G52CPP
C++ Programming
Lecture 13

Dr Jason Atkin

Last Lecture

```
#include <string>
#include <iostream>

using namespace std;

int main()
{
   string s1( "Test string" );
   int i = 1;

   cin >> i;

   cout << s1 << " " << i << endl;

   cerr << s1.c_str() << endl;
}
```

Header files for string and i/o

Look in std namespace
for the names which follow
e.g. cin, cout, string

Overloaded operator - input

Overloaded operator - output

Convert string to const char*

This lecture

- Inheritance
- Virtual functions

When is a duck a duck?

and when is it a
musical
instrument

What is a duck

- Which question defines a duck?
  - Does it have a beak?
  - Does it ‘quack’?
  - Does it fly?
  - Does it look like a duck?
- To be a duck, what does it need to do?
  - We need to understand what we mean by a duck **in the current context**
- In program terms, the properties are defined by the operations and attributes
  - So know what these are!

What is inheritance?

- Inheritance models the ‘is-a’ relationship
  - i.e. the sub-class object **is-a** type of base class object
  - **Be sure that inheritance really is what you want before you use it**
- Define a new class (sub-class/derived class) in terms of a current class (superclass/base class)
  - Take the general class and extend it
- Why do it?
  - Get all member functions and data of the base class, for free, without having to (re-)write them yourself
- How can we extend it?
  - Add functionality?
  - Change or refine functionality? (within reason)
  - Remove functionality? (and still work as base class?)

Using inheritance

- Use the `:` notation (after the class name)

```
class MyClass : public MySuperClass
{
}
```

Maximum access level,
assume `public` for the moment

- Equivalent of Java's '`extends`', i.e.:

```
class MyClass extends MySuperClass
```

- A class can have multiple base classes
  - See later lecture – some complexities

Inheritance

- Define a new class (sub-class or derived class) in terms of a current class (super/base class)
- Inheritance models the ‘**is-a**’ relationship
  - If we have a class which models a bird,
  - And we want a class for a specific species of bird
  - Then we can take the general class and extend it

```
class Bird
{
public:
  void eat();
  void sit();
};
```

```
class FlyingBird
: public Bird
{
public:
    void fly();
};
```

Note: Function implementation is probably in associated **.cpp** files

A new access type: protected

- Reminder: `public` access
  - Anything can access the member
- Reminder: `private` access
  - Only class members can access the members
  - NOT even sub-class members
  - The main reason for being a class member
- New idea: `protected` access
  - Like `private` but also allows sub-class members to access the members
- Note: No concept of (Java-like) package-level access in C++

Base-class access rights

Think: `public -> protected -> private`

`class MyClass : public MySuperClass`

- “At most **`public`** access” (i.e. no change)
- **`public`**/**`protected`** members are inherited with the same access as in the base class
- The most common form of inheritance

`class MyClass : protected MySuperClass`

- “At most **`protected`** access”
- **`public`**/**`protected`** members are inherited as **`protected`** members of the sub-class

