Chap.7. Constructors & other tools.

• Constructors.

Objective: To initialize the value in class. ← Mainly.

Guide line: ① Has the same name with "class name"

② Automatically called when it is born.

Example.
```
class Day of Year {
    Public:
        Day of Year (int Month, int Day): Month[month], Day[day] {"can be empty"}
    Private:
        int month;
        int day;
}
```
* usually initialize here.

• Constructor Overloading and default

⇒ As long as the data member in constructor, it can be overloading

: Default construct ⇒ the constructor that don't have data member

ex.
```
Class Day of Year {
    Day of Year (int Month, int day): month[Month], day[Day] {}
    Day of Year ( ): month[1], day[1] {}
    Day of Year (int Month=5) month[Month], day[1] {}
```
Default constructor

• Copy constructor.

Type:
```
class X {
    X ((const X &d): X1[x1], X2[x2] { }
}
```
const reference - just see, not modify.

Application: ① Initialize.: ex. Day of Year day2{day1} or Day of Year Day3 = day1;

② Return value

• Pass the parameter:

① Just see the data member ⇒ Called by Const. Reference

② Modify the data member ⇒ Called by Pointer.

• Const. member function:

⇒ If the member function that "don't modify the data member", add "const." @ the end of function.

```
class Day of Year {
    :
    void output ( ) const: {cout << month << day << endl;}
}.
```
Promise won't modify data member.

Chap. 8. Operator Overloading & Friend function

- Operator Overloading

Why: Because in "self-defined type", the compiler don't know how to process the operator like +, −, ×, = ..., so we must tell compiler how to do it.

How: ① In member function: (Operator can be treated as function & we would like to do we usually did)

```
class Complex {
    ① const Complex operator+ (const Complex& rhs) ② const
    {
        Complex result {rhs};   // copy constructor
        re = rhs.re,  im = rhs.im;
        return result }
```

① ∵ 為了讓結果不會出現在等式左邊，方便維護

② ∵ 這個函數不會改變 "caller" 的 data member 故加 const 來保護

② In non-member function

```
class Complex { ...
}

const Complex operator+ (const Complex& lhs, const Complex& rhs) {
    return Complex { lhs.re() + rhs.re(), lhs.im() + rhs.im() }
}
```

∵ non-member function ∴ have to use public function to access value.

⇒ Which one is perfered? Ans: Non-member function

Ex.

```
Complex a {1, 3};
Complex b = a + 1;   ⇒ can work in both case.
Complex C = 1 + a    ⇒ can only work in non-member function.
```

∵ In member function case, the caller must be complex, but in non-member function ⇒ compiler will help you automatically turn into complex

⇓

```
Complex C = Complex {1, 0} + a;
```

- Operator "="

```
⇒ Complex& operator= (const Complex& rhs) {
      re = rhs.re, im = rhs.im
      return *this }
```

↓ 誰 call "=" function, 我就指向誰

```
Complex& operator += (Const Complex& rhs) {
re += rhs.re, im += rhs.im
return *this }
```

use += to apply +.

```
const Complex operator+ (const Complex& lhs, : rhs)
{   complex result {lhs};
    result += rhs;
    return result
}
```

• Friend function:

⇒ Friend function isn't member function but can access the private member.

mainly used in Non-member function.

ex. ① Non friend

```
const Complex operator+(const complexd lhs, const Complexd rhs){
    return Complex(lhs.re()+rhs.re(), lhs.im()+rhs.im());
```

② friend function:

```
const Complex operator+(const Complexd lhs, const Complexd rhs)
{ return Complex{lhs.re+rhs.re, lhs.im+rhs.im};
```

⇒ don't need another function to access the data member.

• I/O in user define type.

⇒ It also has to be "Friend Function"

Output:

```
ostreamd operator<<(ostream &OS, Complexd rhs)
{ os << rhs.re << rhs.im;
```

※ return os 才能連接能夠本來的 data type 在一起.

```
  return OS
}
```

Input:

```
istreamd operator>>(istream &is, Complexd rhs)
{   is >> rhs.re >> rhs.im;
    return is
}
```

• Prefix & Postfix.

Prefix (++i):

```
Code: intd operator++()
   { return *this+=1}
```

Post-fix (i++)

```
code:
 intd operator++(int)
 { int old{*this}
   *this+=1;
   return old
 }
```

Can only do one time

i++(i++)++

return old ⇒ old++

⇒ i still remain

Chap. 9. String.

