G52CPP
C++ Programming
Lecture 12

Dr Jason Atkin

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

Last lecture

- **`this`** and **`static`** members

- References
  - Act like pointers
  - Look like values

- More **`const`**
  - And **`mutable`**

This lecture

- Function pointers
- Virtual and non-virtual functions

Function pointers

Function pointers

- Functions are stored in memory
- You can ask for the address of them
- You can store these in function pointers
- Used a lot in low-level programming
  - Operating system calls often want function pointers
- And for ‘callback functions’
  - Allows something to call you back
  - ‘Call this function when an event happens’
  - Event driven programming is VERY common

Simplest functions… some code

```
void f1() { printf( "f1(); " ); }
void f2() { printf( "f2(); " ); }

int main()
{
   void (*g1)() = NULL;
   void (*g2)() = NULL;

   printf("Test 1: " );
   g1 = &f1;
   (*g1)();
   g1();     // Short way

   printf("\nTest 2: " );
   g2 = &f2;
   (*g2)();
   g2();     // Short way
```

Note: Not function calls, but strings!

g1 and **g2** are function pointers of type **void function()**

Note: The **&** is optional. As for arrays. Function name is a pointer to it.

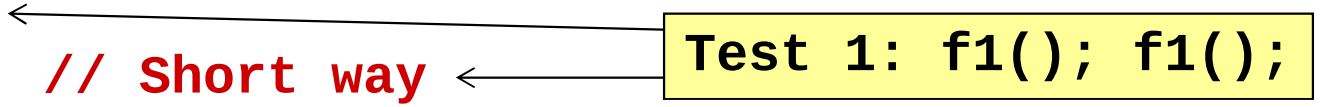
Simplest functions… the output

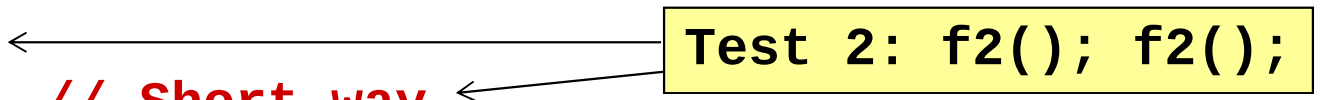
```
void f1() { printf( "f1(); " ); }
void f2() { printf( "f2(); " ); }

int main()
{
   void (*g1)() = NULL;
   void (*g2)() = NULL;

   printf("Test 1: " );
   g1 = &f1;
   (*g1)();
   g1();     // Short way

   printf("\nTest 2: " );
   g2 = &f2;
   (*g2)();
   g2();     // Short way
```





Assignment of pointers

```
void f1() { printf( "f1(); " ); }
void f2() { printf( "f2(); " ); }

int main()
{
   void (*g1)() = NULL;
   void (*g2)() = NULL;

   g1 = &f1;
   g2 = &f2;

   // Assignment of function pointers
   printf("\nTest 3: " );
   g2 = g1;
   (*g2)();
   g2();     // Short way
```

Assignment

```
void f1() { printf( "f1(); " ); }
void f2() { printf( "f2(); " ); }

int main()
{
   void (*g1)() = NULL;
   void (*g2)() = NULL;

   g1 = &f1;
   g2 = &f2;

   // Assignment of function pointers
   printf("\nTest 3: " );
   g2 = g1;
   (*g2)();
   g2();      // Short way
```

Test 3: f1(); f1();

Returning values

```
int f3() { printf( "f3(); " ); return 3; }
int f4() { printf( "f4(); " ); return 4; }

int main()
{
  int (*g3)() = NULL;
  int (*g4)() = NULL;

  printf("\nTest 4: " );
  g3 = &f3;
  g4 = &f4;
  printf( "Result = %d ", g3() );
  printf( "Result = %d ", g4() );
```

Returning values

```
int f3() { printf( "f3(); " ); return 3; }
int f4() { printf( "f4(); " ); return 4; }

int main()
{
  int (*g3)() = NULL;
  int (*g4)() = NULL;

  printf("\nTest 4: " );
  g3 = &f3;
  g4 = &f4;
  printf( "Result = %d ", g3() );
  printf( "Result = %d ", g4() );
```

```
Test 4:
f3(); Result = 3
 f4(); Result = 4
```

Passing parameters

```
int f3() { printf( "f3(); " ); return 3; }

int f5( int i )
{ printf( "f5( %d ); ", i ); return i+1; }

int f6( int i )
{ printf( "f6( %d ); ", i ); return i-2;  }

int main()
{
   int (*g3)() = &f3;
   int (*g5)( int ) = &f5;
   int (*g6)( int ) = &f6;

   printf("\nTest 5: " );      g5( 1 );    g6( 2 );
   printf("\nTest 6: " );      g5( g3() );
   printf("\nTest 7: " );      g6( g4() );
   printf("\nTest 8: " );      g6( g5( g3() ) );
```

