G52CPP
C++ Programming
Lecture 19

Dr Jason Atkin

http://www.cs.nott.ac.uk/~jaa/cpp/g52cpp.html

This lecture and beyond

- This Lecture
  - Multiple Inheritance
- This afternoon, 2pm – how to create programs fast
  - Optional
- Thursday 2nd May, 4pm, Lecture 20
  - Wrapping up (incl smart pointers)
- Friday 3rd May, 10am, Revision Lecture
  - Revision and exam strategy
- Friday 3rd May, 2pm - Optional
  - Any questions / examples
  - What do you want to ‘revise’?

When is a duck an instrument?

Multiple Inheritance

Multiple inheritance

- In Java you can **`implement`** multiple interfaces, but only **`extend`** one class
- **In C++ you can inherit from (extend) multiple classes**
- At times it makes sense to inherit from multiple base classes
  - Maybe something can be both a duck and an instrument?
  - You inherit all of the behaviour (i.e. function **implementations**), not just the interface
- **But be careful of multiple inheritance**
  - **There are dangers, and confusing elements**
  - **There may be easier ways (e.g. composition)**

What re-use options are there?

- There are other ways to support re-use:

1. Composition/aggregation
   - Models the ‘**has a**’ or ‘**is a part of**’ relationship
   - ***Composition*** is a stronger form
     - The ‘part’ only exists while the containing class exists
2. Inheritance
   - ‘**Is a**’ or ‘**is a type of**’
   - **Implementation**: Make the ‘type of’ a sub-class
3. Uses / association
   - **Implementation**: Maintain a pointer or reference between them, to get to the other object
     - Create the other object separately, then set pointer to it
     - Other object is separate – needs to be destroyed separately

Musical Duck

Base classes

```
class Duck
{
public:
        // Constructor
        Duck( int weight = 1 )
                : weight(weight)
        {}

        // Get the weight
        int GetWeight() const
        { return weight; }

//protected:
        int weight;
};
```

Two classes.
Both have a weight,
one has a volume.

```
class Instrument
{
public:
   Instrument(  int weight = 1,
                int volume = 1 )
      : weight(weight)
      , volume(volume)
   {}

   int GetVolume() const
   { return volume; }

   int GetWeight() const
   { return weight; }

//protected:
   int volume;
   int weight;
};
```

Musical Duck 1 : Composition

```
class MusicalDuck1
{
public:
   // Constructor
   MusicalDuck1(
     int weight = 1,
     int volume = 2 )
   : d(weight)
   , i(weight,volume)
   {}

   // Contains a ‘Duck’
   Duck d;

   // Contains ‘Instrument’
   Instrument i;

   // Get instrument volume
   int GetVolume() const
   { return i.GetVolume(); }

   // Get weights
   int GetInstWeight() const
   { return i.GetWeight(); }

   int GetDuckWeight() const
   { return d.GetWeight(); }
   };
};
```

Data from contained objects is available to the container object. Have to expose any methods manually.

Musical Duck 2 : Inheritance

```
class MusicalDuck2
: public Duck
, public Instrument
{
public:
    // Constructor
    MusicalDuck2(
        int weight = 1,
        int volume = 2 )
    : Duck(weight)
    , Instrument(weight,volume)
    { }
```

```
// GetVolume() is inherited
// and available

// GetWeight() is inherited
// (twice) and available

};
```

GetVolume() is available automatically

GetWeight() is available from both base classes (i.e. twice)
How do we differentiate between them?

