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

```
#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

```
#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.

```
#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

```
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.

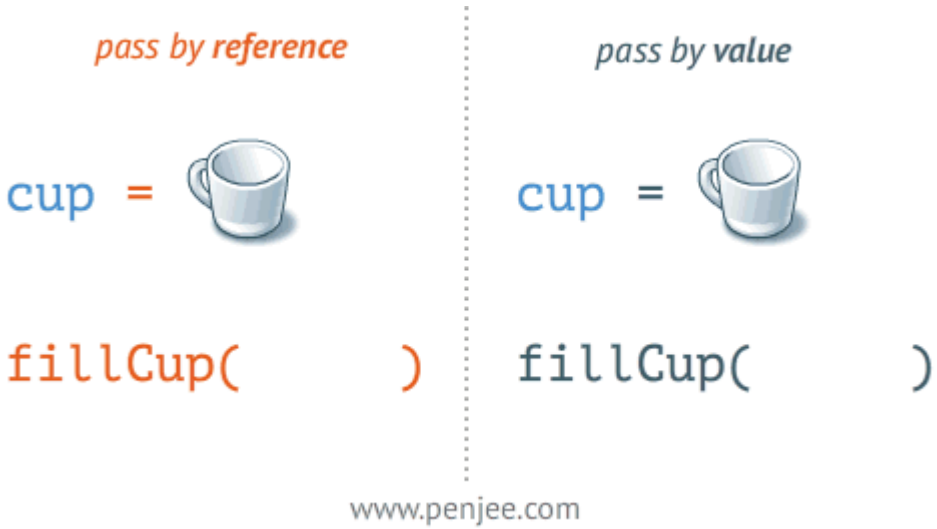
```
//declare integer x, let's assume its address is 0x123456
int x = 10;

//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.




```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:

```
//declare a integer x, assume its address is 0x123456
int x = 12;

//&x is getting the address of x, which this line will print out 0x123456
std::cout<<&x<<std::endl; //output: 0x123456

//declare a integer pointer pInt pointing to the address of x, which is 0x123456
int* pInt = &x;

//output the value of pInt, which is 0x123456
std::cout<<pInt<<std::endl; //output: 0x123456

//output the value of pointee of pInt, which means printing the value in address 0x123456, which
is 12
std::cout<<*pInt<<std::endl; //output: 12

//change the value of x, which mean changing the value in address 0x123456
x = 24;

//output the value of pointee of pInt, which means printing the value in address 0x123456, which
is 24
std::cout<<*pInt<<std::endl;    //output: 24

//change the value of pointee of pInt to 48, which means changing the value in address 0x123456
to 48
*pInt = 48;

//print out the value of x, which means printing the value on 0x123456, which is 48
std::cout<<x<<std::endl;    //output: 48
```

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 random location inside the memory pool, result in hard fault (the program suddenly stop while running).

```
float* foo = NULL;
unsigned char* bar = nullptr;

//**never do this, as foo and var are both pointing to nothing**
*foo = 123; //result: your program GG
*bar = 'd'; //result: your program GG
```

Therefore, you should always check the existence of pointee before dereferencing.

```
if(foo != NULL){
    *foo = 123;
}
if(bar != nullptr){
    *bar = 'd';
}
```

On class task: make a swap function

```
int a = 3, b = 10;     //a and b can be any value
//do a swap function
std::cout<<a<<" "<<b<<std::endl; //10 3
```

Array as a Pointer

Array is actually a pointer pointing to its first element, and a string is actually a character pointer pointing to the first character. For example:

```
int int_array[10]={0,1,2,3,4,5,6,7,8,9};
std::cout<<int_array<<std::endl; //print the address of first element
std::cout<<*int_array<<std::endl; //print the value of first element, which is 0

char buff[12]="hello world";
std::cout<<*buff<<std::endl;    //print out h
```

Memory offset

If we plus or minus some value with the pointer, we can access its neighbor, and this is how array works.

```
int int_array[10]={0,1,2,3,4,5,6,7,8,9};
std::cout<<int_array<<" "<<*int_array<<std::endl;        //0x100000 0
std::cout<<int_array+1<<" "<<*(int_array+1)<<std::endl; //0x100004 1
std::cout<<int_array+2<<" "<<*(int_array+2)<<std::endl; //0x100008 2
```

