Notes on Embedded C

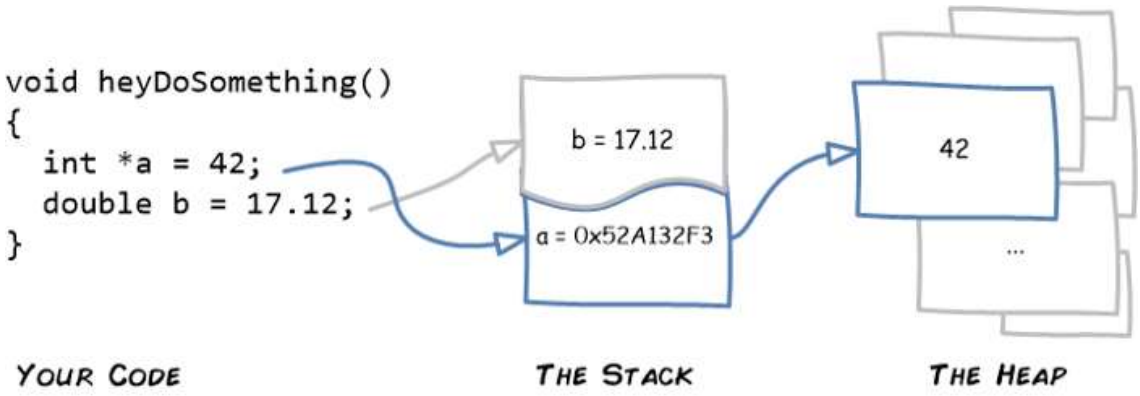
Edition 2 | Source: Internet

Contents

1. Pointing to the heap in C/C++

Some languages, like classic C, put you in charge of your program's memory management. If you want to put something on the heap, you need to speak up and put it there. You also need to keep track of your data using a *pointer*.

A pointer is a memory address. In C/C++, you use an asterisk * to create a pointer. Here's an example that defines one:



In this code, *a is an integer, like you'd expect (and it stores the value 42). But a (with no asterisk) is a pointer — a kind of numeric code that records the memory address where the value 42 is stored.

This example uses a pointer to put an ordinary number on the heap, but you can stuff arrays and big objects there too. If you need more space, the heap grows to fit your needs. It's slower and less efficient than the stack, but far more flexible.

When you need to find your data on the heap, you use the pointer. It's a bit like following a numeric code at the library to find the shelf that has the book you want. Or following a postal address to get to the stash house that has your goods.

2. Variadic functions

Variadic functions are functions that accept variable number of arguments. The type of the parameters is assumed beforehand and is hard coded. Below are two examples of the same.

Example 1

```
#include <stdarg.h>

#include <stdio.h>


double average(int count, ...)

{
```

```
    va_list ap;
    int j;
    double sum = 0;

    va_start(ap, count); /* Requires the last fixed parameter (to
get the address) */
    for (j = 0; j < count; j++)
    {
        sum += va_arg(ap, int); /* Increments ap to the next
argument. */
    }
    va_end(ap);

    return(sum / count);
}

int main(void)
{
    printf("%f\n", average(3, 8, 3, 1));
    printf("%f\n", average(5, 8, 2, 5, 7, 9));
    return(0);
}
```

Output:

```
4.000000
6.200000
```

Example 2

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

```
#include <stdarg.h>

/* print all args one at a time until a negative argument is seen;
   all args are assumed to be of int type */
void printargs(int arg1, ...)
{
    va_list ap;
    int i;

    va_start(ap, arg1);
    for (i = arg1; i >= 0; i = va_arg(ap, int))
        printf("%d ", i);
    va_end(ap);
    putchar('\n');
}

int main(void)
{
    printargs(5, 2, 14, 84, 97, 15, -1, 48, -1);
    printargs(84, 51, -1, 3);
    printargs(-1);
    printargs(1, -1);
    return(0);
}
```

Output:

```
5 2 14 84 97 15
84 51
```

1

3. Pointers and Local variables in functions

```
#include <stdio.h>


int *fun_1(void)

{

    static int x = 10;  //x is stored in the .data section of
memory. So it does not gets deleted even when the function returns.

    return(&x);

}


int *fun_2(void)

{

    int y = 20;  //y is stored in stack memory.

    int *a = &y;  //Pointers are used to store data on the heap
memory. But a is stored in stack memory.

    return(a);  //Returns the address and then a is destroyed.

}


int *fun_3(void)

{

    int z = 30;  //Local variables are created in stack and they
have scope. So, they are not accessible anywhere.

    return(&z);  // z is created on stack. So, it is destroyed when
the function returns. Therefore, &z is invalid address.

}


int main(void)

{
```

```
    int *p = fun_1();
    printf("%d\n", *p);  //Prints 10.

    p = fun_2();
    printf("%d\n", *p);  //Prints 20.

    p = fun_3();
    printf("%d\n", *p);  //Segmentation fault occurs and the program
terminates here.

    return(0);
}
```

Output:

```
10
20
<Segmentation fault>
```

4. Ternary operator

```
#include <stdio.h>

int main(void)
{
    int a, b, result;

    printf("Enter first number: ");
    scanf("%d", &a);
    printf("Enter second number: ");
    scanf("%d", &b);
```

```
    result = (a > b)? a : b;

    printf("Highest is: %d\n", result);


    (a > b)? printf("First number is highest") : printf("Second
number is highest");


    return(0);

}
```

Output:

```
Enter first number: 87

Enter second number: 765

Highest is: 765

Second number is highest
```

5. Terminating a program using exit() while inside a function without return type

```
#include <stdio.h>

#include <stdlib.h>


void foo(void)

{

    printf("Inside foo() before exit()\n");

    exit(1);  //Exiting with failure. exit(0) for exiting with
success.

    printf("Inside foo() after exit()\n");

}
```

```
int main(void)
{
    printf("Inside main() before foo()\n");
    foo();
    printf("Inside main() after foo()\n");
    return(0);
}
```

Output:

```
Inside main() before foo()
Inside foo() before exit()
```

6. Structure Packing and Padding

```
#include <stdio.h>

#pragma pack(push, 4)  //4-byte alignment.
typedef struct
{
      int x;
      char y;
      int z;
}packed_4;

#pragma pack(push, 1)  //1-byte alignment.
typedef struct
{
      char a;
      int b;
```

```
    packed_4 s1;  //4-byte alignment.
}nested_packed_1;


#pragma pack(pop)  //Pops out to previous alignment rule. ie,
pack(push, 4) in this case.


int main(void)
{
    printf("Size of int: %d\n", (size_t)sizeof(int));
    printf("Size of packed_4: %d\n", (size_t)sizeof(packed_4));
    printf("Size of nested_packed_1: %d\n",
(size_t)sizeof(nested_packed_1));


    return(0);
}
```

Output:

```
Size of int: 4
Size of packed_4: 12
Size of nested_packed_1: 17
```

7. Arguments and Options

```
#include <stdio.h>
#include <stdlib.h>  //For atof().
#include <string.h>  //For strcmp().


int main(int argc, char *argv[])
{
```

```
    if(argc == 2 && (strcmp(argv[1], "-h") || strcmp(argv[1], "--
help")))

    {

        printf("Usage: calc [option] [operand 1] [operand 2]\n");

        printf("-a, --add - Addition\n-s, --substract -
Substraction\n-m, --multiply - Multiplication\n-d, --divide -
Division\n");

        return(0);

    }

    else if(argc != 4)

    {

        printf("Usage: calc [option] [operand 1] [operand 2]\n");

        printf("-a, --add - Addition\n-s, --substract -
Substraction\n-m, --multiply - Multiplication\n-d, --divide -
Division\n");

        return(-1);

    }

    else

    {}


    if(strcmp(argv[1], "-a") == 0 || strcmp(argv[1], "--add") == 0)

    {

        printf("Addition is: %f\n", atof(argv[2]) + atof(argv[3]));

        return(0);

    }

    else if(strcmp(argv[1], "-s") == 0 || strcmp(argv[1], "--
substract") == 0)

    {

        printf("Substraction is: %f\n", atof(argv[2]) -
atof(argv[3]));

        return(0);

    }
```

```
    else if(strcmp(argv[1], "-m") == 0 || strcmp(argv[1], "--
multiply") == 0)

    {

        printf("Multiplication is: %f\n", atof(argv[2]) *
atof(argv[3]));

        return(0);

    }

    else if(strcmp(argv[1], "-d") == 0 || strcmp(argv[1], "--
divide") == 0)

    {

        printf("Division is: %f\n", atof(argv[2]) / atof(argv[3]));

        return(0);

    }

    else

    {

        printf("Bad option(s) or/and argument(s)\n");

        return(-1);

    }

}
```

Output:

```
>>./calc --add 2 3

Addition is: 5.000000
```

8. Memory Mapping an SFR

```
#define REGISTER_NAME (*(volatile unsigned char *) 0x1234)
```

More specifically,

```
*(volatile unsigned char *) 0x1234
```

is an integer constant `0x1234`, that is typecast to a pointer (an address) that points to a volatile unsigned char (a single byte), So, the data that is to be read or written to should be having a data type

of `volatile unsigned char`. The pointer has been dereferenced (by the star on the far left). This means that it is accessing (whether reading or writing) the value at the address `0x1234`. The `volatile` means that the value of the byte that is pointed to, can change due to something outside the main code, either through an Interrupt Service Routine (ISR), or more likely in this case, the hardware itself (because we are memory mapping a register).

```
(*(volatile unsigned char *) 0x1234)
```

is the same as above, but with extra parentheses around it to avoid confusion.

```
foo = (*(volatile unsigned char *) 0x1234);
```

is taking the value stored at address `0x1234` and storing it in the variable foo.

```
(*(volatile unsigned char *) 0x1234) = 0xAA;
```

is storing the value `0xAA` in address `0x1234`.

