Tutorial para Iniciantes

C é uma linguagem de alto nível de uso geral que foi originalmente desenvolvida por Dennis M. Ritchie para desenvolver o sistema operacional UNIX no Bell Labs. C foi originalmente implementado pela primeira vez no computador DEC PDP-11 em 1972.

Em 1978, Brian Kernighan e Dennis Ritchie produziram a primeira descrição publicamente disponível de C, agora conhecida como o padrão K & R.

O sistema operacional UNIX, o compilador C e essencialmente todos os programas aplicativos UNIX foram escritos em C. C agora se tornou uma linguagem profissional amplamente utilizada por várias razões -

- Fácil de aprender
- Linguagem estruturada
- Produz programas eficientes
- Pode lidar com atividades de baixo nível
- Pode ser compilado em uma variedade de plataformas de computador

Fatos sobre C

C foi inventado para escrever um sistema operacional chamado UNIX.

C é um sucessor da linguagem B, que foi introduzida por volta do início dos anos 70.

A linguagem foi formalizada em 1988 pelo American National Standard Institute (ANSI).

O sistema operacional UNIX foi totalmente escrito em C.

Hoje C é a linguagem de programação do sistema mais utilizada e popular.

A maioria dos softwares de última geração foram implementados usando C.

O Linux OS e o RDBMS MySQL mais populares de hoje foram escritos em C.

Por que usar C?

C foi inicialmente usado para o trabalho de desenvolvimento de sistemas, particularmente os programas que compõem o sistema operacional. C foi adotado como uma linguagem de desenvolvimento de sistema porque produz código que é executado quase tão rapidamente quanto o código escrito em linguagem assembly. Alguns exemplos do uso de C podem ser -

- Sistemas operacionais
- Compiladores de Linguagem
- Montadores
- Editores de texto
- Spoolers de impressão
- Drivers de rede
- Programas modernos
- Bancos de dados
- Interpretadores de Linguagem

Serviços de utilidade pública

O programa AC pode variar de 3 linhas para milhões de linhas e deve ser C Programas escrito em um ou mais arquivos de texto com extensão **".c"** ; por exemplo, *hello.c* . Você pode usar **"vi"** , **"vim"** ou qualquer outro editor de texto para escrever seu programa C em um arquivo.

Este tutorial pressupõe que você saiba como editar um arquivo de texto e como escrever código-fonte dentro de um arquivo de programa.

C - Configuração do Ambiente

Se você quiser configurar seu ambiente para a linguagem de Configuração do ambiente local programação C, precisará das duas ferramentas de software a seguir disponíveis no seu computador: (a) Editor de texto e (b) O compilador C.

Isso será usado para digitar seu programa. Exemplos de alguns editores Editor de texto incluem o Windows Notepad, o comando OS Edit, Brief, Epsilon, EMACS e vim ou vi.

O nome e a versão dos editores de texto podem variar em diferentes sistemas operacionais. Por exemplo, o Bloco de Notas será usado no Windows e o vim ou o vi poderão ser usados no Windows, bem como no Linux ou UNIX.

Os arquivos criados com o editor são chamados de arquivos de origem e contêm os códigos-fonte do programa. Os arquivos de origem para programas em C geralmente são nomeados com a extensão " **.c** ".

Antes de iniciar sua programação, certifique-se de ter um editor de texto no lugar e você tem experiência suficiente para escrever um programa de computador, salvá-lo em um arquivo, compilá-lo e, finalmente, executá-lo.

O código fonte escrito no arquivo fonte é a fonte legível para o seu O compilador C programa. Ele precisa ser "compilado", em linguagem de máquina para que sua CPU possa realmente executar o programa de acordo com as instruções dadas.

O compilador compila os códigos-fonte em programas executáveis finais. O compilador mais utilizado e gratuito é o compilador GNU C / C ++, caso contrário você pode ter compiladores da HP ou Solaris se você tiver os respectivos sistemas operacionais.

A seção a seguir explica como instalar o compilador GNU C / C ++ em vários sistemas operacionais. Continuamos mencionando o C / C ++ juntos porque o compilador GNU gcc funciona para as linguagens de programação C e C ++.

Se você estiver usando **Linux ou UNIX** , verifique se o GCC está Instalação no UNIX / Linux instalado em seu sistema, digitando o seguinte comando a partir da linha de comando -

```
$ gcc -v
```

Se você tem o compilador GNU instalado em sua máquina, então deve imprimir uma mensagem como segue -

```
Using built-in specs.
Target: i386-redhat-linux
Configured with: ../configure --prefix=/usr .......
Thread model: posix
gcc version 4.1.2 20080704 (Red Hat 4.1.2-46)
```

Se o GCC não estiver instalado, você deverá instalá-lo usando as instruções detalhadas disponíveis em https://gcc.gnu.org/install/

Este tutorial foi escrito com base no Linux e todos os exemplos dados foram compilados no sabor Cent OS do sistema Linux.

Instalação no Mac OS

Se você usa o Mac OS X, a maneira mais fácil de obter o GCC é baixar o ambiente de desenvolvimento Xcode do site da Apple e seguir as instruções de instalação simples. Uma vez que você tenha a configuração do Xcode, você poderá usar o compilador GNU para C / C ++.

O Xcode está atualmente disponível em developer.apple.com/technologies/tools/ .

Instalação no Windows

Para instalar o GCC no Windows, você precisa instalar o MinGW. Para instalar o MinGW, acesse a página do MinGW, www.mingw.org , e siga o link para a página de download do MinGW. Faça o download da versão mais recente do programa de instalação do MinGW, que deve ser denominado MinGW- <version> .exe.

Ao instalar o Min GW, no mínimo, você deve instalar o gcc-core, o gcc-g ++, o binutils e o runtime do MinGW, mas você pode querer instalar mais.

Adicione o subdiretório bin de sua instalação do MinGW à sua variável de ambiente **PATH** , para que você possa especificar essas ferramentas na linha de comandos por seus nomes simples.

Após a conclusão da instalação, você poderá executar gcc, g ++, ar, ranlib, dlltool e várias outras ferramentas GNU na linha de comando do Windows.

C - Estrutura do Programa

Antes de estudar os blocos básicos de construção da linguagem de programação C, vamos dar uma olhada em uma estrutura mínima do programa em C, de modo que possamos tomá-la como referência nos próximos capítulos.

Exemplo Hello World

O programa AC consiste basicamente nas seguintes partes -

- Comandos do pré-processador
- Funções
- Variáveis
- Declarações e Expressões
- Comentários

Vamos olhar para um código simples que imprimiria as palavras "Hello World" -

```c
#include <stdio.h>

int main() {
   /* my first program in C */
   printf("Hello, World! \n");

   return 0;
}
```

Vamos dar uma olhada nas várias partes do programa acima -

A primeira linha do programa *#include <stdio.h>* é um comando pré-processador, que diz ao compilador C para incluir o arquivo stdio.h antes de ir para a compilação real.

A próxima linha *int main ()* é a principal função onde a execução do programa começa.

A próxima linha /*...*/ será ignorada pelo compilador e foi colocada para adicionar comentários adicionais no programa. Então, essas linhas são chamadas de comentários no programa.

A próxima linha *printf (...)* é outra função disponível em C que faz com que a mensagem "Hello, World!" para ser exibido na tela.

A próxima linha **retorna 0;** termina a função main () e retorna o valor 0.

Vamos ver como salvar o código-fonte em um arquivo e Compile e execute o programa C como compilá-lo e executá-lo. A seguir estão os passos simples -

Abra um editor de texto e adicione o código acima mencionado.

Salve o arquivo como *hello.c*

Abra um prompt de comando e vá para o diretório em que você salvou o arquivo.

Digite *gcc hello.c* e pressione enter para compilar seu código.

Se não houver erros no seu código, o prompt de comando o levará para a próxima linha e gerará o arquivo executável *a.out* .

Agora, digite *a.out* para executar seu programa.

Você verá a saída *"Hello World"* impressa na tela.

```
$ gcc hello.c
$ ./a.out
Hello, World!
```

Certifique-se de que o compilador gcc esteja em seu caminho e que você o esteja executando no diretório que contém o arquivo de origem hello.c.

C - Sintaxe Básica

Você viu a estrutura básica de um programa em C, então será fácil entender outros blocos básicos de construção da linguagem de programação C.

Um programa AC consiste em vários tokens e um token é uma palavra-chave, um identificador, uma constante, um literal de string ou um símbolo.
Por exemplo, a seguinte instrução C consiste em cinco tokens - Tokens em C

```
printf("Hello, World! \n");
```

Os tokens individuais são -

```
printf
(
"Hello, World! \n"
)
;
```

Em um programa C, o ponto-e-vírgula é um terminador de instrução. Isto é, cada afirmação individual deve terminar com um ponto e vírgula. Indica o fim de uma entidade lógica. Ponto e vírgula

Dada a seguir são duas declarações diferentes -

```
printf("Hello, World! \n");
return 0;
```

Os comentários são como ajudar o texto em seu programa C e são ignorados pelo compilador. Eles começam com / * e terminam com os caracteres * / como mostrado abaixo - Comentários

```
/* my first program in C */
```

Você não pode ter comentários dentro de comentários e eles não ocorrem em literais de string ou caractere.

Identificador AC é um nome usado para identificar uma variável, função ou qualquer outro item definido pelo usuário. Um identificador começa com uma letra A a Z, a a z ou um sublinhado '_' seguido de zero ou mais letras, sublinhados e dígitos (0 a 9). Identificadores

C não permite caracteres de pontuação como @, $ e% nos identificadores. C é uma linguagem de programação que **diferencia maiúsculas de minúsculas** . Assim, *Manpower* e *mão de obra* são dois identificadores diferentes em C. Aqui estão alguns exemplos de identificadores aceitáveis -

```
mohd       zara    abc    move_name  a_123
myname50   _temp   j      a23b9      retVal
```

A lista a seguir mostra as palavras reservadas em C. Essas palavras reservadas não podem ser usadas como constantes ou variáveis ou quaisquer outros nomes de identificadores. Palavras-chave

auto	outro	longo	interruptor
pausa	enum	registo	typedef

caso	extern	Retorna	União
Caracteres	flutuador	baixo	sem assinatura
const	para	assinado	vazio
continuar	vamos para	tamanho de	volátil
padrão	E se	estático	enquanto
Faz	int	struct	_Packed
em dobro			

Uma linha contendo apenas espaços em branco, possivelmente com um comentário, é conhecida como uma linha em branco, e um compilador C a ignora totalmente.

Espaço em branco em C

Espaços em branco é o termo usado em C para descrever espaços em branco, tabulações, caracteres de nova linha e comentários. O espaço em branco separa uma parte de uma instrução da outra e habilita o compilador a identificar onde um elemento em uma instrução, como int, termina e o próximo elemento começa. Portanto, na seguinte declaração -

```
int age;
```

deve haver pelo menos um caractere de espaço em branco (geralmente um espaço) entre int e age para o compilador ser capaz de distingui-los. Por outro lado, na seguinte declaração -

```
fruit = apples + oranges;   // get the total fruit
```

não são necessários caracteres em branco entre frutas e =, ou entre = e maçãs, embora você esteja livre para incluir alguns, se desejar aumentar a legibilidade.

C - Tipos de dados

Os tipos de dados em c referem-se a um sistema extensivo usado para declarar variáveis ou funções de diferentes tipos. O tipo de variável determina quanto espaço ocupa no armazenamento e como o padrão de bits armazenado é interpretado.

Os tipos em C podem ser classificados da seguinte forma -

Sr. Não.	Tipos e Descrição
1	**Tipos Básicos** Eles são tipos aritméticos e são classificados em: (a) tipos inteiros e (b) tipos de ponto flutuante.

2	**Tipos enumerados** Eles são novamente tipos aritméticos e são usados para definir variáveis que podem apenas designar certos valores inteiros discretos em todo o programa.
3	**O tipo vazio** O especificador de tipo *void* indica que nenhum valor está disponível.
4	**Tipos derivados** Eles incluem (a) tipos de ponteiro, (b) tipos de matriz, (c) tipos de estrutura, (d) tipos de união e (e) tipos de função.

Os tipos de matriz e os tipos de estrutura são referidos coletivamente como tipos agregados. O tipo de uma função especifica o tipo do valor de retorno da função. Veremos os tipos básicos na seção seguinte, onde outros tipos serão abordados nos próximos capítulos.

A tabela a seguir fornece os detalhes dos tipos inteiros padrão com seus Tipos inteiros tamanhos de armazenamento e intervalos de valores -

Tipo	Tamanho do armazenamento	Faixa de valor
Caracteres	1 byte	-128 a 127 ou 0 a 255
caracter não identifcado	1 byte	0 a 255
caractere assinado	1 byte	-128 a 127
int	2 ou 4 bytes	-32.768 a 32.767 ou -2.147.483.648 a 2.147.483.647
int não assinado	2 ou 4 bytes	0 a 65.535 ou 0 a 4.294.967.295
baixo	2 bytes	-32.768 a 32.767
curta não assinada	2 bytes	0 a 65.535
longo	4 bytes	-2,147,483,648 a 2,147,483,647
sem assinatura longa	4 bytes	0 a 4.294.967.295

Para obter o tamanho exato de um tipo ou uma variável em uma plataforma específica, você pode usar o operador **sizeof** . As expressões *sizeof (type)* produzem o tamanho do armazenamento do objeto ou tipo em bytes. Dada a seguir é um exemplo para obter o tamanho do tipo int em qualquer máquina -

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

int main() {
   printf("Storage size for int : %d \n", sizeof(int));

   return 0;
}
```

Quando você compila e executa o programa acima, ele produz o seguinte resultado no Linux -

```
Storage size for int : 4
```

A tabela a seguir fornece os detalhes dos tipos de ponto flutuante padrão com tamanhos de armazenamento e intervalos de valores e sua precisão -

Tipos de ponto flutuante

Tipo	Tamanho do armazenamento	Faixa de valor	Precisão
flutuador	4 bytes	1,2E-38 a 3,4E + 38	6 casas decimais
em dobro	8 bytes	2,3E-308 a 1,7E + 308	15 casas decimais
longa dupla	10 byte	3,4E-4932 a 1,1E + 4932	19 casas decimais

O arquivo de cabeçalho float.h define macros que permitem usar esses valores e outros detalhes sobre a representação binária de números reais em seus programas. O exemplo a seguir imprime o espaço de armazenamento obtido por um tipo flutuante e seus valores de intervalo -

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

int main() {
   printf("Storage size for float : %d \n", sizeof(float));
   printf("Minimum float positive value: %E\n", FLT_MIN );
   printf("Maximum float positive value: %E\n", FLT_MAX );
   printf("Precision value: %d\n", FLT_DIG );

   return 0;
}
```

Quando você compila e executa o programa acima, ele produz o seguinte resultado no Linux -

```
Storage size for float : 4
Minimum float positive value: 1.175494E-38
Maximum float positive value: 3.402823E+38
Precision value: 6
```

O tipo void especifica que nenhum valor está disponível. É usado em três tiposO tipo vazio

de situações -

Sr. Não.	Tipos e Descrição
1	**Função retorna como vazia** Existem várias funções em C que não retornam nenhum valor ou você pode dizer que elas retornam nulas. Uma função sem valor de retorno tem o tipo de retorno como vazio. Por exemplo, **void exit (int status);**
2	**Argumentos de função como nulos** Existem várias funções em C que não aceitam nenhum parâmetro. Uma função sem parâmetro pode aceitar um vazio. Por exemplo, **int rand (void);**
3	**Ponteiros para anular** Um ponteiro do tipo void * representa o endereço de um objeto, mas não o seu tipo. Por exemplo, uma função de alocação de memória **void * malloc (size_t size);** retorna um ponteiro para void que pode ser convertido para qualquer tipo de dados.

