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 –

Live Demo

```
#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( ) {
   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 {
   type [member_name] : width ;
};
```

The following table describes the variable elements of a bit field –

Sr.No.	Element & Description
1	type

	An integer type that determines how a bit-field's value is interpreted. The type may be int, signed int, or unsigned int.
2	member_name The name of the bit-field.
3	width The number of bits in the bit-field. The width must be less than or equal to the bit width of the specified type.

The variables defined with a predefined width are called bit fields. A bit field can hold more than a single bit; for example, if you need a variable to store a value from 0 to 7, then you can define a bit field with a width of 3 bits as follows –

```
struct {
   unsigned int age : 3;
} Age;
```

The above structure definition instructs the C compiler that the age variable is going to use only 3 bits to store the value. If you try to use more than 3 bits, then it will not allow you to do so. Let us try the following example –

Live Demo

```
#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;
}
```

When the above code is compiled it will compile with a warning and when executed, it produces the following result –

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

Typedef

The C programming language provides a keyword called typedef, which you can use to give a type a new name. Following is an example to define a term BYTE for one-byte numbers –

```
typedef unsigned char BYTE;
```

After this type definition, the identifier BYTE can be used as an abbreviation for the type unsigned char, for example..

```
BYTE  b1, b2;
```

By convention, uppercase letters are used for these definitions to remind the user that the type name is really a symbolic abbreviation, but you can use lowercase, as follows –

```
typedef unsigned char byte;
```

You can use typedef to give a name to your user defined data types as well. For example, you can use typedef with structure to define a new data type and then use that data type to define structure variables directly as follows –

Live Demo

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

typedef struct Books {
   char title[50];
   char author[50];
   char subject[100];
   int book_id;
} Book;

int main( ) {

   Book book;

   strcpy( book.title, "C Programming");
   strcpy( book.author, "Nuha Ali");
   strcpy( book.subject, "C Programming Tutorial");
   book.book_id = 6495407;

   printf( "Book title : %s\n", book.title);
   printf( "Book author : %s\n", book.author);
   printf( "Book subject : %s\n", book.subject);
   printf( "Book book_id : %d\n", book.book_id);

   return 0;
}
```

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

```
Book  title : C Programming
Book  author : Nuha Ali
Book  subject : C Programming Tutorial
Book  book_id : 6495407
```

typedef vs #define

#define is a C-directive which is also used to define the aliases for various data types similar to typedef but with the following differences –

- typedef is limited to giving symbolic names to types only where as #define can be used to define alias for values as well, q., you can define 1 as ONE etc.
- typedef interpretation is performed by the compiler whereas #define statements are processed by the pre-processor.

The following example shows how to use #define in a program –

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

#define TRUE  1
#define FALSE 0

int main( ) {
   printf( "Value of TRUE : %d\n", TRUE);
   printf( "Value of FALSE : %d\n", FALSE);

   return 0;
}
```

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

```
Value of TRUE : 1
Value of FALSE : 0
```

C - Input and Output

When we say Input, it means to feed some data into a program. An input can be given in the form of a file or from the command line. C programming provides a set of built-in functions to read the given input and feed it to the program as per requirement.

When we say Output, it means to display some data on screen, printer, or in any file. C programming provides a set of built-in functions to output the data on the computer screen as well as to save it in text or binary files.

The Standard Files

C programming treats all the devices as files. So devices such as the display are addressed in the same way as files and the following three files are automatically opened when a program executes to provide access to the keyboard and screen.

Standard File	File Pointer	Device
Standard input	stdin	Keyboard
Standard output	stdout	Screen
Standard error	stderr	Your screen

The file pointers are the means to access the file for reading and writing purpose. This section explains how to read values from the screen and how to print the result on the screen.

The getchar() and putchar() Functions

The int getchar(void) function reads the next available character from the screen and returns it as an integer. This function reads only single character at a time. You can use this method in the loop in case you want to read more than one character from the screen.