9. Convert Hexadecimal to Binary

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


int main(void)
{
    uint8_t hex, bit;
    int8_t i;  //Since its value would become -1.
    printf("Enter the 1-byte Hex number: 0x");
    scanf("%x", &hex);
    printf("The Bin equivalent is: 0b");
    for(i = 7; i >= 0; i--)
    {
        bit = ((hex >> i) & 1);
        printf("%d", bit);
    }
    return(0);
}
```

10. Convert Byte to Binary

```
#define BYTE_TO_BINARY_PATTERN "%c%c%c%c%c%c%c%c"

#define BYTE_TO_BINARY(byte)  \

      (byte & 0x80 ? '1' : '0'), \

      (byte & 0x40 ? '1' : '0'), \

      (byte & 0x20 ? '1' : '0'), \

      (byte & 0x10 ? '1' : '0'), \

      (byte & 0x08 ? '1' : '0'), \

      (byte & 0x04 ? '1' : '0'), \

      (byte & 0x02 ? '1' : '0'), \

      (byte & 0x01 ? '1' : '0')
```

Usage in driver code:

```
printf("Leading text ""BYTE_TO_BINARY_PATTERN"\n",
BYTE_TO_BINARY(byte));
```

For multi-byte types:

```
printf("m: "BYTE_TO_BINARY_PATTERN" "BYTE_TO_BINARY_PATTERN"\n",
BYTE_TO_BINARY(m>>8), BYTE_TO_BINARY(m));
```

11. How does free() know the size of memory to be deallocated?

The free() function is used to deallocate memory while it is allocated using malloc(), calloc() and realloc(). The syntax of the free is simple. We simply use free with the pointer. Then it can clean up the memory.

```
free(ptr);
```

The free() is not taking any size as parameter, but only pointer. So the question comes, that how the free() function know about the size of the block to deallocate?

When we use the dynamic memory allocation techniques for memory allocations, then this is done in the actual heap section. It creates one word larger than the requested size. This extra word is used to store the size. This size is used by free() when it wants to clear the memory space.

12. Setter and Getter Functions

“Setter” function is used to Set or Update value to the variable. “Getter” function is used to retrieve value of the variable.

13. Interrupt Handling in Microcontroller

1. Microcontroller finishes the current instruction it is executing and saves address of next instruction (which is in the Program Counter) on to the Stack.

2. It also saves current status of all the interrupts internally (i.e., not on the Stack).

3. It jumps to the memory location of "Interrupt Vector Table". Interrupt Vector Table contains addresses of all ISRs.

4. Microcontroller fetches address of the ISR from Interrupt Vector Table and jumps to it. It starts executing the Interrupt Service Routine until it reaches last instruction of ISR.

5. Upon executing last instruction, the microcontroller returns to the place where it was interrupted. First it gets the program counter (PC) address from Stack by Popping top bytes of the Stack into PC. Then, it starts to execute from that address.

14. Bit rate vs Baud rate

```
baud rate = number of symbols per second

bit rate = number of bits per second

         = (baud rate)*(bits per symbol)*(number of channels)
```

15. Static vs Global variables

Global variables are variables which are defined outside all the functions. Static global variables are private to the source file where they are defined and do not conflict with other variables in other source files which would have the same name. Static local variables are initialized only once, even if the initialization statement is run more than once. Static local variable does not have global scope. But address of Static local variable has global scope. Note that, address of ordinary local variable has only local scope.

16. memset() function

memset() is used to fill a block of memory with a particular value.

The syntax of memset() function is as follows :

```
// ptr ==> Starting address of memory to be filled
// x   ==> Value to be filled
// n   ==> Number of bytes to be filled starting
//         from ptr to be filled
void *memset(void *ptr, int x, size_t n);
```

Note that ptr is a void pointer, so that we can pass any type of pointer to this function.

Let us see a simple example in C to demonstrate how memset() function is used:

```
// C program to demonstrate working of memset()
#include <stdio.h>
#include <string.h>
int main()
{
    char str[50] = "GeeksForGeeks is for programming geeks.";
    printf("\nBefore memset(): %s\n", str);
    // Fill 8 characters starting from str[13] with '.'
    memset(str + 13, '.', 8*sizeof(char));
    printf("After memset():  %s", str);
    return 0;
}
```

Output:

```
Before memset(): GeeksForGeeks is for programming geeks.
After memset(): GeeksForGeeks........programming geeks.
```

Explanation: (str + 13) points to first space (0 based index) of the string “GeeksForGeeks is for programming geeks.”, and memset() sets the character ‘.’ starting from first ‘ ‘ of the string up to 8 character positions of the given string and hence we get the output as shown above.

```
// C program to demonstrate working of memset()
#include <stdio.h>
#include <string.h>
```

```
void printArray(int arr[], int n)
{
   for (int i=0; i<n; i++)
      printf("%d ", arr[i]);
}
int main()
{
    int n = 10;
    int arr[n];
    // Fill whole array with 0.
    memset(arr, 0, n*sizeof(arr[0]));
    printf("Array after memset()\n");
    printArray(arr, n);
    return 0;
}
```

Output:

```
0 0 0 0 0 0 0 0 0 0
```

Exercise:

Predict the output of below program.

```
// C program to demonstrate working of memset()
#include <stdio.h>
#include <string.h>
void printArray(int arr[], int n)
{
   for (int i=0; i<n; i++)
      printf("%d ", arr[i]);
}
int main()
```

```
{
    int n = 10;
    int arr[n];
    // Fill whole array with 100.
    memset(arr, 10, n*sizeof(arr[0]));
    printf("Array after memset()\n");
    printArray(arr, n);
    return 0;
}
```

Note that the above code doesn't set array values to 10 as memset works character by character and an integer contains more than one byte (or characters).

However, if we replace 10 with -1, we get -1 values. Because representation of -1 contains all 1s in case of both char and int.

17. Dangling pointer

A pointer pointing to a memory location that has been deleted (or freed) is called dangling pointer.

Example:

```
#include <stdlib.h>
int main()
{
      int* ptr = (int*)malloc(sizeof(int));
      *ptr = 123;
      free(ptr);
      //ptr has now become a dangling pointer.
      return(0);
}
```

18. Void pointer

Void pointer is a specific pointer type – void * – a pointer that points to some data location in storage, which doesn't have any specific type. Void refers to the type. Basically, the type of data that it points to is can be any. If we assign address of char data type to void pointer it will become char Pointer, if int data type then int pointer and so on. Any pointer type is convertible to a void pointer hence it can point to any value.

Important Points

void pointers cannot be dereferenced. It can however be done using typecasting the void pointer

Pointer arithmetic is not possible on pointers of void due to lack of concrete value and thus size.

Example:

```
#include<stdlib.h>

int main()

{

      int x = 4;

      float y = 5.5;

      //A void pointer

      void *ptr;

      ptr = &x;

      // (int*)ptr - does type casting of void.

      // *((int*)ptr) dereferences the typecasted.

      // void pointer variable.

      printf("Integer variable is = %d", *( (int*) ptr) );

      // void pointer is now float

      ptr = &y;

      printf("\nFloat variable is= %f", *( (float*) ptr) );

      return 0;

}
```

Output:

```
Integer variable is = 4

Float variable is= 5.500000
```

19. NULL Pointer

NULL Pointer is a pointer which is pointing to nothing. In case, if we don't have address to be assigned to a pointer, then we can simply use NULL.

```
#include <stdio.h>

int main()

{

      // Null Pointer

      int *ptr = NULL;

      printf("The value of ptr is %u", ptr);

      return 0;

}
```

Output:

```
The value of ptr is 0
```

Important Points

NULL vs Uninitialized pointer – An uninitialized pointer stores an undefined value. A null pointer stores a defined value, but one that is defined by the environment to not be a valid address for any member or object.

NULL vs Void Pointer – Null pointer is a value, while void pointer is a type

20. Wild pointer

A pointer which has not been initialized to anything (not even NULL) is known as wild pointer. The pointer may be initialized to a non-NULL garbage value that may not be a valid address.

```
int main()

{

      int *p;  /* wild pointer */

      int x = 10;
```

```
    p = &x;  //p is not a wild pointer now.

    return 0;

}
```

21. Double Pointer (Pointer to Pointer)

We already know that a pointer points to a location in memory and thus used to store address of variables. So, when we define a pointer to pointer. The first pointer is used to store the address of second pointer. That is why they are also known as double pointers.

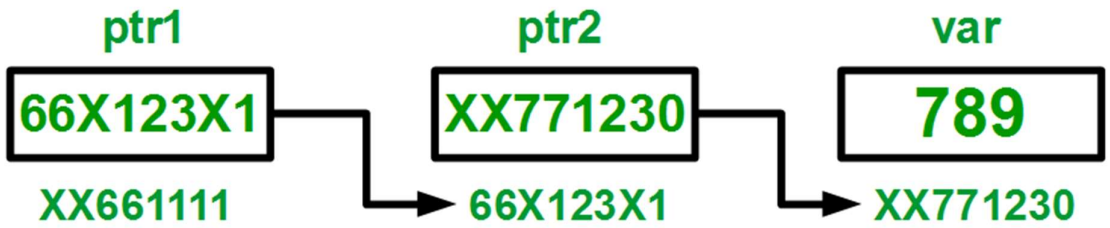
How to declare a pointer to pointer in C?

Declaring Pointer to Pointer is similar to declaring pointer in C. The difference is we have to place an additional '*' before the name of pointer.

Syntax:

```
int **ptr;    // declaring double pointers
```

Below diagram explains the concept of Double Pointers:



The above diagram shows the memory representation of a pointer to pointer. The first pointer ptr1 stores the address of the second pointer ptr2 and the second pointer ptr2 stores the address of the variable.

Let us understand this more clearly with the help of below program:

```
// C program to demonstrate pointer to pointer

#include <stdio.h>

int main()

{

    int var = 789;
```

```
    // pointer for var
    int *ptr2;
    // double pointer for ptr2
    int **ptr1;
    // storing address of var in ptr2
    ptr2 = &var;
    // Storing address of ptr2 in ptr1
    ptr1 = &ptr2;
    // Displaying value of var using
    // both single and double pointers
    printf("Value of var = %d\n", var );
    printf("Value of var using single pointer = %d\n", *ptr2 );
    printf("Value of var using double pointer = %d\n", **ptr1);
    return 0;
}
```

Output:

```
Value of var = 789
Value of var using single pointer = 789
Value of var using double pointer = 789
```

Call by reference function

The call by reference method of passing arguments to a function copies the address of an argument into the formal parameter. Inside the function, the address is used to access the actual argument used in the call. It means the changes made to the parameter affect the passed argument.

To pass a value by reference, argument pointers are passed to the functions just like any other value. So accordingly you need to declare the function parameters as pointer types as in the following function swap(), which exchanges the values of the two integer variables pointed to, by their arguments.

