Pointers

Address in C

Before you get into the concept of pointers, let's first get familiar with address in C.

If you have a variable *var* in your program, `&var` will give you its address in the memory, where `&` is commonly called the reference operator.

You must have seen this notation while using `scanf()` function. It was used in the function to store the user inputted value in the address of *var*.

```
scanf("%d", &var);
```

You may obtain different value of address while using this code.

In above source code, value 5 is stored in the memory location 2686778. *var* is just the name given to that location.

Pointer variables

In C, you can create a special variable that stores the address (rather than the value). This variable is called pointer variable or simply a pointer.

How to create a pointer variable?

```
data_type* pointer_variable_name;
int* p;
```

Above statement defines, *p* as pointer variable of type `int`.

Reference operator (&) and Dereference operator (*)

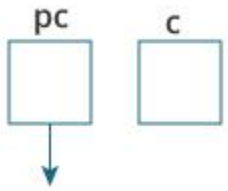
As discussed, `&` is called reference operator. It gives you the address of a variable.

Likewise, there is another operator that gets you the value from the address, it is called a dereference operator `*`.

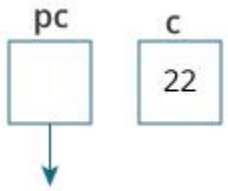
Below example clearly demonstrates the use of pointers, reference operator and dereference operator.

Note: The `*` sign when declaring a pointer is not a dereference operator. It is just a similar notation that creates a pointer.

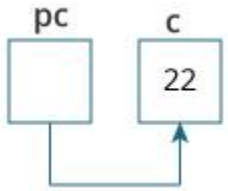
Explanation of the program



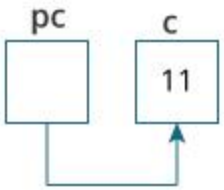
Here, a pointer *pc* and a normal variable *c*, both of type `int`, is created.
Since *pc* and *c* are not initialized at first, pointer *pc* points to either no address or a random address.
And, variable *c* has an address but contains a random garbage value.



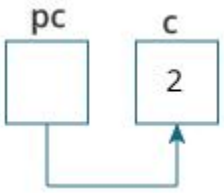
This assigns 22 to the variable *c*, i.e., 22 is stored in the memory location of variable *c*.
Note that, when printing `&c` (address of c), we use `%u` rather than `%d` since address is usually expressed as an unsigned integer (always positive).



This assigns the address of variable *c* to the pointer *pc*.
You see the value of *pc* is same as the address of *c* and the content of *pc* is 22 as well.



This assigns 11 to variable *c*.
Since, pointer *pc* points to the same address as *c*, value pointed by pointer *pc* is 11 as well.



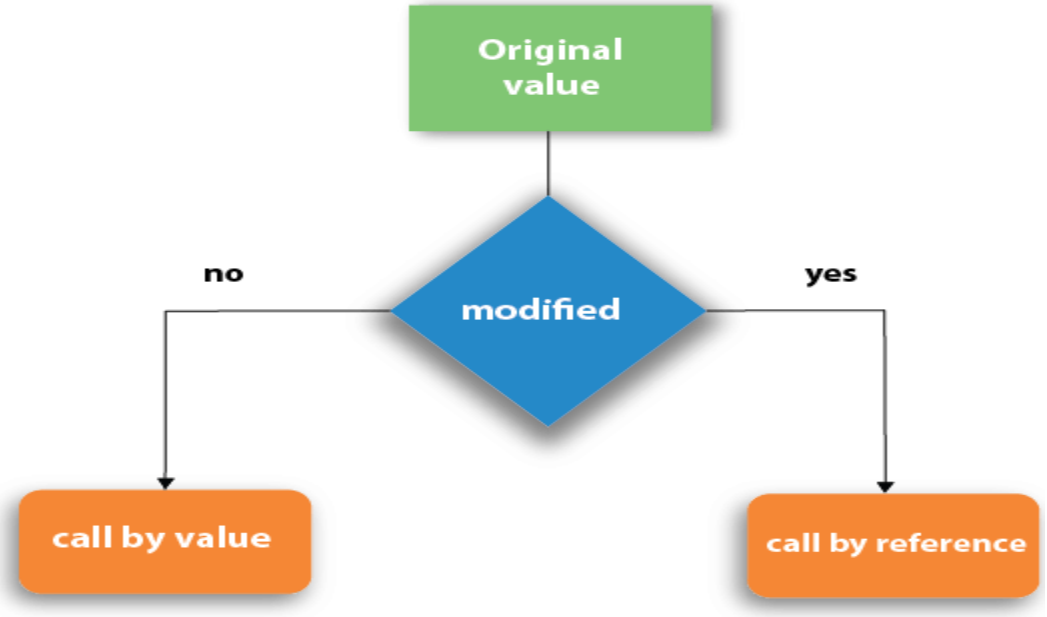
This change the value at the memory location pointed by pointer *pc* to 2.
Since the address of the pointer *pc* is same as the address of *c*, value of *c* is also changed to 2