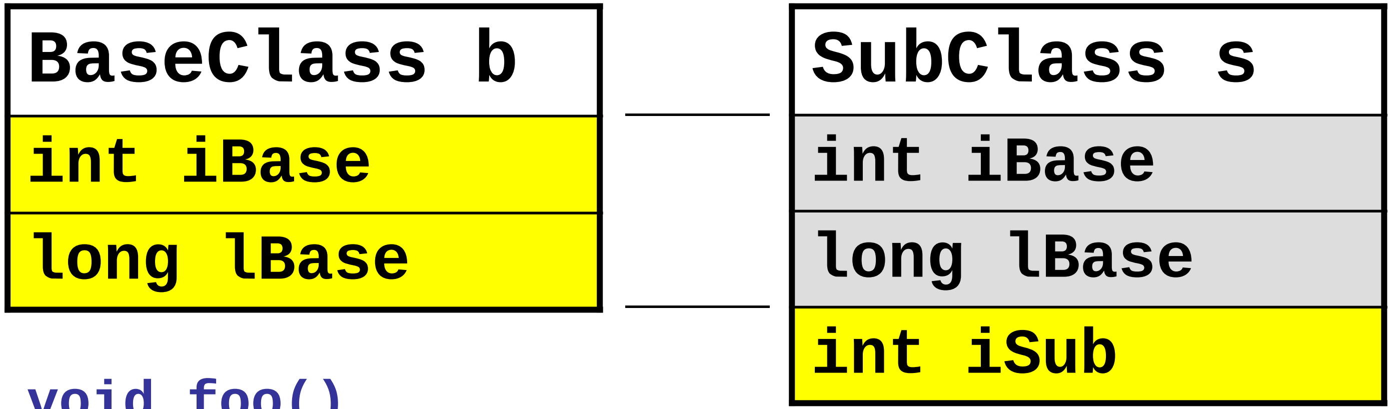
`class MyClass : private MySuperClass`

- “At most **`private`** access”
- **`public`**/**`protected`** members are inherited as **`private`** members of the sub-class
- ***Consider whether composition is more appropriate***

Base class and derived class

```
class BaseClass
{
public:
   int iBase;
   long lBase;
};
```

```
class SubClass
: public BaseClass
{
public:
   int iSub;
};
```



BaseClass b

int iBase

long lBase

SubClass s

int iBase

long lBase

int iSub

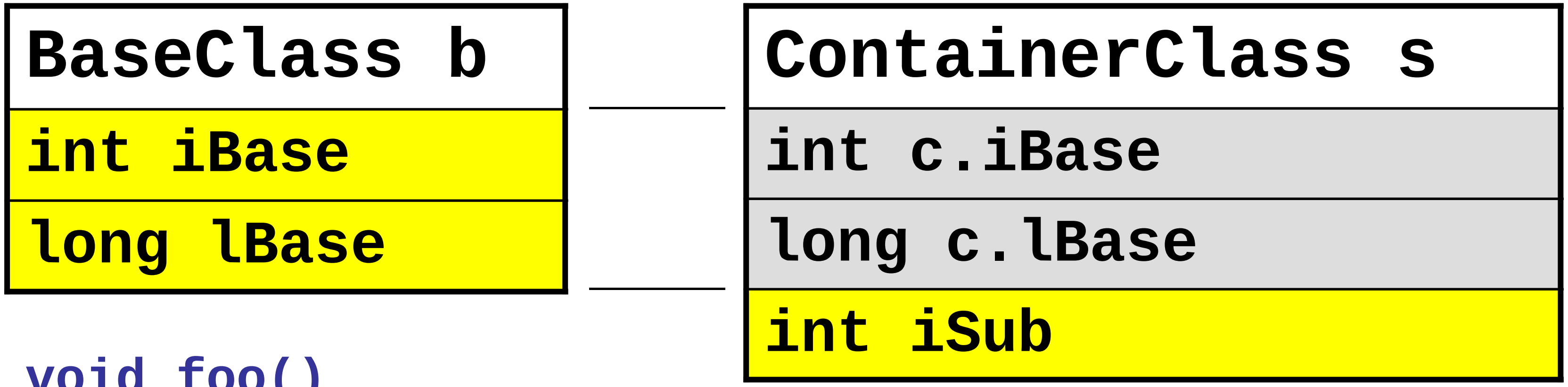
```
void foo()
{
   BaseClass b;
   SubClass s;
}
```

Simple single-inheritance: the base class part appears inside the sub-class

Comparison : composition

```
class Class1
{
public:
   int iBase;
   long lBase;
};
```

```
class ContainerClass
{
public:
   Class1 c;
   int iSub;
};
```



