C++ for Java Programmers

Basics all Finished!
- Everything we have covered so far:
  - Operators
  - Variables
  - Arrays
  - Null Terminated Strings
  - Structs
  - Functions

Introduction
- Many problems can only be solved by representing a significant amount of information.
- C/C++ provide us with mechanisms for dynamic memory allocation and management of data structures during runtime.
- This means we can write flexible programs where sizes of data structures don’t have to be set at compile time.
- Java uses exactly the same mechanisms in the background but hides them from you.

45 mins of pure fun
- Today:
  - Pointers
  - Pointers
  - Even more on Pointers

Dynamic Structures...
- Access to these Dynamic structures are through pointers.
- a pointer is just a variable whose value is the location of another object.
- C++ provides two fundamental pointer operators:
  - the address operator &
  - the de-referencing operator *

LValues and Rvalues
- In C++ there are two kinds of expressions

<table>
<thead>
<tr>
<th>LVALUES</th>
<th>RVALUES</th>
</tr>
</thead>
<tbody>
<tr>
<td>Those that represent objects that can be modified and evaluated.</td>
<td>Those that represent objects which can only be evaluated.</td>
</tr>
</tbody>
</table>

- Although assignment typically uses an object name as its left operand, the syntax only requires it to be an LVALUE. This is important because not all objects actually have names. (eg. elements in an array).
Quick Example

```cpp
#include <iostream.h>

int main ()
{
    int a = 1;
    int b;
    int c[3];
    b = 5;
    cout << b << endl;
    c[0] = 2*a;
    return 0;
}
```

- **lvalues**
- **rvalues**

### Pointer Basics

- A pointer is a variable whose value represents the location of another object.
- Pointer objects are defined in conjunction with the unary indirection operator `*` (also known as the dereferencing operator)
- Let's set up three pointers:
  ```cpp
  int *iPtr;
  char *cPtr;
  float *fPtr;
  ```

### Null Pointers

- So what is in these pointers at the moment? ...who knows - they are all uninitialised. This is bad bad bad.
- There is one literal that we can assign to any type of pointer. This is the value 0, which in this context is known as the null address:
  ```cpp
  int *iPtr = 0;
  char *cPtr = 0;
  float *fPtr = 0;
  ```

### Pointer Representation

- **Uninitialised Pointers**
- **Null Pointers**

### Pointer Assignment

- The assignment `=` is fine for use with pointer objects, so long as the pointers are of the same type.
  ```cpp
  Ptr1 = Ptr2;
  ```
- This will result in Ptr1 and Ptr2 pointing to the same object. There is still only one object, whatever it is - it just has two things pointing at it.
- We've used a pointer as the right operand here, but it could have been any valid expression.

### Example of Mistakes

- ```cpp
  int i = 1;
  int *iPtr;
  char *cPtr;

  iPtr = i;
  i = iPtr;
  iPtr = cPtr;
  ```
- Illegal: i is not an int.
- Illegal: iPtr is not an int.
Quick BUG highlight

- Some people juxtapose the * so the statement reads more logically:
  
  ```c
  char* cPtr;
  ```

- But it can lead to a misunderstanding. For example:
  
  ```c
  char* a, b;
  ```

- Hence this statement does not lead to the creation of two pointers. Always define only one pointer per statement.

The Address Operator

- & is the addressing operator
- It returns the memory address of a variable in pointer form. Take the following code segment:

```c
int number = 13;
int *Ptr;
Ptr = &number;
```

sets pointer to the memory location for number

Dereferencing

- The value for number can now be accessed directly or indirectly.
- The indirect uses the dereferencing pointer (the * operation computes the lvalue of the object at which the pointer is aimed).

```c
cout << *Ptr << endl;
```

Hence we can also modify the number variable by:

```c
*Ptr = 99;
```

Initialising Pointers

- Remember if you don’t know where a pointer is heading yet it’s a good idea to initialise it as null. However pointers can include various initialisation expressions:

```c
int a = 3;
int b = 7;
int *Ptr1 = &b;
int *Ptr2 = Ptr1;
int *Ptr3 = &a;
```

What can you point at?

- practically anything:
  - variables
  - your own types
  - arrays
  - structs
  - instances of classes
  - even functions.

- essentially any sort of data we can store in memory....
C++ allows pointer types whose objects are pointers...to other pointers.

Or even pointers to pointers to pointers...

The following is a definition of a pointer to an int* object:

```
int **PtrPtr1;
```

Each asterisk in a definition indicates another level of dereferencing.

Consider the following example:

```
int num = 25;
int *Ptr = &num;
int **PtrPtr = &Ptr;
PtrPtr = &Ptr;
```

The ability to access and change the value of something whose name is not part of the assignment statement makes pointer programming powerful but confusing.

Errors involving pointers are often subtle and hard to track down.

A simple technique is to draw your own diagrams – this can be so useful even experienced professional C++ coders use this technique to help debug code.

The modifier const can be applied to pointer declarations.

However where you place this keyword can give the statement a drastically different meaning...

```
const char *Ptr1 = &ch;
char const *Ptr2 = &ch;
char *const Ptr3 = &ch;
```

Mean the same Thing – the pointer is not a constant, but what its pointing at is.

