## CISC/CMPE320

- Reminders:
- Fill out the teamwork survey. (116 out of 155, so far)
- TA names and email addresses have been added to the course web site.
- See next slide for info on the tutorial on Monday, the 18th, 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.
- You will have Jira accounts by Monday and Kanchan will show you what to do with them!
- Not everyone needs to bring a laptop or even to log into Jira. One person per table would be fine.

## Jira Tutorial Links

- Also provided in onQ in week 2 content:
- An archive of several short videos from Jira University:

https://www.youtube.com/playlist?list=PLaD4FvsFdarSWUyuv6cto4gunIvVzDTGD

- In Lynda.com search for the following courses:
  - “Agile Project Management: Comparing Agile Tools”, then “Atlassian: Jira”
  - “Learning Jira Software”, look for “Overview of Jira”, “Creating Issues” and “Creating Boards”.

## Today

- Compiler completion of C++ standards.
- Philosophy and mechanics of C++.
- C++ Types and their behaviour.

## C++ Standards, Cont.

- But:
- It takes a while (and a considerable amount of work) to update compilers to the new standards.
- See:

http://en.cppreference.com/w/cpp/compiler_support

for a summary of how various compilers are doing.

- GCC version 4.8.1 and later is now up-to-date with C++11. We are using version 6.3.0
- Other big compilers like Visual Studio and Clang are also now up-to-date with C++11.

## C++ Standards, Cont.

- Compilation is *not* standardized, just the syntax.
- How does Java compare, for example?

## The “Philosophy” of C++

- From “The Design and Evolution of C++” by (*who else!*) Bjarne Stroustrup, 1994:
- C++ is designed to be a statically typed, general-purpose language that is as efficient and portable as C.
- C++ is designed to directly and comprehensively support multiple programming styles (procedural programming, data abstraction, object-oriented programming, and generic programming).
- C++ is designed to give the programmer choice, even if this makes it possible for the programmer to choose incorrectly.

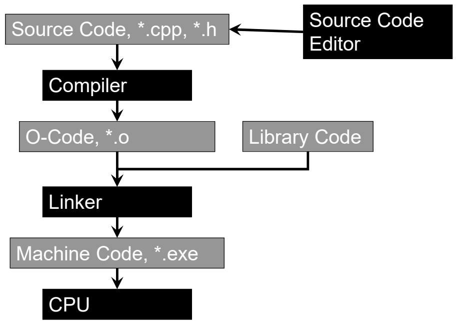
## The “Philosophy” of C++, Cont.

- C++ is designed to be as compatible with C as possible, therefore providing a smooth transition from C.
- C++ avoids features that are platform specific or not general purpose.
- C++ does not incur overhead for features that are not used (the "zero-overhead principle").
- C++ is designed to function without a sophisticated programming environment.

## ISO C++ and C

- The C++ standard library has incorporated the standard C library with just a few optimizations.
- C++ also has the “Standard Template Library”, or “STL”, that contains many pre-defined data structures, for example.
- Generally, well-written C code will run in C++.
- But some C “things” will not work in C++ (for example, in C you can use **`new`** or **`class`** as variable names, since they are not C keywords.)

## Running a C++ Program



Source Code, *.cpp, *.h

Source Code Editor

Compiler

O-Code, *.o

Library Code

Linker

Machine Code, *.exe

CPU

## Running a C++ Program, Cont.

- Thankfully, an IDE simplifies this process for you.
- A project (a set of source code and resource files) is “built” – which consists of both compilation and linking.
- MinGW (“Minimalist GNU for Windows”), for example uses the g++.exe program for this. The IDE makes up the command line calls to this program.
- Building creates a *.o file and then a *.exe file.
- Then you run the *.exe program.
- Each stage of this process can unearth errors.

## Who Reports the Errors?

- An Eclipse pre-compiler does a pretty good (but not perfect!) job of reporting syntax errors and warnings. Errors may prevent building. You can (but should not!) ignore warnings.
- Build errors are reported by the compiler and are seen in the console window.
- Run-time errors are reported by the OS, which is running the *.exe file.

