Universidade Federal de Santa Maria
Prof. Cesar Tadeu Pozzer
Disciplina: Lógica e Algoritmo - ELC1064
24/06/2023

A Linguagem C

A linguagem C nasceu na década de 70, por Dennis Ritchie, e foi derivada da linguagem B, criada por Ken Thompson. O B, por sua vez, originou-se da linguagem BCPL, inventada por Martin Richards. Uma boa referência da linguagem C por ser vista em [1].

A linguagem C é considerada de baixo (ou médio) nível, em oposicao a ***Java e C#*** que são linguagens de alto nível. A linguagem Assembly é a de mais baixo nível existente. Linguagens de mais baixo nível permitem uma comunicação mais direta/eficaz/otimizada com o computador, e por isso são mais difíceis de aprender e depurar. Quanto mais baixo o nível da linguagem de programação, mais linhas de código são necessárias e maior a probabilidade de existirem erros de programação. As linguagens Java e C# têm estrutura de comandos e sintaxe muito parecidas com a linguagem C, porém oferecem um nível de abstração maior. A linguagem C permite mesclar comandos Assembly no mesmo código fonte.

Introdução a Programação em Linguagem C

Antes de um algoritmo ser executado no computador, ele deve ser traduzido em uma linguagem que o computador possa compreender. Para isso existem os compiladores, que varrem o "algoritmo", verificam a existência de erros e o convertem na notação da máquina sendo usada. O código executável de um programa para PC não é igual ao código do mesmo programa para Mac ou Linux, por exemplo.
Outra forma de executar programas é com o uso de interpretadores, como no caso da linguagem Lua, Javascript, etc. Interpretadores são frequentemente utilizados para fazer processamento de linguagens de script, como no caso engines de jogos e aplicativos gráficos, como o 3D Studio, dentre outros.
Para transformar um algoritmo em linguagem C, várias adaptações devem ser feitas. C, como qualquer outra linguagem, tem um nível de formalismo muito maior que um algoritmo. Em C, as variáveis devem ter um tipo específico. Por exemplo:

Tipo	Descrição do tipo
`int`	Usado para armazenar números inteiros. Ex: 30, -10, 100188
`float`	Usado para armazenar números reais. Ex: 12.1, 0.003, 1000.23456
`char`	Usado para armazenar caracteres ou números inteiros pequenos. Ex: 'a', 'M', '+', 120, -70

Em vez de usar Inicio e Fim, deve-se usar os caracteres '{' '}' para definição de blocos de dados ou funções. A linguagem C define um grande número de bibliotecas de funções. Algumas delas são:

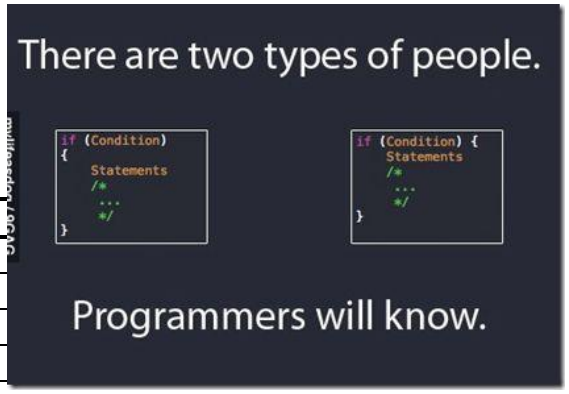
Nome da biblioteca	O que contém
`math.h`	Funções matemáticas, como seno, cosseno, etc.
`stdio.h`	Funções de leitura e impressão de dados
`string.h`	Funções de manipulação de texto

A existência destas funções reduz o tempo de criação de uma aplicação, e tornam o código mais compacto e legível. Cabe ressaltar que não existem funções complexas como, por exemplo, tomar água ou trocar uma lâmpada.

Estrutura de um programa C

Todo programa escrito em C é composto por funções. Uma função é um bloco de programa onde são descritas as operações que devem ser realizadas. Existe uma única função que deve estar presente em todos os programas, que é a função main( ), e é o ponto de partida de execução do programa. A sintaxe de uma função é a seguinte:

```
tipo_retorno nome_função( parametros )
{
   //comandos
}
```



Exemplo 1 – Programa que somente imprime uma mensagem na tela

Linha	Comando C
1	`void main( void )`
2	`{`
3	`   printf("Ola mundo");`
4	`}`

O seguinte exemplo mostra a sintaxe de um programa simples, que imprime na tela o valor de uma expressão aritmética.

Linha	Comando C
1	`#include <stdio.h>`
2	`int main( void )`
3	`{`
4	`   int dado1, dado2, resp;`
5	`   dado1 = 20;`
6	`   scanf("%d", &dado2 );`
7	`   /*Isto é um comentário*/`
8	`   resp = dado1*dado2;`
9	`   printf("O valor da expressão é %d", resp);`
10	`   return 0;`
11	`}`

Na linha 1, a diretiva de compilação `#include` faz o compilador adicionar o arquivo stdio.h ao código do programa. Este arquivo adicionado contém protótipos de várias funções de entrada e saída. A inclusão é necessária, pois na linha 6 é chamada a função `scanf()`, que faz a leitura do teclado de um número inteiro e atribui a variável `dado2`; e na linha 9 é chamada a função `printf()`, que faz a impressão do valor da expressão, calculada na linha 8, na tela do computador.

Na linha 2 está declarada a função `main()`. Para este exemplo, a função deve retornar um valor inteiro (int) e não recebe nenhum parâmetro (`void`). O corpo da função main é delimitado pelo bloco "{ }" , que tem início na linha 4 e termina na linha 10. Qualquer código fora deste bloco não irá pertencer a função `main()`.

O uso de variáveis é necessário para fazer a manipulação dos dados do programa. Na linha 4 são declaradas 3 variáveis do tipo inteiro (variáveis que somente podem armazenar valores inteiros). A variável chamada `dado1` é inicializada na linha seguinte com o valor 20. A variável `dado2` é lida do teclado pela chamada da função `scanf()`. O programa somente segue a execução para o próximo comando quando o usuário digitar algum valor e pressionar a tecla <Enter>. Na linha 7 é adicionado um comentário ao programa, que será ignorado pelo compilador na geração do código executável. Comentários são de grande importância na documentação do código fonte.

