物件導向程式設計

Dynamic Memory

Joseph Chuang-Chieh Lin
Dept. CSIE, Tamkang University

Platform

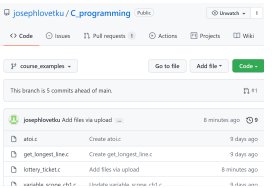
- Dev-C++

  Click here to download.

  Note: Please use this version otherwise you can't compile your programs/projects in Win10.

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

My GitHub page:
click the link here to visit.



- Other resources:
  - MIT OpenCourseWare - Introduction to C++ [link].
  - Learning C++ Programming [Programiz].
  - GeeksforGeeks [link]

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.

```
// 使用 make_shared 函式
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>();
```

```
// 前面也可以使用 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));
}
```

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;
...
```

Remark

- Using a library container (e.g., vector, see STL in the future lectures, if it's possible) is better (safer, easier, and more efficient) and even more pronounced under the new standard.

```
vector<int> v1(10); // v1 has 10 elements with value 0
vector<int> v2(10, 1); // v2 has 10 elements with value 1
vector<int> v3{1, 2, 3}; // v3 has two elements with values 1, 2, and 3
v1.push_back(9); // add 9 into the rear of v1
...
```

Class Exercise:
Try to use `new` and `delete` instead (1%)

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
```

Exercise (3%)

- https://onlinegdb.com/2oqsenisJp
- Design a **constructor** which can assign values of the data members of `Vehicle`.
- Prompt the user to input the number *n* of `vehicles`.
- **Use `new` and `delete` to construct a set of *n* vehicles.**
- Print all the vehicles with total prices and brands.

Sample input & output

```
2
Constructor works here!
100 200 300 Volswagen
Constructor works here!
200 300 400 BMW
VolswagenTotal price: 600
VolswagenTotal price: 900
destructor of 'Vehicle' works here
destructor of 'Vehicle' works here
```