

Chapter Four

Structures and Class

Record: - Structure

Thus far you have worked with variable's whose data types are very simple: they are a numbers of either integer or floating-point format with a specific range. These types of variables, who only have a single value to their name, are known as basic variables or primitives. Everything in computers is built on numbers, but not necessarily singular types. Sometimes it's advantageous to group common variables into a single collection. For example a date would require a day, month, and year. From what we have discussed currently explained, you could create three separate variables for each, like so:

```
int day, month, year;
```

This isn't so bad, but what happens if you want to store two dates and not one? You'd have to create three more variables and give them unique names:

```
int day1, month1, year1;  
int day2, month1, year2;
```

This begins to become a hassle. Not only do you have to create many variables, but you have to keep giving them unique names. C++ provides a way to collect similar variables into a single *structure*.

An array is a data structure which holds multiple numbers of objects having the same basic property (data type) in a contiguous memory slots. Programmer can also reserve contiguous memory space for aggregates of elements of arbitrary data types each. A data type which is created to reserve such type of memory space is called user defined data type.

User defined data types can be equally used like predefined data types to declare variable identifiers. For example we can define simple, array, or pointer variable identifier from this user defined data type.

Structure

The term *structure* in C++ means both a user-defined type which is a grouping of variables as well as meaning a variable based on a user-defined structure type. For the purpose of distinction we

will refer to the user-defined type side as *structure definition* and the variable side as *structure variable*.

A structure definition is a user-defined variable type which is a grouping of one or more variables. The type itself has a name, just like 'int', 'double', or 'char' but it is defined by the user and follows the normal rules of identifiers. Once the type has been defined through the C++ 'struct' keyword, you can create variables from it just like you would any other type.

Since a structure definition is a grouping of several types: it is a group of one or more variables. These are known as elements or member variables as they are members of the structure definition they are part of. Following through with our hinted example, a structure definition could be a 'date' which might be made up of three 'int' member variables: 'day', 'month', and 'year'.

Before creating a structure variable you must create a structure definition. This is a blue print for the compiler that is used each time you create a structure variable of this type. The structure definition is a listing of all member variables with their types and names.

When you create a structure variable based on a structure definition, all of the member variables names are retained. The only name you have to give is that of the new structure variable. The element names within that variable will be the same as in the structure type. If you create two structure variables from 'date', both will have all three member variables with the same name in both: 'day', 'month', and 'year'.

Member variables are distinguished by the structure variable they are part of. You wouldn't simply be able to use 'day' by itself; instead you'd have to refer to both the structure variable's name (that you gave when you created it) as well as 'day'.

struct Specification: Defining Structures

Defining a structure is giving the compiler a blue print for creating your type. When you create a variable based on the structure definition, all of the member variables are created automatically and grouped under the name you gave. Note that when you create a variable of any kind, you must give it a unique name that is different than its type. Below is the syntax for a structure definition:

```
struct structname
{
    datatype1 variable1;
    datatype2 variable2;
```

```
};
```

Discussion: Writing a structure definition begins with the word ‘struct’ followed by the type-to-be, and ended with a structure block that is ultimately terminated with a semicolon. **Do not forget this trailing semi-colon when defining a structure!**

The name of a structure definition is known as the structure *tag*. This will be the name of the type that you create, like ‘int’ or ‘float’. It is the type that you will specify when creating a structure variable. This structure block is remarkably similar to a statement block since it starts and ends with curly braces. But don’t forget that it ultimately ends with a semi-colon. Within the structure block you declare all the member variables you want associated with that type. Declare them as you would normal variables, but do not try to initialize them. This is simply a data blue print, it is not logic or instructions and the compiler does not execute it.

The *data members* (synonym for member variables) of a structure won’t actually be created until a variable based on the structure is created. Technically an ‘int’ is just this as well. It’s a description of a storage unit. That storage unit isn’t reserved until you create a variable with it.

