C++ Course 3: C++ Language Basics 2

2020 by Oleksiy Grechnyev

const Modifier

Variables with **const** modifier are *read-only*

```
const int a = 17;
a = 19;  // ERROR ! a is read-only!
const int b; // ERROR ! b is not initialized!
auto c = a; // c is int, auto ignores const
decltype(auto) d = a; // d is const int
```

const is part of the *data type*.

```
using CULL = const unsigned long long;   // const in a type declaration
CULL a = 17;  // a is const unsigned long long
array<const int, 4> cia{4, 3, 2, 1};   // const in a template argument
```

constexpr Modifier

constexpr is a *compile-time* constant

```
constexpr int SIZE = 18;
constexpr const char * NAME = "Lucifer";  // Type "const char *"
static_assert(SIZE == 18); // Compile-time check
int cArray[SIZE];  // C Array
array<int, SIZE> cppArray; // C++ Array class
```

constexpr variables must be of a *literal* type (int, char, const char * ...)

References

```
int a = 17;
int b = a;   // Regular variable initialized with value 17
int &c = a;  // Reference to a
const int &d = a;  // const reference to a
```

Now **c** is a reference (alias) to **a**. Let us re-assign **c**:

```
c = 18;   // Also changes a, d
// Now a == 18, b == 17, c == 18, d == 18
// d = 19; // Error !

auto x = d;  // x is int, auto ignores both ref and const
decltype(auto) y = c; // y is ref to a
decltype(auto) z = d; // y is const ref to a
```

References in range for loops

```
vector<string> vs{"Zero", "One", "Two", "Three", "Four"};

// Copy: inefficient !
for (string s : vs)  // or (auto s : vs)
    cout << s << " ";

// Use string & to change elements
for (string & s : vs)   // or (auto & s : vs)
    s += s;  // “Zero”  -> “ZeroZero”

// Use const string & for read-only access
for (const string & s : vs)   // or (const auto & s : vs)
    cout << s << " ";
```

Pointers

Pointer type variable keeps a *memory address* (Low level!)

int * = pointer to int

&a = Address of variable **a** (*address of* operator)

***p** = Value which is found at address **p** (*dereferencing* operator)

```
int a = 13;
int *p = &a;
cout << "a = " << a << ", p = " << p << ", *p = " << *p << endl;
```

Pointers

Pointer type variable keeps a *memory address* (Low level!)

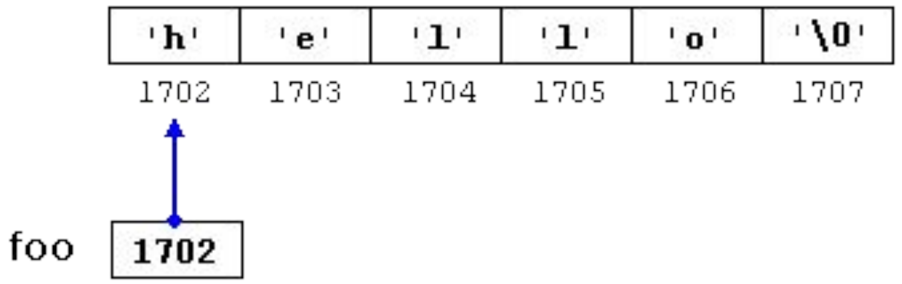
int * = pointer to int

&a = Address of variable **a** (*address of* operator)

***p** = Value which is found at address **p** (*dereferencing* operator)

```
int a = 13;
int *p = &a;
cout << "a = " << a << ", p = " << p << ", *p = " << *p << endl;
```

```
a = 13, p = 0x67fb7c, *p = 13
```