So you can see, `any_array[n]` is totally equivalent to `*(any_array+n)`. Notice that each address is storing 1 byte, so if you plus `n` to the address, the memory offset is `n` times the size of variable, like in the example, the size of integer is 4 byte, so the offset is `n*4`. To get the size of a variable, we can use `sizeof(myVar)`, and so if we do

Memory Allocation `new`

Aside from assigning a reference to a pointer, you can allocate a new memory space for the pointer, it may be a single variable or array. Remember, delete the memory after finishing using it, or you will be wasting memory. If you keep allocating without deleting it, it will result in memory leaks, and the program may crash at anytime.

```
int* x = nullptr;   //deinfe a pointer

x = new int;         //allocate a size of a integers to x
//some usage for x
delete x;

x = new int[12];    //allocate a size of 12 integers to x
//some usage for x
delete [] x;         //release allocated memory of x
```

Void Pointer `void*`

`void*` is a pointer which can point to address of any type of variable. In common, `int*` 's pointee is always a `int`, `float*` 's pointee is always a `float`, and that is how `int*` differ from `float*` , and also for other types of pointer except `void*`. `void*` 's pointee can be any variable type. Notice that before dereferencing void pointer, you need to cast it as a typed pointer.

```
#include <iostream>
int myInt = 24;
float myFloat = 0.01;
void* myPtr = NULL;
int main(){
  myPtr = &myInt;
  std::cout<<&myInt<<" "<<myPtr<<std::endl; //0x654321 0x654321
  std::cout<<*(int*)myPtr<<std::endl;       //24
  *myPtr = 39;                               //error
  *(int*)myPtr = 39;                         //myInt successfully become 39
  myPtr = &myFloat;
  std::cout<<&myFloat<<" "<<myPtr<<std::endl;//0x654325 0x654325
  std::cout<<(*myPtr)<<std::endl;            //error
  return 0;
}
```

Function Pointer

Every function is stored in a memory address, and when you calling a function, you are running code in that memory address. Therefore, a function is also a pointer and it can be stored as a variable.

```
#include <iostream>
int main(){
  std::cout<<(void*)main<<std::endl;     //displace the address of function main
}
```

Declare a C style function pointer is a bit annoying, or you can use standard function.

```
//functions.h

void f1(){}
int f2(){return 1;}
float f3(bool f){return f;}
```

```
//C style function pointers
#include <iostream>
#include "functions.h"

//<return_type>(*<pointer_name>)(parameters_type) = nullptr;
void(*ptr1)() = f1;
int(*ptr2)() = f2;
float(*ptr3)(bool) = f3;

int main(){
  void vptr = (void*)f3;                                //use of void pointer
  std::cout<<(float(*)(bool)vptr)(1)<<std::endl;     //casting to function pointer

  ptr1();   //call the pointee of function pointer
  return 0;
}
```

```
//C++ standard function
#include <functional>
#include "functions.h"
std::function<void()> fptr1 = f1;
std::function<int()> fptr = f2;
std::function<float(bool)> fptr = f3;

ptr1();     //call the pointee of function pointer
```

Data Structure

This chapter we will talk about `enum`, `enum struct` and `struct`.

Enum

Enum is actually using enumerator to represent different numerical values. This will make the code look nicer. Maybe an example will be clearer.

```
//enum <enum name> = {enumerator,enumerator,enumerator,...};

enum robotics_subteam = {kRoboCon, kRov, kSmartCar};
std::cout<<kRoboCon<<std::endl;      //output: 0
std::cout<<kRov<<std::endl;          //output: 1
std::cout<<kSmartCar<<std::endl;     //output: 2
```

In the above case, we are actually using 0 to represent RoboCon, 1 to represent ROV, 2 to represent smart car. It will look better if we use enumerator rather than using 0,1,2 to represent our sub-teams, but at the end both are using integers to do operation.

We can also change the data type stored in enum (but must still be integer) and change the value stored in the enumerator in enum:

```
enum robotics_subteam:uint8_t = {
  kRoboCon = 2,
  kRov,      //implicitly kRov is 3 (previous +1)
  kSmartCar = 5
}
```

If there is multiple enum having same enumerator, you need to add `<enum_name>::` before the enumerator. For example:

```
std::cout<<robotics_subteam::kSmartCar;
```

See details in namespace.

