Intro to Programming in C++

Important Note: This document is aimed at intro to programming students at MSSM. It presents a simplified subset of the c++ language and intentionally "glosses over" certain details. Most notably, students work almost exclusively with the *string* class and we avoid using library functions that work with "c-style" null-terminated character strings. Basically, we pretend that char* doesn't exist for as long as possible to avoid muddying the waters for beginning programmers.

Vocabulary

Type

Type is a concept in computer programming that indicates the possible values and operations supported by a variable, function parameter, literal, or expression. Examples of some basic (primitive) types in c++ include: int, double, string, bool, and char.

As an example, if something has type *int*, we know that it can hold positive or negative integral values, and that it supports a variety of arithmetic operations such as addition, subtraction, multiplication, and so on.

Value

Literals, expressions, and variables have values. The *Type* tells you what values are possible, and what operations can be done on those values. For example, in c++ a *double* variable can hold numbers in the range +- 1x10^308 with about 15 decimal places of precision.

Name/Identifier

A *name* or *identifier* gives the programmer a way to refer to a particular variable or a function later on in the program. Names in C++ consist of letters, numbers, and the underscore _ character. Spaces are not allowed in names.

Object

The term Object means a lot of things in computer science. At this point, just think of it as a thing with a reserved chunk of memory associated with it. This memory is where the value of the object is stored. The fact that it has memory associated with it is what allows the value of the object to be changed: a new value can be placed into the memory.

Statement

A statement is a section of code that tells the computer to carry out some action. It is a bit like a sentence in written human language. In c++, a single line of code terminated by a semicolon is a statement. In addition, larger chunks of code are also considered statements, such as conditional (if/else, or switch) statements, loops, or blocks of code enclosed in "curly braces"

Expression

An expression is any combination of literals, variables, operators, and function calls that can be *evaluated* (turned into a value). The basic characteristics of an expression that you should remember are that it:

- has a Type
- can be evaluated (and therefore has a value)

Examples:

```
3*x + 4         // type int (assuming x is int)
7               // type int,    value 7
sin(4*angle)    // type double (because the sin function
                //    return type is double)
7/10            // type int,    value 0
3.4/10          // type double, value 0.34
```

Variable

A variable in computer science has these four defining characteristics: A variable:

- has a Type
- has a Value

- has a Name (aka Identifier)
- is an Object (has an area of memory/storage associated with it)

Examples:

```
int x = 3;      // x is a variable of type int
                // it has been assigned the value 3

double y{10.4}; // y is a variable of type double
                // it has been initialized with the
                // value 10.4

vector<string> names;  // names is a variable of type
                       //   "vector of strings"
                       // it has the default value of
                       // "empty vector"
```

Literal

A *literal* is a way to specify a single, fixed value in a program. The *type* of the value is indicated by the presence (or absence) of certain symbols such as quotation marks. The two basic characteristics to remember about a literal are:

- Type
- Value

Examples:

```
10          type: int     value: 10
"Hello"     type: string  value: Hello
'x'         type: char    value: x
3.4         type: double  value: 3.4
```

C++ Syntax

Statements

Declaring Variables

```
int x = 3;  // both of these statements are equivalent
int x{3};   // and they both declare AND initialize the variable x

double y;   // this declares y, but does not initialize
            // the value.  In almost all cases, you should
            // initialize variables.  A notable exception is
            // when you immediately follow the declaration
            // with "cin >> y"

string s;   // "non-primitive" types such as strings and vector
            // don't need to be explicitly initialized, since
            // they are automatically set to a default
            // value (in this case, the *empty string* "")
```

Assignment Statements

An assignment statement is used either to initially set or to update the value of a variable. A single "Equals" sign is used for assignment. Examples:

```
x = 100;
name = "Bob";
total = average({1, 7, 10, 20});
```

In an assignment statement, the expression on the right hand side of the assignment operator (=) is evaluated, and the resulting value is then placed into the variable on the left hand side. For example, given a variable y with the value 27, the

statement shown below will first subtract 7 from the value of y (27), resulting in a value of 20. This value is then copied into the variable *y*. Therefore, after the statement runs, *y* will have the value 20.

```
y = y - 7;
```

Input and Output Statements

cin and *cout* are used to read and write text to the console or terminal window. *cin* and *cout* are "stream" objects.

