```
class Box
{
  private:
    double length;
    double width;
    double height;
};
```

```
Box::Box()
{
  length=0;
  width=0;
  height=0;
}
```

```
Box::Box(double l, double w, double h)
{
  length = l;
  width = w;
  height = h
  more code?
}
```

```
Box::Box(double l, double w, double h):
    length(l), width(w), height(h)
  {

    more code?

  }
```

one or other

initialization list

The use of an initialization list is sometimes required for the implementation of class inheritance, constants, and references

Use it

Classes

recall that a structure, which is a user defined data type, models an object. For example,

```
struct Box
{
    double length;
    double width;
    double height;
};
```

← called the attributes of the struct Box

This structure models a box

Note: A structure has just data types or fixed attributes with no actions (read functions) on them. This is what we used in CS I. There could be actions / functions on these attributes. They (both actions and attributes) are public - they are accessible anywhere in the main program

A <u>class</u>* is a user defined data type which is a generalization of a structure which has actions/functions on the attributes/data types which allows for information hiding.

<u>Example</u> – this is a trivial class, no actions

```
class Box
{
  private:
      double length;
      double width;
      double height;

  public:
      ≡
};
```

data members: attribute names of the object here (sometimes actions placed here but not usually)

data methods: actions or functions of the object placed here

* structs are classes where everything is public. struct is maintained for compatability with C. classes were solely created in C++ – they are free to evolve in ways that struct cannot since it is a leftover from C. classes created for encapsulation

Note: use a full name for the private attributes - dont use variables

Every class needs a Constructor

A constructor is a function of the class that is called when a new object (evey object too) of the class is declared. It initializes the objects of the class as they are created.

when we declare a Language defined type, say int, the Language/compilier creates a variable of said type, initializes it (if we did) and stores it - thus the compilier is responsible for allocating memory for the int variables.

Since we are creating new types/objects, we must create the object of the defined type, initialize it and allocate memory for it - this is the rôle of the Constructor. Constructors initialize the data members, whichare usually private, for an object

There are 4 ways to write a constructor

(i) default Constructor - called when an object is created with no initial values

(ii) constructor with initialization list - called when an object is created with initial values

(iii) constructor with default values assigned using initialization list

(iv) constructor without initialization list

<u>Note:</u> it is best to always have a default constructor and one of the 3 remaining types

<u>Note:</u> initialization list constructor is the best of the remaining 3 - more used / more general

(i) Default Constructor

prototype ClassName ( );

implementation

```
ClassName :: ClassName( )
{
}  → C++ statements or functions can be here
```

(ii) Constructor with list

let $v_1, v_2, ---, v_n$ be variable symbols/names

prototype

ClassName(type v_1, type v_2, ---, type v_n);

implementation

```
ClassName :: ClassName(type v1, ---, type vn) : attributename1 (v1),
                                                attributename2 (v2),
                                                  :
                                                attributenamen (vn)
{
}  → C++ statements or functions can be here
     to initialize remaining data member
```

Order matters

(iii) constructor with list assigning default values

prototype

ClassName (type v_1 = something , ---- , type v_n = something);

implementation

order matters

ClassName :: ClassName (type v_1,---, type v_n) : attributename$_1$ (v_1) ,
attributename$_2$ (v_2),
⋮
attributename$_n$ (v_n)

{
→ C++ statements or functions can be here to initialize data members that remain to be initialized
}

(iv) constructor without list

prototype

ClassName (type v_1, --- , type v_n);

implementation

ClassName:: ClassName (type v_1, ---, type v_n)
{
attributename$_1$ = v_1 ;
⋮
attributename$_n$ = v_n ;
}

Note: see Deitel + Deitel (3rd edition pg 414 - 415) for an additional way to set up a constructer which gives both default values and supplied values at the same time

