MTH 4300, Lecture 26

Abstract Classes;
Virtual Destructors;
`dynamic_cast`;
Exception Handling

E Fink

May 13, 2025

1. Abstract Classes

Let's return to our Message example from last time.

L26x1_message.cpp

We had as our base type a generic `Message` type. Question: why was that type even there? We didn't intend to create any "generic" `Message` objects – we created objects representing specific types of actual message. And the one method we did create for the class just said "No content", which seems pretty worthless. So why does it exist at all?

The answer is: the `Message` class exists so I can do exactly the type of things that I did in `main()` – for example, create a single container that is capable of holding different types of `Messages`, and then apply the various `display()` methods to all these objects.

Abstract Classes

So, the `Message` base class is really NOT meant to have its own objects – its purpose is just to hold the methods that are shared by a collection of similar classes.

If we want to get across this point especially strongly – that a class exists not to be instantiated, but just to collect the shared features for a family of inherited classes – we can actually arrange for our class to have object creation outlawed.

To achieve this: in your base class, create at least one function which:

- is `virtual`;
- does not have any implementation at all in the base class – not even an empty pair of curly braces;
- and has its declaration end with `= 0;` (somewhat bizarre).

If you do this, the method in question is called a *pure virtual method*, and the base class in question is called an *abstract class*. Let's go back to the message example, and make the `Message` class abstract.

Abstract Classes

Abstract classes serve as “interfaces”: they generally are meant to make it easier to perform operations that are defined in other classes. (Much like a gas pedal makes it easier for a driver to use their car’s engine.)

With abstract classes, keep note of the following points:

- To repeat, you cannot declare objects of these class types.
- However, you CAN create pointers to these types – but these pointers should point to objects which are of one of the derived subtypes.
- Every pure virtual method that is left unimplemented in the base class should be implemented in the derived subtypes (if you neglect to do this, then the derived subtype will ALSO be abstract).

2. Virtual Destructors

Recall that destructors are called whenever an object ceases to exist – when it goes out of scope, or when it is explicitly `deleted`. Their purpose is to release allocated memory.

Inheritance can make this a tricky business. See **L26x2_destruct.cpp** . The base class `Human` does not allocate extra memory on the heap, but `Parent` does.

```
Parent *a = new Parent("Alice", 5);
Human *b = new Parent("Bob", 5);

delete a;
delete b;
```

Due to the default of static binding, `delete b` calls the destructor for `Human` – which doesn't perform any array deallocation! So `Bob`'s child list is still allocated in memory.

Virtual Destructors

The solution is simple: make `Human`'s destructor virtual as well. Let's do that now!

In fact, this good general practice: if there is even a chance that an inherited class will have dynamically allocated memory, simple write a virtual destructor:

```
virtual ~Base() {} // That's all you need to do!
```

While we're talking about this, we should mention that a destructor to a derived class will automatically call a destructor to the base class. Furthermore, the destructors will get called in the opposite order to the constructors: derived class destructors get called first, then base class destructors get called.

3. dynamic_cast

We saw that for a given method in a base class, if it is declared as virtual, then the caller's *dynamic* type determines the version of the method that gets call; and otherwise, the caller's *static* type determined which version gets called.

But for this to work, there needs to be a version of the method in question in the base class to make virtual in the first place.

Consider **L26x3_fly.cpp** . A Bird is an Animal, so a Animal* can be assigned the address of a Bird.

We want to call fly() on an Animal* that happens to point to a Bird. But, since the Animal class has no fly() method at all, you can't use virtual to solve the problem.

dynamic_cast

Instead: use `dynamic_cast`. The syntax is
`dynamic_cast<Inherited*>(base_class_ptr)`
This line takes a pointer to a base class, and converts its type to `Inherited*`, if possible (if the dynamic type actually is `Inherited*`). If not possible, this function will return `nullptr`.

Now, let's uncomment the code in `fly.cpp`, to see how to exploit this.

Important: for technical reasons, *to use this feature, you need to have at least one virtual method in the base class.* It makes sense to make the base class destructor virtual in this case.

In general, `dynamic_cast` should be used sparingly.

4. Exception Handling

Most code operates under various assumptions about the values and resources they are provided. Denominators should be non-zero; logarithms should be supplied positive values; arrays should be written to in-bounds; ifstreams should be opened with the name of an existing file; etc.

When these assumptions are violated, program behavior can be unpredictable. To avoid this, one can make their code generate *exceptions*. These are segments of code that interrupt program flow upon encountering them.

Exceptions make program behavior more predictable – by forcing your program to terminate in undesirable circumstances, or by redirecting the program to code that can rescue program execution.

Exception Handling

To introduce an exception in your program, simply include the code

```
throw an_exception_value;
```

in a line of code, typically an if-statement, where you have detected exceptional behavior. In the example below, `an_exception_value` will be a string literal, although there are better choices. If a `throw` line is encountered, the default behavior is for the program to terminate with a clear error message.

L26x4_throw.cpp

Every thrown exception has a type. There are certain common types of standard types of exceptions that are built into the language, that can be accessed if you `#include <stdexcept>`. These exception types include `std::invalid_argument`, `std::domain_error`, `std::length_error`, `std::out_of_range`, etc.

L26x5_exception.cpp

Exception Handling

The exception mechanism separates **detection** of errors from the **handling** of them. So far, we have not been handling our exceptions – in the absence of handling, programs with thrown exception will simply terminate in a predictable way.

If we wish to handle your exceptions, we can contain them in a `try-catch` statement. When an exception is `thrown` within in a `try` block, the program will stop immediately, and move to the `catch` block (or blocks). The first catch statement which has an argument matching the type of the thrown exception will then have its block executed instead.

The handling can entail lots of different things, depending on the situation: e.g., closing the program with a helpful error message, asking for the reentry of input, or creating a log message.

L26x6_try.cpp

The standard exceptions have a member function, `.what()`, which produces an explanatory message assigned to the exception.

Exception Handling

It is frequently the case that code in a `try` block calls a function, which perhaps itself calls a function, and so on . . . and then code that `throws` an exception is located in one of these function calls.

In that case, the stack frame for the function call containing the `throw` gets removed from the stack, and all of its local variables get destructed. Then, the frame for the function which called *that* one also gets removed from the stack; and this continues until the top of the stack is the frame for the function containing the `try` block. At that point, the appropriate `catch` block gets executed.

This is known as *stack unwinding.* See **L26x7_unwind.cpp** .