**CS311 More on C++ (Reference)**

**C++ Files**

* .h files include class and function declarations/prototypes
* .cpp files include definitions of member functions
* .cpp files typically #include the .h files they use
* .h file must be included where its declarations are used
* each .cpp file is compiled to object module
* object modules are linked together to form a program
* On Empress, .C is the same as .cpp
* On Empress, the compiler is g++. g++ <list all .cpp files here>
* On Empress, the executable is a.out

**C++ Features You Might Not Have Used Before**

**The Floating Point Types**

* corresponds to floating point real numbers
* three kinds of floating point types:
  + float (usually 4 bytes)
  + **double (usually 8 bytes)**
  + **long double (usually 8 bytes)**
* example literals   
  1.0  
  -3.000001e-10  
  30.01E40

**The Character Type**

* represents an ASCII character code
* requires one byte
* example **special literals**   
  '\0' null character   
  '\n' newline (or linefeed)   
  '\r' return   
  '\t' tab

**The Character String Type**

* represents a sequence of characters
* usually declared as a char\*
* example string literals   
  char\* s = "Hello world!";  
  cout << "This is another character string.\n";  
  cout << s << endl; // prints: Hello World!

**New Type via Enumerations**

* a shorthand for defining a list of constants

enum Day {MON, TUE, WED, THU, FRI, SAT, SUN};  
// Day is a new type with values MON, ..., SUN   
Day d = MON; // can use MON as a value

* when you display the value of MON, it is 0 and SUN is 6.

**Type Alias via Typedef**

* Use typedef to create an alias for types

typedef el\_t int; // el\_t is used to mean int

* allows you to name a new type   
  typedef int\* IntPointer; // IntPointer is now a synonym for int\*  
  typedef char Buffer[20]; // Buffer is a synonym for char[20]  
  Buffer buf; /// buf is an array of 20 char

**Class Definition Basics**

class Circle  
{  
private: // the data members  
 int radius;  
 int centerX;  
 int centerY;  
 **const float PI = 3.14159;**public: // the Interface – member function prototypes  
 Circle(**int newX, int newY, int newRadius**)  
 {  
 centerX = newX;  
 centerY = newY;  
 radius = newRadius;  
 }  
};

**Two things to note in this example:**

1. const float PI says PI is an alias (never changes its value) for 3.14159.
2. The constructor Circle takes 3 arguments used to initialize the data members. An example to create a Circle object: Circle myC(0, 0, 1);

**Other features you can use:**

* the constructor's **'init list'** allows construction of data members   
  Circle(int newX, int newY, int newRadius**): centerX(newX), centerY(newY), radius(newRaidus)** {}
* formals may have **default values** and if an actual parameter is omitted, formal takes on the default value   
   Circle(int newX = 0, int newY = 0, int newRadius = 1**): centerX(newX), centerY(newY), radius(newRaidus)** {}

**You can pass a stream to a function**

void Complex::print( **ostream& out** )// output stream is being passed  
 {  
 out << "(";  
 out << re; // re of this object  
 out << "+";  
 out << im; // im of this object  
 out << "i)";  
 }

* C++ also allows us to **define a << operator**
* operator << cannot be a member function; must pass the object to it.

**ostream&** operator<< ( **ostream& out**, Complex c )  
{  
 c.print( out );  
 **return out;** // stream but be returned}  
  
**The  *main* function exit status**

#include <iostream> // allows use of >> and << for I/O  
using namespace std; // not having to say std:: in front of cin int main() // ***int* is the  *exit status* for main**{  
 cout << "Hello Everyone!\n";  
 **return 0; // 0 means program terminated  *ok; 1 means failure***}

**The class string #include<string.h>**

* may use array indexing or use iterators to traverse
* may use functions such as strcpy, strlen, strcmp  
  int main()  
  {  
   **string s** = "hello";  
   cin >> s;  
   cout << s;  
   for ( int i = 0; i < **s.size**(); ++i )  
   cout << s[i];  
   for ( **string::iterator** i = **s.begin**(); i != **s.end**(); ++i )  
   cout << **\*i**;  
  }

