Conception Objets Avancée

Smart pointers

Giuseppe Lipari

CRIStAL - Université de Lille

Outline

Outline

Dynamic memory

- Dynamic memory is useful
  - It is not necessary to pre-allocate all memory at program startup
  - While the program executes, the programmer can decide how much memory he needs, and ask the OS to allocate memory correspondingly
  - When that memory is not needed anymore, it can be released, to be re-allocated later
  - Main operators: new and delete
- However, it is the programmer responsibility to allocate / deallocate memory
  - If the programmer makes an error in allocating / deallocating memory, the program suffers from strange bugs

Memory leaks

- A typical error:

```
int function()
{
    MyClass *p = new MyClass;
    ...
    // who deletes p ?
}
```

- What happens to the allocated object?
  - If its address is stored somewhere, then eventually someone will delete it
  - However, if we do not delete it, and `p` goes out of scope, we lose the value of its address, and nobody will be able to delete it anymore
  - In the second case, the object stays allocated until the program ends
  - This bug is called memory leak

Use after free

- Another typical error it to access the object after it has been deleted

```
void function(Myclass *q) {
    // ...
    if (cond) delete q;
    // ...
}

int main() {
    MyClass *ptr = new MyClass();

    function(ptr);

    // Here ptr may be invalid !

    cout << ptr->getValue() << endl;
}
```

- The application may crash when a pointer is used after the object has been deleted
  - even worse, memory may be corrupted and strange problems may appear later on
- This problem is called use after free

Ownership

- Who is in charge of deleting the object?
  - The part of the program that has the responsibility of deleting the object is called the owner of the object
- For example:

```
class A {
  B *p;
public:
  A() : p(new B) {}
  ~A() { delete p; }
};
```

- Class `A` is the owner of an object of class `B`, it creates it when it is created, and deletes it when it is deleted

Ownership II

- A counterexample:

```
int function() {
  vector<A *> v;

  for (int i=0; i<5; i++) v.push_back(new A);

  ...
  // what happens to the memory?
}
```

- In this case, vector `v` is not responsible for deleting the objects; so, when the function finishes, we have a memory leak
- In other words, the vector cannot be responsible for deleting the object pointed by its elements, so it is not the owner
- it's the programmer that should take care of deleting the objects it has created, inside `function()` or elsewhere

Ownership III

- It is not always possible to identify one single owner for an object
- For example, the same object can be used by many parts of a program with pointers at different points in time
- Questions:
  - Which of the parts of the program is responsible for its deletion when the object is not used anymore?
  - When an object is deleted, how to make sure that all pointers to that object are invalidated, or not used anymore? (otherwise we run into memory corruption)
- Of course, the answers depend on the context, on the application structure, etc.
- We will now analyse techniques to reduce or eliminate this kind of problems

Outline

RAII

- A safer way to address the problem of creating dynamic objects is to use a wrapper class

```
MyClass *function() {
    std::unique_ptr<MyClass> p = new MyClass;

    // if something bad happens (i.e. an exception),
    // the object is deleted when p goes out of scope

    return p.release(); // returns the naked ptr
}
```

- the `unique_ptr` object is the owner of the object
  - it is responsible for its deletion
- if `p` goes out of scope without calling `release()`, the object is de-allocated
- Otherwise, with the `release()` method `p` loses ownership and it is not responsible for deallocating the object anymore

Naked versus smart pointers

- We say that a pointer is naked (or *raw*) if it is not wrapped in another class
- The `unique_ptr<>` is also called a smart pointer
- RAII
  - this technique of wrapping resource into a class is called Resource Acquisition Is Initialization, or RAII for short
  - The wrapper (the smart pointer) is the owner of the memory and it is responsible for its deletion
  - It can *release* ownership by calling `release()`.

