C++ Notes

More info at:

cplusplus.com
cppreference.com
isocpp.org
learncpp.com

by **NeptuneLuke**

Index

Escape characters

\n	new line
\b	backspace
\t	horizontal tab
\v	vertical tab
****	backslash
\0	null char (string character terminator)
\?	?
\“	“
\‘	‘

Keywords

auto	the compiler will deduce the variable type automatically
void	void type / typeless
sizeof(type)	return the size in bytes of the type
sizeof(var)	return the size in bytes of the variable
break	interrupts the scope in which is contained and restart the computation from the first instruction right after the while/do while/for/switch structure
continue	interrupts the current iteration of the scope in which is contained and goes back to the condition part of the while/do while/for/switch structure
new	(actually and operator) allocates in the heap a memory area big as the variable type declared and the address of that area is stored in the pointer
delete	(actually and operator) deallocates the variable from the heap, eliminates the "object" pointed by the pointer, not the pointer itself, and after the delete, the pointer contains a garbage value, so it is good practice to explicitly set the pointer to null do not use delete two times if the pointer is not set to null after the first delete
using	the most typical way to introduce visibility of components is by means of using declarations: **using** namespace std;

<table>
<tr><td></td><td>The above declaration allows all elements in the std namespace to be accessed in an unqualified manner (without the std:: prefix)

Note that explicit qualification is the only way to guarantee that name collisions never happens

it can also be used as an aliasing tool, similar to typedef:

using new_name = current_name

using vInt = std::vector<int>;</td></tr>
<tr><td>extern</td><td>used to define a “shared” variable between files only once, for example a global variable

header.hpp
<pre>extern int global_x;   // declared
void print_global_x();</pre>file_1.cpp
<pre>#include "header.hpp"

int global_x;   // defined

int main () {

   global_x = 5;   // initialized
   print_global_x();
}</pre>file_2.cpp
<pre>#include <iostream>
#include "header.hpp"

void print_global_x() {

   cout << global_x;
}</pre></td></tr>
</table>

	also used to make a variable declared once and only once (not even temporarily in a for loop, for example)
typedef	it is used to aliasing fundamental and custom data types, or to rename pointers to a more meaningful name **typedef current_name new_name** typedef std::vector<int> vInt; typedef unsigned long long int ulli; typedef int* iPtr; iPtr p1, typedef is still in C++ for backward compatibility only, and should be replace by **using**

Operators

AND **&&**	it works with short circuit evaluation, meaning that evaluates the second argument of the logic expression only if necessary
OR **||**	it works with short circuit evaluation, meaning that evaluates the second argument of the logic expression only if necessary
ternary operators **? :**	result = (condition) ? option1 : option2 ;

	the equivalent version is the following:
	
	if(condition)
	  result = option1
	else
	  result = option2

increment/subtract
++ --
	x++;
	x--;
	increment/subtract the value of x after processing the current statement
	
	++x;
	--x;
	increment/subtract the value of x before processing the current statement
	
	int x += ++i;
	increment i, then add i to x
	
	int x += i++;
	add i to x, then increment i

bitwise left shift **<<**	takes two numbers, left shifts the bits of the first, and the second decides the number of places to shift x << y
bitwise right shift **>>**	takes two numbers, right shifts the bits of the first, and the second decides the number of places to shift x >> y
bitwise AND **&**	takes two numbers and does AND on every bit, the result is true only if both bits are 1

bitwise OR `|`	takes two numbers and does OR on every bit, the result is true only if any of the two bits is 1
bitwise XOR `^`	takes two numbers and does XOR on every bit, the result is true if the two bits are different
bitwise NOT `~`	takes one numbers and inverts all of its bits

Variables

All variables in C++ are basically a numerical value. Variables can be modified by some keywords such as **`short`**, **`signed`**, **`unsigned`**, **`long`**, **`long long`**.

Not all systems use the same size for variables for the same programming language. To avoid potential errors, we can use the following type "variant" **`typexx_t`**, where **`type`** is the type of the variable, **`xx`** is the number of bits used to store it, **`_t`** is a standard used suffix in C++ that stands for type.