## Variable Types

- How many variable types do you have in C? (not counting structs)?
- You can have the atomic types, pointers and arrays of these types. Anything else?

- How many types can you have in C++?
- Infinitely many!
- Thousands defined in the STL.
- You can define as many as you want.
- *Oh* – the atomic types are still there too…

## Variable Declaration

- C++ is declarative and statically typed.
- ie. “Declare before you use” and “You can’t change the type of a variable, once declared”.
- Unlike C, you can declare variables anywhere, as long as you satisfy the above.
- Initialization is good practice, but is optional.
- Examples:

```
int numStudents(40); // C++ style declare
bool flagVal(true);
double aNum;
```

## Variable Declaration, Cont.

- This is what we are used to from Java:

```
int numStudents = 40;
bool flagVal = true;
```

- For a C++ “style” declaration, use:

```
int numStudents(40);
bool flagVal(true);
```

- Looks like you are invoking a constructor doesn’t it?

## Type Categories in C++

- Fundamental types:
  - **`bool`**, **`char`**, integer types and floating point types
- Enumerated types.
- The **`void`** type.
- Pointer types (like **`int*`**)
- Array types (like **`int[]`**)
- Reference types (like **`int&`**)
- Data structures (like **`vector`**) and classes

## `bool` and `char` Types

- **`bool`** literal values: **`true`** and **`false`**
- **`char`** literal values: **`'H'`**
- Escape characters:

| | |
|---|---|
| – **`\n`** | newline |
| – **`\t`** | tab |
| – **`\b`** | backspace |
| – **`\r`** | carriage return |
| – **`\a`** | BEL |
| – **`\?`** | ? |
| – **`\\`** | \ |
| – **`\'`** | ' |
| – **`\"`** | " |
| – **`\0##`** | octal |
| – **`\x##`** | hex |

## Integer Types

- Simplify things a bit (forget **`unsigned`**…):

| | |
|---|---|
| – **`short`** | 2 bytes |
| – **`int`** | 4 bytes |
| – **`long`** | 4 bytes |
| – **`long long`** | 8 bytes |

- To get a **`long long`** literal append an **`l`** or better yet, an **`L`** to the end of the number.
- What is “unsigned” anyways?

## Floating Point Types

| | |
|---|---|
| • **`float`** | 4 bytes |
| • **`double`** | 8 bytes |
| • **`long double`** | 12 bytes |

- Literal floating point numbers have a decimal place and/or an exponent: **`4.5`**, **`2.3e-7`**, for example.
- A **`float`** literal has an **`f`** or **`F`** appended to it.
- A **`long double`** gives you easy access to extended precision numbers.

## TestTypes Demo

- Your compiler will determine the amount of memory allocated to the numeric types.
- Use the **`sizeof()`** function to find out.
- Do integers wrap around as they do in Java?
- See the demo…
- Note that there is some variation on these types depending on the compiler you are using.

## 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`
  - `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…

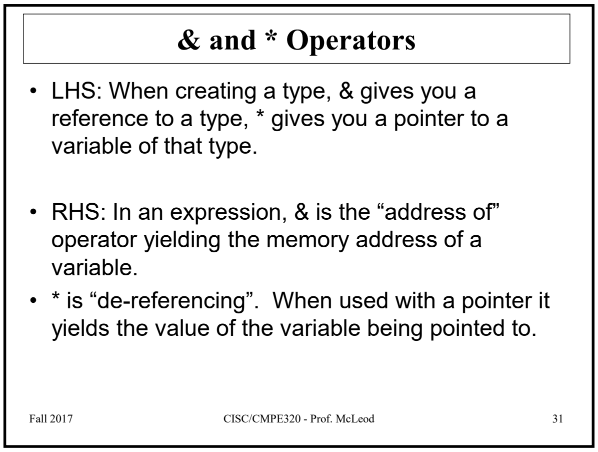
- Pointers are not references and references are not pointers!
- A reference is an "alias for its initializer". [From "C++ Gotchas" by Stephen C. Dewhurst]
- The alias must refer to something. You cannot have a null reference or a reference to `void` or even a reference to a reference.

## 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.

Fall 2017 CISC/CMPE320 - Prof. McLeod 31