

C++

程式語言（二）

Introduction to Programming (II)

Resource Management

Joseph Chuang-Chieh Lin

Dept. CSE, NTOU

Platform/IDE

- Dev-C++



<https://www.pngegg.com/en/search?q=Dev-C>

- Codeblocks



<https://icons8.com/icons/set/code-blocks>

- OnlineGDB (<https://www.onlinegdb.com/>)



- Real-Time Collaborative Online IDE (<https://ide.usaco.guide/>)



Textbooks (We focusing on C++11)

- *Learn C++ Programming by Refactoring* (由重構學習 C++ 程式設計). Pang-Feng Liu (劉邦鋒). NTU Press. 2023.
- *C++ Primer. 5th Edition.* Stanley B. Lippman, Josée Lajoie, Barbara E. Moo. 2019.
- *Effective C++.* Scott Meyers. O'Reilly. 2016.
- *Thinking in C++. Vol. 1: Introducing to Standard C++.* 2nd Edition. Bruce Eckel. Prentice Hall PTR. 2000.

Useful Resources

- Tutorialspoint
 - <https://www.tutorialspoint.com/cplusplus/index.htm>
 - Online C++ Compiler
- Programiz
 - <https://www.programiz.com/cpp-programming>
- LEARN C++
 - <https://www.learncpp.com/>
- MIT OpenCourseWare - Introduction to C++
 - <https://ocw.mit.edu/courses/6-096-introduction-to-c-january-iap-2011/pages/lecture-notes/>
- Learning C++ Programming
 - <https://www.programiz.com/cpp-programming>
- GeeksforGeeks
 - <https://www.geeksforgeeks.org/c-plus-plus/>



Recall for Dynamic Memory Allocation in C++

Purpose of using dynamic memory

- Properly freeing dynamic objects turns out to be a surprisingly rich source of bugs.
- Programs tend to use dynamic memory for one of three purposes:
 1. They don't know how many objects they'll need.
 2. They don't know the precise type of the objects they need.
 3. They want to share data between several objects.

new and delete?

- In C++, people are used to use `new` operator (cf., `malloc()` in C) to allocate memory and `delete` (cf., `free()` in C) to free memory allocated by `new`.
- However, using these operators to manage memory is considerably more error-prone.
- From C++ 11 and newer versions, we are encouraged to use **smart pointers** to manage dynamic objects.
 - They are safer and easier.

Smart Pointers (the `shared_ptr` class)

```
shared_ptr<string> p1;  
unique_ptr<int> p2;
```

*Actually there is also `make_unique` but it's in C++14 standard.

```
//use make_shared function  
shared_ptr<int> p3 = make_shared<int>(42);  
//42  
shared_ptr<string> p4 = make_shared<string>(10, '9');  
//9999999999  
shared_ptr<int> p5 = make_shared<int>();
```

```
//we can also use "auto"  
auto p3 = make_shared<int>(42);  
//42  
auto p4 = make_shared<string>(10, '9');  
//9999999999  
auto p5 = make_shared<int>();
```


An Example

<https://onlinegdb.com/dSS35GJ2l>

```
#include <iostream>
#include <memory>
```

```
using namespace std;
```

```
class Grade {
private:
```

```
    int math;
    int eng;
    int sum;
```

```
public:
```

```
    Grade() = default;
```

```
    Grade(int m, int e): math(m), eng(e) {};
```

```
    ~Grade() { cout << "destructor of 'Grade' works here" << endl; } ;
```

```
    void SumUp() { sum = math + eng; }
```

```
    int ShowSum() { return sum; }
```

```
};
```

```
int main()
{
    auto ptr = make_shared<Grade>(100, 90);
    ptr->SumUp();
    cout << "The total grades: "
         << ptr->ShowSum() << endl;
    return 0;
}
```

```
The total grades: 190
destructor of 'Grade' works here
```

Copying and Assigning shared_ptr

- When we copy or assign a shared_ptr, each shared_ptr keeps track of **how many** other shared_ptrs point to the same object.

```
auto p = make_shared<int>(42); // object to which p points has one user
auto q(p); // p and q point to the same object; q is a copy of p
// object to which p and q point has two users

auto r = make_shared<int>(42); // int to which r points has one user
r = q; // assign to r, making it point to a different address
// increase the use count for the object to which q points
// reduce the use count of the object to which r had pointed
// the object r had pointed to has no users; that object is
// automatically freed
cout << r.unique(); // print out whether p.use_count() is 1 or not
cout << r.use_count(); // print out number of objects sharing with r
```