Here are some of the fixed width types:
`int8_t`
`int16_t`
`char8_t`
`char16_t`
`char32_t`

Bool

`bool` values are 1 byte size. Can only be `0` or 1.

```
bool b = 0;   // false
bool b = 1;   // true
```

Char

The **char** value is the smallest type of value. All sizes of C++ objects are expressed in terms of multiples of **char**, so by definition the size of **char** is 1.

char / unsigned char values are 1 byte size.
Can be any value [0 , 255];

signed char can be any value [-128 , 127];

To have chars with more size, we can use:
char16_t is the UTF-16 equivalent for **char**
char32_t is the UTF-16 equivalent for **char**

```
char c = 'a';   // copy initialization
char c {'a'};   // list initialization
char c = 97;    // 97 is 'a' in ASCII

char c = ' \' ';    // declare char '
char c = ' \" ';    // declare char "
char c = ' \0 ';    // declare char null character

char c = 97;    // 97 is 'a' in ASCII
c++;            // 98 is 'b' in ASCII
```

Int

int / signed int values are 4 bytes (32 bits) size.
Can be any value [-2,147,483,648 , 2,147,483,647].

unsigned int values are 4 bytes (32 bits) size.
Can be any value [0 , 4,294,967,295].

short / short int values are 2 bytes (16 bits) size.
Can be any value [-32,768 , 32,767].

unsigned short values are 2 bytes (16 bits) size.
Can be any value [0 , 65,535].

`long / long int` values are a minimum of 4 bytes (32 bits) , maximum 8 bytes (64 bits).
Can be any value [-2,147,483,647 ， 2,147,483,647].

`long long / long long int` values are a minimum of 8 bytes (64 bits).
Can be any value [-9,223,372,036,854,775,808 ， 9,223,372,036,854,775,807].

```
int x = 0;   // copy initialization

int x {0};   // list initialization
int x {numerical expression};

int x (5);   // direct initialization
int x (numerical expression);


// Declaring various types of numbers
int x;                // initialized with a random number
int x = 017;          // octal number
int x = 0x0f;         // hexadecimal number
int x = 0b00001111;   // binary number


// Declaring numbers like chars
char c = 'a';   // a (97 ASCII)
int x = 'a';    // 97
int x = c;      // 97
```

Float and Double

Float stands for number with floating point precision. Double stands for number with double floating point precision.

`float` values are 4 bytes (32 bits) size, with 7 digits of precision.
`double` values are 8 bytes (64bits) size, with 15 digits of precision.
`long double` values are 12 bytes size, with 15 digits of precision.

```
// float numbers must end with f
float f = 3.14159f;
float f = 2.0f;

// double can be written in scientific notation
double d = 3.14159;
double d = 5.27e8;
double d = 5.27e-8;

long double d = 0.3333333333333333333L;
```

String and Char arrays

Char arrays are the C way to get strings. In C++ there is the string type.

Char arrays

```
char s [] = "hello";
char s [] = { 'h', 'e', 'l', 'l', 'o' };

// this is how the compiler sees the char array
// the compiler that adds the '\0' at the end to know when to stop
char s [] = { 'h', 'e', 'l', 'l', 'o', '\0' };
```

Strings

```
string s = "hello";                     // copy initialization
string s { "hello" };                   // value initialization
string s = ("hello");                   // direct initialization
string s { 'h', 'e', 'l', 'l', 'o' };   // list initialization
```

Casting

The casting consists in changing the type of a variable. There are three main ways to cast variables.

