C++ Programming[1]

These notes present a concise review of C++ additions to the C language; note that most all C syntax can also be used in C++. We recommend you supplement this material with an online tutorial (e.g., www.cplusplus.com). ** Please use these notes as supplementary to the "Review of C Programming" notes – review the C notes first.

C++ can be used for procedural or object-oriented programming. In procedural programming, a logical step-by-step sequence of calculations, e.g., historically described in a flow chart, is used to specify each function or procedure, and a single high-level main program executes a sequence of function calls. Code is then written to represent these steps, and data is defined as needed. Readability of procedural code is improved by careful definition and use/re-use of simple functions that are then composed. In object-oriented programming, abstract classes are first defined to organize data and functions into "objects" that can be used and re-used in functions. Objects offer an organizational structure that makes data easier to group and manipulate. Most first-year engineering programming courses, including Engr101 at the University of Michigan, teach C++ (and/or Matlab) as a procedural programming language. In C++, one learns to write a special function called `main()` that serves as the entry point to the program. The `main()` function in turn calls functions that initialize variables, execute the main procedure(s) until they complete/end, then gracefully exit.

The basic structure of a C++ program, given a `.cpp` file extension, is shown below. The "#" indicates a preprocessor statement (e.g., `#include` cuts/pastes declaration/definition information into the code). The curly braces {} are used to group parts of code together. The function, called `main`, has no input arguments, which would be specified in the parenthetical list including data type, and returns an integer data type (`int`) indicating whether the program succeeded (0) or exited with a non-zero error code. The double slash (//) denotes a comment. Edit code using your favorite editor (e.g., `nano, pico, gedit, emacs, vi, vim` on Linux) – just be sure the editor recognizes C++ syntax so you will have appropriate help with color-coding key words, matching curly braces, and indenting code blocks properly. In the below example, `cout` is used to print data to the screen. An analogous command `cin` reads data from the keyboard.

[1] Primary references: www.cplusplus.com; W. Savitch, *Problem Solving in C++*, Addison Wesley, 2008; J. Holloway, *Introduction to engineering programming: solving problems with algorithms,* Wiley, 2004; Stroustrup, B., *C++ Programming Language,* 3rd ed., Pearson Ed, 1994.

Send questions/comments to Prof. Atkins (ematkins@umich.edu)

```
#include <iostream>     // Input/output stream definitions.
using namespace std;    // Indicates use of standard C++ objects/commands.
int main()
{         // The block of code defining main() is grouped by the curly braces
  cout << "Hello world." << endl;  // cout = console output.
          // A semicolon designates the end of each complete code statement.
  return 0;
}
```

Functions and Procedures (note new material on pass-by-reference)

There are three ways to pass variables into a C++ function or procedure:

1. Pass by value (shown in the example above)
2. Pass by reference (&)
3. Pass by memory address pointer (*)

In pass-by-value, each variable is copied into a local memory block used only by the function into which the variable is passed. This protocol eliminates the possibility that this variable could be inadvertently changed by the function since the function only uses a "copy" of the original. The drawback to copy-by-value is that processing and memory overhead are required to copy the variable. This overhead is trivial when only a few values are copied but can become an issue when [repeatedly] passing large data structures by value.

Pass-by-reference is denoted in the function declaration and implementation with an ampersand (&), e.g., `double &x,` rather than simply `double x`. With this strategy, a function receiving this input argument "references" the same memory block for the variable as was used in the calling function. Pass-by-reference is useful to reduce the copy overhead when passing large data structures and is useful when a programmer wants a procedure to be able to change an input argument variable's value(s). Pass-by-reference is also convenient from a syntax perspective because the variable can be used in function code exactly as if it had been passed by value.

