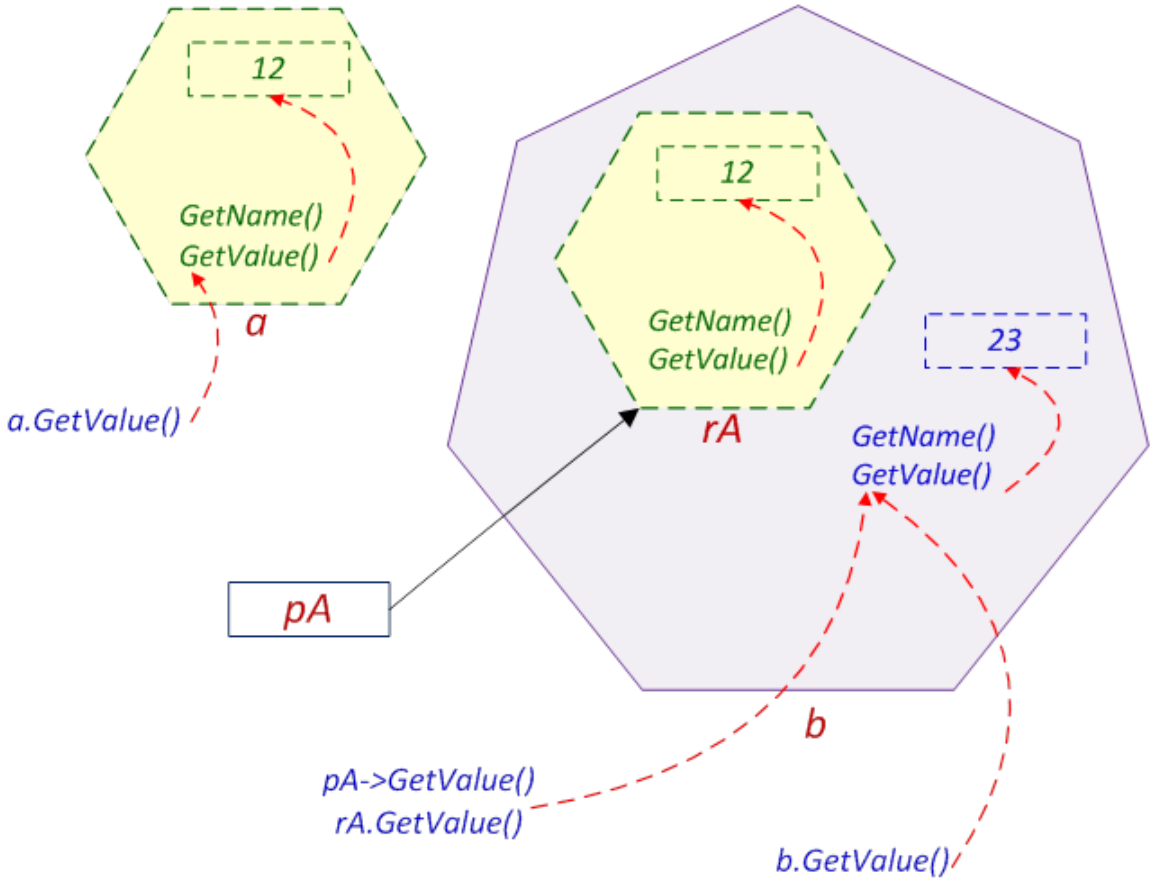
```
class Base {
 protected:
     int value;
 public:
     Base(int a=0) : value(a) { }
     const char* GetName() const { return "Base"; }
     int GetValue() const { return value; }
 };

class Derived: public Base {
 protected:
     int value;
 public:
     Derived(int a=0, int b=0) : Base(a), value(b) { }
     const char* GetName() const { return "Derived"; }
     int GetValue() const { return value; }
 };
```

```
int main() {
  Base a;
  Derived b(12, 23);

  a = b;
  Base &rA = b;
  Base *pA = &b;

  cout<<"a \t" <<  a.GetName()<<"\t"<<  a.GetValue()<<endl;
  cout<<"a \t" <<  b.GetName()<<"\t"<<  b.GetValue()<<endl;
  cout<<"rA\t" << rA.GetName()<<"\t"<< rA.GetValue()<<endl;
  cout<<"pA\t" << pA->GetName()<<"\t"<< pA->GetValue()<<endl;
}
```

```
a       Base    12
b       Derived 23
rA      base    12
pA      Base    12
```

Polymorphism

- **Early and late binding**
  - **Dynamic late binding**



Late binding example

Polymorphism

- **Early and late binding**
- **Dynamic late binding**

```
class Base {
 protected:
     int value;
 public:
     Base(int a=0) : value(a) { }
     virtual const char* GetName() const { return "Base"; }
     virtual int GetValue() const { return value; }
};
```

```
a       Base      12
b       Derived   23
rA      Derived   23
pA      Derived   23
```

Polymorphism

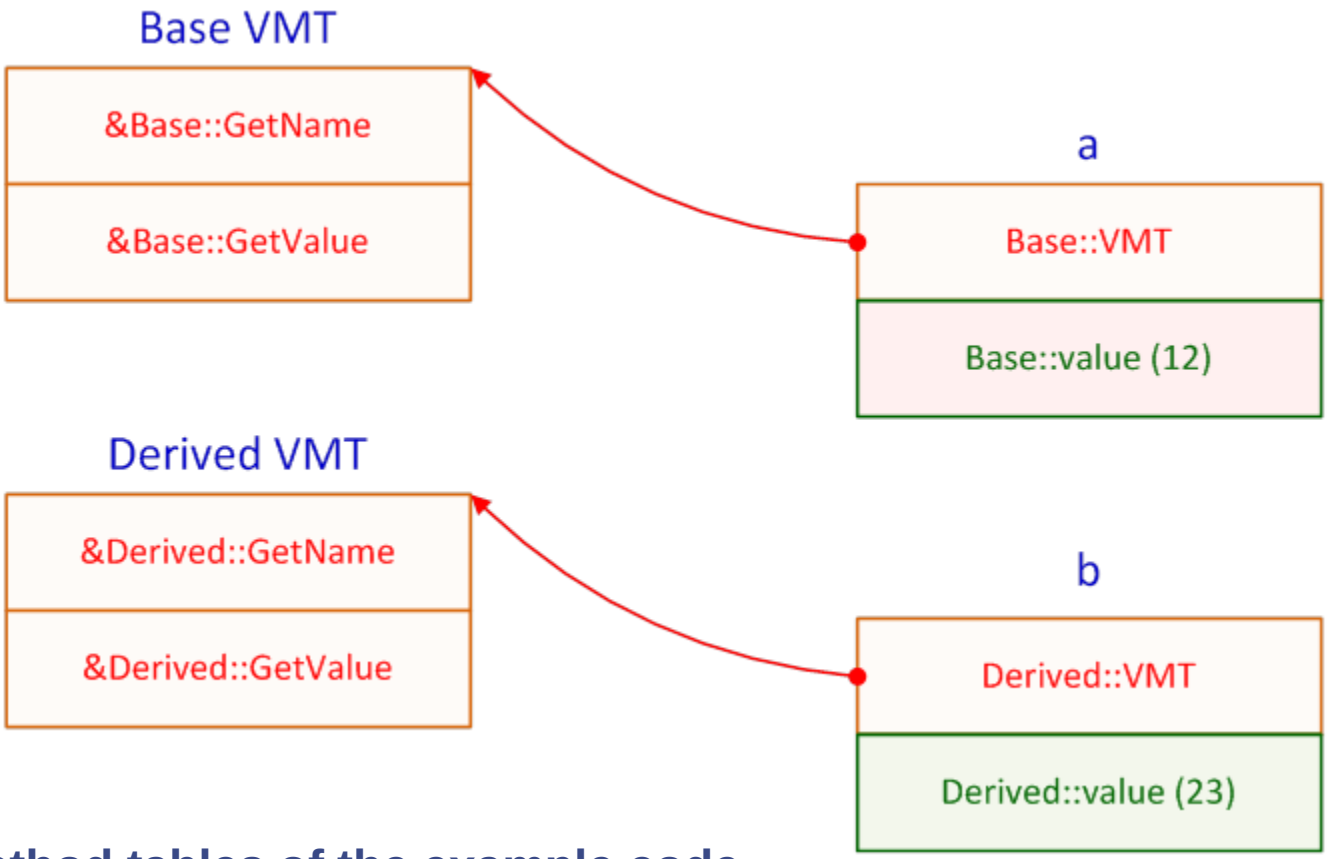
❖ Early and late binding

▪ Virtual method table

- In case a class has one or more virtual member functions, compilers complete the object with a "virtual pointer" to the global data table called virtual method table (V*MT*) or virtual function table (*VFTable*).
- VMT contains function pointers to the virtual member functions redefined the last of the given class and the base classes.
- VMTs for classes are created at run-time when the first constructor is called.

Polymorphism

- **Early and late binding**
- **Virtual method table**



Virtual method tables of the example code

Polymorphism

- Virtual member functions
- Redefining virtual functions
- Early and late binding
- Virtual destructors
- Abstract classes and interfaces
- Run-time type informations in case of classes

Polymorphism

❖ Virtual destructors

```
class Base {
 protected:
     int value;
 public:
     Base(int a=0) : value(a) { }
     virtual const char* GetName() const { return "Base"; }
     virtual int GetValue() const { return value; }
     virtual ~Base() {}
};
```

Polymorphism

- Virtual member functions
- Redefining virtual functions
- Early and late binding
- Virtual destructors
- Abstract classes and interfaces
- Run-time type informations in case of classes

Polymorphism

❖ Abstract classes and interfaces

```
// the geometrical Point class
class Point {
   protected:
      int x, y;
   public:
      Point(int a = 0, int b = 0) : x(a), y(b) {}
      Point(const Point& p) : x(p.x), y(p.y) {}
      int GetX() const { return x; }
      int GetY() const { return y; }
      void SetX(int a) { x = a; }
      void SetY(int a) { y = a; }
      void PrintOut() const { cout<<'('<<x<<','<<y<<')'<< endl; }
};

// an abstract class needed for moving - interface
class IMove {
   public:
      virtual void Move(int a, int b) = 0;
      virtual void Move(const Point& p) = 0;
};
```

Polymorphism

❖ Abstract classes and interfaces

```
// a Point that is able to move
class MovingPoint: public Point, public IMove {
    public:
        MovingPoint(int a=0, int b=0) : Point(a,b) {}
        void Move(int a, int b) { x = a; y = b; }
        void Move(const Point& p) {
            x = p.GetX();
            y = p.GetY();
        }
};

int main() {
    Point fixPoint(12, 23);
    fixPoint.PrintOut();          // (12, 23)

    MovingPoint movingPoint;
    movingPoint.PrintOut();       // (0, 0)
    movingPoint.Move(fixPoint);
    movingPoint.PrintOut();       // (12, 23)
}
```

Polymorphism

- Virtual member functions
- Redefining virtual functions
- Early and late binding
- Virtual destructors
- Abstract classes and interfaces
- Run-time type informations in case of classes

Polymorphism

❖ Run-time type informations in case of classes

```
#include <typeinfo>
#include <iostream>
using namespace std;

class Ancestor {
   public:
      virtual void Vf(){} // without this, RTTI is not stored
      void FunctAncestor() {cout<<"Ancestor"<<endl;}
  };

class Descendant : public Ancestor {
   public:
      void FunctDescendant() {cout<<"Descendant"<<endl;}
 };
```

Polymorphism

❖ Run-time type informations in case of classes

```
int main() {
   Descendant * pDescendant = new Descendant;
   Ancestor * pAncestor = pDescendant;

    // the type_info of the pointer:
   const type_info& tiAncestor = typeid(pAncestor);
   cout<< tiAncestor.name() <<endl;

   // the type_info of the Descendant:
   const type_info& tiDescendant = typeid(*pAncestor);
   cout<< tiDescendant.name() <<endl;

   // points to the Descendant?
   if (typeid(*pAncestor) == typeid(Descendant))
      dynamic_cast<Descendant *>(pAncestor)
                                    ->FunctDescendant();

   // points to the Descendant?
   if (dynamic_cast<Descendant*>(pAncestor))
      dynamic_cast<Descendant*>(pAncestor)
                                    ->FunctDescendant();
   delete pDescendant;
}
```

Polymorphism

❖ Run-time type informations in case of classes

```
#include <iostream>
#include <string>
#include <typeinfo>
using namespace std;

class Animal {
  protected:
     int legs;
  public:
     virtual const string Species() = 0;
     Animal(int n) {legs=n;}
     void Info() {
       cout<<"A(n) "<<Species()<<" has "
          <<legs<<" leg(s)."<<endl;
      }
};
```

```
class Fish : public Animal {
  protected:
     const string Species() {return "fish";}
  public:
     Fish(int n=0) : Animal(n) {}
     void Swims() {cout<<"swims"<<endl;}
};

class Bird : public Animal {
  protected:
     const string Species() {return "bird";}
  public:
     Bird(int n=0) : Animal(n) {}
     void Flies() {cout<<"flies"<<endl;}
};

class Mammal : public Animal {
  protected:
     const string Species() {return "mammal";}
  public:
     Mammal(int n=4) : Animal(n) {}
     void Runs() {cout<<"runs"<<endl;}
};
```

Polymorphism

❖ Run-time type informations in case of classes

```
int main() {
    const int db=3;
    Animal* p[db] = {new Bird, new Fish, new Mammal};

    // accessing data without RTTI
    for (int i=0; i<db; i++)
      p[i]->Info();

    // processing with the help of RTTI
    for (int i=0; i<db; i++)
       if (dynamic_cast<Fish*>(p[i]))          // Fish?
             dynamic_cast<Fish*>(p[i])->Swims();
       else
       if (typeid(*p[i])==typeid(Bird))              // Bird?
             dynamic_cast<Bird*>(p[i])->Flies();
       else
       if (typeid(*p[i])==typeid(Mammal))       // Mammal?
             dynamic_cast<Mammal*>(p[i])->Runs();

    for (int i=0; i<db; i++)
        delete p[i];
}
```

Polymorphism

❖ Run-time type informations in case of classes

- The version of the code above that does not use run-time type information

```
int main() {
    const int db=3;
    Animal* p[db] = {new Bird, new Fish, new Mammal};

    for (int i=0; i<db; i++)
      p[i]->Info();

    for (int i=0; i<db; i++)
       if (p[i]->Species()=="fish")
            static_cast<Fish*>(p[i])->Swims();
       else
       if (p[i]->Species()=="bird")
            static_cast<Bird*>(p[i])->Flies();
       else
       if (p[i]->Species()=="mammal")
            static_cast<Mammal*>(p[i])->Runs();

    for (int i=0; i<db; i++)
        delete p[i];
}
```

Polymorphism

❖ Run-time type informations in case of classes

- Result

```
A(n) bird has two leg(s).
A(n) fish has 0 leg(s).
A(n) mammal has 4 leg(s).
flies
swims
runs
```

Chapter III. Object-oriented programming in C++

- Introduction to the object-oriented world
- Classes and objects
- Inheritance (derivation)
- Polymorphism
- Class templates

Class templates

- A step-be-step tutorial for creating and using class templates
- Defining a generic class
- Instantiation and specialisation
- Value parameters and default template parameters
- The "friends" and static data members of a class template
- The Standard Template Library (STL) of C++

Class templates

❖ A step-be-step tutorial for creating and using class templates

- A simplified class handling an one-dimensional integer array of 32 elements with bound index checking

```
#include <iostream>
#include <cassert>
#include <cstring>
using namespace std;

class IntArray {
  public:
     IntArray(bool initialise = true) : size(32) {
        if (initialise) memset(storage, 0, 32*sizeof(int));
     }
     int& operator [](int index);
     const int size;
  private:
     int storage[32];
};

int & IntArray::operator [](int index) {
  if (index<0 || index>=32) assert(0);      // index error
  return storage[index];                    // success
}

int main() {
    IntArray a;
    a[ 7] = 12;
    a[29] = 23;
    for (int i=0; i<a.size; i++)
        cout<<a[i]<<'\t';
}
```

Class templates

❖ **A step-be-step tutorial for creating and using class templates**

- A class template

```
#include <iostream>
#include <cassert>
#include <cstring>
using namespace std;

template <class type, int numberOfElements>
class Array {
  public:
     Array(bool initialise=true): size(numberOfElements) {
        if (initialise)
         memset(storage,0,numberOfElements*sizeof(type)); }
     type& operator [](int index);
     const int size;
  private:
     type storage[numberOfElements];
};
```

Class templates

❖ A step-be-step tutorial for creating and using class templates

- In the case of external member functions, the name of the class has to be used together with the generic type and the parameter *Array<type, numberOfElements>*:

```
template <class type, int numberOfElements>
type & Array<type, numberOfElements>::operator [](int index) {
  if (index<0 || index>=numberOfElements)
     assert(0);                    // index error
  return storage[index];     // success
}
```

Class templates

❖ A step-be-step tutorial for creating and using class templates

- let's see how a class template can help us

```
int main() {
    Array<int, 32> a;
    a[ 7] = 12;
    a[29] = 23;
    for (int i=0; i<a.size; i++)
        cout<<a[i]<<'\t';
}
```

- The template created from that can be used also for storing character sequences and objects.

```
const int asize=8;
Array<char *, asize> s1;
s1[2] = (char*)"C++";
s1[4] = (char*)"java";
s1[7] = (char*)"C#";
for (int i=0; i<s1.size; i++)
    if (s1[i]) cout<<s1[i]<<'\t';
```

```
const int asize=8;
Array<string, asize> s2 (false);
s2[2] = "C++";
s2[4] = "java";
s2[7] = "C#";
for (int i=0; i<s2.size; i++)
    cout<<s2[i]<<'\t';
```

Class templates

❖ A step-be-step tutorial for creating and using class templates

•

```
Array<double, 3> *dt;
dt = new Array<double, 3>;
(*dt)[0] =12.23;
 dt[0][1]=34.45;
 for (int i=0; i<dt->size; i++)
      cout<<(*dt)[i]<<'\t';
 delete dt;
```

```
Array<double, 3> *dm;
dm = new Array<double, 3> [5];
dm[0][1] =12.23;
dm[4][2]=34.45;
for (int i=0; i<5; i++) {
  for (int j=0; j<dm[0].size; j++)
      cout<<dm[i][j]<<'\t';
  cout<<endl;
}
delete []dm;
```

```
Array<int, 3> m[5];
m[0][1] = 12;
m[4][2] = 23;
for (int i=0; i<5; i++) {
  for (int j=0; j<m[0].size; j++)
      cout<<m[i][j]<<'\t';
  cout<<endl;
}
```

```
Array< Array<int,3>, 5> p(false);
p[0][1] = 12;
p[4][2] = 23;
for (int i=0; i<p.size; i++) {
  for (int j=0; j<p[0].size; j++)
      cout<<p[i][j]<<'\t';
  cout<<endl;
}
```

Class templates

- A step-be-step tutorial for creating and using class templates
- Defining a generic class
- Instantiation and specialisation
- Value parameters and default template parameters
- The "friends" and static data members of a class template
- The Standard Template Library (STL) of C++

Class templates

❖ Defining a generic class

- A parametrized or *generic class* makes it possible for us to use the parametrized class as a template to create new classes.

➢ So a given class definition can be used for all types.

```
template <class type1, … class typeN>
class Classname {
    …
};

//or

template <typename type1, … typename typeN>
class Classname {
    …
};
```

Class templates

❖ Defining a generic class

- The non-**inline** member functions of the class should be defined as follows:

```
template <class type1, … class typeN >
functtype Classname< type1, … typeN> ::
                                    Functname(parameter_list) {
    …
}

//or

template <typename type1, … typename typeN >
functtype Classname< type1, … typeN> ::
                                    Functname(parameter_list) {
    …
}
```

Class templates

❖ Defining a generic class

```
template <typename type>
class Point {
   protected:
      type x, y;
   public:
      Point(type a = 0, type b = 0) : x(a), y(b) {}
      Point(const Point& p) : x(p.x), y(p.y) {}
      type GetX() const { return x; }
      type GetY() const { return y; }
      void SetX(type a) { x = a; }
      void SetY(type a) { y = a; }
      void PrintOut() const { cout<<'('<<x<<','<<y<<')'<< endl; }
};
```

Class templates

❖ Defining a generic class

- The same *Point* class becomes much more complicated if one part of its member functions is defined outside the class:

```
template <typename type>
class Point {
   protected:
      type x, y;
   public:
      Point(type a = 0, type b = 0) : x(a), y(b) {}
      Point(const Point& p) : x(p.x), y(p.y) {}
      type GetX() const;
      type GetY() const { return y; }
      void SetX(type a);
      void SetY(type a) { y = a; }
      void PrintOut() const;
};
template <typename type>
type Point<type>::GetX() const { return x; }

template <typename type>
void Point<type>::SetX(type a) { x = a; }

template <typename type>
void Point<type>::PrintOut() const {
     cout<<'('<<x<<','<<y<<')'<< endl;
}
```

Class templates

- A step-be-step tutorial for creating and using class templates
- Defining a generic class
- Instantiation and specialisation
- Value parameters and default template parameters
- The "friends" and static data members of a class template
- The Standard Template Library (STL) of C++

Class templates

❖ Instantiation and specialisation

- A template can be defined in many ways.

➢ In **implicit instantiation**, type parameters are replaced by concrete types. First the version of the given type of a class is created (if it has not yet been created), then the object instance:

```
Point<double> p1(1.2, 2.3), p2(p1);
Point<int> *pp;  // the class Point<int> is not created
```

Class templates

❖ Instantiation and specialisation

- In **explicit instantiation**, the compiler is asked to create an instance of the class by using the given types, so when the object is being created, the class is ready to be used:

```
template class Point<double>;
…
Point<double> p1(1.2, 2.3), p2(p1);
```

Class templates

❖ Instantiation and specialisation

- Among the following declarations:

➢ the first one is a general template,

➢ the second one is a version tailored to pointers,

➢ the third is a version specialised to **void*** pointers.

```
template <class type> class Point {
  // the class template above
};

template <class type> class Point <type *> {
  // has to be created
};

template <> class Point <void *> {
  // has to be created
};
```

```
Point<double> pa;
Point<int *> pp;
Point<void *> pv;
```

Class templates

❖ Instantiation and specialisation

```
template <typename T1, typename T2>
class DataStream {
   public:
      DataStream() { cout << "DataStream<T1,T2>"<<endl;}
      // …
};

template < typename T1, typename T2>     // Specialisation
class DataStream<T1*, T2*> {
   public:
      DataStream() { cout << "DataStream<T1*,T2*>"<<endl;}
      // …
};

template < typename T1>              // Partial specialisation
class DataStream<T1, int> {
   public:
      DataStream() { cout << "DataStream<T1, int>"<<endl;}
      // …
};
```

Class templates

❖ Instantiation and specialisation

```
template <>                          // Complete specialisation
class DataStream<char, int> {
   public:
      DataStream() { cout << "DataStream<char, int>"<<endl;}
      // …
} ;

int main() {
   DataStream<char, int> s4 ;          // Complete specialisation
   DataStream<int, double> s1 ;
   DataStream<double*, int*> s2;      // Specialisation
   DataStream<double, int> s3 ;      // Partial specialisation
}
```

Class templates

- A step-be-step tutorial for creating and using class templates
- Defining a generic class
- Instantiation and specialisation
- Value parameters and default template parameters
- The "friends" and static data members of a class template
- The Standard Template Library (STL) of C++

Class templates

❖ Value parameters and default template parameters

- C++ supports **default template parameters**.

```
#include <iostream>
#include <cassert>
#include <cstring>
using namespace std;

template <typename type=int, int numberOfElements=32>
class Array {
  public:
    Array(bool initialise=true): size(numberOfElements) {
        if (initialise)
          memset(storage, 0, numberOfElements*sizeof(type));
    }
    type& operator [](int index) {
        if (index<0 || index>=numberOfElements) assert(0);
        return storage[index];
     }
     const int size;
  private:
     type storage[numberOfElements];
};
```

Class templates

❖ Value parameters and default template parameters

- Example

```
#include <iostream>
#include <string>
using namespace std;

template<typename Type=int, int MaxSize=100>
class Stack {
   Type array[MaxSize];
   int sp;
  public:
   Stack(void) { sp = 0; };
   void Push(Type data) {
      if (sp < MaxSize) array[sp++] = data;
   }
   Type Pop(void) {
      return array[sp > 0 ? --sp : sp];
   }
   bool isEmpty(void) const { return sp == 0; };
};
```

Class templates

❖ Value parameters and default template parameters

- Example

```
int main(void) {
  Stack<double,1000> dStack; // stack of 1000 double elements
  Stack<string> sStack;      // stack of 100 string elements
  Stack<> iStack;            // stack of 100 int elements

  int a=102, b=729;
  iStack.Push(a);
  iStack.Push(b);
  a=iStack.Pop();
  b=iStack.Pop();

  sStack.Push("language");
  sStack.Push("C++");
  do {
     cout << sStack.Pop()<<endl;;
  } while (!sStack.isEmpty());
}
```

Class templates

- A step-be-step tutorial for creating and using class templates
- Defining a generic class
- Instantiation and specialisation
- Value parameters and default template parameters
- The "friends" and static data members of a class template
- The Standard Template Library (STL) of C++

Class templates

❖ The "friends" and static data members of a class template

- A class template may also have **friends**, which may behave differently.

```
#include <iostream>
using namespace std;

class ClassA {
   void Operation() { cout << "Operation carried out."<< endl; };
   template<typename T> friend class BClass;
};

template<class T> class BClass {
   public:
      void Execute(ClassA& a) { a.Operation(); }
};

int main() {
   BClass<int> b;
   BClass<double> c;
   ClassA a;
   b.Execute(a);
   c.Execute(a);
}
```

Class templates

❖ The "friends" and static data members of a class template

```
#include <iostream>
using namespace std;

// Forward declarations
template <typename T> class Class;
template <typename T> void Funct(Class<T>&);

template <typename T> class Class {
    friend void Funct<T>(Class<T>&);
  public:
    T GetData(){return data;}
    void SetData(T a){data=a;}
  private:
    T data;
};
```

```
template<typename T> void Funct(Class<T>& x) {
    cout<<"Result: "<<x.GetData()<<endl;
}

int main() {
    Class<int> obj1;
    obj1.SetData(7);
    Funct(obj1);
    Class<double> obj2;
    obj2.SetData(7.29);
    Funct(obj2);
}
```

Class templates

❖ The "friends" and static data members of a class template

```
#include <iostream>
using namespace std;

template<typename type> class Class {
  public:
    static int ID;
    static type data;
    Class() {}
};

// Definitions of static data members
template <typename type> int Class<type>::ID = 23;
template <typename type> type Class<type>::data = 12.34;

int main() {
    Class <double> dObj1, dObj2;

    cout << dObj1.ID++   << endl;          // 23
    cout << dObj1.data-- << endl;          // 12.34

    cout << dObj2.ID   << endl;            // 24
    cout << dObj2.data << endl;            // 11.34

    cout <<Class<double>::ID << endl;      // 24
    cout <<Class<double>::data << endl;    // 11.34
}
```

Class templates

- A step-be-step tutorial for creating and using class templates
- Defining a generic class
- Instantiation and specialisation
- Value parameters and default template parameters
- The "friends" and static data members of a class template
- The Standard Template Library (STL) of C++

Class templates

❖ The Standard Template Library (STL) of C++

▪ The structure of STL

The elements of the Library can be grouped into five groups:

- containers – data structures making it possible to store data in memory (*vector*, *list*, *map*, *set*, *deque*, …)
- adaptors – higher-level data structures based on containers (*stack*, *queue*, *priority_queue*)
- algorithms - operations that can be carried out on data stored in containers (*sort*, *copy*, *search*, *min*, *max*, …)

Class templates

❖ The Standard Template Library (STL) of C++

▪ The structure of STL

The elements of the Library can be grouped into five groups:

- iterators – generic pointers that ensure access to the data stored in containers (*iterator*, *const_iterator*, *ostream_iterator<>*, …)
- function objects – functions are covered by classes, for other components (*divides*, *greater_equal*, *logical_and*, …).

Class templates

❖ The Standard Template Library (STL) of C++

▪ The structure of STL

Short description	Header file
Managing, sorting data in containers and searching in them	*<algorithm>*
Associative container for storing bits: bitset	*<bitset>*
Associative containers that store elements: *multiset* (may have the same elements more times), and *set* (stores only unique elements)	*<set>*
Associative container storing key/value pairs in a 1:1 relation (*map*), or in a 1:n relation (*multiset*)	*<map>*
Predefined iterators, datastream iterators	*<iterator>*
Container: dynamic array	*<vector>*
Container: double ended queue	*<deque>*
Container: linear list	*<list>*
Container adaptor: queue	*<queue>*
Container adaptor: stack	*<stack>*

Class templates

❖ The Standard Template Library (STL) of C++

▪ STL and C++ arrays

```
#include <iostream>
#include <iterator>
#include <algorithm>
using namespace std;

void PrintOut(const int x[], int n) {
    static ostream_iterator<int> out(cout,"\t");
    cout<< "\t";
    copy(x, x+n, out);
    cout<<endl;
}

void IntPrintOut(int a) {
    cout << "\t" << a << endl;
}
```

```
int main() {
    const int db = 7;
    int data[db]={2, 7, 10, 12, 23, 29, 80};

    cout << "Original array: " << endl;
    PrintOut(data, db);

    cout << "Next permutation: " << endl;
    next_permutation(data,data+db);
    PrintOut(data, db);

    cout << "In the reverse order: " << endl;
    reverse(data,data+db);
    PrintOut(data, db);

    cout << "Random shuffle: " << endl;
    for (int i=0; i<db; i++) {
       random_shuffle(data,data+db);
       PrintOut(data, db);
    }

    cout << "The greatest element: ";
    cout << *max_element(data,data+db) << endl;

    cout << "Finding an element:";
    int *p=find(data,data+db, 7);
    if (p != data+db)
      cout << "\tfound" <<endl;
    else
      cout << "\tnot found" <<endl;
```

Class templates

❖ The Standard Template Library (STL) of C++

- **STL and C++ arrays**

```
    cout << "Sort: " << endl;
    sort(data,data+db);
    PrintOut(data, db);

    cout << "Printing out each element in a new line:"
                                                << endl;
    for_each(data, data+db, IntPrintOut);
    PrintOut(data, db);

    cout << "Swap: " << endl;
    swap(data[2],data[4]);
    PrintOut(data, db);

    cout << "Filling the container: " << endl;
    fill(data,data+db, 123);
    PrintOut(data, db);
}
```

Class templates

❖ The Standard Template Library (STL) of C++

▪ Using STL containers

```
#include <vector>
#include <algorithm>
#include <iterator>
#include <iostream>
using namespace std;

double Sum(const vector<double>& dv) {
    vector<double>::const_iterator p; // constant iterator
    double s = 0;
    for (p = dv.begin(); p != dv.end(); p++)
        s += *p;
    return s;
}

bool isOdd (int n) {
    return (n % 2) == 1;
}

int main() {
    // output operator
    ostream_iterator<double>out(cout, " ");
    double data[]  = {1.2, 2.3, 3.4, 4.5, 5.6};

    // Initialising the vector with the elements of the array
    vector<double> v(data, data+5);

    // Printing out the vector
    copy(v.begin(), v.end(), out);  cout << endl;
    cout<<"Sum of the elements: "<<Sum(v)<<endl;

    // Adding elements to the vector
    for (int i=1; i<=5; i++)
        v.push_back(i-i/10.0);
    copy(v.begin(), v.end(), out);  cout << endl;

    // Adding 4.5 to all elements
    for (int i=0; i<v.size(); i++)
        v[i] += 4.5;
    copy(v.begin(), v.end(), out);  cout << endl;

    // Converting all elements to an integer
    vector<double>::iterator p;
    for (p=v.begin(); p!=v.end(); p++)
        *p = int(*p);
    copy(v.begin(), v.end(), out);  cout << endl;

    // Deleting each second element
    int index = v.size()-1;
    for (p=v.end(); p!=v.begin(); p--)
        if (index-- % 2 ==0)
            v.erase(p);
    copy(v.begin(), v.end(), out);  cout << endl;

    // Sorting the elements of the vector
    sort(v.begin(), v.end() );
    copy(v.begin(), v.end(), out);  cout << endl;

    // Searching for 7 in the vector
    p = find(v.begin(), v.end(), 7);
    if (p != v.end() )
        cout << "found"<< endl;
    else
        cout << "not found"<< endl;

    // The number of odd elements
    cout<< count_if(v.begin(), v.end(), isOdd)<< endl;
}
```

Class templates

❖ The Standard Template Library (STL) of C++

▪ Using STL container adaptors

- The adapted *stack* functions are summarised in the following table:

void push(const value_type& a)	inserting in the stack,
void pop()	removing the top element of the stack,
value_type& top()	accessing the top element of the stack,
const value_type& top() const	accessing the top element of the stack,
bool empty() const	returns true, if the stack is empty,
size_type size()const	the number of elements in the stack,
operator== and operator<	the operations "equals" and "smaller than".

Class templates

❖ The Standard Template Library (STL) of C++

▪ Using STL container adaptors

```
#include <iostream>
#include <stack>
#include <vector>
using namespace std;

int main() {
   int number=2013, base=16;
   stack<int, vector<int> > istack;

   do {
     istack.push(number % base);
     number /= base;
   } while (number>0);

   while (!istack.empty()) {
      number = istack.top();
      istack.pop();
      cout<<(number<10 ? char(number+'0'):
                         char(number+'A'-10));
   }
}
```

TRƯỜNG ĐẠI HỌC BÁCH KHOA HÀ NỘI

KỸ THUẬT LẬP TRÌNH HỆ CƠ ĐIỆN TỬ

Programming Engineering in Mechatronics

Giảng viên: TS. Nguyễn Thành Hùng

Đơn vị: Bộ môn Cơ điện tử, Viện Cơ khí

Hà Nội, 2018

Chapter IV. Graphical User Interface in C++/CLI

- 1. Specialties of CLI, standard C++ and C++/CLI
- 2. The window model and the basic controls
- 3. Text and binary files, data streams
- 4. The GDI+

Specialties of CLI, standard C++ and C++/CLI

There are several ways to develop applications for a computer running the Windows operating system:

- We implement the application with the help of a development kit and it will operate within this run-time environment. The file cannot be run directly by the operating system (e.g. MatLab, LabView) because it contains commands for the run-time environment and not for the CPU of the computer. Sometimes there is a pure run-time environment also available beside the development kit for the use of the application developed, or an executable (exe) file is created from our program, which includes the run-time needed for running the program.
- The development kit prepares a stand-alone executable application file (exe), which contains the commands written in machine code runnable on the given operating system and processor (native code). This file is run while developing and testing the program. Such tools are e.g. Borland Delphi and Microsoft Visual Studio, frequently used in industry.

Specialties of CLI, standard C++ and C++/CLI

1.1. Compiling and running native code under Windows

1.2. Problems during developing and using programs in native code

1.3. Platform independence

1.4. Running MSIL code

1.5. Integrated development environment

1.6. Controllers, visual programming

1.7. The .NET framework

1.8. C#

1.9. Extension of C++ to CLI

1.10. Extended data types of C++/CLI

Specialties of CLI, standard C++ and C++/CLI

1.11. The predefined reference class: String

1.12. The System::Convert static class

1.13. The reference class of the array implemented with the CLI array template

1.14. C++/CLI: Practical realization in e.g. in the Visual Studio 2008

1.15. The Intellisense embedded help

1.16. Setting the type of a CLR program.

Specialties of CLI, standard C++ and C++/CLI

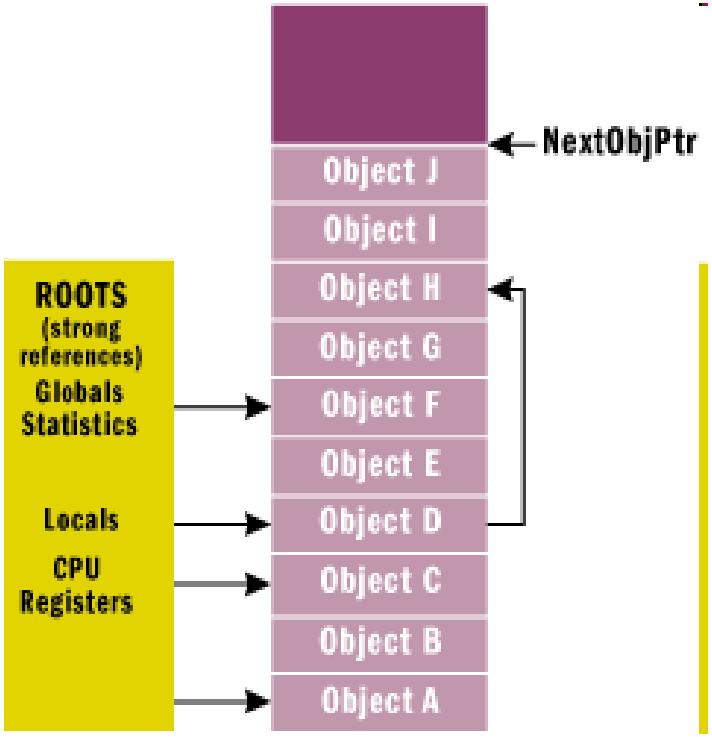
❖ 1.1. Compiling and running native code under Windows

The process of compliation is the following:

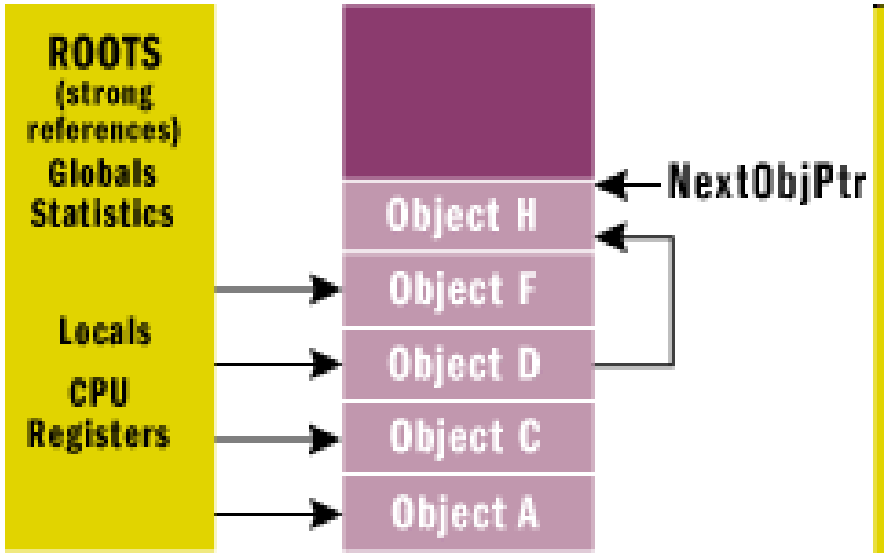
- C++ sources are stored in files with the extension .cpp, headers in files with the extension .h. There can be more than one of them, if the program parts that logically belong together are placed separately in files, or the program has been developed by more than one person.
- Preprocessor: resolving *#define* macros, inserting *#include* files into the source.
- Preprocessed C source: it contains all the necessary function definitions.
- C compiler: it creates an .OBJ object file from the preprocessed sources.
- OBJ files: they contain machine code parts (making their names public – export) and external references to parts in other files.
- Linker: after having resolved references in OBJ files and files with the extension .LIB that contain precompiled functions (e.g. printf()), having cleaned the unnecessary functions and having specified the entry point (function main()), the runnable file with the extension .EXE is created, which contains the statements in machine code runnable on the given processor.

Specialties of CLI, standard C++ and C++/CLI

❖ 1.2. Problems during developing and using programs in native code



The memory before cleaning



The memory after cleaning

Specialties of CLI, standard C++ and C++/CLI

❖ 1.3. Platform independence

- CPUs made by many manufacturers (Intel, ARM, MIPS, etc.)
- The 32 bits and 64 bits operating system
- Operating system: Windows (XP, Vista, 7, 8, 10), Linux, Unix, Mac OS, Android, …

❖ 1.4. Running MSIL code

The CIL (Common Intermediate Language) code is transformed into a file with .EXE extension, where it is runable. But this code is not the native code of the processor, so the operating system must recognize that one more step is necessary. This step can be done in two ways, according to the principles used in Java system:

- interpreting and running the statements one by one. This method is called **JIT** (Just In Time) execution. Its use is recommended for the step by step running of the source code and for **debug** including break points.
- generating native code from all statements at the same time and starting it. This method is called AOT (Ahead of Time), and it can be created by the Native Image Generator (NGEN). We use it in the case of well functioning, tested, ready programs (**release**).

Specialties of CLI, standard C++ and C++/CLI

❖ 1.5. Integrated development environment

- The integrated development environment (**IDE**) includes a text editor, a compiler and a runner in one program.

Specialties of CLI, standard C++ and C++/CLI

❖ 1.6. Controllers, visual programming

Applications that run on operating systems with a graphical user interface (GUI) consist of two parts at least:

- The code part that contains the algorithm of the program
- The interface that implements the user interface (**UI**)

The two parts are logically linked: events (**event**) happening in the user interface trigger the run of the defined subprograms of the algorithm part (these subprograms are called **function**s in C type languages). Hence these functions are called "**event handler functions**"

Specialties of CLI, standard C++ and C++/CLI

❖ 1.7. The .NET framework

Parts of the framework:

- *Common Language Infrastructure* (CLI), and its realization the *Common Language Runtime* (CLR): the common language compiler and run-time environment.
- *Base Class Library*: the library of the basic classes.
- WinForms: controls preprepared for the Windows applications, inherited from the Base Class Library.
- Additional parts: these could be the ASP.NET system that supports application development on the web, the ADO.NET that allows access to databases and Task Parallel Library that supports multiprocessor systems.

Specialties of CLI, standard C++ and C++/CLI

❖ 1.8. C#

- The .NET framework and the pure managed code can be programmed with C# easily.
- It is recommended to amateurs and students in higher education (not for programmers – their universal tools are the languages K&R C and C++).
- The .NET framework contains a command line C# compiler and we can also download freely the Visual C# Express Edition from Microsoft.
- Their goal with this is to spread C# (and .NET).

Specialties of CLI, standard C++ and C++/CLI

❖ 1.9. Extension of C++ to CLI

- The C++ compiler developed by Microsoft can be considered as a standard C++ as long as it is used to compile a native win32 application.
- However, in order to reach CLI new data types and operations were needed.
- The defined language cannot be considered as C++ because the statements and data types of MC do not fit in C++ standard definition.
- The language was called **C++/CLI** and it was standardized (ECMA-372).

Specialties of CLI, standard C++ and C++/CLI

❖ 1.10. Extended data types of C++/CLI

- In C++ the class on the managed heap is called **reference class** (**ref class**).

The reference class behaves differently compared to the C++ class:

- Static samples do not exist, only dynamic ones
- It is not pointer that points to it but **handle** (handler) and its sign is **^**. Handle has pointer like features, for instance the sign of a reference to a member function is **->**. Correct declaration is *String ^text;* in this case the text does not have any content yet given that its default constructor creates an empty, string with length of 0 (“”).
- When creating we do not use the **new** operator but the **gcnew.** An example: *text=gcnew String("");* creation of a string with length of 0 with a constructor. Here we do not have to use the ^ sign, its usage would be wrong.
- Its deletion is not handled by using the **delete** operator but by giving a value of handle **nullptr**. After a while the garbage collector will free up the used space automatically. An example: *text=nullptr;* delete can be used as well, it will call the destructor but the object will stay in the memory.

Specialties of CLI, standard C++ and C++/CLI

❖ 1.10. Extended data types of C++/CLI

The reference class behaves differently compared to the C++ class:

- It can be inherited only **publicly** and only from **one parent** (multiple inheritances are possible only with an interface class).
- There is the option to create an **interior pointer** to the reference class that is initiated by the garbage collector. This way, however, we loose the security advantages of the managed code (e.g preventing memory overrun).
- The reference class – similarly to the native one – can have data members, methods, constructors (with overloading). We can create properties (**property**) that contain the data in themselves (**trivial** property) or contain functions (scalar property) to reach the data after checking (e.g. the age cannot be set as to be a negative number). Property can be virtual as well or multidimensional, in the latest case it will have an index as well. Big advantage of property is that it does not have parenthesis, compared to a native C++ function that is used to reach member data. An example: *int length=text->Length*; the *Length* a read only property gives the number of the characters in the string.

Specialties of CLI, standard C++ and C++/CLI

❖ 1.10. Extended data types of C++/CLI

The reference class behaves differently compared to the C++ class:

- Beside the destructor that runs when deleting the class (and for this it can be called deterministic) can contain a **finalizer**() method which is called by the GC (*garbage collector*) when cleaning the object from the memory. We do not know when GC calls the finalizer that is why we can call it non-deterministic.
- The **abstract** and the **override** keywords must be specified in each case when the parent contains virtual method or property.
- All data and methods will be **private** if we do not specify any access modifier.
- If the virtual function does not have phrasing, it has to be declared as **abstract***: virtual type functionname() abstract;* or *virtual type functionname() =0;* (the =0 is the standard C++. the abstract is defined as =0). It is mandatory to override it in the child. If we do not want to override the (not purely) virtual method, then we can create a new one with the **new** keyword.

Specialties of CLI, standard C++ and C++/CLI

❖ 1.10. Extended data types of C++/CLI

The reference class behaves differently compared to the C++ class:

- It can be set at the reference class that no new class could be created from it with inheritance (with overriding the methods), and it could be only instantiated. In this case the class is defined as **sealed**. The compiler contains a lot of predefined classes that could not be modified e.g. the already mentioned String class.
- We can create an Interface class type for multiple inheritances. Instead of reference we can write an **interface** class/struct (their meaning is the same at the interface). The access to all the members of the interface (data members, methods, events, properties) is automatically public. Methods and properties cannot be expanded (mandatorily abstract), while data can only be static. Constructors cannot be defined either. The interface cannot be instantiated, only **ref/value class/struct** can be created from it with inheritance. Another interface can be inherited from an interface. A derived reference class (**ref class**) can have any interface as base class. The interface class is usually used on the top of the class hierarchy, for example the Object class that is inherited by almost all.

Specialties of CLI, standard C++ and C++/CLI

❖ 1.10. Extended data types of C++/CLI

The reference class behaves differently compared to the C++ class:

- We can use **value class** to store data. What refers to it is not a handle but it is a static class type (that is, a simple unspecified variable). It can be derived from an interface class (or it can be defined locally without inheritance).
- Beside function pointers we can define a **delegate** also to the methods of a (reference) class that appears as a procedure that can be used independently. This procedure is secured, and errors are not faced that cause a mix up of the types and is possible with pointers of a native code. Delegate is applied by the .NET system to set and call the event handler methods, that belong to the events of the controls.

Specialties of CLI, standard C++ and C++/CLI

❖ 1.10. Extended data types of C++/CLI

Operation	K&R C	C++	Managed C++ (VS 2002)	C++/CLI (VS 2005-)
Memory allocation for the object (dynamic variable)	malloc(…), calloc()	new …	_gc new ...	gcnew ...
Memory unallocation	free(...)	delete ...	Automatic, after ...=nullptr GC::Collect()	<- similarly as in 2002
Referring to an object	Pointer (*)	Pointer (*)	_nogc Pointer: to native data, _gc Pointer: to managed data	Pointer (*): to native data, Handle (^): to managed data

Specialties of CLI, standard C++ and C++/CLI

❖ 1.11. The predefined reference class: String

- The System::String class was created on the basis of C++ string type in order to store text.
- The text is stored with the series of Unicode characters (**wchar_t**).
- Its default constructor creates a 0 length (“”) text.
- Its other constructors allow that we create it from char*, native string, wchar_t* or from an array that consists of strings.
- String is a reference class, we create a handle (**^**) to it and we can reach its properties and methods with **->.** Properties and methods that are often used:
  - String->Length length. An example: *s=”ittykitty”; int i=s->Length;* after the value of *i* will be 9
  - String[ordinal number] character (0.. as by arrays). An example: value of *s[1]* will be the *‘t’* character.

Specialties of CLI, standard C++ and C++/CLI

❖ 1.11. The predefined reference class: String

- String is a reference class, we create a handle (**^**) to it and we can reach its properties and methods with **->.** Properties and methods that are often used:
  - String->Substring(from which ordinal number, how many) copying a part. An example: the value of *s->Substring(1,3)* will be *”tty”.*
  - String->Split(delimiter) : it separates the string with the delimiter to the array of words that are contained in it. An example*: s=”12;34”; t=s->Split(‘;’);* after t a 2 element array that contains strings (the string array has to be declared). The 0. its element is *“12”*, and the 1. its elements is *“34”.*
  - in what -> IndexOf(what) search. We get a number, the initiating position of the what parameter in the original string (starting with 0 as an array index). If the part was not found, it returns -1. Note that it will not be 0 because 0 is a valid character position. As an example: with the s is *“ittykitty”,* the value of *s->IndexOf(“ki”)* will be *4*, but the value *of s->IndexOf(“dog”)* will be *-1*.

❖ 1.11. The predefined reference class: String

- String is a reference class, we create a handle (**^**) to it and we can reach its properties and methods with **->.** Properties and methods that are often used:
  - Standard operators are defined: ==, !=, +, +=. By native (char*) strings the comparing operator (==) checks whether the two pointers are equal, and it does not check the equality of their content. When using String type the == operator checks the equality of the contents using operator overloading. Similarly, the addition operator means concatenation. As an example: the value *of s+", hey"* will be *"ittykitty, hey".*
  - String->ToString() exists as well because of inheritance. It does not have any pratical importance since it returns the original string. On the other hand, there is no method that converts to a native string (char*). Let us see a function as an example that performs this conversion:

Specialties of CLI, standard C++ and C++/CLI

❖ 1.11. The predefined reference class: String

```
char * Managed2char(String ^s)
{
    int i, size=s->Length;
    char *result=(char *)malloc(size+1); // place for the
    converted string
    memset(result,0,size+1); // we fill the converted with
    end signs
    for (i=0; i<size;i++)     // we go through the characters
        result[i]=(char)s[i]; // here we will got a warning:
        s[i]
    //stored on 2 bytes unicode wchar_t type character.
    //Converting ASCII from this the accents will disappear
    return result; // we will return the pointer to the result
}
```

Specialties of CLI, standard C++ and C++/CLI

❖ 1.12. The System::Convert static class

- The *System* namespace contains a class called *Convert.* The *Convert* class has numerous overloaded static methods, which help the data conversion tasks.
- For performing the most common text <-> number conversions the *Convert::ToString(NumericType)* and the *Convert::ToNumericType(String)* methods are defined.

```
int total=Convert::ToInt32(s1)+Convert::ToInt32(s2);
```

- The Convert class, however, performs the real <-> string conversion according to the region and language settings (**CultureInfo**). The *CultureInfo* can be set for the current program, if for example we got a text file that contains real numbers in English format.

```
// c is the instance of the CultureInfor reference class
System::Globalization::CultureInfo^ c;
// Like we were in the USA
c = gcnew System::Globalization::CultureInfo("en-US");
System::Threading::Thread::CurrentThread->CurrentCulture = c;
// from now onwards in the program the decimal separator is
the point, the list delimiter is the comma
```

Specialties of CLI, standard C++ and C++/CLI

❖ 1.12. The System::Convert static class

- The methods of the Convert class can appear also in the methods of the data class. For example the instance created by the *Int32* class has a *ToString()* method to convert to a string and a *Parse()* method to convert from a string. These methods can be parameterized in several ways. We often use hexadecimal numbers in computer/hardware related programs.

```
if (checkBox7->Checked) c|=0x40;
if (checkBox8->Checked) c|=0x80;
sc="C"+String::Format("{0:X2}",c);// A 2 character hex number
is created from the byte type c. C is the command;
//if the value of c was 8: "C08" will be the output, if c was
255 "CFF".
serialPort1->Write(sc); // we sent it to the hardware
s=serialPort1->ReadLine(); // the answer was returned
// let us convert the answer to an integer
status = Int32::Parse(s, System::Globalization::NumberStyles::
AllowHexSpecifier);
```

Specialties of CLI, standard C++ and C++/CLI

❖ 1.13. The reference class of the array implemented with the CLI array template

- Declaration*: cli::***array<type***, dimension=1>^ arrayname***, the *dimension* is optional; in this case its value is 1. The ^ is the sign of the *ref class*, the *cli::* is also omissible, if we use at the beginning of our file the *using namespace cli;* statement.
- We have to allocate space for the array with the **gcnew** operator before using – since it is a reference class when declaring a variable only the handle is created, and it is not pointing to anywhere. We can make the allocation in the declaration statement as well: we can list the elements of the array between { } as used in C++.
- Array's property: **Length** gives the number of elements of the onedimensional array. For arrays passed to a function we do not have to pass the size, like in the basic C. The size can be used in the loop statement, which does not address out from the array:

```
for (i=0; i<arrayname->Length; i++)…
```

❖ 1.13. The reference class of the array implemented with the CLI array template

For the basic array algorithms static methods were created, and those are stored in the *System::Array* class:

- **Clear**(array, from where, how many) deletion. The value of the array elements will be 0, false, null, nullptr (depending on the base type of the array),
- **Resize**(array, new size) in case of resizing (expanding) after the old elements it fills the array with the values used with Clear().
- **Sort**(array) sorting the elements of the array. It can be used by default to order numerical data in ascendant order. We can set keys and a comparing function to sort any type data.
- **CopyTo**(target array, starting index) copying elements. Note: the = operator **duplicates the reference** only. If an element of the array is changed, this changed element is reached using the other reference as well. Similarly, the == oparetor that the two references are the same but it does not compare the elements themselves.

Specialties of CLI, standard C++ and C++/CLI

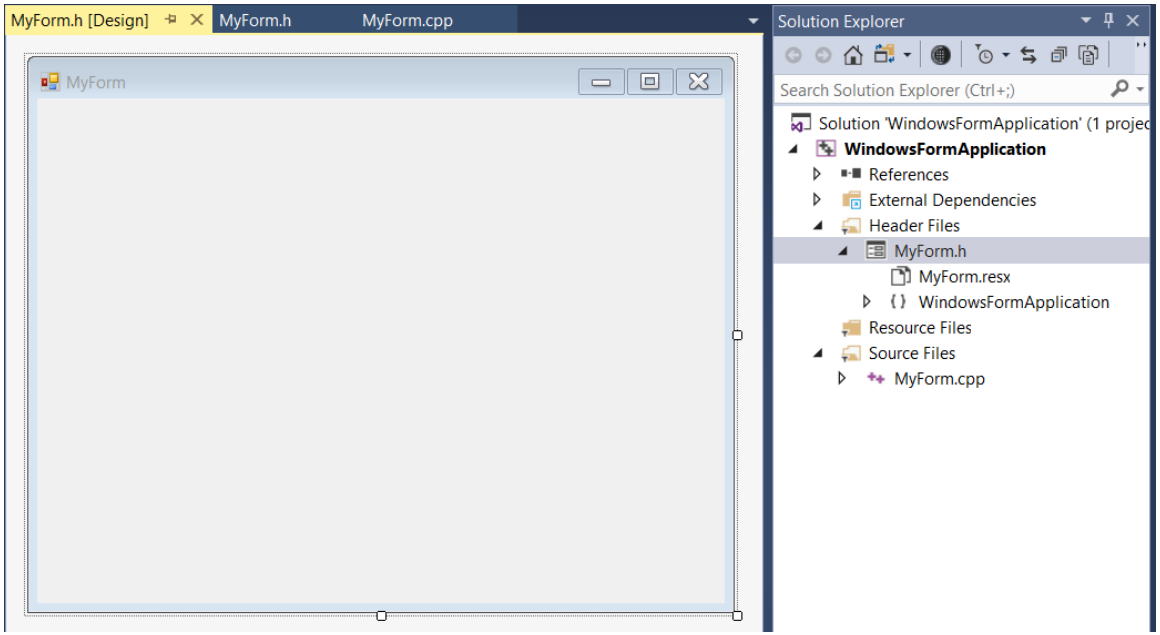
❖ 1.13. The reference class of the array implemented with the CLI array template

```
array<int>^ numbers; // managed array type, reference
Random ^r = gcnew Random();// random number generator instance
int piece = 5, max = 90, i; // we set how many numbers we need and the
  highest number.
                              //It could be set as an input after
                 conversion.
numbers = gcnew array<int>(piece); // managed array on the heap created
for (i = 0; i<numbers->Length; i++)
    numbers[i] = r->Next(max) + 1;// the raw random numbers are in the
       array
Array::Sort(numbers); // with the embedded method we set the numbers in
  order
                        // check: two identical next to each other?
bool rightnumber = true;
for (i = 0; i<numbers->Length - 2; i++)
    if (numbers[i] == numbers[i + 1]) rightnumber = false;
```

Specialties of CLI, standard C++ and C++/CLI

❖ C++/CLI: Practical realization in e.g. in the Visual Studio 2017

- Select Visual C++ **CLR** and **CLR Empty Project** and type in **WindowsFormApplication** for the project name. The, OK.
- **Project->Add New Item...** .
  Select **UI** under **Visual C++**.
  Leave the Form name as given by default **MyForm.h**.
  Then, click **Add**.



Specialties of CLI, standard C++ and C++/CLI

❖ C++/CLI: Practical realization in e.g. in the Visual Studio 2017

- We need to edit the **MyForm.cpp** file:

```
#include "MyForm.h"

using namespace WindowsFormApplication;

[STAThreadAttribute]
int main(cli::array<System::String ^> ^args)
{
    // Enabling Windows XP visual effects before any
      controls are created
    Application::EnableVisualStyles();
    Application::SetCompatibleTextRenderingDefault
      (false);

    // Create the main window and run it
    Application::Run(gcnew MyForm());
    return 0;
}
```

Specialties of CLI, standard C++ and C++/CLI

❖ C++/CLI: Practical realization in e.g. in the Visual Studio 2017

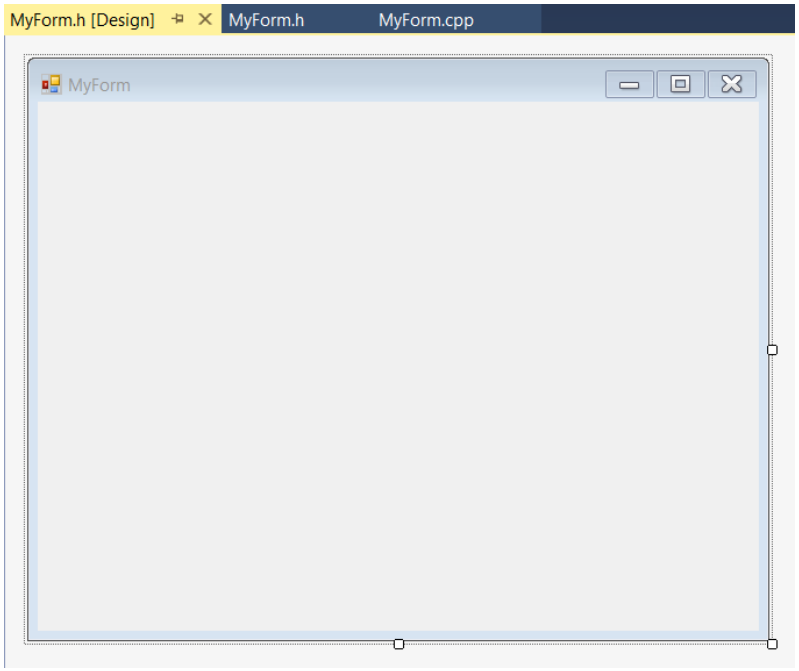
The **System** namespace provides functions to work with UI controls.

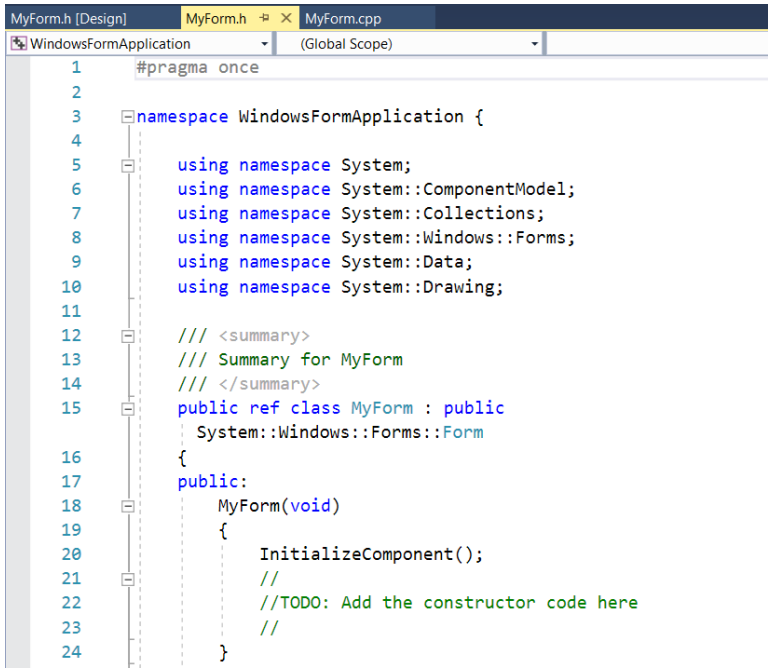
- At the right-mouse click on **WindowsFormApplication**, we get the **Properties** window.
  Configuration Properties->**Linker**->**System**
  Select **Windows (/SUBSYSTEM:WINDOWS)** for **SubSystem**.
  Advanced->**Entry Point**, type in **main**.
  The, hit **OK**.
- Hit **F5**, then we will have to run result, the **Form**.
-

Specialties of CLI, standard C++ and C++/CLI

❖ C++/CLI: Practical realization in e.g. in the Visual Studio 2017

- Using the "*View*/**Designer**" menuitem, we can select the graphical editor, while with the "*View*/**Code**" menuitem the source program.

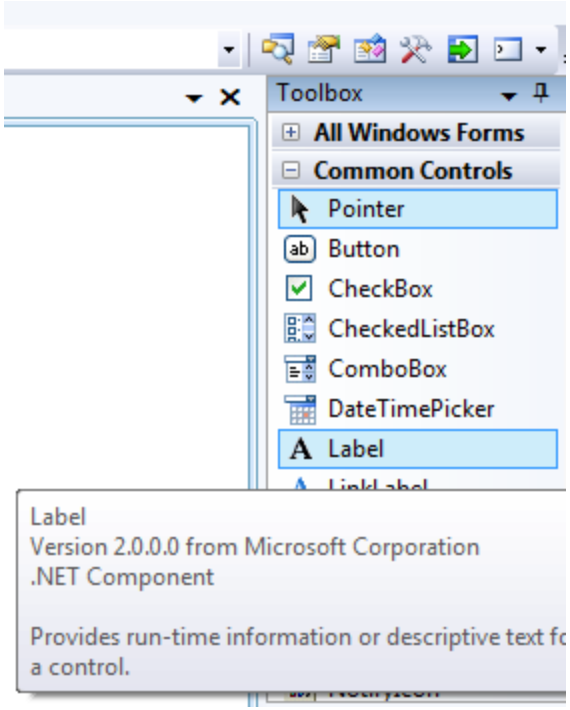




Specialties of CLI, standard C++ and C++/CLI

❖ C++/CLI: Practical realization in e.g. in the Visual Studio 2017

- Selecting the "*View/Designer*" menuitem we will need the Toolbox where the additional controls can be found (the toolbox contains additional elements only in designer state). In case it is not visible we can set it back with the "*View/Toolbox*" menuitem.

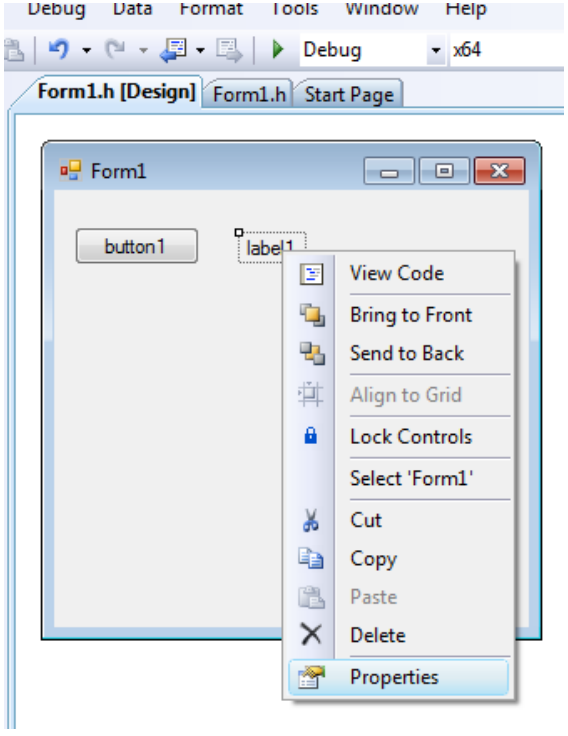


The Toolbox

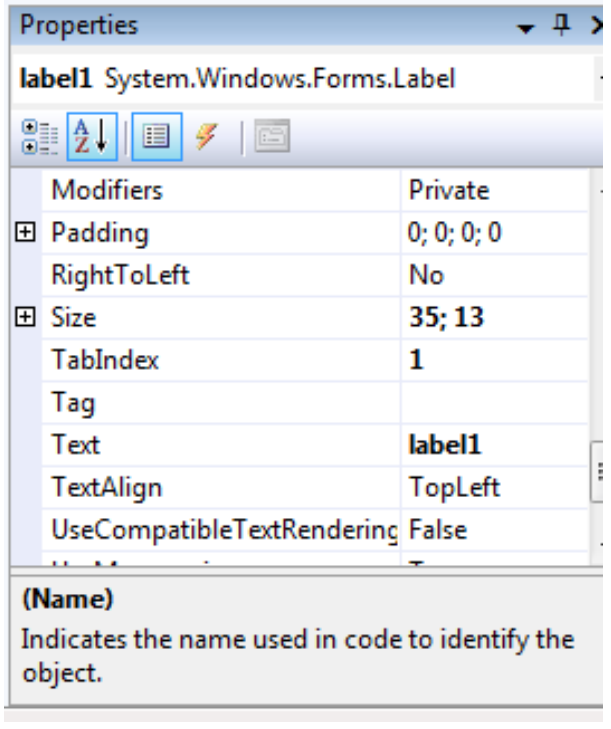
Specialties of CLI, standard C++ and C++/CLI

❖ C++/CLI: Practical realization in e.g. in the Visual Studio 2017

- After selecting the control and with right mouse click we can achieve the setting in the window, opened with the "*Properties*" menuitem. It is important to note that these settings refer to the currently selected control and the properties windows of the certain controls differ from each other. On the next figure we select the "*Properties*" window of the label1 control:



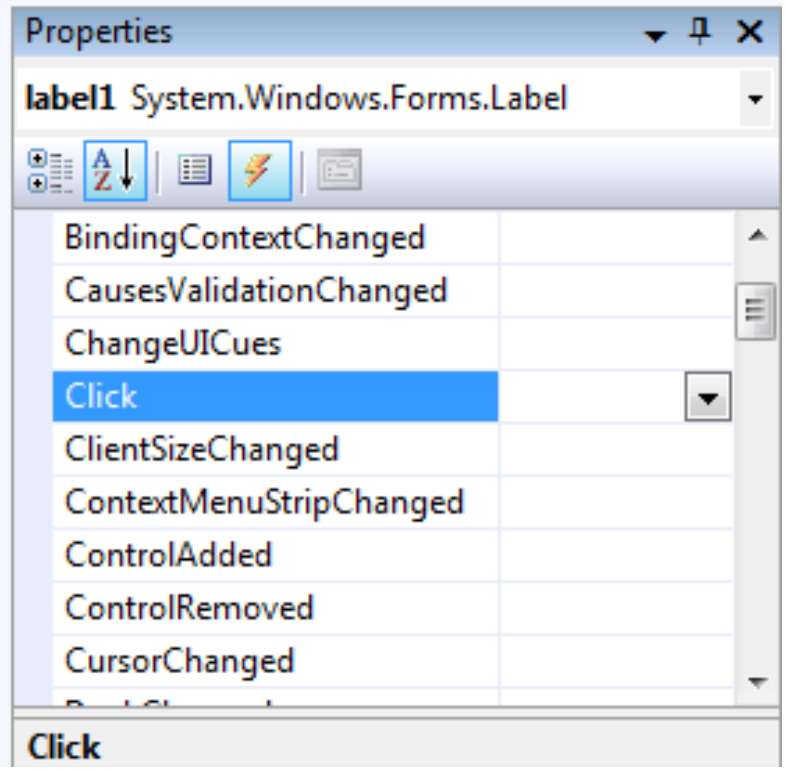
The Properties Window



Specialties of CLI, standard C++ and C++/CLI

❖ C++/CLI: Practical realization in e.g. in the Visual Studio 2017

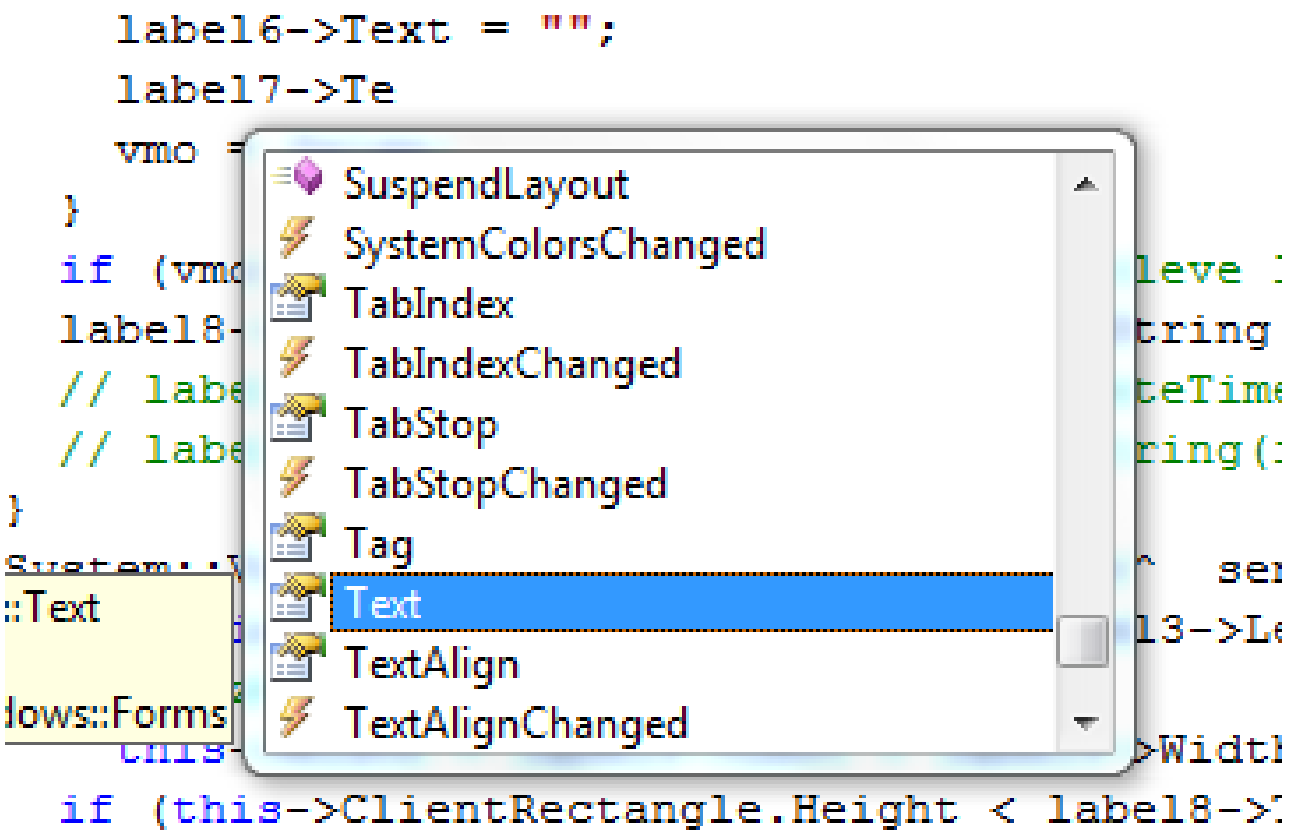
- The same window serves for selecting the event handlers. We have to click on the blitz icon ( ) to define the event handlers. In this case all the reacting options will appear that are possible for all the events of the given control. In case the right side of the list is empty then the control will not react to that event.