Every class should have a destructer

syntax

```
class  ClassName
{
  private:
       ≡  attribute names
  public:
       constructors
     ~ClassName( ); ← destructor
};
```

objects that are created by a constructor must be cleaned up in an orderly manner. The tasks involved in cleaning up include releasing memory and closing files. The destructor destroys objects after they are out of scope. this is done automatically (with out the destructor ~ClassName()) but there are times - namely if we are allocating memory dynamically - where space needs to be freed up - thus we must destroy/destruct the objects manually. On exit from the function block, the object is destroyed - but the object may have accumulated resources (memory, disk blocks, network connections) during its lifetime - but still holds space in memory. the destructor releases the resources held by any objects created

Also need copy constructors and
assignment operators
(see Dattatri pg 44-45

Member Functions for Classes
(ie. actions on class objects)

Note: if we have the following prototype of a function (for any program - whether it has classes or not)

type function (type variablename, ..., type variablename);

we could merely use the following prototype

type function (type, ---, type);

for example

The following prototype

```
int maximum (int x, int y, int z);
```

could be written as (the prototype)

```
int maximum (int , int, int);
```

the compiler at this stage ignores parameter names. They are necessary at the implementation

Member Functions *

prototype

return type functionName (argument list);

in h. file inside the class declaration

implementation

return type ClassName :: functionName (argument list)

in .cpp file

{

statements

}

Calling a member function in main

classObjectName . functionName (argument list);

*Note: member functions are always attached to an object just as an accelerator for a car is a function, it is always attached to a particular car.

namely it is the object. FunctionName. This works/means apply the function to the object.

For example, if we have a class called ClassBox, and a defined member function Volume ( ) - which calculates volume, the statement

```
box1.Volume( );
```

will apply the Volume function to the object or argument box1; i.e. it calculates the volume of box1.

Note: an application program (i.e. main statements) that manipulates the objects of a class can only access the public members of those objects. To do this we use the class member access operator

syntax: objectName.member

For example,

box1.length

will return/calculate the length of box1 if the length is a public member of the class

box1.length = 10

would assign to box1 a length of 10

The main point of classes is to have private members – this is called information hiding or encapsulation. The question is then how to access them? only public members may be accessed with the dot operator.

This is done almost universally* with get() and set() methods – which are public.

* Almost every class will have get() and set() methods

the get() function will "get" a particular privata data member/data attribute, where set() methods will (also called update functions) initialize or "set" the values of a particular data member/data attribute

Note: (see Prinz + Prinz pg 275)
get functions read/access members
set functions manipulate members

Note: not all private members need these functions, but most do

All this work allows for info hiding - they guard or control changes to the data members/attributes

Syntax / Example
of class with get() and set()

in .h file

```
class ClassName
{
private:

    type_1  attributename1;
    type_2  attributename2;
        ⋮

public:

    type_1'  getattributeName1() const;
    type_2'  getattributeName2() const;
        ⋮

    type_k'  setName(type v_1, ..., type v_n);
        ⋮
};
```

real names/words not symbols

see page 8 for const explanation

usually same return type as the attribute name member

usually void return type for setName

$v_1, \ldots, v_n$ are variable symbols or letters implementation

← in .cpp file

```
type'_1   ClassName :: getattributeName1 ( )  const
{
    return attributeName1 ;
}

type'_2   ClassName :: getattributeName2( )  const
{
    return attributeName2;
}
  =

type'_K   ClassName :: setName(type v_1, type v_2, ... type v_n)
{
    attributeName1 = v_1 ;
    attributeName2 = v_2 ;
         :
     etc
}
```

const modifier

get functions are usually declared const (constant)
use const when functions do not modify the object:
they only read it. this follows the principle of least privilege

print functions should also be const.
(see Deitel 4th edition pg 473-475)

---

the const keyword at the end indicates to the compilier that the host object will not modify or be modified by the function.

(see Wang pg 103 for more on read only variables and parameters

---

A primary application of const member functions is to support the passing of arguments by constant reference. Function parameters may be declared as const. For example,

```
void printRect (const Rectangle & r)
{
  cout << "Length is" << r.getLength () << endl;
  :
  cout << "Area is" << r.area () << endl;
}
```

the const keyword prohibits alteration during execution of the function

(see Ford/Topp pg 302 computing using C++...)