OOP

CH1

- `a*= b+c;` ⇒ `a = a*(b+c);`
- C ⇒ `double a = (double) b;`
  C++ ⇒ `double a = static_cast<double>(b);`
- Post-increment : `a++` ⇒ use current value. THEN +1
  Pre-increment : `++a` ⇒ +1 first. THEN use new value

CH2

- `a+=b=c;` ⇒ `(a+=(b=c));`
  `a*b/c*d;` ⇒ `(((a*b)/c)*d);`
- `if((x>=0)&&(y>1))`
  if x is negative, they y will NOT be examined.
  `if((x>=0)||(y>1))`
  if x is nonnegative, then y will NOT be examined.

> △ `y=9`
> `if((--x)&&(--y))`
> ⇒ If x is 1 then y still 9; otherwise y will be 8

- `int q = i>>4;` // $q = i / 2^4$
  `int q = i<<2;` // $q = i * 2^2$
- **Bitwise AND &**
  `int i=53;`
  `int r= i & 0xf;`
  // $53_{10} = 110101_2$
  $0xf = 15_{10} = 001111_2$
  $r = 000101_2 = 5_{10}$
- **Bitwise OR |**
  `int i=53;`
  `int r= i | 0xb;`
  // $53_{10} = 110101_2$
  $0xb = b_{10} = 001011_2$
  $r = 110111_2 = 55_{10}$
- **Bitwise exclusive OR ^** (找不同)
  `int i=53;`
  `int r= i ^ 0xf;`
  // $r = 111010_2 = 58_{10}$

CH3

- 0.0 ~ 1.0 rand() / static_cast<double>(RAND_MAX)
  1 ~ b rand() % b + 1
- formal parameter : in function declaration and definition
  actual argument : in function call

CH4

- call-by-value func(a,b) void func(int x, int y){...}
- call-by-pointer-value func(&a, &b) void func(int*x, int*y)
- call-by-reference func(a,b) void func(int &x, int &y){...}
  → reference MUST be initialized
- call-by-constant-reference void func(const int&x, const int&y)
- Order of Mixed Parameter Lists :

  void func(int& a, int b, double& c)

  a: int (passed by reference)
  b: int (passed by value) can be variable, constant...
  c: double (passed by reference)

- Overloading : same name but different parameter lists
- Default argument : void func(int a, int b=10)
  默認
  ⇒ func(30) = fun(30,10)

  → 只能在參數列尾巴

  ① void a(int, int=0, char*=0);
  ② void b(char*, int=0, int=0);
  ③ void c(int, int=0, char*=0);

CH5

- Array type array_name [size]
  ↓
  must be CONSTANT

CH6

< array : collection of same type
structure : collection of different types

1. must define struct first

```
struct name
{
  int a;
  double b;
};
```
↳ No memory

2. declare variable
   name x;

3. x.a → int
   x.b → double

```
void func(S1 p1, S1& p2, S1*p3, const S1& p4, const S1*p5);
void f(){
  S1 a1, a2, a3, a4, a5;
   :
  func(a1, a2, &a3, a4, &a5);
}
```

call-by-value → avoid : BIG
call-by-pointer-value
call-by-reference

- Class

1.
```
class name{
public:
        int a;          → access specifier
        void output();  → data member declaration
                        → member function declaration
};
```

2. name today, birthday;

3. today.a
   birthday.a
   today.output()
   > different

→ function definition (must specify the class it belongs to)

```
void name :: output() { ... }
```
↓ scope resolution operator

< public : interface to the outside world (Member function) usually
private : private to this class only (Data member)

CH7

· Constructors (ctors) 建構子

→ ctor's name MUST be SAME as class name

→ No return type

Declaration

```
class DayofYear {
public:
    DayofYear(int month, int day);
private:
    int monthvalue;
    int dayvalue;
};
```

< DayofYear();

↓

Defalt ctor ⇒ 1/1

Calling

```
void f(){
    DayofYear date1(7,4), date2(3,6);
}
```

Definition

```
DayofYear::DayofYear(int month, int day)
    : monthvalue(month), dayvalue(day)
{ /* can be empty */ }
```

→ member initialization list
data member initialized here

· Copy Constructor ① variable initialization ② argument passing ③ value return

```
DayofYear::DayofYear(const DayofYear& src)
  : month(src.month), day(src.day) { }
void f(){
    DayofYear date2(date1);
    DayofYear date3 = date2;
    date3 = date1;
}
```

