# C++ Course 2: C++ Language Basics 1

2020 by Oleksiy Grechnyev

# C++ history

- 1979-1998 Development
- C++ 98 : First Official Standard
- C++ 11 : New (Big) Standard
  Rvalue references + move semantics
  Lambda expressions + **std::function**
  Concurrency (**thread**, **future**, ...)
  Smart pointers **(shared_ptr**, **unique_ptr**)
  **auto** + **decltype**
- C++ 14 : Small changes only
- C++ 17 : **filesystem**, **any**, **optional**, ...
- C++ 20: ???

# Integer types

*Integer types*
**char** (8 bit) , **short** (16 bit), **int** (32 bit), **long** (32/64), **long long** (64 bit), **size_t** (32/64)

*Sign modifiers*
**signed** , **unsigned** , **signed** = default (except **char**)

For example:
**int** : −2147483648 .. 2147483647
**unsigned int** : 0 .. 4294967295

*Fixed-width types* (C++ 11):
**int8_t**, **int16_t**, **int32_t**, **int64_t**
**uint8_t**, **uint16_t**, **uint32_t**, **uint64_t**

# The danger of mixing signed and unsigned numbers

```
int a = -10;
unsigned int b = 1;
cout << "a + b = " << a + b << endl;
```

# The danger of mixing signed and unsigned numbers

```
int a = -10;
unsigned int b = 1;
cout << "a + b = " << a + b << endl;
```

Result (OUCH !):

```
a + b = 4294967287
```

**int** and **unsigned int** have the same size (32 bit)
**a**+**b** has type **unsigned int**

But **a** + **(int)b** is OK !
The sum is now **int** (signed).

# numeric_limits : Example 2_1

*Type alias*

```
using MyType = long long;
// typedef long long MyType;    // C++ 98
```

**numeric_limits<MyType>** : Type information

```
cout << boolalpha; // Write bool as true/false
cout << "sizeof(MyType) = " <<  sizeof(MyType) << endl; // Size in bytes
cout << "is_signed = " << numeric_limits<MyType>:: is_signed << endl;
cout << "is_integer = " << numeric_limits<MyType>:: is_integer << endl;
cout << "is_exact = " << numeric_limits<MyType>:: is_exact << endl;
cout << "has_infinity = " << numeric_limits<MyType>:: has_infinity << endl;
cout << "has_quiet_NaN = " << numeric_limits<MyType>:: has_quiet_NaN << endl;
cout << "digits = " << numeric_limits<MyType>:: digits << endl;
cout << "digits10 = " << numeric_limits<MyType>:: digits10 << endl;
cout << "lowest() = " << numeric_limits<MyType>:: lowest() << endl;
cout << "min() = " << numeric_limits<MyType>:: min() << endl;
cout << "max() = " << numeric_limits<MyType>:: max() << endl;
```

# numeric_limits : Example 2_1

*Type alias*

```
using MyType = long long;
// typedef long long MyType;    // C++ 98
```

**numeric_limits<MyType>** : Type information

```
sizeof(MyType) = 8
is_signed = true
is_integer = true
is_exact = true
has_infinity = false
has_quiet_NaN = false
digits = 63
digits10 = 18
lowest() = -9223372036854775808
min() = -9223372036854775808
max() = 9223372036854775807
```

# Other types

Boolean:
**bool** (8 bit) : **false** (0), **true** (1)

Floating point:
**float** (32 bit), **double** (64 bit), **long double** (128 bit ?)
Always use **double** !

Small difference between 2 large doubles:

```
double a = 1.0e15;
double b = 1.0e15 + 0.1234;
cout << endl << "b - a = " << b-a << endl;
```

# Other types

Boolean:
**bool** (8 bit) : **false** (0), **true** (1)

Floating point:
**float** (32 bit), **double** (64 bit), **long double** (128 bit ?)
Always use **double** !

Small difference between 2 large doubles:

```
double a = 1.0e15;
double b = 1.0e15 + 0.1234;
cout << endl << "b - a = " << b-a << endl;
```

```
b - a = 0.125
```

Or, if you take 1.0e+16

```
b - a = 0
```

# Literals

```
1234                int
4000000000u         unsigned int
8’000’000’000ll     long long
8000000000ull       unsigned long long
1.23e-4             double
1.23e-4f            float
1.23e-4d            long double
’M’                 char
”Dina Meyer”        const char[11]  (including ‘\0’, NOT std::string !)
false               bool

Hexadecimal, octal, binary (C++ 14) literals

0xFF                int (HEX) = 255
0100                int (OCTAL) = 64
0b100               int (BIN) = 4
```

# Operators

| Precedence level goes down the table | | |
|---|---|---|
| **Operator name** | Associativity | Operators |
| **Scope resolution** (included in C++ ) | left to right | :: |
| Primary | left to right | () [] . -> **dynamic_cast typeid** |
| Unary | right to left | ++ -- + - ! ~ & * **(type_name) sizeof new delete** |
| Pointer to Member(C++) | left to right | *. -> |
| Multiplicative | left to right | * / % |
| Additive | left to right | + - |
| Bitwise Shift | left to right | << >> |
| Relational | left to right | < > <= >= |
| Equality | left to right | == != |
| Bitwise AND | left to right | & |
| Bitwise Exclusive OR | left to right | ^ |
| Bitwise Inclusive OR | left to right | \| |
| Logical AND | left to right | && |
| Logical OR | left to right | \|\| |
| Conditional | right to left | ? : |
| Assignment | right to left | = += -= *= /= <<= >>= %= &= ^= \|= |
| Comma | right to left | , |

# True story: "negative" bit shift

**int a = 12 << -1;** // What is wrong with this ?
The compiler gives a WARNING :
warning: left shift count is negative
Negative shift is NOT ALLOWED !!!
In particular **12 << -1** is NOT the same as **12 >> 1** !!!
The behavior is UNDEFINED !