Implicit casting

Is when the operands of an operation are of different types. Before executing the calculation, the compiler temporarily converts them into a common type with an automatic/implicit casting. The implicit casting cannot always be done and must respect the following rule: the value of the operand with lesser priority is cast in the type of the operand with greater priority, based on the following hierarchy.

```
low                                high
char -> int -> long -> float -> double

int i;
float f;
double d;

f * i          // i is cast into a float, the expression returns a float

d * (f / i)    // i is cast into a float, (f / i) returns a float which
               // is cast into a double, the expression returns a double

f * (d / i)    // i is cast into a double, (d * i) returns a double, f is
               // cast into a double, the expression returns a double

i = f * d      // f is cast into a double, the expression returns a
               // double which is then cast into an int, causing
               // information loss
```

Explicit casting

Is explicitly stated by the programmer. It does not follow the low to high hierarchy, but can be used both ways.

```
(type_to_be_converted_to) expression

double d;
float f;
int i = (int) f / (int) d;   // f and d are cast into an int, causing an
                             // information loss, the expression returns
                             // an int
```

```
float f, r;
double d = (double) (f = r * r * 3.14159)   // the expression is cast
                                            // into a double
```

Method casting

Is explicitly stated by the programmer through some methods, provided by the C++ STL. One of the most used is **`static_cast<type>(value)`**.

```
int i = static_cast<int>('a');   // returns 97, which is the value of
                                 // 'a' in ASCII

char c = static_cast<char>(97);  // returns 'a'

float f = static_cast<float>(115);     // returns 115.0f
int i = static_cast<int>(3.14159f);    // returns 3
```

Constants

Constants are a special type of variable, that cannot be changed after the first initialization. Constants must be declared before the main function. They can be declared in two ways:

C style

Declared by the **`#define`** directive, the constant is not stored in memory, but its value is replaced in the program by the compiler.

```
#define PI 3.14159;
```

C++ style

Declared by the **`const`** keyword, the constant is stored in memory, just like any other variable, so it is typed and scoped.

```
const float PI = 3.14159;
```

In C++ there is no actual need for the **`#define`** to be preferred over **`const`**.

Enumerations

Enumerations are a useful way to group some values. Every value of the enumeration has a code, starting from zero. The code of each value can be explicitly stated or not, and in that case, the code is the previous code plus one.

```
enum enum_name { ..., ..., ... };


enum Days { MON, TUE, WED, THU, FRI, SAT, SUN };

for (i = MON; i <= SUN; i++)
        cout << i << " ";
// output: 0 1 2 3 4 5 6


enum EscapeChars { BELL = '\a', BACKSPACE = '\b', TAB = '\t',
                   NEWLINE = '\n', VTAB = '\v', RETURN = '\r' };

enum Foo { a,           // a = 0
           b = 1,       // b = 1
           c = 10,      // c = 10
           d,           // d = 11
           e = 1,       // e = 1
           f,           // f = 2
           g = f + c    // g = 12
         };


enum Color { red, green = 20, blue };
Color col = red;
int i = blue;    // i == 21
i = col;         // i = 0

enum class Color { red, green = 20, blue };
Color col = Color::red;
int i = Color::blue;    // i == 21
i = col;                // i = 0
```

Stack and Heap

Every program uses different memory areas:

- text/code segment: stores the instructions of the program
- data segment: stores global variables
- stack
- heap

The text segment, data segment and the stack are all under the control of the compiler, and are statically managed memory. The developer can't change how they work.
The heap is where the dynamic memory is handled and the programmer can freely use it.

Stack

The stack is the place where the static memory is managed. It is used if the developer knows exactly how much data the program will need and it is not too big.
To wrap it up:

- it has finite memory, in most systems is 4 MB
- it is stored in the RAM
- it contains local variables, function calls, and return statements
- the compiler controls the memory lifetime through the scope mechanism
- it is faster to allocate memory to
- if too much data is stored in the stack, then a stack overflow occurs

Heap

The heap is the place where the dynamic memory is managed. It is used if the developer doesn't know exactly how much data the program will need at runtime or if they need to allocate a lot of data.
To wrap it up:

- has finite memory, but a lot more than the stack (as big as the currently free RAM)
- is stored in the RAM

- the developer must manually store and then free the memory that variables hold through the **`new`** and **`delete`** operators
- is slower to allocate memory to
- can cause memory leaks

