CIS 265 – Core Concepts and Code Snippets

Christopher Burns – ckburns@cox.net

This is a digest of some of the concepts we've covered up to this point in the class. It is not as detailed as the lecture notes, but you may find this helpful as a quick reference when trying to tackle the homework, lab assignments, and midterm.

I would also suggest implementing some of the code snippets for concepts that you feel need a better understanding of.

If you would like to see something else added, or have any questions, let me know.

Functions – prototypes

When a compiler compiles your program, it read through it starting at the first line going to the end. If it encounters a word (token) that it does not understand it will end with an error. This is true with functions you've written.

If you have a call to a function that the compiler has not yet seen, it will tell you that it encountered something it could not identify. One solution is to be sure that the implementation of your function occurs earlier in your source file than where it is called. This is often inconvenient. A better solution is to create prototypes for your functions, and placing them near the beginning of your program ensures that the compiler will be able to recognize that function

A Function prototype is not the implementation of your function. Its only purpose is to add the name of your function, and the parameters it accepts, to the list of symbols it understands. Your function prototype must match your function implementation in both name, parameters, and return value.

example – function prototype

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

// prototype of add function
int add( int a, int b );

void main()
{
  int i, j, k;
  i = 1;
  j = 2;
  // compiler will recognize "add( int, int )"
  //  because it has already seen the function prototype
  k = add( i, j );
}

// implementation of function add
int add( int a, int b )
{
  return a + b;
}
```

Functions – overloading

It is possible to have multiple implementations of the same function name. Once useful example of this is with math functions.

example – function overloading

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

int   add( int a, int b );
int   add( int a, int b, int c );
float add( float a, float b );

void main()
{
  int i = 1, j = 2, k, l;
  float x = 1.2, float y = 2.5, float z;

  k = add( i, j );
  l = add( i, j, k );
  z = add( x, y );
}

int add( int a, int b )
{
  return a + b;
}

int add( int a, int b, int c )
{
  return a + b;
}

float add( float a, float b )
{
  return a + b;
}
```

Strings – basic usage

Strings in the C language are implemented as an array of characters. The final character of any string must be the NULL character. Basic string manipulation can occur by treating the string as an array, and manipulating its characters thusly. Don't forget that when doing this you'll often have to set the NULL termination yourself.

example – find length of a string

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

char aString[] = "This is a string";

void main()
{
  int i = 0;

  // find the null character to see how long the string is
  while ( aString[ i ] != '\0' )  // '\0' is NULL represented as
                                  //   a character
  {
    i++;
  }
  cout << "The string contains " << i << " characters" << endl;
}
```

example – reverse a string

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

const int MAX_STRING_LENGTH = 64;
char aString[] = "Reverse me";
char anotherString[ MAX_STRING_LENGTH ];

void main()
{
  int length = 0;
  // first find out how long the string is
  while ( aString[ length ] != '\0' )
  {
    length++;
  }
  // reverse the character of the string
  for ( int i = 0; i < length; i++ )
  {
    anotherString[ i ] = aString[ length - i - 1 ];
  }
  // null terminate reversed string
  anotherString[ length ] = '\0';
  cout << aString << " reversed is " << anotherString << endl;
}
```

Strings – using the standard library

The standard C library provides several string operation functions. Most of these seem cryptic and rudimentary at first. They provide the basic features that are needed for string parsing and can be used as building blocks to create more complex string operations. Once you've spent some time using these (see Deitel HTPC++ 4th ed page 363) you'll begin to get a good feeling of how they can be best applied.

example – copy text to a string, append to it, and get its length

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

const int MAX_STRING_LENGTH = 64;
char aString[ MAX_STRING_LENGTH ];

void main()
{

  strcpy( aString, "copy this into my string" );

  strcat( aString, ", then append this." );

  int length = strlen( aString );

  cout << "My string is " << aString << " and it is " <<
          length << " characters in length." << endl;
}
```

Pointers – basic

When your program is executed, it is loaded into the computer's memory. Variables you've created also exist in the computer's memory. The C language allows you to get the memory location of a variable, a.k.a. a pointer. This seems to be a complex feature at first, and its usefulness even questionable.

Pointers are actually one of the most powerful features of the C language. They allow you to pass to location of a variable to functions so that they can manipulate the actual value of the variable as well as allowing you to look at memory around the variable.

example – getting a pointer to a variable and using it

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

void main()
{
  int i = 12; // variable i has a value of 12

  int* ptrI;  // int* reads as pointer to integer

  ptrI = &i;  // &i returns the address where value of i lives
              //    in memory

  cout << "i     = " << i << endl;
  cout << "&i    = " << &i << endl;
  cout << "ptrI  = " << ptrI << endl;
  cout << "*ptrI = " << *ptrI << endl; // *ptrI returns the value
                                       //  the memory location
                                       //  contained in ptrI
}
```

```
i     = 12
&i    = 0012FED4
ptrI  = 0012FED4
*ptrI = 12
```