The Event handlers

Specialties of CLI, standard C++ and C++/CLI

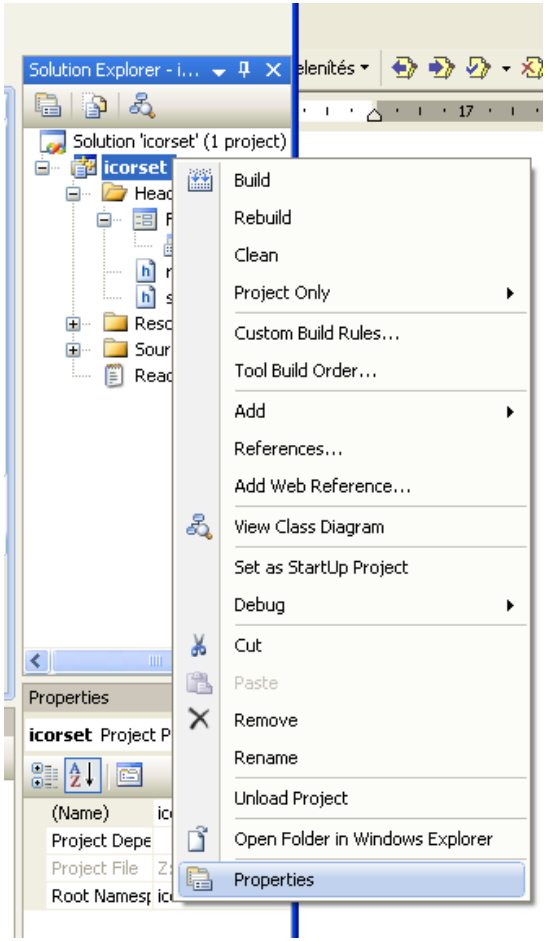
❖ 1.15. The Intellisense embedded help



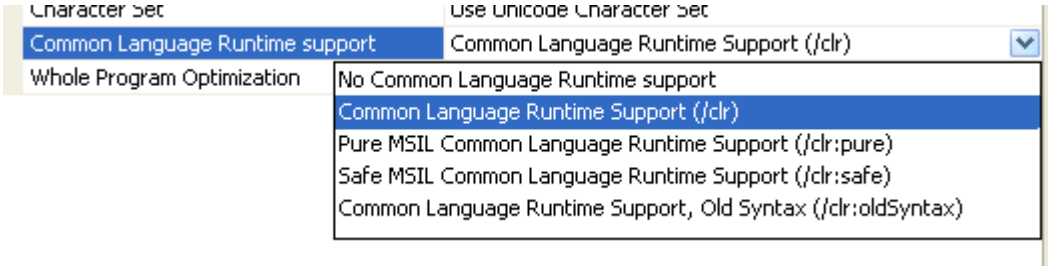
The Intellisense window

Specialties of CLI, standard C++ and C++/CLI

❖ 1.16. Setting the type of a CLR program



Solution Explorer menu



Project properties

Specialties of CLI, standard C++ and C++/CLI

❖ 1.16. Setting the type of a CLR program

- "*No common Language Runtime Support*" – there is no managed code. It is the same if we create a Win32 console application or a native Win32 project. With this setting it is not capable of compiling the parts of the .NET system (handles, garbage collector, reference classes, assemblies).
- "*Common Language Runtime Support*" – there is native and managed code compiling as well. With this setting we can create mixed mode programs, that is, if we started to develop our program with the default window settings and we would like to use native code data and functions, then we have to set the drop down menu to this item.
- "*Pure MSIL Common Language Runtime Support*" – purely managed code compiling. The default setting of programs created from the "*Windows Form Application*". This is the only possible setting of C# compiler. Note: this code type can contain native code data that we can reach through managed code programs.

❖ 1.16. Setting the type of a CLR program

- "*Safe MSIL Common Language Runtime Support*" – it is similar to the previous one but it cannot contain native code data either and it allows the security check of the CRL code with a tool created for this purpose (`peverify.exe`)
- "*Common Language Runtime Support, Old Syntax*" – this also creates a mixed code program but with Visual Studio 2002 syntax. (*_gc new* instead of *gcnew*). This setting was kept to ensure compatibility with older versions, however, it is not recommended to be used.

Chapter IV. Graphical User Interface in C++/CLI

- 1. Specialties of CLI, standard C++ and C++/CLI
- 2. The window model and the basic controls
- 3. Text and binary files, data streams
- 4. The GDI+

2. The window model and the basic controls

2.1. The Form basic controller

2.2. Often used properties of the Form control

2.3. Events of the Form control

2.4. Updating the status of controls

2.5. Basic controls: Label control

2.6. Basic controls: TextBox control

2.7. Basic controls: Button control

2.8. Controls used for logical values: CheckBox

2.9. Controls used for logical values: RadioButton

2.10. Container object control: GroupBox

2. The window model and the basic controls

2.11. Controls inputting discrete values: HscrollBar and VscrollBar

2.12. Control inputting integer numbers: NumericUpDown

2.13. Controls with the ability to choose from several objects: ListBox and ComboBox

2.14. Control showing the status of progressing: ProgressBar

2.15. Control with the ability to visualize PixelGrapic images: PictureBox

2.16. Menu bar at the top of our window: MenuStrip control

2.17. The ContextMenuStrip control which is invisible in basic mode

2.18. The menu bar of the toolkit: the control ToolStrip

2. The window model and the basic controls

2.19. The status bar appearing at the bottom of the window, the StatusStrip control

2.20. Dialog windows helping file usage: OpenFileDialog, SaveFileDialog and FolderBrowserDialog

2.21. The predefined message window: MessageBox

2.22. Control used for timing: Timer

2.23. SerialPort

2. The window model and the basic controls

❖ 2.1. The Form basic controller

```
#pragma region Windows Form Designer generated code
        /// <summary>
        /// Required method for Designer support - do not
          modify
        /// the contents of this method with the code editor.
        /// </summary>
        void InitializeComponent(void)
        {
            this->SuspendLayout();
            //
            // MyForm
            //
            this->AutoScaleDimensions = System::Drawing::SizeF
              (8, 16);
            this->AutoScaleMode =
              System::Windows::Forms::AutoScaleMode::Font;
            this->ClientSize = System::Drawing::Size(620,
              455);
            this->Name = L"MyForm";
            this->Text = L"MyForm";
            this->ResumeLayout(false);

        }
#pragma endregion
    };
}
```

2. The window model and the basic controls

❖ 2.2. Often used properties of the Form control

- **Text** – title of the form. This property can be found at each control that contains text (as well).
- **Size** – the size of the form, by default in pixels. It contains the Width and Height properties that are directly accessible. Also the visible controls have these properties.
- **BackColor** – color of the background. By default it has the same color as the background of the controls defined in the system (*System::Drawing::SystemColors::Control*). This property will be important if we would like to delete the graphics on the form because deletion means filling with a color.
- **ControlBox** – the system menu of the window (minimalizer, maximalizer buttons and windows menu on the left side). It can be enabled (by default) and disabled.
- **FormBorderStyle** – We can set here whether our window can be resized or it should have a fix size or whether it had a frame or not.

2. The window model and the basic controls

❖ 2.2. Often used properties of the Form control

- **Locked** – we can prohibit resizing and movement of the window with the help of this.
- **AutoSize** – the window is able to change its size aligning to its content
- **StartPosition** – when starting the program where should the form appear on the Windows desktop. Its application: if we use a multiscreen environment, then we can set the x,y coordinates of the second screen, our program will be lunched there then. It is useful to set this property in a conditional statement because in case the program is lunched in one screen only the form will not be visible.
- **WindowState** – we can set here whether our program would be a window (Normal), whether it would run full screen (Maximized) or whether it would run in the background (Minimized). Of course, like any of the other properties, it is reachable during run-time as well, that is, if the program lunched in the small window would like to maximalize itself (for example because it would like to show many things) then we have an option for this setting as well: *this->WindowState=FormWindowState::Maximized;*

2. The window model and the basic controls

❖ 2.3. Events of the Form control

- **Load** – a program part that appears when starting the program before displaying it.

```
private: System::Void Form1_Load(System::Object^  sender, System::EventArgs^
  e) {
    this->Text = "quadratic";
    textBox1->Text = "1";  textBox2->Text = "-2"; textBox3->Text = "1";
    label1->Text = "x^2+"; label2->Text = "x+";   label3->Text = "=0";
    label4->Text = "x1=";  label5->Text = "x2=";  label6->Text = "";
    label7->Text = "";     label8->Text = "";
    button1->Text = "Solve it";
    if (this->ClientRectangle.Width < label3->Left + label3->Width) // the
      form is not wide enough
        this->Width = label3->Left + label3->Width + 24;
    if (this->ClientRectangle.Height < label8->Top + label8->Height)
        this->Height = label8->Top + label8->Height + 48; // it is not high
          enough
    button1->Left = this->ClientRectangle.Width - button1->Width - 10; //
      pixel
    button1->Top = this->ClientRectangle.Height - button1->Height - 10;
}
```

2. The window model and the basic controls

❖ 2.3. Events of the Form control

- In case we find in this function that running of the program does not make sense (we would process a file but we could not find it, we would like to communicate with a hardware but we could not find it, we would like to use the Internet but we do not have connection etc.) then after displaying a window of an error message we can leave the program. Here comes a hardware example:

```
if (!controller_exist) {
    MessageBox::Show("No iCorset controller.",
        "Error", MessageBoxButtons::OK);
    Application::Exit();
}
else {
    // controller exist, initialize the controller.
}
```

2. The window model and the basic controls

❖ 2.3. Events of the Form control

- **Resize** – An event handler that runs when resizing our form (minimalizing, maximalizing, setting it to its normal size could be also considered here). It runs when loading the program, this way the increase of the form size from the previous example could have been mentioned here as well, and in this case the form could not be resized to smaller in order to ensure the visibility of our controls. In case we have a graphic which size depens on the window size, then we can resize it here.
- **Paint** – the form has to be repainted.
- **MouseClick, MouseDoubleClick** – we click once or we double click on the Form with the mouse. In case we have other controls on the form then this event runs if we do not click on neither of the controls just on the empty area. In one of the arguments of the event handler we got the handle to the reference class

2. The window model and the basic controls

❖ 2.3. Events of the Form control

- **MouseDown, MouseUp** – we clicked or released one of the mouse buttons on the Form.The Button propety of the MouseEventArgs contains which button was clicked or released. The next program part saves the coordinates of the clicks into a file, this way for example we can create a very basic drawing program:

```
// if save is set and we pushed the left button
if (toolStripMenuItem2->Checked && (e->Button ==
  System::Windows::Forms::MouseButtons::Left)) {
    // we write the two coordinates into the file, x and y as int32
    bw->Write(e->X); // we write int32
    bw->Write(e->Y); // that is 2*4 byte/point
}
```

2. The window model and the basic controls

❖ 2.3. Events of the Form control

- **MouseMove**: - the function running when moving the mouse. It works independently from the buttons of the mouse. In case our mouse is moved over the form, its coordinates can be read. The next program part displays these coordinates in the title of the window (that is, in the Text property), of course after the needed conversions:

```
private: System::Void Form1_MouseMove(System::Object^  sender,
  System::Windows::Forms::MouseEventArgs^  e) {
    // coordinates into the header, nobody looks at those ever
    this->Text = "x:" + Convert::ToString(e->X) + " y=" +
      Convert::ToString(e->Y);
}
```

2. The window model and the basic controls

❖ 2.3. Events of the Form control

- **FormClosing** – our program got a *Terminate()* Windows message for some reason. The source of the message could be anything: the program itself with *Application::Exit()*, the user with clicking on the "*close window*", or the user with the Alt+F4 key combination, we are before stopping the operating system etc. When this function runs the Form is closed, its window disappears, resources used by it will be unallocated. In case our program decides that this is not possible yet, the program stop can be avoided by setting the Cancel member of the event's parameter to true, and the program will run further. The operating system however, if we would like to prevent it from stopping, it will close our program after a while. In the next example the program let itself to be closed only after a question appearing in a dialog window:

```
void Form1_FormClosing(System::Object^  sender,
  System::Windows::Forms::FormClosingEventArgs^  e) {
    System::Windows::Forms::DialogResult d;
    d = MessageBox::Show("Are you sure that you would like to use the
      airbag?",
        " Important security warning ", MessageBoxButtons::YesNo);
    if (d == System::Windows::Forms::DialogResult::No) e->Cancel =
      true;
}
```

❖ 2.3. Events of the Form control

- **FormClosed** – our program is already in the last step of closure process, the window do not exist anymore. There is no way back from here, this is the last event.

2. The window model and the basic controls

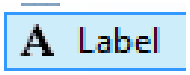
❖ 2.4. Updating the status of controls

```
int i;
for (i = 0; i<100; i++)
{
    this->Text = Convert::ToString(i);
}
```

```
int i;
for (i = 0; i<100; i++)
{
    this->Text = Convert::ToString(i);
    Application::DoEvents();
    Threading::Thread::Sleep(500);
}
```

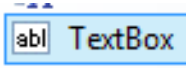
2. The window model and the basic controls

❖ 2.5. Basic controls: Label control



- The simplest control is the Label which displays text. It's String ^ type property called **Text** includes the text to be displayed.

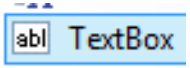
❖ 2.6. Basic controls: TextBox control



- TextBox control can be used for **entering** text (String^) (in case we need to enter numbers, we use the same control, but in this case the processing starts with a conversion). The **Text** property contains the text, which can be rewritten from the program and the user can also change it while the program is running.
- TextBox has a default event as well: it is **TextChanged**, which runs after each and every change (per character).

2. The window model and the basic controls

❖ 2.6. Basic controls: TextBox control




```
double a, b, c, d, x1, x2, e1, e2; // local variables
                                   // in case we forget to give values
                  to any of the variables -> error
a = Convert::ToDouble(textBox1->Text); // String -> double
b = Convert::ToDouble(textBox2->Text);
c = Convert::ToDouble(textBox3->Text);
d = Math::Pow(b, 2) - 4 * a * c; // the method for exponentiation
  exists as well
if (d >= 0) // real roots
{
    x1 = (-b + Math::Sqrt(d)) / (2 * a);
    x2 = (-b - Math::Sqrt(d)) / (2 * a);
    label4->Text = "x1=" + Convert::ToString(x1);
    label5->Text = "x2=" + Convert::ToString(x2);
    // checking
    e1 = a * x1 * x1 + b * x1 + c; // this way we typed less than in
      Pow
    e2 = a * x2 * x2 + b * x2 + c;
    label6->Text = "...=" + Convert::ToString(e1);
    label7->Text = "...=" + Convert::ToString(e2);
}
```

2. The window model and the basic controls

❖ 2.7. Basic controls: Button control

- Button control denotes a command button that "*sags*" when clicking on it. We use command button(s) if the number of currently selectable functions are low.

```
private: System::Void button1_Click(System::Object^  sender,
  System::EventArgs^  e) {
}
```

❖ 2.8. Controls used for logical values: CheckBox

- Text property of the CheckBox control is the text (String^ type), written next to it. Its bool type property is the **Checked**, which is true in case it is checked. CheckedState property can take up three values: apart from 'on' and 'off' it has a third, middle value as well, which can be set up only from the program, when it is running, however it is considered to be checked. In case there are more CheckBoxes in a Form, these are independent of each other: we can set any of them checked or unchecked. Its event: **CheckedChanged** occurs when the value of the Checked property changes.

2. The window model and the basic controls

❖ 2.8. Controls used for logical values: CheckBox


```
switch (checkBox1->Checked)
{
case true:
     checkBox1->Enabled = false;
     step();
     break;
case false:
     while (Math::Abs(f(xko)) > eps) step();
     break;
}
write();
```

2. The window model and the basic controls

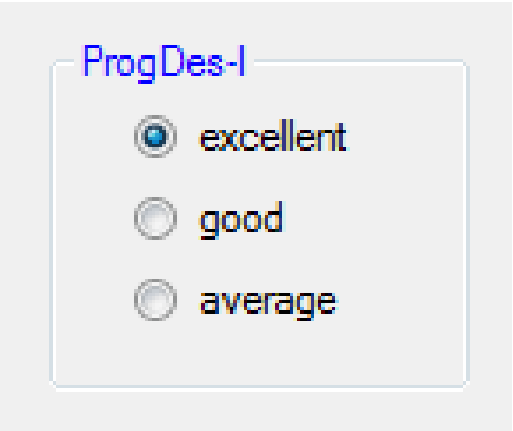
❖ 2.9. Controls used for logical values: RadioButton

- RadioButton is a circular option button. It is similar to the CheckBox, but within one container object (such as Form: we put other objects in it) only one of them can be checked at the same time. It was named after the old radio containing waveband switch: when one of the buttons was pressed, all the others were deactivated. When one of the buttons is marked/activated by the circle (either by the program: Checked = true, or by the user clicking on it), the previous button (and all the others) becomes deactivated (its Checked property changes to false).

2. The window model and the basic controls

❖ 2.10. Container object control: GroupBox 

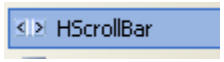
- GroupBox is a rectangular frame with text (Text, String^ type) on its top left line. We placed controls in it, which are framed. On the one hand, it is aesthetic, as the logically related controls appear within one frame, on the other hand it is useful for RadioButton type controls. Furthermore, controls appearing here can be moved and removed with a single command by customizing the appropriate property of GroupBox.



Part of the program's window

2. The window model and the basic controls

❖ 2.11. Controls inputting discrete values: HscrollBar and VscrollBar



- The two controls that are called scrollbars differ only in direction. They do not have labels. In case we would like to indicate their end positions or their current status, we can do this by using separate label controls. The current value, where the scrollbar is standing, is the integer numerical value found in Value property. This value is located between the values of Minimum and Maximum properties. The scrollbar can be set to the Minimum property, this is its left/upper end position. However, for its right/lower end position we have a formula, which includes the quick-change unit of the scrollbar, called LargeChange property (the value changes this much, when we click on the empty space with the mouse): *Value_max=1+Maximum-LargeChange*. LargeChange and SmallChange are property values set by us. When moving the scrollbar a Change event is running with the updated value.

2. The window model and the basic controls

❖ 2.11. Controls inputting discrete values: HscrollBar and VscrollBar


```
int mx;
mx=254 + hScrollBar1->LargeChange; // We would like to have
255 in the right position
hScrollBar1->Maximum = mx; // max. 1 byte
hScrollBar2->Maximum = mx;
hScrollBar3->Maximum = mx;
```

```
System::Drawing::Color c; // color variable
r = hScrollBar1->Value ;  // from 0 to 255
g = hScrollBar2->Value;
b = hScrollBar3->Value;
c = Color::FromArgb(Convert::ToByte(r),Convert::ToByte(g),
Convert::ToByte(b));
label1->Text = "R=" + Convert::ToString(r); //we can also
have them written
label2->Text = "G=" + Convert::ToString(g);
label3->Text = "B=" + Convert::ToString(b);
```

❖ 2.12. Control inputting integer numbers: NumericUpDown

- With the help of NumericUpDown control we can enter an integer number. It will appear in Value property, between the Minimum and Maximum values. The user can increase and decrease the Value by 1, when clicking on the up and down arrows. All the integer numbers between the Minimum and Maximum values appear among the values to choose from. Event: ValueChanged is running after every change.

2. The window model and the basic controls

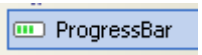
❖ 2.13. Controls with the ability to choose from several objects: ListBox and ComboBox

- ListBox control offers an arbitrary list to be uploaded, from which the user can choose. Beyond the list, ComboBox contains a TextBox as well, which can get the selected item as well and the user is also free to type a string. This is the Text property of ComboBox. The control's list property is called Items, which can be upgraded by using Add() method and can be read indexed. SelectedIndex points the current item. Whenever selection changes, SelectedIndexChanged event is running. ComboBox control is used by several controls implementing more complex functions: for example OpenFileDialog.

```
comboBox1->Items->Add("Excellent");
comboBox1->Items->Add("Good");
comboBox1->Items->Add("Average");
comboBox1->Items->Add("Pass");
comboBox1->Items->Add("Fail");
private: System::Void comboBox1_SelectedIndexChanged(System::
Object^  sender, System::EventArgs^  e) {
  if (comboBox1->SelectedIndex>=0)
    label4->Text=comboBox1->Items[comboBox1->SelectedIndex]->
    ToString();
}
```

2. The window model and the basic controls

❖ 2.14. Control showing the status of progressing: ProgressBar



- With the help of ProgressBar we can give information about the status of the process, how much is still left, the program has not frozen, it is working. We have to know in advance when the process will be ready, when it should reach the maximum. The control's properties are similar to HScrollBar, however the values cannot be modified with the help of the mouse. Certain Windows versions animate the control even if it shows a constant value. Practically, this control used to be placed in the StatusStrip in the bottom left corner of our window. When clicking on it, its Click event is running, however, it is something we normally do not deal with.

2. The window model and the basic controls

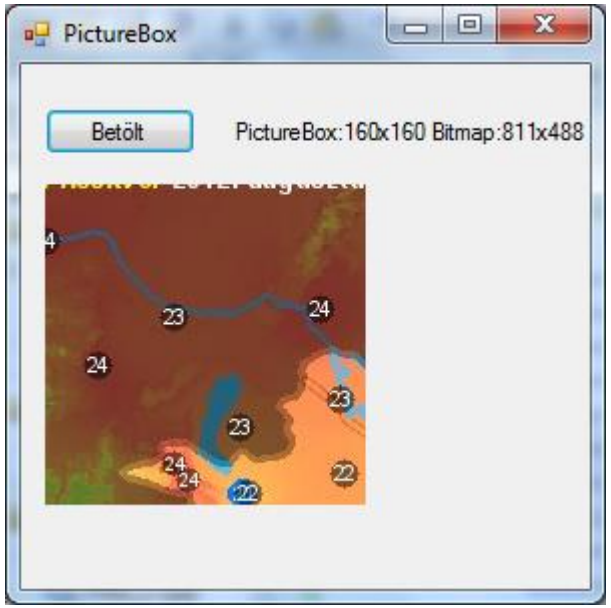
❖ 2.15. Control with the ability to visualize PixelGrapic images: PictureBox 

- This control has the ability to visualize an image. Its **Image** property contains the reference of the Bitmap to be visualized. Height and Width properties show its size, while Left and Top properties give its distance from the left side and top of the window measured in pixels. It is practical to have the size of the PictureBox exactly the same as the size of the Bitmap to be visualized (which can be loaded from almost all kinds of image files) in order to avoid resizing.
- The **SizeMode** property contains the info about what to do if you need to resize the image: in case of *Normal* value, there is no resizing, the image is placed to the left top corner. If Bitmap is larger, the extra part will not be displayed. In case of **StretchImage** value, Bitmap is resized to be as large as the PictureBox. In case of **AutoSize** option, the size of the PictureBox is resized according to the size of the Bitmap. In the program below we display a temperature map (found in idokep.hu) in the PictureBox control. SizeMode property is preset in the Designer, the size parameters of the images are shown in the label:

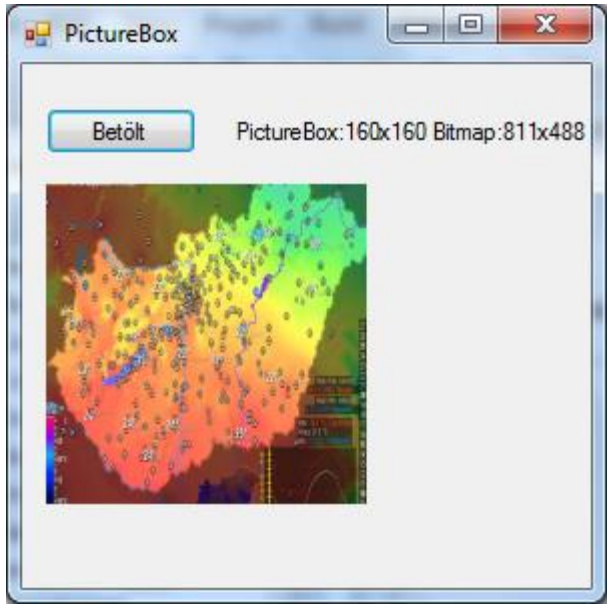
2. The window model and the basic controls

❖ 2.15. Control with the ability to visualize PixelGrapic images: PictureBox


```
Bitmap ^ bm=gcnew Bitmap("mo.png");
label1->Text="PictureBox:"+Convert::ToString(pictureBox1->
Width)+"x"+
     Convert::ToString(pictureBox1->Height)+" Bitmap:"+
     Convert::ToString(bm->Width)+"x"+Convert::ToString(bm->
     Height);
pictureBox1->Image=bm;
```



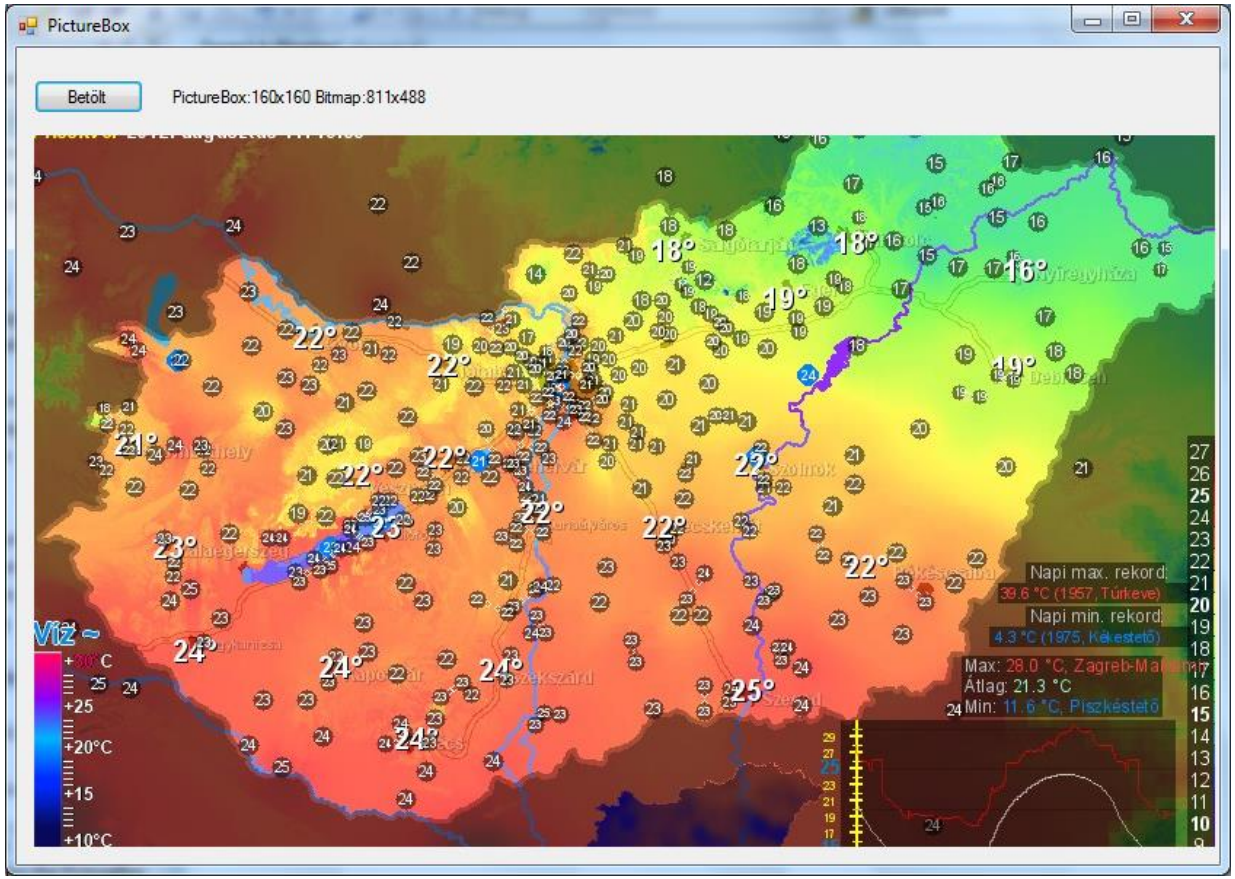
Normal size picturebox on the form



Stretched size picturebox on the form

2. The window model and the basic controls

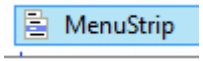
❖ 2.15. Control with the ability to visualize PixelGrapic images: PictureBox



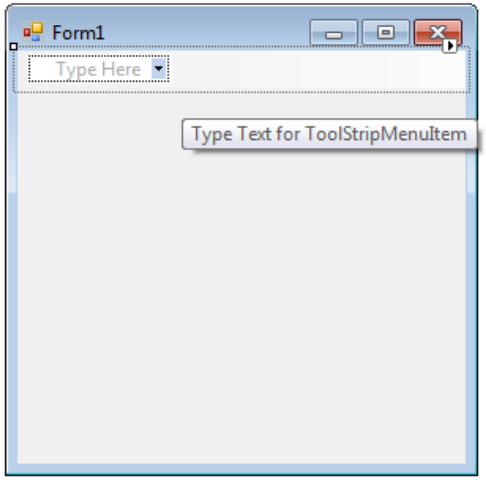
Automatic sized picturebox on the form

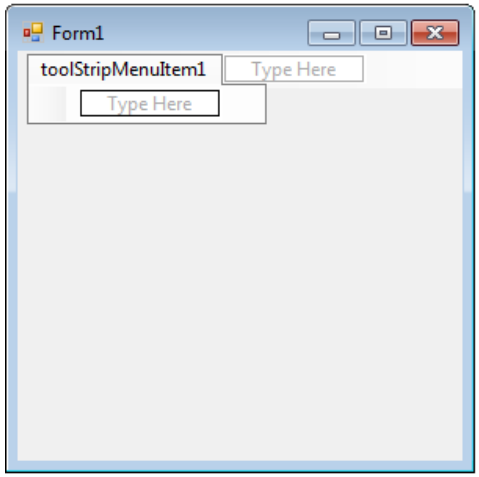
2. The window model and the basic controls

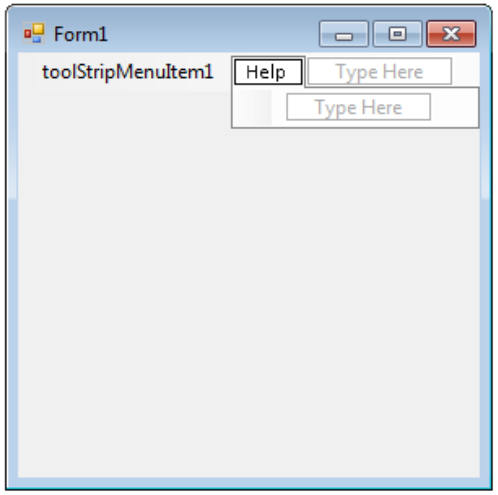
❖ 2.16. Menu bar at the top of our window: MenuStrip control

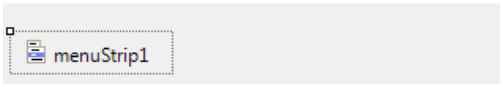


- In case our program implements so many functions that we would require a large number of command buttons in order to start them, it is practical to group functions hierarchically and settle them into a menu.





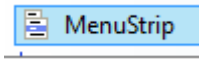


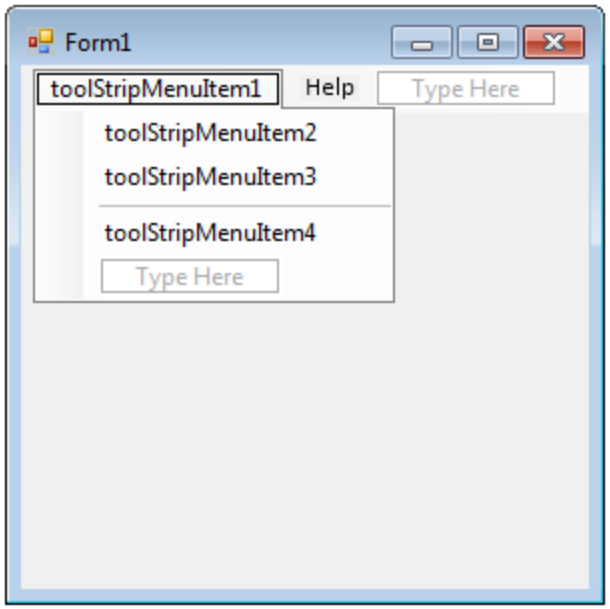


Menustrip **Menuitem on the menustrip** **The Help menu**

2. The window model and the basic controls

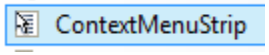
❖ 2.16. Menu bar at the top of our window: MenuStrip control



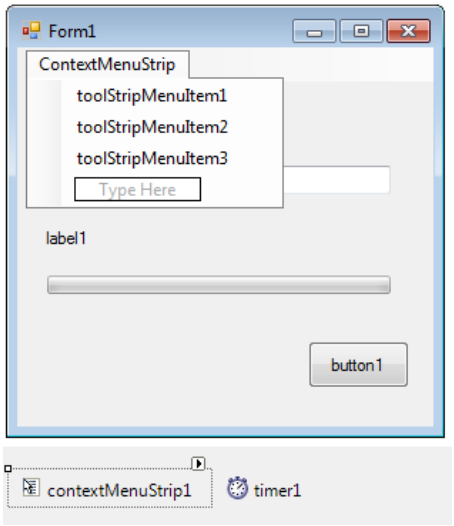


The submenu

2. The window model and the basic controls

❖ 2.17. The ContextMenuStrip control which is invisible in basic mode 

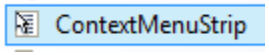
- The main menu of the MenuStrip control is always visible. The ContextMenuStrip can only be seen while designing, during running it is only visible if it is visualized from the program. It is recommended to use it with the local menu appearing at the mouse cursor after clicking the right mouse button. We create the items of the menu in Designer, we write the Click event hanler of the menu items in the editor then in the MouseDown or MouseClick event handler of the Form (they have a coordinate parameter) we visualize the ContextMenu.



The contextmenu

2. The window model and the basic controls

❖ 2.17. The ContextMenuStrip control which is invisible in basic mode

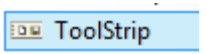


- The event control Form_MouseClick looks like this:

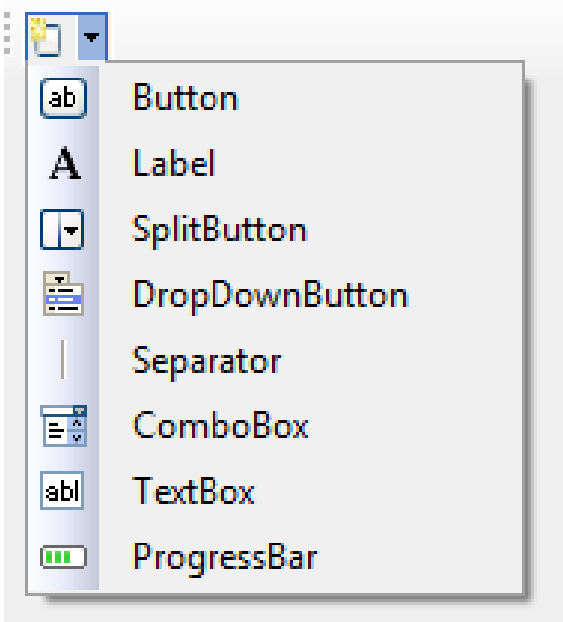
```
private: System::Void Form1_MouseClick(System::Object^  sender
, System::Windows::Forms::MouseEventArgs^  e) {
  if (e->Button == Windows::Forms::MouseButtons::Right) //
  only for the right button
    contextMenuStrip1->Show(this->ActiveForm, e->X, e->Y); //
    we show the menu to the Form
}
```

2. The window model and the basic controls

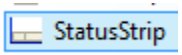
❖ 2.18. The menu bar of the toolkit: the control ToolStrip



- The toolkit contains graphical command buttons next to each other. The Image property of the buttons contains the visualized image. Since they are command buttons, they run the Click event when clicking on them. The figure below shows the elements of the toolstrip, with the button contatining the image at the top, which will bear a name starting with *toolStripButton*:



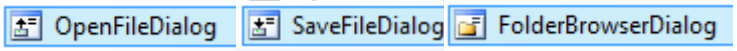
Toolkit on toolstrip

❖ 2.19. The status bar appearing at the bottom of the window, the StatusStrip control 

- A status bar visualises status information, therefore it is useful to create a label and a progressbar for it. The name of the label placed over it starts with toolStripLabel, that of the progressbar will start with toolStripProgressBar. It is not clicked in general, so Click event handlers are not written for it.

2. The window model and the basic controls

❖ 2.20. Dialog windows helping file usage: OpenFileDialog, SaveFileDialog and FolderBrowserDialog



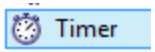
- Almost all computer programs in which modifications carried out can be saved contain the Open file and Save file functions. Selecting a file to be saved or to be opened is done with the same window type in each software because the developers of .NET created a uniform control for these windows.

```
System::Windows::Forms::DialogResult dr;
openFileDialog1->FileName = "";
openFileDialog1->Filter = "CSV files (*.csv)|*.csv";
dr=openFileDialog1->ShowDialog(); // open dialog window for a
csv file
filename = openFileDialog1->FileName; // getting the file name.
if (dr==System::Windows::Forms::DialogResult::OK ) // if it
is not the Cancel button that was clicked
  sr = gcnew StreamReader(filename); // opened for reading
```

❖ 2.21. The predefined message window: MessageBox

- A MessageBox is a dialog window containing a message and is provided by the Windows. In a message box, the user can choose one of the command buttons.
- MessageBox::Show() can be called with one string argument: in that case, it will not have a header, and will have only the "*OK*" button.

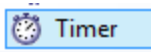
2. The window model and the basic controls

❖ 2.22. Control used for timing: Timer 

- The Timer control makes it possible to run a code portion at given time intervals. It should be put under the form in the Designer, because it is not visible during execution.
- Its property named Interval contains the interval in milliseconds, and if the Enabled property is set to false, the timer can be disabled. If it is enabled and Interval contains a useful value (>=20 ms), the Timer executes its default event handler, the Tick. Progammers have to make sure that the code portion finish before the next event is occured (that is when time is out). The following code portion sets up the timer1 to run in every second:

```
timer1->Interval=1000; // every second
timer1->Enabled=true; // go !
```

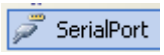
2. The window model and the basic controls

❖ 2.22. Control used for timing: Timer 

- The programmed timer of the next code prints out the current time in seconds into the title of the form:

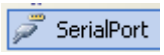
```
DateTime^ now=gcnew DateTime(); // variable of type time.
now=now->Now; // what's the time?
this->Text=Convert::ToString(now); // show in the title of
the form.
```

2. The window model and the basic controls

❖ 2.23. SerialPort 

- The SerialPort control makes possible the communication between a serial port of rs-232 standard and the peripherial devices connected to it (modem, SOC, microcontroller, Bluetooth device). The port has to be parameterized then opened. Then, textual information (Write, WriteLine, Read, ReadLine) and binary data (WriteByte, ReadByte) can be sent and received. It also has an event handler function: if data arrive, the DataReceived event is run. The following code portion checks all available serial ports, and searches the hardware named "*iCorset*" by sending a "*?*" character. If it is not found, it exits with an error message. In the case of virtual serial ports (USBs), the value of the BaudRate parameter can be anything.

2. The window model and the basic controls

❖ 2.23. SerialPort 


```
array<String^>^ portnames;
bool there_is_a_controller=false;
int i;
portnames=Ports::SerialPort::GetPortNames();
i=0;
while (i<portnames->Length && (!there_is_a_controller))
{
    if (serialPort1->IsOpen) serialPort1->Close();
    serialPort1->PortName=portnames[i];
    serialPort1->BaudRate=9600;
serialPort1->DataBits=8;
    serialPort1->StopBits=Ports::StopBits::One;
serialPort1->Parity = Ports::Parity::None;
    serialPort1->Handshake = Ports::Handshake::None;
    serialPort1->RtsEnable = true;
    serialPort1->DtrEnable = false;
    serialPort1->ReadTimeout=200; // 0.2 s
    serialPort1->NewLine="\r\n";
    try {
        serialPort1->Open();
        serialPort1->DiscardInBuffer();// usb !
        serialPort1->Write("?");
        s=serialPort1->ReadLine();
        serialPort1->Close();
    } catch (Exception^ ex) {
        s="Timeout";
    }
    if (s=="iCorset") there_is_a_controller=true;
    i++;
}
if (! there_is_a_controller) {
    MessageBox::Show("No iCorset Controller.",
"Error",MessageBoxButtons::OK);
    Application::Exit();
}
```

Chapter IV. Graphical User Interface in C++/CLI

- 1. Specialties of CLI, standard C++ and C++/CLI
- 2. The window model and the basic controls
- 3. Text and binary files, data streams
- 4. The GDI+

3. Text and binary files, data streams

3.1. Preparing to handling files

3.2. Methods of the File static class

3.3. The FileStream reference class

3.4. The BinaryReader reference class

3.5. The BinaryWriter reference class

3.6. Processing text files: the StreamReader and StreamWriter reference classes

3.7. The MemoryStream reference class

3. Text and binary files, data streams

❖ 3.1. Preparing to handling files

- Contrary to graphics or to controls, file handling namespace is not inserted in the form when a new project is created. This is the task of programmers to insert it at the beginning of the form1.h, where the other namespaces are:

```
using namespace System::IO;
```

Another thing to do is to decide what the file to be handled contains and what we would like to do with it:

- Only deleting, renaming, copying it or checking whether it exists.
- We would like to handle it by bytes (as a block containing bytes) if we are brave enough: virus detection, character encoding etc.
- It is a binary file with fixed-length record structure.
- If it is a text file with variable-length lines (Strings), and the lines end with line feed.

3. Text and binary files, data streams

❖ 3.1. Preparing to handling files

The file name can be given in two ways:

- The file name is given in the code, therefore it is hard coded. If the file is only for the usage of programmer and it is always the same file, than this is a simple and rapid way of providing the name of the file.
- Users can select the file by using **OpenFileDialog** or **SaveFileDialog** (see section Section 2.20, "Dialog windows helping file usage: OpenFileDialog, SaveFileDialog and FolderBrowserDialog"). In this case, the name of the file is stored in the **FileName** property of these dialog boxes.

❖ 3.2. Methods of the File static class

- bool File::Exists(String^ filename) It examines the existence of the file given in filename, if it exists, the output is true if not, it is false. By using it we can avoid some errors: opening a non-existing file, overwriting an important data file by mistake.
- void File::Delete(String^ filename) It deletes the file given in filename. As opposed to current operating systems, deletion does not mean moving to a recycle bin but a real deletion.
- void File::Move(String^ oldname, String^ newname) It renames the disk file named oldname to newname. If there are different paths in the filenames, the file moves into the other directory.
- void File::Copy(String^ sourcefile, String^ targetfile) This method is similar to Move, except that the source file does not disappear but the file is duplicated. A new file is created on the disk, with the content of the source file.

❖ 3.2. Methods of the File static class

FileStream^ File::Open(String^ filename, FileMode mode) Opening the given file. The FileStream^ does not get its value with gcnew but with this method. It does not have to be used for text files but for all the other files (containing bytes or binary files containing records) opening should be used. The values of mode:

- FileMode::Append we go to the end of the text file and switch to write mode. If the file does not exist, a new file is created.
- FileMode::Create this mode creates a new file. If the file already exists, it is overwritten. In the directory of the path, the current user should have a right for writing.
- FileMode::CreateNew this mode also creates a new file but if the file already exists, we get an exception.
- FileMode::Open opening an existing file for reading/writing. This mode is generally used after creating the file, e.g. after using the FileOpenDialog.

❖ 3.2. Methods of the File static class

FileStream^ File::Open(String^ filename, FileMode mode) Opening the given file. The FileStream^ does not get its value with gcnew but with this method. It does not have to be used for text files but for all the other files (containing bytes or binary files containing records) opening should be used. The values of mode:

- FileMode::OpenOrCreate we open an existing file, if it does not exist, a file with the given name is created.
- FileMode::Truncate we open an existing file and delete its content. The length of the file will be 0 byte.

3. Text and binary files, data streams

❖ 3.3. The FileStream reference class

If files are processed by bytes or they are binary, we need to create a *FileStream^* object for the file. The class instance of *FileStream* is not created by *gcnew* but by *File::Open()*, so the physical disk file and *FileStream* are assigned to each other. With the help of *FileStream* the actual file pointer is accessable, and it can be moved. The measure unit of the position and the movement is byte; its data type is 64 bit integer so that it should manage files bigger than 2GB. Its most frequently used properties and methods:

- **Length**: read-only property, the actual size of the file in bytes.
- **Name**: the name of the disk file that we opened.
- **Position**: writable/readable property, the current file position in bytes. The next writing operation will write into this position, the next reading will read from here.
- **Seek**(how much, with regard to what) method for movng the file pointer. With regard to the Position property, it can be given how we understand movement: from the beginning of the file (SeekOrigin::Begin), from the current position (SeekOrigin::Current), from the end of the file (SeekOrigin::End). This operation must be used also when we attach BinaryReader or BinaryWriter to FileStream since they do not have a Seek() method.

❖ 3.3. The FileStream reference class

Its most frequently used properties and methods:

- int ReadByte(), WriteByte(unsigned char) methods for reading and writing data of one byte. Reading and writing happens in the current position. At the level of the operating system, file reading is carried out into a byte array; these functions are realized as reading an array with one element.
- int Read(array<unsigned char>, offset, count): a method for reading bytes into a byte array. The bytes will be placed from the array's element with the index offset and the count is maximum number of bytes to read. Its return value is: how many bytes could be read.

❖ 3.3. The FileStream reference class

Its most frequently used properties and methods:

- Write(array<unsigned char>,offset, count): a method for writing a block of bytes from a byte array. Writing begins at at the element with the index offset and it writes maximum count elements.
- Flush(void): clears buffers for this stream and causes any buffered data to be written to the file.
- Close(): closing FileStream. Files should be closed after use in order to avoid data loss and running out of resources.

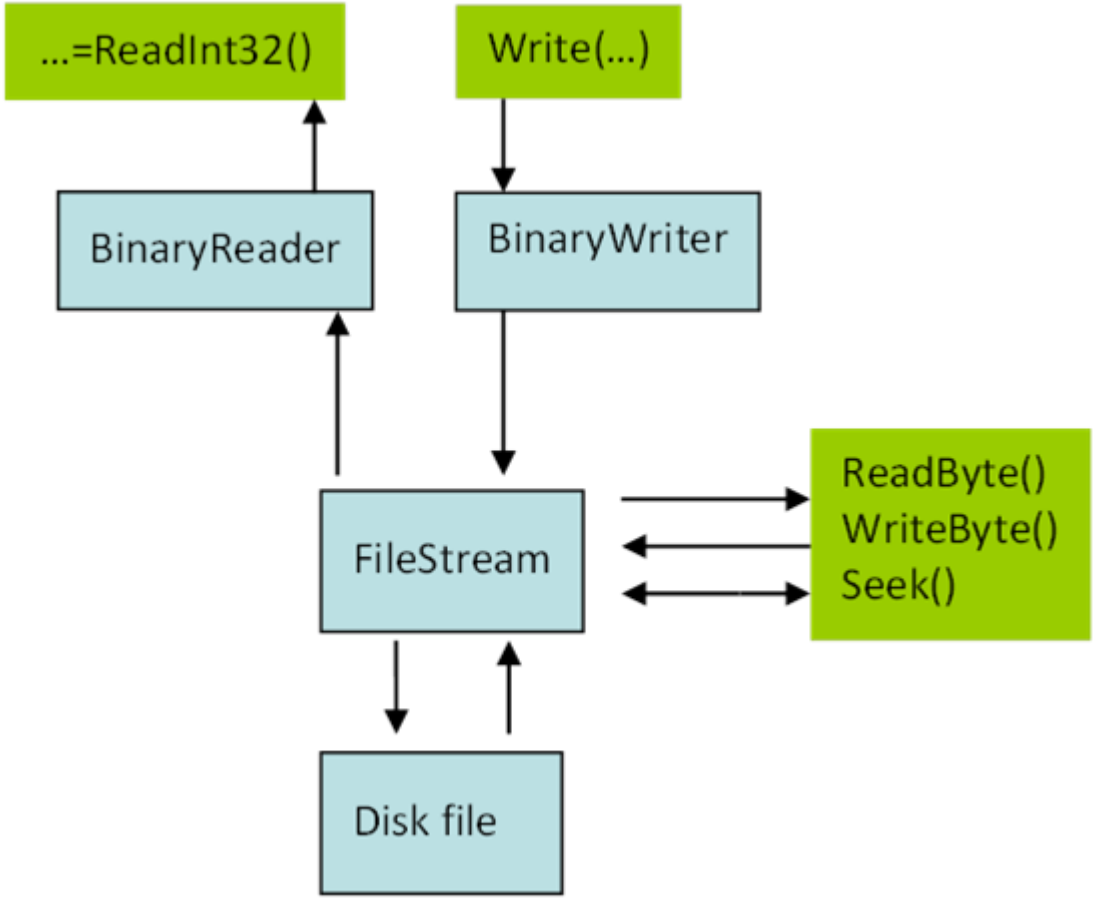
❖ 3.4. The BinaryReader reference class

- If we want to read non-byte type binary data from a file, we use BinaryReader. In the BinaryReader ‘s constructor we give the opened FileStream handle as an argument. BinaryReader is created with a regular gcnew operator. It is important to note that BinaryReader is not able to open the disk file and to assign it to a FileStream. BinaryReader contains methods for the basic data types: ReadBool(), ReadChar(), ReadDouble(), ReadInt16(), ReadInt32(), ReadInt64(), ReadUInt16(), ReadString(), ReadSingle() etc. The file pointer is incremented with the length of the read data. BinaryReader also should be closed after use with the method Close() before closing the FileStream.

❖ 3.5. The BinaryWriter reference class

- If we want to write binary data into *FileStream*, we use *BinaryWriter*. It is created similarly to *BinaryReader*, with the operator *gcnew*. The difference is that *Reader* contained methods with a given return data type but *Writer* contains a method with a given parameter and without a return value, with a number of overloaded versions. The name of the method is *Write* and it can be used with several data types: from *Bool* to *cli::Array^* in the order of complexity. The overview of binary file processing can be seen below:

❖ 3.5. The BinaryWriter reference class



Binary file processing

❖ 3.6. Processing text files: the StreamReader and StreamWriter reference classes

- Text files are composed of *variable-length* lines that are legible for human beings as well. In these files, characters are stored in ASCII, Unicode, UTF-8 etc. encoding, one line of a text file corresponds to the data type *String^* of the compiler. Lines end with CR/LF (two characters) under DOS/Windows-based systems. Because of variable-length lines, text files **can only be processed sequentially**: reading the 10th line can be done by reading the first 9 lines and finally the requested 10th line. When the file is opened, it cannot be calculated at which byte the 10th line starts in the file, only after all preceding lines have been read.

❖ 3.6. Processing text files: the StreamReader and StreamWriter reference classes

- Text files have an important role in realizing *communication* between different computer programs. Since they can be read for example in NotePad, humans can also modify their content. Text files are used, among other things, to save databases (in that case, a text file is called dump and it contains SQL statements that create the saved database on an empty system), to communicate with Excel (comma or tabulator separated files with CSV extension) and even e-mails are transferred as text files between incoming and outgoing e-mail servers. Measuring devices also create text files containing the measurement results. In these files, each line contains one measurement data in order that these data could be processed or visualized with any software (even with Excel) by a user carrying out the measurement.

3. Text and binary files, data streams

❖ 3.6. Processing text files: the StreamReader and StreamWriter reference classes

- Text files can be processed by reference variables of type StreamReader and StreamWriter classes. For that purpose, the gcnew operator should be used, and the name of the file should be specified in its constructor. A FileStream is not needed to be defined because StreamReader and StreamWriter use exclusively disk files; therefore they can create for themselves their own FileStream (BaseStream) with which programmers do not have to deal. The most frequently used method of StreamReader is ReadLine(), which reads the next line of the text file and its most frequently used property is EndOfStream, which becomes true accessing the end of the file. Attention: EndOfStream shows the state of the latest reading operation, ReadLine() returns a zero-length string at the end of the file, and the value of EndOfStream will be true. Thus, the ordinary pre-test loops can be used (while (! StreamReader->EndOfStream) …). One only has to examine if the length of the currently read line is greater than 0. The most frequently used method of StreamWriter is WriteLine(String), which writes the string passed as a parameter and `the newline character in the text file. Write(String) is the same but it does not write the newline character. The newline character(s) (CR,LF,CR/LF) can be set by the NewLine property.

3. Text and binary files, data streams

❖ 3.6. Processing text files: the StreamReader and StreamWriter reference classes

```
private: System::Void button1_Click(System::Object^  sender,
System::EventArgs^  e) {
int m, credits, grade, creditsum = 0, gradesum = 0;
System::Windows::Forms::DialogResult dr;
String^ subject, ^line, ^outputstring = "";
openFileDialog1->FileName = "";
openFileDialog1->Filter = "CSV files (*.csv)|*.csv";
dr=openFileDialog1->ShowDialog(); // file open dialogbox, csv
file
filename = openFileDialog1->FileName; // getting the file name
if (dr==System::Windows::Forms::DialogResult::OK ) // if it
is not the Cancel button that was clicked
{
  sr = gcnew StreamReader(filename); // open for reading
  sw = gcnew StreamWriter(tmpname); // open the file for
  writing
  while (!sr->EndOfStream) // always using a pre-test loop
  {
    line = sr->ReadLine(); // a line is read
    if ((line->Substring(0,1) != csch) && (line->Length > 0))
    // if it is not a separating character, it is processed
    {
      m = 0; // all lines are read from the first character
      subject = newdata(line,  m); // lines are split
      credits = Convert::ToInt32(newdata(line, m)); // into 3
      parts
      grade = Convert::ToInt32(newdata(line,  m));
```

3. Text and binary files, data streams

❖ 3.6. Processing text files: the StreamReader and StreamWriter reference classes

```
            // composing the output string
            outputstring = outputstring + "subject:"+subject + "
            credits:" +
                    Convert::ToString(credits) +" grade:" +
                    Convert::ToString(grade) + "\n";
            // and a weighted average is counted
            creditsum = creditsum + credits;
            gradesum = gradesum + credits * grade;
            sw->WriteLine(line+csch+Convert::ToString(credits*grade
            ));
          } else { // is not processed but written back to the file.
            sw->WriteLine(line);
          } // if
        } // while
        sr->Close(); // do not forget to close the file
        wa = (double) gradesum / creditsum; // otherwise the result
        is integer
        // the previous line contained \n at its end, the result is
        written in a new line
        outputstring = outputstring + "weighted average:" + Convert
        ::ToString(sa);
        label1->Text = outputstring;
        sw->WriteLine(csch + "weighted average"+ csch + Convert::
        ToString(wa));
        sw->Close(); // output file is also closed,
        File::Delete(filename); // the old data file is deleted
        // and the temporary file is renamed to the original data
        file.
        File::Move(tmpname, filename);
        }
    }
```

❖ 3.7. The MemoryStream reference class

- One can also create sequential files, composed of bytes that are not stored on a disk but in the memory. A great advantage of streams created in the memory is the speed (a memory is at least two times faster than a storage device), its disadvantage is its smaller size and that its content is lost if the program is exited. MemoryStream has the same methods as FileStream: it reads/writes a byte or an array of bytes. It can be created by the gcnew operator. The maximal size of a MemoryStream can be set in the parameter of the constructor. If no parameter is given, MemoryStream will allocate memory dynamically for writing. Using that class has two advantages as compared to arrays: on one hand, automatic allocation and on the other hand, a MemoryStream can easily be transformed into a FileStream if memory runs out by using File::Open() instead of gcnew.

KỸ THUẬT LẬP TRÌNH HỆ CƠ ĐIỆN TỬ

Programming Engineering in Mechatronics

Giảng viên: TS. Nguyễn Thành Hùng

Đơn vị: NCM Robot, Khoa Cơ điện tử, Trường Cơ khí

Hà Nội, 2022

Chapter IV. Graphical User Interface

1. What is QT?
2. QT Framework
3. QT in Visual Studio
4. Qt5 C++ GUI Development

Chapter IV. Graphical User Interface

1. What is QT?
2. QT Framework
3. QT in Visual Studio
4. Qt5 C++ GUI Development

What is QT?