Enum struct

A enum struct is just a enum with scope, which mean you always need `::` to access the enumerators, and there is no other difference.

```
enum struct robotics_subteam = {kRoboCon, kRov, kSmartCar};
std::cout<<kSmartCar<<std::endl;         //GG
std::cout<<robotics_subteam::kSmartCar<<std::endl;          //output: 1
```

Struct

Struct is a data structure which can store multiple data type in one, including function. In practice, we use capital letters to define struct and class (class will talk later). If you need to access elements inside the struct, you need to use dot.

```
//define a struct Foo, and Foo will become a new data type for you
struct Human{
  int age = 0;           //the items are seperated with semicolon
  void PrintAge(){
    std::cout<<age<<std::endl;  //variable inside struct can be access without dot (.) when it
is accessed iside its own struct
  }
};  //don't forget this semicolon

int main(){
  Human dipsy;       //now you have a new Human, and its name is dipsy
  dipsy.PrintAge(); //tell dipsy to print his age, output 0
  dipsy.age = 18;   //set dipsy's age to 18
  dipsy.PrintAge(); //tell dipsy to print his age, output 18
  return 0;
}
```

If the struct does not contain default value for the variable, you can declare it using `{}`. The above example have default value for age, so it can't declare it using `{}`. The below one is a valid one.

```
struct Human{
  int age;
  float GPA;
  char* name;
  bool inRelationship;
  void Greet(){
    std::cout<<"Hello, my name is "<<name<<"!"<<std::endl;
  }
};

//declare Human peter, assigning value according to its sequence
//age=18, GPA=4.0, name=peter, inRelationship=false
Human peter = {18,4.0,"peter",false};
int main(){
  peter.Greet();    //Hello, my name is peter!
  return 0;
}
```

For struct pointer, we use `->` instead of `.` when we are dealing elements inside the struct pointee. For example,

```
struct Human{
  int age;
  float GPA;
  char* name;
  bool inRelationship;
  void Greet(){
    std::cout<<"Hello, my name is "<<name<<"!"<<std::endl;
  }
};

//declare a human pointer pointing to nothing
Human* pHuman = nullptr;

void PlayedOneYearOfSmartCar(){
  pHuman->age++;
  pHuman->GPA = 4.3;
  pHuman->inRelationship = true;
}

int main(){
  Human peter = {18,4.0,"peter",false};
  pHuman = &peter;  //let pHuman pointing to peter
  std::cout<<peter.inRelationship<<std::endl;//this prints false
  PlayedOneYearOfSmartCar();
  std::cout<<peter.inRelationship<<std::endl;//now it is true
  return 0;
}
```

Now you can see the power of playing one year smart car and the `->` operator~

Namespace

Namespace is a space to place variables. Inside a namespace, the name of variables, function, enum, struct, class must be unique, while across namespace, there can be variables having same name.

To access things inside a namespace, we use `::` , `<namespace_name>::<thing's_name>`

```cpp
#include <iostream>

int y;
float y;    //invalid

void foo(){
  //anything
}

int foo(){  //invalid
  //anything
}


namespace N{
  int a;
  int b;
  void bar(){
    //anything
  }
}

namespace n{
  int a;    //valid
  float b;  //valid
  int bar(){    //valid
    //anything
  }
}

int main(){
  N::a = 10;    //accessing tha a in namespace N
  n::a = 20;    //accessing tha a in namespace n
  std::cout<<N::a<<" "<<n::b<<endl; //10 20
  return 0;
}
```

using and using namespace

`using` is a wonderful thing for lazy programmers to type less `::` to access things in namespace.

`using namespace` makes visible all the names of the namespace, instead stating `using` on a specific object of the namespace makes only that object visible. Visible means no need `::`.

```cpp
#include <iostream>

namespace foo{
  struct A{
    int x,y,z;
  };
  struct B{
    int x,y,z;
  };
}

namespace bar{
  struct C{
    int x,y,z;
  };
  struct D{
    int x,y,z;
  };
}

using namespace foo;//expose everything in namespace foo, which is A and B
using bar::C;   //expose C in namespace bar only

int main(){
  A a = {1,2,3};    //success, A is visible
  B b = {1,2,3};    //success, B is visible
  C c = {1,2,3};    //success, C is visible
  D d = {1,2,3};    //fail, D is not visible
  return 0;
}
```

Class

Class is something similar with struct, and is the core thing in OOP. The specific properties of an object (variables) are called attributes, and capabilities of an object (functions) are called methods. You can use it directly like a struct, but there is no `{}` style declaration and be aware of `public` and `private`.

