Current Assignments

- Homework 2 is available and is due in three days (June 19th).

- Project 1 due in 6 days (June 23rd)

  Write a binomial root solver using the quadratic equation.

Last Week

- You used an integrated development environment (emacs) to write, compile, and run your program.

- You wrote C++ programs that:
  - Declared variables
  - Performed mathematical operations
  - Read input, wrote output
  - Performed boolean operations
  - Made decisions (branched) based on boolean expressions

This Week

- You will learn how emacs makes programming easier (tabify, jump to error, window control, etc).

- You will write C++ programs that do everything the programs from last week did plus:
  - Variable scoping
  - Iterative control structures
  - Other control structures such as "switch," "break," and "continue."
  - Use all the control structures in concert

Scope of Variables

- The scope of a variable *x* is the portion of the program from which *x* can be "seen."
- Up to this point we have only placed variable declarations at the start of the program so that we wouldn't have to worry about scope resolution.
- With iterative control structures scope starts to become an issue.

Scope of Variables

- Braces are the most common way of defining the scope of a variable.

- Variables declared after an "{" are destroyed when the matching close brace "}" is encountered.

- This allows us to return memory allocated to variables when we don't need them anymore.

Scope of Variables, Example 1

```
int main
{
   {
    int x = 0;
   }
   cout << x << endl;
return 0;
}
```

Scope of Variables, Example

```
int main
{
   int y = 0;
   {
    int x = 0;
   }
   cout << y << endl;
return 0;
}
```

Nesting Scope

- Scopes “{ … }” can be nested just like we have been doing with “if” statements.

Nesting Scope, Example 1

```
{
int x = 0;
{
   int y = 0;
   // x exists, y exists
}
// x exists, y destroyed
}
```

Nesting Scope, Example 1

```
{
int x = 0;
    {
            int y = 0;
            // x exists, y exits, z does not yet exist
    }
    {
            int z = 0;
            // x exists, y destroyed, z exists
    }
// x exists, y destroyed, z destroyed
}
```

Shadowing Variables

Variable scope resolution can become tricky when there are two or more variables with the same name in different scopes.

The compiler would not have allowed this:

```
int x = 0;

int x = 1;
```

We would have received a "variable redefinition" warning.

But…

Shadowing Variables

… it does allow this:

```
int x = 0;
{
  int x = 1;
}
```

This is useful because we don't want to have to use a different name for every single variable in a program. Here one x is said to "shadow" the other.

Shadowing Variables

If I use "x" at different points in the program I could be getting different variables.

```
int x = 0;
// here x = 0
{
  int x = 1;
  // here x = 1
}
// here x = 0
```

Shadowing Variables

When a name is used the name is searched for in the current scope first, then the scope enclosing this one, then the one enclosing that one, etc.

```
int x = 0;
{
    int x = 1;
    {
        int x = 2;
        cout << x;
    }
}
```

```
int x = 0;
{
    int x = 1;
    {
        int x = 2;
    }
    cout << x;
}
```

Other Kind of Variable Scope

- There are lots of other kinds of scope rules and exceptions that we will see throughout the semester (e.g.):

Global scope (global variables)

File scope

Class scope

Static Variables

Iteration

- We can execute any algorithm with just "if" statements.
- But often we need to repeat the same piece of code over and over. Sometimes thousands or millions of times. Writing the same code with if statements alone in this case is impractical.
- Instead we use iterative control structures. Iteration is simply the process of doing something over and over.

The “while” loop

- The “while” loop is the most general type of iteration.
- The “if” control structure said if a certain condition is true then execute the following block of code.
- The “while” loop says so long as the condition is true keep executing the block of code over and over.

The “while” loop, Syntax

```
while( boolean_expression )
{
  statements…
}
```

So long as the boolean_expression is true the statements will be executed over and over.

The “while” loop, Syntax

```
while( boolean_expression )
{
  statements…
}
```

Typically the the block of statements will alter the boolean_expression such that it will eventually be false. If this does not happen an “infinite loop” will occur.

Symptoms of an Infinite Loop

If an infinite loop occurs the same lines of code are executed over and over forever. This is a common "run-time" error.

Symptom 1: Your program prints the same thing over and over in rapid succession.