- A software development framework
  - Qt framework
    - APIs
  - Qt Creator IDE
    - Design and debug
  - Tools and toolchains
    - Simulator, complier, device toolchains
- Qt is released on 1991 by Trolltech
  - Nokia acquired Trolltech in 2008
  - Free and open source software to puclic
  - C+ is the primary programming language

Chapter IV. Graphical User Interface

1. What is QT?
2. QT Framework
3. QT in Visual Studio
4. Qt5 C++ GUI Development

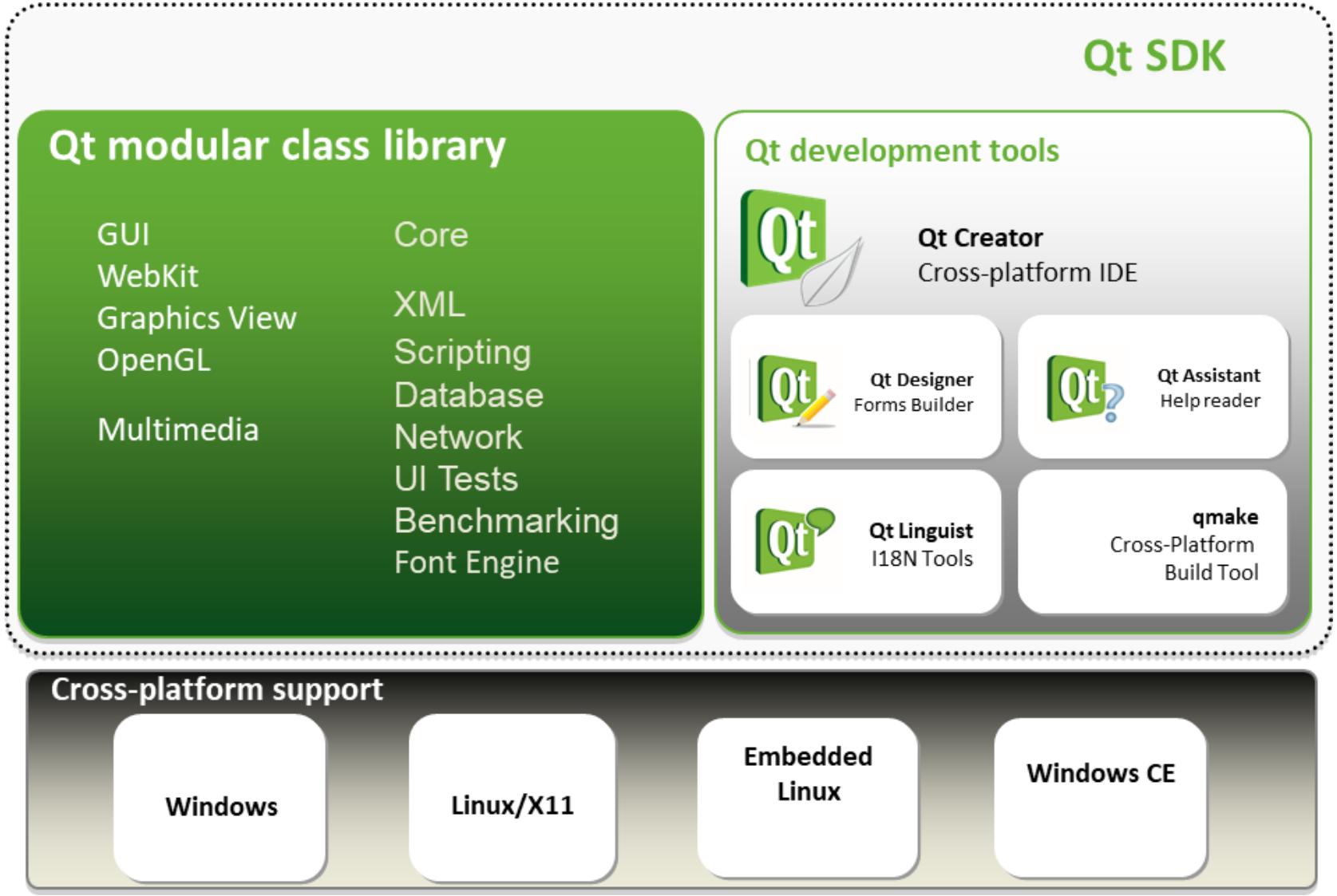
QT Framework

- Qt is cross-platform application and UI framework.
- Qt provides a well defined API that can make development quick and easy.
- Webkit
  - Well accepted open source web browser
  - Rapidly create real-time web content and services
  - Use HTML and Java Script integrated in native code

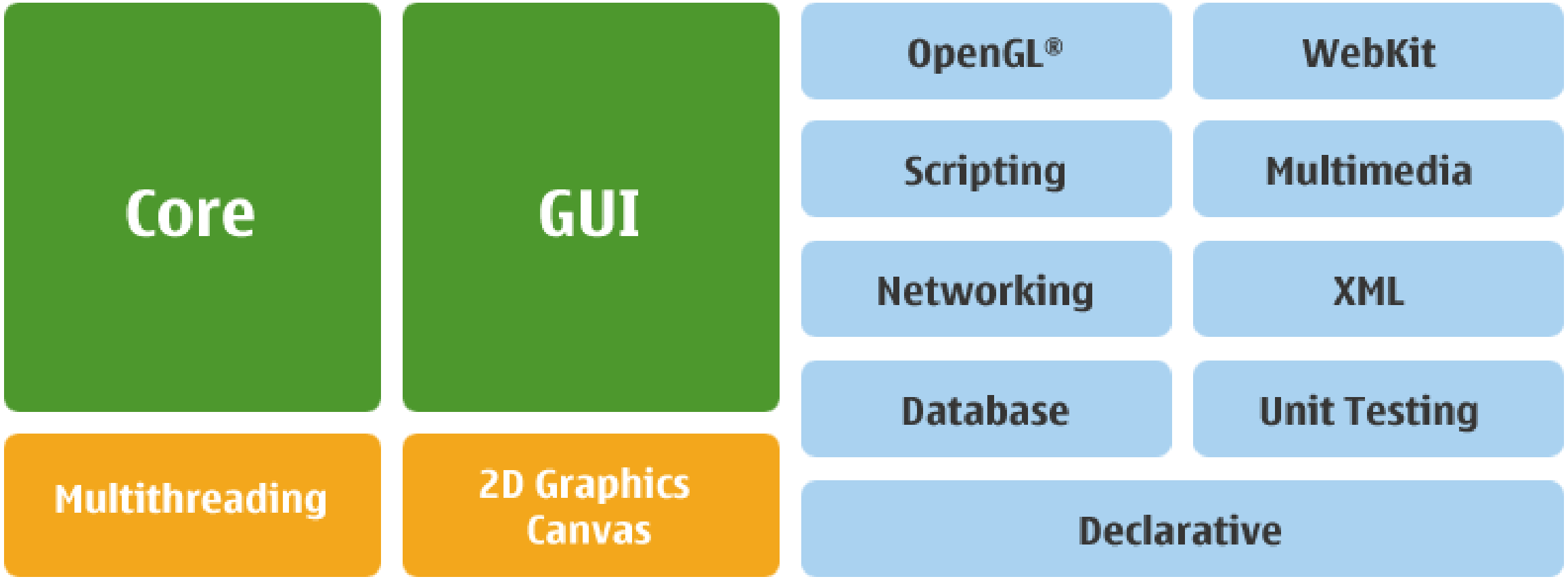
QT Framework

- 3D Graphics with OpenGL and OpenGL ES
  - Easily incorporate 3D Graphics in your applications
  - Get maximum graphics performance
- Multithreading support
- Network connectivity
- Advanced GUI development

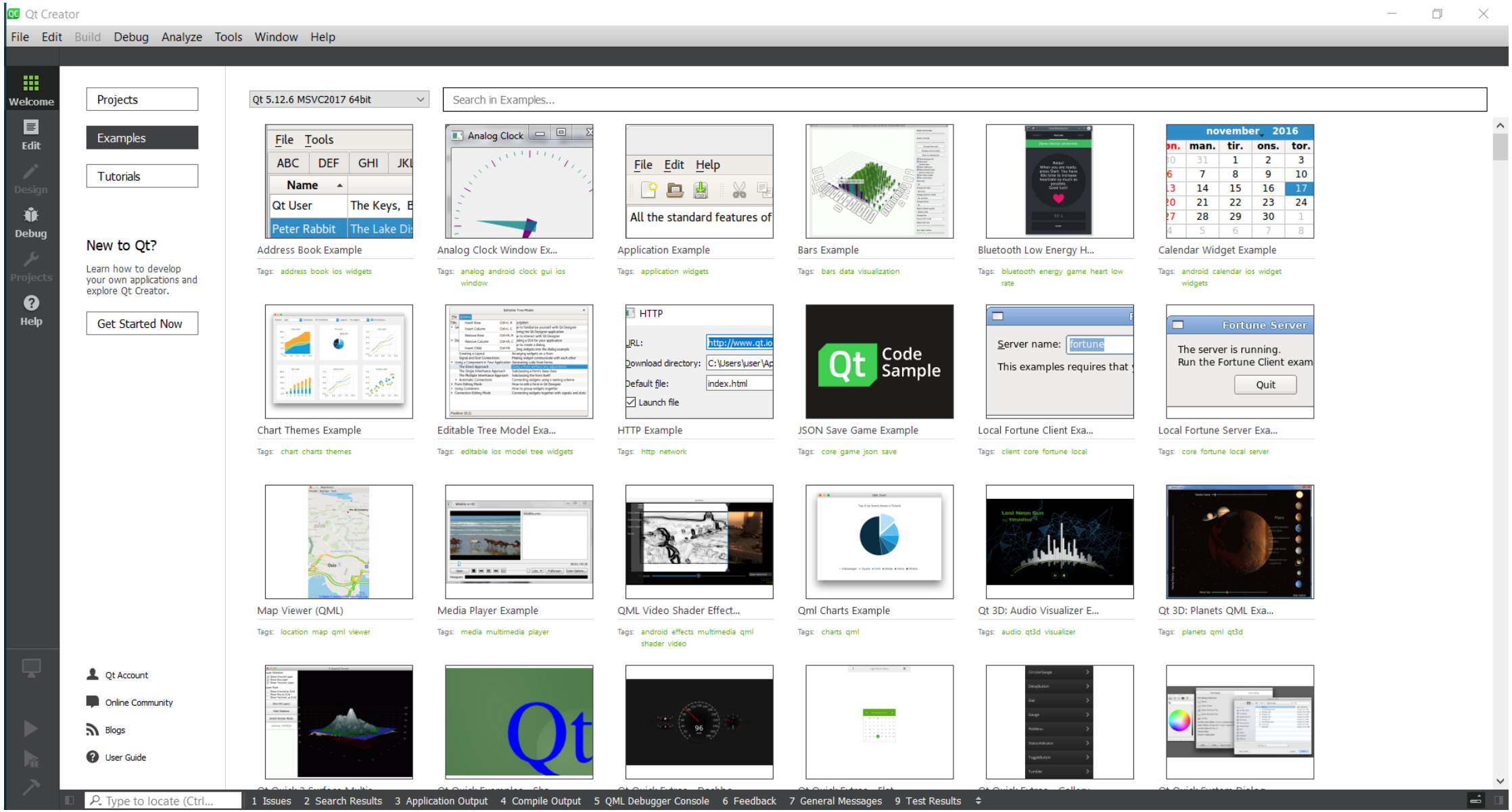
QT Framework



Qt Framework – Application Classes

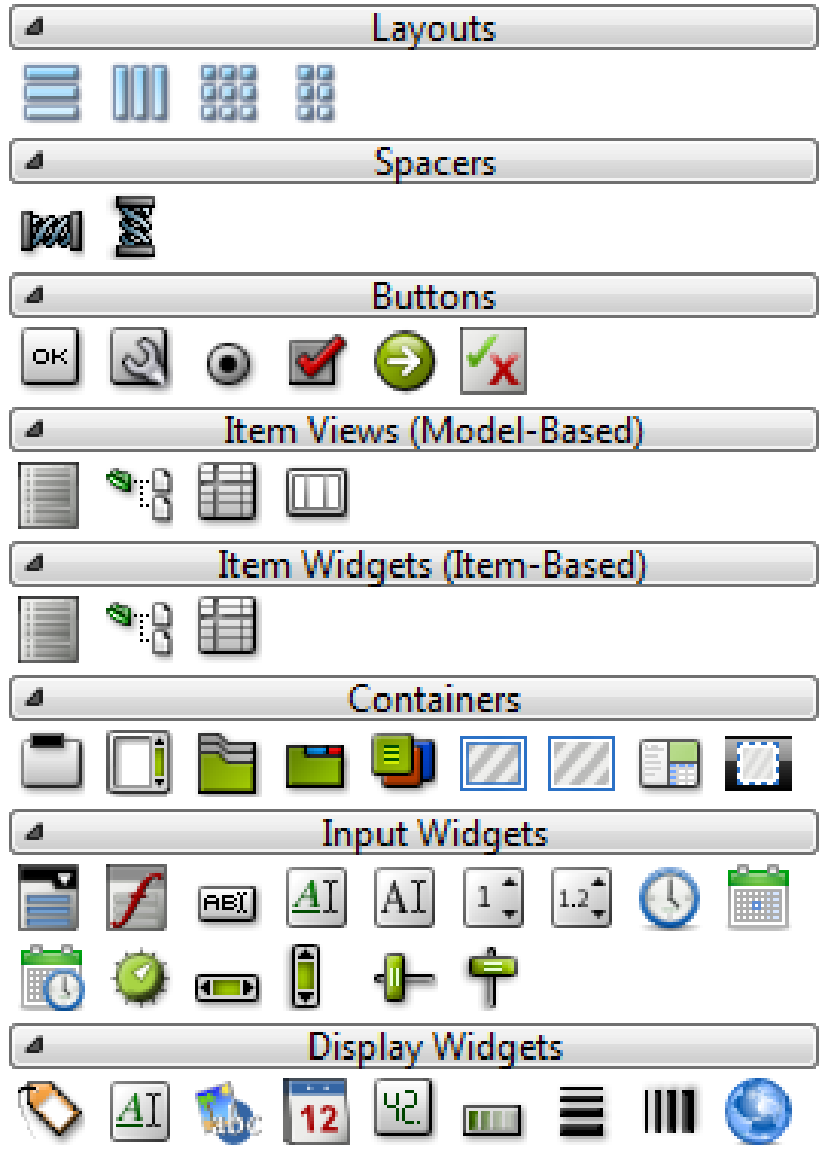


QT Creator – Development tools

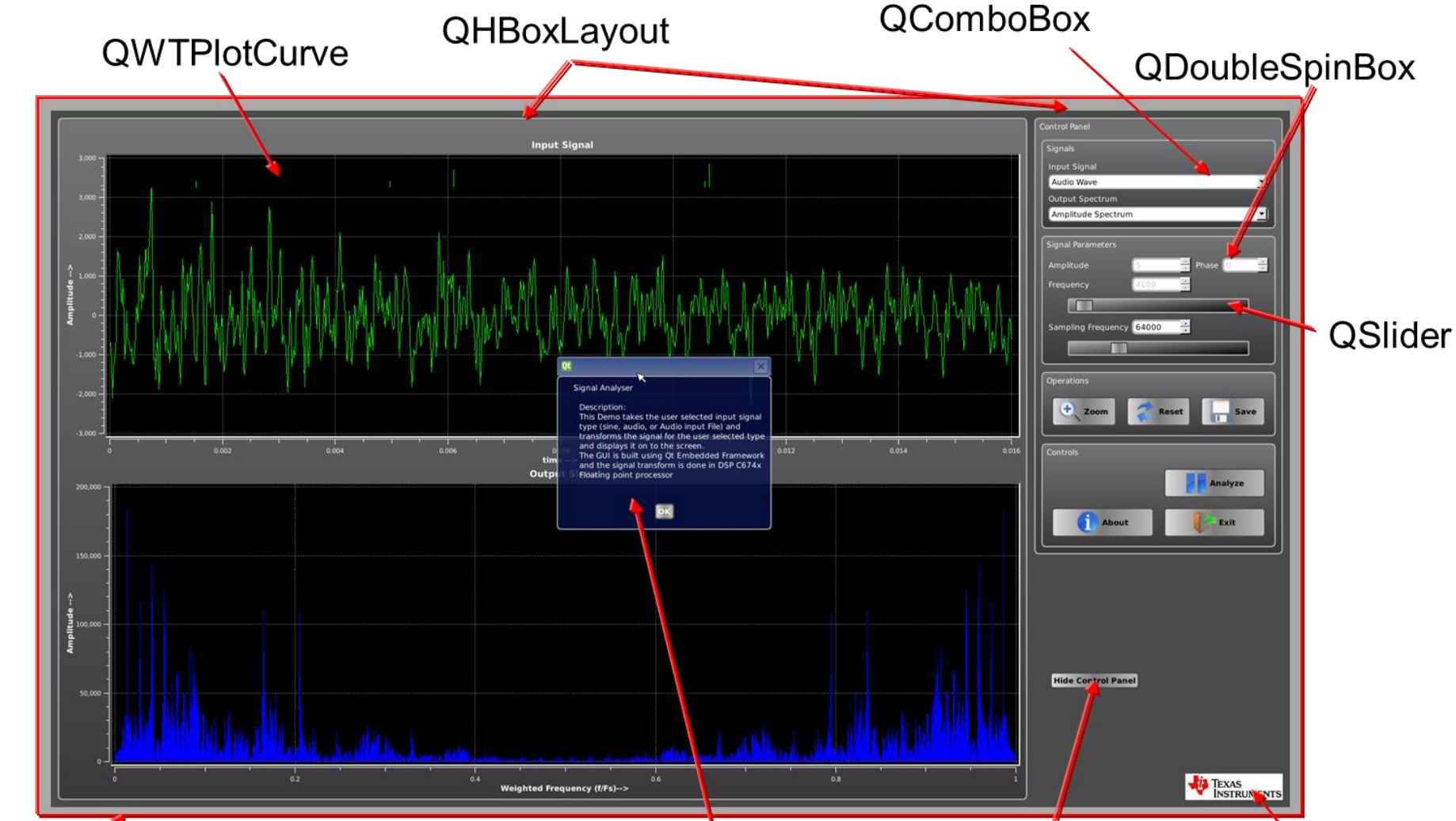


Widgets

- Qt UI framework is based on widgets
- Widgets respond to UI events (key presses/mouse movements), and update their screen area
- Each widget has a parent, that affects its behavior, and is embedded into it
- Most Qt classes are derived from QWidget
  - Ex, QGLWidget, QPushbutton …
    QPushButton * myButton = new QPushButton(…);
    myButton->doSomethingAPI();



Widgets


QWTPlotCurve
QHBoxLayout
QComboBox
QDoubleSpinBox
QSlider
QMainWindow
QMessageBox
QPushButton
Qlabel
Input Signal
Control Panel
Signals
Input Signal
Audio Wave
Output Spectrum
Amplitude Spectrum
Signal Parameters
Amplitude
Phase
Frequency
Sampling Frequency
64000
Operations
Zoom
Reset
Save
Controls
Analyze
About
Exit
Signal Analyser
Description:
This Demo takes the user selected input signal type (sine, audio, or Audio input File) and transforms the signal for the user selected type and displays it on to the screen.
The GUI is built using Qt Embedded Framework and the signal transform is done in DSP C674x Floating point processor
OK
Hide Control Panel
Amplitude -->
Weighted Frequency (f/Fs)-->
Texas Instruments

Signals & Slots

- Signals & Slots
  - Signals and slots are used for communication between objects. The signals and slots mechanism is a central feature of Qt and probably the part that differs most from the features provided by other frameworks.
- Signals
  - Events occur and cause a signal
  - Widgets contain a list of predefined signals, but you can subclass a widget to add your own signal
  - Example – button press, or Process complete
- Slots
  - Slots are the functions which are assigned to handle a signal.
  - Widgets contain a list of predefined slots.
  - You can subclass a widget and add your own slots

Running Supplied Demo Applications

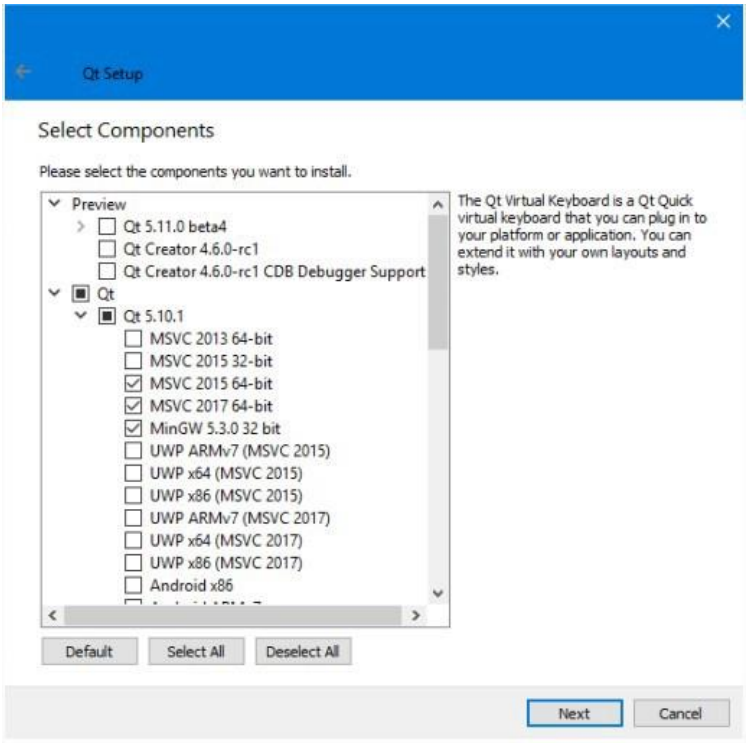
- There are over 300 demo and example applications supplied in the SDK.
  - They come from the QT SDK
  - Wide variety of applications. The same application from QT Demo.
- Use QT Creator to pull in the project and build and run it on the target.

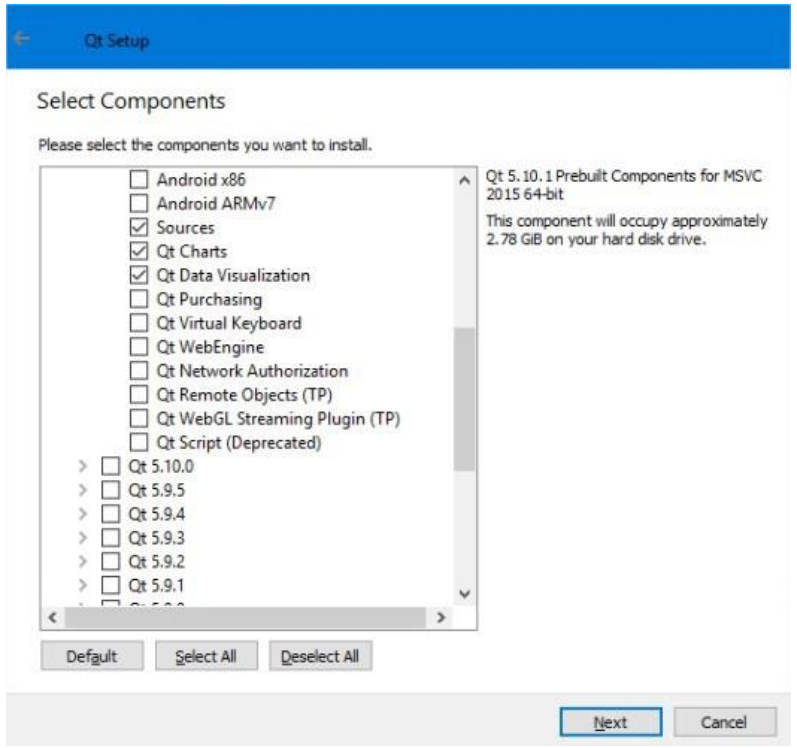
Chapter IV. Graphical User Interface

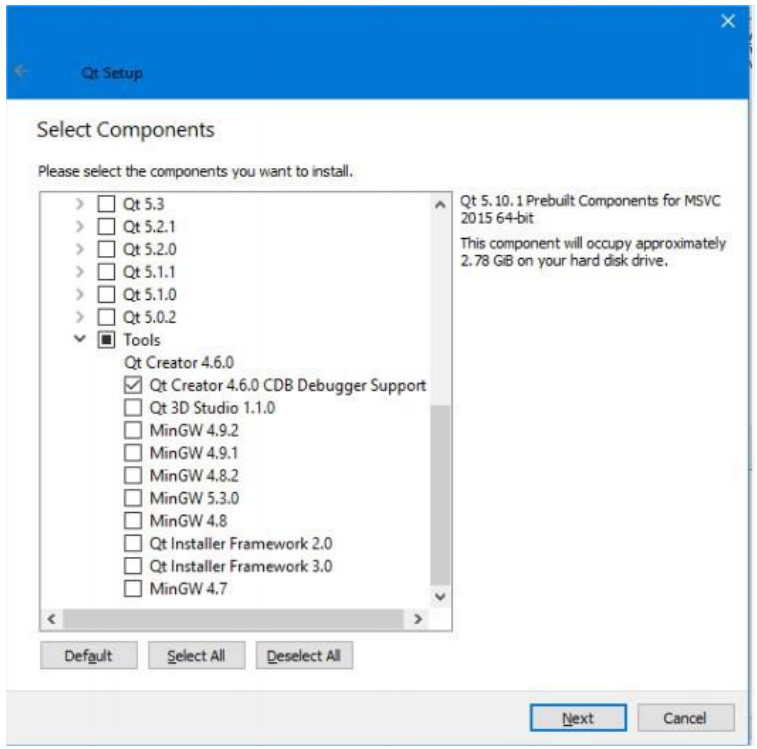
1. What is QT?
2. QT Framework
3. QT in Visual Studio
4. Qt5 C++ GUI Development

QT in Visual Studio

- Download and Install Qt
  - Download the open source version of Qt that is suitable for your operating system https://www.qt.io/download
  - Start the installation: select Tools, Qt Source, Qt chart and also Qt Data Visualization

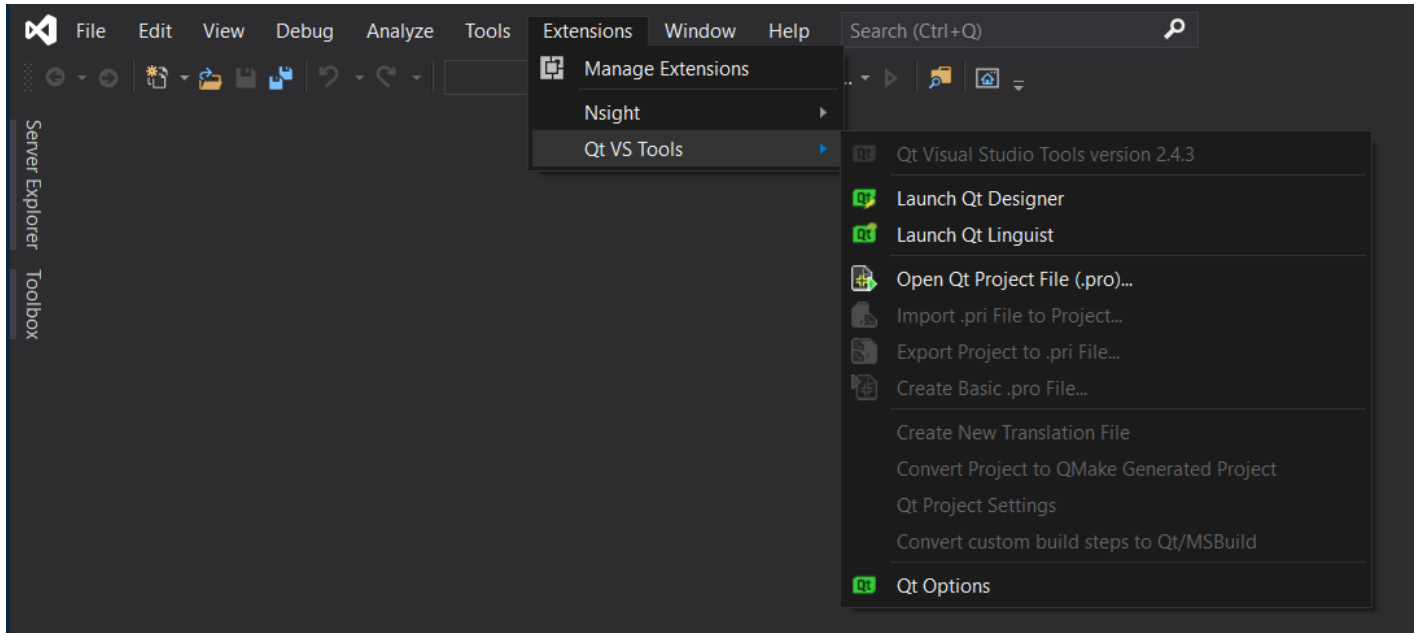






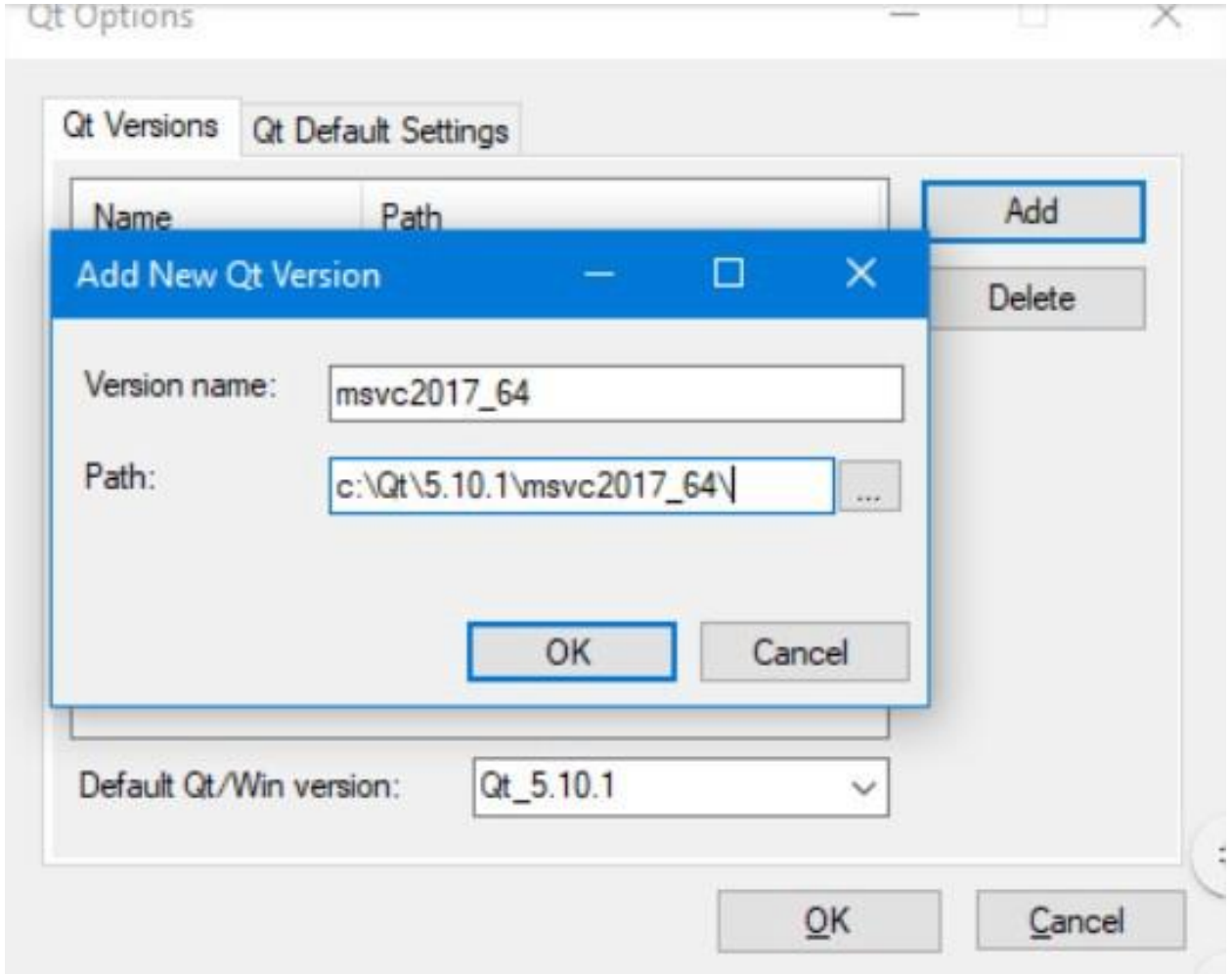
QT in Visual Studio

- Download and Install Qt Visual Studio Tools
  - Download Qt Visual Studio Tools and Install https://marketplace.visualstudio.com/items?itemName=TheQtCompany.QtVisualStudioTools2019
  - Open Visual studio, if the Qt VS Tools were correctly installed, you should find a new menu item “Qt VS Tools”.



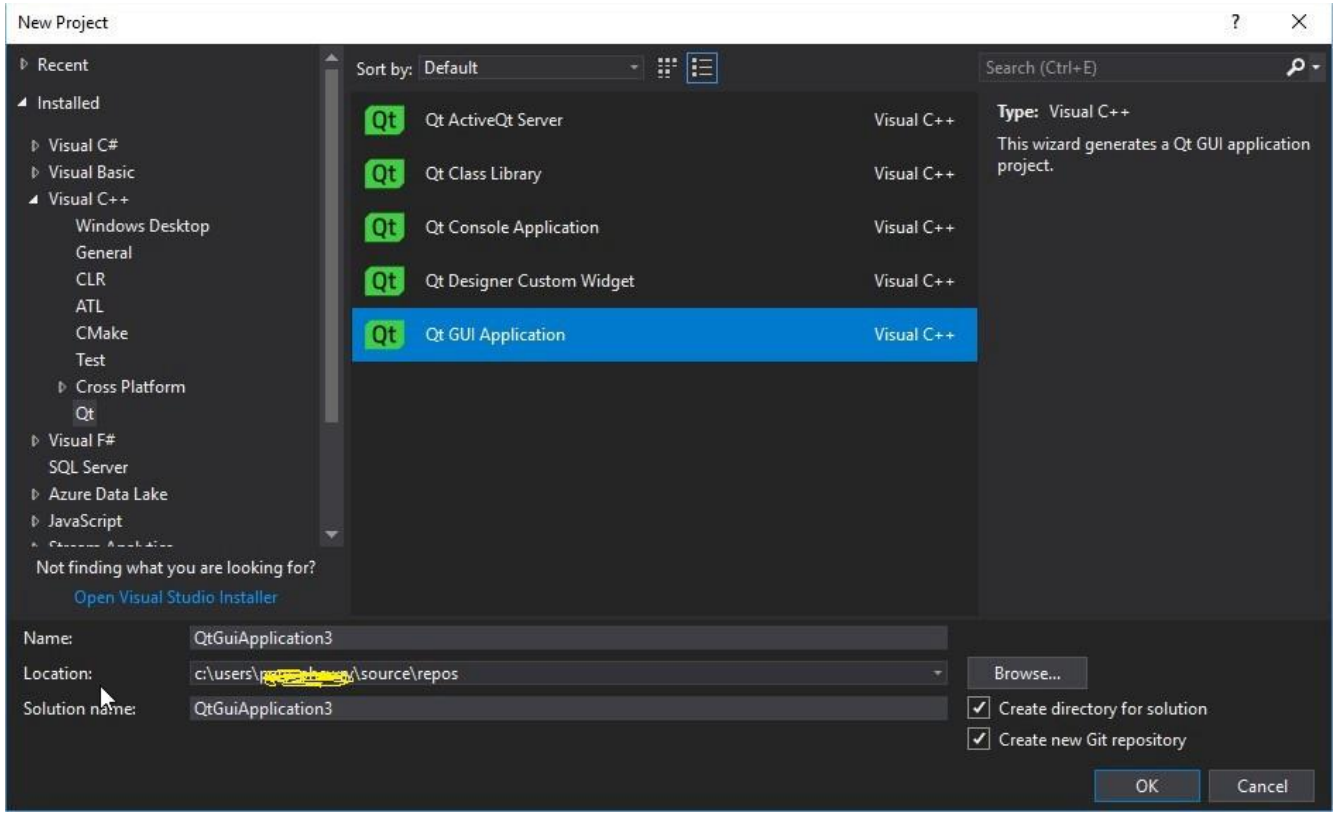
QT in Visual Studio

- Add the path towards the components for the compiler: Qt VS Tools -> Qt Options-> Add button, Use the folder where you installed Qt



QT in Visual Studio

- Creating Qt GUI Application Projects
  - Select New **Project > Installed > Templates > Visual C++ > Qt > Qt GUI Application**
  - In the **Name** field, enter **AddressBook**, and then select **OK**
  - To acknowledge the **Welcome** dialog, select **Next**
  - Select the modules to include in the project, and then select **Next**
  - In the **Base class** field, enter **QWidget** as the base class type



QT in Visual Studio

- Creating Qt GUI Application Projects
  - Select the **Lower case filenames** check box to only use lower case characters in the names of the generated files
  - Select the **Precompiled header** check box to use a precompiled header file
  - Select the **Add default application** icon check box to use a default application icon for the application
  - Select **Finish** to create the project
  - Select **Build > Build Solution** to build it, and then select **Debug > Start Debugging** to run it. For now, the result is an empty window.

Chapter IV. Graphical User Interface

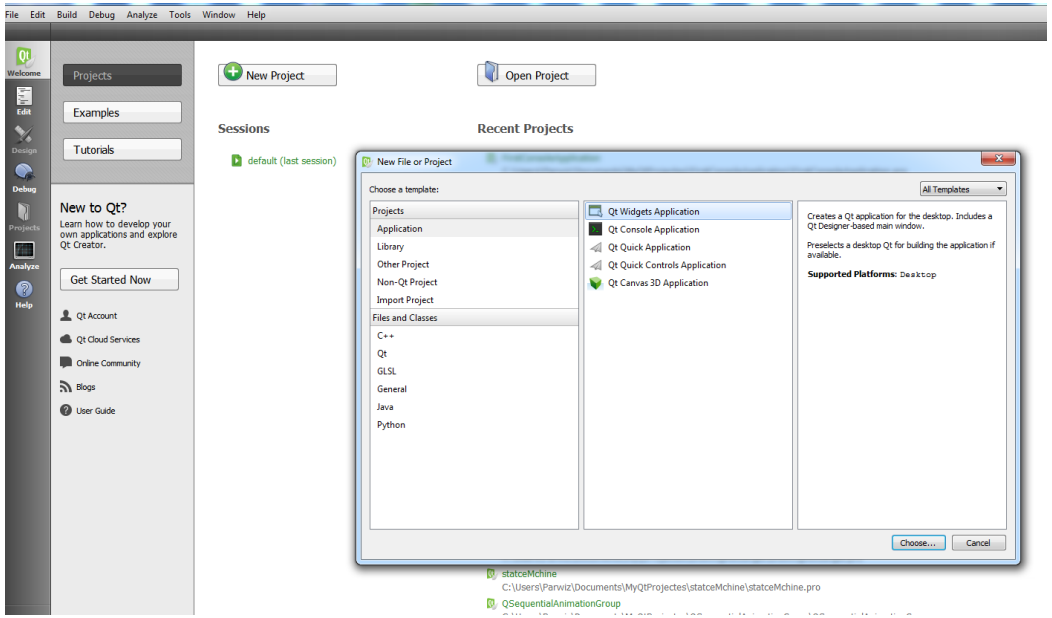
1. What is QT?
2. QT Framework
3. QT in Visual Studio
4. Qt5 C++ GUI Development

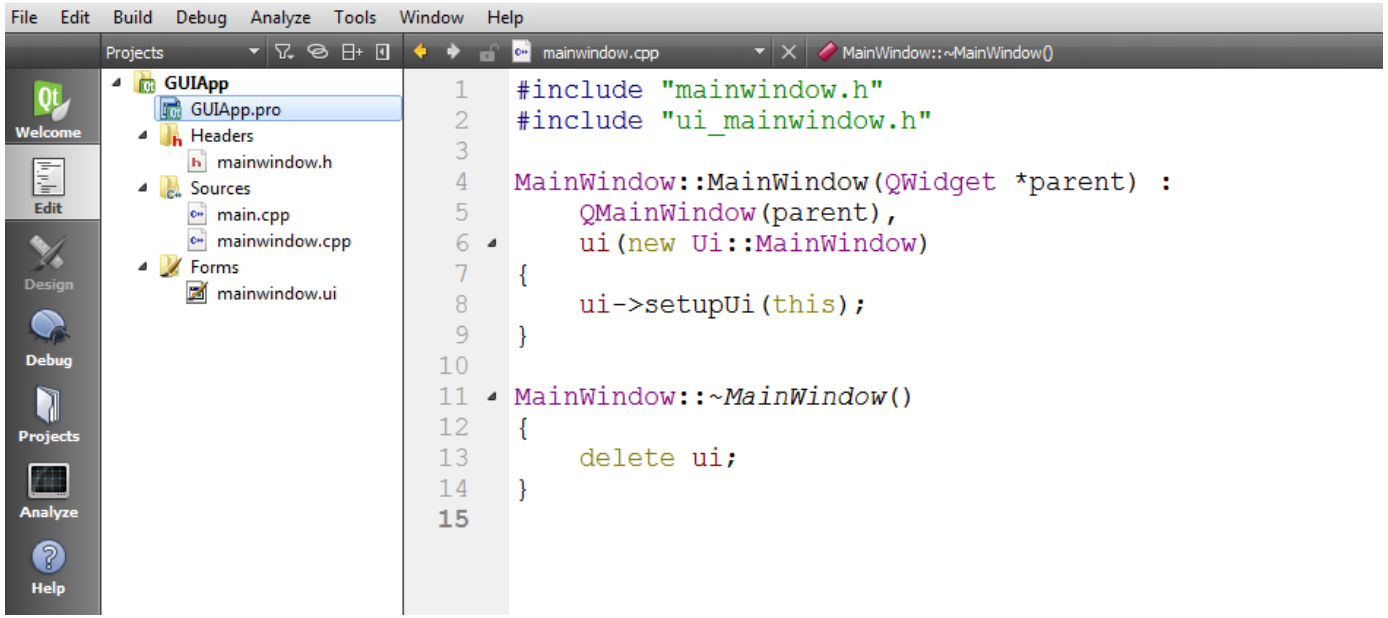
Qt5 C++ GUI Development

- First GUI Application
- Signal And Slots
- Layout Management
- Qt5 Style Sheets
- QPushButton
- QCheckBox
- QRadioButton
- QComboBox
- QListWidget
- QMessageBox
- QMenu And QToolbar
- QFileDialog
- QProgressBar
- Draw Text & Line with QPainter
- Draw Rectangle
- Draw Ellipse

Qt5 C++ GUI Development

- First GUI Application
  - Create a new Project in your Qt5 framework
  - Choose Qt Widget Application
  - Click on next and finish your project

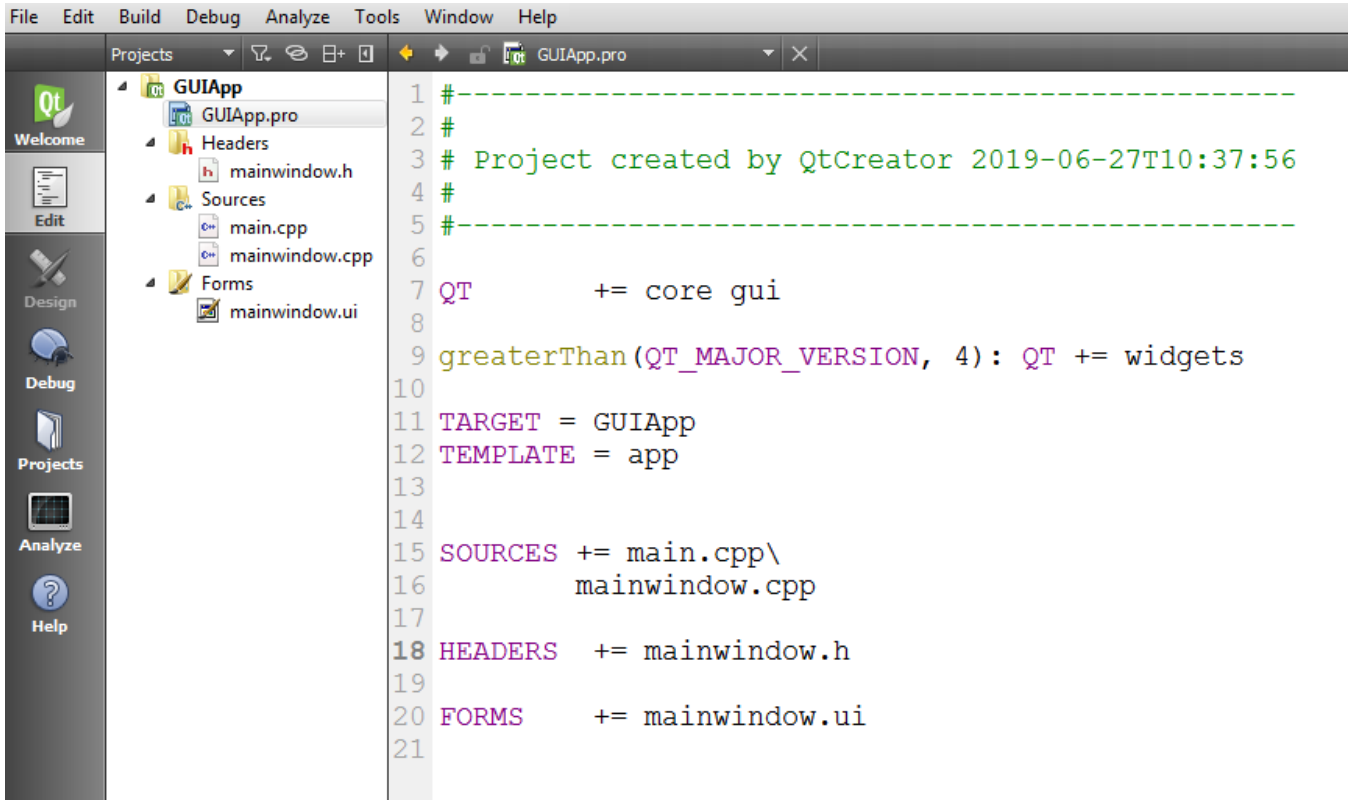




https://codeloop.org/qt5-c-first-gui-application/

Qt5 C++ GUI Development

- First GUI Application
  - .PRO file: PRO files include references to project libraries, assets, and source code files, as well as other files such as application resources (.QRC files), project includes (.PRI files), translation sources (.TS files), phrase books (.QPH files), and style sheets (.QSS files).Qt projects are used for creating applications that run on the Qt framework.



https://codeloop.org/qt5-c-first-gui-application/

- First GUI Application
  - The header file .h:

mainwindow.h

```
1  #ifndef MAINWINDOW_H
2  #define MAINWINDOW_H
3
4  #include <QMainWindow>
5
6  namespace Ui {
7  class MainWindow;
8  }
9
10 class MainWindow : public QMainWindow
11 {
12     Q_OBJECT
13
14 public:
15     explicit MainWindow(QWidget *parent = 0);
16     ~MainWindow();
17
18 private:
19     Ui::MainWindow *ui;
20 };
21
22 #endif // MAINWINDOW_H
```

https://codeloop.org/qt5-c-first-gui-application/

- First GUI Application
  - The .cpp files:

main.cpp

```cpp
#include "mainwindow.h"
#include <QApplication>

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    MainWindow w;
    w.show();

    return a.exec();
}
```

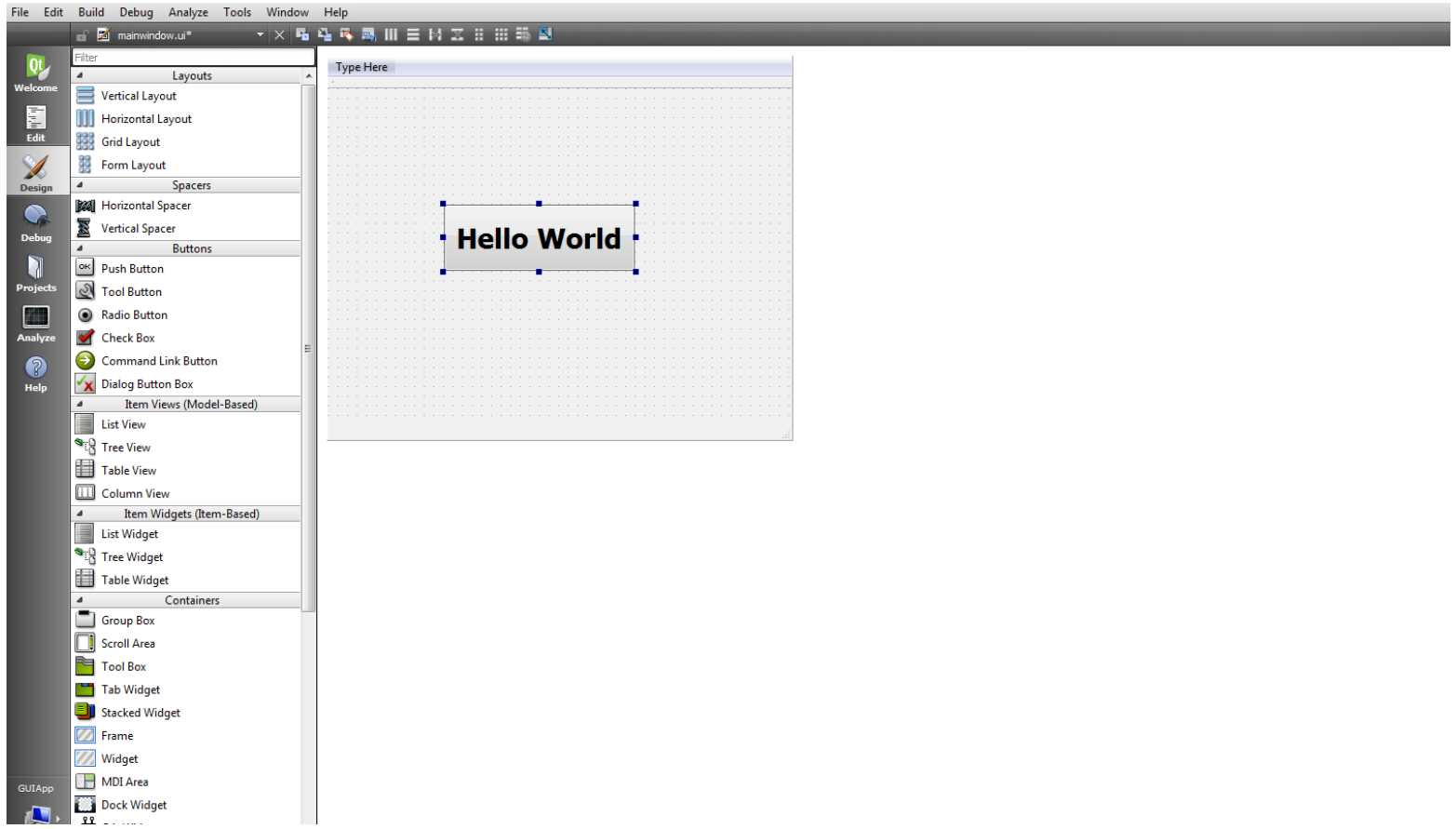
mainwindow.cpp

```cpp
#include "mainwindow.h"
#include "ui_mainwindow.h"

MainWindow::MainWindow(QWidget *parent) :
    QMainWindow(parent),
    ui(new Ui::MainWindow)
{
    ui->setupUi(this);
}

MainWindow::~MainWindow()
{
    delete ui;
}
```

https://codeloop.org/qt5-c-first-gui-application/

Qt5 C++ GUI Development

- First GUI Application
  - The .ui file: we design our GUI application in here



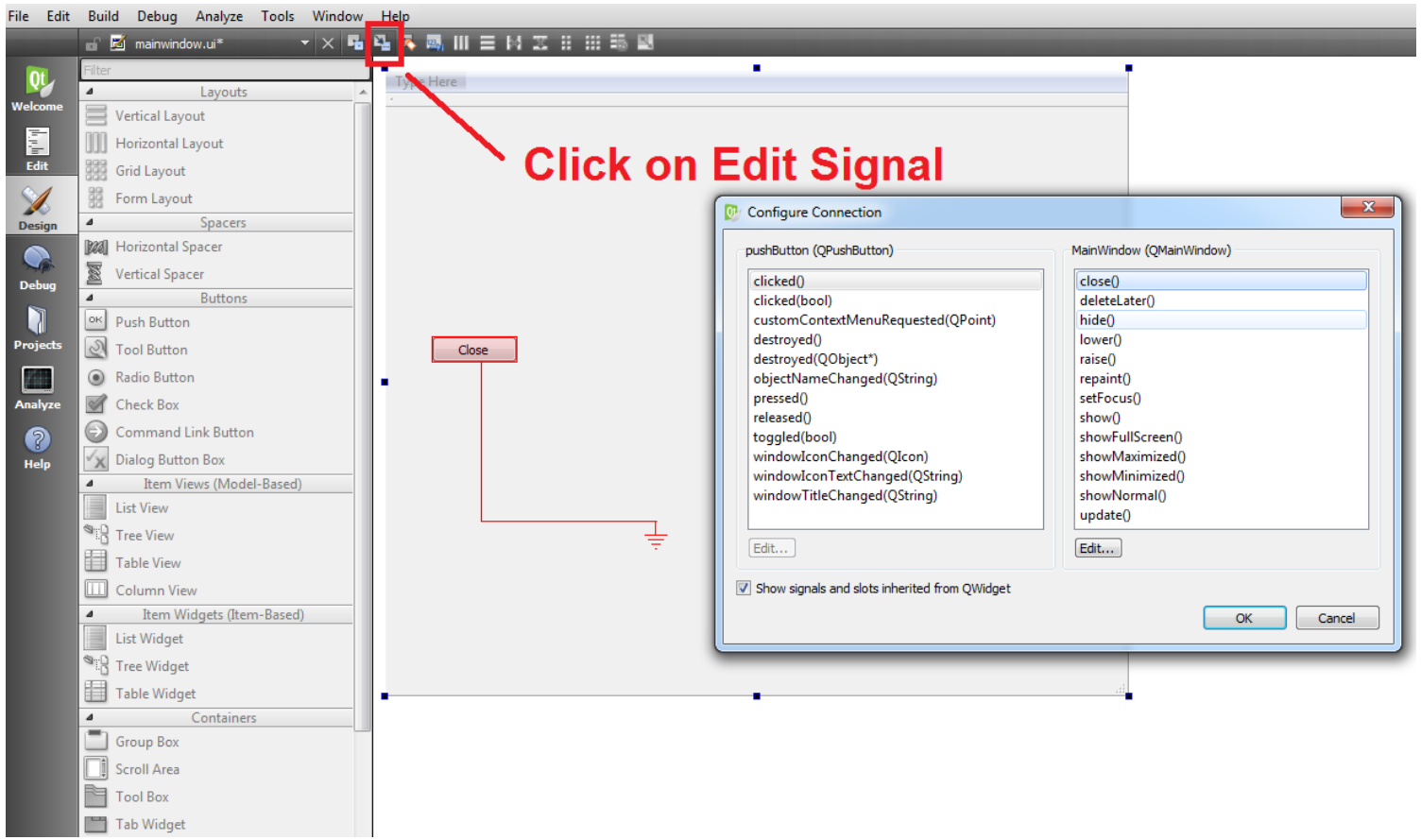
https://codeloop.org/qt5-c-first-gui-application/

- First GUI Application
  - Run the project and this will be the result

https://codeloop.org/qt5-c-first-gui-application/

Qt5 C++ GUI Development

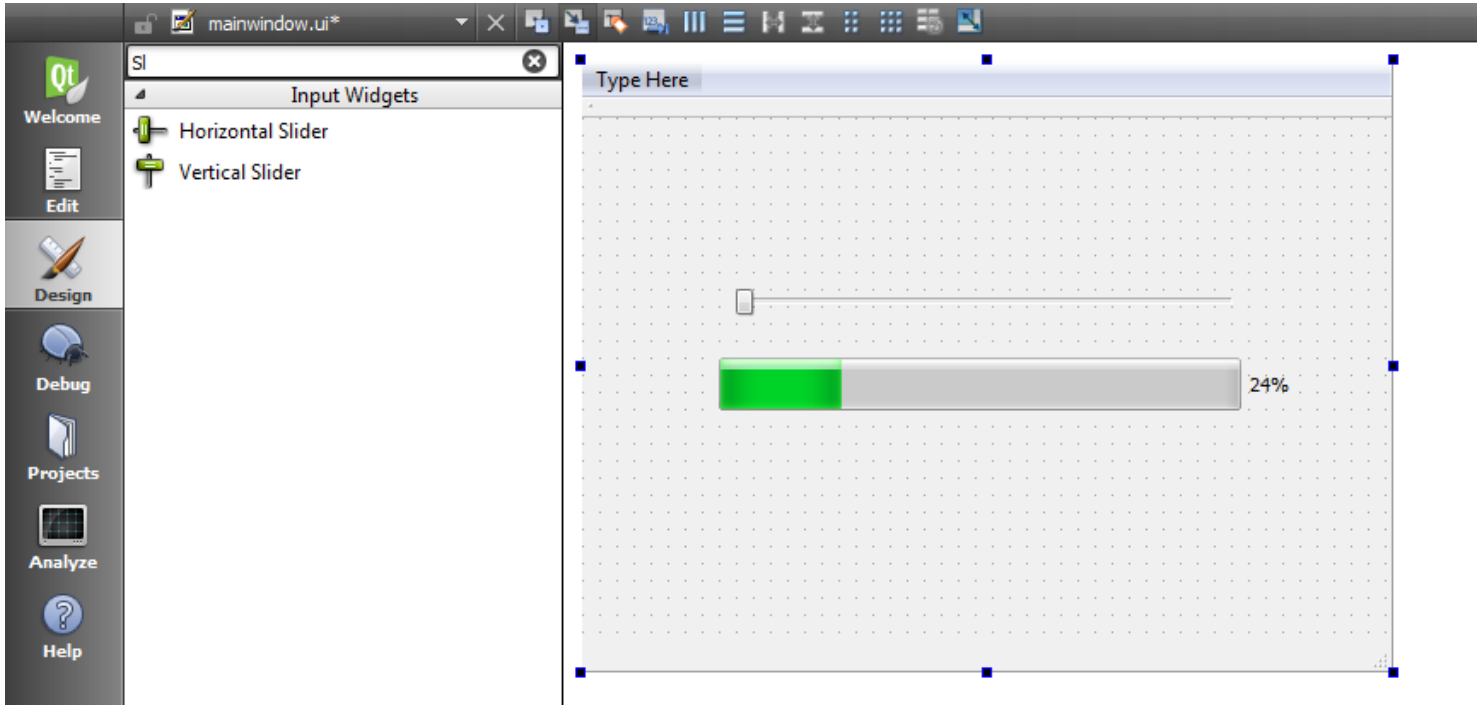
- Signal And Slots
  - Creating New Project in Qt5 C++
  - Open your mainwindow.ui
  - Add a QPushButton



https://codeloop.org/qt5-c-signal-and-slots-introduction/

Qt5 C++ GUI Development

- Signal And Slots
  - Creating New Project in Qt5 C++
  - Open your mainwindow.ui
  - Add a QProgressbar and a Horizontal Slider

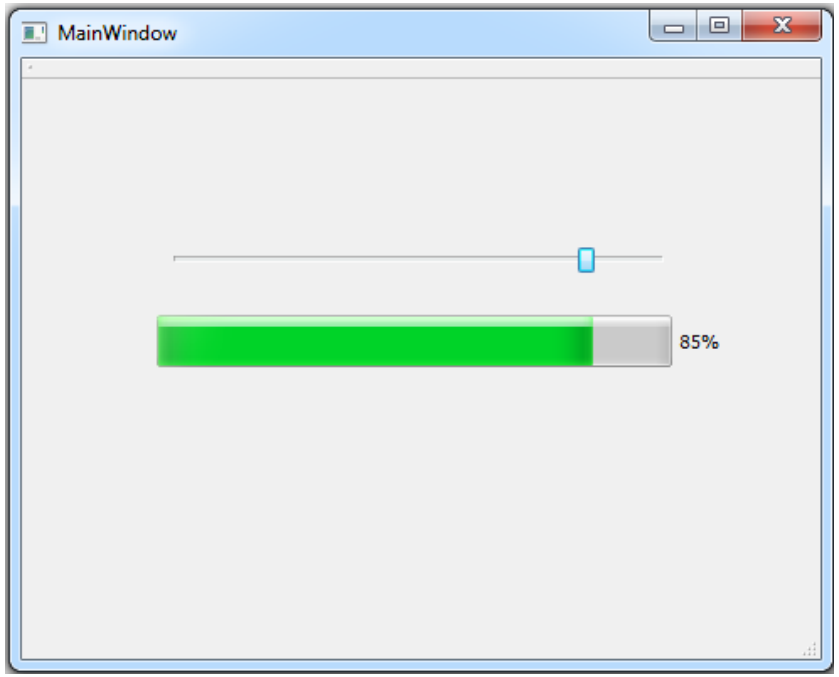


https://codeloop.org/qt5-c-signal-and-slots-introduction/

- Signal And Slots
  - For connecting the signal and slots you need to open your mainwindow.cpp and in the constructor add this line of code

connect(ui->horizontalSlider, SIGNAL(valueChanged(int)), ui->progressBar, SLOT(setValue(int)));

```
1  #include "mainwindow.h"
2  #include "ui_mainwindow.h"
3
4  MainWindow::MainWindow(QWidget *parent) :
5      QMainWindow(parent),
6      ui(new Ui::MainWindow)
7  {
8      ui->setupUi(this);
9
10     connect(ui->horizontalSlider, SIGNAL(valueChanged(int)),
11             ui->progressBar, SLOT(setValue(int)));
12 }
13
14 MainWindow::~MainWindow()
15 {
16     delete ui;
17 }
```



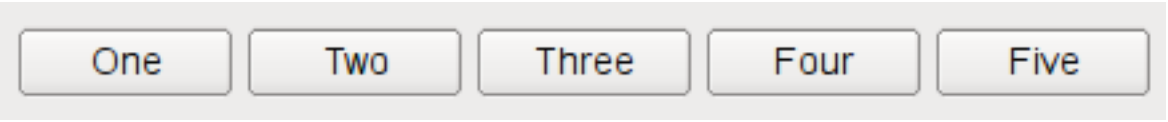
https://codeloop.org/qt5-c-signal-and-slots-introduction/

- Layout Management
  - All QWidget subclasses can use layouts to manage their children. The QWidget::setLayout() function applies a layout to a widget.
  - When a layout is set on a widget in this way, it takes charge of the following tasks:
    - Positioning of child widgets
    - Sensible default sizes for windows
    - Sensible minimum sizes for windows
    - Resize handling
    - Automatic updates when contents change:
      - Font size, text or other contents of child widgets
      - Hiding or showing a child widget
      - Removal of child widgets

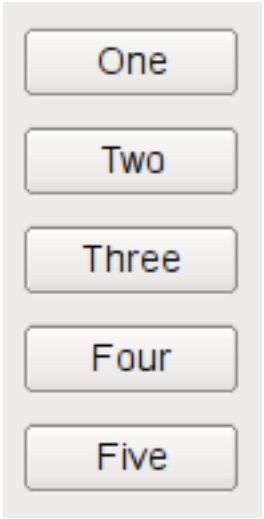
https://codeloop.org/qt5-layout-management-introduction/

- Layout Management
  - The built-in layout managers: QHBoxLayout, QVBoxLayout, QGridLayout, and QFormLayout

QHBoxLayout

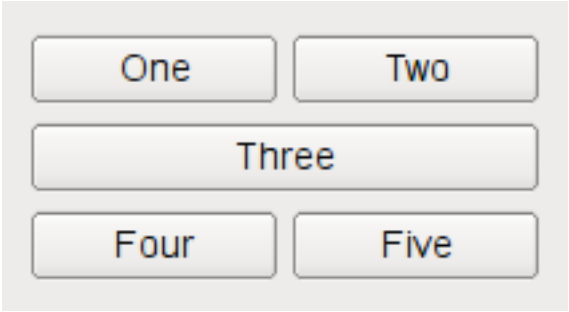


QVBoxLayout



QFormLayout

QGridLayout



https://codeloop.org/qt5-layout-management-introduction/

Qt5 C++ GUI Development

- Qt5 Style Sheets
  - allows you to customize the appearance of widgets
  - subclass Qstyle
  - HTML Cascading Style Sheets (CSS)
  - QApplication::setStyleSheet() and QWidget::setStyleSheet()
  - Example

```
1 QLineEdit { background: yellow }
2 QCheckBox { color: red }
```

https://codeloop.org/qt5-style-sheets-introduction-and-example/

- Qt5 Style Sheets
  - The Style Sheet Syntax And Rules
    - HTML CSS
    - A *style rule* is made up of a selector and a declaration.
    - The *selector* specifies which widgets are affected by the rule
    - The *declaration* specifies which properties should be set on the widget

```
1 QPushButton { color: red }
```

selector *declaration*

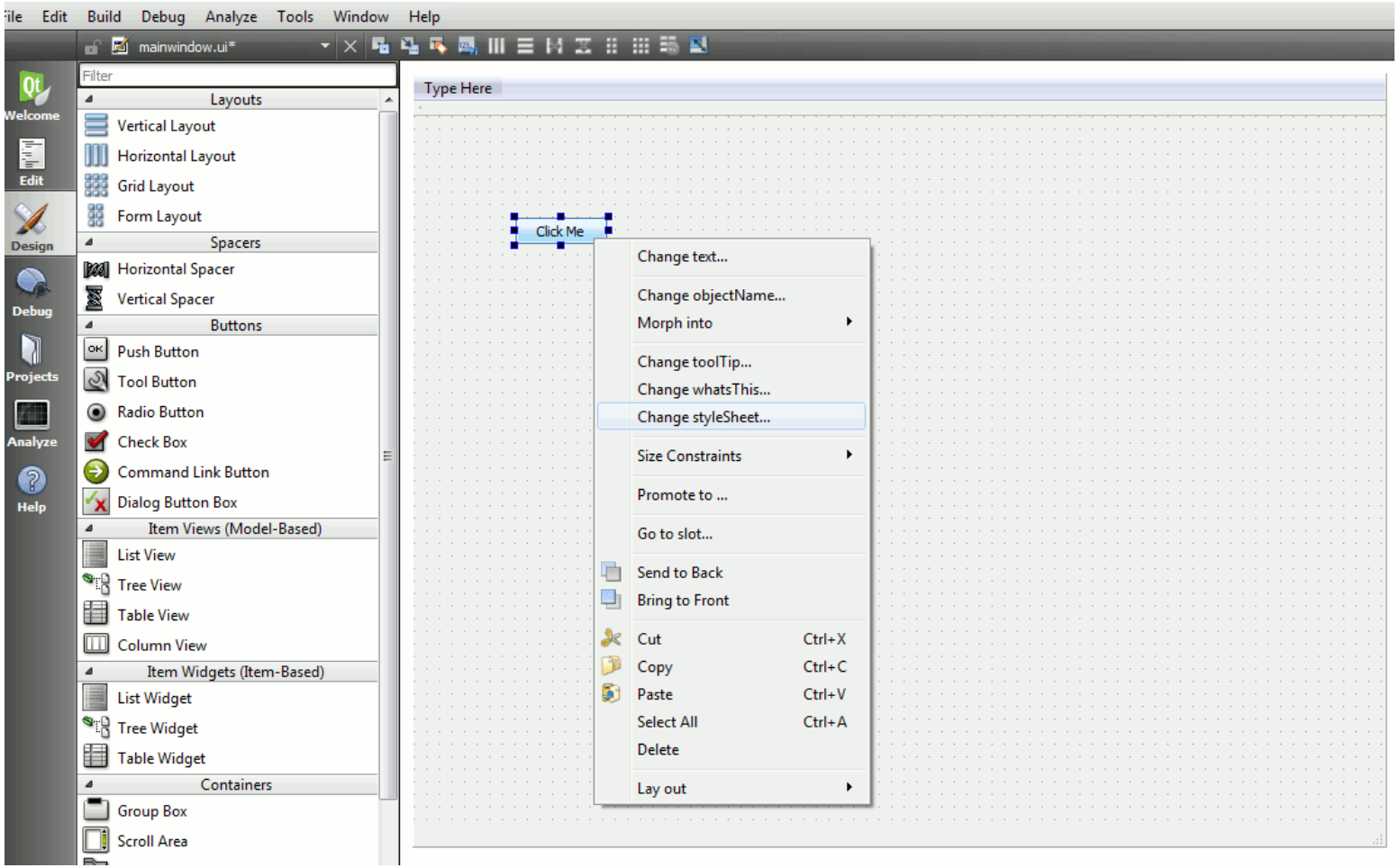
```
1 QPushButton, QLineEdit, QComboBox { color: red }
```

```
1 QPushButton { color: red }
2 QLineEdit { color: red }
3 QComboBox { color: red }
```

https://codeloop.org/qt5-style-sheets-introduction-and-example/

Qt5 C++ GUI Development

- Qt5 Style Sheets
  - Example

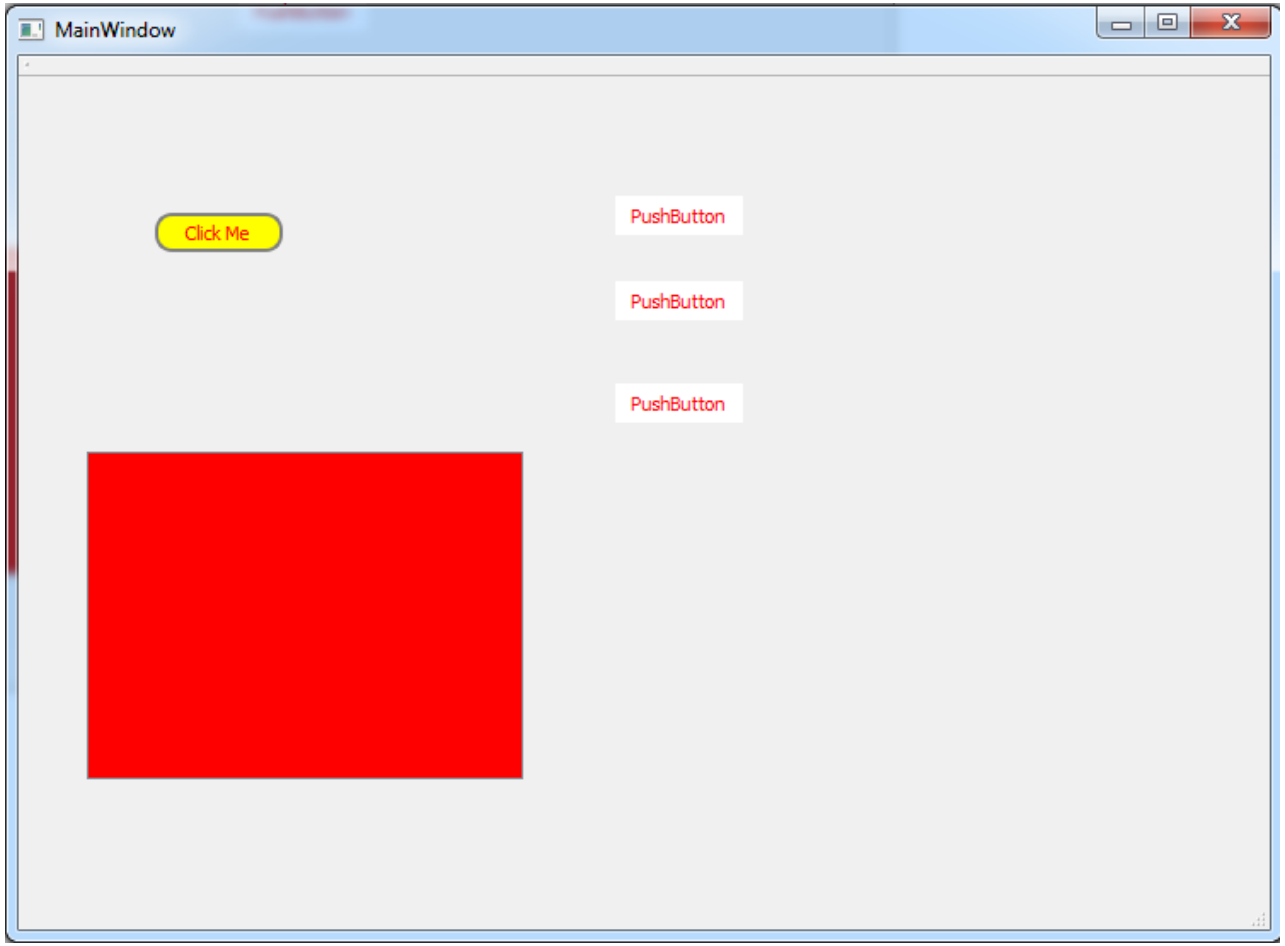


https://codeloop.org/qt5-style-sheets-introduction-and-example/

Qt5 C++ GUI Development

- Qt5 Style Sheets
  - Example

```
QPushButton#pushButton {

border: 2px solid gray;
border-radius: 10px;
padding: 0 8px;
background: yellow;


}


QPushButton {

border: 0px;
padding: 0 8px;
background: white;
color:red;


}


QListWidget {

background:red;


}
```



https://codeloop.org/qt5-style-sheets-introduction-and-example/

- QPushButton
  - The most commonly used widget in any graphical user interface

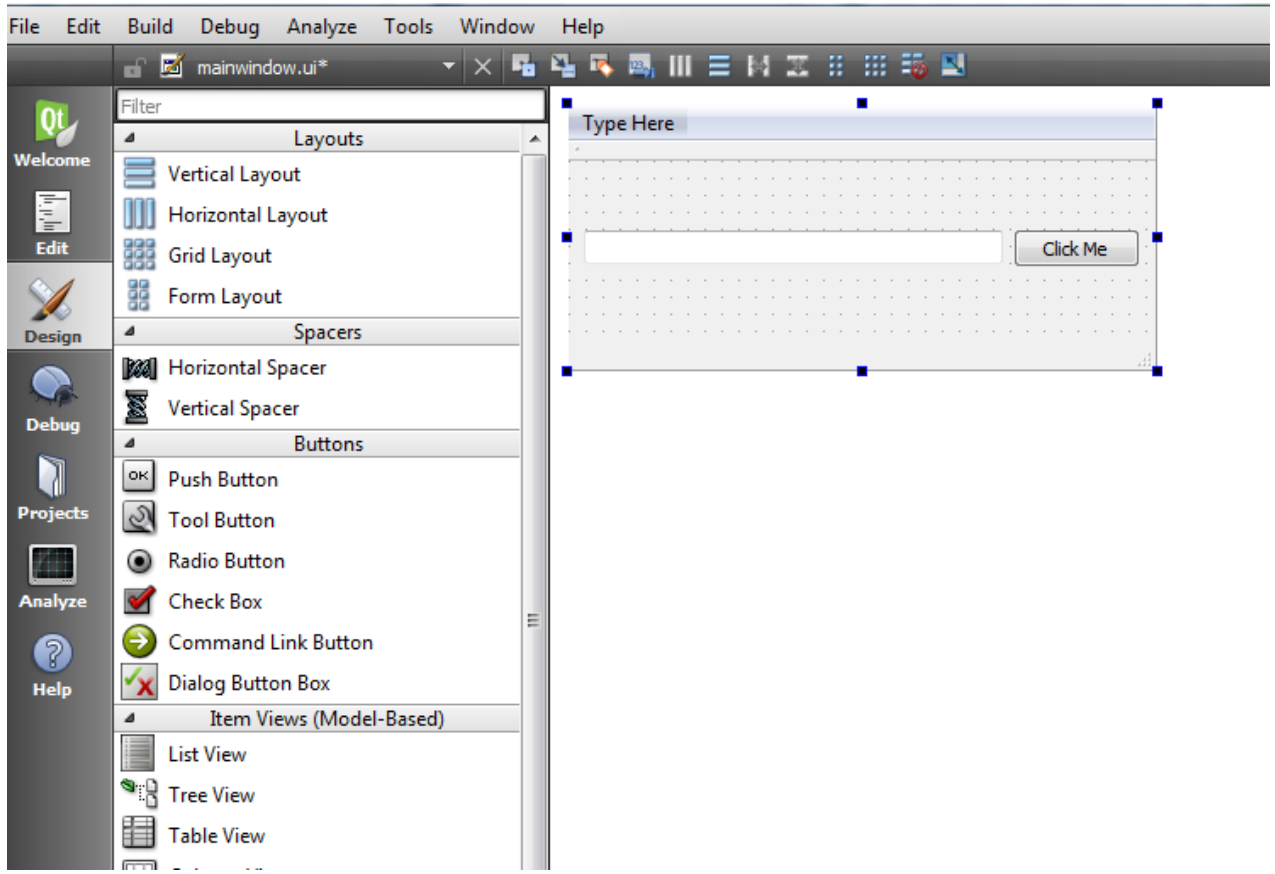
```
1 QPushButton *button = new QPushButton("&Download", this);
```

  - A push button emits the signal clicked() when it is activated by the mouse, the Spacebar or by a keyboard shortcut.
  - Push buttons also provide less commonly used signals, for example pressed() and released().