Common mistakes when working with pointers

Suppose, you want pointer *pc* to point to the address of *c*. Then,

```
int c, *pc;

// Wrong! pc is address whereas,
// c is not an address.
pc = c;

// Wrong! *pc is the value pointed by address whereas,
// &c is an address.
*pc = &c;

// Correct! pc is an address and,
// &c is also an address.
pc = &c;

// Correct! *pc is the value pointed by address and,
// c is also a value (not address).
*pc = c;
```

Call by value and Call by reference in C



Call by value in C

- In call by value method, the value of the actual parameters is copied into the formal parameters. In other words, we can say that the value of the variable is used in the function call in the call by value method.
- In call by value method, we can not modify the value of the actual parameter by the formal parameter.
- In call by value, different memory is allocated for actual and formal parameters since the value of the actual parameter is copied into the formal parameter.
- The actual parameter is the argument which is used in the function call whereas formal parameter is the argument which is used in the function definition.

Call by Value Example: Swapping the values of the two variables????????

Call by reference in C

- In call by reference, the address of the variable is passed into the function call as the actual parameter.
- The value of the actual parameters can be modified by changing the formal parameters since the address of the actual parameters is passed.

- In call by reference, the memory allocation is similar for both formal parameters and actual parameters. All the operations in the function are performed on the value stored at the address of the actual parameters, and the modified value gets stored at the same address.

Call by reference Example: Swapping the values of the two variables?????

Pointers and Arrays

First check Program PA from pointer and Array folder

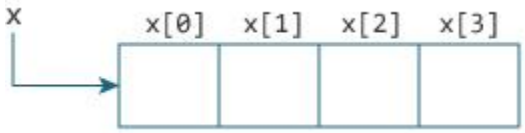
There is a difference of 4 bytes between two consecutive elements of array *x*. It is because the size of `int` is 4 bytes (on our compiler).

Notice that, printing *&x[0]* and *x* gave us the same result.

Relation between Arrays and Pointers

Consider an array:

```
int x[4];
```



From the above example, it's clear that *x* and `&x[0]` both contains the same address. Hence, `&x[0]` is equivalent to *x*.

And, `x[0]` is equivalent to `*x`.

Similarly,

- `&x[1]` is equivalent to `x+1` and `x[1]` is equivalent to `*(x+1)`.
- `&x[2]` is equivalent to `x+2` and `x[2]` is equivalent to `*(x+2)`.
- ...
- Basically, `&x[i]` is equivalent to `x+i` and `x[i]` is equivalent to `*(x+i)`.

PAA.c

In most contexts, array names "decays" to pointers. In simple words, array names are converted to pointers. That's the reason why you can use pointer with the same name as array to manipulate elements of the array. However, you should remember that **pointers and arrays are not same**

AAP.c

In this example, `&x[2]` (address of the third element of array *x*) is assigned to the pointer *ptr*. Hence, `3` was displayed when we printed `*ptr`.

And, printing `*ptr+1` gives us the fourth element. Similarly, printing `*ptr-1` gives us the second element.

Dynamic Memory Allocation

An array is a collection of fixed number of values of a single type. That is, you need to declare the size of an array before you can use it.

Sometimes, the size of array you declared may be insufficient. To solve this issue, you can allocate memory manually during run-time. This is known as dynamic memory allocation in C programming.