Copying unique pointers

- The key idea is that there is only one owner of the memory block
- So, we cannot copy `unique_ptr`, otherwise there would be two owners for the same memory block
- rather, we need to pass ownership from one `unique_ptr` to another one

```
unique_ptr<MyClass> p = new MyClass;
unique_ptr<MyClass> q;

q = p;            // error, cannot copy!
q = std::move(p); // correct, transfers ownership
```

- How does it work?
  - The raw pointer is copied from `p` to `q`
  - The raw pointer in `p` is set to `nullptr`
  - When `p` goes out of scope, it performs a `delete nullptr` which results in nothing
  - see UniquePtrEx.cpp

Moving

- It turns out this moving ownership semantic is useful in many other places too, so it has been generalised
- to understand why `std::move()` works, and what it means we need to do a digression on different types of references
  - lvalue references,
  - universal references,
  - rvalue references.

Outline

Lvalues

- Before explaining how to implement the *move semantic*, we have to understand what is a lvalue and what is a rvalue
- an lvalue is an expression that has a well defined address, and to which we can assign values
- An lvalue is typically something that can stay at the left of an assignment

```
int x;

x = 5;          // x is an lvalue

int fun();

fun() = 7;      // ERROR: fun is not an lvalue

int& get() {
   return x;
}
get() = 5;      // get() is an lvalue
```

Rvalues

- A rvalue is the complement of an lvalue
  - It is everything that cannot be safely assigned a value and should not stay at the left of an assignment
- Example:

```
(x + y) = 5;     // ERROR: (x+y) is an rvalue
```

  - Typically, an rvalue is a temporary object that will disappear as soon as the effects of the expression in which it is used are completely over

```
int fun();
x = fun();
```

  - The return value of `fun()` is stored in a temporary location in memory until it is copied into x;
  - After that, the temporary location is deleted or reused

References

- It is an error to store the address of a temporary object in a pointer variable or in a reference, because the temporary is going to be deleted soon

```
int fun();
int &x = fun();   // error
```

- When we use `x`, it may refer to a location that does not exist anymore!
- With objects this fact is more evident

```
vector<MyClass> function();
vector<MyClass>  v = function();   // 1)
vector<MyClass> &r = function();   // 2)
```

  - The function returns a temporary object of type `vector`
  - In 1), the object is copied into `v`, then deleted
  - In 2), the object address is copied into `r`, then deleted (ERROR!)

rvalue references

- The C++ compiler will prevent you from obtaining a reference to a temporary object
  - the lines marked with `ERROR` in the previous code are *compiler errors*
- It does it by providing two different types of references:
  - lvalue references are the normal references that you have used until now, and they are denoted by the single character &
  - rvalue references are special references to temporary objects, and they are denoted by a double &&.
  - the compiler makes it impossible to assign a rvalue to a lvalue reference:

```
vector<MyClass> function();
vector<MyClass>  v = function();  // ok, copy
vector<MyClass> &r = function();  // compiler error !!
```

Rvalue lifetime and references in C++

- It is important to understand when a temporary object can be deleted
- The standard says:

  all effects of the expression must be completed

- For example, if the temporary is passed *by const reference* to a function, the temporary is deleted only when the function is over

```
MyClass get();                    // returns a temporary
int function(const MyClass &p);   // takes a constant reference
int x = function(get());          // correct!
```

- However, the reference must be const: C++ does not allow to have a non-const reference to a temporary object

```
const MyClass &p = get();  // correct!
MyClass &r = get();        // error!
```

Rvalue lifetime and references, cont.

