Who Am I?

Chris Fischer, B.S.

Software Engineer for Hologic, Inc.

What is C++

- A High Level, Compiled, Unmanaged, Third-Generation Programming Language.
- What’s in the name?

Where did it come from?

- Created by Bjarne Stroustrup at AT&T in the 1970's
- In the world of industrial software, C++ is viewed as a solid, mature, mainstream tool. It has widespread industry support which makes it "good" from an overall business perspective.

Compiled

- Compiled (versus Interpreted).
  - Interpreters validate and run a program one line at a time. The program will run until a syntax error is encountered or until the program terminates.
  - Tend to be slower
  - User has access to source code
  - Good for developing prototype systems
  - Usually designed for simpler languages such as BASIC, Visual Basic, Dbase, SQL

Compiled

Compilers are different.

- Check syntax for errors
- When ALL syntax errors are corrected (through many "*go-'rounds"* correcting and recompiling), a object file (*.o, sometimes seen as *.obj) is generated
- Object files must be linked
- Create an executable file (in our case a.out, but sometimes *.exe)
- Once the compile process is complete the executable file can be run
- Much faster
- User does not usually have access to the source code

Unmanaged

- C++ is Unmanaged (unlike Java and C#).
- You have direct control of what happens in memory, which can be very powerful, but very error prone.
- Java and C#, which both have roots in C#, does not allow this (why?)

Third Generation Language

- First Generation: Machine code (0's and 1's)
- Second Generation: Assembly
- Third Generation: C, C++, Java, C#
- Fourth Generation:
  SQL, macro languages, MATLAB

Other C++'s

- There are many different implementations of C++
- They're all slightly different, because C++ is (basically) platform specific.
- A C++ program compiled on machine A probably won't run on Machine B.

Compilers and File Extensions

- Use .cc for C++ source files
- .C, .cpp, cxx also used.
- Two compilers (maybe more?) available on strauss
  - CC – Sun WS C++ Compiler
  - g++ - GNU C++ Compiler

a.out

- Produces a file called a.out, you run it by typing “a.out”.
- To produce a different filename, use the –o option

CC –o temp.out temp.cc

Getting there..

- There are 6 basic “steps” in creating a C++ program.

1) Editing
2) Preprocessing
3) Compiling
4) Linking
5) Loading
6) Executing

What is compiling?

- Actually 3 steps
- Preprocessing
- Compiling
- Linking

Hello World

- Every programmer new to a language should run a Hello, World program. This is a program that just prints “Hello, World” and exits. Here’s the one for C++

```
// Hello World Program for C++
// Author: Chris Fischer  Date: 9-3-03
#include <iostream>

int main(void)
{
   std::cout <<“Welcome to C++!\n”;
   return 0; //indicates successful termination
}
```

iostream or iostream.h

- Both are C++ classes/libraries for input and output operation.
- What is a library anyways (code reuse)?
- Which one of these should I #include?
- Why?
- What is #include?
  What does the # sign represent?
- What is a using statement?
  Where do I put them?

A more detailed example

```
#include <iostream>
using namespace std;
int main(void)
{
   int a,b;
   std::cin >> a >> b;
   std::cout <<“You entered “<< a << “ and “ << b << std::endl;
   cout <<“Thank you.”;

   return 0;
}
```

Variables and basic assignments

- What is a variable?

```
int a;
int b=0;
a=3;
int c;
cout << c;
float b=1.44;
```

Arithmetic operators

- addition ‘+’
- subtraction ‘-’
- multiplication ‘*’
- division ‘/’
- modulus (remainder of division) ‘%’

Precedence and Associativity

1: ()
2: *, /, %
3: +, -
4: <<, >>
5: <, <=, >, >=
6: ==, !=
7: =

All left to right except for =

Assignment and Equality Operators

- What is the value of the following code?

```
int a=0;

if ( a = 0 )
   cout <<“Equal!”;
else
   cout <<“Not Equal!”;
```

Assignment and Equality Operators

- == and = are very often mistakenly exchange for one another.
- While not foolproof, one good way to minimize this is, whenever you compare a variable to a constant, instead of writing

```
if ( a == 7 )        instead do      if ( 7 == a )
```

- Why do we do this?