Passing parameters

```
int f3() { printf( "f3(); " ); return 3; }

int f5( int i )
{ printf( "f5( %d ); ", i ); return i+1; }

int f6( int i )
{ printf( "f6( %d ); ", i ); return i-2;  }

int main()
{
   int (*g3)() = &f3;
   int (*g5)( int ) = &f5;
   int (*g6)( int ) = &f6;

   printf("\nTest 5: " );      g5( 1 );    g6( 2 );
   printf("\nTest 6: " );      g5( g3() );
   printf("\nTest 7: " );      g6( g4() );
   printf("\nTest 8: " );      g6( g5( g3() ) );
```

```
Test 5: f5( 1 ); f6( 2 );
Test 6: f3(); f5( 3 );
Test 7: f4(); f6( 4 );
Test 8: f3(); f5( 3 ); f6( 4 );
```

Callback function

```
int f5( int i )
{
  printf("f5( %d ); ",i);
  return i+1;
}

// Execute a callback
   function ten times
int Do10(
  int (*func)(int)      )
{
  int i = 0, j = 0;
  for ( ; i < 10 ; i++ )
      j = func( j );
  return j;
}
```

```
int f6( int i )
{
  printf("f6( %d ); ",i);
  return i-2;
}

int main()
{
  int (*g5)( int ) = &f5;
  int (*g6)( int ) = &f6;

  printf("\nTest 9: " );
  Do10( g5 );
  printf("\nTest 10: " );
  Do10( g6 );
  …
```

Callback function

```
int f5( int i )
{
   printf("f5( %d ); ",i);
   return i+1;
}

// Execute a callback
   function ten times
int Do10(
   int (*func)(int)      )
{
   int i = 0, j = 0;
   for ( ; i < 10 ; i++ )
       j = func( j );
   return j;
}
```

```
int f6( int i )
{
   printf("f6( %d ); ",i);
   return i-2;
}

int main()
{
   int (*g5)( int ) = &f5;
   int (*g6)( int ) = &f6;

   printf("\nTest 9: " );
   Do10( g5 );
   printf("\nTest 10: " );
   Do10( g6 );
   …
```

```
Test 9: f5( 0 ); f5( 1 ); f5( 2 ); f5( 3 ); f5( 4 );
  f5( 5 ); f5( 6 ); f5( 7 ); f5( 8 ); f5( 9 );
```

Callback function

```
int f5( int i )
{
   printf("f5( %d ); ",i);
   return i+1;
}

// Execute a callback
   function ten times
int Do10(
   int (*func)(int)      )
{
   int i = 0, j = 0;
   for ( ; i < 10 ; i++ )
       j = func( j );
   return j;
}
```

```
int f6( int i )
{
   printf("f6( %d ); ",i);
   return i-2;
}

int main()
{
   int (*g5)( int ) = &f5;
   int (*g6)( int ) = &f6;

   printf("\nTest 9: " );
   Do10( g5 );
   printf("\nTest 10: " );
   Do10( g6 );
   …
```

```
Test 10: f6( 0 ); f6( -2 ); f6( -4 ); f6( -6 ); f6( -8 );
  f6( -10 ); f6( -12 ); f6( -14 ); f6( -16 ); f6( -18 );
```

Function pointer typedef

You can using **`typedef`** to create a new type which can simplify the code considerably:

```
/* Make name fptr1 mean function pointer:
  return int, int parameter */
typedef int (*fptr1)(int);

/* return void, float parameter */
typedef void (*fptr2)(float);

/* Usage: */
fptr1 f= …;
fptr2 g= … , myfptr = …;
```

Arrays of function pointers

```
#include <cstdio>

int f1( int i )
   { return i; }
int f2( int i )
   { return i*2; }
int f3( int i )
   { return i*3; }

/* typedef makes the name
   fptr mean a function
   pointer: int <f> (int) */
typedef int (*fptr)(int);
```

1. Addresses of functions are stored in an array
2. Can call functions by index in the array rather than by name

```
int main()
{
   int i,j,k;

   fptr fptrarray[6];
   fptrarray[0] = &f1;
   fptrarray[1] = &f2;
   fptrarray[2] = &f3;

   i = fptrarray[2](2);

   j = fptrarray[0](i);

   k = fptrarray[2](j);

   printf( "%d %d %d\n",
       i, j, k );
   return 0;
}
```