- things which are `public` can be accessed inside and outside the class
- things which are `private` can only be accessed inside the class

```
#includes <iostream>
using namespace std;
class Foo{
  //default is private
  int x = 2;

  public:   //below this is public
  float y = 4.3;
  int GetX(){
    return x;
  }
  void Troll(){
    for(int i=0; i<x; i++){
      cout<<Troll<<endl;
    }
  }

  private:  //below this is private again
  bool z;
};

int main(){
  Foo foo;
  cout<<foo.x<<endl;     //GG, x is private
  cout<<foo.GetX()<<endl;//success, GetX() is public, and GetX is inside class so it can access
x which is private inside class
  cout<<foo.y<<endl;     //success, y is public
}
```

It is suggested to keep variables private and make getter and setter function for variables to prevent memory leak, and that is why `public:` and `private:` exist.

Separation of Code

Normally we separate the prototype and implementation of a class into a header file and a source file. It is a good practice to reduce the building time of code as the compiler can just compile the files which have been changed but not the whole bunch of code.

Taking the previous class as an example to show separation of code, we put the prototype in `foo.h` and implementation in `foo.cpp`, and the `main.cpp` will be clean.

```
//foo.h
#ifndef FOO_H   //header guard
#define FOO_H
class Foo{
  public:
  float y = 4.3;
  int GetX(){return x;} //usually function with only 1 line we keep in header file
  void Troll();
  private:
  int x = 2;
  bool z;
};
```

```
//foo.cpp
#include "foo.h"
#include <iostream>
using namespace std;
void Foo::Torll(){           //you need <class_name>::<method_name>
  for(int i=0; i<x; i++){
    cout<<Troll<<endl;
  }
}
```

```
//main.cpp
#include <iostream>
#include "foo.h"
int main(){
  Foo foo;
  cout<<foo.x<<endl;    //GG, x is private
  cout<<foo.GetX()<<endl;//success, GetX() is public, and GetX is inside class so it can access
x which is private inside class
  cout<<foo.y<<endl;    //success, y is public
}
```

Constructor

Constructor is the function which will run once the object is created. It is have the same name as object. Constructor must be public and no return type.

```
class Rect{
  public:
  Rect(int x, int y, int w, int h){
    m_x = x; m_y = y; m_w = w; m_h = h;
  }

  private:
  int m_x, m_y, m_w, m_h;
}
```

Special Constructors :

same class can have multiple constructors (function overload)

- Default constructor : constructor don't need to supply arguments (or all have default values)
- Conversion constructor : only need to supply one argument, others may be default parameters)
- Copy constructor : with only one parameter that has the same type as the object

Member Initialization List

It is the easiest way to initialize the values of variable inside the object. It initializes variables by placing a list behind the constructor. `Constructor(int param1, float param2):var1(param1), var2(param2)`. Why we always use member initialization list instead of something like previous topic?

- It is the only way to initialize a `const`
- It is the only way to initialize a member object with no default constructor (constructor with no argument needed to give).
- Execution efficiency as member initialization can be compiled more nicely.
- Parameters of constructor can have same name as object attribute.

```
//this class do not have default parameter
class A{
public:
    A(int x):x(x) {}    //this is declare x(class A) to have value as x(parameter), valid
    int x;
};

class B{
public:
  // 'a' and 'y' MUST be initialized in an initializer list; here will call constructor A(int) directly, and y will initialize with value as 2
    B() : a(3), y(2){
  //you cannot place a(3) inside {}, because a shall have been constructed when entering the {} as a is inside the root scope of class B. However there is no default constructor to contruct a before entering the {}.
  //you cannot change value of y here also because the value of y has alread fixed when entering the {}
    }

private:
    A a;    //this will try to call default constructor of A, however there isn't,
             //the program will expect it initialized in member initialization list
    const int y;    //const can never be changed after initialization (declaration)
};
```

reference:https://stackoverflow.com/questions/926752/why-should-i-prefer-to-use-member-initialization-list

Note : be aware of the initialization sequence of variable, as this may cause a difference. Variable are initialized using the sequence you wrote it in the prototype, but not using the sequence in the member initialization list. For example:

```
class Human {
    bool m_can_get_married;
    int  m_age;
public:
    Human(int age)
     : m_age(age), m_can_get_married(m_age >= 18 ? true : false)
    {}
};
```

Although `m_can_get_married` is behind `m_age` in the initializing list, it is before `m_age` in the prototype definition, so it is initialized before `m_age`, however its value depends on the value of `m_age`, there will be a logical bug. You will be forever alone if you do programming like this.

