C++ Lecture Level 1

[1. Basic C++ types & statements 3](#_Toc306190107)

[1.1 Basic types 3](#_Toc306190108)

[1.1.1 Print format 3](#_Toc306190109)

[1.1.2 Cast 3](#_Toc306190110)

[1.2 Const 4](#_Toc306190111)

[1.2.1 Const variable 4](#_Toc306190112)

[1.2.2 Const for function 4](#_Toc306190113)

[1.3 Pointer & reference 5](#_Toc306190114)

[1.4 Basic statements 6](#_Toc306190115)

[1.4.1 Complex statement 6](#_Toc306190116)

[1.4.2 Loop(for/do…while) 6](#_Toc306190117)

[1.4.3 Switch & case 6](#_Toc306190118)

[1.4.4 Goto 6](#_Toc306190119)

[2. C++ file 7](#_Toc306190120)

[2.1 Header file 7](#_Toc306190121)

[2.2 CPP file 7](#_Toc306190122)

[2.3 MISC 7](#_Toc306190123)

[3. Function 7](#_Toc306190124)

[3.1 Parameters 7](#_Toc306190125)

[3.2 Return 8](#_Toc306190126)

[3.3 Assert 8](#_Toc306190127)

[3.4 Inline 8](#_Toc306190128)

[3.5 C functions 8](#_Toc306190129)

[3.6 MISC 8](#_Toc306190130)

[4. Class 9](#_Toc306190131)

[4.1 Members and functions 9](#_Toc306190132)

[4.2 Constructor & destructor 9](#_Toc306190133)

[4.3 Virtual function 10](#_Toc306190134)

[4.5 Inheritance 10](#_Toc306190135)

[4.6 MISC 10](#_Toc306190136)

[5. Operators 11](#_Toc306190137)

[5.1 Concept 11](#_Toc306190138)

[5.2 Rules 11](#_Toc306190139)

[5.3 Samples 11](#_Toc306190140)

[5.4 Caution 11](#_Toc306190141)

[6. Memory 12](#_Toc306190142)

[6.1 new/delete and malloc/free 12](#_Toc306190143)

[6.2 Memory fragment 12](#_Toc306190144)

[7. Practice 12](#_Toc306190145)

# 1. Basic C++ types & statements

## 1.1 Basic types

### 1.1.1 Print format

%% a percent sign

%c a character with the given number

%s a string

%d a signed integer, in decimal

%u an unsigned integer, in decimal

%o an unsigned integer, in octal

%x an unsigned integer, in hexadecimal (lower case)

%e a floating-point number, in scientific notation

%f a floating-point number, in fixed decimal notation

%g a floating-point number, in %e or %f notation

%X like %x, but using upper-case letters

%E like %e, but using an upper-case "E"

%G like %g, but with an upper-case "E" (if applicable)

%p a pointer (outputs the pointer value's address in hexadecimal)

The print information is useful for debugging. But the print should be only existed in debug and release version, never output information in final version because the performance.

### 1.1.2 Cast

* static\_cast
  + any type conversions that the compiler performs implicitly can be made explicit through use of static\_cast

1. Cast basic types
2. Cast pointers/reference from derived to base
3. Cast pointers/reference from base to derived. It’s **not safe.**
4. Cast pointer to void\* or from void\* to any other pointer.

* dynamic\_cast

1. Cast pointers/reference from derived to base. The same as static\_cast.
2. Cast pointers/reference from base to derived.

It returns NULL if the pointer doesn’t point to an instance of the target type.

There should be virtual functions for it to work.

RTTI(Run-Time Type Identification). *But the RTTI needs more memory and takes a little performance, so RTTI is disabled in some games like some NDS games, some PSP games.*

1. Cast pointers to void\*. But can’t not cast pointers from void\*.

* const\_cast

You’d better not to use it.