Example defining a student struct,

```
struct student
{
    int id;
    char name[15];
};
```

Example: The follow defines a structure called ‘date’ which contains three ‘int’ member variables: ‘day’, ‘month’, and ‘year’:

```
struct date {
    int day;
    int month;
    int year;
};
struct date
{
    int day, month, year;
};
```

Note: You *cannot* initialize member variables in a structure definition. The following is *wrong* and will *not* compile:

```
struct date{
    int day = 24, month = 10, year = 2001;
};
```

A structure definition has the same type of scoping as a variable. If you define a structure in a function, you will only be able to use it there. If you define it in a nested statement block, you will only be able to use it inside there and any statement blocks nested within it. But the most common place is defining it globally, as in outside of any functions or blocks. Typically, you'll want to be able to create variables of the defined structure anywhere in your program, so the definition will go at the top. The following is an empty program that defines a 'date' structure globally:

```
#include <iostream.h>

struct date{
    int day, month, year;
};
int main(){
    return 0;
}
```

Declaring and using struct data types

Once you have defined a structure you can create a variable from it just as you would any other variable.

```
student std1;
date birthday;
```

The above declaration statements would create a variable called 'birthday' whose type is the structure 'date'. The variable contains three parts: 'day', 'month', and 'year'. What this actually does is set aside a whole block of memory that can contain all of the member variables. Each member variable then occupies a chunk of it for their individual storage units. The member variables are not actually created one at a time.

Storage for member variables exist at some *offset* from the beginning of the glob of memory reserved by the entire structure. For example, in our 'date' structure variable the first member variable is 'day' so it exists at offset 0. The next member variable, 'month' in this case, will exist at the next available offset. If 'day' uses 4 bytes (32 bits), then 'month' will be at offset 4:

```
int i;
student std1;
```

The above statements are similar in nature because both declare a variable of a given type. The former is a built in type, which is integer while the later declares a variable type of user-defined type. std1 will have the characteristics of representing id and name of a student.

Initializing Structure Variables

You cannot initialize member variables in the structure definition. This is because that definition is only a map, or plan, of what a variable based on this type will be made of. You can, however, initialize the member variables of a structure *variable*. That is, when you create a variable based on your structure definition you can pass each member variable an initializer.

To initialize a structure variable's members, you follow the original declaration with the assignment operator (=). Next you define an initialization block which is a list of initializers separated by commas and enclosed in curly braces. Lastly, you end it with a semi-colon. These values are assigned to member variables in the order that they occur. Let's look at an example:

```
date nco_birthday = { 19, 8, 1979 };
student std1={"Ababe", "Scr/2222/22"};
```

This creates a variable called 'nco_birthday' and initializes it to a list of values. The values are assigned to the member variables in the order they are declared in the structure definition. Remember what is mentioned about each member variable having an offset. The same order in which each member is given an offset is the order in which each is assigned a value in an initialization. So the first initializer is used to initialize the first member variable in the structure, next is the second, and so on and so forth. This order of initialization continues until the values run out.

If you try to assign more values than are member variables, you will get a compiler error. However, you can assign *fewer* values than there are member variables. If there are no more values to be assigned, the assignment will simply end. For example, if we had omitted the last value, '1979', then no value would be assigned to 'year'.

It is possible to use any expression that you normally would. But remember that the expression must result in a value. Here is an example of initialization with things other than literals:

```
int myday = 19;
int mymonth = 5;
date nco_birthday = { myday, mymonth + 3, 2001 - 22 };
```

Although you can assign a value to a variable in the same way you initialize it, the same is not true with structures. So while this works:

```
int x;
x = 0;
```

This doesn't:

```
date nco_birthday;
nco_birthday = { 19, 8, 1979 };
```

Assigning values to multiple members of a structure variable is only possible when that variable is first created. Once a structure variable has been declared, you must access each member individually.

Accessing members of a structure variable

There's not much you can do with the structure itself; much of your time with structure variables will be spent using its members. You can use a member variable in any place you'd use a normal variable, but you must specify it by the structure variable's name *as well as* the member variable's name using the member operator.

To specify that you want a member of a specific structure variable, you use the structure member operator which is the period (also known as a "dot"). Simply use the structure's name, follow with the period, and end with the member:

```
structure.member
```

Example: to reading and displaying values to and from structure s1.

```
cin>>s1.id; //storing to id item of s1
cin>>s1.name; //storing a name to s1
cout<<s1.id; //displaying the content of id of s1.
cout<<s1.name; //displaying name
```

Example:-a program that creates student struct and uses it to store student information.

```
#include<iostream.h>
#include<conio.h>
struct student
{
    int id;
    char name[15];
};

void main()
{
    //creating three student variables
    student s1,s2;
    cout<<"\n Enter Student Id";
    cin>>s1.id;
    cout<<"\nEnter Name";
    cin>>s1.name;
    cout<<"\n Enter Student Id";
    cin>>s2.id;
    cout<<"\nEnter Name";
    cin>>s2.name;

    cout<<"\nStudents Information";
    cout<<"\n Student id\t Student Name";
```

```

        cout<<endl<<s1.id<<"\t"<<s1.name;
        cout<<endl<<s2.id<<"\t"<<s2.name;
        getch();
    }

```

The above program shows you how to capture data in student variables.

