C++ Programming - Intermediate

Author: Dipsy Wong

Preprocessor directives

Preprocessor directives are some codes for the compiler to read. During the compile time, the compiler will compile the code according to the preprocessor directives by replacing the code to be compiled.

A. `#define`

```
#include <iostream>
#define MY_AGE 18
int main(){
  std::cout<<MY_AGE;
  return 0;
}
```

The output will be `18`. But `MY_AGE` is not a variable, it is fixed by `#define` (mentioned in constant section in basic section). It is done by replacing `MY_AGE` with 18 during the compile time, so the code actually look like this in the view of compiler:

```
#include <iostream>
int main(){
  std::cout<<18;
  return 0;
}
```

And this is the reason why value of `MY_AGE` cannot be changed.

B. `#include`, `#ifndef` and `#endif`

`#include` works by copying the code. For example,

`main.cpp`

```
#include "hello.h"
#include "foo.h"
...
```

This will copy all of the content of `hello.h` to `main.cpp`.

However, if `hello.h` also have to include `foo.h` , there will be two copies of `foo.h` in the compilation. At this moment, we need to use `#ifndef` `#endif` to make sure the content of `foo.h` only include once.

```
#ifndef FOO_H
#define FOO_H
int bar(){
  return 1234;
}
#endif
```

Then, when `foo.h` is included the first time, compiler will find `FOO_H` haven't be defined, then it will compile code between `#ifndef` and `#endif` , and define `FOO_H` , so when `foo.h` is included the second time, compiler will find `FOO_H` is already defined and will not compile code between `#ifndef` and `#endif`.

As a result, these are very useful for `.h` files.

C. `#pragma`

You may see `#pragma once` in some codes for embedded system too. It is a non-standard way to make sure the file is only included once, and it supports most of the processors. You may choose either the `#ifndef` - `#define` - `#endif` sequence or `#pragma once` , but we recommend the former one.

Function Overload

Function overload means functions having same name, but different parameters, in the same scope. For example:

```
int max(int a, int b){return (a>b?a:b);}
float max(float a, float b){return (a>b?a:b);}
double max(double a, double b){return (a>b?a:b);}
```

For different types of variable pass to function max, different version of max is called. You may even have different implementations

```cpp
#include <iostream>
void print(){
  std::cout<<"param is nothing"<<std::endl;
}
void print(int x){
  std::cout<<"param is integer with value = "<<x<<std::endl;
}
void print(float x){
  std::cout<<"param is float with value = "<<x<<std::endl;
}
int main(){
  float x = 1.234;
  print();           //param is nothing
  print(x);          //param is float with value = 1.234
  print((int) x);    //param is float with value = 1
  return 0;
}
```

Note: if you overload a function which have default parameters, it will cause ambiguity error

```cpp
#include <iostream>
/*version 1*/
void print(){
  std::cout<<"param is nothing"<<std::endl;
}

/*version 2*/
void print(int x = 0){
  std::cout<<"param is integer with value = "<<x<<std::endl;
}

int main(){
  print();           //are you calling version 1, or version 2 with parameter x = 0
  return 0;
}
```

Inline Function

The inline functions are a C++ enhancement feature to increase the execution time of a program. Functions can be instructed to compiler to make them inline so that compiler can replace those function definition wherever those are being called. Compiler replaces the definition of inline functions at compile time instead of referring function definition at runtime. NOTE- This is just a suggestion to compiler to make the function inline, if function is big (in term of executable instruction etc) then, compiler can ignore the “inline” request and treat the function as normal function.

```
inline int max(int a, int b) {  //fine
  return (a > b) ? a : b;
}

inline int min(int a, int b) {  //still fine
  char msg = "this line is totally rubbish";
  return (a < b) ? a : b;
}
```

This will replace all `max(some_int, another_int)` into `(some_int > another_int) ? some_int : another_int` during compile time, which can help reducing running time as calling a function waste some time.