**Assertions**

* allows statement of assumptions
* if the assertion is false, the program aborts with an error message
* should be able to delete them without affecting the program execution   
  #include <assert.h>  
  class Coins  
  {  
   Coins(int q, int n, int d, int p)  
   {  
   **assert(q >= 0 && n >= 0 && d >= 0 && p >= 0);** }

**Use of Const**

* const parameter allows you to protect a parameter from accidental modification   
  void print( const int j )  
  {  
   for ( int i = j; i >= 0; --j ) /// error is caught!  
   cout << i;  
  }  
  const with & is common for using pass by reference to improve efficiency  
  void print( const Coins& c )   
  {  
   cout << c.total() << endl;  
  }
* const member functions promise not to change the object

void inspect() const; // inspect will not change \*this

* Fred const\* p means "p points to a **constant Fred**": the Fred object can't be changed [via p](http://www.parashift.com/c++-faq-lite/ptr-to-const-aliasing.html).
* Fred\* const p means "p is **a const pointer** to a Fred": you can't change the pointer p, but you can change the Fred object [via p](http://www.parashift.com/c++-faq-lite/ptr-to-const-aliasing.html).

**Use of Static**

* Static duration means that the object or variable is allocated when the program starts and is de-allocated when the program ends. (it is like “global”)

static const float PI = 3.14159;

**Friend functions**

* Friend functions are not members of the class but can access its private members. Friend functions are not called with the receiver object. Objects must be passed as parameters.

friend Complex operator+(const Complex&, const Complex&);

* Useful when both arguments to the operator need to be converted into the object type.

Complex Result = 1 + 2; // converts 1 and 2 into complex numbers

**Standard C++ Exceptions**

* bad\_alloc is thrown when global operator **new** fails
* bad\_cast is thrown when dynamic cast type doesn't match
* bad\_typeid is thrown when typeid is called on null pointer
* bad\_exception is thrown if thrown exception isn't in throw spec
* ios::failure is thrown on I/O error

**What are Reference Variables in C++?**

If you have variable A, it comes with its location. E.g. A; // location of A is 1080.

***Reference variables will share the same location but with a different name.***

Reference variables must be initialized in its declaration and the reference cannot be changed.

**e.g. int &B = A; // B and A share the same location**

**A = 3;**

**B = 5; // also changes A to be 5**

You can place & in front of any variable to denote its location instead of its value.

**e.g. &A** means the location of A

**Are pointers different from reference variables?**

**Yes.** Pointers are basically variables that contain addresses (i.e. locations). The addresses can change allowing the pointer to point to different places. In fact, reference variables are implemented as constant pointers in C++ (i.e. pointers that point to the same location all the time).

Pointers and & can be used together. In

**e.g. int \*P; // pointer P is declared**

**int A; // integer variable is declared**

**P = &A; // makes P point to the location of A**

**\*P = 6; // will also change A’s value to be 6**

***Therefore, a pointer does not necessarily mean a pointer to the heap memory. You can point to anything.***

**Is there a pointer to a static array?**

**Yes.** Arrays in C++ are unique in the sense that the ***array name can be treated as a pointer to the location of the array*** even if it is a static array allocated on the run-time stack.

**e.g.**

**int m[] = {4,2,3}; // static array m**

**cout << \*m; // displays 4 because \*m refers to the first element of m**

**cout << \*(m+2) ; // displays 3 the 3rd element of m**

**Is it different from a dynamically allocated array?**

**Yes.** You can create dynamically allocated arrays.

**e.g.**

**m = new int[n]; // where the value of n is decided at run time and m is created in the heap**

You can still use the regular array notation m[i] to access each slot of the array, but you have to de-allocate the array explicitly by saying **delete [] m;**

**What is Pass by Reference?**

Reference variables are used to pass parameters to a function by reference.

**void f(int& A) // A will share the location with the function’s actual argument**

**{ A = 4; }**

**f(B); // by calling f, B will change to 4**

This achieves the same result as the following:

**void f(int\* P) // a pointer/location is received**

**{ \*P = 4; }**

**f(&B); // passing the location of B so B will change to 4**

Don’t forget that arrays are automatically passed by reference.

