Data Structures Exam 1
Dynamic Memory Allocation:
new and delete keywords

```
int *pointer = new int;
int * arr = new int[size];
delete pointer;
delete [] arr;
```

Pass by Alias (Reference)

```
void foo(int & bar)
{
bar = 1;
}
int P;
foo(P);
```

P is changed to 1

Pass by Value - function gets separate copy of object, duplicating wastes time and space
Pass by Reference - function gets direct access to object

Const
Const can be used to indicate

1. Variable does not change
2. Member function does not change its object
3. Function does not change its parameter

```
void MyClass::foo(const int & bar) const
{ const double PI = 3.14; }
```

const - foo does not change the variable passed as bar
const - foo does not change the MyClass it's invoked on
const - PI does not change

Recursion
A function that is defined in terms of itself is called recursive

Base Case - Idea behind recursion is to reduce a problem to the point where it is simple to solve without using recursion (base case). A recursive definition for a function must always reach a base case.

```
int factorial (int n)
if (n== 1) // base case
return 1;
else
return n * factorial(n-1);
```

```
int main( ) {
  MyString A;
  MyString B("Hello");
  MyString C(A);
  MyString *ptr = new MyString( );
  if( B== A)
     cout << A;
  cin >> B;
  C = A + B;
  ptr = &B;
  A = *ptr;
  myFunc1(C);
  myFunc2(C);
}
....
void myFunc1(MyString S) {
```

default constructor

constructor

copy constructor

default constr.

operator==

stream-insertion - friend operator<<

stream-extraction - friend operator>>

operator + member

assignment operator

assignment op.

copy constructor

Functions for Classes

1. Copy constructor and the rule of three (big five in C++ 11): used to create a copy of an object
2. Stream insertion operator: used to print out object
3. Destructor: Used to free up memory

Copy Constructor

Often we need to make a new object that is a copy/clone of an existing object.
Passing by value makes a copy so when calling

```
void foobar (Pear ob) { }
```

ob is initialized by running the copy constructor
Compiler provides default copy constructor but one should be provided, especially for classes with pointers

Shallow Copy

Suppose class Pear contains a pointer to a Banana.
If we use the default copy constructor, then

```
Pear P;
Pear Q(P);
```

The object Q will have a pointer that points to the same Banana as the pointer in P. This means that changes to P also change Q. We say Q is a shallow copy.

Deep Copy

To get a separate copy we write:

```
class Pear {
Banana ptr;
Pear(Pear & other) : ptr ( new Banana( *(other.ptr ) ) ) { }
```

This assumes Banana has its own copy constructor

Rule of Three

All three should either be default or coded

1. Copy constructor
2. The copy assignment operator (P = Q)
3. The destructor

In C++ 11

4. Move constructor
5. Move assignment operator

Stream Insertion Operator

```
Pear P;
cout << P;
```

This is achieved by a friend function (not a member function) with signature

```
ostream & operator << (ostream & out, const Pear & ob) {
```

Inheritance
Allows one class to be a specialization of another class
Allows for code reuse and abstraction (diff. classes can be treated the same)

Inheritance in C++
```
class Derived : public Base
{
//Inherited member variables // from base additional member variables
new constructors and destructors
//inherited functions; // from base overridden functions; //
replacing Base new functions;
};
```

Access Control

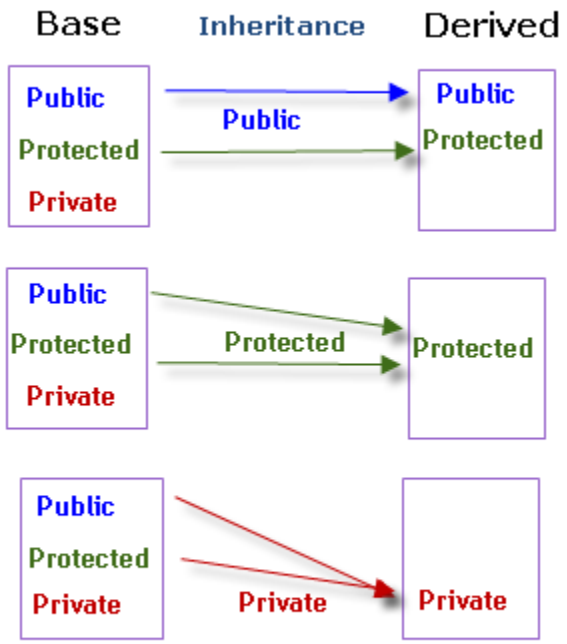

Fig: Visibility after inheritance

Polymorphism
The derived class is a special case of the Base class. Thus it can be used wherever the Base class can. But not vice versa.
```
Base *R = new Derived(); //okah
Derived *S = new Base(); //not allowed
```

Base constructor executed before derived class constructor, default constructor for base class is used unless otherwise specified
Derived destructor executed before base class destructor

Virtual Functions, Pure Functions, and Abstract Base Classes

Casting

```
Base *R = new Derived(); // Okay
```

To use the member functions of Derived, one has to “remind” the compiler that we know that the object R is pointing to is a Derived object:

```
Derived Q = static_cast<Derived> (R);
```

Overriding

The derived class can provide its own version of a function that appears in the base class
(Note that a function is determined by its name and the types of its arguments (and not its return type).)
This raises the question of which version of a function is executed

Virtual and Nonvirtual

If a function is labeled virtual in the Base class, then the Derived version of a function is always used. Otherwise which version is used is determined by how the object is referenced. (We omit the details).

Comment on Virtual Functions

Note that if a function is labeled virtual in the Base class, it is automatically virtual in the Derived class.

Destructors

Destructors should be virtual: the Derived version should always be run on a Derived object

Pure Functions

A virtual function might have no implementation at all in the Base class. This is indicated by writing

```
virtual void scream ( ) = 0;
```

in the class definition.

Abstract Base Classes

An abstract base class is one with at least one pure function. We cannot instantiate an object of an abstract base class