C++ Class Design
A Checklist Approach

Alan Bellingham

episys

Who I am

- First language – Algol 60? (1976/77)
- First professional program – Fortran IV (1978)
- Full time programmer since 1985
- At Episys since 1988
- C since 1989
- C++ since 1991

episys

Why a checklist

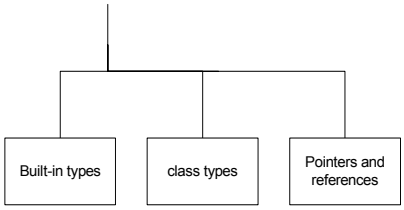
- Pilots use checklists
- Programmers tend to just jump in
- 'things get forgotten'

What we will be covering

- The various types of class objects
- The difference between struct and class
- Exception safety
- The magic four
- Swapping
- The Interface Principle
- Comparison functions
- Arithmetic functions
- Serialisation functions
- Other functions
- The checklist itself

Available Types

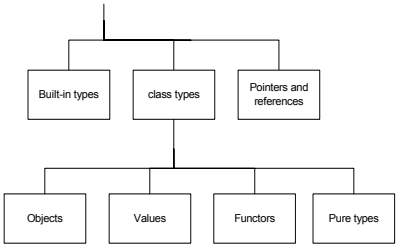
- Built-in types
- User-defined types
- Pointers or references



episys

Class Types

- Two common types: those that have value and object semantics.
- I'm going to break them down into four types



episys

Object types

- One whose identity is relevant
- You consider two objects to be the same when they exist at the same address in memory
- It's important for object types that they do not get copied willy-nilly
- Since the use of standard containers will cause objects to be copied, you should not put objects into these containers :- instead, you will store pointers to them
- An object is quite likely to exist in a polymorphic hierarchy

Value Types

- One in which identity is not relevant
- It doesn't matter whether it's copied, it doesn't matter what its address is, you can quite happily stuff containers full of them
- They're unlikely to be found in a polymorphic hierarchy.
- The built-in types are value types.

Functors

- Also known as a function object
- Functors often have no data at all
- Often have but a single member function – usually operator()(…)
- Examples in the Standard Library include the std::less<> template
- Usually very simple as a result, since their usual purpose is just to carry a single function, and they tend to be defined entirely inline
- They're not usually found in hierarchies

Pure types

- Even simpler than the functor is what I call the pure type.
- The pure type doesn't contain any data.
- It doesn't contain any functions.
- It exists purely for one purpose – to be of a different type from some other pure type.
- If this sounds a little strange, consider exceptions – you catch an exception by its type, and a lot of the time, the only thing that matters is its type.
- Pure types will often exist in hierarchies – you may have a file_error type, and derived types of file_not_found_error, file_locked_error, etc.

Difference between struct and class

- Real difference is default access – struct has public access by default, and class has private access by default

```
class c : base {
   int d;
};
```

- c::d is private member, base is private base class

- ```
  struct s : base {
     int d;
  };
  ```

- s::d is public member, base is public base class
- Mostly be using structs in order to save typing 'public'

Exception safety

- We need to consider the subject of exception safety.
- Exceptions are part of life with C++ - they don't usually occur (hence they are 'exceptional'), but they can do, and you need to write your code knowing this. This affects class design.
- There are two forms of exception safety. These are basic (or weak) exception safety, and strong exception safety
- There is, of course, the case of not being exception safe.


  ```

No exception safety

- Like our pilot getting into the cockpit, starting the engine, and just taking off, not even checking the fuel level
- It'll probably work most of the time, but if an exception is thrown, then there is a good chance that either resources are leaked (this may be memory, database locks, screen windows, or anything), or that an existing object is left in an inconsistent state.

No exception safety (cont)

- Example of a leak:

```
void f()
{
     int * i = new int;
     maybe_throw_an_exception();
     int * j = new int;
     delete i;
     delete j;
}
```

- The first int allocated on the heap has been leaked.

No exception safety (cont)

- Example of inconsistent state:

```
void newstring::copy(string const& rhs)
{
  alloc_len = rhs.alloc_len;
  delete [] data;
  maybe_throw_an_exception();
  data = new char[alloc_len];
  strcpy(data, rhs.data);
}
```

- We still have a pointer to deleted data
- What do we do in the d'tor?

Basic exception safety

- If a function fails because of an exception, then the basic exception safety guarantee tells us that
  - no resources are leaked
  - everything is in a safe and consistent state

Basic exception safety (cont)

- For example:

