Lecture 3

More on Flow Control,
More on Functions,
and Intro to Streams

"Absolute C++"
Sections 1.3, 2.2, 4.2

Flow Control -- big if/else statements

Consider the following code:

```
int x;
cin >> x;

if (x == 0)
  cout << "x is zero" << endl;
else if (x == 1)
  cout << "x is one" << endl;
else if (x == 2)
  cout << "x is two" << endl;
else
  cout << "x is not 0,1 or 2" << endl;
```

- Here we only have a single line of code to be executed when the *if* statement is true. No braces ({,}) are used.

Flow Control -- Be careful with if/else

Be careful, though:

```
if (fuelGaugeReading < 0.75)
  if (fuelGaugeReading < 0.25)
    cout << "Fuel is very low. << endl;
else
  cout << "Fuel over 3/4, don't stop!" << endl;
```

- This does **not** produce the desired effect.
- If the reading is between 0.25 and 0.74, what is displayed?
- This is why scope delimiters can be very important

Flow Control -- Be careful with if/else

The right way:

```
if (fuelGaugeReading < 0.75)
{
  if (fuelGaugeReading < 0.25)
    cout << "Fuel is very low. << endl;
}
else
  cout << "Fuel over 3/4, don't stop!" << endl;
```

- Now, we'll get the desired results.
- You might want to *always* use scope delimiters to avoid confusion and mistakes down the road.

Flow Control -- big if/else statements

OK, remember the code from a few slides ago…

```
int x;
cin >> x;

if (x == 0)
  cout << "x is zero" << endl;
else if (x == 1)
  cout << "x is one" << endl;
else if (x == 2)
  cout << "x is two" << endl;
else
  cout << "x is not 0,1 or 2" << endl;
```

- There is nothing wrong with this code, but can be inefficient.

Flow Control -- switch statement

A better way:

```
int x;
cin >> x;        // read in number from console

switch(x)
{
  case 0:
     cout << "x is zero" << endl;
     break;
  case 1:
     cout << "x is one" << endl;
     break;
  case 2:
     cout << "x is two" << endl;
     break;
  default:
     cout << "x is not 0,1 or 2" << endl;
}
```

Flow Control -- switch statement

A switch statement takes the form:

```
switch(integerValue)
{
  case integerValue1:
      statement1;   // multiple statements allowed
      break;
  case integerValue2:
      statement2;   // multiple statements allowed
      break;

  default:
      statementN;   // multiple statements allowed
}
```

- What happens if `break` is omitted in a given `case` statement?

Demonstration #1

switch statement

Flow Control -- for loop

`for (int cntr = init; cntr <= final; cntr += incr)`

- A `for` loop contains three distinct parts:
  - an initialization
  - a test for completion
  - an increment operation
- Initialization
  - The "counter" variable is often declared right in the `for` statement.
  - You have a chance here to set an initial value
- Test
  - When this expression evaluates to `false` (0), the `for` loop terminates.
- Increment
  - An operation which is performed at the "end" of the `for` loop.
- Let's see an example...

Flow Control -- for loop

Say we need to loop 10 times:

```
for (int x=0; x<10; x++)
{
  cout << "Ron DiNapoli" << endl;
}
```

- Initialize -- `x = 0`
- Test -- `x < 10`
- Increment -- `x++`

Flow Control -- for loop

Any (or all) of the three statements in a for loop may be omitted:

```
for (;;)
{
  // Loop forever!
}
```

- Most common use is to create an "infinite loop"
  - same as using `while(true);`

Demonstration #2

for loop

The void type

- We've been using it, but we've never talked about it.
- `void` is used to neatly specify that no return value is required
- can also be used to specify that a function takes no parameters

```
// doNothing() is a function which takes no parameters and
// returns no value
void doNothing(void)
{
  int x = 1;  // well, something, but really nothing :-)
}
```

- You cannot create a variable of type `void`.
- That's because it really isn't a type--the compiler would have no idea how big a "void" is.

Function Overloading

- What do you suppose happens when compiling the following code:

```
void myPrint(int x)
{
  cout << "Integer is: " << x << endl;
}

void myPrint(string s)
{
  cout << "String is: " << s << endl;
}
```

- The `myPrint` function is seemingly defined twice.
- Is this legal?