The int putchar(int c) function puts the passed character on the screen and returns the same character. This function puts only single character at a time. You can use this method in the loop in case you want to display more than one character on the screen. Check the following example –

```
#include <stdio.h>
int main( ) {

   int c;

   printf( "Enter a value :");
   c = getchar( );

   printf( "\nYou entered: ");
   putchar( c );

   return 0;
}
```

When the above code is compiled and executed, it waits for you to input some text. When you enter a text and press enter, then the program proceeds and reads only a single character and displays it as follows –

```
$./a.out
Enter a value : this is test
You entered: t
```

The gets() and puts() Functions

The char *gets(char *s) function reads a line from stdin into the buffer pointed to by s until either a terminating newline or EOF (End of File).

The int puts(const char *s) function writes the string 's' and 'a' trailing newline to stdout.

NOTE: Though it has been deprecated to use gets() function, Instead of using gets, you want to use fgets().

```
#include <stdio.h>
int main( ) {

   char str[100];

   printf( "Enter a value :");
   gets( str );

   printf( "\nYou entered: ");
   puts( str );

   return 0;
}
```

When the above code is compiled and executed, it waits for you to input some text. When you enter a text and press enter, then the program proceeds and reads the complete line till end, and displays it as follows –

```
$./a.out
Enter a value : this is test
You entered: this is test
```

The scanf() and printf() Functions

The int scanf(const char *format, ...) function reads the input from the standard input stream stdin and scans that input according to the format provided.

The int printf(const char *format, ...) function writes the output to the standard output stream stdout and produces the output according to the format provided.

The format can be a simple constant string, but you can specify %s, %d, %c, %f, etc., to print or read strings, integer, character or float respectively. There are many other formatting options available which can be used based on requirements. Let us now proceed with a simple example to understand the concepts better –

```
#include <stdio.h>
int main( ) {

   char str[100];
   int i;

   printf( "Enter a value :");
   scanf("%s %d", str, &i);

   printf( "\nYou entered: %s %d ", str, i);

   return 0;
}
```

When the above code is compiled and executed, it waits for you to input some text. When you enter a text and press enter, then program proceeds and reads the input and displays it as follows –

```
$./a.out
Enter a value : seven 7
You entered: seven 7
```

Here, it should be noted that scanf() expects input in the same format as you provided %s and %d, which means you have to provide valid inputs like "string integer". If you provide "string string" or "integer integer", then it will be assumed as wrong input.

Secondly, while reading a string, scanf() stops reading as soon as it encounters a space, so "this is test" are three strings for scanf().

C - Preprocessors

Previous Page

Next Page

The C Preprocessor is not a part of the compiler, but is a separate step in the compilation process. In simple terms, a C Preprocessor is just a text substitution tool and it instructs the compiler to do required pre-processing before the actual compilation. We'll refer to the C Preprocessor as CPP.

All preprocessor commands begin with a hash symbol (#). It must be the first nonblank character, and for readability, a preprocessor directive should begin in the first column. The following section lists down all the important preprocessor directives –

Sr.No.	Directive & Description
1	#define Substitutes a preprocessor macro.

2	#include Inserts a particular header from another file.
3	#undef Undefines a preprocessor macro.
4	#ifdef Returns true if this macro is defined.
5	#ifndef Returns true if this macro is not defined.
6	#if Tests if a compile time condition is true.
7	#else The alternative for #if.
8	#elif #else and #if in one statement.

9	#endif Ends preprocessor conditional.
10	#error Prints error message on stderr.
11	#pragma Issues special commands to the compiler, using a standardized method.