Pointers

A pointer is a special type of variable that stores memory addresses. A pointer can only store the type for which was declared and every pointer has the same size (provided the same type).
The operators to be familiar with when working with pointers are **`&`** , **`*`**.
`&` returns the address of the variable
`*` returns the value the pointer is pointing to

```
int i = 10;     // stores the number 10 in some memory cell
int* p = &i;    // stores the address of the variable i (so the address
                // where the number 10 is stored) in the variable p
                // the variable p is stored in another memory address

cout << i;       // print 10
cout << &i;      // print 0xBFB23AAC (memory address of i)
cout << p;       // print 0xBFB23AAC (the content of p which is the
                 // memory address of i)
cout << &p;      // print 0xBFB23AA8 (memory address of p)
cout << *p;      // print the content to which p is pointing to (10)
```

Null pointers

A special value for pointers is **`NULL`**. A **`NULL`** pointer is a pointer that is pointing to the address `0x00000000`. The pointer is still stored in memory, so it has a memory address. Another value identical to **`NULL`** is **`0`**.
In C++ (since C++ 11), the **`NULL`** keyword should not be used anymore and instead should be replaced by **`nullptr`**.
The keywords **`nullptr`**, **`NULL`** and **`0`** all mean the same value.

```
int i = 10;          // stores the number 10 in some memory cell
int* p = nullptr;    // stores the address nullptr
                     // the variable p is stored in another memory
                     // address
```

```
cout << i;        // print 10
cout << &i;       // print 0xBFB23AAC (memory address of i)
cout << p;        // print 0 (the content of p which nullptr = 0)
cout << &p;       // print 0xBFB23AA8 (memory address of p)
cout << *p;       // segmentation fault
```

Things to avoid while working with pointers

- writing into an uninitialized pointer through dereference
- writing into a null pointer
- calling delete twice on a pointer

```
int* p;
*p = 55;            // writing into a random memory cell

int* p = nullptr;
*p = 55;            // writing into a pointer pointing nowhere

Foo* p = new Foo();  // allocates a pointer in the heap
delete p;            // deallocates the pointer from the heap
delete p;            // deallocates a random memory part
                     // anything can happen! avoid at all cost
```

Why should we set pointers to null?

Setting a pointer to null is like initializing an int to `0`.
It ensures it doesn't point to a random memory location. This helps avoid undefined behavior if the pointer is accidentally used before being assigned a valid address.
We should also set pointers to null after using delete on them, because a dangling pointer leads to undefined behavior if accessed later.

Dynamically allocate pointers in the heap

To dynamically allocate a pointer in the heap, you should create a new pointer, then delete, and then set it to null.

```
int* p = new int {10}   // allocates pointer in the heap
int* p = new int;       // allocates pointer in the heap
delete p;               // deallocates pointer from the heap
p = nullptr;            // set the pointer to null


Foo* p = new Foo();
delete p;
p = nullptr;


// If we are using a pointer to an array
int* p = new int[10];
delete[] p;
p = nullptr;
```

Functions

Functions are a useful way to organize the code. Functions take arguments that are only scoped in the function itself.

Functions can be defined before the main function.

```
int sum(int x, int y) {   // definition, body of the function
   return x + y;
}

int main() {

   sum(3,2);
}
```

Functions can also be declared before the main function and defined after the main function. The definition of the function is called prototype and is the signature of the function, that means the return type, the name of the function, the type and name of the arguments.

```
int sum(int x, int y);    // declaration, signature of the function

int main() {

   sum(3,2);
}

int sum(int x, int y) {   // definition, body of the function
   return x + y;
}
```

If a function doesn't require arguments, then the signature can be blank or the **`void`** keyword can be used.

```
void print() {
   cout << "hello";
}

void print(void ) {
   cout << "hello";
}
```

Passing arguments by value

It's the default passing mechanism. It copies the actual value of the argument into the parameter of the function, so that the changes made to the parameter inside the function have no effect on the argument passed in the call of the function. This is a relatively inexpensive operation for fundamental types such as int, but if the parameter is of a large compound type, it may result in a certain overhead.

Passing arguments by reference

Using the **&** operator, it copies the address of the argument into the parameter of the function, so that the changes made to the parameter inside the function affect the argument passed in the call of the function.