```
void newstring::copy(string const& rhs)
{
     delete [] data;
     alloc_len = 0;
     data = 0;
     data = new char[rhs.alloc_len];  // throw?
     strcpy(data, rhs.data);
     alloc_len = rhs.alloc_len;
}
```

- Object remains in a stable state – it's safe to call delete on a null pointer. We don't have incorrect alloc_len saying more memory allocated than exists
- We've lost the original content, and failed to copy the new data
- This brings us on to the subject of strong exception safety

Strong exception safety

- The strong exception guarantee says
  - Not only will you not leak any resources
  - Not only will no objects be left in an inconsistent state
  - But also the program state will not change in the event of an exception.
- In database terms - commit/rollback
- Hardest of all, but does have the advantage that provides basic exception safety guarantee as well.
- Works best if you design it in from the start.

Strong exception safety (cont)

- Example:

```
void newstring::copy(string const& rhs)
{
     char* temp = new char[rhs.alloc_len]; // throw?
     strcpy(temp, rhs.data);
     delete [] data;
     data = temp;
     alloc_len = rhs.alloc_len;
}
```

- Here, the only place that can throw is the allocation.
- But if we throw here, nothing has happened. We have *not* changed the object at all, and so we can try again if we like, or we can (virtually) shrug our shoulders and try something else.
- This technique is often known as the 'off-to-the-side' method.
- Do all the dangerous bits first on temporaries of some sort, and then finish off doing only safe operations.

Is it safe?

- These are safe
  - Any assignment or operation on basic types cannot throw an exception
  - delete is always safe
  - (Not quite true. It's possible for a destructor to throw an exception. If it does, it's evil.)
  - Certain functions are safe. E.g., we used strcpy(), which we know does not throw an exception (especially since it's a C library function, and C doesn't have exceptions).
- These are not safe
  - Anything that can allocate memory can also throw an exception
  - A c'tor can throw an exception
  - In general, check the documentation. If it doesn't say it can't, assume it can

The magic four

- Given a simple class or struct like the following

```
struct newstring
{
    char* data;
    int    alloc_len;
};
```

C++ will implicitly generate four functions
  - the default constructor
  - the copy constructor
  - the assignment operator
  - the destructor
- Important to know what compiler will generate, default implementation may not be actually desired.

The default constructor

- Used to initialise an object when you create an instance without passing any parameters. E.g.

```
struct newstring
{
    char* data;
    int    alloc_len;
};

newstring s;
newstring s2 = new newstring;
```

- In both these cases, the default c'tor will be called.

The default constructor (cont)

- What would the compiler generate in this case? Well, the answer is that C++ will default-initialise each member in turn.
- Here, not be terribly useful - default initialisation of built-in types just does nothing (for class types, default initialisation calls the default c'tor).
- Note - this is different from zero initialisation.

The default constructor (cont)

- For this struct, we need a default c'tor, e.g.

```
struct newstring
{
    newstring()
    : data(0)
    , alloc_len(0)
    {}

  private:
    char* data;
    int     alloc_len;
};
```


The default constructor (cont)

- Now consider a different struct.

```
struct simple
{
    char a;
    char b;
};
```

- Does this need you to define a default c'tor?
- Well, no and yes
  - Without a user-defined default c'tor, the contents will be in an unpredictable state
  - Will be in legal state, values of both members will be valid chars
  - Probably better to give it c'tor to set the values to zero

The default constructor (cont)

- I previously mentioned that default initialisation of class types calls the default c'tors for those types

```
struct doublestring
{
    std::string a;
    std::string b;
};
```

- We know that default initialisation of a std::string puts it into a known state. So for this struct, we don't need to provide a default c'tor.
- Or do we?

The default constructor (cont)

- Rule (12.1 [class ctor] para 5) states that the compiler will generate an implicit default c'tor only if there are no user-declared c'tors.
- If you have another c'tor, then you may need to declare and define your own default c'tor. But it can be simple:

```
struct doublestring
{
    doublestring(char* A, char* B)  // user declared
    : a(A), b(B)
    {}

    doublestring()  // no longer implicit
    {}

    std::string a;
    std::string b;
};
```


The default constructor (cont)

- There is also the case that an implicit c'tor cannot be generated:

```
struct simpleref
{
    char& a;
};
```

- The compiler is completely unable to generate code to initialise the member a, and yet it must initialise a.
- In this case, you'll have a compilation failure, so it's not something you can forget to do.

The copy constructor

- The implicit copy c'tor is more likely to 'just work' than the default c'tor, since what it does is simple – it copies each member (and base) in order. (Some so-called experts say that the copying is bitwise – it isn't)
- Not always what you want. Using our newstring example:

```
struct newstring
{
    newstring() …
  private:
    char* data;
    int     alloc_len;
};
newstring a;
newstring b(a);
```

- Member copy will copy pointer, and both objects point at same data. Both think they own it, they'll both try destroying it. Bad news.

The copy constructor

- So, if you have a raw resource pointer, you definitely need to think about writing a copy constructor yourself.