Preprocessors Examples

Analyze the following examples to understand various directives.

```
#define MAX_ARRAY_LENGTH 20
```

This directive tells the CPP to replace instances of MAX_ARRAY_LENGTH with 20. Use *#define* for constants to increase readability.

```
#include <stdio.h>

#include "myheader.h"
```

These directives tell the CPP to get stdio.h from System Libraries and add the text to the current source file. The next line tells CPP to get myheader.h from the local directory and add the content to the current source file.

```
#undef  FILE_SIZE
```

```
#define FILE_SIZE 42
```

It tells the CPP to undefine existing FILE_SIZE and define it as 42.

```
#ifndef MESSAGE
  #define MESSAGE "You wish!"
#endif
```

It tells the CPP to define MESSAGE only if MESSAGE isn't already defined.

```
#ifdef DEBUG
  /* Your debugging statements here */
#endif
```

It tells the CPP to process the statements enclosed if DEBUG is defined. This is useful if you pass the *-DDEBUG* flag to the gcc compiler at the time of compilation. This will define DEBUG, so you can turn debugging on and off on the fly during compilation.

Predefined Macros

ANSI C defines a number of macros. Although each one is available for use in programming, the predefined macros should not be directly modified.

Sr.No.	Macro & Description
1	__DATE__ The current date as a character literal in "MMM DD YYYY" format.

2	__TIME__ The current time as a character literal in "HH:MM:SS" format.
3	__FILE__ This contains the current filename as a string literal.
4	__LINE__ This contains the current line number as a decimal constant.
5	__STDC__ Defined as 1 when the compiler complies with the ANSI standard.

Let's try the following example –

Live Demo

```
#include <stdio.h>


int main() {


   printf("File :%s\n", __FILE__ );

   printf("Date :%s\n", __DATE__ );

   printf("Time :%s\n", __TIME__ );

   printf("Line :%d\n", __LINE__ );
```

```
  printf("ANSI :%d\n", __STDC__ );


}
```

When the above code in a file test.c is compiled and executed, it produces the following result –

```
File :test.c

Date :Jun 2 2012

Time :03:36:24

Line :8

ANSI :1
```

Preprocessor Operators

The C preprocessor offers the following operators to help create macros –

The Macro Continuation (\) Operator

A macro is normally confined to a single line. The macro continuation operator (\) is used to continue a macro that is too long for a single line. For example –

```
#define  message_for(a, b)  \
  printf(#a " and " #b ": We love you!\n")
```

The Stringize (#) Operator

The stringize or number-sign operator ('#'), when used within a macro definition, converts a macro parameter into a string constant. This operator may be used only in a macro having a specified argument or parameter list. For example –

Live Demo

```
#include <stdio.h>

#define  message_for(a, b)  \
   printf(#a " and " #b ": We love you!\n")

int main(void) {
   message_for(Carole, Debra);
   return 0;
}
```

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

```
Carole and Debra: We love you!
```

The Token Pasting (##) Operator

The token-pasting operator (##) within a macro definition combines two arguments. It permits two separate tokens in the macro definition to be joined into a single token. For example –

Live Demo

```c
#include <stdio.h>

#define tokenpaster(n) printf ("token" #n " = %d", token##n)

int main(void) {
  int token34 = 40;
  tokenpaster(34);
  return 0;
}
```

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

```
token34 = 40
```

It happened so because this example results in the following actual output from the preprocessor –

```
printf ("token34 = %d", token34);
```

This example shows the concatenation of token##n into token34 and here we have used both stringize and token-pasting.

The Defined() Operator

The preprocessor defined operator is used in constant expressions to determine if an identifier is defined using #define. If the specified identifier is defined, the value is true

(non-zero). If the symbol is not defined, the value is false (zero). The defined operator is specified as follows –

Live Demo

```
#include <stdio.h>

#if !defined (MESSAGE)
  #define MESSAGE "You wish!"
#endif

int main(void) {
  printf("Here is the message: %s\n", MESSAGE);
  return 0;
}
```

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

```
Here is the message: You wish!
```

Parameterized Macros

One of the powerful functions of the CPP is the ability to simulate functions using parameterized macros. For example, we might have some code to square a number as follows –

```
int square(int x) {
  return x * x;
}
```