Pointers – with arrays

Arrays are a simple collection of the same data type. The elements of the array are stored in memory together and arranged ascending. This allows us to easily use a pointer to reference elements of an array. Pointer math, so called because you are typically changing what memory location you are pointing to by adding or subtracting from the memory address, can be used to change what element of an array is being refered to. This is not only similar to using the array index notation array[index] to change what element is being looked at. Manipulating an array with array notation or pointer math is equivalent.

example – the mid function

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

// The BASIC language has a useful built in string function called mid
// Here is a C version of it:
//  mid will take a copy "length" characters from "inString"
//   starting at "begin"
//  example, mid( "Universe", 2, 3, outString ), outString would = "ive"
char* mid( char* inString, int begin, int length, char* outString )
{
  strncpy( outString, inString + begin, length );
  // note the second parameter: "inString + begin"
  //  this is pointer math.
  //  this parameter is the address of the start of
  //  the inString plus begin

  outString[ length ] = '\0'; // NULL termination!

  return outString;
  // notice we are returning a pointer to a string
  //  note that this is a pointer to a string
  //  declared in main and passed to us
}

void main()
{
  int begin = 12;
  int length = 8;
  char aString[] = "Once upon a midnight dreary";
  char anotherString[256];

  cout << "input string = " << aString << endl;
  cout << "mid of input string with begin = " << begin <<
          " and length = " << length << endl;
  cout << mid( aString, begin, length, anotherString ) << endl;
}
```

References

Pointers are quite powerful. So powerful that both novice and seasoned C programmers can easily get themselves hurt if they're not careful. C++ added references to the C language. References have most of the power of pointers, but make it harder for you to get into trouble.

References can be thought of as immutable pointers. That is to say that they point to a location in memory, but you cannot change what location they point at. Unless you have a need to do some pointer math, you'll find references more intuitive to work with.

When using a reference, treat it as an alias to the variable it is referencing. You do not need to use special operators like `*` or `->` to perform actions on it. Although it is in essence a pointer, it can be manipulated the same way the variable it refers to can be.

example – references in code

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

void main()
{
  int i = 21;
  int& refI = i; // int& reads as reference to integer

  cout << "i    = " << i << endl;
  cout << "refI = " << refI << endl;
  // variable refI can be used just like i
  //  do not need to use * like with pointers

  refI++;
  // this also increments i, since refI "refers" to i
  cout << "i    = " << i << endl;
  cout << "refI = " << refI << endl;
}
```

example – references in function calls, "passing by reference"

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

void doubleIt( int );
void doubleItRef( int& );

void main()
{
  int i = 21;
  cout << "i    = " << i << endl;
  doubleIt( i ); // i is actually being passed as a value
  cout << "i    = " << i << endl;
  doubleItRef( i ); // i is actually being passed as a reference
  cout << "i    = " << i << endl;
}

void doubleIt( int number )
{
  number*=2;
}

void doubleItRef( int& number )
{
  number*=2;
}
```

Structs

There are numerous ways to create your own data types in C. One of these is a struct. A struct data type can contain a collection of variables. It is useful for organizing variables of a single purpose into a manageable collection.

example – declaring and using a struct

```
struct point3D
{
  float x;
  float y;
  float z;
};

void main()
{
  point3D point;

  point.x = 5.5f;
  point.y = 20.0f;
  point.z = 8.2f;
}
```

Classes

Classes are identical to structs, except that the member variables and member functions they contain are initially considered private. Private members can be used by any function that belongs to the class, but cannot be used by any code outside of the class. The key words `private` and `public` can be used to change this designation for groups of variables and functions.

Since your class will likely have internal data variables and functions that should not be used outside of your class, you declare these as private. Some of these private variables are data items that need to be validated, state information, worker functions that are only used by functions within your class.

If the state of these variables needs to be accessed outside your class, you can declare it as public. Object oriented design techniques prefer that instead of simply exposing the variable, you add accessor methods. These methods can get and set the value of the private member variable.

example – declaring and using a class

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

class point3D
{
private:
  float x;
  float y;
  float z;
public:
  void setPoint( float a, float b, float c )
  {
    x = a;
    y = b;
    z = c;
  }
  float scaler()
  {
    return sqrt( x*x + y*y + z*z );
  }
  // overloading the << operator, a hint of things to come
  //  this allows us to cout << the class
  friend ostream& operator << (ostream& os, const point3D& s)
  {
    return os << s.x << ", " << s.y << ", " << s.z;
  }
};

void main()
{
  point3D point1;
  point1.setPoint( 5.5f, 20.0f, 8.2f );
  cout << "point1 = " << point1 << endl;
  cout << "scaler of point vector = " << point1.scaler() << endl;
}
```

