

DAILY ASSESSMENT

Date:	19-06-2020	Name:	Yamunashree N
Course:	C Programming	USN:	4AL17EC097
Topic:	Module 5: Structures & Unions	Semester & Section:	6 TH SEM & 'B' Section
Github Repository:	yamunashree-course		

SESSION DETAILS

Structures & Unions
Structures XP 39

1/5

Structures

A **structure** is a **user-defined data type** that groups related variables of different data types.

A **structure declaration** includes the keyword **struct**, a **structure tag** for referencing the **structure**, and curly braces { } with a list of variable declarations called **members**.

For example:

```
struct course {  
    int id;  
    char title[40];  
    float hours;  
};
```

This struct statement defines a new data type named **course** that has three members. Structure members can be of any data type, including basic types, strings, arrays, pointers, and even other structures, as you will learn in a later lesson.

Do not forget to put a semicolon after **structure** declaration.
A **structure** is also called a **composite** or **aggregate** data type. Some languages refer to structures as **records**.



Report

Structures

A structure is a user-defined data type that groups related variables of different data types.

A structure declaration includes the keyword `struct`, a structure tag for referencing the structure, and

curly braces `{ }` with a list of variable declarations called members.

For example:

```
struct course {  
    int id;  
    char title[40];  
    float hours;  
};
```

Structures with Structures

The members of a structure may also be structures.

For example, consider the following statements:

```
typedef struct {  
    int x;  
    int y;  
} point;  
typedef struct {  
    float radius;  
    point center;  
} circle;
```



Unions

A union allows to store different data types in the same memory location.

It is like a structure because it has members. However, a union variable uses the same memory

location for all its member's and only one member at a time can occupy the memory location.

A union declaration uses the keyword union, a union tag, and curly braces { } with a list of members.

Union members can be of any data type, including basic types, strings, arrays, pointers, and

structures.

For example:

```
union val {  
    int int_num;  
    float fl_num;  
    char str[20];  
};
```

Pointers to Unions

A pointer to a union points to the memory location allocated to the union.

A union pointer is declared by using the keyword union and the union tag along with * and the

pointer name.

For example, consider the following statements:

```
union val {  
    int int_num;  
    float fl_num;  
    char str[20];  
};  
union val info;
```



```
union val *ptr = NULL;
ptr = &info;
ptr->int_num = 10;
printf("info.int_num is %d", info.int_num);
```

Unions as Function Parameters

A function can have union parameters that accept arguments by value when a copy of the union

variable is all that is needed.

For a function to change the actual value in a union memory location, pointer parameters are required.

For example:


```
union id {
    int id_num;
    char name[20];
};
void set_id(union id *item) {
    item->id_num = 42;
}
void show_id(union id item) {
    printf("ID is %d", item.id_num);
}
```



DAILY ASSESSMENT

Date:	19-06-2020	Name:	Yamunashree N
Course:	C Programming	USN:	4AL17EC097
Topic:	Module 6: Memory Management	Semester & Section:	6 TH SEM & 'B' Section

SESSION DETAILS

Memory Management
Working With Memory

XP 39

▶ ? ▶ ? 1/2

Memory Management

Understanding memory is an important aspect of C programming. When you declare a variable using a basic data type, C automatically allocates space for the variable in an area of memory called the **stack**.

An **int** variable, for example, is typically allocated 4 bytes when declared. We know this by using the **sizeof** operator:

```
int x;  
printf("%d", sizeof(x)); /* output: 4 */
```

Try It Yourself

As another example, an **array** with a specified size is allocated **contiguous blocks** of memory with each block the size for one element:

```
int arr[10];  
printf("%d", sizeof(arr)); /* output: 40 */
```



Report

Memory Management in C

When the compiler encounters a sequence of characters enclosed in the double quotation marks, it appends a null character `\0` at the end by default. One of the things that makes C such a versatile language is that the programmer can scale down a program to run with a very small amount of memory.

When C was first written, this was an important feature because computers weren't nearly as powerful as they are today. With the current demand for small electronics, from mobile phones to tiny medical devices, there's a renewed interest in keeping the memory requirements small for some software. C is the go-to language for most programmers who need a lot of control over memory usage.

To better understand the importance of memory management, consider how a program uses memory. When you first run a program, it loads into your computer's memory and begins to execute by sending and receiving instructions from the computer's processor. When the program needs to run a particular function, it loads that function into yet another part of memory for the duration of its run, then abandons that memory when the function is complete. Plus, each new piece of data used in the main program takes up memory for the duration of the program.

There are two ways in which memory can be allocated in C:

- by declaring variables
- by explicitly requesting space from C

We have discussed variable declaration in other lectures, but here we will describe requesting dynamic memory allocation and memory management.

C provides several functions for memory allocation and management:

- `malloc` and `calloc`, to reserve space
- `realloc`, to move a reserved block of memory to another allocation of different dimensions
- `free`, to release space back to C

These functions can be found in the `stdlib` library

What happens when a pointer is declared?



Whenever a pointer is declared, all that happens is that C allocates space for the pointer.

For example,

```
char *p;
```

allocates 4 consecutive bytes in memory which are associated with the variable p. p's type is declared to

be of pointer to char. However, the memory location occupied by p is not initialised, so it may contain

garbage.

It is often a good idea to initialise the pointer at the time it is declared, to reduce the chances of a

random value in p to be used as a memory address:

```
char *p = NULL;
```

At some stage during your program you may wish p to point to the location of some string A common

error is to simply copy the required string into p: `strcpy(p, "Hello");`

Often, this will result in a "Segmentation Fault". Worse yet, the copy may actually succeed.

```
//a.c
```

```
#include <stdio.h>
```

```
main()
```

```
{
```

```
    char *p;
```

```
    char *q = NULL;
```

```
    printf("Address of p = %u\n", p);
```

```
    strcpy(p, "Hello");
```

```
    printf("%s\n", p);
```

```
    printf("About to copy \"Goodbye\" to q\n");
```

```
    strcpy(q, "Goodbye");
```

```
    printf("String copied\n");
```



```
printf("%s\n", q);  
}
```

When p and q are declared, their memory locations contain garbage. However, the garbage value in p

happens to correspond to a memory location that is not write protected by another process. So the strcpy

is permitted. By initialising q to NULL, we are ensuring that we cannot use q incorrectly. Trying to

copy the string "Goodbye" into location 0 (NULL) results in a run-time Bus Error, and a program crash.