C - Variáveis

Uma variável nada mais é do que um nome dado a uma área de armazenamento que nossos programas podem manipular. Cada variável em C possui um tipo específico, que determina o tamanho e o layout da memória da variável; o intervalo de valores que podem ser armazenados nessa memória; e o conjunto de operações que podem ser aplicadas à variável.

O nome de uma variável pode ser composto de letras, dígitos e o caractere de sublinhado. Deve começar com uma letra ou um sublinhado. Letras maiúsculas e minúsculas são distintas porque C faz distinção entre maiúsculas e minúsculas. Com base nos tipos básicos explicados no capítulo anterior, haverá os seguintes tipos básicos de variáveis -

Sr. Não.	Tipo e Descrição
1	**Caracteres** Normalmente, um único octeto (um byte). Este é um tipo inteiro.
2	**int** O tamanho mais natural do inteiro para a máquina.
3	**flutuador** Um valor de ponto flutuante de precisão única.

4	**em dobro** Um valor de ponto flutuante de precisão dupla.
5	**vazio** Representa a ausência de tipo.

A linguagem de programação C também permite definir vários outros tipos de variáveis, as quais abordaremos em capítulos subseqüentes como Enumeração, Ponteiro, Matriz, Estrutura, União etc. Para este capítulo, vamos estudar apenas os tipos básicos de variáveis.

Uma definição de variável informa ao compilador onde e quanto armazenamento deve ser criado para a variável. Uma definição de variável especifica um tipo de dados e contém uma lista de uma ou mais variáveis desse tipo da seguinte maneira:

Definição Variável em C

```
type variable_list;
```

Aqui, **tipo** deve ser um tipo de dados C válido, incluindo char, w_char, int, float, double, bool ou qualquer objeto definido pelo usuário; e **variable_list** pode consistir em um ou mais nomes de identificadores separados por vírgulas. Algumas declarações válidas são mostradas aqui -

```
int    i, j, k;
char   c, ch;
float  f, salary;
double d;
```

A linha **int i, j, k;** declara e define as variáveis i, j e k; que instrui o compilador a criar variáveis chamadas i, j e k do tipo int.

Variáveis podem ser inicializadas (atribuídas a um valor inicial) em sua declaração. O inicializador consiste em um sinal de igual seguido por uma expressão constante da seguinte maneira -

```
type variable_name = value;
```

Alguns exemplos são -

```
extern int d = 3, f = 5;    // declaration of d and f.
int d = 3, f = 5;           // definition and initializing d and f.
byte z = 22;                // definition and initializes z.
char x = 'x';               // the variable x has the value 'x'.
```

Para definição sem um inicializador: variáveis com duração de armazenamento estático são implicitamente inicializadas com NULL (todos os bytes possuem o valor 0); o valor inicial de todas as outras variáveis é indefinido.

Uma declaração de variável fornece garantia ao compilador de

Declaração Variável em C

que existe uma variável com o tipo e o nome fornecidos, de modo que o compilador possa continuar a compilação adicional sem exigir detalhes completos sobre a variável. Uma definição de variável tem seu significado somente no momento da compilação, o compilador precisa de definição de variável real no momento de vincular o programa.

Uma declaração de variável é útil quando você está usando vários arquivos e você define sua variável em um dos arquivos que estarão disponíveis no momento da vinculação do programa. Você usará a palavra-chave **extern** para declarar uma variável em qualquer lugar. Embora você possa declarar uma variável várias vezes em seu programa em C, ela pode ser definida apenas uma vez em um arquivo, uma função ou um bloco de código.

Exemplo

Tente o seguinte exemplo, onde as variáveis foram declaradas no topo, mas elas foram definidas e inicializadas dentro da função principal -

```
#include <stdio.h>

// Variable declaration:
extern int a, b;
extern int c;
extern float f;

int main () {

   /* variable definition: */
   int a, b;
   int c;
   float f;

   /* actual initialization */
   a = 10;
   b = 20;

   c = a + b;
   printf("value of c : %d \n", c);

   f = 70.0/3.0;
   printf("value of f : %f \n", f);

   return 0;
}
```

Quando o código acima é compilado e executado, ele produz o seguinte resultado -

```
value of c : 30
value of f : 23.333334
```

O mesmo conceito se aplica na declaração de função onde você fornece um nome de função no momento de sua declaração e sua definição real pode ser dada em qualquer outro lugar. Por exemplo -

```
// function declaration
int func();

int main() {

   // function call
   int i = func();
```

```
}

// function definition
int func() {
   return 0;
}
```

Existem dois tipos de expressões em C

Lvalores e Rvalores em C

lvalue - Expressões que se referem a um local de memória são chamadas de expressões "lvalue". Um lvalue pode aparecer como o lado esquerdo ou direito de uma atribuição.

rvalue - O termo rvalue refere-se a um valor de dados que é armazenado em algum endereço na memória. Um rvalor é uma expressão que não pode ter um valor atribuído a ele, o que significa que um rvalor pode aparecer no lado direito, mas não no lado esquerdo de uma atribuição.

Variáveis são lvalores e, portanto, podem aparecer no lado esquerdo de uma atribuição. Literais numéricos são rvalues e, portanto, podem não ser atribuídos e não podem aparecer no lado esquerdo. Dê uma olhada nas seguintes declarações válidas e inválidas -

```
int g = 20; // valid statement


10 = 20; // invalid statement; would generate compile-time error
```

C - Constantes e Literais

Constantes referem-se a valores fixos que o programa não pode alterar durante sua execução. Esses valores fixos também são chamados de **literais** .

As constantes podem ser de qualquer um dos tipos básicos de dados, como *uma constante inteira, uma constante flutuante, uma constante de caractere ou uma string literal* . Existem constantes de enumeração também.

As constantes são tratadas como variáveis regulares, exceto que seus valores não podem ser modificados após sua definição.

Um literal inteiro pode ser uma constante decimal, octal ou hexadecimal. Um prefixo especifica a base ou base: 0x ou 0X para hexadecimal, 0 para octal e nada para decimal.

Literais Inteiros

Um literal inteiro também pode ter um sufixo que é uma combinação de U e L, para unsigned e long, respectivamente. O sufixo pode ser maiúsculo ou minúsculo e pode estar em qualquer ordem.

Aqui estão alguns exemplos de literais inteiros -

```
212         /* Legal */
215u        /* Legal */
0xFeeL      /* Legal */
078         /* Illegal: 8 is not an octal digit */
032UU       /* Illegal: cannot repeat a suffix */
```

A seguir, outros exemplos de vários tipos de literais inteiros -

```
85         /* decimal */
0213       /* octal */
0x4b       /* hexadecimal */
30         /* int */
30u        /* unsigned int */
30l        /* long */
30ul       /* unsigned long */
```

Literais de ponto flutuante

Um literal de ponto flutuante tem uma parte inteira, um ponto decimal, uma parte fracionária e uma parte exponencial. Você pode representar literais de ponto flutuante no formato decimal ou exponencial.

Enquanto representa a forma decimal, você deve incluir o ponto decimal, o expoente ou ambos; e ao representar a forma exponencial, você deve incluir a parte inteira, a parte fracionária ou ambas. O expoente assinado é introduzido por e ou E.

Aqui estão alguns exemplos de literais de ponto flutuante -

```
3.14159       /* Legal */
314159E-5L    /* Legal */
510E          /* Illegal: incomplete exponent */
210f          /* Illegal: no decimal or exponent */
.e55          /* Illegal: missing integer or fraction */
```

Constantes de personagem

Literais de caracteres são colocados entre aspas simples, por exemplo, 'x' pode ser armazenado em uma variável simples do tipo **char** .

Um literal de caractere pode ser um caractere simples (por exemplo, 'x'), uma seqüência de escape (por exemplo, '\ t') ou um caractere universal (por exemplo, '\ u02C0').

Existem certos caracteres em C que representam um significado especial quando precedidos por uma barra invertida, por exemplo, newline (\ n) ou tab (\ t).

Aqui, você tem uma lista de códigos de seqüência de escape -

A seguir está o exemplo para mostrar alguns caracteres de seqüência de escape -

```
#include <stdio.h>

int main() {
   printf("Hello\tWorld\n\n");

   return 0;
}
```

Quando o código acima é compilado e executado, ele produz o seguinte resultado -

```
Hello World
```

Literais de cordas

Literais de string ou constantes são colocados entre aspas duplas "". Uma sequência contém caracteres semelhantes aos literais de caracteres:

caracteres simples, seqüências de escape e caracteres universais.

Você pode dividir uma linha longa em várias linhas usando literais de string e separá-las usando espaços em branco.

Aqui estão alguns exemplos de literais de string. Todas as três formas são seqüências idênticas.

```
"hello, dear"

"hello, \

dear"

"hello, " "d" "ear"
```

Existem duas maneiras simples em C para definir constantes - Definindo Constantes

- Usando o **#define** preprocessor.
- Usando a palavra-chave **const** .

Dada a seguir é o formulário para usar o #define preprocessor O pré-processador #define para definir uma constante -

```
#define identifier value
```

O exemplo a seguir explica isso em detalhes -

```
#include <stdio.h>

#define LENGTH 10
#define WIDTH  5
#define NEWLINE '\n'

int main() {
   int area;

   area = LENGTH * WIDTH;
   printf("value of area : %d", area);
   printf("%c", NEWLINE);

   return 0;
}
```

Quando o código acima é compilado e executado, ele produz o seguinte resultado -

```
value of area : 50
```

Você pode usar o prefixo **const** para declarar constantes com um A palavra-chave const tipo específico da seguinte maneira -

```
const type variable = value;
```

O exemplo a seguir explica isso em detalhes -

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

```c
int main() {
   const int  LENGTH = 10;
   const int  WIDTH = 5;
   const char NEWLINE = '\n';
   int area;

   area = LENGTH * WIDTH;
   printf("value of area : %d", area);
   printf("%c", NEWLINE);

   return 0;
}
```

Quando o código acima é compilado e executado, ele produz o seguinte resultado -

```
value of area : 50
```

Note que é uma boa prática de programação definir constantes em CAPITAL.

C - Classes de Armazenamento

Uma classe de armazenamento define o escopo (visibilidade) e o tempo de vida de variáveis e / ou funções dentro de um programa em C. Eles precedem o tipo que eles modificam. Nós temos quatro classes de armazenamento diferentes em um programa C -

- auto
- registo
- estático
- extern

A classe de armazenamento **automático** é a classe de armazenamento padrão para todas as variáveis locais.

A classe de armazenamento automático

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

O exemplo acima define duas variáveis na mesma classe de armazenamento. 'auto' só pode ser usado dentro de funções, isto é, variáveis locais.

A classe de armazenamento de **registro** é usada para definir variáveis locais que devem ser armazenadas em um registro em vez de RAM. Isso significa que a variável tem um tamanho máximo igual ao tamanho do registro (geralmente uma palavra) e não pode ter o operador '&' unário aplicado a ela (pois não tem um local de memória).

O registro Storage Class

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

O registrador só deve ser usado para variáveis que requerem acesso rápido, como contadores. Deve-se notar também que definir 'registrador' não significa que a variável

será armazenada em um registrador. Isso significa que ele pode ser armazenado em um registro, dependendo das restrições de hardware e implementação.

A classe de armazenamento **estática** instrui o compilador a manter uma variável local em existência durante a vida útil do programa, em vez de criá-lo e destruí-lo toda vez que ele entrar e sair do escopo. Portanto, tornar as variáveis locais estáticas permite que elas mantenham seus valores entre as chamadas de função.

A classe de armazenamento estática

O modificador estático também pode ser aplicado a variáveis globais. Quando isso é feito, faz com que o escopo da variável seja restrito ao arquivo no qual ela está declarada.