BaseClass b

int iBase

long lBase

ContainerClass s

int c.iBase

long c.lBase

int iSub

```
void foo()
{
   Class1 b;
   ContainerClass s;
}
```

Simple composition: the contained class part appears inside the containing class

Example: overriding methods

```
class BaseClass
{
public:
   char* foo() { return "BaseFoo"; }
   char* bar() { return "BaseBar"; }
};
class SubClass : public BaseClass
{
public:
   char* foo() { return "SubFoo"; }
   // No override for bar()
};
int main()
{
   SubClass* pSub = new SubClass;
   printf("foo=%s bar=%s\n", pSub->foo(), pSub->bar() );
   delete pSub;
}
```

`bar()` from base class is available unchanged in sub-class

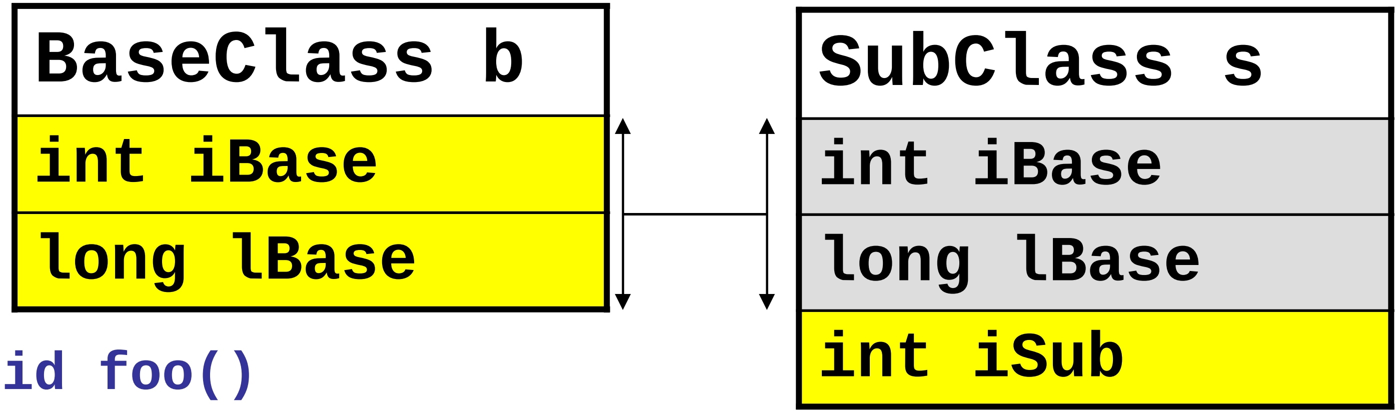
sub-class “overrides” (replaces) the `foo()` function from the base class

Using dynamically allocated memory

Sub-class objects ARE base class objects

```
class BaseClass
{
public:
   int iBase;
   long lBase;
};
```

```
class SubClass
: public BaseClass
{
public:
   int iSub;
};
```



BaseClass b

int iBase

long lBase

SubClass s

int iBase

long lBase

int iSub

```
void foo()
{
   SubClass* pSub = new SubClass();
   BaseClass* pBase = pSub; // POINTERS!
   // Same applies to references!
   delete pSub;
}
```

Question

Consider functions which exist in the base-class, and are overridden in the sub-class

When called using a base class (type) pointer (or reference), which of the following is true?

a) The sub-class versions of functions are used (because the object is really of the sub-class type) [Note: this is the usual case in Java]

b) The base-class versions of functions are used (because the pointer type is used to determine the function to use)

Example follows, on the next slide, for clarity

Example: Overridden function

```
class BaseClass
{
public:
   char* foo() { return "BaseFoo"; }
};

class SubClass : public BaseClass
{
public:
   char* foo() { return "SubFoo "; }
};

int main()
{
   SubClass* pSub = new SubClass;
   BaseClass* pSubAsBase = pSub; // Pointers

   printf( "foo  S=%s SaB=%s\n",
             pSub->foo(), pSubAsBase->foo() );
   delete pSub;
}
```