Na linha 8 é avaliada uma expressão de multiplicação, que consiste em armazenar na variável `resp` o valor da multiplicação da variável `dado1` com `dado2`. O valor da expressão é mostrado na tela pela chamada da função `printf()`.
Ao final, o programa retorna o valor 0 ao sistema operacional, indicando que a finalização do programa ocorreu de forma correta.

Um programa pode ter qualquer número de funções. O uso de funções é uma boa forma de estruturação do programa e facilita a reutilização de código. Variáveis declaradas dentro de uma função somente são visíveis somente dentro da própria função.

Variáveis podem ter qualquer nome, desde que iniciem com letras, como por exemplo: `a`, `dado`, `v_13df`, `teste`, `tmp`, etc. Não são permitidos caracteres especiais como -, +, *, etc, bem como acentuação. Aconselha-se dar nomes significativos as variáveis, pois tornam o código fonte mais legível, como por exemplo: `media`, `temp`, `opcao`, `resp`, etc.

Observa-se que comandos e declarações de variáveis são terminadas com ponto-e-vírgula. Caso algum “;” for esquecido (erro de sintaxe), será gerada uma mensagem de erro pelo compilador indicando a linha onde o erro ocorreu.

Outra observação é que o C é "Case Sensitive", isto é, *maiúsculas e minúsculas fazem diferença*. Se for declarada uma variável com o nome soma ela será diferente de `Soma`, `SOMA`, `SoMa` ou `sOmA`. Da mesma maneira, os comandos do C “`if`” e “`for`”, por exemplo, só podem ser escritos em minúsculas pois senão o compilador não irá interpretá-los como sendo comandos, mas sim como variáveis.

Este mesmo programa poderia ser escrito de forma mais estruturada, colocando a operação de multiplicação em uma função específica.

```
1   #include <stdio.h>
2
3   int produto(int n1, int n2);
4
5   int produto(int n1, int n2)
6   {
7      int resp;
8      resp = n1 * n2;
9      return resp;
10  }
11
12  int main( void )
13  {
14     int dado1, dado2, resp;
15     dado1 = 20;
16     scanf("%d", &dado2 );
17     /*chamada da funcao produto passando dado1 e dado2*/
18     resp = produto(dado1, dado2);
19     printf("O valor da expressão é %d", resp);
20     return 0;
21  }
```

Neste exemplo, foi criada a função `multiplica()`, que recebe dois parâmetros inteiros e retorna outro inteiro, representando o produto de `n1` com `n2`. Esta função é chamada de dentro da função `main()`, na linha 18. Quando a função é chamada, a execução do programa sai da função `main()` e vai para a função `produto()` e a variável `dado1` é copiada na variável `n1` e a variável `dado2` é copiada na variável `n2`. Quando a função produto retorna (linha 9), o valor de retorno é atribuído a variável `resp` na linha 18. Cabe lembrar que as variáveis são locais à função onde foram declaradas, logo a variável `resp` da função `produto()` não é a mesma da função `main()`. Quando a função `produto()` retorna, a execução continua na função `main()`.
Na linha 3 é declarado o protótipo da função `produto()`, que é usada para indicar ao compilador da existência da função produto. A declaração do protótipo é obrigatória caso a função seja chamada antes de sua declaração, o que não ocorre neste exemplo.

Identação de código

Criar um código identado é fundamental para a compreensão. Utilize 3 espaços e nunca utilize tabulação, para evitar que os códigos apresentem formatação diferente ao serem abertos em outros editores.

Errado	Correto

`int main()`
`{`
`int a,b,c;`
`for(a=0;a<10;a++){`
`b = a*a; if(a>b)`
`a=7;`
`else b=6;}`
`return 0;`
`}`
	`int main()`
	`{`
	`  int a, b;`
	`  for(a=0; a<10; a++)`
	`  {`
	`  b = a*a;`
	`  if( a > b )`
	`  a = 7;`
	`  else`
	`  b = 6;`
	`  }`
	`  return 0;`
	`}`

Tratamento de Warnings

Deve-se tomar cuidado com warnings. Em diversos casos pode causar erros de execução. Eis alguns exemplos.

Código	**Mensagem**

`int a;`
`int *p;`
`p = a;`
	"main.c:10: warning: assignment makes pointer from integer without a cast"

`int a = 10;`
`double d = 11.5;`
`a = d;`
	"main.c:10: warning: conversion from ‘double’ to ‘int’. Possible loss of data"

`int a = 10;`
`int v[10];`
`a = v;`
	"main.c:10: warning: assignment makes integer from pointer without a cast"

`int a;`
`if(a>10)`
`...`
	"main.c:10: warning: local variable 'a' used without having been initialized"

No compilador gcc/g++ existem vários argumentos de compilação para exibição de warnings (Geralmente todos iniciam com W). Eis alguns:

Argumento	Função
`-Wall`	Enable all preprocessor warnings
`-W`	Enable extra warnings
`-Wconversion`	Warn about possibly confusing type conversions
`-Wshadow`	Warn when one local variable shadows another
`-Wunused`	Warn when a variable is unused
`--fatal-warnings`	treat warnings as errors

Consulte http://www.network-theory.co.uk/docs/gccintro/gccintro_31.html para exemplos em C destes erros. Para uma lista completa de argumentos de compilação do gcc/g++, utilize gcc –v –help

Operadores

Existem vários tipos de operadores em C, que podem ser classificados, segundo sua função, nos seguintes grupos. Ao final do conteúdo são apresentados os operadores **bitwise**:

Tipo do Operador	Exemplo	Descrição
Aritméticos		
`+, -, *, /,`	`f = a+b;`	Operações aritméticas convencionais
`% (módulo)`	`c = a%b;`	c recebe o resto da divisão inteira de a por b
Atribuição		
`=`	`a = 10;`	Inicializa a com 10
Incremento		
`++`	`cont++;`	Incrementa cont em 1
`--`	`cont--;`	Decrementa cont em 1
`+=`	`cont+=5`	Incrementa cont em 5
`-=`	`cont-=3`	Decrementa cont em 3
Relacionais		
`==`	`a==b`	Testa se a é igual a b
`< , <=`	`a<=b`	Testa se a é menor ou igual a b
`>, >=`	`a>b`	Testa se b é menor que a
`!=`	`a!=b`	Testa se a é diferente de b
Lógicos		
`&&`	`a==1 && b==1`	E (and) lógico
`||`	`a==1 || b==1`	Ou (or) lógico
`!`	`a=!a`	Negação (not)

Deve-se ter muito cuidado ao utilizar o operador ++ antes da variável, como no seguinte exemplo. Qual será o valor de b impresso na tela?

```
int i = 2;
int b = i + (++i);
printf("%d", b);
```

Entrada e Saída padrões

Os dois comandos mais básicos para entrada/saída são o **scanf** e o **printf**. O comando `scanf()` é usado para capturar valores via teclado e o `printf()` para exibir valores na tela.