Numbers

- **Integers**
- Internal representations of **integers** is a simple function of **powers of 2**. Since the computer only understands a 1 or 0, all values must be converted to base 2 in order to be stored in a computer.
- For Example, the number 107 stored in binary would look like:

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
0	1	1	0	1	0	1	1
128	64	32	16	8	4	2	1

- The value is calculated by selecting, or not selecting values associated with a power of 2. So, 107 is represented as:
- `107 = 64 + 32 + 8 + 2 + 1`
- When storing integers, larger numbers are possible by simply using more bits (2 or 4 byte integers).

Numbers

- **Real Numbers**
  Real numbers are more complicated to represent than integers because you have to deal with 4 distinct components.
- Thus, when we store data using **float** or **double** as the data type, the internal representation becomes more complicated than an **integer** datatype. In order to make sense of it, we are going to have to review a bit about what we know about real numbers in the decimal system.
- A real number can be expressed as -123.456, for example. Some scientific calculators would use the format
  $-1.23456 * 10^{2}$.
  Computers typically use something closer to the second form.

Numbers

- **In Decimal, we would say that -1.23456 * 10 ^{2} consists of the following components:**

1. the sign (-)
2. the Radix is 10 (base 10)
3. the Exponent is 2
4. the Mantissa is 1.23456

- When representing real numbers, the component parts are:

1. Sign bit indicating whether number is positive or negative.
2. The base or radix for exponentiation - this is almost always 2
3. The exponent to which the base is raised (sometimes this is offset by a fixed number called a bias)
4. The mantissa or significand,
   an unsigned integer representing the number

Numbers

A **float** in C++ is typically: composed of 32 bits (4 bytes) comprised of:

1. A sign bit.
2. 8 bit exponent (bias of 127)
3. 23 bit mantissa

A **double** in C++ is typically: composed of 64 bits (8 bytes) comprised of:

1. A sign bit.
2. 11 bit exponent (bias of 1023)
3. 52 bit mantissa

Floats

- Floating Point is wildly inaccurate. Look at this example.

```
float a=1000.43
float b=1000.0;
cout << a – b << endl;
```

This outputs .0429993

Chars

- **Characters**
- **A character representation is stored in a single byte**.
- A computer's natural language is a bit pattern.
- It is humans that require symbols to read.
- A byte can store 256 different bit patterns and application developers use standard representations to determine what symbol (character) the individual bit patterns represent.
- A number of standards have been used to represent character data. As typically happens when people are working separately on separate products they often choose different values to represent a common symbol. For example, the "a" is stored on IBM mainframes using the **EBCDIC** standard as **1000 0001** (81 Hex, or 129 Dec) and another standard, **ASCII** uses **0110 0001** (61 Hex, or 97 Dec).

Chars

- In order for computers to exchange text data, there had to be a **standard for communication** that said "everybody shall use this bit pattern as an "a". One of the first and most successful of these standards was a table called American Standard Code for Information Interchange or **ASCII** for short. It defined only 128 characters of the possible 256 combinations in one byte and left the remaining 128 up in the air. It did not include non-english characters like ç.
- There are some ASCII extensions which define the other 128 characters, but they are not necessarily standard.

Coding Standards

- This is just how we, as humans, format our computer code.
- The computer does not care at all about this. How you format your code will not (directly) affect how your programs runs, at all.
- However, programming (in practice) is a team sports. Coding conventions can help the team work together better.
- So why are these important? Why might it be bad?

The Good

Common standards a few good things happen:

- Programmers can go into any code and figure out what's going on.
- New people can get up to speed quickly.
- People new to C++ are spared the need to develop a personal style and defend it to the death.
- People new to C++ are spared making the same mistakes over and over again.
- People tend to make fewer mistakes in consistent environments.

The Bad

You’ll hear lots of reasons why coding standards are bad / pointless. Some of the reasons are even almost valid. You may hear

- The standard is usually stupid because it was made by someone who doesn't understand C++.
- The standard is usually stupid because it's not what I do.
- Standards reduce creativity.
- Standards are unnecessary as long as people are consistent.
- Standards enforce too much structure.
- People ignore standards anyway.