Pointers

Pointers can be changed (Unlike references)

int a = 13;
int *p = &a; // Now p points to a and *p = 13
int b = 17;
p = &b; // Now p points to b and *p = 17

p++; // increases **p** by **sizeof(int)** = 4 bytes
p += 11; // increases **p** by **11*sizeof(int)** = 44 bytes

p[n] == *(p + n) // Array-like indexing (C-arrays and pointers are VERY similar)

void * is a memory address without type

nullptr Null pointer (also you can see 0, NULL)
if (p) … // True if p != nullptr

Pointers and const

```
// p1  is a pointer to int, both p1 and *p1 can be changed
int * p1 = &a;
*p1 = 22 ;   // OK
p1 = &b;  // OK
// p2  is a pointer to const int, p2 can be changed, but not p2
const int * p2 = &a;
*p2 = 22 ;   // ERROR !
p2 = &b;  // OK
// p3  is a constant pointer to int, *p3 can be changed, but not p3
int * const p3 = &a;
*p3 = 22 ;   // OK
p3 = &b;  // ERROR
// p4  is a constant pointer to const int, neither p4 nor *p4 can be changed
const int * const p4 = &a;
*p4 = 22 ;  p4 = &b;  // ERROR
```

Pointers as array iterators

A C-style Array (NOT class) :

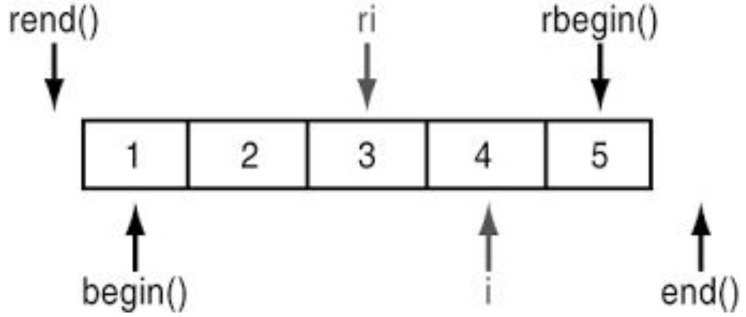
```
const string names[] = {"Karen", "Lucia", "Anastasia", "Margaret", "Alice"};
```

Print it with pointers :

```
const string * eit = names + 5;    // Position just after the last element
for (const string * it = names; it != eit; ++it)
     cout << *it << " ";
```

Or using C++ iterator style:

```
for (const auto *it = begin(names); it != end(names); ++it)
     cout << *it << " ";
```



Using new and delete

!!! Don't use this without a serious reason !!!

!!! A good C++ code DOES NOT use **new** + **delete** !!!

new : Creates object in the heap

string *pS = new string("Eowyn"); // Returns pointer to string

delete: Must be called to free memory, otherwise **MEMORY LEAK** !

delete pS; // Calls the destructor of ***pS**, then frees memory

Array version (don't use this, use **std::vector** instead)

int * myArray = new int[size]; // Reserves memory for **size** ints

…

delete [] myArray;

Extra slides:

C-style memory management (malloc, free)

C-style memory operations (memset, memcpy, memmove)

Functions

A function is defined as

<return_type> **myFunction**(*<par1>*, *<par2>*, ..) { <body> }

For example (**void** = no return):

```
void hello() {
     cout << "Hello There !" << endl;
}
```

```
int add(int a, int b){
     return a + b;
}
```

Using **auto**:

```
auto add(int a, int b) -> int {return a + b;}
```

```
auto add(int a, int b) {return a + b;}              // C++ 14
```

```
decltype(auto) add(int a, int b) {return a + b;}  // C++ 14
```

Reference parameters

Passing argument by reference :

Function can change argument variables !

```
void change(string & s){
     s = "Hello " + s;   // Modify s
}
```

```
string s = "Medea";
change(s); // Now we change s !
cout << "s = " << s << endl;
```

const & parameters are used for class types to avoid copying large objects

Not needed the for primitives (int, double ...)

```
string addStrings(const string & s1, const string & s2){
     return s1 + s2;
}
```

Returns **string** , not **string &** or **const string &** !!!

