CISC/CMPE320

- Reminders:
- Fill out the teamwork survey. (143 out of 155, so far)
- Attend the tutorial today after class, offered by Kanchan Nair.

Jira Tutorial – Sept. 18

- Go to ELL 321 first. If it fills up and you do not want to stand, go to ELL 333.
- If you end up in 333 - watch the material at the links provided on the next slide while you are waiting for Kanchan to finish her presentation in ELL 321.
- 130 people were successfully sent "invitations" to Jira. 25 failed, but I don't know which ones failed…

Today

- Divide by zero, enums, constants.
- Casting.
- Pointers and References.
- Arrays (*if we have time*).

Aside - Divide by Zero Example

- How does Eclipse/GCC handle a divide by zero?
- Check out TestZeroDivide.cpp.
- Note: We have compilation errors and warnings as well as run-time errors. Note that the run-time error results from the crashing of the executable, and is reported by the OS, not Eclipse.
- How does Java handle a divide by zero?

Enumerated Types

- Useful in supplying cases for **`switch`** statements, or building a bunch of constants.
- Example:

```
enum MonthLengths = {JAN = 31, FEB = 28, MAR
  = 31, APR = 30, MAY = 31, JUN = 30, JUL =
  31, AUG = 31, SEP = 30, OCT = 31, NOV = 30,
  DEC = 31};
```

- The members of the **`enum`** have integer values and appear as named constants.
- If you do not assign values, they are given incremental values starting from zero.

Constants

- Prefix the type declaration with the keyword **`const`**.
- By convention, use all capital letters for the name of the constant.
- For example:

```
const double CM_IN_INCH(2.54);
```

- You must assign a value at the time of declaration.

The `void` Type

- Used to indicate that a function does not return anything.
- Can also be used to declare a pointer to anything:

```
void*
```


Casting Numeric Types

- In C++ you can cast between types in six different ways:
  - `(type)expresssion` } C style – don't use!
  - `type(expression)` } C style – don't use!
  - `static_cast<type>(expression)`
  - `const_cast<type>(expression)`
  - `dynamic_cast<type>(expression)`
  - `reinterpret_cast<type>(expression)`
- Where ***`type`*** is the desired type and ***`expression`*** is what you are casting.

Aside - C Style Casts

- These are crude and too powerful. For example:
- The cast will remove any const property.
- The cast does not check to see if the cast is even possible or if it makes sense at all (like casting a string to an int or *visa-versa*).

`static_cast` Casting

- Best, general purpose technique for atomic types. For example:

```
int aVal = static_cast<int>(4.8);
```

- **`aVal`** contains **`4`** – casting truncates, not rounds.
- You don't always have to cast. For example an **`int`** can always be stored in a **`double`** because of implicit casting or "type coercion":

```
double dVal(5);
```


References and Pointers, First Pass…

- Pointers are not references and references are not pointers!
- A reference is an "alias for its initializer". [From "C++ Gotchas" by Stephen C. Dewhurst]
- The alias must refer to something. You cannot have a null reference or a reference to `void` or even a reference to a reference.

Pointers

- A pointer is a variable that stores a memory address.
- It occupies memory (a reference may not occupy any memory).
- If the memory address was obtained using the & operator ("address of") on another variable, then the pointer indirectly references the value contained by the other variable. Accessing this value through the pointer (using de-referencing) is called *indirection*.

Pointers, Cont.

- In Java, pointers are implicit – you have no control over them. In C and C++ pointers are explicit, you have complete control!
- Pointers can be `NULL` (or better yet, use `nullptr` in C++11)
- You can even have such a thing as **`void*`** as a pointer type.

& and * Operators

- LHS: When creating a type, & gives you a reference to a type, * gives you a pointer to a variable of that type.
- RHS: In an expression, & is the "address of" operator yielding the memory address of a variable.
- * is "de-referencing". When used with a pointer it yields the value of the variable being pointed to.

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};
```


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…