IB9N7 C++ for Quantitative Finance

Lecture 13: Small objects

J.F.Reizenstein@warwick.ac.uk

11 February 2016
(Week 19)

Outline

In this lecture, we'll cover the following topic:

- Some functions every object may have
- Some small objects

Important constructors and connected issues

Important constructors: default constructor

Default constructor

- "Default" here means "can be called with no parameters". (Therefore it either has an empty formal parameter list or all parameters have default values .)
- Declaration (within the class definition):

```
MyClass();
```

- Definition (if defining outside the class definition, else omit scope):

```
MyClass::MyClass()
{
    ‹body›
}
```

Default constructors: usage notes

- The default constructor is invoked when a class is instantiated with the following syntax:

```
MyClass c;  // Initialisation by default constructor
```

- If you do not declare a constructor, a default constructor with an empty body is automatically provided for you (it is said to be **implicitly generated**).
- If you do declare a constructor, a default constructor is not implicitly generated. It is therefore possible to have classes without a default constructor (i.e. all constructors must take arguments).

Important constructors: copy constructor

Copy constructor

- "Copy" here means it can be called with a single argument which is of the same type as the class.
  Normally use a const reference.
  Therefore, a declaration may look like this:

```
MyClass(const MyClass& src);
```

- Invoked when an object is instantiated from another object of the same class.
- Compiler tries to provide a copy one and a move one if you don't declare one (regardless of whether you declare other constructors).
  The compiler-provided copy constructor does a shallow copy of all data members.

Copy constructors

Initialisation and assignment

Assuming that `c` is of type `MyClass`:

```
MyClass d = c;  // Initialisation by copy constructor (assignment-like syntax)
MyClass e(c);  // Initialisation by copy constructor (function-like syntax)
d = c;  // Assignment by copy assignment operator (see operator overloading lecture later)
```

N.B. if `c` were of a type other than `MyClass` then these would not be straightforward copy operations.

Conversion constructors

A constructor that accepts
- a single argument with no default value,
- of different type to the constructor's class

is called a **conversion constructor**.

Creates an object from an argument with a different type.

Important: Implicitly called by compiler when
an object of first type is expected,
but given an object of second type.
To avoid implicit conversion: prefix constructor declaration with keyword `explicit` (recommended).

Special member functions

C++ will automatically generate the following if we do not declare them:

- default constructor (if we do not declare any constructor);
- copy constructor;
- move constructor (sometimes);
- copy assignment operator;
- move assignment operator (sometimes);
- destructor.

For this reason, these are known as the **special member functions**.

You can use `= default` and `= delete` to explicitly specify whether you want the default versions.

Rule of three

Rule of three

If custom defining any one of:

1. copy constructor;
2. copy assignment operator;
3. destructor,

should (usually) custom define all three! (Not a compiler requirement.)

Rule of three: rationale

Principle of rule of three

If you have to custom define one of the above special member functions, it is usually because the compiler-generated version does not fit the needs of the class.

The compiler-generated versions of the others will probably not fit the needs of the class either.

Not defining the move constructor or assignment operator is unlikely to result in a bug (but may be a missed opportunity for optimisation), so is not part of the rule of three.

Can satisfy rule of three if don't define any of them.

Dynamic memory and virtual destructors

If `delete` is called (possibly implicitly) on a pointer where the static type of the pointer is different to the dynamic type, then:

- if the destructor is not `virtual`, the destructor corresponding to the static type is invoked;
- the destructor corresponding to the dynamic type is not necessarily invoked;
- get undefined behaviour; the object might only be partially destroyed!

For an example of this, see the lab exercises.

Virtual destructors

If there is any possibility of the above happening, then the destructor should be declared `virtual`.

- More generally, if there are any `virtual` methods in a class, then you should declare the destructor to be `virtual` too.
- This may mean having to define the destructor(!) instead of relying on the automatically provided version (which would not be virtual).
- An empty destructor body will often suffice.

Virtual destructors

- When a pointer is `delete`d, the destructor is used through the pointer. It obeys the same binding rule as other member functions.
- A base class should always declare its destructor as `virtual` if you want to be able to delete any derived class via the base class pointer.
- Note that, by declaring the destructor as `virtual` in the class definition, the destructor is declared! Therefore, you also have to implement it, rather than relying on the implicitly generated one. This is usually the empty function body.

```
class Base{
public:
    virtual ~Base(){}
};
class Derived : public Base {};
int main(){
    std::unique_ptr<Base> b = make_unique<Derived>();
}
```


Shallow copy and deep copy

Shallow copy: Copies pointers, but not what they point to.

Deep copy: Makes a copy of pointed to object, and sets pointer to point to new object.

Default copy constructor/copy assignment does shallow copy.

When that is what is wanted, it may still be helpful to make it explicit.

If need deep copy (e.g. have data members which are pointers):

Must define own rule-of-three functions.

When defining copy constructor:

Must define copy for all data members, not just those which are/contain pointers.