Constexpr

Constexpr, constant expression, defines an expression that can be evaluated at compile time. It can be a variable or function with one line of return only. For example:

```
constexpr float foo = 1+2+3+4+5+6+7;    //valid
constexpr int max(int a, int b) {   //valid
  return (a > b) ? a : b;
}
constexpr int min(int a, int b) {   //compile error
  char msg = "this line is totally rubbish";
  return (a < b) ? a : b;
}
```

This will replace all `foo` to result of `1+2+3+4+5+6+7` which is `28`; `max(some_int, another_int)` into `(some_int > another_int) ? some_int : another_int` during compile time, which can help reducing running time as calling a function waste some time. Note that constexpr function must be in recursion form instead of iteration.

```
//recursion form, correct
constexpr int factorial(int n){
  return n <= 1? 1 : (n * factorial(n - 1));
}

//iteration form, which need multiple line, compilation error
constexpr int factorial(int n){
  int x = 1;
  for(int i=1;i<=n;i++){
    x*=i;
  }
  return x;
}
```

Boolean Function

Boolean function is a coding style, it will do something like a void function and at the same time return the action is successful or not. For example:

```
#include <iostream>
bool FindSomewhereToDate(){
  //do something, if success return 1, else return 0
}

int main(){
  if(FindSomewhereToDate()){
    //if success
    std::cout<<"you get somewhere to date";
  }
  else{
    std::cout<<"you don't have a girlfriend/boyfriend";
  }
  return 0;
}
```

Notice: actually the return 0 in the `int main()` is for this purpose. If the return value of the main function is not 0, the system will know there is an error.

If a single true false value is not enough, you can return a enum value.

Array

An array is a variable will can store list of things. If a variable acts like a box, array is the cabinet. Array can be declared like this:

```
//a boolean array which can store 100 true false values;
bool bool_array[100];

//an integer array which can store 10 integers
//with first element is 0, second is 1, etc.
int int_array[10]={0,1,2,3,4,5,6,7,8,9};
```

Be careful, array which have variable length can only be declared inside scope.

```
#include <iostream>
int x = 10;
bool y[x]; //invalid array declaration
int main(){
  bool z[x]; //valid array declaration
  return 0;
}
```

To get and set value of array, we use `[]` operator. Always remember, the very first index is always `0` !

```
std::cout<<a[0]; //print the first element of array a.
a[12] = true; //set the value of the element having index 12 in array a to true
```

There is also multi-dimension array. It works like array of array, which still can be used easily.

```
//a integer array which have 3 row (actually each row is another integer //array), each row
have 5 element
int matrix[3][5]={
  {0,1,2,3,4},
  {5,6,7,8,9},
  {10,11,12,13,14}
};

std::cout<<matrix[1][3];//get matrix's index 1 row's index 3 element, which is 8
matrix[1][3] = 100; //set matrix's index 1 row's index 3 element's value to 100
```

If you wish to initialize each elements of an array with certain values, you may use a for loop. Or, if you just want to zero all the elements, you may do the following.

```cpp
#include <cstring> // need this for memset()
int main(){
  int a[n]; // n is any arbitrary number
  memset(a, 0, sizeof(a)); // or memset(a, 0, sizeof(int) * n); or memset(a, 0, a[0] * n);
  int matrix[n][m]; // n & m are any arbitrary numbers
  memset(matrix, 0, sizeof(matrix[0][0]) * n * m); // or memset(matrix, 0, sizeof(int) * n *
m);
}
```

Note: `sizeof` would return the allocated space (in byte) for certain variable type.

Warning: Never access index which is out of array boundary

```cpp
int a[10];
a[11] = 100;    //out of bound, memory leak, program die
a[10] = 200;    //out of bound, memory leak, program die
//the value inside [] can only be 0<=integer<10
```

String

String is actually array of characters. They can be used like an array.

```cpp
#include <iostream>
#include <cstring>  //for data type string, standard string

int main(){
    char a[5] = {'h','e','l','l','o'}; //char array style declaration
    char b[6] = "world";    //string style declaration, which always end by a null character ('\0') implicitly
    std::string str = "lol";        //standard string style declaration, actually is a dynamic length character array

    std::cout<<a<<" "<<b<<std::endl;    //output hello world
    std::cout<<str<<std::endl;      //output lol

  //output hello in vertical
    for (int i=0; i<5; i++){
        std::cout<<a[i]<<std::endl;
    }

  //output world in vertical
    for (int i=0; i<5; i++){
        std::cout<<b[i]<<std::endl;
    }

  //output lol in vertical
    for (int i=0; i<5; i++){
        std::cout<<str[i]<<std::endl;
    }
  return 0;
}
```