https://codeloop.org/qt5-qpushbutton-with-signal-and-slots/

Qt5 C++ GUI Development

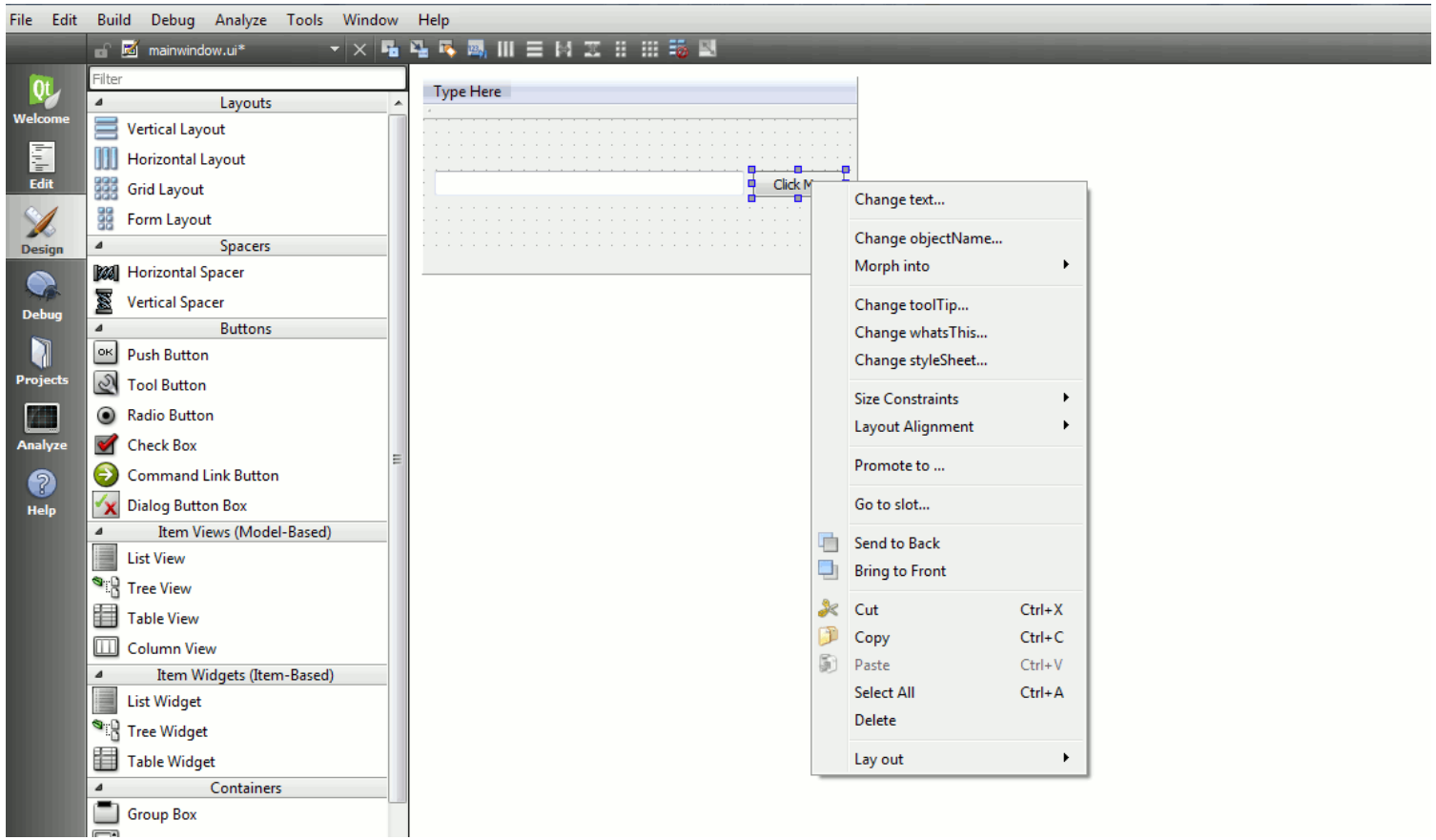
- QPushButton
  - Example



https://codeloop.org/qt5-qpushbutton-with-signal-and-slots/

Qt5 C++ GUI Development

- QPushButton
  - Example



https://codeloop.org/qt5-qpushbutton-with-signal-and-slots/

Qt5 C++ GUI Development

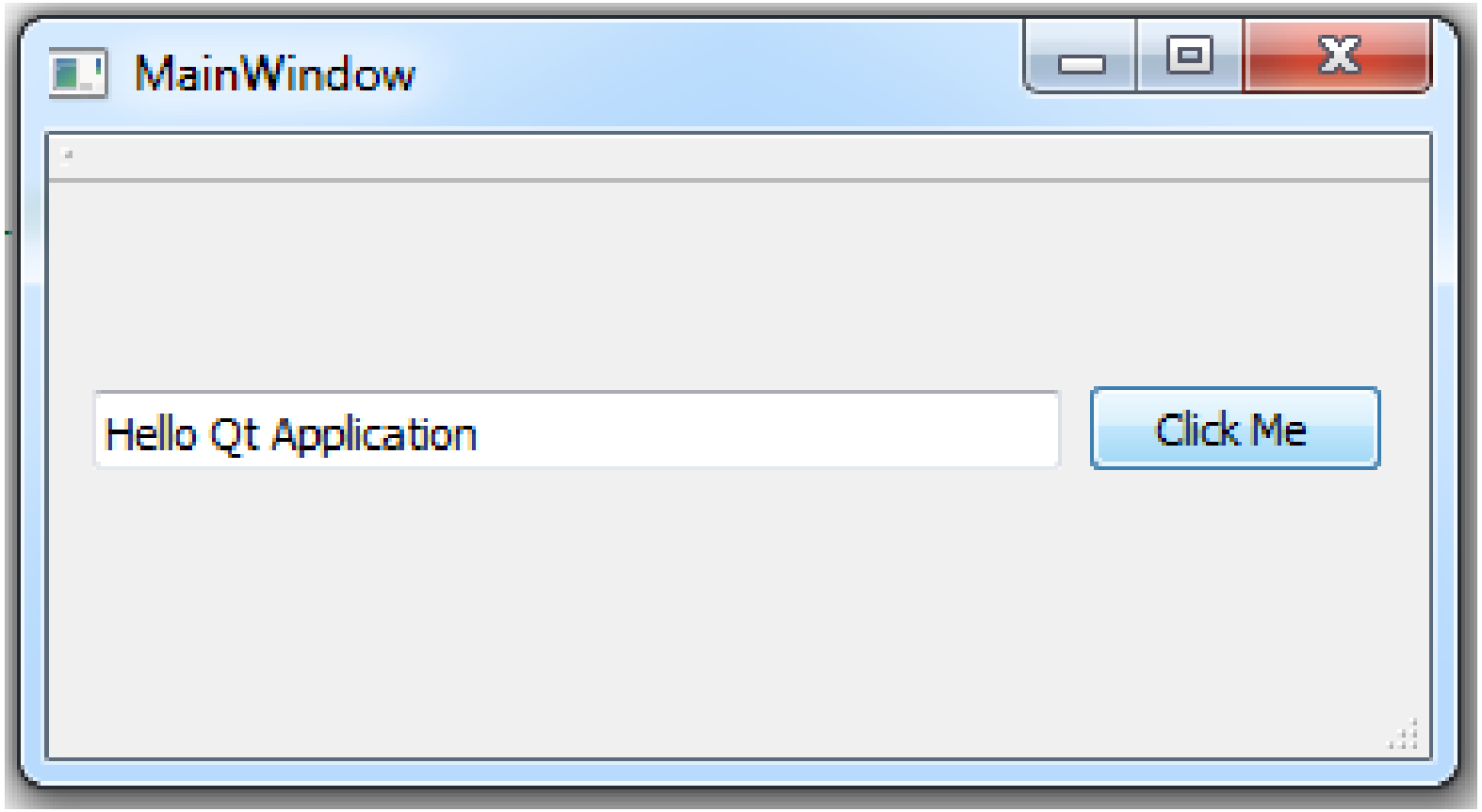
- QPushButton
  - Example

mainwindow.cpp

```
1  #include "mainwindow.h"
2  #include "ui_mainwindow.h"
3
4  MainWindow::MainWindow(QWidget *parent) :
5      QMainWindow(parent),
6      ui(new Ui::MainWindow)
7  {
8      ui->setupUi(this);
9  }
10
11 MainWindow::~MainWindow()
12 {
13     delete ui;
14 }
15
16 void MainWindow::on_pushButton_clicked()
17 {
18
19     ui->lineEdit->setText("Hello Qt Application");
20
21 }
```

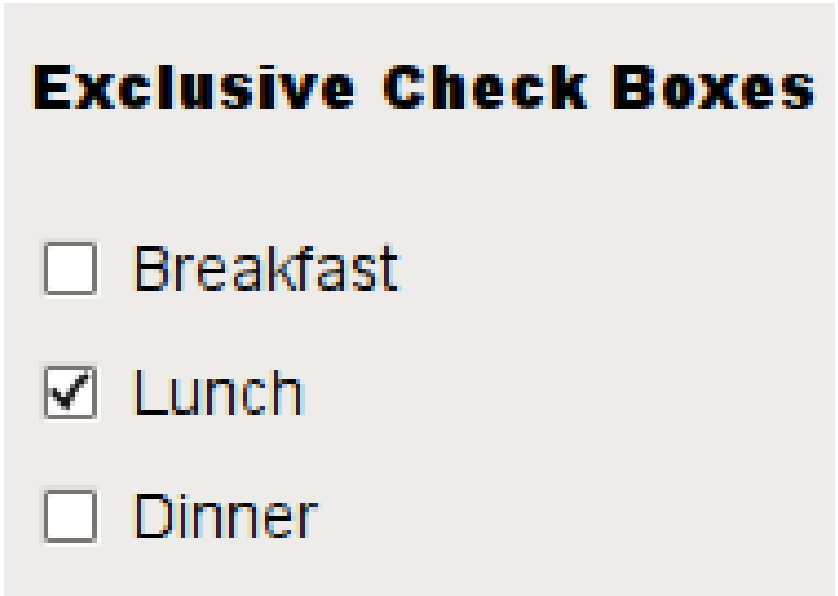
https://codeloop.org/qt5-qpushbutton-with-signal-and-slots/

- QPushButton
  - Example

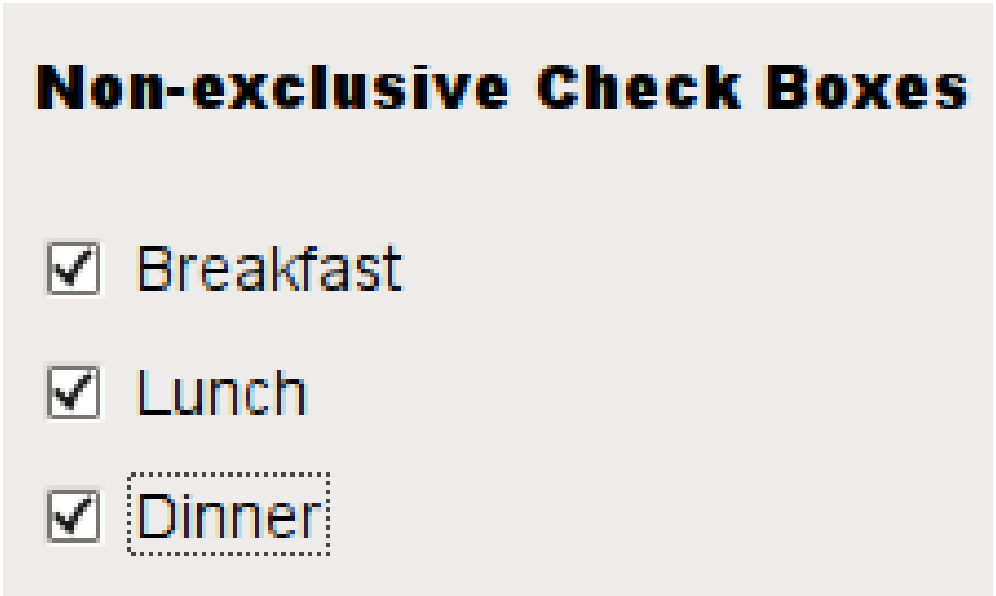


https://codeloop.org/qt5-qpushbutton-with-signal-and-slots/

- QCheckBox
  - A QCheckBox is an option button that can be switched on (checked) or off (unchecked).
  - Signal: stateChanged()
  - Slot: isChecked()



Checkboxes Exclusive



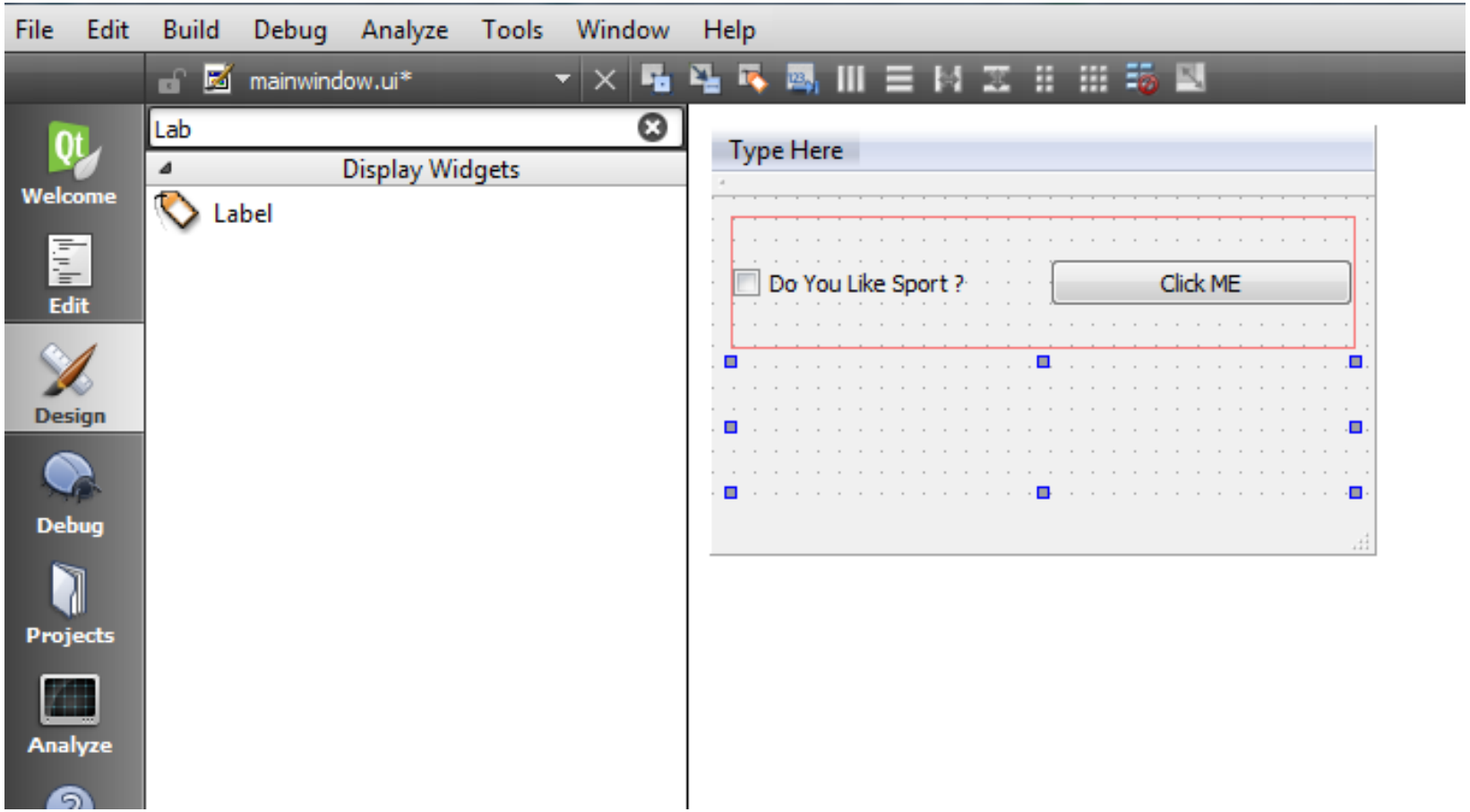
Checkboxes Non Exclusive

https://codeloop.org/how-to-create-qcheckbox-in-qt5-gui/

Qt5 C++ GUI Development

- QCheckBox
  - Example

mainwindow.ui



https://codeloop.org/how-to-create-qcheckbox-in-qt5-gui/

Qt5 C++ GUI Development

- QCheckBox
  - Example

mainwindow.cpp

mainwindow.cpp

```
1  #include "mainwindow.h"
2  #include "ui_mainwindow.h"
3
4  MainWindow::MainWindow(QWidget *parent) :
5      QMainWindow(parent),
6      ui(new Ui::MainWindow)
7  {
8      ui->setupUi(this);
9
10     ui->checkBox->setChecked(true);
11
12 }
13
14 MainWindow::~MainWindow()
15 {
16     delete ui;
17 }
```

https://codeloop.org/how-to-create-qcheckbox-in-qt5-gui/

Qt5 C++ GUI Development

- QCheckBox
  - Example

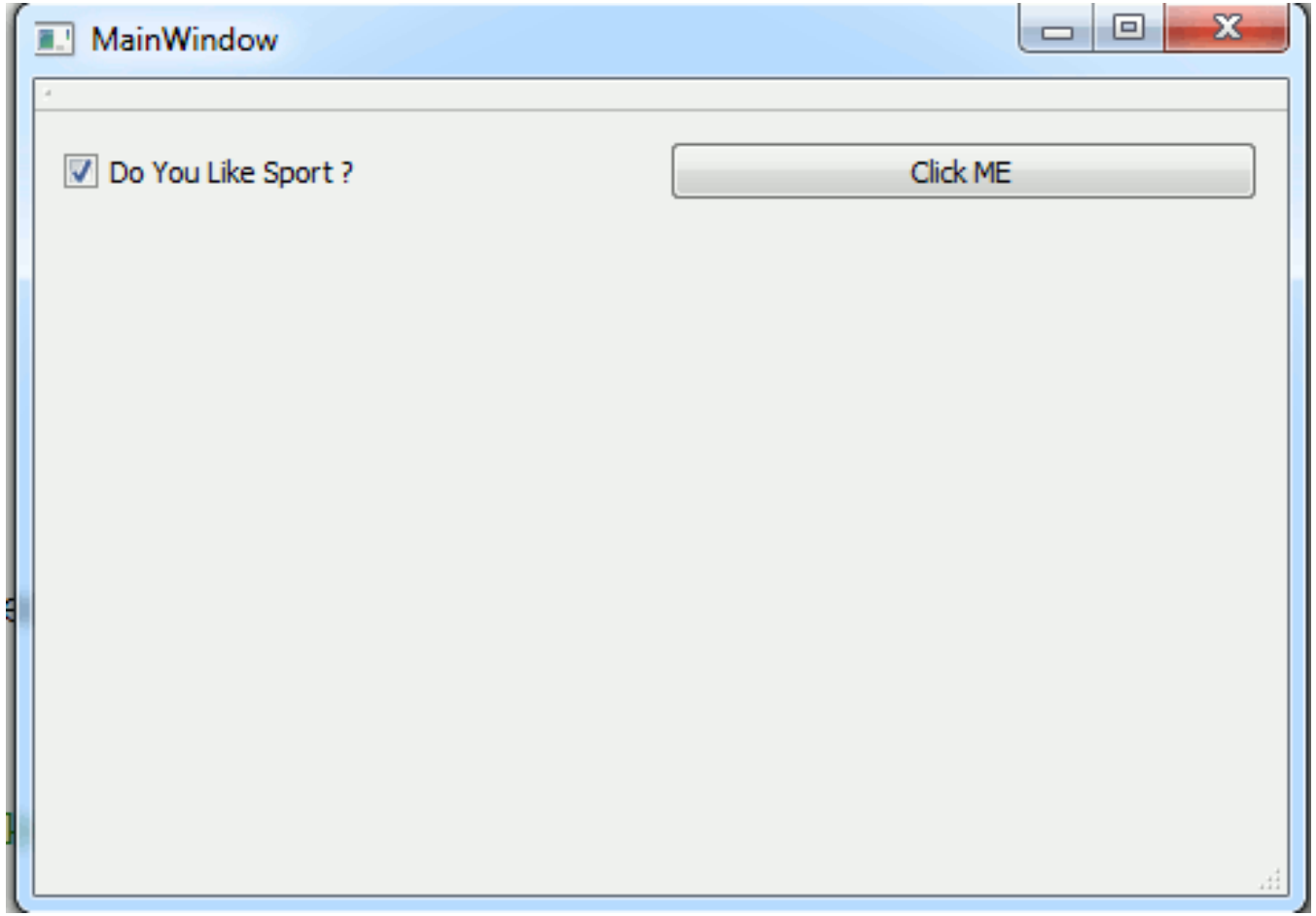
mainwindow.cpp

mainwindow.cpp

```cpp
#include "mainwindow.h"
#include "ui_mainwindow.h"

MainWindow::MainWindow(QWidget *parent) :
    QMainWindow(parent),
    ui(new Ui::MainWindow)
{
    ui->setupUi(this);

    ui->checkBox->setChecked(true);

}

MainWindow::~MainWindow()
{
    delete ui;
}

void MainWindow::on_pushButton_clicked()
{

    if(ui->checkBox->isChecked()) {
        ui->label->setText("Yes I Like Sport");


    }else {


        ui->label->setText("No I Dont Like Sport");

    }

}
```

https://codeloop.org/how-to-create-qcheckbox-in-qt5-gui/

- QCheckBox
  - Example

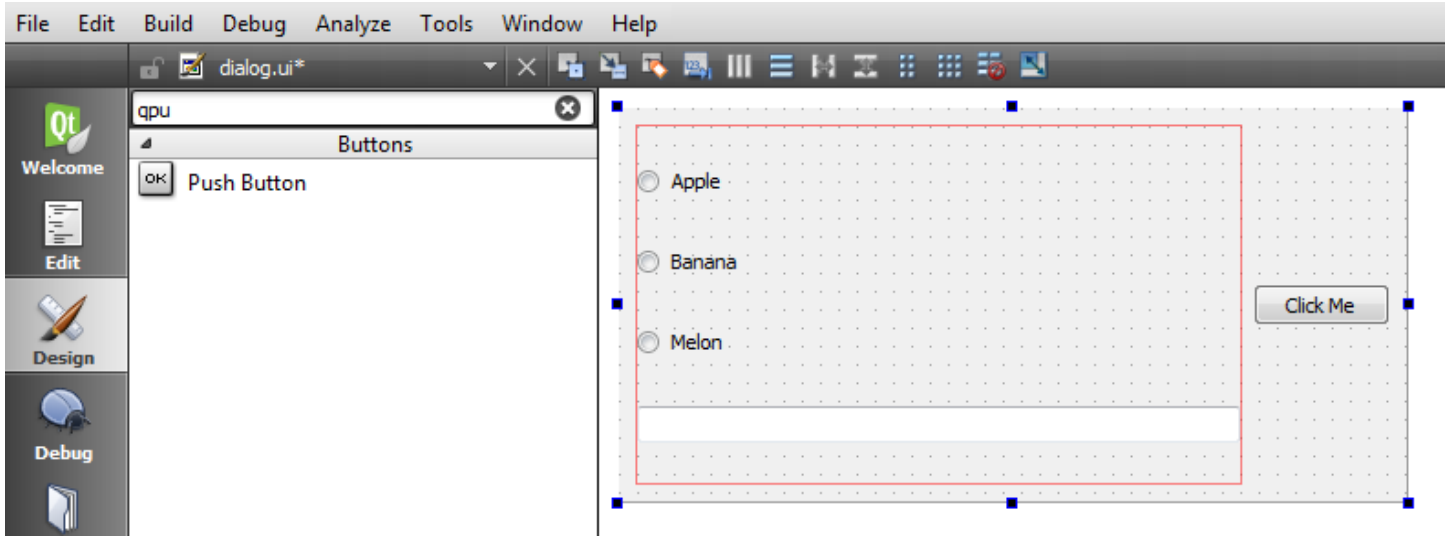


https://codeloop.org/how-to-create-qcheckbox-in-qt5-gui/

- QRadioButton
  - A QRadioButton is an option button that can be switched on (checked) or off (unchecked).
  - Radio buttons typically present the user with a “one of many” choice.

```
1 QRadioButton *button = new QRadioButton("Search from the &cursor", this);
```

  - Example



https://codeloop.org/qt5-gui-how-to-create-qradiobutton/

Qt5 C++ GUI Development

- QRadioButton
  - Example

dialog.cpp

```
1  #include "dialog.h"
2  #include "ui_dialog.h"
3
4  Dialog::Dialog(QWidget *parent) :
5      QDialog(parent),
6      ui(new Ui::Dialog)
7  {
8      ui->setupUi(this);
9
10     ui->radioButton->setChecked(true);
11 }
12
13 Dialog::~Dialog()
14 {
15     delete ui;
16 }
```

https://codeloop.org/qt5-gui-how-to-create-qradiobutton/

Qt5 C++ GUI Development

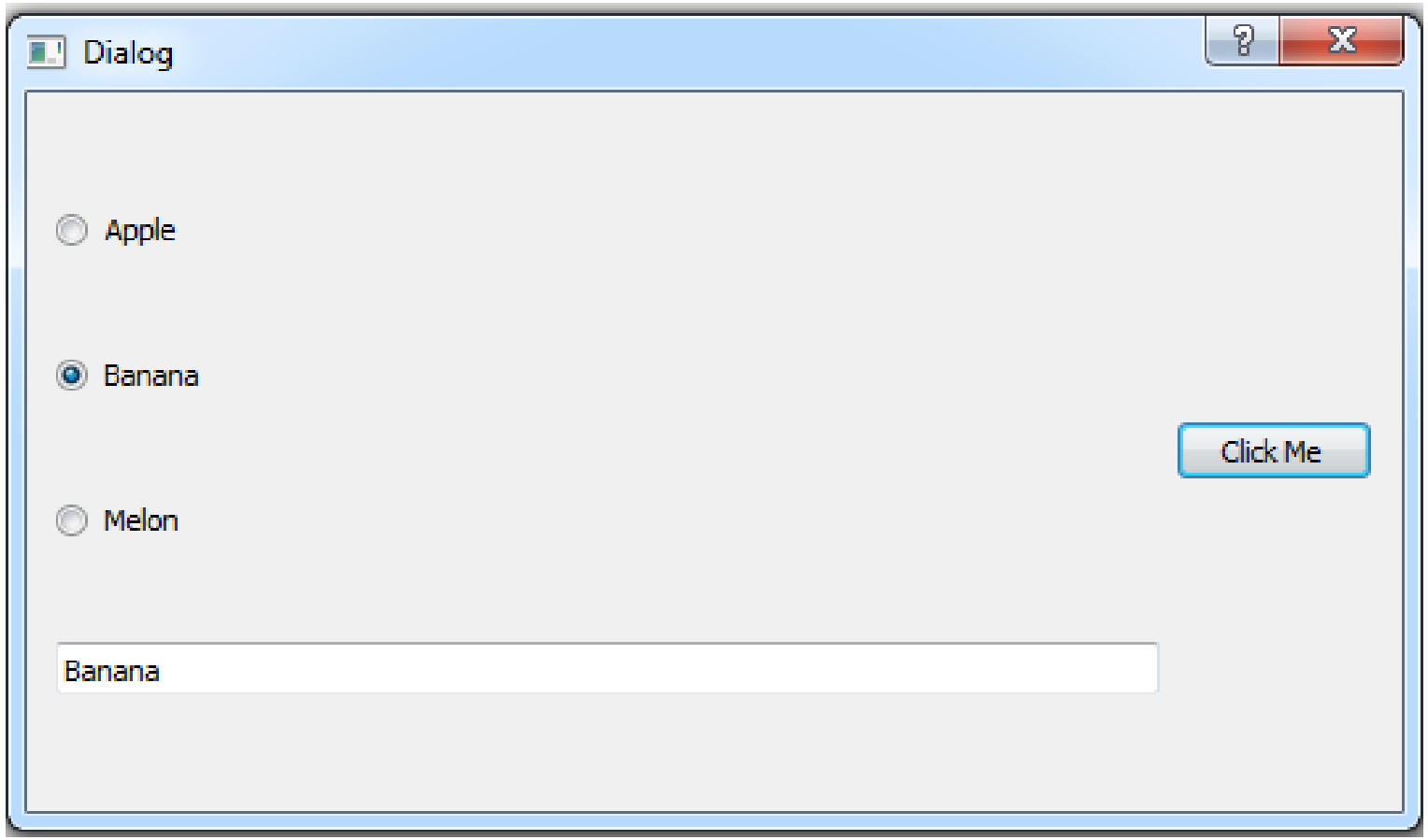
- QRadioButton
  - Example

dialog.cpp

```cpp
#include "dialog.h"
#include "ui_dialog.h"

Dialog::Dialog(QWidget *parent) :
    QDialog(parent),
    ui(new Ui::Dialog)
{
    ui->setupUi(this);

    ui->radioButton->setChecked(true);
}

Dialog::~Dialog()
{
    delete ui;
}

void Dialog::on_pushButton_clicked()
{

    if(ui->radioButton->isChecked()) {

        ui->lineEdit->setText(ui->radioButton->text());

    }

    else if(ui->radioButton_2->isChecked()) {

        ui->lineEdit->setText(ui->radioButton_2->text());
    }

    else if(ui->radioButton_3->isChecked()){
       ui->lineEdit->setText(ui->radioButton_3->text());
    }

}
```

https://codeloop.org/qt5-gui-how-to-create-qradiobutton/

- QRadioButton
  - Example



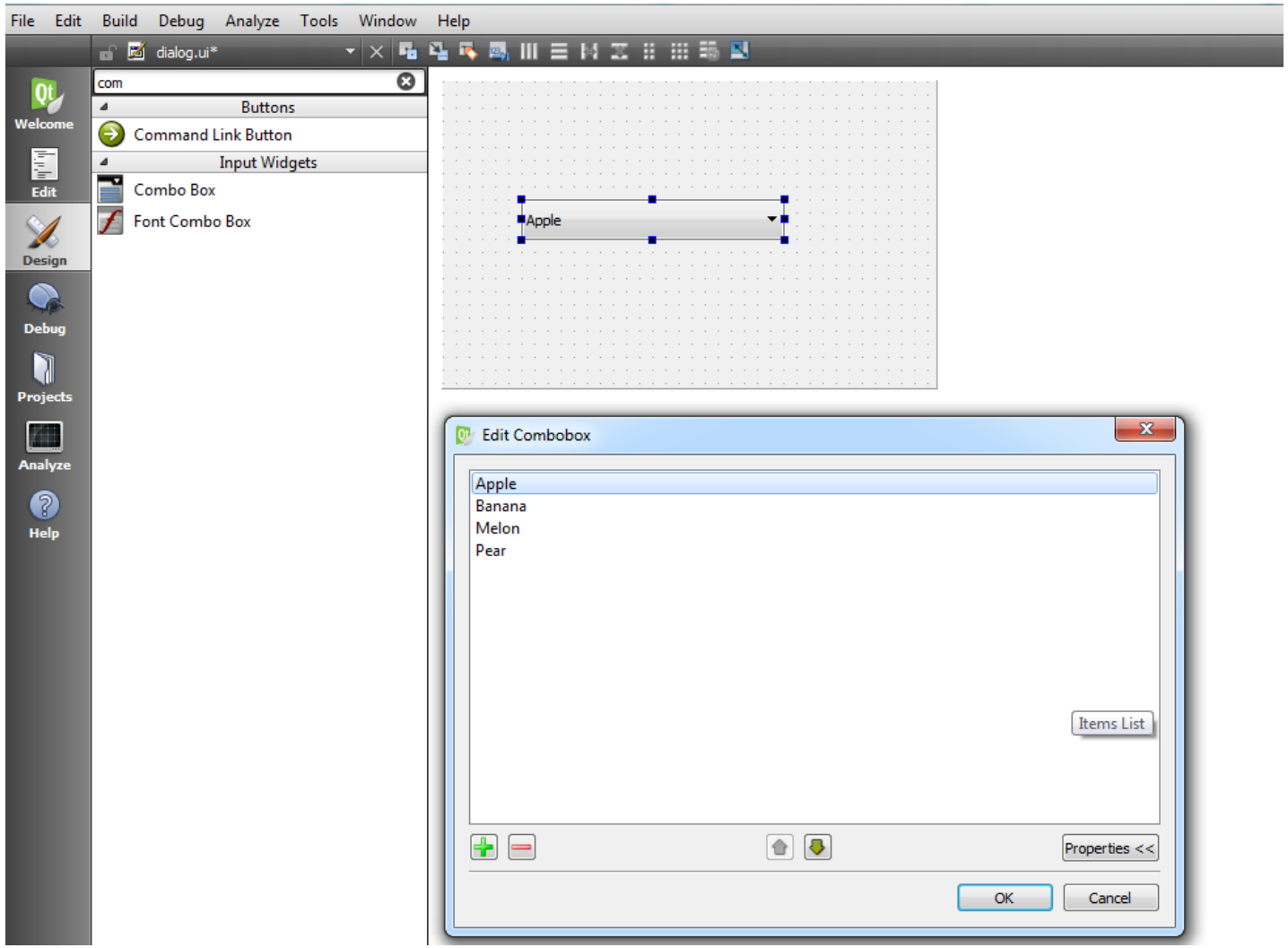
https://codeloop.org/qt5-gui-how-to-create-qradiobutton/

- QComboBox
  - A combobox is a selection widget that displays the current item, and can pop up a list of selectable items.
  - A combobox may be editable, allowing the user to modify each item in the list.
  - Example

https://codeloop.org/qt5-gui-development-how-to-create-combobox/

Qt5 C++ GUI Development

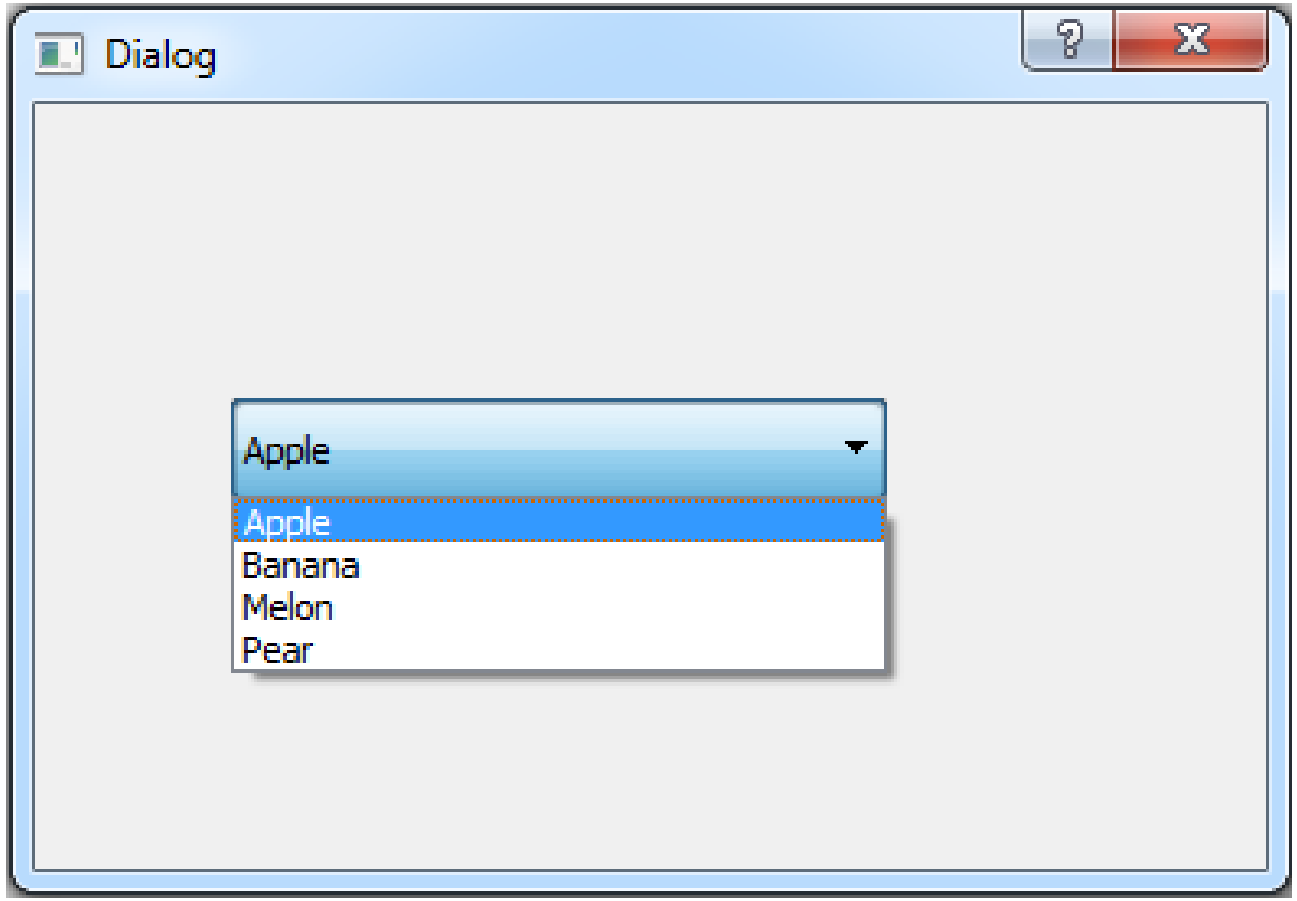
- QComboBox
  - Example: first way



https://codeloop.org/qt5-gui-development-how-to-create-combobox/

Qt5 C++ GUI Development

- QComboBox
  - Example: first way



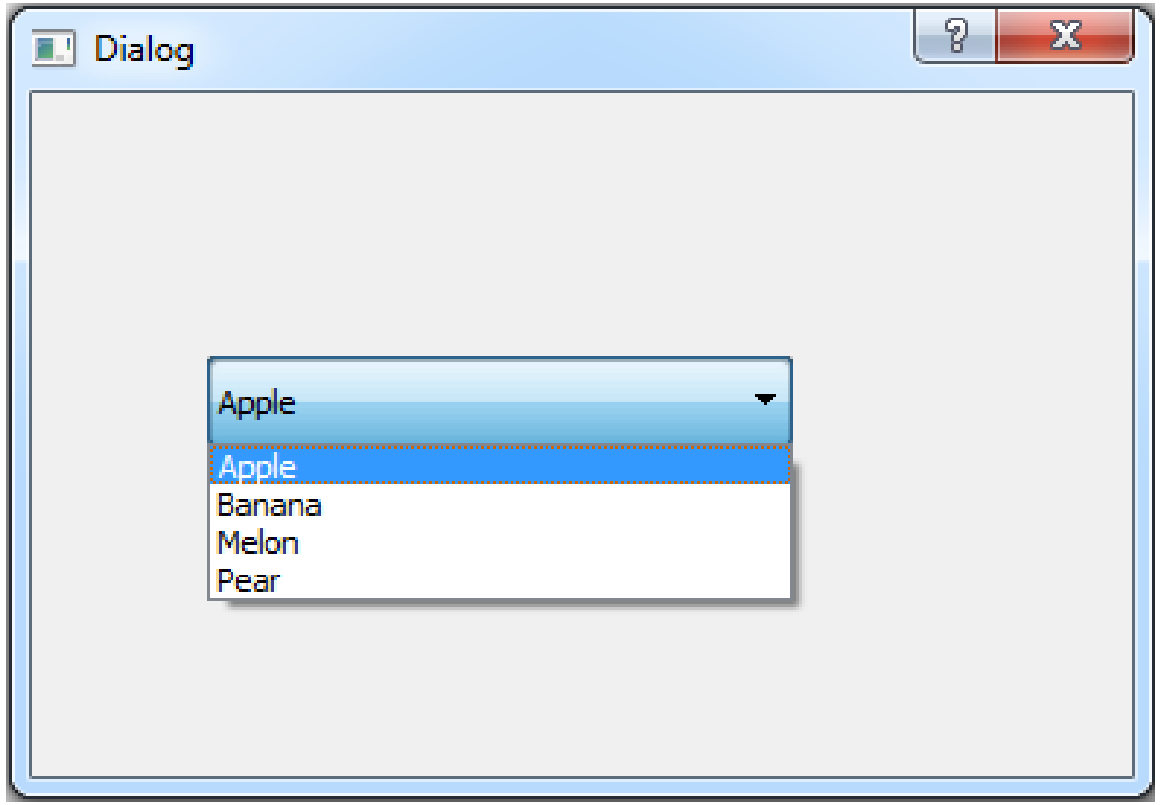
https://codeloop.org/qt5-gui-development-how-to-create-combobox/

Qt5 C++ GUI Development

- QComboBox
  - Example: second way

dialog.cpp

```
1  #include "dialog.h"
2  #include "ui_dialog.h"
3
4  Dialog::Dialog(QWidget *parent) :
5      QDialog(parent),
6      ui(new Ui::Dialog)
7  {
8      ui->setupUi(this);
9
10     ui->comboBox->addItem("Apple");
11     ui->comboBox->addItem("Banana");
12     ui->comboBox->addItem("Pear");
13     ui->comboBox->addItem("Melon");
14 }
15
16 Dialog::~Dialog()
17 {
18     delete ui;
19 }
```

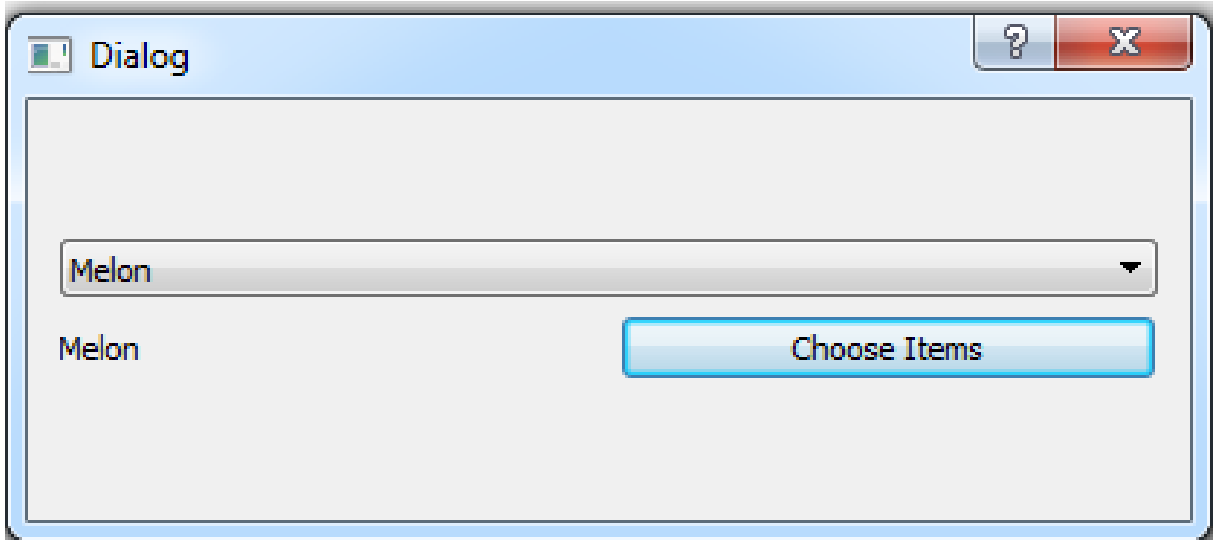


https://codeloop.org/qt5-gui-development-how-to-create-combobox/

- QComboBox
  - Example: second way

dialog.cpp

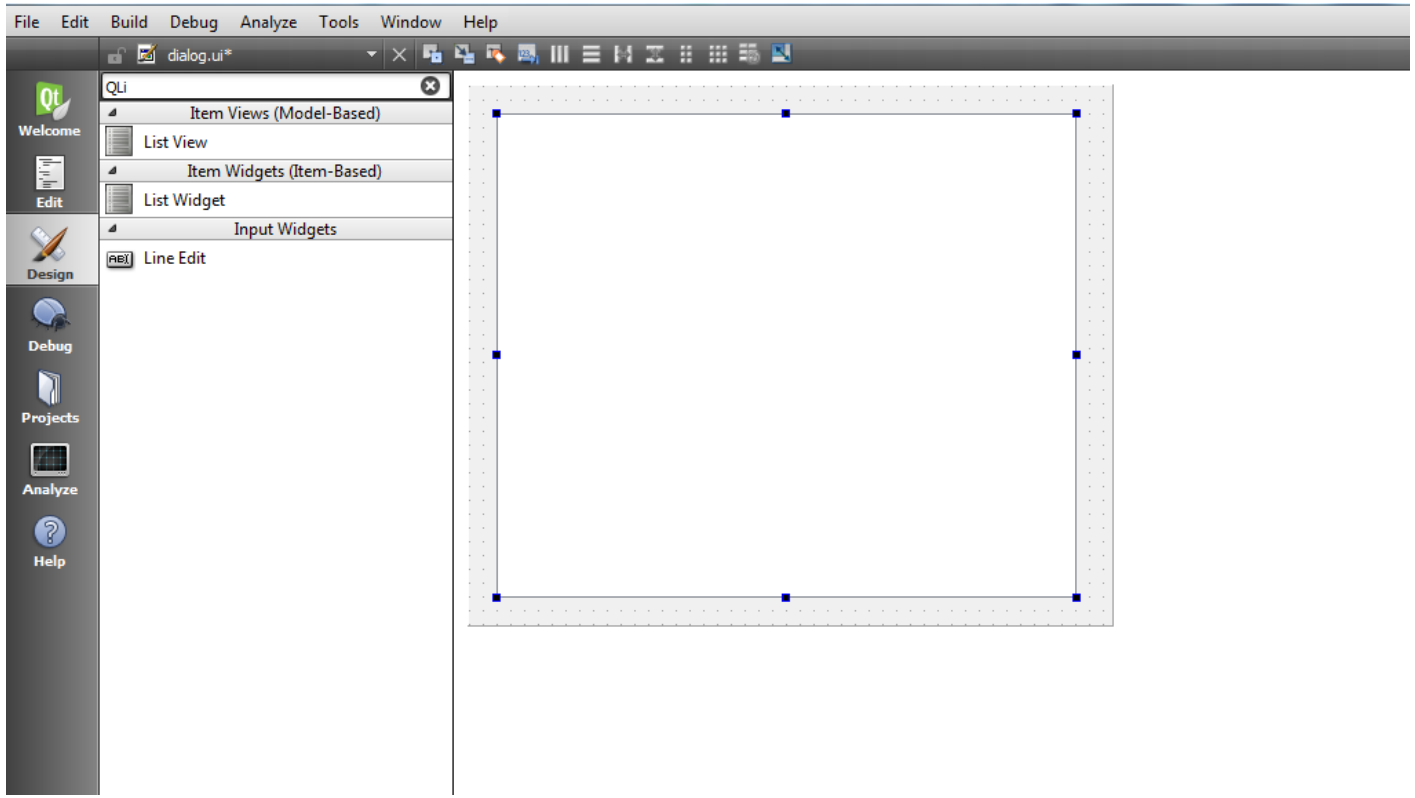
```cpp
#include "dialog.h"
#include "ui_dialog.h"

Dialog::Dialog(QWidget *parent) :
    QDialog(parent),
    ui(new Ui::Dialog)
{
    ui->setupUi(this);

    ui->comboBox->addItem("Apple");
    ui->comboBox->addItem("Banana");
    ui->comboBox->addItem("Pear");
    ui->comboBox->addItem("Melon");
}

Dialog::~Dialog()
{
    delete ui;
}

void Dialog::on_pushButton_clicked()
{
    ui->label->setText(ui->comboBox->currentText());
}
```



https://codeloop.org/qt5-gui-development-how-to-create-combobox/

Qt5 C++ GUI Development

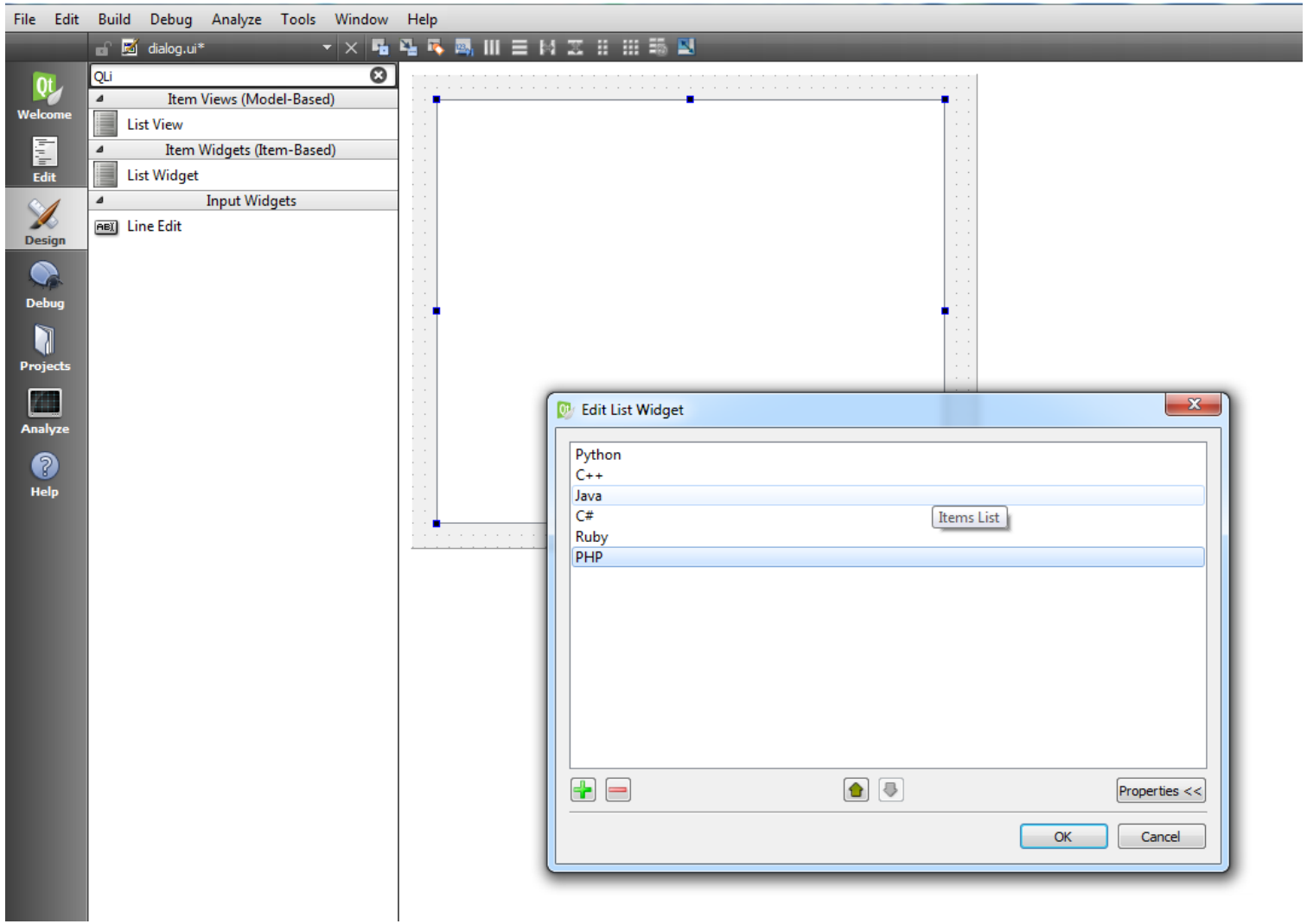
- QListWidget
  - QListWidget is a convenience class that provides a list view similar to the one supplied by QListView, but with a classic item-based interface for adding and removing items.
  - Example:



https://codeloop.org/qt5-gui-development-how-to-create-qlistwidget/

Qt5 C++ GUI Development

- QListWidget
  - Example: first way

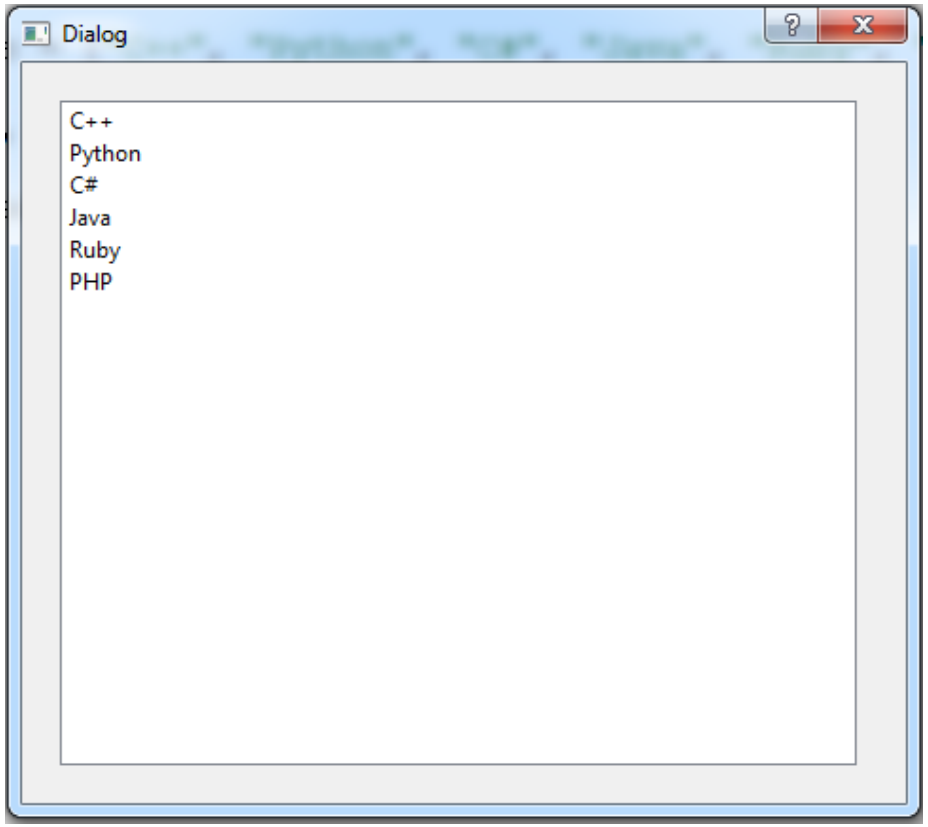


https://codeloop.org/qt5-gui-development-how-to-create-qlistwidget/

- QListWidget
  - Example: second way

dialog.cpp

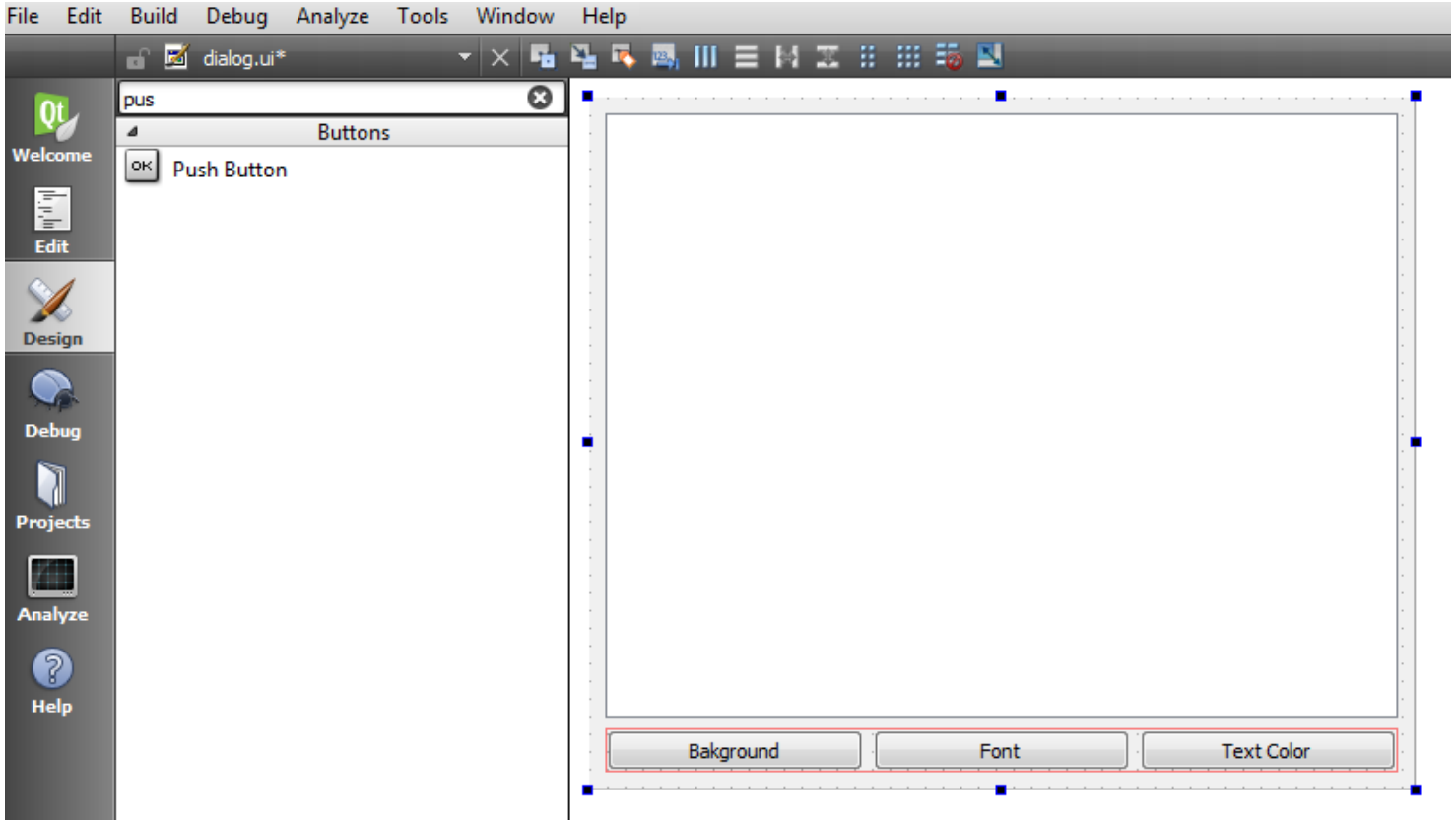
```cpp
#include "dialog.h"
#include "ui_dialog.h"

Dialog::Dialog(QWidget *parent) :
    QDialog(parent),
    ui(new Ui::Dialog)
{
    ui->setupUi(this);

    QStringList languages = {"C++", "Python", "C#", "Java", "Ruby", "PHP"};

    foreach(QString item, languages) {

        ui->listWidget->addItem(item);

    }
}

Dialog::~Dialog()
{
    delete ui;
}
```



https://codeloop.org/qt5-gui-development-how-to-create-qlistwidget/

Qt5 C++ GUI Development

- QListWidget
  - Example: Design UI



https://codeloop.org/qt5-gui-development-how-to-create-qlistwidget/

Qt5 C++ GUI Development

- QListWidget
  - Example: Coding

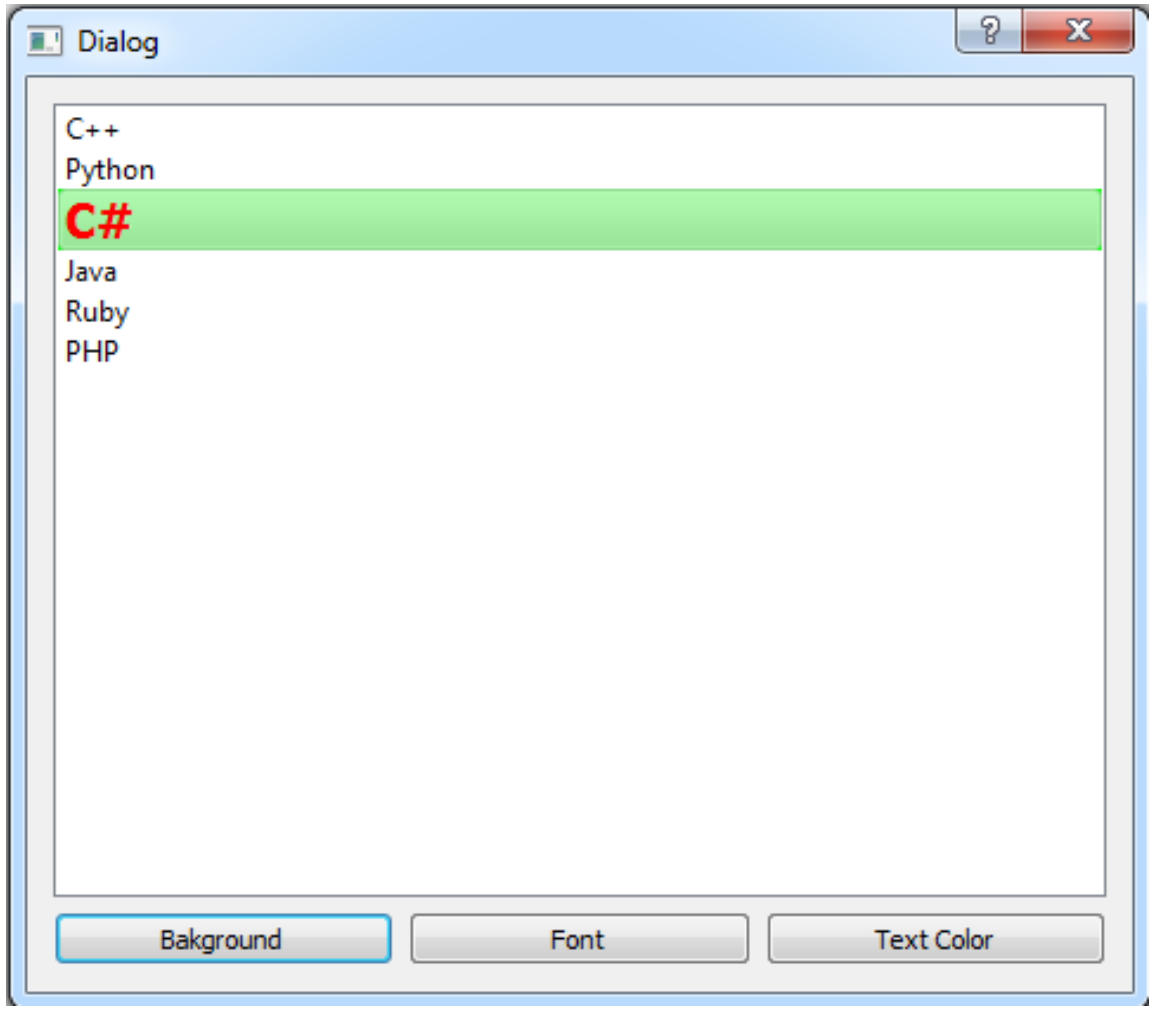
dialog.cpp

```cpp
#include "dialog.h"
#include "ui_dialog.h"

Dialog::Dialog(QWidget *parent) :
    QDialog(parent),
    ui(new Ui::Dialog)
{
    ui->setupUi(this);

    QStringList languages = {"C++", "Python", "C#", "Java", "Ruby", "PHP"};

    foreach(QString item, languages) {

        ui->listWidget->addItem(item);

    }
}

Dialog::~Dialog()
{
    delete ui;
}

void Dialog::on_pushButton_clicked()
{

    QListWidgetItem *item = ui->listWidget->currentItem();
    item->setTextColor(Qt::red);

}

void Dialog::on_pushButton_2_clicked()
{
    QFont font("Times", 15, QFont::Bold);
    QListWidgetItem *item = ui->listWidget->currentItem();
    item->setFont(font);

}

void Dialog::on_pushButton_3_clicked()
{
    QListWidgetItem *item = ui->listWidget->currentItem();
    item->setBackgroundColor(Qt::green);

}
```

https://codeloop.org/qt5-gui-development-how-to-create-qlistwidget/

Qt5 C++ GUI Development

- QListWidget
  - Example:



https://codeloop.org/qt5-gui-development-how-to-create-qlistwidget/

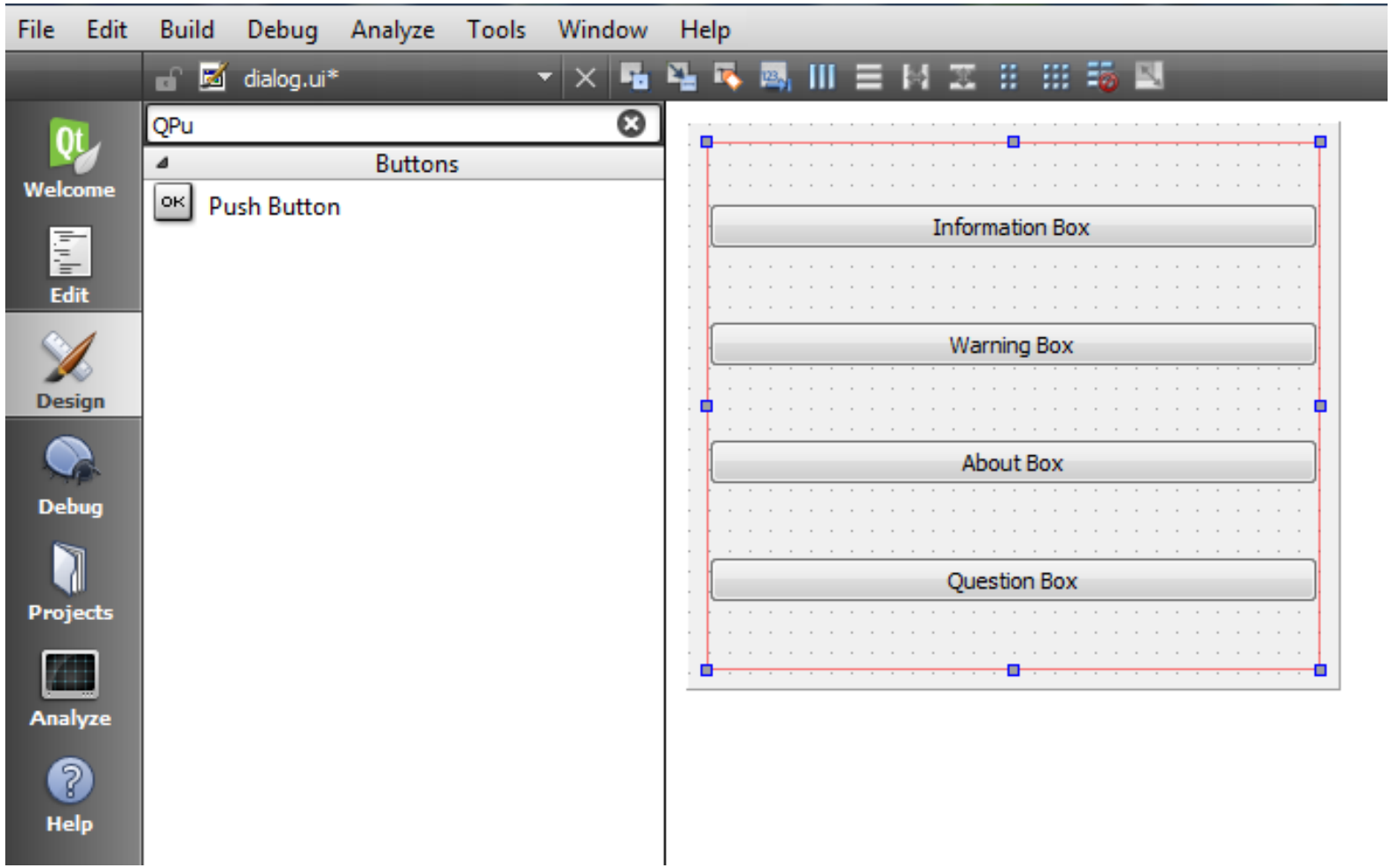
- QMessageBox
  - QMessageBox supports four predefined message severity levels, or message types, which really only differ in the predefined icon they each show.
  - The following rules are guidelines:

	Question	For asking a question during normal operations.
	Information	For reporting information about normal operations.
	Warning	For reporting non-critical errors.
	Critical	For reporting critical errors.

https://codeloop.org/how-to-create-qmessagebox-in-qt5-gui/

Qt5 C++ GUI Development

- QMessageBox
  - Example



https://codeloop.org/how-to-create-qmessagebox-in-qt5-gui/

Qt5 C++ GUI Development

- QMessageBox
  - Example

dialog.cpp

```cpp
#include "dialog.h"
#include "ui_dialog.h"

Dialog::Dialog(QWidget *parent) :
    QDialog(parent),
    ui(new Ui::Dialog)
{
    ui->setupUi(this);
}

Dialog::~Dialog()
{
    delete ui;
}

void Dialog::on_pushButton_clicked()
{
    // Information MessageBox

}

void Dialog::on_pushButton_2_clicked()
{
    //Warning MessageBox


}

void Dialog::on_pushButton_3_clicked()
{
    //About MessageBox
}

void Dialog::on_pushButton_4_clicked()
{
    //Question MessageBox
}
```

dialog.h

```cpp
#ifndef DIALOG_H
#define DIALOG_H

#include <QDialog>
#include<QMessageBox>

namespace Ui {
class Dialog;
}

class Dialog : public QDialog
{
    Q_OBJECT

public:
    explicit Dialog(QWidget *parent = 0);
    ~Dialog();

private slots:
    void on_pushButton_clicked();

    void on_pushButton_2_clicked();

    void on_pushButton_3_clicked();

    void on_pushButton_4_clicked();

private:
    Ui::Dialog *ui;
};

#endif // DIALOG_H
```

https://codeloop.org/how-to-create-qmessagebox-in-qt5-gui/

Qt5 C++ GUI Development

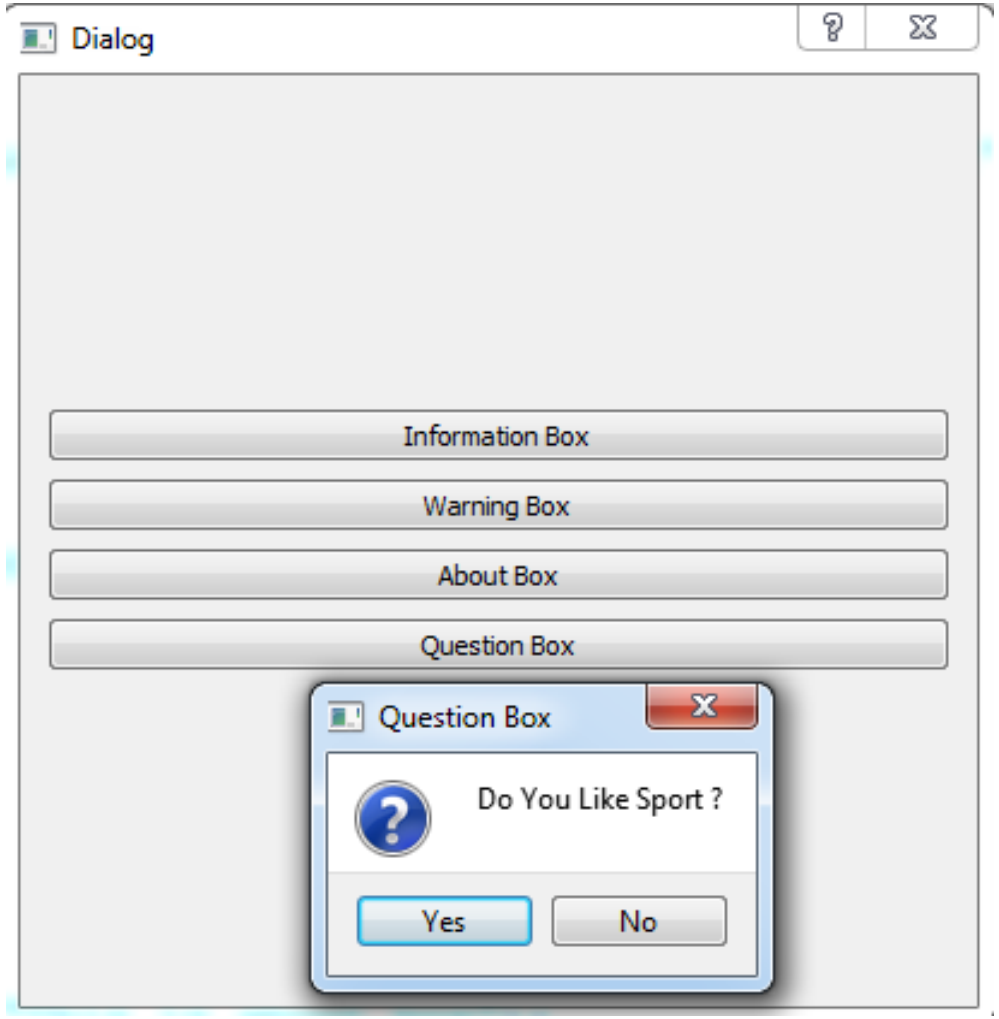
- QMessageBox
  - Example

```
1  void Dialog::on_pushButton_clicked()
2  {
3      // Information MessageBox
4
5
6      QMessageBox::information(this, "Information Box", "This is information text");
7
8  }
9
10 void Dialog::on_pushButton_2_clicked()
11 {
12     //Warning MessageBox
13
14     QMessageBox::warning(this, "Warning Box", "This is a warning box");
15
16
17 }
18
19 void Dialog::on_pushButton_3_clicked()
20 {
21     //About MessageBox
22
23     QMessageBox::about(this, "About Box", "This is about box");
24 }
25
26 void Dialog::on_pushButton_4_clicked()
27 {
28     //Question MessageBox
29
30     QMessageBox::question(this, "Question Box", "Do You Like Sport ? ",
31                           QMessageBox::Yes | QMessageBox::No );
32 }
```

https://codeloop.org/how-to-create-qmessagebox-in-qt5-gui/

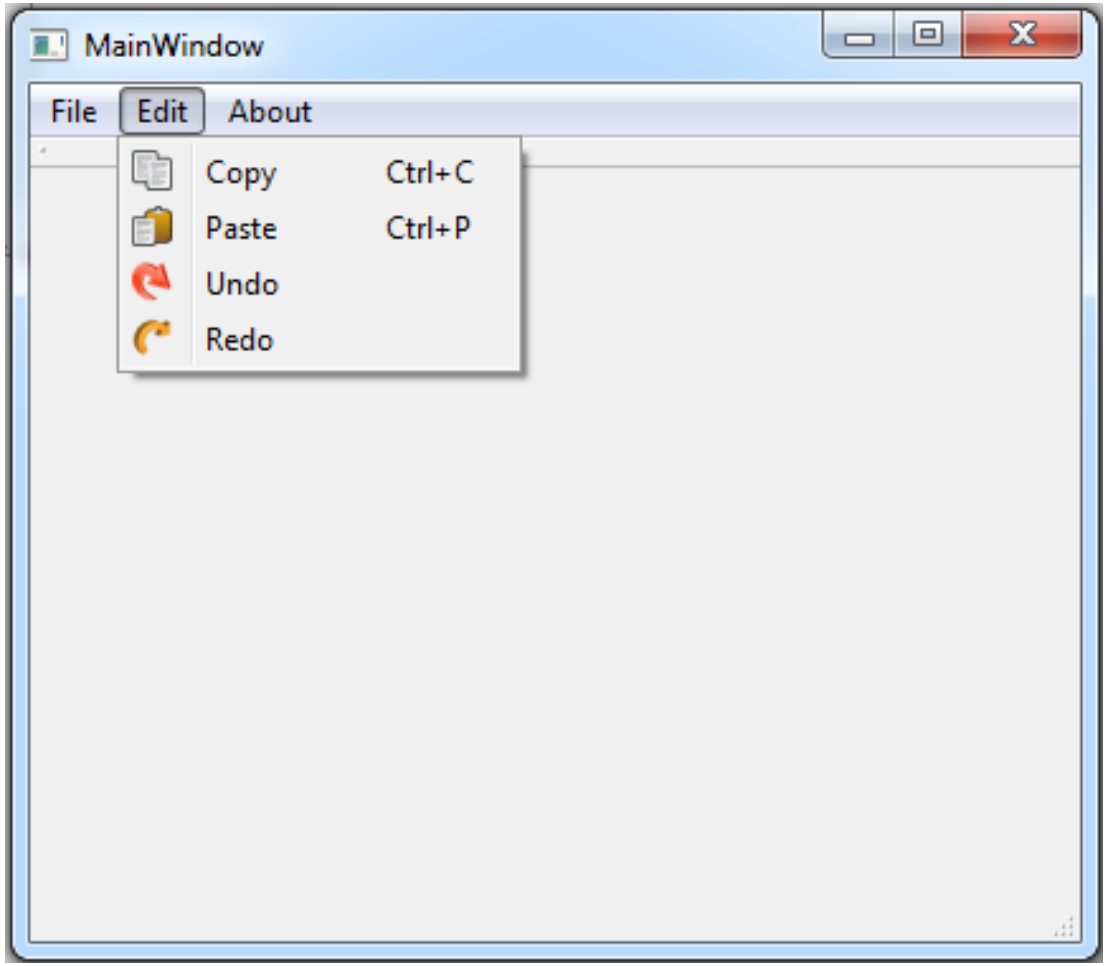
Qt5 C++ GUI Development

- QMessageBox
  - Example



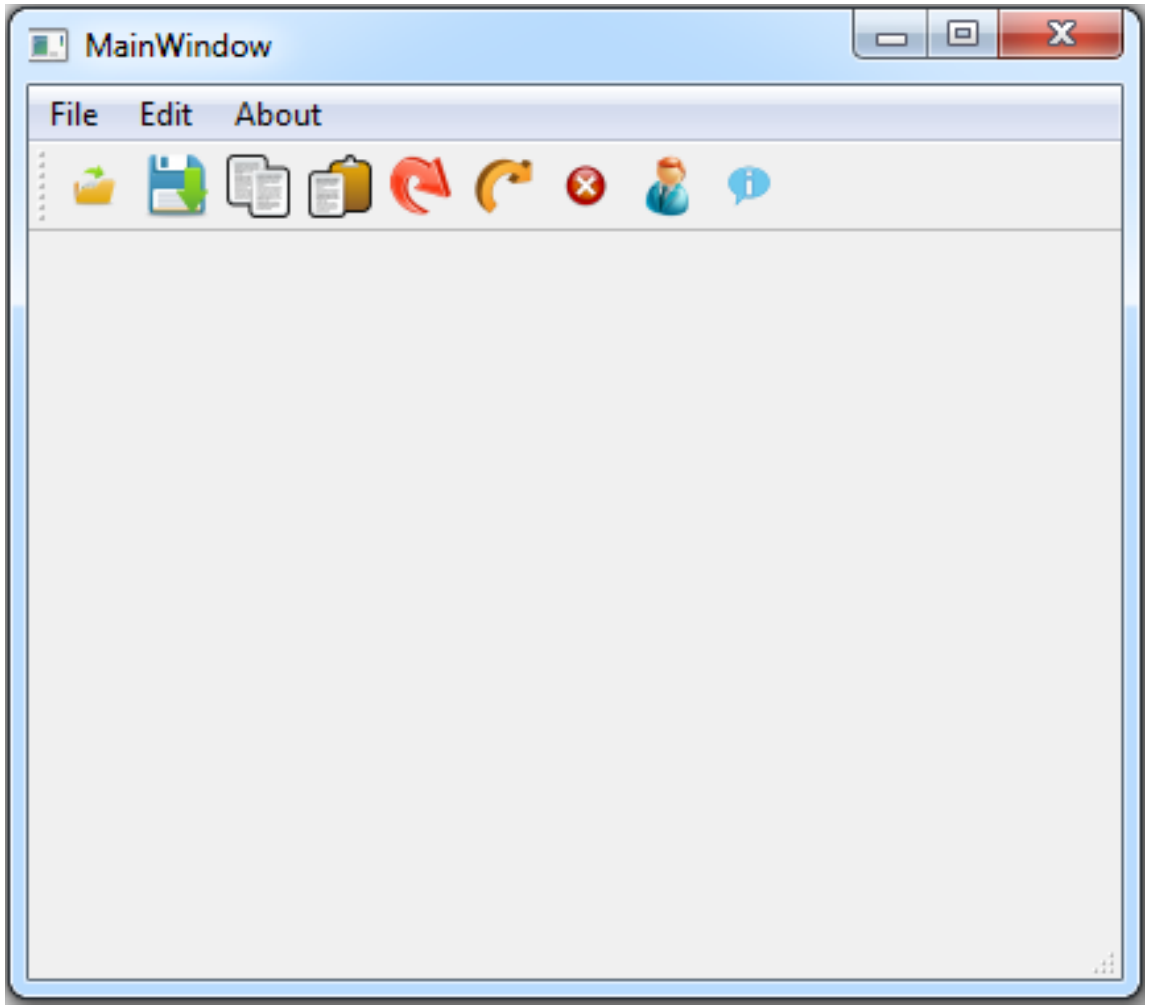
https://codeloop.org/how-to-create-qmessagebox-in-qt5-gui/

- QMenu And QToolbar
  - What is a Menu?