CH8

CH9

- **C-string** `#include <cstring>`
  - → null character '\0' always at the end
    - ↳ end of string
  - < char s[4] = "abc";
    char s[3] = {'a','b','c'};
  - assignment : char* strcpy (char*dest, const char* src);
  - comparison : int strcmp (const char* str1, const char* str2);
    - str1 - str2 ⊖ ⇒ str1 < str2 0 ⇒ str1 == str2 ⊕ ⇒ str1 > str2
  - string length : size_t strlen (const char*);
  - Concatenate 連接 : char* strcat (char*dest, const char*src);
  - input string w/whitespaces : getline (char*s, streamize n)

```
char a[80];
cin.getline (a,80);
```

- **Class string** `#include <string>`

```
int main{
   string phrase;
   string adj("friend"), noun("ants");
   string wish = "Bon appetite!";
   phrase = "I love " + adj + " " + noun;
   cout << phrase; // I love friend ants
}
```

  - assignment : s3 = "Hello Mom!"
  - concatenate : s3 = s1 + s2
  - input string w/whitespaces :

```
string line;
getline (cin, line);
```

Quiz 1

Name:_____(20%) Student ID:_____(40%)

1. (40%, 4% off for each error)

Please fill in the table with the value of each variable after executing the corresponding line of the following code. (Fill in "X" if the value of a variable is unknown, and the latest previous values if the corresponding line is not executed.) (pow(x, y) = x^y)

a	b	c	d		# include <cmath>
1	2	X		line1	int a = 1, b = 2, c;
			0	line2	int d = a / b + static_cast<float>(a) / b;
1				line3	if ((a % b == 1) || (a = 3))
2		3	4	line4	d = (c = pow(b, ++a))--;
			7	line5	d = ((16 >> 1) > 10) ? 6 : 7;
	-1	1		line6	b += -1 * 3; c = 1;
	1		0	line7	d = static_cast<int>((b *= -2 + 1) != c);

Quiz 2

Name:_____(20%) Student ID:_____(40%)

1. (25%) True and False Test

(a) (○) In function calls, C++ use positional argument mapping.

(b) (✕) Multiple(Repeated) function declarations will always cause compilation error.

(c) (✕) Function cannot call itself.

(d) (○) In general, inline function makes executable larger.

(e) (○) “int arr[10]; arr[-2] = 0” will not cause compilation error. (題目少一個分號，送分)

2. (15%) Explain the reason if the following groups of function declarations will cause compilation error, or write “No error.” if they won’t.

Questions	Answers
(a) void f(int x, char y); int f(int x, char y);	(a)
(b) float g(int x, float y=2.31, char z);	(b)
(c) void h(int x); void h();	(c)

(a) Function overloading does not depend on return type.
(Function names and parameter lists are the same，so compiler does not know which function to call.)
P.S.
Difference on the number of parameters, the data type of parameters, and the order of parameters does matter.

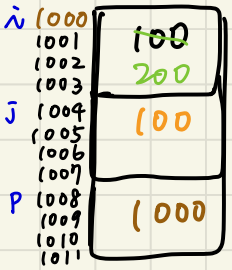
(b) Non-default arguments cannot follow with default arguments.

(c) No error.

CH10

- **Pointer**

```
int i = 100;
int *p = &i; // its value is a memory address where int at.
             // p points to int or p is a pointer to int
int **pp = &p;
int j = *p; //j = i
*p = 200; //i = 200
```



→ pointer value is an address
→ address value is an integer

- **Pointer assignment**

```
int i = 100, j = 200, *pi = &i, *pj = &j;
*pi = *pj; // i = j ⇒ i = 200
pi = pj; // pi points to what pj points to
*pi = 300; // j = 300
```

- **call-by-value**

```
int i = 99
int *p = &i;
cout << *p; //99
sneaky(p);
cout << *p << i; //99 99
void sneaky(int *temp){
   cout << *temp; //99
   *temp = 99;
}
```