Ambos comandos podem receber formatadores, que vao dizer como a informação deve ser lida/escrita. Os mais comuns são: %d para valores inteiros, %c para caracteres, %s para strings (sequencia de caracteres), e %f para números reais, como nos seguintes exemplos:

```
printf("Ola mundo");   // imprime uma mensagem
printf("%d %f", a, b); // imprime a como sendo inteiro e b como float
printf("%c %s", c, s); // imprime c como char e s como string
printf("A variavel cont vale %d", cont); // mensagem e valor
printf("\n %d \n", cont);  // imprime uma linha em branco antes e após cont
printf("\t %d", cont);  //usa espaço de tabulação
printf("%.3f", media);  // imprime o valor media com 3 casas decimais
printf("%5d", cont);    // imprime cont com no minimo 5 caracteres
printf("%8.3f", media); // imprime media com no minimo 8 caracteres, sendo 3 de casas decimais

scanf("%d", &a);    // lê o valor da variável inteira a
scanf("%f", &nota); // lê o valor da variável nota como sendo um número real
scanf("%s", nome);  // lê uma sequencia de caracteres (texto)
scanf("%[^\n]", str);  // lê uma sequencia de caracteres ateh encontrar o \n
```

Controle de fluxo

Para efetuar tomada de decisões, desvios condicionais e controle de laços de repetição, C fornece um amplo conjunto de construções para as mais diversas necessidades. O comando principal, que é a base para os outros é o `if` (se, em português). A sintaxe é a seguinte.

```
if (expressão for verdadeira)
{
   /* bloco de comandos */
}
```

ou

```
if (expressão for verdadeira)
{
   /* bloco de comandos */
}
else /*senao */
{
   /* bloco de comandos se expressão for falsa */
}
```

```
if( 3 == 2 && 3 == 2 || 3 == 3  )
{
    printf("verdade 1");
}
if( 3 == 2 && 3 == 3 || 3 == 2 )
{
    printf("verdade 2");
}
if( 3 == 3 || 3 == 2 && 3 == 2 )
{
    printf("verdade 3");
}
if( 5 == 3 || 3 == 3 && 3 == 2 )
{
    printf("verdade 4");
}
```

Em C, uma expressão é verdadeira se tiver valor diferente de 0 (zero) e falsa se for igual a 0.

```
int a;
scanf("%d", &a);
if(a>5)
   printf(" A eh maior que 5");
else if(a<=5)
   printf(" A eh menor ou igual a 5");
```

Além de operadores relacionais, pode-se também usar operadores lógicos

```
if( (a < 4 && b <=10) || c!=6 )
{
   d -= 30;
}
```

Neste exemplo deve-se cuidar a precedência dos parênteses. A expressão é verdadeira em duas situações, devido ao OU lógico:
1) se (a menor 4 **E** b menor igual 10)
2) se (c diferente 6)

Quando o encadeamento de if-else se tornar muito grande, pode-se usar a estrutura switch, que torna o código mais estruturado e legível. A sintaxe é a seguinte:

```
switch ( expressao )
{
   case opcao1:
      /* comandos executados se expressao == opcao1 */
   break;
   case opcao2:
      /* comandos executados se expressao == opcao2 */
   break;
   default:
      /* se expressao for diferente de todas as opções anteriores */
   break;
}
```

O valor da expressão deve ser um número inteiro ou uma constante caractere. Caso expressão for igual a `opcao1`, serão executados todos os comandos até o comando `break`. A opção `default` é opcional.

Para tratar laços de repetição, pode-se usar o `for(;;)`, `while()` ou `do-while()`. Os comandos `for` e `while` tem funcionalidades semelhantes, porém o `for` é mais compacto, e deve ser usado quando se sabe de antemão quantas interações vão ser necessárias:

`for(inic; condição; incremento )`
`{`
`}`
	`while (condicao)`
	`{`
	`}`
		`do`
		`{`
		`}while(condicao);`

`for(int i=0; i<=20; i++)`
`{`
`  printf(" %d", i);`
`}`
	`int i=0;`
	`while( i <= 20 )`
	`{`
	`  printf(" %d", i);`
	`  i++;`
	`}`
		`int i=0;`
		`do`
		`{`
		`  printf(" %d", i);`
		`  i++;`
		`} while( i <= 20 );`

Nos dois casos, inicialmente é testada a condição. Se ela for verdadeira, o laço executa e a variável `i` é incrementada, de forma iterativa. Deve-se usar preferencialmente o for quando se sabe com certeza quantas iterações vão ser executadas e o while quando não se tem esta informação. Deve-se ter cuidado para NÃO colocar “;” após o `for` ou `while`, pois o bloco do laço não é mais executado (isso não causa erro de compilação).
O comando `do-while()` é muito semelhante ao `while`, com a diferença que o laço primeiramente é executado para posterior teste condicional:

Tipo de Dados

1. Tipos básicos

A linguagem C oferece vários tipos básicos para definição de valores do tipo caractere, inteiro e de ponto flutuante. Deve-se usar o tipo mais adequado ao dado que se deseja representar. A tabela a seguir ilustra os tipos, com respectivos tamanhos e representatividades.

Tipo	Espaço ocupado	Representatividade
`char`	1 byte	-128 a 127
`unsigned char`	1 byte	0 a 255
`short int`	2 bytes	-32.768 a 32.767
`unsigned short int`	2 bytes	0 a 65.535
`int`	4 bytes	-2.147.483.648 a 2.147.483.647
`long long`	8 bytes	-9.223.372.036.854.775.808 a 9.223.372.036.854.775.807
`unsigned int`	4 bytes	0 a 4.294.967.295
`float`	4 bytes	$\pm 10^{-38}$ a 10^{38}
`double`	8 bytes	$\pm 10^{-308}$ a 10^{308}

A declaração de variáveis segue o mesmo modelo dos exemplos anteriores. A inicialização pode ocorrer juntamente com a declaração da variável. Variável não inicializada leva a erros na execução do programa.