Example 2: This program demonstrates using member variables for user input, output, and mathematical operations.

```

#include <iostream.h>
struct date
{
    int day, month, year;
};
int main()
{
    date birth;
    cout << "Enter your birth date!" << endl;
    cout << "Year: ";
    cin >> birth.year;
    cout << "Month: ";
    cin >> birth.month;
    cout << "Day: ";
    cin >> birth.day;
    cout << "You entered " << birth.month << "/"<< birth.day
        << "/" << birth.year << endl;
    cout << "You were born in the "<< ((birth.year / 100) + 1)
        << "th Century!"<< endl;
    return 0;
}

```

Variables with Definition

The syntax of ‘struct’ also allows you to create variables based on a structure definition without using two separate statements:

```

struct tag
{
    member(s);
} variable;

```

The structure variables created in this way will have the same scope as their structure definition. This is a nice thing to use when you want to group some variables in one place in your program without it affecting other things. You can create a structure definition as well as variables from it in that one local place and not have to ever use it again.

Example: A ‘point’ variable right after the ‘pointtag’ structure is defined:

```

struct pointtag{
    int x, y;
} point;

```

In this, ‘point’ is a variable just as if we had declared it separately. In fact, this statement is identical to:

```

struct pointtag{
    int x, y;
};
pointtag point;

```

Rather than defining a structure under a name and then creating variables which refer to the named definition, the definition becomes part of the variable declaration. It is even possible to omit the structure tag which may make more sense in this situation:

```

struct{
    int x, y;
} point;

```

The above creates a structure variable called ‘point’ which has two member variables. Because the structure definition is not named, it cannot be used elsewhere to create variables of the same structure type. However, like in any variable declaration, you can create multiple variables of the same type by separating them with commas:

```

struct{
    int x, y;
} point1, point2;

```

Now we have two structure variables of the same nameless structure type. Even more fun we can initialize these structure variables as we normally would:

```

struct{
    int x, y;
} point1 = { 0, 0}, point2 = {0, 0};

```

This creates the two structure variables, ‘point1’ and ‘point2’, and initializes both members, ‘x’ and ‘y’, on each to zero (0). You don’t need to make a name-less definition to do this. You could still have a structure tag *and* initialize the variables created after the definition.

Note: Not only can you write name-less structures, but you can write them without declaring any variables. This will cause a warning on smarter compilers, but is perfectly legal:

```

struct
{
    int x, y;
};

```

The above structure definition can’t be used to create any variables because it has no name. Likewise, no variables are declared after it either. So this is a nice way to fill up your programs with useless source code. But why stop there? You could make an empty, name-less definition:

```

struct { };

```

Array of structs

```
#include<iostream.h>
#include<conio.h>
struct student {
    int id;
    char name[15];
};

void main()
{
    clrscr();
    //creating 5 student using an array
    student s[5];
    int i;
    for(i=0; i < 5; i ++)
    {
        cout<<"\n Enter Student Id";
        cin>>s[i].id;
        cout<<"\nEnter Name";
        cin>>s[i].name;
    }
    cout<<"\n Displaying student Info";
    cout<<"\nStudent Id \t Student Name";
    cout<<"\n=====";

    for( i = 0; i < 5; i++)
        cout<<endl<<s[i].id<<"\t\t\t"<<s[i].name;

    getch(); }
```

Memory map of the above struct declaration.

0	id	1
	name	Tameru
1	id	2
	name	Hassen
2	id	3
	name	Selamawit
3	id	4
	name	Asia
4	id	5
	name	Micheal

If you use s[0].id you will be referring to 1, and s[0].name will refer to Tameru.