Question:

When functions are called from base-class pointers/references, which functions are called?

i.e. what do these do?

pSub->foo()

pSubAsBase->foo()

Object is of type SubClass
Pointer is of type BaseClass

Answer to the question

You can choose which you want to apply (by making the function `virtual` or not)

a) The functions in the sub-class are used

(because the object is really of the sub-class type)

This method applies if the functions are **`virtual`**

b) The functions in the base-class are used (because the pointer type is used to determine the function to use)

This method applies if **`virtual`** is not specified

Example: virtual functions

```
class BaseClass
{
public:        char* foo() { return "BaseFoo"; }
               virtual char* bar() { return "BaseBar"; }
};
```

```
class SubClass : public BaseClass
{
public:        char* foo() { return "SubFoo"; }
               virtual char* bar() { return "SubBar "; }
};
```

```
int main()
{
       SubClass*  pSub  = new SubClass;
       BaseClass* pSubAsBase = pSub;
       printf( "pSubAsBase->foo() %s\n", pSubAsBase->foo() );
       printf( "pSubAsBase->bar() %s\n", pSubAsBase->bar() );
       delete pSub;
}
```

Calling base-class functions

- If a function is virtual, you can still call the base class version from the sub-class version
  - Useful so that you don’t need to repeat code
- From Java you can call the (immediate) super-class version of a method from within a method
  - Uses the `super.foo()` notation
- The C++ version is more flexible…
  - You can call any base-class version, not just the *immediate* base-class
- C++ uses the scoping operator `::`
  - Example…

Example of scoping operator

```
class Base
{
public:
   virtual void DoSomething()
   { x = x + 5; }
private:
   int x;
};
class Derived : public Base
{
public:
   virtual void DoSomething()
   {
       y = y + 5;
       Base::DoSomething();
   }
private:
   int y;
};
```

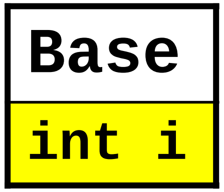
Base class version of **`DoSomething()`** adds 5 to x

Derived class version of **`DoSomething()`** adds 5 to y THEN calls the base class version, which will add 5 to x

This EXPLICITLY calls the base-class version

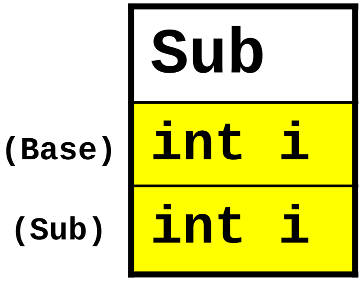
Reminder: scoping

```
#include <cstdio>
int i = 1; // Global

struct Base
{
  int i;

  Base()
  : i(3)
  {}
};

struct Sub : public Base
{
  int i;

  Sub()
  : i(2)
  {}

  void modify()
  {
      int i = 7;   // Local
      ::i = 4;     // Global
      Sub::i = 5;  // Sub's i
      Base::i = 6; // Base's i
  }
};

int main()
{
  Sub s;
  printf( "%d %d %d\n",
      i, s.i, s.Base::i );
  s.modify();
  printf( "%d %d %d\n",
      i, s.i, s.Base::i );
  return 0;
}
```