Musical Duck 1 : Composition

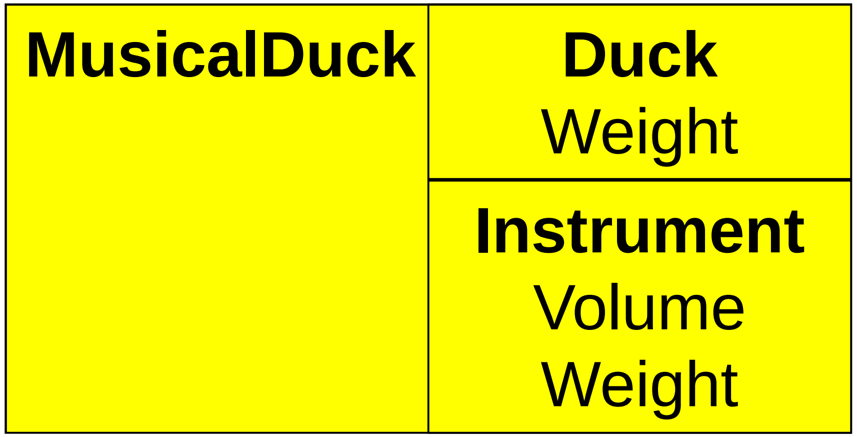
```
class MusicalDuck1
{
public:
   // Constructor
   MusicalDuck1(
      int weight = 1,
      int volume = 2 )
   : d(weight)
   , i(weight,volume)
   { }

// Contains a ‘Duck’
Duck d;
// Contains ‘Instrument’
Instrument i;
…
};
```

```
MusicalDuck1 mduck1;
printf( "Musical duck at %p\n",
       &mduck1 );
printf( "Duck at %p\n",
       &mduck1.d );
printf( "Duck.weight at %p\n",
       &mduck1.d.weight );
printf( "Instrument at %p\n",
       &mduck1.i );
printf("Instr.Volume at%p\n",
       &mduck1.i.volume );
printf( "Instr.Weight at %p\n",
       &mduck1.i.weight );
```

```
Musical duck at 0x22ccd0
Duck at 0x22ccd0
Duck.weight at 0x22ccd0
Instrument at 0x22ccd4
Instr.Volume at 0x22ccd4
Instr.Weight at 0x22ccd8
```

Musical Duck 1 : Composition

```
class MusicalDuck1
{
public:
   // Constructor
   MusicalDuck1(
      int weight = 1,
      int volume = 2 )
   : d(weight)
   , i(weight,volume)
   { }

// Contains a ‘Duck’
Duck d;
// Contains ‘Instrument’
Instrument i;
…
};
```



MusicalDuck

Duck
Weight

Instrument
Volume
Weight

```
Musical duck at 0x22ccd0
Duck at 0x22ccd0
Duck.weight at 0x22ccd0
Instrument at 0x22ccd4
Instr.Volume at 0x22ccd4
Instr.Weight at 0x22ccd8
```

Musical Duck 2 : Inheritance

```
class MusicalDuck2
: public Duck
, public Instrument
{
public:
   // Constructor
   MusicalDuck2(
        int weight = 1,
        int volume = 2 )
   : Duck(weight)
   , Instrument(weight,volume)
   { }

…
};
```

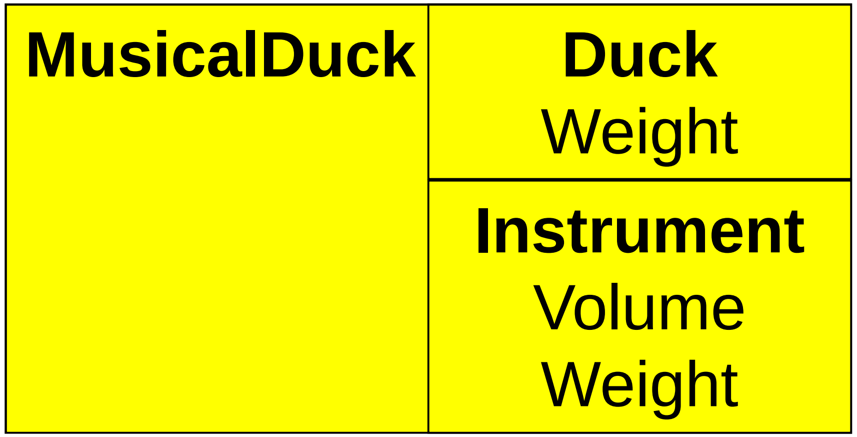
```
MusicalDuck2 mduck2;
printf( "Musical duck at %p\n",
   &mduck2 );
printf( "Duck at %p\n",
   (Duck*)(&mduck2) );
printf( "Duck.weight at %p\n",
   &mduck2.Duck::weight );
printf( "Instrument at %p\n",
   (Instrument*)(&mduck2) );
printf( "Instr.Volume at %p\n",
   &mduck2.volume );
printf( "Instr.Weight at %p\n",
   &mduck2.Instrument::weight );
```

```
Musical duck at 0x22ccd0
Duck at 0x22ccd0
Duck.weight at 0x22ccd0
Instrument at 0x22ccd4
Instr.Volume at 0x22ccd4
Instr.Weight at 0x22ccd8
```

Important notes:

Important notes:

- The base-class information is contained within the sub-class structure
- Casting a pointer can change the address: **`(Instrument*)(&mduck2)`**
- Composition may be easier in many cases
- Main difference is that you have to wrap/expose the functions yourself