The following program declares and uses a book struct. Also swaps the content of two book struct variables.

```
#include<iostream.h>
#include<conio.h>
struct Book {
    int id;
    char title[15];
};

void main() {
```

```

//creating three Book variables
Book b1,b2,temp;
cout<<"\n Enter Book Id";
cin>>b1.id;
cout<<"\nEnter Title";
cin>>b1.title;
cout<<"\n Enter Book Id";
cin>>b2.id;
cout<<"\nEnter Title";
cin>>b2.title;
cout<<"\n Book Information";
cout<<"\n Before Changing Contents";
cout<<"\n Book id\t Title";
cout<<"\n===== ";
cout<<endl<<b1.id<<"\t\t\t"<<b1.title;
cout<<endl<<b2.id<<"\t\t\t"<<b2.title;
//swapping content
temp=b1;
b1=b2;
b2=temp;
cout<<"\nAfter swapping contents";
cout<<"\n Book Information";
cout<<"\n Book id\t Title";
cout<<"\n===== ";

cout<<endl<<b1.id<<"\t"<<b1.title;
cout<<endl<<b2.id<<"\t"<<b2.title;
getch();
}

```

Declaring struct types as part of a struct

A structure definition contains multiple variables, but not necessarily just primitives. You can define a structure to have *structure member variables*.

Now if you have data's like birth of day of an employee, published year of a book, address of a person. What are you going to do? You must be able to incorporate this type of data's in other structs. The following program declares two structs one for address and other for student.

```

#include<iostream.h>
#include<conio.h>
struct Address{
    int kebele;
    char Kefle_ketema[20];
    char roadname[20];
};
struct Student {
    int id;
    char name[15];
    char section[6];
    //declaring address type within student
    Address studaddress;
};
void main() {
    clrscr();

```

```

//creating Student type that encapsulates Address
Student s1;
cout<<"\n Enter Student Id";
cin>>s1.id;
cout<<"\nEnter Student Name";
cin>>s1.name;
cout<<"\n Enter Section";
cin>>s1.section;
//reading address attributes
cout<<"\nEnter Kebele";
cin>>s1.studaddress.kebele;
cout<<"\nEnter Street Name";
cin>>s1.studaddress.roadname;
cout<<"\nEnter Kefle Ketema";
cin>>s1.studaddress.Kefle_ketema;
cout<<"\n Student Information";
cout<<"\n id\t Name\t Section \t Kebele";
cout<<" \t Street Name \t Kefele Ketema ";
cout<<"\n===== ";
cout<<endl<<s1.id<<"\t"<<s1.name;
cout<<"\t"<<s1.section<<"\t"<<s1.studaddress.kebele;
cout<<"\t"<<s1.studaddress.roadname<<"\t";
cout<<s1.studaddress.Kefle_ketema;
getch();
}

```

The memory map of student s;

id		1
name		Asefa
section		RDDOB02
studaddress	kebele	21
	Kefle_ketema	Arada
	roadname	Gahandi

Using the above figure. You should be able to understand how data is accessed. Example s.id refers id of student, but s.studaddress.kebele refers kebele of student where the kebele is referenced by studaddress, intern part of student.

Example: This assumes that ‘date’ and ‘time’ are structures that have already been declared.

```

struct moment
{
    date theDate;
    time theTime;
};

```

These structure members can then be accessed as normal member variables, but then *their* variables must be accessed as well. If there is a variable based on ‘moment’ called ‘birth’, then we would need to write the following to assign ‘19’ to “birth’s date’s day”:

```
birth.theDate.day = 19;
```

A structure only has to be declared before it can be used in another structure's definition. The following is perfectly acceptable:

```
struct date;
struct time;
struct moment
{
    date theDate;
    time theTime;
};
struct date
{
    int day, month, year;
};
struct time
{
    int sec, min, hour;
};
```

To be able to define a structure you only must know the types and the names of the member variables declared inside. With the above we declare the structures 'date' and 'time' but do not define them until later. This simply acknowledges that they exist and they can therefore be used within 'moment'.

What if 'date' and 'time' hadn't defined? It would still be legal, but then I would not be able to use 'moment' at all. Why? Since 'date' or 'time' have not been defined, the compiler does not know how big they are supposed to be or what kind of data they contain. You couldn't then create a variable based on 'moment' because the compiler doesn't know how big of a memory block to allocate. Likewise if you try to use a structure that has been declared *before* it has been defined, you will encounter the same problem.

Defining Structure in Structure

It is possible to define a structure inside a structure definition and create variables from it at the same time. For example:

```
struct moment
{
    struct date
    {
        int day, month, year;
    } theDate;

    struct time
    {
        int second, minute, hour;
    } theTime;
```

```
};
```

The drawback of the above is that the ‘date’ and ‘time’ definitions cannot be used elsewhere without also referring to the parent structure. If the ‘date’ definition isn’t going to be used elsewhere anyway, the structure tag can simply be omitted. Thus we could remove the ‘date’ and ‘time’ structure type identifiers and it would work fine. You can write any number of variables in a structure definition and a valid variable declaration statement (minus initialization of course) is valid inside a structure definition.