We can rewrite above the code using a macro as follows –

```
#define square(x) ((x) * (x))
```

Macros with arguments must be defined using the #define directive before they can be used. The argument list is enclosed in parentheses and must immediately follow the macro name. Spaces are not allowed between the macro name and open parenthesis. For example –

Live Demo

```
#include <stdio.h>

#define MAX(x,y) ((x) > (y) ? (x) : (y))

int main(void) {
  printf("Max between 20 and 10 is %d\n", MAX(10, 20));
  return 0;
}
```

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

```
Max between 20 and 10 is 20
```

C - Header Files

Previous Page

Next Page

A header file is a file with extension .h which contains C function declarations and macro definitions to be shared between several source files. There are two types of header files: the files that the programmer writes and the files that comes with your compiler.

You request to use a header file in your program by including it with the C preprocessing directive #include, like you have seen inclusion of stdio.h header file, which comes along with your compiler.

Including a header file is equal to copying the content of the header file but we do not do it because it will be error-prone and it is not a good idea to copy the content of a header file in the source files, especially if we have multiple source files in a program.

A simple practice in C or C++ programs is that we keep all the constants, macros, system wide global variables, and function prototypes in the header files and include that header file wherever it is required.

Include Syntax

Both the user and the system header files are included using the preprocessing directive #include. It has the following two forms –

```
#include <file>
```

This form is used for system header files. It searches for a file named 'file' in a standard list of system directories. You can prepend directories to this list with the -I option while compiling your source code.

```
#include "file"
```

This form is used for header files of your own program. It searches for a file named 'file' in the directory containing the current file. You can prepend directories to this list with the -I option while compiling your source code.

Include Operation

The #include directive works by directing the C preprocessor to scan the specified file as input before continuing with the rest of the current source file. The output from the preprocessor contains the output already generated, followed by the output resulting from the included file, followed by the output that comes from the text after the #include directive. For example, if you have a header file header.h as follows –

```
char *test (void);
```

and a main program called *program.c* that uses the header file, like this –

```
int x;
#include "header.h"
```

```
int main (void) {
  puts (test ());
}
```

the compiler will see the same token stream as it would if program.c read.

```
int x;
char *test (void);

int main (void) {
  puts (test ());
}
```

Once-Only Headers

If a header file happens to be included twice, the compiler will process its contents twice and it will result in an error. The standard way to prevent this is to enclose the entire real contents of the file in a conditional, like this –

```
#ifndef HEADER_FILE
#define HEADER_FILE

the entire header file file

#endif
```

This construct is commonly known as a wrapper #ifndef. When the header is included again, the conditional will be false, because HEADER_FILE is defined. The preprocessor will skip over the entire contents of the file, and the compiler will not see it twice.

Computed Includes

Sometimes it is necessary to select one of the several different header files to be included into your program. For instance, they might specify configuration parameters to be used on different sorts of operating systems. You could do this with a series of conditionals as follows –

```
#if SYSTEM_1
  # include "system_1.h"
#elif SYSTEM_2
  # include "system_2.h"
#elif SYSTEM_3
  ...
#endif
```

But as it grows, it becomes tedious, instead the preprocessor offers the ability to use a macro for the header name. This is called a computed include. Instead of writing a header name as the direct argument of #include, you simply put a macro name there –

```
#define SYSTEM_H "system_1.h"
...
```

```
#include SYSTEM_H
```

SYSTEM_H will be expanded, and the preprocessor will look for system_1.h as if the #include had been written that way originally. SYSTEM_H could be defined by your Makefile with a -D option.