The operators « and » are the stream insertion and extraction operators. In class I normally refer to them as the stream output and stream input operators.

```
cout << "Hello World" << endl;   // send the string "Hello World" to the
                                 // output window, followed by a "newline" character

// to read a value from the user...
double x;    // first create a variable to hold the value
cin >> x;    // then read into that variable
```

Conditional Statement: If and If/Else

And *if statement* can be used to execute (run) a statement or set of statements if a certain condition is true. An *if* statement can include an *else clause* which contains statements to be executed if the condition is **not** true. The basic form of an *if* statement is as follows:

```
if (condition) {
    ... these statements are run if the condition is true
}
```

or

```
if (condition) {
    ... these statements are run if the condition is true
}
else {
    ... these statements are run if the condition is false
}
```

Note: although the "curly braces" are not strictly required by c++ if there is only a single statement in an *if* or *else*, in the introduction to programming class it is expected that students will **always** use curly braces. Many companies and other institutions follow this same convention to avoid confusion and errors in their code.

The *condition* of an *if* statement can be any expression whose value is of type bool. In other words, it is a "true/false" question. Often, the condition of an *if* statement will use one or more of the *conditional operators*:

conditional operator	name/meaning
==	Equality (Equals)
<	Less Than
>	Greater Than
<=	Less than or Equals
>=	Greater than or Equals
!=	Inequality (Not Equals)

Examples:

```
if (x < 10) {
    // do something
}

if (name == "Bob") {
    // do something
}
```

Note that a single "equals sign" (=) is the *assignment operator* **not** the test for equality.

More complex conditions can be created by using the *logical operators* to combine conditions.

logical operator	name/meaning
&&	and
||	or
!	not

Examples:

```
if (x > 50 && y < 100) {
    // do this if x is in the range 51 to 99 (assuming x is an integer)
}

if (answer == "yes" || answer == "ok") {
    // do this if answer is either "yes" or "ok"
}
```

Note: Be careful not to write something like this: `if (10 < x < 100) {`

It will almost certainly not give the result you intend or expect.
The condition should instead be written using an *AND* operator, like this: `if (10 < x && x < 100) {`

Looping Statement: while

The *while* statement allows a statement or block of statements to be executed multiple times for as long as a condition is true. The basic form of a *while* statement looks almost the same as an *if* statement.

```
while (condition) {
    // statement(s)
}
```

As a more concrete example, here is a while statement that counts down from 10 to 1, printing out those values:

```
int x = 10;
while (x > 0) {
    cout << x << endl;
    x = x - 1;
}
```

The following example allows the user to enter values as long as the value entered is greater than 10. Note: to make sure the while loop is entered the first time, the variable is initialized to a value greater than 10.

```
double value = 11;  // make sure we get into the loop
while (value > 10) {
    cin >> value;
    cout << "You entered: " << value << endl;
}
cout << "That was less than or equal to 10... guess you're done" << endl;
```

Looping Statement: for

It is very common for loops to involve "counting" through a series of values, or simply to execute a piece of code a specific number of times. In this case, it is more convenient to use a *for* loop. A for loop provides a convenient way to initialize a variable, update that variable each time through the loop, and to test a condition to determine whether the loop should continue.

The basic format of a *for* loop is as follows:

```
for (initialization; condition; update) {
    // statements to repeat go here
}
```

The *initialization* section is typically used to declare and initialize a loop variable. The *condition* section often compares the loop variable to a value. The *update* section typically modifies the loop variable (adding one to it, for example.)

Here is an example of a *for* loop that counts from 1 to 10 and displays those numbers:

```
for (int i = 1; i <= 10; i++) {
    cout << i << endl;
}
```

Note: In the example above, the *postincrement* operator (++) is used to add one to the variable i each time through the loop. This is basically a shorthand way of writing $i = i + 1$.