Let’s say we want to be able to use ‘date’ elsewhere, but not ‘time’. The following program demonstrates how the structure definitions would be written as well as uses the defined structure types in an extended “birth date” sample:

```
#include <iostream.h>
struct date { int day, month, year; } ;
struct moment
{
    date theDate;
    struct
    {
        int sec, min, hour;
    } theTime;
};
int main()
{
    moment birth;
    cout << "Enter your birth moment!" << endl;
    cout << "Year: ";
    cin >> birth.theDate.year;
    cout << "Month: ";
    cin >> birth.theDate.month;
    cout << "Day: ";
    cin >> birth.theDate.day;
    cout << "Hour (military): ";
    cin >> birth.theTime.hour;
    cout << "Minute: ";
    cin >> birth.theTime.min;
    cout << "Second: ";
    cin >> birth.theTime.sec;

    cout << "You entered " << birth.theDate.month << "/"
        << birth.theDate.day << "/" << birth.theDate.year
        << " @ " << birth.theTime.hour << ":"
        << birth.theTime.min << ":" << birth.theTime.sec
        << endl;

    if (birth.theTime.hour > 20 || birth.theTime.hour < 8)
        cout << "You were born early in the morning!"
            << endl;

    return 0;
}
```

```
}
```

Any number of structure definitions can be nested; you can get extremely complex with structures, which is why they are sometimes known as *complex types*.

Structure, Reference and Pointer

References to structure variables, work the same way as normal references. To create one you would write out the name of the structure type, followed by the reference name which is preceded by the reference operator (&):

```
struct_name &reference;
```

The following creates a structure variable based on 'date' and a reference to it:

```
date birth;  
date &mybirth = birth;
```

Both of these, 'birth' and 'mybirth', would have access to the same member variables and their values. Thus they can be used interchangeably:

```
birth.year = 1981;  
mybirth.year -= 2;
```

Can you guess what the value of 'birth.year' would be from the above? It would be '1979'. The reference 'mybirth' is just an alias to 'birth' and its group of member variables. Remember that a reference is not a real variable (it has no value), but simply a nickname for another.

Utilizing pointers with structures, unfortunately, adds a previously unseen complexity. A pointer to a variable cannot be used to modify the variable's value until it has been dereferenced. This case is no different from structures. But it affects the way you access the structure variable's members. Recall that a pointer simply contains a memory address. The pointer knows nothing of what this memory address is used for, which is why you have to dereference a pointer to a specific type. Thus, you cannot access a structure variable's members until you dereference a pointer to the structure type:

```
date birth;  
date *p = &birth;  
(*p).year = 1979;
```

The pointer 'p' above, had to be dereferenced before the member variable 'year' could be accessed through it. It was surrounded in parenthesis because the indirection operator (asterisk '*') has a lower precedence than the member operator. Thus the following would not work:

```
*p.year = 1979;
```

This would be seen as “get the value pointed to by ‘p.year’” and ‘p.year’ is an invalid identifier; hence the parenthesis around the indirection operation. This method of accessing a structure variable’s members is cumbersome and requires two operations simply to get at the member variable: indirection and then member. For this purpose, there is an operator specifically for accessing a structure variable’s members through a pointer. This is a member operator known specifically as a pointer-to-member operator or a dash followed by a greater than sign ‘->’ (also known as an “arrow”):

```
p->year = 1979;
```

This operator only works on pointers to structure variables. You cannot use it on normal structure variables for members. The following would not work:

```
birth->year = 1979;
```

The left operand must be a pointer to a variable with members, and the right operand must be the name of a member variable within that.

Classes

A class is basically a structure with member functions as well as member data. Classes are central to the programming methodology known as object-oriented programming.

A class is a logical method to organize data and functions in the same structure. They are declared using keyword **class**, whose functionality is similar to that of the C keyword **struct**, but with the possibility of including functions as members, instead of only data.