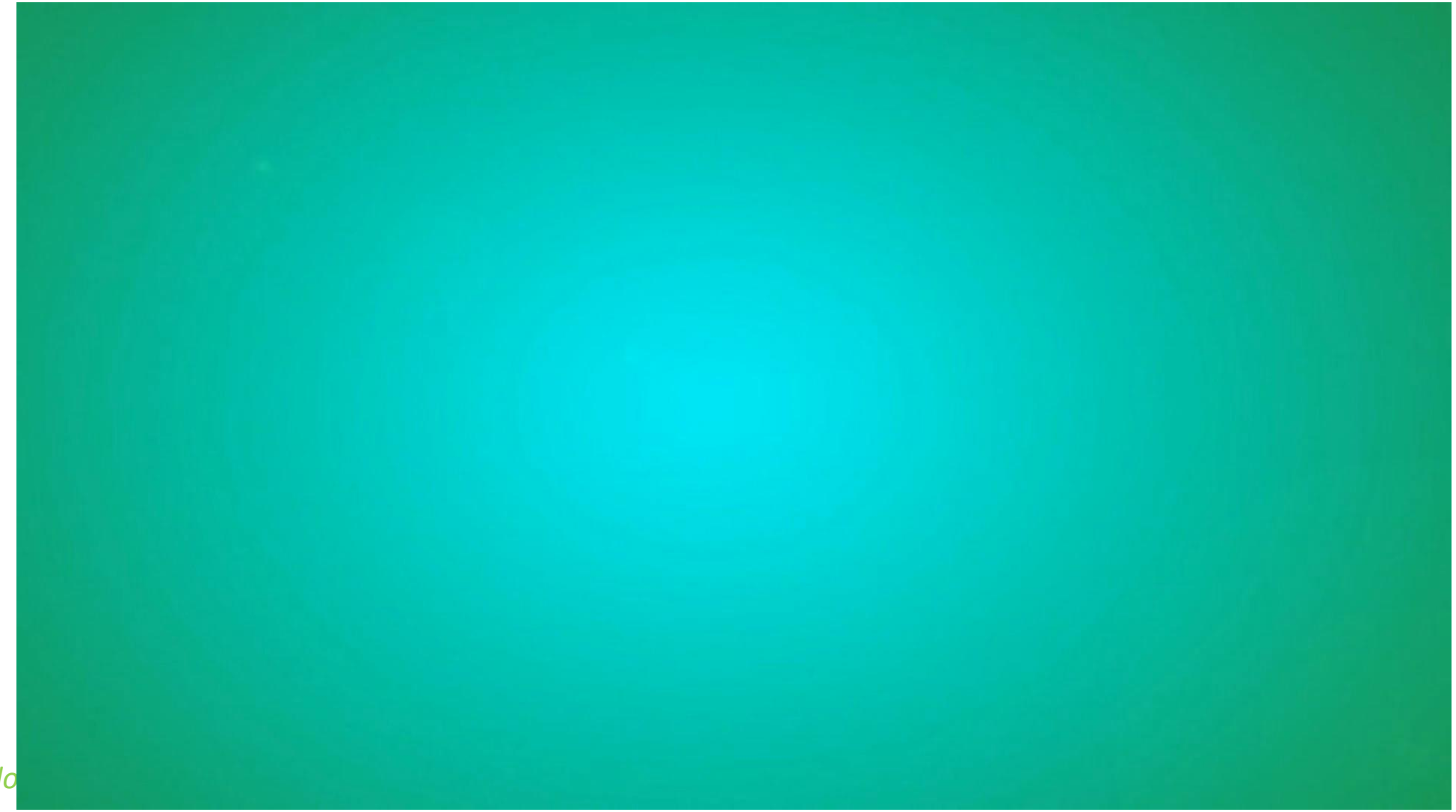
https://codeloop.org/qt5-gui-creating-qmenu-and-qtoolbar/

- QMenu And QToolbar
  - What is a QToolbar?



https://codeloop.org/qt5-gui-creating-qmenu-and-qtoolbar/

- QMenu And QToolbar



- QFileDialog
  - QFileDialog class enables a user to traverse the file system in order to select one or many files or a directory.

```
fileName = QFileDialog::getOpenFileName(this,
    tr("Open Image"), "/home/jana", tr("Image Files (*.png *.jpg *.bmp)"));
```

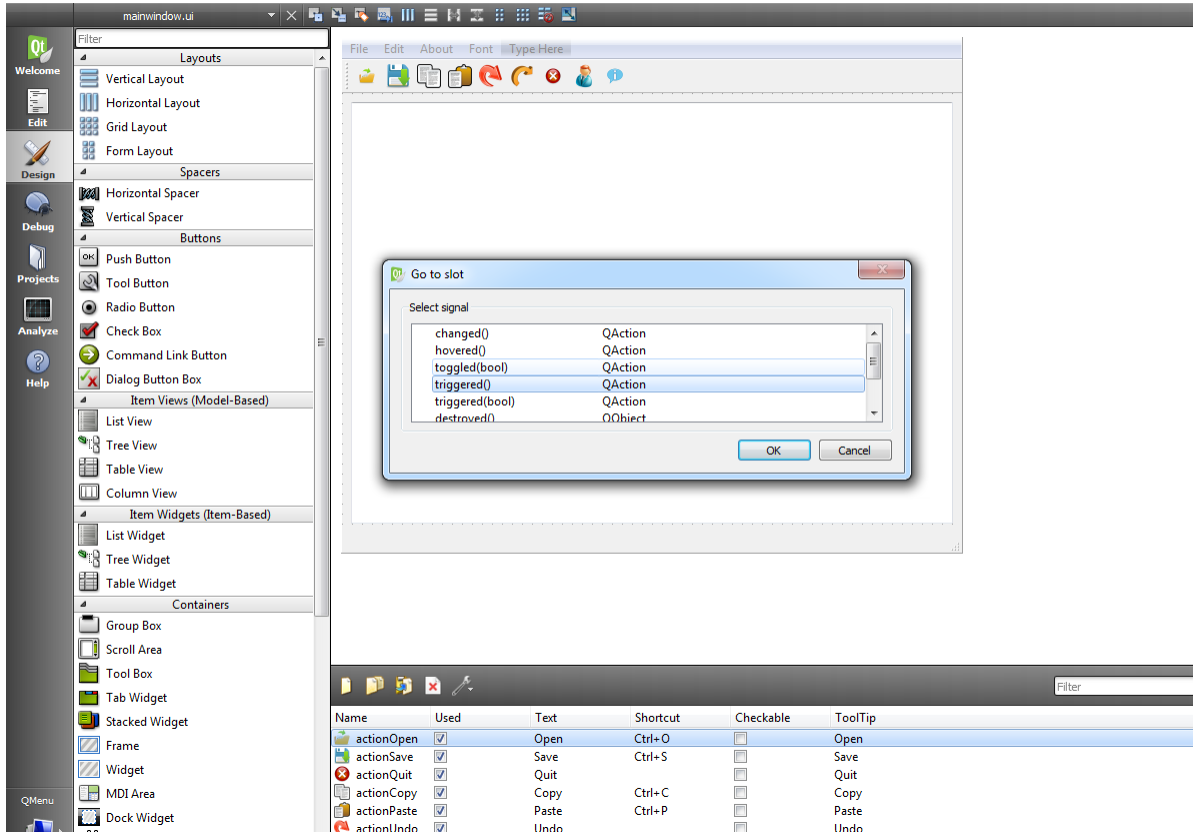
```
"Images (*.png *.xpm *.jpg);;Text files (*.txt);;XML files (*.xml)"
```

```
QFileDialog dialog(this);
dialog.setFileMode(QFileDialog::AnyFile);
```

https://codeloop.org/how-to-create-qfiledialog-in-qt5-gui/

Qt5 C++ GUI Development

- QFileDialog
  - Example: right click on your Open menu item in Signals And Slot Editor, after that choose Go To Slot and from the dialog choose triggered() like this.



https://codeloop.org/how-to-create-qfiledialog-in-qt5-gui/

Qt5 C++ GUI Development

- QFileDialog
  - Example:

mainwindow.h

```
#ifndef MAINWINDOW_H
#define MAINWINDOW_H

#include <QMainWindow>
#include<QFontDialog>
#include<QColorDialog>
#include<QFileDialog>


namespace Ui {
class MainWindow;
}

class MainWindow : public QMainWindow
{
    Q_OBJECT

public:
    explicit MainWindow(QWidget *parent = 0);
    ~MainWindow();

private slots:
    void on_actionPrint_triggered();

    void on_actionFont_triggered();

    void on_actionColor_triggered();

    void on_actionOpen_triggered();

private:
    Ui::MainWindow *ui;
};

#endif // MAINWINDOW_H
```

https://codeloop.org/how-to-create-qfiledialog-in-qt5-gui/

- QFileDialog
  - Example:

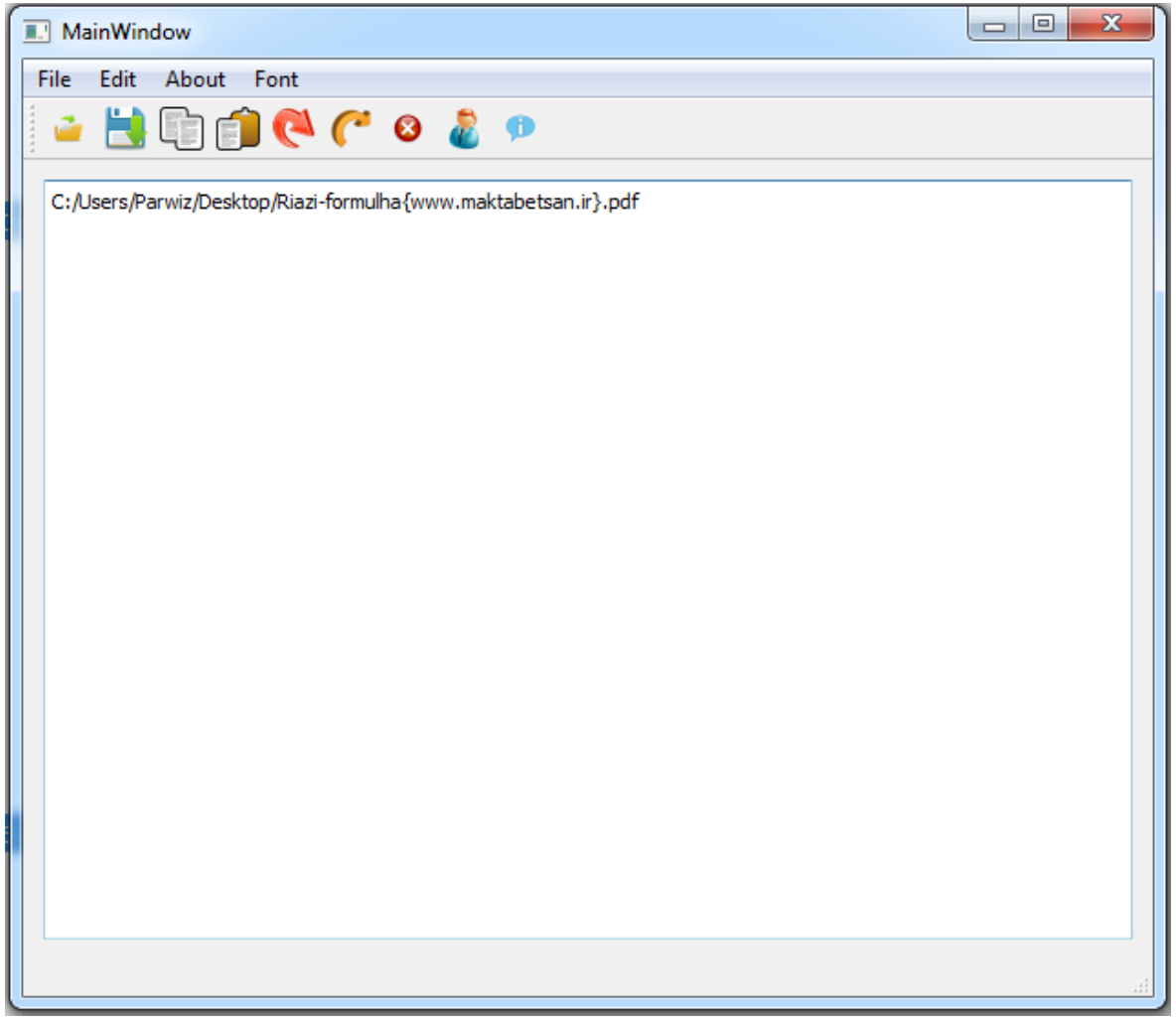
mainwindow.cpp

```
78 void MainWindow::on_actionOpen_triggered()
79 {
80
81     QString file = QFileDialog::getOpenFileName(this, "Open A File", "C://");
82     ui->textEdit->setText(file);
83
84 }
```

https://codeloop.org/how-to-create-qfiledialog-in-qt5-gui/

Qt5 C++ GUI Development

- QFileDialog
  - Example:



https://codeloop.org/how-to-create-qfiledialog-in-qt5-gui/

- QFileDialog
  - Example: Opening and saving files
    - *Open Files*

```
void Notepad::open()
{
    QString fileName = QFileDialog::getOpenFileName(this, tr("Open File"), "",
        tr("Text Files (*.txt);;C++ Files (*.cpp *.h)"));

    if (!fileName.isEmpty()) {
        QFile file(fileName);
        if (!file.open(QIODevice::ReadOnly)) {
            QMessageBox::critical(this, tr("Error"), tr("Could not open file"));
            return;
        }
        QTextStream in(&file);
        textEdit->setText(in.readAll());
        file.close();
    }
}
```

https://qt.misfrog.com/posts/%E3%83%A1%E3%83%A2%E3%81%AE%E4%BF%9D%E5%AD%98%E3%81%A8%E3%83%AD%E3%83%BC%E3%83%89

- QFileDialog
  - Example: Opening and saving files
    - *Save Files*

```
void Notepad::save()
{

    QString fileName = QFileDialog::getSaveFileName(this, tr("Save File"), "",
        tr("Text Files (*.txt);;C++ Files (*.cpp *.h)"));

    if (!fileName.isEmpty()) {
        QFile file(fileName);
        if (!file.open(QIODevice::WriteOnly)) {
            // error message
        } else {
            QTextStream stream(&file);
            stream << textEdit->toPlainText();
            stream.flush();
            file.close();
        }
    }
}
```

https://qt.misfrog.com/posts/%E3%83%A1%E3%83%A2%E3%81%AE%E4%BF%9D%E5%AD%98%E3%81%A8%E3%83%AD%E3%83%BC%E3%83%89

- QFileDialog
  - Example: Opening, displaying and saving images
    - create a new **Qt Widgets application**
    - add a **Graphics View** (located under the *Display Widgets*)
    - add two **Push Buttons**: *openButton* and *saveButton*

mainwindow.h

```
1  private:
2      Ui::MainWindow *ui;
3      QPixmap image;
4      QImage  *imageObject;
5      QGraphicsScene *scene;
```

http://creative-punch.net/2014/02/opening-displaying-saving-images-qt/

- QFileDialog
  - Example: Opening, displaying and saving images
    - Adding events for the buttons

```
1  void MainWindow::on_openButton_pressed()
2  {
3
4  }
5
6  void MainWindow::on_saveButton_pressed()
7  {
8
9  }
```

http://creative-punch.net/2014/02/opening-displaying-saving-images-qt/

- QFileDialog
  - Example: Opening, displaying and saving images
    - Opening an image and displaying it on the QGraphicsView

```
01 void MainWindow::on_openButton_pressed()
02 {
03     QString imagePath = QFileDialog::getOpenFileName(
04                     this,
05                     tr("Open File"),
06                     "",
07                     tr("JPEG (*.jpg *.jpeg);;PNG (*.png)" )
08                     );
09
10     imageObject = new QImage();
11     imageObject->load(imagePath);
12
13     image = QPixmap::fromImage(*imageObject);
14
15     scene = new QGraphicsScene(this);
16     scene->addPixmap(image);
17     scene->setSceneRect(image.rect());
18     ui->graphicsView->setScene(scene);
19
20 }
```

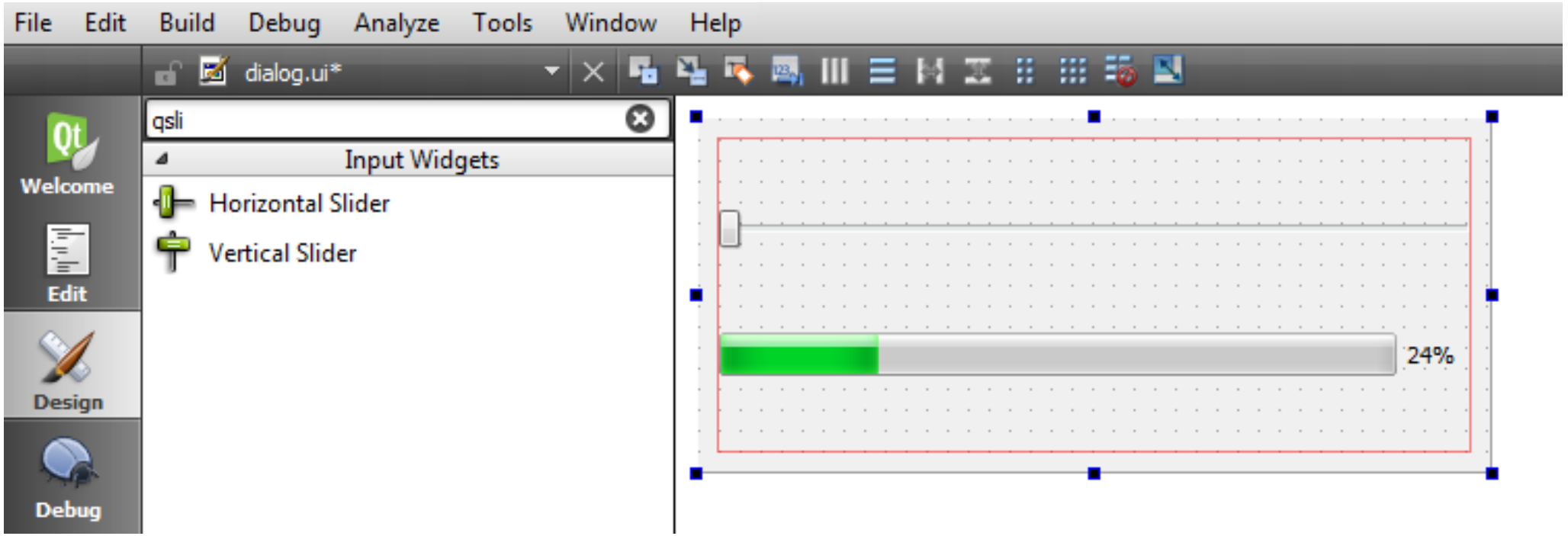
http://creative-punch.net/2014/02/opening-displaying-saving-images-qt/

- QFileDialog
  - Example: Opening, displaying and saving images
    - Saving the image

```
22  void MainWindow::on_saveButton_pressed()
23  {
24      QString imagePath = QFileDialog::getSaveFileName(
25                      this,
26                      tr("Save File"),
27                      "",
28                      tr("JPEG (*.jpg *.jpeg);;PNG (*.png)" )
29                      );
30
31      *imageObject = image.toImage();
32
33      imageObject->save(imagePath);
34  }
```

http://creative-punch.net/2014/02/opening-displaying-saving-images-qt/

- QProgressBar
  - A progress bar is used to give the user an indication of the progress of an operation and to reassure them that the application is still running.
  - Example:



https://codeloop.org/how-to-create-qprogressbar-in-qt5-gui/

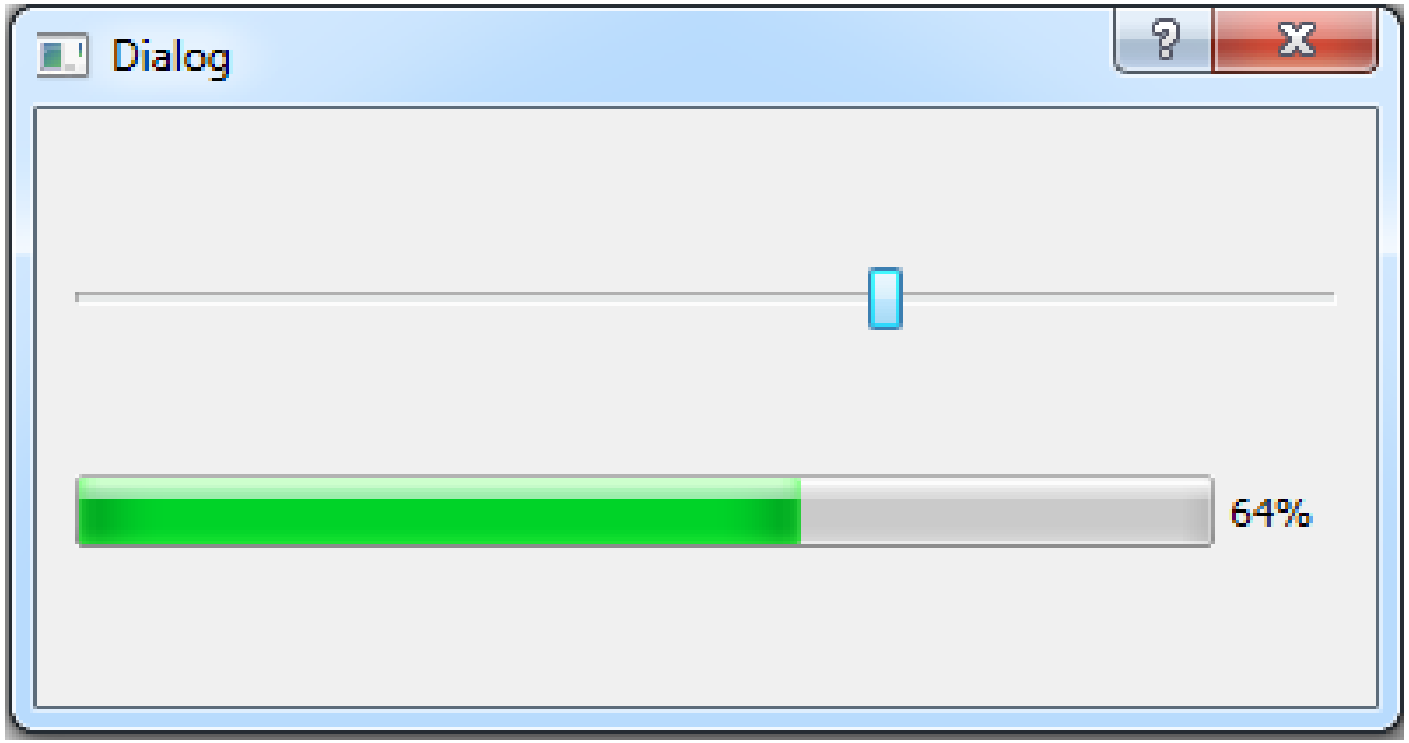
- QProgressBar
  - Example:

dialog.cpp

```cpp
#include "dialog.h"
#include "ui_dialog.h"

Dialog::Dialog(QWidget *parent) :
    QDialog(parent),
    ui(new Ui::Dialog)
{
    ui->setupUi(this);

    connect(ui->horizontalSlider, SIGNAL(valueChanged(int)), ui->progressBar, SLOT(setValue(int)));

}

Dialog::~Dialog()
{
    delete ui;
}
```

https://codeloop.org/how-to-create-qprogressbar-in-qt5-gui/

- QProgressBar
  - Example:



https://codeloop.org/how-to-create-qprogressbar-in-qt5-gui/

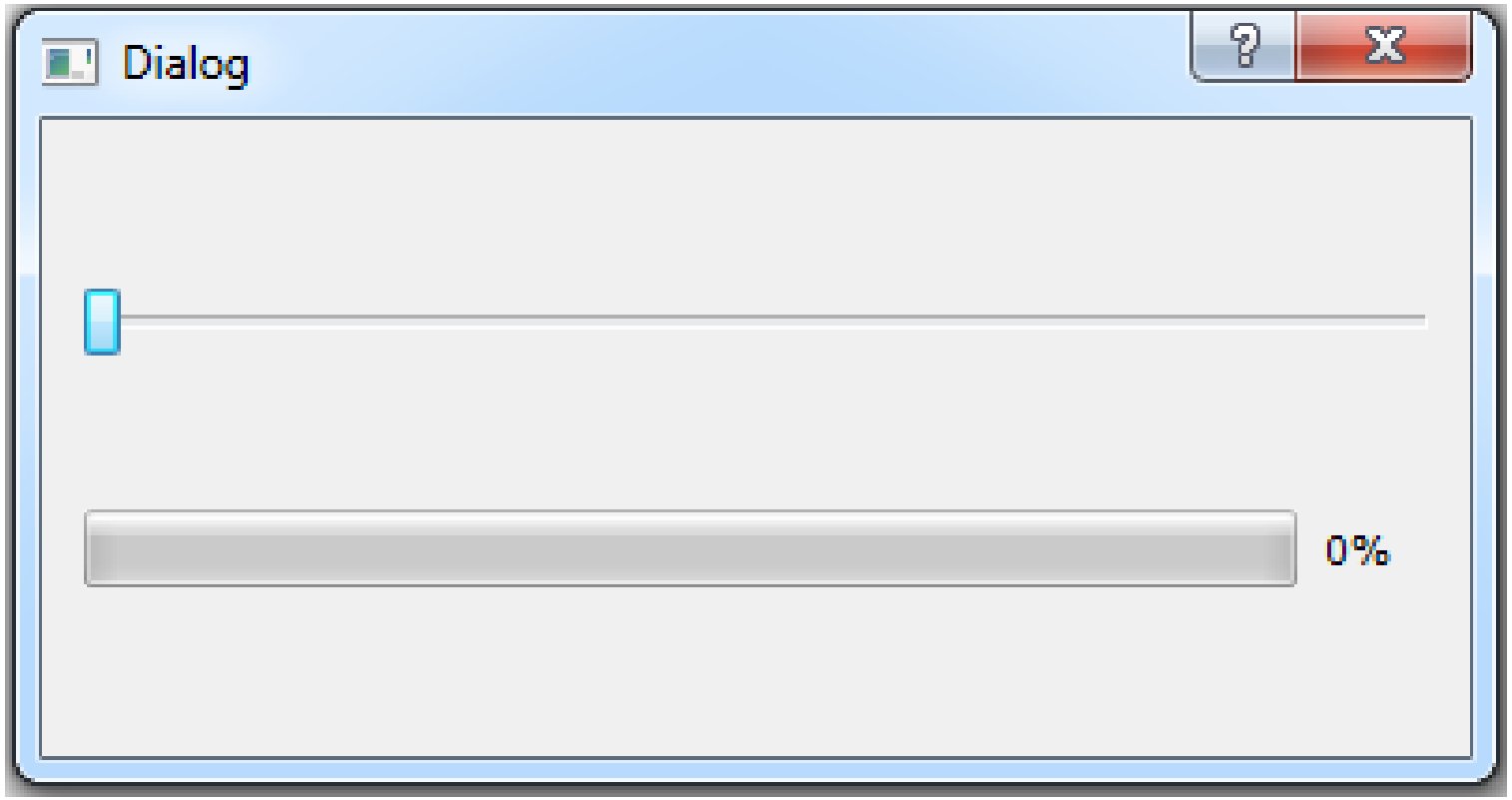
- QProgressBar
  - Example:

dialog.cpp

```cpp
#include "dialog.h"
#include "ui_dialog.h"

Dialog::Dialog(QWidget *parent) :
    QDialog(parent),
    ui(new Ui::Dialog)
{
    ui->setupUi(this);

    ui->progressBar->setValue(ui->horizontalSlider->value());

    connect(ui->horizontalSlider, SIGNAL(valueChanged(int)), ui->progressBar, SLOT(setValue(int)));

}

Dialog::~Dialog()
{
    delete ui;
}
```

https://codeloop.org/how-to-create-qprogressbar-in-qt5-gui/

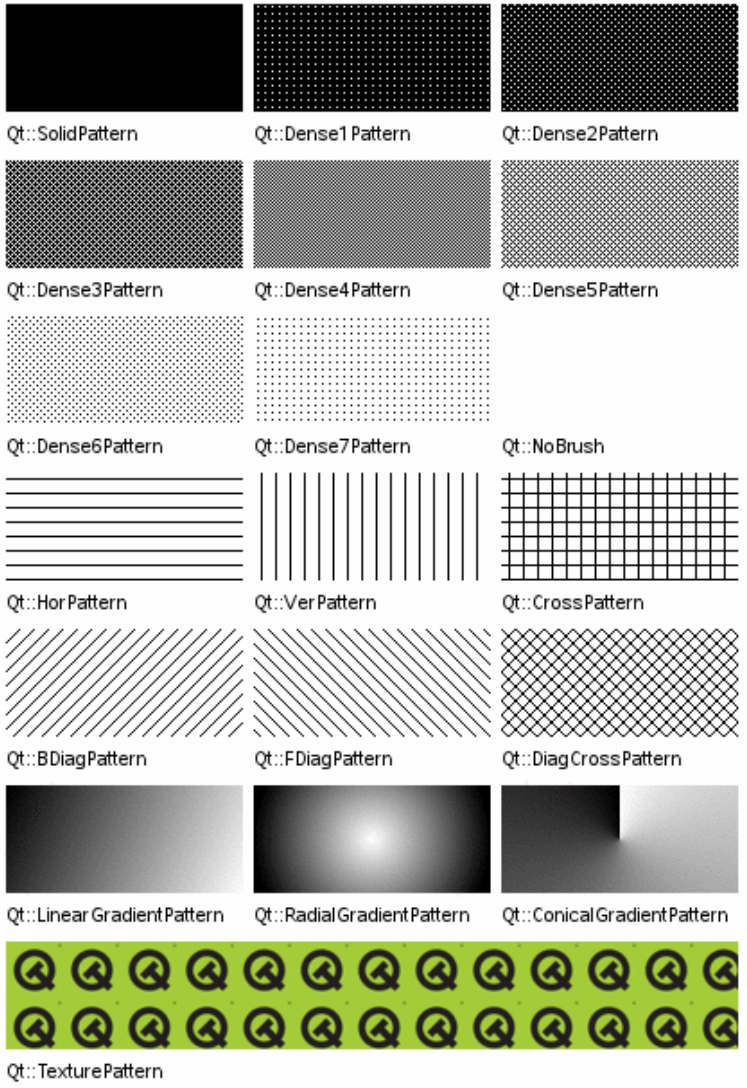
- QProgressBar
  - Example:



https://codeloop.org/how-to-create-qprogressbar-in-qt5-gui/

Qt5 C++ GUI Development

- Draw Text & Line with QPainter
  - The QPainter class performs low-level painting on widgets and other paint devices.
  - QPainter provides highly optimized functions to do most of the drawing GUI programs require.
  - What is Qbrush?
    - A brush has a style, a color, a gradient and a texture.
    - The brush style() defines the fill pattern using the Qt::BrushStyle enum.



https://codeloop.org/how-to-draw-text-line-in-qt5-with-qpainter/

Brush Styles

- Draw Text & Line with QPainter
  - What is QPen?
    - A pen has a style(), width(), brush(), capStyle() and joinStyle().
    - The pen style defines the line type.
    - The brush is used to fill strokes generated with the pen. Use the QBrush class to specify fill styles.

https://codeloop.org/how-to-draw-text-line-in-qt5-with-qpainter/

- Draw Text & Line with QPainter
  - Example:

mainwindow.h

```
1  #ifndef MAINWINDOW_H
2  #define MAINWINDOW_H
3
4  #include <QMainWindow>
5
6  #include<QPainter>
7  #include<QTextDocument>
8
9  namespace Ui {
10 class MainWindow;
11 }
12
13 class MainWindow : public QMainWindow
14 {
15     Q_OBJECT
16
17 public:
18     explicit MainWindow(QWidget *parent = 0);
19     ~MainWindow();
20
21     virtual void paintEvent(QPaintEvent *event);
22
23
24
25 private:
26     Ui::MainWindow *ui;
27 };
28
29 #endif // MAINWINDOW_H
```

https://codeloop.org/how-to-draw-text-line-in-qt5-with-qpainter/

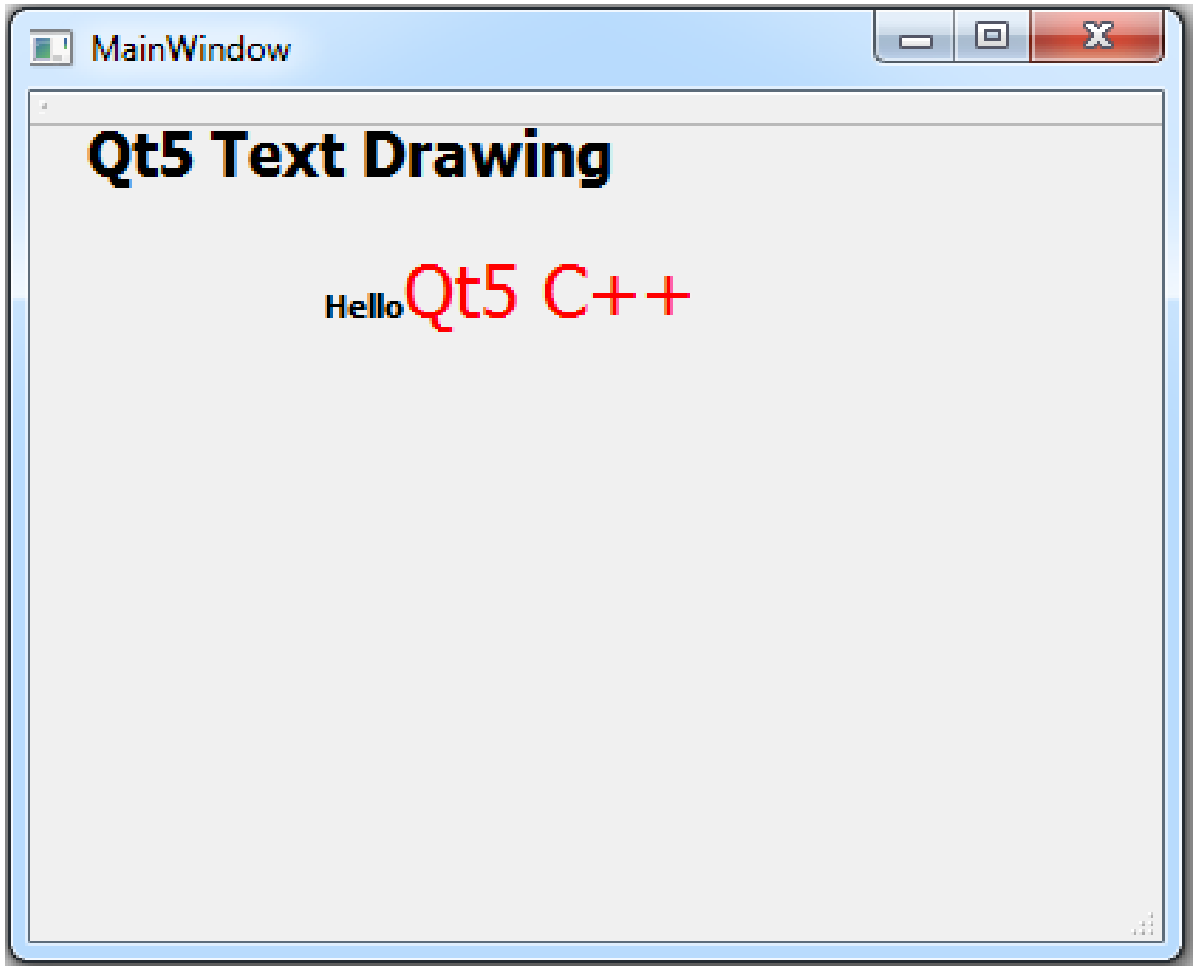
- Draw Text & Line with QPainter
  - Example:

mainwindow.cpp

```
1  void MainWindow::paintEvent(QPaintEvent *event)
2  {
3      //Drawing Texts
4
5      QPainter mytext(this);
6      mytext.setFont(QFont("Times", 16, QFont::Bold));
7      mytext.drawText(QPoint(20,30), "Qt5 Text Drawing");
8
9
10     // Giving Style To Texts
11
12
13     QTextDocument document;
14     QRect rect(0,0,250,250 );
15     mytext.translate(100,50);
16
17     document.setHtml("<b>Hello</b><font color='red' size='30'>Qt5 C++ </font>");
18     document.drawContents(&mytext, rect);
19
20
21
22
23
24
25 }
```

https://codeloop.org/how-to-draw-text-line-in-qt5-with-qpainter/

- Draw Text & Line with QPainter
  - Example:



https://codeloop.org/how-to-draw-text-line-in-qt5-with-qpainter/

Qt5 C++ GUI Development

- Draw Rectangle
  - Example

mainwindow.h

```
1  #ifndef MAINWINDOW_H
2  #define MAINWINDOW_H
3
4  #include <QMainWindow>
5  #include<QPainter>
6
7  namespace Ui {
8  class MainWindow;
9  }
10
11 class MainWindow : public QMainWindow
12 {
13     Q_OBJECT
14
15 public:
16     explicit MainWindow(QWidget *parent = 0);
17     ~MainWindow();
18
19
20     virtual void paintEvent(QPaintEvent *event);
21
22 private:
23     Ui::MainWindow *ui;
24 };
25
26 #endif // MAINWINDOW_H
```

https://codeloop.org/qt5-qpainter-how-to-draw-rectangle/

Qt5 C++ GUI Development

- Draw Rectangle
  - Example

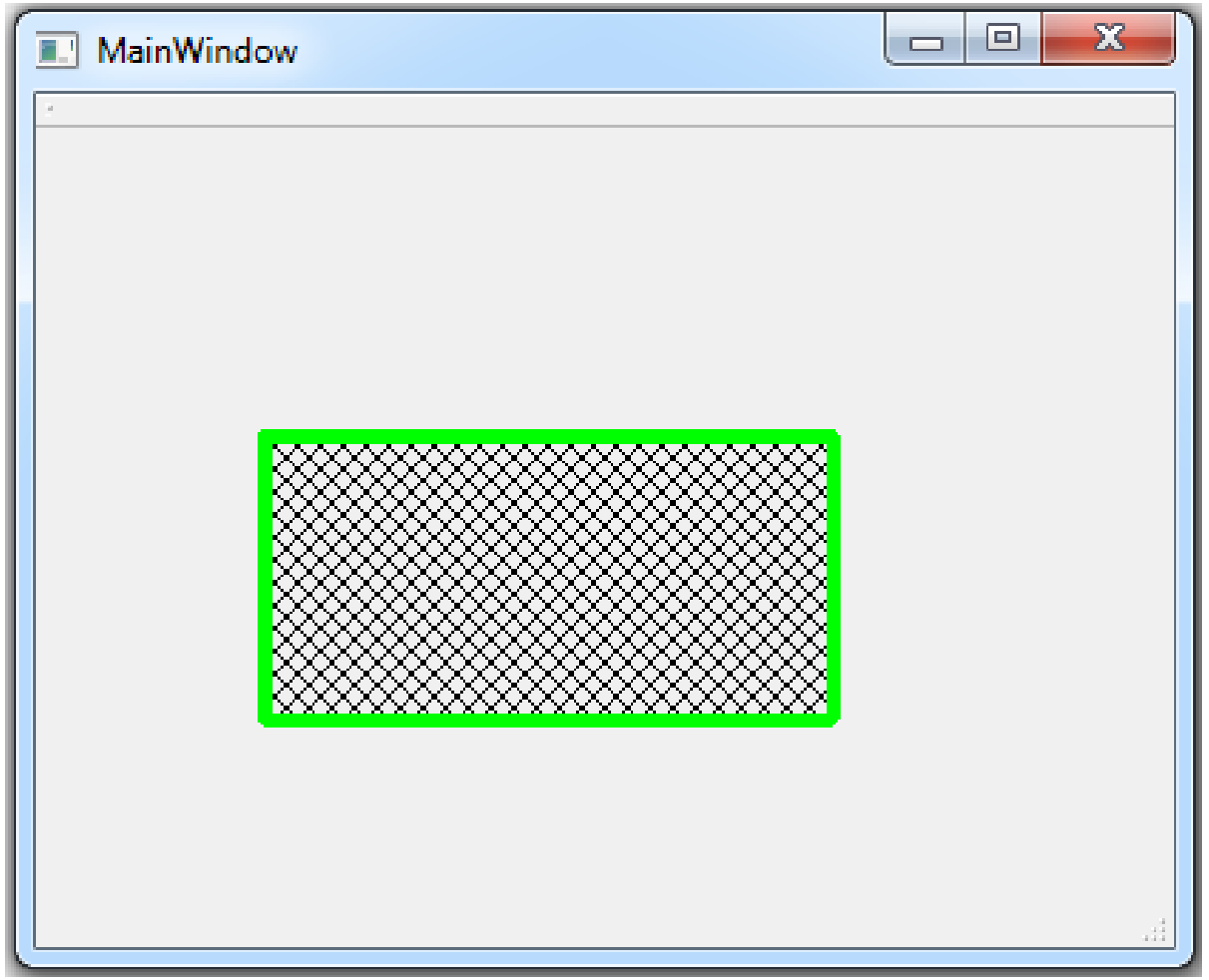
mainwindow.cpp

mainwindow.cpp

```cpp
#include "mainwindow.h"
#include "ui_mainwindow.h"

MainWindow::MainWindow(QWidget *parent) :
    QMainWindow(parent),
    ui(new Ui::MainWindow)
{
    ui->setupUi(this);
}

MainWindow::~MainWindow()
{
    delete ui;
}

void MainWindow::paintEvent(QPaintEvent *event)
{

    QPainter painter(this);
    painter.setBrush(Qt::DiagCrossPattern);


    QPen pen;

    pen.setColor(Qt::green);
    pen.setWidth(5);

    painter.setPen(pen);
    painter.drawRect(QRect(80,120,200,100));



}
```

https://codeloop.org/qt5-qpainter-how-to-draw-rectangle/

- Draw Rectangle
  - Example



https://codeloop.org/qt5-qpainter-how-to-draw-rectangle/

Qt5 C++ GUI Development

- Draw Ellipse
  - Example

mainwindow.h

```
1  #ifndef MAINWINDOW_H
2  #define MAINWINDOW_H
3
4  #include <QMainWindow>
5  #include<QPainter>
6
7  namespace Ui {
8  class MainWindow;
9  }
10
11 class MainWindow : public QMainWindow
12 {
13     Q_OBJECT
14
15 public:
16     explicit MainWindow(QWidget *parent = 0);
17     ~MainWindow();
18
19
20     virtual void paintEvent(QPaintEvent *event);
21
22 private:
23     Ui::MainWindow *ui;
24 };
25
26 #endif // MAINWINDOW_H
```

https://codeloop.org/qt5-gui-qpainter-how-to-draw-ellipse/

Qt5 C++ GUI Development

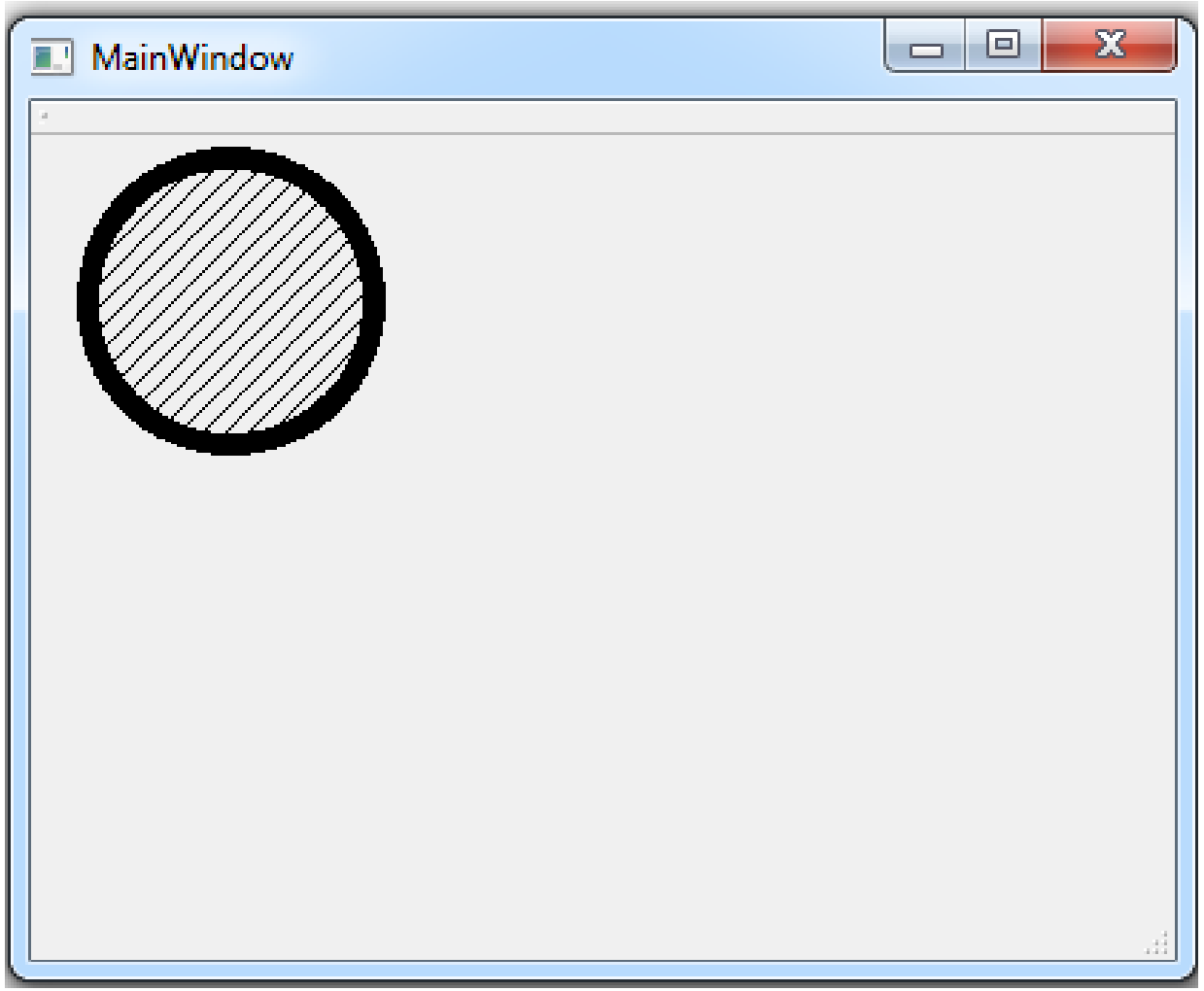
- Draw Ellipse
  - Example

mainwindow.cpp

```cpp
#include "mainwindow.h"
#include "ui_mainwindow.h"

MainWindow::MainWindow(QWidget *parent) :
    QMainWindow(parent),
    ui(new Ui::MainWindow)
{
    ui->setupUi(this);
}

MainWindow::~MainWindow()
{
    delete ui;
}

void MainWindow::paintEvent(QPaintEvent *event)
{

    QPainter ellipsePainter(this);

    ellipsePainter.setBrush(Qt::BDiagPattern);

    QPen pen;
    pen.setWidth(8);
    pen.setBrush(Qt::SolidPattern);
    ellipsePainter.setPen(pen);

    ellipsePainter.drawEllipse(QRect(20,20,100,100));

```

https://codeloop.org/qt5-gui-qpainter-how-to-draw-ellipse/

- Draw Ellipse
  - Example



https://codeloop.org/qt5-gui-qpainter-how-to-draw-ellipse/

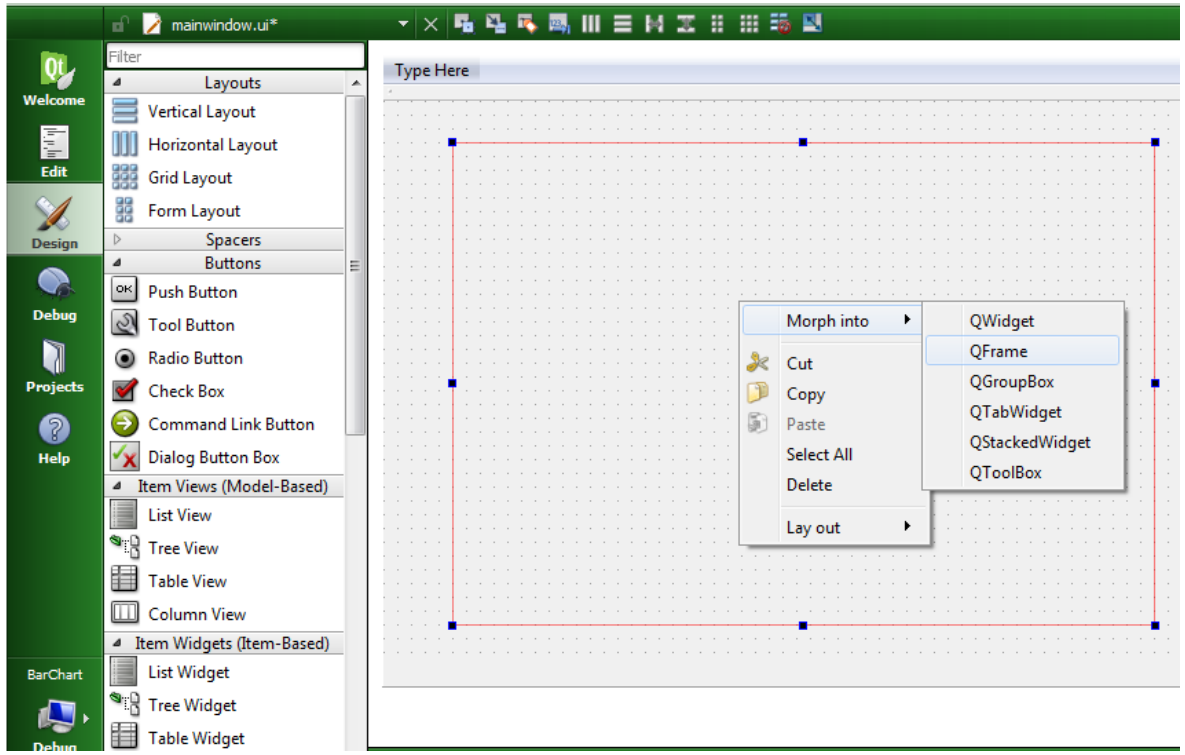
Qt5 C++ GUI Development

- Draw BarChart with QtChart
  - Example

```
1 QT       += core gui charts
```

```
#include <QMainWindow>
#include<QtCharts>
#include<QChartView>
#include<QBarSet>
#include<QBarSeries>
```

  - add Horizontal Layout in our gui window
  - right-click on the layout widget you just dragged to the central widget, and select *Morph into | QFrame*.



https://codeloop.org/qt5-tutorial-creating-barchart-with-qtchart/

- Draw BarChart with QtChart
  - Example

```
#include "mainwindow.h"
#include "ui_mainwindow.h"

MainWindow::MainWindow(QWidget *parent)
    : QMainWindow(parent)
    , ui(new Ui::MainWindow)
{
    ui->setupUi(this);

    QBarSet *set0 = new QBarSet("Bob");
    QBarSet *set1 = new QBarSet("Tom");
    QBarSet *set2 = new QBarSet("John");
    QBarSet *set3 = new QBarSet("Doe");
    QBarSet *set4 = new QBarSet("Ahmad");

    *set0 << 30 << 40 << 10 << 20 << 10 << 60;
    *set1 << 10 << 30 << 42 << 15 << 81 << 75;
    *set2 << 80 << 100 << 70 << 13 << 60 << 20;
    *set3 << 30 << 10 << 80 << 70 << 60 << 45;
    *set4 << 100 << 40 << 70 << 30 << 16 << 42;

    QBarSeries *series = new QBarSeries();
    series->append(set0);
    series->append(set1);
    series->append(set2);
    series->append(set3);
    series->append(set4);

    QChart *chart = new QChart();
    chart->addSeries(series);
    chart->setTitle("BarChart Example In Qt5 C++ ");
    chart->setAnimationOptions(QChart::SeriesAnimations);


    QStringList categories;
    categories << "Jan" << "Feb" << "Mar" << "Apr" << "May" << "Jun";
    QBarCategoryAxis *axis = new QBarCategoryAxis();
    axis->append(categories);
    chart->createDefaultAxes();
    chart->setAxisX(axis, series);


    QChartView *chartView = new QChartView(chart);
    chartView->setParent(ui->horizontalFrame);
}

MainWindow::~MainWindow()
{
    delete ui;
}
```

https://codeloop.org/qt5-tutorial-creating-barchart-with-qtchart/

- Draw BarChart with QtChart
  - Example

https://codeloop.org/qt5-tutorial-creating-barchart-with-qtchart/

- Draw BarChart with QtChart
  - Example

https://codeloop.org/qt5-tutorial-creating-barchart-with-qtchart/

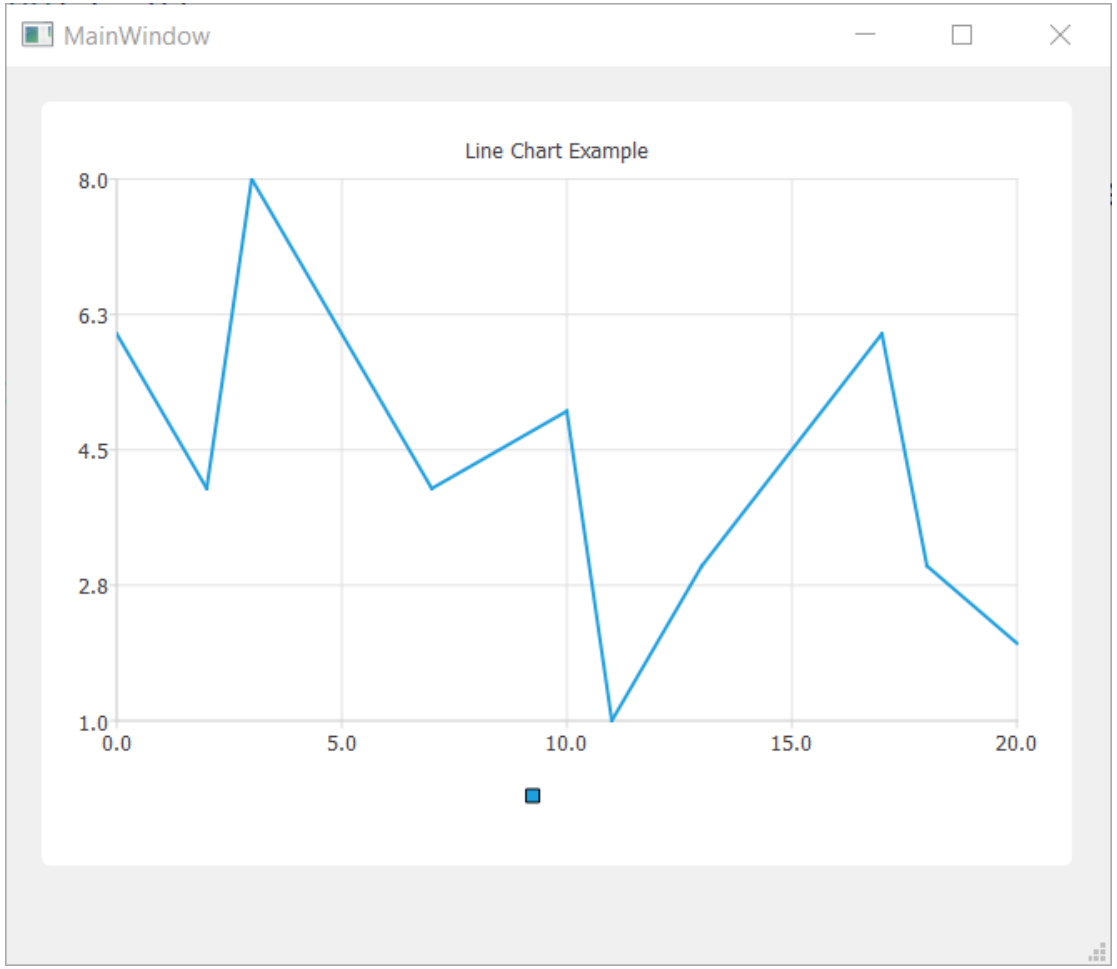
- Draw LineChart with QtChart
  - Example

```
#include <QMainWindow>
#include<QtCharts>
#include<QChartView>
#include<QLineSeries>
```

```
MainWindow::MainWindow(QWidget *parent)
    : QMainWindow(parent)
    , ui(new Ui::MainWindow)
{
    ui->setupUi(this);

    QLineSeries *series = new QLineSeries();

    series->append(0, 6);
    series->append(2, 4);
    series->append(3, 8);
    series->append(7, 4);
    series->append(10, 5);

    *series << QPointF(11, 1) << QPointF(13, 3) << QPointF(17, 6) << QPointF(18, 3)
            << QPointF(20, 2);

    QChart *chart = new QChart();
    //chart->legend()->hide();
    chart->addSeries(series);
    chart->createDefaultAxes();
    chart->setTitle("Line Chart Example");

    chart->legend()->setVisible(true);
    chart->legend()->setAlignment(Qt::AlignBottom);

    QChartView *chartView = new QChartView(chart);
    chartView->setRenderHint(QPainter::Antialiasing);
    chartView->setParent(ui->horizontalFrame);
}
```

https://codeloop.org/qt5-tutorial-creating-linechart-with-qtchart/

- Draw LineChart with QtChart
  - Example



https://codeloop.org/qt5-tutorial-creating-linechart-with-qtchart/

Qt5 C++ GUI Development

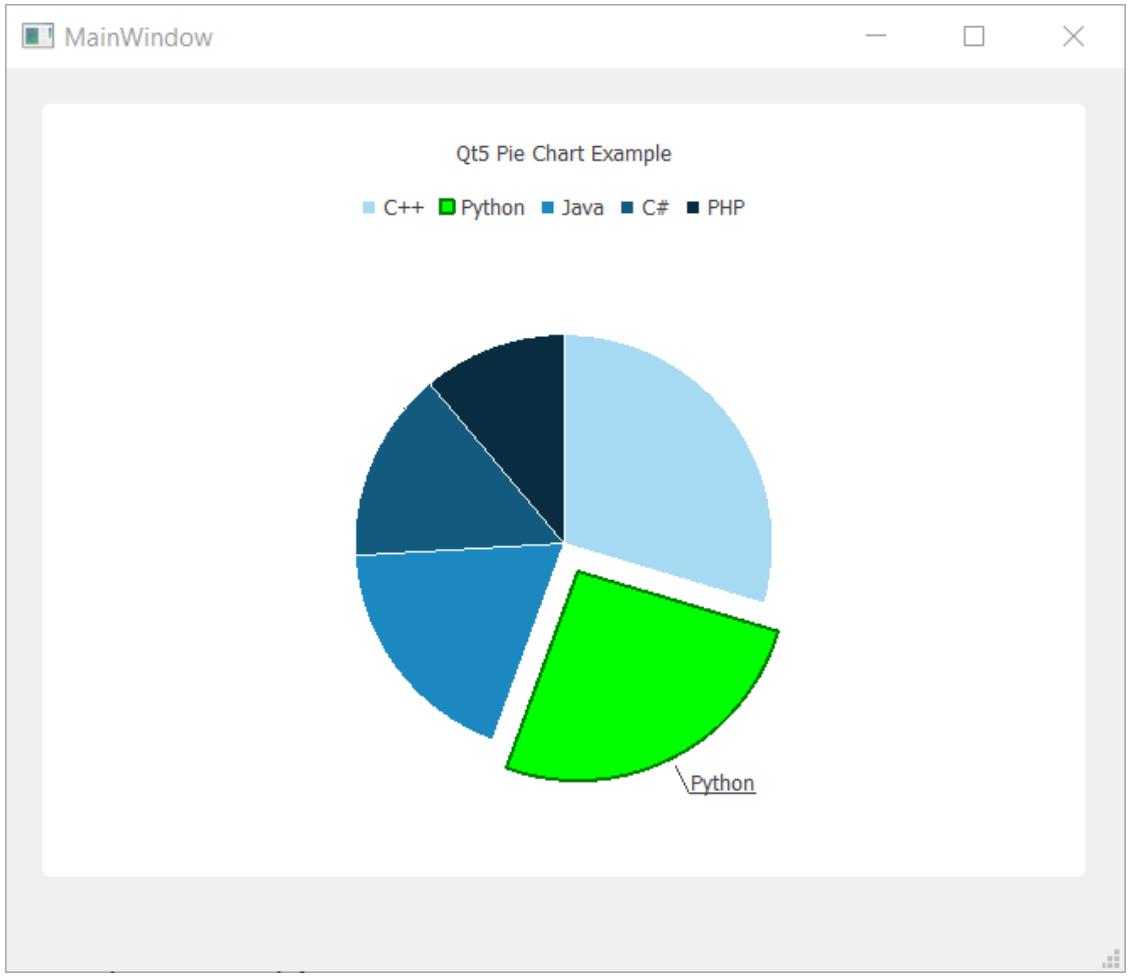
- Draw PieChart with QtChart
  - Example

```
#include <QMainWindow>
#include<QtCharts>
#include<QChartView>
#include<QPieSeries>
#include<QPieSlice>
```

```
MainWindow::MainWindow(QWidget *parent)
    : QMainWindow(parent)
    , ui(new Ui::MainWindow)
{
    ui->setupUi(this);

    QPieSeries *series = new QPieSeries();

    series->append("C++", 80);
    series->append("Python", 70);
    series->append("Java", 50);
    series->append("C#", 40);
    series->append("PHP", 30);


    QPieSlice *slice = series->slices().at(1);
    slice->setExploded(true);
    slice->setLabelVisible(true);
    slice->setPen(QPen(Qt::darkGreen, 2));
    slice->setBrush(Qt::green);


    QChart *chart = new QChart();
    chart->addSeries(series);
    chart->setTitle("Qt5 Pie Chart Example");


    QChartView *chartview = new QChartView(chart);
    chartview->setParent(ui->horizontalFrame);
}
```

https://codeloop.org/qt5-tutorial-creating-piechart-with-qtchart/

- Draw PieChart with QtChart
  - Example



https://codeloop.org/qt5-tutorial-creating-piechart-with-qtchart/

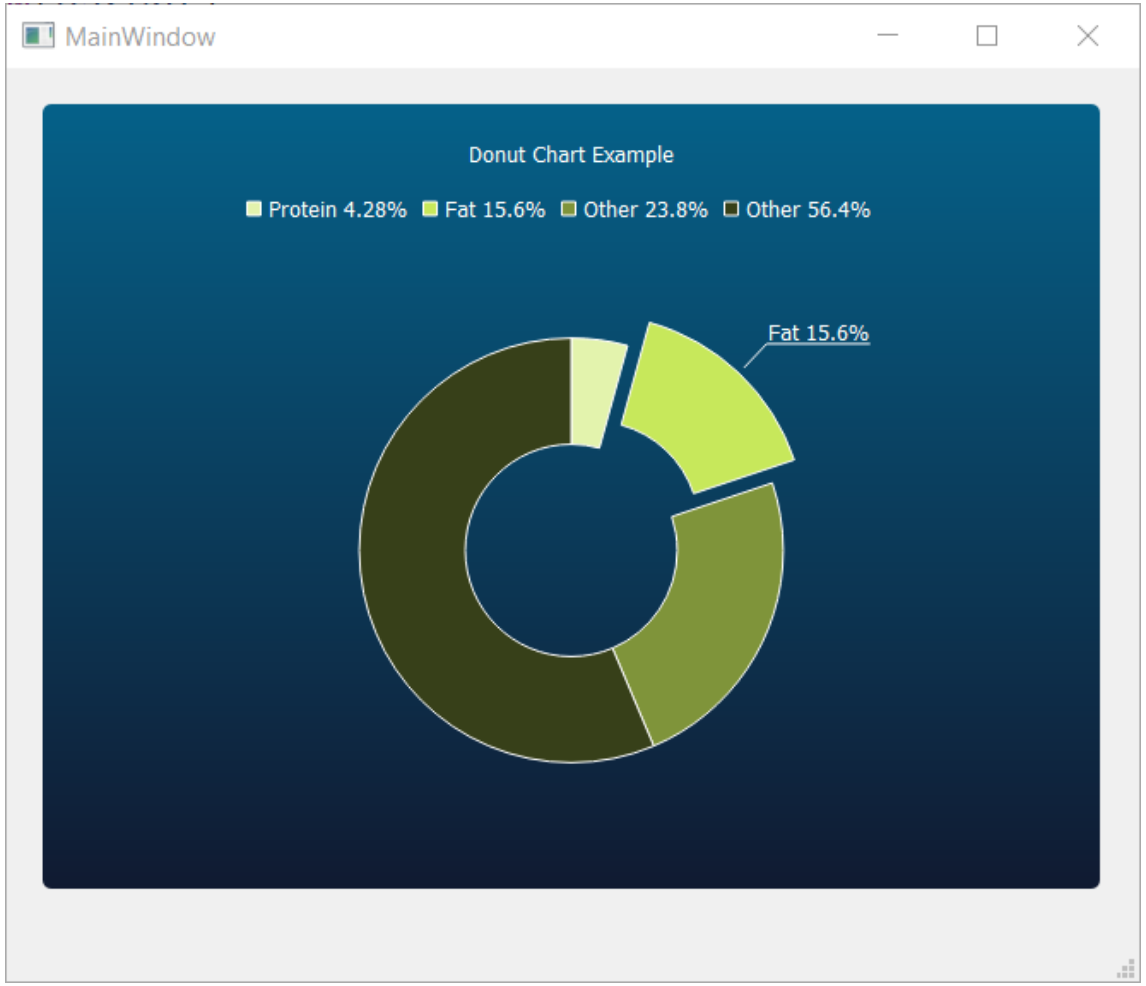
- Draw DonutChart with QtChart
  - Example

```
#include <QMainWindow>
#include<QtCharts>
#include<QChartView>
#include<QPieSeries>
#include<QPieSlice>
```

```
MainWindow::MainWindow(QWidget *parent)
    : QMainWindow(parent)
    , ui(new Ui::MainWindow)
{
    ui->setupUi(this);

    QPieSeries *series = new QPieSeries();
    series->setHoleSize(0.35);
    series->append("Protein 4.28%", 4.28);
    QPieSlice *slice = series->append("Fat 15.6%", 15.6);

    slice->setExploded();
    slice->setLabelVisible();
    series->append("Other 23.8%", 23.8);
    series->append("Other 56.4%", 56.4);

    QChart *chart = new QChart();
    chart->addSeries(series);
    chart->setAnimationOptions(QChart::SeriesAnimations);
    chart->setTitle("Donut Chart Example");
    chart->setTheme(QChart::ChartThemeBlueCerulean);

    QChartView *chartview = new QChartView(chart);
    chartview->setRenderHint(QPainter::Antialiasing);

    chartview->setParent(ui->horizontalFrame);
}
```

https://codeloop.org/qt5-tutorial-creating-donutchart-with-qtchart/

- Draw DonutChart with QtChart
  - Example



https://codeloop.org/qt5-tutorial-creating-donutchart-with-qtchart/

TRƯỜNG ĐẠI HỌC BÁCH KHOA HÀ NỘI

KỸ THUẬT LẬP TRÌNH HỆ CƠ ĐIỆN TỬ

Programming Engineering in Mechatronics

Giảng viên: TS. Nguyễn Thành Hùng

Đơn vị: Bộ môn Cơ điện tử, Viện Cơ khí

Hà Nội, 2018

Chapter V. Hardware Interface Programming

- 1. Introduction
- 2. Serial Port
- 3. Read Data
- 4. Send Data
- 5. Real-Time Data Transfer
- 6. Digital Image Processing

1. Introduction

- A hardware interface specifies the plugs, sockets, cables and electrical signals that pass through each line between the CPU and a peripheral device or communications network.

