C++ Introduction

1. Table of Content

2. Languages

Python / JS	C# / Java	C++	C / Assembly	Binary
High Level	High Level	Low Level	Low Level	Native
Interpreted	Compiled	Compiled	Compiled	Executable
Managed Memory / GC	Managed Memory / GC	Unmanaged Memory	Unmanaged Memory	Unmanaged Memory
Secured Memory Access	Secured Memory Access	Unsafe Memory Access	Unsafe Memory Access	Unsafe Memory Access
No Pointers	Unsafe Pointer Only	Pointers	Pointers	Pointers
POO	POO	POO	No Objects	No Objects

3. Memory

Understanding Memory Management

Detailed Pointer Tutorial

3.1. Binary

Decimal	Binary	Formula
0	0000	0

Decimal	Binary	Formula
1	0001	1
2	0010	2
3	0011	2 + 1
4	0100	4
5	0101	4 + 1
6	0110	4 + 2
7	0111	4 + 2 + 1
8	1000	8
15	1111	8 + 4 + 2 + 1

3.2. System Memory

Address	Data
0	0000 0000
1	0000 0000
2	0000 0000
...	0000 0000
$2^x - 1$	0000 0000

In 32-bit system, each address is a 32 bits (4 bytes) integer. Which means that the system is able to address 2^{32} bytes (4 GB) of memory. In theory a 64-bit system is capable of addressing 2^{64} bytes (18.4 Million TB) of memory. In practice 64-bit CPU only use 42 to 52 bits addressing with a standard of 48 bits (256 TB).

3.3. Reference and Dereference

Pointers and Values can be obtain and modify using `reference(&)` and `dereference(*)` operators.

```
int value = 5;
int *ptr = &value; // Reference operator : ptr == address of value
int value2 = *ptr; // Dereference operator : value2 == value pointed by ptr
```

3.4. Simple Pointer

Pointers can be obtain and modify using `reference` `&` operator.

```
int value = 5;
int *ptr = &value;
```

Memory can be allocated using `new` operator.

```
int *p = new int;
```

Address	Data
p	0000 0000

Values can be read or modified using `dereference` `*` operator.

```
*p = 5;
int value = *p; // value == 5
```

Address	Data
p	0000 0101

Memory can be free/deallocated using `delete` operator.

```
delete p;
```

Address	Data
p	???? ????

3.5. Array

```
int a[3] = { 7, 53, 97 };
int *p = a;
```

Address	Data
p	0000 0111
p + 1	0011 0101
p + 2	0110 0001

3.6. Reference

Reference is a pointer that is dereferenced automatically on use. It has the same behaviour as regular pointers but it uses non-pointer syntax.

```
int x = 2;
int& rx = x;
```

4. Headers And Preprocessor

C++ projects are divided in two type of files. Code files (.cpp) contains regular C++ and hold most of the implementation. The header file (.h or .hpp) are used to propagate declarations. Header files are used to centralize declarations in one place, allowing to import all of them when needed.

```
#pragma once                        // Ensure this file is imported only once

#include <myfile>                   // Import standard file content
#include "myfile.h"                 // Import file content

#define MACRO value                 // Replace MACRO with value
#define MACRO(param) (expression)   // Replace MACRO with expression with parameters
#undef MACRO                        // Remove MACRO definition

#if                                 // Conditional compilation
#else                               // Conditional compilation
#endif                              // Required after #if, #ifdef

#ifdef MACRO                        // if MACRO is defined
#ifndef MACRO                       // if MACRO is not defined
```

5. Syntax

5.1. Literals

Literals represent fixed values.

```
42              // Decimal Integer (base 10)
052      // Octal Interger (base 8)
0x2A     // Hexadecimal Integer (base 16)
0b101010 // Binary Integer (base 2)
3.14     // Floating Point
'A'      // Character
"ABC"    // String
true     // Boolean
false    // Boolean
```

5.2. Comments

```
// Single Line Comment

int a = 0; // Single Line Comment

/*
Comment
Block
*/
```

5.3. Code Documentation

Comments can also be used to document your code. Multiple documentation parser can be used with different syntaxes. For example Doxygen or JavaDoc.

Example with Doxygen

```
/**
 * \brief Adds two numbers.
 *
 * This function takes two numbers, adds them, and then returns the result.
 *
 * \param x The first number to add.
 * \param y The second number to add.
 * \return The sum of the two numbers.
 */
int add(int x, int y) {
    return x + y;
}
```

5.4. Instructions

5.4.1. End of Statement and

In C++ `;` is required to specify the end of statement. Most of the time the end of statement is also the end of line.