DEFINING CLASSES AND MEMBER FUNCTIONS

```
class class_name {  
permission_label_1: member1; permission_label_2: member2; ...  
} object_name;
```

where *class_name* is a name for the class (user defined *type*) and the optional field *object_name* is one, or several, valid object identifiers. The body of the declaration can contain *members*, that can be either data or function declarations, and optionally *permission labels*, that can be any of these three keywords: **private:**, **public:** or **protected:**. They make reference to the permission(ENCAPSULATION) which the following *members* acquire:

- **private** members of a class are accessible only from other members of their same class or from their "*friend*" classes.
- **protected** members are accessible from members of their same class and *friend* classes, and also from members of their *derived* classes.
- Finally, **public** members are accessible from anywhere the class is visible.

If we declare members of a class before including any permission label, the members are considered **private**, since it is the default permission that the members of a class declared with the **class** keyword acquire.

For example:

```
class CRectangle {
    int x, y;
    public: void set_values (int,int);
    int area (void);
} rect;
```

Declares class **CRectangle** and an object called **rect** of this class (type). This class contains four members: two variables of type **int** (**x** and **y**) in the **private** section (because private is the default permission) and two functions in the **public** section: **set_values()** and **area()**, of which we have only included the prototype.

Notice the difference between class name and object name: In the previous example, **CRectangle** was the class name (i.e., the user-defined type), whereas **rect** was an object of type **CRectangle**. Is the same difference that **int** and **a** have in the following declaration:

`int a;` **int** is the *class name* (type) and **a** is the *object name* (variable).

On successive instructions in the body of the program we can refer to any of the public members of the object **rect** as if they were normal functions or variables, just by putting the object's name followed by a point and then the class member (like we did with C **structs**). For example:

```
rect.set_value (3,4); myarea = rect.area();
```

but we will not be able to refer to **x** or **y** since they are private members of the class and they could only be referred from other members of that same class. Confused? Here is the complete example of class **CRectangle**:

```
// classes example
#include <iostream.h>
```

```

class CRectangle {
    int x, y;
public:
    void set_values (int,int);
    int area (void)
    {return (x*y);}
};

void CRectangle::set_values (int a, int b)
{ x = a; y = b; }

int main () {
    CRectangle rect;
    rect.set_values (3,4);
    cout << "area: " << rect.area();
}

```

The new thing in this code is the operator `::` of scope included in the definition of **set_values()**. It is used to declare a member of a class outside it. Notice that we have defined the behavior of function **area()** within the definition of the **CRectangle** class - given its extreme simplicity. Whereas **set_values()** has only its prototype declared within the class but its definition is outside. In this outside declaration we must use the operator of scope `::`.

The scope operator (`::`) specifies the class to which the member being declared belongs, granting exactly the same scope properties as if it was directly defined within the class. For example, in the function **set_values()** of the previous code, we have referred to the variables **x** and **y**, that are members of class **CRectangle** and that are only visible inside it and its members (since they are **private**).

The only difference between defining a class member function completely within its class and to include only the prototype, is that in the first case the function will automatically be considered *inline* by the compiler, while in the second it will be a normal (not-inline) class member function. The reason why we have made **x** and **y** **private** members (remember that if nothing else is said all members of a class defined with keyword *class* have **private** access) it is because we have already defined a function to introduce those values in the object (**set_values()**) and therefore the rest of

the program does not have a way to directly access them. Perhaps in a so simple example as this you do not see a great utility protecting those two variables, but in greater projects it may be very important that values cannot be modified in an unexpected way (unexpected from the point of view of the object).

One of the greater advantages of a class is that we can declare several different objects from it. For example, following with the previous example of class **CRectangle**, we could have declared the object **rectb** in addition to the object **rect** :

```
// class example
#include <iostream.h>
class CRectangle {
int x, y;
public: void set_values (int,int);
int area (void) {return (x*y);}
};
void CRectangle::set_values (int a, int b)
{ x = a; y = b; }
int main () {
CRectangle rect, rectb; rect.set_values (3,4); rectb.set_values (5,6);
cout << "rect area: " << rect.area() << endl;
cout << "rectb area: " << rectb.area() << endl; }
```

Notice that the call to **rect.area()** does not give the same result as the call to **rectb.area()**. This is because each object of class **CRectangle** has its own variables **x** and **y**, and its own functions **set_value()** and **area()**

.

On that is based the concept of **object** and **object-oriented programming**. In that data and functions are properties of the object, instead of the usual view of objects as function parameters in structured programming. In this and the following sections we will discuss advantages of this methodology.

In this concrete case, the class (type of object) to which we were talking about is **CRectangle**, of which there are two instances, or objects: **rect** and **rectb**, each one with its own member variables and member functions.