Return by value or reference

Functions almost always return by value, e.g.

```
string somefun( … ){
     string s;
     …
     return s;    // Return Value Optimization : No copy !
}
```

Returning reference OK for *getters*, otherwise ERROR !!!

??? Reference to what exactly ???

```
const string & bad1(){
     string s = "HAHAHA !";
     return s;    // ERROR !!! Reference to a local variable !
}
```

```
const string & bad2(){
     return string("HAHAHA !");  //  ERROR !!! Ref to a temporary !
}
```

Factory function should return **unique_ptr** (best), or the object by value

Function overloading

Overloading: Same name, different parameter list

```
void print(int a) { … }
void print(int a, double b) { … }
void print(const string &) { … }
```

```
print(17);  // print(int)
print(22.8);  // print(int), double is converted to int
print(4, 5.7);  // print(int, double)
print(13, 23);  // print(int, double), int is converted to double
print(string("Nel Zelpher"));  // print(string)
print("Maria Traydor");  // print(string), const char * is converted to string
```

Automatic type conversion

Compiler error if ambiguity

```
void print(double a, int b) { … }
void print(int a, double b) { … }
```

```
print(1, 2);  // ERROR !! Ambiguity !!!
```

Extra Slide: Parameter Default Values

Function templates

Function template to add two objects (anything which can be added by "+"):

Template Instantiation : **T** is replaced by a concrete type (e.g. **int** or **string**)

```
template <typename T>  // OR "template <class T>"
T tmplAdd(T a, T b) {
    return a + b;
}
```

In many cases, type **T** can be inferred automatically (*automatic instantiation*):

tmplAdd(3, 4) // T == int

tmplAdd(3.2, 4.2) // T == double

tmplAdd(string("Hello "), string("World !")) // Note: does not work with string literals!

tmplAdd(3, 4.2) // Error ! T == int ? Or T == double ? Cannot infer!

You can always specify **T** explicitly (*explicit instantiation*):

tmplAdd<double>(3, 4.2) // T == double

tmplAdd<string>("Hello ", "World !") // T == string (Literals converted to string)

Function templates

Function template to add two objects (anything which can be added by "+"):

Template Instantiation : **T** is replaced by a concrete type (e.g. **int** or **string**)

```
template <typename T>  // OR "template <class T>"
T tmplAdd(T a, T b) {
    return a + b;
}
```

- Works with both primitive types and classes (unlike Java)
- Template Instantiation is done at *compile time*: Separate machine code is created for every **T**! No single machine code for every **T** is created! (also unlike Java)
- *Duck typing*: "If it walks like a duck and it quacks like a duck, ..."
- Template defined in a **.cpp** file can be used in this **.cpp** file only!
- Define "library" templates in **.h** files! Then everybody can use them!
- *Template specialization* is different from *Template Instantiation*!

Do it yourself: Look at the second template example in e3_2

Namespaces

In a large project ...

a.cpp : **void fatalError(const string & s);**
b.cpp : **void fatalError(const string & s);**
OpenCV library : **void fatalError(const string & s);**
ffmpeg library : **void fatalError(const char * s);**

Name conflict !!!
Solution: namespaces. Namespace is a named scope.

```
namespace A{
	void fatalError(const string & s);
}
…
A::fatalError("Camera device not found !");
```

Namespace tree

```
namespace A{
    namespace A{
        namespace X{
            namespace A{
               ...
            }
        }
    }
}
```

:: is the Root scope

```
A::A::X::A::myFuction(17);   // Starts from the current scope
::myFuction(17);             // :: is the root scope
::A::A::X::A::myFuction(17);   // Starts from the root scope
```

using statement

using std::vector; // Use only class vector from namespace std
using namespace std; // Use the entire namespace std
..
vector <int> vi; // Without std:: prefix !

using can lead to a name conflict
Solve it with **::A::A::myFun();** (Starting from the root !)

using rules:

1. Never in header (*.h) files ! Important! Otherwise namespace leaks !
2. Cannot be used inside class definition (Syntax error).
3. Best: inside function or method body.
4. OK but not best: In CPP file.