5.4.2. Line Continuation

In C++ `\` can be used at the end of the line to ignore the newline (\n) and continue the current line on the next one.

5.4.3. Scope

In C++ the context in which a name is visible is called its scope. Scope are delimited by `{` and `}`. Scope access is resolved using `::` operator. Most common scopes include function, namespace, classes, enum, struct, loops, if/else etc.

```
{       // Open Scope
}   // Close Scope
::  // Scope resolution operator
```

Example

```
class Base
{ // open Class Base scope 'S1'
    void f();
} // close scope 'S1'

void Base::f()   // '::' reopen scope 'S1'
{ // Open function scope 'S2'
        int x = 0; // x belong to 'S2' scope
} // Close scopes 'S1' and 'S2'
// x is out of scope and is deleted
```

5.4.4. Const

In C++ `const` keyword is used to specify that something cannot be modified. It can be used in a lot of different places like, variable declaration, function parameters, classes body etc.

```
const int x = 0; // x is const and can't be modified
```

5.5. Types

5.5.1. Types

Variables are used to store data. Variable can be of different `types`.

```
void    // no type or unspecified type
bool    // true or false
int             // integer
char    // single character
float   // floating point number (with decimals)
double  // float with double precision (number of decimal digits)
```

5.5.2. Declaration

There are many syntaxes to declare variables. Commonly declaration operation is reference like `type variableName;` or `type variableName = value;`.

```
int x;                          // Declarations
int x = 255;                    // Declarations + Assignment

int a, b, c;                    // Multiple Declarations
int a = 0, b = 1, c = 2;        // Multiple Declarations + Assignment

int a[10];                      // Declare Array of Size 10
int a[] = {0, 1, 2};            // Declare + Assign Array of Size 3
int a[2][2] = {{1, 2}, {4, 5}}; // Declare + Assign Array of Array

int* p;                         // Declare Pointer of int
int& r = x;                     // Declare + Assign Reference

const int c = 3;                // Constants have to be initialized
const int* p=a;                 // Content of p is constant
int* const p=a;                 // p is constant

auto it = m.begin();            // auto type
```

5.6. Operators

5.6.1. Assignment

Assignment operator is call using `=`.

```
int i = 1; // Assign 1 to i
```

Assignment can be combine with Arithmetic and Bitwise operator (see below).

Example :

```
i += 2; // i = i + 2 = 1 + 2 = 3
i *= 2; // i = i * 2 = 3 * 2 = 6
```

5.6.2. Increment And Decrement

Increment and Decrement operator are used to respectively increase and decrease a variable by 1.

```
i++; // Increment, Assign, Return pre-increment value
++i; // Increment, Assign, Return post-increment value
i--; // Decrement, Assign, Return pre-increment value
--i; // Decrement, Assign, Return post-increment value
```

5.6.3. Arithmetic

```
a + b;      // a plus b
a - b;      // a minus b
a * b;      // a multiply by b
a / b;      // a divide b
a % b;      // a modulo b
```

5.6.4. Relational

```
a > b;  // a greater than b
a >= b; // a greater than or equal to b
a < b;  // a less than b
a <= b; // a less than or equal to b
a == b; // a equal to b
a != b; // a not equal to b
```

5.6.5. Logical

```
a && b;  // a and b
a || b;  // a or b
!a;      // not a
```

5.6.6. Bitwise

```
a & b;  // bitwise a and b
a | b;  // bitwise a or b
a << 1; // bitwise shift left a by 1
a >> 1; // bitwise shift right a by 1
```

5.7. Conditions

5.7.1. If/Else

```
if(a){
        // Do Stuff if a
}
else if (b){
        // Do Stuff if not a and b
}
else{
        // Do Stuff if not a and not b
}
```

One line conditional statements can also be written without scope.

```
int x;

if(conditionA) x = 0; // if a
else if (conditionB) x = 1; // if not a and b
else x = 2; // if not a and not b

if(conditionA)
        x = 0; // if a
else if (conditionB)
        x = 1; // if not a and b
else
        x = 2; // if not a and not b
```

5.7.2. Ternary

Ternary syntax `condition ? a : b;`

```
int test = 0;
bool result1 = test == 0 ? true : false; // result1 = true
int result2 = test != 0 ? 1 : 2; // result2 = 2;
```

5.7.3. Switch

```
switch(expression){
        case x:
                // do stuff if x
                break; // exit out of switch
        case y:
                // do stuff if y
                break; // exit out of switch
        default:
                // do stuff if no other case match
}
```