```
newstring::newstring(newstring const& rhs)
: data(new char[rhs.alloc_len])
, alloc_len(rhs.alloc_len)
{
    strcpy(data, rhs.data);
}
```


The assignment operator

- The assignment operator is like the copy constructor, in that it just copies members one by one.
- However, the same rules apply as for the copy constructor, and one extra.
  - Remember, if any of your members are pointers to resources, then you need to define an assignment operator. Otherwise, you have multiple owning pointers to the same resource.
- The extra point is that of exception safety. If a constructor fails, then the object never gets to actually exist. If assignment fails, the left hand side didn't end up getting updated fully.

The assignment operator (cont)

- This is an exception-unsafe way

```
newstring&
newstring::operator=(newstring const& rhs)
{
    delete [] data;
    data = new char[rhs.alloc_len];
    strcpy(data, rhs.data);
    alloc_len = rhs.alloc_len;
    return *this;
}
```

- If an exception is thrown in the new expression, this is bad news

The assignment operator (cont)

- Here is an exception safe method:

```
newstring&
newstring::operator=(newstring const& rhs)
{
     char * temp = new char[rhs.alloc_len];
     strcpy(temp, rhs.data);
     delete [] data;
     data = temp;
     alloc_len = rhs.alloc_len;
     return *this;
}
```


The assignment operator (cont)

- However, a neater way is to use the following idiom:

```
newstring&
newstring::operator=(newstring const& rhs)
{
     newstring temp(rhs);
     swap(temp);
     return *this;
}
```

- Not **necessarily** as efficient as previous method, and requires a swap method, but it's definitely safe, and does not require updating if more members get added.
- Note that the original data pointer has now been put into temp, gets deleted when that goes out of scope.
- It needn't check to see if lhs is the same as rhs.

The destructor

- The implicit d'tor simply default destructs each member (and base). This is bad news if you have raw pointers to resources, because default destruction of a built-in type just does nothing (for a class type, it calls the d'tor for that class). So:

```
struct newstring
{
    newstring()
    : data(0)
    , alloc_len(0)
    {}

  private:
    char* data;
    int     alloc_len;
};
```


The destructor (cont)

- In this case, the memory pointed to by data will be leaked. This then is the major purpose for writing a destructor – to clean up resources (memory, database locks, whatever) previous allocated. In this case:

```
 struct newstring
{
    newstring(); // As previous

    ~newstring()
    {
       delete [] data;
    }

  private:
    char* data;
    int     alloc_len;
};
```


The destructor (cont)

- On the other hand, going back to the doublestring example:

```
struct doublestring
{
    std::string a;
    std::string b;
};
```

  In this case, we may rely on the implicit destructor to do the right thing.

Virtual destructors

- As always, there is a caveat – if the class is meant to be a base class for public inheritance (and you could then have a pointer-to-base actually pointing at a derived), then you should strongly consider making the destructor virtual. In that case, you will have to declare it, and also define it too.
- (A proposal exists that would obviate having to define it as well, but that proposal is at an early stage and it'd be years yet before it got into the next standard and compilers started implementing it.)
- A final note on destructors – if you declare a pure virtual destructor (in other words, one that has the " = 0" syntax), then you still need to define it. In this respect, it's different from other pure virtual functions, because a derived class destructor will call the base class destructor.

The magic four and various class types

- If you don't have any non-static member variables, then you don't normally need to supply any of the four functions – so pure types and functors will be left out.
- Value types will expect to be copied and assigned, so you should consider all four.
- Object types will need a constructor (not necessarily the default constructor) and a destructor.
- Since object semantics are different from value semantics, you will not (usually) want to allow client code to make copies or assignments. The usual way to do this is to declare the copy constructor and assignment operator as private, and not actually define them.

The swap function

- A common extra function is swap(). We've already seen this function being used above.
- Principle is that a pair of objects may have their internal members exchanged. This happens by the exchange of the underlying primitives – an operation known to be exception safe – and must not be able to throw under any circumstances.
- An example

```
void
newstring::swap(newstring& rhs)
{
    std::swap(data, rhs.data);
    std::swap(alloc_len, rhs.alloc_len);
}
```


The swap function (cont)

- If you do this, then it's also worth specialising std::swap for your class:

```
inline template<newstring> void
swap(newstring& lhs, newstring& rhs)
{
    lhs.swap(rhs);
}
```

- This allows standard library algorithms that use std::swap internally to deal with your class more efficiently.

The Interface Principle

- The above specialisation of std::swap is a free function – in other words, it's not bound to a class. Yet is it part of the class interface?
- The answer is yes. Conditionally. Although it's not a member of the class, it comes with the class, and we should therefore consider it as important to the class.
- What it does show is that some methods may be free functions, yet are meaningless without the class. They are frequently functions that require a pair of operands which have equal weight. An example is comparison functions.

Comparison functions

- Value types are often used as keys in std::set or std::map containers.
- For this, we need to be able to sort them, and for this, the standard library algorithms make use of operator<(). The assumption is that if !(a < b) and !(b < a), then the values a and b are equivalent.
- A simple implementation of this would look like this:

```
inline bool
operator<(newstring const& a, newstring const& b)
{
    return (a.compare(b) < 0);
}
```


Comparison functions (cont)

- This relies on an actual member function newstring::compare, which returns the values –x, 0 or +x depending on which value was greater. For our example, we could use strcmp()

```
void
newstring::compare(newstring& rhs)
{
    return strcmp(data, rhs.data);
}
```

- Once we have that member function, it's easy enough to provide other comparison functions.

Arithmetic functions

- Some value types work like basic numeric types – they're expected to be multiplied, etc. In this case, you will want to look at which functions are provided, and how.
- The basic arithmetic operators are +, -, * and /, but you also need to consider the assignment equivalents.
- For instance, let's consider the + operation. If you provide this, then you'll probably want to provide += as well:

```
struct s
{
  s& operator+=(s const& rhs)
  {
    // actual implementation
    return *this;
  }
  // … actual data
};
```


Arithmetic functions (cont)

```
s&
operator+(s const& lhs, s const& rhs)
{
    s temp(lhs);
    return temp += rhs;
}
```

- Again, we see the use of a free function as part of the interface, and that free function makes use of a member function to do its work.
- Could do it the other way around, but better to make the free function the one that treats both arguments the same
- When you change the behaviour of operator+=(), the behaviour of operator+() also changes.

Serialisation functions

- In this case, you usually want to serialise to or from a basic stream. This way, you can supply a file stream, or a string stream, or some specialised stream of your own that the items can be written to, or read from. Again, we'll use a free function and a member:

```
struct s
{
  std::ostream& writeOn(std::ostream& os) const {
    // actual implementation
    return os;
  }
};

std::ostream&
operator<<(std::ostream& os, s const& S) {
  return S.writeOn(os);
}
```


Serialisation functions (cont)

- Note that we use the most abstract stream type possible.
- If s is the base for a class hierarchy, you can make the member function a virtual function overridden in derived classes. The free function need not repeated for each derived class – because the virtual function will dispatch to the correct implementation as required.
- Presence of an output serialisation function implies the presence of an input one, but doesn't require it – you may be serialising purely for logging purposes.

Other functions

- I've not covered all common functions. Some I've left out are:
  - Factory functions
    - Usually a static member function
    - Implies private c'tor
    - Either an AddRef/Release (reference counted) or just a destroy member
  - clone() (always virtual)
  - operator new
    - Which requires operator delete
    - And operator new[]
    - And operator delete[]
- There are other common functions you can come up with which are missing here. If so, I'd be happy to hear of them – this is the v1.0 of this talk, and I'd like to expand it in future

The checklist itself

- The default constructor
  - Is the compiler generated one sufficient or do we need to write it ourselves?
- The copy constructor
  - Is the compiler generated one sufficient, or do we need to write it ourselves?
  - Or should we disable it?
- The assignment operator
  - Is the compiler generated one sufficient, or do we need to write it ourselves?
  - Or should we disable it?
- The destructor
  - Is the compiler generated one sufficient, or do we need to write it ourselves?
  - Should it be virtual?

The checklist itself (cont)

- The swap function
  - Is this a value type? If so, a std::swap specialisation is a good idea.
- Comparison functions
  - Is this type ever going to be used as a map or set key? If so, operator<()
- Arithmetic equivalents
  - Is this a value type? Does it have arithmetic-like functionality?
- Serialisation functions
  - Do we want the ability to serialise the class?
  - Do we want input serialisation as well as output?

Conclusion

- So there we are. We have a list of functions that we need to consider when designing a class. Of course, a class that we design will usually have other functions – otherwise it will quite likely be lacking the functionality that we wanted in the first place. But by using our checklist, we should ensure that we don't have unwanted behaviour as well.
- Questions?