IB9N7 C++ for Quantitative Finance

Lecture 11: Resources

J.F.Reizenstein@warwick.ac.uk

28 January 2016
(Week 17)

Outline

Today we consider how C++ deals with resources, which are things which a program must be careful to finish using promptly. We introduce

- writing to files
- pointers
- the heap / dynamic allocation
- destructors.

Writing out

Writing to a file

An **ofstream** behaves like `cout` but writes to a file.

```
1  #include<iostream>
2  #include<fstream>
3  int main(){
4      int i;
5      {
6          std::ofstream ofs("output.txt");
7          ofs<<"My␣favourite␣number␣is␣"<<2+40<<"\n";
8          std::string a = "Have␣a␣nice␣day\n";
9          ofs<<a;
10         std::cin >> i;
11     }
12     std::cin >> i;
13 }
```

There are similar ways to append to a file, read from a file, and read and write to a string.

Destructors

How?

Note how it is the end of the life of the variable `ofs` which is the point where the file is closed by the program. Any class can define a member function called its destructor which runs when its lifetime ends - including early returns from a function, `throw`, `break`, `continue` etc.

- It is always named with a tilde followed by the same name as the class:

```
~MyClass();
```

- It takes exactly zero parameters and has no return type (not even `void`), and thus cannot be overloaded.

Rules for destructors

It is a good idea to obey the following rules.

- A destructor should be short and simple, not doing anything which might be slow.
- A destructor should not do anything which might throw an exception.

When an object is destroyed, its members will also be destroyed. This happens automatically. The practice of

relying on the lifetime of an object to match the use of a resource and using the object's destructor to clear it up is good and reliable. It is known as "RAII", for "Resource Allocation is Initialization".

example of destructor

```
1  #include<iostream>
2  using std::cout;
3  class A{
4  public:
5      ~A(){cout<<"ello␣";}
6  };
7  class B{
8  public:
9      ~B();
10 };
11 B::~B(){cout<<"world!";}
12 int main(){
13     B b;
14     A a;
15     cout<<"H";
16 }
```

Pointers

Pointers: concept

Pointers and memory addresses

A pointer is a variable whose contents is a memory address.

- Pointers are a powerful concept inherited from C.
- Allow direct access to the memory.
- Also involved in decoupling, polymorphism, etc.

Pointers

- A variable is specified as a pointer by using an asterisk (`*`) in the declaration.
- The asterisk is also used as the **dereference** operator (sometimes called the **indirection** operator).
- Pointers must also specify the data type of the entities they can point at.
- The memory address of a variable is found using the reference operator (`&`).
- Can read `&` as “address of” and `*` as “at” or “contents of”.

Example: syntax

```
int *p; // declare and define p to be a pointer to an int
```


Pointer declarations

Warning

When declaring multiple variables in one statement, the asterisk only binds with the next identifier, not all identifiers in the list.

```
1 int *p1, p2;
2 int* p1, p2;
3 // are both the same as
4 int *p1; // there is (to be) an int at location p1
5 int p2;
```

Pointers and dereferencing example

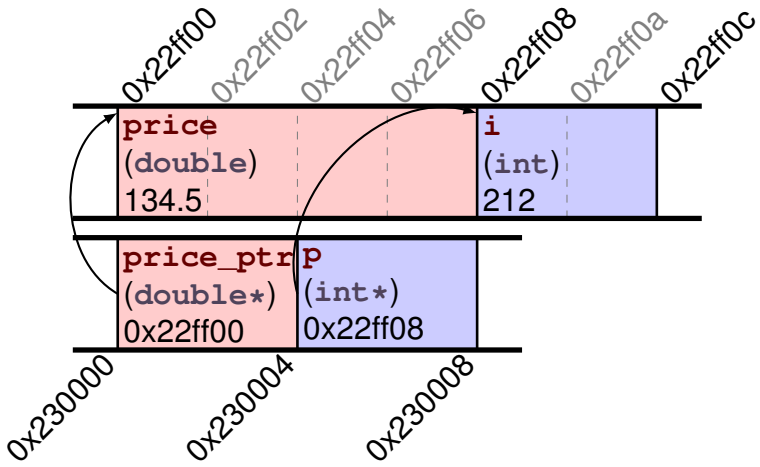
Example: pointers and dereferencing

```
1 double price = 134.5;
2 double *price_ptr; // declares a pointer to a double
3 price_ptr = &price; // assigns the address to the pointer
4 // (can also declare and initialise in one statement as usual)
5 std::cout << *price_ptr << std::endl;
```