Here the opposite. The pointer is a constant, the char is not.

We could even have:

```
const char *const Ptr4 = &ch;
```

The C++ syntax can make understanding these declarations difficult.

Solution – read the declaration backwards replacing " with "is a pointer to".

```c
#include <iostream.h>

void Swap(char *Ptr1, char *Ptr2){
    char c = *Ptr1;
    *Ptr1 = *Ptr2;
    *Ptr2 = c
}

test main(){
    char a = 'y';
    char b = 'n';
    Swap(a,b);
    cout << a << b << endl;
}
Pointers as Parameters

- The above program shows that using pointers we can mimic passing by reference.
- To us C++ programmers this seems as though we are just replicating what we did using the & operator to pass by reference.
- However to a C programmer this is crucial because C doesn’t provide standard reference parameters. Life is easier for a C++ coder.

Arrays of Pointers

- Because a pointer is just a type of variable (i.e. one that stores a memory address) it can often be treated just like you would any other variable.
- Hence you can quite happily created an array of pointers if you so wish.
  ```
  int* Ptrl4[5];
  ```
- This code for example would set up an array of five pointers to integers.

Pointers to Arrays

- Never confuse an array of pointers with a pointer to an array!
- In C++ the name of an array is considered to be a constant pointer.
- The name of the array is associated with a particular memory location, and this association cannot be changed.
- In fact it is associated with the location of the first element in the array.

Example

```
int ant[5];
int dec[7];
int *Ptr1 = ant;
int *Ptr2 = &ant[0];
int *Ptr3 = dec;
int *Ptr4 = &dec[4];
```
**Pointer Arithmetic**

- There are only two arithmetic operations that you can perform on pointers: **addition and subtraction**.
- Hence all of the following expressions are valid:
  
  ```
  Ptr++;  
  Ptr--;  
  Ptr2 = Ptr1 - 2;  
  Ptr = (x) + 4;
  ```
- Best use an example to explain this so here goes:

**Pointer Arithmetic Example**

- Say we have a integer pointer `Ptr`, with a current value (which of course is a memory address) of 2000.
- Now we have the expression: `Ptr++`
- What’s the value of `Ptr` now? You might expect it be 2001 but it is in fact **2004**.
- This is because when the pointer is incremented it moves to the next int held in memory – 4 bytes across.

**In general**

- Each time a pointer is incremented it points to the memory location of the next element of its base type.

```
ptr
ptr+1
ptr+2
ptr+3
ptr+4
```

**Pointer Arithmetic Program**

```
#include <iostream.h>

int main(){
  int cubes[] = {1,4,9,16,25,36};
  int *ptr;
  ptr = cubes;
  for(int i=0; i<6; i++) {
    cout << *(Ptr + i) << endl;
  }
}
```

**The Indexing Deception…**

- So what does this mean? It means these are the same

```
int array[] = {1,3,5};
int[[array = {1,3,5};
cout << array[0];
cout << *(array);
cout << *(array+1);
cout << *(array+2);
```

- These are identical apart from the pointer ‘array’ in the left hand code is a constant. On the right it is not.

**Character String Processing**

- All this is particularly for our good old null terminated strings (remember them?)
- Null terminated strings are of course character arrays. The name of the string is just a pointer to the first letter.
- We can use pointer arithmetic to manipulate arrays – and that’s exactly what the functions in `<iostream>` are doing.
The interchangeability of pointer and array notation is most obvious when you pass char strings to functions.

Both of the following are possible implementations of the strlen library function:

```
int strlen(const char s[]) {
    int i;
    for(i=0; s[i]!='\0'; ++i);
    return i;
}
```

```
int strlen(const char *s) {
    int i;
    for(i=0; *s++ !='\0'; ++i);
    return i;
}
```

Pointers to Functions

- Earlier we said that you can use a pointer to point to almost anything – even functions.
- The primary use of this is as a parameter to another function.
- This allows the destination function the ability to invoke a variety of actions
- Just like an array name points to its elements, a function name is a pointer to the location of the code that constitutes its actions.

For example, when we call FuncPtr here it is the same as calling toUpper:

```
int (*FuncPtr)(int i);
FuncPtr = toUpper;
```

You could use this characteristic to send a certain function:

```
void Display(char word[], int (*FuncPtr)(int i)) {
    ...doStuff...
    word = (*FuncPtr)(word);
}
```

So what is the point?

- Well we have:
  - an alternative notation for manipulating arrays
  - a way to access command line arguments
  - is that it?
- No Sireee. The main role of pointers is for creating dynamic objects.
- Dynamic Memory allows to solves lots of problems that might otherwise be intangible...

Problems with Pointers

- Wild pointers are the devils work.
- Pointers give you tremendous power, but if one accidentally contains the wrong value they can be a bugger to find.
- Take as much care as you possibly can not to make pointer errors.
- Classic example is the uninitialised pointer.

Tomorrow

- We’ve pretty much covered the fundamentals of pointers…now you’ve just got to learn how to apply them
- Tomorrow we’ll be looking at Dynamic Memory Allocation – the real role of pointers.
- Extra lab has been organised for Friday at 4pm just for those digital business students who have a clash.