Using Structures in C++

Intro

Structures form a very large building block with which to collect like data into one collective unit. They are a versatile data structure in which to clump data together in convenient little packages! They are essentially classes with all members defined as public access with no private or protected access modes available. They are most commonly used for conglomerating data and also support member functions, but that attribute is rarely used. I believe this is because when multiple functions are involved, the need for private and protected variables and functions increases.

Definition and Syntax

Arrays are one of the most widely used data structures in programming languages. One downfall of using such a data type is that one must use homogeneous data types, an array can only hold multiple items of the same type. Structures overcome this problem by allowing the programmer to have an unlimited number of items of different data types! Objects contained within the structure are referred to as a members. Here's an example of what a structure declaration looks like:

```
struct name
{ type1 Member1;
  type2 Member2;
  typeX Memberx;
} copyname,listname[x];
```

Example 1

Here's our general structure definition in Example 1. We use the struct keyword followed by a name for the structure. This name must be defined if we would like to make multiple copies of our structure. If we only need one instance, we can declare copyname as our instance name and disregard the name field. We can use listname[x] if we would we know that we will be immediately needing x copies of our structure.

In Example 2, we have declared our structure and have members that we can access. Before we can actually access any members, we have to have a copy to work with. This is called an instance. So far we have created a so-called template for constructing copies of the Games structure, but we still haven't actually made one. We can create a copy 2 different ways. We can make a copy in our declaration, or we can declare one later on in our code. Here's the first way:

```
struct Games
{ char Name[80];
  char Rating;
  BOOL Played;
  long NumberOfKills;
}
```

Example 2

```
struct Games
{ char Name[80];
  char Rating;
  BOOL Played;
  long NumberOfKills;
} Unreal, Blizzard[3];
typedef Games* Gamesptr;
```

Example 3

Now we have a named structure (Games) that will allow us to make instances later on in our code, and we also have Unreal which IS an instance that we can use. Since we have named our structure Games, we can make however many copies we like later on in the code if we would like to. Blizzard is also an instance, but it is a list of instances, actually it is an array of instances. Notice that we also declare Gamesptr to be a pointer to a Games structure. This is if we want to point to an instance or if we want to dynamically allocate instances of the Games structure. We will get to accessing the members in the next section.

The second way of created instances, is if we create an instance later in our code using what we filled in the name field as shown in Example 4. Here we create an instance named Unreal by using the what we filled in the name field in our structure declaration. We also create a Games pointer (GamesList) using the typedef that we created (Gameptr). We then dynamically allocate an array of structure instances. Realize that up to this point, all instances have been created during compile time and none during run-time. Utilizing the Gameptr will allow us to make a number of instances of a structure during run time when we know how many we will need according to run-time data instead of wasting unused instances.

```
void main()
{  Games Unreal;
   Gameptr GamesList;
   GamesList = new Games[30];
   ...
   ...
   delete GamesList;
}
```

Example 4

```
struct
{ char Name[80];
  char Rating;
  BOOL Played;
  long NumberOfKills;
} Data,DataList[90];
```

An Unnamed Struct Type

Sometimes we just want to create a structure and immediately use it without needing any instances later on in the code. For this reason, we can leave out the name field and create copies in our declaration. In this last example we leave the name field blank, but create an instance called Data, then create an array of structures called DataList with 90 items. We have those instances to work with, but should we need more copies later on in the code, it will not be possible since we didn't fill in our name field.

Accessing Members

Since we have an instance to work with, we can use the period to access the members of a given instance. We are allowed to access members because we created the instance Unreal in our definition using Example 3. We set each member to whatever we want, gaining access to it through our instance Unreal and using the period.

Below that we declare a second instance called Quake. Notice that since we named our structure Games, we were able to create another copy. We access the members and set them the same as we did with Unreal.

Lastly we are accessing our dynamically allocated array of structures. Notice first that in order to access a certain element in the array we use the same type of indexing as arrays, by using brackets and index number, starting with 0 as the first element. Also notice that instead of using the period to access members, we use the arrow. This is because we are dynamically allocating the array. If we were going to access Blizzard as in Example 3 we would use the period since it wasn't dynamically allocated, it was created during compile time. Now that we have a pretty good idea what structures are, how to declare them, and how to access them, let's cover some operations we can do.

```
void main()
{ Unreal.Rating=1;
  strcpy(Unreal.Name,"Unreal");
  Unreal.Played=True;
  Unreal.NumberOfKills=100;

  Games Quake;
  Quake.Rating=2;
  strcpy(Quake.Name,"Quake");
  Quake.Played=True;
  Quake.NumberOfKills=100;

  GamesList[0]->Rating=1;
  strcpy(GamesList[0]->Name, "Warcraft2");
 Gamesist[0]->Played = False;
 GamesList[0]->NumberOfKills = 0;
}
```

Operations

Notice in our above example that we can directly access and manipulate individual members of our structure using a period or arrow. When we deal with multiple instances of the same type of class, type being of the type filled in the name field, we can do aggregate operations.

Here we are using an aggregate operation of equality. Aggregate means that the operation acts on the structure as a whole whereas normal operations deal with individual members. We are able to use the equality operator because both structures are instances of the same structure type (Games). Here's an example of an aggregate operation that is illegal.

```
void main()
{ Games game1,game2;

  game1.Played=1;
  strcpy(game1.Name,"Roadwarriors");
  game1.Rating=4;
  game2=game1;
}
```

Aggregate Equality Operation

```
struct Games
{ BOOL LikedIt;
}game1;

struct OtherGames
{ BOOL LikedIt;
};

void main()
{ Games game2;
  OtherGames game3;

    game1.LikedIt=1;
  game2=game1;                    Illegal
  game3=game2;                    Illegal
  game3.LikedIt=game2.LikedIt;    Legal
}
```

Types MUST be the same!

Here we have two structures that look exactly alike, except they are named two deferent types. Since game2 and game1 are of the same type (Games), we are allowed to set them equal to each other. Now notice that we are not allowed to set game3 equal to game1 or game2!! We are then forced to set each member separately. Even if members are exactly the same, the types MUST be the same, so this example of aggregate equality is illegal!

Thanks for reading. Structures are a very useful means of gathering heterogeneous data under one common tag. I hope you feel like you actually learned something, instead of having a dazed look on your face! If you have any questions, comments, rude remarks please give me some feedback!

Contact Information

I just wanted to mention that everything here is copyrighted, feel free to distribute this document to anyone you want, just don't modify it! You can get a hold of me through my website or direct email. Please feel free to email me about anything. I can't guarantee that I'll be of ANY help, but I'll sure give it a try :-)

Email : Justin Deltener <deltener@mindtremors.com>
Webpage : www.inversereality.org