INTRODUÇÃO À LINGUAGEM C

Versão 2.0

Gerência de Atendimento ao Cliente

CENTRO DE COMPUTAÇÃO

UNICAMP

ÍNDICE

1. Introdução

A linguagem C foi criada por Dennis Ritchie, em 1972, no centro de Pesquisas da Bell Laboratories. Sua primeira utilização importante foi a reescrita do Sistema Operacional UNIX, que até então era escrito em assembly.

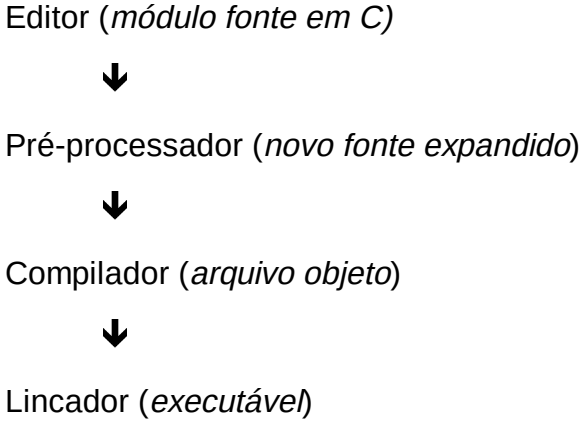
Em meados de 1970 o UNIX saiu do laboratório para ser liberado para as universidades. Foi o suficiente para que o sucesso da linguagem atingisse proporções tais que, por volta de 1980, já existiam várias versões de compiladores C oferecidas por várias empresas, não sendo mais restritas apenas ao ambiente UNIX, porém compatíveis com vários outros sistemas operacionais.

O C é uma linguagem de propósito geral, sendo adequada à programação estruturada. No entanto é mais utilizada escrever compiladores, analisadores léxicos, bancos de dados, editores de texto, etc..

A linguagem C pertence a uma família de linguagens cujas características são: portabilidade, modularidade, compilação separada, recursos de baixo nível, geração de código eficiente, confiabilidade, regularidade, simplicidade e facilidade de uso.

Visão geral de um programa C

A geração do programa executável a partir do programa fonte obedece a uma seqüência de operações antes de tornar-se um executável. Depois de escrever o módulo fonte em um editor de textos, o programador aciona o compilador que no UNIX é chamado pelo comando **cc**. Essa ação desencadeia uma seqüência de etapas, cada qual traduzindo a codificação do usuário para uma forma de linguagem de nível inferior, que termina com o **executável** criado pelo lincador.



2. Sintaxe

A sintaxe são regras detalhadas para cada construção válida na linguagem C. Estas regras estão relacionadas com os **tipos**, as **declarações**, as **funções** e as **expressões**.

Os **tipos** definem as propriedades dos dados manipulados em um programa.

As **declarações** expressam as partes do programa, podendo dar significado a um **identificador**, alocar memória, definir conteúdo inicial, definir funções.

As **funções** especificam as ações que um programa executa quando roda.

A determinação e alteração de valores, e a chamada de funções de I/O são definidas nas **expressões**.

As **funções** são as entidades operacionais básicas dos programas em C, que por sua vez são a união de uma ou mais funções executando cada qual o seu trabalho. Há funções básicas que estão definidas na **biblioteca C**. As funções **printf()** e **scanf()** por exemplo, permitem respectivamente escrever na tela e ler os dados a partir do teclado. O programador também pode definir novas funções em seus programas, como rotinas para cálculos, impressão, etc.

Todo programa C inicia sua execução chamando a função **main(),** sendo obrigatória a sua declaração no programa principal.

Comentários no programa são colocados entre **/*** e ***/** não sendo considerados na compilação.

Cada instrução encerra com **;** (ponto e vírgula) que faz parte do comando.

Ex:

```
main()     /* função obrigatória */
        {
        printf("oi");
    }
```