Na programação C, quando **estática** é usada em uma variável global, faz com que apenas uma cópia desse membro seja compartilhada por todos os objetos de sua classe.

```
#include <stdio.h>

/* function declaration */
void func(void);

static int count = 5; /* global variable */

main() {

   while(count--) {
      func();
   }

   return 0;
}

/* function definition */
void func( void ) {

   static int i = 5; /* local static variable */
   i++;

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

Quando o código acima é compilado e executado, ele produz o seguinte resultado -

```
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
```

A classe de armazenamento **externa** é usada para fornecer uma referência de uma variável global que é visível para TODOS os arquivos de programa. Quando você usa 'extern', a variável não pode ser inicializada, no entanto, ela aponta o nome da variável em um local de armazenamento que foi definido anteriormente.

A classe de armazenamento externa

Quando você tem vários arquivos e define uma variável ou função global, que também será usada em outros arquivos, o *extern* será usado em outro arquivo para fornecer a

referência da variável ou função definida. Apenas para entender, *extern* é usado para declarar uma variável global ou função em outro arquivo.

O modificador externo é mais comumente usado quando há dois ou mais arquivos compartilhando as mesmas variáveis globais ou funções, conforme explicado abaixo.

Primeiro Arquivo: main.c

```
#include <stdio.h>

int count ;
extern void write_extern();

main() {
   count = 5;
   write_extern();
}
```

Segundo arquivo: support.c

```
#include <stdio.h>

extern int count;

void write_extern(void) {
   printf("count is %d\n", count);
}
```

Aqui, *extern* está sendo usado para declarar *count* no segundo arquivo, onde como ele tem sua definição no primeiro arquivo, main.c. Agora, compile esses dois arquivos da seguinte maneira -

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

Produzirá o programa executável **a.out** . Quando este programa é executado, produz o seguinte resultado -

```
count is 5
```

C - Operadores

Um operador é um símbolo que informa ao compilador para executar funções matemáticas ou lógicas específicas. A linguagem C é rica em operadores integrados e fornece os seguintes tipos de operadores -

- Operadores aritméticos
- Operadores Relacionais
- Operadores lógicos
- Operadores bit a bit
- Operadores de atribuição
- Operadores Diversos

Vamos, neste capítulo, examinar a maneira como cada operador trabalha.

A tabela a seguir mostra todos os operadores aritméticos suportados pela linguagem C. Suponha que a variável **A** detenha 10 e a variável **B** detenha 20 então - Operadores aritméticos

Mostrar exemplos

Operador	Descrição	Exemplo
+	Adiciona dois operandos.	A + B = 30
-	Subtrai o segundo operando do primeiro.	A - B = -10
*	Multiplica os dois operandos.	A * B = 200
/	Divide o numerador por de-numerador.	B / A = 2
%	Operador de módulo e resto depois de uma divisão inteira.	B% A = 0
++	O operador de incremento aumenta o valor inteiro em um.	A ++ = 11
-	O operador Decrement diminui o valor inteiro em um.	A-- = 9

A tabela a seguir mostra todos os operadores relacionais suportados por C. Suponha que a variável **A** detenha 10 e a variável **B** detenha 20 então - Operadores Relacionais

Mostrar exemplos

Operador	Descrição	Exemplo
==	Verifica se os valores de dois operandos são iguais ou não. Se sim, a condição se torna verdadeira.	(A == B) não é verdade.
! =	Verifica se os valores de dois operandos são iguais ou não. Se os valores não forem iguais, a condição se tornará verdadeira.	(A! = B) é verdade.
>	Verifica se o valor do operando à esquerda é maior que o valor do operando da direita. Se sim, a condição se torna verdadeira.	(A> B) não é verdade.
<	Verifica se o valor do operando à esquerda é menor que o valor do operando da direita. Se sim, a condição se torna verdadeira.	(A <B) é verdade.
> =	Verifica se o valor do operando à esquerda é maior ou igual ao valor do operando da direita. Se sim, a condição se torna verdadeira.	(A> = B) não é verdade.

<=	Verifica se o valor do operando à esquerda é menor ou igual ao valor do operando à direita. Se sim, a condição se torna verdadeira.	(A <= B) é verdade.

A tabela a seguir mostra todos os operadores lógicos suportados pela linguagem C. Assuma que a variável **A** contém 1 e a variável **B** contém 0, então - Operadores lógicos

Mostrar exemplos

Operador	Descrição	Exemplo
&&	Chamado Lógico E operador. Se ambos os operandos forem diferentes de zero, a condição se tornará verdadeira.	(A && B) é falso.
||	Chamado Lógico OU Operador. Se algum dos dois operandos for diferente de zero, a condição se tornará verdadeira.	(A || B) é verdadeiro.
!	Operador NÃO Lógico Chamado. Ele é usado para reverter o estado lógico de seu operando. Se uma condição for verdadeira, o operador Lógico NOT tornará falsa.	! (A && B) é verdade.

O operador bit a bit trabalha em bits e executa a operação bit a bit. As tabelas de verdade para &, | e ^ são as seguintes - Operadores bit a bit

p	q	p & q	p | q	p ^ q
0	0	0	0	0
0	1	0	1	1
1	1	1	1	0
1	0	0	1	1

Assume A = 60 e B = 13 em formato binário, eles serão os seguintes -

A = 0011 1100

B = 0000 1101

A & B = 0000 1100

A | B = 0011 1101

A ^ B = 0011 0001

~ A = 1100 0011

A tabela a seguir lista os operadores bit a bit suportados por C. Suponha que a variável 'A'

contenha 60 e a variável 'B' contenha 13, então -

Mostrar exemplos

Operador	Descrição	Exemplo
E	Binary AND Operator copia um pouco para o resultado, se existir em ambos os operandos.	(A & B) = 12, ou seja, 0000 1100
|	Operador binário OR copia um bit se existir em um dos operandos.	(A | B) = 61, ou seja, 0011 1101
^	Operador XOR binário copia o bit se estiver definido em um operando, mas não em ambos.	(A ^ B) = 49, ou seja, 0011 0001
~	Binary Ones Complement Operator é unário e tem o efeito de 'flipping' bits.	(~ A) = -60, isto é. 1100 0100 no formulário de complemento de 2.
<< <<	Operador Shift Esquerdo. O valor dos operandos esquerdos é movido para a esquerda pelo número de bits especificado pelo operando direito.	A << 2 = 240 ou seja, 1111 0000
>>	Operador de deslocamento à direita binário. O valor dos operandos esquerdos é movido para a direita pelo número de bits especificado pelo operando direito.	A >> 2 = 15 ou seja, 0000 1111

A tabela a seguir lista os operadores de atribuição suportados pela linguagem C -

Operadores de atribuição

Mostrar exemplos

Operador	Descrição	Exemplo
=	Operador de atribuição simples. Atribui valores de operandos do lado direito ao operando do lado esquerdo	C = A + B atribuirá o valor de A + B a C
+ =	Adicionar operador de atribuição E. Adiciona o operando direito ao operando esquerdo e atribui o resultado ao operando esquerdo.	C + = A é equivalente a C = C + A
- =	Subtrair e operador de atribuição. Subtrai o operando direito do operando esquerdo e atribui o resultado ao operando esquerdo.	C - = A é equivalente a C = C - A
* =	Operador Multiply AND assignment. Multiplica o operando da direita pelo operando da esquerda e atribui o resultado ao operando da esquerda.	C * = A é equivalente a C = C * A
/ =	Divide AND operador de atribuição. Ele divide o operando da esquerda com o operando da	C / = A é equivalente a C = C / A

	direita e atribui o resultado ao operando da esquerda.	
% =	Módulo E operador de atribuição. Ele usa módulo usando dois operandos e atribui o resultado ao operando esquerdo.	C% = A é equivalente a C = C% A
<< =	Esquerda e operador de atribuição.	C << = 2 é o mesmo que C = C << 2
>> =	Operador de deslocamento e atribuição à direita.	C >> = 2 é o mesmo que C = C >> 2
& =	Operador Bitwise AND assignment.	C & = 2 é o mesmo que C = C e 2
^ =	Operador exclusivo de OR ou de atribuição de bits.	C ^ = 2 é o mesmo que C = C ^ 2
| =	Operador OR ou com atribuição de bits.	C | = 2 é o mesmo que C = C | 2

Operadores Diversos ↦ sizeof & ternary

Além dos operadores discutidos acima, existem alguns outros operadores importantes, incluindo **sizeof** e **? :** suportado pela linguagem C.

Mostrar exemplos

Operador	Descrição	Exemplo
tamanho de()	Retorna o tamanho de uma variável.	sizeof (a), onde a é inteiro, retornará 4.
E	Retorna o endereço de uma variável.	&uma; retorna o endereço real da variável.
*	Ponteiro para uma variável.	*uma;
? :	Expressão Condicional.	Se a condição é verdadeira? então valor X: caso contrário, valor Y

Precedência dos operadores em C

A precedência do operador determina o agrupamento de termos em uma expressão e decide como uma expressão é avaliada. Certos operadores têm precedência mais alta que outros; por exemplo, o operador de multiplicação tem uma precedência maior que o operador de adição.

Por exemplo, x = 7 + 3 * 2; aqui, x é atribuído 13, não 20, porque o operador * tem uma precedência maior que +, portanto, ele primeiro é multiplicado com 3 * 2 e, em seguida, adiciona 7.

Aqui, os operadores com a maior precedência aparecem na parte superior da tabela, e aqueles com os mais baixos aparecem na parte inferior. Dentro de uma expressão, os

operadores de precedência mais alta serão avaliados primeiro.

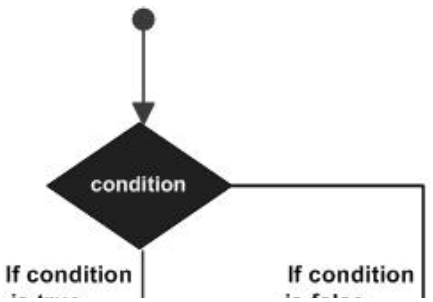
Mostrar exemplos

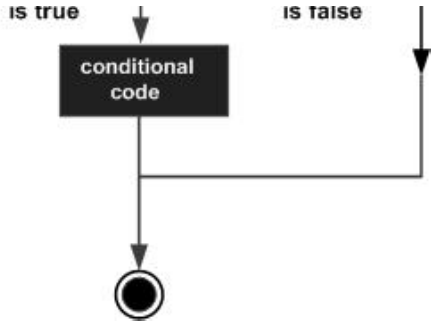
Categoria	Operador	Associatividade
Postfix	() [] ->. ++ - -	Esquerda para a direita
Unário	+ -! ~ ++ - - (tipo) * & tamanho do	Direita para esquerda
Multiplicativo	* /%	Esquerda para a direita
Aditivo	+ -	Esquerda para a direita
Mudança	<< >>	Esquerda para a direita
Relacional	<=>> =	Esquerda para a direita
Igualdade	==!	Esquerda para a direita
Bit a bit E	E	Esquerda para a direita
Bit a bit XOR	^	Esquerda para a direita
Bit a bit OU	|	Esquerda para a direita
E lógico	&&	Esquerda para a direita
OR lógico	||	Esquerda para a direita
Condicional	?:	Direita para esquerda
Tarefa	= + = - = * = / =% = >> = << = & = ^ = | =	Direita para esquerda
Vírgula	,	Esquerda para a direita

C - Tomada de Decisão

As estruturas de tomada de decisão exigem que o programador especifique uma ou mais condições a serem avaliadas ou testadas pelo programa, juntamente com uma declaração ou instruções a serem executadas se a condição for determinada como verdadeira e, opcionalmente, outras instruções a serem executadas se a condição está determinado a ser falso.

Veja abaixo a forma geral de uma estrutura típica de tomada de decisão encontrada na maioria das linguagens de programação -





A linguagem de programação C assume qualquer valor **diferente de zero** e **não nulo** como **true** e, se for **zero** ou **nulo** , será assumido como valor **falso** .

A linguagem de programação C fornece os seguintes tipos de declarações de tomada de decisão.

Sr. Não.	Declaração & Descrição
1	**se declaração** Uma **instrução if** consiste em uma expressão booleana seguida de uma ou mais instruções.
2	**if ... else statement** Uma **instrução if** pode ser seguida por uma **instrução else** opcional, que é executada quando a expressão booleana é falsa.
3	**aninhado se instruções** Você pode usar um **if if else se** instrução dentro de outro **if** ou **else if** statement (s).
4	**mudar a indicação** Uma instrução **switch** permite que uma variável seja testada quanto à igualdade em relação a uma lista de valores.
5	**instruções de troca aninhadas** Você pode usar uma instrução **switch** dentro de outra instrução de **switch** .

Nós cobrimos **operador condicional? :** no capítulo anterior, que pode ser usado para substituir **if ... else** statements. Tem a seguinte forma geral -

: Operador O ?

```
Exp1 ? Exp2 : Exp3;
```

Onde Exp1, Exp2 e Exp3 são expressões. Observe o uso e a colocação dos dois pontos.

O valor de um? expressão é determinada assim -

Exp1 é avaliado. Se é verdade, então Exp2 é avaliado e se torna o valor de todo? expressão.

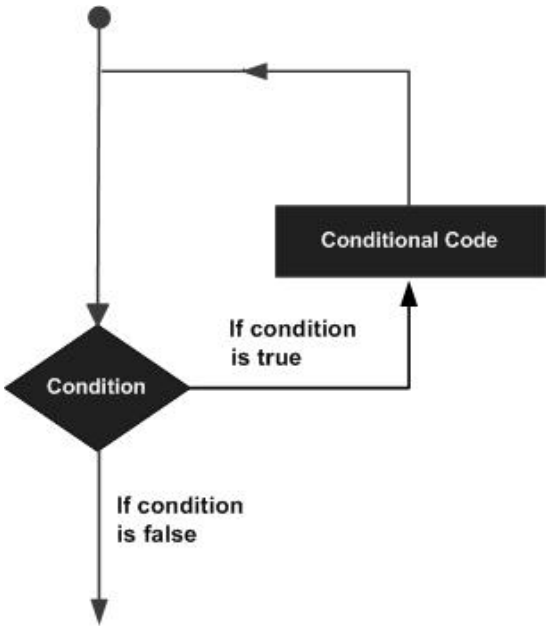
Se Exp1 for falso, Exp3 será avaliado e seu valor se tornará o valor da expressão.

C - Loops

Você pode encontrar situações, quando um bloco de código precisa ser executado várias vezes. Em geral, as instruções são executadas seqüencialmente: a primeira instrução em uma função é executada primeiro, seguida pela segunda e assim por diante.

As linguagens de programação fornecem várias estruturas de controle que permitem caminhos de execução mais complicados.

Uma instrução de loop nos permite executar uma instrução ou grupo de instruções várias vezes. Dada a seguir é a forma geral de uma declaração de loop na maioria das linguagens de programação -



A linguagem de programação C fornece os seguintes tipos de loops para lidar com os requisitos de loop.

Sr. Não.	Tipo de loop e descrição
1	**enquanto loop** Repete uma instrução ou grupo de instruções enquanto uma determinada condição é verdadeira. Ele testa a condição antes de executar o corpo do loop.
2	**para loop** Executa uma seqüência de instruções várias vezes e abrevia o código que gerencia a variável de loop.
3	**fazer ... while loop** É mais como uma instrução while, exceto pelo fato de testar a condição no final do corpo do loop.

4	**loops aninhados** Você pode usar um ou mais loops dentro de qualquer outro loop while, for, ou do..while.

As instruções de controle de loop alteram a execução de sua sequência normal. Quando a execução deixa um escopo, todos os objetos automáticos que foram criados nesse escopo são destruídos.

Declarações de controle de loop

C suporta as seguintes instruções de controle.

Sr. Não.	Declaração de controle e descrição
1	**declaração de quebra** Encerra a instrução **loop** ou **switch** e transfere a execução para a instrução imediatamente após o loop ou comutador.
2	**continuar instrução** Faz com que o loop pule o restante de seu corpo e repita imediatamente sua condição antes de reiterar.
3	**declaração goto** Transfere o controle para a instrução rotulada.

Um loop se torna um loop infinito se uma condição nunca se tornar falsa. O loop for é tradicionalmente usado para essa finalidade. Como nenhuma das três expressões que formam o loop 'for' é necessária, você pode fazer um loop infinito deixando a expressão condicional vazia.

O Loop Infinito