Casting static type

One can use the casting operators to convert between static types in the hierarchy *where necessary*.
e.g.

```
Derived * p3 = dynamic_cast<Derived *>(p2);
```

Notes:

- If a `dynamic_cast` fails on a pointer, value will be `nullptr`. `dynamic_cast` is only possible if there are `virtual` methods.
- If a `dynamic_cast` fails on a reference, throws `bad_cast` exception (no such thing as a null reference).
- One can also use `static_cast` when one is sure the conversion is valid (undefined behaviour if not).

Pointer arithmetic

- A pointer is simply a memory address.
- We can move to another point relative to this in memory using the standard arithmetic operations + and − with (signed or unsigned) integers. The units for arithmetic operations are multiples of the size of the underlying data type!
- Only valid if the original pointer and the result of the expression point to elements of the same allocated memory block, else undefined behaviour.
- When two pointers to elements of the same array object are subtracted, the result a value of (implementation-defined) type **ptrdiff_t** representing the difference of the index of the two array elements.

Small types

pair

- Sometimes, a very simple class is needed which contains a small number of elements, and you can use some library types instead of defining your own classes.
- If we have types **A1** and **A2** then we can write **std::pair<A1,A2>** for a type which has two public data members: **first** of type **A1** and **second** of type **A2**. This comes from <**utility**>
- We can create one with std::make_pair. For example:

```
std::string s = "hello";
auto x = make_pair(s,3.8); //x has type std::pair<std::string,double>
std::pair<std::string,double> y(s, 3.8); //the same
std::cout<<x.first;
```


tuple

- If we have types **A1**, **A2** etc. then we can write **std::tuple<A1,A2,A3>** for a type which has a public data member of each type. This comes from <**tuple**>
- The elements do not have known names, we access them with a nonmember function **get**.
- We can create one with std::make_tuple. For example:

```
std::string s = "hello";
auto x1 = make_tuple(s,3.8,&s); //x has type std::tuple<std::string,double,std::string*>
std::cout<<std::get<1>(x1)<<"\n"; //prints 3.8
```

tuple (2)

- **pair** and **tuple** can be constructed and assigned wherever it makes sense, and they have comparison and < operators.
- We can create a tuple of references with **std::tie**, and pull out of a tuple as follows.

```
std::string s = "hello", t;
double d;
auto x1 = make_tuple(s,3.8);
std::tie(t,d) = x1;
```

- This can be useful for returning multiple values from a function.

array

- If we have **A1** is a type, and **N** is a **size_t** known at compile time then we can write **std::array**<**A1**,**N**> for a type which has **N** public data members of type **A1** which live adjacent in memory. This comes from <**array**>
- The elements do not have known names, we access them with **at** or `[]`.
- For example:

```
std::array<int, 3> a{2,3,64};
std::array<int, 4> b{2}; // contains 2,0,0,0
std::array<int, 4> c; //uninitialised
std::cout<<a.at(64)<<"\n"; //prints 3.8
```

- Note that a **vector**<**array**<**int**,42>> only has one memory block.

Enumerated data types

For variables with a restricted range of values, where the permitted values form a collection of named constants.

```
enum class Teacher {JEREMY, JESSIE, MATTHEW};
```

Defines:

> **Teacher::JEREMY** = 0, **Teacher::JESSIE** = 1, and **Teacher::MATTHEW** = 2.
>
> Not a normal class, or normal **int**s! Strongly typed.

Enum names at run-time

As usual with variable declarations, your program can't normally know the variable identifier that was originally associated to a memory location or an enum value.

i.e. No built in way to convert from `"JEREMY"` to **JEREMY** and vice versa.

Write conversion functions yourself as necessary, example on next slide.

Can convert between integral types and enum types using **static_cast**.

> But, just use them as named constants, not magic numbers (normally pretend you don't know their values)!

Enum conversions

```
std::string Teacher_to_string(Teacher t)
{
    return sb == Teacher::JEREMY ? "Jeremy" :
           sb == Teacher::JESSIE ? "Jessie" : "Matthew" ;
}

Teacher string_to_Teacher(std::string s)
{
    if (s == "Jeremy")
        return Teacher::JEREMY;
    else if (s == "Jessie")
        return Teacher::JESSIE;
    else if (s == "Matthew")
        return Teacher::MATTHEW;
    throw std::runtime_error( s + " was not a valid teacher name");
}
```


sort

- This is part of a future topic, but here is an application of the < operator.
- If **a** is a vector of objects which are comparable with <, and < defines a “strict weak ordering” (e.g. a total order), then you can include **<algorithm>** and write

```
std::sort(a.begin(),a.end());
```

  to get **a** to be sorted in ascending order. Elements will have switched positions.
- Alternatively, a less-than function taking two elements and returning **bool** can be provided as a third argument to std::sort. This could be a function name, a **std::function**, a lambda or an object with an overloaded `operator()`.

Lab objectives

-