1. Cast const pointers/reference to non cost.
2. Cast volatile pointers/reference to non volatile.

* reinterpret\_cast

1. Cast variable between the basic types. It’s not safe.
2. Cast pointers/reference between integers.

## 1.2 Const

### 1.2.1 Const variable

1. Concept
2. What is const variable?

char chr;

const char\* const pChr1 = &chr; //Both the pointer and its pointed value are const.

const char\* pChr2 = &chr; //The pointed value is const.

char\* const pChr3 = &chr; //The pointer itself is const

char\* pChr4 = &chr; //It’s a normal pointer.

const int C\_Interval = 15; //A constant value.

1. Why it is useful?

It protects values from being modified unexpectedly.

It makes your codes safer for the others to use.

It avoids unnecessary debug.

1. When to use?

Use const whenever you need.

1. Difference from define?

Const value is typed, the compiler can check more for us.

1. Practice
2. Please use it as much as possible when the target variables and instances are not allowed or not necessary to be modified.
3. Only expose the constant value in the header file when it’s referenced by the others.
4. Maintain the relationship properly between constants if they are relative.

const int C\_Interval = 15;

const int C\_DoubleInterval = C\_Interval \* 2;

1. The const member variable can only be initialized in the initialize list. And const member not for class, but for every instance. If you want a const variable for class, you can use enum.

### 1.2.2 Const for function

* + 1. parameter

void funtion(**const** A& a…)

Please take parameters as const when there is no need to modify them, it makes your interface safer for the others. People will be happier to pass their parameters to your function as const.

* + 1. return value

**const** B& funtion(const A& a…)

It’s better to return member/global values (pointer and reference) as const if you don’t want its pointed values to be modified unexpectedly.

* + 1. member function itself (postfix)

const B& funtion(const A& a…) **const**

No member is allowed to be modified in side this kind of functions.

It keeps your object in safe.

The mutable member is an exception.

## 1.3 Pointer & reference

* 1. Concept

1. It’s a way to share data.
2. It benefits both memory and performance.
   1. Practice

|  |  |  |  |
| --- | --- | --- | --- |
|  | Practice | Pointer | Reference |
| Member Variables |  | 1) Point to existing instance. Should be notified when the existing instance is destroyed. 2) Point to self created/duplicated instance. Remember to destroy it. | 1) Usually refer to existing instance. Be careful when the existing instance is destroyed.  2) Can only be initialized in the initialize list. |
| Parameters | Always pass object parameter as pointer or reference to avoid unnecessary copy. | 1) Could be null. 2) Assert it. | 1) Could not be null. |
| Returned Type | Do not return pointer or reference of a temporary variable. | 1) Return pointer if it could return null. | 1) Return reference if there is always a valid value to return. |
| Temp Variables | It helps to avoid unnecessarily copy operation. | 1) Be careful when point to an instance that is in a different or smaller life scope. 2) Be sure there is the code to free the memory that allocated for it. | 1) Be careful when the referenced instance is in a different or smaller life scope. 2) Use reference to hold returned value from a function if it’s already returning a reference. |

## 1.4 Basic statements

### 1.4.1 Complex statement

1. Don’t use too complex statement.
2. Be careful of the order of every statement.

*If ( p != NULL && p->function(…) ).* In most cases there is no bug, but there is bug for NDS because the order for NDS compiler is from right to left.

### 1.4.2 Loop(for/do…while)

1. If there is a “for loop” inside another. Please put the longer loop in the inner.
2. You’d better not modify the loop iterate inside the loop codes.
3. “++i” is better than “i++”.
4. If there is “new” or “malloc” in loop, be careful of “continue”, “break” and “return”.
5. Be careful of infinite loop especial when the loop condition is complex.

### 1.4.3 Switch & case

1. Always add the bracket.
2. Don’t forget the break.
3. Consider the default. Please assert it if you think it would never enter the default.

switch(\_value)

{

case 0:

{

//...

}

break;

case 1:

{

//...

}

break;

default:

{

//...

}

break;

}

### 1.4.4 Goto

It is powerful but very risky.

* Risk:

It could skip important lines like initialization. Then possibly cause run time crash.

* Suggestion:

You’d better not use it. It’s not 100% forbidden to use goto, but once you use it you’d better got a very good reason.