C - Type Casting

Previous Page

Next Page

Converting one datatype into another is known as type casting or, type-conversion. 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 –

Live Demo

```
#include <stdio.h>

main() {
```

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

   double mean;


   mean = (double) sum / count;

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

}
```

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.

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 –

Live Demo

```
#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 –

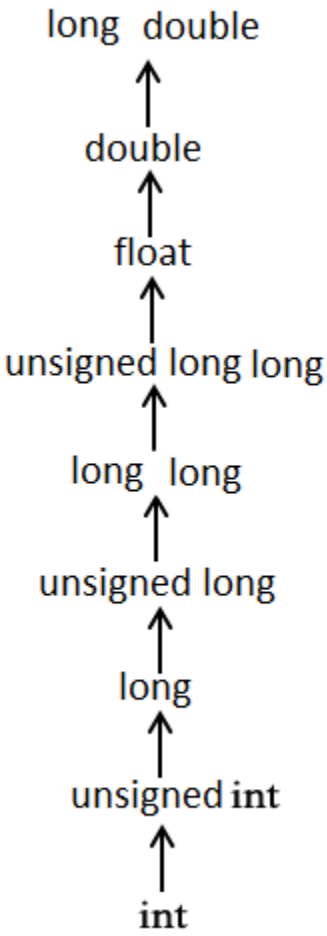
```
Value of sum : 116
```

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 –

Live Demo

```
#include <stdio.h>


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.

C - Error Handling

Previous Page

Next Page

As such, C programming does not provide direct support for error handling but being a system programming language, it provides you access at lower level in the form of

return values. Most of the C or even Unix function calls return -1 or NULL in case of any error and set an error code errno. It is set as a global variable and indicates an error occurred during any function call. You can find various error codes defined in <error.h> header file.

So a C programmer can check the returned values and can take appropriate action depending on the return value. It is a good practice, to set errno to 0 at the time of initializing a program. A value of 0 indicates that there is no error in the program.

errno, perror(). and strerror()

The C programming language provides perror() and strerror() functions which can be used to display the text message associated with errno.

- The perror() function displays the string you pass to it, followed by a colon, a space, and then the textual representation of the current errno value.
- The strerror() function, which returns a pointer to the textual representation of the current errno value.

Let's try to simulate an error condition and try to open a file which does not exist. Here I'm using both the functions to show the usage, but you can use one or more ways of printing your errors. Second important point to note is that you should use stderr file stream to output all the errors.

```
#include <stdio.h>

#include <errno.h>

#include <string.h>
```

```c
extern int errno ;

int main () {

  FILE * pf;
  int errnum;
  pf = fopen ("unexist.txt", "rb");

  if (pf == NULL) {

    errnum = errno;
    fprintf(stderr, "Value of errno: %d\n", errno);
    perror("Error printed by perror");
    fprintf(stderr, "Error opening file: %s\n", strerror( errnum ));
  } else {

    fclose (pf);
  }
```

```
  return 0;

}
```

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

```
Value of errno: 2

Error printed by perror: No such file or directory

Error opening file: No such file or directory
```

Divide by Zero Errors

It is a common problem that at the time of dividing any number, programmers do not check if a divisor is zero and finally it creates a runtime error.

The code below fixes this by checking if the divisor is zero before dividing –

Live Demo

```
#include <stdio.h>

#include <stdlib.h>


main() {


  int dividend = 20;

  int divisor = 0;

  int quotient;
```

```
  if( divisor == 0){
     fprintf(stderr, "Division by zero! Exiting...\n");
     exit(-1);
  }

  quotient = dividend / divisor;
  fprintf(stderr, "Value of quotient : %d\n", quotient );

  exit(0);
}
```

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

```
Division by zero! Exiting...
```

Program Exit Status

It is a common practice to exit with a value of EXIT_SUCCESS in case of program coming out after a successful operation. Here, EXIT_SUCCESS is a macro and it is defined as 0.

If you have an error condition in your program and you are coming out then you should exit with a status EXIT_FAILURE which is defined as -1. So let's write above program as follows –

Live Demo

```c
#include <stdio.h>

#include <stdlib.h>


main() {


   int dividend = 20;

   int divisor = 5;

   int quotient;


   if( divisor == 0) {

      fprintf(stderr, "Division by zero! Exiting...\n");

      exit(EXIT_FAILURE);

