This course has already ended.
The latest instance of the course can be found at: Object oriented programming with C++: 2023 Autumn

4 Smart pointers and resource management¶

- **You can download the template for the programming tasks of the module as a `zip` file from `this link`.**

Contents

- 4 Smart pointers and resource management
  - 4.1 Unique pointer
  - 4.2 Shared pointer
  - 4.3 Weak pointer
  - 4.4 Programming task

As we have noticed so far, even though C++ provides many helpful tools in its standard library, we still need dynamic memory management and pointers to manage the memory. Doing this properly needs care and may be difficult in larger programs, to prevent memory leaks or dangling (outdated) pointers.

Use of dynamic memory with raw pointers, e.g. `new` and `delete`, makes the the code error-prone, because releasing memory requires an explicit `delete` call. If we are careless with this, some memory may be left unreleased if we exit from the current program scope unexpectedly, for example through an exception. A well-known solution is to utilize Resource Acquisition is Initialization (RAII) principle, which is a way of tying a resource lifetime to the object lifetime.

> Point of interest iconResource Acquisition is Initialization (RAII)
> previous | next
> **Resource Acquisition is Initialization (RAII)** is a resource management principle applicable to C++ (along with some other object-oriented languages). The principle means that a resource lifetime is tied to the object lifetime: the resources needed by the object are **allocated at the constructor**, and **released at the destructor**. As long as the object is properly deleted, the resources will also be released. Particularly, when an object is a local variable allocated from the stack, the resources will be released whenever the program leaves the scope where the local variable was declared. This way one can ensure that the resources are released also when an exception occurs (to be discussed in next module), or when the program block terminates e.g. through `break` or `return`.

Smart pointers are a RAII-friendly solution for safer memory management. Smart pointer is a template class that wraps an actual pointer with logic that helps in proper handling of the pointer, particularly tracking when the memory should be released. A smart pointer allocates the memory when it is initialized, and releases the memory when a destructor is called. We will discuss three kinds of smart pointers: **unique pointer**, **shared pointer** and **weak pointer**, that differ on how the pointer ownership is defined. Smart pointers are defined in the **memory** header.

Note

Related stackoverflow question: "RAII and smart pointers in C++"

4.1 Unique pointer¶

C++ Primer Chapter 12.1.5 (unique_ptr)

Unique pointer (unique_ptr) is a sole owner of the pointer it holds. The pointer cannot be shared, i.e. unique pointer does not allow copying of the pointer through construction or assignment. (You can, however, use the *move* constructor or assignment operator but that is outside the scope of this course).

unique_ptr of an object can be used just as a normal pointer is used, because it has both `*` and `->` dereferencing operators. `unique_ptr` is defined in **memory** header.

Below example illustrates the use of unique pointer.

```
1  int main() {
2      // Create new unique pointers to
3      // dynamically allocated troll objects
4      std::unique_ptr<Creature> uptr(new Troll("Hemmo"));
5      std::unique_ptr<Creature> uptr2 = std::make_unique<Troll>("Pate"); // since c++14
6
7      // these cause compiler errors:
8      // unique_ptr cannot be copied
9      std::unique_ptr<Creature> second = uptr; // error
10     std::unique_ptr<Creature> third(uptr);
11
12     // use of reference is possible normally
13     std::cout << uptr->getName() << std::endl;
14
15     // memory allocated for Troll is automatically released
16 }
```

Since C++14, unique_ptr also has a special function for allocating an object, shown on line 5 above. Alternatively, you can just give the constructor a pointer, like on line 4.

The pointer can be "stolen" from *unique_ptr* using the *release* function. It returns a raw pointer out of the unique pointer, and assigns null value to the original *unique_ptr*. After this, the programmer is again responsible of proper handling and releasing of the raw pointer. For example (adding to above example):

```
Creature *cp = uptr.release();
```

C++ uses the **nullptr** constant to indicate null pointers, and it can also be assigned to a *unique_ptr*. `uptr = nullptr;` would set uptr to nullptr and cause the memory allocated for Troll to be released. After this uptr can be set again using the *reset(some_pointer)* function.

Unique pointer can be returned from a function as it uses the move constructor and assignment operators (i.e. the unique_ptr being assigned to *steals* the pointer from the returned unique_ptr), so there is no risk of multiple objects of the *unique_ptr* type having the same raw pointer.

Uses of unique_ptr

- Providing exception safety (we will see exceptions in the next module) for managing dynamically allocated memory.