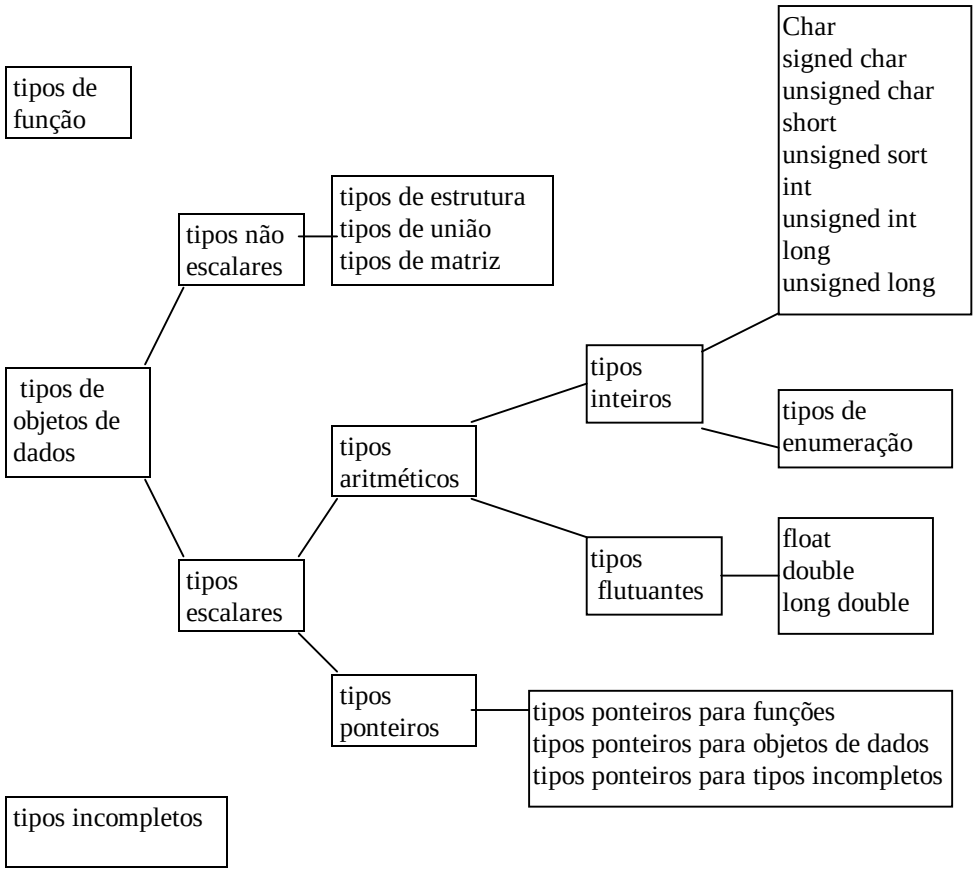
2.1. Identificadores

São nomes usados para se fazer referência a variáveis, funções, rótulos e vários outros objetos definidos pelo usuário. O primeiro caracter deve ser uma letra ou um sublinhado. Os 32 primeiros caracteres de um identificador são significativos. É case sensitive, ou seja, as letras maiúsculas diferem das minúsculas.

```
int x;  /*é  diferente de  int X;*/
```

2.2. Tipos

Quando você declara um **identificador** dá a ele um tipo. Os tipos principais podem ser colocados dentro da classe do *tipo de objeto de dado*. Um tipo de objeto de dados determina como valores de dados são representados, que valores pode expressar, e que tipo de operações você pode executar com estes valores.



2.2.1. Tipos Inteiros

char	[0,128)	igual a signed char ou unsigned char
signed char	(-128,128)	inteiro de pelo menos 8 bits
unsigned char	(0,256)	mesmo que signed char sem negativos
short	$(2^{-15},2^{15}$	inteiro de pelo menos 16 bits tamanho pelo menos igual a char
unsigned sort	$[0,2^{16})$	mesmo tamanho que sort sem negativos
int	$(2^{-15},2^{15})$	inteiro de pelo menos 16 bits; tamanho pelo menos igual a short
unsigned int	$[0,2^{16})$	mesmo tamanho que int sem negativos
long	$(2^{-31},2^{31})$	inteiro com sinal de pelo menos 32 bits; tamanho pelo menos igual a int
unsigned log	$[0,2^{32})$	mesmo tamanho que long sem valores negativos

Uma implementação do compilador pode mostrar um faixa maior do que a mostrada na tabela, mas não uma faixa menor. As potencias de 2 usadas significam:

2^{15} 32.768
2^{16} 65536
2^{31} 2.147.483.648
2^{32} 4.294.967.298

2.2.2. Tipos Flutuantes

float	$[3.4^{-38}, 3.4^{+38}]$	pelo menos 6 dígitos de precisão decimal
double	$(1.7^{-308},1.7^{+308})$	pelo menos 10 dígitos decimais e precisão maior que do float
long double	$(1.7^{-308},1.7^{+308})$	pelo menos 10 dígitos decimais e precisão maior que do double

Ex:

```
main()
        {
        char c;
        unsigned char uc;
        int i;
        unsigned int ui;
        float f;
        double d;
        printf("char          %d",sizeof(c));
        printf("unsigned char %d",sizeof(uc));
        printf("int           %d",sizeof(i));
        printf("unsigned int  %d",sizeof(ui));
        printf("float         %d",sizeof(f));
        printf("double        %d",sizeof(d));
        }
```

2.3. Operadores

2.3.1. Operador de atribuição

O operador de atribuição em C é o sinal de igual "=". Ao contrário de outras linguagens, o operador de atribuição pode ser utilizado em expressões que também envolvem outros operadores.

2.3.2. Aritméticos

Os operadores *, /, + e - funcionam como na maioria das linguagens, o operador % indica o resto de uma divisão inteira.

```
i+=2;        ->    i=i+2;
x*=y+1;      ->    x=x*(y+1);
d-=3;        ->    d=d-3;
```

Ex:

```
main()
        {
        int x,y; x=10; y=3;
        printf("%d\n",x/y);
        printf("%d\n",x%y);
        }
```

2.3.3. Operadores de relação e lógicos

Relação refere-se as relações que os valores podem ter um com o outro e lógico se refere às maneiras como essas relações podem ser conectadas. Verdadeiro é qualquer valor que não seja 0, enquanto que 0 é falso. As expressões que usam operadores de relação e lógicos retornarão 0 para falso e 1 para verdadeiro.

Tanto os operadores de relação como os lógicos tem a precedência menor que os operadores aritméticos. As operações de avaliação produzem um resultado 0 ou 1.

relacionais		lógicos	
>	maior que	&&	and
>=	maior ou igual	||	ou
<	menor	!	not
<=	menor ou igual		
==	igual		
!=	não igual		

Ex:

```
main()
        {
        int i,j;
        printf("digite dois números: ");
        scanf("%d%d",&i,&j);
        printf("%d == %d é %d\n",i,j,i==j);
        printf("%d != %d é %d\n",i,j,i!=j);
        printf("%d <= %d é %d\n",i,j,i<=j);
        printf("%d >= %d é %d\n",i,j,i>=j);
        printf("%d <  %d é %d\n",i,j,i< j);
        printf("%d >  %d é %d\n",i,j,i> j);
        }
```

Ex:
```
main()
        {
        int x=2,y=3,produto;
        if ((produto=x*y)>0) printf("é maior");
        }
```

2.3.4. Incremento e decremento

O C fornece operadores diferentes para incrementar variáveis. O operador soma 1 ao seu operando, e o decremento subtrai 1. O aspecto não usual desta notação é que podem ser usado como operadores pré-fixo(++x) ou pós-fixo(x++).