- The CPU socket on the motherboard determines which CPU chips can be used in the computer.

- Peripheral cards, such as a high-end graphics cards, plug into the bus on the motherboard. The most common buses are PCI and PCI Express.

https://www.pcmag.com/encyclopedia/term/51988/standards-hardware-interfaces

1. Introduction

- The most widely used hardware interface for attaching external devices to computers is USB. It connects printers, cameras, music players, flash drives and auxiliary hard and optical drives.
- FireWire is also used for camcorders and hard disks.
- In addition, SATA is a common hard drive and optical drive interface.
- The GPIB IEEE 488 standard is used for process control instruments.
- The de facto standard for connecting devices to local networks (LANs) is Ethernet, which is also used to hook up a cable or DSL modem.

https://www.pcmag.com/encyclopedia/term/51988/standards-hardware-interfaces

2. Serial Port

- Serial ports provide an easy way to communicate between many types of hardware and your computer.
- They are relatively simple to use and are very common among peripherals and especially DIY projects.
- Many platforms such as Arduino have built in serial communication so they are really easy to set up and use.
- Many times you may want your project to communicate with your computer in order to have a cool interactive output, a neat sensor that passes data to your computer, or anything else you could possibly dream up.

2. Serial Port

❖ Visual Studio Serial Port control

- Visual Studio has a control that performs serial input and output. It is the SerialPort control and found in the Components toolbox tab.

- It is a non visual control. Its main properties are set to a common communications rate and are: BaudRate: 9600, DataBits: 8, Parity: None, PortName: COM1, StopBits: One.
- Its main event is: DataReceived which occurs when data is received from the port.

2. Serial Port

Visual Studio Serial Port control: Constructors

Description	
SerialPort()	Initializes a new instance of the SerialPort class.
SerialPort(IContainer^)	Initializes a new instance of the SerialPort class using the specified IContainer object.
SerialPort(String^)	Initializes a new instance of the SerialPort class using the specified port name.
SerialPort(String^, Int32)	Initializes a new instance of the SerialPort class using the specified port name and baud rate.
SerialPort(String^, Int32, Parity)	Initializes a new instance of the SerialPort class using the specified port name, baud rate, and parity bit.
SerialPort(String^, Int32, Parity, Int32)	Initializes a new instance of the SerialPort class using the specified port name, baud rate, parity bit, and data bits.
SerialPort(String^, Int32, Parity, Int32, StopBits)	Initializes a new instance of the SerialPort class using the specified port name, baud rate, parity bit, data bits, and stop bit.

2. Serial Port

Visual Studio Serial Port control: Properties

BaseStream	Gets the underlying Stream object for a SerialPort object.
BaudRate	Gets or sets the serial baud rate.
BreakState	Gets or sets the break signal state.
BytesToRead	Gets the number of bytes of data in the receive buffer.
BytesToWrite	Gets the number of bytes of data in the send buffer.
CanRaiseEvents	Gets a value indicating whether the component can raise an event. (Inherited from Component)
CDHolding	Gets the state of the Carrier Detect line for the port.
Container	Gets the IContainer that contains the Component. (Inherited from Component)
CtsHolding	Gets the state of the Clear-to-Send line.
DataBits	Gets or sets the standard length of data bits per byte.
DesignMode	Gets a value that indicates whether the Component is currently in design mode. (Inherited from Component)
DiscardNull	Gets or sets a value indicating whether null bytes are ignored when transmitted between the port and the receive buffer.
DsrHolding	Gets the state of the Data Set Ready (DSR) signal.
DtrEnable	Gets or sets a value that enables the Data Terminal Ready (DTR) signal during serial communication.
Encoding	Gets or sets the byte encoding for pre- and post-transmission conversion of text.
Events	Gets the list of event handlers that are attached to this Component. (Inherited from Component)

2. Serial Port

Visual Studio Serial Port control: Properties

Handshake	Gets or sets the handshaking protocol for serial port transmission of data using a value from Handshake.
IsOpen	Gets a value indicating the open or closed status of the SerialPort object.
NewLine	Gets or sets the value used to interpret the end of a call to the ReadLine() and WriteLine(String) methods.
Parity	Gets or sets the parity-checking protocol.
ParityReplace	Gets or sets the byte that replaces invalid bytes in a data stream when a parity error occurs.
PortName	Gets or sets the port for communications, including but not limited to all available COM ports.
ReadBufferSize	Gets or sets the size of the SerialPort input buffer.
ReadTimeout	Gets or sets the number of milliseconds before a time-out occurs when a read operation does not finish.
ReceivedBytesThreshold	Gets or sets the number of bytes in the internal input buffer before a DataReceived event occurs.
RtsEnable	Gets or sets a value indicating whether the Request to Send (RTS) signal is enabled during serial communication.
Site	Gets or sets the ISite of the Component. (Inherited from Component)
StopBits	Gets or sets the standard number of stopbits per byte.
WriteBufferSize	Gets or sets the size of the serial port output buffer.
WriteTimeout	Gets or sets the number of milliseconds before a time-out occurs when a write operation does not finish.

2. Serial Port

Visual Studio Serial Port control: Methods

Close()	Closes the port connection, sets the IsOpen property to false, and disposes of the internal Stream object.
CreateObjRef(Type)	Creates an object that contains all the relevant information required to generate a proxy used to communicate with a remote object. (Inherited from MarshalByRefObject)
DiscardInBuffer()	Discards data from the serial driver's receive buffer.
DiscardOutBuffer()	Discards data from the serial driver's transmit buffer.
Dispose()	Releases all resources used by the Component. (Inherited from Component)
Dispose(Boolean)	Releases the unmanaged resources used by the SerialPort and optionally releases the managed resources.
Equals(Object)	Determines whether the specified object is equal to the current object. (Inherited from Object)
GetHashCode()	Serves as the default hash function. (Inherited from Object)
GetLifetimeService()	Retrieves the current lifetime service object that controls the lifetime policy for this instance. (Inherited from MarshalByRefObject)
GetPortNames()	Gets an array of serial port names for the current computer.
GetService(Type)	Returns an object that represents a service provided by the Component or by its Container. (Inherited from Component)
GetType()	Gets the Type of the current instance. (Inherited from Object)
InitializeLifetimeService()	Obtains a lifetime service object to control the lifetime policy for this instance. (Inherited from MarshalByRefObject)
MemberwiseClone()	Creates a shallow copy of the current Object. (Inherited from Object)

2. Serial Port

Visual Studio Serial Port control: Methods

MemberwiseClone(Boolean)	Creates a shallow copy of the current MarshalByRefObject object. (Inherited from MarshalByRefObject)
Open()	Opens a new serial port connection.
Read(Byte[], Int32, Int32)	Reads a number of bytes from the SerialPort input buffer and writes those bytes into a byte array at the specified offset.
Read(Char[], Int32, Int32)	Reads a number of characters from the SerialPort input buffer and writes them into an array of characters at a given offset.
ReadByte()	Synchronously reads one byte from the SerialPort input buffer.
ReadChar()	Synchronously reads one character from the SerialPort input buffer.
ReadExisting()	Reads all immediately available bytes, based on the encoding, in both the stream and the input buffer of the SerialPort object.
ReadLine()	Reads up to the NewLine value in the input buffer.
ReadTo(String)	Reads a string up to the specified value in the input buffer.
ToString()	Returns a String containing the name of the Component, if any. This method should not be overridden. (Inherited from Component)
Write(Byte[], Int32, Int32)	Writes a specified number of bytes to the serial port using data from a buffer.
Write(Char[], Int32, Int32)	Writes a specified number of characters to the serial port using data from a buffer.
Write(String)	Writes the specified string to the serial port.
WriteLine(String)	Writes the specified string and the NewLine value to the output buffer.

2. Serial Port

Visual Studio Serial Port control: Events

DataReceived	Indicates that data has been received through a port represented by the SerialPort object.
Disposed	Occurs when the component is disposed by a call to the Dispose() method. (Inherited from Component)
ErrorReceived	Indicates that an error has occurred with a port represented by a SerialPort object.
PinChanged	Indicates that a non-data signal event has occurred on the port represented by the SerialPort object.

2. Serial Port

❖ Visual Studio Serial Port control: Remarks

- Use this class to control a serial port file resource. This class provides synchronous and event-driven I/O, access to pin and break states, and access to serial driver properties. Additionally, the functionality of this class can be wrapped in an internal Stream object, accessible through the BaseStream property, and passed to classes that wrap or use streams.
- The SerialPort class supports the following encodings: ASCIIEncoding, UTF8Encoding, UnicodeEncoding, UTF32Encoding, and any encoding defined in mscorlib.dll where the code page is less than 50000 or the code page is 54936. You can use alternate encodings, but you must use the ReadByte or Write method and perform the encoding yourself.

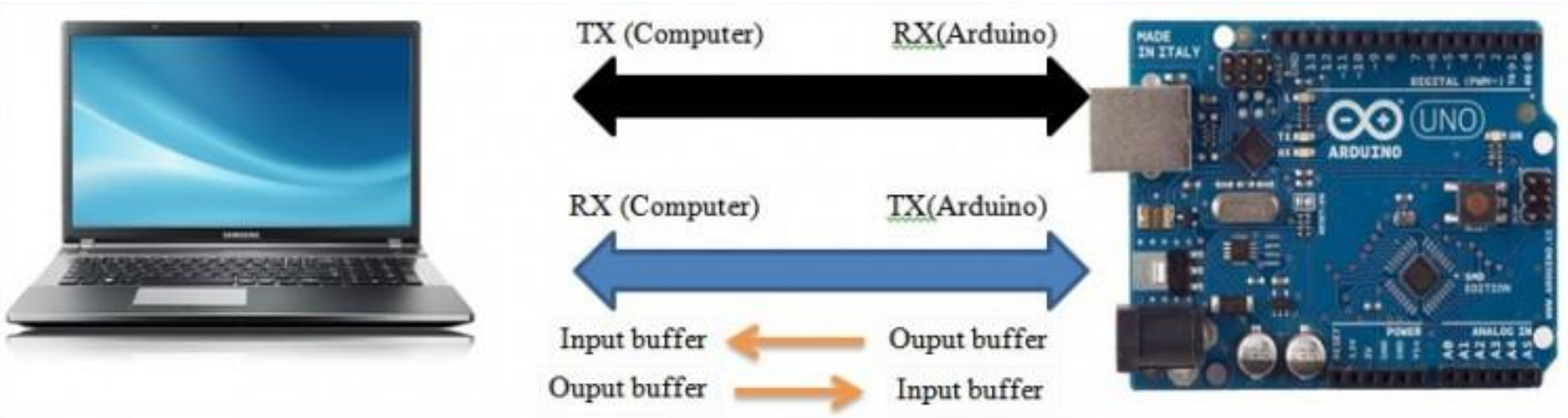
2. Serial Port

❖ Visual Studio Serial Port control: Remarks

- You use the GetPortNames method to retrieve the valid ports for the current computer.
- If a SerialPort object becomes blocked during a read operation, do not abort the thread. Instead, either close the base stream or dispose of the SerialPort object.

2. Serial Port

- Serial Port data over USB



2. Serial Port

- Serial Port data over USB

USB

Comm port


```
serialPort1.Open();
serialPort1.PortName = L"COM4";
serialPort1.Read(byteArray, 0, 1);
serialPort1.Write(byteArray, 0, 1);
serialPort1.Close();
```

system.io.ports.serialport

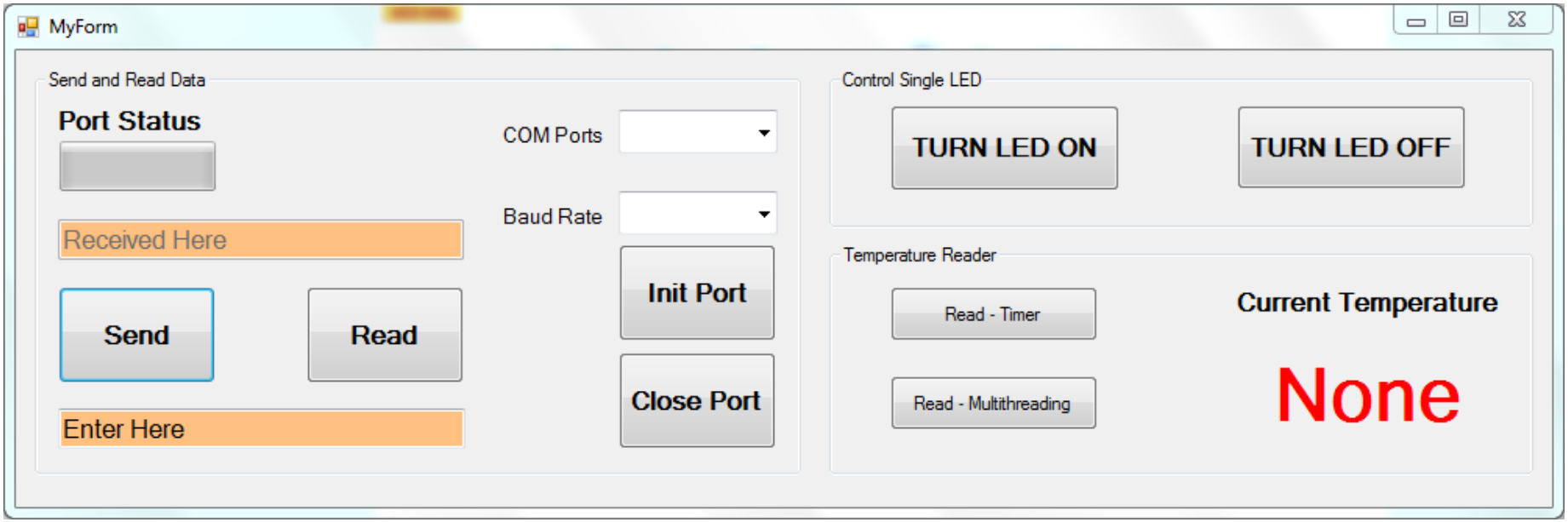
```
Serial.begin(9600);
Serial.available();
Serial.read();
Serial.write(value);
```

Arduino Serial Port Reference

We need to use serial port objects in order to communicate with each device

2. Serial Port

- Visual Studio Serial Port control in C++/CLI



2. Serial Port

❖ Visual Studio Serial Port control in C++/CLI

- Find Ports

```
public ref class MyForm : public System::Windows::Forms::Form
{
public:
    MyForm(void)
    {
        InitializeComponent();
        //
        //TODO: Add the constructor code here
        findPorts();
        textBoxReceivedData->Enabled = false;
        //
    }

 // find available ports
 private: void findPorts(void)
 {
     // get port names
     array<Object^>^ objectArray = SerialPort::GetPortNames();
     // add string array to combobox
     this->comboBoxPorts->Items->AddRange(objectArray);
 }
```

2. Serial Port

❖ Visual Studio Serial Port control in C++/CLI

- Initialize Port

```
// Init button-----------------------------------
private: System::Void buttonInitialPort_Click(System::Object^  sender, System::EventArgs^  e) {
    this->textBoxReceivedData->Text = String::Empty;
    if (this->comboBoxPorts->Text == String::Empty || this->comboBoxBaudRate->Text == String::Empty)
        this->textBoxSendData->Text = "Please Select Port Settings";
    else {
        try {
            // make sure port isn't open
            if (!this->serialPort1->IsOpen) {
                this->serialPort1->PortName = this->comboBoxPorts->Text;
                this->serialPort1->BaudRate = Int32::Parse(this->comboBoxBaudRate->Text);
                this->textBoxSendData->Text = "Enter Message Here";
                //open serial port
                this->serialPort1->Open();
                this->progressBarPortStatus->Value = 100;
            }
            else
                this->textBoxSendData->Text = "Port isn't openned";
        }
        catch (UnauthorizedAccessException^) {
            this->textBoxSendData->Text = "UnauthorizedAccess";
        }
    }
}
```

2. Serial Port

❖ Visual Studio Serial Port control in C++/CLI

- Close Port

```
    // close port button
private: System::Void buttonClosePort_Click(System::Object^  sender, System::EventArgs^  e) {
    //close serialPort
    this->serialPort1->Close();
    // update progress bar
    this->progressBarPortStatus->Value = 0;
    // Enable read button
    this->buttonReadData->Enabled = true;
    // Enable the init button
    this->buttonInitialPort->Enabled = true;
}
```

3. Read Data

❖ Visual Studio Serial Port control in C++/CLI

- Read Data

```
    // Read button ------------------------------------
    //this will start the asyn for backgroundwork
private: System::Void buttonReadData_Click(System::Object^  sender, System::EventArgs^  e) {
    // check if port is ready for reading
    if (this->serialPort1->IsOpen) {
        // Reset the text in the result label.
        this->textBoxReceivedData->Text = String::Empty;

        // this will read manually
        try {
            this->textBoxReceivedData->Text = this->serialPort1->ReadLine();
        }
        catch (TimeoutException^) {
            this->textBoxReceivedData->Text = "Timeout Exception";
        }
        // Disable the init button
        // the asynchronous operation is done.
        this->buttonInitialPort->Enabled = false;
    }
    else
        // give error warning
        this->textBoxReceivedData->Text = "Port Not Opened";
}
```

3. Read Data

❖ Example: Read data from Arduino

- Arduino code

```
void setup() {
  // put your setup code here, to run once:
  Serial.begin(9600);
}

void loop() {
  // put your main code here, to run repeatedly:
  Serial.println("Giao tiep Arduino and PC");
  delay(200);
}
```

4. Send Data

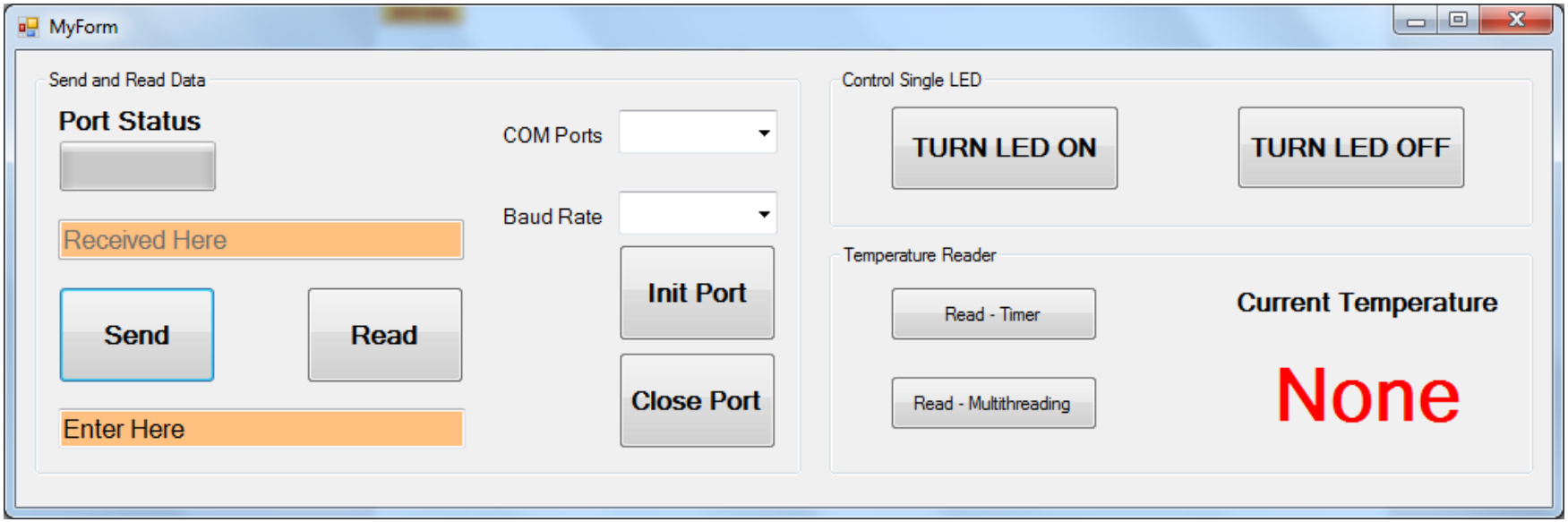
- Visual Studio Serial Port control in C++/CLI
- Send Data

```
    // Send button
private: System::Void buttonSendData_Click(System::Object^  sender, System::EventArgs^  e) {
    // add sender name
    String^ name = this->serialPort1->PortName;
    // grab text and store in send buffer
    String^ message = this->textBoxSendData->Text;
    // write to serial
    if (this->serialPort1->IsOpen)
        //this->_serialPort->WriteLine(String::Format("<{0}>: {1}",name,message));
        this->serialPort1->WriteLine(message);
    else
        this->textBoxSendData->Text = "Port Not Opened";
}
```

4. Send Data

❖ Example: Turn led ON/OFF

- C++/CLI code



4. Send Data

❖ Example: Turn led ON/OFF

- C++/CLI code

```
// grab text and store in send
buffer
String^ message = "ON";
// write to serial
if (this->serialPort1->IsOpen)
this->serialPort1-
>Write(message);
else
this->textBoxSendData->Text =
"Port Not Opened";
```

Turn led ON

```
// grab text and store in send
buffer
String^ message = "OFF";
// write to serial
if (this->serialPort1->IsOpen)
this->serialPort1-
>Write(message);
else
this->textBoxSendData->Text =
"Port Not Opened";
```

Turn led OFF

4. Send Data

❖ Example: Turn led ON/OFF

- Arduino code

```
String x;//khai bao bien dang chuoi
// do ki tu nhap tu ban phim mang gia tri theo bang ma ASCII
int led = 11;

void setup() {
  // put your setup code here, to run once:
  Serial.begin(9600);
  // khai bao ngo ra chan 11
  pinMode(led, OUTPUT);
  // xuat ra monitor cau lenh
  //Serial.println("Nhap trang thai led ( on or off ) :");
}
```

4. Send Data

❖ Example: Turn led ON/OFF

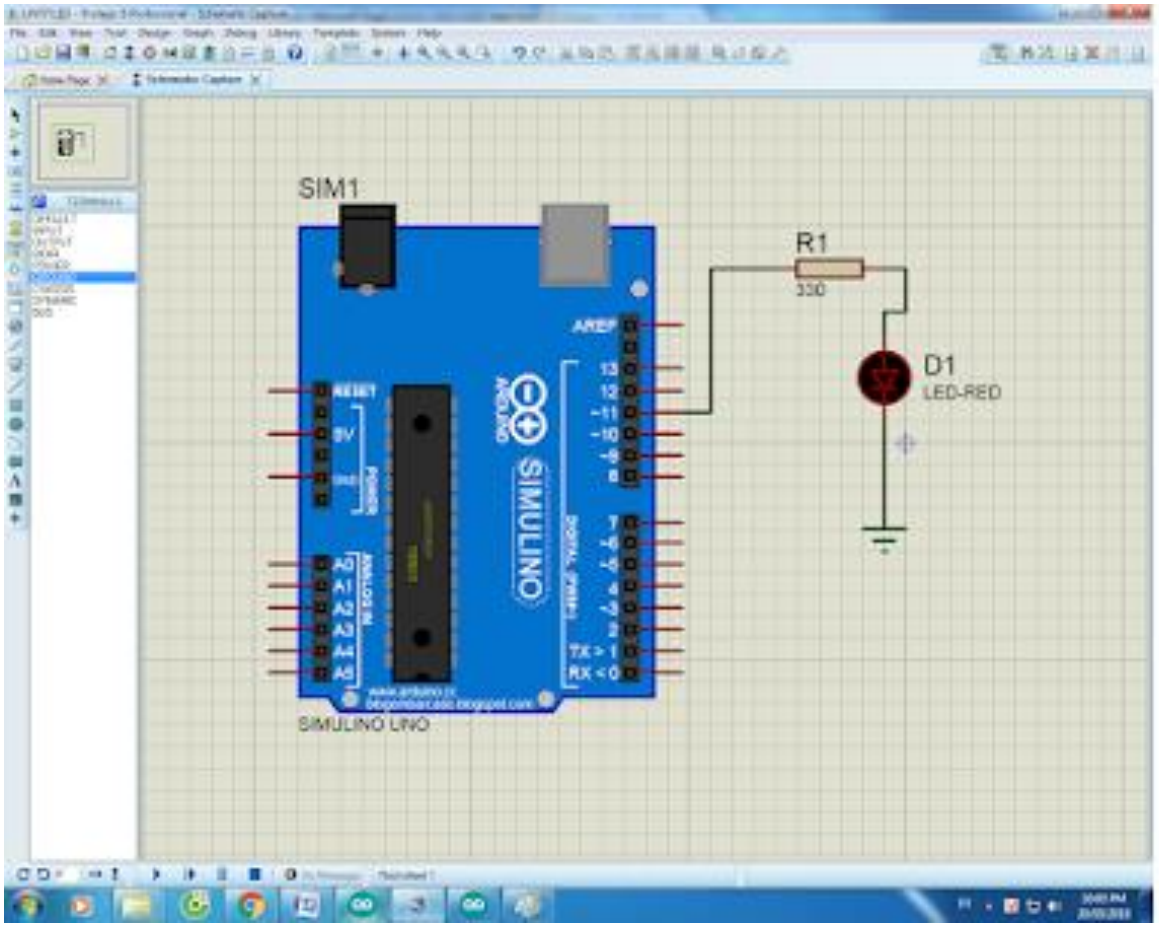
- Arduino code

```
void loop() {
  // put your main code here, to run repeatedly:
  if(Serial.available())// neu co du lieu truyen toi thi thuc hien cac lenh sau
  {
    x = Serial.readString();// gan du lieu truyen toi vao chuoi x
    //Serial.println(x);
    if(x == "ON" ){ //neu x = ON thi chan 11 xuat muc cao => led sang
      digitalWrite(11, HIGH);
    }
    if(x == "OFF" ){//neu x = OFF thi chan 11 xuat muc thap => led tat
      digitalWrite(11, LOW);
    }
  }
}
```

4. Send Data

❖ Example: Turn led ON/OFF

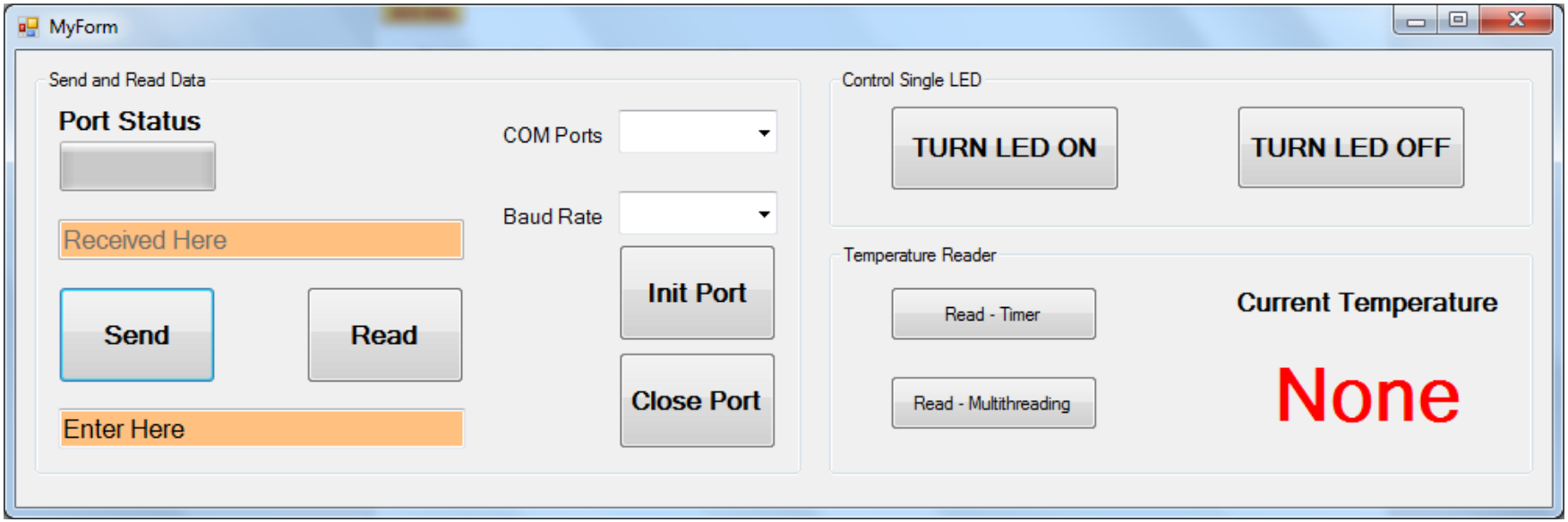
- Circuit Diagram



5. Real-Time Data Transfer

❖ Serial Data Received Event Handler

- Example: Read and Display Temperature



5. Real-Time Data Transfer

❖ Serial Data Received Event Handler

- Example: Read and Display Temperature

- C++/CLI Code

```
private: System::Void serialPort1_DataReceived(System::Object^  sender,
System::IO::Ports::SerialDataReceivedEventArgs^  e) {
// check if port is ready for reading
if (this->serialPort1->IsOpen) {
// Reset the text in the result label.
this->labelTemperature->Text = String::Empty;

// this will read manually
try {
this->labelTemperature->Text = this->serialPort1->ReadLine();
}
catch (TimeoutException^) {
this->labelTemperature->Text = "None";
}
// Disable the init button
// the asynchronous operation is done.
this->buttonInitialPort->Enabled = false;
}
else
// give error warning
this->textBoxReceivedData->Text = "Port Not Opened";
}
```

5. Real-Time Data Transfer

❖ Serial Data Received Event Handler

- Example: Read and Display Temperature

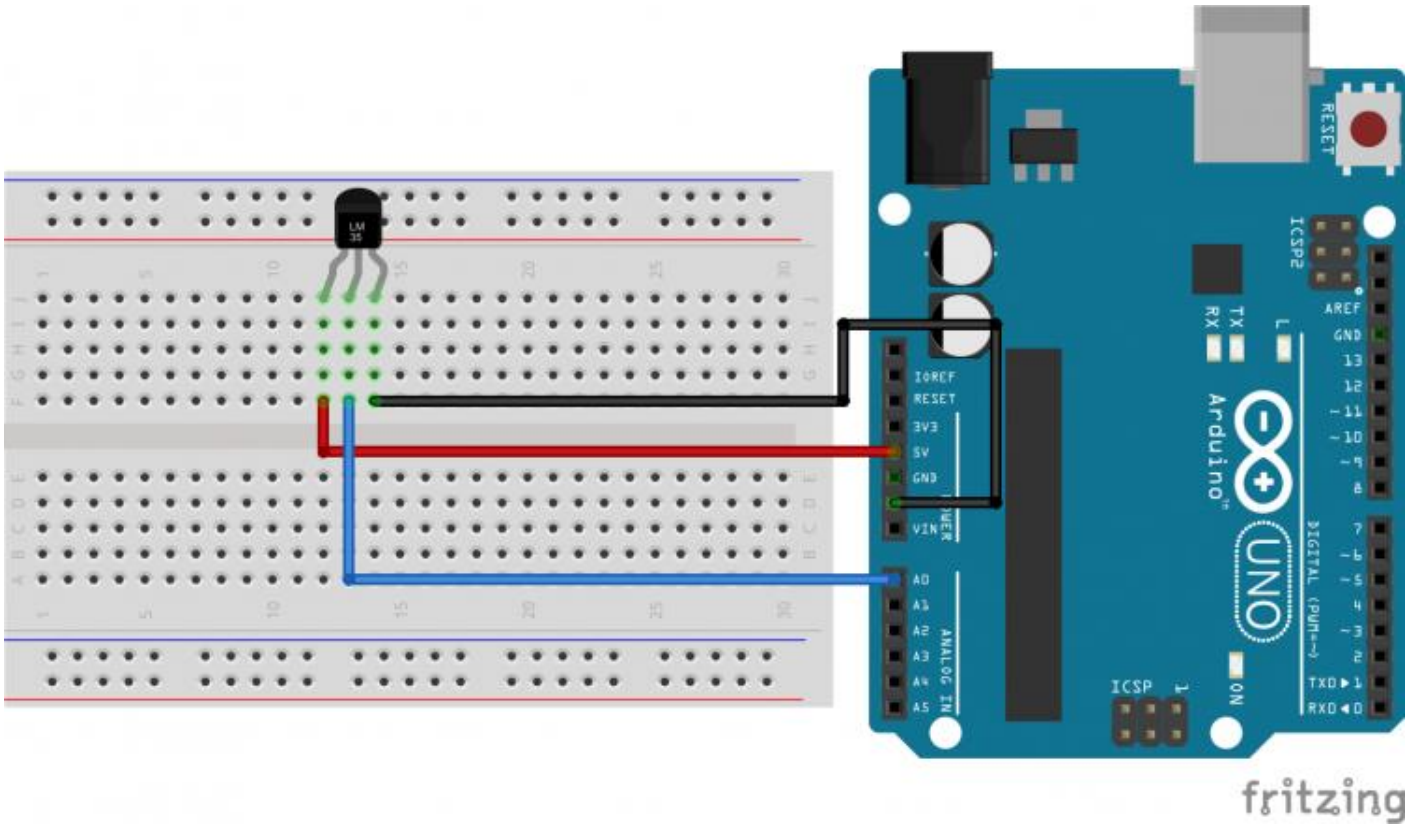
- Arduino Coding

```
int sensorPin = A0;// chân analog kết nối tới cảm biến LM35

void setup() {
  // put your setup code here, to run once:
  Serial.begin(9600);  //Khởi động Serial ở mức baudrate 9600
  // Bạn không cần phải pinMode cho các chân analog trước khi dùng nó
}

void loop() {
  // put your main code here, to run repeatedly:
  //đọc giá trị từ cảm biến LM35
  int reading = analogRead(sensorPin);

  //tính ra giá trị hiệu điện thế (đơn vị Volt) từ giá trị cảm biến
  float voltage = reading * 5.0 / 1024.0;

  // ở trên mình đã giới thiệu, cứ mỗi 10mV = 1 độ C.
  // Vì vậy nếu biến voltage là biến lưu hiệu điện thế (đơn vị Volt)
  // thì ta chỉ việc nhân voltage cho 100 là ra được nhiệt độ!

  float temp = voltage * 100.0;

  Serial.println(temp);
  delay(1000);//đợi 1 giây cho lần đọc tiếp theo
}
```

5. Real-Time Data Transfer

❖ Serial Data Received Event Handler

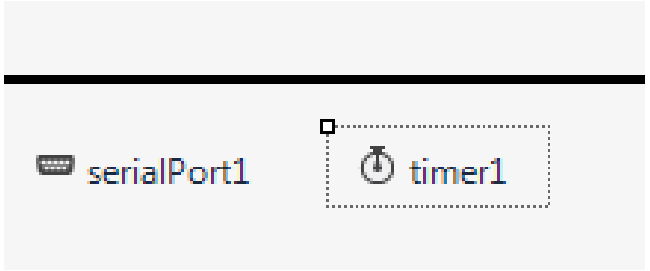
- Example: Read and Display Temperature
- Circuit Diagram



5. Real-Time Data Transfer

❖ Timer

- Example: Read and Display Temperature



- C++/CLI Code

```
private:
    /// <summary>
    /// Required designer variable.
    String^ currTemp;
    /// </summary>

private: System::Void MyForm_Load(System::Object^  sender, System::EventArgs^  e) {
    timer1->Enabled = false;
    currTemp = "None";
}
private: System::Void buttonUsingTimer_Click(System::Object^  sender, System::EventArgs^  e) {
    timer1->Enabled = true;
}
private: System::Void MyForm_FormClosed(System::Object^  sender,
  System::Windows::Forms::FormClosedEventArgs^  e) {
    timer1->Enabled = false;
}
```

5. Real-Time Data Transfer

❖ Timer

- Example: Read and Display Temperature

- C++/CLI Code

```
delegate void SetLabelTextDelegate(String^ text, Label^ label);
void SetLabelText(String^ text, Label^ label)
{
    // InvokeRequired required compares the thread ID of the
    // calling thread to the thread ID of the creating thread.
    // If these threads are different, it returns true.
    if (label->InvokeRequired)
    {
        SetLabelTextDelegate^ d = gcnew SetLabelTextDelegate(this, &MyForm::SetLabelText);
        this->Invoke(d, gcnew array<Object^> { text, label });
    }
    else
    {
        label->Text = text;
    }
}
```

5. Real-Time Data Transfer

❖ Timer

- Example: Read and Display Temperature

- C++/CLI Code

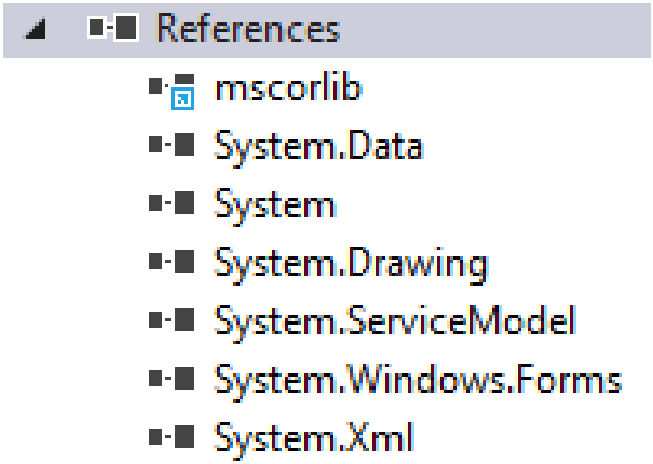
```
private: System::Void timer1_Tick(System::Object^  sender, System::EventArgs^  e) {
    // check if port is ready for reading
    if (this->serialPort1->IsOpen) {
        // Reset the text in the result label.
        SetLabelText(String::Empty, labelTemperature);

        // this will read manually
        try {
            currTemp = serialPort1->ReadLine();
            SetLabelText(currTemp, labelTemperature);
        }
        catch (TimeoutException^) {
            SetLabelText(currTemp, labelTemperature);
        }
        // Disable the init button
        // the asynchronous operation is done.
        this->buttonInitialPort->Enabled = false;
    }
    else
        SetLabelText(currTemp, labelTemperature);
}
```

5. Real-Time Data Transfer

❖ Multithreading

- Example: Read and Display Temperature

- C++/CLI Code




```
using namespace System::ServiceModel;
using namespace System::IO;
using namespace System::IO::Ports;  //add this,represents a serial port resource
using namespace System::Threading;
using namespace System::Threading::Tasks;
using namespace System::Diagnostics;
```

5. Real-Time Data Transfer

❖ Multithreading

- Example: Read and Display Temperature

- C++/CLI Code

```
private: void receive_serial_stream()
{
    while (true)
    {
        // check if port is ready for reading
        if (this->serialPort1->IsOpen) {
            // Reset the text in the result label.
            SetLabelText(String::Empty, labelTemperature);

            // this will read manually
            try {
                currTemp = serialPort1->ReadLine();
                SetLabelText(currTemp, labelTemperature);
            }
            catch (TimeoutException^) {

            }
            //Set poll frequency to 2Hz
            Thread::Sleep(500);
        }
        else
        {
            Thread::Sleep(1000);
        }
    }
}
```

5. Real-Time Data Transfer

❖ Multithreading

- Example: Read and Display Temperature

- C++/CLI Code

```
private: System::Void buttonUsingMultithreading_Click(System::Object^  sender,
  System::EventArgs^  e) {
    Thread^ get_serial_stream = gcnew Thread(gcnew ThreadStart(this,
      &MyForm::receive_serial_stream));
    get_serial_stream->IsBackground = true;
    get_serial_stream->Start();
}
```

KỸ THUẬT LẬP TRÌNH HỆ CƠ ĐIỆN TỬ

Programming Engineering in Mechatronics

Giảng viên: TS. Nguyễn Thành Hùng

Đơn vị: NCM Robot, Khoa Cơ điện tử, Trường Cơ khí

Hà Nội, 2022

Chapter V. Hardware Interface Programming

- 1. Giới thiệu
- 2. Các chuẩn giao tiếp và giao thức truyền thông công nghiệp
- 3. Truyền thông nối tiếp với Qt
- 4. Đa luồng trong Qt
- 5. Lập trình giao tiếp với Arduino
- 6. Lập trình giao tiếp với camera

1. Giới thiệu

❖ *Giao diện* là sự kết nối và tương tác giữa phần cứng, phần mềm và người dùng.

➢ Người dùng "giao tiếp" với phần mềm.

➢ Phần mềm "giao tiếp" với phần cứng và phần mềm khác.

➢ Phần cứng "giao tiếp" với phần cứng khác.

➢ Giao diện phải được thiết kế, phát triển, thử nghiệm và thiết kế lại.

1. Giới thiệu

- *Giao diện phần cứng*: là phích cắm, ổ cắm, cáp và tín hiệu điện truyền qua chúng. Ví dụ như USB, FireWire, Ethernet, ATA/IDE, SCSI và PCI.
- *Phần mềm / Giao diện lập trình*: là ngôn ngữ, mã và thông điệp mà các chương trình sử dụng để giao tiếp với nhau và với phần cứng.

- Hệ điều hành Windows, Mac và Linux
- Email SMTP
- Giao thức mạng IP
- Trình điều khiển phần mềm kích hoạt các thiết bị ngoại vi
- …

1. Giới thiệu

- *Giao diện người dùng*: bàn phím, chuột, các câu lệnh và menu được sử dụng để giao tiếp giữa người dùng và máy tính.

- Định dạng & Chức năng: định dạng (tiêu đề, nội dung, đoạn giới thiệu, v.v.), chức năng (đọc, ghi, truyền, nhận, kiểm tra lỗi, phương pháp truy cập, giao thức liên kết dữ liệu, …)

- Ngôn ngữ & Lập trình:

- Giao diện người dùng, Giao thức, API và ABI

Chapter V. Hardware Interface Programming

- 1. Giới thiệu
- 2. Các chuẩn giao tiếp và giao thức truyền thông công nghiệp
- 3. Truyền thông nối tiếp với Qt
- 4. Đa luồng trong Qt
- 5. Lập trình giao tiếp với Arduino
- 6. Lập trình giao tiếp với camera

2. Các chuẩn giao tiếp và giao thức truyền thông công nghiệp

❖ *Cổng nối tiếp*

- Các cổng hoạt động theo nguyên lý nối tiếp
- Các cổng nối tiếp thông dụng được sử dụng trong truyền thông công nghiệp như: COM, RS232/RS422/RS485, v.v.
- Giao tiếp nối tiếp chậm hơn so với giao tiếp song song, tuy nhiên được dùng phổ biến để truyền dữ liệu dài bởi chi phí thấp hơn.
- RS232 (DB9 hay COM): Tốc độ truyền của cổng RS232 được dùng phổ biến như: 9600, 14400, 28800 và 33600. Tốc độ truyền dữ liệu có thể ở mức 20 kb/s. Chiều dài cáp tối đa là 15 mét.

2. Các chuẩn giao tiếp và giao thức truyền thông công nghiệp

❖ *Cổng nối tiếp*

- RS422: Với chiều dài đường truyền là 40 feet (12m) thì tốc độ truyền tối đa là 10 Mbits/s, 400 feet (122m) là 1 Mbits/s và 4000 feet (1219m) là 100 kbits/s. Hiện nay chuẩn truyền thông công nghiệp RS422 gần như không được sử dụng.

- RS485: Có thể coi RS485 là một phiên bản nâng cấp của RS422. RS485 cho phép kết nối và truyền dữ liệu với tối đa 32 cặp thu phát trên đường truyền cùng một lúc. Với chiều dài đường truyền là 40 feet (12m) thì tốc độ truyền tối đa là 10 Mbits/s, 400 feet (122m) là 1 Mbits/s và 4000 feet (1219m) là 100 kbits/s.

2. Các chuẩn giao tiếp và giao thức truyền thông công nghiệp

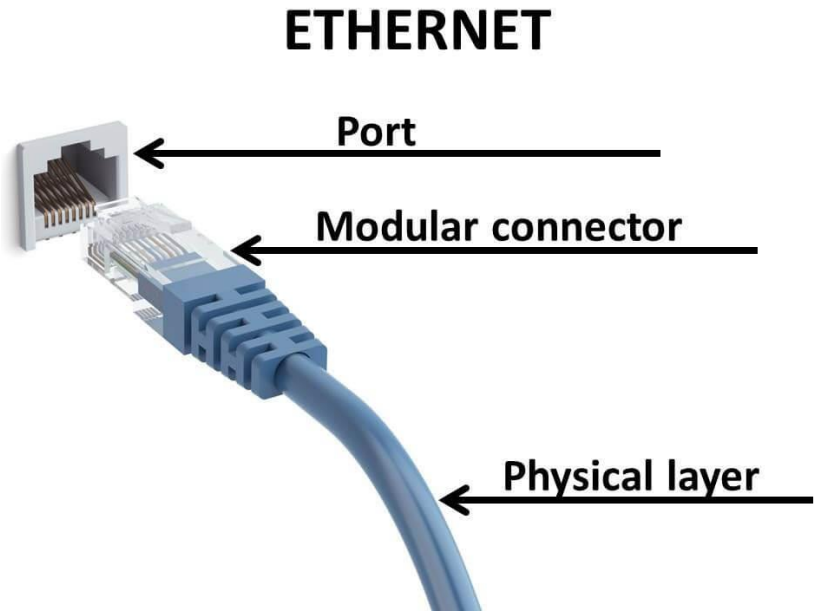
❖ *USB (Universal Serial Bus)*

- Kết nối các thiết bị (điện thoại, máy tính bảng, máy chụp ảnh, máy quay phim, máy nghe nhạc hoặc các thiết bị công nghiệp khác như bộ thu thập dữ liệu, remote I/O, v.v. với máy tính.
- USB có 2 loại chính là cổng USB 2.0 và cổng USB 3.0
- Về lí thuyết thì tốc độ ghi chép dữ liệu của USB 2.0 là 60 MB/s, còn USB 3.0 là 600 – 625 MB/s.

2. Các chuẩn giao tiếp và giao thức truyền thông công nghiệp

❖ *Ethernet (LAN, RJ45)*

- Ethernet là một dạng công nghệ truyền thông dùng để kết nối các mạng LAN cục bộ, cho phép các thiết bị có thể giao tiếp với nhau thông qua một giao thức – một bộ quy tắc hoặc ngôn ngữ mạng chung.
- So với công nghệ mạng LAN không dây, Ethernet thường ít bị gián đoạn hơn – cho dù là do nhiễu sóng vô tuyến, trở ngại vật lý hay băng thông.
- Tốc độ chuẩn cho hệ thống Ethernet hiện nay là 100-Mbps, 1000-Mbps.

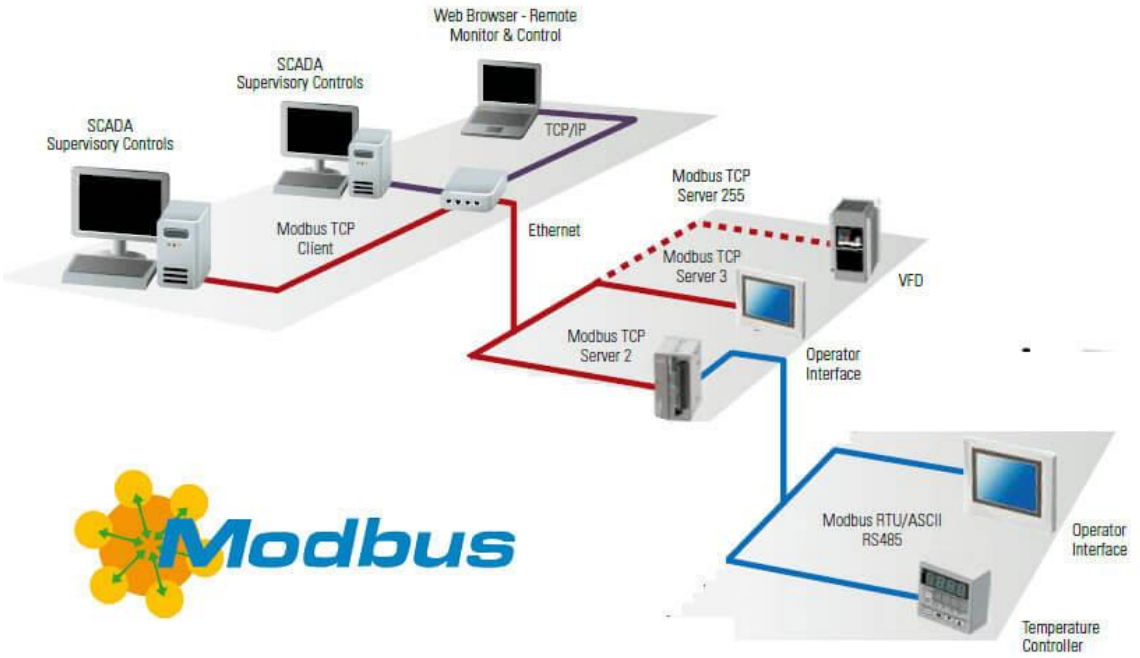


2. Các chuẩn giao tiếp và giao thức truyền thông công nghiệp

❖ *MODBUS*

➢ Modbus là một chuẩn truyền thông công nghiệp được Modicon phát triển từ năm 1979 để thay thế các chuẩn truyền thông truyền thống trước đó.

➢ Cách thức hoạt động của Modbus là dựa trên nguyên tắc Master – Slave (bên nhận – bên gửi tín hiệu), nhằm truyền dữ liệu từ các thiết bị đầu cuối về PLC hoặc SCADA.



2. Các chuẩn giao tiếp và giao thức truyền thông công nghiệp

❖ *MODBUS*

- Modbus đã trở thành một chuẩn truyền thông công nghiệp tiêu chuẩn và phổ biến nhờ sự ổn định, đơn giản, dễ sử dụng và miễn phí.
- Các thiết bị chỉ cần cùng chung một chuẩn với nhau thì có thể giao tiếp với nhau mà không cần quan tâm về loại thiết bị hay hãng sản xuất.
- Nhờ đó, các nhà sản xuất đã tích hợp chuẩn Modbus vào sản phẩm của họ để tăng tính linh hoạt mà không cần trả phí bản quyền.

2. Các chuẩn giao tiếp và giao thức truyền thông công nghiệp

❖ *UART (Universal Asynchronous Receiver – Transmitter)*

- UART là một mạch tích hợp được sử dụng trong việc truyền dẫn dữ liệu nối tiếp giữa máy tính và các thiết bị ngoại vi.
- Trong UART, giao tiếp giữa hai thiết bị có thể được thực hiện theo hai phương thức là giao tiếp dữ liệu nối tiếp và giao tiếp dữ liệu song song.
- UART thường được sử dụng trong các bộ vi điều khiển có các yêu cầu chính xác và chúng cũng có sẵn trong các thiết bị liên lạc khác nhau như giao tiếp không dây, thiết bị GPS, mô-đun Bluetooth và nhiều ứng dụng khác.

2. Các chuẩn giao tiếp và giao thức truyền thông công nghiệp

❖ *UART (Universal Asynchronous Receiver – Transmitter)*

- UART là một mạch tích hợp được sử dụng trong việc truyền dẫn dữ liệu nối tiếp giữa máy tính và các thiết bị ngoại vi.
- Trong UART, giao tiếp giữa hai thiết bị có thể được thực hiện theo hai phương thức là giao tiếp dữ liệu nối tiếp và giao tiếp dữ liệu song song.
- UART thường được sử dụng trong các bộ vi điều khiển có các yêu cầu chính xác và chúng cũng có sẵn trong các thiết bị liên lạc khác nhau như giao tiếp không dây, thiết bị GPS, mô-đun Bluetooth và nhiều ứng dụng khác.

Chapter V. Hardware Interface Programming

- 1. Giới thiệu
- 2. Các chuẩn giao tiếp và giao thức truyền thông công nghiệp
- 3. Truyền thông nối tiếp với Qt
- 4. Đa luồng trong Qt
- 5. Lập trình giao tiếp với Arduino
- 6. Lập trình giao tiếp với camera

3. Truyền thông nối tiếp với Qt

❖ Giới thiệu về QtSerialPort

- QtSerialPort cung cấp chức năng cơ bản, bao gồm cấu hình, hoạt động I/O, nhận và thiết lập tín hiệu điều khiển của sơ đồ chân RS-232.
- Mô-đun QtSerialPort là một mô-đun bổ sung cho thư viện Qt5, cung cấp một giao diện duy nhất cho cả phần cứng và cổng nối tiếp ảo.
- Giao diện nối tiếp, do tính đơn giản và độ tin cậy của chúng, vẫn còn phổ biến trong một số ngành công nghiệp như phát triển hệ thống nhúng, người máy, v.v.
- Sử dụng mô-đun QtSerialPort, các nhà phát triển có thể giảm đáng kể thời gian cần thiết để triển khai các ứng dụng Qt yêu cầu quyền truy cập vào giao diện nối tiếp.

https://wiki.qt.io/Qt_Serial_Port

3. Truyền thông nối tiếp với Qt

❖ Giới thiệu về QtSerialPort

- Các tính năng sau không được mô-đun này hỗ trợ:

➢ Các tính năng đầu cuối, chẳng hạn như tiếng vang (echo), điều khiển CR/LF, v.v.

➢ Chế độ văn bản

➢ Định cấu hình thời gian chờ và độ trễ khi đọc hoặc viết

➢ Thông báo thay đổi tín hiệu sơ đồ chân

➢ Các điều kiện thu phát đặc biệt, như lỗi Framing, lỗi chẵn lẻ và lỗi điều kiện Break.

https://wiki.qt.io/Qt_Serial_Port

3. Truyền thông nối tiếp với Qt

❖ Qt Serial Port: Chức năng

- QSerialPort: QSerialPort là lớp cơ sở của mô-đun và cung cấp một tập hợp các phương thức và thuộc tính cơ bản để truy cập tài nguyên trên các cổng nối tiếp. Hỗ trợ các hệ điều hành sau:

Hệ điều hành	Hỗ trợ	Chú ý
Windows XP/Vista/7/8/10	Có	Hỗ trợ đầy đủ
Windows CE	Không (từ Qt5.7)	Chỉ được thử nghiệm trên 5 và 6 nền tảng trong trình giả lập
Gnu/Linux	Có	Hỗ trợ đầy đủ
MacOSX	Có	Hỗ trợ đầy đủ
Others Unix	Có	Tất cả tương thích với POSIX

https://wiki.qt.io/Qt_Serial_Port

3. Truyền thông nối tiếp với Qt

❖ Qt Serial Port: Chức năng

- QSerialPortInfo: QSerialPortInfo là một lớp trợ giúp. Lớp này cung cấp thông tin về các cổng nối tiếp có sẵn trên hệ thống. Hỗ trợ các hệ điều hành sau:

Hệ điều hành	Hỗ trợ	Chú ý
Windows XP/Vista/7/8/10	Có	Hỗ trợ đầy đủ (sử dụng SetupAPI)
Windows CE	Không (từ Qt5.7)	Chỉ được thử nghiệm trên 5 và 6 nền tảng trong trình giả lập
Gnu/Linux	Có	Hỗ trợ đầy đủ (sử dụng libudev, sysfs hoặc tìm kiếm đơn giản trong /dev)
MacOSX	Có	Hỗ trợ đầy đủ
Others Unix	Có	Tất cả tương thích với POSIX (chỉ tìm kiếm đơn giản trong /dev)

https://wiki.qt.io/Qt_Serial_Port

3. Truyền thông nối tiếp với Qt

❖ Qt Serial Port: Functionality

- QSerialPort:
  - How to use?

```
#include <QSerialPort>
```

```
#include <QSerialPortInfo>
```

  - To link against the module, add this line to your qmake .pro file:

```
QT += serialport
```

https://doc.qt.io/qt-5/qtserialport-index.html

3. Truyền thông nối tiếp với Qt

Qt Serial Port: Example

```
#include <QCoreApplication>
#include <QDebug>

#include <QSerialPort>
#include <QSerialPortInfo>

QT_USE_NAMESPACE

int main(int argc, char *argv[])
{
    QCoreApplication a(argc, argv);
    // Example use QSerialPortInfo
    foreach (const QSerialPortInfo &info, QSerialPortInfo::availablePorts()) {
        qDebug() << "Name : " << info.portName();
        qDebug() << "Description : " << info.description();
        qDebug() << "Manufacturer: " << info.manufacturer();

        // Example use QSerialPort
        QSerialPort serial;
        serial.setPort(info);
        if (serial.open(QIODevice::ReadWrite))
            serial.close();
    }
    return a.exec();
}
```

https://wiki.qt.io/Qt_Serial_Port

Chapter V. Hardware Interface Programming

- 1. Giới thiệu
- 2. Các chuẩn giao tiếp và giao thức truyền thông công nghiệp
- 3. Truyền thông nối tiếp với Qt
- 4. Đa luồng trong Qt
- 5. Lập trình giao tiếp với Arduino
- 6. Lập trình giao tiếp với camera

4. Đa luồng trong Qt

- Lớp QThread được sử dụng để xử lý tất cả các loại chức năng đa luồng

- QThread: Lớp này là cơ sở của tất cả các luồng trong Qt.
- QThreadPool: Lớp này có thể được sử dụng để quản lý các luồng và giúp giảm chi phí tạo luồng bằng cách cho phép các luồng hiện có được sử dụng lại cho các mục đích mới.
- QRunnable: Lớp này cung cấp một cách khác để tạo luồng và nó là cơ sở của tất cả các đối tượng có thể chạy trong Qt.

4. Đa luồng trong Qt

- Lớp QThread được sử dụng để xử lý tất cả các loại chức năng đa luồng

- QMutex, QMutexLocker, QSemaphore, QWaitCondition, QReadLocker, QWriteLocker và QWriteLocke: Các lớp này được sử dụng để giải quyết các tác vụ đồng bộ hóa liên luồng. Tùy thuộc vào tình huống, các lớp này có thể được sử dụng để tránh các vấn đề như các luồng ghi đè tính toán của nhau, các luồng cố gắng đọc hoặc ghi vào một thiết bị chỉ có thể xử lý một luồng tại một thời điểm và nhiều vấn đề tương tự. Thông thường cần phải xử lý thủ công các vấn đề như vậy khi tạo các ứng dụng đa luồng.

4. Đa luồng trong Qt

- Lớp QThread được sử dụng để xử lý tất cả các loại chức năng đa luồng

- QtConcurrent: Không gian tên này có thể được sử dụng để tạo các ứng dụng đa luồng bằng cách sử dụng API cấp cao. Nó giúp việc viết các ứng dụng đa luồng trở nên dễ dàng hơn mà không cần phải xử lý các vấn đề về mutexes, semaphores và đồng bộ hóa liên luồng.
- QFuture, QFutureWatcher, QFututeIterator và QFutureSynchronizer: Tất cả các lớp này đều được sử dụng cùng với không gian tên QtConcurrent để xử lý kết quả hoạt động đa luồng và không đồng bộ.

4. Đa luồng trong Qt

- Có hai cách tiếp cận khác nhau đối với đa luồng trong Qt.
  - Cách tiếp cận đầu tiên, dựa trên QThread, là cách tiếp cận cấp thấp cung cấp nhiều tính linh hoạt và khả năng kiểm soát đối với các luồng nhưng đòi hỏi nhiều mã hóa và cẩn thận hơn để hoạt động hoàn hảo.
  - Cách tiếp cận thứ hai dựa trên không gian tên QtConcurrent (hoặc khung Qt Concurrency), là cách tiếp cận cấp cao để tạo và chạy nhiều tác vụ trong một ứng dụng.