Pass-by-pointer is denoted in the function declaration and implementation with a star (*), e.g., `double *x`, in the argument. An equivalent and acceptable representation is an empty array, e.g., `double x[]`. When passing by pointer, the variable references a memory address inside the function. To access the memory address, one uses the variable alone; to access an element of the memory block to which the variable points, one must use the pointer dereference operator *, e.g., *x = 3.14; or an array subscript, e.g., x[0] = 3.14; Arrays should be passed by pointer to the array, as shown in examples below. Pointers to single variables (one-dimensional arrays) can also be passed by memory. When passing by pointer, the values stored in the memory block referenced by the pointer variable CAN be changed in the function as a side effect of the function with new values available to the calling function. Although a function can also allocate

memory for the pointer, e.g., `x = new double [5];`, the new memory address is NOT passed back to the calling function because the pointer itself was "passed by value"; any memory allocated locally in a function (to a local variable or a pointer argument) must be deleted/freed prior to returning from this function – otherwise the memory is "leaked" which can cause problems when the function is repeatedly called (thus building up the amount of memory leaked). If a function needs to locally allocate memory to a pointer variable but then make this memory available to the calling function, a "pointer-to-the-pointer" must be passed into the function, e.g., `double **y;`. Try using pointers with functions for yourself! Pointers offer a powerful capability but one that will cause compiler errors/warnings (if you're lucky) and segmentation faults (if you're not).

Built-in C++ Classes

A major convenience in C++ relative to C is the availability of built-in "classes" that are easy to use even if you don't augment, modify, or even completely understand the code in these classes. Unbeknownst to most students just learning to program, the very first C++ program one usually writes relies on the use of C++ classes or objects. Indeed, `cout` is actually a C++ object of type `ostream`. `ostream` is the "output stream" C++ class, and when one enters the statement `cout << "Hello world\n";` one is using the overloaded operator `<<` which is defined as a member function of class `ostream` to print text to the screen. Other classes referenced above are `istream` (input stream), the data type of objects such as `cin`, and the file handling classes `ifstream` and `ofstream`. Typically, a class is declared in a header (.h) file and implemented in one or more source (.cpp) file(s). This holds true for classes built into C++, as one `#include`'s header files such as `iostream` declaring the `istream` and `ostream` classes, and `fstream` declaring `ifstream` (input file stream) and `ofstream` (output file stream) classes. In fact, the `fstream` classes "inherit" data and functions from the corresponding "base" class in `iostream`; inheritance allows derived classes (e.g., `ifstream`) to reuse the data and functions from base class(es) (e.g., `istream`) improving code compactness, convenience, and function consistency.

Two additional built-in C++ classes very commonly used by engineers are the `vector` class and the `string` class. Both classes "hide" dynamic memory management details from the programmer/user for convenience. Descriptions and lists of of member functions available in each of these classes are available online: http://www.cplusplus.com/reference/vector/vector/ and http://www.cplusplus.com/reference/string/string/. The vector class is designed to generally support array dynamic memory management for any valid data type, e.g., `double, int, char,` the user-defined `student struct` defined above. To do this the `vector` class relies on the notion of a C++ template as specified in the standard template library (STL). By using STL, the user can specify a placeholder data type as part of the function declaration, implementation, or call, then substitute the actual C++ data type at runtime.

An example program for vector addition using the `vector` class is shown below. Note that the `vector` class doesn't make the code much simpler in this case so there is no real advantage to using the `vector` class over a pointer variable since either version allows the user to choose array size at runtime. In other cases, particularly where vector size needs to incrementally change, the `vector` class can simplify code. The cost is that the `vector` class also adds overhead thus execution time relative to direct use of arrays/pointers.

Numerous other built-in C++ classes are available; before you embark on manually writing your own class, unless of course asked to do so for a homework assignment, look around to see if a class to support your needs is already available.

```
#include <iostream>
#include <vector>
using namespace std;

int main()
{
  vector <double> x, y;
  int size;

  // Many use push_back() as a convenient way to add indidivual elements
  // to a vector.  This is OK for small vectors, but for large vectors this
  // strategy has the same problem as incrementally increasing the size of
  // vectors and matrices in Matlab.
  // In this application we can ask the user for a size
  // and "resize()" the vectors appropriately.

  cout << "Enter size:\n";
  cin >> size;
  x.resize(size);
  y.resize(size);

  cout << "Enter x:\n";
  for (int i=0;i<size;i++)
    cin >> x[i];

  cout << "Enter y:\n";
  for (int i=0;i<size;i++)
    cin >> y[i];

  cout << "Result:  ";
  for (int i=0;i<size;i++)
    cout << x[i] + y[i] << "  ";
  cout << endl;

  return 0;
}
```

User-defined Classes

While full mastery of object-oriented programming is beyond the scope of these notes, this chapter does at least introduce the concept of a user-defined class. Organization of code through use of C++ classes or "objects" is viewed as essential for large programs. Such organization improves code modularity, readability, and re-usability as a minimum. To define a class, a class declaration is created and typically stored in a header (.h) file of the same name as the class. This declaration describes the data and function "members" contained in the class. Short member function implementations may also be defined, but longer function definitions are typically split into one or more C++ source files (.cpp) to promote readability of the header file class declaration.