- In C:

  Use Char array to show the string.

  ⇒ But the cons of char Array is that the name of char arr. is the "Const Pointer" that points to the first elements of array ⇒ so it can't be on the left side of "=".

- input:

No matter in C (Char. Array) or C++ (string), when the input encounter space, tab or "\n", it will automatically stop.

So we will use the function getline(cin, "string name") to input the full sentense.

- In C++: We use string to replace char array

⇒ so that it is the STL class that can process. lots of operator like "+", "=" ....

Command Line Arguments:

We can send the Arguments from keyboard to main function.

ex. copy foo.txt foo2.txt (keyboard)

```
int main (int argc, char **argv)
```

argc specified how many arguments are supply

**argv specified the name of the file.

argv[0]: copy (the name of the program that is invoked)

argv[1]: foo.txt (the name of first parameter)

argv[2]: foo2.txt (~ second ~)

Chap.10 Pointer and Dynamic memory

- Pointer def.

⇒ The variable that store the "Address" of the objects.

ex. `int* pi = &i`

Data type: pointer the store int's address.

- Meaning of. "&" and "*"

⇒ "&" is to get the address of the objects.

"*" ① "Declaration time" means the variable. will store the address.

② "Definition time" will get the "value of the object that you point."

ex. `int i;`

`int *pi = &i;`

`*pi = 6;` (i=6)

means variable store address.

`*pi = i`

- Constant pointer vs. Pointer to constant.

(i). const. pointer. : Pointer that points to "constant variable"

ex. `const int *cpi, int *pi;`

```
int i, const int ci;
cpi = &ci ; (O)
cpi = &i ; (O)
pi = &ci ; (X)
```

(ii). Pointer to const. : Pointer points to "fixed object." (指向的对象不变)

```
int* const pi ;
int i , j ;
pi = &i ; (O)
*pi = j ; (O) i=j
pi = j ; (X) pi is fixed to point @ i.
```

- Array name and pointer arithmetic.

"Array name can be seened as a constant pointer that points to the first element in Array"

ex. `int Arr[100] ; int *pi; int j;`

`pi = &Arr[0] ;`

pi[0], // = Arr[0] (o)

*Arr = 5, // Arr[0] = 5 (o)

Arr = dj; (X) ∵ Arr is a const pointer.

⇒ can't be on left side.

Arr[0]
Arr[1]
Arr[2]
Arr[3]
Arr[4]
Arr[5]
Arr[6]

- Arithmetic (the code must have meaning)

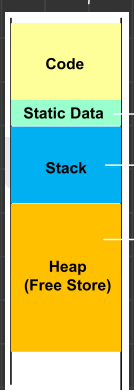
ex. *(Arr+5) = 6; // Arr[5] = 6.

int *pi1 = &Arr[0], *pi2 = &Arr[3];

pi1 - pi2; (指標差)

- Dynamic memory.

(i) Principle.



Static Data → Static local variable & Global variable

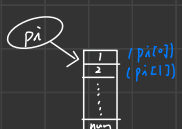
Stack → Local variable.

Heap (Free Store) → Dynamic variable. ⇒ when we instantly need storage @ Runtime, we'll ask this part to give us space, but we must return it. to avoid "memory leakage"

(ii). 語法.

int* pi = new int [index];

→ ask heap to borrow us the space.



delete [] pi;

→ return the borrowing space to heap.

<補充>. Object use "." to call data member

Pointer use "→" to call data member.

- Destructor:

⇒ if we borrow the space @ the time when object born, we must return before it die

ex. Class IntArr {

int size, *arr;

public:

IntArr(int sz): size[sz], arr[new int[sz]]{ };

~IntArr() { delete [] arr; }

<key point>
- We can have lots of constructor
- But only can have one destructor

• Copy constructor and Assignment operator.

⇒ If you doesn't define by yourself, the compiler will automatically make one, but it sometimes cause serious problem

Ex.