++x incrementa x antes de utilizar o seu valor.
x++ incrementa x depois de ser utilizado.

Ex:
```
main()
        {
        int x=0;
        printf("x= %d\n",x++);
        printf("x= %d\n",x);
        printf("x= %d\n",++x);
        printf("x= %d\n",x);
        }
```

2.3.5. Precedência

O nível de precedência dos operadores é avaliado da esquerda para a direita. Os parênteses podem ser utilizados para alterar a ordem da avaliação.

```
++ -- mais alta
* / %
+ -   mais baixa
```

2.3.6. Operador cast

Sintaxe:
(tipo)expressão

Podemos forçar uma expressão a ser de um determinado tipo usando o operador cast.

Ex:
```
main()
        {
        int i=1;
        printf("%d/3 é: %f",i,(float) i/3);
        }
```

2.3.7. Operador sizeof

O operador sizeof retorna o tamanho em bytes da variável, ou seja, do tipo que está em seu operando. É utilizado para assegurar a portabilidade do programa.

3. Funções Básicas da Biblioteca C

3.1. Função printf()

Sintaxe:

printf("expressão de controle",argumentos);

É uma função de I/O, que permite escrever no dispositivo padrão (tela).

A expressão de controle pode conter caracteres que serão exibidos na tela e os códigos de formatação que indicam o formato em que os argumentos devem ser impressos. Cada argumento deve ser separado por vírgula.

\n	nova linha	%c	caractere simples
\t	tab	%d	decimal
\b	retrocesso	%e	notação científica
\"	aspas	%f	ponto flutuante
\\	barra	%o	octal
\f	salta formulário	%s	cadeia de caracteres
\0	nulo	%u	decimal sem sinal
		%x	hexadecimal

Ex:

```
main()
        {
        printf("Este é o numero dois: %d",2);
        printf("%s está a  %d milhões de milhas\ndo sol","Vênus",67);
        }
```

Tamanho de campos na impressão:

Ex:

```
main()
        {
        printf("\n%2d",350);
        printf("\n%4d",350);
        printf("\n%6d",350)
        }
```

Para arredondamento:

Ex:
```
main()
        {
        printf("\n%4.2f",3456.78);
        printf("\n%3.2f",3456.78);
        printf("\n%3.1f",3456.78);
        printf("\n%10.3f",3456.78);
        }
```

Para alinhamento:

Ex:
```
main(){
        printf("\n%10.2f %10.2f %10.2f",8.0,15.3,584.13);
        printf("\n%10.2f %10.2f %10.2f",834.0,1500.55,4890.21);
        }
```

Complementando com zeros à esquerda:

Ex:
```
main()
        {
        printf("\n%04d",21);
        printf("\n%06d",21);
        printf("\n%6.4d",21);
        printf("\n%6.0d",21);
        }
```

Imprimindo caracteres:

Ex:
```
main(){
        printf("%d %c %x %o\n",'A','A','A','A');
        printf("%c %c %c %c\n",'A',65,0x41,0101);
        }
```

A tabela ASCII possui 256 códigos de 0 a 255, se imprimirmos em formato caractere um número maior que 255, será impresso o resto da divisão do número por 256; se o número for 3393 será impresso A pois o resto de 3393 por 256 é 65.

3.2. Função scanf()

Também é uma função de I/O implementada em todos compiladores C. Ela é o complemento de printf() e nos permite ler dados formatados da entrada padrão (teclado). Sua sintaxe é similar a printf().

```
scanf("expressão de controle", argumentos);
```

A lista de argumentos deve consistir nos endereços das variáveis. C oferece um operador para tipos básicos chamado operador de endereço e referenciado pelo símbolo "&" que retorna o endereço do operando.

Operador de endereço &:

A memória do computador é dividida em bytes, e são numerados de 0 até o limite da memória. Estas posições são chamadas de endereços. Toda variável ocupa uma certa localização na memória, e seu endereço é o primeiro byte ocupado por ela.

Ex:

```
main()
        {
        int num;
        printf("Digite um número: ");
        scanf("%d",&num);
        printf("\no número é %d",num);
        printf("\no endereço e %u",&num);
        }
```

3.3. Função getchar()

É a função original de entrada de caractere dos sistemas baseados em UNIX. getchar() armazena a entrada até que ENTER seja pressionada.

Ex:

```
main()
        {
        char ch;
        ch=getchar();
        printf("%c\n,ch);
        }
```

3.4. Função putchar()

Escreve na tela o argumento de seu caractere na posição corrente.

Ex:

```
main()
        {
        char ch;
        printf("digite uma letra minúscula : ");
        ch=getchar();
        putchar(toupper(ch));
        putchar('\n');
        }
```

Há inumeras outras funções de manipulação de *char* complementares às que foram vistas, como isalpha(), isupper(), islower(), isdigit(), isespace(), toupper(), tolower().

4. Estruturas de Controle de Fluxo

Os comandos de controle de fluxo são a essência de qualquer linguagem, porque governam o fluxo da execução do programa. São poderosos e ajudam a explicar a popularidade da linguagem. Podemos dividir em três categorias. A primeira consiste em instruções condicionais if e switch. A segunda são os comandos de controle de loop o while, for e o do-while. A terceira contém instruções de desvio incondicional goto.

4.1. If

sintaxe:

```
if (condição)comando;
else comando;
```

Se a condição avaliar em verdadeiro (qualquer coisa menos 0), o computador executará o comando ou o bloco, de outro modo, se a cláusula else existir, o computador executará o comando ou o bloco que é seu objetivo.