Classes – Encapsulation

Unless a member variable needs to be accessed outside a class, it should be private. Even if it needs to be accessed outside the class it should be private and accessible only through the class's methods. This provides a level of data protection that can allow member variables to act as read-only or to be validated when assigned a value. This is called encapsulation.

Typically public get/set methods are created to access private member variables. The usual naming convention is "get" or "set" followed by the name of the member variable. If the member variable is a boolean, the prefix "is" is usually used instead of "get"; this makes the code more readable.

Set methods can validate the input data before setting the private member. If the member variable is calculated, or only set at object construction, it can be made read only by not implementing a set accessor. These both help to provide data protection.

It is usually a good idea to have Get/set accessor methods, but it does add a some overhead. If a variable of a class needs to be accessed a large number of times over a small amount of time, performance constraints may dictate that the variable be made public and the accessor bypassed.

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

class MyClass
{
private:
  int value;
  char name[25];
public:
  int getValue()
  {
    return value;
  }
  void setValue( int input )
  {
    if ( input > 0 ) // validate input
      value = input;
  }
  const char* getName() // always return pointers as constants
  {
    return name;
  }
  void setName( char* input )
  {
    int i = strlen( input );
    if ( i > 0 && i < 25 )
      strcpy( name, input );
  }
};
```

Classes – Using pointers

Classes exist in memory like other variables, and thus you can have a pointer to that location. Getting the address of an object works the same way as for a regular variable. Accessing members of the object requires using the -> operator instead of the . operator (see example.)

Classes can sometimes be rather large, so C++ provides operators to allocate memory for them, and also to free memory when done. These are the new and delete operators respectively. When a new instance of a class is created, a pointer is returned.

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

class MyClass
{
private:
  int value;
public:
  int getValue() { return value; }
  void setValue( int input ) { value = input; }
};

void main()
{
  MyClass localClass;
  localClass.setValue( 21 );

  MyClass* ptrToLocal = &localClass;
  cout << "Value is " << ptrToLocal->getValue() << endl;

  MyClass* ptrToNewClass = new MyClass;
  ptrToNewClass->setValue( 33 );
  delete ptrToNewClass;
}
```

Rules of thumb for pointers to classes:

- Pass objects to functions as pointers.
- Create the object locally (on the stack) if it is small (since the stack has limited space), if it is only needed in one function. Advantage of creating it locally is that memory access to the object will be faster.
- Create the object using new (in the heap) if it is large, if it needs to be accessed by multiple functions, needs to have a long lifespan.

Classes – Constructors and destructors

When an instance of a class is created, a special function in the class is called automatically. This function is called the constructor and has the same name as the class and no return type. Constructors provide a convenient place to initialize variables within your class as well as any basic startup task. Destructors are called when a class is removed from memory either by leaving scope or through the delete operation. A class' destructor has the same name as the class preceded by a ~ and has no return type.

Constructors can be overloaded so that they can be passed parameters that might be used to initialize its variables. There are two special constructor types, the default constructor and the copy constructor. Even if you do not implement these two, they are created automatically by the compiler. You can implement them to add your own special operations.

Default Constructor

This is the basic constructor with no parameters. It is called when a new object is created, but there are no parameters or assignments.

Copy Constructor

When a new variable is created from an existing object, this constructor is called. The default copy constructor performs a shallow copy of the existing object; that is it does a straight member variable to member variable assignment. Generally you do not need to create a special copy constructor unless a deep copy is required; there are member variables in your class that require special copying methods, like arrays.

Overloaded Constructors

It's often nice to pass values to your constructor so it can use them to initialize its member variables.

```
class MyClass
{
private:
  int value;
public:
  MyClass() // default constructor
  {
    value = 0;
  }
  MyClass( const MyClass& copyMe ) // copy constructor
  {
    value = copyMe.value;
  }
  MyClass( int inputValue ) // overloaded constructor
  {
    value = inputValue;
  }
  int getValue() { return value; }
  void setValue( int input ) { value = input; }
};
```

```
void main()
{
  MyClass class1; // default constructor called
  MyClass class2 = class1; // copy constructor called
  MyClass class3( class2 ); //copy constructor called
  MyClass class4( 2 ); // overloaded constructor called
}
```