  The unique_ptr is an example of RAII, and the pointer owned by a unique_ptr object is released by the destructor of unique_ptr. This is, by far, the most important application of unique_ptr, and the code-segment above is an example. A simplified typical usage is:

  ```
    void func() {
      std::unique_ptr<int> up(new int);

      // do something fancy with the pointer
      ++*up; // increase the value of the allocated integer --> This is an example fancy operation.
   }
   // the allocated memory is released by the destructor of the unique_ptr before exiting the function sco
   // if an exception occurs within the function, it is guaranteed that the unique_ptr destructor will
   // be called.
  ```

- Passing ownership of dynamically allocated memory to a function and returning dynamically allocated memory from a function using move semantics.
- Storing pointers in containers

The last two usages of unique_ptr are not very common since other smart pointer types are usually preferred. However, by making use of move semantics, the management of allocated memory can be handed over to different functions, and sometimes provide convenience over shared_ptr.

4.2 Shared pointer¶

C++ Primer Chapter 12.1.1 (The shared_ptr class)

Shared pointer (shared_ptr) can safely be stored in multiple places in the program. The *shared_ptr* type holds a reference count that tracks the number of copies that the program currently has of the pointer.

shared_ptr makes a good use of operator overloading: for example, when a shared_ptr is assigned to another shared_ptr, the reference count is automatically increased. Similar to *unique_ptr*, it can be used just as a normal pointer, because it has both `*` and `->` dereferencing operators. `shared_ptr` is defined in **memory** header.

Although *shared_ptr* can be constructed using a pointer that is allocated in place, one should **avoid using the new operator for memory allocation**. Instead, new objects should be allocated using the **make_shared** function, which does the memory allocation and initialization, and initializes the reference counter properly. **make_shared** returns a new shared pointer of a given type.

Below example is based on the earlier defined classes of Trolls and Dragons. We create a vector that holds shared pointers to *Creature* type objects. Remember that *Creature* was an abstract base class of *Troll* and *Dragon* classes.

```
1  #include <string>
2  #include <vector>
3  #include <iostream>
4  #include <list>
5  #include <memory>
6
7  class Creature {
8  public:
9      Creature(const std::string& n, const std::string& t, int h) :
10     name_(n), type_(t), hitpoints_(h) { }
11     virtual ~Creature() { std::cout << "destructor: " << name_ << std::endl; }
12
13     const std::string& GetName() const { return name_; }
14     const std::string& GetType() const { return type_; }
15     int GetHP() const { return hitpoints_; }
16     virtual std::string WarCry() const { return "(nothing)"; }
17
18 private:
19     std::string name_;
20     const std::string type_;
21     int hitpoints_;
22 };
23
24
25 class Troll : public Creature {
26 public:
27     Troll(const std::string& n) : Creature(n, "Troll", 10) { }
28
29     virtual std::string WarCry() const { return "Ugazaga!"; }
30 };
31
32
33 class Dragon : public Creature {
34 public:
35     Dragon(const std::string& n) : Creature(n, "Dragon", 50) { }
36
37     virtual std::string WarCry() const { return "Whoosh!"; }
38 };
39
40
41
42 int main() {
43     // create an empty vector. Members are shared pointers to Creatures
44     std::vector< std::shared_ptr<Creature> > animals;
45
46     // Allocate new troll as a shared pointer, store in local variable
47     std::shared_ptr<Troll> tr = std::make_shared<Troll>("Peikko");
48
49     // add the troll to vector
50     animals.push_back(tr);
51
52     // create new Dragon and add it directly to vector
53     animals.push_back(std::make_shared<Dragon>("Rhaegal"));
54
55     for (auto i : animals) {
56         // i-> is a dereference of pointer to Creature type (virtual)
57         // i. refers to functions in shared_ptr class
58         std::cout << i->GetName() << " count: " << i.use_count() << std::endl;
59     }
60     animals.clear();
61
62     std::cout << "Animals cleared" << std::endl;
63 }
```

It is good to pay attention on line 15. The *auto* type declaration causes *i* to hold *shared_ptr<Creature>* type objects. Using the dereference operator points directly to a *Creature* object, but on the other hand, using the dot notation is also possible. This way we can access the member functions of *shared_ptr* class. The *use_count* function returns the current reference count of the shared pointer.

When the creatures are output using *i*, the pointer to "Peikko" is stored in three places: variable *tr*, in the vector, and temporarily in the *i* variable. On the other hand, the pointer to "Rhaegal" is directly created when added to the vector, so when the for loop outputs it, it is only stored in two places (vector and *i*). Then the vector is cleared on line 20. This causes the reference count to "Rhaegal" to drop to 0, which causes it to be deleted, and thus the destructor for "Rhaegal" is called (we have a printout in the destructor function) and the memory is released. Finally, "Peikko" is released when the main function ends.

With shared pointers we do not need to worry about releasing the memory: the *shared_ptr* class does it automatically when the last pointer to an object is destroyed.

