**Structures**

Table of Contents

[Dot Operator 3](#_Toc65505796)

[Nested Structure 4](#_Toc65505797)

[Bit-Fields 5](#_Toc65505798)

[Union 6](#_Toc65505799)

A structure is an aggregate (or conglomerate) data type, that is composed of two or more related variables called members.

struct varName  
{  
 type var1;  
 type var2;  
} x, y;

C

struct date *//4+13+4 = 21 bytes*{  
 int day; *//4 bytes* char month[13]; *//13 bytes* int year; *//4 bytes*} d1, d2;  
*//d1 and d2 are vars  
// that can contain the variables in the structure date d3, d4[100];  
// more vars can be declared*

C

The structure does not need a name but not giving it a name means more variables cannot be assigned to it in the future.

## **Dot Operator**

The dot operator can be used to assign values to the variables inside a structure.

d3.day = 19;  
d3.month[0] = 'a';  
  
scanf("%d %s %d", &d1.day, &d1.month, &d1.year);  
*//d1 contains all three variables*date \*p;  
p = &d1; *//\*p points to the data in date*p -> day; *// -> is called an arrow operator*p -> month; *// month data type can be accessed like this*

C

Trees [We will get back to this later]

1

2

3

4

5

1 is called the root. It is also the parent of 2 and 3. 2 and 3 are called the children.

struct node  
{  
 int data;  
 node \*child1; *//points to 4 and 5 node \*child2;*}

C

Nested Structure

struct date  
{  
 int day;  
 char month[13];  
 int year;  
}  
  
struct student  
{  
 char name[100];  
 int id;  
 date birth; *//a structure that contains the date of birth*}CoolBoy  
  
CoolBoy.birth.day; *//gives access to day inside birth inside student*

C

Bit-Fields

Code can be optimized. For example, simply declaring int day gives that integer 216 bits of data, but only 5 bits are required to store that maximum day (31).

struct date  
{  
 unsigned day: 5; *//maximum 5 bits of memory can be used unsigned month: 4;* unsigned year: 12;  
}

C

This results in just 21 bits of memory being needed now, while originally 168 bits were given.

Union

Unions store data in a certain memory space. This data can be interpreted as one of the declared data types. The union takes the memory of the largest data type.

union month  
{  
 int m1;  
 char m2[13];  
}

C

Using struct, this would use 17 bytes of memory, but union uses 13 bytes.

Example of Usage

#include <stdio.h>  
#include <string.h>  
  
union month  
{  
 int num;  
 char name[13];  
};  
  
struct structure  
{  
 union month mon;  
};  
  
int main()  
{  
 */\*  
 This code shows a real life use of unions.  
 It assumes that the type of data being used is unknown (like user input).  
 \*/* char months[12][13] = {"January", "February", "March"};  
 struct structure b = {01};  
 */\*  
 If the data is interpreted as an integer (a.mon.num)  
 and has a value less than 13, print the integer,  
 the array member corresponding to that integer  
 and the string form of the data (which won’t be sensible characters).  
 Otherwise, if the data has a value greater than 13  
 when being interpreted as an integer, it must not be an integer at all.  
 Then print the data as a string  
 and then print the integer value of that string.  
 \*/* if(b.mon.num < 13)  
 printf("Integer: %s %s", months[b.mon.num], b.mon.name);  
 else printf("String: %s %d", b.mon, b.mon);  
}

C