More on shared_ptr

- shared_ptrs **automatically**
 - **destroy** their objects (by a destructor of the class).
 - **free** the associated memory.

```
// factory returns a shared_ptr pointing to a dynamically allocated object
shared_ptr<Foo> factory(T arg)
{
    // process arg as appropriate
    // shared_ptr will take care of deleting this memory
    return make_shared<Foo>(arg);
} // the object will be appropriately deleted with the allocated memory freed
```

```
void use_factory(T arg)
{
    shared_ptr<Foo> p = factory(arg); // use p
} // p goes out of scope; the memory to which p points is automatically freed
```

More on shared_ptr

- shared_ptrs **automatically**
 - **destroy** their objects (by a destructor of the class).
 - **free** the associated memory.

```
// factory returns a shared_ptr pointing to a dynamically allocated object
shared_ptr<Foo> factory(T arg)
{
    // process arg as appropriate
    // shared_ptr will take care of deleting this memory
    return make_shared<Foo>(arg);
}
```

```
// factory returns a shared_ptr pointing to a dynamically allocated object
shared_ptr<Foo> use_factory(T arg)
{
    shared_ptr<Foo> p = factory(arg); // use p
    return p; // reference count is incremented when we return p
} // p goes out of scope; the memory to which p points is NOT freed
```

Managing Memory Directly (new & delete)

```
int *pi = new int;  
string *ps = new string;  
int *pi = new int(1024);  
string *ps2 = new string(10, '9');  
// allocate and initialize a const int  
const int *pci = new const int(1024);  
// allocate and initialize an empty string  
const string *pcs = new const string;
```

```
int i, *pi1 = &i, *pi2 = nullptr;  
double *pd = new double(33), *pd2 = pd;  
delete i; // error: i is not a pointer  
delete pi1; // undefined: pi1 refers to a local  
delete pd; // ok  
delete pd2; // undefined: the memory pointed to by pd2 was already freed  
delete pi2; // ok: it is always ok to delete a null pointer
```

Using shared_ptrs with new

```
shared_ptr<double> p1;  
shared_ptr<int> p2(new int(42)); //direct initialization
```

Note that the following initialization is wrong:

```
shared_ptr<int> p1 = new int(42);  
//error: we must use direct initialization
```

Note that the following implicit conversion is also wrong:

```
shared_ptr<int> clone(int p) {  
    return new int(p);  
}
```

 **correction**

```
shared_ptr<int> clone(int p) {  
    return shared_ptr<int>(new int(p));  
}
```

Direct initialization

- `shared_ptr<T> (T*)` constructor is declared **explicit**.

```
shared_ptr<int> p1 = new int(42);  
// converting from int* to a shared_ptr<int>.  
// But copy-initialization only considers non-explicit  
// converting constructors.
```

- A constructor is called here:

```
shared_ptr<int> p2(new int(42));  
// the argument list: (new int(42))
```

Dynamic Arrays

```
int *pia = new int[10]; // uninitialized 10 ints
int *pia2 = new int[10](); //initialized to be 10 0's;
string *psa = new string[10]; // block of 10 empty strings
string *psa2 = new string[10](); // block of 10 empty strings
int *pia3 = new int[5]{0,1,2,3,4};
string *psa3 = new string[10]{"a", "b", string(3,'x')};
// the first three elements are initialized from given initializers
// remaining elements are value initialized
```

```
// Freeing dynamic arrays
delete [] pia;
delete [] psa;
...
```




Challenges of Resource Management

Resource Management

- Managing memory, file (handles), network connection, etc.
- **Goal:** Efficiently to prevent:
 - Memory leaks (memory not deallocated)
 - Dangling pointers (accessing deleted memory)
 - Double deletion errors.