4. Đa luồng trong Qt

❖ Đa luồng mức thấp sử dụng QThread

- Sử dụng lớp con Qthread: Example

➢ Tạo ứng dụng *MultithreadedCV*

➢ Thêm thư viện OpenCV

➢ Thêm hai tiện ích nhãn vào giao diện

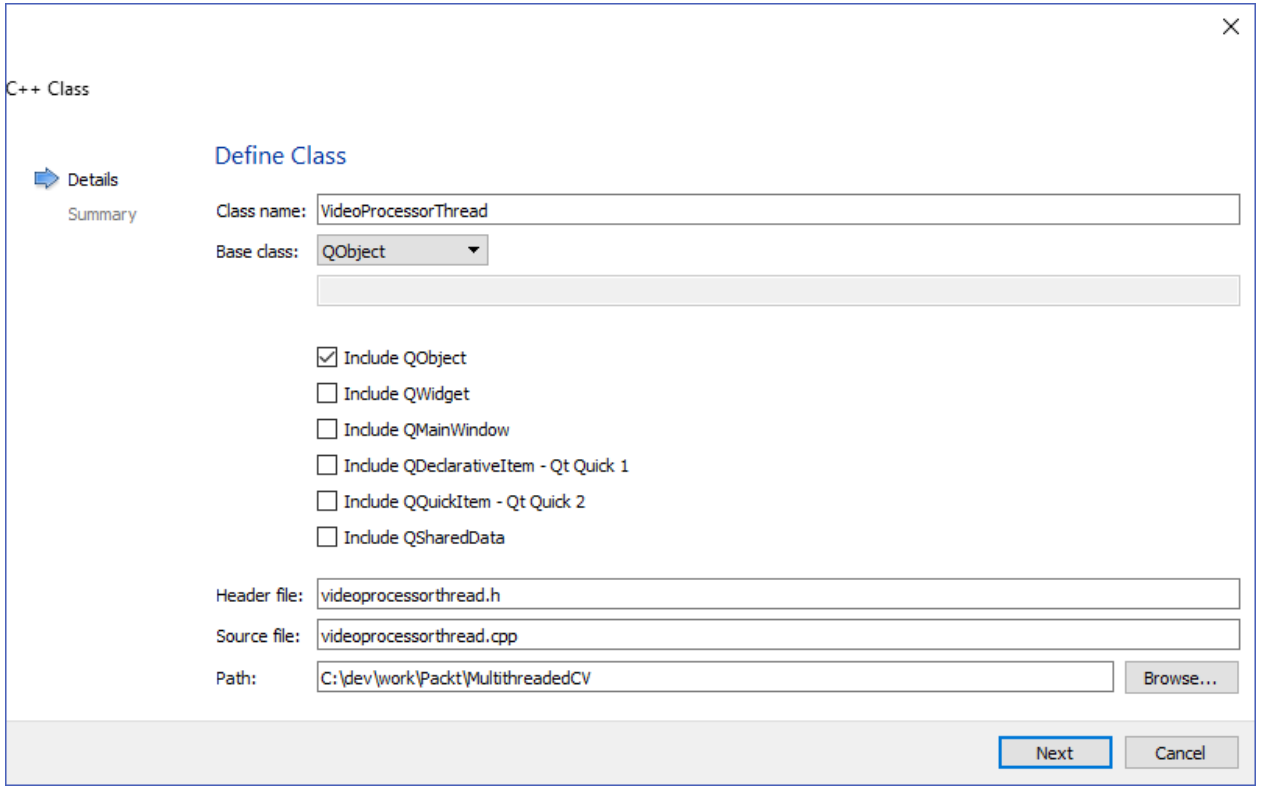
Object	Class
MainWindow	QMainWindow
centralWidget	QWidget
inVideo	QLabel
outVideo	QLabel

4. Đa luồng trong Qt

❖ Đa luồng mức thấp sử dụng QThread

- Sử dụng lớp con Qthread: Example

➢ Tạo một lớp mới có tên là *VideoProcessorThread*



4. Đa luồng trong Qt

❖ Đa luồng mức thấp sử dụng QThread

- Sử dụng lớp con Qthread: Example

➢ Tạo một lớp mới có tên là *VideoProcessorThread*

```h
#ifndef VIDEOPROCESSORTHREAD_H
#define VIDEOPROCESSORTHREAD_H

#include <QObject>
#include <QThread>
#include <QPixmap>
#include "opencv2/opencv.hpp"

class VideoProcessorThread : public QThread
{
    Q_OBJECT
public:
    explicit VideoProcessorThread(QObject *parent = nullptr);

signals:
    void inDisplay(QPixmap pixmap);
    void outDisplay(QPixmap pixmap);

public slots:

private:
    void run() override;
};

#endif // VIDEOPROCESSORTHREAD_H

```

```cpp
#include "videoprocessorthread.h"

VideoProcessorThread::VideoProcessorThread(QObject *parent) : QThread(parent)
{

}

void VideoProcessorThread::run()
{
    using namespace cv;
    VideoCapture camera(0);
    Mat inFrame, outFrame;
    while(camera.isOpened() && !isInterruptionRequested())
    {
        camera >> inFrame;
        if(inFrame.empty())
            continue;

        bitwise_not(inFrame, outFrame);

        emit inDisplay(
                    QPixmap::fromImage(
                        QImage(
                            inFrame.data,
                            inFrame.cols,
                            inFrame.rows,
                            inFrame.step,
                            QImage::Format_RGB888)
                        .rgbSwapped()));

        emit outDisplay(
                    QPixmap::fromImage(
                        QImage(
                            outFrame.data,
                            outFrame.cols,
                            outFrame.rows,
                            outFrame.step,
                            QImage::Format_RGB888)
                        .rgbSwapped()));
    }
}

```

4. Đa luồng trong Qt

❖ Đa luồng mức thấp sử dụng QThread

- Sử dụng lớp con Qthread: Example

➢ Hàm *start:* được sử dụng để bắt đầu một luồng nếu nó chưa được bắt đầu.

➢ Hàm *terminate*: buộc một luồng phải kết thúc.

➢ Hàm *wait*: sử dụng để chặn một luồng (buộc chờ đợi) cho đến khi luồng kết thúc hoặc đạt đến giá trị thời gian chờ (tính bằng mili giây).

4. Đa luồng trong Qt

❖ Đa luồng mức thấp sử dụng QThread

- Sử dụng lớp con Qthread: Example

➢ mainwindow.h

```
1  #ifndef MAINWINDOW_H
2  #define MAINWINDOW_H
3
4  #include <QMainWindow>
5
6  #include "videoprocessorthread.h"
7
8  namespace Ui {
9  class MainWindow;
10 }
11
12 class MainWindow : public QMainWindow
13 {
14     Q_OBJECT
15
16 public:
17     explicit MainWindow(QWidget *parent = 0);
18     ~MainWindow();
19
20 private:
21     Ui::MainWindow *ui;
22
23     VideoProcessorThread processor;
24 };
25
26 #endif // MAINWINDOW_H
27
```

4. Đa luồng trong Qt

❖ Đa luồng mức thấp sử dụng QThread

- Sử dụng lớp con Qthread: Example

➢ mainwindow.cpp

```
23  MainWindow::~MainWindow()
24  {
25      processor.requestInterruption();
26      processor.wait();
27
28      delete ui;
29  }
```

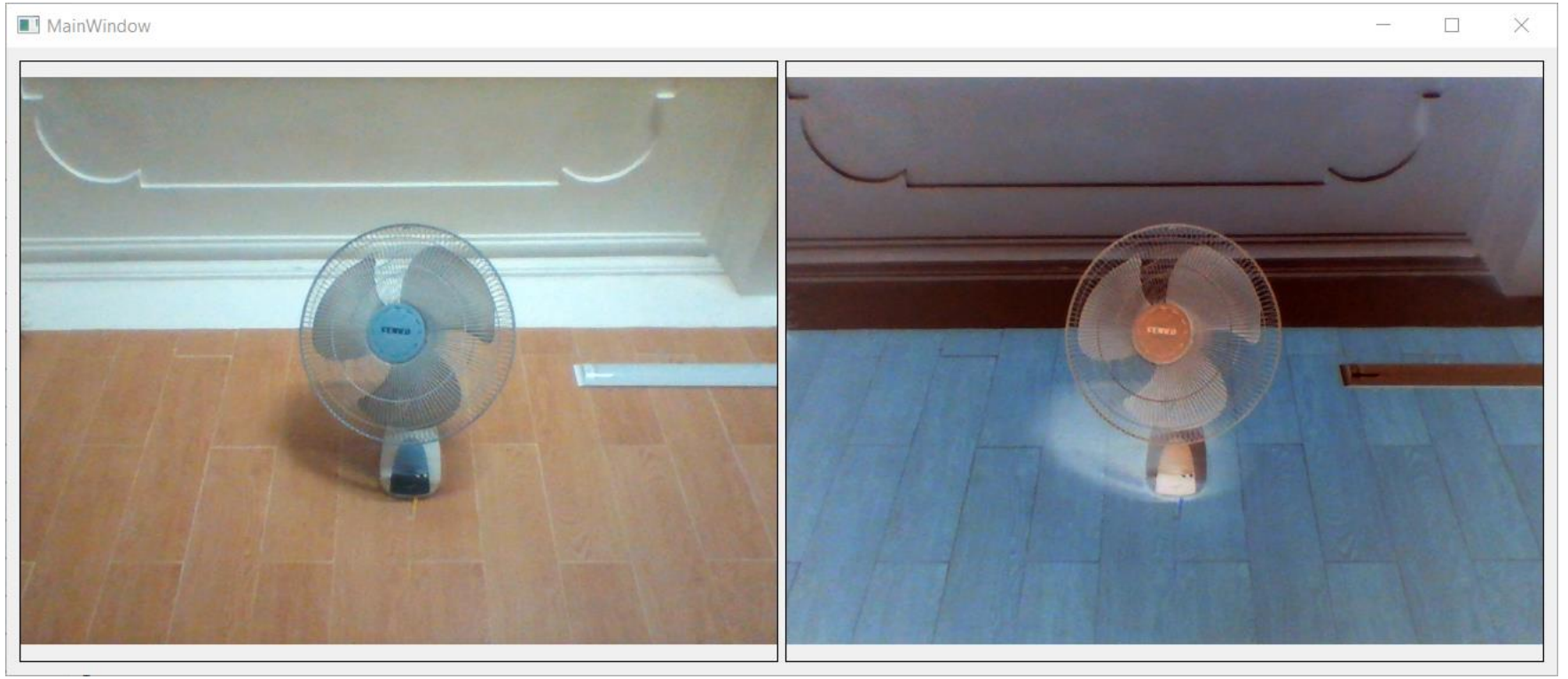
```
4   MainWindow::MainWindow(QWidget *parent) :
5       QMainWindow(parent),
6       ui(new Ui::MainWindow)
7   {
8       ui->setupUi(this);
9
10      connect(&processor,
11              SIGNAL(inDisplay(QPixmap)),
12              ui->inVideo,
13              SLOT(setPixmap(QPixmap)));
14
15      connect(&processor,
16              SIGNAL(outDisplay(QPixmap)),
17              ui->outVideo,
18              SLOT(setPixmap(QPixmap)));
19
20      processor.start();
21  }
```

4. Đa luồng trong Qt

❖ Đa luồng mức thấp sử dụng QThread

- Sử dụng lớp con Qthread: Example

➢ Build and Run program



4. Đa luồng trong Qt

❖ Đa luồng mức thấp sử dụng QThread

- Sử dụng hàm *moveToThread*: Example

➢ Tạo lớp mới *VideoProcessor*

```h
#ifndef VIDEOPROCESSOR_H
#define VIDEOPROCESSOR_H

#include <QObject>
#include <QPixmap>
#include <QDebug>
#include <QMutex>
#include <QReadWriteLock>
#include <QSemaphore>
#include <QWaitCondition>
#include "opencv2/opencv.hpp"

class VideoProcessor : public QObject
{
    Q_OBJECT
public:
    explicit VideoProcessor(QObject *parent = nullptr);

signals:
    void inDisplay(QPixmap pixmap);
    void outDisplay(QPixmap pixmap);

public slots:
    void startVideo();
    void stopVideo();

private:
    bool stopped;
};
```

```cpp
#include "videoprocessor.h"

VideoProcessor::VideoProcessor(QObject *parent) : QObject(parent)
{

}

void VideoProcessor::stopVideo()
{
    qDebug() << Q_FUNC_INFO;
    stopped = true;
}
```

4. Đa luồng trong Qt

❖ Đa luồng mức thấp sử dụng QThread

- Sử dụng hàm *moveToThread*: Example

➢ Tạo lớp mới *VideoProcessor*

```
14  void VideoProcessor::startVideo()
15  {
16      using namespace cv;
17      VideoCapture camera(0);
18      Mat inFrame, outFrame;
19      stopped = false;
20      while(camera.isOpened() && !stopped)
21      {
22          camera >> inFrame;
23          if(inFrame.empty())
24              continue;
25
26          bitwise_not(inFrame, outFrame);
27
28          emit inDisplay(
29                      QPixmap::fromImage(
30                          QImage(
31                              inFrame.data,
32                              inFrame.cols,
33                              inFrame.rows,
34                              inFrame.step,
35                              QImage::Format_RGB888)
36                          .rgbSwapped()));
37
38          emit outDisplay(
39                      QPixmap::fromImage(
40                          QImage(
41                              outFrame.data,
42                              outFrame.cols,
43                              outFrame.rows,
44                              outFrame.step,
45                              QImage::Format_RGB888)
46                          .rgbSwapped()));
47      }
48  }
49
```

4. Đa luồng trong Qt

❖ Đa luồng mức thấp sử dụng QThread

- Sử dụng hàm *moveToThread*: Example

➢ Mainwindow.h

```
#ifndef MAINWINDOW_H
#define MAINWINDOW_H

#include <QMainWindow>

#include "videoprocessor.h"
#include <QThread>

namespace Ui {
class MainWindow;
}

class MainWindow : public QMainWindow
{
    Q_OBJECT

public:
    explicit MainWindow(QWidget *parent = 0);
    ~MainWindow();

private:
    Ui::MainWindow *ui;

    VideoProcessor *processor;
};

#endif // MAINWINDOW_H
```

4. Đa luồng trong Qt

❖ Đa luồng mức thấp sử dụng QThread

- Sử dụng hàm *moveToThread*: Example

➢ Mainwindow.cpp

```
4   MainWindow::MainWindow(QWidget *parent) :
5       QMainWindow(parent),
6       ui(new Ui::MainWindow)
7   {
8       ui->setupUi(this);
9
10      processor = new VideoProcessor();
11
12      processor->moveToThread(new QThread(this));
13
14      connect(processor->thread(),
15              SIGNAL(started()),
16              processor,
17              SLOT(startVideo()));
18
19      connect(processor->thread(),
20              SIGNAL(finished()),
21              processor,
22              SLOT(deleteLater()));
23
24      connect(processor,
25              SIGNAL(inDisplay(QPixmap)),
26              ui->inVideo,
27              SLOT(setPixmap(QPixmap)));
28
29      connect(processor,
30              SIGNAL(outDisplay(QPixmap)),
31              ui->outVideo,
32              SLOT(setPixmap(QPixmap)));
33
34      processor->thread()->start();
35  }
```

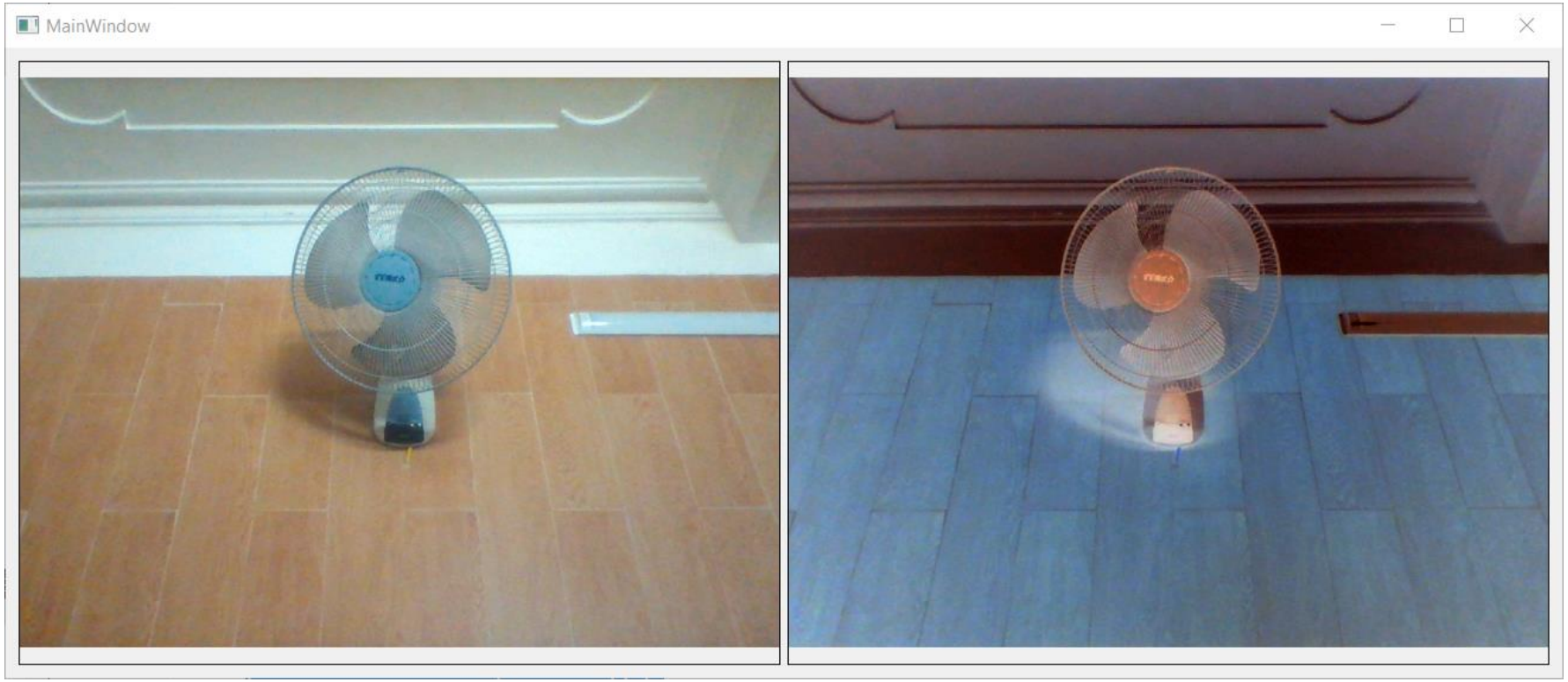
```
37  MainWindow::~MainWindow()
38  {
39      processor->stopVideo();
40      processor->thread()->quit();
41      processor->thread()->wait();
42
43      delete ui;
44  }
45
46
```

4. Đa luồng trong Qt

❖ Đa luồng mức thấp sử dụng QThread

- Sử dụng hàm *moveToThread*: Example

➢ Build and Run program



4. Đa luồng trong Qt

❖ Công cụ đồng bộ hóa luồng

- Mutex

➢ Giả sử rằng một luồng đang đọc một biến lớp *Mat* có tên là *image* mọi lúc bằng cách sử dụng các dòng mã sau:

```
forever
{
  image = imread("image.jpg");
}
```

➢ Luồng thứ hai khác đang sửa hình ảnh này mọi thời gian

```
forever
{
    cvtColor(image, image, CV_BGR2GRAY);
    resize(image, image, Size(), 0.5, 0.5);
}
```

❖ Công cụ đồng bộ hóa luồng

- Mutex

➢ Nếu hai luồng này chạy cùng một lúc, thì tại một số điểm, hàm *imread* của luồng đầu tiên có thể được gọi sau *cvtColor* và trước hàm *resize* trong luồng thứ hai.

➢ Giải pháp cho vấn đề này được gọi là **tuần tự hóa truy cập**, và trong lập trình đa luồng, nó thường được giải quyết bằng cách sử dụng các đối tượng **mutex**.

➢ Một **mutex** chỉ đơn giản là một phương tiện bảo vệ và ngăn một cá thể đối tượng bị truy cập bởi nhiều luồng tại một thời điểm.

4. Đa luồng trong Qt

Công cụ đồng bộ hóa luồng

- Mutex

- Với luồng 1

```
forever
{
    imageMutex.lock();
    image = imread("image.jpg");
    imageMutex.unlock();
}
```

- Với luồng 2

```
forever
{
    imageMutex.lock();
    cvtColor(image, image, CV_BGR2GRAY);
    resize(image, image, Size(), 0.5, 0.5);
    imageMutex.unlock();
}
```

4. Đa luồng trong Qt

❖ Công cụ đồng bộ hóa luồng

- Mutex: khi làm việc với Qt, tốt nhất là sử dụng lớp *QMutexLocker* để đảm nhận việc khóa và mở khóa mutex.

➢ Với luồng 1 và 2

```
forever
{
    QMutexLocker locker(&imageMutex);
    image = imread("image.jpg");
}

forever
{
    QMutexLocker locker(&imageMutex);
    cvtColor(image, image, CV_BGR2GRAY);
    resize(image, image, Size(), 0.5, 0.5);
}
```

❖ Công cụ đồng bộ hóa luồng

- Các khóa đọc - ghi

➢ Mutex khá hữu ích cho việc tuần tự hóa truy cập, nhưng chúng không thể được sử dụng hiệu quả cho các trường hợp như tuần tự hóa đọc-ghi

➢ Sử dụng các **Khóa Đọc-Ghi** để kiểm soát tuần tự đọc – ghi: *QReadWriteLock*

```
forever
{
    lock.lockForRead();
    read_image();
    lock.unlock();
}
```

```
forever
{
    lock.lockForWrite();
    write_image();
    lock.unlock();
}
```

Công cụ đồng bộ hóa luồng

- Các khóa đọc - ghi

➢ Để dễ dàng hơn, chúng ta có thể sử dụng các lớp *QReadLocker* và *QWriteLocker* để khóa và mở khóa *QReadWriteLock* tương ứng

```
forever
{
    QReadLocker locker(&lock);
    Read_image();
}
```

```
forever
{
    QWriteLocker locker(&lock);
    write_image();
}
```

4. Đa luồng trong Qt

❖ Công cụ đồng bộ hóa luồng

- Semaphore

➢ Trong lập trình đa luồng, chúng ta cần đảm bảo rằng nhiều luồng có thể truy cập một số tài nguyên giống nhau có giới hạn.

➢ Vấn đề này và các vấn đề tương tự trong lập trình đa luồng thường được xử lý bằng cách sử dụng các semaphore.

➢ Semaphore tương tự như mutex nâng cao, không chỉ có khả năng khóa và mở khóa mà còn theo dõi số lượng tài nguyên có sẵn.

4. Đa luồng trong Qt

❖ Công cụ đồng bộ hóa luồng

- Semaphore

➢ Giả sử chúng ta có 100 megabyte dung lượng bộ nhớ có sẵn để được sử dụng bởi tất cả các luồng của chúng ta và mỗi luồng yêu cầu X số megabyte để thực hiện nhiệm vụ của nó, tùy thuộc vào luồng.

➢ X không giống nhau trong tất cả các luồng và giả sử là nó được tính toán bằng cách sử dụng kích thước của hình ảnh sẽ được xử lý trong luồng hoặc bất kỳ phương pháp nào khác.

➢ Chúng ta có thể sử dụng lớp *QSemaphore* để đảm bảo các luồng của chúng ta chỉ truy cập vào không gian bộ nhớ có sẵn chứ không phải nhiều hơn.

❖ Công cụ đồng bộ hóa luồng

- Semaphore

➢ Tạo một semaphore

```
QSemaphore memSem(100);
```

➢ Bên trong mỗi luồng, trước và sau quá trình thâm dụng bộ nhớ, chúng ta sẽ thu nhận và giải phóng không gian bộ nhớ cần thiết

```
memSem.acquire(X);
process_image(); // memory intensive process
memSem.release(X);
```

❖ Công cụ đồng bộ hóa luồng

- Các điều kiện chờ

➢ Một vấn đề phổ biến khác trong lập trình đa luồng có thể xảy ra là một luồng nhất định phải chờ một số điều kiện khác với luồng đang được thực thi bởi hệ điều hành.

➢ Người ta sẽ mong đợi rằng luồng cần đợi một điều kiện, sẽ chuyển sang trạng thái ngủ sau khi nó giải phóng khóa mutex hoặc khóa đọc-ghi để các luồng khác tiếp tục hoạt động và khi điều kiện được đáp ứng, nó sẽ được đánh thức bởi một luồng khác.

➢ Lớp *QWaitCondition* được dành riêng để xử lý các vấn đề như chúng ta vừa đề cập.

Công cụ đồng bộ hóa luồng

- Các điều kiện chờ

- Ví dụ luồng đọc ảnh

```
forever
{
    mutex.lock();
    imageExistsCond.wait(&mutex);
    read_image();
    mutex.unlock();
}
```

- Luồng đánh thức luồng đọc ảnh

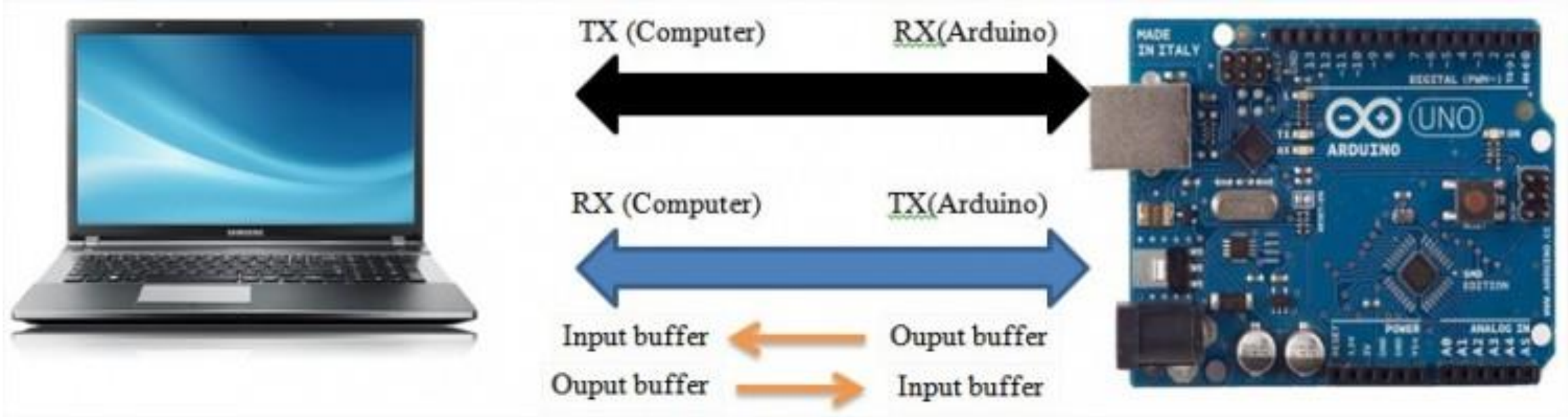
```
forever
{
    if(QFile::exists("image.jpg"))
        imageExistsCond.wakeAll();
}
```

Chapter V. Hardware Interface Programming

- 1. Giới thiệu
- 2. Các chuẩn giao tiếp và giao thức truyền thông công nghiệp
- 3. Truyền thông nối tiếp với Qt
- 4. Đa luồng trong Qt
- 5. Lập trình giao tiếp với Arduino
- 6. Lập trình giao tiếp với camera

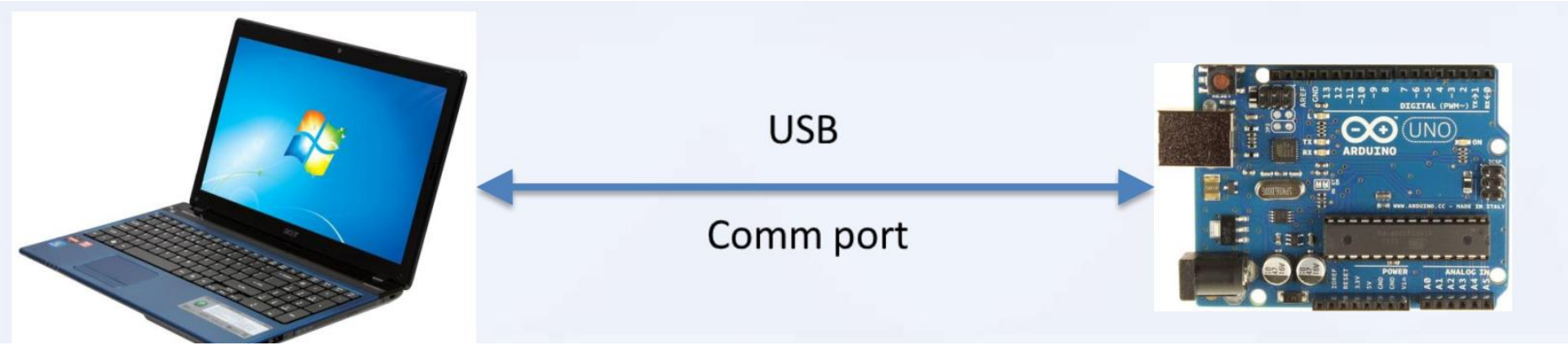
5. Lập trình giao tiếp với Arduino

- Serial Port data over USB



5. Lập trình giao tiếp với Arduino

- Serial Port data over USB



5. Lập trình giao tiếp với Arduino

❖ Qt Serial Port Example

- Create new **Qt Widgets Application**

Table1 - Components, Names, and Types Used in Qt5 Widgets

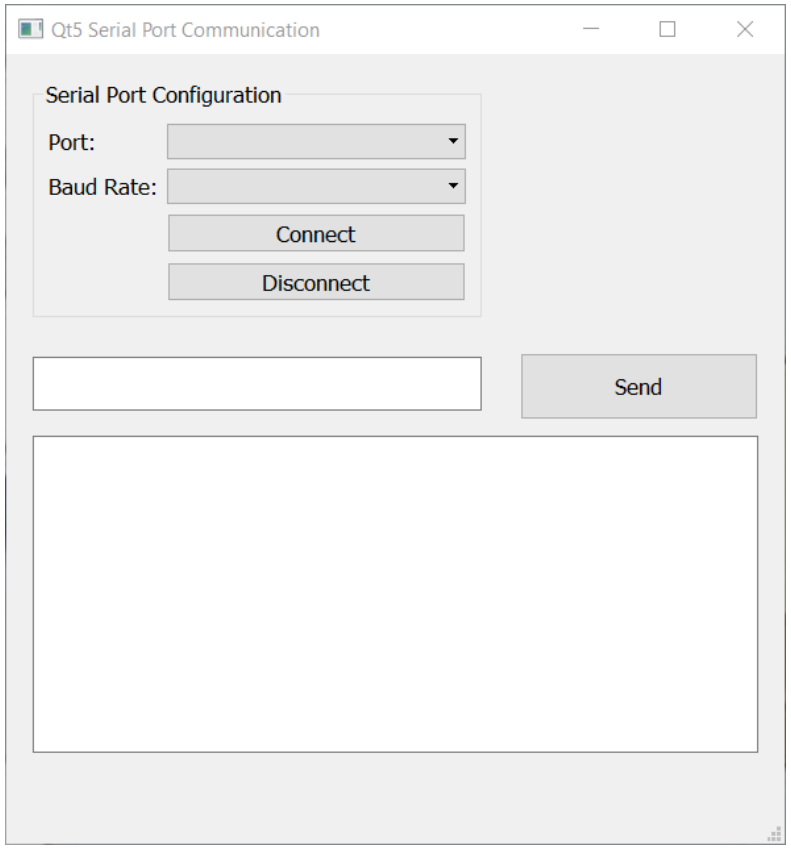
Name	Component	Description / Function
	QLabel	Application Title Label
	QFormLayout	Layout for organizing components in GroupBox
	QGroupBox	Serial Settings GroupBox
	QLabel	Door Label
comboBoxPort	QComboBox	ComboBox for Serial Ports
	QLabel	BaudRate Label
comboBoxBaudRate	QComboBox	ComboBox with BaudRate Values
pushButtonConnect	QPushButton	Connect button
pushButtonDisconnect	QPushButton	Disconnect button
lineEditSendData	QLineEdit	Command Text Input
pushButtonSendData	QPushButton	Button to send commands by serial
textEditGetData	QTextEdit	Application Log Text Box (Serial Reception)

https://www.embarcados.com.br/serie/comunicacao-serial-com-arduino-utilizando-qt5/

5. Lập trình giao tiếp với Arduino

❖ Qt Serial Port Example

- Create new **Qt Widgets Application**



https://www.embarcados.com.br/serie/comunicacao-serial-com-arduino-utilizando-qt5/

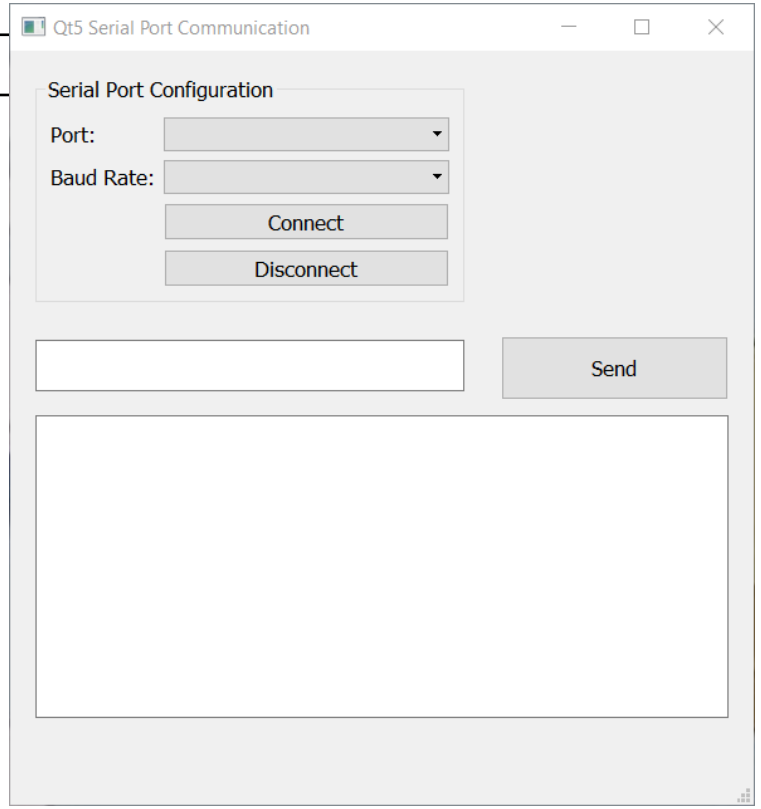
5. Lập trình giao tiếp với Arduino

❖ Qt Serial Port Example

- Create new **Qt Widgets Application**

Table1 - Components, Names, and Types Used in Qt5 Widgets

Name	Component	Description / Function
	QLabel	Application Title Label
	QFormLayout	Layout for organizing components in GroupBox
	QGroupBox	Serial Settings GroupBox
	QLabel	Door Label
comboBoxPort	QComboBox	ComboBox for Serial Ports
	QLabel	BaudRate Label
comboBoxBaudRate	QComboBox	ComboBox with BaudRate Values
pushButtonConnect	QPushButton	Connect button
pushButtonDisconnect	QPushButton	Disconnect button
lineEditSendData	QLineEdit	Command Text Input
pushButtonSendData	QPushButton	Button to send commands by serial
textEditGetData	QTextEdit	Application Log Text Box (Serial Reception)

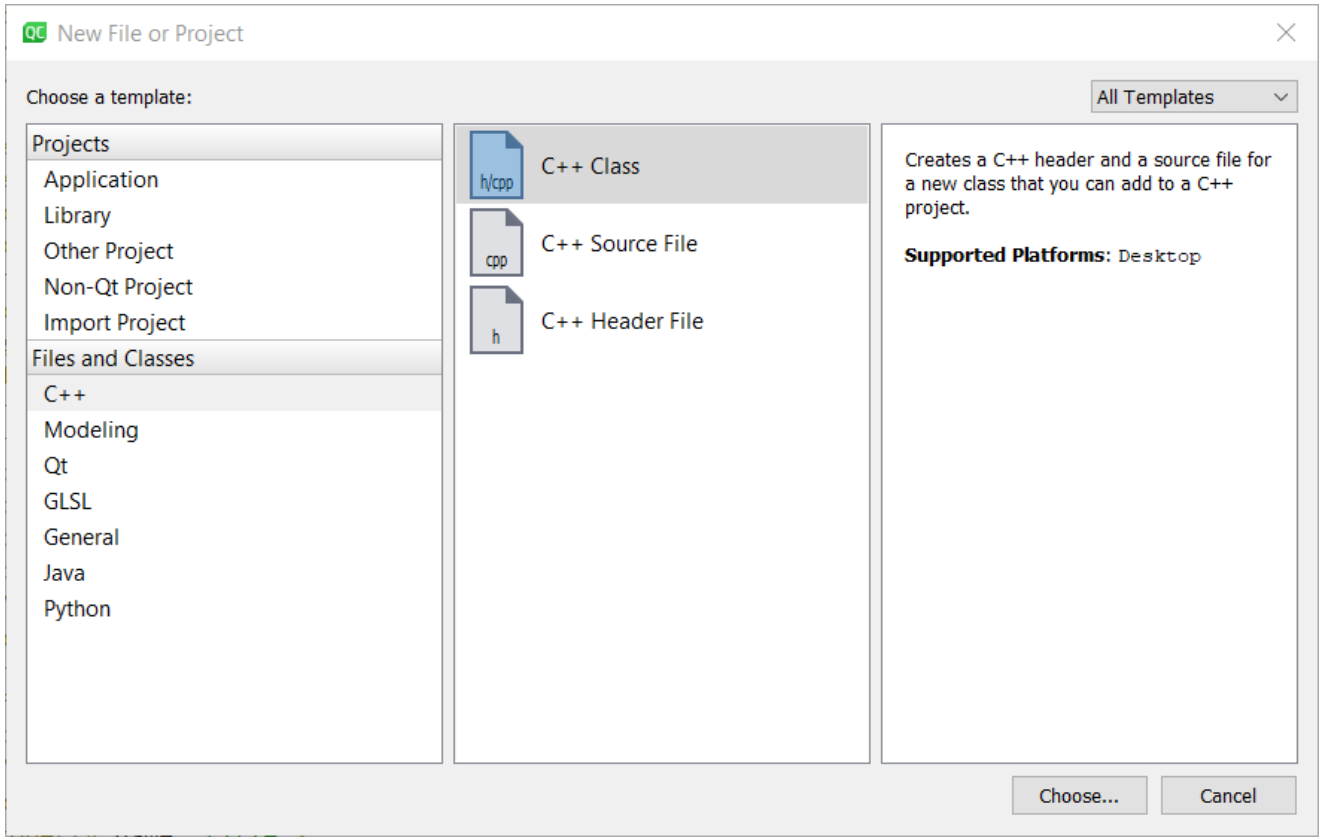


https://www.embarcados.com.br/serie/comunicacao-serial-com-arduino-utilizando-qt5/

5. Lập trình giao tiếp với Arduino

❖ Qt Serial Port Example

- Add New > C++ Class

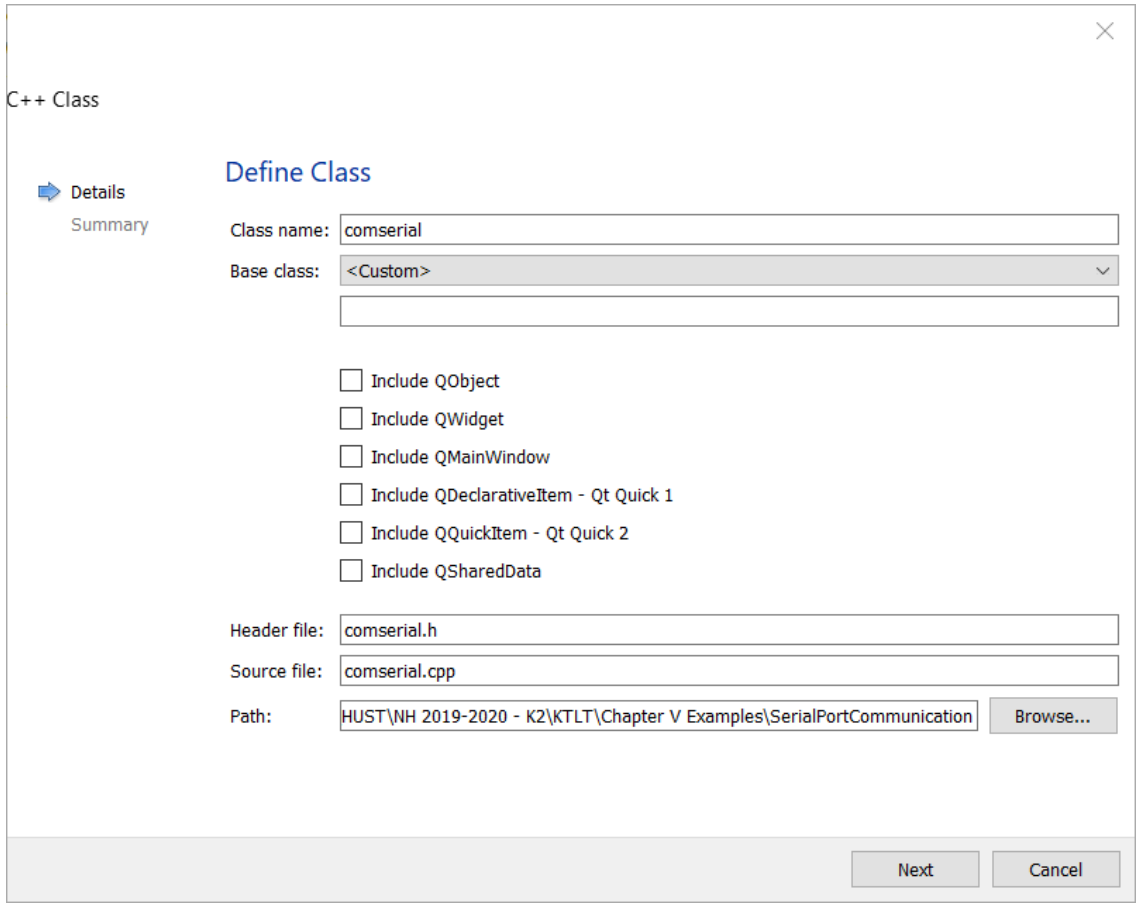


https://www.embarcados.com.br/serie/comunicacao-serial-com-arduino-utilizando-qt5/

5. Lập trình giao tiếp với Arduino

❖ Qt Serial Port Example

- Add New > C++ Class

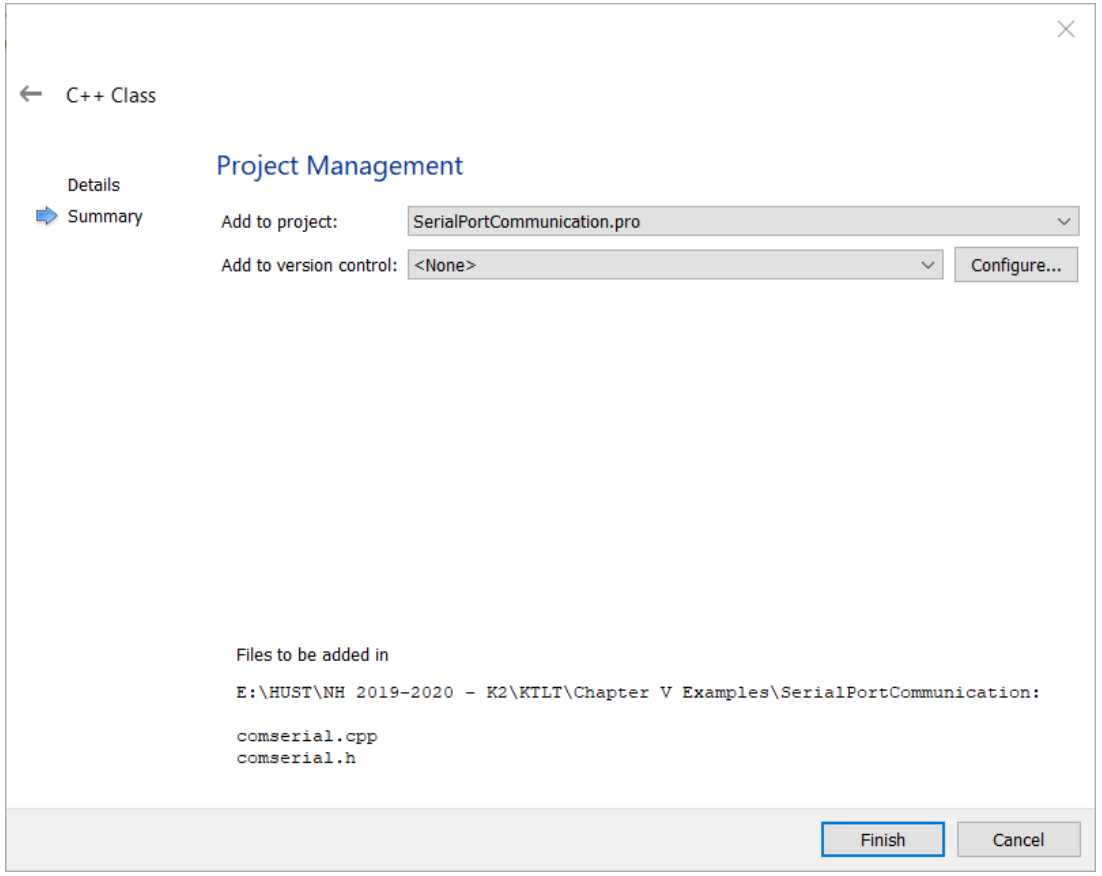


https://www.embarcados.com.br/serie/comunicacao-serial-com-arduino-utilizando-qt5/

5. Lập trình giao tiếp với Arduino

❖ Qt Serial Port Example

- Add New > C++ Class



https://www.embarcados.com.br/serie/comunicacao-serial-com-arduino-utilizando-qt5/

5. Lập trình giao tiếp với Arduino

❖ Qt Serial Port Example

- Edit the .pro file

```
QT       += core gui serialport

greaterThan(QT_MAJOR_VERSION, 4): QT += widgets

CONFIG += c++11

# The following define makes your compiler emit warnings if you use
# any Qt feature that has been marked deprecated (the exact warnings
# depend on your compiler). Please consult the documentation of the
# deprecated API in order to know how to port your code away from it.
DEFINES += QT_DEPRECATED_WARNINGS

# You can also make your code fail to compile if it uses deprecated APIs.
# In order to do so, uncomment the following line.
# You can also select to disable deprecated APIs only up to a certain version of Qt.
#DEFINES += QT_DISABLE_DEPRECATED_BEFORE=0x060000    # disables all the APIs deprecated before Qt 6.0.0

SOURCES += \
    comserial.cpp \
    main.cpp \
    mainwindow.cpp

HEADERS += \
    comserial.h \
    mainwindow.h

FORMS += \
    mainwindow.ui
```

Qt5

```
QT  +=  serialport
```

Qt4

```
CONFIG  +=  serialport
```

https://www.embarcados.com.br/serie/comunicacao-serial-com-arduino-utilizando-qt5/

5. Lập trình giao tiếp với Arduino

- Qt Serial Port Example
  - Edit the comserial.h

```
#ifndef COMSERIAL_H
#define COMSERIAL_H

/**
 * Adding QDebug to send background debug in terminal
 * QSerialPort and QSerialPortInfo to manipulate serial device
 */
#include <QDebug>
#include <QtSerialPort/QSerialPort>
#include <QtSerialPort/QSerialPortInfo>

class comserial
{
public :
    comserial(QSerialPort *myDev);
    ~ comserial();

    QStringList ConnectedDevices();

    bool   Connect(QString  Port, uint32_t bd);
    bool   Disconnect(void);

    qint64  Write(const char *cmd);
    QString  Read( );
    QString  Read(int bufferSize);

protected :
    QSerialPort  * devSerial;
};

#endif // COMSERIAL_H
```

https://www.embarcados.com.br/serie/comunicacao-serial-com-arduino-utilizando-qt5/

5. Lập trình giao tiếp với Arduino

- Qt Serial Port Example
  - Edit the comserial.cpp

```
#include "comserial.h"

/*
  * @brief CommSerial :: CommSerial
  * Constructor method function, start variables and private objects for class use
  * @param
  * @return
  */
comserial::comserial(QSerialPort *myDev)
{
    devSerial = myDev;
}

/*
  * @brief CommSerial :: ConnectedDevices
  * Function that checks all available ports / dev / tty *
  * and then arrow the port with serial.setport and test if it is a
  * serial port
  * @param void
  * @return QStringList with / dev devices which are serial ports
  */
QStringList comserial::ConnectedDevices()
{
    QStringList devs;

    foreach (const QSerialPortInfo info, QSerialPortInfo::availablePorts()) {
        devSerial->setPort(info);
        if (devSerial->open(QIODevice::ReadWrite)) {
            devSerial->close();
            devs << info.portName();
        }
    }
    return devs;
}
```

https://www.embarcados.com.br/serie/comunicacao-serial-com-arduino-utilizando-qt5/

5. Lập trình giao tiếp với Arduino

❖ Qt Serial Port Example

- Edit the comserial.cpp

```
/*
  * @brief CommSerial :: Connect
  * Function that receives the parameters Port, BaudRate and FlowControl and makes
connection
  * @param QString Port (Serial Port), uint32_t bd (BaudRate), uint8_t fc (FlowControl)
  * @return
  */
bool comserial::Connect(QString Port, uint32_t bd)
{
    /* Device Serial Port */
    devSerial->setPortName(Port);
    qDebug() << "Serial port device: " << Port;

     /* Connect SerialPort */
    /* BaudRate */
    switch (bd) {
    case 2400:
        qDebug() << "Baudrate: 2400";
        devSerial->setBaudRate(QSerialPort::Baud2400);
        break;
    case 4800:
        qDebug() << "Baudrate: 4800";
        devSerial->setBaudRate(QSerialPort::Baud4800);
        break;
    case 9600:
        qDebug() << "Baudrate: 9600";
        devSerial->setBaudRate(QSerialPort::Baud9600);
        break;
    case 19200:
        qDebug() << "Baudrate: 19200";
        devSerial->setBaudRate(QSerialPort::Baud19200);
        break;
    case 115200:
        qDebug() << "Baudrate: 115200";
        devSerial->setBaudRate(QSerialPort::Baud115200);
        break;
    }

    /* FlowControl */
    devSerial->setFlowControl(QSerialPort::NoFlowControl);

    /* Additional Settings */
    devSerial->setDataBits(QSerialPort::Data8);
    devSerial->setParity(QSerialPort::NoParity);
    devSerial->setStopBits(QSerialPort::OneStop);

    if(devSerial->open(QIODevice::ReadWrite)) {
        qDebug() << "Successfully opened serial port!";
        return true;
    }
    else {
        qDebug() << "Failed to open serial port!";
        qDebug() << "Error: " << devSerial->error();
        return false;
    }
}
```

https://www.embarcados.com.br/serie/comunicacao-serial-com-arduino-utilizando-qt5/

5. Lập trình giao tiếp với Arduino

❖ Qt Serial Port Example

- Edit the comserial.cpp

```
/*
  * @brief CommSerial :: Disconnect
  * Disconnect function, performs component cleaning and closes
  * @param
  * @return
  */
bool comserial::Disconnect()
{
    devSerial->clear();
    devSerial->close();

    if(devSerial->error() == 0 || !devSerial->isOpen()) {
        qDebug() << "Successfully closed serial port!";
        return true;
    }
    else {
        qDebug() << "Failed to close serial port! ERROR: " << devSerial->error();
        return false;
    }
}
```

https://www.embarcados.com.br/serie/comunicacao-serial-com-arduino-utilizando-qt5/

5. Lập trình giao tiếp với Arduino

- Qt Serial Port Example
  - Edit the comserial.cpp

```
/*
  * @brief ComSerial :: Write
  * Serial write function, gets a character pointer already in the write const char *
format
  * @param const char * cmd
  * @return void
  */
qint64 comserial::Write(const char *cmd)
{
    qint64 sizeWritten;
    sizeWritten = devSerial->write(cmd,qstrlen(cmd));

    return sizeWritten;
}

/*
  * @brief CommSerial :: Read
  * Function Reads what comes from the serial after writing it, and returns a QString
  * @param
  * @return QString
  */
QString comserial::Read()
{
     QString bufRxSerial;
     /* Awaits read all the data before continuing */
     while (devSerial->waitForReadyRead(20)) {
         bufRxSerial += devSerial->readAll();
     }
     return bufRxSerial;
}
```

https://www.embarcados.com.br/serie/comunicacao-serial-com-arduino-utilizando-qt5/

5. Lập trình giao tiếp với Arduino

❖ Qt Serial Port Example

- Edit the comserial.cpp

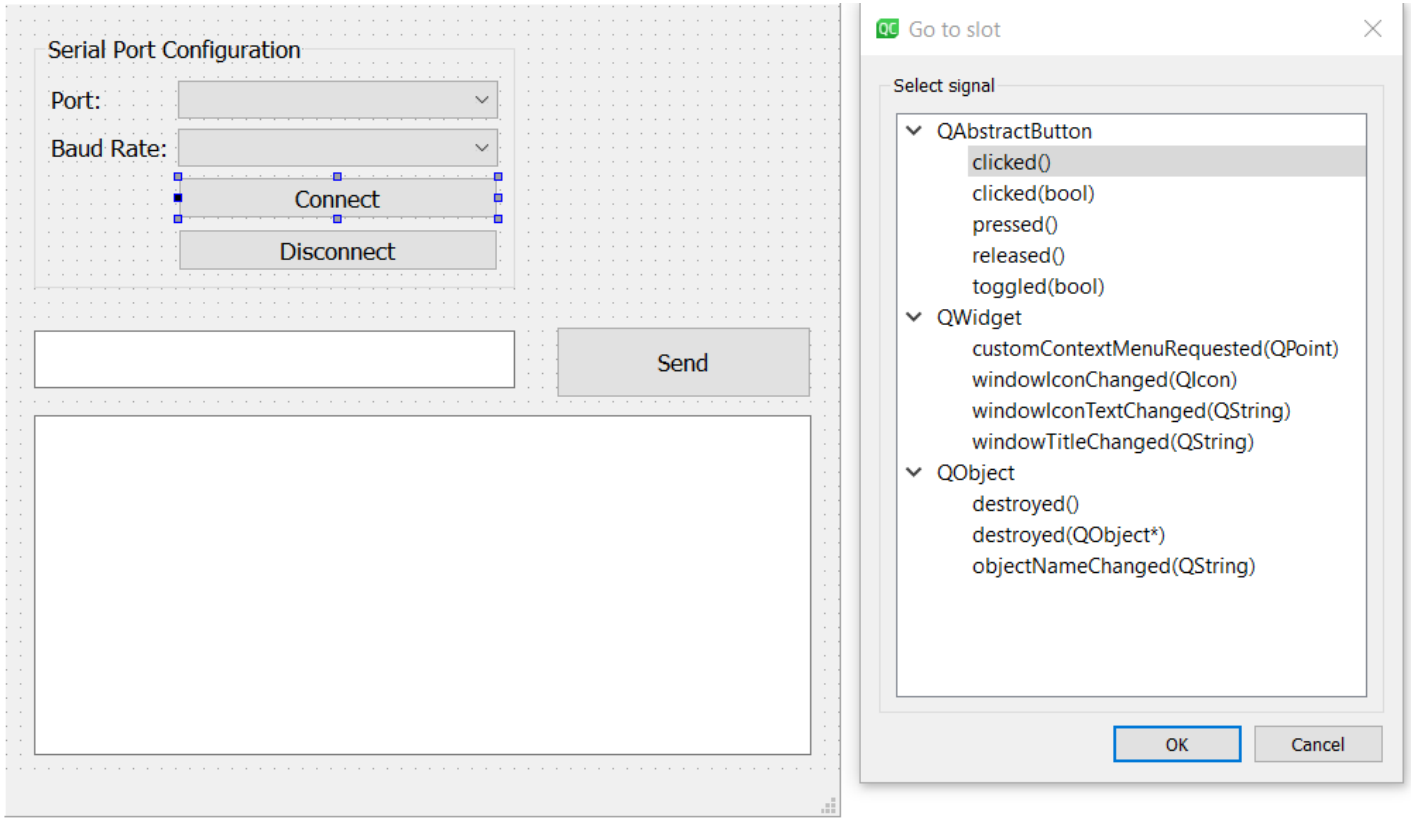
```
/*
  * @brief CommSerial :: Read
  * Function Reads what comes from the serial after writing it, and returns a QString
  * In this case an integer with the receive buffer number is sent
  * @param int
  * @return QString
  */
QString comserial::Read(int bufferSize)
{
    char *buf = new char[bufferSize];

    if (devSerial->canReadLine()) {
        devSerial->read(buf, sizeof(buf));
    }

    QString str = QString::fromUtf8(buf, bufferSize);
    delete [] buf;

    return str;
}
```

https://www.embarcados.com.br/serie/comunicacao-serial-com-arduino-utilizando-qt5/

5. Lập trình giao tiếp với Arduino

❖ Qt Serial Port Example

- Generate the Slot functions



https://www.embarcados.com.br/serie/comunicacao-serial-com-arduino-utilizando-qt5/

5. Lập trình giao tiếp với Arduino

- Qt Serial Port Example
  - Mainwindow.h

```
#ifndef MAINWINDOW_H
#define MAINWINDOW_H

#include <QMainWindow>
#include "comserial.h"

QT_BEGIN_NAMESPACE
namespace Ui { class MainWindow; }
QT_END_NAMESPACE

class MainWindow : public QMainWindow
{
    Q_OBJECT

public:
    MainWindow(QWidget *parent = nullptr);
    ~MainWindow();

private slots:
    void on_pushButtonConnect_clicked();
    void on_pushButtonDisconnect_clicked();
    void on_pushButtonSendData_clicked();

    void WriteData(const QByteArray  data);
    void ReadData();

private:
    Ui::MainWindow *ui;
    QSerialPort *devserial;
    comserial *procSerial;
};
#endif // MAINWINDOW_H
```

https://www.embarcados.com.br/serie/comunicacao-serial-com-arduino-utilizando-qt5/

5. Lập trình giao tiếp với Arduino

- Qt Serial Port Example
  - Mainwindow.cpp

```
MainWindow::MainWindow(QWidget *parent)
    : QMainWindow(parent)
    , ui(new Ui::MainWindow)
{
    ui->setupUi(this);

    /* Create Object the Class QSerialPort*/
    devserial = new QSerialPort(this);

    /* Create Object the Class comserial to manipulate read/write of the my way */
    procSerial = new comserial(devserial);

    /* Load Device PortSerial available */
    QStringList DispSerial = procSerial->ConnectedDevices();

    /* Inserting in ComboxBox the Devices */
    ui->comboBoxPort->addItems(DispSerial);

    /* Enable PushButton "Conector" if any port is found.
     * If an error occurs, it is reported in the Log
     */
    if(DispSerial.length() > 0) {
        ui->pushButtonConnect->setEnabled(true);
        ui->textEditGetData->append("### Serial port ready for use.");
    }
    else { ui->textEditGetData->append("### No serial ports were detected!"); }


    /* Connect Objects -> Signal and Slots
     * DevSerial with Read Data Serial
     * TextEdit "textEditGetData" with getData() after send data WriteData() [Not apply here in
version 5.X]
     */
    connect(devserial, SIGNAL(readyRead()), this, SLOT(ReadData()));
}
```

https://www.embarcados.com.br/serie/comunicacao-serial-com-arduino-utilizando-qt5/

5. Lập trình giao tiếp với Arduino

❖ Qt Serial Port Example

- Mainwindow.cpp

```
void MainWindow::on_pushButtonConnect_clicked()
{
    bool statusOpenSerial;

    statusOpenSerial = procSerial->Connect(ui->comboBoxPort->currentText(), ui-
>comboBoxBaudRate->currentText().toInt());
    if (statusOpenSerial) {
        ui->pushButtonDisconnect->setEnabled(true);
        ui->pushButtonConnect->setEnabled(false);
        ui->textEditGetData->append("### Successfully opened serial port!");
    }
    else {
        ui->textEditGetData->append("Failed to open serial connection.");
    }
}
```

https://www.embarcados.com.br/serie/comunicacao-serial-com-arduino-utilizando-qt5/

5. Lập trình giao tiếp với Arduino

❖ Qt Serial Port Example

- Mainwindow.cpp

```
void MainWindow::on_pushButtonDisconnect_clicked()
{
    bool statusCloseSerial;

    statusCloseSerial = procSerial->Disconnect();

    /* BUG: There is a bug in the close of QtSerialPort, already known to the community where
      * when used with waitForReadyRead, gives a SerialPort 12 Timeout error and does not
terminate the connection
      * however the error is reported but the device is terminated.
      * To get around the disconnect issue I checked with isOpen right after closing the
connection
      * serial.
      */

    if (statusCloseSerial) {
        ui->pushButtonDisconnect->setEnabled(false);
        ui->pushButtonConnect->setEnabled(true);
        ui->textEditGetData->append("### Successfully closed serial port!");
    }
    else {
        ui->textEditGetData->append("### Failed to close serial connection.");
    }
}
```

https://www.embarcados.com.br/serie/comunicacao-serial-com-arduino-utilizando-qt5/

5. Lập trình giao tiếp với Arduino

❖ Qt Serial Port Example

- Mainwindow.cpp

```
void MainWindow::ReadData()
{
    QString data = procSerial->Read();
    qDebug() << "Input: " << data << endl;
    ui->textEditGetData->append(data);

}
```

https://www.embarcados.com.br/serie/comunicacao-serial-com-arduino-utilizando-qt5/

5. Lập trình giao tiếp với Arduino

❖ Example: Read data from Arduino (1)

- Arduino code

```
void setup() {
  // put your setup code here, to run once:
  Serial.begin(9600);
}

void loop() {
  // put your main code here, to run repeatedly:
  Serial.println("Giao tiep Arduino and PC");
  delay(200);
}
```

5. Lập trình giao tiếp với Arduino

❖ Example: Read data from Arduino (2)

- Arduino code

```
#define HW_MODELO "ARDUINO UNO"

String serialCmd = "";
bool flagControlRxSerial = false;


void setup() {
    Serial.begin(9600);
    /* Limiting to 10 bytes for our commands. */
    serialCmd.reserve(10);
}
```

```
void loop() {

  /* It was always checking if the reception of the serial was over,
      This flagControlRxSerial variable will become true for this.
  */
    if (flagControlRxSerial) {
        if( serialCmd == "version\n" ) {
            Serial.print("\tFirmware: ");
            Serial.print(FW_VERSAO);
        }
        else if ( serialCmd == "hardware\n") {
            Serial.print("\tBoard : ");
            Serial.print(HW_MODELO);
        }
        else if ( serialCmd == "help\n" || serialCmd == "?\n") {
            Serial.print("\tCommands : \n\thardware - Current board model\n\tversion - Current Firmware Version");
        }
        else if (serialCmd == "\n") {
            Serial.print("\n");
        }
        else {
            Serial.print("\t* Invalid command *");
        }
        /* Resetting values for new reception */
        serialCmd = "";
        flagControlRxSerial = false;
    }
}
```

5. Lập trình giao tiếp với Arduino

❖ Example: Read data from Arduino (2)

- Arduino code

```
void serialEvent() {
    /* Loop checking if any bytes are available in serial */
    while (Serial.available()) {
      /* What comes by serial is a char typecast for the variable character */
        char caractere = (char)Serial.read();
        /* Our variable is concatenated with our serialCmd variable which is a String */
        serialCmd += caractere;
        /* If a new line arrives, our control flag (flagsControlRxSerial)
         *  becomes true and in the main loop it can already be used.
        */
        if (caractere == '\n') {
            flagControlRxSerial = true;
        }
    }
}
```

5. Lập trình giao tiếp với Arduino

❖ Qt Serial Port Example

- Mainwindow.cpp

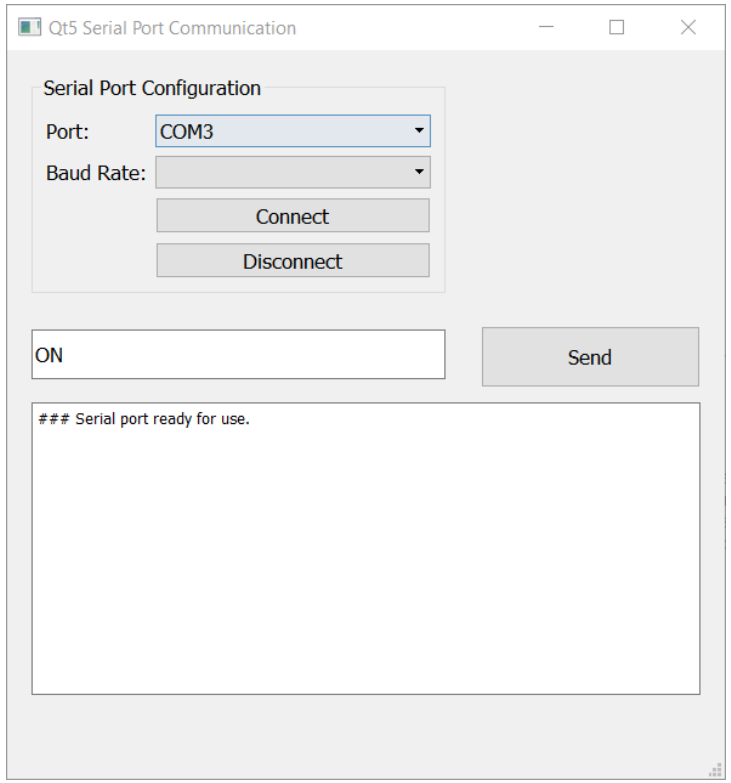
```
void MainWindow::WriteData(const QByteArray data)
{
    procSerial->Write(data);
}

void MainWindow::on_pushButtonSendData_clicked()
{
    QString Cmd = ui->lineEditSendData->text()+"\n";
    qDebug() << "Output: " << Cmd << endl;
    WriteData(Cmd.toUtf8());
}
```

https://www.embarcados.com.br/serie/comunicacao-serial-com-arduino-utilizando-qt5/

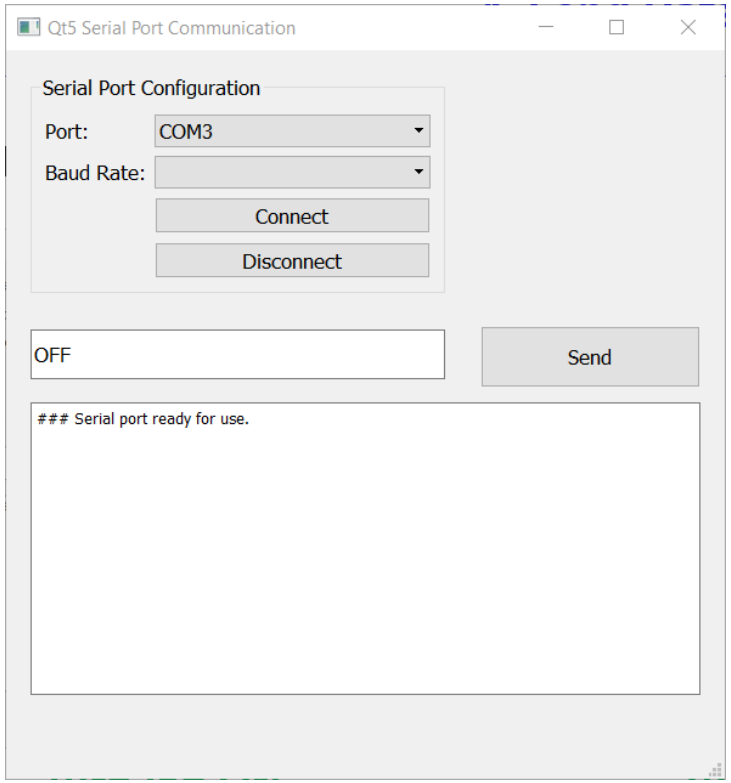
5. Lập trình giao tiếp với Arduino

❖ Example: Turn led ON/OFF

- Qt code



Turn led ON



Turn led OFF

5. Lập trình giao tiếp với Arduino

❖ Example: Turn led ON/OFF

- Arduino code

```
String x;//khai bao bien dang chuoi
// do ki tu nhap tu ban phim mang gia tri theo bang ma ASCII
int led = 11;

void setup() {
  // put your setup code here, to run once:
  Serial.begin(9600);
  // khai bao ngo ra chan 11
  pinMode(led, OUTPUT);
  // xuat ra monitor cau lenh
  //Serial.println("Nhap trang thai led ( on or off ) :");
}
```

5. Lập trình giao tiếp với Arduino

❖ Example: Turn led ON/OFF

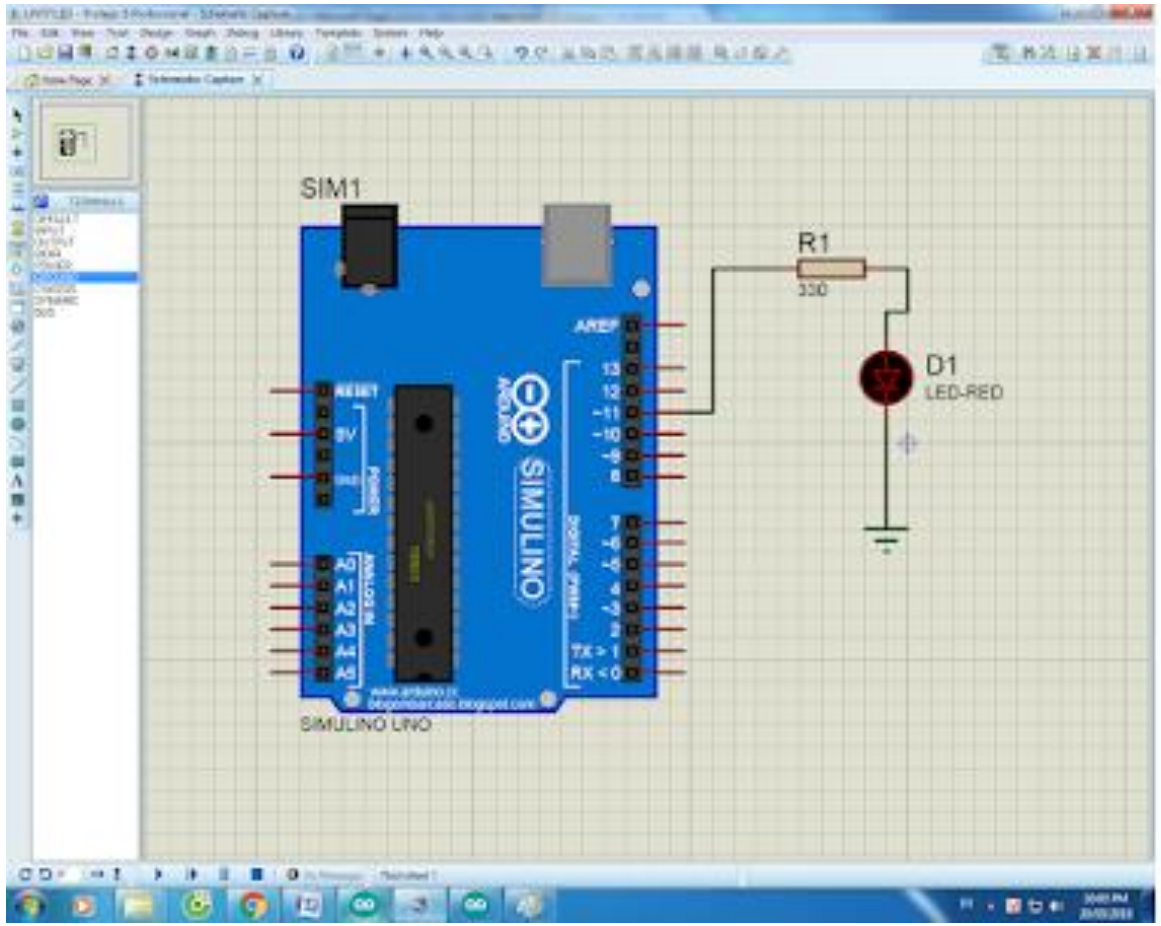
- Arduino code

```
void loop() {
  // put your main code here, to run repeatedly:
  if(Serial.available())// neu co du lieu truyen toi thi thuc hien cac lenh sau
  {
    x = Serial.readString();// gan du lieu truyen toi vao chuoi x
    //Serial.println(x);
    if(x == "ON" ){ //neu x = ON thi chan 11 xuat muc cao => led sang
      digitalWrite(11, HIGH);
    }
    if(x == "OFF" ){//neu x = OFF thi chan 11 xuat muc thap => led tat
      digitalWrite(11, LOW);
    }
  }
}
```