Ex:

```
main()
        {
        int a,b;
        printf("digite dois números:");
        scanf("%d%d",&a,&b);
        if (b) printf("%d\n",a/b);
        else printf("divisão por zero\n");
        }
```

Ex:

```
#include <stdlib.h>
#include <time.h>
main(){
int num,segredo;
srand(time(NULL));
segredo=rand()/100;
printf("Qual e o numero: ");
scanf("%d",&num);
if (segredo==num)
        {printf("Acertou!");
        printf("\nO numero e %d\n",segredo);}
else if (segredo<num)
                        printf("Errado, muito alto!\n");
                else printf("Errado, muito baixo!\n");}
```

4.2. If-else-if

Uma variável é testada sucessivamente contra uma lista de variáveis inteiras ou de caracteres. Depois de encontrar uma coincidência, o comando ou o bloco de comandos é executado.

Ex

```
#include <stdlib.h>
#include <time.h>
main()
{
      int num,segredo;
      srand(time(NULL));
      segredo=rand()/100;
      printf("Qual e o numero: ");
      scanf("%d",&num);
      if (segredo==num)
            {printf("Acertou!");
            printf("\nO numero e %d\n",segredo);}
      else if (segredo<num)
            printf("Errado, muito alto!\n");
            else printf("Errado, muito baixo!\n");
}
```

4.3. Operador ternário

Sintaxe:
condição?expressão1:expressão2

É uma maneira compacta de expressar if-else.

Ex:

```
main()
        {
        int x,y,max;
        printf("Entre com dois números: ");
        scanf(%d,%d,&x,&y);
        max=(x>y)?1:0;
        printf("max= %d\n",max);
        }
```

4.4. Switch

sintaxe:

```
    switch(variável)
      {
      case constante1:
          seqüência de comandos
          break;
```

```
    case constante2:

  seqüência de comandos
        break;
    default:
        seqüência de comandos
    }
```

Uma variável é testada sucessivamente contra uma lista de variáveis inteiras ou de caracteres. Depois de encontrar uma coincidência, o comando ou o bloco de comandos é executado.

Se nenhuma coincidência for encontrada o comando default será executado. O default é opcional. A seqüência de comandos é executada até que o comando break seja encontrado.

Ex:

```
main()
    {
    char x;
    printf("1. inclusão\n");
    printf("2. alteração\n");
    printf("3. exclusão\n");
    printf("  Digite sua opção:");
    x=getchar();
    switch(x)
        {
        case '1':
            printf("escolheu inclusão\n");
            break;
        case '2':
            printf("escolheu alteração\n");
            break;
        case '3':
            printf("escolheu exclusão\n");
            break;
        default:
            printf("opção inválida\n");
        }
    }
```

4.5. Loop for

Sintaxe:

```
    for(inicialização;condição;incremento) comando;
```

O comando for é de alguma maneira encontrado em todas linguagens procedurais de programação.

Em sua forma mais simples, a incialização é um comando de atribuição que o compilador usa para estabelecer a variável de controle do loop. A condição é uma expressão de relação que testa a variável de controle do loop contra algum valor para determinar quando o loop terminará. O incremento define a maneira como a variável de controle do loop será alterada cada vez que o computador repetir o loop.

Ex:
```
main()
        {
        int x;for(x=1;x<100;x++)printf("%d\n",x);}
```

Ex:
```
main()
        {
        int x,y;
        for (x=0,y=0;x+y<100;++x,++y)
                printf("%d ",x+y);
        }
```

Um uso interessante para o for é o loop infinito, como nenhuma das três definições são obrigatórias, podemos deixar a condição em aberto.

Ex:
```
main()
        {
        for(;;) printf("loop infinito\n");
        }
```

Outra forma usual do for é o for aninhado, ou seja, um for dentro de outro.

Ex:
```
main()
        {
        int linha,coluna;
        for(linha=1;linha<=24;linha++)
                {
                for(coluna=1;coluna<40;coluna++)
                        printf("-");
                        putchar('\n');
                }
        }
```

4.6. While

Sintaxe:
```
        while(condição) comando;
```

Uma maneira possível de executar um laço é utilizando o comando while. Ele permite que o código fique sendo executado numa mesma parte do programa de acordo com uma determinada condição.

- o comando pode ser vazio, simples ou bloco
- ele é executado desde que a condição seja verdadeira
- testa a condição antes de executar o laço

Ex:

```
main()
        {
        char ch;
        while(ch!='a') ch=getchar();
        }
```

4.7. Do while

Sintaxe:

```
        do
        {
        comando;
        }
        while(condição);
```

Também executa comandos repetitivos.

Ex:

```
main()
        {
        char ch;
        printf("1. inclusão\n");
        printf("2. alteração\n");
        printf("3. exclusão\n");
        printf("  Digite sua opção:");
        do
                {
                ch=getchar();
                switch(ch)
                        {
                        case '1':
                                printf("escolheu inclusao\n");
                                break;
                        case '2':
                                printf("escolheu alteracao\n");
                                break;
                        case '3':
                                printf("escolheu exclusao\n");
                                break;
                        case '4':
                                printf("sair\n");
                        }
                }
        while(ch!='1' && ch!='2' && ch!='3' && ch!='4');
        }
```

4.8. Break

Quando o comando break é encontrado em qualquer lugar do corpo do for, ele causa seu término imediato. O controle do programa passará então imediatamente para o código que segue o loop.

Ex:

```
main()
        {
        char ch;
        for(;;)
                {
                ch=getchar();
                if (ch=='a') break;
                }
        }
```

4.9. Continue

Algumas vezes torna-se necessário "saltar" uma parte do programa, para isso utilizamos o "continue".