```
/* function definition to swap the values */
int swap(int *x, int *y)
```

```
{
      int temp;
      temp = *x;    /* save the value at address x */
      *x = *y;      /* put y into x */
      *y = temp;    /* put temp into y */
      return 0;
}
```

Let us now call the function swap() by passing values by reference as in the following example −

```
#include <stdio.h>
      /* function declaration */
      void swap(int *x, int *y);
      int main () {
      /* local variable definition */
      int a = 100;
      int b = 200;
      printf("Before swap, value of a : %d\n", a );
      printf("Before swap, value of b : %d\n", b );
      /* calling a function to swap the values.
      * &a indicates pointer to a ie. address of variable a and
      * &b indicates pointer to b ie. address of variable b.
      */
      swap(&a, &b);
      printf("After swap, value of a : %d\n", a );
      printf("After swap, value of b : %d\n", b );
      return 0;
}
```

Let us put the above code in a single C file, compile and execute it, to produce the following result −

```
Before swap, value of a :100

Before swap, value of b :200

After swap, value of a :200

After swap, value of b :100
```

It shows that the change has reflected outside the function as well, unlike call by value where the changes do not reflect outside the function.

22. Call by value function

The call by value method of passing arguments to a function copies the actual value of an argument into the formal parameter of the function. In this case, changes made to the parameter inside the function have no effect on the argument.

By default, C programming uses call by value to pass arguments. In general, it means the code within a function cannot alter the arguments used to call the function. Consider the function swap() definition as follows.

```
      /* function definition to swap the values */

      int swap(int x, int y) {

      int temp;

      temp = x; /* save the value of x */

      x = y;    /* put y into x */

      y = temp; /* put temp into y */

      return 0;

}
```

Now, let us call the function swap() by passing actual values as in the following example −

```
#include <stdio.h>

/* function declaration */
```

```
void swap(int x, int y);

int main ()

{

      /* local variable definition */

      int a = 100;

      int b = 200;

      printf("Before swap, value of a : %d\n", a );

      printf("Before swap, value of b : %d\n", b );

      /* calling a function to swap the values */

      swap(a, b);

      printf("After swap, value of a : %d\n", a );

      printf("After swap, value of b : %d\n", b );

      return 0;

}
```

Let us put the above code in a single C file, compile and execute it, it will produce the following result.

```
Before swap, value of a :100

Before swap, value of b :200

After swap, value of a :100

After swap, value of b :200
```

It shows that there are no changes in the values, though they had been changed inside the function.

23. volatile and const volatile keyword

Used to avoid compiler optimizations for the specified variable. Normally used with interrupts and GPIO pins. Used to tell that the value of the variable can change at any time.

```
volatile int x;
```

`const` means that the value isn't modifiable by the program.

And volatile means that the value is subject to sudden change (possibly from outside the program).

In fact, the C Standard gives an example of a valid declaration which is both `const` and `volatile`. The example is:

```
const volatile int real_time_clock;
```

where, `real_time_clock` may be modifiable by hardware, but cannot be assigned to, incremented, or decremented.

24. static extern is invalid

`static` and `extern` keywords cannot be used together, while declaring a variable as below:

```
static extern int num;
```

Because, `static` restricts the variable to the file in which it is declared and `extern` is used to access the variable from another file to this file. Both of these are contradictory and creates a conflict.

25. Inline Function

Inline Function are those function whose definitions are small and be substituted at the place where its function call is happened. Function substitution is totally compiler choice.

```
#include <stdio.h>

// Inline function in C

inline int foo()

{

    return 2;

}

// Driver code

int main()

{

    int ret;

    // inline function call

    ret = foo();

    printf("Output is: %d\n", ret);

    return 0;

}
```

Output:

```
In function `main':

undefined reference to `foo'
```

Why this error happened?

This is one of the side effects of GCC the way it handles inline function, when compiled, GCC performs inline substitution as the part of optimisation. So, there is no function call present (foo) inside main. Normally GCC's file scope is "not extern linkage". That means inline function is never ever provided to the linker which is causing linker error, mentioned above.

How to remove this error?

To resolve this problem use "static" before inline. Using static keyword forces the compiler to consider this inline function in the linker, and hence the program compiles and runs successfully.

Example:

```
#include <stdio.h>

// Inline function in C

static inline int foo()

{

      return 2;

}

// Driver code

int main()

{

      int ret;

      // inline function call

      ret = foo();

      printf("Output is: %d\n", ret);

      return 0;

}
```

Output:

```
Output is: 2
```

26. Storage class

A storage class defines the scope (visibility) and life-time of variables and/or functions within a C Program. They precede the type that they modify. We have four different storage classes in a C program −

auto

register

static

extern

1) auto

The auto storage class is the default storage class for all local variables.

```
{
      int mount;
      auto int month;
}
```

The example above defines two variables with in the same storage class. 'auto' can only be used within functions, i.e., local variables.

2) register

The register storage class is used to define local variables that should be stored in a register instead of RAM. This means that the variable has a maximum size equal to the register size (usually one word) and can't have the unary '&' operator applied to it (as it does not have a memory location).

```
{
      register int miles;
}
```

The register should only be used for variables that require quick access such as counters. It should also be noted that defining 'register' does not mean that the variable will be stored in a register. It means that it MIGHT be stored in a register depending on hardware and implementation restrictions.

3) static

The static storage class instructs the compiler to keep a local variable in existence during the life-time of the program instead of creating and destroying it each time it comes into and goes out of scope. Therefore, making local variables static allows them to maintain their values between function calls.

The static modifier may also be applied to global variables. When this is done, it causes that variable's scope to be restricted to the file in which it is declared.

Example:

```
#include <stdio.h>

/* function declaration */

void func(void);

static int count = 5; /* global variable only accessible inside this
file. It cannot be accessed with extern keyword from another file.
*/

int main()

{

      while(count--)

      {

            func();

      }

      return 0;

}

/* function definition */

void func(void)

{

      static int i = 5; /* Local static variable. It gets
initialized only once */

      i++;
```

```
      printf("i is %d and count is %d\n", i, count);
}
```

When the above code is compiled and executed, it produces the following result −

```
i is 6 and count is 4
i is 7 and count is 3
i is 8 and count is 2
i is 9 and count is 1
i is 10 and count is 0
```

4) extern

The extern storage class is used to give a reference of a global variable that is visible to ALL the program files. When you use 'extern', the variable cannot be initialized however, it points the variable name at a storage location that has been previously defined.

When you have multiple files and you define a global variable or function, which will also be used in other files, then extern will be used in another file to provide the reference of defined variable or function. Just for understanding, extern is used to declare a global variable or function in another file.

The extern modifier is most commonly used when there are two or more files sharing the same global variables or functions as explained below.

First File: main.c

```
#include <stdio.h>
int count;
extern void write_extern();
int main()
{
      count = 5;
      write_extern();
}
```

Second File: support.c

```
#include <stdio.h>

extern int count;

void write_extern(void)

{

      printf("count is %d\n", count);

}
```

Here, extern is being used to declare count in the second file, where as it has its definition in the first file, main.c. Now, compile these two files as follows −

```
$ gcc main.c support.c
```

It will produce the executable program a.out. When this program is executed, it produces the following result.

```
count is 5
```

27. Structure declaration and definition

```
struct student

{

      int id;

      char name[20];

};

int main()

{

      struct student record;

      record.id = 12;

      strcpy(record.name, "Raju");

      return(0);

}
```

OR

```
struct student
{
      int id;
      char name[20];
}record;
int main()
{
      record.id = 12;
      strcpy(record.name, "Raju");
      return(0);
}
```

28. typedef and struct

```
typedef struct {
  int x, y;
} Point;
Point point_new(int x, int y)
{
  Point a;
  a.x = x;
  a.y = y;
  return a;
}
```

29. Passing structure to function

It can be done in below 3 ways.

1) Passing structure to a function by value

2) Passing structure to a function by address (reference)
3) Declare structure variable as global

Example program – passing structure to function in C by value:

In this program, the whole structure is passed to another function by value. It means the whole structure is passed to another function with all members and their values. So, this structure can be accessed from called function. This concept is very useful while writing very big programs in C.

```
#include <stdio.h>
#include <string.h>
struct student
{
      int id;
      char name[20];
      float percentage;
};
void func(struct student record);
int main()
{
      struct student record;
      record.id = 1;
      strcpy(record.name, "Raju");
      record.percentage = 86.5;
      func(record);
      return 0;
}
void func(struct student record)
{
      printf(" Id is: %d \n", record.id);
      printf(" Name is: %s \n", record.name);
      printf(" Percentage is: %f \n", record.percentage);
```

```
}
```

Output:

```
Id is: 1

Name is: Raju

Percentage is: 86.500000
```

Example program – Passing structure to function in C by address:

In this program, the whole structure is passed to another function by address. It means only the address of the structure is passed to another function. The whole structure is not passed to another function with all members and their values. So, this structure can be accessed from called function by its address.

```
#include <stdio.h>

#include <string.h>

struct student

{

      int id;

      char name[20];

      float percentage;

};

void func(struct student *record);

int main()

{

      struct student record;

      record.id=1;

      strcpy(record.name, "Raju");

      record.percentage = 86.5;

      func(&record);

      return 0;
```

```
}
void func(struct student *record)
{
      printf(" Id is: %d \n", record->id);
      printf(" Name is: %s \n", record->name);
      printf(" Percentage is: %f \n", record->percentage);
}
```

Output:

```
Id is: 1
Name is: Raju
Percentage is: 86.500000
```

Example program to declare a structure variable as global in C:

Structure variables also can be declared as global variables as we declare other variables in C. So, when a structure variable is declared as global, then it is visible to all the functions in a program. In this scenario, we don't need to pass the structure to any function separately.

```
#include <stdio.h>
#include <string.h>
struct student
{
      int id;
      char name[20];
      float percentage;
};
struct student record;  //Global declaration of structure
void structure_demo();
int main()
{
```

```
	record.id=1;
	strcpy(record.name,"Raju");
	record.percentage = 86.5;
	structure_demo();
	return 0;
}
void structure_demo()
{
	printf(" Id is: %d \n",record.id);
	printf(" Name is: %s \n",record.name);
	printf(" Percentage is: %f \n",record.percentage);
}
```

Output:

```
Id is: 1
Name is: Raju
Percentage is: 86.500000
```

30. Static Function

In C, functions are global by default. The "static" keyword before a function name makes it static. For example, below function fun() is static.