# 2. C++ file

## 2.1 Header file

1. ifndef/define/endif

#ifndef \_\_MY\_CLASS\_H

#define \_\_MY\_CLASS\_H

//…

#endif //\_\_MY\_CLASS\_H

1. Avoid including other header files in header files if possible.

The better way is to declare class in the header file and include the wanted header file in the CPP file.

* It helps to limit the complex and save the compiling time.
* It helps to avoid compiling error when tries to include header files each other.

1. No implement in header file except inline. It helps to save compile time if you just want to change the implement but without modification of interface.

It’s better to separate the declaration and definition of inline.

## 2.2 CPP file

1. Include other header files in CPP. Include them from low level to high level.
2. It only contains the implement for is relevant header file.

## 2.3 MISC

1. #include

#include <filename.h> for API

#include ”filename.h” for non API

(set in INCLUDE environment variable)

1. Put header files and CPP files to different folders.

# 3. Function

## 3.1 Parameters

* Pass parameters.

1. Pass-by-reference

It’s better to pass parameters by reference.

Always do:

void foo(const A& a);

No need to do:

void foo(const int& a);

1. Pass-by-value

It does unnecessary copy operations.

* Many parameters

If there is a function takes a lot of parameters, please consider to take them as a class or structure if it’s logical.

## 3.2 Return

1. Don’t return magic number. Declare enum to maintain them.
2. Always return value as (const) pointer or reference if it refers to something is already stored somewhere safely.
3. No need to return normal values (neither pointer nor reference) as const.
4. It is danger to return temp variable as pointer or reference.
5. Return value in an efficient way.

return String(s1 + s2);

The above is better than the following.

String temp(s1 + s2);

return temp;

## 3.3 Assert

Use Assert when there is illegal thing. You can add assert for the parameters at the begin of function, add assert where you do supposition.

It’s very important to use assert.

## 3.4 Inline

1. The inline keyword should be with the definition not only the declaration.
2. The function with definition in the declaration becomes inline.
3. Use inline function instead of “#define function”.
4. Do not inline big functions. Though compiler will not treat big function as inline, but it is not good practice.
5. You’d better not inline constructor(s) and destructor(s).

## 3.5 C functions

extern “C”

{

void foo(int x, int y);

… // More functions

}

Or

extern “C”

{

#include “myheader.h”

… // More header files.

}

## 3.6 MISC

1. Keep one function as simpler as possible. Implement multiple functions for multiple functionalities.
2. Please add comments when the logic goes complex.
3. You’d better avoid declare static value inside the function.
4. Please consider as many as input cases for you function.
5. Please add if(condition) to filter the invalid input.
6. Please do not commit the codes that do debug output in every frame.
7. Consider complex function as a class.

# 4. Class

## 4.1 Members and functions

1. There should be a good reason to put stuff together as an object.
2. Is it useful?
3. Is it duplicated?
4. Does it make sense?
5. Does it make things simpler or more complex?
6. Is there a better way?
7. Usually, all the member variables should be declared as **protect or private**.
8. It’s easier to monitor and validate your member variables from being modified.
9. It’s easier for you to upgrade your class if you want to do rename or remove.
10. Only expose the interfaces that are necessarily to be so.
11. It’s easier and more straightforward for the others to use your class.
12. It prevents the others from misusing your class by calling the internal functions that may against your design.
13. It’s easier for you to update your class by modifying the internal functions but still keeping the same exposed interfaces.
14. Const function
15. Define functions as const when there is no need to modify the member variables.
16. Be careful when providing the methods for the others to modify your instance.
17. Always provide function like setXXX() to modify you member variable directly.
18. Be careful when return pointer or reference that point to your internal member variables. Always return them as const unless there is a good reason.
19. It’s easier to debug your object if you limit the way to modify it.

## 4.2 Constructor & destructor

1. Constructor
   1. Initialize **ALL** class members in the constructor(s).

You have basically 2 choices, you always initialize your object to a **valid** state or you initialize to an **invalid** state, but it should not be something in between.