Constructors and destructors

Objects generally need to initialize variables or assign dynamic memory during their process of creation to become totally operative and to avoid returning unexpected values during their execution. For example, what would happen if in the previous example we called the function **area()** before having called function **set_values**? Probably an indetermined result since the members **x** and **y** would have never been assigned a value.

In order to avoid that, a class can include a special function: a *constructor*, which can be declared by naming a member function with the same name as the class. This constructor *function* will be called automatically when a new instance of the class is created (when declaring a new object or allocating an object of that class) and only then. We are going to implement **CRectangle** including a *constructor*:

```
// classes example
#include <iostream.h>
class CRectangle {
int width, height;
public: CRectangle (int,int);
int area (void) {return (width*height);}
};
CRectangle::CRectangle (int a, int b) { width = a; height = b; }
int main () {
    CRectangle rect (3,4);
    CRectangle rectb (5,6);
    cout << "rect area: " << rect.area() << endl;
    cout << "rectb area: " << rectb.area() << endl;
}
```

As you can see, the result of this example is identical to the previous one. In this case we have only replaced the function **set_values**, that no longer exists, by a class *constructor*. Notice the way

in which the parameters are passed to the constructor at the moment at which the instances of the class are created:

```
CRectangle rect (3,4); CRectangle rectb (5,6);
```

You can also see how neither the prototype nor the later constructor declaration includes a return value, not even **void** type. This must always be thus. A constructor never returns a value nor does the **void** have to be specified, as we have shown in the previous example.

The **Destructor** fulfills the opposite functionality. It is automatically called when an object is released from the memory, either because its scope of existence has finished (for example, if it was defined as a local object within a function and the function ends) or because it is an object dynamically assigned and it is released using operator **delete**.

The destructor must have the same name as the class with a tilde (~) as prefix and it must return no value.

The use of destructors is specially suitable when an object assigns dynamic memory during its life and at the moment of being destroyed we want to release the memory that it has used.

// example on constructors and destructors

```
#include <iostream.h>

class CRectangle {
int *width, *height;
public: CRectangle (int,int);
~CRectangle ();
int area (void)
{return (*width * *height);} };

CRectangle::CRectangle (int a, int b)
{ width = new int; height = new int; *width = a; *height = b; }

CRectangle::~~CRectangle () { delete width; delete height; }

int main () {
CRectangle rect (3,4), rectb (5,6);
cout << "rect area: " << rect.area() << endl;
cout << "rectb area: " << rectb.area() << endl;
return 0;
```

}

THE DOT OPERATOR AND THE SCOPE RESOLUTION OPERATOR

Both the dot operator and the scope resolution operator are used with member names to specify of what thing they are a member. For example, suppose you have declared a class called DayOfYear and you declare an object called today as follows:

```
DayOfYear today;
```

You use the dot operator to specify a member of the object today. For example, output is a member function for the class DayOfYear (defined in Display 6.3), and the following function call will output the data values stored in the object today:

```
today.output( );
```

You use the scope resolution operator, ::, to specify the class name when giving the function definition for a member function. For example, the heading of the function definition for the member function output would be as follows:

```
void DayOfYear::output( )
```

Remember, the scope resolution operator, ::, is used with a class name, whereas the dot operator is used with an object of that class.

Exercise

1. **Employee Information:** You are developing an employee management system that stores information about employees. Define a structure called "Employee" with fields such as ID, name, age, and salary. Write a C++ program that creates an array of Employee structures, prompts the user to enter details for multiple employees, and then displays the information of each employee.
2. **Student Database:** You are working on a student database system for a school. Define a structure called "Student" with fields such as roll number, name, age, and grade. Write a C++ program that allows the user to enter details for multiple students, stores them in an array of Student structures, and then displays the information of each student.
3. **Book Inventory:** You are designing a book inventory system for a library. Define a structure called "Book" with fields such as ISBN, title, author, and price. Write a C++ program that allows the user to enter details for multiple books, stores them in an array of Book structures, and then displays the information of each book.
4. **Bank Account Management:** You are developing a program to manage bank accounts. Define a structure called "Account" with fields such as account number, holder name, balance, and account type. Write a C++ program that allows the user to create and manage multiple bank accounts using Account structures.
5. **Product Catalog:** You are creating a product catalog for an online store. Define a structure called "Product" with fields such as product ID, name, price, and quantity. Write a C++ program that allows the user to add products to the catalog, update their details, and display the catalog.
6. **Contact Management:** You are implementing a contact management system for a mobile phone. Define a structure called "Contact" with fields such as name, phone number, email, and address. Write a C++ program that allows the user to add contacts, search for contacts by name or phone number, and display the contact list.
7. **Car Rental System:** You are designing a car rental system for a rental company. Define a structure called "Car" with fields such as registration number, model, make, and rental price per day. Write a C++ program that allows the user to add cars to the rental inventory, rent them out, and calculate rental charges.
8. **Employee Payroll:** You are developing a payroll system for a company. Define a structure called "Payroll" with fields such as employee ID, name, hours worked, hourly rate, and total pay. Write