**Can you return a reference?**

**Yes.**  You can return the location instead of the value by using & in the return type.

**e.g.**

**int& f(int X[], int i) // array location is received because arrays are passed by reference**

**{**

**return X[i]; } // returns the location of X[i] (you cannot return the location of a static local variable)**

**int m[] = {1,2,3};**

**f(m,1)= 6; // the location of m[1] gets 6**

**cout << m[1] ; // 6 will be displayed**

Note that using f(m,1) on the left hand side of “=” treats it as a location but in other contexts, the value in the location will be used.

**e.g. cout << f(m,1); // displays 2**

**n = f(m,1); // assigns 2 to n**

Note that returning a reference can be accomplished by using a pointer.

**int\* f(int X[], int i) // array location is received**

**{**

**return &X[i]; } // returns the location of X[i]**

**int m[] = {1,2,3};**

**\*f(m,1)= 6; // the location of m[1] gets 6**

**cout << m[1] ; // 6 will be displayed**

But in this case, cout << f(m,1) will display an address.

**Are there pointers to functions:**

**Yes.** Given function Fun, Fun names the function but it is really a pointer to the body of the function.

Fun body of Fun

Therefore, \*Fun is the body of the function.

This allows you to pass functions as arguments to other functions!

**e.g.**

**float g( float (\*f)(float) ) // you can pass any function that takes a float argument and return a float type**

**{ return f(3.3); } // it will call that function with 3.3**

**cout<< g(doubleit); // calls g with the function named doubleit**

**cout << g(tripleit); // calls g with the function named tripleit**

**Can you create pointers to objects?**

**Yes.** You can create a pointer to an object.

**e.g.**

**girl \*P; // P can point to girl objects**

**girl Mary; // Mary is a girl object**

**P = &Mary; // P points to Mary (or you can do \*P = Mary)**

To access parts of Mary, you can dereference the pointer using P-> or (\*P).

**e.g. P ->age or (\*P).age**

**this** is a special pointer to reference the receiver object as you write member functions.

**Example to put it all together:**

**slist& slist::operator=(const slist& OtherOne) // OtherOne is received by reference**

**{**

**if (&OtherOne != this) // not the same because OtherOne and this point to different places**

**{**

**// this’s object has to be emptied first.**

**while (! this->isEmpty() )**

**this->deleteRear();**

**// this’s object has to be built up by allocating new cells**

**Node\* P; // local pointer for OtherOne’s nodes**

**P = OtherOne.Front; // P points to a node**

**while (P != NULL)**

**{**

**this->addRear(P->elem); //P’s element is added to this’s object**

**P = P->Next; // Go to the next node in OtherOne**

**}**

**}// end of if**

**return \*this; // return the resulting object (i.e. what this points to)**

**}**

**But slist& as the return type will return it as a reference to the object instead of returning a copy of the object.**

**Are object pointers related to inheritance?**

**Yes.**  A pointer for the parent class can point to any sub-class object.

Assume that person is the parent class and girl is the derived class.

**e.g.**

**person \*P; // P can point to person objects**

**girl Mary;**

**P = &Mary; // is allowed because girl is a sub-class of person**

**girl \*Q; // Q can point to girl objects**

**person Jones;**

**Q = &Jones; // is not allowed**

**Now that we have looked at pointers, we can talk about virtual functions and polymorphism.**

**What is Polymorphism?**

**Polymorphism in C++ means the same function or operation name denotes many different functions or operations of different classes.**

* static polymorphism - via overloading of operators so that the same operator can be used for any object.

In the context of choosing the function to call:

* Static binding means choosing the function at compilation time (as in the example we saw aboven\*\*).
* Dynamic binding means choosing the function at run time.

Dynamic binding occurs when you use **virtual member functions** **along with pointers to objects.**

In front of the function prototype, you add the word virtual. **E.g. virtual void f();**

If girl inherits **virtual function f** from person,

but also has its own function f, **there are two F's available for girl**. What happens?