```
static int fun(void)
{
	printf("I am a static function ");
}
```

Unlike global functions in C, access to static functions is restricted to the file where they are declared. Therefore, when we want to restrict access to functions, we make them static. Another reason for making functions static can be reuse of the same function name in other files.

For example, if we store following program in one file file1.c

```
/* Inside file1.c */

static void fun1(void)

{

      puts("fun1 called");

}
```

And store following program in another file file2.c

```
/* Inside file2.c  */

int main(void)

{

      fun1();

      getchar();

      return 0;

}
```

Now, if we compile the above code with command “gcc file2.c file1.c”, we get the error “undefined reference to “fun1””. This is because fun1() is declared static in file1.c and cannot be used in file2.c.

31. Type casting

Type casting is a way to convert a variable from one data type to another data type. For example, if you want to store a 'long' value into a simple integer then you can type cast 'long' to 'int'. You can convert the values from one type to another explicitly using the cast operator as follows −

```
(type_name) expression
```

Consider the following example where the cast operator causes the division of one integer variable by another to be performed as a floating-point operation −

Example:

```
#include <stdio.h>

int main(void)

{

      int sum = 17, count = 5;
```

```
      double mean;

      mean = (double) sum / count;

      printf("Value of mean : %f\n", mean );

      return(0);

}
```

When the above code is compiled and executed, it produces the following result −

```
Value of mean : 3.400000
```

It should be noted here that the cast operator has precedence over division, so the value of sum is first converted to type double and finally it gets divided by count yielding a double value.

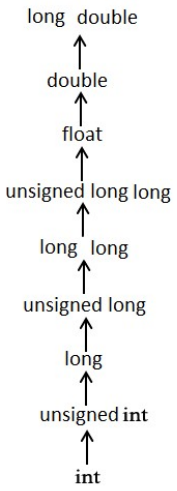
Type conversions can be implicit which is performed by the compiler automatically, or it can be specified explicitly through the use of the cast operator. It is considered good programming practice to use the cast operator whenever type conversions are necessary.

32. Integer Promotion

Integer promotion is the process by which values of integer type "smaller" than int or unsigned int are converted either to int or unsigned int. Consider an example of adding a character with an integer −

Example:

```
#include <stdio.h>

main()

{

      int i = 17;

      char c = 'c'; /* ascii value is 99 */

      int sum;

      sum = i + c;

      printf("Value of sum : %d\n", sum );

}
```



When the above code is compiled and executed, it produces the following result.

```
Value of sum : 116
```

Here, the value of sum is 116 because the compiler is doing integer promotion and converting the value of 'c' to ASCII before performing the actual addition operation.

Usual Arithmetic Conversion

The usual arithmetic conversions are implicitly performed to cast their values to a common type. The compiler first performs integer promotion; if the operands still have different types, then they are converted to the type that appears highest in the following hierarchy –

The usual arithmetic conversions are not performed for the assignment operators, nor for the logical operators && and ||. Let us take the following example to understand the concept −

Example:

```
#include <stdio.h>

void main()

{

      int i = 17;

      char c = 'c'; /* ascii value is 99 */

      float sum;

      sum = i + c;

      printf("Value of sum : %f\n", sum );

}
```

When the above code is compiled and executed, it produces the following result −

```
Value of sum : 116.000000
```

Here, it is simple to understand that first c gets converted to integer, but as the final value is double, usual arithmetic conversion applies and the compiler converts i and c into 'float' and adds them yielding a 'float' result.

33. Dynamic Memory Allocation

As you know, you have to declare the size of an array before you use it. Hence, the array you declared may be insufficient or more than required to hold data. To solve this issue, you can allocate memory dynamically.

Dynamic memory management refers to manual memory management. This allows you to obtain more memory when required and release it when not necessary.

Although C inherently does not have any technique to allocate memory dynamically, there are 4 library functions defined under <stdlib.h> for dynamic memory allocation.

Function	Use of Function
malloc()	Allocates requested size of bytes and returns a pointer first byte of allocated space
calloc()	Allocates space for an array elements, initializes to zero and then returns a pointer to memory
free()	Deallocate the previously allocated space
realloc()	Change the size of previously allocated space

Note

Dynamic Memory Allocation cannot be initialised as global.

```
#include <stdio.h>

#include <stdlib.h>


int num;

int *p = &num;  //No Error.

int *ptr = (int *)malloc(sizeof(int));  //Causes Error.


int main(void)

{

      int *poiter = (int *)malloc(sizeof(int));  //No Error.

      return(0);

}
```

malloc()

The name malloc stands for "memory allocation".

The function malloc() reserves a block of memory of specified size and return a pointer of type void which can be casted into pointer of any form.

Syntax of malloc()

```
<pointer type> *ptr = (cast-type*) malloc(byte-size);
```

Here, ptr is pointer of cast-type. The malloc() function returns a pointer to an area of memory with size of byte size. If the space is insufficient, allocation fails and returns NULL pointer.

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

This statement will allocate either 200 or 400 according to size of int 2 or 4 bytes respectively and the pointer points to the address of first byte of memory.

calloc()

The name calloc stands for "contiguous allocation".

The only difference between malloc() and calloc() is that, malloc() allocates single block of memory whereas calloc() allocates multiple blocks of memory each of same size and sets all bytes to zero.

Syntax of calloc()

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

This statement will allocate contiguous space in memory for an array of n elements. For example:

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

This statement allocates contiguous space in memory for an array of 25 elements each of size of float, i.e, 4 bytes.

free()

Dynamically allocated memory created with either calloc() or malloc() doesn't get freed on its 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.

Example #1: Using C malloc() and free()

Write a C program to find sum of n elements entered by user. To perform this program, allocate memory dynamically using malloc() function.

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
      int num, i, *ptr, sum = 0;
      printf("Enter number of elements: ");
      scanf("%d", &num);
      ptr = (int*) malloc(num * sizeof(int));   //memory allocated
using malloc
      if(ptr == NULL)
      {
            printf("Error! memory not allocated.");
            exit(0);
      }
      printf("Enter elements of array: ");
      for(i = 0; i < num; ++i)
      {
            scanf("%d", ptr + i);
            sum += *(ptr + i);
      }
```

```
    printf("Sum = %d", sum);
    free(ptr);
    return 0;
}
```

Example #2: Using calloc() and free()

Write a C program to find sum of n elements entered by user. To perform this program, allocate memory dynamically using calloc() function.

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int num, i, *ptr, sum = 0;
    printf("Enter number of elements: ");
    scanf("%d", &num);
    ptr = (int*) calloc(num, sizeof(int));
    if(ptr == NULL)
    {
        printf("Error! memory not allocated.");
        exit(0);
    }
    printf("Enter elements of array: ");
    for(i = 0; i < num; ++i)
    {
        scanf("%d", ptr + i);
        sum += *(ptr + i);
    }
```

```
	printf("Sum = %d", sum);
	free(ptr);
	return 0;
}
```

realloc()

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

Syntax of realloc()

```
<pointer> = realloc(<pointer>, <new size>);
```

Here, ptr is reallocated with size of new-size.

Example #3: Using realloc()

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
	int *ptr, i, n1, n2;
	printf("Enter size of array: ");
	scanf("%d", &n1);
	ptr = (int*) malloc(n1 * sizeof(int));
	printf("Address of previously allocated memory: ");
	for(i = 0; i < n1; ++i)
	{
		printf("%u\t",ptr + i);
	}
```

```
      printf("\nEnter new size of array: ");
      scanf("%d", &n2);
      ptr = realloc(ptr, n2 * sizeof(int));
      for(i = 0; i < n2; ++i)
      {
            printf("%u\t", ptr + i);
      }
      return 0;
}
```

Output:

```
Enter size of array: 2
Addresses of previously allocated memory:26855472
26855476

Enter new size of array: 4
Addresses of newly allocated memory:26855472
26855476
26855480
26855484
```

34. Union

A union is a special data type available in C that allows to store different data types in the same memory location. You can define a union with many members, but only one member can contain a value at any given time. Unions provide an efficient way of using the same memory location for multiple-purpose.

Defining a Union

To define a union, you must use the union statement in the same way as you did while defining a structure. The union statement defines a new data type with more than one member for your program. The format of the union statement is as follows −

```
union <union name>

{

      member definition;

      member definition;

      ...

      member definition;

} <one or more union variables separated by comma>;
```

The union tag is optional and each member definition is a normal variable definition, such as int i; or float f; or any other valid variable definition. At the end of the union's definition, before the final semicolon, you can specify one or more union variables but it is optional. Here is the way you would define a union type named Data having three members i, f, and str −

```
union Data

{

      int i;

      float f;

      char str[20];

} data;
```

Now, a variable of Data type can store an integer, a floating-point number, or a string of characters. It means a single variable, i.e., same memory location, can be used to store multiple types of data. You can use any built-in or user defined data types inside a union based on your requirement.

The memory occupied by a union will be large enough to hold the largest member of the union. For example, in the above example, Data type will occupy 20 bytes of memory space because this is the maximum space which can be occupied by a character string. The following example displays the total memory size occupied by the above union −

Example:

```
#include <stdio.h>

#include <string.h>

union Data

{

      int i;

      float f;

      char str[20];
```

```
};
int main()
{
      union Data data;
      printf( "Memory size occupied by data : %d\n", sizeof(data));
      return 0;
}
```

When the above code is compiled and executed, it produces the following result −

```
Memory size occupied by data : 20
```

Accessing Union Members

To access any member of a union, we use the member access operator (.). The member access operator is coded as a period between the union variable name and the union member that we wish to access. You would use the keyword union to define variables of union type. The following example shows how to use unions in a program −

Example:

```
#include <stdio.h>
#include <string.h>
union Data
{
      int i;
      float f;
      char str[20];
};
int main()
{
      union Data data;
      data.i = 10;
      data.f = 220.5;
      strcpy( data.str, "C Programming");
      printf( "data.i : %d\n", data.i);
```

```
      printf( "data.f : %f\n", data.f);

      printf( "data.str : %s\n", data.str);

      return 0;

}
```

When the above code is compiled and executed, it produces the following result −

```
data.i : 1917853763

data.f : 4122360580327794860452759994368.000000

data.str : C Programming
```

Here, we can see that the values of i and f members of union got corrupted because the final value assigned to the variable has occupied the memory location and this is the reason that the value of str member is getting printed very well.

Now let's look into the same example once again where we will use one variable at a time which is the main purpose of having unions −

Example:

```
#include <stdio.h>

#include <string.h>

union Data

{

      int i;

      float f;

      char str[20];

};

int main()

{

      union Data data;

      data.i = 10;

      printf("data.i : %d\n", data.i);

      data.f = 220.5;
```

```
      printf( "data.f : %f\n", data.f);

      strcpy( data.str, "C Programming");

      printf( "data.str : %s\n", data.str);

      return 0;
}
```

When the above code is compiled and executed, it produces the following result −

```
data.i : 10

data.f : 220.500000

data.str : C Programming
```

Here, all the members are getting printed very well because one member is being used at a time.