a C++ program that allows the user to enter details for multiple employees, calculate their total pay, and display the payroll.

9. Course Management: You are creating a course management system for a university. Define a structure called "Course" with fields such as course code, name, instructor, and schedule. Write a C++ program that allows the user to add courses, assign instructors, update schedules, and display course details.
10. Movie Database: You are designing a movie database for a streaming service. Define a structure called "Movie" with fields such as title, genre, director, and release year. Write a C++ program that allows the user to add movies to the database, search for movies by title or genre, and display movie details
11. Pointer to Structure: You are developing a database management system for storing employee information. Write a C++ program that defines a structure for an employee with fields like name, age, and salary. Use a pointer to structure to create and display employee records.
12. Pointer to Object: You are creating a banking application that manages customer accounts. Write a C++ program that defines a class for a bank account with fields like account number, balance, and customer name. Use a pointer to an object to create and manipulate account objects.
13. Define a class called **Rectangle** to represent rectangles. Include member variables for length and width, member functions to set and get the dimensions, and a member function to calculate the area.
14. Create a class named **Employee** to store employee details such as name, ID, and salary. Include appropriate member functions for setting and getting these details.
15. Implement a class **Circle** to represent circles. Include member variables for radius, member functions to set and get the radius, and a member function to calculate the area of the circle.
16. Define a class **Student** to store student information such as name, roll number, and marks. Include appropriate member functions for setting and getting this information.
17. Create a class **BankAccount** to represent a bank account. Include member variables for account number, balance, and account holder name, along with member functions for deposit, withdrawal, and displaying account details.
18. Implement a class **Car** to represent cars. Include member variables for make, model, year, and price, along with member functions to set and get these details.

19. Define a class **Book** to store information about books such as title, author, and price. Include appropriate member functions for setting and getting this information.
20. Create a class **Time** to represent time in hours, minutes, and seconds. Include member functions to set and get the time, and a member function to display the time in 12-hour format.
21. Implement a class **Person** to store personal information such as name, age, and address. Include appropriate member functions for setting and getting this information.
22. Define a class **Rectangle** to represent rectangles. Include member variables for length and width, member functions to set and get the dimensions, and a member function to calculate the perimeter.
23. **Bank Account**: Create a class representing a bank account with methods to deposit, withdraw, and check balance. Ensure that the account balance is encapsulated and accessible only through class methods.
24. **Student Record**: Design a class to store student records with attributes such as name, roll number, and marks. Implement methods to set and get these attributes, ensuring encapsulation.
25. **Employee Management System**: Develop a class for an employee with attributes like name, employee ID, and salary. Include methods to set and get these attributes, as well as calculate bonuses or increments.
26. **Car Rental System**: Design a class for managing car rentals. Include attributes like car model, rental duration, and rental fee. Implement methods to calculate the rental cost and display rental details.
27. **Library Management System**: Create a class representing a library book with attributes such as title, author, and availability status. Implement methods to check out and return books, ensuring proper encapsulation.
28. **Inventory Management**: Develop a class to manage inventory items in a store. Include attributes like item name, quantity, and price. Implement methods to update inventory levels and display item details.
29. **Ticket Booking System**: Design a class for managing ticket bookings with attributes like seat number, booking status, and passenger details. Implement methods to book and cancel tickets, ensuring data encapsulation.
30. **Hospital Management System**: Create a class for managing patient records in a hospital. Include attributes like patient name, age, and medical history. Implement methods to add new patients, update records, and retrieve patient information.

31. **Online Shopping Cart:** Design a class representing a shopping cart for an online store. Include attributes like product name, quantity, and price. Implement methods to add, remove, and update items in the cart, ensuring encapsulation.
32. **Restaurant Ordering System:** Develop a class for managing orders in a restaurant. Include attributes like order ID, items ordered, and total cost. Implement methods to add new orders, calculate bills, and display order details.