The Ugly?

- We're going to make up a coding standard for our class. Everyone in our class will use it. It'll be posted on the website.
- It's going to be fairly simple (read: incomplete).
- The whole point of this is to get you thinking about structuring/commenting your programs – not to rigidly enforce it.

So here it is

1) Each Source File should have a comment block at the top, with your name, the date, the program filename, and a short description in it *(also, for Prof. Conrad, your section number)*

2) Variables names should be self describing, exceptions being loop counters in for loops, etc.

   - int myWeightInPounds; int timeoutInMsec;

3) Function names should also be self describing

   - checkForErrors() instead of errorCheck(), dumpDataToFile() instead of dataFile().

4) Use consistent case. camelCaseIsFine

5) Indentation – always exactly 3 spaces, not tabs.

   (Or, use the automatic indentation that your editor gives you)

6) No magic numbers – any number other than 0 or 1 should have a constant defined for it. Example:

   - const int squareFeetInSquareYard = 9;

So here it is

7) Use spaces in all assignment statements, and always use brackets in control / repetition statements

```
for ( int i = 0; i < someConstant; i++ )
{
   //do something
}

if ( a == b )
{
   //do something
}
```

8) Use meaningful comments to describe complex situations. Avoid useless comments.

Decision Statements

- If-Else – simplest decision structure

```
if (some condition is true)
{
   //do something
}
else
{
   //do something else
}
```

Decision Statements

- The conditional in a C++ if can evaluate to 2 different things – a boolean or an int

- if (**someNumber > someOtherNumber**)

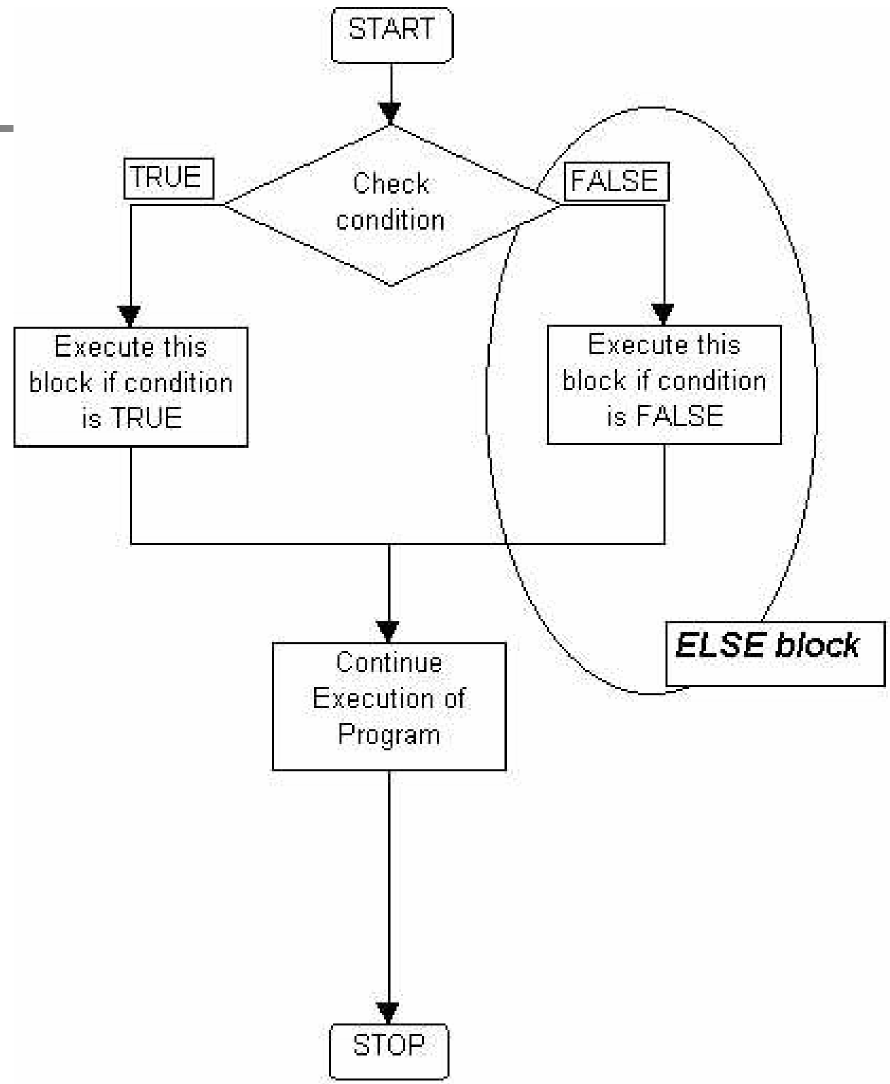
  will evaluate to a boolean (either true or false)