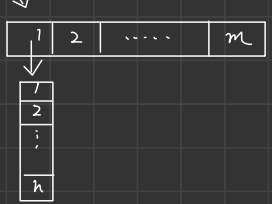
```
Class IntArr {
  int size, *arr;
  public:
    IntArr(int sz) : size(sz), arr(new int[sz]) { }
   ~IntArr() { delete [] arr; }
    #ifdef DEBUG
    IntArr& operation=(const IntArr& src);
    IntArr(const IntArr&);
    #endif
};
IntArr a(100), b(100);
b = a;  (淺 b 和 a 指向相同地方)
a[20]=100;  上行使得 b[20]=100. }
```

So we must define our own "=".

Ex.
```
IntArr& IntArr::operator=(const IntArr& src) {
    int size = src.size;
    int *ptr = new int[size];
    for(int i=0, i<size, ++i){
        ptr[size] = src.arr[size]; }
    delete [] arr;   (刪除 caller 的 Array 內容)
    arr = ptr;       (讓 arr 指向和 ptr 同向)
    return *this;
}
```

Advanced: Construct dynamic 2D Array. (m x n)

Concept:



Code:
```
int **arr = new int*[m];
for(int i=0; i<m; ++i){
    int*[i] = new int[n];}
```

Data type: dynamic array store pointer

Delete:
```
for(int i=0; i<m; ++i){
    delete [] arr[i];}
delete [] arr;
```

Recall: Array name is a constant pointer points to the first element in array.

Chap12. I/O.

• I/O Manipulator

Output I/O or True/False.

```
cout << boolalpha << noboolalpha << showbase << noshowbase
<< showpoint << noshowpoint << showpos << noshowpos
<< uppercase << nouppercase << left << right
<< dec << hex << oct << fixed << scientific << defaultfloat
<< setfill('#') << setprecision(4) << setw(8) << ... since C++11
```

Need to include <iomanip>

• File I/O.

⇒ ① We must first include. <fstream>.

② We can give some parameter to determine the input/output file's name by using: int main (int argc, char **argv)

argument count — pointed to the first letter of file

ex. Input: copy file1.txt file2.txt.

Data type

ifstream: ifs (argv[1]);
if (!ifs) { cout << can not open; }
Debug
To check if it is open successfully, if not will return "True"

ofstream: ofs (argv[2]);
if (!argv[2]) { cout << can not output; }

• Check the file is end or not.

```
ex. char next;
    ifs.get(next); (Use get() to input 1 char a time)
    while(!ifs.eof()){
        cout << next;
        ifs.get(next); }
```

• istringstream & ostringstream

Very important! Because it helps us able to not know the amount of input when we are coding!!!

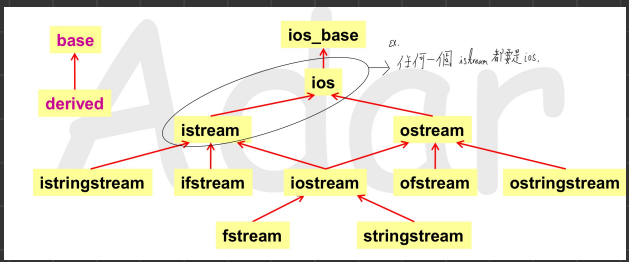
• Must include <sstream> before we use.

```
ex. #include <iostream>, <sstream>, <string>, using namespace std;
string repeator(const string& str, int times){
    ostringstream oss;
    for(int i=0; i<n; ++i){
        oss << str << " "; }
    return oss.str(); }

int main(){
    cout << repeator("Coding",3) <<endl;
    return 0; }
```

Output:
Coding! Coding! Coding!

〈The relationship between each I/O〉



- We can imagine that we want to output the result by part, so we utilize the oss to help us store the result in str by part. And finally return the oss.str() (string type.)

Input.
1 3 5 7

Output
1
3
5
7

```
#include <iostream>, <sstream>, <string>, using namespace std;
void num_per_line (const string &str) {
    istringstream iss(s);
    int num;
    while (iss >> num) cout<<num<<endl; } }
int main() {
    string input;
    getline (cin, input);
    num_per_line (input);
    return 0; }
```

Normally, it will return iss, but it isn't bool.
So compiler will use "bool operator"
, when iss don't have input, it will return 0.

- We first use getline to access a string that contains space. then we store it into iss.
then we can use the characteristic that when the input encounter space or tab, it will automatically stop.
, and it will help us to divide the input to pieces.

Chap. 14. Inheritance

- In OOP, class is used to represent a concept.

Based (generalize) Ex. Employee.
↓
Derived (specific) Ex. Manager.