```
#include <stdio.h>

int main () {

   for( ; ; ) {
      printf("This loop will run forever.\n");
   }

   return 0;
}
```

Quando a expressão condicional está ausente, supõe-se que seja verdadeira. Você pode ter uma inicialização e expressão de incremento, mas os programadores C mais comumente usam a construção for (;;) para significar um loop infinito.

NOTA - Você pode terminar um loop infinito pressionando as teclas Ctrl + C.

C - Funções

Uma função é um grupo de instruções que juntas executam uma tarefa. Todo programa C tem pelo menos uma função, que é **main ()** , e todos os programas mais triviais podem

definir funções adicionais.

Você pode dividir seu código em funções separadas. Como você divide seu código entre diferentes funções depende de você, mas logicamente a divisão é tal que cada função executa uma tarefa específica.

Uma **declaração de** função informa ao compilador sobre o nome de uma função, tipo de retorno e parâmetros. Uma **definição de** função fornece o corpo real da função.

A biblioteca padrão C fornece inúmeras funções internas que seu programa pode chamar. Por exemplo, **strcat ()** para concatenar duas strings, **memcpy ()** para copiar uma localização de memória para outra localização e muitas outras funções.

Uma função também pode ser referida como um método ou uma sub-rotina ou um procedimento, etc.

Definindo uma Função

A forma geral de uma definição de função na linguagem de programação C é a seguinte -

```
return_type function_name( parameter list ) {
   body of the function
}
```

Uma definição de função na programação C consiste em um *cabeçalho de função* e um *corpo de função* . Aqui estão todas as partes de uma função -

Tipo de retorno - Uma função pode retornar um valor. O **return_type** é o tipo de dados do valor que a função retorna. Algumas funções executam as operações desejadas sem retornar um valor. Nesse caso, o return_type é a palavra-chave **void** .

Nome da Função - Este é o nome real da função. O nome da função e a lista de parâmetros juntos constituem a assinatura da função.

Parâmetros - Um parâmetro é como um marcador de posição. Quando uma função é chamada, você passa um valor para o parâmetro. Este valor é referido como parâmetro ou argumento real. A lista de parâmetros refere-se ao tipo, ordem e número dos parâmetros de uma função. Parâmetros são opcionais; isto é, uma função pode não conter parâmetros.

Corpo da Função - O corpo da função contém uma coleção de instruções que definem o que a função faz.

Dada a seguir é o código fonte para uma função chamada **max ()** . Esta função usa dois parâmetros num1 e num2 e retorna o valor máximo entre os dois -

Exemplo

```
/* function returning the max between two numbers */
int max(int num1, int num2) {

   /* local variable declaration */
   int result;

   if (num1 > num2)
```

```
      result = num1;
   else
      result = num2;

   return result;
}
```

Declarações de Função

Uma **declaração de** função informa ao compilador sobre um nome de função e como chamar a função. O corpo real da função pode ser definido separadamente.

Uma declaração de função tem as seguintes partes -

```
return_type function_name( parameter list );
```

Para a função acima definida max (), a declaração da função é a seguinte -

```
int max(int num1, int num2);
```

Nomes de parâmetros não são importantes na declaração de função, apenas o seu tipo é necessário, então o seguinte também é uma declaração válida -

```
int max(int, int);
```

A declaração de função é necessária quando você define uma função em um arquivo de origem e chama essa função em outro arquivo. Nesse caso, você deve declarar a função na parte superior do arquivo que está chamando a função.

Chamando uma Função

Ao criar uma função C, você fornece uma definição do que a função precisa fazer. Para usar uma função, você terá que chamar essa função para executar a tarefa definida.

Quando um programa chama uma função, o controle do programa é transferido para a função chamada. Uma função chamada executa uma tarefa definida e quando sua instrução de retorno é executada ou quando sua chave de fechamento de finalização de função é atingida, ela retorna o controle de programa de volta ao programa principal.

Para chamar uma função, basta passar os parâmetros necessários junto com o nome da função e, se a função retornar um valor, você poderá armazenar o valor retornado. Por exemplo -

```
#include <stdio.h>

/* function declaration */
int max(int num1, int num2);

int main () {

   /* local variable definition */
   int a = 100;
   int b = 200;
   int ret;

   /* calling a function to get max value */
   ret = max(a, b);
```

```
   printf( "Max value is : %d\n", ret );

   return 0;
}

/* function returning the max between two numbers */
int max(int num1, int num2) {

   /* local variable declaration */
   int result;

   if (num1 > num2)
      result = num1;
   else
      result = num2;

   return result;
}
```

Nós mantivemos max () junto com main () e compilamos o código fonte. Ao executar o executável final, ele produziria o seguinte resultado -

```
Max value is : 200
```

Argumentos da Função

Se uma função é usar argumentos, ela deve declarar variáveis que aceitam os valores dos argumentos. Essas variáveis são chamadas de **parâmetros formais** da função.

Parâmetros formais se comportam como outras variáveis locais dentro da função e são criados após a entrada na função e destruídos na saída.

Ao chamar uma função, existem duas maneiras pelas quais os argumentos podem ser passados para uma função -

Sr. Não.	Tipo de chamada e descrição
1	**Ligue por valor** Este método copia o valor real de um argumento no parâmetro formal da função. Nesse caso, as alterações feitas no parâmetro dentro da função não afetam o argumento.
2	**Ligue por referência** Este método copia o endereço de um argumento no parâmetro formal. Dentro da função, o endereço é usado para acessar o argumento real usado na chamada. Isso significa que as alterações feitas no parâmetro afetam o argumento.

Por padrão, C usa **chamada por valor** para passar argumentos. Em geral, isso significa que o código dentro de uma função não pode alterar os argumentos usados para chamar a função.

C - Regras do Escopo

Um escopo em qualquer programação é uma região do programa onde uma variável definida pode ter sua existência e além dessa variável não pode ser acessada. Existem três lugares onde as variáveis podem ser declaradas na linguagem de programação C -

Dentro de uma função ou um bloco que é chamado de variáveis **locais** .

Fora de todas as funções que são chamadas de variáveis **globais** .

Na definição de parâmetros de função que são chamados parâmetros **formais** .

Vamos entender quais são variáveis **locais** e **globais** e parâmetros **formais** .

Variáveis Locais

Variáveis declaradas dentro de uma função ou bloco são chamadas de variáveis locais. Eles podem ser usados apenas por instruções que estão dentro dessa função ou bloco de código. As variáveis locais não são conhecidas por funções externas às suas. O exemplo a seguir mostra como as variáveis locais são usadas. Aqui todas as variáveis a, b e c são locais para a função main ().

```
#include <stdio.h>

int main () {

  / * declaração de variável local * / int a , b ; int c ;



  / * inicialização real * /
  a = 10 ;
  b = 20 ;
  c = a + b ;

  printf ( "valor de a =% d, b =% de ec =% d \ n" , a , b , c );

  return 0 ;
 }
```

Variáveis globais

Variáveis globais são definidas fora de uma função, geralmente no topo do programa. As variáveis globais mantêm seus valores durante toda a vida útil do programa e podem ser acessadas dentro de qualquer uma das funções definidas para o programa.

Uma variável global pode ser acessada por qualquer função. Ou seja, uma variável global está disponível para uso em todo o seu programa após sua declaração. O programa a seguir mostra como variáveis globais são usadas em um programa.

```
#include <stdio.h>

/* global variable declaration */
int g;

int main () {
```

```
  /* local variable declaration */
  int a, b;

  /* actual initialization */
  a = 10;
  b = 20;
  g = a + b;

  printf ("value of a = %d, b = %d and g = %d\n", a, b, g);

  return 0;
}
```

Um programa pode ter o mesmo nome para variáveis locais e globais, mas o valor da variável local dentro de uma função terá preferência. Aqui está um exemplo -

```
#include <stdio.h>

/* global variable declaration */
int g = 20;

int main () {

  /* local variable declaration */
  int g = 10;

  printf ("value of g = %d\n",  g);

  return 0;
}
```

Quando o código acima é compilado e executado, ele produz o seguinte resultado -

```
value of g = 10
```

Parâmetros Formais

Parâmetros formais, são tratados como variáveis locais com-em uma função e eles têm precedência sobre variáveis globais. A seguir, um exemplo -

```
#include <stdio.h>

/* global variable declaration */
int a = 20;

int main () {

  /* local variable declaration in main function */
  int a = 10;
  int b = 20;
  int c = 0;

  printf ("value of a in main() = %d\n",  a);
  c = sum( a, b);
  printf ("value of c in main() = %d\n",  c);

  return 0;
}

/* function to add two integers */
int sum(int a, int b) {
```

```
   printf ("value of a in sum() = %d\n",  a);
   printf ("value of b in sum() = %d\n",  b);

   return a + b;
}
```

Quando o código acima é compilado e executado, ele produz o seguinte resultado -

```
value of a in main() = 10
value of a in sum() = 10
value of b in sum() = 20
value of c in main() = 30
```

Inicializando Variáveis Locais e Globais

Quando uma variável local é definida, ela não é inicializada pelo sistema, você deve inicializá-la você mesmo. Variáveis globais são inicializadas automaticamente pelo sistema quando você as define da seguinte forma -

Tipo de dados	Valor Padrão Inicial
int	0
Caracteres	'\ 0'
flutuador	0
em dobro	0
ponteiro	NULO

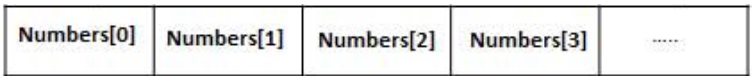
É uma boa prática de programação inicializar as variáveis adequadamente, caso contrário, seu programa poderá produzir resultados inesperados, porque as variáveis não inicializadas terão algum valor de lixo já disponível em seu local de memória.

C - Matrizes

Arrays é um tipo de estrutura de dados que pode armazenar uma coleção sequencial de tamanho fixo de elementos do mesmo tipo. Uma matriz é usada para armazenar uma coleção de dados, mas geralmente é mais útil pensar em uma matriz como uma coleção de variáveis do mesmo tipo.

Em vez de declarar variáveis individuais, como number0, number1, ... e number99, você declara uma variável de matriz, como números, e usa números [0], números [1] e ..., números [99] para representar variáveis individuais. Um elemento específico em uma matriz é acessado por um índice.

Todas as matrizes consistem em locais de memória contíguos. O endereço mais baixo corresponde ao primeiro elemento e o endereço mais alto ao último elemento.



Declarando Matrizes

Para declarar um array em C, um programador especifica o tipo dos elementos e o número de elementos requeridos por um array como segue -

```
type arrayName [ arraySize ];
```

Isso é chamado *de* matriz *unidimensional* . O **arraySize** deve ser uma constante inteira maior que zero e o **tipo** pode ser qualquer tipo de dado C válido. Por exemplo, para declarar uma matriz de 10 elementos chamada **equilíbrio** do tipo double, use esta instrução -

```
double balance[10];
```

Aqui o *equilíbrio* é uma matriz variável que é suficiente para conter até 10 números duplos.

Inicializando matrizes

Você pode inicializar uma matriz em C, um por um ou usando uma única instrução da seguinte maneira -

```
double balance[5] = {1000.0, 2.0, 3.4, 7.0, 50.0};
```

O número de valores entre chaves {} não pode ser maior que o número de elementos que declaramos para o array entre colchetes [].

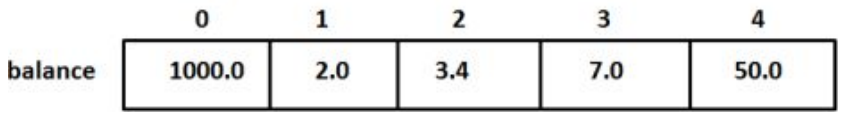
Se você omitir o tamanho da matriz, será criada uma matriz grande o suficiente para manter a inicialização. Portanto, se você escrever -

```
double balance[] = {1000.0, 2.0, 3.4, 7.0, 50.0};
```

Você criará exatamente o mesmo array que no exemplo anterior. A seguir é um exemplo para atribuir um único elemento da matriz -

```
balance[4] = 50.0;
```

A declaração acima atribui o 5^{o} elemento na matriz com um valor de 50,0. Todas as matrizes têm 0 como o índice de seu primeiro elemento, que também é chamado de índice base e o último índice de uma matriz será o tamanho total da matriz menos 1. Abaixo, é apresentada a representação pictórica da matriz discutida acima -



Acessando Elementos da Matriz

Um elemento é acessado pela indexação do nome da matriz. Isso é feito colocando o índice do elemento dentro de colchetes após o nome da matriz. Por exemplo -

```
double salary = balance[9];
```

A declaração acima levará o 10º elemento da matriz e atribuir o valor a variável salário. O exemplo a seguir mostra como usar todos os três conceitos mencionados acima, viz. declaração, atribuição e acesso a matrizes -

```
#include <stdio.h>

int main () {

   int n[ 10 ]; /* n is an array of 10 integers */
   int i,j;

   /* initialize elements of array n to 0 */
   for ( i = 0; i < 10; i++ ) {
      n[ i ] = i + 100; /* set element at location i to i + 100 */
   }

   /* output each array element's value */
   for (j = 0; j < 10; j++ ) {
      printf("Element[%d] = %d\n", j, n[j] );
   }

   return 0;
}
```

Quando o código acima é compilado e executado, ele produz o seguinte resultado -

```
Element[0] = 100
Element[1] = 101
Element[2] = 102
Element[3] = 103
Element[4] = 104
Element[5] = 105
Element[6] = 106
Element[7] = 107
Element[8] = 108
Element[9] = 109
```

Matrizes em detalhe

Matrizes são importantes para C e devem precisar de muito mais atenção. Os seguintes conceitos importantes relacionados ao array devem ser claros para um programador C -

Sr. Não.	Conceito e Descrição
1	**Matrizes multidimensionais** C suporta matrizes multidimensionais. A forma mais simples do array multidimensional é o array bidimensional.
2	**Passando matrizes para funções** Você pode passar para a função um ponteiro para uma matriz, especificando o nome da matriz sem um índice.

3	**Retornar matriz de uma função** C permite que uma função retorne uma matriz.
4	**Ponteiro para um array** Você pode gerar um ponteiro para o primeiro elemento de uma matriz simplesmente especificando o nome da matriz, sem nenhum índice.

C - Ponteiros

Ponteiros em C são fáceis e divertidos de aprender. Algumas tarefas de programação C são executadas mais facilmente com ponteiros e outras tarefas, como alocação de memória dinâmica, não podem ser executadas sem o uso de ponteiros. Por isso, torna-se necessário aprender ponteiros para se tornar um programador C perfeito. Vamos começar a aprendê-las em etapas simples e fáceis.

Como você sabe, cada variável é um local de memória e cada localização de memória tem seu endereço definido que pode ser acessado usando o operador E comercial (&), que denota um endereço na memória. Considere o seguinte exemplo, que imprime o endereço das variáveis definidas -