String methods (Not for standard string, which can directly use `=` and `+` operator)

method	description
`strcpy(str1,str2)`	copy the content of `str2` to `str1`
`strcat(str1,str2)`	append the content of `str2` to `str1`

There are more methods left for you to discover (#

Reference

Reference is the address of variable, which is meant by `&` operator. First, we need to know memory. Memory is the way our C++ program store variable value, which is store in an address inside the memory pool. When we are getting or setting value of the variable, actually we are accessing its address.

```
//declare integer x, let's assume its address is 0x123456
int x = 10;

//define integer y have same address as x, 0x123456, which mean x and y always have same value.
int& y = x;

//display the value of y, which mean display the value stored in 0x123456 which is 10
std::cout<<y<<std::endl;

//set value of x to 20, which mean set the value stored in 0x123456 to 20
x = 20;

//display the value of y, which mean display the value stored in 0x123456 which is 20
std::cout<<y<<std::endl;

//set value of y to 50, which mean set the value stored in 0x123456 to 50
y = 50;

//display the value of x, which mean display the value stored in 0x123456 which is 50
std::cout<<x<<std::endl;
```

So you can see x and y always have same value.

Note: The memory address of variable is fixed when defined, so the operation `int& y = x` can only be done in declaration.

Pass by Reference VS Pass by Copy

With the above knowledge, you can easily classify what is the different between pass by reference and pass by copy.

*pass by **reference***

cup = 

```
fillCup(      )
```

*pass by **value***

cup = 

```
fillCup(      )
```

www.penjee.com

```cpp
#include <iostream>

//this is a pass by reference function
int addByReference(int& x, int& y){
  x += y;
  return x;
}

//this is a pass by copy function
int addByCopy(int x, int y){
  x += y;
  return x;
}

int main(){
  int a = 3, b = 9, c = 3, d = 9;

  //the two functions have exactly the same operation, so their return value are the same
  std::cout<<addByReference(a,b)<<std::endl;     //12
  std::cout<<addByCopy(c,d)<<std::endl;      //12

  //however, the operation x += y is changing the value of different address, their actual effects are different
  std::cout<<a<<std::endl;  //12
  std::cout<<c<<std::endl;  //3
  return 0;
}
```

Why the last two outputs are different?

As we know that, passing parameters to function is like assigning variables, when you do `addByReference(a,b)`, it is doing `int& x = a; int &y = b;`, so integer `x` and `y` have the same address as `a` and `b` respectively. Therefore, when we do `x += y`, it is actually adding the value in address of `b` to address of `a`. That's why after calling the function, the value of `a` is changed.

On the other hand, when you do `addByCopy(c,d)`, it is doing `int x = c; int y = d;`. Although their values are the same, it is creating brand new integers `x` and `y` in a new address in the memory pool. They are just copying the values of `c` and `d` and they are independent to each other now. When we do `x += y`, it is changing the value in address of `x` but not address of `c`. That is why after calling the function, the value of `c` is unchanged.

Little Tips: if you add `const` to the parameter of functions, the function will accept constant as parameter

```
int addByReference(int& x, const int& y){
  x += y;
  return x;
}
int a = 3;
addByReference(a,9);     //now it is valid
```

Pointer

Pointer Basics

Pointer is a variable type which can store a memory address. Pointer will be useful when we need to deal with variables across function scope. Let us call the variable that "the pointer is pointing to" "pointee". To get the address of a variable, we use `&`. To get the value of pointee, namely dereferencing, we add a `*` before the pointer. For example:

Note: To declare a pointer with no initial pointee, we can use `NULL` (C style) or `nullptr` (c++0x style). Both of them meaning a pointer pointing to nothing. `nullptr` recommended.

Be careful, never change the value of null pointer. This will change the value in a r