Data and function members of a class may be available or "hidden" from users, in which case they only support internal class operations. Three access designators are available: `public` (everyone can use), `protected` (everyone can read but not write), and `private` (only member functions have access). By default class member data is `private` and class member functions are `public`, allowing general use of functions but not general modification to class data. This is a desirable access configuration because it maximizes class functionality to the user but ensures the user won't inadvertently set data values improperly. A function that is not a member of the class can be declared as a `friend`, which offers this function access to private and protected data. Functions with the same name as the class are called constructors; a constructor is called each time a new class variable (or instance) is declared. Multiple versions of the constructor will typically be defined to offer the user flexibility in initializing data member values during the declaration statement. A destructor, designated by the class name preceded by a tilde (~), is called when the class instance goes out of scope. The destructor de-initializes a class instance before it goes out of scope and is critical to delete (free) memory dynamically allocated for each class instance. A group of accessor functions return private data member values to the user. Operator functions are common to classes that perform mathematical computations. An operator function is named "operator" followed by the operator symbol being defined (or overloaded), e.g., operator+() is a function defining how the + operator works for this class. Nontraditional operators such as << and >> for data output and input can also be defined. The following pages show two example codes I use in Engr151 lectures to illustrate the declaration, implementation, and use of C++ classes. The codes are structured as follows:

Example 1: A user-defined `Vector` class
Class declaration: `Vector.h`
Class implementation: `Vector.cpp`
Example `main()` function using the class: `vector_main.cpp`

