Structures and Unions in C

Lecture 18

Objective

Be able to use compound data structures in programs.

Be able to pass compound data structures as function arguments, either by value or by reference.

Structures

Arrays require that all elements be of the same data type.

Many times it is necessary to group information of different data types

An example is a materials list for a product (list typically includes a name for each item, a part number, dimensions, weight, and cost)

C supports data structures that can store combinations of character, integer floating point and enumerated type data

They are called a struct.

Arrays of Structures

```
#include <stdio.h>

typedef struct {

        int month;

        int day;

        int year;

    } Date;
```

```
Date birthdays[NFRIENDS];

int check_birthday(Date birthdays[], int N, Date today)

{

    int i;

    for (i = 0; i < N; i++) {

        if ((today.month == birthdays[i].month) && (today.day == birthdays[i].day))

            return i;

    }

    return -1;

}
```

Pointers to Structures

- Pointers to structures are crucial for dynamic memory management and handling large data efficiently.
- Use the arrow operator (->) to access members of the structure through a pointer.

```
struct Person {

  char name[50];

  int age;

};

struct Person *ptr;

ptr->name = "John";

ptr->age = 30;
```

Passing Structures by Reference

Saves memory and time as copies of structures are not created.

```
void displayPerson(const struct Person *person) {
   printf("Name: %s\n", person->name);
   printf("Age: %d\n", person->age);
}
```

Pass by Value vs Pass by Reference

```
Date create_date1(Date d, int month, int day, int year)

{

    d.month = month;

    d.day = day;

    d.year = year;

    return (d);

}

int main(){
    Date p = {1,10,2015};
    p = change_date(p,1, 18, 2018);
    printf("%d %d %d\n", p.month, p.day, p.year);
}
```

```
void create_date2(Date *d, int month, int day, int year)

{

    d->month = month;

    d->day = day;

    d->year = year;

}

int main(){
    Date p;
    create_date2(&p, 1, 18, 2018);
}
```

Limitation of Structure

Suppose we have a requirement to create a new type color to store color of an object. It should accept color input in all formats. We end up with following structure definition.

```
struct color
{
    struct color_rgb  rgb;   // rgb color value
    struct color_rgba rgba;  // rgba color value
    unsigned int  value;     // hexadecimal or integer value
    char name[20];           // Unique string representing color name
};
```

Limitation of Structure

The above structure approach has one big problem.

At any time one object will store one color value.

Hence, you will either use rgb, rgba or value or name to store color.

In no case you will you two different formats of color.

But, still we are wasting memory for all members of color structure unknowingly.

Introducing Union

To overcome such type of real world problems we use union.

Unlike structures union occupies **single memory location** to store all its members.

So, unions are helpful when you want to store value in single member from a set of members.So you only need a single variable to store data of different types at different times.

Size of union is defined according to size of largest member data type.

They are efficient in terms of memory usage because they share the same memory for all fields.

For example, if there are three members (char, short and int) in union, the size of union will be sizeof(int).

Syntax of Union

```
union union_name
{
   union_member1;
   union_member2;
   ...
   ...
   ...
   union_member_N;
};
```

Example of Unions in C

```
union Data {

   int i;

   float f;

   char str[20];

};
```

Declaration along with union definition

You can declare a union variable right after union definition. The union definition must be followed by union variable declaration.

```
union color
{
    struct color_rgb  rgb;     // color in rgb
    struct color_rgba rgba;   // color in rgba
    unsigned int value;        // hexadecimal or decimal color value
    char name[20];            // unique string color name
} console_color;
```

Declaration after union definition

This approach to declare union variable sets you free to declare union variable anywhere in program. You are not only restricted to declare union along with its definition.

```
union color

{

    struct color_rgb  rgb;    // color in rgb

    struct color_rgba rgba;   // color in rgba

    unsigned int value;       // hexadecimal or decimal color value

    char name[20];            // unique string color name

};

union color console_color;
```

How to access union members?

Dot/period operator in C: We use dot operator to access union members.

```
console_color.value = 999;
```

Arrow operator in C: We use arrow operator to access members of a pointer to structure or union.

```
console_color->value = 999;
```

Unions (for inspecting bytes of int)

```
#include <stdio.h>
union intChar {
   int i;
   unsigned char c[4];
} n;

int main(){
   scanf("%d",&n.i);
   printf("%d %d %d %d",n.c[0],n.c[1],n.c[2],n.c[3]);
}
```

Input : 2147483647

Output: 255 255 255 127

Comparing Structures and Unions

- Memory Allocation:
  - Structure: Memory size is the sum of all member sizes.
  - Union: Memory size is the size of the largest member.
- Usage Context:
  - Structure: When you want to store multiple different values.
  - Union: When you want to use the same memory for different types.

- Structures: members are “and”ed together
- Unions: members are “xor”ed together

Example

https://docs.google.com/document/d/1OmXodTCkjfa-doxdM4FEawXxgeMdSZe73TpH4iIGslg/edit?usp=sharing

https://www.javatpoint.com/c-union

Constants

Allow consistent use of the same constant throughout the program

- Improves clarity of the program
- Reduces likelihood of simple errors
- Easier to update constants in the program

Constants - Define once, use throughout the program

```
int array[10]; for (i=0; i<10; i++) {

…

}
```

$\longrightarrow$

```
#define SIZE 10 //Preprocessors directive

int array[SIZE]; //Constants are capitalized by convention

for (i=0; i<SIZE; i++) {

…..

}
```

Homework:
bit-field