35. Linked List

Example:

```
#include <stdio.h>

#include <string.h>

#include <stdlib.h>

struct node

{

      int data;

      struct node *next;

};

struct node *head = NULL;


// display the list

void printList()

{
```

```
      struct node *p = head;
      printf("\n[");

      //start from the beginning
      while(p != NULL)
      {
        printf(" %d ",p->data);
        p = p->next;
      }
      printf("]");
}

//insertion at the beginning
void insertatbegin(int data)
{
      //create a link
      struct node *lk = (struct node*) malloc(sizeof(struct node));
      lk->data = data;

      // point it to old first node
      lk->next = head;

      //point first to new first node
      head = lk;
}

void insertatend(int data)
{
      //create a link
```

```
	struct node *lk = (struct node*) malloc(sizeof(struct node));
	lk->data = data;
	lk->next = NULL;
	struct node *linkedlist = head;

	// point it to old first node
	while(linkedlist->next != NULL)
	{
		linkedlist = linkedlist->next;
	}
	//point first to new first node
	linkedlist->next = lk;
}

void insertafternode(struct node *list, int data)
{
	struct node *lk = (struct node*) malloc(sizeof(struct node));
	lk->data = data;
	lk->next = list->next;
	list->next = lk;
}

void deleteatbegin()
{
	head = head->next;
}

void deleteatend()
{
```

```
	struct node *linkedlist = head;
	while (linkedlist->next->next != NULL)
	{
		linkedlist = linkedlist->next;
	}
	linkedlist->next = NULL;
}

void deletenode(int key)
{
	struct node *temp = head, *prev = NULL;
	if (temp != NULL && temp->data == key)
	{
		head = temp->next;
		return;
	}

	// Find the key to be deleted
	while (temp != NULL && temp->data != key)
	{
		prev = temp;
		temp = temp->next;
	}

	// If the key is not present
	if (temp == NULL)
	{
		return;
	}
```

```
	// Remove the node
	prev->next = temp->next;
}

int searchlist(int key)
{
	struct node *temp = head;
	while(temp != NULL)
	{
		if (temp->data == key)
		{
			return 1;
		}
			temp=temp->next;
	}
	return 0;
}

void main()
{
	int k=0;
	insertatbegin(12);
	insertatbegin(22);
	insertatend(30);
	insertatend(44);
	insertatbegin(50);
	insertafternode(head->next->next, 33);
	printf("Linked List: ");
```

```
      // print list
      printList();
      deleteatbegin();
      deleteatend();
      deletenode(12);
      printf("\nLinked List after deletion: ");

      // print list
      printList();
      insertatbegin(4);
      insertatbegin(16);
      printf("\nUpdated Linked List: ");
      printList();
      k = searchlist(16);
      if (k == 1)
      {
            printf("\nElement is found");
      }
      else
      {
            printf("\nElement is not present in the list");
      }
}
```

If we compile and run the above program, it will produce the following result −

Output:

```
Linked List:
[ 50  22  12  33  30  44 ]
Linked List after deletion:
```

```
[ 22  33  30 ]

Updated Linked List:

[ 16  4  22  33  30 ]

Element is found
```

36. Splitting a string using strtok_s()

Use string.h header file for getting access to strtok_s().

```
char* token = NULL;
char* next = NULL;
char s[] = "this is a string";

for(token = strtok_s(s, " ", &next); token != NULL; token = strtok_s(NULL,
" ", &next))
{
    puts(token);
}
```

37. Pointer to Structure

```
struct dog

{

char name[10];

char breed[10];

int age;

char color[10];

};

struct dog spike;

// declaring a pointer to a structure of type struct dog

struct dog *ptr_dog

ptr_dog = &spike;

(*ptr_dog).name    //refers to the name of dog

(*ptr_dog).breed    //refers to the breed of dog

ptr_dog→name    //refers to the name of dog

ptr_dog→breed    //refers to the breed of dog
```

38. Easy way to find the number of elements in an array

```
int arr[] = {10, 5, 15, 12, 90, 80};

int n = sizeof(arr)/sizeof(arr[0]);
```

39. Function Pointers

In C, like normal data pointers (int *, char *, etc), we can have pointers to functions. Following is a simple example that shows declaration and function call using function pointer.

```
#include <stdio.h>

// A normal function with an int parameter

// and void return type

void fun(int a)

{

      printf("Value of a is %d\n", a);

}


int main()

{

      // fun_ptr is a pointer to function fun()

      void (*fun_ptr)(int) = &fun;


      /* The above line is equivalent of following two

      void (*fun_ptr)(int);

      fun_ptr = &fun;

      */


      // Invoking fun() using fun_ptr

      (*fun_ptr)(10);
```

```
    return 0;
}
```

Output:

```
Value of a is 10
```

Why do we need an extra bracket around function pointers like fun_ptr in above example?

If we remove bracket, then the expression "void (*fun_ptr)(int)" becomes "void *fun_ptr(int)" which is declaration of a function that returns void pointer. See following post for details.

Following are some interesting facts about function pointers.

1) Unlike normal pointers, a function pointer points to code, not data. Typically, a function pointer stores the start of executable code.

```
#include <stdio.h>

int main(void)

{

    void (*lets_jump)(void) = (void*) 0x1233;

    lets_jump();

    return(0);

}
```

Output:

```
<jumps to the memory location 0x1233 and executes the code present
there>
```

2) Unlike normal pointers, we do not allocate de-allocate memory using function pointers.

3) A function's name can also be used to get functions' address. For example, in the below program, we have removed address operator '&' in assignment. We have also changed function call by removing *, the program still works.

```
#include <stdio.h>

// A normal function with an int parameter
```

```
// and void return type
void fun(int a)
{
      printf("Value of a is %d\n", a);
}


int main()
{
      void (*fun_ptr)(int) = fun; // & removed


      fun_ptr(10); // * removed


      return 0;
}
```

Output:

```
Value of a is 10
```

4) Like normal pointers, we can have an array of function pointers. Below example in point 5 shows syntax for array of pointers.

5) Function pointer can be used in place of switch case. For example, in below program, user is asked for a choice between 0 and 2 to do different tasks.

```
#include <stdio.h>
void add(int a, int b)
{
      printf("Addition is %d\n", a+b);
}
void subtract(int a, int b)
{
```

```
      printf("Subtraction is %d\n", a-b);
}
void multiply(int a, int b)
{
      printf("Multiplication is %d\n", a*b);
}


int main()
{
      // fun_ptr_arr is an array of function pointers
      void (*fun_ptr_arr[])(int, int) = {add, subtract, multiply};
      unsigned int ch, a = 15, b = 10;


      printf("Enter Choice: 0 for add, 1 for subtract and 2 for
multiply\n");
      scanf("%d", &ch);


      if (ch > 2) return 0;


      (*fun_ptr_arr[ch])(a, b);


      return 0;
}
```

Output:

```
Enter Choice: 0 for add, 1 for subtract and 2 for multiply
2
Multiplication is 150
```

6) Like normal data pointers, a function pointer can be passed as an argument and can also be returned from a function.

For example, consider the following C program where wrapper() receives a void fun() as parameter and calls the passed function.

```
// A simple C program to show function pointers as parameter
#include <stdio.h>

// Two simple functions
void fun1() { printf("Fun1\n"); }
void fun2() { printf("Fun2\n"); }

// A function that receives a simple function
// as parameter and calls the function
void wrapper(void (*fun)())
{
      fun();
}

int main()
{
      wrapper(fun1);
      wrapper(fun2);
      return 0;
}
```

This point in particular is very useful in C. In C, we can use function pointers to avoid code redundancy. For example a simple qsort() function can be used to sort arrays in ascending order or descending or by any other order in case of array of structures. Not only this, with function pointers and void pointers, it is possible to use qsort for any data type.

```
// An example for qsort and comparator
#include <stdio.h>
```

```
#include <stdlib.h>

// A sample comparator function that is used
// for sorting an integer array in ascending order.
// To sort any array for any other data type and/or
// criteria, all we need to do is write more compare
// functions. And we can use the same qsort()
int compare (const void * a, const void * b)
{
return ( *(int*)a - *(int*)b );
}

int main ()
{
int arr[] = {10, 5, 15, 12, 90, 80};
int n = sizeof(arr)/sizeof(arr[0]), i;

qsort (arr, n, sizeof(int), compare);

for (i=0; i<n; i++)
     printf ("%d ", arr[i]);
return 0;
}
```

Output:

```
5 10 12 15 80 90
```

Similar to qsort(), we can write our own functions that can be used for any data type and can do different tasks without code redundancy. Below is an example search function that can be used for

any data type. In fact, we can use this search function to find close elements (below a threshold) by writing a customized compare function.

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

// A compare function that is used for searching an integer
// array
bool compare (const void * a, const void * b)
{
return ( *(int*)a == *(int*)b );
}

// General purpose search() function that can be used
// for searching an element *x in an array arr[] of
// arr_size. Note that void pointers are used so that
// the function can be called by passing a pointer of
// any type. ele_size is size of an array element
int search(void *arr, int arr_size, int ele_size, void *x,
           bool compare (const void * , const void *))
{
      // Since char takes one byte, we can use char pointer
      // for any type/ To get pointer arithmetic correct,
      // we need to multiply index with size of an array
      // element ele_size
      char *ptr = (char *)arr;

      int i;
      for (i=0; i<arr_size; i++)
            if (compare(ptr + i*ele_size, x))
```

```
            return i;

      // If element not found
      return -1;
}

int main()
{
      int arr[] = {2, 5, 7, 90, 70};
      int n = sizeof(arr)/sizeof(arr[0]);
      int x = 7;
      printf ("Returned index is %d ", search(arr, n,
                                    sizeof(int), &x, compare));
      return 0;
}
```

Output:

```
Returned index is 2
```

The above search function can be used for any data type by writing a separate customized compare().

7) Many object-oriented features in C++ are implemented using function pointers in C.

40. Call back function using function pointer – Example

```
#include <stdio.h>

int call_back_function(int a)
```

```
{
    printf("Hello: %d\n", a);
    return(0);
}

int foo(int (*ptr)(int), int num)
{
    printf("Processing Complete - Calling Call Back Function\n");
    ptr(num);
    return(0);
}

int main(void)
{
    int b = 789;
    int (*hello)(int);
    hello = call_back_function;
    foo(hello, b);
    return(0);
}
```

41. Function pointer without a name in C

A normal function pointer would look like this:

```
void (*fun_ptr)(void);
```

However, I have seen this syntax used to cast something to a void function pointer:

```
(void (*)(void *))
```

As you can see it has no name and only has (*) in place of a name. What does this mean? Is it only used for casting?

The syntax `(void (*)(void *))` is a cast.

The destination type of the cast is a function pointer that takes a single parameter of type `void *` and returns `void`. An example of a function whose type matches this cast is:

```
void abc(void *param);
```

42. C program to check Endianness

unsigned int x = 0x7654;

char *c = (char*) &x;

Big endian format:

Byte address | 0x01 | 0x02 |

Byte content | 0x76 | 0x54 |

Little endian format:

Byte address | 0x01 | 0x02 |

Byte content | 0x54 | 0x76 |

Quick way to remember:

LLL – **L**east significant byte in **L**east address, it means **L**ittle endian.

C program to find out if the underlying architecture is little endian or big endian.