   }

   quotient = dividend / divisor;

   fprintf(stderr, "Value of quotient : %d\n", quotient );


   exit(EXIT_SUCCESS);

}
```

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

```
Value of quotient : 4
```

C - Variable Arguments

Previous Page

Next Page

Sometimes, you may come across a situation, when you want to have a function, which can take variable number of arguments, i.e., parameters, instead of predefined number of parameters. The C programming language provides a solution for this situation and you are allowed to define a function which can accept variable number of parameters based on your requirement. The following example shows the definition of such a function.

```
int func(int, ... ) {
   .
   .
   .
}

int main() {
   func(1, 2, 3);
```

```
  func(1, 2, 3, 4);
}
```

It should be noted that the function func() has its last argument as ellipses, i.e. three dotes (...) and the one just before the ellipses is always an int which will represent the total number variable arguments passed. To use such functionality, you need to make use of stdarg.h header file which provides the functions and macros to implement the functionality of variable arguments and follow the given steps –

- Define a function with its last parameter as ellipses and the one just before the ellipses is always an int which will represent the number of arguments.
- Create a va_list type variable in the function definition. This type is defined in stdarg.h header file.
- Use int parameter and va_start macro to initialize the va_list variable to an argument list. The macro va_start is defined in stdarg.h header file.
- Use va_arg macro and va_list variable to access each item in argument list.
- Use a macro va_end to clean up the memory assigned to va_list variable.

Now let us follow the above steps and write down a simple function which can take the variable number of parameters and return their average –

Live Demo

```
#include <stdio.h>

#include <stdarg.h>
```

```
double average(int num,...) {

  va_list valist;
  double sum = 0.0;
  int i;

  /* initialize valist for num number of arguments */
  va_start(valist, num);

  /* access all the arguments assigned to valist */
  for (i = 0; i < num; i++) {
     sum += va_arg(valist, int);
  }

  /* clean memory reserved for valist */
  va_end(valist);

  return sum/num;
}
```

```
int main() {

  printf("Average of 2, 3, 4, 5 = %f\n", average(4, 2,3,4,5));

  printf("Average of 5, 10, 15 = %f\n", average(3, 5,10,15));

}
```

When the above code is compiled and executed, it produces the following result. It should be noted that the function average() has been called twice and each time the first argument represents the total number of variable arguments being passed. Only ellipses will be used to pass variable number of arguments.

```
Average of 2, 3, 4, 5 = 3.500000

Average of 5, 10, 15 = 10.000000
```

C - Memory Management

Previous Page

Next Page

This chapter explains dynamic memory management in C. The C programming language provides several functions for memory allocation and management. These functions can be found in the <stdlib.h> header file.

Sr.No.	Function & Description

1	void *calloc(int num, int size); This function allocates an array of num elements each of which size in bytes will be size.
2	void free(void *address); This function releases a block of memory block specified by address.
3	void *malloc(size_t size); This function allocates an array of num bytes and leave them uninitialized.
4	void *realloc(void *address, int newsize); This function re-allocates memory extending it upto newsize.

Allocating Memory Dynamically

While programming, if you are aware of the size of an array, then it is easy and you can define it as an array. For example, to store a name of any person, it can go up to a maximum of 100 characters, so you can define something as follows –

```
char name[100];
```

But now let us consider a situation where you have no idea about the length of the text you need to store, for example, you want to store a detailed description about a topic. Here we need to define a pointer to character without defining how much memory is

required and later, based on requirement, we can allocate memory as shown in the below example –

Live Demo

```
#include <stdio.h>

#include <string.h>


int main() {


   char name[100];

   char *description;


   strcpy(name, "Zara Ali");


   /* allocate memory dynamically */

   description = malloc( 200 * sizeof(char) );


   if( description == NULL ) {

      fprintf(stderr, "Error - unable to allocate required memory\n");