- **price_ptr** contains the address of **price**.
- The first asterisk indicates **price_ptr** is a pointer to a **double**, rather than a **double** itself.
- The second asterisk is used to dereference the pointer and retrieve the (pointed-to) value of **price**.

Pointers by picture

If you're having a hard time visualising how pointers work, you might find the following picture helpful.



Pointers by picture: explanation

In the diagram

```
1 double price = 134.5; // declare and initialise a normal variable
2 int i = 212; // ditto
3 double *price_ptr = &price; // declare 'price_ptr' to be a
      pointer to a double, and initialise it to contain the memory address
      of 'price'
4 int *p = &i; // declare 'p' to be a pointer to an integer, and initialise
       it to contain the memory address of 'i'
```


Assigning to pointed variables

A dereferenced pointer can appear on the LHS of an assignment (lvalue).

Example: Assignment

```
*price_ptr = 4.5;
```

assigns 4.5 to the location given by **price_ptr**.

i.e. changes the value of **price**.

Assigning pointers

Pointers can point to different variables (of the same type) during their lifetime, since they are just variables which hold a memory address:

Example: Assigning pointers

```
1  int i1 = 1, i2 = 2, i3 = 3;
2  int *ptr = &i1;
3
4  std::cout << *ptr << std::endl; // 1
5
6  ptr = &i2;
7  std::cout << *ptr << std::endl; // 2
8
9  ptr = &i3;
10 *ptr += i2; // now, i3 = 5
11 std::cout << *ptr << std::endl; // 5
```


Pointers and initialisation

The following line creates and dereferences a pointer:

```
1  double *ptr; // not initialised: bad!
2  double val = *ptr; // using before initialised: very bad!!
```

It has not been initialised before the dereferencing, so have undefined behaviour. Chances are it does not point to memory we own. Something is likely to go wrong.

Pointers: best practice

- Pointers must be initialised before they can be safely dereferenced.
- If it is not possible to assign a meaningful address to a pointer when it is declared, make it a **null pointer** (see later).

operators on pointers

If a pointer is uninitialised, or its pointee's lifetime is over, nothing may be done with it until it is assigned to something. Such a pointer may be referred to as junk, toast, trash, or rubbish. Barring that, comparisons (!=, ==), assignment, copying etc. work simply.

Segmentation faults

Segmentation fault

A segmentation fault is an error which occurs when a program attempts to access a memory location that does not belong to it.

This can happen with pointers (and native arrays, or square bracket syntax of accessing vector elements), among other situations.
In Windows, a program which causes a segmentation fault is terminated by Windows, and the familiar message "‹program› has stopped working" appears.

Segmentation faults in the debugger

If you are running your program through the debugger when the segmentation fault occurs, you may instead see the following message, after which the debugger will hopefully pause your code on the line causing the fault:

Null pointers

Null pointer

A null pointer is a pointer whose value is null (think zero, but note this is not quite accurate).

Dereferencing a null pointer is undefined behaviour and usually causes the program to crash (but it's clear why!).

A conversion from an initialised pointer to `bool` will return `true` iff the pointer is not null. To get a null pointer, use the keyword `nullptr`. For example

```
1 int *a = nullptr;
2 ...
3 if(a){...}
4 if(a!=nullptr){...}// same as the previous line
```


Null pointers and initialisation

If you use pointers in your code, always initialise them to either a valid value or a null pointer (do not let your pointers initialise to junk!).