```
#include <stdio.h>
int main ()
{
  unsigned int x = 0x7654;
  char *c = (char*) &x;

  printf ("*c is: 0x%x\n", *c);
  if (*c == 0x54)
  {
```

```
    printf ("Underlying architecture is little endian. \n");
  }
  else
  {
     printf ("Underlying architecture is big endian. \n");
  }
   return(0);
}
```

43. Initializing multi-dimensional array pointer

```
int arr[3][4][5];
int (*buffer)[4][5];
buffer = &arr;
```

44. Referring to an element in multi-dimensional array pointer

`arr[2][3][4]` is same as `*(*(*(buffer+2)+3)+4)` and `*(*(buffer+2)+3)+4` gives the address of `arr[2][3][4]`.

45. Bit-Shift and Bitwise operators

Let, the initial value of `REGISTER` be `0bXXXX XXXX` (0 otherwise 1, we really "Don't Care"). The code given below:

```
REGISTER = REGISTER << 3;
```

Otherwise,

```
REGISTER <<= 3;
```

Shifts contents in the `REGISTER` to 3 places towards left. So, the final value of `REGISTER` would be `0bXXXX X000`

Similarly, let the initial value of `REGISTER` be `0bXXXX XXXX` (0 otherwise 1, we really "Don't Care"). The code given below:

```
REGISTER = REGISTER >> 3;
```

Otherwise,

```
REGISTER >>= 3;
```

Shifts contents in the `REGISTER` to 3 places towards right. So, the final value of `REGISTER` would be `0b000X XXXX`

Let, the initial value of `REGISTER` be `0bXXXX XXXX` (0 otherwise 1, we really "Don't Care"). If we want to change the 3rd bit from right (i.e., bit 2, also called as 2nd place, hereafter), we can do something like below.

```
REGISTER = REGISTER | (1 << 2);
```

Otherwise,

```
REGISTER |= (1 << 2);
```

This will change the `REGISTER` value to `0bXXXX X1XX`. Here, `1<<2` means, shift, binary digit 1 from right to 2nd place. When 1 is OR-ed with 1 or 0 (Depicted by X, i.e. "Don't Care"), the result is always 1. Similar is the case if we want to make the 2nd place to 0. But, here, we use AND. Remember 0 when AND-ed with 1 or 0 always results in 0. Below is the code.

```
REGISTER = REGISTER & ~(1 << 2);
```

Otherwise,

```
REGISTER &= ~(1 << 2);
```

On several occasions, we might want to change few bits in a register but not affect the other bits in the same register. If we want to change 7 bits of this 8-bit register, we would need to write the above line of code 7 times (What if the register were 32 bit long and if we wanted to change 31 bits in it?), which is inefficient. There is a much easier way to this. Say for instance, if we want to change, 2nd place to 1, 5th place to 0 and 6th place to 1. We can write the below 2 lines of code for that.

```
REGISTER &= 0b1001 1011;

REGISTER |= 0b0100 0100;
```

Final, `REGISTER` value will be `0bX10X X1XX`. Just, as we wanted.

If we want to compliment a particular bit without affecting other bits of the register, we could do an XOR operation like below. This will, compliment the 2nd bit. Remember, `X^1 = ~X`, and, `X^0 = X`.

```
REGISTER = REGISTER ^ (1 << 2);
```

Otherwise,

```
REGISTER ^= (1 << 2);
```

On several occasions, we might want to compliment few bits in a register but not affect the other bits in the same register. Say for instance, if we want to compliment, 2nd place, 5th place and 6th place. We can write the below single line of code for that.

```
REGISTER ^= 0b0110 0100
```

46. Pointer Arithmetic

```
int* p = (int *) 0x1F4;  //Decimal value is 500.

int* q = (int *) 0x226;  //Decimal value is 550.

int i;

i = q - p;  //Gives the number of integer blocks between 500 and
550, which is, (550-500)/2, i.e. 25. Assuming size of int is 2
bytes. So, the value of i will be 25.

p++;  //Increments p by 2 bytes, assuming size of int is 2 bytes.

p--;  //Decrements p by 2 bytes, assuming size of int is 2 bytes.

p = p + 3;  //Increments p by 6 bytes, assuming size of int is 2
bytes.

p = p - 3;  //Decrements p by 6 bytes, assuming size of int is 2
bytes.
```

Always read from right to left in these below scenarios as they are unary operators.

```
++(*p)  //Read data and increment data.

++*p  //Read data and increment data.

*++p  //Increment pointer and read data.

*p++  //Read data and increment pointer. Here, although ++ has
higher priority it is executed after * because it is postfix.

(*p)++  //Read data and increment data.
```

47. enum

Example 1

```
#include<stdio.h>

enum week{Mon, Tue, Wed, Thur, Fri, Sat, Sun};
```

```
int main()
{
    enum week day;
    day = Wed;
    printf("%d",day);
    return(0);
}
```

Output

```
2
```

Example 2

```
#include<stdio.h>
enum year{Jan, Feb, Mar, Apr, May, Jun, Jul,
          Aug, Sep, Oct, Nov, Dec};
int main()
{
   int i;
   for (i=Jan; i<=Dec; i++)
      printf("%d ", i);
   return 0;
}
```

Output

```
0 1 2 3 4 5 6 7 8 9 10 11
```

Example 3

```
#include <stdio.h>
enum day {sunday = 1, monday, tuesday = 5,
```

```
          wednesday, thursday = 10, friday, saturday};

int main()
{
    printf("%d %d %d %d %d %d %d", sunday, monday, tuesday,
            wednesday, thursday, friday, saturday);
    return 0;
}
```

Output

```
1 2 5 6 10 11 12
```

Example **4**

```
#include <stdio.h>
enum state  {working, failed};
enum result {failed, passed};
int main()
{
return 0;
}
```

Output

```
main.c:4:14: error: redeclaration of enumerator 'failed'
 enum result {failed, passed};
              ^~~~~~
main.c:3:23: note: previous definition of 'failed' was here
 enum state  {working, failed};
          ^~~~~~
```

48. Bit Fields

Suppose your C program contains a number of TRUE/FALSE variables grouped in a structure called status, as follows –

```
struct {

   unsigned int widthValidated;

   unsigned int heightValidated;

} status;
```

This structure requires 8 bytes of memory space but in actual, we are going to store either 0 or 1 in each of the variables. The C programming language offers a better way to utilize the memory space in such situations.

If you are using such variables inside a structure then you can define the width of a variable which tells the C compiler that you are going to use only those number of bytes. For example, the above structure can be re-written as follows –

```
struct {

   unsigned int widthValidated : 1;

   unsigned int heightValidated : 1;

} status;
```

The above structure requires 4 bytes of memory space for status variable, but only 2 bits will be used to store the values.

If you will use up to 32 variables each one with a width of 1 bit, then also the status structure will use 4 bytes. However as soon as you have 33 variables, it will allocate the next slot of the memory and it will start using 8 bytes. Let us check the following example to understand the concept –

```
#include <stdio.h>

#include <string.h>

/* define simple structure */

struct {

   unsigned int widthValidated;

   unsigned int heightValidated;

} status1;

/* define a structure with bit fields */

struct {

   unsigned int widthValidated : 1;
```

```
   unsigned int heightValidated : 1;
} status2;
int main(void) {
   printf("Memory size occupied by status1 : %d\n",
sizeof(status1));
   printf( "Memory size occupied by status2 : %d\n",
sizeof(status2));
   return 0;
}
```

When the above code is compiled and executed, it produces the following result –

```
Memory size occupied by status1 : 8
Memory size occupied by status2 : 4
```

Bit Field Declaration

The declaration of a bit-field has the following form inside a structure –

```
struct {
<data type> <member name> : <width>;
};
```

Example

```
#include <stdio.h>
#include <string.h>
struct {
   unsigned int age : 3;
} Age;
int main( ) {
   Age.age = 4;
   printf( "Sizeof( Age ) : %d\n", sizeof(Age) );
   printf( "Age.age : %d\n", Age.age );
```

```
    Age.age = 7;
    printf( "Age.age : %d\n", Age.age );
    Age.age = 8;
    printf( "Age.age : %d\n", Age.age );
    return 0;
}
```

Output

```
Sizeof( Age ) : 4
Age.age : 4
Age.age : 7
Age.age : 0
```

49. Assigning Pointer address manually

```
void * p = (void *)0x28FF44;
```

Or if you want it as a char *

```
char * p = (char *)0x28FF44;
```

If you're pointing to something you really, really aren't meant to change, add a `const`.

```
const void * p = (const void *)0x28FF44;
const char * p = (const char *)0x28FF44;
```

Defining memory location of Memory Mapped Registers using Pointers

```
#define REGISTER *((uint8_t*) 0x123456)
```

Note: Accessing Non-Memory Mapped Registers, is possible only by using Assembly Language.

50. Format Specifiers

```
%c    Character
%d    Signed integer
```

```
%e or %E   Scientific notation of floats

%f    Float values

%.2f  Float values by rounding to 2 decimal places

%g or %G   Similar as %e or %E

%hi   Signed integer (short)

%hu   Unsigned Integer (short)

%i    Unsigned integer

%l or %ld or %li Long

%lf   Double

%Lf   Long double

%lu   Unsigned int or unsigned long

%lli or %lld    Long long

%llu  Unsigned long long

%o    Octal representation

%p    Pointer

%s    String

%u    Unsigned int

%x or %X   Hexadecimal representation

%n    Prints nothing

%%    Prints % character
```

51. Bootloader, Bootstrap Loader and Start-up Code

A bootloader is responsible for moving, and in some cases decompressing, the operating system kernel from long term storage (i.e., flash or hard disk) into main memory (RAM) for execution by the CPU. A bootstrap loader is another term for bootloader.

Start-up code, however, is slightly different. Start-up code refers to the handful of subroutines that run when the kernel is first handed control by the bootloader. These routines are written in machine code for the target architecture and configure parameters within target CPU that enable the execution of other programs as well as perform critical tasks such as setting up the stack, zeroing the BSS, and copying the data section of the kernel from long term storage to main memory. Once executed, the

start-up code jumps to the entry point of the kernel, usually defined under the symbol name ‘main’ for higher level system initialization such as the instantiation of device drivers and system periphery.

In other words,

Bootloader:

Is not part of your application software. It is a piece of code to load your application code into the controller/processor. It is used only when you want to load a new software (version) onto the controller.

Example: a bootloader may communicate with the PC (using USB or UART) to load your application software from PC into the microcontroller's program memory. After that it can be executed.

Start-up code:

Is a part of your application software. It is automatically generated by the compiler and added (at the program start) to your application software. Usually you don't know, nor you have to take care about it. The start-up code performs standard functions like data memory erase, stack pointer setup, standard peripheral setup. This is performed each time immediately after reset/power-on and may take only some milliseconds. You won't recognize this. Then automatically your application software starts.

52. Header Include Guard

Header Include Guard is used to disable multiple inclusion of header file into a C project. Typical syntax is as below,

my-header-file.h

