Chapter Two
Structures

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.

- For example a date would require a day, month, and year.
  - We can declare three separate variables like
    - 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.
- But we can define structure that 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’.

struct Specification: Defining Structures

- Syntax for defining structure

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

 };
```

- 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 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.

Example defining a student struct

- 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.

```
#include <iostream.h>

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

Declaring and using sturct data types

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

- 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:

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.

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

- 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.

- 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
};
```

Accessing members of a structure variable

- 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>
using namespace std;
struct student
{
    int id;
    char name[15];
};
int main() {
    student s1,s2;
    cout<<"enter student id"<<endl;
    cin>>s1.id;
    cout<<"Enter student name"<<endl;
    cin>>s1.name;
    cout<<"enter student id"<<endl;
    cin>>s2.id;
    cout<<"Enter student name"<<endl;
    cin>>s2.name;

    cout<<"Student Inforamtion"<<endl;
    cout<<"**********************"<<endl;
    cout<<"student id \t student name"<<endl;
    cout<<s1.id<<"\t\t\t"<<s1.name<<endl;
    cout<<s2.id<<"\t\t\t"<<s2.name<<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.

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;
```

- Or you can even initialize it.

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

Array of structures

- An array of structures refers to an array in which each element is of structure type.
- To declare an array of structures, firstly, a structure is defined and then an array of that structure is declared. The syntax for declaring an array of structures is:

```
structure_name structure_variable[size];
```

```cpp
#include <iostream>
using namespace std;
struct student{
        int id;
        char name[15];
    };
    int main(){
        student s[5];
        int i=0;
        for(i=0;i<5;i++){
            cout<<"enter studnet id"<<endl;
            cin>>s[i].id;
            cout<<"enter student name"<<endl;
            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;

    return 0;
}
```

- 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

```
#include<iostream>
using namespace std;
struct Book {
        int id;
        char title[15];
      };
int 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;
          }
```

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.
- The following program declares two structs one for address and other for student.

- The memory map of student s;

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

- 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:
- The drawback of the above is that the ‘date’ and ‘time’ definitions cannot be used elsewhere without also referring to the parent structure.

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

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

```
#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.theDate.hour << “:”
      << birth.theDate.min << “:” << birth.theDate.sec
      << endl;

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

   return 0;
}
```

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.

Passing structure to function

- we can pass a structure variable to a function as argument like we pass any other variable to a function.
- Structure variable is passed using call by value.
- To take a structure variable as argument, function must declare a structure argument in it's declaration.
- Any change in the value of formal parameter inside function body, will not affect the value of actual parameter.

- Example

```
struct employee {
    char name[100];
    int age;
    float salary;
    char department[50];
};
void printEmployeeDetails(employee emp);
```

- We can also pass address of a structure to a function.
- In this case, any change in the formal parameter inside function's body will reflect in actual parameter also.
- To take a structure pointer as parameter a function declare a structure pointer as it's formal parameter.

```
void printEmployeeDetails( employee *emp);
```

- Like any other inbuilt data type, we can also pass individual member of a structure as function argument.

```cpp
#include <iostream>
using namespace std;

struct student
{
    int id;
    string name;
    string dep;
} s1;
student getStudent();
void showstudnet(student );
void showstudnet2(student *p);
void showDep(string);
int main() {
    student temp=getStudent();
    showstudnet(temp);
    showstudnet2(&temp);
    showDep(s1.dep);
    return 0;
}   student getStudent(){
        cout<<"enter student id"<<endl;
        cin>>s1.id;
        cout<<"Enter student name"<<endl;
        cin>>s1.name;
        cout<<"enter the dep of the student"<<endl;
        cin>>s1.dep;
        return s1;//returning  a strucure
    }
//passing structure elemenet
void showDep (string str){
    cout<<"the student is in dep of "<<str<<endl;
}
//passing structure by reference
void showstudnet2(student *s){
    cout<<"Student Inforamtion"<<endl;
    cout<<"**********************"<<endl;
    cout<<"student id \t student name"<<endl;
    s->id=9;
    s->name="kebede";
    cout<<s->id<<"\t\t\t"<<s->name<<endl;
}
    //passing structure by value
    void showstudnet(student s){
        cout<<"Student Inforamtion"<<endl;
        cout<<"**********************"<<endl;
        cout<<"student id \t student name"<<endl;
        cout<<s.id<<"\t\t\t"<<s.name<<endl;
```

Passing array of structure to a function

- Like a normal array of elements we can pass array of structure to a function.
- For a normal array
  - Function prototype:
    - return type functionName (datatype[], int size);
  - Fucntion call:
    - functionName(variableName, size);
  - Function definition
    - return type functionName (datatype variableName [], int size);

```
#include <iostream>

using namespace std;
void acceptnumber();
int mini(int [], int);
int maxx(int [], int);

int main() {
    acceptnumber();
    return 0;
}
void acceptnumber(){
    int n;
    cout<<"how many numbers do you want to compare"<<endl;
    cin>>n;
    int nums[n];
    cout<<"enter the "<<n<<"numbers"<<endl;
    for (int i=0;i<n;i++)
    {
        cin>>nums[i];
    }
    int min=mini(nums,n);
    cout<<"min="<<min<<endl;
    int max=maxx(nums,n);
    cout<<"max"<<max<<endl;
}
```

```
int mini(int num[],int n)
{
    int m=num[0];
    for (int i=1;i<n;i++){
        if(num[i]<m){
            m=num[i];
        }

    }
    return m;
}
```

```
int maxx(int num[],int n)
{
    int m=num[0];
    for (int i=1;i<n;i++){
        if(num[i]>m){
            m=num[i];
        }

    }
    return m;
}
```

- For array of structure
  - Function prototype:
    - return type functionName (structName [], int size);
  - Fucntion call:
    - functionName(structName, size);
  - Function definition
    - return type functionName (structName variableName [], int size);

```cpp
#include <iostream>

using namespace std;
struct student {
    string name;
    int id;
};
void print(student [], int n);
void accept();
int main() {
    accept();
    return 0;
}
```

```cpp
void accept(){
    student s[3];
    for (int i=0;i<3;i++){
        cin>>s[i].name;
        cin>>s[i].id;
    }
    print(s,3);
}
void print(student s[],int n){
    for (int i=0;i<3;i++){
        cout<<s[i].name;
        cout<<"\t";
        cout<<s[i].id;
        cout<<endl;
    }
```