Note: **main()** must NEVER be in any namespace !!!

Anonymous namespaces and the static keyword

fun() is only visible in this *.cpp* file, not in other *.cpp* files:

```
namespace{
	void fun();
}
```

Keyword **static** :

1. In *.cpp*: like anonymous namespace
   static void fun();   // Visible in this cpp file only (also **inline** is similar)
2. Inside function/method: hidden global (!!! SUPER - EVIL !!! !!! NEVER !!!)
   static int myVar; // Global var, not in stack !
3. Inside class: static method/field (belongs to class, not instance)
   static MyClass & getInstance();

Headers

header.h is a piece of code included with **#include "header.h"**

#include is a preprocessor directive, no semicolon (;) !

#include <...> Search in system/project include directories

#include "..." Search in current directory, then system/project include directories

C++ system headers:

#include <iostream> **// C++ standard library headers**

#include <cstdio> **// C standard library headers in namespace std**

#include <stdio.h> **// C standard library headers**

Other headers:

#include <boost/asio.hpp> **// Library headers**

#include <libavcodec/avcodec.h>

#include "myfile.h" **// My header**

Include guard

In myfile2.h :

```
#include "myfile.h"
```

In main.cpp :

```
#include "myfile2.h"
#include "myfile.h"   // Included twice !!! BAD !!!
```

We want to include **myfile.h** only once

#ifndef _MYFILE_H_

#define _MYFILE_H_

<All code goes here>

#endif

Or, not in the official C++ standard, but works for all modern compilers

#pragma once

Learn it yourself: C++ preprocessor (#if, #ifdef, #define ...)

Function/class declarations and definitions

Function declaration (in **myfile.h**) :

```
int add(int a, int b);    // We need this to call add
```

Function definition (in **myfile.cpp**):

```
int add(int a, int b) {
        return a+b;
}
```

Class definition (in **MyClass.h**):

```
struct MyClass{    // Class definition
        void method();        // Method declaration
};
```

Class method definition (in **MyClass.cpp**):

```
MyClass::method() { … }
```

Can we define a function in a header?

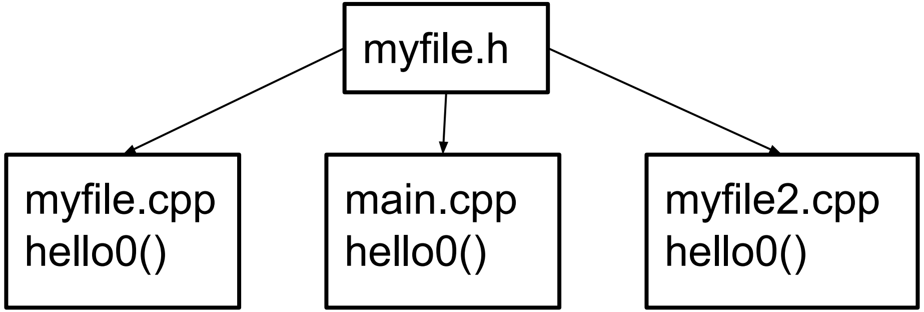
in myfile.h:

```
void hello0() { std::cout << "hello0()" << std::endl;}          // Error !
```

Can we define a function in a header?

in myfile.h:

```
void hello0() { std::cout << "hello0()" << std::endl;}        // Error !
```



But if we REALLY REALLY want to define a function in myfile.h ???

```
inline void hello1() { std::cout << "hello1()" << std::endl;}
static void hello2() { std::cout << "hello2()" << std::endl;}
namespace {
        void hello3() { std::cout << "hello3()" << std::endl;}
}
```

Note: templates are always in .h, no troubles with them!

Global variables : EVIL !