- People copypasted some bad code from the internet
- IGNORED the warnings
- Gave the code to Android developers
- Algorithm developers : "My code works !"
- Android developers : "Nothing works !"
- The results are different on different platforms! BAD!
- Took a *long time* to fix this.

# Make this code work on platform XXX !

Suppose some algorithm "works on my computer" but not on Android!

The correct question is "WHY are the results different ???"
C++ code should work the same everywhere! This is called standard!

Does the code conform to a certain C++ standard (e.g. C++ 17)?

**YES**:
Than it should work on every compiler and every platform (if not it is a compiler/library bug)

**NO**:
The code relies on *undefined* or *compiler-specific* behavior.
Then this is a BAD code and BAD algorithm and the author is responsible.
"Works on my computer" is not a good argument.

# Operators 1

**&a** Address of a variable **a** (This is NOT a reference declaration !)
***b** Pointer **b** dereferencing (This is NOT a pointer declaration !)

```
int a = 17;
int *b = &a;   // &a = address of a
cout << *b;   // Prints 17
```

**a.c** Member access operator
**b->c** Member access operator (pointer), equivalent to (*b).c
**ns::c** Scope resolution (namespace members and static class members)

```
std::string a = ”Mary had a little lamb”;
std::string *b = &a;
std::cout << a.length() << std::endl;
std::cout << b->length() << std::endl;
```

# Operators 2

**condition ? value1 : value2** Conditional (a.k.a. ternary) operator

```
cout <<  (a > 0 ? a : -a);
```

**a = b** Assignment operator

```
a = b = (c = d + 13)*2;
```

, Comma operator (Evaluates all expressions, returns the last one)

```
int a = 13;
int b = (a++, ++a, a+1);
```

Most operators can be *overloaded* in C++ for your classes (except for **a.b** and **a::b** !)

# Variable declaration and initialization : Example 2_2

```
int i1(17);    // This does not work for class fields !
int i2 = 17;  // Forbidden with an explicit constructor
int i3 = int(17);  // No copy/move here!
int i4{17};        // List initialization with std::initializer_list
int i5 = {17};   // Forbidden with an explicit constructor
```

All this declarations call constructor **once**, no assignment/copy/move !

Does not exist in C++ !

```
Warrior w{
        name: ”Karin Koenig”,
        weapon: ”Rapier”,
        age: 25
};
```

# auto, decltype, decltype(auto)

**auto** = Automatic type inference

```
int a = 13;
auto b = a;   // b is int = 13
```

**decltype(a)** = Type of variable **a**

```
decltype(a)  c = 14;  // c is int = 14
```

**decltype(auto)** = A version of **auto** following **decltype** rules (C++ 14)

```
int & d = a; // d is a reference to a
auto e = d; // e is int = 13, ref is ignored
decltype(auto)  f = d; // f is a ref to a
```

And now we try to change variable **a** ...

```
a = 22;
```

What are the values of **a, b, c, d, e, f** ?

# auto, decltype, decltype(auto)

**auto** = Automatic type inference

```
int a = 13;
auto b = a;   // b is int = 13
```

**decltype(a)** = Type of variable **a**

```
decltype(a)  c = 14;  // c is int = 14
```

**decltype(auto)** = A version of **auto** following **decltype** rules (C++ 14)

```
int & d = a; // d is a reference to a
auto e = d; // e is int = 13, ref is ignored
decltype(auto)  f = d; // f is a ref to a
```

And now we try to change variable **a** ...

```
a = 22;
```

a==**22**, b==13, c==14, d==**22**, e==13, f==**22**

# Move and swap operations

**std::move** = Move an object (No copying)

```
string s1("Brianna");
string s2 = move(s1);
```

**std::swap** = Swap two objects (No copying)

```
string s3("Mira");
string s4("Visas");
swap(s3, s4);
```

What are the values of **s1**, **s2**, **s3**, **s4** ?

# Move and swap operations

**std::move** = Move an object (No copying)

```
string s1("Brianna");
string s2 = move(s1);
```

**std::swap** = Swap two objects (No copying)

```
string s3("Mira");
string s4("Visas");
swap(s3, s4);
```

s1 == “”

s2 == “Brianna”

s3 == “Visas”

s4 = “Mira”

Extra slides: what does std::move() really do?