```
#include <stdio.h>

int main () {

   int  var1;
   char var2[10];

   printf("Address of var1 variable: %x\n", &var1  );
   printf("Address of var2 variable: %x\n", &var2  );

   return 0;
}
```

Quando o código acima é compilado e executado, ele produz o seguinte resultado -

```
Address of var1 variable: bff5a400
Address of var2 variable: bff5a3f6
```

O que são ponteiros?

Um **ponteiro** é uma variável cujo valor é o endereço de outra variável, isto é, endereço direto da localização da memória. Como qualquer variável ou constante, você deve declarar um ponteiro antes de usá-lo para armazenar qualquer endereço variável. A forma geral de uma declaração de variável de ponteiro é -

```
type *var-name;
```

Aqui, **type** é o tipo base do ponteiro; deve ser um tipo de dados C válido e **var-name** é o nome da variável do ponteiro. O asterisco * usado para declarar um ponteiro é o mesmo asterisco usado para multiplicação. No entanto, nesta declaração, o asterisco está sendo usado para designar uma variável como um ponteiro. Dê uma olhada em algumas das

declarações de ponteiro válidas -

```
int    *ip;    /* pointer to an integer */
double *dp;    /* pointer to a double */
float  *fp;    /* pointer to a float */
char   *ch     /* pointer to a character */
```

O tipo de dados real do valor de todos os ponteiros, seja inteiro, flutuante, caractere ou não, é o mesmo, um número hexadecimal longo que representa um endereço de memória. A única diferença entre ponteiros de diferentes tipos de dados é o tipo de dados da variável ou constante para o qual o ponteiro aponta.

Como usar ponteiros?

Existem algumas operações importantes, que faremos com a ajuda de ponteiros com muita frequência. **(a)** Definimos uma variável de ponteiro, **(b)** atribuímos o endereço de uma variável a um ponteiro e **(c)** finalmente acessamos o valor no endereço disponível na variável de ponteiro. Isso é feito usando o operador unário ***** que retorna o valor da variável localizada no endereço especificado pelo seu operando. O exemplo a seguir faz uso dessas operações -

```
#include <stdio.h>

int main () {

   int  var = 20;   /* actual variable declaration */
   int  *ip;        /* pointer variable declaration */

   ip = &var;  /* store address of var in pointer variable*/

   printf("Address of var variable: %x\n", &var  );

   /* address stored in pointer variable */
   printf("Address stored in ip variable: %x\n", ip );

   /* access the value using the pointer */
   printf("Value of *ip variable: %d\n", *ip );

   return 0;
}
```

Quando o código acima é compilado e executado, ele produz o seguinte resultado -

```
Address of var variable: bffd8b3c
Address stored in ip variable: bffd8b3c
Value of *ip variable: 20
```

Ponteiros NULL

É sempre uma boa prática atribuir um valor NULL a uma variável de ponteiro caso você não tenha um endereço exato a ser atribuído. Isso é feito no momento da declaração da variável. Um ponteiro atribuído a NULL é chamado de ponteiro **nulo** .

O ponteiro NULL é uma constante com um valor zero definido em várias bibliotecas padrão. Considere o seguinte programa -

```c
#include <stdio.h>

int main () {

   int  *ptr = NULL;

   printf("The value of ptr is : %x\n", ptr  );

   return 0;
}
```

Quando o código acima é compilado e executado, ele produz o seguinte resultado -

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

Na maioria dos sistemas operacionais, os programas não têm permissão para acessar a memória no endereço 0 porque essa memória é reservada pelo sistema operacional. No entanto, o endereço de memória 0 tem um significado especial; sinaliza que o ponteiro não se destina a apontar para um local de memória acessível. Mas por convenção, se um ponteiro contiver o valor nulo (zero), é assumido que aponta para nada.

Para verificar um ponteiro nulo, você pode usar uma instrução 'if' da seguinte maneira -

```c
if(ptr)     /* succeeds if p is not null */
if(!ptr)    /* succeeds if p is null */
```

Ponteiros em Detalhe

Os ponteiros têm muitos conceitos, mas são muito importantes para a programação em C. Os seguintes conceitos de ponteiro importantes devem estar claros para qualquer programador C -

Sr. Não.	Conceito e Descrição
1	**Aritmética ponteiro** Existem quatro operadores aritméticos que podem ser usados em ponteiros: ++, -, +, -
2	**Matriz de ponteiros** Você pode definir matrizes para manter um número de ponteiros.
3	**Ponteiro para ponteiro** C permite que você tenha ponteiro em um ponteiro e assim por diante.
4	**Passando ponteiros para funções em C** Passar um argumento por referência ou por endereço permite que o argumento transmitido seja alterado na função de chamada pela função chamada.
5	**Retorna o ponteiro das funções em C** C permite que uma função retorne um ponteiro para a variável local, variável estática e memória alocada dinamicamente também.

C - cordas

Strings são na verdade array unidimensional de caracteres terminados por um caractere **nulo** '\ 0'. Portanto, uma cadeia terminada com nulo contém os caracteres que compõem a cadeia, seguidos de um **nulo** .

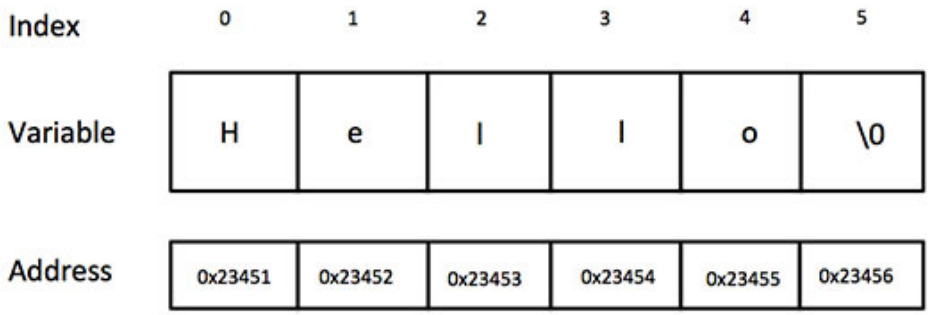
A declaração e inicialização a seguir criam uma string consistindo da palavra "Hello". Para manter o caractere nulo no final da matriz, o tamanho da matriz de caracteres que contém a seqüência de caracteres é um mais do que o número de caracteres na palavra "Hello".

```
char greeting[6] = {'H', 'e', 'l', 'l', 'o', '\0'};
```

Se você seguir a regra de inicialização da matriz, poderá escrever a declaração acima da seguinte maneira -

```
char greeting[] = "Hello";
```

A seguir está a apresentação de memória da cadeia definida acima em C / C ++ -



Na verdade, você não coloca o caractere *nulo* no final de uma constante de string. O compilador C coloca automaticamente o '\ 0' no final da string quando inicializa a matriz. Vamos tentar imprimir a string acima mencionada -

```
#include <stdio.h>

int main () {

   char greeting[6] = {'H', 'e', 'l', 'l', 'o', '\0'};
   printf("Greeting message: %s\n", greeting );
   return 0;
}
```

Quando o código acima é compilado e executado, ele produz o seguinte resultado -

```
Greeting message: Hello
```

C suporta uma ampla gama de funções que manipulam strings terminadas em null -

Sr. Não.	Função e Propósito
1	**strcpy (s1, s2);** Copia a cadeia s2 na cadeia s1.

2	**strcat (s1, s2);** Concatena a string s2 no final da string s1.
3	**strlen (s1);** Retorna o tamanho da string s1.
4	**strcmp (s1, s2);** Retorna 0 se s1 e s2 forem iguais; menor que 0 se s1 <s2; maior que 0 se s1> s2.
5	**strchr (s1, ch);** Retorna um ponteiro para a primeira ocorrência do caractere ch na string s1.
6	**strstr (s1, s2);** Retorna um ponteiro para a primeira ocorrência da string s2 na string s1.

O exemplo a seguir usa algumas das funções mencionadas acima -

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

int main () {

   char str1[12] = "Hello";
   char str2[12] = "World";
   char str3[12];
   int  len ;

   /* copy str1 into str3 */
   strcpy(str3, str1);
   printf("strcpy( str3, str1) :  %s\n", str3 );

   /* concatenates str1 and str2 */
   strcat( str1, str2);
   printf("strcat( str1, str2):   %s\n", str1 );

   /* total lenghth of str1 after concatenation */
   len = strlen(str1);
   printf("strlen(str1) :  %d\n", len );

   return 0;
}
```

Quando o código acima é compilado e executado, ele produz o seguinte resultado -

```
strcpy( str3, str1) :  Hello
strcat( str1, str2):   HelloWorld
strlen(str1) :  10
```

C - Estruturas

As matrizes permitem definir o tipo de variáveis que podem conter vários itens de dados do mesmo tipo. Da mesma forma **estrutura** é outro tipo de dados definidos pelo usuário disponível em C que permite combinar itens de dados de diferentes tipos.

Estruturas são usadas para representar um registro. Suponha que você queira acompanhar seus livros em uma biblioteca. Você pode acompanhar os seguintes atributos sobre cada livro:

- Título
- Autor
- Sujeito
- ID do livro

Definindo uma Estrutura

Para definir uma estrutura, você deve usar a instrução **struct** . A instrução struct define um novo tipo de dados, com mais de um membro. O formato da instrução struct é o seguinte -

```
struct [structure tag] {

   member definition;
   member definition;
   ...
   member definition;
} [one or more structure variables];
```

A **tag de estrutura** é opcional e cada definição de membro é uma definição de variável normal, como int i; ou float f; ou qualquer outra definição de variável válida. No final da definição da estrutura, antes do ponto-e-vírgula final, você pode especificar uma ou mais variáveis de estrutura, mas é opcional. Aqui está a maneira como você declararia a estrutura do livro -

```
struct Books {
   char  title[50];
   char  author[50];
   char  subject[100];
   int   book_id;
} book;
```

Acessando Membros da Estrutura

Para acessar qualquer membro de uma estrutura, usamos o **operador de acesso de membro (.)** . O operador de acesso de membro é codificado como um período entre o nome da variável de estrutura e o membro da estrutura que desejamos acessar. Você usaria a palavra-chave **struct** para definir variáveis do tipo de estrutura. O exemplo a seguir mostra como usar uma estrutura em um programa -

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

struct Books {
```

```
   char  title[50];
   char  author[50];
   char  subject[100];
   int   book_id;
};

int main( ) {

   struct Books Book1;        /* Declare Book1 of type Book */
   struct Books Book2;        /* Declare Book2 of type Book */

   /* book 1 specification */
   strcpy( Book1.title, "C Programming");
   strcpy( Book1.author, "Nuha Ali");
   strcpy( Book1.subject, "C Programming Tutorial");
   Book1.book_id = 6495407;

   /* book 2 specification */
   strcpy( Book2.title, "Telecom Billing");
   strcpy( Book2.author, "Zara Ali");
   strcpy( Book2.subject, "Telecom Billing Tutorial");
   Book2.book_id = 6495700;

   /* print Book1 info */
   printf( "Book 1 title : %s\n", Book1.title);
   printf( "Book 1 author : %s\n", Book1.author);
   printf( "Book 1 subject : %s\n", Book1.subject);
   printf( "Book 1 book_id : %d\n", Book1.book_id);

   /* print Book2 info */
   printf( "Book 2 title : %s\n", Book2.title);
   printf( "Book 2 author : %s\n", Book2.author);
   printf( "Book 2 subject : %s\n", Book2.subject);
   printf( "Book 2 book_id : %d\n", Book2.book_id);

   return 0;
}
```

Quando o código acima é compilado e executado, ele produz o seguinte resultado -

```
Book 1 title : C Programming
Book 1 author : Nuha Ali
Book 1 subject : C Programming Tutorial
Book 1 book_id : 6495407
Book 2 title : Telecom Billing
Book 2 author : Zara Ali
Book 2 subject : Telecom Billing Tutorial
Book 2 book_id : 6495700
```

Estruturas como Argumentos da Função

Você pode passar uma estrutura como um argumento de função da mesma forma que você passa qualquer outra variável ou ponteiro.

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

struct Books {
   char  title[50];
   char  author[50];
   char  subject[100];
```

```
   int   book_id;
};

/* function declaration */
void printBook( struct Books book );

int main( ) {

   struct Books Book1;        /* Declare Book1 of type Book */
   struct Books Book2;        /* Declare Book2 of type Book */

   /* book 1 specification */
   strcpy( Book1.title, "C Programming");
   strcpy( Book1.author, "Nuha Ali");
   strcpy( Book1.subject, "C Programming Tutorial");
   Book1.book_id = 6495407;

   /* book 2 specification */
   strcpy( Book2.title, "Telecom Billing");
   strcpy( Book2.author, "Zara Ali");
   strcpy( Book2.subject, "Telecom Billing Tutorial");
   Book2.book_id = 6495700;

   /* print Book1 info */
   printBook( Book1 );

   /* Print Book2 info */
   printBook( Book2 );

   return 0;
}

void printBook( struct Books book ) {

   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);
}
```

Quando o código acima é compilado e executado, ele produz o seguinte resultado -

```
Book title : C Programming
Book author : Nuha Ali
Book subject : C Programming Tutorial
Book book_id : 6495407
Book title : Telecom Billing
Book author : Zara Ali
Book subject : Telecom Billing Tutorial
Book book_id : 6495700
```

Ponteiros para Estruturas

Você pode definir ponteiros para estruturas da mesma forma que você define o ponteiro para qualquer outra variável -

```
struct Books *struct_pointer;
```

Agora, você pode armazenar o endereço de uma variável de estrutura na variável de ponteiro definida acima. Para encontrar o endereço de uma variável de estrutura, coloque

o '&'; operador antes do nome da estrutura da seguinte forma -

```
struct_pointer = &Book1;
```

Para acessar os membros de uma estrutura usando um ponteiro para essa estrutura, você deve usar o operador → da seguinte maneira -

```
struct_pointer->title;
```

Vamos reescrever o exemplo acima usando o ponteiro de estrutura.

