G52CPP
C++ Programming
Lecture 11

Dr Jason Atkin

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

Last lecture

- **`new`** and **`delete`**
- Inheritance
- Virtual functions

Uninitialised variables

```
class MyClass
{
public:
    int ai[4];
    short j;
};
```

- Member data of basic types will be uninitialised
- Use the initialisation list to initialise variable
- Use the constructor to set values for arrays
  - The default constructors do nothing

This lecture

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

- References
  - – Act like pointers
  - – Look like values

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

The **`this`** pointer

The **`this`** pointer

- An object is a collection of data (its state)
- A class defines the ***structure*** of the object and what you can do with it (a design for an object)
  - e.g. Clothing, cars, programs, etc
- **For functions to actually do something to an object, they need to know which object to affect**
- (Non-static) member functions have an **implicit** extra parameter saying which object to act on
  - Parameter ***type*** is a ***pointer to object*** (of correct class)
  - And the parameter ***name*** is **`this`**
- Note: **`this`** exists in Java too, as you know
  - As an object reference to the current object

The `this` pointer

```
class DemoClass
{
public:
   int GetValue()
   {
      return m_iValue;
   }

   void SetValue(int iValue)
   {
      m_iValue = iValue;
   }

private:
   int m_iValue;
};
```

- **`GetValue()`** is effectively:

```
int GetValue(DemoClass* this)
{
   return m_iValue;
}
```

- **`SetValue(int)`** is effectively:

```
void SetValue(
DemoClass* this, int iValue )
{
   m_iValue = iValue;
}
```

- i.e. you can refer to `m_iValue` as `this->m_iValue`
- Not always obvious because you can miss out the `this->`

Static methods and attributes

- **`static`** members are shared between all objects of that class
- **NOT** associated with a specific object
  - Same as **`static`** in Java
- Static member functions **do not** have a **`this`** pointer
- **Both static and non-static member data and functions are class members**
  - i.e. They have access to **`private`** members

```
class MyClass
{
public:
  static int var;
  static void foo();
};

int MyClass::var = 25;

void MyClass::foo()
{
   var = 32;
}

int main()
{
   MyClass::var = 15;
   MyClass::foo();
}
```

Static methods/functions

- Declaration of static member function:
  `static void foo();`
- Usually in .h file

- Definition of static member function
  ```
  void MyClass::foo()
  {
      var = 32;
  }
  ```
- Usually in .cpp file
- No 'static' keyword in cpp file

- Call of static function
  `MyClass::foo();`

```
class MyClass
{
public:
  static int var;
  static void foo();
};

int MyClass::var = 25;

void MyClass::foo()
{
   var = 32;
}

int main()
{
   MyClass::var = 15;
   MyClass::foo();
}
```

Static data members / attributes

- Declaration of static data member:
  `static int var;`
- Usually in a header file

- Definition and initialisation of static member
  `int MyClass::var = 25;`
- Usually in .cpp file
- Done ONCE

- Use of static member
  `var = 32;` `// Within class`
  `MyClass::var = 15;`

```
class MyClass
{
public:
  static int var;
  static void foo();
};

int MyClass::var = 25;

void MyClass::foo()
{
   var = 32;
}

int main()
{
   MyClass::var = 15;
   MyClass::foo();
}
```

References

A short intro

We'll see many examples later

References

- A way to give a new name to an item
- **Look like normal variables**
  - Usage syntax is same as for non-pointer variables
- **Act like pointers**
  - To work out what will happen with a reference, think “what would happen if it was a pointer”

- Opinions on references vary:
  - Some say “use pointers whenever you can do so”
  - Others say “use references whenever you can do so”
  - My view:
    - “If it acts like a pointer, it should look like a pointer”
    - Looking like a non-pointer and acting like a pointer is a recipe for disaster (***my own opinion only***)

The really confusing part…

- As if that was not confusing enough...

… references are labelled with an **&**

  – Like the address-of operator, but **NOT** the address-of operator

- Example:

```
int i = 1;
int& j = i;
j = 2;
int* pi = &i;
*pi = 3;
```

j is a reference to **i**
Just another name for **i**
Anything done to **j** will apply to **i**

Notice that the pointer does the same kind of thing
***pi** is another name for i

Example: references.cpp

```
#include <cstdio>

int main( int argc, char* argv[] )
{
   int i = 9;
   int& j = i;
   j = 4;

   printf( "i=%d, j=%d\n", i, j );

   return 1;
}
```

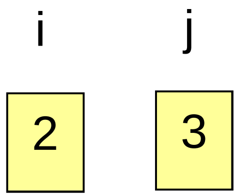
What is the output?