In an exam…

- Any exam question will have a similar type of form to the last few slides

- i.e. ‘What is the output of this code…’

1. Work out what points to what
2. And resolve the function calls

v-tables

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;
}
```

Virtual and non-virtual functions

- For normal/default (non-`virtual`) functions:
  - Type of **pointer** determines function to call
  - i.e. the **apparent** type of the object, from pointer type
  - Use the type of the object the compiler thinks it is:
    - Type of pointer (or reference) to the object
    - Type of the member function making the call (hidden `this` ptr)
    - Easier for the compiler, type is known at **compile-time**
- Virtual function:
  - Finds out the **actual** function to call based upon the object type **AT RUNTIME** - much more difficult - slower
  - i.e. look-up: 'which function should I really call'
  - Works in the same way as Java functions

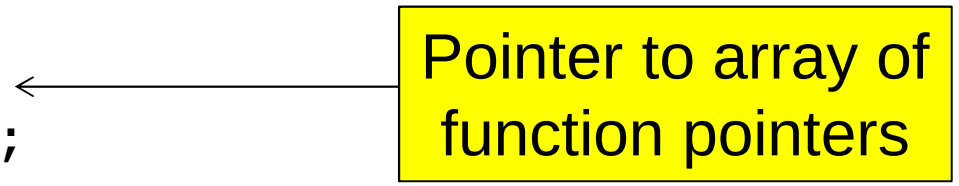
Possible Implementation

- **How could virtual functions be implemented?**

- One possible implementation uses **vtables** (virtual function tables) and **vpointers** (pointers to a vtable)

- This is equivalent to having a hidden pointer:

```
struct EMPLOYEE
{
    void** vpointer;
    char strName[64];
    int iEmployeeID;
} Employee;
```



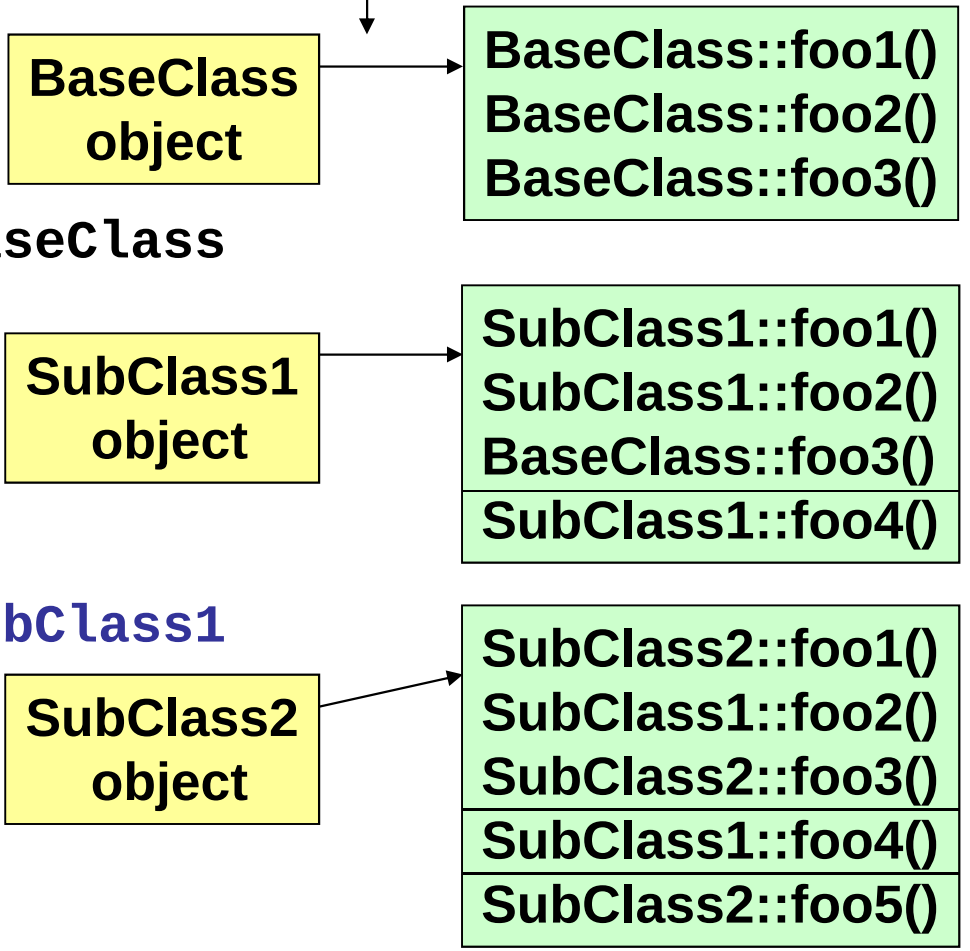
The **vtable** (virtual function table)