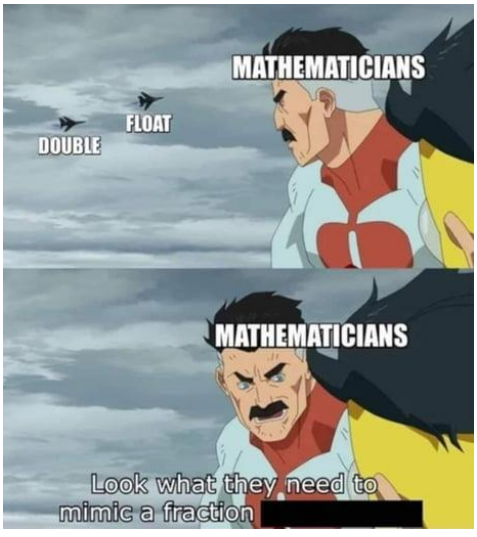
```
float a, b, c = 10.4;
char nome = 20, letra='m';
unsigned char byte = 200;
unsigned int v = 1000;
double pi = 3.1415;
```

Na linguagem C#, o tipo char tem representação semelhante ao unsigned short int da linguagem C. A linguagem C# também oferece o tipo bool (para valores booleanos).

Deve-se ter os seguintes cuidados no uso de variáveis:

- Deve-se garantir que não será atribuído valor superior ao da representatividade de cada tipo, o que pode levar ao truncamento e alteração da informação armazenada;
- A conversão de números inteiros para flutuantes ocorre de forma automática e sem perda, porém o contrário não;
- Variáveis não inicializadas contém lixo.

```
float f = 4.33;
int a, b, c;
a = 2;
c = a + b;  //b não foi inicializada e contém lixo
a = f;      //erro de conversão
f = a;
a = (int)f; //conversão válida com o uso de casting
```

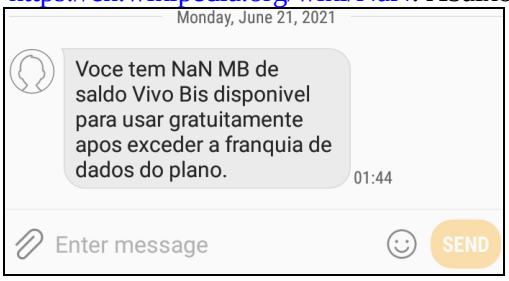


- Tomar cuidado com variáveis de ponto flutuante, pois elas têm precisão finita. Para mais detalhes, acesse: https://en.wikipedia.org/wiki/Floating-point_arithmetic, https://en.wikipedia.org/wiki/IEEE_floating_point. No seguinte exemplo, o número 0.1 não tem representação exata em ponto flutuante. Como a variável f é float e 0.1, por definição, é double (exceto se fosse declarado como 0.1f), o valor 0.1 é representado como double, e consequentemente vai ter mais casas decimais. Logo, o teste condicional é falso.

```c
int main(void)
{
   float f = 0.1; //0.1 não tem representação precisa. Eh
aproximado para 0.0999999999999

   if (f == 0.1)
   {
      printf("Eh igual");
   }
   return 1;
}
```

```c
void main(void)
{
   int   i = 16777217;
   float f = 16777216.0f;

   if( f == i )
   {
       printf(" Iguais ");
   }
}
```

- Deve-se também tomar cuidado ao atribuir valores inteiros a um float. Apesar do float suportar o número 10^{38} ele não tem a mesma precisão em todo intervalo. O número 16.777.216 é o maior número inteiro consecutivo que o float consegue representar, ou seja, 2^{24}. À medida que o número aumenta, a precisão diminui. Como exercício, faça um programa que determina a precisão do float entre 0 e 1038 (precisão inteira e precisão decimal).
- Operações de divisão por zero levam o float a um valor indeterminado, que é representado pelo símbolo NaN (Not a Number) https://en.wikipedia.org/wiki/NaN. Abaixo, exemplo de um uso incorreto de NaN ☺.



De forma complementar a definição de variáveis, pode-se também definir strings que serão substituídas pelos seus identificadores toda vez que encontrados no código fonte (Macro). Isso é feito pela diretiva `#define`. No seguinte exemplo, o identificador é o `MAX_JOGADORES`, e a string correspondente é 5.

```c
#define MAX_JOGADORES 5
#define PI            3.1415
...
/* antes de ser compilada, esta linha será transformada em if (cont<5)*/
if ( cont < MAX_JOGADORES )
   /* comandos */
```

Pode-se criar macros mais elaboradas, como nos seguintes exemplos:

```c
#define MULT(A,B) (A)*(B) //CORRETO
#define MULT(A,B) (A*B)   //ERRADO
...
a = MULT(2+2, 5)   = (2+2)*(5) = 20  //CORRETO. Utiliza MULT(A,B) (A)*(B)
a = MULT(2+2, 5)   = (2+2*5)   = 12  //ERRADO.  Utiliza MULT(A,B) (A*B)
...
#define FOR(a, b, c) for (int(a) = (b); (a) < (c); ++(a))
```

Para maiores detalhes sobre macros, consulte http://gcc.gnu.org/onlinedocs/cpp/Macros.html.

Para uma lista de macros predefinidas, consulte
http://gcc.gnu.org/onlinedocs/cpp/Predefined-Macros.html#Predefined-Macros

2. Vetores e Matrizes

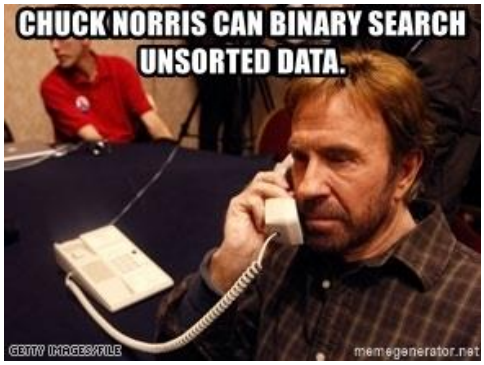
Uma forma simples de implementar tipos de dados mais complexos é por meio de vetores. Eles permitem que conjuntos de dados do mesmo tipo sejam definidos por uma única variável, como no seguinte exemplo.

```
float vet[5] = {1.2, 0.09, 34, 1.01, 50000};
int   tabela[2];
tabela[0] = 1000;
tabela[1] = 88;
```