Example 2 : **Without** references

```
#include <cstdio>

int RefFunction( int a, int b )
{
  a += b;
  return b;
}

int main()
{
  int i = 2;
  int j = 3;
  int k = RefFunction( i, j );
  k += 4;
  printf( "%d %d %d\n", i, j, k );
  return 0;
}
```

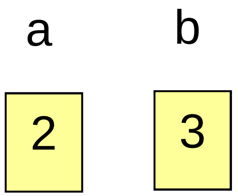
Example 2 : Without references

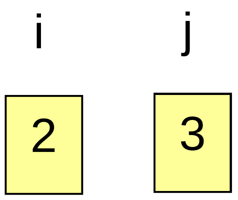
```
#include <cstdio>

int RefFunction( int a, int b )
{
    a += b;
    return b;
}

int main()
{
    int i = 2;
    int j = 3;
    int k = RefFunction( i, j );
    k += 4;
    printf( "%d %d %d\n", i, j, k );
    return 0;
}
```



Example 2 : Without references

```
#include <cstdio>

int RefFunction( int a, int b )
{
  a += b;
  return b;
}

int main()
{
  int i = 2;
  int j = 3;
  int k = RefFunction( i, j );
  k += 4;
  printf( "%d %d %d\n", i, j, k );
  return 0;
}
```



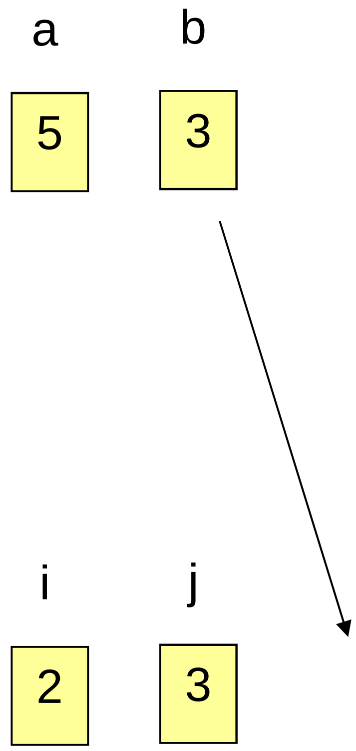


Example 2 : Without references

```
#include <cstdio>

int RefFunction( int a, int b )
{
  a += b;
  return b;
}

int main()
{
  int i = 2;
  int j = 3;
  int k = RefFunction( i, j );
  k += 4;
  printf( "%d %d %d\n", i, j, k );
  return 0;
}
```

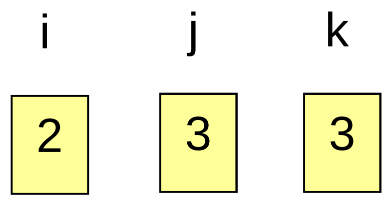
Example 2 : Without references

```
#include <cstdio>

int RefFunction( int a, int b )
{
  a += b;
  return b;
}

int main()
{
  int i = 2;
  int j = 3;
  int k = RefFunction( i, j );
  k += 4;
  printf( "%d %d\n", i, j, k );
  return 0;
}
```



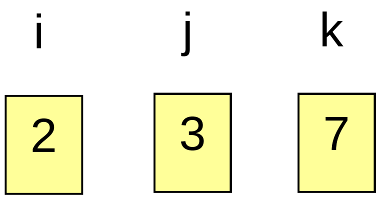
Example 2 : Without references

```
#include <cstdio>

int RefFunction( int a, int b )
{
  a += b;
  return b;
}

int main()
{
  int i = 2;
  int j = 3;
  int k = RefFunction( i, j );
  k += 4;
  printf( "%d %d %d\n", i, j, k );
  return 0;
}
```



Example 2 : Without references

```
#include <cstdio>

int RefFunction( int a, int b )
{
   a += b;
   return b;
}

int main()
{
   int i = 2;
   int j = 3;
   int k = RefFunction( i, j );
   k += 4;
   printf( "%d %d %d\n", i, j, k );
   return 0;
}
```



Passing parameters

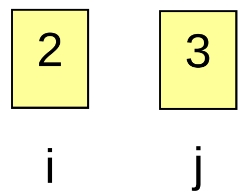
- When a function is called, the values of the parameters are copied into the stack frame for the new function
- i.e. function gets a **copy** of the variable
- Not so for references
  - Then the parameter refers to the **same** variable

Example 2 : With References

```
#include <cstdio>

int& RefFunction( int& a, int& b )
{
   a += b;
   return b;
}

int main()
{
   int i = 2;
   int j = 3;
   int& k = RefFunction( i, j );
   k += 4;
   printf( "%d %d %d\n", i, j, k );
   return 0;
}
```

