6.096 Lab 1

Due: 7 January 12:30:00

September 20, 2011

1 Additional Material

1.1 Constants

A constant is an expressions with a fixed value. Kinds of constants:

- Literals: used to express particular values within the source code; can be integers (ex: 1, −0, -17), floating points (ex: 3.1415926535897, 1., 6.096, 3), characters (ex: 'a', ' ', '\n'), strings (ex: "6.096", "a"), booleans (ex: true, false)
- Defined constants (`#define`): user-defined named constants that do not require memory-consuming variables. When the preprocessor sees the `#define`, it will replace every subsequent occurrance of the identifier in the source code.

```
1  #include <iostream>
2
3  using namespace std;
4
5  #define PI 3.14159
6  #define NEWLINE '\n'
7
8  int main()
9  {
10     double r = 5.0;
11     double circle;
12
13     circle = 2 * PI * r; // circle = 2 * 3.14159 * r;
14     cout << circle << NEWLINE; // cout << circle << '\n';
15
16     return 0;
17 }
```

- Declared constants (`const`): user defined constants with `const` prefix with a specific type that are declared the same way as variables. The value of a `const` variable cannot be modified.

```
1 const int pathwidth = 100;
2 // pathwidth = 2; this will cause a compiler error!
3 const char tabulator = '\t';
4 cout << "tabulator =" << tabulator << '\n';
```

1.2 L-values, R-values

lvalue is short for "left hand side value" (of an assignment).

Examples of **non**-lvalue expressions:

- `3+3 // you can't assign 3+3 to something`
- `"str" // the literal "str" can't take on another value`
- `const int i = 3 // can't change the value of const variable`

Examples of lvalue expressions:

- `int var // var is an lvalue because we can assign it with some value`
- `float x`

rvalue is short for "right hand side value" because rvalues can appear on the right hand side of an assignment. Anything with a well-defined value can be an rvalue, including an assigment: `(x = 5)` can be used as an rvalue whose value is 5, e.g. `y = (x=5);`.

1.3 Increment, Decrement operators (++, --)

`a++` and `++a` are shorthand for `a = a + 1` with a big warning sign:

- `++a` will increment `a` and then return the value (so it will return one greater than the original value)
- `a++` will return the current value and then increment
- `--a` will decrement `a` and then return the value (so it will return one less than the original value)
- `a--` will return the current value and then decrement

```
1 // this code outputs 0 to 9
2 for(int i = 0; i < 10;)
3 {
4     cout << i++ << "\n";
5 }
6
```

```
// this code outputs 1 to 10
for(int i = 0; i < 10;)
{
    cout << ++i << "\n";
}
```

1.4 Assignment operators

Assignment ops include `+=`, `-=`, `*=`, `/=`, `%=`, etc.; `a += 5` is equivalent to `a = a + 5`, etc.

1.5 Type Conversions

Used for changing between data types. Also called "casts." Type conversions are implicit when changing from smaller data type to a bigger data type or data type of same size (e.g. `float` to `double` or `int` to `float`). Type conversions usually must be explicitly stated when changing from bigger datatype to smaller datatype or when there could be a loss of accuracy (e.g. `int` to `short` or `float` to `int`), but implicitly converting from a `double` to an `int` will not generate a compiler error (the compiler will give a warning, though).

```
int x = (int)5.0; // float should be explicitly "cast" to int
short s = 3;
long l = s; // does not need explicit cast, but
            // long l = (long)s is also valid
float y = s + 3.4; // compiler implicitly converts s
                   // to float for addition
```

1.6 Operator precedence

Like arithmetic, C++ operators have a set order by which they are evaluated. The table of operators and their precedence is listed in the following table:

1	() [] -> . ::	Grouping, scope, array/member access
2	! ~ * & sizeof (type cast) ++ –	(most) unary operations, sizeof and typecasts
3	* / %	Multiplication, division, modulo
4	+ -	Addition and subtraction
5	<< >>	Bitwise left and right shift
6	< <= > >=	Comparisons: less than, etc.
7	== !=	Comparisons: equal and not equal
8	&	Bitwise AND
9	^	Bitwise exclusive OR
10	|	Bitwise inclusive (normal) OR
11	&&	Logical AND
12	||	Logical OR
13	?:	Conditional expression (ternary operator)
14	= += -= *= /= %=, etc.	Assignment operators
15	,	Comma operator