this pointer

this pointer is a pointer pointing to itself, useful when you need to separate variables of same name.

```
class Rect{
  public:
  Square(int length, int width):length(length),width(width){}
  void SetLength(int length){
    length = length;             //WTF are you doing?
  }
  void SetWidth(int width){
    this->width = width;         //this is actually changing the width
  }
  private:
  int length, width;
}
```

So for a good code practice, we add `m_` before the member of class to state it is the member of class, not the parameters of functions.

```
class Foo{
  public:
  Foo(int bar)  //WTF are you doing?
  }
  void SetX(int bar){
    m_bar = bar;
  }
  private:
  int m_bar;
}
```

Standard Library

All type in standard library are in standard namespace.

Vector

Vector is a standard container similar to array, but its length and memory are managed automatically. You need to include `vector` to use vector. You can treat it like an array using `[]` and `=` operator. You can use `push_back()` or `emplace_back()` to append elements at the back while lengthen the array, and `pop_back()` to get and remove the last element. You can use `size()` to get the length of vector.

The difference between emplace and push is that emplace will construct the object while pushing.

```
#include <iostream>
#include <vector>
using namespace std;

class Foo{
  int x, y;
  public:
  Foo(int x, int y):x(x),y(y){}
}

vector<int> myIntVec = {1,2,3,4};   //You need to tell compiler what variable type is this
vector handling by putting the variable into <> behind vector
vector<Foo> myFooVec;
int main(){
    myIntVec.push_back(6);  //{1,2,3,4,6}
    myIntVec[4] = 5;         //{1,2,3,4,5}
    cout<<myIntVec.size();  //5
    myFooVec.push_back(Foo(1,2));   //{Foo(1,2)}
    myFooVec.emplace_back(3,4);     //{Foo(1,2),Foo(3,4)}
}
```

Use of `auto` Type

`auto` is a variable type which the compiler will automatically determine what is the best variable type for it. For example:

```
vector<int> myIntVec = {1,2,3,4};
auto myInt = myIntVec[2];   //compiler will choose the best variable type for it, which may be
int

//for each entry in myIntVec
for(auto& entry: myIntVec){ //better to use reference or changes on entry will not affect
myIntVec
    cout<<entry;
}
```

Iterator

Iterator, aka cursor, pointing an element in vector. The iterator (cursor) can refer to the starting element of vector by `begin()` or element after the last element of vector by `end()`.

Iterator	position
`myVector.begin()`	`{ (0) , 1 , 2 , 3 , 4 , 5 , 6 , 7 , 8 , 9 ,,,,}`
`myVector.begin()+1`	`{ 0 , (1) , 2 , 3 , 4 , 5 , 6 , 7 , 8 , 9 ,,,,}`
`myVector.end()`	`{ 0 , 1 , 2 , 3 , 4 , 5 , 6 , 7 , 8 , 9 ,(),,,}`
`myVector.end()-1`	`{ 0 , 1 , 2 , 3 , 4 , 5 , 6 , 7 , 8 , (9) ,,,,}`

```
//http://www.cplusplus.com/reference/vector/vector/begin/
#include <iostream>
#include <vector>

int main ()
{
  std::vector<int> myvector;
  for (int i=1; i<=5; i++) myvector.push_back(i);

  std::cout << "myvector contains:";
  for (std::vector<int>::iterator it = myvector.begin() ; it != myvector.end(); ++it)
    std::cout << ' ' << *it;
  std::cout << '\n';

  return 0;
}
```

Iterator is needed for `insert()` and `erase()` . Element at the back can be iterated faster $O(1)$ while middle is slower $O(n)$.

`insert()` will insert the entry at the cursor position and move entries at or behind the cursor position backward.

`erase()` will remove the entry at the cursor position.

```
#include <iostream>
#include <vector>
using namespace std;

vector<int> myVec = {1,2,3,4};
int main(){
  myVec.insert(myVec.begin(),0);     //{0,1,2,3,4}
  myVec.insert(myVec.end(),5);       //{0,1,2,3,4,5}
  myVec.erase(myVec.end()-2);        //{0,1,2,3,5}
  return 0;
}
```

Pair

Pair is a standard data structure containing two variable: `first` and `second`. No header need to be included.

```
#include <iostrea>
using namespace std;
pair<int,int> myPair = {1,2};    //you need to define the variable types inside pair
int main(){
  cout<<myPair.first<<","<<myPair.second<<endl;
  return 0;
}
```