- if statements can also evaluate to an int – the following is valid in C++

- if (**1**) { cout <<“yes!”; }

- With integers, 0 evaluates to false, and any other number evaluates to true

If/Else



START
Check condition
TRUE
FALSE
Execute this block if condition is TRUE
Execute this block if condition is FALSE
ELSE block
Continue Execution of Program
STOP

Simple Example

```
if ( result == answer )
    cout << "You are correct" << endl;
else {
    cout << "Sorry, you're wrong."
    cout << "Try again." << endl;
  }
```

Nested Example

```
if (somenumber==1)
    //do something
else if (somenumber==2)
    //do something else
else if (somenumber==3)
    //do something else
else
    //do nothing
```

Comparison Operators

- Equal to ==
- Not equal to !=
- Greater than >
- Less than <
- Greater than or equal >=
- Less than or equal <=

Boolean Operators

- Not Operator !
- Or Operator ||
- AND Operator &&

Decision Statements in C++

- Couple of C++ *pitfalls* to watch out for.
- First, what is a pitfall?
- C++ code that
  - compiles
  - links
  - runs
  - then does something different than you expect

if statements

- Normally, our decision statements test truth

  if (a > 6) { do something }
- a > 6 will evaluate to boolean true / false
- C++ will also accept an int instead of a boolean. With this, 0 is false, everything else is true
  - if (1) cout <<“true” // will cout “true”
- So, one C++ pitfall are statements like this

  if (-0.5 <= x <= 0.5) return 0;

- This expression does *not* test the mathematical condition

  -0.5 <= x <= 0.5

Instead, it first computes -0.5 <= x, which is 0 or 1, and then compares the result with 0.5.

Ternary Operator

- <condition> ? *true result* : *false result*

cout << (grade >= 60 ? "Passed" : "Failed");

Examples

- cout << (outsidetemp<32&&issnowing==true?“No Class!”:”We have class”);

Examples

```
const int freezing_point=32;
if ( ( outside_temp < freezing_point ) && ( is_snowing == true ) )
    cout <<“No Class!”;
 else
    cout <<“We have class”;
```

Examples

- cout << (outsidetemp<32&&issnowing==true?“No Class!”:”We have class”);

- ```
  if (outsidetemp<32&&issnowing==true)
      cout <<“No Class!”;
  else
      cout <<“We have class”;
  ```

- ```
  const int freezing_point=32;
  if ( ( outside_temp < freezing_point ) && ( is_snowing == true ) )
      cout <<“No Class!”;
  else
      cout <<“We have class”;
  ```

More Examples

- ```
  const int freezing_point=32;
  if (
     ( ( outside_temp < freezing_point ) && ( is_snowing == true ) ) ||
     ( raining_fire == true ) )
       cout <<"No Class!";
   else
       cout <<"We have class";
  ```

- ```
  const int freezing_point=32;
  if (
     ( ( outside_temp < freezing_point ) && ( is_snowing == true ) ) ||
     ( !raining_fire == false ) )
       cout <<"No Class!";
   else
       cout <<"We have class";
  ```

Switch Statement

```
switch ( grade )
{
   case ‘A’:
   case ‘a’:
      cout <<“A work!”;
   break;
   case ‘B’:
   case ‘b’:
      cout <<“B work..”;
   break;
   case ‘C’:
   case ‘c’:
      cout <<“C work….”;
   break;
   default:
      cout <<“Grade less than C”;
   break;  //optional – will exit anyhow
}
```