- força a próxima iteração do loop
- pula o código que estiver em seguida

Ex:

```
main()
        {
        int x;
        for(x=0;x<100;x++)
                {
                if(x%2)continue;
                printf("%d\n",x);
                }
        }
```

5. Matrizes

A matriz é um tipo de dado usado para representar uma certa quantidade de variáveis que são referenciados pelo mesmo nome. Consiste em locações contíguas de memória. O endereço mais baixo corresponde ao primeiro elemento.

5.1. Matriz unidimensional

Sintaxe:

```
tipo nome[tamanho];
```

As matrizes tem 0 como índice do primeiro elemento, portanto sendo declarada uma matriz de inteiros de 10 elementos, o índice varia de 0 a 9.

Ex:

```
main(){
        int x[10];
        int t;
        for(t=0;t<10;t++)
                {
                x[t]=t*2;
                printf("%d\n",x[t];
                }
        }
```

Ex:

```
main()
        {
        int notas[5],i,soma;
        for(i=0;i<5;i++)
                {
                printf("Digite a nota do aluno %d: ",i);
                scanf("%d",&notas[i]);
                }
        soma=0;
        for(i=0;i<5;i++) soma=soma+notas[i];
        printf("Media das notas: %d.",soma/5);
        }
```

5.2. Matriz Multidimencional

Sintaxe:

```
tipo nome[tamanho][tamanho] ...;
```

Funciona como na matriz de uma dimensão (vetor), mas tem mais de um índice.

Ex:

```
main(){
    int x[10][10];
    int t,p=0;
    for(t=0;t<10;t++,p++)
        {
        x[t][p]=t*p;
        printf("%d\n",x[t][p]);
        }
    }
```

5.3. Matrizes estáticas

Os vetores de dados podem ser inicializados como os dados de tipos simples, mas somente como variáveis globais. Quando for inicializar uma matriz local sua classe deve ser *static*.

Ex:

```
main()
        {
        int i;
        static int x[10]={0,1,2,3,4,5,6,7,8,9};
        for(i=0;i<10;i++) printf("%d\n",x[i]);
        }
```

5.4. Limites das Matrizes

A verificação de limites não é feita pela linguagem, nem mensagem de erros são enviadas, o programa tem que testar os limites das matrizes.

Ex:

```
main()
        {
        int erro[10],i;
        for(i=0;i<100;i++)
                {
                erro[i]=1;
                printf("%d\n",erro[i];
                }
        }
```

6. Manipulação de Strings

Em C não existe um tipo de dado string, no seu lugar é utilizado uma matriz de caracteres. Uma string é uma matriz tipo char que termina com '\0'. Por essa razão uma string deve conter uma posição a mais do que o número de caracteres que sedeseja. Constantes strings são uma lista de caracteres que aparecem entre aspas, não sendo necessário colocar o '\0', que é colocado pelo compilador.

Ex:

```
main()
        {
        static re[]=“lagarto”;
        puts(re);
        puts(&re[0]);
        putchar('\n');
        }
```

6.1. Função gets()

Sintaxe:

```
gets(nome_matriz);
```

É utilizada para leitura de uma string através do dispositivo padrão, até que o ENTER seja pressionado. A função gets() não testa limites na matriz em que é chamada.

Ex:

```
main()
	{
	char str[80];
	gets(str);
	printf("%s",str);
	}
```

6.2. Função puts()

Sintaxe:

```
puts(nome_do_vetor_de_caracteres);
```

Escreve o seu argumento no dispositivo padrão de saída (vídeo), coloca um '\n' no final. Reconhece os códigos de barra invertida.

Ex:

```
main()
	{
	puts("mensagem");
	}
```

6.3. Função strcpy()

Sintaxe:

```
strcpy(destino,origem);
```

Copia o conteúdo de uma string.

Ex:

```
main(){
	char str[80];
	strcpy(str,"alo");
	puts(str);
	}
```

6.4. Função strcat()

Sintaxe:

strcat(string1,string2);

Concatena duas strings. Não verifica tamanho.

Ex:

```
main()
        {
        char um[20],dois[10];
        strcpy(um,"bom");
        strcpy(dois," dia");
        strcat(um,dois);
        printf("%s\n",um);
        }
```

6.5. Função strcmp()

Sintaxe:

strcmp(s1,s2);

Compara duas strings, se forem iguais devolve 0.

Ex:

```
main()
        {
        char s[80];
        printf("Digite a senha:");
        gets(s);
        if (strcmp(s,"laranja"))
                printf("senha inválida\n");
        else
                printf("senha ok!\n") ;
        }
```

Além das funções acima, há outras que podem ser consultadas no manual da linguagem, como strlen() e atoi().

7. Ponteiros

Sintaxe:
tipo *nomevar;

É uma variável que contém o endereço de outra variável. Os ponteiros são utilizados para alocação dinâmica, podendo substituir matrizes com mais eficiência. Também fornecem a maneira pelas quais funções podem modificar os argumentos chamados, como veremos no capítulo de funções.

7.1. Declarando Ponteiros

Se uma variável irá conter um ponteiro, então ela deve ser declarada como tal:

```
int x,*px;

px=&x;  /*a variável px aponta para x */
```

Se quisermos utilizar o conteúdo da variável para qual o ponteiro aponta:

```
y=*px;
```

O que é a mesma coisa que:

```
y=x;
```

7.2. Manipulação de Ponteiros

Desde que os pointers são variáveis, eles podem ser manipulados como as variáveis podem. Se py é um outro ponteiro para um inteiro então podemos fazer a declaração:
py=px;

Ex:

```
main(){
        int x,*px,*py;
        x=9;
        px=&x;
        py=px;
        printf("x= %d\n",x);
        printf("&x= %d\n",&x);
        printf("px= %d\n",px);
        printf("*px= %d\n",*px);
        printf("*px= %d\n",*px);
        }
```

7.3. Expressões com Ponteiros

Os ponteiros podem aparecer em expressões, se px aponta para um inteiro x, então *px pode ser utilizado em qualquer lugar que x seria.