Classes – The this pointer

Sometimes your object needs to refer to itself. It can do this by using a special token called the "this" pointer. Here are three examples of what it can be used for:

Avoiding name collisions

Your member functions that refer to member variables can be made easier to read by using the this pointer refer to member variables. This allows get/set functions to use more logical parameter variable names, as well as making the usage of member variables more visibly obvious.

```
class MyClass
{
private:
  int value;
public:
  MyClass& setValue( int );
};

void MyClass::setValue( int value )
{
  this->value = value;
}
```

Cascading method calls

By returning a reference to yourself from a function, that reference can be used to call another method of your object. This allows methods calls to cascade, or build on each other.

```
class MyClass
{
private:
  int value;
public:
  int getValue();
  MyClass& setValue( int );
};

MyClass& MyClass::setValue( int value )
{
  this->value = value;
  return *this;
}

int MyClass::getValue()
{
  return this->value;
}
```

```
void main()
{
  MyClass class1;
  cout << class1.setValue( 2 ).getValue();
}
```

Passing a reference to yourself so others can refer back to you

Sometimes you need to let someone know who you are so they can call you back, other times you might want to add yourself to a collection so you can be found again.

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

class MyClass
{
private:
  int value;
public:
  MyClass();
  int getValue();
  MyClass& setValue( int );
};

MyClass& MyClass::setValue( int value )
{
  this->value = value;
  return *this;
}

int MyClass::getValue()
{
  return this->value;
}

vector< MyClass* > myClasses;

MyClass::MyClass()
{
  this->value = (int)myClasses.size();
  myClasses.push_back( this );
}

void main()
{
  MyClass class1, class2, class3;
  for( int i = 0; i < (int)myClasses.size(); i++ )
  {
    cout << i << "'s value " << myClasses[i]->getValue();
  }
}
```

Classes – Operator Overloading

Operators are language tokens that you have used in C/C++ to perform a variety of tasks. Some of these include arithmetic operations (+, -, /, *), comparison operations (= =. !=, <, >…), array operations ([ ]), and pointer operations (*, &, new, delete); a complete list can be found in any C++ text book.

Just as it is convenient to add two numbers using operators, x = y + z, it is also convenient to perform functions with objects using operators. A common example of operator overloading is a string class, appending string1 to string2 can be written as string1+=string2. This works if the string class has overloaded the += operator

Hint: when overloading operators, think of the operators as functions… also remember that operators often cascade so they may need to return references. If you think about how the operator has to behave, you'll see why temp holders are needed in some of the operator implementations (e.g. postfix ++.)

Overloaded operators can be implemented as member, or non-member functions. Generally, they are only implemented as non-members if their return type is different than the class they are implemented for. This is the case with the stream operators as they return i/ostreams.

Common operator overloads

Operator	Type	Example

=	member	`MyClass& operator= ( const MyClass setMe )`
		`{`
		`  value = setMe;`
		`  return *this;`
		`}`

+	member	`MyClass operator+ ( const MyClass addMe )`
		`{`
		`  MyClass temp;`
		`  temp.value = this->value + addMe.value;`
		`  return temp;`
		`}`

+=	member	`MyClass& operator+=( const MyClass addMe )`
		`{`
		`  value = value + addMe.value;`
		`  return *this;`
		`}`

==	member	`bool operator==( const MyClass checkMe )`
		`{`
		`  if ( this->value == checkMe.value )`
		`  return true;`
		`  else`
		`  return false;`
		`}`

>>	non-member	`friend ostream& operator<< ( ostream& out,`
		`  MyClass printMe )`
		`{`
		`  out << "MyClass value " << printMe.value;`
		`  return out;`
		`}`

<<	non-member	`friend istream& operator>> ( istream& in,` `  MyClass& getMe )`

		{
		  string temp;
		  in >> temp;
		  getMe.value = temp;
		  return in;
		}

[ ]	member	int operator[](int index)
		{
		  return value[index];
		}

++
(prefix)
	member	MyClass& operator++()
		{
		  this->value++;
		  return *this;
		}

++
(postfix)
	member	MyClass operator++(int)
		{
		  MyClass temp = *this;
		  this->value++;
		  return temp;
		}

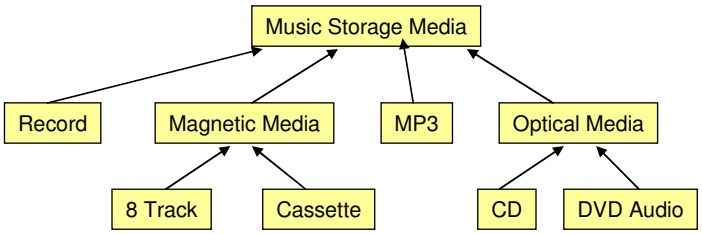
Classes – Inheritance

Inheritance is a very large topic, and this section just shows some of the basic mechanics. Revisit the class slides for more detailed examples.

Inheritance allows classes to "inherit" the methods and interface of a parent class. Establishing a logical parent/child relationship is important when creating an inheritance hierarchy. The child, or sub class, has an "is-a" relationship with the parent or base class: a Cassette Tape "is-a" Music Storage Media. Member variables of a class usually have a "has-a" relationship with their class: Cassette Tape "has-a" Recording Length.

Like class variables, a class that is being inherited can be declared public, protected, or private. Public inheritance follows the "is-a" relationship. Private and protected generally are "has-a" relationships, and can be used to implement aggregation. We will talk about public inheritance here.

When creating classes, groups of similar items can be abstracted to common elements. This abstracted class is called a base class. As an example, suppose you want to write a set of classes to help you organize your music collection. You should first draw out what your classes might be and how they inter-relate:



The "Music Storage Media" is the base class which would contain functions that are common to all of its type. These might include media type, album title, song list, location, and date. All classes that inherit from this base class would also have these functions.

Child classes are not limited to what was defined in the parent class. They can override methods of the parent and add methods of their own. These child classes can act as base classes for other classes. The "Magnetic Media" class would override the media type method of "Music Storage Media" and add an additional method for "Tape Type".

Base classes are often abstract. This means that although they define methods that children will use, they do not necessarily have to implement them. This is done through the use of virtual functions.

