CISC/CMPE320

- Notices:
- Teamwork survey is done – thanks! (153 out of 155, so good to go!)
- How was the tutorial yesterday?
- In the process of manually creating accounts for the 25 (*I know who you are now*…).

Today

- Finish up first pass at Pointers and References using the demo program.
- Arrays.
- Structs.
- Operators.

References and Pointers, Cont.

- This gets even more interesting when moving things in and out of functions. *Later…*
- Pointers tend to be overused in C++ code – try to use a reference first.
- See TestSimplePointers.cpp

Pointers, Review Questions

- I can have many pointers aliased to a fundamental type variable, does this mean that an **`int`** is an Object, for example?
- What is stored in a pointer?
- What does it mean to "de-reference" a pointer, and how do you do it?
- If I add one to a pointer (pointer arithmetic), then how many bytes are added to the memory address and how does the system know?

Pointers, Review Questions Cont.

- I can use pointer arithmetic to access memory positions away from the original pointer:
  - Can I access any memory position I want?
  - Can I access and change any memory position I want?
  - *Is this a good idea*?
- How much memory does a pointer occupy?
- How do you get the address of where the pointer is stored?
- Am I using 32 bit pointers on a 64 bit machine? Why?

Aside – Declaring Multiple Variables

- This is when you do something like:

```
int x, y, z;
```

- But this does not work with operators. For example:

```
int* x, y;
```

- **`x`** is a pointer, but **`y`** is an **`int`**

Aside - `typedef` Keyword

- If you prefix a type with the **`typedef`** keyword, then you can create a synonym for the type.
- For example:

```
typedef long double dprecision;
```

- Now, you can use **`dprecision`** instead of using **`long double`** for a type.

Arrays

- To declare a variable to hold five integers:

```
int anArray[5];
```

- Or you can use an array initializer:

```
int anArray[] = {2, 7, 3, 0, 1};
```

- or

```
int anArray[5] = {2, 7, 3, 0, 1};
```


Rewatch video at this point for array discussion

Arrays, Cont.

- What is in an uninitialized array?
- Can I use pointer arithmetic with an array?
- Can I access values outside the array bounds?
- See ArrayExample.cpp
- **`vector`**s are much better to use than arrays – more about this class later…

Arrays and Strings

- In C, you had to use char[] to store a string literal:

```
char oldString[] = "Hello there!";
```

- These "C-strings" end with the character **`'\0'`**, also called the "null character".
- Manipulating C-strings means manipulating arrays – generally a real pain…
- In C++ use the **`string`** class instead!
- Defined in the **`string`** library, and uses the **`std`** namespace.
- More about **`string`**s later…

Structures

- A kind of **`class`**.
- Defined as an aggregate of other types.
- For example:

```
struct address {
     string name;
     int streetNumber;
     string street;
     string city;
     string province;
     string postalCode;
};
```

Note ";"

Structures, Cont.

- See StructureDemo.cpp
- (Also note use of pointers with a function, and use of **`const`** in parameter list – getting a bit ahead of ourselves!!)
- Note how Eclipse gives you the membership list for an **`address`**.

Member Selection Operators

- The example also demonstrated the use of the two member selection operators:
- The "dot operator":

  object.member

- De-referencing and membership:

  pointer->member

- The latter is the same as:

  (*pointer).member

- ("Members" are attributes or methods (or member functions)…)

Structures, Cont.

- A struct is a primitive object definition with no privacy for it's members.
- Why use them in C++?

Operators

- Discussed in order of highest to lowest precedence.
- Highest Precedence:
- Scope resolution (when defining only):

  class_name*::*member

- Or ***namespace_name*::*member***

Operators, Cont.

- Next Highest Precedence (level 2):
- Member selection **.** (see slide 13).
- Subscripting arrays: ***pointer*[*expr*]**
- **( )** when used with function calls or value constructors.
- Post increment or post decrement: ***var*++ *var*--**
- Type identification: **typeid(*type or expr*)**
- Casts like **static_cast<*type*>(*expr*)**

Aside – `typeid()`

- Used to find out the types of pointers.
- More useful in a function:
- A trivial example:

```
int aVal = 100;
int* ptr_aVal = &aVal;
cout << typeid(ptr_aVal).name() << endl;
```

- Displays: Pi

Operators, Cont.

- Level 3 Precedence:
- **sizeof(*type*)**
- Pre-increment and pre-decrement.
- Complement: **~*expr***
- Not: **!*expr***
- Negation: **-*expr***
- Address of and dereference (**&** and *****)
- The heap operators: **new**, **delete** and **delete[]**
- C – style cast: **(*type*)*expr***

Operators, Cont.