Neste exemplo, `vet` é um vetor com 5 posições para números do tipo `float`. A inicialização de vetores também pode ocorrer juntamente com a declaração. Em C, a primeira posição do vetor é a de índice 0 e a última é a de índice N-1, como mostrado na inicialização do vetor de inteiros tabela. A atribuição `tabela[2] = 90` causa invasão de memória pois a posição de índice 2 não está definida. Deve-se observar que o tamanho dos vetores é definido em tempo de compilação, ou seja, o compilador irá reservar espaço suficiente na memória para comportar os vetores `vet` e `tabela`.

```
for(int i=0; i<5; i++)
   printf("%f", vet[i]);
```

O seguinte programa pode ser usado para achar o maior valor do vetor, e sua posição.




```
int indice = 0;
float max;
max = vet[0];
for(int i=1; i<5; i++)
{
   if( vet[i] > max )
   {
      max = vet[i];
      indice = i;
   }
}
printf("O maior valor é %f, na posicao %d", max, indice);
```

A linguagem C assume que o programador sabe o que está fazendo, por isso não verifica se a indexação do vetor está dentro dos limites do vetor. Em C# ou Java, uma indexação fora do vetor gera um erro de execução (array index out of bounds exception) no momento, e informa em que local o erro ocorreu. Em C, uma indexação fora do vetor pode funcionar em alguns momentos e em outros não (pior tipo de erro para o programador), e quando ocorre geralmente faz o programa travar, sem dizer que erro ocorreu e onde. Essa não verificação da linguagem C a torna mais rápida que outras linguagens que fazem verificação.

Pode-se também criar vetores multidimensionais, chamados matrizes. A sintaxe e forma de utilização são muito semelhantes.

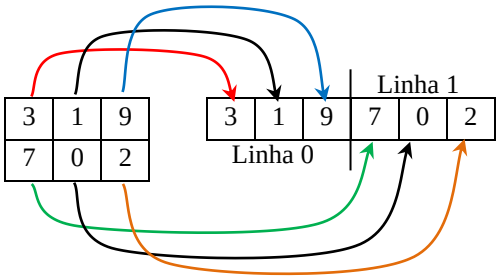
```
int mat[10][30]; //define uma matriz com 300 elementos. 10 linhas e 30 colunas
char tabuleiro[8][8];
float cubo[5][5][5]; //matriz tridimensional
int mat2[10][4][50][600][3]; //matriz com 5 dimensões
```

O seguinte programa ilustra a inicialização da matriz `mat` com valores randômicos entre 0 e 99. Observe o uso do operador %.

```
#include <math.h>
for(int i=0; i<10; i++)
   for(int j=0; j<30; j++)
      mat[i][j] = rand()%100;
```

Vetores e matrizes são passados por ***referência*** em funções (passa-se apenas o endereço inicial do vetor). Isso significa que alterações dentro da função refletem no dado original. Isso será estudado no capítulo de ponteiros.

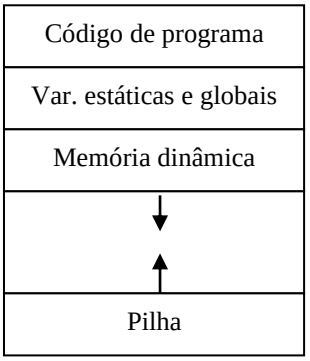
Existem duas formas para se passar uma matriz estática para uma função, como mostrado no seguinte exemplo. Na função f1() passa-se o endereço inicial da matriz. Deve-se observar que a matriz é alocada em memória como um vetor, logo a necessidade de fazer o cálculo de indexação `lin * num_col + col`. Na função f2(), é obrigatório informar o número de colunas que a matriz possui, pela mesma necessidade do cálculo de indexação da função f1(), só que nesse caso este cálculo é feito automaticamente. Observe a necessidade do uso de um *casting* de matriz para ponteiro na chamada da função f1(). Neste exemplo utiliza-se defines para definir as dimensões da matriz.




```c
#include<stdio.h>
#define COL 3
#define LIN 2

void f1(int *p)
{
    for(int l=0; l<LIN; l++)
    {
        for(int c=0; c<COL; c++)
        {
            int pos = l*COL + c;
            printf(" %d", *(p+pos) );
        }
    }
}
void f2(int m[][3]) //ou f2(int m[LIN][COL]) ou f2(int m[2][3])
{
    for(int l=0; l<LIN; l++)
    {
        for(int c=0; c<COL; c++)
        {
            printf(" %d", m[l][c] );
        }
    }
}

void f3(int *v) //ou f3(int v[]) ou f3(int v[3])
{
    for(int c=0; c<COL; c++)
    {
        printf("\n %d", v[c] );
    }
}

int main()
{
    int m[LIN][COL] = {{1,2,3},{4,5,6}};
    int vet[COL] = {7,6,7};
    f1( (int*)m );
    f2( m );
    f3( vet );

    getchar();
    return 0;
}
```

Outro exemplo que representa uma matriz em um vetor pode ser visto na API OpenGL. O OpenGL utiliza matrizes 4x4 para representar as transformações. Para se ler ou setar uma matriz, utiliza-se um vetor de 16 posicoes, como mostrado no seguinte exemplo:

```
GLfloat m[16]; //usado para leitura das matrizes
glGetFloatv (GL_PROJECTION_MATRIX, m); //ou GL_MODELVIEW_MATRIX para a modelview
for(int l=0; l<4; l++)
{
   printf("\n");
   for(int c=0; c<4; c++)
   {
       printf(" %.2f", m[l * 4 + c] );
   }
}
```



3. Ponteiros e Alocação Dinâmica

A alocação da memória RAM do computador é divida em 4 áreas:

a) Código do programa
b) Variáveis estáticas e globais
c) Memória dinâmica – função `malloc()`
d) Pilha – chamadas de funções

Todas essas áreas podem ser acessadas sabendo-se o endereço de memória apropriado. Uma variável do tipo ponteiro é usada para armazenar um endereço de memória. Esse endereço pode representar o local onde uma função ou uma variável (ou vetor) está armazenada. Em termos de sintaxe, a diferença entre um ponteiro e uma variável normal é a presença do asterisco antes do nome da variável.

Os endereços vão da posição 0 (zero) até a quantidade máxima de memória disponível na máquina. Para arquiteturas de 32 bits, pode-se endereçar até 4 GB.