Example 2 : With References

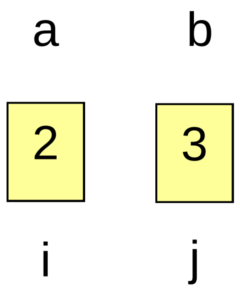
```
#include <cstdio>

int& RefFunction( int& a, int& b )
{
  a += b;
  return b;
}

int main()
{
  int i = 2;
  int j = 3;
  int& k = RefFunction( i, j );
  k += 4;
  printf( "%d %d %d\n", i, j, k );
  return 0;
}
```



Example 2 : With References

```
#include <cstdio>

int& RefFunction( int& a, int& b )
{
  a += b;
  return b;
}

int main()
{
  int i = 2;
  int j = 3;
  int& k = RefFunction( i, j );
  k += 4;
  printf( "%d %d %d\n", i, j, k );
  return 0;
}
```

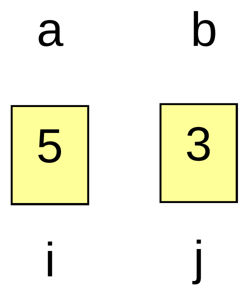
New names for same variables:



Example 2 : With References

```
#include <cstdio>

int& RefFunction( int& a, int& b )
{
  a += b;
  return b;
}

int main()
{
  int i = 2;
  int j = 3;
  int& k = RefFunction( i, j );
  k += 4;
  printf( "%d %d %d\n", i, j, k );
  return 0;
}
```

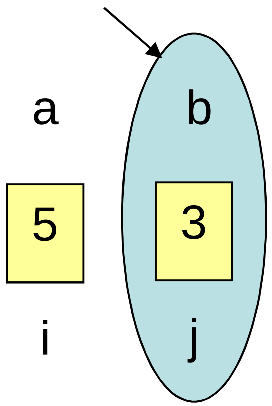
a += b:



Example 2 : With References

```
#include <cstdio>

int& RefFunction( int& a, int& b )
{
  a += b;
  return b;
}

int main()
{
  int i = 2;
  int j = 3;
  int& k = RefFunction( i, j );
  k += 4;
  printf( "%d %d %d\n", i, j, k );
  return 0;
}
```

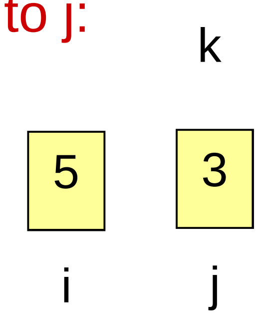
Return reference to b



Example 2 : With References

```
#include <cstdio>

int& RefFunction( int& a, int& b )
{
   a += b;
   return b;
}

int main()
{
   int i = 2;
   int j = 3;
   int& k = RefFunction( i, j );
   k += 4;
   printf( "%d %d %d\n", i, j, k );
   return 0;
}
```

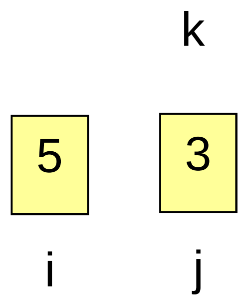
k is a reference to j:



Example 2 : With References

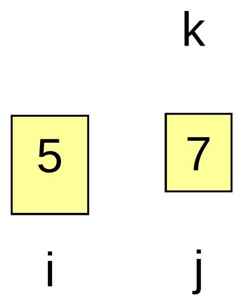
```
#include <cstdio>

int& RefFunction( int& a, int& b )
{
    a += b;
    return b;
}

int main()
{
    int i = 2;
    int j = 3;
    int& k = RefFunction( i, j );
    k += 4;
    printf( "%d %d %d\n", i, j, k );
    return 0;
}
```

k += 4:



Example 2 : With References

```
#include <cstdio>

int& RefFunction( int& a, int& b )
{
    a += b;
    return b;
}

int main()
{
    int i = 2;
    int j = 3;
    int& k = RefFunction( i, j );
    k += 4;
    printf( "%d %d %d\n", i, j, k );
    return 0;
}
```



References vs pointers

- Changing what they refer to:
  - Pointers can be made to point to something else
  - References always bind to a single object, at creation, and cannot be bound to a new object
  - i.e. you can't make them refer to something else
- References always have to refer to something
  - Must give them a ***thing to refer to*** on ***initialisation***
  - No such thing as a **`NULL`** reference
- Pointers need **`*`** or **`->`** to dereference them, to access the thing pointed to
  - References do not (use reference name itself, or .)
- **Java object references act like C/C++ pointers, NOT C++ references**. But they have the syntax of C++ references (e.g. **`.`** not **`->`**)