There are 4 library functions defined under `<stdlib.h>` makes dynamic memory allocation in C programming. **They are `malloc()`, `calloc()`, `realloc()` and `free()`.**

C malloc()

The name "malloc" stands for memory allocation.

The `malloc()` function reserves a block of memory of the specified number of bytes. And, it returns a pointer of type `void` which can be casted into pointer of any form.

Syntax of malloc()

```
ptr = (cast-type*) malloc(byte-size)
```

Example:

```
ptr = (int*) malloc(100 * sizeof(int));
```

Considering the size of `int` is 4 bytes, this statement allocates 400 bytes of memory. And, the pointer *ptr* holds the address of the first byte in the allocated memory.

However, if the space is insufficient, allocation fails and returns a NULL pointer.

C calloc()

The name "calloc" stands for contiguous allocation.

The `malloc()` function allocates a single block of memory. Whereas, `calloc()` allocates multiple blocks of memory and initializes them to zero.

Syntax of calloc()

```
ptr = (cast-type*)calloc(n, element-size);
```

Example:

```
ptr = (float*) calloc(25, sizeof(float));
```

This statement allocates contiguous space in memory for 25 elements each with the size of `float`.

C free()

Dynamically allocated memory created with either `calloc()` or `malloc()` doesn't get freed on their own. You must explicitly use `free()` to release the space.

Syntax of free()

```
free(ptr);
```

This statement frees the space allocated in the memory pointed by `ptr`.

---

realloc()

If the dynamically allocated memory is insufficient or more than required, you can change the size of previously allocated memory using `realloc()` function

---

Syntax of realloc()

```
ptr = realloc(ptr, x);
```

Here, *ptr* is reallocated with new size *x*.

Structure

Structure is a collection of variables (can be of different types) under a single name.

For example: You want to store information about a person: his/her name, citizenship number and salary. You can create different variables *name*, *citNo* and *salary* to store these information separately.

What if you need to store information of more than one person? Now, you need to create different variables for each information per person: *name1*, *citNo1*, *salary1*, *name2*, *citNo2*, *salary2* etc.

A better approach would be to have a collection of all related information under a single name `Person` structure, and use it for every person.

How to define a structure?

Keyword `struct` is used for creating a structure.

Syntax of structure

```
struct structure_name
{
    data_type member1;
    data_type member2;
    .
    .
    data_type memeber;
};
```

Here is an example:

```
struct Person
{
    char name[50];
    int citNo;
    float salary;
};
```

Here, a derived type `struct Person` is defined.

Create structure variable

When a structure is defined, it creates a user-defined type. However, no storage or memory is allocated. To allocate memory of a given structure type and work with it, we need to create variables.

Here's how we create structure variables:

```
struct Person
{
    char name[50];
    int citNo;
    float salary;
};

int main()
{
    struct Person person1, person2, p[20];
    return 0;
}
```

Another way of creating a structure variable is:

```
struct Person
{
    char name[50];
    int citNo;
    float salary;
} person1, person2, p[20];
```

In both cases, two variables *person1*, *person2*, and an array variable *p* having 20 elements of type **struct Person** are created.

How to Access members of a structure?

There are two types of operators used for accessing members of a structure.

1. Member operator(.)
2. Structure pointer operator(->) (will be discussed in structure and pointers)

Suppose, you want to access salary of *person2*. Here's how you can do it:

```
person2.salary
```

Keyword typedef

Keyword typedef can be used to simplify syntax of a structure.

`typedef` is a keyword used in C language to assign alternative names to existing datatypes. Its mostly used with user defined datatypes, when names of the datatypes become slightly complicated to use in programs. Following is the general syntax for using `typedef`,

```
typedef <existing_name> <alias_name>
```

Lets take an example and see how `typedef` actually works.

```
typedef unsigned long ulong;
```

The above statement define a term `ulong` for an `unsigned long` datatype. Now this `ulong` identifier can be used to define `unsigned long` type variables.

```
ulong i, j;
```

Application of typedef

`typedef` can be used to give a name to user defined data type as well. Lets see its use with structures.

```
typedef struct
{
    type member1;
    type member2;
    type member3;
} type_name;
```

Here **type_name** represents the stucture definition associated with it. Now this **type_name** can be used to declare a variable of this stucture type.

```
type_name t1, t2;
```

`typedef` and Pointers

`typedef` can be used to give an alias name to pointers also. Here we have a case in which use of `typedef` is beneficial during pointer declaration.