5.8. Loops

5.8.1. While

```
// do 'a' 0 or more times while condition is true
while(condition){
        a;
}
```

5.8.2. Do While

```
// do 'a' 1 or more times while condition is true
do {
        a;
}
while (condition);
```

5.8.3. For

```cpp
// do x, then do a and z while y is true
for(x; y; z){
        a;
}

// Example
int sum = 0;
for(int i = 0; i < 10; ++i){
        sum += i;
}
```

5.8.4. For Each

```cpp
// Range-based loop
for(type x : y){
        a;
}

// Example
int array[5] = {1, 2, 3, 4, 5}
int sum = 0;
for(int element : array){
        sum += element;
}
```

For Each is often used with `auto` variable type for readability purposes : `for(auto element : array)`.

For performances and as a good practice it is also recommended to use const reference as the type if possible : `for(const auto& element : array)`.

5.8.5. Break

`break` is used to jump out of a loop. Is other words it interrupt the loop and jump to the `}` that close the loop scope.

```cpp
for(int i = 0; i < 10; ++i){
        if(i == 2) break; // break after 3 iteration
}
// jump here after 3 iteration
```

5.8.6. Continue

`continue` is used to break the current iteration of the loop and jump to the next one.

```
int sum = 0;
for(int i = 0; i < 5; ++i){
        if(i == 2) continue; // skip 3rd iteration
        sum++;
}
// evolution of i   : [0, 1, 2, 3, 4]
// evolution of sum : [1, 2, 2, 3, 4]
```

5.9. Arrays

5.9.1. Declaration

```
int a[4];                                // uninitialized array, can be filled using wri
int a[4] = {2, 4, 8, 16};       // initialized array
int a[] = {2, 4, 8, 16};        // initialized array can omit size
```

5.9.2. Read

```
// a == [2, 4, 8, 16]
int x = a[0]; // x == 2
int x = a[2]; // x == 8
```

5.9.3. Write

```
// a == [2, 4, 8, 16]
a[0] = 0; // a == [0, 4, 8, 16]
```

5.9.4. Multidimensional

```
// declare
int a2D[2][4] = {
        {1, 2, 4, 8},
        {16, 32, 64, 128}
}

// read
int x = a2D[1][1]; // x == 32

// write
a2D[1][1] = 0; // a2D == [[1, 2, 4, 8], [16, 0, 64, 128]]
```

6. Function

Functions define block of code that can be executed when called. They can take parameters and/or return value.

6.1. Return

In C++ `return` keyword is used to return a value from a function.

```
int f(){
        return 2;
}
int x = f(); // x == 2
```

6.2. Declare Function

Declare function with no return and no parameters

```
void f();
void f(){
        // code
}
```

Declare function with return value and parameters

```
ReturnType f(Type1 param1, Type2 param2, ..., Type3 param3 = defaultValue, ...);
int sum(int x, int y = 1){
        return x + y;
}
```

6.3. Call Function

```
f();

ReturnType x = f(param1, param2, ..., param3);
int x = sum(5, 5); // x == 10

ReturnType x = f(param1, param2, ...); // using default value
int x = sum(5); // x == 6
```

6.4. Overloading

Multiple function can share the same name with different parameters. Doing so is call function overloading.

```
int sum(int x, int y) { return x + y }
float sum(float x, float y) { return x + y }

int r = sum(5, 5);              // call int function, r == 10
float r = sum(5.0, 5.0) // call float function, r == 10.0
```

7. Namespace

Namespaces provides a scope to the identifiers (types, functions, variables, etc) inside of it. Namespaces are used to organize code and prevent name collisions.

```
namespace N         // Create namespace
{
        class T {};
        void f();
}
N::T t;             // Create object t from T define in namespace N
N::f();             // Call function f for namespace N
using namespace N;  // Make namespace content usable without N::
```

8. Enums

Enums are used to store a group of constant values. Enums can be unscoped or scoped.

8.1. Unscope Enums

For unscoped enum, the scope is the surrounding scope.

```
// simple enum
enum Color {
  RED,
  GREEN,
  BLUE
};
```

```
// enum with assigned values
enum Color {
  RED = 10,
  GREEN = 20,
  BLUE = 30
};
```

```
// Declare and Assign Enum
Color color = RED;
```

8.2. Scope Enums

For scoped enum, the scope is the enum-list itself.

To create Scope Enum the declaration should specify `class` keyword.
Specifying enum scope is know required to get enum values.