Then girl's objects will use the local definition, and

person objects will use the base definition even though f is virtual.

**e.g.**

**girl Mary;**

**person Jones;**

**Mary.f(); // uses f of girl**

**Jones.f(); // uses f of person**

But now let’s create a pointer P. What P points to will determine which f to use:

**e.g.**

**person\* P; // P can point to persons**

**P = &Jones;**

**P->f(); // uses f of person**

**P = &Mary; // OK because girl is a sub-class of person**

**P->f(); // uses f of girl because P points to a girl and f is virtual**

Is this useful? Yes. If you have an array of pointers to persons and some slots point to girls and some slots point to boys, you can call f on each slot of the array without worrying about whether it is girl or boy. An appropriate one will be used.

**What are pure virtual functions?**

C++ allows you to create a special kind of virtual function called a **pure virtual function** (or **abstract function**) that has no body at all. A pure virtual function simply acts as ***a placeholder*** that is meant to be defined by derived classes. A pure virtual function is useful when we have a function that we want to put in the base class, but only the derived classes know what it should do.

To create a pure virtual function, rather than define a body for the function, we assign the function the value 0.

**e.g.    virtual int f() = 0; // a pure virtual function**

**What are abstract classes?**

Any class with one or more pure virtual functions becomes an **abstract base class**, which means that it cannot be instantiated. Until the pure virtual function is defined in the derived class, the derived class cannot be instantiated either to create its objects.

**e.g.**

**In the Animal Class**

**virtual const char\* Speak() = 0; // pure virtual function**

**---- you cannot create an Animal object**

**In the Dog Class that is derived from the Animal class**

**virtual const char\* Speak() { return "Bow wow"; } // defines Speak**

**--- you can create a Dog object**

**What are interface classes?**

An **interface class** is an **abstract** class that has **no members variables**, and where **all of the functions are pure virtual**. Interfaces are useful when you want to simply define the functionality that derived classes must implement but you never expect to have an object of that class.

Interface classes are often named beginning with an **I**.

Any class inheriting from an interface class must provide implementations for all functions in order to be instantiated.

Interface classes have become extremely popular in many languages because they are easy to use, easy to extend, and easy to maintain.

**Background: What Are Other Ways In Which Classes Are Related?**

**As we cover design patterns, you will see different ways in which classes are related.**

**Composition**

Compositions are complex classes that contain other classes as member variables (sub-part relationships).

When a composition is destroyed, all of the sub-objects are destroyed as well.

* If you destroy a car, its engine, and other parts should be destroyed as well.
* If you destroy a PC, you would its RAM and CPU should be destroyed as well.
* Typically use normal member variables
* Can use pointer values if the composition class automatically handles allocation/deallocation
* Responsible for creation/destruction of subclasses

**Aggregation**

An **aggregation** is a specific type of composition where no ownership between the complex object and the sub-objects is implied. When an aggregate is destroyed, the sub-objects are not destroyed.

Member variables are typically either references or pointers that are used to ***point at objects that have been created outside the scope of the class.***

* Typically use pointer variables **that point to objects** that lives outside the scope of the aggregate class
* Can use reference variables that point to objects that lives outside the scope of the aggregate class
* Not responsible for creating/destroying subclasses

**Friend classes**

Similar to friend functions, a friend class is a class whose members have access to the private or protected members of another class.

**e.g.**

**class Square {**

**friend class Rectangle; // rectangle is a friend of square**

**private: int side; ..}**

**In class Rectangle, the following member function can operate on a square object**

**void Rectangle::convert (Square a)**

**{width = a.side;**

**height = a.side;}**

Rectangle is considered a friend class by Square, but Square is not considered a friend by Rectangle. Therefore, the member functions of Rectangle can access the protected and private members of Square but not the other way around. Another property of friendships is that they are not transitive: The friend of a friend is not considered a

***As you find other aspects of C++ that you have never seen before, look them up on the internet or in C++ books immediately.***

***Features to eliminate from your program:***

* ***Duplicated code (the same code is used in multiple places)***
* ***Dependency on a particular data type***
* ***Any feature that makes it difficult to reuse your code***