G52CPP
C++ Programming
Lecture 10

Dr Jason Atkin

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

Last lecture

- Constructors
  - Default constructor – needs no parameters
- Default parameters
- Inline functions
  - Like safe macros in some ways
- Function definitions outside the class declaration
  - i.e. .h files and .cpp files

This lecture

- new and delete
- Inheritance
- Virtual functions

new and delete

For reference purposes

We will see plenty of examples of use over the next few weeks

new vs malloc

```
MyClass* pOb = new MyClass;
```

- **`new`** knows how big the object is
  - No call to **`sizeof()`** is needed (unlike **`malloc()`**)
- **`new`** creates an object (and returns a ***pointer***)
  - Allocates memory (probably in same way as **`malloc()`**)
- **`new`** knows how to create the object in memory
  - C++ objects can consist of more than the visible data members (an example later, with hidden vtable ptrs)
- **`new`** calls the constructor (**`malloc()`** will not!)
- **`new`** throws an exception (**`bad_alloc`**) if it fails
  - By default, unless you tell it not to (e.g. **`new(nothrow) int`**)
  - Some older compilers may return NULL – but new ones should not (**`malloc()`** returns NULL on failure)

delete

```
MyClass* pOb = new MyClass;

delete pOb;
```

- **`delete`** destroys an object
  - It cares about the object type
  - Calls the destructor of the class it thinks the thing is (using pointer type) **and then** frees the memory

delete, new[] and delete[]

- **`new`** and **`delete`** have a **`[]`** version for creating and destroying arrays
  - Default constructor is called for the elements
    - Same as for arrays created on the stack

- You MUST match together:

  `new` and **`delete`**

  `new []` and **`delete []`**

  `malloc()` and **`free()`**

Example : new and delete

```
class MyClass
{
public:
        int ai[4];
        short j;
};
int main()
{
  MyClass* pOb = new MyClass;
  MyClass* pObArray = new MyClass[4];

  pOb->ai[2] = 3;
  pObArray[3].j = 5;
  pObArray[1].ai[3] = 5;

  delete pOb;
  delete [] pObArray;
  return 0;
}
```

Can pass values to constructor here inside **( )**

<u>Could</u> use empty **()** with **new** to pass no parameters

Uses default constructor for each object in array

delete [] to match **new []**

Can **new/delete** basic types

```
int* pInt = new int;
int* pIntArray = new int[50];
int* pInt2 = new int(4);

*pInt = 65;
pIntArray[1] = 9;

delete pInt;
delete [] pIntArray;
delete pInt2;
```

Array of 50 elements NOT PARAM FOR CONSTRUCTOR!

Pass an initial value of 4 to ‘constructor’ NOT AN ARRAY

`malloc()` just declares memory, and you tell the compiler to treat it as if it was a struct, array or type

`new` actually constructs something of that type

Comments on **`delete`**

- You **MUST** **`delete`** anything which you create using **`new`**

```
MyClass* pOb1 = new MyClass;
delete pOb1;
MyClass* pOb2 = new MyClass(5);
delete pOb2;
```

- You MUST **`delete`** any arrays which you create using **`new … []`**

```
MyClass* pObArray = new MyClass[6];
delete [] pObArray;
```

- You MUST **`free`** any memory which you **`malloc`**/**`alloc`**/**`calloc`**/**`realloc`**

Pointer problems

- The same kind of problems can occur with **`new`** and **`delete`** as with **`malloc()`** and **`free()`**:
  - Memory leak (leaking memory – less available)
    - Not calling **`delete`** on all of the objects or arrays that you **`new`**
  - Dereferencing a pointer after you have freed/deleted the memory it points to
    - Effects may not be immediately obvious!
  - Calling **`delete`** multiple times on same pointer
- Plus some new ones:
  - Not matching the array and non-array **`new`** & **`delete`**

```
int* p = new int; delete [] p; // WRONG!
int* p = new int[4]; delete p; // WRONG!
```

- And references don't help
  - The same problems with references as with pointers

Constructors and destructors

- Constructor is called:
  - When objects are created on the stack
  - Upon creation of globals/static locals
  - When new is used to create an object
  - **NOT called when `malloc()` is called**
- Destructor is called:
  - When objects on the stack are destroyed
  - When globals and static locals are destroyed
  - When **`delete`** is used to destroy an object
  - **NOT called when `free()` is called**
- **`malloc()`** and **`free()`** do not create objects
  - They allocate memory and **you** tell the compiler to treat the memory as if it held a struct/object/array/etc
  - Safe for C-style structs but **not safe for C++ style structs and classes**

What **new** really does

When you call new:

- e.g. using `MyClass* ob = new MyClass;`

the compiler generates code to:

- Call **`operator new`** (to allocate the memory)
  - You can change the way that new allocates memory
    - Look up “operator new” for details
  - You can create an object at a specific memory location
    - Look up “placement new” for details
- Create the object
  - Including hidden data (e.g. **`vpointer`**s)
  - Constituents get constructed first
    - i.e. base class first, aggregated objects first
  - Uses the initialisation list to provide initial values
- Calls the constructor code

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 lecture 19 – 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*

- Note: You do NOT get access to `private` base-class members, whatever you use

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 : aggregation

```
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 aggregation: 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

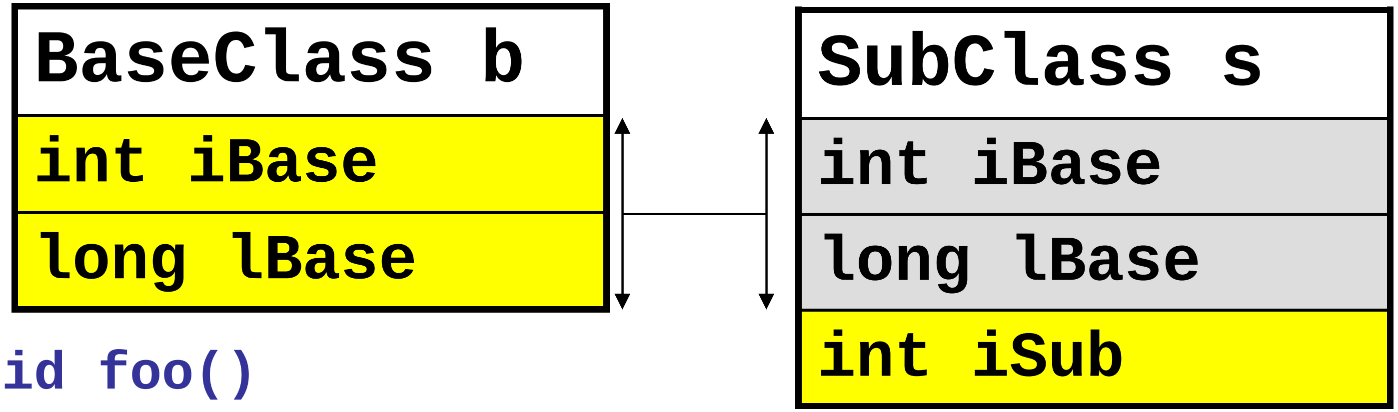
sub-class "overrides" (replaces) the foo() function from the base class

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;
};
```




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

Next lecture

- The **`this`** pointer and static members
- References
  - – Act like pointers
  - – Look like values
- More **`const`**
  - And **`mutable`**