Base

int i



Sub

(Base) int i

(Sub) int i

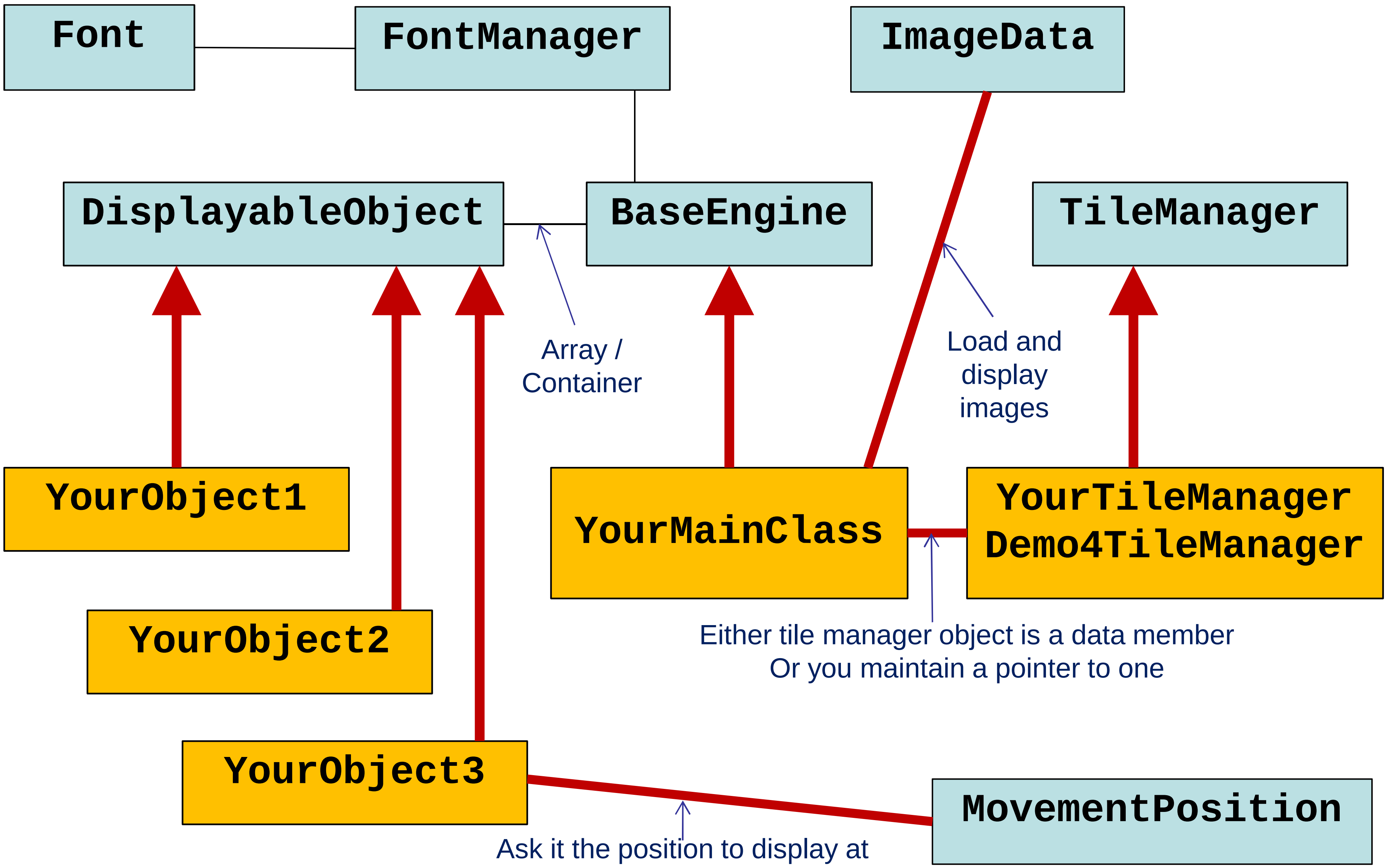
Reminder: The scoping operator

- You can use the scoping operator to call global functions or access global variables
  - use `::` with nothing before it
- Also used to denote that a function is a class member in a definition, e.g.

```
void Sub::modify() { … }
```

- Left of scoping operator is
  - **blank** (to access a global variable/function)
  - **class name** (to access member of that class)
  - **namespace name** (to use that namespace)

The Coursework Framework



Either tile manager object is a data member
Or you maintain a pointer to one

Ask it the position to display at

Next lecture

- Discussion of the coursework requirements

- How to make C++ programs for windows really quickly
  - Not relevant for the coursework though