```
class BaseClass
{ public:
virtual void foo1();
virtual void foo2();
virtual void foo3();
}

class SubClass1 : public BaseClass
{ public:
virtual void foo1();
virtual void foo2();
virtual void foo4();
}
class SubClass2 : public SubClass1
{ public:
virtual void foo1();
virtual void foo3();
virtual void foo5();
}
```

Object has a **hidden** pointer to the **vtable** for its class (**vpointer**)

Class has a **vtable** (which functions to call)



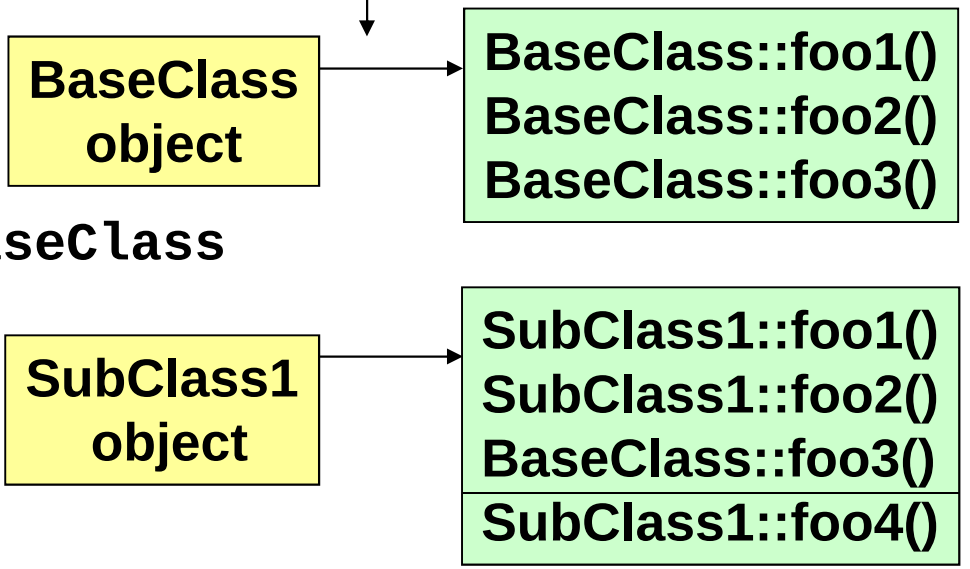
The **vtable** (virtual function table)

```
class BaseClass
{ public:
virtual void foo1();
virtual void foo2();
virtual void foo3();
}

class SubClass1 : public BaseClass
{ public:
virtual void foo1();
virtual void foo2();
virtual void foo4();
}
```

Object has a **hidden** pointer to the **vtable** for its class (**vpointer**)

Class has a **vtable** (which functions to call)



The index in the array matters!!!
The caller only needs to know which index is which function.
Sub-classes keep the index the same as the base class…
… and just add new functions.

Possible Implementation (1)

- One possible implementation for **vtables** (virtual function tables) and **vpointers** (pointers to a vtable) can be simulated in as follows:

```
struct EMPLOYEE
{
    void** vpointer;
    char strName[64];
    int iEmployeeID;
} Employee;
```

Possible Implementation (2)

- Define the functions that could be called

```
char* vGetManagerTypeName()
{ return “Manager”; }

char* vGetEmployeeTypeName()
{ return “Employee”; }

char* vGetDirectorTypeName()
{ return “Director”; }
```

Possible Implementation (3)

- Define the functions that could be called
- Create the arrays of function pointers (void* pointers)

```
void* pEmployeeFunctions[] =
        {       &vGetEmployeeTypeName,
                &vGetNameEmployee,
                &vGetEmployeeID };
void* pManagerFunctions[] =
        {       &vGetManagerTypeName,
                &vGetNameManager,
                &vGetManagerEmployeeID };
void* pDirectorFunctions[] =
        {       &vGetDirectorTypeName,
                &vGetNameDirector,
                &vGetDirectorEmployeeID };
```

`void*` so that Function types can differ

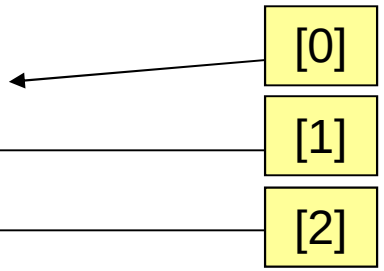
Must cast to correct type before use

Possible Implementation (4)