```
#ifndef MY_HEADER_FILE_H

#define MY_HEADER_FILE_H

// Code body for header file

#endif
```

An example to illustrate the issue of multiple file/header inclusion is given below.

header-1.h

```
typedef struct {

    …

} MyStruct;

int myFunction(MyStruct *value);
```

header-2.h

```
#include "header-1.h"
```

```
int myFunction2(MyStruct *value);
```

main.c

```
#include "header-1.h"

#include "header-2.h"

int main() {

    // do something

}
```

This code has a serious problem: the detailed contents of MyStruct is defined twice, which is not allowed. This would result in a compilation error that can be difficult to track down, since one header file includes another. If you instead, did it with header guards:

header-1.h

```
#ifndef HEADER_1_H

#define HEADER_1_H

typedef struct {

    …

} MyStruct;

int myFunction(MyStruct *value);

#endif
```

header-2.h

```
#ifndef HEADER_2_H

#define HEADER_2_H

#include "header-1.h"

int myFunction2(MyStruct *value);

#endif
```

main.c

```
#include "header-1.h"

#include "header-2.h"

int main() {

    // do something
```

```
}
```

This would then expand to:

```
#ifndef HEADER_1_H
#define HEADER_1_H
typedef struct {
    …
} MyStruct;
int myFunction(MyStruct *value);
#endif
#ifndef HEADER_2_H
#define HEADER_2_H
#ifndef HEADER_1_H // Safe, since HEADER_1_H was #defined before.
#define HEADER_1_H
typedef struct {
    …
} MyStruct;
int myFunction(MyStruct *value);
#endif
int myFunction2(MyStruct *value);
#endif
int main() {
    // do something
}
```

When the compiler reaches the second inclusion of `header-1.h`, `HEADER_1_H` was already defined by the previous inclusion. So, now everything boils down to:

```
#define HEADER_1_H
typedef struct {
    …
} MyStruct;
```

```
int myFunction(MyStruct *value);

#define HEADER_2_H

int myFunction2(MyStruct *value);

int main() {

    // do something

}
```

And thus, there is no compilation error.

Note: There are multiple different conventions for naming the header guards. Some people like to name it `HEADER_2_H_`, some include the project name like `MY_PROJECT_HEADER_2_H`. The important thing is to ensure that the convention you follow makes it so that each file in your project has a unique header guard.

53. Memory Layout in C

After compiling a C program, a binary executable file(.exe) is created, and when we execute the program, this binary file loads into RAM in an organized manner. After being loaded into the RAM, memory layout in C Program has six components which are **text segment, initialized data segment, uninitialized data segment, command-line arguments, stack, and heap**. Each of these six different segments stores different parts of code and have **their own read, write permissions**. If a program tries to access the value stored in any segment differently than it is supposed to, it results in a **segmentation fault** error.

Scope of article

- This article discusses how a program is loaded into RAM when a C program executes, which helps programmers decide the amount of memory the program uses for its execution.
- This article explains each of these sections in memory layout in C with examples.
- This article does not discuss how a program compiles and the state of CPU registers and how their value change during program compilation.

Introduction

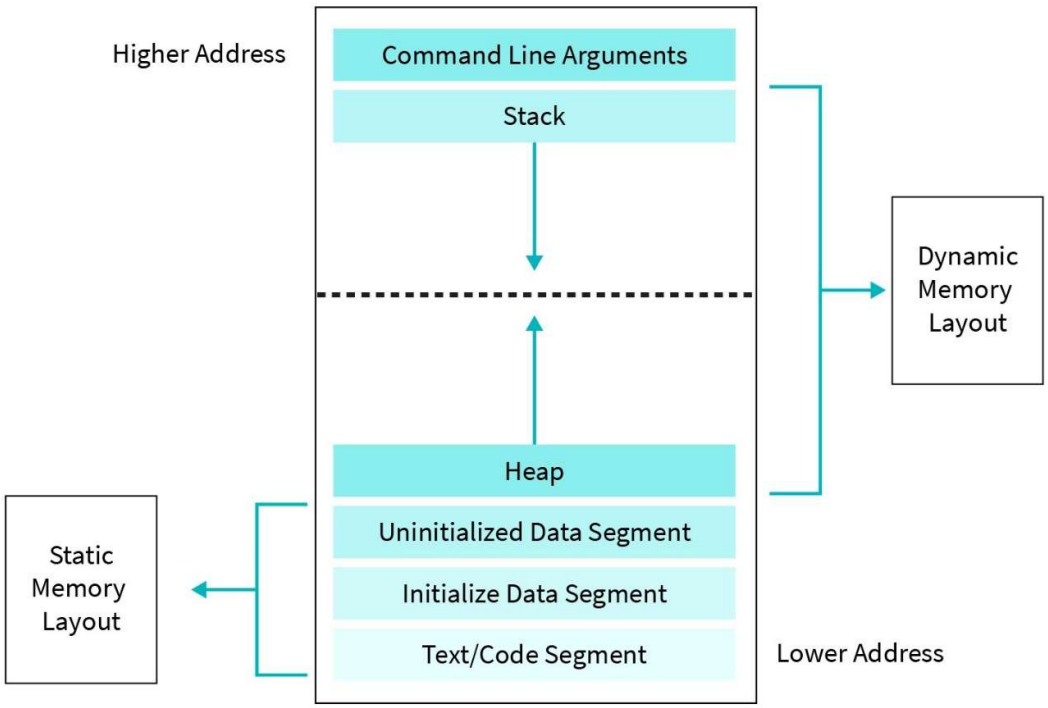
When we execute a C program, the executable code of the file loads into RAM in an organized manner. Computers do not access program instructions directly from secondary storage because the access time of secondary storage is longer when compared to that of RAM. RAM is faster than secondary storage but has a limited storage capacity, so it is necessary for programmers to utilize this limited storage efficiently. Knowledge of memory layout in C is helpful to programmers because they can decide the amount of memory utilized by the program for its execution.

A C program memory layout in C mainly comprises six components these are heap, stack, code segment, command-line arguments, uninitialized and initialized data segments. Each of these **segments has its own read and write permissions**. A segmentation fault occurs when a

program tries to access any of the segments in a way that is not allowed, which is also a common reason for the program to crash.

Diagram for memory structure of C

The diagram mentioned below shows a visual representation of how RAM loads a program written in C into several segments.



Memory Layout

Let us discuss each of these data segments in detail.

Text Segment

- After we compile the program, a binary file generates, which is used to execute our program by loading it into RAM. This binary file contains instructions, and these instructions get stored in the text segment of the memory.
- Text segment has read-only permission that prevents the program from accidental modifications.
- Text segment in RAM is shareable so that a single copy is required in the memory for frequent applications like a text editor, shells, etc.

Initialized data segment

Initialized data segment or **data segment** is part of the computer's virtual memory space of a C program that contains values of all external, global, static, and constant variables whose values are initialized at the time of variable declaration in the program. Because the values of variables can change during program execution, this memory segment has **read-write** permission. We can further

classify the data segment into the **read-write and read-only areas**. const variable comes under the read-only area. The remaining types of variables come in the read-write area. For example,

```
const hello* = "Data segment";
```

Here, the pointer variable hello comes under the read-write area, and the value of the string literal "Data segment" lies comes under initialized read-only data segment.

```
#include<stdio.h>


/* global variables stored in the read-write part of
   initialized data segment
 */
int global_var = 50;
char* hello = "Hello World";
/* global variables stored in the read-only part of
   initialized data segment
 */
const int global_var2 = 30;


int main() {
    // static variable stored in initialized data segment
    static int a = 10;
    // ...
    return 0;
}
```

In this example, variables global_var and pointer hello are declared outside the scope of main() function because of which they are stored in the read-write part of the initialized data segment but, global variable global_var2 is declared with the keyword const and hence it is stored in the read-only part of initialized data segment. Static variables like a are also stored in this part of the memory.

Uninitialized data segment

An uninitialized data segment is also known as **bss (block started by symbol)**. The program loaded allocates memory for this segment when it loads. Every data in bss is **initialized to arithmetic 0** and **pointers to null pointer** by the kernel before the C program executes. BSS also contains all

the static and global variables, initialized with arithmetic 0. Because values of variables stored in bss can be changed, this data segment has **read-write permissions**.

```
#include <stdio.h>


// Uninitialized global variable stored in the bss segment

int global_variable;


int main()

{

    // Uninitialized static variable stored in bss

    static int static_variable;


    // ..

    printf("global_variable = %d\n", global_variable);

    printf("static_variable = %d\n", static_variable);

    return 0;