O operador * tem maior precedência que as operações aritméticas, assim a expressão abaixo pega o conteúdo do endereço que px aponta e soma 1 ao seu conteúdo.

```
y=*px+1;
```

No próximo caso somente o ponteiro será incrementado e o conteúdo da próxima posição da memória será atribuído a y:

```
y=*(px+1);
```

Os incrementos e decrementos dos endereços podem ser realizados com os operadores ++ e --, que possuem procedência sobre o * e operações matemáticas e são avaliados da direita para a esquerda:

```
*px++; /* sob uma posição na memória */
*(px--); /* mesma coisa de *px-- */
```

No exemplo abaixo os parênteses são necessários, pois sem eles px seria incrementado em vez do conteúdo que é apontado, porque os operadores * e ++ são avaliados da direita para esquerda.

```
(*px)++ /* equivale a x=x+1; ou *px+=1 */
```

Ex:

```
main()
        {nt x,*px;
        x=1;
        px=&x;
        printf("x= %d\n",x);
        printf("px= %u\n",px);
        printf("*px+1= %d\n",*px+1);
        printf("px= %u\n",px);
        printf("*px= %d\n",*px);
        printf("*px+=1= %d\n",*px+=1);
        printf("px= %u\n",px);
        printf("(*px)++= %d\n",(*px)++);
        printf("px= %u\n",px);
        printf("*(px++)= %d\n",*(px++));
        printf("px= %u\n",px);
        printf("*px++-= %d\n",*px++);
        printf("px= %u\n",px);}
```

7.4. Ponteiros para ponteiros

Um ponteiro para um ponteiro é uma forma de indicação múltipla. Num ponteiro normal, o valor do ponteiro é o valor do endereço da variável que contém o valor desejado. Nesse caso o primeiro ponteiro contém o endereço do segundo, que aponta para a variável que contém o valor desejado.

float **balanço;
balanço é um ponteiro para um ponteiro float.

Ex:

```
main()
        {
        int x,*p,**q;
        x=10;
        p=&x;
        q=&p;
        printf("%d",**q);
        }
```

7.5. Problemas com ponteiros

O erro chamado de ponteiro perdido é um dos mais difíceis de se encontrar, pois a cada vez que a operação com o ponteiro é utilizada, poderá estar sendo lido ou gravado em posições desconhecidas da memória. Isso pode acarretar em sobreposições sobre áreas de dados ou mesmo área do programa na memória.

```
int,*p;
x=10;
*p=x;
```

Estamos atribuindo o valor 10 a uma localização desconhecida de memória. A conseqüência desta atribuição é imprevisível.

8. Ponteiros e Matrizes

Em C existe um grande relacionamento entre ponteiros e matrizes, sendo que eles podem ser tratados da mesma maneira. As versões com ponteiros geralmente são mais rápidas.

8.1. Manipulando Matrizes Através de Ponteiros

Considerando a declaração da matriz **int a[10];**

Sendo **pa** um ponteiro para inteiro então:

```
pa=&a[0];  /*passa o endereço inicial do vetor a para o ponteiro pa */
```

pa=a; /* é a mesma coisa de pa=&a[0]; */

x=*pa; /*(passa o conteúdo de a[0] para x */

Se **pa** aponta para um elemento particular de um vetor **a**, então por definição **pa+1** aponta para o próximo elemento, e em geral **pa-i** aponta para **i** elementos antes de **pa** e **pa+i** para **i** elementos depois.

Se **pa** aponta para **a[0]** então:

*(pa+1) aponta para a[1]

pa+i é o endereço de a[i] e *(pa+i) é o conteúdo.

É possível fazer cópia de caracteres utilizando matrizes e ponteiros:

Ex: (versão matriz)
```
main()
        {
        int i=0;
        char t[10];
        static char s[]="abobora";
        while(t[i]=s[i])i++;
        printf("%s\n",t);
        }
```

Ex: (versão ponteiro)
```
main()
        {
        char *ps,*pt,t[10],s[10];
        strcpy(s,"abobora");
        ps=s;
        pt=&t[0];
        while(*ps)*pt++=*ps++;
        printf("%s",t);
        }
```

8.2. String de Ponteiros

Sendo um ponteiro para caracter **char *texto**;:, podemos atribuir uma constante string para texto, que não é uma cópia de caracteres, somente ponteiros são envolvidos. Neste caso a string é armazenada como parte da função em que aparecem, ou seja, como constante.

Char *texto="composto"; /* funciona como **static char texto[]=“composto”;** */

Ex:

```
main()
        {
        char *al="conjunto";
        char re[]=”simples”;
        puts(al);
        puts(&re[0]); /* ou puts(re); */
        for(;*al;al++) putchar(*al);
        putchar('\n');
        }
```

8.3. Matrizes de Ponteiros

A declaração de matrizes de ponteiros é semelhante a qualquer outro tipo de matrizes:

```
int *x[10];
```

Para atribuir o endereço de uma variável inteira chamada **var** ao terceiro elemento da matriz de ponteiros:

```
x[2]=&var;
```

Verificando o conteúdo de **var**:

```
*x[2]
```

As matrizes de ponteiros são tradicionalmente utilizadas para mensagens de erro, que são constantes :

```
char *erro[]={"arquivo não encontrado\n","erro de leitura\n"};
printf("%s",erro[0]);
printf("%s",erro[1]);
```

Ex:

```
main()
        {
        char *erro[2];
        erro[0]="arquivo nao encontrado\n";
        erro[1]="erro da leitura\n";
        for(;*erro[0];)
        printf("%c",*erro[0]++);
```

9. Funções

É uma unidade autônoma de código do programa é desenhada para cumprir uma tarefa particular. Geralmente os programas em C consistem em várias pequenas funções. A declaração do tipo da função é obrigatória no C do UNIX. Os parâmetros de recepção de valores devem ser separados por vírgulas.