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

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

/* function declaration */
void printBook( struct Books *book );
int main( ) {

   struct Books Book1;        /* Declare Book1 of type Book */
   struct Books Book2;        /* Declare Book2 of type Book */

   /* book 1 specification */
   strcpy( Book1.title, "C Programming");
   strcpy( Book1.author, "Nuha Ali");
   strcpy( Book1.subject, "C Programming Tutorial");
   Book1.book_id = 6495407;

   /* book 2 specification */
   strcpy( Book2.title, "Telecom Billing");
   strcpy( Book2.author, "Zara Ali");
   strcpy( Book2.subject, "Telecom Billing Tutorial");
   Book2.book_id = 6495700;

   /* print Book1 info by passing address of Book1 */
   printBook( &Book1 );

   /* print Book2 info by passing address of Book2 */
   printBook( &Book2 );

   return 0;
}

void printBook( struct Books *book ) {

   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);
}
```

Quando o código acima é compilado e executado, ele produz o seguinte resultado -

```
Book title : C Programming

Book author : Nuha Ali

Book subject : C Programming Tutorial
```

```
Book book_id : 6495407
Book title : Telecom Billing
Book author : Zara Ali
Book subject : Telecom Billing Tutorial
Book book_id : 6495700
```

Bit Fields

Os campos de bits permitem o empacotamento de dados em uma estrutura. Isso é especialmente útil quando a memória ou armazenamento de dados é um prêmio. Exemplos típicos incluem -

- Embalagem de vários objetos em uma palavra de máquina. Por exemplo, os sinalizadores de 1 bit podem ser compactados.
- Leitura de formatos de arquivos externos - formatos de arquivo não padrão podem ser lidos, por exemplo, inteiros de 9 bits.

C nos permite fazer isso em uma definição de estrutura, colocando: comprimento de bit após a variável. Por exemplo -

```
struct packed_struct {
   unsigned int f1:1;
   unsigned int f2:1;
   unsigned int f3:1;
   unsigned int f4:1;
   unsigned int type:4;
   unsigned int my_int:9;
} pack;
```

Aqui, o packed_struct contém 6 membros: Quatro sinalizadores de 1 bit f1..f3, um tipo de 4 bits e um my_int de 9 bits.

C empacota automaticamente os campos de bits acima da forma mais compacta possível, desde que o comprimento máximo do campo seja menor ou igual ao tamanho inteiro da palavra do computador. Se este não for o caso, alguns compiladores podem permitir a sobreposição de memória para os campos enquanto outros armazenariam o próximo campo na próxima palavra.

C - sindicatos

Uma **união** é um tipo de dados especial disponível em C que permite armazenar diferentes tipos de dados no mesmo local de memória. Você pode definir uma união com muitos membros, mas apenas um membro pode conter um valor a qualquer momento. As uniões fornecem uma maneira eficiente de usar o mesmo local de memória para vários propósitos.

Definindo uma União

Para definir uma união, você deve usar a instrução **union** da mesma forma que definiu uma estrutura. A instrução union define um novo tipo de dados com mais de um membro para o seu programa. O formato da declaração da união é o seguinte -

```
union [union tag] {
   member definition;
   member definition;
   ...
   member definition;
} [one or more union variables];
```

A **tag union** é opcional e cada definição de membro é uma definição de variável normal, como int i; ou float f; ou qualquer outra definição de variável válida. No final da definição do sindicato, antes do ponto-e-vírgula final, você pode especificar uma ou mais variáveis de união, mas é opcional. Aqui está a maneira como você definiria um tipo de união chamado Data com três membros i, f e str -

```
union Data {
   int i;
   float f;
   char str[20];
} data;
```

Agora, uma variável de tipo de **dados** pode armazenar um número inteiro, um número de ponto flutuante ou uma seqüência de caracteres. Isso significa que uma única variável, ou seja, a mesma localização de memória, pode ser usada para armazenar vários tipos de dados. Você pode usar qualquer tipo de dados interno ou definido pelo usuário dentro de uma união com base em sua necessidade.

A memória ocupada por um sindicato será grande o suficiente para manter o maior membro do sindicato. Por exemplo, no exemplo acima, o tipo de dados ocupará 20 bytes de espaço de memória, porque esse é o espaço máximo que pode ser ocupado por uma cadeia de caracteres. O exemplo a seguir exibe o tamanho total da memória ocupado pela união acima -

```
#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;
}
```

Quando o código acima é compilado e executado, ele produz o seguinte resultado -

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

Acessando Membros da União

Para acessar qualquer membro de um sindicato, usamos o **operador de acesso de**

membro (.) . O operador de acesso de membro é codificado como um período entre o nome da variável de união e o membro do sindicato que desejamos acessar. Você usaria a **união de** palavras-chave para definir variáveis do tipo de união. O exemplo a seguir mostra como usar uniões em um programa -

```
#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;
}
```

Quando o código acima é compilado e executado, ele produz o seguinte resultado -

```
data.i : 1917853763
data.f : 4122360580327794860452759994368.000000
data.str : C Programming
```

Aqui, podemos ver que os valores de **i** e **f** membros do sindicato foram corrompidos porque o valor final atribuído à variável ocupou o local da memória e essa é a razão pela qual o valor do membro **str** está sendo impresso muito bem.

Agora vamos olhar para o mesmo exemplo mais uma vez, onde vamos usar uma variável de cada vez, que é o principal objetivo de ter sindicatos -

```
#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;
}
```

Quando o código acima é compilado e executado, ele produz o seguinte resultado -

```
data.i : 10
data.f : 220.500000
data.str : C Programming
```

Aqui, todos os membros estão sendo impressos muito bem porque um membro está sendo usado por vez.

Campos C-Bit

Suponha que seu programa C contenha um número de variáveis TRUE / FALSE agrupadas em uma estrutura chamada status, como segue -

```
struct {
   unsigned int widthValidated;
   unsigned int heightValidated;
} status;
```

Essa estrutura requer 8 bytes de espaço de memória, mas, na verdade, armazenaremos 0 ou 1 em cada uma das variáveis. A linguagem de programação C oferece uma maneira melhor de utilizar o espaço da memória em tais situações.

Se você estiver usando essas variáveis dentro de uma estrutura, então você pode definir a largura de uma variável que diz ao compilador C que você vai usar apenas o número de bytes. Por exemplo, a estrutura acima pode ser reescrita da seguinte maneira -

```
struct {
   unsigned int widthValidated : 1;
   unsigned int heightValidated : 1;
} status;
```

A estrutura acima requer 4 bytes de espaço de memória para a variável de status, mas apenas 2 bits serão usados para armazenar os valores.

Se você usar até 32 variáveis, cada uma com uma largura de 1 bit, a estrutura de status também usará 4 bytes. No entanto, assim que você tiver 33 variáveis, ele alocará o próximo slot da memória e começará a usar 8 bytes. Vamos verificar o seguinte exemplo para entender o conceito -

```
#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;
}
```

Quando o código acima é compilado e executado, ele produz o seguinte resultado -

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

Declaração de campo de bits

A declaração de um campo de bits tem o seguinte formulário dentro de uma estrutura -

```
struct {
   type [member_name] : width ;
};
```

A tabela a seguir descreve os elementos variáveis de um campo de bits -

Sr. Não.	Elemento e Descrição
1	**tipo** Um tipo inteiro que determina como o valor de um campo de bits é interpretado. O tipo pode ser int, signed int ou unsigned int.
2	**nome do membro** O nome do campo de bits.
3	**largura** O número de bits no campo de bits. A largura deve ser menor ou igual à largura de bit do tipo especificado.

As variáveis definidas com uma largura predefinida são chamadas de **campos de bits** . Um campo de bits pode conter mais que um único bit; por exemplo, se você precisa de uma variável para armazenar um valor de 0 a 7, então você pode definir um campo de bit com uma largura de 3 bits da seguinte maneira -

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

A definição de estrutura acima instrui o compilador C que a variável age usará apenas 3

bits para armazenar o valor. Se você tentar usar mais de 3 bits, isso não permitirá que você faça isso. Vamos tentar o seguinte exemplo -

```
#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;
}
```

Quando o código acima é compilado, ele será compilado com um aviso e, quando executado, produzirá o seguinte resultado -

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

C - typedef

A linguagem de programação C fornece uma palavra-chave chamada **typedef** , que você pode usar para dar um novo nome a um tipo. A seguir, um exemplo para definir um termo **BYTE** para números de um byte -

```
typedef unsigned char BYTE;
```

Após essa definição de tipo, o identificador BYTE pode ser usado como uma abreviação para o tipo **unsigned char, por exemplo.** .

```
BYTE  b1, b2;
```

Por convenção, letras maiúsculas são usadas para essas definições para lembrar ao usuário que o nome do tipo é realmente uma abreviação simbólica, mas você pode usar letras minúsculas, como segue -

```
typedef unsigned char byte;
```

Você também pode usar **typedef** para dar um nome aos seus tipos de dados definidos pelo usuário. Por exemplo, você pode usar typedef com estrutura para definir um novo tipo de dados e, em seguida, usar esse tipo de dados para definir variáveis de estrutura diretamente da seguinte maneira -

```c
#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;
}
```

Quando o código acima é compilado e executado, ele produz o seguinte resultado -

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

typedef vs #define

#define é uma diretiva C que também é usada para definir os aliases para vários tipos de dados semelhantes ao **typedef,** mas com as seguintes diferenças -

> **typedef** é limitado a dar nomes simbólicos a tipos somente onde como **#define** pode ser usado para definir alias para valores também, q., você pode definir 1 como ONE etc.
>
> **A** interpretação **typedef** é executada pelo compilador, enquanto as instruções **#define** são processadas pelo pré-processador.

O exemplo a seguir mostra como usar #define em um programa -

```c
#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;
}
```

Quando o código acima é compilado e executado, ele produz o seguinte resultado -

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

C - Entrada e Saída

Quando dizemos **Input** , isso significa alimentar alguns dados em um programa. Uma entrada pode ser dada na forma de um arquivo ou da linha de comando. A programação C fornece um conjunto de funções incorporadas para ler a entrada dada e alimentá-la ao programa conforme a necessidade.

Quando dizemos **Saída** , significa exibir alguns dados na tela, na impressora ou em qualquer arquivo. A programação C fornece um conjunto de funções internas para produzir os dados na tela do computador, bem como salvá-los em arquivos de texto ou binários.

Os arquivos padrão

A programação C trata todos os dispositivos como arquivos. Assim, dispositivos como o monitor são endereçados da mesma forma que os arquivos e os três arquivos a seguir são abertos automaticamente quando um programa é executado para fornecer acesso ao teclado e à tela.

Arquivo Padrão	Ponteiro de Arquivo	Dispositivo
Entrada padrão	stdin	Teclado
Saída padrão	stdout	Tela
Erro padrão	stderr	Sua tela

Os ponteiros de arquivo são os meios para acessar o arquivo para fins de leitura e escrita. Esta seção explica como ler valores da tela e como imprimir o resultado na tela.

As funções getchar () e putchar ()

A **função int getchar (void)** lê o próximo caractere disponível na tela e o retorna como um inteiro. Esta função lê apenas um caractere de cada vez. Você pode usar este método no loop caso queira ler mais de um caractere na tela.

A função **int putchar (int c)** coloca o caractere passado na tela e retorna o mesmo caractere. Essa função coloca apenas um caractere por vez. Você pode usar este método no loop caso queira exibir mais de um caractere na tela. Verifique o seguinte exemplo -

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

   int c;

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

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

   return 0;
}
```

Quando o código acima é compilado e executado, ele espera que você insira algum texto. Quando você insere um texto e pressiona enter, o programa continua e lê apenas um único caractere e o exibe da seguinte maneira -

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

As funções gets () e puts ()

A função **char * gets (char * s)** lê uma linha do **stdin** no buffer apontado por **s** até uma nova linha de finalização ou EOF (End of File).

A função **int puts (const char * s)** escreve a string 's' e 'a' à direita da **stdout** .

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

   char str[100];

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

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

   return 0;
}
```

Quando o código acima é compilado e executado, ele espera que você insira algum texto. Quando você insere um texto e pressiona enter, o programa prossegue e lê a linha completa até o final e a exibe da seguinte maneira -

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

As funções scanf () e printf ()

A função **scanf int (const char * format, ...)** lê a entrada do **stdin do** fluxo de entrada padrão e varre aquela entrada de acordo com o **formato** fornecido.

A **função int printf (const char * format, ...)** grava a saída no **stdout do** fluxo de saída **padrão** e produz a saída de acordo com o formato fornecido.

O **formato** pode ser uma string constante simples, mas você pode especificar% s,% d,% c,% f, etc., para imprimir ou ler strings, integer, character ou float, respectivamente. Existem muitas outras opções de formatação disponíveis que podem ser usadas com base nos requisitos. Vamos agora prosseguir com um exemplo simples para entender melhor os

conceitos -

```
#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;
}
```

Quando o código acima é compilado e executado, ele espera que você insira algum texto. Quando você insere um texto e pressiona enter, o programa continua e lê a entrada e a exibe da seguinte maneira -

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

Aqui, deve-se notar que scanf () espera entrada no mesmo formato que você forneceu% se% d, o que significa que você tem que fornecer entradas válidas como "string integer". Se você fornecer "string string" ou "integer integer", então será assumido como entrada errada. Em segundo lugar, durante a leitura de uma string, scanf () pára de ler assim que encontra um espaço, então "this is test" são três strings para scanf ().

C - File I / O

O último capítulo explicou os dispositivos de entrada e saída padrão manipulados pela linguagem de programação C. Este capítulo aborda como os programadores C podem criar, abrir, fechar arquivos de texto ou binários para seu armazenamento de dados.

Um arquivo representa uma seqüência de bytes, independentemente de ser um arquivo de texto ou um arquivo binário. A linguagem de programação C fornece acesso a funções de alto nível, bem como chamadas de baixo nível (no nível do sistema operacional) para manipular arquivos em seus dispositivos de armazenamento. Este capítulo irá guiá-lo pelas importantes chamadas para o gerenciamento de arquivos.

Abrindo Arquivos

Você pode usar a função **fopen ()** para criar um novo arquivo ou abrir um arquivo existente. Essa chamada inicializará um objeto do tipo **FILE** , que contém todas as informações necessárias para controlar o fluxo. O protótipo desta chamada de função é o seguinte -