- While the “const” limitation can save the programmer from some strange bug, it does not allow us to easily implement the move semantic

```
vector<MyClass> function();
vector<MyClass>  v = function();
```

- We would like to avoid to copy the temporary vector produced by `function()` into `v`
- To do this, we should modify the copy constructor to actually do the following things:
  - Copy the internal representation of the temporary `vector` (the pointer to the data) into `v`
  - Set the internal representation of the temporary to `nullptr`, so that the memory does not get de-allocated when the temporary is deleted
- However, the copy constructor takes a `const` reference, so it cannot modify the temporary, and it is forbidden to take a non-const reference to a temporary

Rvalue references

- The standard committee was aware of the problem with the move semantic, so C++11 now includes one more type of reference: the rvalue reference
- An rvalue reference is a *mutable* reference to a temporary object
- The syntax is &&

```
MyClass get();

// 1. const reference, can be used on anything
void function(const MyClass & temp_obj);
// 2. rvalue reference, can be used on temp. objects
void function(MyClass && temp_obj);
// 3. lvalue reference, cannot be used on temp. objects
void function(MyClass & temp_obj);

function(get()); // calls version 2
```

- notice they are all different functions, because they have different types

Overloading

- Since the rvalue reference is a new type, we can use it to overload a function

```
void function(const MyClass &p) {
    // called with normal objects
}

void function(MyClass &&p) {
   // called with temporary objects only
}
```

1. If we call function passing a lvalue, then the first one is called
2. If we call function passing a rvalue (i.e. a temporary object), then the second one is called

Order of evaluation

- When passing a temporary object to a function `fun()`, the compiler looks for a function in the following order:
  - If a function `fun()` that takes a rvalue reference is found, it is called; else ...
  - if a function `fun()` that takes a const lvalue reference is found, it is called; else ...
  - if a function `fun()` that takes an object by value is found, the temporary object if first copied, than the function is called; else ...
  - if a function `fun()` that takes arguments of another types is found, the compiler checks if a type conversion is possible; else ...
  - it issues an error.

Outline

Move constructor

- In addition to the copy constructor, it is possible to define a move constructor:

```
class MyClass {
    int *array = nullptr;
public:
    MyClass() { array = new int[10]; }
    MyClass(const MyClass &other) {
        array = new int[10];
        for (int i=0; i<10; i++) array[i] = other.array[i];
    }
    MyClass(MyClass &&other) {
        array = other.array;
        other.array = nullptr;
    }
    ~MyClass() { delete array; }
};
```

Example of move constructor

- An example of move constructor is in ArrayWrapper.cpp
- Notice the flag `-fno-elide-constructors` in the makefile:
  - the compiler can optimise the code by automatically eliminating copies and by enabling the *copy elision* optimization
  - the compiler sees what we are trying to do, and automatically builds the new object in place
  - in the example, `b` is directly built by `myArrayCreation`
  - try to remove the flag `-fno-elide-constructors` and see what happens
- However, even removing the flag, the compiler correctly checks that everything is in place:
  - make the move constructor private, and see what happens when you compile

Rvalues or lvalues?

- Sometimes, we have to “force” the use of a move constructor, because the object we are passing to the function is not a rvalue reference
- For example, the UniquePtrEx.cpp code shown before:

```
unique_ptr<MyClass> p = new MyClass;
unique_ptr<MyClass> q;

q = p;    // moves ownership?
```

- The code above does not compile! In fact, the assignment tries to assign p to q, and p is a lvalue !
- Therefore, the compiler tries to call operator:

```
unique_ptr& operator=(const unique_ptr &p) = delete;
```

- The `delete` syntax means that the operator has been removed from the interface, and calling it results in a compilation error

std::move

- To actually call the correct assignment operator (the one that takes a rvalue reference), we have to cast the type of the operand
- This can be done by using the `std::move()` function:

```
unique_ptr<MyClass> p = new MyClass;
unique_ptr<MyClass> q;

q = std::move(p);    // now it works
```

- This works because `std::move()` performs a cast into a rvalue reference, so that the following operator is called:

```
operator=(unique_ptr&& p);
```

