Classes

Declaration
```
class classname
{
public:
  classname(params);
  ~classname();
  type member1;
  type member2;
protected:
  type member3;
private:
  type member4;
};
```

Example
```
class Square
{
public:
  Square();
  Square(float w);
  void setWidth(float w);
  float getArea();
private:
  float width;
};
```

public members are accessible from anywhere the class is visible.

private members are only accessible from the same class or a friend (function or class).

protected members are accessible from the same class, derived classes, or a friend (function or class).

constructors may be overloaded just like any other function. You can define two or more constructors as long as each constructor has a different parameter list.

Definition of Member Functions
```
return_type classname::functionName(params)
{
    statements;
}
```

Examples
```
Square::Square()
{
    width = 0;
}

void Square::setWidth(float w)
{
    if (w >= 0)
      width = w;
    else
      exit(-1);
}

float Square::getArea()
{
    return width*width;
}
```

Definition of Instances
```
classname varName;

classname* ptrName;
```

Example
```
Square s1();
Square s2(3.5);

Square* sPtr;
sPtr=new Square(1.8);
```

Accessing Members
```
varName.member=val;
varName.member();

ptrName->member=val;
ptrName->member();
```

Example
```
s1.setWidth(1.5);
cout << s.getArea();

cout<<sPtr->getArea();
```

Inheritance

Inheritance allows a new class to be based on an existing class. The new class inherits all the member variables and functions (except the constructors and destructor) of the class it is based on.

Example
```
class Student
{
public:
  Student(string n, string id);
  void print();
protected:
  string name;
  string netID;
};

class GradStudent : public Student
{
public:
  GradStudent(string n, string id,
              string prev);
  void print();
protected:
  string prevDegree;
};
```

Visibility of Members after Inheritance

Inheritance Specification	Access Specifier in Base Class		
	private	protected	public
private	-	*private*	*private*
protected	-	*protected*	*protected*
public	-	*protected*	*public*

Operator Overloading

C++ allows you to define how standard operators (+, -, *, etc.) work with classes that you write. For example, to use the operator + with your class, you would write a function named `operator+` for your class.

Example
Prototype for a function that overloads + for the `Square` class:
```
Square operator+ (const Square &);
```

If the object that receives the function call is not an instance of a class that you wrote, write the function as a friend of your class. This is standard practice for overloading << and >>.

Example
Prototype for a function that overloads << for the `Square` class:
```
friend ostream & operator<<
         (ostream &, const Square &);
```

Make sure the return type of the overloaded function matches what C++ programmers expect. The return type of relational operators (<, >, ==, etc.) should be `bool`, the return type of << should be `ostream &`, etc.

Exceptions

Example
```
try
{
  // code here calls functions that might
  // throw exceptions
  quotient = divide(num1, num2);

  // or this code might test and throw
  // exceptions directly
  if (num3 < 0)
    throw -1;  // exception to be thrown can
               // be a value or an object
}
catch (int)
{
  cout << "num3 can not be negative!";
  exit(-1);
}
catch (char* exceptionString)
{
  cout << exceptionString;
  exit(-2);
}
//  add more catch blocks as needed
```

Function Templates

Example
```
template <class T>
T getMax(T a, T b)
{
  if (a>b)
    return a;
  else
    return b;
}


// example calls to the function template
int a=9, b=2, c;
c = getMax(a, b);

float f=5.3, g=9.7, h;
h = getMax(f, g);
```

Class Templates

Example
```
template <class T>
class Point
{
public:
  Point(T x, T y);
  void print();
  double distance(Point<T> p);
private:
  T x;
  T y;
};


// examples using the class template
Point<int> p1(3, 2);
Point<float> p2(3.5, 2.5);
p1.print();
p2.print();
```