Quando se declara uma variável inteira, por exemplo, esta vai estar associada a um endereço de memória. Pode-se utilizar um ponteiro para guardar este endereço, como no seguinte exemplo. É usado o símbolo "*" para diferenciar ponteiros de variáveis normais. Pode-se criar ponteiros para qualquer tipo de dados.

```
int a;
int *pont;
pont = &a; //neste caso pont está apontando para o endereço da variável a.
*pont = 10; //isso equivale a fazer a = 10;
printf("\n%d %d", &a, pont); //imprime o endereço das variáveis, que são iguais
printf("\n%X %X", &a, pont); //imprime o endereço em hexadecimal
```

Cada variável é armazenada em um endereço específico da memória RAM. Variáveis do tipo int e float ocupam 4 bytes, enquanto que as do tipo char ocupam apenas 1. Variáveis ponteiros ocupam sempre 4 bytes, independente do tipo que forem: char, int, double, etc.

```
int a, b, c, *p1, *p2, **pp; //p é um ponteiro e pp um ponteiro de ponteiro
a = 10;   //atribui o valor 10 na variável a, que está no endereço 100
b = 20;   //atribui o valor 20 na variável b, que está no endereço 104
p1 = &c;  //faz p1 apontar para o endereço onde esta a variável c
*p1 = 30; //atribui 30 a variável c via o ponteiro p1.
p2 = p1;  //faz p2 apontar para onde aponta p1
pp = &p1; //faz pp apontar para o endereço onde está p1
printf("%d", *p1) ; //imprime 30, que é o valor que está onde p1 está apontando
printf("%d", &p1);  //imprime 112, que é o endereço onde p1 está armazenado
printf("%d", p1);   //imprime 108, que é o endereço de onde p1 está apontando
printf("%d", *pp);  //imprime 108
printf("%d", **pp); //imprime 30
```

Variável	Valor	Endereço
a	10	100
b	20	104
c	30	108
*p1	108	112
*p2	108	116
**pp	112	120

Como mostrado no exemplo anterior, existem duas formas de se referenciar uma variável: pelo seu valor ou pelo seu endereço. Neste exemplo, o nome “a” representa o valor da variável, e o nome “&a” representa o endereço da variável. Ponteiros sempre armazenam endereços, por isso a atribuição p1 = &c.

Ponteiros podem ser acessados de 3 formas distintas:

1. O endereço onde ele aponta: p1
2. A informação que está armazenada no endereço onde o ponteiro aponta: *p1
3. E o endereço onde o ponteiro está alocado: &p1.

A alocação de vetores segue uma representação sequencial da memória utilizada, como mostrado na seguinte figura. Guarda-se apenas o endereço inicial do vetor. As demais posições são calculadas com deslocamentos em relação ao endereço base. Deve-se observar que v = &v[0].

```
int a = 10, v[4] = {1,2,3,4}, *p;
p = &v[0];
p = v;
for(int i=0; i<4; i++)
{
   printf("%d", *(p + i) ); //ou p[i]
}
for(int i=0; i<4; i++)
{
   printf("%d", *p );
   p++;
}
```

Variável	Valor	Endereço
a	10	100
v	1	104
	2	108
	3	112
	4	116
*p	104	120
	112	124

A representação de matrizes também é feita de maneira sequencial. Inicialmente são armazenados os elementos da primeira linha, seguidos pelos da segunda linha e assim por diante. Para matrizes de maior dimensão, utiliza-se a mesma estratégia. O seguinte exemplo apresenta duas formas de imprimir os valores desta matriz com o uso de ponteiros.

```
int m[2][3] = {{1,2,3},{4,5,6}}, *p;
p = (int*)m;
for(int l=0; l<2; l++)
{
   for(int c=0; c<3; c++)
   {
      int pos = l*3 + c;
      printf("%d", *(p + pos) ); //ou p[pos]
   }
}
for(int i=0; i<6; i++)
{
   printf("%d", *p );
   p++;
}
```

Variável	Valor	Endereço
a	10	100
m	1	104
	2	108
	3	112
	4	116
	5	120
	6	124
		128
*p	104	132
		136

Neste exemplo pode-se observar como é feito a aritmética de ponteiros. Quando um ponteiro é incrementado, desloca-se um número de bytes igual ao tipo do ponteiro, ou seja, para ponteiros inteiros, o deslocamento é de 4 bytes. Se p aponta para o endereço 100, p++ faz p apontar para o endereço 104, pois cada inteiro gasta 4 bytes.

Ponteiros podem ser usados para acessar matrizes n-dimensionais de dados. Dada uma matriz bidimensional, tem-se a seguinte equivalência de comandos:

```
int m[2][2];
```

`m[0][0];`	`*(*(m + 0) + 0);`	`*(*m);`
`m[0][1];`	`*(*(m + 0) + 1);`	`*(*m + 1);`
`m[1][0];`	`*(*(m + 1) + 0);`	`*(*(m + 1));`
`m[1][1];`	`*(*(m + 1) + 1);`	`*(*(m + 1) + 1);`

- `m` aponta para o endereço base da matriz bidimensional.
- `*(m + 0)` aponta para o endereço da primeira linha da matriz.
- `*(m + 1)` aponta para o endereço da segunda linha da matriz e assim por diante.

Uma aplicação típica de ponteiros é para implementar a função troca(), que troca o valor de duas variáveis. Observe que a função recebe como argumento dois endereços de memória, que são atribuídos a dois ponteiros.

```
void troca(int *a, int *b)
{
   int c;
   c = *a;
   *a = *b;
   *b = c;
}
void main(void)
{
   int v1 = 1, v2 = 2;
   printf("\n %d %d", v1, v2);
   troca(&v1, &v2);
   printf("\n %d %d", v1, v2);
}
```

Just C/C++ things

Caso não se saiba de antemão o espaço que será necessário para armazenar os dados, como por exemplo, o número de valores a serem utilizados no cálculo da média, deve-se então usar uma estratégia de alocação de memória em tempo de execução (alocação dinâmica). Em C, isso pode ser feito com o uso de ponteiros, que são variáveis cujo conteúdo é um endereço de memória. O símbolo NULL (`stddef.h`) indica que o ponteiro está apontando para o endereço de memória zero. O seguinte exemplo ilustra a alocacao de dois vetores

Following may be the declaration for NULL Macro depending on the compiler.

```
#define NULL ((char *)0)

or

#define NULL 0L

or

#define NULL 0
```

```
float *vet = NULL;    /* vet é um ponteiro para float */
int   *tabela = NULL; /* tabela é um ponteiro para int */

vet    = (float*)malloc( 3*sizeof(float) );
tabela = (int*)  malloc( 2*sizeof(int)   );
tabela[0] = 100000;
tabela[1] = 88;
free(vet);  //libera o espaço alocado, mas vet continua apontando para o mesmo endereço.
vet = NULL; //agora vet aponta para NULL
```

Como deveria ser o protótipo da função free() para que ela liberasse o espaço alocado e ao mesmo tempo fizesse o ponteiro apontar para NULL?