Outline

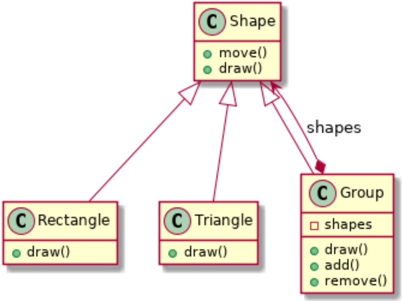
Smart Pointers

- `unique_ptr` allows to wrap one pointer, and guarantees unique ownership
  - When the `unique_ptr` is deleted, the pointed object is deleted
  - When the `unique_ptr` is moved, the internal pointer is reset to null
- Sometimes, however, an object has multiple owners
- Suppose, for example, that an object needs to be referred by different parts of a program at different points in time
  - Not clear who is the owner

Multiple ownership

Example:

- Suppose we want to group `Shapes`
  - A group is just a collection of shapes
  - We want to move all shapes in a group with a single move operation
  - We want to delete a group by deleting all shapes in a group
  - We want to add a `Shape` to a group, or remove a `Shape` from a group
- We can create a special kind of Shape called `Group` :
  - it contains a list of `Shapes`, and it is a Shape itself

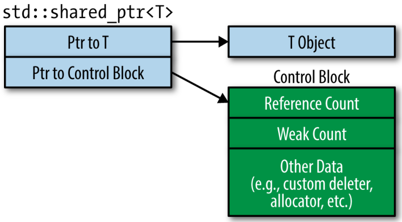


Membership

- We have several lists of `Shape` to manage:
  - the list of all shapes in the program
  - a list inside each group
- Which one is the *owner* of the shapes ?
  - who is responsible for deleting the object when it is not necessary anymore?
- Group is owner
  - every time we remove a shape from a group, we should pass the ownership to the general list
- The general list is owner
  - if we delete a group, we should remove all objects from the general list which then deletes them
- So, a unique pointer is not enough

Shared ptr

- A shared pointer permits to have a *shared* ownership for an object
- It uses a reference counter
  - it counts how many pointers point to the object
  - when the counter reacher zero, we can delete the object
- Therefore, a shared pointer contains two things
  1. A pointer to the object
  2. A pointer to a control block (which contains the reference counter)



Example

- To create the first shared pointer, we must use `make_shared()`

```
class MyClass {
public:
    MyClass(int i) { ... }
};

// Invokes the constructor of MyClass,
// and creates the control block
shared_prt<MyClass> p = make_shared<MyClass>(5);

auto q = p;    // a new shared pointer
```

See examples/shared.cpp

Problem

- The main problem with shared pointers (and with reference counters) is when we have two objects that point to each other

```
class A;
class B {
   shared_ptr<A> p;
public:
   B(shared_ptr<A> x) : p(x) {}
};
class A {
   shared_ptr<B> p;
public:
   A() {}
   void set(shared_ptr<B> x) { p = x; }
};

int main() {
   {  auto a = make_shared<A>();
      auto b = make_shared<B>(a);
      a.set(b); }
   // has everything been deleted?
}
```

Difference between reference counters and garbage collector

- In reference counting, every object has a counter that *counts* how many pointers point to the object
  - if there are circular data structures, it does not work
- A garbage collector periodically looks for unreferenced objects
  - it explores the graph of references, and deletes unconnected subgraphs of objects
- Reference counters are predictable
  - the overhead can be predicted quite easily
  - can be used with parallel threads
- Garbage collectors are less predictable
  - While the garbage collector execute we have to stop all parallel tasks
  - it is difficult to estimate the complexity of exploring the graph
  - it is difficult to estimate when the garbage collector will do its job

Weak pointers

- To solve the issue of the circular references, we can use a `weak_ptr`
  - a weak pointer does not increment the reference counter

```
auto p = make_shared<MyClass>(1);

weak_ptr<MyClass> w = p;      // get the pointer

// before using, you must convert to a shared ptr
shared_ptr<MyClass> sw = w.lock();
if (sw) { /* is valid */ }
else { /* is invalid */ }
```

Example

See http://en.cppreference.com/w/cpp/memory/weak_ptr

```
std::weak_ptr<int> gw;

void observe()
{
    std::cout << "use_count == " << gw.use_count() << ": ";
    if (auto spt = gw.lock()) {
        std::cout << *spt << "\n";
    }
    else {
        std::cout << "gw is expired\n";
    }
}

int main()
{
    {
        auto sp = std::make_shared<int>(42);
        gw = sp;
        observe();
    }
    observe();
}
```