```
FILE *fopen( const char * filename, const char * mode );
```

Aqui, **filename** é uma string literal, que você usará para nomear seu arquivo, e o **modo de** acesso pode ter um dos seguintes valores -

Sr. Não.	Modo e descrição
1	**r** Abre um arquivo de texto existente para fins de leitura.
2	**W** Abre um arquivo de texto para gravação. Se não existir, um novo arquivo será criado. Aqui o seu programa começará a escrever conteúdo desde o início do arquivo.
3	**uma** Abre um arquivo de texto para gravação no modo de anexação. Se não existir, um novo arquivo será criado. Aqui o seu programa irá começar a adicionar conteúdo no conteúdo do arquivo existente.
4	**r +** Abre um arquivo de texto para leitura e gravação.
5	**w +** Abre um arquivo de texto para leitura e gravação. Primeiro trunca o arquivo para comprimento zero se existir, caso contrário, cria um arquivo se ele não existir.
6	**um +** Abre um arquivo de texto para leitura e gravação. Cria o arquivo se ele não existir. A leitura começará do começo, mas a escrita só pode ser acrescentada.

Se você for lidar com arquivos binários, então você usará os seguintes modos de acesso ao invés dos mencionados acima -

```
"rb", "wb", "ab", "rb+", "r+b", "wb+", "w+b", "ab+", "a+b"
```

Fechando um arquivo

Para fechar um arquivo, use a função fclose (). O protótipo desta função é -

```
int fclose( FILE *fp );
```

A função **fclose (-)** retorna zero em caso de sucesso ou **EOF** se houver um erro ao fechar o arquivo. Essa função realmente libera todos os dados ainda pendentes no buffer para o arquivo, fecha o arquivo e libera qualquer memória usada para o arquivo. O EOF é uma constante definida no arquivo de cabeçalho **stdio.h** .

Existem várias funções fornecidas pela biblioteca padrão C para ler e gravar um arquivo, caractere por caractere ou na forma de uma cadeia de comprimento fixo.

Escrevendo um arquivo

A seguir, a função mais simples de escrever caracteres individuais em um fluxo -

```
int fputc( int c, FILE *fp );
```

A função **fputc ()** escreve o valor do caractere do argumento c no fluxo de saída referenciado por fp. Ele retorna o caractere escrito escrito no sucesso, caso contrário, **EOF,** se houver um erro. Você pode usar as seguintes funções para gravar uma string terminada em null em um fluxo -

```
int fputs( const char *s, FILE *fp );
```

A função **fputs ()** grava a string **s** no fluxo de saída referenciado por fp. Retorna um valor não negativo ao sucesso, caso contrário, o **EOF** é retornado em caso de qualquer erro. Você pode usar a **função fprintf (FILE * fp, const char * format, ...)** para gravar uma string em um arquivo. Tente o seguinte exemplo.

Certifique-se de ter o **diretório / tmp** disponível. Se não estiver, então antes de prosseguir, você deve criar esse diretório em sua máquina.

```
#include <stdio.h>

main() {
   FILE *fp;

   fp = fopen("/tmp/test.txt", "w+");
   fprintf(fp, "This is testing for fprintf...\n");
   fputs("This is testing for fputs...\n", fp);
   fclose(fp);
}
```

Quando o código acima é compilado e executado, ele cria um novo arquivo **test.txt** no diretório / tmp e grava duas linhas usando duas funções diferentes. Vamos ler esse arquivo na próxima seção.

Lendo um arquivo

Dada a seguir é a função mais simples para ler um único caractere de um arquivo -

```
int fgetc( FILE * fp );
```

A função **fgetc ()** lê um caractere do arquivo de entrada referenciado por fp. O valor de retorno é o caractere lido ou, em caso de erro, retorna **EOF** . A seguinte função permite ler uma string de um fluxo -

```
char *fgets( char *buf, int n, FILE *fp );
```

As funções **fgets ()** lêem até n-1 caracteres do fluxo de entrada referenciado por fp. Ele copia a cadeia de leitura para o buffer **buf** , anexando um caractere **nulo** para finalizar a cadeia.

Se essa função encontrar um caractere de nova linha '\ n' ou o final do arquivo EOF antes de ler o número máximo de caracteres, ele retornará apenas os caracteres lidos até esse ponto, incluindo o novo caractere de linha. Você também pode usar a função **fscanf (FILE * fp, const char * format, ...)** para ler seqüências de caracteres de um arquivo, mas ele pára de ler depois de encontrar o primeiro caractere de espaço.

```
#include <stdio.h>

main() {

   FILE *fp;
   char buff[255];

   fp = fopen("/tmp/test.txt", "r");
   fscanf(fp, "%s", buff);
   printf("1 : %s\n", buff );

   fgets(buff, 255, (FILE*)fp);
   printf("2: %s\n", buff );

   fgets(buff, 255, (FILE*)fp);
   printf("3: %s\n", buff );
   fclose(fp);

}
```

Quando o código acima é compilado e executado, ele lê o arquivo criado na seção anterior e produz o seguinte resultado -

```
1 : This
2: is testing for fprintf...


3: This is testing for fputs...
```

Vamos ver um pouco mais detalhadamente sobre o que aconteceu aqui. Primeiro, **fscanf ()** lê apenas **Isto** porque depois disso, ele encontrou um espaço, a segunda chamada é para **fgets ()** que lê a linha restante até encontrar o fim da linha. Finalmente, a última chamada **fgets ()** lê a segunda linha completamente.

Funções binárias de E / S

Existem duas funções, que podem ser usadas para entrada e saída binárias -

```
size_t fread(void *ptr, size_t size_of_elements, size_t number_of_elements, FILE *a_file);

size_t fwrite(const void *ptr, size_t size_of_elements, size_t number_of_elements, FILE *a_file);
```

Ambas estas funções devem ser usadas para ler ou escrever blocos de memórias - geralmente matrizes ou estruturas.

C - Pré-processadores

O **pré** - **processador C** não faz parte do compilador, mas é uma etapa separada no processo de compilação. Em termos simples, um pré-processador C é apenas uma ferramenta de substituição de texto e instrui o compilador a fazer o pré-processamento

necessário antes da compilação real. Vamos nos referir ao pré-processador C como CPP.

Todos os comandos do pré-processador começam com um símbolo de hash (#). Deve ser o primeiro caractere não vazio e, para facilitar a leitura, uma diretiva de pré-processador deve começar na primeira coluna. A seção a seguir lista todas as diretivas importantes do pré-processador -

Sr. Não.	Diretiva e Descrição
1	**#definir** Substitui uma macro de pré-processador.
2	**#incluir** Insere um cabeçalho específico de outro arquivo.
3	**#undef** Não define uma macro de pré-processamento.
4	**#ifdef** Retorna true se essa macro for definida.
5	**#ifndef** Retorna true se essa macro não estiver definida.
6	**#E se** Testa se uma condição de tempo de compilação é verdadeira.
7	**#outro** A alternativa para #if.
8	**#elif** #else e #if em uma declaração.
9	**#fim se** Termina o pré-processador condicional.
10	**#erro** Imprime mensagem de erro no stderr.

11	**#pragma** Emite comandos especiais para o compilador, usando um método padronizado.

Exemplos de pré-processadores

Analise os exemplos a seguir para entender várias diretivas.

```
#define MAX_ARRAY_LENGTH 20
```

Esta diretiva diz ao CPP para substituir instâncias de MAX_ARRAY_LENGTH por 20. Use *#define* para constantes para aumentar a legibilidade.

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

Essas diretivas dizem ao CPP para obter o stdio.h das **Bibliotecas** do **Sistema** e adicionar o texto ao arquivo de origem atual. A próxima linha informa ao CPP para obter **myheader.h** do diretório local e adicionar o conteúdo ao arquivo de origem atual.

```
#undef  FILE_SIZE
#define FILE_SIZE 42
```

Diz ao CPP para indefinir o FILE_SIZE existente e defini-lo como 42.

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

Ele informa ao CPP para definir MENSAGEM somente se MENSAGEM não estiver definida.

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

Ele informa ao CPP para processar as instruções incluídas, se o DEBUG estiver definido. Isso é útil se você passar o flag *-DDEBUG* para o compilador gcc no momento da compilação. Isso definirá o DEBUG, para que você possa ativar e desativar a depuração durante a compilação.

Macros predefinidas

ANSI C define um número de macros. Embora cada um esteja disponível para uso na programação, as macros predefinidas não devem ser modificadas diretamente.

Sr. Não.	Macro e Descrição
1	**__ENCONTRO__** A data atual como um literal de caractere no formato "MMM DD AAAA".

2	**__TEMPO__** A hora atual como um literal de caractere no formato "HH: MM: SS".
3	**__ARQUIVO__** Isso contém o nome do arquivo atual como um literal de string.
4	**__LINHA__** Contém o número da linha atual como uma constante decimal.
5	**__STDC__** Definido como 1 quando o compilador está em conformidade com o padrão ANSI.

Vamos tentar o seguinte exemplo -

```
#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__ );

}
```

Quando o código acima em um arquivo **test.c** é compilado e executado, ele produz o seguinte resultado -

```
File :test.c
Date :Jun 2 2012
Time :03:36:24
Line :8
ANSI :1
```

Operadores de pré-processador

O pré-processador C oferece os seguintes operadores para ajudar a criar macros -

O operador de continuação de macro (\)

Uma macro é normalmente confinada a uma única linha. O operador de continuação de macro (\) é usado para continuar uma macro que é muito longa para uma única linha. Por exemplo -

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

O operador Stringize (#)

O operador de string ou de sinal de número ('#'), quando usado em uma definição de macro, converte um parâmetro de macro em uma constante de string. Este operador pode ser usado apenas em uma macro que tenha um argumento especificado ou uma lista de parâmetros. Por exemplo -

```
#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;
}
```

Quando o código acima é compilado e executado, ele produz o seguinte resultado -

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

O Operador Token Colando (##)

O operador de colagem de tokens (##) dentro de uma definição de macro combina dois argumentos. Ele permite que dois tokens separados na definição de macro sejam unidos em um único token. Por exemplo -

```
#include <stdio.h>

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

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

Quando o código acima é compilado e executado, ele produz o seguinte resultado -

```
token34 = 40
```

Isso aconteceu porque este exemplo resulta na seguinte saída real do pré-processador -

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

Este exemplo mostra a concatenação do token ## n no token34 e aqui usamos tanto o **string** quanto o **token- pasting** .

O Operador Definido ()

O operador **definido pelo** pré-processador é usado em expressões constantes para determinar se um identificador é definido usando #define. Se o identificador especificado estiver definido, o valor será true (diferente de zero). Se o símbolo não estiver definido, o valor é falso (zero). O operador definido é especificado da seguinte maneira -

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

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

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

Quando o código acima é compilado e executado, ele produz o seguinte resultado -

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

Macros parametrizadas

Uma das funções poderosas do CPP é a capacidade de simular funções usando macros parametrizadas. Por exemplo, podemos ter algum código para o quadrado de um número da seguinte forma -

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

Podemos reescrever acima do código usando uma macro da seguinte forma -

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

As macros com argumentos devem ser definidas usando a diretiva **#define** antes que possam ser usadas. A lista de argumentos é colocada entre parênteses e deve seguir imediatamente o nome da macro. Os espaços não são permitidos entre o nome da macro e os parênteses abertos. Por exemplo -

```
#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;
}
```

Quando o código acima é compilado e executado, ele produz o seguinte resultado -

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

C - Arquivos de cabeçalho

Um arquivo de cabeçalho é um arquivo com extensão **.h** que contém declarações de função C e definições de macro a serem compartilhadas entre vários arquivos de origem. Existem dois tipos de arquivos de cabeçalho: os arquivos que o programador escreve e os arquivos que vêm com o seu compilador.

Você solicita o uso de um arquivo de cabeçalho em seu programa, incluindo-o com a diretiva de pré-processamento de C **#include** , como se você tivesse visto a inclusão do arquivo de cabeçalho **stdio.h** , que vem junto com seu compilador.

Incluir um arquivo de cabeçalho é igual a copiar o conteúdo do arquivo de cabeçalho, mas não o fazemos porque ele será propenso a erros e não é uma boa idéia copiar o conteúdo de um arquivo de cabeçalho nos arquivos de origem, especialmente se tem vários arquivos de origem em um programa.

Uma prática simples em programas C ou C ++ é que mantemos todas as constantes, macros, variáveis globais de todo o sistema e protótipos de funções nos arquivos de cabeçalho e incluímos esse arquivo de cabeçalho onde quer que seja necessário.

Incluir sintaxe

Tanto o usuário quanto os arquivos de cabeçalho do sistema são incluídos usando a diretiva de pré-processamento **#include** . Tem as duas formas a seguir -

```
#include <file>
```

Este formulário é usado para arquivos de cabeçalho do sistema. Ele procura por um arquivo chamado 'arquivo' em uma lista padrão de diretórios do sistema. Você pode preceder diretórios a esta lista com a opção -I enquanto compila seu código-fonte.

```
#include "file"
```

Este formulário é usado para arquivos de cabeçalho do seu próprio programa. Ele procura por um arquivo chamado 'file' no diretório que contém o arquivo atual. Você pode preceder diretórios a esta lista com a opção -I enquanto compila seu código-fonte.

Incluir Operação

A diretiva **#include** funciona direcionando o pré-processador C para varrer o arquivo especificado como entrada antes de continuar com o restante do arquivo de origem atual. A saída do pré-processador contém a saída já gerada, seguida pela saída resultante do arquivo incluído, seguida pela saída que vem do texto após a diretiva **#include** . Por exemplo, se você tiver um cabeçalho de arquivo de cabeçalho.h da seguinte maneira -

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

e um programa principal chamado *program.c* que usa o arquivo de cabeçalho, como este -

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

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

o compilador verá o mesmo fluxo de token como se fosse lido por program.c.

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

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

Cabeçalhos Únicos

Se um arquivo de cabeçalho for incluído duas vezes, o compilador processará seu conteúdo duas vezes e isso resultará em um erro. A maneira padrão de evitar isso é incluir todo o conteúdo real do arquivo em um condicional, como este -

```
#ifndef HEADER_FILE
#define HEADER_FILE

the entire header file file

#endif
```

Essa construção é comumente conhecida como um wrapper **#ifndef** . Quando o cabeçalho é incluído novamente, o condicional será falso, porque HEADER_FILE está definido. O pré-processador ignorará todo o conteúdo do arquivo e o compilador não o verá duas vezes.