Example 2: A user-defined `Vehicle` class
Class declaration and implementation: `Vehicle.h`
Example `main()` function using the class: `vehicle_main.cpp`

```
// Vector.h
//
#ifndef MY_VECTOR
#define MY_VECTOR

#include <iostream>
using namespace std;

class Vector
{
 private:

  int size;
  double *x;

 public:
  Vector(int=0);  // Default constructor (input is size)
  Vector(const Vector &); // Copy constructor
  ~Vector(); // Destructor

  // Math operator functions
  double dot(const Vector &);
  Vector cross(const Vector &);  // 3D only!!!!

  Vector operator=(const Vector &);  // Assignment operator (!)

  Vector operator+(const Vector &);
  Vector operator+(double);
  Vector operator-(const Vector &);
  Vector operator-(double);
  Vector operator*(double);
  double &operator[](int);
  double mag();
  bool operator==(const Vector &);

  // Read and print
  friend ostream & operator<<(ostream &, const Vector &);
  friend istream & operator>>(istream &, Vector &);
};


#endif
```

```
// Vector.cpp
//
#include "Vector.h"
#include <cmath>  // For magnitude function mag()

Vector::Vector(int i)
{
  size=i;
  if(size==0)
    {
      x=NULL;
      return;
    }
  x=new double[size];
  int k=0;
  while(k<size)
    {
      x[k]=0;
      k++;
    }
}


Vector::Vector(const Vector &v)
{
  size = v.size;
  if (v.x == NULL)
    x = NULL;
  else {
    x = new double[size];
    for (int i = 0; i < v.size; i++)
      x[i] = v.x[i]; // You can also use v[i]
  }
}

Vector::~Vector()
{
  if (x != NULL)  delete [] x;
  return;
}

double Vector::dot(const Vector &v)
{
  double ret=0;
  if (size==v.size)
    for (int i=0; i<size; i++)
      ret+=x[i]*v.x[i];
  return ret;
}

Vector Vector::cross(const Vector &v) // 3D only!!!!
{
  if (size==3 && size==v.size) {
    Vector ret(3);
    ret.x[0]=x[1]*v.x[2]-x[2]*v.x[1];
    ret.x[1]=x[2]*v.x[0]-x[0]*v.x[2];
```

```
    ret.x[2]=x[0]*v.x[1]-x[1]*v.x[0];
    return ret;
  }
  Vector ret(0);
  return ret;
}

Vector Vector::operator=(const Vector &v)
{
  if (size != v.size) {
    size = v.size;
    delete []x;
    x = new double[size];
  }

  for (int i = 0; i < size; i++)
    x[i] = v.x[i];

  return *this;
}

Vector Vector::operator+(const Vector &v)
{
  if(size>=v.size)
    {
      Vector ret(size);
      for(int i=0; i<v.size; i++)
      ret.x[i]=x[i]+v.x[i];
      for(int i=v.size; i<size; i++)
      ret.x[i]=x[i];
      return ret;
    }
  else
    {
      Vector ret(v.size);
      for(int i=0; i<size; i++)
      ret.x[i]=x[i]+v.x[i];
      for(int i=size; i<v.size; i++)
      ret.x[i]=v.x[i];
      return ret;
    }
}

Vector Vector::operator+(double blah)
{
  Vector ret(*this);
  for (int i=0; i<size; i++)
    ret.x[i] += blah;
  return ret;
}

Vector Vector::operator-(const Vector &v)
{
  if(size>=v.size)
    {
      Vector ret(size);
```

```
      for(int i=0; i<v.size; i++)
      ret.x[i]=x[i]-v.x[i];
      for(int i=v.size; i<size; i++)
      ret.x[i]=x[i];
      return ret;
    }
  else
    {
      Vector ret(v.size);
      for(int i=0; i<size; i++)
      ret.x[i]=x[i]-v.x[i];
      for(int i=size; i<v.size; i++)
      ret.x[i]= -v.x[i];
      return ret;
    }
}

Vector Vector::operator-(double blah)
{
  Vector ret(*this);
  for (int i=0; i<size; i++)
    ret.x[i] -= blah;
  return ret;
}

Vector Vector::operator*(double blah)
{
  Vector ret(*this);
  for (int i=0; i<size; i++)
    ret.x[i] *= blah;
  return ret;
}

// Needs to be global (in error condition branch of operator[])
// Set this to 0 instead of M_PI - less fun but probably more intuitive.
double vector_placeholder=0.0;

double & Vector::operator[](int d)
{
  if (d >= 0 && d < size)
    return x[d];
  else {
    cout << "operator[] index error blah blah blah!\n";
    if (size > 0) return x[0];  // Return first element if x has elements
    else   // Otherwise return a valid double & to prevent exception
condition.
      return vector_placeholder;
  }
}

double Vector::mag()
{
  double magblah=0.0;
  for (int i=0;i<size;i++) magblah += x[i]*x[i];
  return sqrt(magblah);
}
```

```
bool Vector::operator==(const Vector &v)
{
  bool r=1;
  if(size==v.size)
    for(int i=0;i<v.size;i++)
      if(x[i]!=v.x[i])  r=0;
  else    r=0;
  return r;
}

ostream & operator<<(ostream &os, const Vector &v)
{
  os << '[';
  for (int i=0;i<v.size;i++)
    os << v.x[i] << ", ";
  if (v.size>0) os << "\b\b";
  os << ']';
  return os;
}

// Additional code added to deal with dynamic memory after today's lecture
istream & operator>>(istream &is, Vector &v)