You can then test to see if (maybe) you can dereference them:

Example

```
1 int* ptr = nullptr;
2 ...
3 if (ptr) // boolean conversion rule: nullptr=false, else=true
4 {
5    std::cout << *ptr << std::endl;
6 }
```


pointer function parameters

We have learned that a function which needs to know about a class instance should take it as a (possibly const) reference. It would work to take a pointer instead, it is less self-documenting. A pointer might be useful if the parameter is supposed to be optional - the user could pass nullptr instead.

Accessing object members

Suppose `ptr` is a (valid) pointer to a `Point`.

One can of course access members by dereferening the pointer first and using the dot operator (though parentheses are needed):

```
1 (*ptr).get_x();
```

More convenient: use the `->` (arrow) operator, eg

```
2 ptr->get_x();
```


Dynamic memory

Automatic vs. dynamic memory

Dynamic memory

Dynamic memory refers to memory where the lifetime of the allocation is not controlled automatically by the compiler.

Dynamically allocated memory is requested from the **free store** (aka pool or heap, but the term *free store* is preferred in connection to the C++ dynamic memory operators).

So far, we have always been using automatic memory (normal variable definitions), although some objects we have used (like `vector`) have used dynamic memory behind the scenes.

Dynamic memory is used in connection with the `new` keyword, although we usually avoid this keyword. Useful for dynamic polymorphism.

Dynamic memory: requesting and releasing

In C++, dynamic memory is

1. requested with the `new` keyword.
   - `new` returns a pointer to the start of the allocated memory if successful.
   - If unsuccessful, raises an exception.
   - Cannot expect consecutive use of `new` to return nearby blocks of memory.
2. released (and destroyed) with the `delete` keyword (only).
   - Can only `delete` memory addresses returned by a previous call to `new`, else undefined.
   - Cannot `delete` same memory multiple times (unless reallocated in between), else undefined.
   - Deleting a null pointer is safe, and has no effect.
   - Best to `delete` as soon as the memory is no longer needed (locality).

If `delete` is never called (either directly or indirectly e.g. by a smart pointer class), then the memory is never released (bad)!

Dynamic memory and initialisation

When creating dynamic memory, can (should) optionally explicitly initialise also:

- Use function-style initialisation syntax: parentheses (containing an optional expression-list) after `new` and the data type.
  Arguments in the optional expression-list are passed to the constructor in the case of object types, and have the obvious meaning for fundamental types.
- No parentheses means default initialisation.
- Recall default initialisation for fundamental types means uninitialised!
  For object types, means default constructor.
- Value/direct initialisation are preferred to default initialisation.

Syntax example: fundamental types

Example for illustrating syntax only

Create memory for a single fundamental type (`int`) dynamically. (Normally only used for arrays or class types.)

```
1 int *a = new int; // the memory pointed to by a is default
    initialised (meaning uninitialised since fundamental type)
2 int *b = new int(); // the memory pointed to by b is value
    initialised (here, to 0)
3 int *c = new int(5); // the memory pointed to by c is direct
    initialised to 5
4 ...
5 delete c;
6 delete b;
7 delete a;
```


Memory leaks

If memory is assigned using `new`, it **must** be freed/released (eventually) using `delete` (ditto for each of the other methods of allocating dynamic memory).

The part of code responsible for calling `delete` may not be same as the part of code that called `new`. The code responsible for handling the `delete` is said to **own** the pointer. Better to use smart pointers (discussed later).

Memory leaks

A **memory leak** occurs when dynamically allocated memory becomes unreachable. e.g. the program has finished using it, and discarded its address, but failed to release it (and cannot release it later, as it no longer has the address).

Smart pointers

More problems with raw pointers

An object allocated from the heap is a resource and it must be deleted. We should make it easy to do this reliably. Very often, pass pointers around between functions, classes etc.

- Need to signal to some other part of the code that it should `delete` later.
- Not always clear which code should be responsible.
- Subject to programmer error and omission.
- Sometimes difficult for a single piece of code to take that responsibility.