- `enum Color{}` becomes `enum class Color{}`.
- `Color color = RED` becomes `Color color = Color::RED`

9. Classes And Structures

Technically classes are identical to structures except that Structures default access is public while classes default access is private. Therefore all concept showcased with classes can be applied to structures as well. With that said, Structures are commonly used to store data and declare new basic types while Classes are used to define Objects.

9.1. Member Access

In order to access members of struct and classes, C++ provide `.` and `->` operator.

```
.       // Access member of object or object reference
->      // Access member of object pointer

// Examples
myObject.x;
myObjectPtr->x;
```

9.2. Structures

Structure can be used as a container that can encapsulate data and functions.
They are commonly used to store data and declare new basic types.

```
// Declare
struct StructName {
        Type1 member1;
        Type2 member2;
        void f() {
                // code
        };
        ...
}
StructName s; // Create variable of type StructName

// Example
struct Vector2D {
        int x = 0;
        int y = 0;
        int LengthSquared(){
                return x*x + y*y;
        }
}
Vector2D v;
v.x = 1;
int x = v.x; // x == 1
int lengthSquared = v.LengthSquared();
```

9.3. Classes

```
class Base {

public:
        int x;                                  // Declare member variable (attribute)
        void Method();                          // Declare member function (method)
        const void constMethod() {} // Does not modify object
};

// Implement function Method of class Base
void Base::Method() {
        // do stuff
}

Base base;              // Create an object base of class Base
base.Method();  // Call function Method on object base
```

9.4. Access specifiers

Access specifiers `public`, `protected` and `private` are used to control object members accessibilities.

- `private` members can only be accessed from inside the class
- `protected` members can only be accessed from inside the class or inherited child classes
- `public` members are accessible anywhere

```
class Base {
public:
        int x; // x is public

private:
        int y; // y is private
};
```

9.5. Accessor and Mutator (Getter and Setter)

Accessors (Getters) are used to return inaccessible members of an object (like protected are private ones).

Mutators (Setters) are used to edit inaccessible members of an object (like protected are private ones).

```
class Base {
private:
        int x; // x is private

public:
        int GetX(); // Accessor of x
        void SetX(int i); // Mutator of x
};

int i = GetX(); // i == x
SetX(2); // x == 2
```

9.6. Constructor And Destructor

```
class Base {
public:
        int x;

        // Default constructor
        Base() {}

        // Constructor with parameter
        Base(int i){
                x = i;
        }

        // Constructor with parameter, using initialization list
        Base(int i) : x(i) {}

        // Copy constructor
        Base(const Base& t){
                x = t.x;
        }

        // Destructor (cleanup routine)
        ~T();
};

Base b0;        // call default constructor
Base b1(1); // call constructor that will initialize x to 1
Base b2(b1) // call copy constructor using b1 as source to copy from
```

Destructors are automatically called when the object is deleted. For example it is called when :

- The object is out of scope (end of function, loop, etc).
- At the end of the program.
- When delete is called.

9.7. Operator Overloading

In C++, operators can be overload for specific data types. For example the string class can overload the `+` operator to perform concatenate operation.

```
class Base {
public:
        int x;

        // Add(+) operator overload
        Base operator+(const Base& other){
                Base b;
                b.x = x + other.x;
                return b;
        }
};

Base b1, b2;
Base b3 = b1 + b2; // call + operator overload
```

9.8. Static

`static` keyword can be used in classes body to declare variables and methods that belong to the class itself. Such variables are share by all objects of the class and therefore have the same value. Static variables and methods can also be called directly from the class itself like this `Base::staticVariable`, `Base::staticMethod()`.

```
class Base {
public:
        // StaticMember : Data shared by all Base instances
        static int staticMember;
        // Static method. Can access static data but not instance data.
        static void staticMethod();

};

// Initialization of static member (required)
int Base::staticMember = 2;

// Reading static member
int x = Base::staticMember;

// Call to static method
Base::staticMethod();
```

9.9. Inheritance

In C++ classes can inherit attributes and methods from another class.

```
class Base {
public:
        // Virtual methods can be override by derived classes
        virtual void VirtualMethod();
};

// Derived class inherits all members of the Base class
class Derived: public Base
{
public:
    // Override parent method
        virtual void VirtualMethod() override;
};
```

Classes can also inherit from multiple base classes which is known as multiple inheritance.

```
class Derived: public Base1, public Base2 {

};
```