Symptom 2: Your program appears to freeze and do nothing. In actuality it is working as hard as it can but getting nowhere. Be careful though since your program may just be waiting for input.

The "while" loop, Examples

```
int n = 0, x = 15;
while( n < x)
{
  cout << n;
  n = n + 1;
}
```

```
int x = 0, n = 15;
int fact = 1;
while( x < n)
{
  x = x + 1;
  fact = fact*x;
}
cout << fact;
```

The "do …while" loop

- The do … while loop is very similar to the while loop.
- Where the while loops condition was a pre-condition the "do … while" has a post-condition.

  i.e with a while loop the condition is checked at the start of the loop, a "do… while" checks at the end.
- The effect is that the loop always executes at least once.

The “while” loop, Syntax

```
do
{
  statements…
}
while ( boolean_expression );
```

The statements are executed at least once, then so long as the boolean_expression is true the statements will be executed over and over.

The “do …while” loop, Example

```
int month = -1;
do
{
   cout << “Please input month {1..12}\n”;
   cin >> month;
}
while ((month < 1) || (month > 12));
```

The "do …while" loop, Example

```
int x = 0, n = 15;
int fact = 1;
while( x < n)
{
  x = x + 1;
  fact = fact*x;
}
cout << fact;
```

```
int x = 0, n = 15;
int fact = 1;
do
{
  x = x + 1;
  fact = fact*x;
}while(x < n);
cout << fact;
```

The for loop

- Early programmers used goto statements to accomplish the same tasks as while and for loops but there are many different ways to do the same kind of loop with goto.
- For and while loops constrain the sorts of code you are likely to see and make it easy to recognize right away what the programmer is trying to do.

The for loop

- Anything that can be done with a for loop can be done with a while loop, and vice versa.
- While loops and for loops are equivalent.

The for - while loop equivalence

- If they are equivalent whey bother with the for loop?
- When a programmer uses a for loop it almost always means they are counting something: elements in a list, branches in a tree, it is almost always something discrete.
- Programmers use a while loop when the terminating condition is not so clear, for example reading input from a user.

The for loop, syntax

```
for ( init_variables; pre-condition; statements )
{
   statements…
}
```

e.g.

```
for ( int i = 0; i < 100; i++ )
{
   statements…
}
```

Note: i only has scope within the loop

The for loop, syntax

```
for ( init; pre-condition; statements )
{
   statements…
}
```

Init is executed only once when the loop starts. Pre-condition and statements are executed on every loop.

Pre-condition is executed at the start of the loop and “statements” are evaluated at the end.

The for loop, syntax

```
for ( init; pre-condition; statements )
{
   statements…
}
```

Init can have multiple declarations and statements can have many statements separated by commas:

```
for ( int i = 0, j=10; i < j; i += 5, j++ )
{
   statements…
}
```

The for - while loop equivalence

```
for (int i = 0; i < 100; i++ )
{
   statements…
}
```

```
int i = 0;
while( i < 100 )
{
   statements…
   i++;
}
```

These two sections of code do exactly the same thing

The for loop, syntax

Note: Microsoft Visual C++ 6.0 does not use standard scoping and for loop syntax. MSVC 6 allows you to initialize variables in the for loop but not declare them there. This means they have scope outside the for loop.

Instead of:

```
for ( int i = 0; i < 0; i++ )
{
statements…
}
```

You must write:

```
{
  int i;
  for ( i = 0; i < 0; i++ )
  {
     statements…
  }
}
```

The for loop, example

```
int result = 1, base = 2, power = 0;


cin >> power;


for ( i = 1; i <= power; i++)
{
   result = result * base;
}
cout << result;
```

Nested Control Structures

All control structures can be nested in C++.

Typically a program will consist of an outer loop (usually a "while" loop), that runs the program over and over until some condition is met and…

Selection control structures such as "if" statements that decide which code to run.

Nested Control Structures

```
for ( int i = 0; i < 2;  i++ )
{
   for ( int j = 0; j < 4; j++)
   {
        cout << i << “ ”<< j << “ ”;
   }
   cout << endl;
}
```

0 0 0 1 0 2 0 3

1 0 1 1 1 2 1 3