Examples of Memory Leaks

```
void leaky() {  
    int *p = new int(16);  
    std::cout << "value= " << *p << endl;  
    // memory pointed by p is not deleted  
}  
void leaky_array() {  
    for (int i = 0; i<100; i++) {  
        arr = new int[10];  
    }  
    delete[] arr;  
    // Only last allocation is freed!  
}
```



```
void leaky() {  
    int *p = new int(16);  
    std::cout << "value= " << *p << endl;  
    delete p;  
}  
void leaky_array() {  
    for (int i = 0; i<100; i++) {  
        arr = new int[10];  
        ...  
        delete[] arr;  
    }  
}
```

Examples of Dangling Pointers

Pointers that points to memory which has been freed or deallocated.

```
void dangling() {
    int *p = new int(16);
    std::cout << "value= " << *p << endl;
    delete p;
    std::cout << "value= " << *p << endl;
}

int* getPointer() {
    int x = 37;
    return &x;
    // returning address of a
    // local variable!
}

int main() {
    int *ptr = getPointer();
    std::cout << *ptr << endl; // error!
    return 0;
}
```



```
void dangling() {
    int *p = new int(16);
    std::cout << "value= " << *p << endl;
    delete p;
    p = nullptr; // safer!
}

int* getPointer() {
    static int x = 37;
    return &x;
    // though not recommended...
}

int main() {
    int *ptr = getPointer();
    std::cout << *ptr << endl; // fine
    return 0;
}
```

Examples of Dangling Pointers

Pointers that points to memory which has been freed or deallocated.

```
void dangling() {
    int *p = new int(16);
    std::cout << "value= " << *p << endl;
    delete p;
    std::cout << "value= " << *p << endl;
}

int* getPointer() {
    int x = 37;
    return &x;
    // returning address of a
    // local variable!
}

int main() {
    int *ptr = getPointer();
    std::cout << *ptr << endl; // error!
    return 0;
}
```



```
void dangling() {
    auto p = std::make_unique<int>(16);
    std::cout << "value= " << *p << endl;
    // no need for manual delete
}

std::unique_ptr<int> getPointer() {
    return std::make_unique<int>(37);
    // Safe. No need for manual delete
}

int main() {
    auto ptr = getPointer();
    std::cout << *ptr << endl; // fine
    return 0;
}
```

Examples of Double Deletion

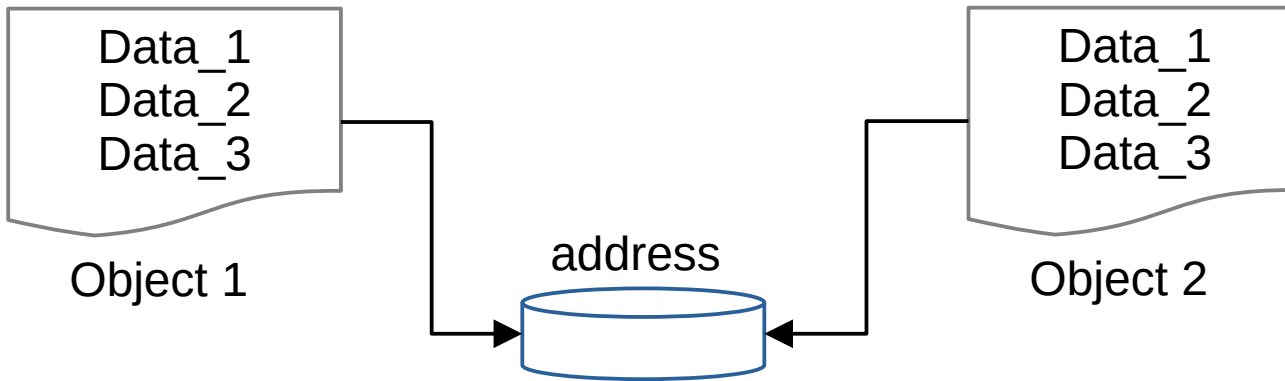
```
int main() {  
    int* ptr = new int(10);  
    delete ptr; // OK for first deletion  
    delete ptr; // undefined for the  
                // second deletion  
}
```



```
int main() {  
    int* ptr = new int(10);  
    delete ptr; // OK for first deletion  
    ptr = nullptr;  
    delete ptr; // safe now  
}
```