Sintaxe:

```
tipo nome(parâmetros);
{ comandos}
```

9.1. Função sem Retorno

Quando uma função não retorna um valor para a função que a chamou ela é declarada como **void**.

Ex:

```
void inverso();
        main()
        {
        char *vet="abcde";
        inverso(vet);
        }
void inverso(s)
        char *s;
        {
        int t=0;
        for(;*s;s++,t++);
        s--;
        for(;t--;)printf("%c",*s--);
        putchar('\n');
        }
```

9.2. Função com Retorno

O Tipo de retorno da função deve ser declarado.

Ex:

```
int elevado();
main()
        {
        int b,e;
        printf("Digite a base e expoente x,y : ");
        scanf("%d,%d",&b,&e);
        printf("valor=%d\n",elevado(b,e));
        }
int elevado(base,expoente)
        int base,expoente;
```

```
{
int i;
if (expoente<0) return;
i=1;
for(;expoente;expoente--)i=base*i;
return i;
}
```

9.3. Parâmetros Formais

Quando uma função utiliza argumentos, então ela deve declarar as variáveis que aceitaram os valores dos argumentos, sendo essas variáveis os parâmetros formais.

Ex:

```
int pertence(string,caracter) /* pertence(char *string,char caracter) */
        char *string,caracter;
        {
        while (*string) if (*string==caracter) return 1;
        else string++;
        return 0;
        }
```

9.3.1. Chamada por Valor

O valor de um argumento é copiado para o parâmetro formal da função, portanto as alterações no processamento não alteram as variáveis.

Ex:

```
int sqr();
main()
        {
        int t=10;
        printf("%d %d",sqr(t),t);
        }

int sqr(x)
        int x;
        {
        x=x*x;
        return(x)
        }
```

9.3.2. Chamada por Referência

Permite a alteração do valor de uma variável. Para isso é necessário a passagem do endereço do argumento para a função.

Ex:
```
void troca();
main()
        {
        int x=10,y=20;
        troca(&x,&y);
        printf("x=%d y=%d\n",x,y);
        }

void troca(a,b)
        int *a,*b;
        {
        int temp;
        temp=*a;
        *a=*b;
        *b=temp;
        }
```

9.4. Classe de Variáveis

Uma função pode chamar outras funções, mas o código que compreende o corpo de uma função (bloco entre {}) está escondido do resto do programa, ele não pode afetar nem ser afetado por outras partes do programa, a não ser que o código use variáveis globais. Existem três classes básicas de variáveis: locais, estáticas e globais.

9.4.1. Variáveis locais

As variáveis que são declaradas dentro de uma função são chamadas de locais. Na realidade toda variável declarada entre um bloco { } podem ser referenciadas apenas dentro deste bloco. Elas existem apenas durante a execução do bloco de código no qual estão declaradas. O armazenamento de variáveis locais por default é na pilha, assim sendo uma região dinâmica.

Ex:
```
void linha;
main(){
        int tamanho;
        printf("Digite o tamanho: ");
        scanf("%d",&tamanho);
        linha(tamanho);
        }
void linha(x)
        int x;
        {
        int i;
        for(i=0;i<=x;i++)putchar(95);
        /* A variável i na função linha não é reconhecida pela função main.*/
        }
```

9.4.2. Variáveis Globais

São conhecidas por todo programa e podem ser usadas em qualquer parte do código. Permanecem com seu valor durante toda execução do programa. Deve ser declarada fora de qualquer função e até mesmo antes da declaração da função main. Fica numa região fixa da memória própria para esse fim.

Ex:

```
void func1(),func2();
int cont;
main()
        {
        cont=100;
        func1();
        }
void func1()
        {
        int temp;
        temp=cont;
        func2();
        printf("cont é = %d",cont);
        }
void func2()
        {
        int cont;
        for(cont=1;cont<10;cont++) printf(".");
        }
```

9.4.3. Variáveis Estáticas

Funcionam de forma parecida com as variáveis globais, conservando o valor durante a execução de diferentes funções do programa. No entanto só são reconhecidas na função onde estão declaradas. São muitos utilizadas para inicializar vetores.

Ex:

```
main()
        {
        int i;
        static int x[10]={0,1,2,3,4,5,6,7,8,9};
        for(i=0;i<10;i++) printf("%d\n",x[i]);
        }
```

9.5. Funções com Matrizes

9.5.1. Passando Parâmetros Formais

É necessário passar somente o endereço e não uma cópia da matriz.

Ex:
```
void mostra();
main()
        {
        int t[10],i;
        for(i=0;i<10;i++)t[i]=i;
        mostra(t);
        }
void mostra(num)
        int num[];   /* ou declarar    int *num;    */
        {
        int i;
        for(i=0;i<10;i++)printf("%d",num[i]);
        }
```

9.5.2. Alterando o Valores da Matriz

Ex:
```
void maiusc();
main()
        {
        char s[80];
        gets(s);
        maiusc(s);
        }

void maiusc(string)
        char *string;
        {
        register int t;
        for(t=0;string[t];t++)
                {
                string[t]=toupper(string[t]);
                printf("%c",string[t]);
                }
        }
```

10. Argumentos da Linha de Comando

No ambiente C existe uma maneira de passar argumentos através da linha de comandos para um programa quando ele inicia. O primeiro argumento (argc) é a quantidade de argumentos que foram passados quando o programa foi chamado; o segundo argumento (argv) é um ponteiro de vetores de caracteres que contém os argumentos, um para cada string.