{
  // Delete old vector (simple - don't care if new size is the same)
  if (v.size) delete v.x;

  cout << "Enter size of vector:\n";
  is >> v.size;
  if (v.size <= 0) {
    cout << "Error - negative or zero size entered!\n";
    v.x=NULL;
  } else {
    v.x = new double [v.size];
    cout << "Enter vector elements:\n";
    for (int i=0;i<v.size;i++)
      is >> v.x[i];
  }
  return is;
 }
```

```
// vector_main.cpp:  main program to test the Vector class
//
#include "Vector.h"

int main()
{
  Vector a(2), b(3), c;

  // Test output operator
  cout << "a = " << a << endl;
  cout << "b = " << b << endl;
  cout << "c = " << c << endl;

  // Test input operator
  cin >> c;
  cout << "c = " << c << endl;

  // Now test remaining operators
  a = c + c;
  cout << "c + c = " << a << endl;

  b = a + 2;
  cout << "a + 2 = " << b << endl;

  c = b - c;
  cout << "b - c = " << c << endl;

  c = c - 3;
  cout << "c - 3 = " << c << endl;

  c = c * 1.5;
  cout << "c * 1.5 = " << c << endl;

  cout << "Magnitude of c = " << c.mag() << endl;
  cout << "a dot b = " << a.dot(b) << endl;
  cout << "a cross b = " << a.cross(b) << endl;

  if (a == b)
    cout << "equality test failed!\n";
  else
    cout << "equality test passed!\n";

  // Test index function
  cout << "c = " << c << endl;
  cout << " c[0] = " << c[0] << endl;
  cout << " c[1] = " << c[1] << endl;
  cout << " c[2] = " << c[2] << endl;
  cout << " c[100] = " << c[100] << endl;  // Out of bounds

  return 0;
}
```

```
// Vehicle.h
//
#ifndef MY_VEHICLE_H
#define MY_VEHICLE_H

#include <iostream>
using namespace std;

class Vehicle
{
 protected:
  string manufactName;
  int numCylinders;
  string owner;

 public:
  Vehicle(string name = "", int cyl = 0, string own = "") {
    manufactName = name;
    numCylinders = cyl;
    owner = own;
  }

  Vehicle(const Vehicle & v)  {
    manufactName = v.manufactName;
    numCylinders = v.numCylinders;
    owner = v.owner;
  }

  Vehicle operator=(const Vehicle & v)  {
    manufactName = v.manufactName;
    numCylinders = v.numCylinders;
    owner = v.owner;
    return *this;
  }

  string get_manufacturer() { return manufactName; }
  int get_cylinders() { return numCylinders; }
  string get_owner() { return owner; }

  void set_manufacturer(string & s) { manufactName = s; }
  void set_cylinders(int & i) { numCylinders = i; }
  void set_owner(string & s) { owner = s; }

  friend ostream & operator<<(ostream & os, Vehicle v)  {
    os << "Manufacturer: " << v.manufactName << endl;
    os << "Cylinders: " << v.numCylinders << endl;
    os << "Owner: " << v.owner << endl;
    return os;
  }

  friend istream & operator>>(istream & in, Vehicle & v)  {
    int c; string m = "" , o = "";
    getline(in, m);
    in >> c;
    getline(in, o);
    getline(in, o);
```

```
    v.manufactName = m;
    v.numCylinders = c;
    v.owner = o;
    return in;
  }
};

#endif
```

```
// vehicle_main.cpp:  Test program for Vehicle/Truck classes
//
#include <iostream>
using namespace std;
#include "Vehicle.h" // Not needed for our implementation
#include "Truck.h"

int main()
{
  Vehicle v1;
  Truck t1;
  string s;
  int i;
  double d;

  // Set properties of each class; print to screen
  s = "Ford";  v1.set_manufacturer(s);
  i = 4;  v1.set_cylinders(i);
  s = "Snoopy";  v1.set_owner(s);
  cout << "Vehicle Test #1:\n";
  cout << v1 << endl;

  s = "Toyota";  t1.set_manufacturer(s);
  i = 6; t1.set_cylinders(i);
  s = "Spike";  t1.set_owner(s);
  d = 1500.5; t1.set_load_capacity(d);
  i = 3333; t1.set_towing_capacity(i);
  cout << "Truck Test #1:\n";
  cout << t1 << endl;

  Vehicle v2(v1);  // Copy constructor tests
  Truck t2(t1);
  s = "Flying Ace"; v2.set_owner(s);
  s = "Land Rover of America"; t2.set_manufacturer(s);
  s = "Easter Beagle"; t2.set_owner(s);
  cout << "Vehicle Test #2:\n" << v2 << endl;
  cout << "Truck Test #2:\n" << t2 << endl;

  cout << "Test of get operators:\n";
  cout << "v1 owner = " << v1.get_owner() << endl;
  cout << "t2 owner = " << t2.get_owner() << endl;
  cout << "t1 cylinders = " << t1.get_cylinders() << endl;
  cout << "t2 manufacturer = " << t2.get_manufacturer() << endl;
  cout << "t1 towing capacity = "<< t1.get_towing_capacity() << endl;

  cout << "Finally, test >> operators.\n";
  cout << "First, enter a vehicle (manufacturer, cylinders, owner):\n";
  cin >> v1;
  cout << v1 << endl;

  cout << "Next, enter a truck (manuf, cyl, owner, load cap, towing cap).\n";
  cin >> t1;
  cout << t1 << endl;

  return 0;
}
```