• Constant Pointer & Pointer to Constant

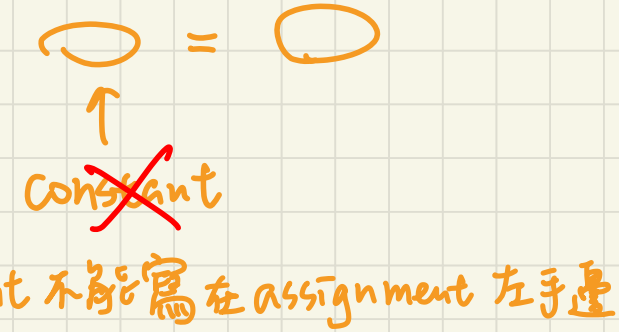
```
const int ci = 0, cj = 10;
const int *pci = &ci; // pci is a pointer to constant int ci
j = *pci; // ok
*pci = 50; // error, *pci is a constant int
pci = &j; // ok
pci = &cj; // ok, pci now points to constant int cj
           // pci's value can be changed (指向的address)
```

pci: variable

```
int * const cpi = &i; // cpi is a constant pointer to int
int * const cpj = &cj; // error, cpj can't point to constant int
j = *cpi; // ok
*cpi = 40; // ok
cpi = &j;
cpi = &ci; // error, cpi is a constant pointer 不能指別人
```

i可改，cpi不能改

```
const int *const cpci = &ci; // cpci is a constant pointer to constant int
const int *const cpcj = &j; // ok, 可以讀j的值，但不能改
j = *cpci; // ok
*cpci = 50; // error, *cpci is a constant int
cpci = &cj;
cpci = &j; // error, cpci is a constant pointer
```



constant 不能寫在 assignment 左手邊

- **Array name → a constant pointer**

```
void func1(int arr[]);
void func2(int *arr);
void f(){
    int arr1[10], arr2[10];
    int *pi = arr1; // ok, arr1 is actually a constant pointer to int
    pi[3] = 100; // ok, arr1[3] = 100
    pi = arr2; //ok
    pi[3] = 200; //ok, arr2[3] = 200
    func1(arr1); func2(arr1); func2(pi); func1(pi); //ok
    arr1 = arr2; //error, arr1 is a constant pointer
}
```

- **Pointer Arithmetic**

```
int ia[100], *pi1 = ia; *pi2 = &ia[4];   // ia ⇔ ia[0]
cout << pi1 << pi2; // 0x28fa80  0x28fa90  差16 bytes (int 1個4 bytes)
int x = pi2 - pi1; //4 (差4格)
int y = pi1 - pi2; //-4
ia[n]=30 ⇔ pi1[n] ⇔ *(pi1+n) ⇔ *(n+pi1) ⇔ *(ia+n)   // *(n+pi1): not preferred
*++pi1 = *pi2++ // ia[1] = ia[4] and then pi2 points to ia[5]
pi1 += 20; // pi1 points to ia[21]      pi1 points to ia[1]
*(1-pi1) / *(pi*3) / *(pi/4) ⇒ meaningless
int arr1[100], arr2[100], *pi = &arr1[10], *pj = &arr1[50]; double arr3[100];
int z = pj-pi; // 50-10=40
z = pi-pj; // 10-50=-40
z = pi-arr2; // ok, but meaningless!
z = pi-arr3; // error, different kinds of pointers
int sum = pi+pj; // error, meaningless! same for *, /, % ...
```

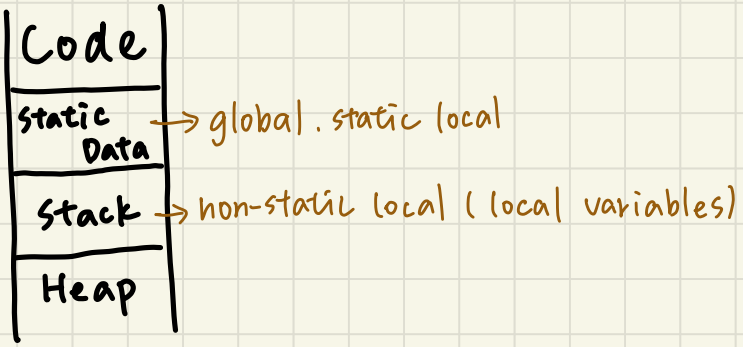
• Dynamic Memory → Heap

```
int *p = new int (5);      // *p = 5, 可不write
        ⋮
delete p;
```

new ① 跟heap要 4 bytes memory空間
② 初始化為 5
③ assign給 p

```
int *s = new int [num];
for(int i=0; i<num; ++i)
    cin >> s[i];
delete [] s;
```

Memory leak：借完記憶体，突然改变 pointer的位置，就無法釋放原本記憶体
Dangling Pointer：已經釋放記憶体，又去改值 (*p=1) ∴可能改到別的地方的值