# Simplest cmake projects

My first CMake project (**CMakeLists.txt**)

```
add_executable(hello hello.cpp)
```

My second CMake project

```
# This is a comment
cmake_minimum_required(VERSION 3.1)

project(hello)

set(CMAKE_CXX_STANDARD 14)

set(SRCS
#    somefile.h somefile.cpp
     hello.cpp
)

add_executable(${PROJECT_NAME} ${SRCS})
```

# How to build a CMake project ?

```
mkdir build
cd build
cmake ..
cmake --build .
```

Rebuild after you have edited some source files ...

```
cmake --build .
```

Using *generators* (Example: Windows, MinGW)

```
mkdir build
cd build
cmake -G "MinGW Makefiles" ..
cmake --build .
```

CMake does not call the C++ compiler directly.
Generators use low-level build systems (**make**, **nmake**, **ninja**, **...**)
and IDEs (Visual Studio, Code.Blocks, xcode)

# if statement : Example 2_3

```
if (a > 0)
        cout << ”a is positive” << endl;
else if (0 == a)  {
        cout << ”a is equal to zero” << endl;
} else
        cout << ”a is negative” << endl;
```

**{...}** is a *block*

```
{
        statement1;
        statement2;
        statement3;
}
```

# Switch statement

```
    switch (m) {
    case 1:
        cout << "January" << endl;
        break;
    case 2:
        cout << "February" << endl;
        break;
    case 3:
        cout << "March" << endl;
        break;
...
    default:
        cout << "Wrong Month !" << endl;
    }
```

**switch** works only for integer and **enum** types !

Don’t forget **break** !

# Loops

```
for (int i=0;  i<10; ++i)
    cout << i << endl;


int j=0;
while (j < 10)
    cout << j++ << endl;


int k=0;
// This runs at least once !
do
    cout << k++ << endl;
while (k < 10);


for (char c: string("Tower"))
    cout << c;
```

# Loops: bad style

```
int i;
cout << "Enter a number (0 = exit) :" << endl;
cin >> i;
while (i != 0) {
    cout << i << " * 2 = " << i*2 << endl;

    cout << "Enter a number (0 = exit) :" << endl;
    cin >> i;
}
```

# Loops: bad style

```
int i;
cout << "Enter a number (0 = exit) :" << endl;
cin >> i;
while (i != 0) {
    cout << i << " * 2 = " << i*2 << endl;

    cout << "Enter a number (0 = exit) :" << endl;
    cin >> i;
}
```

A piece of code is repeated 2 times = BAD

# Loops: good style

```
int i;
for (;;) {
    cout << "Enter a number (0 = exit) :" << endl;
    cin >> i;
    if (0 == i)
        break;
    cout << i << " * 2 = " << i*2 << endl;
}
```

"Infinite loop", which runs until

**break** exits the loop

# Library of the day : OpenCV : Example 2_4

```
#include <iostream>
#include <opencv2/opencv.hpp>

int main(){
     using namespace std;
     cv::VideoCapture cam(cv::CAP_ANY); // Open the camera
     if (!cam.isOpened())
          throw runtime_error("Cannot open camera");

     for (;;) {
          cv::Mat img;
          cam.read(img); // Read frame
          // Select the central roi and apply photo negative
          int w = img.cols, h = img.rows;
          cv::Mat m(img, cv::Rect(w/3, h/3, w/3, h/3));
          cv::bitwise_not(m, m);
          // Show frame
          cv::imshow("img", img);
          if (27 == cv::waitKey(1))
               break;
     }
     return 0;
}
```

# Library of the day : OpenCV : Example 2_4

```
cmake_minimum_required(VERSION 3.1)
project(e2_4)
set(CMAKE_CXX_STANDARD 14)
# Find package OpenCV
find_package(OpenCV REQUIRED)
# Add opencv include dirs
include_directories( ${OpenCV_INCLUDE_DIRS} )
message("OpenCV_INCLUDE_DIRS = ${OpenCV_INCLUDE_DIRS}")
message("OpenCV_LIBS = ${OpenCV_LIBS}")
set(SRCS
     main.cpp
)
add_executable(${PROJECT_NAME} ${SRCS})
target_link_libraries(${PROJECT_NAME} ${OpenCV_LIBS})
```

Sometimes you need to specify OpenCV directory:

**cmake -DOpenCV_DIR=/home/seymour/opencv/411/lib/cmake/opencv4 ..**

This must be the directory with *.cmake files !

Alternatively, you can specify CMake package search path:

**cmake -DCMAKE_PREFIX_PATH=/home/seymour/opencv/411 ..**

# Thank you for your attention !

# What does std::move() really do?

In reality **std::move()** does not move anything anywhere! For example:
**string a = "Some text !";**
**string b = move(a);**

**move(a)** casts **a** to *rvalue reference* **string &&**, this selects the move assignment **operator=** of **std::string** over the copy assignment version. This operator performs the actual move of the data. In general, this cast selects rvalue versions of constructor (move ctor), assignment, and various methods.

This logic is implemented in **vector**, **string** and all standard constructors. For you classes, you have to implement move logic by yourself if you implement any custom copy/move operations at all.

**std::swap(a, b)** is overloaded for different types. You can create an efficient version for your own class.