Incluído computado

Às vezes é necessário selecionar um dos vários arquivos de cabeçalho diferentes a serem incluídos em seu programa. Por exemplo, eles podem especificar parâmetros de configuração para serem usados em diferentes tipos de sistemas operacionais. Você poderia fazer isso com uma série de condicionais da seguinte maneira -

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

Mas à medida que cresce, torna-se tedioso, em vez disso, o pré-processador oferece a capacidade de usar uma macro para o nome do cabeçalho. Isso é chamado de **inclusão computada** . Em vez de escrever um nome de cabeçalho como o argumento direto de **#include** , você simplesmente coloca um nome de macro lá -

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

SYSTEM_H será expandido e o pré-processador procurará system_1.h como se o **#include** tivesse sido escrito dessa forma originalmente. SYSTEM_H pode ser definido pelo seu Makefile com uma opção -D.

C - Type Casting

A conversão de tipos é uma maneira de converter uma variável de um tipo de dados para outro tipo de dados. Por exemplo, se você quiser armazenar um valor 'longo' em um inteiro simples, então você pode digitar 'long' para 'int'. Você pode converter os valores de um tipo para outro explicitamente usando o **operador de** conversão da seguinte maneira -

```
(type_name) expression
```

Considere o exemplo a seguir, em que o operador de conversão faz com que a divisão de uma variável inteira por outra seja executada como uma operação de ponto flutuante -

```
#include <stdio.h>

main() {

   int sum = 17, count = 5;
   double mean;

   mean = (double) sum / count;
   printf("Value of mean : %f\n", mean );
}
```

Quando o código acima é compilado e executado, ele produz o seguinte resultado -

```
Value of mean : 3.400000
```

Deve-se notar aqui que o operador de elenco tem precedência sobre a divisão, então o valor da **soma** é primeiro convertido em tipo **duplo** e, finalmente, é dividido por contagem, resultando em um valor duplo.

As conversões de tipo podem ser implícitas, o que é executado pelo compilador automaticamente, ou pode ser especificado explicitamente através do uso do **operador de** conversão . É considerado uma boa prática de programação usar o operador de conversão sempre que forem necessárias conversões de tipo.

Promoção Integer

Promoção inteira é o processo pelo qual os valores do tipo inteiro "menor" que **int** ou **unsigned int** são convertidos para **int** ou **unsigned int** . Considere um exemplo de adicionar um caractere com um inteiro

```
#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 );
}
```

Quando o código acima é compilado e executado, ele produz o seguinte resultado -

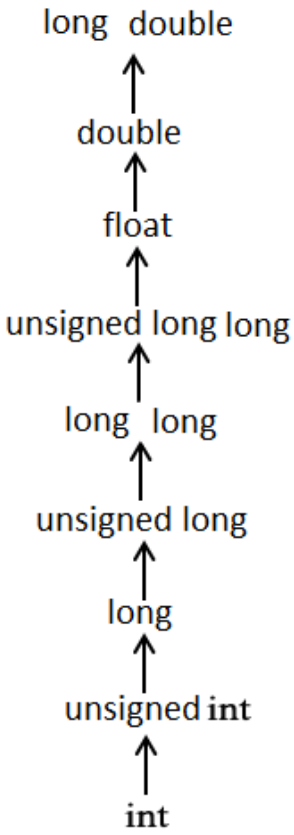
```
Value of sum : 116
```

Aqui, o valor da soma é 116 porque o compilador está fazendo promoção inteira e convertendo o valor de 'c' para ASCII antes de executar a operação de adição real.

Conversão aritmética usual

As **conversões aritméticas usuais** são implicitamente executadas para converter seus

valores em um tipo comum. O compilador primeiro executa a *promoção de inteiros* ; se os operandos ainda tiverem tipos diferentes, eles serão convertidos para o tipo que aparecer mais alto na hierarquia a seguir -



As conversões aritméticas usuais não são realizadas para os operadores de atribuição, nem para os operadores lógicos && e ||. Vamos dar o seguinte exemplo para entender o conceito -

```
#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 );
}
```

Quando o código acima é compilado e executado, ele produz o seguinte resultado -

```
Value of sum : 116.000000
```

Aqui, é simples entender que o primeiro c é convertido em inteiro, mas como o valor final é duplo, a conversão aritmética usual é aplicada e o compilador converte i e c em 'float' e adiciona-os produzindo um resultado 'float'.

C - Tratamento de Erros

Como tal, a programação C não fornece suporte direto para o tratamento de erros, mas sendo uma linguagem de programação do sistema, fornece acesso a um nível inferior na forma de valores de retorno. A maioria das chamadas de função C ou até Unix retorna -1 ou NULL em caso de qualquer erro e define um código de erro **errno** . Ele é definido como uma variável global e indica que ocorreu um erro durante qualquer chamada de função. Você pode encontrar vários códigos de erro definidos no arquivo de cabeçalho <error.h>.

Portanto, um programador C pode verificar os valores retornados e pode executar a ação apropriada, dependendo do valor de retorno. É uma boa prática, definir errno como 0 no momento da inicialização de um programa. Um valor de 0 indica que não há erro no programa.

errno, perror (). e strerror ()

A linguagem de programação C fornece as funções **perror ()** e **strerror ()** que podem ser usadas para exibir a mensagem de texto associada a **errno** .

> A função **perror ()** exibe a string que você passa para ela, seguida por dois-pontos, um espaço e depois a representação textual do valor atual do errno.
>
> A função **strerror ()** , que retorna um ponteiro para a representação textual do valor atual do errno.

Vamos tentar simular uma condição de erro e tentar abrir um arquivo que não existe. Aqui estou usando as duas funções para mostrar o uso, mas você pode usar uma ou mais maneiras de imprimir seus erros. O segundo ponto importante a ser observado é que você deve usar o fluxo de arquivos **stderr** para gerar todos os erros.

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

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;
}
```

Quando o código acima é compilado e executado, ele produz o seguinte resultado -

```
Value of errno: 2
Error printed by perror: No such file or directory
Error opening file: No such file or directory
```

Divida por zero erros

É um problema comum que no momento de dividir qualquer número, os programadores não verificam se um divisor é zero e finalmente cria um erro de tempo de execução.

O código abaixo corrige isso, verificando se o divisor é zero antes de dividir -

```
#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);
}
```

Quando o código acima é compilado e executado, ele produz o seguinte resultado -

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

Status de saída do programa

É uma prática comum sair com um valor EXIT_SUCCESS no caso do programa ser lançado após uma operação bem-sucedida. Aqui, EXIT_SUCCESS é uma macro e é definida como 0.

Se você tiver uma condição de erro em seu programa e sair, deverá sair com um status EXIT_FAILURE, que é definido como -1. Então, vamos escrever acima do programa da seguinte forma -

```
#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);
}
```

Quando o código acima é compilado e executado, ele produz o seguinte resultado -

```
Value of quotient : 4
```

C - Recursão

A recursão é o processo de repetir itens de maneira semelhante. Em linguagens de programação, se um programa permite chamar uma função dentro da mesma função, ela é chamada de chamada recursiva da função.

```
void recursion() {
   recursion(); /* function calls itself */
}

int main() {
   recursion();
}
```

A linguagem de programação C suporta recursão, ou seja, uma função para se chamar. Mas enquanto estiver usando a recursão, os programadores precisam ter cuidado para definir uma condição de saída da função, caso contrário ela entrará em um loop infinito.

Funções recursivas são muito úteis para resolver muitos problemas matemáticos, como calcular o fatorial de um número, gerar séries de Fibonacci, etc.

Número Fatorial

O exemplo a seguir calcula o fatorial de um determinado número usando uma função recursiva -

```
#include <stdio.h>

unsigned long long int factorial(unsigned int i) {

   if(i <= 1) {
      return 1;
   }
   return i * factorial(i - 1);
}

int  main() {
   int i = 12;
   printf("Factorial of %d is %d\n", i, factorial(i));
   return 0;
}
```

Quando o código acima é compilado e executado, ele produz o seguinte resultado -

```
Factorial of 12 is 479001600
```

Série de Fibonacci

O exemplo a seguir gera a série Fibonacci para um determinado número usando uma função recursiva -

```c
#include <stdio.h>

int fibonacci(int i) {

   if(i == 0) {
      return 0;
   }

   if(i == 1) {
      return 1;
   }
   return fibonacci(i-1) + fibonacci(i-2);
}

int  main() {

   int i;

   for (i = 0; i < 10; i++) {
      printf("%d\t\n", fibonacci(i));
   }

   return 0;
}
```

Quando o código acima é compilado e executado, ele produz o seguinte resultado -

```
0
1
1
2
3
5
8
13
21
34
```

C - Argumentos Variáveis

Às vezes, você pode se deparar com uma situação, quando deseja ter uma função, que pode receber um número variável de argumentos, ou seja, parâmetros, em vez de um número predefinido de parâmetros. A linguagem de programação C fornece uma solução para esta situação e você tem permissão para definir uma função que pode aceitar um número variável de parâmetros com base em sua necessidade. O exemplo a seguir mostra a definição de tal função.

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

int main() {
```

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

Deve-se notar que a função **func ()** tem seu último argumento como elipses, ou seja, três pontos (**...**) e o outro logo antes das elipses é sempre um **int** que representará o número total de argumentos variáveis passados. Para usar essa funcionalidade, você precisa usar o arquivo de cabeçalho **stdarg.h,** que fornece as funções e macros para implementar a funcionalidade de argumentos variáveis e seguir as etapas fornecidas -

Defina uma função com seu último parâmetro como elipses e a logo antes das elipses é sempre um **int** que representará o número de argumentos.

Crie uma variável do tipo **va_list** na definição da função. Esse tipo é definido no arquivo de cabeçalho stdarg.h.

Use o parâmetro **int** e a macro **va_start** para inicializar a variável **va_list** em uma lista de argumentos. A macro va_start é definida no arquivo de cabeçalho stdarg.h.

Use a **variável va_arg** macro e **va_list** para acessar cada item na lista de argumentos.

Use uma macro **va_end** para limpar a memória atribuída à variável **va_list** .

Agora vamos seguir os passos acima e escrever uma função simples que pode pegar o número variável de parâmetros e retornar sua média -

```
#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));
}
```

Quando o código acima é compilado e executado, produz o seguinte resultado. Deve-se notar que a função **average ()** foi chamada duas vezes e cada vez que o primeiro

argumento representa o número total de argumentos variáveis sendo passados. Somente elipses serão usadas para passar um número variável de argumentos.

```
Average of 2, 3, 4, 5 = 3.500000
Average of 5, 10, 15 = 10.000000
```

C - Gerenciamento de Memória

Este capítulo explica o gerenciamento de memória dinâmica em C. A linguagem de programação C fornece várias funções para alocação e gerenciamento de memória. Essas funções podem ser encontradas no arquivo de cabeçalho **<stdlib.h>** .

Sr. Não.	Descrição da função
1	**void * calloc (int num, tamanho int);** Essa função aloca uma matriz de elementos **numéricos** , cada qual tamanho em bytes será **tamanho** .
2	**void free (void * address);** Esta função libera um bloco de memória especificado pelo endereço.
3	**void * malloc (int num);** Essa função aloca uma matriz de **num** bytes e os deixa não inicializados.
4	**void * realloc (void * endereço, int newsize);** Esta função **realoca a** memória estendendo-a até **newsize** .

Alocando Memória Dinamicamente

Durante a programação, se você está ciente do tamanho de um array, então é fácil e você pode defini-lo como um array. Por exemplo, para armazenar um nome de qualquer pessoa, ele pode ir até um máximo de 100 caracteres, para que você possa definir algo da seguinte forma -

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

Mas agora vamos considerar uma situação em que você não tem idéia sobre o tamanho do texto que precisa armazenar, por exemplo, você deseja armazenar uma descrição detalhada sobre um tópico. Aqui, precisamos definir um ponteiro para caractere sem definir a quantidade de memória necessária e, posteriormente, com base no requisito, podemos alocar memória, conforme mostrado no exemplo a seguir -

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

```c
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 );
}
```

Quando o código acima é compilado e executado, produz o seguinte resultado.

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

O mesmo programa pode ser escrito usando **calloc ();** única coisa é que você precisa substituir malloc com calloc da seguinte forma -

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

Portanto, você tem controle total e pode passar qualquer valor de tamanho ao alocar memória, ao contrário dos arrays em que, uma vez definido o tamanho, não é possível alterá-lo.

Redimensionando e liberando memória

Quando o programa é lançado, o sistema operacional libera automaticamente toda a memória alocada pelo seu programa, mas como uma boa prática quando você não está mais precisando de memória, você deve liberar essa memória chamando a função **free ()** .

Alternativamente, você pode aumentar ou diminuir o tamanho de um bloco de memória alocado chamando a função **realloc ()** . Vamos verificar o programa acima novamente e fazer uso das funções realloc () e free () -

```c
#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);
}
```

Quando o código acima é compilado e executado, produz o seguinte resultado.

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

Você pode tentar o exemplo acima sem realocar memória extra, e a função strcat () dará um erro devido à falta de memória disponível na descrição.

C - Argumentos da linha de comando

É possível passar alguns valores da linha de comando para seus programas em C quando eles são executados. Esses valores são chamados de **argumentos de linha de comando** e muitas vezes são importantes para o programa, especialmente quando você deseja controlar o programa de fora, em vez de codificar esses valores dentro do código.

Os argumentos da linha de comando são manipulados usando argumentos da função main () onde **argc** se refere ao número de argumentos passados, e **argv []** é uma matriz de ponteiro que aponta para cada argumento passado para o programa. A seguir, um exemplo simples que verifica se existe algum argumento fornecido pela linha de comando e age de acordo.

```
#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");
   }
}
```

Quando o código acima é compilado e executado com um único argumento, ele produz o seguinte resultado.

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

Quando o código acima é compilado e executado com dois argumentos, ele produz o resultado a seguir.

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

Quando o código acima é compilado e executado sem passar nenhum argumento, ele produz o resultado a seguir.

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

Deve-se notar que **argv [0]** contém o nome do próprio programa e **argv [1]** é um ponteiro para o primeiro argumento de linha de comando fornecido, e * argv [n] é o último argumento. Se nenhum argumento for fornecido, argc será um e, se você passar um argumento, **argc** será configurado como 2.

Você passa todos os argumentos da linha de comando separados por um espaço, mas se o próprio argumento tiver um espaço, você poderá passar esses argumentos colocando-os entre aspas duplas "" ou aspas simples "". Vamos reescrever o exemplo acima mais uma vez, onde vamos imprimir o nome do programa e também passamos um argumento de linha de comando colocando entre aspas duplas -

```
#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");
   }
}
```

Quando o código acima é compilado e executado com um único argumento separado por espaço, mas dentro de aspas duplas, ele produz o resultado a seguir.

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


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