In **myfile.h** :

```
extern int x;   // Declaration
```

In **myfile.cpp** :

```
int x = 17;         // Definition. Initialize here if needed
```

Global Constant (not evil), in **myfile.h** :

```
constexpr int GLOBAL_CONST = 666;
```

Global variables : EVIL !

In **myfile.h** :

```
extern int x;   // Declaration
```

In **myfile.cpp** :

```
int x = 17;         // Definition. Initialize here if needed
```

Global Constant (not evil), in **myfile.h** :

```
constexpr int GLOBAL_CONST = 666;
```

Global variable means:

- Cannot run 2 or more copies of the code
- This includes multithread
- This includes unit tests

Real cases: Global variables, memory leak, seg faults ...

Case 1:

- CV engineer: My C++ algorithm code "Works"!
- Request: "Make it work in different threads ..." WTF ??? It's absurd!
- It actually had global variables
- It also had bad memory leak
- Took a lot of effort to make the code really work

Case 2:

- CV engineer: My C++ algorithm code "Works"!
- Warning: int(..) {} function does not return a value
- Undefined! Forbidden operation in C++
- In Debug : The code "works" ...
- In Release: Segmentation fault!

How to write good algorithm code in C++?

"The code works, here is the demo" is not good enough!
The real (deployable) algorithm code in C++ must fulfill the following requirements:

- No memory leaks
- No crashes/SEG faults
- No global variables
- All code in namespaces
- No pointers and **new/delete/malloc/free** unless absolutely necessary
- No platform-dependent stuff (e.g. **#include<windows.h>**)
- The algorithm and GUI/demos are cleanly separated
- **README.md**, Doxygen comments
- Ideally: Handle all pathological cases: All black/all white/empty images, negative/zero parameters: Exceptions or error codes: Not random OpenCV exceptions!

Otherwise it is not a deployable C++ code, as it cannot be used in other people's projects without rewriting!

Tool (not library !) of the day: Doxygen

Document your code with docstrings: special comments:

```
/** @brief Multiply a and b
*
* This function is super-fun !
* @param[in]  a   Input parameter a
* @param[in]  b   Input parameter b
* @return         The product of a and b */
double mul(double a, double b);
```

Or, short comment:

```
/// Divide a by b
double div(double a, double b);
```

Then run **doxygen** to generate HTML documentation (See e.g. OpenCV docs).
Real C++ programmers *always* do this.

Thank you for your attention !

C-style memory management

Allocate memory in the heap, C-style

void * malloc(size_t n); // Allocate n bytes

```
int * iBuffer = (int *) malloc(25 * sizeof(int));   // 25 elements
```

Allocate memory in the heap and set it to 0, C-style

void * calloc(size_t n1, size_t n2); // Allocate **n1*n2** bytes

```
char * cBuffer = (char *) calloc(45, sizeof(char));   // 45 elements
```

Set **count** bytes to **ch**

void* memset(void* dest, int ch, std::size_t count); // Writes to dest

```
memset(cBuffer, (int)'Z', 45);   // Fill the buffer with 'Z' (45 bytes)
```

Free memory block in the heap

```
free(iBuffer);    free(cBuffer);
```

Using memcpy and memmove

Fast memory copy (**num** bytes)

void * memcpy (void * destination, const void * source, size_t num);

```
char s[100];    // A buffer
memcpy(s, "Jessica", 8);  // Copy 8 bytes
```

Memory copy with overlap

void * memmove (void * destination, const void * source, size_t num);

```
memmove(s + 4, s, 8);  // Copy 8 bytes from s to s+4
cout << s << endl;
```

```
JessJessica
```

Parameter default values

```
int add(int a = 10, int b = 5){
     return a + b;
}
```

```
cout << add(1, 2);  // 3
cout << add(1);     // 6
cout << add();      // 15
```

Such parameters must be **to the right** in the parameters list !

```
int add(int a, int b = 5){return a + b;}     // OK
int add(int a = 10, int b){return a + b;}    // ERROR !!!
```

Slide