MusicalDuck

Duck
Weight

Instrument
Volume
Weight

If data or methods are available from multiple base classes you need to **disambiguate**

Use scoping to do this:

```
&mduck2.Duck::weight
&mduck2.Instrument::weight
```

Casting Pointers and References

- **I used C-style casting to keep the code short**
  - **DO NOT DO THIS!!!**
- Use **`static_cast`** (for sub-class to base class) or **`dynamic_cast`** (for base class to sub-class)
  - Dynamic cast will check (at runtime) that the pointer really is to an object of that type
- **IMPORTANT:** If you cast pointers or references when multiple inheritance is being used, then addresses may change
  - Normally, casting a pointer just changes the type, but leaves the address unchanged
  - If you go to or from a second (or later) base class, the address (pointer value) will change!
  - If you go back again (to sub-class), the pointer value changes back again (use dynamic cast if necessary, to check the type)

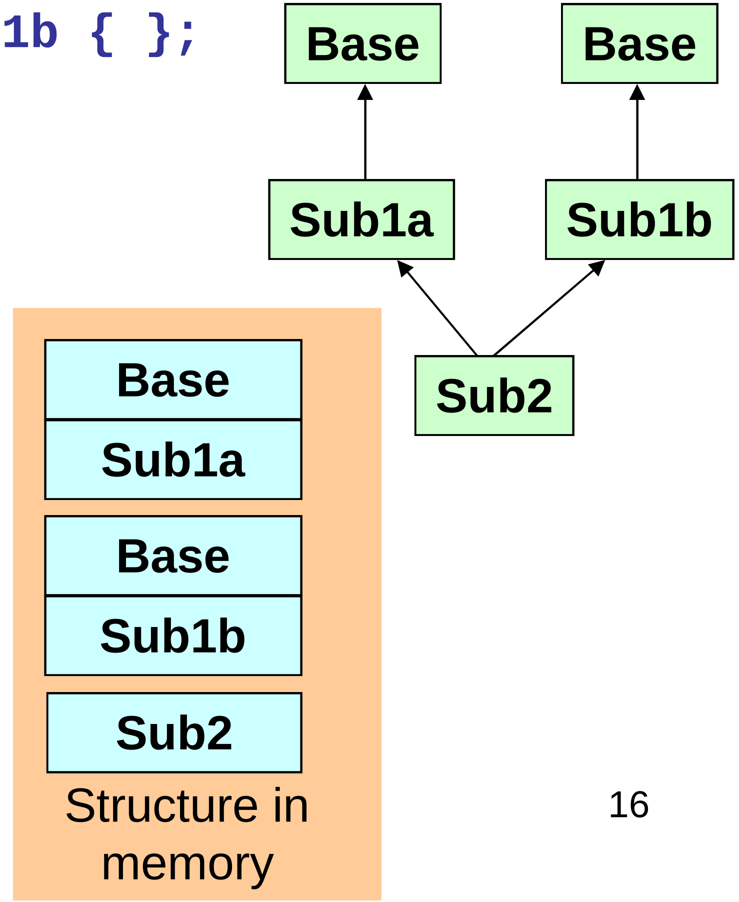
Shared base classes

Shared base classes

Sub1 and **Sub2** each have a copy of **i**, which they inherit. **Sub2** has 2 copies

```
#include <cstdio>

struct Base { int i; };
struct Sub1a : public Base { Sub1a() {i=1;} };
struct Sub1b : public Base { Sub1b() {i=2;} };
struct Sub2 : public Sub1a, public Sub1b { };

int main()
{
   printf( "Sizes: %d %d %d %d\n",
        sizeof(Base), sizeof(Sub1a),
        sizeof(Sub1b), sizeof(Sub2) );

   Sub2 ob;
// printf( "%d\n", ob.i ); WRONG!!!
   printf( "%d\n", ob.Sub1a::i );
   printf( "%d\n", ob.Sub1b::i );
};
```

```
Output:
4 4 4 8
1
2
```