Conditional Statement: switch

A *switch* statement can sometimes be a good alternative to a series of *if/else if/else if/ else* statements. The basic form of a switch is as follows:

```
switch (expression) {
    ... cases
}
```

The expression must evaluate to a integer or integer-like value (int, char, or an enumeration). The cases specify *literal* values, and when the switch statement is executed the flow of control will "jump" to the appropriate case if a match is found. Here is a more complete example:

```
switch (c) {
    case 'a':
    case 'A':
        cout << "Aye Aye Captain!" << endl;
        break;
    case 'b':
    case 'B':
        cout << "Bees go Buzz!" << endl;
        break;
    default:
        cout << "Nothing special about that character, I guess..." << endl;
        break;
}
```

In the example above, if the variable c contains the character 'a', 'A', 'b', or 'B' it will jump to the appropriate location and display a message. If c is any other character, it will jump to the *default* and display that message. Note that the *break* keyword is needed to jump to the end of the switch after the appropriate case is processed. Without a break, the flow of control will "fall through" and the code for the next case (or default) will be executed.

Functions

Basics

A function is a named, reusable chunk of code that can be called/executed elsewhere in your code. Functions can optionally accept one or more arguments, and can optionally return a value to the caller.

The basic form of a function is as follows:

```
returntype functionName(parameters)
{
    function body
}
```

The first line of the function is called the *signature*. It includes the "return type" the name of the function and any parameters. The *signature* indicates whether the function returns a value, and if so, what type that value is. It also indicates whether the function accepts arguments, and if so, the number and types of those arguments.

For example, the following signature describes a function named "multiply" which has two parameters that are of type double, and which returns a value that is a double:

```
double multiply(double x, double y)
```

After the signature comes the *body* of the function. The body contains the statements that are executed when the function is called. If the function returns a value, the body will contain a "return statement" that indicates the value that should be returned by the function. Continuing the example above, here is an example of a very simple function definition that includes the function body:

```
double multiply(double x, double y)
{
    return x * y;
}
```

If a function does not return a value, use *void* as the return type:

```
void sayHello(string name)
{
    cout << "Hello " << name << endl;
}
```

Calling functions

When calling functions, type the function name, followed by parentheses (even if there are no parameters):

```
cout << multiply(3, 7) << endl;
sayHello("Bob");
funcWithNoParams();
```

References

Normally, when you call a function and use a variable as an argument to the function, only the *value* contained in that variable is actually passed into the function. In other words, the parameter holds a **copy** of the argument. Any changes to a parameter inside the function does not change the original variable. The example below demonstrates this situation.

```
void func(double x)
{
    x = x + 100;       // this line modifies x, but does not affect the variable y below
}

int main()
{
    int y = 10;
    func(y);
    cout << y << endl;  // this will print out 10
}
```

If a function parameter is a *reference* (denoted using the & symbol), then the parameter becomes an *alias* for the variable passed into the function as an argument, rather than a new variable holding a copy of the value. The example below shows this difference.

```
void func(double& x)
{
    x = x + 100;       // this line modifies the variable y that was passed in
}

int main()
{
    int y = 10;
    func(y);
    cout << y << endl;  // this will print out 110
}
```

vectors

The primitive types (int, double, char, etc) work with individual values. However, it is important to be able to work with groups or collections of related values without needing to create a separate variable for each value. C++ provides the *vector* type as one way of working with multiple values at once. A vector can store an arbitrary number of values (elements) all

having the same type. For example, you can create a vector to hold integers, or a vector to hold strings. The syntax for creating vectors is as follows:

```
vector<int> numbers;   // declares a variable named "numbers" which can hold an arbitrary
                          number of integers
vector<int> names;     // declares a variable which can hold an arbitrary number of strings
```

Vectors hold their values in sequentially numbered "slots." The individual values within a vector are referred to as the *elements* of the vector, and the numbers that refer to the locations of those values within the vector are called the *indices* of the elements. Vectors in c++ are "zero-based" which means that the first element in the vector is at index 0, the second is at index 1, and so on.