• Typedef → Not a new type

```
typedef int* Pint; // Pint is just an alias for int*
Pint p1, p2;       // int* p1, *p2;
typedef int int32; // size of (int) = 4
typedef long int32; // size of (int) = 2, size of (long) = 4
```

• Destructors → prevent memory leak, only one dtor

```
Class IntArr{
   int size, *arr;
public:
   IntArr(int sz): size(sz){arr = new int[sz];}
   ~IntArr();
};
IntArr::~IntArr(){ // no return type & parameters
    delete[]arr;
}
```

= ~IntArr(){delete[]arr;}

CH11

- **Linkage**
  → defined once . declared many times (consistent)

```
//file1.cpp
int x;          // def. in x=0
int f(){return x;}   // def
int b=1;        // def
extern int c;   // dec
```

```
//file2.cpp
int x=1;            // def of x
int g(){return f();}  // def of g
extern double b;    // dec
extern int c;       // dec
```

link error : x is defined twice
→ compile doesn't error b is declared twice with different types
c is declared twice but not defined

compilation error : in file2, f() is used before declared

< external linkage : 可以使用於多個files (global scope)
internal linkage : 只能在這個file ex. const, typedef

- ODR (One-Definition Rule): must be defined <u>once</u>

- **Conditional compilation**

```
#ifndef ABC
#define ABC
//...
#endif
```

⇒ 不會defined twice

- **Namespace**

  local → class → namspace → global

```
void h();
namespace X {
  void g();
  namespace Y { void f(); void ff(); }
}
void X::Y::ff(){ f(); g(); h(); } //ok, X::Y::f(), X::g(), ::h()
void X::g(){
   f();      // error, no f() in X
   Y::f();   // ok
}
void h(){
   f()       // error, no global f()
   Y::f();   // error, no global Y
   X::f();   // error, no f() in X
   X::Y::f(); // ok
}
```

- **Using delarations**

```
namespace A{
  void func1();
  using B::func2;
}   // B::func2 now in A's scope
```

- **Using directives**

```
namespace A{
  void func1();
  using namespace B;
}   // mak ALL from B accessible in A
```

- **Unnamed Namespace**

```
namespace { No name
  int a;
  void f();
}
```

⇒ a and f have internal linkage

no way to access outside this file

CH12

```
cout.setf(ios_base::dec, ios_base::basefield); // 1234
cout.setf(ios_base::oct, ios_base::basefield); // 2322
cout.setf(ios_base::hex, ios_base::basefield); // 4d2 = 4x256+13x16+2
cout.setf(ios_base::showbase); 0x 4d2
cout.unsetf(ios_base::showbase); 4d
cout.setf(ios_base::scientific, ios_base::floatfield); // 1.234568 e+003   (6 → after 小數點)
cout.setf(ios_base::fixed, ios_base::floatfield); // 1234.567890   (6)
cout.unsetf(ios_base::floatfield); // 1234.57   (6 → # of digits)

cout.width(4); cout.fill('#'); "ab" // ##ab
cout.width(4); "abcdef" // abcdef
cout.width(0); "abcdef" // abcdef
cout.width(4); cout<<12<<':'<<34 // ##12:34   ∵ width() 只有一次性
cout.width(6); cout.setf(ios_base::left, ios_base::adjustfield); "ab"
                                                  // ab####

bool abc{false};
cout<<abc; // 0
cout.setf(ios_base::boolalpha);
cout<<abc; // false

cout.setf(ios-base::showpoint); 12.0 // 12.0000
cout.setf(ios-base::showpos); 12.0 // +12.0000
                              positive
```

- **File Open and Close**

```
method1 : ifstream ifs("filename");
          ofstream ofs("filename");

method2 : ifstream ifs; ifs.open("filename");
          ofstream ofs; ofs.open("filename");

ifs.close();
ofs.close();
```

CH14

· Inheritance

→ a derived class can NOT access private members of base class

```
Employee::Employee(const string&s, short d)
: familyname(s), department(d) {…}

Manager::Manager(string&s, short d, short lvl)
: Employee(s, d), level(lvl) {…}
```