- Level 4:
- Member selection (and dereference) `->`
- Level 5:
- Multiply, Divide, Modulo: `*`, `/`, `%`
- Level 6:
- Add, subtract: `+`, `-`

Operators, Cont.

- Level 7:
- Bitwise shift left: ***`expr << expr`***
- Bitwise shift right: ***`expr >> expr`***
- Level 8:
- `<`, `<=`, `>`, `>=`
- Level 9:
- `==`, `!=`

Operators, Cont.

- Levels 10 to 12:
- Bitwise AND: ***`expr & expr`***
- Bitwise exclusive OR (or XOR): ***`expr ^ expr`***
- Bitwise OR: ***`expr | expr`***
- Levels 13 and 14:
- Logical AND: ***`expr && expr`***
- Logical OR: ***`expr || expr`***
- Level 15:
- Conditional Expression: ***`expr ? expr : expr`***

Operators, Cont. (Still!)

- Level 16:
- All assignment operators:
- `=`
- `*=, /=, %=, +=, -=, <<=, >>=, &=, |=, ^=`
- Level 17:
- Throw exception: **`throw`** ***`expr`***
- Lowest Precedence!:
- Comma: `,`

Notes on Precedence

- Prefix operators (`~`, `!`, `-`, `&`, `*`, pre-increment and pre-decrement) and assignment operators are right associative (*right to left*) – all others are left-associative (*left to right*).
- So:

```
int a = b + c + d;
```

- Means `int a = (b + c) + d;`
- But,

```
int x = y = z;
```

- Means `int x = (y = z);`
- (Yes, the assignment operator returns something!)

Precedence Notes, Cont.

- Use ( ) to control precedence.
- When in doubt, control precedence!
- If an expression gets too long, use intermediate variables to make it more readable.

Bit and Shift Operations

- These operate on all integer types and enums.
- Complement, **~**, carries out a bitwise negation of each bit (1 becomes 0, 0 becomes 1).
- Binary **&**, **|** and **^**:

```
A   B   A&B  A|B  A^B
0   0    0    0    0
0   1    0    1    1
1   0    0    1    1
1   1    1    1    0
```


Bit and Shift Operations, Cont.

- The left shift operator, **<<**, moves all bits in ***val*** to the left by **n** positions: ***val*** **<< n**.
- Zeros are added to the least significant bits.
- This is the same as multiplying a number by 2^n.
- Right shift, **>>**, moves all bits to the right.
- For unsigned integers zeros are added to the most significant bits – same as dividing by 2^n.
- For signed integers, the result is not predicable as the sign bit may be duplicated.

Bitwise Operations, Examples

- Set the n^{th} bit in a number and the rest are zeros:

```
1 << n
```

- To set the n^{th} bit of any number, **x**, and not change the rest:

```
x = x | 1 << n
```

- To check to see if the n^{th} bit is set:

```
if ((x & 1 << n) != 0)…
```


Boolean Expressions

- We have seen the boolean operators already. Here are a few notes:
- Something like

```
a < b < c
```

will compile and run, but may not produce the desired result. Better to use:

```
a < b && b < c
```

- Remember that **&** and **|** are bitwise operators, not logical ones.

Boolean Expressions, Cont.

- The **&&** and **||** logical operators use "short circuit evaluation":
- For **&&** if the LHS is **false** then the RHS is not evaluated.
- For **||** if the LHS is **true** then the RHS is not evaluated.
- (Same as in Java.)

Boolean Expressions, Cont.

- Non zero integers are treated as being **true**, and zero is treated as being **false**. (*Ouch*!)
- So, you can use logical operators, **&& ||** and **!**, with integers.
- For example, the code:

```
int x = 10;
if (x)
```

- is the same as:

```
int x = 10;
if (x != 0)
```

Boolean Expressions, Cont.

- Also, this is legal syntax:

```
if (x = 10)
```

- The assignment operator returns the value being assigned, which in this case is a `true`! But suppose `x` is `12` and you really meant to type `==`…
- *Ouch, again*!

Boolean Expressions, Cont.

- See TestStuff.cpp.
- Applying `!` to a non-zero integer returns **`false`** or zero.
- An **`if`** statement will treat a pointer by itself as an integer – it will be **`true`** unless it is **`NULL`**.
- You can also test assignment statements since the assignment operator returns the value being assigned.