1.7 Ternary operator (?:)

An operator that takes three arguments and defines a conditional statement.

```
1 if(a > b)
2     result = x;
3 else
4     result = y;
```

is equivalent to

```
1 result = a > b ? x : y;
```

1.8 `switch` statement

A type of selection control statement. Its purpose is to allow the value of a variable or expression to control the flow of program execution via multiple possible branches. Omitting `break` keywords to causes the program execution to "fall through" from one block to the next, a trick used extensively. In the example below, if $n = 2$, the fifth case statement (line 10) will match the value of n, so the next line outputs "n is an even number." Execution then continues through the next three case statements and to the next line, which outputs "n is a prime number." This is a classic example of omitting the `break` line to allow for fall through. The `break` line after a case block causes the switch statement to conclude. The `default` block (line 21) is executed if no other cases match, in this case producing an error message if n has multiple digits or is negative. The `default` block is optional in a `switch` statement.

```
1 switch(n) {
```

```
  case 0:
    cout << "You typed zero.\n";
    break;
  case 1:
  case 4:
  case 9:
    cout << "n is a perfect square.\n";
    break;
  case 2:
    cout << "n is an even number.\n";
  case 3:
  case 5:
  case 7:
    cout << "n is a prime number.\n";
    break;
  case 6:
  case 8:
    cout << "n is an even number.\n";
    break;
  default:
    cout << "Only single-digit positive numbers are allowed.\n";
    break;
}
```

1.9 break

Used for breaking out of a loop or switch statement.

```
// outputs first 10 positive integers
int i = 1;
while(true)
{
    if(i > 10)
        break;
    cout << i << "\n";
    ++i;
}
```

1.10 continue

Used for skipping the rest of a loop body and continuing to the next iteration.

```
// print out even numbers in range 1 to 10
for(int i = 0; i <= 10; ++i)
{
```

```
4       if(i % 2 != 0)
5           continue; // skips all odd numbers
6       cout << i << "\n";
7   }
```

1.11 References

- Wikipedia: http://en.wikipedia.org/
- Cplusplus: http://www.cplusplus.com/

2 “Hello, World!”

This section is about writing the canonical “Hello, World!” program and its derivatives. Now is a good time to get used to your development environment. Submit each program in a separate source file named ⟨section⟩.⟨subsection⟩.cpp.

2.1 Hello World I

Write a program that outputs “Hello, World!” by printing a `const char *` with value “Hello, World!”.

2.2 Hello World II

Write a program that outputs “Hello, World!” n times (where n is a nonnegative integer that the user will input) with:

- a `for` loop.
- a `while` loop.
- a `do...while` loop.

3 More Programs

Now that you have had some practice working in your development environment, here are some more challenging tasks. Again, submit your work (source code and answers to questions) in an appropriately named text file.

3.1 Scope

For these questions, you are encouraged to use a computer.

1. Below is a sample program. Use it to answer the following question: What happens if we declare the same name twice within a block, giving it two different meanings?

```
#include <iostream>

using namespace std;

int main()
{
    int arg1;
    arg1 = -1;
    int x, y, z;
    char myDouble = '5';
    char arg1 = 'A';
    cout << arg1 << "\n";
    return 0;
}
```

   Hints: Did your program compile? If so, what does it print? If not, what error message do you get?

2. Below is a sample program. Use it to answer the following question: What happens if we declare an identifier in a block, and then redeclare that same identifier in a block nested within that block?

```
#include <iostream>

using namespace std;

int main()
{
    int arg1;
    arg1 = -1;
    {
        char arg1 = 'A';
        cout << arg1 << "\n";
```

```
12      }
13      return 0;
14  }
```

Hints: Did your program compile? If it does, what does the program output? If not, what error message does it produce?

3. Below is a sample program. Use it to answer the following question: Suppose an identifier has two different declarations, one in an outer block and one in a nested inner block. If the name is accessed within the inner block, which declaration is used?

```
1  #include <iostream>
2
3  using namespace std;
4
5  int main()
6  {
7      int arg1;
8      arg1 = -1;
9      {
10         char arg1 = 'A';
11         cout << arg1 << "\n";
12     }
13     return 0;
14 }
```

4. Below is a sample program. Use it to answer the following question: Suppose an identifier has two different declarations, one in an outer block and one in a nested inner block. If the name is accessed within the outer block, but after the inner block, which declaration is used?

```
1  #include <iostream>
2
3  using namespace std;
4
5  int main()
6  {
7      int arg1;
8      arg1 = -1;
9      {
10         char arg1 = 'A';
11     }
12     cout << arg1 << "\n";
13     return 0;
14 }
```

5. Below is a sample program that will not compile. Why not? By moving which line can we get the code to compile?

```
1 using namespace std;
2
3 int main()
4 {
5     cout << "Hello, World!\n";
6     return 0;
7 }
8
9 #include <iostream>
```

3.2 Basic Statistics

Given a list of N integers, find its mean (as a `double`), maximum value, minimum value, and range. Your program will first ask for N, the number of integers in the list, which the user will input. Then the user will input N more numbers.