A simplified implementation of shared_ptr

The *shared_ptr* can be considered as a structure composed of two pointers: one to the object and another to the use count (or reference count). Therefore, a simplified implementation would be:

```
template <typename T>
struct shared_ptr{
    T* obj_;
    int* use_count_;

    // constructors -- Rule of 3 or 5
    shared_ptr(): obj_(nullptr), use_count_(nullptr){

    }
    // overloads -- do not forget copy constructor

    // destructors
    ~shared_ptr(){
        // when should the members be destroyed?
    }

    shared_ptr& operator= (const shared_ptr& lhs){
        // assignments.. modify the use count
    }
    // other overloads of operator=

    T& operator*() const {
        // return a reference to the object
    }

    T* operator->() const {
        // return the object pointer
    }
};
```

Use of shared_ptr

The shared_ptr is used when two pieces of code need access to some object but neither has has exclusive ownership of the object in the sense that neither has an exclusive right to destroy the object. The object is destroyed when the object's reference count goes to zero.

shared_ptr should not be used for passing a pointer from one owner to another. The ownership passing should be done using *unique_ptr*, which is very light weight in terms of added complexity.

4.3 Weak pointer¶

C++ Primer Chapter 12.1.6 (weak_ptr)

Weak pointer (weak_ptr) does not track the reference count of the pointer it owns. Otherwise it works quite similarly to a shared pointer. A shared pointer (or another weak pointer) can be assigned to a weak pointer, but this does not affect the reference count of the shared pointer. Therefore, it is possible that an object controlled by a shared pointer is deleted while there are weak pointers pointing to it. However, we can check it this has happened using the *expired* function. If it returns *true* we should not try to use the weak pointer, because the object it points to has been deleted.

Weak pointer cannot be dereferenced directly because of the risk of handling an expired pointer. Instead, we must use the *lock* function, that returns a new shared pointer based on the weak pointer, or *nullptr* if the weak pointer was expired. Weak pointer can only be initialized from a shared pointer.

The following example shows the usage of a weak pointer.

```
int main() {
    std::weak_ptr<Creature> wp1(std::make_shared<Dragon>("Puff"));

    // Puff dies immediately, because reference count remains 0

    // try to use the pointer, by making a shared copy of it
    // unfortunately sp == nullptr, so we cannot anymore use it
    std::shared_ptr<Creature> sp = wp1.lock();
    if (!sp) {
        std::cout << "Oh no, dragon has died!" << std::endl;
    }

    // let's try again...
    std::shared_ptr<Creature> sp2 = std::make_shared<Troll>("Peikko");
    std::weak_ptr<Creature> wp2 = sp2;

    // now the weak pointer is valid, because sp2 is still alive
}
```

4.4 Programming task¶

Points 25 / 25 | My submissions 22 ▾ | Deadline Friday, 15 October 2021, 19:59 | To be submitted alone

⚠ This course has been archived (Saturday, 17 December 2022, 19:59).

Restricted pointer

Objective: Learn to implement your own kind of smart pointers. Templates become more familiar again.

In this exercise you should implement a generic smart pointer of your own. Your smart pointer class should be called *RestrictedPtr*, and it should be able to contain a pointer to any any type of data (use a template!). There is a little twist in this restricted pointer compared to the traditional shared pointer, as **its reference count can be only 3 or less**. This means there may be at most 3 copies of a RestrictedPtr pointing to the same memory address.

The RestrictedPtr should have:

- Constructor for a case when no parameters are given, and
- Constructor for a case when a raw pointer is given as a parameter. In this case the constructor should obviously store the given pointer into this instance.
- Copy constructor. This should make a copy of another *RestrictedPtr* and its raw pointer should point to the same memory location as the instance it was copied from. If there are already 3 or more copies of that instance, this copied instance's raw pointer will point to *nullptr* and the reference count of this instance will be set to 1.
- Destructor. If the reference count of this instance drops to 0, the destructor should release the memory allocated to the raw pointer.
- Copy assignment. This is basically the same as with the copy constructor, but for copy assignment there might also be a situation when an already initialized RestrictedPtr is assigned to another RestrictedPtr, and the assigned instance's reference count is 1 (it is the only copy of RestrictedPtr pointing to that address). Because of this the copy assignment should also release the memory allocated to the raw pointer in that situation.

The RestrictedPtr should also have the methods

- *GetData()* should return the data where the instance's raw pointer points to. This returned data/type should be modifiable.
- *GetPointer()* should return the raw pointer of this instance.
- *GetRefCount()* should return the number of RestrictedPtrs that contain the same raw pointer.

You have got the artistic freedom to implement the reference counter for the RestrictedPtr class yourself.

restricted_ptr.hpp

Choose File No file chosen

restricted_ref_counter.hpp

Choose File No file chosen

Submit