Derived Is A Based → dell compiler. → use Public. inheritance. can access the data members. ① Based & ② Protected in based class.

- In public: if. Derived class doesn't re write it. , then we'll just use. based's public data member.

- ctors ⇒ Derived ctor. response to call. based's ctor.
  ① Use. based ctor. to initialize. inheritance data member.

⇒ order of ctor & dtor.

- first.
- second.
- third.

ctor. ↓ ↑ dtors.

- Types of Data member and. Inheritance

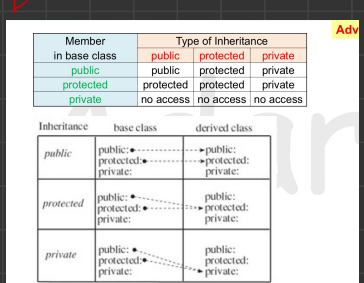
(~.) Data member.

- Public: All the member can be used.
- Protected: ① It can only be used ② member function & friend function & ③ the derived class's member function & friend function
- Private: It can only be used ^ member function & friend function

(~.) Inheritance. ⇒ also have ① Public ② Protected ③ Private inheritance.

- Representation: ① Class X: public B
  ② Class Y: protected B
  ③ Class Z: private B
- Access Control to Base classes.

Member in base class	Type of Inheritance		
	public	protected	private
public	public	protected	private
protected	protected	protected	private
private	no access	no access	no access



Chap.15 Polymorphism and virtual functions

From "Inheritance concept", we already know that "Based pointer can also points to the derived data", but if we have the same function both in "Based" and "Derived class", how can the compiler know which function we would like to use?

Example code

```
Class B.{
    void print_msg();
    ⋮ }
Class D: public B {
    void print_msg();   ⇒ Redefine
    ⋮ }
int main() {
  B b, *bp[dB]; D d, *dp[dd];
    bp→print_msg(); Call class B's
    dp→print_msg(); Call class D's
    bp[&d];   legal action: "D is a B"
    bp→print_msg(); What we want: Call class D's
                    But what actually is: Call class B's
```

Why this happen? Because the compiler just call the pointer's type function. it don't consider what's stored in it. (Statically binding)

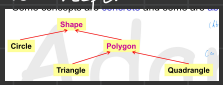
- How we let compiler know it must call the function depend on the content it stored. (Dynamically binding)

- (I) Call the function by Pointer / Reference
- (II) Use "virtual function"

⇒ Polymorphism

- Virtual function:

In OOP, we usually use Class to represent a concept.



- Shape and polygon is counted as a "Abstract class" ⇒: it don't have specific definition.
- Circle, Triangle and Quadrangle, is counted as "concrete class", because it has specific def.

Revised code

```
Class B.{
    virtual void print_msg();   add virtual @ the beginning of base function.
    ⋮ }
Class D: public B {
    void print_msg();   ⇒ Override
    ⋮ }
int main() {
  B b, *bp[dB]; D d, *dp[dd];
    bp→print_msg(); Call class B's
    dp→print_msg(); Call class D's
    bp[&d];   legal action: "D is a B"
    bp→print_msg(); Now it'll know we want to call class D's.
```

* Pure virtual function in abstract class.

⇒ In abstract class, some member function don't have specific definition, but we still need to define it. in this case we can use "Pure virtual function.

— Example code —

```
Class Shape {
    Public:
        virtual void rotate(int) = 0;      } → pure virtual function.
        virtual bool is_closed() = 0;      }
            : }
Class Circle: public shape{
        Public:
            void rotate(int);
            void bool is_closed(){ return true;}
            Circle( x,y, r):X{x}, Y{y}, Radius{r} {}
        privates
            double X, Y, Radius;}
  int main(){
     Shape s; → illegal move, you must override "All" the pure virtual function to create the object.
     Circle c(0.0, 5);
  }
```

⇒ Important usage:

```
void draw_shapes ( Shape **sarr, int size ){
    for(int i=0; i <size; ++i)
      sarr[i]→ draw(); Circle / triangle / rectangle..
  }
```

* Pure virtual vs. Simple virtual vs. non-virtual

- Pure virtual: Just to be the base of the derived function ⇒ so you must override all virtual function.
- Simple virtual: Based class has defined the function, but derived class can choose to override it or not.
- non-virtual: It suit for all based and declared class ⇒ Never change it!!!