* 1. You must write a default constructor if there is a customized constructor.
  2. Initialize list

1. It’s better to initialize the members in the initialize list.
2. It only initializes the base and its own member.
3. It works depends on the order that the members are being declared.
4. Initialize the const member variable and reference variable in the initialize list.
   1. Initialize order: Base class 🡪 member variable declare order.
   2. Don’t initialize members by the virtual function of this class.
5. Destructor
   1. Virtual the destructor of the base class.
   2. Destroy **ALL** the class members in the destructor(s).
   3. Order: Reversed member variable declare order 🡪 base class.
6. initialize() and destroy()

It is recommended to implement function initialize() and destroy(), then call them in constructor(s) and destructor.

* 1. Check NULL pointer before allocation.
  2. Check NULL pointer before deallocation, set NULL pointer after deallocation.
  3. The application will crash when accessing wild pointer.
  4. The application will crash when tries delete one pointer twice.

## 4.3 Virtual function

1. The function in the derived becomes virtual if there is a same function declared as virtual in its parent class.
2. There is no need to declare the constructor(s) as virtual.
3. Remember to declare the destructor of the base class as virtual.

## 4.5 Inheritance

It is a good way to reuse the code, but never ill-use it:

1. Don’t use inheritance with no mean. If class A has no relationship with class B, never let B derive from A.
2. If B is kind of A, and all the function and properties is mean for B. Let B derive from A.

## 4.6 MISC

1. Avoid generating duplicated codes.
2. You don’t want to maintain all of them for actually only one modification.
3. It would be buggy if you miss one of them when doing the above action.
4. Please make them as a function so you can call it when it’s needed.
5. Always avoid including the header files in the other header files.
6. It complicates the reference and slows down the compiling.
7. It causes compiling error when including each other.
8. Remove the old codes if they are no longer needed.
9. It would confuse the others.
10. It is wasting both memory and CPU process.
11. It could cause potential bugs.
12. Add assert.
13. Assert the input parameters.
14. Assert when the instance is going against your design.
15. It proves your innocence when someone is using your class in wrong way.

# 5. Operators

## 5.1 Concept

1. Overload function.
2. Cast.

## 5.2 Rules

1. It doesn’t work for basic types.
2. Can not overload “.”.
3. Can not overload character that is not actually an operator symbol.
4. Can not change their priorities.

## 5.3 Samples

* + 1. Assignment operator

1. Here are the major steps.

Object & Object::operator=(const Object & rhs)

{

//Step 1, check if it’s itself

if (this == &rhs)

return \*this;

//Step 2, clean up the old data.

Cleanup();

//step 3, assignment

Base::operate =(other); //If there is base class

Setup(rhs);

//step4, return its reference.

return \*this;

}

1. We’d better implement assignment operators to avoid memory leak by using default assignment function.
2. You can private it if you don’t want to code it without trouble.
   * 1. Cast operator

operator const char\* () { return m\_str; }

## 5.4 Caution

Only use it when it making things more convenient, otherwise you are making trouble.

# 6. Memory

## 6.1 new/delete and malloc/free

The new/delete always is override in game engine to get good memory management.

1. Pair use new/delete, new[]/delete[], malloc/free.
2. Always set the pointer to NULL when it is initialized(like constructor) or after deallocation.
3. Be careful when using new/malloc in loop with “break”, “continue” or “return”.
4. Check if the pointer is NULL before you assign memory address to it.
5. Check the pointer after new/malloc.
6. Always delete/free memory after if you new/delete it. Don’t leave memory leak in engine.

## 6.2 Memory fragment

There are many ways that can cause memory fragment. It can be solved by memory management. But we can avoid memory fragment by good practice

1. Assign memory for the full-time existing variable as early as possible.
2. Assign big memory-block as early as possible.
3. Pre-assign memory block, like use reserve to assign enough memory for vector.

# 7. Practice

1. Always check compiling warning and try to clean them. If you can’t find the reasons of some errors, maybe you can get information from warning.
2. Don’t forget to add assert when you has supposition.
3. Check the boundary of array.

Don’t access array out of boundary. This always causes some bugs that can’t be fixed easily because you don’t know what memory you accessed.

1. When use float, write as “0.1f” but not “0.1”.
2. Don’t copy code.
3. No hardcode. You can use const/define/enum, or load parameters from config file.