Classes can have a polymorphic behaviour. This term is used to describe the ability of objects to have many different type. This concept is used to managed multiple objects from a common ancestor. Think of managing a list of Vehicles that contain cars, bikes, trucks, etc.

```
class Polygon {
public:
    float width, height;
        float Area() { return 0.0; }
};

class Rectangle: public Polygon {
  public:
    float area() { return width * height; }
};

class Triangle: public Polygon {
  public:
    float area() { return width * height / 2.0; }
};

Polygon poly[2] = { Rectangle(), Triangle() };

for(auto& p : poly){
        poly.width = 2;
        poly.height = 2;
        float area = poly.area();
}
```

9.10. Interfaces

Interfaces are used to define methods that a class must implement. In C++ Interfaces are defined using abstract classes (declare pure virtual methods). They are inherited by child classes that have to implements defined methods.

```
// pure virtual method declaration
virtual void pureVirtual() = 0;

// virtual method implementation
virtual void pureVirtual() override { /* implementation */ }
```

10. Templates

Template is a tool used to pass data type as a parameter in a class or a function. It allow developer to write code once and reuse it with multiple different type. Templates are expended at compile time to generate function and classes for each required parameter type.

```
// Templated function (class parameter)
template<class T>
T f(T t);

template<class T> class X          // Templated Class
{
        X(T t);                         // Constructor Declaration
};
template<class T> X<T>::X(T t) {}  // Constructor Definition
```

Example :

```
template<class T>
T add(T a, T b){
        return a + b;
}

int intResult = add<int>(1, 1); // intResult = 2
float floatResult = add<float>(0.5, 0.5); // floatResult = 1
```

11. Cast

Cast operation is used to convert a type to another compatible type.

```
dynamic_cast<T>(x)          // Converts x to a T, run time check.
static_cast<T>(x)           // Converts x to a T, compilation time check
reinterpret_cast<T>(x)      // Reinterpret x bits as T
const_cast<T>(x)            // Remove const
```

12. Error Handling

12.1. Exceptions

```
try {
  // try do stuff
  throw exception; // Throw an exception when a problem arise
}
catch (int exception) {
  // handle error if integer exception is thrown
}
catch (...) {
  // handle error if any other type of exception is thrown
}
```

12.2. Assertions

An assertion statement specifies a condition that you expect to be true at a point in your program. If that condition is not true, the assertion fails, execution of your program is interrupted, and the Assertion Failed dialog box appears.

```
assert(condition); // terminate with error if condition is false
```

13. Smart Pointers

Smart pointer are used to avoid memory leaks.

Unique Pointer :

- Ensure only one copy of an object exists.
- Copy operation is unsupported but the pointer can be moved.
- Free memory when pointer is invalidated.

Shared Pointer :

- Count references to the object.
- Free memory when count reach 0.

Weak Pointer :

- Reference an object without being the owner.
- Mostly use to break circular reference.
- Weak Pointer should always be checked before use.

14. Data Structures

Data Structures define ways to organize data. They are playing a huge role in computer science. Storing data is the right data structure allow to process a large amount of data in the shortest amount of time. An efficient data structure makes use of minimum memory space and takes the minimal possible time to execute.

Having access to low level functionalities, especially pointers, is mandatory to implement efficient data structures. Therefore in C++, it is not rare to develop new data structures specific to the problem that need to be solve. Doing so allow to improve code readability, execution time, code complexity and optimize memory usage.

Data Structures

15. Advanced Concept

15.1. SizeOf

`sizeof()` operator return the size of a type in bytes.

```
int sizeOfInt = sizeof(int); // 4

int a[3];
int sizeOfArray = sizeof(a); // 12
int lengthOfArray = sizeof(a) / sizeof(int); // 3
```

15.2. Inline

Normal function are called by pointer during execution. Instead, `inline` keyword can be used to specify to the compilator that you want the function content to be copy/paste in place each time it is called. This is useful on small function that are called a lot to increase performances.

```
inline f();
```

15.3. Friend

`friend` keyword can be used in a class body and give protected or private member access to a function or another class.

```
class Base {
private:
        int privateVariable;
protected:
        int protectedVariable;
public:
        friend class FriendClass;
        friend void friendFunction(Base& b);
};

class FriendClass {
public:
        int SumC1(Base& b) {
                // Base objects variable can be use directly despite being private/protected
                return b.privateVariable + b.protectedVariable;
        }
};

void friendFunction(Base& b){
        // Base objects variable can be use directly despite being private/protected
        return b.privateVariable + b.protectedVariable;
}
```

16. References

Learn C++ (learncpp.com)

Learn C++ (cplusplus.com)

Learn C++ (programiz.com)

Understanding Memory Management

Detailed Pointer Tutorial

Data Structures