```
typedef struct EMPLOYEE
{
        void** vtable;
        char strName[64];
        int iEmployeeID;
} Employee;
```

Add the vtable pointer to the class

```
void* pEmployeeFunctions[3] =
{
   &vGetEmployeeTypeName,
   &vGetName,
   &vGetEmployeeID
};
```



Fake the vtable
An array of pointers

```
// Create the structs
Employee e1 = { pEmployeeFunctions, "Employee 1", 1 };
Employee e2 = { pEmployeeFunctions, "Employee 2", 2 };
```

When you create the objects set the vtable pointer

Using the vptr to find the function

- How to use a vtable:
  1. Extract the pointer to the array of virtual functions from the object
  2. Find function pointer at the correct place in the array
  3. Cast the function pointer to the correct type
  4. Call the function
- You need to know: array index, function return type and parameter value types

```
Char* GetTypeName( Employee* e )
{
   void* vfn = e->vtable[0];          // Extract
   char* (*fn)(/*Params*/) = vfn;     // Cast
   return (*fn)();                    // Call
}
```

What to know

- Some equivalent of a **`vpointer`** exists in objects with virtual functions
  - Just one pointer is needed in each object
- Only virtual functions appear in **`vtables`**
  - No need to record non-virtual functions
- Looking up which function to call is slower than calling a non-virtual function
  - But can be done!

1. Go to the object itself
2. Go to the **`vtable`** (following the **`vpointer`**)
3. Look up which function to call from index
4. Call the function

Example: virtual functions

```
#include <cstdio>
class BaseClass {
public:
   virtual char* bar() { return "BaseBar"; }
   char* bar2() { return this->bar(); }
};
class SubClass : public BaseClass {
public:
   char* bar() { return "SubBar "; }
};
int main()
{
   BaseClass* pBase = new BaseClass;
   SubClass*  pSub  = new SubClass;
   BaseClass* pSubAsBase = pSub;
   printf( "bar  B=%s S=%s SaB=%s\n",
       pBase->bar(), pSub->bar(), pSubAsBase->bar() );
   printf( "bar2  B=%s S=%s SaB=%s\n",
       pBase->bar2(), pSub->bar2(), pSubAsBase->bar2() );
   delete pBase; delete pSub;
}
```

BaseClass vtable:
BaseClass::bar()

SubClass vtable:
SubClass::bar()

Notes on virtual functions

- **`virtual`**-ness is inherited
  - If a function is **`virtual`** in the base class then the function is **`virtual`** in the derived class(es)
    - Including destructor! (If **`virtual`** in base class then destructors of derived classes are **`virtual`**)
  - Even when the keyword **`virtual`** is not used in the *derived* class
- All functions in **Java** are **`virtual`** by default
  - Unless you make them '**`final`**'
  - Java, **`final`** functions are different: they cannot be re-implemented in sub-classes

Pure virtual functions

- A function in a base class may be available ***ONLY*** for re-implementation in sub-class
  - Like '`abstract`' function in Java
- Rather than giving a dummy implementation for a function in a base class, make it **pure virtual**
  - In declaration in class body use `=0` not `{}` or `;`
  - e.g. `virtual void purevirtfoo() = 0;`
  - It exists as a place-holder in vtable, but with no function to call (function pointer `= NULL`/`0`?)
- A class with at least one pure virtual function is an abstract class (rather than a concrete class)

Should a function be `virtual`?

- If member function is called **from a base class function** or **through a base class pointer** **AND** the behaviour should depend on class type **then** the member function has to be `virtual`
  - Otherwise when it is called by the base class, or through a base-class pointer, the base-class version will be called, not your modified version
- Utility functions will often **not** be `virtual`
  - When functionality is not expected to be changed in sub-classes
  - Faster to call these functions – no look-up needed
  - Makes it easier for function to be `inline,` which can make code even faster: no function call needed

To be clear … `sizeof()`

- Functions act on objects of the class type
  - Even member functions
- They are not actually in the objects
  - The just have a hidden '**`this`**' parameter saying which object they apply to
- The object is the collection of data
  - But also includes any hidden data and pointers that the compiler adds to implement features (e.g. **`vtable`**)
- Adding a member function to an existing class will not **usually** make the **objects** bigger
  - Exception: adding the first virtual function may add a **`vtable`** pointer (or equivalent)
  - Understand why this is the case!

Next lecture

- Automatically created methods:
  - Default Constructor
  - Copy Constructor
  - Assignment operator
  - Destructor
- Conversion constructors
- Conversion Operators
- Friends