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.

● `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.

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;**

<u>What is Pass by Reference?</u>

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.

<u>Can you return a reference?</u>

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.

<u>Are there pointers to functions:</u>

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

$$\begin{tikzcd}\text{Fun} \arrow[r] & \text{body of Fun}\end{tikzcd}$$

Therefore, *Fun is the body of the function.

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

e.g.

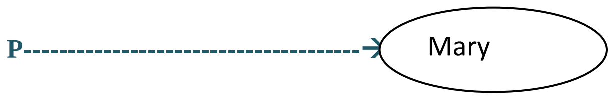
```
float g(    float (*f)(float)   )     // you can pass any function that takes a float argument and return a float type
{  return f(3.3); }    // it will call that function with 3.3

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

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

<u>Can you create pointers to static objects?</u>

Yes. You can create a pointer to an object. Recall that P = <right side> meant <right side> had to be another pointer or &<variable>.

e.g.

```
girl *P;            // P can point to girl objects
girl Mary;          // Mary is a girl object (not in the heap)
P = &Mary;          // P points to Mary; note that you cannot do P = Mary
```



Mary is not in the heap.

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

Note that *P = Mary; would cause a segmentation fault. Why?

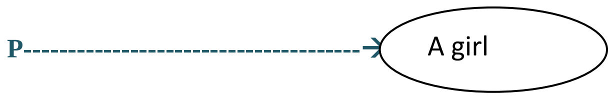
<u>this</u> is a special pointer to the receiver object that you can use as you write member functions.

<u>Can you create pointers to dynamic objects?</u>

Yes. You can create a pointer to a dynamic object in the heap.

e.g.

```
girl *P;              // P can point to girl objects
P = new girl;        // P points to a new girl object in the heap.
```



in the heap.

To access parts of the girl, you can dereference the pointer using P-> or (*P).
e.g. P ->age   or (*P).age

Virtual Functions and Polymorphism

Virtual functions promote polymorphism when used with pointers to objects.

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 *G;  // G can point to girl objects

person Jones;

G = &Jones; // is not allowed.  Why?
```

What is Polymorphism?

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

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

But we want to talk about polymorphism in relation to inheritance.

Polymorphism and Inheritance

In the context of choosing the function to call in an inheritance hierarchy:

- **Static choice** means choosing the function at compilation time (as in the example we saw above).

  e.g. with no pointers or references
  ```
  girl      Mary;
  person   Jones;
  Mary.F();          // uses F of girl
  Jones.F();          // uses F of person
  ```

- What happens if a pointer is created to an object? Still static.

  e.g. with &
  ```
  person *P;          // P can point to person objects
  girl Mary;          // girl object
  P = &Mary;          // is allowed because girl is a sub-class of person
  ```

```
    P->F();             // calls F of person because P is a person pointer

e.g. with new
    person *P;                    // P can point to person objects
    girl* Mary = new girl;       // a new girl object in the heap
    P = Mary;                    // is allowed because girl is a sub-class of person
    P->F();                      // calls F of person because P is a person pointer
```

What is a virtual function?

In front of the function prototype of a parent class, 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 now?

Let's create a pointer P. **What P points to will determine which f to use**: **(see VirtualFunction)**

Dynamic choice occurs when you use virtual functions along with pointers to objects.

```
e.g. with &
person* P;       // P can point to persons
P = &Jones;      // P points to person Jones
P->f();           // uses f of person
girl Mary;
P = &Mary;       // P points to 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

e.g. with new
person* P = Jones;        // P can point to a new persons object in the heap
P =  Jones;               // P points to person Jones
P->f();                   // uses f of person
P = new girl;             // P points to a new girl; 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, in two ways!

- 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. **(See VirtualFuncWithArrays)**
- If you have **a client function that is passed a pointer to a person** object, the function can be written in terms of the person pointer which could point to a girl or a boy. The function is re-usable.
  ```
  void G(person* P)         // P can point to persons
  {  P->f();  // calls appropriate f based on what P points to  }
  ```
  (see VirtualFuncion – client3)

This leads to the next topic below.

Abstract and Interface Classes

Abstract (and Interface) classes are found in many design patterns to promote maintainability and reusability.

Abstract (and Interface) classes provide the common interface for all sub-classes to use.

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 defining a body for the function, we assign the function the value 0 when you create a prototype in a header file.

e.g. virtual int f() = 0; // a pure virtual function in the base
// it should be defined in a derived class

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. (I put "pure" in quotes because in some later examples, you will see default implementation of virtual functions although the class is used only for the purpose of defining a common interface.)

e.g.

In the Animal Class - to provide a common interface

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

You cannot create an Animal object; you can create Animal pointers

In the Dog Class that is derived from the Animal class

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

What are interface classes? (See InterfaceClass)

An interface class is an abstract class that has no data members, 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.

What is #included where?

Let's say the class hierarchy is Person (Interface class) with Girl and Boy (Concrete Sub-classes).

Girl header #includes "person.h"
Boy header #includes "person.h"

The client wants to create pointers to Girl and Boy objects (the concrete sub-classes). Person object cannot be created.

```
#include "girl.h"
#include "boy.h"
person* G = new girl;
person* B = new boy;
```

The client is then written in terms of Person functions.
g++ *.cpp   to compile all the cpp files.

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***