```
#include <string>
using namespace std;

class MusicStorageMedia
{
private:
  string title;
  string artist;
public:
  // note how the constructor is setting the default values
  //  This is called an initialization list.
  MusicStorageMedia():title("undefined"),artist("undefined") {}

  // getMediaType is a pure virtual function as it is
  //  set equal to 0
  virtual string getMediaType() = 0;

  // The more accessors we can get into the base class
  //  the less we have to define in the sub classes!
  string getAlbumTitle() { return title; }
  void setAlbumTitle( string title ) { this->title = title; }
  string getArtistName() { return artist; }
  void setArtistName( string artist ) { this->artist = artist; }
};

class Record : public MusicStorageMedia
{
public:
  // enum types are a collection of constants.
  //  Defining them this way helps for type checking
  //  and keeps things simpler
  enum RecordSpeed { Unknown = 0, Record33, Record45, Record78 };
  string getMediaType() { return "Record"; }
  RecordSpeed getSpeed() { return speed; }
  Record() { speed = Unknown; }
private:
  RecordSpeed speed;
};

class MagneticMedia : public MusicStorageMedia
{
  // implementation added here
};

class CassetteTape : public MagneticMedia
{
   string getMediaType() { return "Cassette"; }
  // implementation added here
};
```

Classes – Polymorphism

The before example showed how inheritance can help in logic structuring of your data types and code reuse, polymorphism is what makes using inheritance convenient.

The class "CassetteTape" is-a "MagneticMedia" is-a "MusicStorageMedia", and class "Record" is-a "MusicStorageMedia". Thus, they are siblings with a root base class. They can be referred to by their base class type. Thus, a vector or array of "MusicStorageMedia" can contain "Cassette", "Record", and any other class that is-a "MusicStorageMedia".

```
vector< MusicStorageMedia* > myMusic;

CassetteTape* myCassette = new CassetteTape();
Record* myRecord = new Record();

myMusic.push_back( myCassette );
myMusic.push_back( myRecord );
```

When referring to the objects in the "myMusic" vector, you will only be able to see them as "MusicStorageMedia". They will both support the "getMediaType" function, and use their individual implementation, but you will not be able to access the Record's "getSpeed" function.

```
cout << myMusic[0]->getMediaType() << endl;
cout << myMusic[1]->getMediaType() << endl;
//cout << myMusic[1]->getSpeed() << endl; // can't do this
```

Most C++ compilers have an option to support runtime type information, or RTTI. Using this we can determine what type of object has been polymorphed, and thus access its specific capabilities. The typeid operator returns a type_info object that can be used to compare an object to a class type.

Using a special type of casting, "dynamic_cast" we can reinterpret a base type as a child type. This will only work if the object really is the child type.

```
if ( typeid( *myMusic[1] ) == typeid( Record ) )
{
  cout << dynamic_cast< Record* >( myMusic[1] )->getSpeed();
}

// This would be bad!
//cout << dynamic_cast< Record* >( myMusic[0] )->getSpeed();
```