G52CPP
C++ Programming
Lecture 10

Dr Jason Atkin

Last lecture

- Constructors
  - Default constructor – needs no parameters
- Default parameters
- Inline functions
  - Work exactly as if the function had been called
  - But copy the code into the caller function
- Function definitions outside the class declaration
  - i.e. .h files and .cpp files

This lecture

- References
- new and delete

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

QUESTION: 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;
}
```

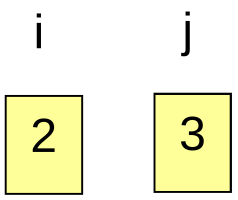
Question: What is the output?

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;
}
```

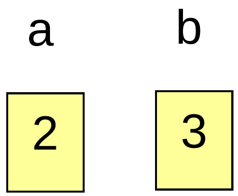
Example 2 : Without references

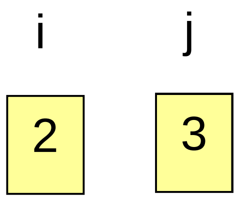
```
#include <cstdio>

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;
}
```



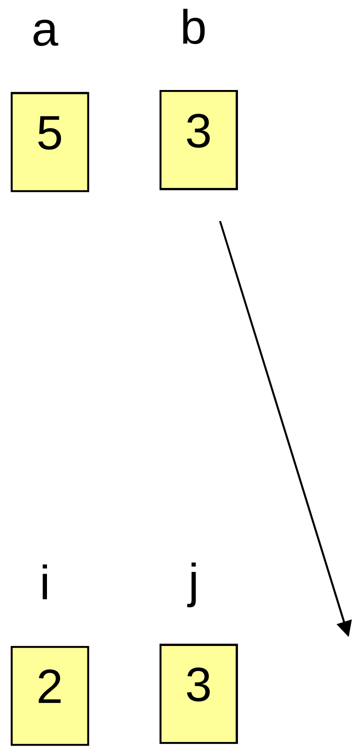


Example 2 : Without references

```
#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;
}
```

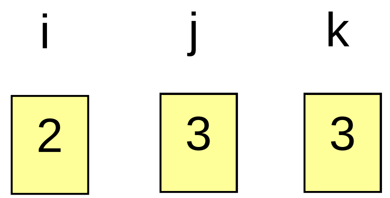
Example 2 : Without references

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



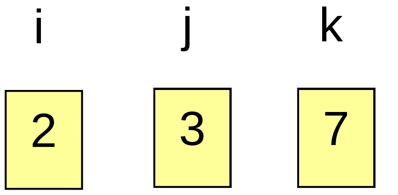
Example 2 : Without references

```
#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

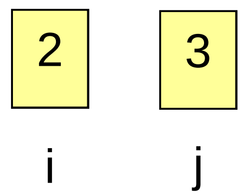
- When a function is called, the values of the parameters are copied into the stack frame for the new function
- i.e. function gets a **copy** of the variable
- Not so for references
  - Then the parameter refers to the **same** variable

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

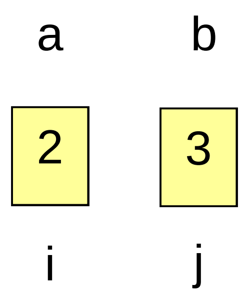
```
#include <cstdio>

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;
}
```

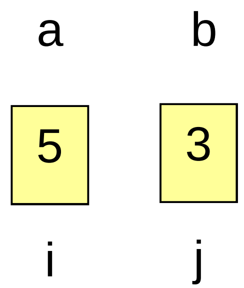
New names for same variables:



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;
}
```

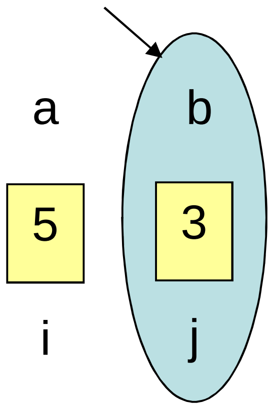
a += b:



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;
}
```

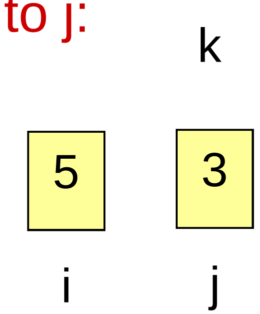
Return reference to b



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;
}
```

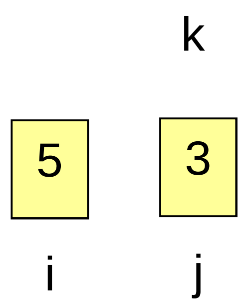
k is a reference to j:



Example 2 : With References

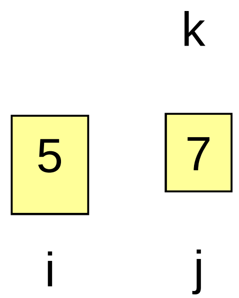
```
#include <cstdio>

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 `->`)

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 later

- 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 with operator overloads later

- 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

New and Delete

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;
}
```

Create a new object of type MyClass on the heap

Really creates the object, e.g. calls the constructor

Uses default constructor for each object in array

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

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)

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

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 ( )

Could 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**

Next lecture

- const

- Constants
- const pointers
- const references
- const member functions