Lecture 15.2

Topics:

1. Derived Data Type – `struct`

---

1. Derived Data Type – `struct`

There are times when a single variable may not be sufficient or able to represent meaningful data. For examples, a fraction is defined with a numerator and a denominator both of integer of type **`int`**; one can use two variables of type **`int`** to represent a single fraction.

However, it is cumbersome to carry around this pair of variables at all time or to name these variables as **`num1`** and **`denom1`** to refer to just one fraction. It will become harder when many fractions are declared because the number of variables and notational issues are getting in the way of the solution logic.

1.1 `struct` – Definition

A new data type is derived from existing types can be used to accomplish the above task. This new derived data type is called **`struct`** (structure). A **`struct`** can be created by using template. This template or data type is used to declare one or more variables of this **`struct`** type.

Structure Type – `struct`

A general form of a struct in C is defined as follows,

```
struct StructName {
  DataType1 varType1;
  DataType2 varType2;
  DataType3 varType3;
  …
};
```

where

- `struct` is the keyword,
- `StructName` is the user's defined name for the struct type. It can be any name and should start with an uppercase character,
- `DataType1, DataType2, DataType3` are the existing data types,
- `varType1, varType2, varType3` are the names of the members of the `struct`. These are variable names, and should start with lowercase characters.

Example

```
struct Fraction {
  int iNum;
  int iDenom
};

struct Sample {
  int iFirst;
```

```
  char cSecond;
};
```

The above statement defines a new ***data type*** called ***`struct Sample`***. Each variable of this type consists of two elements: an integer variable called `iFirst` and a character variable called `cSecond`.

At this juncture, *there is no variable declared; the system does not set aside any memory for storage yet*! This statement basically tells the compiler what the `struct Sample` looks like and in what way it should convey the data template for this structure.

The keyword **`struct`** introduces the statement. The name `Sample` is a **tag** and it names the kind of structure being defined. This tag is not a ***variable name***, since we are not declaring a variable; it is only a ***type name***.

Thus, in general, ***`a structure is a data type`*** whose format is defined by the programmer.

1.2 Declaring Structure Variable

Using the data type struct sample, we can declare one or more variables to be of that type. For examples,

```
struct Sample sample1;
struct Sample sample2,sample3;
```

Or in some cases, combined declarations can be done as follows,

```
struct Sample {
  int iFirst;
  char cSecond;
} sample2,sample3;
```

This way of declaring structure variables is not as clear but may be used.

Note that as soon as each of these statements is executed, the corresponding memory spaces are set aside. This memory provides enough space to hold all items in the structure; in the above example, four (4) bytes for the integer and one (1) byte for the character. (In some situation, the compiler may allocate more bytes so that the next variable will come out on the even address).

1.3 Accessing Structure Elements

Structure uses the dot operator ("**.**" or membership operator) to refer to its elements. For examples, **`sample1.iFirst`** refers to the first element of the variable **`sample1`**, which is of **`struct Sample`** type. The variable name preceding the **dot** is the structure name while the name following it is the specific element in the structure.

Structures can be initialized or assigned values.

- *Initialization*

```
struct Sample sample1 = { 123, 'a' };
```

- *Assigning Values*

```
sample1.iFirst = 123;
sample1.cSecond = 'a';
```

- *Structure Assignment*

The value of one structure variable can be assigned to another structure variable of the same type. For example, the following statement is a valid one

```
sample1 = sample2;      /*Each element of sample1
                          will have value of the
                          corresponding one in
                          sample2*/
```

Each element of a structure variable must be treated as proper and appropriate single variable case. That means the address, value can be referred to. For examples,

`&sample1`	is the address of the structure.
`&sample1.iFirst`	denotes the address of the `iFirst` element in the struct `sample1`. While `sample1.iFirst` denotes its value.

1.4 Nested (Contained) Structures

Example

```
struct NestedSample         /*Defines structure type*/
{
  struct Sample sample4;    /*Structure within structure*/
  int iThird;           /*Regular integer variable*/
};
struct NestedSample nest1;       /*Declares structture
                                   variable*/
struct NestedSample nest2 = { { 123, 'a' }, 456 };
                    /*Declares & initializes structure
                      variable nest2 */
```

To access the nested structure elements, the dot (.) operators must be used according to the level of nesting. For examples,

```
nest2.sample4.iFirst
```

refers to the element `intFirst` in structure `sample4` in structure `nest2`.