A seguinte figura apresenta a configuração da memória antes da alocação dos dois vetores. Observe que os ponteiros foram inicializados com NULL, ou seja, endereço 0.

Variável	Valor	Endereço
vet	0	100
tabela	0	104
		108

Apos a alocação, tem-se a seguinte configuração. O ponteiro vet passa a apontar para o endereço 112 e o ponteiro tabela para o endereço 124. Observe que o espaço apontado por vet contém lixo, pois ainda não foi inicializado.

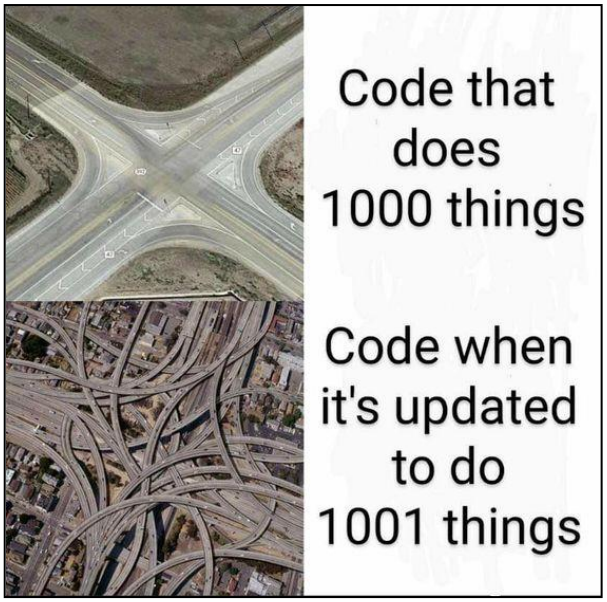
Variável	Valor	Endereço
vet	112	100
tabela	124	104
		108
	Lixo	112
	Lixo	116
	Lixo	120
	100000	124
	88	128
		132

O operador `sizeof( tipo )` retorna o número de bytes ocupado pelo tipo (ver tabela de tipos básicos). A função `malloc( int size )` aloca size bytes e retorna o endereço inicial da região alocada. Este endereço retornado é atribuído ao ponteiro que passa a apontar para a região alocada. A função `free()` libera o espaço alocado. Para se usar a função `malloc()` deve-se incluir a biblioteca `stdlib.h`.

Programas interessantes:

1) Espaço ocupado por uma alocação dinânima: Uma questão curiosa refere-se a situações onde são realizadas diversas alocações consecutivas, como ilustrado no seguinte código.

```c
#define TAM 1

int main(void)
{
  char *a;
  int i;
  int novo, velho;

  a = (char*)malloc(TAM);
  velho = (int)a;

  for(i=0; i < 200; i++)
  {
     a = (char*)malloc(TAM);
     novo = (int)a;

     printf(" %d ", novo-velho); //subtrai os endereços.
     velho = novo;
  }
  getchar();
  return 0;
}
```

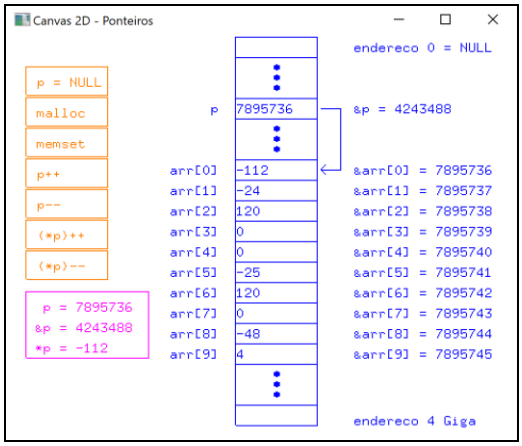


Usando a IDE Codeblocks, com o compilador GCC, com TAM = 1, usa-se 16 bytes para cada alocação. Usando TAM = 11, 24 bytes. Usando o Visual Studio em modo debug, com TAM=1, gasta-se 64 bytes a cada alocação, e em modo Release 32 bytes. Passando para TAM=1024, 1088 em debug e 1048 bytes em release. Esse espaço extra alocado deve ser usado para algum controle do programa/compilador, visto que em modo debug ele gasta mais. Quando a alocação é grande, o espaço perdido é menor, logo, quando possivel, deve-se fazer menor número de alocações e com maior espaço.

2) Visualização da memória RAM do computador: O seguinte programa ilustra a leitura sequencial de um bloco de memória RAM, e somente para quando ocorrer violação de acesso.


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

#define SCREEN 1000
#define OFFSET 1000

int main(void)
{
     char *c = NULL;
     char op = 0;
     int cont=0;

     c = (char*)malloc(sizeof(char));
     c -= OFFSET; //move o ponteiro para um endereço anterior

     while(op != 27)  //ESC para sair
     {
        printf("%c", *c); //imprime a RAM em modo sequencial até TRAVAR
        c++;
        cont++;
        if( cont == SCREEN )
        {
           op = getchar();
           printf("\n %c", op);
           cont = 0;
        }
     }
     return 0;
}
```

3) Visão gráfica da RAM: O seguinte programa ilustra graficamente endereços e valores da memória RAM para um programa em tempo real. Ver material de programação gráfica em linguagem C.



A alocação de memória em linguagens como C# e Java é semelhante à linguagem C. Utiliza-se o operador `new()` para isso ao invés de `malloc()`. A grande diferença está na forma como a memória é liberada. Em C, existe uma função específica – `free()` – que deve ser chamada pelo programador. Se o programador não liberar a memória ocorre o chamado memory leak (vazamento de memória), que faz o programa ocupar cada vez mais memória, até que esta seja esgotada. Em C# e Java existe o garbage collector, que decide quando liberar a memória. Isso ocorre quando não existem mais referências para um objeto. Quando o garbage collector executa podem ocorrer pequenos travamentos na aplicação. Novamente, por serem linguagens de mais alto nível, o programador perde um pouco o controle de como o programa funciona.
A passagem de parâmetros por referência em C# é feita por pela palavra-chave `ref`. A sintaxe é mais simples que a passagem por referência com ponteiros em C. C# tambem permite o uso de ponteiros, por meio da instrução `fixed`, porém o código deve estar dentro de um contexto `unsafe`.

4. Questões de Ponteiros e alocação dinâmica para discussão