   } else {

      strcpy( description, "Zara ali a DPS student in class 10th");
```

```
  }

  printf("Name = %s\n", name );
  printf("Description: %s\n", description );
}
```

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

```
Name = Zara Ali
Description: Zara ali a DPS student in class 10th
```

Same program can be written using calloc(); only thing is you need to replace malloc with calloc as follows –

```
calloc(200, sizeof(char));
```

So you have complete control and you can pass any size value while allocating memory, unlike arrays where once the size defined, you cannot change it.

Resizing and Releasing Memory

When your program comes out, operating system automatically release all the memory allocated by your program but as a good practice when you are not in need of memory anymore then you should release that memory by calling the function free().

Alternatively, you can increase or decrease the size of an allocated memory block by calling the function realloc(). Let us check the above program once again and make use of realloc() and free() functions –

Live Demo

```
#include <stdio.h>

#include <stdlib.h>

#include <string.h>


int main() {


   char name[100];

   char *description;


   strcpy(name, "Zara Ali");


   /* allocate memory dynamically */

   description = malloc( 30 * sizeof(char) );

   if( description == NULL ) {

      fprintf(stderr, "Error - unable to allocate required memory\n");

   } else {
```

```
    strcpy( description, "Zara ali a DPS student.");
  }

  /* suppose you want to store bigger description */
  description = realloc( description, 100 * sizeof(char) );

  if( description == NULL ) {
    fprintf(stderr, "Error - unable to allocate required memory\n");
  } else {
    strcat( description, "She is in class 10th");
  }

  printf("Name = %s\n", name );
  printf("Description: %s\n", description );

  /* release memory using free() function */
  free(description);
}
```

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

```
Name = Zara Ali
```

```
Description: Zara ali a DPS student.She is in class 10th
```

You can try the above example without re-allocating extra memory, and strcat() function will give an error due to lack of available memory in description.

C - Command Line Arguments

Previous Page

Next Page

It is possible to pass some values from the command line to your C programs when they are executed. These values are called command line arguments and many times they are important for your program especially when you want to control your program from outside instead of hard coding those values inside the code.

The command line arguments are handled using main() function arguments where argc refers to the number of arguments passed, and argv[] is a pointer array which points to each argument passed to the program. Following is a simple example which checks if there is any argument supplied from the command line and take action accordingly –

```
#include <stdio.h>


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


  if( argc == 2 ) {
```

```
    printf("The argument supplied is %s\n", argv[1]);
  }
  else if( argc > 2 ) {
    printf("Too many arguments supplied.\n");
  }
  else {
    printf("One argument expected.\n");
  }
}
```

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

```
$./a.out testing
The argument supplied is testing
```

When the above code is compiled and executed with a two arguments, it produces the following result.

```
$./a.out testing1 testing2
Too many arguments supplied.
```

When the above code is compiled and executed without passing any argument, it produces the following result.

```
$./a.out
One argument expected
```

It should be noted that argv[0] holds the name of the program itself and argv[1] is a pointer to the first command line argument supplied, and *argv[n] is the last argument. If no arguments are supplied, argc will be one, and if you pass one argument then argc is set at 2.

You pass all the command line arguments separated by a space, but if argument itself has a space then you can pass such arguments by putting them inside double quotes "" or single quotes ''. Let us re-write above example once again where we will print program name and we also pass a command line argument by putting inside double quotes –

```
#include <stdio.h>

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

  printf("Program name %s\n", argv[0]);

  if( argc == 2 ) {
    printf("The argument supplied is %s\n", argv[1]);
  }
  else if( argc > 2 ) {
    printf("Too many arguments supplied.\n");
  }
```

```
  else {
     printf("One argument expected.\n");
  }
}
```

When the above code is compiled and executed with a single argument separated by space but inside double quotes, it produces the following result.

```
$./a.out "testing1 testing2"


Program name ./a.out
The argument supplied is testing1 testing2
```