5. Lập trình giao tiếp với Arduino

❖ Example: Turn led ON/OFF

- Circuit Diagram

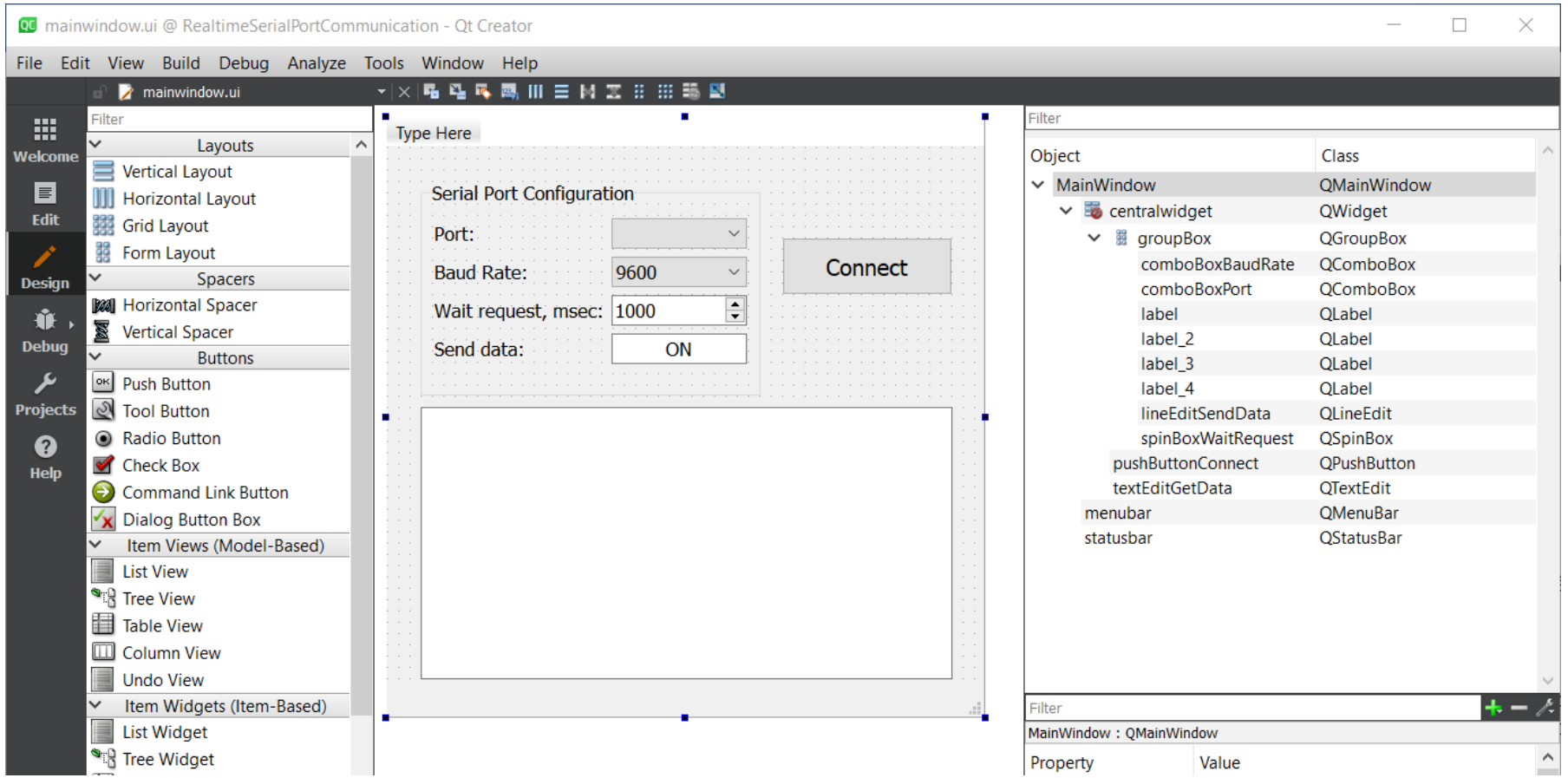


5. Lập trình giao tiếp với Arduino

❖ Truyền thông nối tiếp thời gian thực sử dụng Qthread

➢ Example: Read and Display Temperature

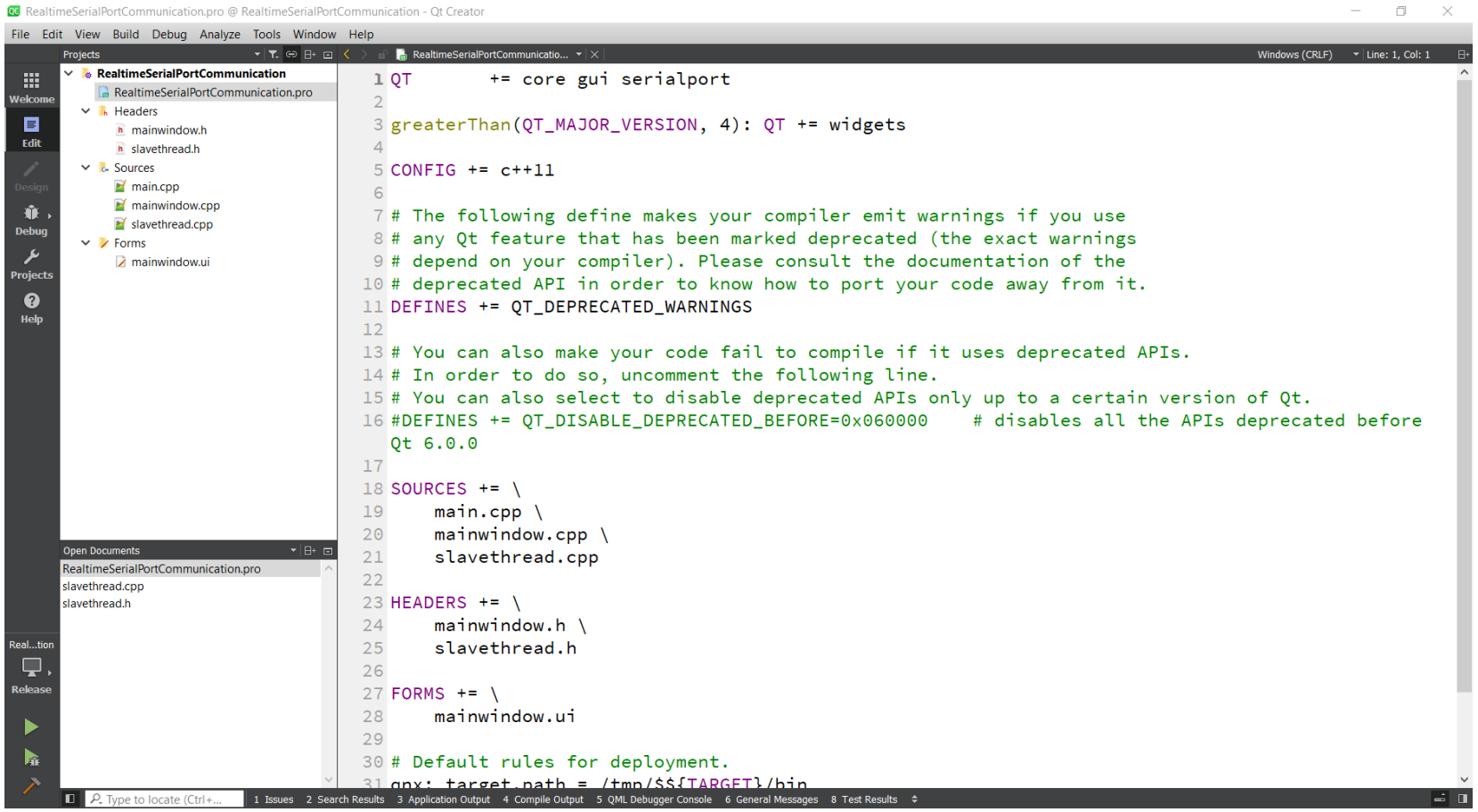
- Giao diện Qt



5. Lập trình giao tiếp với Arduino

❖ Example: Read and Display Temperature

- Giao diện Qt



5. Lập trình giao tiếp với Arduino

❖ Example: Read and Display Temperature

- Mainwindow.h

```
#ifndef MAINWINDOW_H
#define MAINWINDOW_H

#include <QMainWindow>
#include "slavethread.h"

QT_BEGIN_NAMESPACE
namespace Ui { class MainWindow; }
QT_END_NAMESPACE

class MainWindow : public QMainWindow
{
    Q_OBJECT

public:
    MainWindow(QWidget *parent = nullptr);
    ~MainWindow();

private slots:
    void on_pushButtonConnect_clicked();
    void ReadData(const QString &s);
    void processError(const QString &s);
    void processTimeout(const QString &s);
    void activateConnectButton();

private:
    int m_transactionCount = 0;

private:
    Ui::MainWindow *ui;
    SlaveThread m_thread;
};
#endif // MAINWINDOW_H
```

5. Lập trình giao tiếp với Arduino

❖ Example: Read and Display Temperature

- Mainwindow.cpp

```
#include "mainwindow.h"
#include "ui_mainwindow.h"
#include <QSerialPortInfo>

MainWindow::MainWindow(QWidget *parent)
    : QMainWindow(parent)
    , ui(new Ui::MainWindow)
{
    ui->setupUi(this);

    const auto infos = QSerialPortInfo::availablePorts();
    for (const QSerialPortInfo &info : infos)
        ui->comboBoxPort->addItem(info.portName());

    ui->comboBoxPort->setFocus();

    connect(&m_thread, &SlaveThread::request,
this,&MainWindow::ReadData);
    connect(&m_thread, &SlaveThread::error, this,
&MainWindow::processError);
    connect(&m_thread, &SlaveThread::timeout, this,
&MainWindow::processTimeout);

    connect(ui->comboBoxPort,
            QOverload<const
QString&>::of(&QComboBox::currentIndexChanged),
            this, &MainWindow::activateConnectButton);
    connect(ui->spinBoxWaitRequest, &QSpinBox::textChanged,
            this, &MainWindow::activateConnectButton);
    connect(ui->lineEditSendData, &QLineEdit::textChanged,
            this, &MainWindow::activateConnectButton);
}
```

```
MainWindow::~MainWindow()
{
    delete ui;
}

void MainWindow::on_pushButtonConnect_clicked()
{
    ui->pushButtonConnect->setEnabled(false);
    ui->textEditGetData->append(tr("Status: Running, connected to
port %1.")
                                .arg(ui->comboBoxPort->currentText()));
    m_thread.startSlave(ui->comboBoxPort->currentText(),
                        ui->comboBoxBaudRate-
>currentText().toUInt(),
                        ui->spinBoxWaitRequest->value(),
                        ui->lineEditSendData->text());
}

void MainWindow::activateConnectButton(){
    ui->pushButtonConnect->setEnabled(true);
}
```

5. Lập trình giao tiếp với Arduino

Example: Read and Display Temperature

- Mainwindow.cpp

```
void MainWindow::processError(const QString &s)
{
    activateConnectButton();
    ui->textEditGetData->append(tr("Status: Not running, %1.").arg(s));
    ui->textEditGetData->append(tr("No traffic."));
}

void MainWindow::processTimeout(const QString &s)
{
    ui->textEditGetData->append(tr("Status: Running, %1.").arg(s));
    ui->textEditGetData->append(tr("No traffic."));
}

void MainWindow::ReadData(const QString &s)
{
    ui->textEditGetData->append(tr("Traffic, transaction #%1:"
                                   "\n-request: %2"
                                   "-response: %3")
                                .arg(++m_transactionCount)
                                .arg(s)
                                .arg(ui->lineEditSendData->text()));
}
```

5. Lập trình giao tiếp với Arduino

❖ Example: Read and Display Temperature

- Add slavethread class into project

```
#ifndef SLAVETHREAD_H
#define SLAVETHREAD_H

#include <QMutex>
#include <QThread>
#include <QWaitCondition>

//! [0]
class SlaveThread : public QThread
{
    Q_OBJECT

public:
    explicit SlaveThread(QObject *parent = nullptr);
    ~SlaveThread();

    void startSlave(const QString &portName, uint32_t bd, int waitTimeout,
const QString &response);

signals:
    void request(const QString &s);
    void error(const QString &s);
    void timeout(const QString &s);

private:
    void run() override;

    QString m_portName;
    qint32 m_baudRate;
    QString m_response;
    int m_waitTimeout = 0;
    QMutex m_mutex;
    bool m_quit = false;
};
//! [0]

#endif // SLAVETHREAD_H
```

5. Lập trình giao tiếp với Arduino

Example: Read and Display Temperature

- Add slavethread class into project

```
#include "slavethread.h"

#include <QSerialPort>
#include <QTime>

SlaveThread::SlaveThread(QObject *parent) :
    QThread(parent)
{
}

//! [0]
SlaveThread::~SlaveThread()
{
    m_mutex.lock();
    m_quit = true;
    m_mutex.unlock();
    wait();
}
//! [0]
```

```
//! [1] //! [2]
void SlaveThread::startSlave(const QString &portName, uint32_t bd, int
waitTimeout, const QString &response)
{
    //! [1]
    const QMutexLocker locker(&m_mutex);
    m_portName = portName;
    /* BaudRate */
    switch (bd) {
    case 2400:
        m_baudRate = QSerialPort::Baud2400;
        break;
    case 4800:
        m_baudRate = QSerialPort::Baud4800;
        break;
    case 9600:
        m_baudRate = QSerialPort::Baud9600;
        break;
    case 19200:
        m_baudRate = QSerialPort::Baud19200;
        break;
    case 115200:
        m_baudRate = QSerialPort::Baud115200;
        break;
    }
    m_waitTimeout = waitTimeout;
    m_response = response;
    //! [3]
    if (!isRunning())
        start();
}
//! [2] //! [3]
```

5. Lập trình giao tiếp với Arduino

❖ Example: Read and Display Temperature

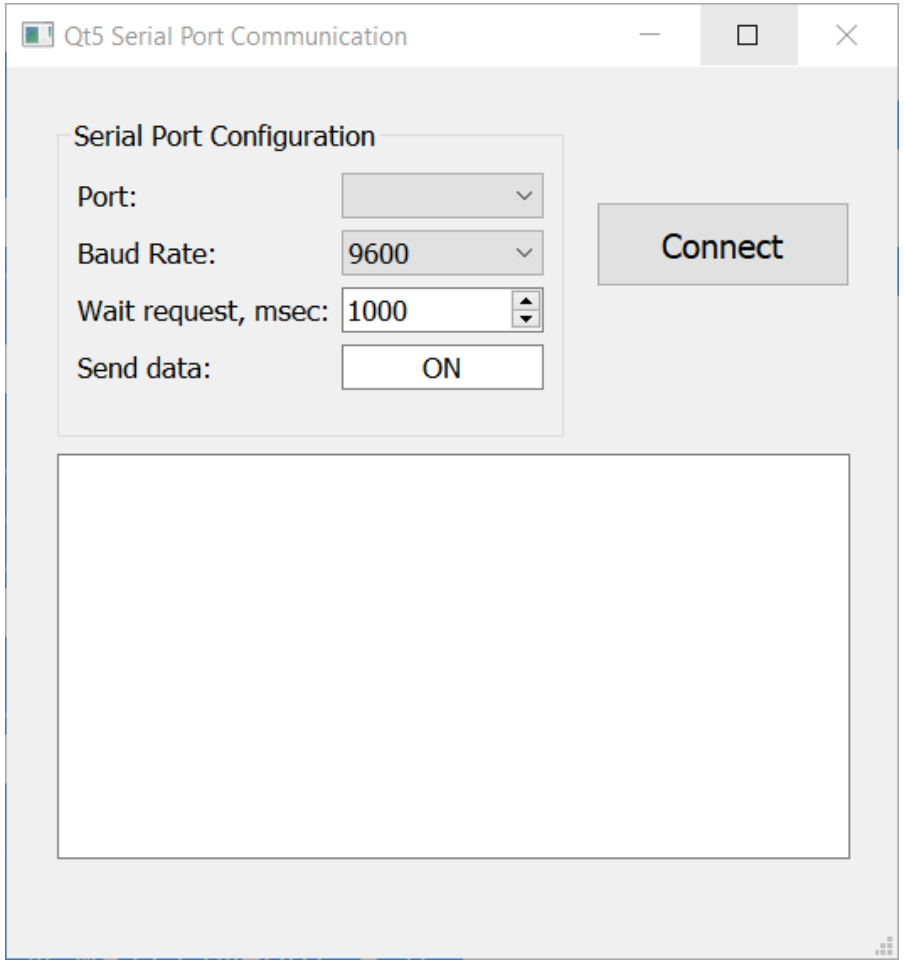
- Add slavethread class into project

```
//! [4]
void SlaveThread::run()
{
    bool currentPortNameChanged = false;

    m_mutex.lock();
//! [4] //! [5]
    QString currentPortName;
    if (currentPortName != m_portName) {
        currentPortName = m_portName;
        currentPortNameChanged = true;
    }

    int currentWaitTimeout = m_waitTimeout;
    QString currentRespone = m_response;
    qint32 currentBaudRate = m_baudRate;
    m_mutex.unlock();
//! [5] //! [6]
    QSerialPort serial;

    while (!m_quit) {
//![6] //! [7]
        if (currentPortNameChanged) {
            serial.close();
            serial.setPortName(currentPortName);
            serial.setBaudRate(currentBaudRate);

            if (!serial.open(QIODevice::ReadWrite)) {
                emit error(tr("Can't open %1, error code %2")
                           .arg(m_portName).arg(serial.error()));
                return;
            }
        }
        if (serial.waitForReadyRead(currentWaitTimeout)) {
//! [7] //! [8]
            // read request
            QByteArray requestData = serial.readAll();
            while (serial.waitForReadyRead(10))
                requestData += serial.readAll();
//! [8] //! [10]
            // write response
            const QByteArray responseData = currentRespone.toUtf8();
            serial.write(responseData);
            if (serial.waitForBytesWritten(m_waitTimeout)) {
                const QString request = QString::fromUtf8(requestData);
//! [12]
                emit this->request(request);
//! [10] //! [11] //! [12]
            } else {
                emit timeout(tr("Wait write response timeout %1")
                             .arg(QTime::currentTime().toString()));
            }
//! [9] //! [11]
        } else {
            emit timeout(tr("Wait read request timeout %1")
                         .arg(QTime::currentTime().toString()));
        }
//! [9]  //! [13]
        m_mutex.lock();
        if (currentPortName != m_portName) {
            currentPortName = m_portName;
            currentPortNameChanged = true;
        } else {
            currentPortNameChanged = false;
        }
        currentBaudRate = m_baudRate;
        currentWaitTimeout = m_waitTimeout;
        currentRespone = m_response;
        m_mutex.unlock();
```

5. Lập trình giao tiếp với Arduino

❖ Example: Read and Display Temperature

- Giao diện



5. Lập trình giao tiếp với Arduino

❖ Example: Read and Display Temperature

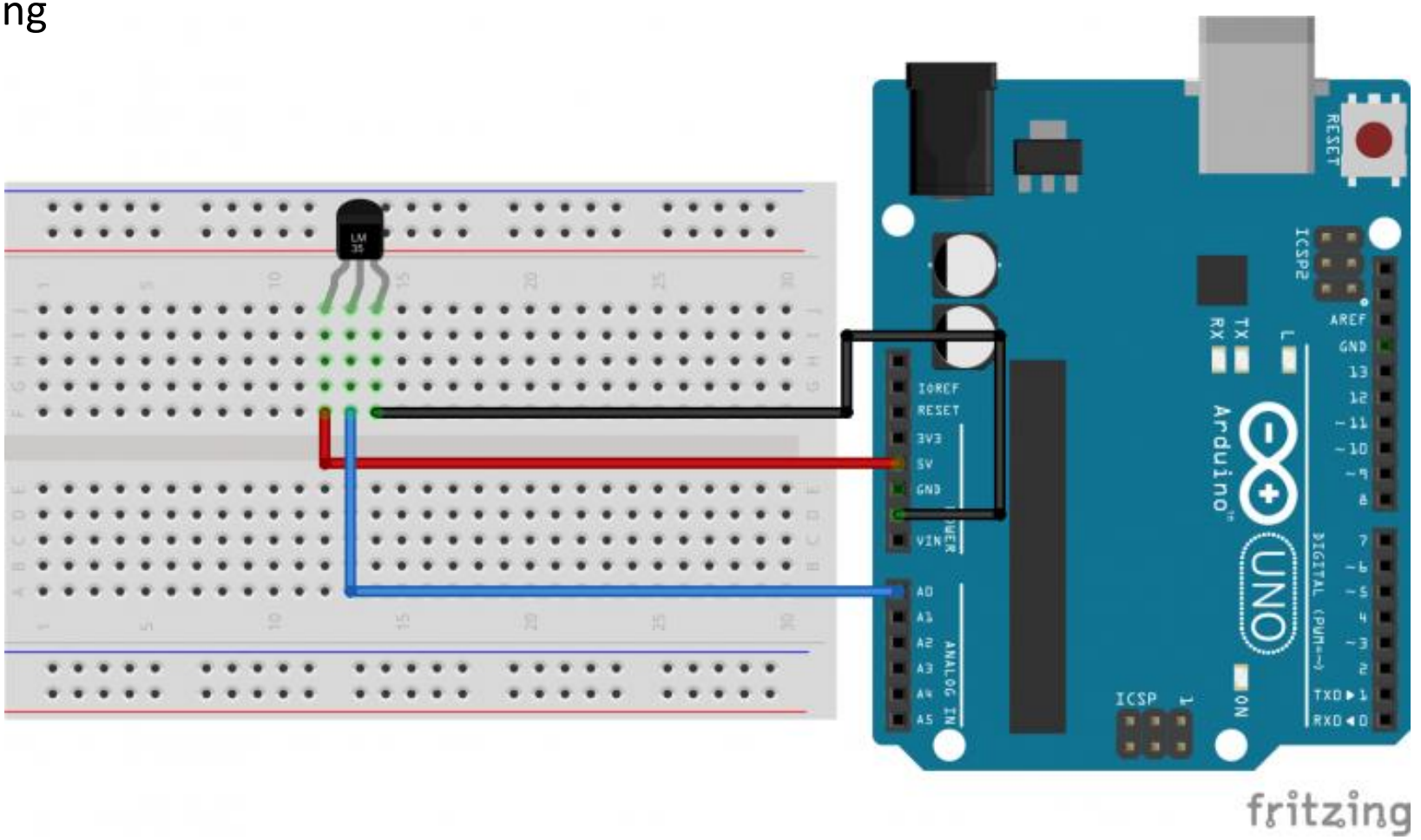
- Arduino Coding

```
int sensorPin = A0;// chân analog kết nối tới cảm biến LM35

void setup() {
  // put your setup code here, to run once:
  Serial.begin(9600);  //Khởi động Serial ở mức baudrate 9600
  // Bạn không cần phải pinMode cho các chân analog trước khi dùng nó
}

void loop() {
  // put your main code here, to run repeatedly:
  //đọc giá trị từ cảm biến LM35
  int reading = analogRead(sensorPin);

  //tính ra giá trị hiệu điện thế (đơn vị Volt) từ giá trị cảm biến
  float voltage = reading * 5.0 / 1024.0;

  // ở trên mình đã giới thiệu, cứ mỗi 10mV = 1 độ C.
  // Vì vậy nếu biến voltage là biến lưu hiệu điện thế (đơn vị Volt)
  // thì ta chỉ việc nhân voltage cho 100 là ra được nhiệt độ!

  float temp = voltage * 100.0;

  Serial.println(temp);
  delay(1000);//đợi 1 giây cho lần đọc tiếp theo
}
```

5. Lập trình giao tiếp với Arduino

❖ Example: Read and Display Temperature

- Arduino Coding

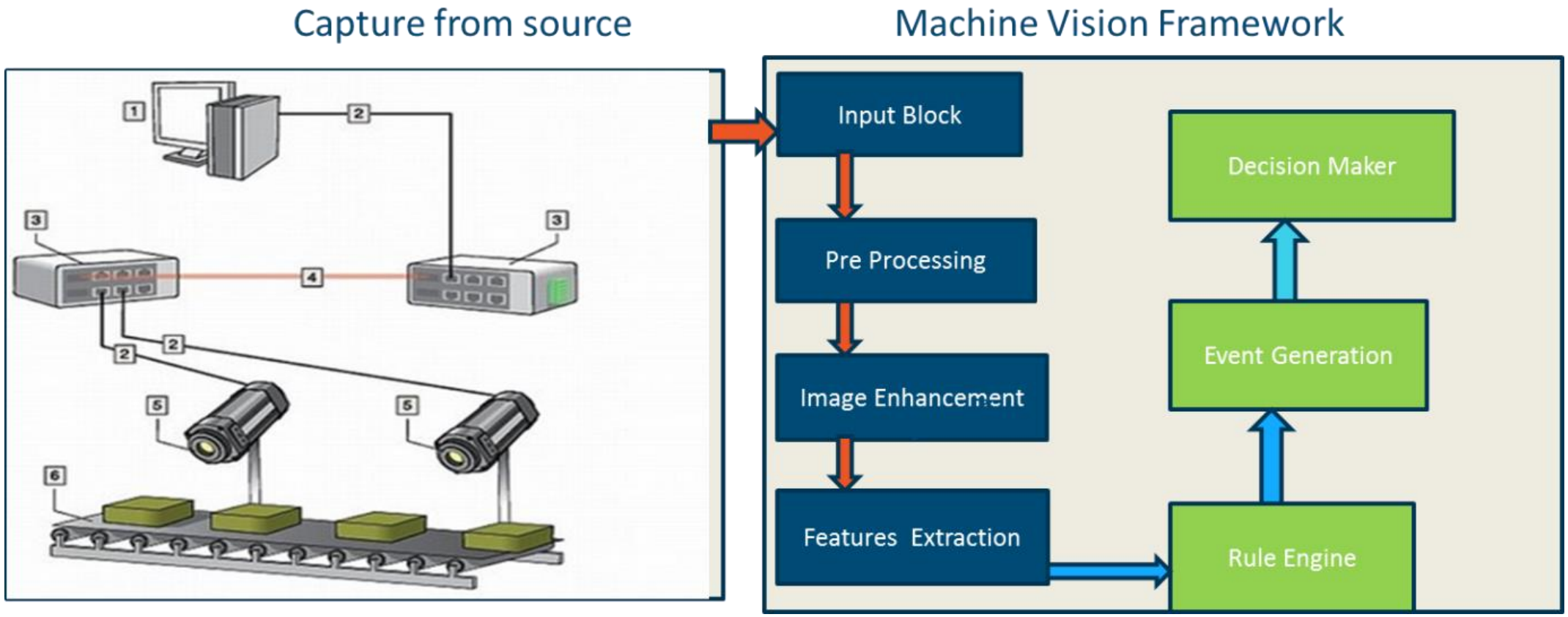


Chapter V. Hardware Interface Programming

- 1. Giới thiệu
- 2. Các chuẩn giao tiếp và giao thức truyền thông công nghiệp
- 3. Truyền thông nối tiếp với Qt
- 4. Đa luồng trong Qt
- 5. Lập trình giao tiếp với Arduino
- 6. Lập trình giao tiếp với camera

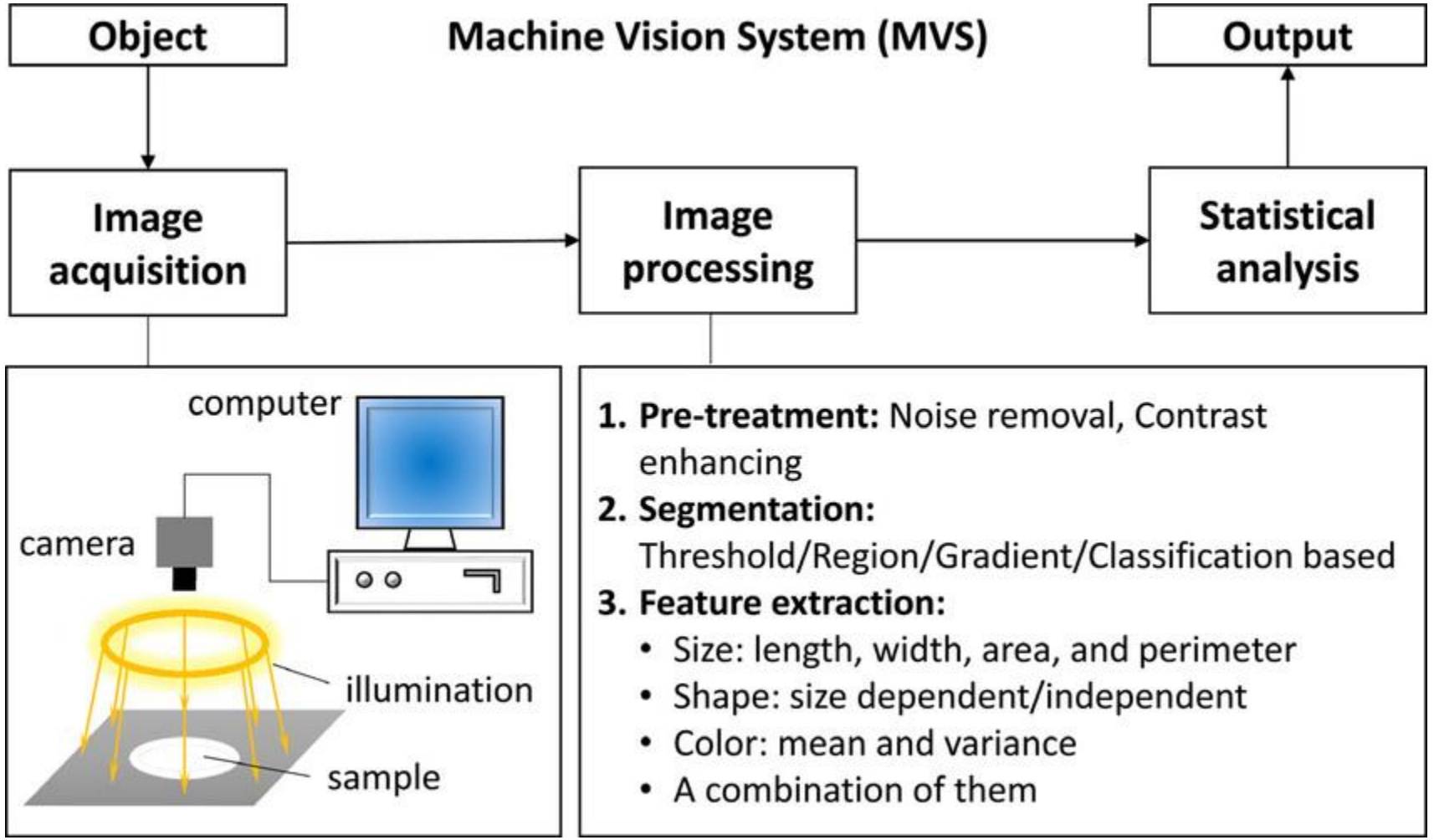
6. Lập trình giao tiếp với camera

❖ Machine Vision Framework



6. Lập trình giao tiếp với camera

❖ Machine Vision System



6. Lập trình giao tiếp với camera

❖ Machine Vision Software

- Machine vision software allows engineers and developers to design, deploy and manage vision applications.
- Vision applications are used by machines to extract and ingest data from visual imagery. Kinds of data available are geometric patterns (or other kinds of pattern recognition), object location, heat detection and mapping, measurements and alignments, or blob analysis.

https://www.trustradius.com/machine-vision

6. Lập trình giao tiếp với camera

❖ Machine Vision Software

Features of Machine Vision Software:

- Suite of tools for building 2D and 3D vision apps
- Support for multiple image types (e.g. analog, digital, color, monochrome, scans, etc.)
- Integratable with third-party smart cameras
- Blob detection & analysis
- Image processing, integration with analytics suites

https://www.trustradius.com/machine-vision

6. Lập trình giao tiếp với camera

❖ Machine Vision Software: Example

Features of Machine Vision Software:

- **Auto ID Tools**: Decoding all standard linear barcodes, Data Matrix, and other symbols, Optical Character Recognition (OCR) and Verification (OCV), etc.
- **Image Processing Tools**: Image arithmetic, image rotation and warping, binary and grayscale morphology, edge enhancement, other image filtering, etc.
- **Image Analysis Tools**: Flaw detection, histogram analysis, blob analysis, template and pattern recognition, object location and orientation detections, etc.
- **Calibrated Dimensional Measurements**: Variety of pre-configured measurements such as line intersection, point-to-point distance, point-to-line normal, etc.

https://www.redlinesolutions.com/index.php/solutions/machine-vision/machine-vision-software

6. Lập trình giao tiếp với camera

❖ Machine Vision Software: Example

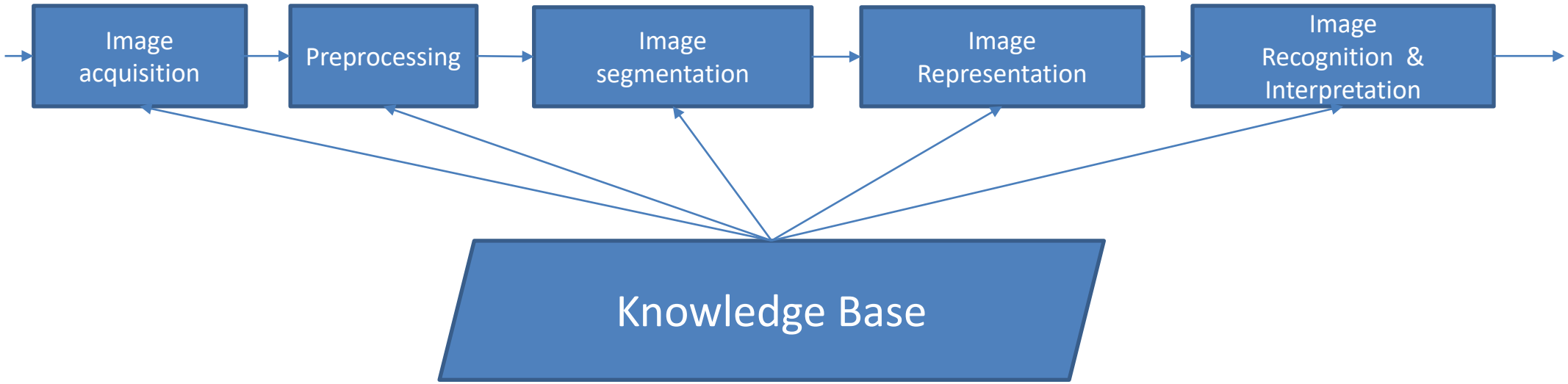
Features of Machine Vision Software:

- **Intellifind Tool**: Geometric pattern match tool for robust pattern location and pattern recognition in noisy images; includes scale measurement.
- **Color Support**: Color imaging, color visualization and other color tools allow color checking and identification applications. Color-based dimensional gauging applications are also possible.
- **Application Specific and Custom Tools**: User-defined expressions and math, custom scripted vision processing tools, etc.

https://www.redlinesolutions.com/index.php/solutions/machine-vision/machine-vision-software

6. Lập trình giao tiếp với camera

❖ Real-Time Image Processing

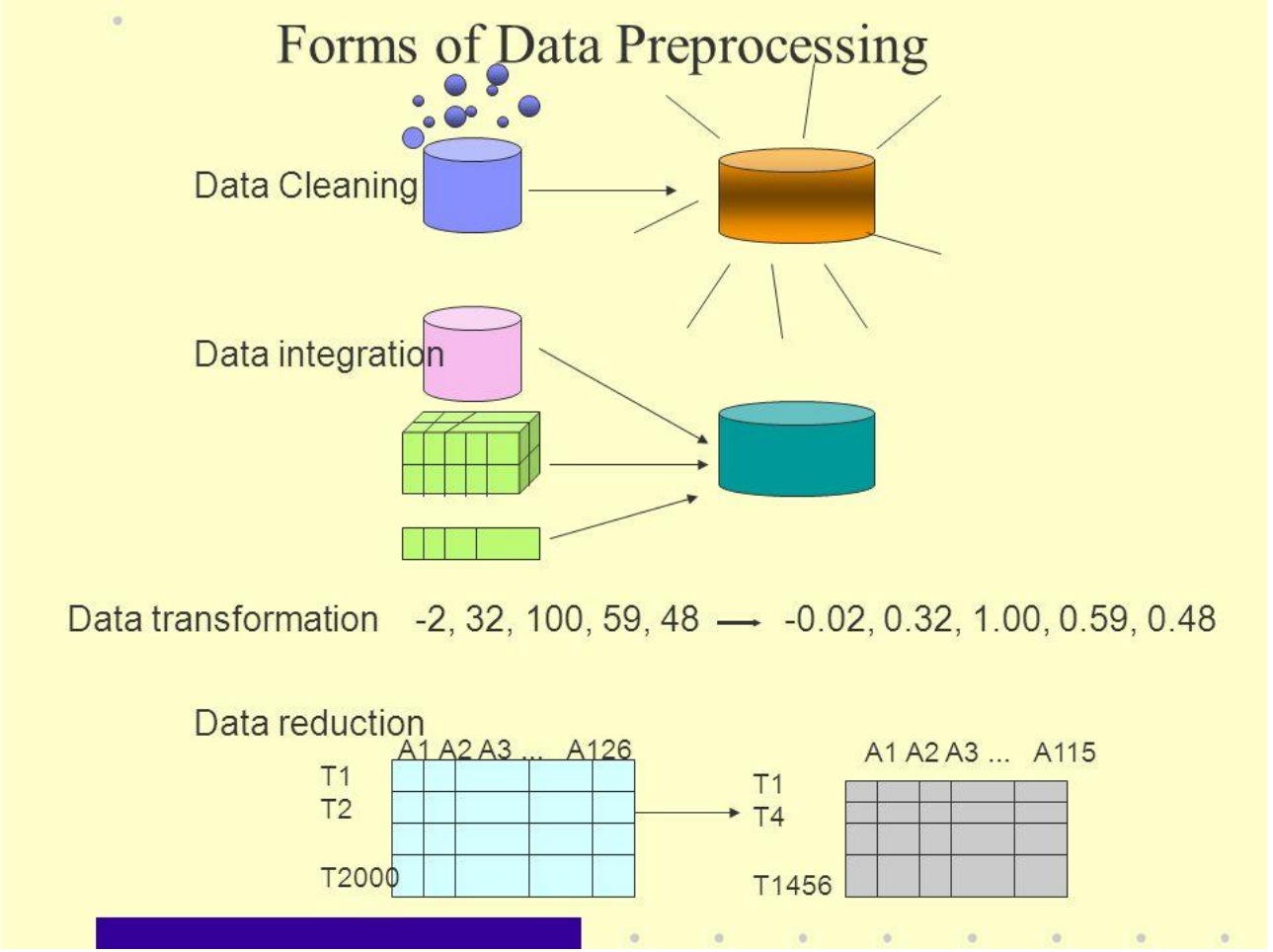


The steps in image processing

6. Lập trình giao tiếp với camera

❖ Real-Time Image Processing

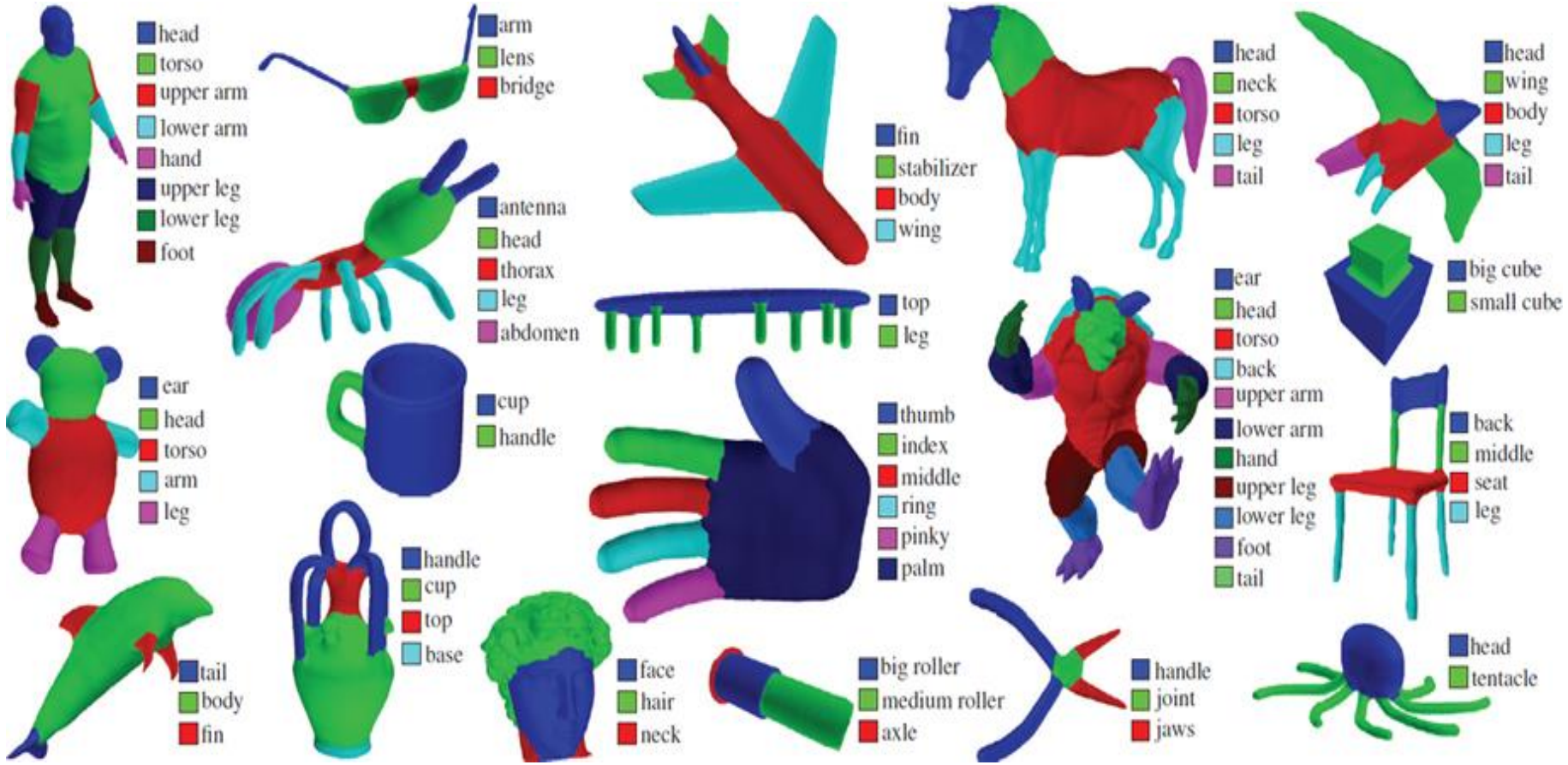
- Preprocessing
  - ➢ Noise Removal
  - ➢ Image Enhancement
  - ➢ …



6. Lập trình giao tiếp với camera

❖ Real-Time Image Processing

- Segmentation
- ➢ Line Detection
- ➢ Edge Detection
- ➢ Thresholding
- ➢ Watershed
- ➢ Color Detection
- ➢ …



6. Lập trình giao tiếp với camera

❖ **Real-Time Image Processing**

- Recognition and Localization

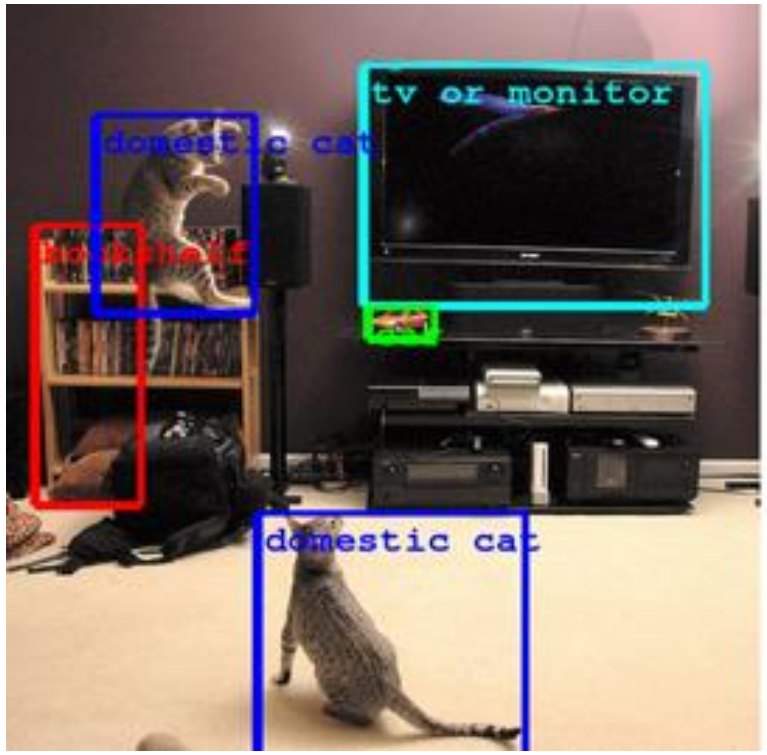
➢ Shape Detection

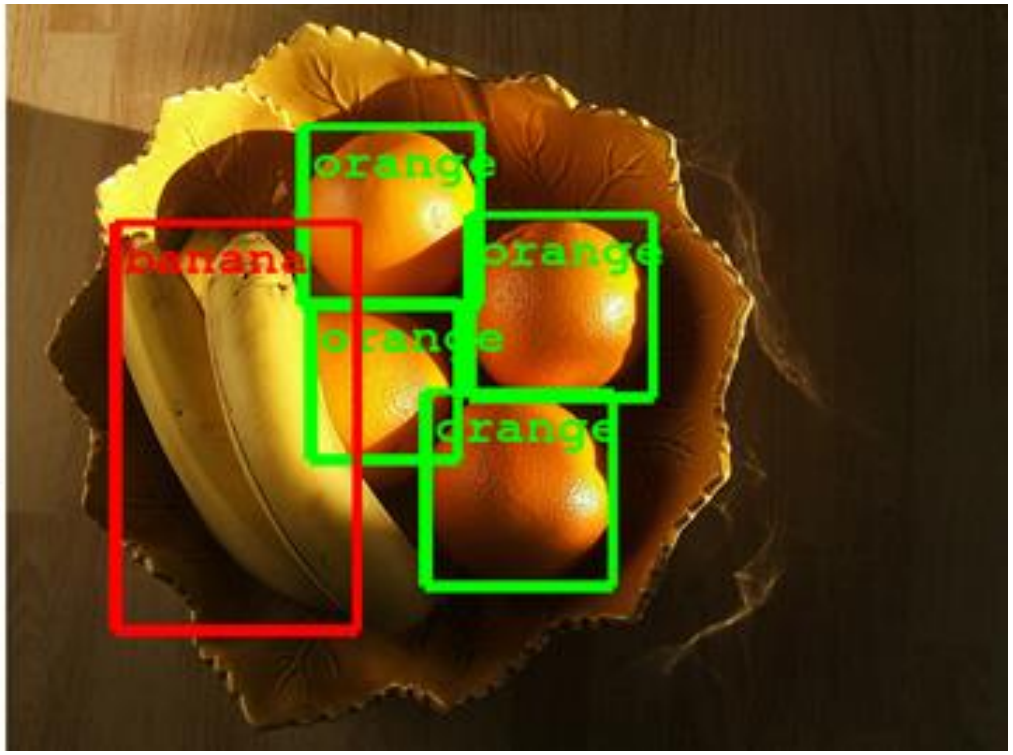
➢ Template Matching

➢ Feature Matching

➢ Deep Learning

➢ …



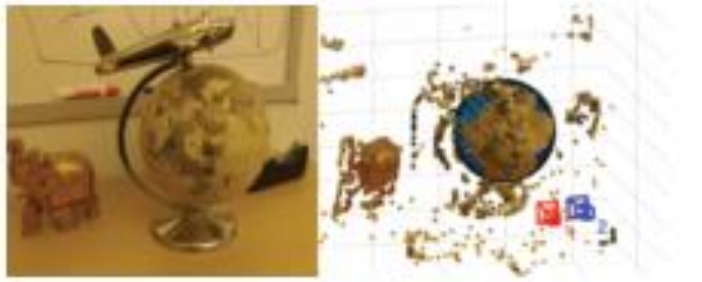


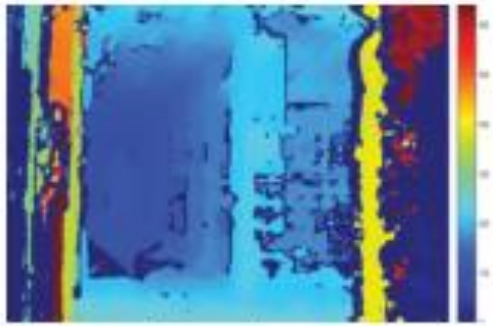
6. Lập trình giao tiếp với camera

❖ **Real-Time Image Processing**

- Camera Calibration
  - ➢ 2-D and 3-D Object Measurement
  - ➢ Robot-Camera Calibration
  - ➢ …

Remove Lens Distortion


Estimate 3-D Structure from Camera Motion


Estimate Depth
Using a Stereo Camera

Measure Planar Objects

6. Lập trình giao tiếp với camera

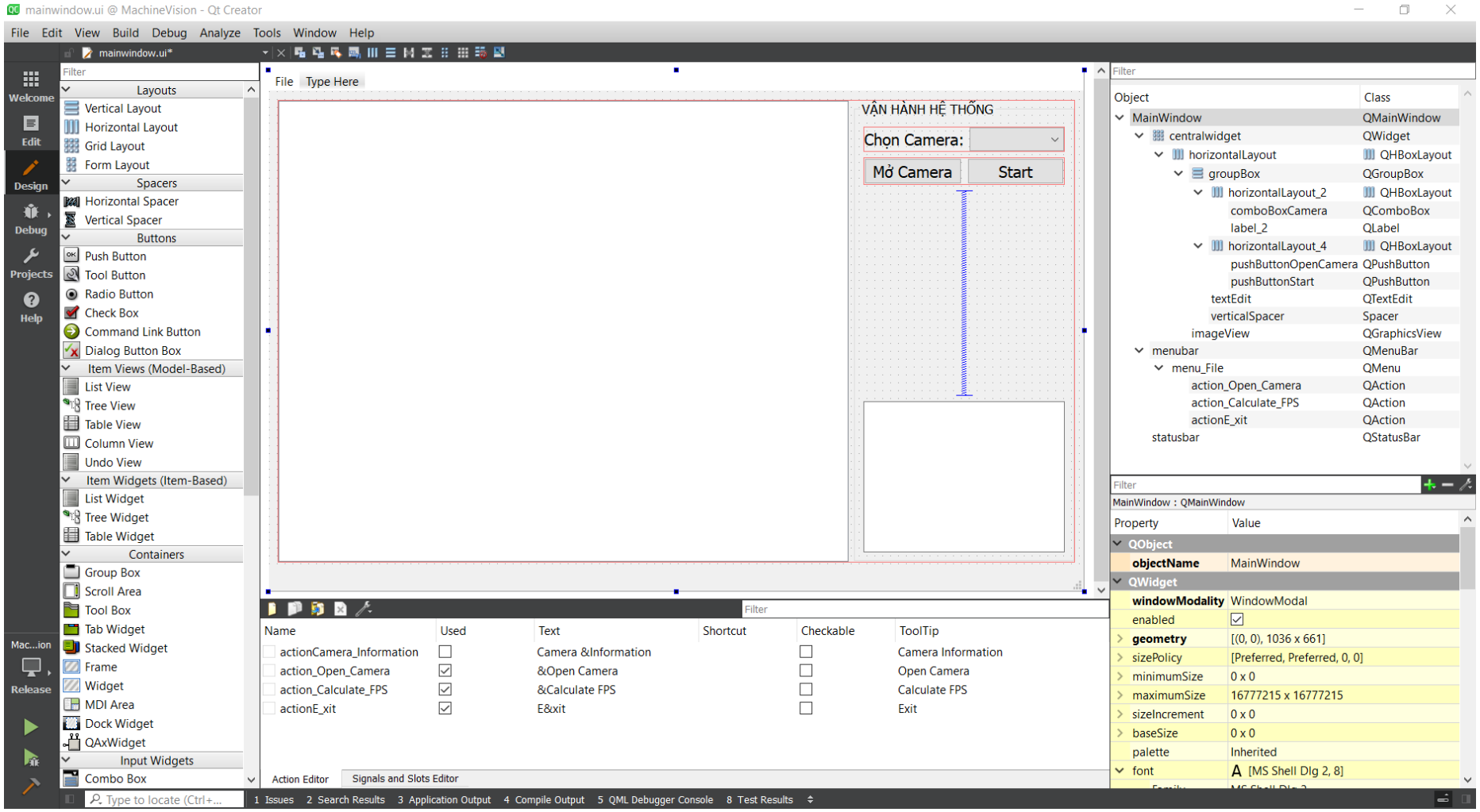
Camera Connection: Using Qthread

```
QT       += core gui multimedia network concurrent

greaterThan(QT_MAJOR_VERSION, 4): QT += widgets printsupport
```

```
win32:CONFIG(release, debug|release): LIBS += -LD:/DEV-LIB/opencv-
4.1.0/build/x64/vc15/lib/ -lopencv_world410
else:win32:CONFIG(debug, debug|release): LIBS += -LD:/DEV-
LIB/opencv-4.1.0/build/x64/vc15/lib/ -lopencv_world410d

INCLUDEPATH += D:/DEV-LIB/opencv-4.1.0/build/include
DEPENDPATH += D:/DEV-LIB/opencv-4.1.0/build/include
```

6. Lập trình giao tiếp với camera

❖ Camera Connection: Using Qthread



6. Lập trình giao tiếp với camera

Camera Connection: Using Qthread

Tên	Kiểu	Nhãn hiển thị	Mô tả / Chức năng
comboBoxCamera	QComboBox		ComboBox hiển thị các máy ảnh kết nối với máy tính mà chương trình phát hiện được
label_2	QLabel	Chọn Camera:	Hiển thị hướng dẫn chọn máy ảnh
pushButtonOpenCamera	QPushButton	Mở Camera	Nút mở máy ảnh được lựa chọn trong comboBoxCamera
pushButtonStart	QPushButton	Start	Nút để bắt đầu chương trình xử lý ảnh
textEdit	QTextEdit		Hộp văn bản với nhật ký ứng dụng
imageView	QGraphicsView		Hiển thị hình ảnh thu được từ máy ảnh hoặc hình ảnh đã được xử lý
menubar	QMenuBar		Thanh menu
menu_File	QMenu	File	Menu file
action_Open_Camera	QAction	Open Camera	Nút mở máy ảnh trên menu file
action_Calculate_FPS	QAction	Calculate FPS	Nút tính tốc độ khung hình máy ảnh trên menu file
actionE_xit	QAction	Exit	Nút thoát chương trình trên menu file
statusbar	QStatusBar		Thanh trạng thái

6. Lập trình giao tiếp với camera

Camera Connection: Using Qthread

- Mainwindow.h

```
#ifndef MAINWINDOW_H
#define MAINWINDOW_H

#include <QMainWindow>
#include <QGraphicsScene>
#include <QGraphicsPixmapItem>
#include <QMutex>
#include <QStandardItemModel>
#include <QTimer>

#include "opencv2/opencv.hpp"

#include "capture_thread.h"

QT_BEGIN_NAMESPACE
namespace Ui { class MainWindow; }
QT_END_NAMESPACE

class MainWindow : public QMainWindow
{
    Q_OBJECT

public:
    MainWindow(QWidget *parent = nullptr);
    ~MainWindow();

private slots:
    void updateFrame(cv::Mat*);
    void updateFPS(float);
    void on_actionE_xit_triggered();
    void on_action_Open_Camera_triggered();
    void on_action_Calculate_FPS_triggered();
    void on_pushButtonStart_clicked();
    void on_pushButtonOpenCamera_clicked();

private:
    void Camera_Information();

private:
    Ui::MainWindow *ui;
    QGraphicsScene *imageScene;
    QStandardItemModel *list_model;
    cv::Mat currentFrame, showFrame;

    // for capture thread
    QMutex *data_lock;
    CaptureThread *capturer;
};
#endif // MAINWINDOW_H
```

6. Lập trình giao tiếp với camera

Camera Connection: Using Qthread

- Mainwindow.cpp

```cpp
#include <QApplication>
#include <QFileDialog>
#include <QMessageBox>
#include <QPixmap>
#include <QKeyEvent>
#include <QDebug>
#include <QCameraInfo>
#include <QGridLayout>
#include <QIcon>
#include <QStandardItem>
#include <QSize>

#include "opencv2/videoio.hpp"

#include "utilities.h"
#include "mainwindow.h"
#include "ui_mainwindow.h"

```

```cpp
MainWindow::MainWindow(QWidget *parent)
    : QMainWindow(parent)
    , ui(new Ui::MainWindow)
    , capturer(nullptr)
{
    ui->setupUi(this);

    imageScene = new QGraphicsScene(this);
    ui->imageView->setScene(imageScene);
    ui->imageView->setHorizontalScrollBarPolicy(Qt::ScrollBarAlwaysOff);
    ui->imageView->setVerticalScrollBarPolicy(Qt::ScrollBarAlwaysOff);
    ui->imageView->setSizePolicy(QSizePolicy::Ignored, QSizePolicy::Ignored);

    ui->statusbar->showMessage("MachineVision is Ready!");

    Camera_Information();

    setWindowState(Qt::WindowMaximized);

    data_lock = new QMutex();
}
```

6. Lập trình giao tiếp với camera

❖ Camera Connection: Using Qthread

- Mainwindow.cpp

```
41  MainWindow::~MainWindow()
42  {
43      delete ui;
44  }
45
46  void MainWindow::Camera_Information()
47  {
48      QList<QCameraInfo> cameras = QCameraInfo::availableCameras();
49
50      foreach (const QCameraInfo &cameraInfo, cameras) {
51          ui->comboBoxCamera->addItem(cameraInfo.description());
52      }
53
54  }
55
56  void MainWindow::on_actionE_xit_triggered()
57  {
58      QApplication::quit();
59  }
```

6. Lập trình giao tiếp với camera

Camera Connection: Using Qthread

- Mainwindow.cpp

```
61  void MainWindow::updateFrame(cv::Mat *mat)
62  {
63      int width = 0, height = 0;
64      width = ui->imageView->width();
65      height = ui->imageView->height();
66      data_lock->lock();
67      currentFrame = *mat;
68      showFrame = currentFrame.clone();
69      cv::resize(showFrame, showFrame, cv::Size(width, height));
70      data_lock->unlock();
71
72      QImage frame(
73          showFrame.data,
74          showFrame.cols,
75          showFrame.rows,
76          showFrame.step,
77          QImage::Format_RGB888);
78      QPixmap image = QPixmap::fromImage(frame);
79
80      imageScene->clear();
81      ui->imageView->resetMatrix();
82      imageScene->addPixmap(image);
83      imageScene->update();
84      ui->imageView->setSceneRect(image.rect());
85  }
```

6. Lập trình giao tiếp với camera

Camera Connection: Using Qthread

- Mainwindow.cpp

```
87  void MainWindow::updateFPS(float fps)
88  {
89      ui->statusbar->showMessage(QString("FPS of current camera is %1").arg(fps));
90  }
91
92  void MainWindow::on_action_Open_Camera_triggered()
93  {
94      if(capturer != nullptr) {
95          // if a thread is already running, stop it
96          capturer->setRunning(false);
97          disconnect(capturer, &CaptureThread::frameCaptured, this,
    &MainWindow::updateFrame);
98          disconnect(capturer, &CaptureThread::fpsChanged, this, &MainWindow::updateFPS);
99          connect(capturer, &CaptureThread::finished, capturer,
    &CaptureThread::deleteLater);
100     }
101
102     // Usually, the Index of the first camera is 0.
103     int camID = ui->comboBoxCamera->currentIndex();
104     capturer = new CaptureThread(camID, data_lock);
105     connect(capturer, &CaptureThread::frameCaptured, this, &MainWindow::updateFrame);
106     connect(capturer, &CaptureThread::fpsChanged, this, &MainWindow::updateFPS);
107     capturer->start();
108     ui->statusbar->showMessage(QString("Capturing Camera %1").arg(camID));
109 }
```

6. Lập trình giao tiếp với camera

Camera Connection: Using Qthread

- Mainwindow.cpp

```
111 void MainWindow::on_action_Calculate_FPS_triggered()
112 {
113     if(capturer != nullptr) {
114             capturer->startCalcFPS();
115         }
116 }
117
118 void MainWindow::on_pushButtonStart_clicked()
119 {
120     if(capturer == nullptr) {
121         return;
122     }
123     capturer->setImageProcessingStatus(true);
124 }
125
126 void MainWindow::on_pushButtonOpenCamera_clicked()
127 {
128     on_action_Open_Camera_triggered();
129 }
130
```

6. Lập trình giao tiếp với camera

Camera Connection: Using Qthread

- capture_thread.h

```
// mode: c++
#ifndef CAPTURE_THREAD_H
#define CAPTURE_THREAD_H

#include <QString>
#include <QThread>
#include <QMutex>
#include <QElapsedTimer>

#include <opencv2/opencv.hpp>
#include <opencv2/videoio.hpp>
#include <opencv2/tracking.hpp>
#include <opencv2/core/ocl.hpp>
#include <opencv2/video/background_segm.hpp>

using namespace std;

class CaptureThread : public QThread
{
    Q_OBJECT
public:
    CaptureThread(int camera, QMutex *lock);
    CaptureThread(QString videoPath, QMutex *lock);
    ~CaptureThread();
    //...
    void setRunning(bool run) {running = run; }
    void startCalcFPS() {fps_calculating = true; }
    enum VideoSavingStatus {
                            STARTING,
                            STARTED,
                            STOPPING,
                            STOPPED
    };
```

6. Lập trình giao tiếp với camera

Camera Connection: Using Qthread

- capture_thread.h

```
35      void setVideoSavingStatus(VideoSavingStatus status) {video_saving_status = status; }
36      void setImageProcessingStatus(bool status) {
37          image_processing_status = status;
38          target_detected = false;
39          if(video_saving_status != STOPPED) video_saving_status = STOPPING;
40      }
41
42  protected:
43      void run() override;
44
45  signals:
46      void frameCaptured(cv::Mat *data);
47      void fpsChanged(float fps);
48      void videoSaved(QString name);
49
50  private:
51      void calculateFPS(cv::VideoCapture &cap);
52      void startSavingVideo(cv::Mat &firstFrame);
53      void stopSavingVideo();
54      void imageProcessing(cv::Mat &frame);
55      void getRandomColors(vector<cv::Scalar>& colors, int numColors);
56
```

6. Lập trình giao tiếp với camera

Camera Connection: Using Qthread

- capture_thread.h

```
57  private:
58      bool running;
59      int cameraID;
60      QString videoPath;
61      QMutex *data_lock;
62      cv::Mat frame;
63
64      // FPS calculating
65      bool fps_calculating;
66      float fps;
67
68      // video saving
69      int frame_width, frame_height;
70      VideoSavingStatus video_saving_status;
71      QString saved_video_name;
72      cv::VideoWriter *video_writer;
73
74      // image analysis
75      bool image_processing_status;
76      bool target_detected;
77  };
78
79  #endif // CAPTURE_THREAD_H
```

6. Lập trình giao tiếp với camera

Camera Connection: Using Qthread

- capture_thread.cpp

```cpp
#include <QtConcurrent>
#include <QDebug>
#include <QtMath>
#include <QTime>

#include "utilities.h"
#include "capture_thread.h"

CaptureThread::CaptureThread(int camera, QMutex *lock):
    running(false), cameraID(camera), videoPath(""), data_lock(lock)
{
    //...
    fps_calculating = false;
    fps = 0.0;

    frame_width = frame_height = 0;
    video_saving_status = STOPPED;
    saved_video_name = "";
    video_writer = nullptr;

    image_processing_status = false;
}
```

6. Lập trình giao tiếp với camera

❖ Camera Connection: Using Qthread

- capture_thread.cpp

```
24  CaptureThread::CaptureThread(QString videoPath, QMutex *lock):
25      running(false), cameraID(-1), videoPath(videoPath), data_lock(lock)
26  {
27      fps_calculating = false;
28      fps = 0.0;
29
30      frame_width = frame_height = 0;
31      video_saving_status = STOPPED;
32      saved_video_name = "";
33      video_writer = nullptr;
34
35      image_processing_status = false;
36  }
```

6. Lập trình giao tiếp với camera

Camera Connection: Using Qthread

- capture_thread.cpp

```
41  void CaptureThread::run() {
42      running = true;
43      cv::VideoCapture cap(cameraID);
44      cv::Mat tmp_frame;
45
46      frame_width = cap.get(cv::CAP_PROP_FRAME_WIDTH);
47      frame_height = cap.get(cv::CAP_PROP_FRAME_HEIGHT);
48
49      while(running) {
50          cap >> tmp_frame;
51          if (tmp_frame.empty()) {
52              break;
53          }
54          if(image_processing_status) {
55              imageProcessing(tmp_frame);
56          }
57          if(video_saving_status == STARTING) {
58              startSavingVideo(tmp_frame);
59          }
60          if(video_saving_status == STARTED) {
61              video_writer->write(tmp_frame);
62          }
63          if(video_saving_status == STOPPING) {
64              stopSavingVideo();
65          }
66
67          cvtColor(tmp_frame, tmp_frame, cv::COLOR_BGR2RGB);
68          data_lock->lock();
69          frame = tmp_frame;
70          data_lock->unlock();
71          emit frameCaptured(&frame);
72          if(fps_calculating) {
73              calculateFPS(cap);
74          }
75      }
76      cap.release();
77      running = false;
78  }
```

6. Lập trình giao tiếp với camera

Camera Connection: Using Qthread

- capture_thread.cpp

```
80  void CaptureThread::calculateFPS(cv::VideoCapture &cap)
81  {
82      const int count_to_read = 100;
83      cv::Mat tmp_frame;
84      QTime timer;
85      timer.start();
86      for(int i = 0; i < count_to_read; i++) {
87              cap >> tmp_frame;
88      }
89      int elapsed_ms = timer.elapsed();
90      fps = count_to_read / (elapsed_ms / 1000.0f);
91      fps_calculating = false;
92      emit fpsChanged(fps);
93      // ....
94  }
95
96
97  void CaptureThread::startSavingVideo(cv::Mat &firstFrame)
98  {
99      saved_video_name = Utilities::newSavedVideoName();
100
101     QString cover = Utilities::getSavedVideoPath(saved_video_name, "jpg");
102     cv::imwrite(cover.toStdString(), firstFrame);
103
104     video_writer = new cv::VideoWriter(
105         Utilities::getSavedVideoPath(saved_video_name, "avi").toStdString(),
106         cv::VideoWriter::fourcc('M','J','P','G'),
107         int(fps)? int(fps): 30,
108         cv::Size(frame_width,frame_height));
109     video_saving_status = STARTED;
110 }
```

6. Lập trình giao tiếp với camera

❖ Camera Connection: Using Qthread

- capture_thread.cpp

```
113 void CaptureThread::stopSavingVideo()
114 {
115     video_saving_status = STOPPED;
116     video_writer->release();
117     delete video_writer;
118     video_writer = nullptr;
119     emit videoSaved(saved_video_name);
120 }
121
122 void CaptureThread::getRandomColors(vector<cv::Scalar>& colors, int numColors)
123 {
124     cv::RNG rng(0);
125     for (int i = 0; i < numColors; i++)
126         colors.push_back(cv::Scalar(rng.uniform(0, 255), rng.uniform(0, 255), rng.uniform(0, 255)));
127 }
```

6. Lập trình giao tiếp với camera

❖ Camera Connection: Using Qthread

- capture_thread.cpp

```
129 void CaptureThread::imageProcessing(cv::Mat &frame)
130 {
131     cv::Mat src = frame.clone();
132     //Prepocessing
133     //chuyển ảnh màu về ảnh xám
134     cv::Mat gray_img;
135     cv::cvtColor(src, gray_img, CV_BGR2GRAY);
136     //denoise
137     cv::Mat denoise_img;
138     cv::GaussianBlur(gray_img, denoise_img, cv::Size(5,5), 1.5, 1.5);
139     //image segmentation
140     //edge-based segmentation
141     cv::Mat edges;
142     cv::Canny(denoise_img, edges, 50, 150);
143     //region-based segmentation
144     cv::Mat thresh_img;
145     cv::threshold(denoise_img, thresh_img, 150, 255, cv::THRESH_BINARY | cv::THRESH_OTSU);
146
147     frame = thresh_img.clone();
148 }
```