STRUCT, UNION AND ENUM

2401 NOTES

By:
THE HOMIES

Structures

A structure allows you to wrap one or more variables that may be in different data types into one. It can contain any valid data type like int ,char, float, array, pointer or even structures. Each variable in the structure is called a structure member.

Defining a structure

To define a structure, you use the *struct* keyword.

```
struct struct_name{ structure_member };
```

The following example defines the *person* structure:

```
struct person{
      char first[32]; //first field an array of chars
      char last[32];  //second field an array of chars
      int age;        //third field an int
    };
```

Note that the above code is only defining a *person* structure. There are two ways to declare a structure variable:

Declaring a structure

You can declare structure variables together with the structure defintion:

```
struct struct_name {
      structure_member;
      ...
    } instance_1,instance_2 instance_n;
```

Or, you can declare the structure variable after you define the structure:

```
struct struct_name instance_1,instance_2 instance_n;
```

Using typedef

For more concise code, you can use the *typedef* keyword to create a synonym for a structure. So when you want to create a strucuture variable, you can omit the keyword *struct*.

```
typedef struct{
      char first[32]; //first field an array of chars
      char last[32];  //second field an array of chars
      int age;        //third field an int
    }person;

person student;
person teacher;
```

Accessing a structure member

To access a structure member, we use the dot operator ‘ . ‘ between the structure name and member:

```
structure_name.structure_member
```

The following example demonstrates how to access the first name of structure *person*:

```
person  student;
student.first = "Richard";
```

We can also use the dot operator to access a *nested structure* and its members:

```
typedef struct{
      person teacher[0];
      person students[50];
    }course;


course COMP2401;

COMP2401.students[0].first = "Richard";
```

Advantages of structures

Having a label that refers to the entire structure is convenient for two reasons.

First, it allows assignment statements between structure variables. This allows you to copy data from one structure to another:

```
struct_intance1 = struct_intance2
```

Second, is passing a structure as a parameter to a function:

```
displayStudent(person info){
    printf(“ %s %s: %d”, info.first info.last, info.age);
}

main()
{
person student;

displayStudent(student
```

Arrays and Structures

Just like you can have an array of any data type, you can have an array of structs.

```
typedef struct{
      char first[32]; //first field an array of chars
      char last[32];  //second field an array of chars
      int age;        //third field an int
    }person;

person students[60]; //array of struct person

strcpy(students[0].first, "Richard");
strcpy(students[0].last, "Ison");
students[0].age = 19;
```

Pointers and Structures

The address of a structure variable can be stored in a pointer variable, just like the address of any other type of variable.

```
struct fraction {
      int x, y;
    };

struct fraction f[3], *g;

f[0].x=3;
f[0].y = 5;

g = &(f[0]);
```

There are two syntaxes to access the bytes of the structure using the pointer variable.

```
(*g).x = 5;
g->y=11; //preferred syntax
```

A pointer variable for a structure can also be used to dynamically allocate memory.

```
g = (struct fraction *) malloc (sizeof(struct fraction));
```

Union

Definition

A union is a type of structure that can be used where the amount of memory used is a key factor. Similarly to the structure the union can contain different types of data types. Each time a new variable is initialized from the union it overwrites the previous and uses that memory location. This is most useful when the type of data being passed through functions is unknown, using a union which contains all possible data types can remedy this problem.

Defining a Union

```
union person {
       char name;
       int age;
       double height;
};
```

Declaring a Union

Since the union is still a type of structure the method for declaring it is as follows:

```
union person {
       char name;
       int age;
       double height;
}newPerson;
```

Or

```
Person newPerson;
```

Initializing a Union

The process of initializing variables in a union is the same as a union but the results are what make unions unique. The dot operator is still used in order to reach the data types inside of the union.

```
Person newPerson;
newPerson.age = 20;
newPerson.height = 6.2;
```

Once the height variable is initialized the variable age is overwritten and no longer exists.

ENUM

Defining an Enumeration

An enumeration provides the data type with a set of values. An enumeration constant is a type of an integer. A variable type takes and stores the values of the enumeration set defined by that type. Enumerations can be used with indexing expressions also as operands with arithmetic and relational operators. Enumerations can be also be used as an alternate way to use “#define”.

Declaring an Enumeration

There are two different types of enumerations declarations:

Creating a named type:

```
enum exampletype {THIS, IS, A, TEST};
```

Creating an unnamed type:

```
enum {THIS, IS, ANOTHER, TEST};
```

Both examples work, it simply comes down to preference, and which one you seem to work faster with.

Example of an Enumeration

```
enum Django {<- defines an enumeration type
    One, Two, Ps3, Four, Five} stuff; <- Variable stuff created to work with
enum Django gamestation = Ps3; <- gamestation is now assigned to the Django set, Ps3.
```

Advantages of Enumerations

There are some advantages to enumerations:

1. Numeric values are automatically assigned
2. It allows your code to become generally understandable by others or yourself
3. Enums allow certain debuggers to print the values of an enumeration constant