Por convenção argv[0] é o nome do programa que foi chamado, portanto argc é pelo menos 1. Cada argumento da linha de comando deve ser separado por um espaço ou tab.

Ex:
```
main(argc,argv)
int argc;
```

```
char *argv[];
        {
        if (argc!=2)
                {
                printf("falta digitar o nome\n");
                exit(0);
                }printf("alo %s",argv[1]);}

Ex:
main(argc,argv)
int argc;
char *argv[];
        {
        int disp,cont;
        if (argc<2)
                {
                printf("falta digitar o valor para contagem\n");
                exit(0);
                }
        if (argc==3&&!strcmp(argv[2],"display"))
                disp=1;
        else disp=0;
        for(cont=atoi(argv[1]);cont;--cont)
                if(disp)printf("%d",cont);
        printf("%c",7);}
```

11. Estruturas

Ao manusearmos dados muitas vezes deparamos com informações que não são fáceis de armazenar em variáveis escalares como são os tipos inteiros e pontos flutuantes, mas na verdade são conjuntos de coisas. Este tipo de dados são compostos com vários dos tipos básicos do C. As estruturas permitem uma organização dos dados dividida em campos e registros.

```
Ex:
struct lapis {
       int dureza;
       char fabricante;
       int numero;
       };

main()
{
int i;
struct lapis p[3];
p[0].dureza=2;
p[0].fabricante='F';
p[0].numero=482;
p[1].dureza=0;
p[1].fabricante='G';
p[1].numero=33;
p[2].dureza=3;
```

```
p[2].fabricante='E';
p[2].numero=107;

printf("Dureza  Fabricante     Numero\n");
for(i=0;i<3;i+ +)
printf("%d\t%c\t\t%d\n",p[i].dureza,p[i].fabricante,p[i].numero);
}
```

Como ocorre com as variáveis, as estruturas também podem ser referenciadas por ponteiros. Assim, definindo-se por exemplo o ponteiro ***p** para a estrutura acima (lapis), pode-se usar a sintaxe **(*p).dureza**. Porém, para referenciar o ponteiro há ainda outra sintaxe, através do operador **->** , como por exemplo, **p->dureza**.

12. Noções de Alocação Dinâmica

Há duas maneiras de armazenar variáveis na memória do computador. Primeiro por variáveis globais e *static* locais, segundo através de alocação dinâmica, quando o C armazena a informação em uma área de memória livre, de acordo com a necessidade. No caso do C standart, a alocação dinâmica fica disponível com a inclusão de **stdio.h**.

Ex.

```
#include <stdio>
main()
{
int *p, t;
p=(int *) malloc(40*sizeof(int));
if (!p)
        printf("memoria insuficiente\n");
else
        {
        for(t=0;t<40;++t) *(p+t)=t;
        for(t=0;t<40;++t) printf("%d ",*(p+t));
        free(p);
        }
}
```

Ex:

```
#include <stdio>
main()
{
int i,quant;
float max,min,*p;
printf ("quantidade de numeros:\n");
scanf("%d",&quant);
if (!(p=(float*)malloc((quant+1)*sizeof(float))))
  {
  printf("sem memoria\n");
  exit(1);
  }
printf("digite %d numeros:\n",quant);
```

```
for (i=1;i<=quant;i++)
  {
   scanf("%f",(p+i));
  }
max=*(p+1);
for (i=2;i<=quant;i++)
{
 if (*(p+i)>=max)
  max=*(p+i);
 }
printf("maior e :%f\n",max);
free(p);
}
```

13. Noções de Maninipulação de Arquivos

Para tratar de arquivos a linguagem C fornece um nível de abstração entre o programador e o dispositivo que estiver sendo usado. Esta abstração é chamada *fila de bytes* e o dispositivo normalmente é o arquivo. Existe um sistema bufferizado de acesso ao arquivo, onde um *ponteiro de arquivo* define vários aspectos do arquivo, como nome, status e posição corrente, além de ter a fila associada a ele.

Ex:

```
#include <stdio.h>
        main ()
                {
                FILE *fp; char ch;int nu,*pn;
                pn=&nu;
                fp=fopen("teste.dat","w");
                printf("Entre com os numeros para gravar e 0 para sair: ");
                scanf("%d",&nu);
                while(nu)
                        {
                        fprintf(fp,"%d ",nu);
                        scanf("%d",&nu);
                        }
                fclose(fp);
                fp=fopen("teste.dat","r");
                while(!feof(fp))
                        {
                        fscanf(fp,"%d ",&nu);
                        printf("%d",nu);
                       }  }
```

14. Bibliografia

- KERNIGHAN, Brian W. e RITCH, Dennis M., C: A Linguagem de Programação, Rio de Janeiro, Campus, 1986.
- PLAUGER, P.J. e BRODIE J. Standart C : guia de referência básica, São Paulo, Mcgraw-Hill, 1991. 306p.
- HANCOCK, Les e KRIEGER, Morris. Manual de Linguagem C, Rio de Janeiro, Campus,1985. 182p.
- MIZRAHI, Viviane V. Treinamento em Linguagem C - módulo 1 e 2, São Paulo, McGraw-Hill, 1990, 241p.
- SCHILDT, Herbert. Turbo C: Guia do Usuário, São Paulo, McGraw-Hill, 1988, 414p.