Function Overloading (cont)

- Yes, it is legal, so long as the *argument list* is different.
- When the compiler compiles this code it can distinguish between the two "versions" by looking at the argument list.
- Consider the following code:

```
void main()
{
  myPrint(1);
}
```

- How does the compiler know which version of "myPrint" to call?
- It looks at the arguments passed. In this case, an integer is passed.
- It looks to see if there is a version of myPrint() that takes a single integer argument.

Function Overloading (cont)

- When it finds it, it will produce compiled code such that when the resulting program is run, we'll get the following output:

```
Integer is: 1
```

- Let's see this in action…

Demonstration #3

Function Overloading

Streams

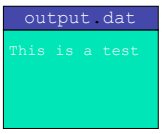
- In the past we've mentioned that when using `cin` and `cout`, we're actually dealing with a *stream* of characters.
- We use the same type of streams to do file I/O
- Let's start with the stream used to write to a file.
- It is called ofstream (for output file stream)
- It is used like this:

```
#include <fstream>
int main()
{
  ofstream outStream;
  outStream.open("output.dat");  // name of file to open
  outStream << "This is a test" << endl;
  outStream.close();         // close the file when done
}
```

Streams (cont)

```
#include <fstream>
int main()
{
  ofstream outStream;
  outStream.open("output.dat");  // name of file to open
  outStream << "This is a test" << endl;
  outStream.close();          // close the file when done
}
```

- Notice how we use outStream just like cout.
- outStream and cout are both streams, but outStream refers to the file "output.dat" instead of the console.
- This program causes the file on the right to be created.



A brief look ahead…

```
outStream.open("output.dat");  // name of file to open
```

- Notice the line of code above.
- From experiences with other languages you might have you can probably guess that :
  - "outStream" is being treated as some form of *data structure*
  - "open()" is some sort of *member* of that data structure
- If you are new to object oriented programming, this may be confusing.
- Bascially, the "open()" *member function* is a function call that pertains to the *object* named "outStream".
- We'll cover more on objects, member functions and *classes* next lecture.

Back to Streams…

```
outStream.close();
```

- If you explicitly open a file stream, you must remember to close it.
  - How else would you open a stream?
  - We'll cover that later :-)
- We've covered how you write to a file, how do you read from one?
- With a different data type called "ifstream"

Reading from streams

```
#include <fstream>
int main()
{
  ifstream inStream;
  inStream.open("input.dat");  // name of file to open
  inStream >> str;
  cout << "We just read in: " << str << endl;
  inStream.close();          // close the file when done
}
```

- Notice the syntax is quite similar to our dealings with "ofstream".
- Once the file is opened we can use "inStream" just like we'd use "cin"--only we'll be *reading* from *input.dat* instead of the console.
- Let's check this out…

Demonstration #4

ofstream and ifstream

Being more careful…

- The code we've written has one major flaw.
- If the open operation fails, we aren't handling the condition properly.
- There are two ways to check that a file was opened.
- Both involve using special *member functions:*
  - The member function *is_open()*
  - The member function *fail()*

```
#include <fstream>
int main()
{
  ofstream outStream;
  outStream.open("output.dat");  // name of file to open
  if (outStream.is_open())
  {
    // Proceed with file manipulation code here…
```

Being more careful…

```
#include <fstream>
int main()
{
  ifstream inStream;
  inStream.open("input.dat");  // name of file to open
  if (inStream.fail())
  {
     cout << "ERROR… could not open stream" << endl;
     return -1;
  }
  inStream >> str;
  cout << "We just read in: " << str << endl;
  inStream.close();          // close the file when done
}
```

- Both ways are valid methods for checking whether or not a file opened. The "fail" method is more generic…

Checking for EOF

- When you don't know how big the file you are reading in from is, you'll need to know how to check for end-of-file.
- There are two ways to do it.
  - is_eof() *member function*
  - Boolean "test" on the stream variable

```
int main()
{
  ifstream inFile("input.dat");  // What's this shortcut?
  long data;
  if (inFile.fail()) return -1;
  while (inFile)               // both lines do the same thing
// while (!inFile.is_eof())     // both lines do the same thing
  {
     inFile >> data;
     cout << "Read in: " << data << endl;
  }
```

Lecture 3

Final Thoughts…