→ ctor/dtor/copy ctor/copy assignment operator are never inherited

```
struct A{
  A(){cout<<"ctor A"<<endl;}
  ~A(){cout<<"dtor A"<<endl;}
};
struct B{
  B(){cout<<"ctor B"<<endl;}
  ~B(){cout<<"dtor B"<<endl;}
};
struct C: public B{
  A a;
  C(){cout<<"ctor C"<<endl;}
  ~C(){cout<<"dtor C"<<endl;}
};
int main(){
  C c;
  return 0;
}
```

output:

```
ctor B
ctor A
ctor C
dtor C
dtor B
dtor A
```

· Protected

```
Class B{
  int a;
protected:
  void f();
public:
  void h();
};
class D: public B{
public:
  void g();
};
void D::g(){
  a=1; // error, B's private
  f(); // ok
  h(); // ok
}
void func(B& b){ // global function
  b.a=1 // error, B's private
  b.f(); // error, B's protected
  b.h(); // ok
}
```

CH15

• Non-Virtual Function

```
class B{                    class D:public B{
public:                     public:
   void mf();                  void mf(); // redefine mf();
};                          };

void f(){
  B b, *pB=&b; D d, *pD=&d;
  b.mf();    // Statically binding ⇒ b is of type B ⇒ call B::mf()
  d.mf();    // Statically binding ⇒ d is of type D ⇒ call D::mf()
  pB->mf();  // Statically binding ⇒ pB is of type B* ⇒ call B::mf()
  pD->mf();  // Statically binding ⇒ pD is of type D* ⇒ call D::mf()
  pB = &d;   // ok, D is derived from B
  pB->mf();  // Statically binding ⇒ pB is of type B* ⇒ call B::mf()
             (though pB actually points to d (an object of type D)
```

• Virtual Functions

```
class B{                    class D:public B{
public:                     public:
   virtual void mf();          void mf(); // override B::mf();
};                          };

void f(){
  B b, *pB=&b; D d, *pD=&d;
  b.mf();    // Statically binding ⇒ b is of type B ⇒ call B::mf()
  d.mf();    // Statically binding ⇒ d is of type D ⇒ call D::mf()
  pB->mf();  // dynamically binding ⇒ pB actually points to b ⇒ call B::mf()
  pD->mf();  // dynamically binding ⇒ pD actually points to d ⇒ call D::mf()
  pB = &d;   // ok, D is derived from B
  pB->mf();  // dynamically binding ⇒ pB actually points to d ⇒ call D::mf()
```

• Abstract Base Class (ABC)

→ A derived class becomes concrete once it overrides ALL inherited pure virtual functions

• Pure Virtual Functions → only declaration, no definition

```
class Base{ public:
   virtual void mf1()=0;      // pure virtual function
   virtual void mf1(int);     // overloaded simple virtual function
   virtual void mf2();        // simple virtual function
   void mf3();                // non-virtual member func
   void mf3(double); };       // overloaded non-virtual member func
class Derived: public Base { public:
   virtual void mf1();        // override Base::mf1
   void mf3(); };             // redefine mf3
void f(){
  Derived d;
  d.mf1();      // ok. call Derived::mf1()
  d.mf1(10);    // error! Derived::mf1 hides Base::mf1
  d.mf2();      // ok, call Base::mf2()
  d.mf3();      // ok, call Derived::mf3()
  d.mf3(10.0);} // error! Derived::mf3 hides Base::mf3
```

SOL:

```
class Derived: public Base { public:
   using Base::mf1; // make ALL things in Base named mf1 visible in Derived
   virtual void mf1();
   void mf3(); };
void f(){
  Derived d;
  d.mf1();
  d.mf1(10); // ok now! call Base::mf1(int)
  d.mf2();
  d.mf3();
  d.Base::mf3(10.0); // ok now! call Base::mf3(double)
}
```

CH18

. Exceptional Handling

```
try { //...
  throw x;
}
catch(int a) { ... }
```

```
struct Range_error{
   int d;
   Range_error(int dd): d{dd}{}
};
char num_to_hexchar(int num){
   if(num<0 || num>15)
      throw Range_error{num};
   if(num<10)
      return '0'+num;
   else
      return 'A'+num-10;
}

void g(int num){
  try {
    char c = num_to_hexchar(num);
  }
  catch(Range_error) // just type
    cerr<<"Oops! There is a range error!";
  }
}
```

```
void h(int num){
  try {
    char c = num_to_hexchar(num);
  }
  catch(Range_error x) // with name object
    cerr<<"Range error! Pass "<<x.d
        <<"to function num_to_hexchar.";
  }
```

→ can throw multiple exceptions

- Exception and Class Hierarchy