`const` references

`const` references

- `const` references make the thing referred to const
  - `const` for pointers can mean either ***unchangable pointer*** or ***the thing pointed at cannot be changed***
  - You cannot make a reference refer to something else anyway, so `const` always means the thing referred to
- `const` references are useful for parameters
  - Passing by value (not reference) means the original variable cannot be accidentally modified
    - May be safer
  - Passing a reference means that no copy is made
    - May be quicker – copying objects can be slow
  - Using a `const` reference means no copy needs to be made, but the original can still not be changed, **like a copy but faster**

The need for references

- Useful if we need to keep the same syntax
  - But avoiding making a copy
  - Sometimes this is vital – see copy constructor

- Useful as return values, to chain functions together
  - Especially returning `*this` from member functions to return reference to current object
    - This will make sense later on, with examples

- References are **necessary** for operator overloading
  - Changing the meaning of operators
  - The syntax means that you cannot use pointers

Warning

- Similar problems with references as with pointers

- e.g. **do NOT return a reference to a local variable**
  - When the local variable vanishes (e.g. the function ends), the reference refers to something that doesn't exist
  - Same symptoms as for pointers – it will look OK until something else uses the memory

`const` members

`const` member data

```
class DemoClass
{
public:
  DemoClass()
  : ci(4)
  , cj(12)
  {}
private:
  int const ci;
  const int cj;
};
```

Note: Relative order of `const` and type only matters for pointers `const *` vs `* const`

- `const` **member** data **MUST** be initialised in the initialisation list for the constructor
  - i.e. an initial value when member data is constructed
- Cannot just be set in constructor body, since construction has occurred by then
- **Compiler error if you miss any**

`const` references and pointers

- Q: If you have a `const` reference (or pointer) to an object, then which ***methods*** can you call using the reference (or pointer)?

```
MyClass ob2;
const MyClass& rob2a = ob2;

rob2a.GetVal(); // ?
rob2a.SetVal(); // ?
```

`const` references and pointers

- Q: If you have a **`const`** reference (or pointer) to an object, then which ***methods*** can you call using the reference (or pointer)?
- A: Only methods which **guarantee** not to change the object (i.e. accessors)
- **These methods are labelled `const`**
  - **They CANNOT alter member data**
  - **The this pointer is const**
- Functions are either mutators or accessors
  - Accessors only access data – should be **`const`**
  - Mutators change data – **cannot** be **`const`**

Which of these lines will not compile?

```
class ConstClass
{
public:
   // Constructor
   ConstClass()
   {}

   // Accessor
   int GetVal() const
   { return _ival; }

   // Mutator
   void SetVal(int ival)
   { _ival = ival; }

private:
   int _ival;
};
```

```
int main()
{
   ConstClass ob2;
   ConstClass& rob2 = ob2;
   const ConstClass& rob2a = ob2;
   ConstClass const& rob2b = ob2;

   rob2.GetVal();
   rob2a.GetVal();
   rob2b.GetVal();

   rob2.SetVal(3);
   rob2a.SetVal(1);
   rob2b.SetVal(2);
}
```

Example: **const** functions

```
class ConstClass
{
public:
   // Constructor
   ConstClass()
   {}

   // Accessor
   int GetVal() const
   { return _ival; }

   // Mutator
   void SetVal(int ival)
   { _ival = ival; }

private:
   int _ival;
};
```

```
int main()
{
   ConstClass ob2;
   ConstClass& rob2 = ob2;
   const ConstClass& rob2a = ob2;
   ConstClass const& rob2b = ob2;

   rob2.GetVal();
   rob2a.GetVal();
   rob2b.GetVal();

   rob2.SetVal(3);

   // The following 2 lines
   // do not compile
   rob2a.SetVal(1);
   rob2b.SetVal(2);
}
```

mutable

- The compiler will **not allow** you to alter member data from a member function declared as **`const`**
  - If you try, then you will get a compilation error
- If you need to alter a **specific** variable within a **`const`** member function, you can declare **that variable** **`mutable`**
- e.g. for a class which caches the last value retrieved:

```
class CachingClass
{
  int _iVal;
  mutable int _lastgot;
public:
  int GetVal() const
        {_lastgot = _iVal; return _iVal; }
  void SetVal( int iVal ) const
        { _iVal = iVal; }
}
```

This can be altered even by **`const`** member functions

mutable

- The compiler will **not allow** you to alter **any** member data from a member function declared as **`const`**
  - If you try, then you will get a compilation error
- If you need to alter a specific variable within a **`const`** member function, you can declare that variable **`mutable`**
- e.g. for a class which caches the last value retrieved:

```
class CachingClass
{
  int _iVal;
  mutable int _lastgot;
public:
  int GetVal() const
        {_lastgot = _iVal; return _iVal; }
  void SetVal( int iVal ) const
        { _iVal = iVal; }
}
```

OK, since `_lastgot` is mutable

Compilation error `const` fn sets `_ival`

Next Lecture

- Function pointers
- Virtual and non-virtual functions
  - v-tables