In Pointers `*` binds to the right and not on the left.

```
int* x, y;
```

By this declaration statement, we are actually declaring `x` as a pointer of type `int`, whereas `y` will be declared as a plain `int` variable.

```
typedef int* IntPtr;
IntPtr x, y, z;
```

But if we use `typedef` like we have used in the example above, we can declare any number of pointers in a single statement.

Nested Structures

You can create structures within a structure in C programming. For example:

```
struct complex
{
 int imag;
 float real;
};

struct number
{
   struct complex comp;
   int integers;
} num1, num2;
```

Suppose, you want to set *imag* of *num2* variable to 11. Here's how you can do it:

```
num2.comp.imag = 11;
```

Structure and Pointer

Structures can be accessed using pointers. Here's how:

```
struct name {
    member1;
    member2;
    .
    .
};

int main()
{
    struct name *ptr, Harry;
}
```

Here, a pointer *ptr* of type **`struct name`** is created. The pointer can access members of *Harry*.

How to Access structure members using pointer??????

n this example, the address of *person1* is stored in *personPtr* pointer variable using code `personPtr = &person1;`.

Now, you can access members of *person1* using `personPtr` pointer. For that we use `->` operator.

*********By the way,

- `personPtr->age` is equivalent to `(*personPtr).age`
- `personPtr->weight` is equivalent to `(*personPtr).weight`

Dynamic memory allocation of structures

Sometimes, the number of structure variables you declared may be insufficient. You may need to allocate memory during run-time.

strDMA.c

Structure and Function

Passing structure to a function

PassstrFun.c

Returning structure from a function

ReturnStrFun.c

Passing structure by reference

Strpassbyreff.c

Nested Structure in C

C provides us the feature of nesting one structure within another structure by using which, complex data types are created. For example, we may need to store the address of an entity employee in a structure. The attribute address may also have the subparts as street number, city, state, and pin code. Hence, to store the address of the employee, we need to store the address of the employee into a separate structure and nest the structure address into the structure employee. Consider the following program.

nSTR.c

The structure can be nested in the following ways

1. By separate structure
2. By Embedded structure

Separate structure

Here, we create two structures, but the dependent structure should be used inside the main structure as a member. Consider the following example.

SP.c

As you can see, doj (date of joining) is the variable of type Date. Here doj is used as a member in Employee structure. In this way, we can use Date structure in many structures.

Embedded structure

The embedded structure enables us to declare the structure inside the structure. Hence, it requires less line of codes but it can not be used in multiple data structures. Consider the following example.

EP.c

Accessing Nested Structure

We can access the member of the nested structure by Outer_Structure.Nested_Structure.member as given below:

e1.doj.dd

e1.doj.mm

e1.doj.yyyy

Unions

Union is a user-defined type similar to a structure in C programming

How to define a union?

We use `union` keyword to define unions. Here's an example:

```
union car
{
  char name[50];
  int price;
};
```

The above code defines a derived type `union car`

Create union variables

When a union is defined, it creates a user-defined type. However, no memory is allocated. To allocate memory for a given union type and work with it, we need to create variables.

Here's how we create union variables:

```
union car
{
  char name[50];
  int price;
};

int main()
{
  union car car1, car2, *car3;
  return 0;
}
```

Another way of creating union variables is:

```
union car
{
  char name[50];
  int price;
} car1, car2, *car3;
```

In both cases, union variables *car1*, *car2*, and a union pointer *car3* of **union car** type are created.

How to access members of a union?

We use `.` to access normal variables of a union. To access pointer variables, we use `->` operator.

In the above example,

- price for `car1` can be accessed using `car1.price`
- price for `car3` can be accessed using `car3->price`

Why this difference in size of union and structure variables?

The size of structure variable is 40 bytes. It's because:

- size of `name[32]` is 32 bytes
- size of `salary` is 4 bytes
- size of `workerNo` is 4 bytes

However, the size of union variable is 32 bytes. It's because the size of union variable will always be the size of its largest element. In the above example, the size of largest element (`name[32]`) is 32 byes.

Only one union member can be accessed at a time