If we want to pass a variable by reference because copying the value could result in a significant overhead, but we want to be sure the variable is not changed, we can use the keyword **`const`**.

```
int sum(int &x, int &y) {   // x and y are changed!
   x++;
   y++;
   return x + y;
}

int main() {

   sum(3,2);
   cout << x << " " << y;   // output: 7
}


// Using const
int sum(const int &x, const int &y) {   // x and y are not changed!
   x++;
   y++;
   return x + y;
}

int main() {

   sum(3,2);
   cout << x << " " << y;   // output: 5
}
```

Inline functions

Calling a function generally causes a certain overhead (stacking arguments, jumps, etc...), and thus for very short functions, it may be more efficient to simply insert the code of the function where it is called, instead of performing the process of calling a function.
Preceding a function declaration with the **`inline`** specifier informs the compiler that inline expansion is preferred over the usual function call mechanism. This does not change at all the behavior of a function, but is merely used to suggest to the compiler that the code generated by the function body should be inserted at each point the function is called, instead of being invoked with a regular function call, although the compiler is free to not inline it, and optimize otherwise.

Note that most compilers already optimize code to generate inline functions when they see an opportunity to improve efficiency, even if not explicitly marked with the **`inline`** specifier.

```
inline void print() {
   cout << "hello";
}
```

Array and Vector

Array

Arrays are a set of values of the same type stored in a contiguous space in memory. The name of an array is also an address, to be more specific the address of the first element of the array.

```
type array_name [array_size];
type array_name [array_size] = { ... , ..., ... };
type array_name [] = { ... , ..., ... };


array_name = &array_name[0]       // first address is the array address
&array_name[i] = array_name + i   // i-th element address is offset from
                                  // the first element


// How to access an array
for(int i=0; i < size(array_name); ++i)
      array_name[i]

for(auto elem : array_name)
      elem                  // elem is array_name[elem]
```

Arrays can be passed to functions in two ways, but always need the size of the array to be specified. Arrays are always passed by reference, because the array is basically an address. If we do not need to modify the array passed in a function we can use the following statement using **`const`**.

```
// or i can pass just the array and calculate the size of the array
// inside the function with the statement
// int array_size = sizeof(array) / sizeof(int);
```

```
void my_func(int array_name[], int array_size) {
    ...
}

int main() {

    int array [] = { 1, 2, 3, 4, 5 };
    int array_size = sizeof(array) / sizeof(int);
    my_func(array, array_size);
}


// Do not modify the array using const
void my_func(const int array_name[], int array_size) {
    ...
}
```

Vector

Vectors are special types of arrays, made possible through the C++ STL with **`#include <vector>`**. Vectors do not need their size to be passed to the function and by default are passed by value.

```
#include <vector>

vector<type> vector_name;
vector<type> vector_name(vector_size);
vector<type> vector_name(vector_size, init_value);
```

Structs

Structs are a way to define custom types. A struct is just a collection of variables, which can be every fundamental data type, classes, and other structs.
Structs should be defined before the main() and functions, so that they would be accessible by the functions. Structs by default are passed by value.

```
struct Test1 {
   int x;
   char c;
};

// we can assign a name to the struct after the definition
struct {
   int x;
   char c;
}Test2;


int main() {

   Test1 t1;
   t1.x = 10;   // to access the fields of the struct we use the .
                // operator

   Test1 t2 = { 10, 'a' };   // another way to initialize

   Test2.x = 10;

   // declare an array of structs
   Test1 array_name [array_size];
   array_name[0].x = 10;
}
```

How much memory does a struct occupy?

Someone could say the sum of the size of every field. Theoretically it's true, but usually compilers for efficiency purposes "align" data, for example doing so that addresses of every field are multiple of 4 bytes.

```
struct Test1 {
   int x;
   char c;
};

// this struct occupies 8 bytes, and not 5
// (4 bytes for int + 1 byte for char)
```

Structs are usually used to implement dynamic data structures (list, stack, queue, tree) with the use of pointers.

```
struct node {
   int x;
   node* next;
};

int main() {

   node* head = nullptr;
}
```