The number of elements currently stored in a vector is called the *size* of the vector. Because of zero-based indexing, the index of the last element in a vector is one less than the size. For example, a vector of size 5 would have elements stored at indices 0 through 4.

When a vector is created, it has a size of zero by default, which means there are no spaces/slots for elements. The size of the vector can be changed in number of different ways, but in our class the most common methods will be one of the following:

- initializing the vector with a specific set of values: `vector<int> number{1,5,2,7};`
- using the `resize()` method to explicitly set the size
- using the `push_back` method to add an element to the vector, increasing its size by one

To access individual elements (values) within a vector, use *indexing notation*. Indexing notation uses square brackets [] to denote the index. Indexing notation can be used to either read a value or to write (change or update) a value within a vector. Example:

```
vector<double> numbers;      // create an empty vector
numbers.resize(10);          // resize it so that it now has 10 elements (zero by default)

cout << numbers[1] << endl;  // display the second element of the vector
                                (remember zero-based indexing!)

numbers[1] = 20;             // set the second element to have the value 20

for (int i = 0; i < numbers.size(); i++) {    // loop over the indices from 0 to size()-1
    cout << numbers[i] << endl;               // display the value of the given element
}
```

Functions of vectors

As with any *type*, vectors may be used as variables, function parameters, or function return types. For example, a function that computes the sum of a group of integers could be written as follows:

```
double sum(vector<int> values)
{
    double total = 0;

    for (int i = 0; i < values.size(); i++) {
        total = total + values[i];
    }

    return total;
}
```

An example of a function that *returns* a vector might be a function that creates a vector of integers between a lower and upper bound (inclusive). Such a function could be written like this:

```
// produce vector of integers from startNumber to endNumber inclusive
// endNumber should be >= startNumber
vector<int> numberRange(int startNumber, int endNumber)
{
    vector<int> result;

    int value = startNumber;
```

```
    while (value <= endNumber) {
        result.push_back(value);
        value = value + 1;
    }

    return result;
}
```

Reminder: In c++, parameters are passed by value (copied) by default, even for "complex" types like vectors. This is different than many other languages like python or javascript, by the way. If you want a function to change a vector that was passed in as a parameter, you can use *reference* notation using the & operator. Normally, a function recieves a complete copy of the vector that was passed in, so if that vector is resized or if the elements are changed, it will not affect the original vector. Here is an example that demonstrates that fact:

```
double sumOfSquares(vector<double> values)
{
    // square all the values of the vector passed in
    for (int i = 0; i < values.size(); i++) {
        values[i] = values[i] * values[i];  // changes the local copy of the vector passed in
    }

    return sum(values);  // then return the sum of those squares
}

...

vector<double> numbers{1,3,5,7};

cout << sumOfSquares(numbers) << endl; // prints out 84

printVector(numbers);   // prints out {1, 3, 5, 7}
```

Common vector operations

You should understand the following basic operations that can be done with vectors:

- Creating vectors

  ```
  vector<string> words;  // create an empty vector
  vector<string> names{"Bob", "Joe", "Jane", "Milo"};    // create a vector with starting elements
  ```

- Accessing elements using indexing notation

  ```
  cout << words[0];    // get the first element
  names[2] = "Janet";  // set the third element
  ```

- Set the size of a vector

  ```
  names.resize(3);  // set the number of elements to 3
                       this will discard extra elements or
                       add new default elements as needed
  ```

- Append an element to a vector ("growing" its size by 1)

  ```
  names.push_back("Bud");
  ```

- Get the current size of a vector

  ```
  cout << "There are: " << names.size() << " names";
  ```

- Loop (interate) over the elements of a vector

  ```
  for (int i = 0; i < words.size(); i++) {
      cout << "Word #" << (i+1) << " is " << words[i] << endl;
  }
  ```

- Insert a value into the middle of a vector

```
values.insert(values.begin() + 3, 101);  // insert the value 101 into location 4 of the
                                            vector.  Moving existing values over to make room
```

- Remove a value from the middle of a vector

```
values.erase(values.begin() + 3);  // remove the 4th element, moving the 5 and higher elements
                                      down to "fill the gap"
```