}
```

Output

```
global_variable = 0

static_variable = 0
```

Here, both the variables global_variable and static_variables are uninitialized. Hence they are stored in the bss segment in the memory layout in C. Before the program execution begins, these values are initialized with value 0 by the kernel. This can be verified by printing the values of the variable as shown in the program.

Stack

The stack segment follows the LIFO (Last In First Out) structure and grows down to the lower address, but it depends on computer architecture. Stack **grows in the direction opposite to heap**. Stack segment **stores the value of local variables and values of parameters passed to a function** along with some additional information like the instruction's return address, which is to be executed after a function call.

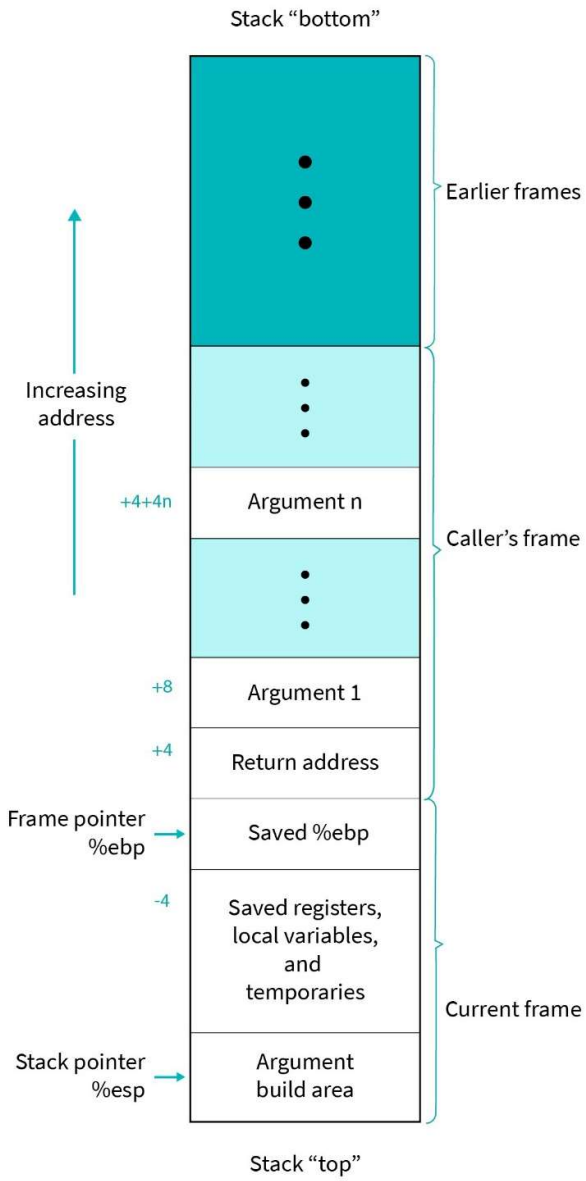
Stack pointer register keeps track of the top of the stack and its value change when push/pop actions are performed on the segment. The values are passed to stack when a function is called stack frame. **Stack frame** stores the value of function temporary variables and some automatic variables

that store extra information like the return address and details of the caller's environment (memory registers). Each time function calls itself recursively, a **new stack frame is created**, which allows a set of variables of one stack frame to not interfere with other variables of a different instance of the function. This is how recursive functions work.

Let us see an example to understand the variables stored in the stack memory segment.

```
#include<stdio.h>


void foo() {

    // local variables stored in the stack

    // when the function call is made

    int a, b;

}


int main() {

    // local variables stored in the stack

    int local = 5;

    char name[26];

    foo();

    // ..

    return 0;

}
```

Here, all the variables are stored in a stack memory layout in C because they are declared inside their parent function's scope. These variables only take the space in memory till their function is executed. For example, in the above code, the first main() starts its execution, and a stack frame for main() is made and pushed into the program stack with data of variables local and name. Then in main, we call foo, then another stack frame is made and pushed for it separately, which contains data of variables a and b. After the execution of foo, its stack frame is popped out, and its variable gets unallocated, and when the program ends, main's stack frame also gets popped out.



Heap

Heap is used for memory which is **allocated during the run time** (dynamically allocated memory). Heap generally begins at the end of bss segment and, they grow and shrink in the opposite direction of the Stack. Commands like malloc, calloc, free, realloc, etc are used to manage allocations in the heap segment which internally use sbrk and brk system calls to change memory allocation within the heap segment. Heap data segment is shared among modules loading dynamically and all the shared libraries in a process.

```
#include <stdio.h>

#include <stdlib.h>

int main() {
```

```
    // memory allocated in heap segment
    char *var = (char*) malloc ( sizeof(char) );
    // ..
    return 0;
}
```

Here, we create a variable of data type char by allocation memory of size 1 byte (equal to size of char in C) at the time of program execution. Because the variable is created dynamically such variables are initialized in the heap segment of the memory.

Command-line arguments

When a program executes with arguments passed from the console like argv and argc and other environment variables, the value of these **variables gets stored in this memory layout in C**.

```
#include<stdio.h>


int main(int argc, char *argv[]) {
    int i;


    // first value in argv stores file name
    printf("File name = %s\n", argv[0]);
    printf("Number of arguments passed = %d\n", argc-1);
    for(i = 1; i < argc; i++) {
        printf("Value of Argument_%d = %s\n", i, argv[i]);
    }


    return 0;
}
```

Output

```
~$ gcc file_1.c -o file_1
~$ ./file_1 100 23 43 69
File name = ./file_1
Number of arguments passed = 4
```

```
Value of Argument_1 = 100

Value of Argument_2 = 23

Value of Argument_3 = 43

Value of Argument_4 = 69
```

This example explains how command-line arguments are passed and used in the program. Here, this segment stores the value of variables argc and argv where argc stores the number of arguments passed and argv stores the value of actual parameters along with file name.

Examples

The size command is used to check the sizes (in bytes) of these different memory segments. Let us see some examples to visualize the memory layout in C, in detail.

A simple C program

```
#include<stdio.h>


int main() {

    return 0;

}
```

Output

```
~$ gcc file_1.c -o file_1

~$ size file_1

   text    data     bss     dec     hex filename

   1418     544       8    1970     7b2 file_1
```

Let us now add a global variable

```
#include<stdio.h>


int global_variable = 5;


int main() {

    return 0;
```

```
}
```

Output

```
~$ gcc file_1.c -o file_1

~$ size file_1

   text    data     bss     dec     hex filename

   1418     548       4    1970     7b2 file_1
```

Adding one global variable increased memory allocated by data segment (Initialized data segment) by 4 bytes, which is the actual memory size of 1 variable of type integer (sizeof(global_variable)).

Let us add one uninitialized static variable that should increase memory occupied by bss.

```
#include<stdio.h>


int global_variable = 5;


int main() {

    static int static_variable_1;

    return 0;

}
```

Output

```
~$gcc file_1.c -o file_1

~$ size file_1

   text    data     bss     dec     hex filename

   1418     548      12    1978     7ba file_1
```

But if we add a static variable with an initialized value, it will be stored in the data segment.

```
#include<stdio.h>


int global_variable = 5;
```

```
int main() {
    static int static_variable_1;
    static int static_variable_2 = 12;
    return 0;
}
```

Output

```
~$gcc file_1.c -o file_1
~$ size file_1
   text    data     bss     dec     hex filename
   1418     552      8    1978     7ba file_1
```

Similarly, if we add a global variable with an uninitialized value, it will be stored in bss.

```
#include<stdio.h>

int global_variable = 5;
int global_variable_in_bss;

int main() {
    static int static_variable_1;
    static int static_variable_2 = 12;
    return 0;
}
```

Output

```
~$gcc file_1.c -o file_1
~$ size file_1
   text    data     bss     dec     hex filename
   1418     552      16    1978     7ba file_1
```

Also, we have classified initialized data segment into two parts:

- read-only area
- read-write area

Let us see two C programs to understand this classification.

Program 1

```
#include <stdio.h>


/*
 * created in the read-write area
 */
char str[] = "Hello";


int main() {
    printf("%s\n",str);


    str[0] = 'Y';


    printf("%s\n",str);
    return 0;
}
```

Output

```
Hello
Yello
```

Program 2

```
#include <stdio.h>


/*
 * created in a read-only area
```

```
 */
char* str = "Hello";


int main() {
    printf("%s\n",str);


    str[0] = 'Y';


    printf("%s\n",str);
    return 0;
}
```

Output

```
Segmentation fault (core dumped)
```

In the first example, the global variable str is a character array and we can change its value but in the second case, we cannot change the character of the string because variable str is string literal and stored in the read-write area of the data segment because of which the second program throws an error.

Conclusion

- When a program in C is executed, binary code is loaded into RAM and is segregated into five different areas which are text segment, initialized data segment, uninitialized data segment, command-line arguments, stack, and heap.
- Code instructions are stored in text segment and this is shareable memory. If arguments are passed when code is executed from the console, the values of arguments are stored in the command line arguments area in memory.
- Initialized data segment stores global, static, external variables that are initialized beforehand in the program. Uninitialized data segment or bss contains all the uninitialized global and static variables.
- Stack stores all local variables and arguments of functions. They also store a function return address of the instruction, which is to be executed after a function call.
- Stack and heap grow opposite to each other.
- Heap stores all dynamically allocated memory in the program and is managed by commands like malloc, calloc, free etc.

54. Passing an array to a function

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

```
void helper_1(int arr[], int element)  //int arr[] creates a pointer
to the first element of the array. Arry is always passed by
reference.

{

    printf("Size of arr inside helper_1: %d\n", sizeof(arr));  //arr
is just a pointer pointing to the first element of the array. Size
of arr is just a word, irrespective of what it points to. 8 bytes in
64 bit architecture.

    for(int i = 0; i < element; i++)

    {

        printf("%d ", arr[i]);

    }

    printf("\n");

}


void helper_2(int *arr, int element)  //Arry is always passed by
reference.

{

    printf("Size of arr inside helper_2: %d\n", sizeof(arr));  //arr
is just a pointer pointing to the first element of the array. Size
of arr is just a word, irrespective of what it points to. 8 bytes in
64 bit architecture.

    for(int i = 0; i < element; i++)

    {

        printf("%d ", arr[i]);

    }

    printf("\n");

}


int main(void)

{

    int arr[] = {1, 2, 3, 4};
```

```
    int element = sizeof(arr)/sizeof(arr[0]);  //Fnding the number
of elements in the array.

    printf("Size of int: %d\n", sizeof(int));

    printf("Size of arr: %d\n", sizeof(arr));

    helper_1(arr, element);  //Passing arr to the function.

    helper_2(arr, element);  //Passing arr to the function.

    return(0);

}
```

Output:

```
Size of int: 4

Size of arr: 16

Size of arr inside helper_1: 8

1 2 3 4

Size of arr inside helper_2: 8

1 2 3 4
```

55. Normal declaration of array of pointers vs. array allocated with malloc()

The first is the difference between declaring pointer to an array as

```
int num[n];

int (*array)[n];

array = &num;
```

and

```
int* array = malloc(n * sizeof(int));
```

In the first version, you are declaring an object with automatic storage duration. This means that the array lives only as long as the function that calls it exists. In the second version, you are getting memory with dynamic storage duration, which means that it will exist until it is explicitly deallocated with `free()`.

The reason that the second version works here is an implementation detail of how C is usually compiled. Typically, C memory is split into several regions, including the stack (for function calls and local variables) and the heap (for malloced objects). The stack typically has a much smaller size than the heap; usually it's something like 8MB. As a result, if you try to allocate a huge array with:

```
int num[n];

int (*array)[n];

array = &num;
```

Then you might exceed the stack's storage space, causing the segmentation fault. On the other hand, the heap usually has a huge size (say, as much space as is free on the system), and so mallocing a large object won't cause an out-of-memory error.

In general, be careful with variable-length arrays in C. They can easily exceed stack size. Prefer malloc unless you know the size is small or that you really only do want the array for a short period of time.