Circular references and weak pointers

```
class A;
class B {
   shared_ptr<A> p;
public:
   B(shared_ptr<A> x) : p(x) {}
};
class A {
   weak_ptr<B> w;
public:
   A() {}
   void set(shared_ptr<B> x) { w = x; }
   void print_value() {
      auto p = w.lock(); // get the pointer, inc ref. counter
      if (p) cout << "Valid" << endl;
      else cout << "Invalid" << endl;
      // p goes out of scope, dec. ref. counter
   }
};
```

Outline

Hiding the implementation

- Even when you are writing a program all by yourself, it is useful to break the playing field into class creator and client programmer
  - the class creator implements the internals of a class, so that it can provide services
  - the client programmers use the class to realize some other behavior (e.g. another class)
  - almost all programmers are at the same time class creators and client programmers
- The class creators must not expose the implementation details to the client programmers
  - The goal is to export only the details that are strictly useful to provide the services

Why ?

- The concept of implementation hiding cannot be overemphasized
- Why it is so important?
  1. The first reason for access control is to keep client programmers' hands off portions they shouldn't touch
  2. This is actually a service to users because they can easily see what's important to them and what they can ignore
  3. The second reason for access control is to allow the library designer to change the internal workings of the class without worrying about how it will affect the client programmer
  4. For example, you might implement a particular class in a simple fashion to ease development, and then later discover that you need to rewrite it in order to make it run faster

Hiding implementation in C++

- In C++, not all implementation details are "hidden"
  - Class declaration in the include file contains private members
  - Include files are distributed along with lib files
  - Those are visible, not *hidden*! Customers can view the private part
- *Hidden* does not mean *secret*
  - It only means that they are not part of the interface
  - Thus, modifications to the private part do not imply modification to the client code, because the interface does not change
  - Code modification/adaptation is expensive, and a potential source of bugs

The pimpl idiom

- Changing the private part implies re-compiling all client files
  - Not so expensive, but annoying
  - Also, it may create unnecessary dependencies between classes
  - Also, sometimes we want to make all implementation *secret*
- To do this, we can use the pimpl idiom

```cpp
// include file
class MyClassImpl;

class MyClass {
  MyClassImpl *pimpl;
public:
    //interface
};

// cpp source file
// definition of private part
class MyClassImpl {...}

MyClass::MyClass() {
  pimpl = new MyClassImpl();
  ...
}
```

Pimpl performance

- pimpl stands for *pointer to implementation*
- All private data is in class `MyClassImpl`, which is not declared to the client
- The drawback is one more level of indirection:
  - All private data must be accessed through a pointer, or redirected to an internal class
  - This causes a slight increment in overhead
  - Another performance loss could be the call to `new` and `delete` operators every time an object is constructed
- However, the pimpl idiom brings us some other important advantages
- Example: `examples/pimpl.h`

Unique_ptr

- One "easy" improvement is to use a smart pointer for the implementation
  - we save one delete in the destructor

```
// include file
class MyClassImpl;

class MyClass {
  std::unique_ptr<MyClassImpl> pimpl;
public:
    //interface
};
```

- however, we have to understand what does it mean to "copy" an object of type `MyClass`
  - if we must copy also the implementation, then a `unique_ptr` is ok
  - if two objects share the implementation, then we must use a shared pointer

Advantages

1. It is possible to change the implementation (class `MyClassImpl`) without requiring the client code to be recompiled (only linked)
2. The implementation can be changed dynamically !
   - I am using a pointer to a class; this can also be a base class of an inheritance hierarchy
   - I can decide to change the implementation while the class is working
   - The class will appear to change its behaviour dynamically!
   - See the "State" pattern (https://en.wikipedia.org/wiki/State_pattern)