Here is a sample input sequence:

```
3 <-- N
2
1
3
```

Three numbers are given: 2, 1, 3. The output should be as follows:

```
Mean: 2
Max: 3
Min: 1
Range: 2
```

3.3 Prime Numbers

Write a program to read a number N from the user and then find the first N primes. A prime number is a number that only has two divisors, one and itself.

3.4 Multiples of numbers

3.4.1 Ternary operator

Write a program that loops indefinitely. In each iteration of the loop, read in an integer N (declared as an `int`) that is inputted by a user, output $\frac{N}{5}$ if N is nonnegative and divisible by 5, and -1 otherwise. Use the ternary operator (`?:`) to accomplish this. (*Hint:* the modulus operator may be useful.)

3.4.2 `continue`

Modify the code from 3.4.1 so that if the condition fails, nothing is printed. Use an `if` and a `continue` command (instead of the ternary operator) to accomplish this.

3.4.3 `break`

Modify the code from 3.4.2 to let the user break out of the loop by entering -1 or any negative number. Before the program exits, output the string "Goodbye!".

3.5 What does this program do?

Do these problems without the use of a computer!

1. What does this snippet do? Try doing out a few examples with small numbers on paper if you're stuck. (*Hint:* Think about numbers in binary notation – in base 2. How would you express a number as a sum of powers of 2? You may also find it useful to note that multiplying by 2^n is equivalent to multiplying by 2 n times. You should also keep in mind the distributive property of multiplication: $a(x+y) = ax + ay$.)

```
// bob and dole are integers
int accumulator = 0;
while(true)
{
  if(dole == 0) break;
  accumulator += ((dole % 2 == 1) ? bob : 0);
  dole /= 2;
  bob *= 2;
}
cout << accumulator << "\n";
```

2. What does this program do? What would the operating system assume about the program's execution?

```
#define O 1 // That's an oh, not a zero
int main()
{
    return O;
}
```

3. What does this program do?

```
// N is a nonnegative integer
double acc = 0;
for(int i = 1; i <= N; ++i)
{
```

```
5       double term = (1.0/i);
6       acc += term * term;
7       for(int j = 1; j < i; ++j)
8       {
9           acc *= -1;
10      }
11  }
12  cout << acc << "\n";
```

4 Factorials Gone Wrong

This section focuses on debugging programs. We will start off with a simple factorial program and do a step by step troubleshooting routine.

4.1 Writing the factorial program

Here is the code for a factorial program. Copy it into your IDE and verify that it compiles.

```
1  #include <iostream>
2
3  using namespace std;
4
5  int main()
6  {
7      short number;
8      cout << "Enter a number: ";
9      cin >> number;
10
11     cout << "The factorial of " << number << " is ";
12     int accumulator = 1;
13     for(; number > 0; accumulator *= number--);
14     cout << accumulator << ".\n";
15
16     return 0;
17 }
```

What do you get when you enter the following values: 0, 1, 2, 9, 10?

4.2 Breaking the program

Run the program and enter -1. What happens? How can you change the code such that it won't exhibit this behavior?

4.3 Breaking the program II

Try entering some larger numbers. What is the minimum number for which your modified program from 4.2 stops working properly? (You shouldn't have to go past about 25. You may find Google's built-in calculator useful in checking factorials.) Can you explain what has happened?

4.4 Rewriting Factorial

Modify the given code such that negative inputs do not break the program and the smallest number that broke the program before now works. Do not hardcode answers.

4.5 Rewriting Factorial II

Since we know that only a small number of inputs produce valid outputs, we can alternatively hardcode the factorials of these inputs. Rewrite the program from the previous part ("Rewriting Factorial") using a switch statement to demonstrate for inputs up to 10 how you would do this. (Of course, the code for inputs above 10 would basically be the same, but you do not need to go through the work of finding all those large factorials.)

4.6 Further testing

Are there any other inputs we have to consider? Why or why not?

6.096 Introduction to C++

January (IAP) 2011