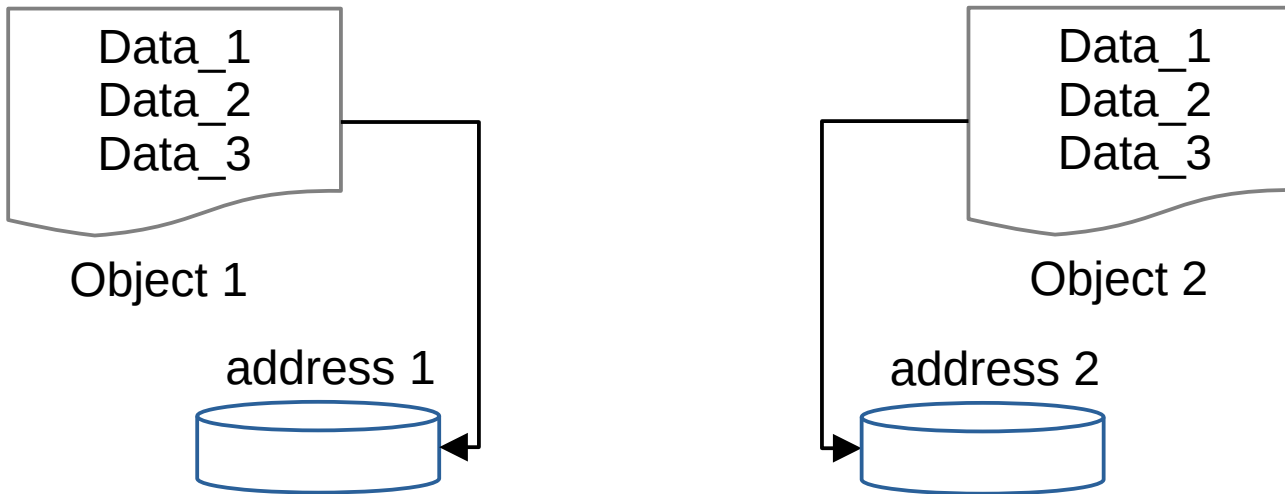
```
class ShallowCopy {  
public:  
    int* data;  
    ShallowCopy(int val) { data = new int(val); }  
    ~ShallowCopy() { delete data; } // GOOD. Destructor deletes memory.  
};  
  
int main() {  
    ShallowCopy obj1(10);  
    ShallowCopy obj2 = obj1; // NO GOOD! Shallow copy here  
                             // Both share same `data` pointer  
    return 0; // NO GOOD! Destructor called twice -> Double deletion occurs!  
}
```

Shallow Copy vs. Deep Copy



Shallow Copy

Shallow Copy vs. Deep Copy



Deep Copy

Solving Double Deletion in a Shallow Copy:

Deep Copy

```
class DeepCopy {
public:
    int* data;
    DeepCopy(int val) { data = new int(val); }
    ~DeepCopy() { delete data; }
    DeepCopy(const DeepCopy& other) { data = new int(*other.data); }
    // create a new memory copy

    DeepCopy& operator=(const DeepCopy& other) {
        if (this == &other) return *this; // self-assignment check
        delete data; // free old memory here
        data = new int(*other.data);
        return *this;
    }
};

int main() {
    DeepCopy obj1(10);
    DeepCopy obj2 = obj1; // Nice! Deep copy is applied here
    return 0; // Safe! Destructor called twice but no double deletion occurs!
}
```

Solving Double Deletion in a Shallow Copy:

Deep Copy (using smart pointers)

```
class DeepCopy {
public:
    std::unique_ptr<int> data;
    DeepCopy(int val) { data = std::make_unique<int>(val); }
    ~DeepCopy() = default; // no need for manual deletion
    DeepCopy(const DeepCopy& other) {
        data = make_unique<int>(*other.data); // create a new copy
    }
    DeepCopy& operator=(const DeepCopy& other) {
        if (this == &other) return *this; // self-assignment check
        data = make_unique<int>(*other.data);
        return *this;
    }
};

int main() {
    DeepCopy obj1(10);
    DeepCopy obj2 = obj1; // Nice! Deep copy is applied here
    return 0; // Safe! Destructor called twice but no double deletion occurs!
}
```

Rule of Three

- for Classes Managing Resources

- Whenever you design a class which uses dynamic memory, implement:

(1). Destructor

```
~className()
```

- Freeing memory.

(2). Copy Constructor


```
className(const className &)
```

- Creating deep copy.

(3). Copy Assignment Operator

```
operator=(className &)
```

- Ensuring correct copy behavior.



Discussions & Questions