Smart pointers

Smart pointers provide all the advantages of raw pointers but help to reduce the associated dangers on the latter.

For example:

- they are automatically initialised to `nullptr`,
- they make the memory deallocation automatic,
- they help enforce ownership semantics,

thereby minimising otherwise common memory leaks and dangling pointers.

Using operator overloading, C++ smart pointers are able to mimic the interfaces and behaviour of traditional (raw) pointers (including dereferencing syntax).

Smart pointers in C++

Require header `<memory>`.

The main one is `unique_ptr`.

There is also `shared_ptr`.

In old versions of C++, there was also `auto_ptr`.

This still exists but is deprecated.

Use `unique_ptr` instead.

It is always easy to convert a `unique_ptr` into a `shared_ptr` when more than one copy of the pointer is needed.

We focus on `unique_ptr`.

Unique pointers and shared pointers

Unique pointers cannot be copied, only moved (transferring the ownership).

- Use these when you can rely on a single owner, i.e. there can be a single object whose end-of-life will call delete.
- Uses exactly the same amount of memory as a raw pointer.
- You can still use raw pointers to the object.
- Accessing the encapsulated pointer does not incur much overhead penalty over the speed of raw pointers.

If more than one entity needs to own the pointer, use a shared pointer instead.

- A shared pointer keeps track of how many instances there are (reference count).
- If a shared pointer is destroyed, it only deletes the

- May need to worry about circular references ☆.

Construction of `unique_ptr` directly

Can create an instance of a **`std::unique_ptr`** just like:

Example (bad)

```
std::unique_ptr<Date> p(new Date{21,1,2026});
```

However, then confusing because there is **`new`** but (correctly) not **`delete`**.

Also, memory leaks can still occur if more than one **`unique_ptr`** is created in the same statement.

Best not to use the above syntax!

`std::make_unique`

It is better to encapsulate construction of smart pointers via a function call, to hide the dangling **`new`** keyword and to eliminate the remaining possibility of memory leaks.

There is a "standard" function called **`make_unique`** for this.

- (And similarly, **`make_shared`** for **`shared_ptr`**s.)

This function is not part of C++11 due (partly) to an oversight.

- It is part of C++14, but our compiler version doesn't have it.

Need to provide it ourselves.

- Definition is short but uses (variadic) templates and so is (for now) incomprehensible.

Using `make_unique`

Example

```
std::unique_ptr<Date> d(std::make_unique<Date
    >(21,1,2026));
//or
auto d = std::make_unique<Date>(21,1,2026);
```

Actually, until we get C++14 support, use the function defined on the next slide and invoke as:

```
auto d = make_unique<Date>(21,1,2026);
```

i.e. without the **`std`** namespace.

The arguments to the constructor are given as the arguments to **`make_unique`**.

Definition of `make_unique`

Until you get a C++14 compiler, define the following in global scope (e.g. in a custom header file).

Note that we do **not** alter namespace **`std`**.

When you get a C++14 compiler, remember to delete your custom version and use the one in **`std`** instead.

```
1 template<typename T, typename... Args>
2 std::unique_ptr<T> make_unique(Args&&... args)
3 {
4     return std::unique_ptr<T>(new T(std::forward<Args
      >(args)...));
5 }
```

Some member functions

- `unique_ptr` objects can be swapped with a non-member `swap`.
- `get` returns the pointer.
- `reset` sets the pointer afresh, deleting the pointee.
- `release` returns the pointer and relinquishes ownership. This is the only way to escape the certainty of ownership.

Pointers in containers

It is natural to have a vector of pointers to objects which owns them. In this case, use a vector of unique pointers.

For example:

- It is much quicker to sort a vector of pointers (smart or otherwise) than a vector of objects (less copying required). However, actual elements no longer contiguous (only the pointers are).
- We will need this for dynamic polymorphism later.
- You cannot have a vector of references (because references are not assignable, which would prevent the dynamic features of the vector from working).

Lab objectives