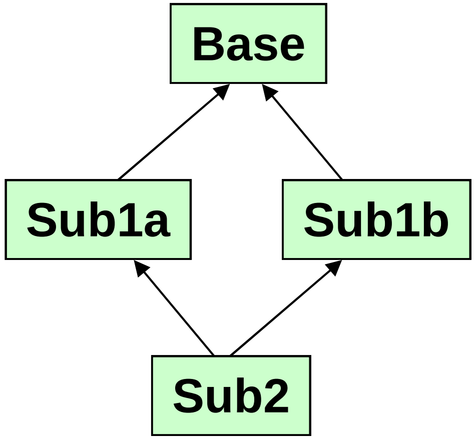


Structure in memory

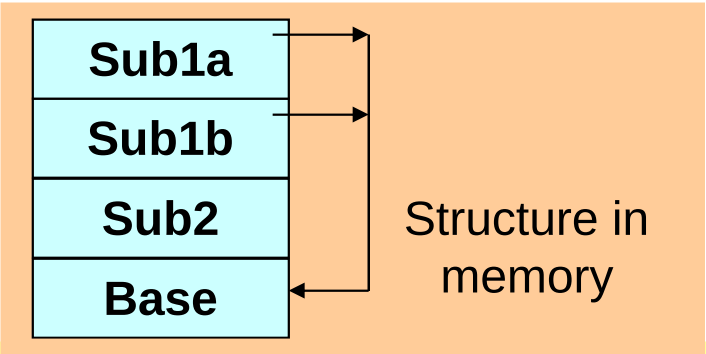
Virtual base classes

```
#include <cstdio>

struct Base { int i; };
struct Sub1a : virtual public Base { Sub1a() {i=1;} };
struct Sub1b : virtual public Base { Sub1b() {i=2;} };
struct Sub2 : public Sub1a, public Sub1b {};

int main()
{
   printf( "Sizes: %d %d %d %d\n",
        sizeof(Base), sizeof(Sub1a),
        sizeof(Sub1b), sizeof(Sub2) );

   Sub2 ob;
   printf( "%d\n", ob.i );
   printf( "%d\n", ob.Sub1a::i );
   printf( "%d\n", ob.Sub1b::i );
};
```

```
Output:
4 8 8 12
2
2
2
```



Can now use ob.i (only one copy)



Structure in memory

Note: Size increased by 4 bytes, for the pointer to virtual base class

Safe multiple inheritance and alternatives

Multiple inheritance dangers

- Be careful if you use multiple inheritance
- Beware of:
  - **Inheriting the same names from multiple base classes**
  - **Inheriting the same base class twice, through two different intermediate classes**
- To resolve the problem:
  - Use scoping operator **::** to dis-ambiguate
  - Or use virtual base classes, to keep one copy
  - Or ensure that only one base class has any data, or any non-abstract methods …

Abstract/pure-virtual base class

```
class PureVirtual
{
  virtual void func1() = 0;
  virtual int func2() = 0;
  virtual double func3(int,double) = 0;
};
```

No implementation is given for any of these functions They must be implemented in **concrete** sub-classes

- No member data is specified
- All functions are pure virtual (i.e. abstract, `= 0`)
  - MUST be implemented in any concrete sub-class
- **This class acts like a Java interface and can be used in the same way**

Should I Use Inheritance?

- Inheritance says this object IS an object of the other type, not just that they have SOME commonality
- Do not assume that inheritance is always the answer
  - Be sure that you really want ‘is-a’ and not ‘has-a’
  - Aggregation or composition are often better options if you just want to reuse some code
  - Although you then have to re-implement function wrappers
- Do not assume that multiple inheritance is needed
  - It is **never** necessary (but is sometimes useful)
- Do you need to treat different sub-class types as the base class? (i.e. need to model ‘is-a’?)
- To be safe, adopt the Java way of having only one base class any data or function implementations
  - i.e. all but one base class is an ‘interface’

Moving on…

Quick creation of C++ programs

- This afternoon I will (optionally for you) show you how to generate code and programs easily using MFC, the application wizard and the class wizard for Windows program development:
  - A Windows application with a ribbon
  - A Single Document Interface application
  - A Multi-Document Interface application
  - A Dialog-based application (easy to create and edit with very little knowledge)
- Even though it’s now over 20 years old, if you want to create a windows application I suggest reading up on MFC. It is easy to do basics with, with low overheads
- Note: Microsoft are pushing .NET now instead, with ‘managed C++ code’ – which makes sense
- Note that my views may be unusual: I tend to use C++ for the low-level or fast work and other languages otherwise, so managed code is of less use to me

Next lecture and beyond

- This afternoon, 2pm – how to create programs fast
  - Optional

- Thursday 2nd May, 4pm, Lecture 20
  - Wrapping up (incl smart pointers)

- Friday 3rd May, 10am, Revision Lecture
  - Revision and exam strategy

- Friday 3rd May, 2pm - Optional
  - Any questions / examples
  - What do you want to ‘revise’?