1.5. Array of Structures

(i) Defining

```
struct Sample sampleArray[2];
```

(ii) Accessing Array Element

```
&sampleArray[0]             /*is the address of the
                              first structure.*/
sampleArray[0].iFirst       /*is the value of
                              intFirst in the
                                  first structure.*/
&sampleArray[0].iFirst      /*is the address of
                               ----- '' ----- */
```

(iii) Initializing

```
struct Sample sampleArray1[2] = { { 1,'a' },
                                  { 2,'b' } };
```

1.6 Pointers to Structures

(i) Defining

```
struct Sample* samplePtr;        /*Declaring*/
```

(ii) Initializing

```
samplePtr = &sample1;       /*Initializing or assigning*/
```

(iii) Accessing

```
samplePtr->iFirst;           /*Link operator*/
(*samplePtr).iFirst;         /*Pointer operator*/
samplePtr->cSecond;          /*Likewise*/
(*samplePtr).cSecond;        /*Likewise*/
```

Example 4

```
/**
 *Program Name:  cis26L1524.c
 *Discussion:    Pointers to struct
 */
#include <stdio.h>
#define LEN 20

struct Name {
  char cFirst[ LEN ];
  char cLast[ LEN ];
};

struct Person {
  struct Name name;
  char cFavFood[ LEN ];
  int iAge;
};

int main( ) {
  struct Person per[ 2 ] = { { { "Egbert", "Snivley" },
                               "eggplant",
                               20 },
                             { { "Rodney", "Swillbelly" },
                               "salmon mousse",
                               21 } };

  struct Person* him;    /* here is a pointer to a structure */
  struct Person* her,*them;

  printf( "Addresses: &per[ 0 ]: %p \t&per[ 1 ]: %p\n",
          &per[ 0 ], &per[ 1 ] );

  him = &per[ 0 ];    /* tell the pointer where to point  */
  her = &per[ 1 ];
  them = her;

  printf( "Addr. of ptr. variable him is %p; Its content is %p\n",
          &him, him );
  printf( "Addr. of ptr. variable her is %p; Its content is %p\n",
          &her, her );
  printf( "Addr. of ptr. variable them is %p; Its content is %p\n",
          &them, them );
```

```
    printf( "\nPointers --- ( him ) : %p\t( him + 1 ): %p\n",
            him, him + 1 );
    printf( "Address of per[ 0 ].name.cFirst is %p\n",
            per[ 0 ].name.cFirst );
    printf( "Address of per[ 0 ].name.cLast is %p\n",
            per[ 0 ].name.cLast );
    printf( "Address of per[ 0 ].name is %p\n",
            per[ 0 ].name );
    printf( "Address of per[ 0 ].iAge is %p\n",
            &per[ 0 ].iAge );

    printf( "Address of per[ 1 ].name.cFirst is %p\n",
            &per[ 1 ].name.cFirst );
    printf( "Address of per[ 1 ].name.cLast is %p\n",
            &per[ 1 ].name.cLast );
    printf( "Address of per[ 1 ].name is %p\n",
            per[ 1 ].name );
    printf( "Address of per[ 1 ].iAge is %p\n",
            &per[ 1 ].iAge );

    printf( "\nhim->iAge is %d: ( *him ).iAge is %d\n",
            him->iAge, ( *him ).iAge );

    printf( "After incrementing him++, then\n" );

    him++;  /* point to the next structure */

    printf( "him->cFavFood is %s:  him->name.cLast is %s\n",
            him->cFavFood, him->name.cLast );

    printf( "\nEnter any key to continue ... " );
    getchar( );

    return 0;
  }
```

OUTPUT

```
Addresses: &per[ 0 ]: 0012FF00  &per[ 1 ]: 0012FF40
Addr. of ptr. variable him is 0012FEFC; Its content is 0012FF00
Addr. of ptr. variable her is 0012FEF8; Its content is 0012FF40
Addr. of ptr. variable them is 0012FEF4; Its content is 0012FF40

Pointers --- ( him ) : 0012FF00 ( him + 1 ): 0012FF40
Address of per[ 0 ].name.cFirst is 0012FF00
Address of per[ 0 ].name.cLast is 0012FF14
Address of per[ 0 ].name is 65626745
Address of per[ 0 ].iAge is 0012FF3C
Address of per[ 1 ].name.cFirst is 0012FF40
Address of per[ 1 ].name.cLast is 0012FF54
Address of per[ 1 ].name is 6E646F52
Address of per[ 1 ].iAge is 0012FF7C

him->iAge is 20: ( *him ).iAge is 20
After incrementing him++, then
him->cFavFood is salmon mousse:  him->name.cLast is Swillbelly

Enter any key to continue ... c
```

1.7 Array of Pointers

```
struct Sample* arrayPtr[2];
```

(i) Initializing

```
arrayPtr[0] = &sampleArray[0];    /*Address value*/
arrayPtr[1] = &sampleArray[1];
```

(ii) Accessing

```
arrayPtr[0]->intFirst;            /*Integer value*/
arrayPtr[1]->chSecond;            /*Character value*/
```