```c
int *p = (int*)malloc(100*sizeof(int));
if( p = NULL)
{
   printf("Erro de alocação");
   exit(1);
}
p[0] = 10;
```

```c
int *p;
unsigned int i;
p = (int*)malloc(sizeof(int));
i = (int)p;
printf("%d %p", i, p);
```

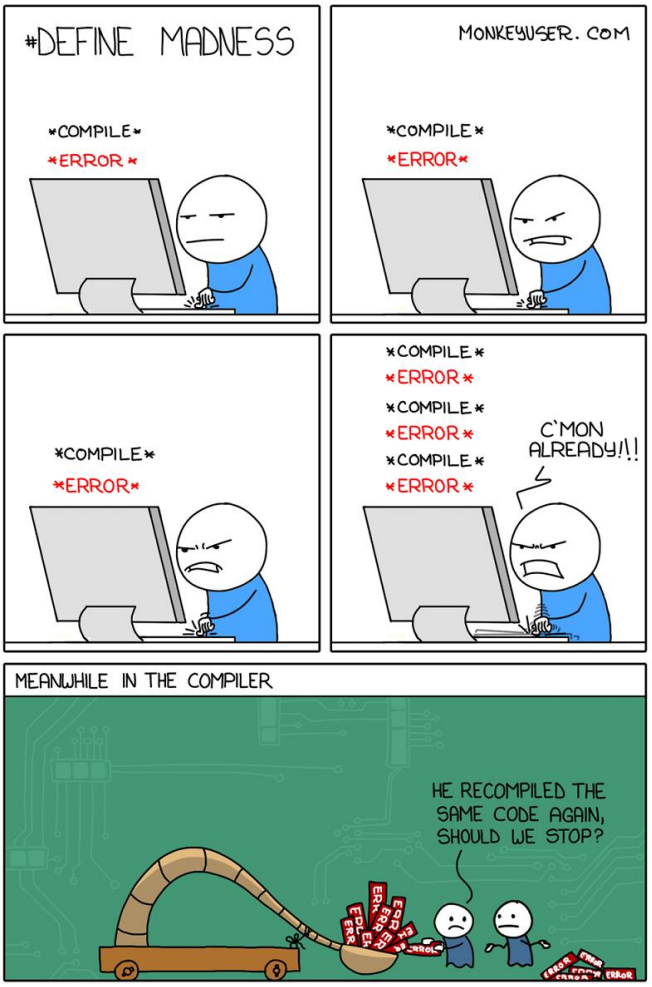
```c
int *p = NULL;
int i;
for(i=0; i<100; i++)
{
  p = (int*)malloc(sizeof(int));
}
```

```c
char *c;
int *p;
int v[]={1,2,3,4};
c = (char*)v;
c+=4;
p = (int*)c;
printf("\np=%d", *p);
```

```c
int *p;
p = NULL;
*p = 10;
```

```c
void *p;
p = (int*)malloc(sizeof(int));
p++; //não se pode incrementar ponteiro void
```

```c
int *p;
unsigned int i;
p = (int*)malloc(sizeof(int));
*p = 100;
i = (int)p;
p = (int*)i;
printf("\np=%d", *p);
```



```
int *p;
p = (int*)malloc(sizeof(int));
free(p);
free(p);
p++;
free(p);
```

5. String

A representação de strings em C é feita por meio de vetores do tipo char. C não possui um tipo básico específico. Cada caractere está associado um valor decimal, segundo a tabela Ascii. Nesta tabela, por exemplo, a letra 'a' tem valor decimal 97 e a letra 'A' valor 65. O digito '1' tem valor decimal 49.

```
for(int i=0;  i<255; i++)
   printf(" %c = %d", i, i);
```

Para se converter um caractere de minúsculo para maiúculo, por exemplo, pode-se utilizar a função `char toupper(char)` ou o seguinte algoritmo. Neste exemplo, o trecho de código `'A' - 'a'` representa a diferença entre a letra minúscula para maiúcula, que vale -32.

```
char c = 'a', b;
b = c + 'A' - 'a';
```

Caracteres podem ser somados da mesma forma como se faz com números inteiros, como no seguinte exemplo.

```
char c = 'a' + 1;
printf("%c", c);  //imprime 'b', pois 97+1 = 98
```

Desta forma, pode-se fazer verificações sobre o valor de caracteres

```
char letra = getchar();
if( letra  >= 'a' && letra <= 'z' )
    printf("eh uma letra minuscula");
```

Outra particularidade do C, é que quando se aloca uma string deve-se reservar um espaço a mais para guardar um caractere terminador de string, o `'\0'` (cujo valor decimal é zero). Sem ele, não se pode saber qual o tamanho da string.

```
char str[3] = "oi";      //aloca 3 espaços, sendo um para o '\0'
char s2[] = "ola mundo"; //espaço calculado automaticamente, incluindo o o '\0'
char cidade[] = {'S','a','n','t','a',' ','M','a','r','i','a'}; // o '\0' é adicionado
char *s = (char*)malloc(100*sizeof(char)); //aloca 100 espaços, sendo 99 uteis.
```

Neste exemplo, `str[0]= 'o'`, `str[1]= 'i'` e `str[2]= '\0'`. Deve-se cuidar para nunca fazer atribuições do seguinte tipo, pois algumas são inválidas:

```
char s[100], *p;
s = "teste";  //erro – deve-se utilizar a funcao strcpy(). Nesta operação o endereço onde s
              aponta deveria ser mudado, mas como ele é um vetor, isso não pode acontecer.
p = "teste";  //funciona, porem a string literal "teste" não pode ser modificada.
s = "teste" + "teste"; //operação inválida
```

O seguinte exemplo ilustra como se comporta o operador `sizeof()` ao se referenciar a vetores e ponteiros de strings.

```
char  a[100] = "abc";
char  *b = (char*)malloc(200);
char  *c = a;
```

```
printf("%d", sizeof(a)); //imprime 100
printf("%d", sizeof(b)); //imprime 4
printf("%d", sizeof(c)); //imprime 4
```

C fornece diversas funções para manipulação de strings, como por exemplo, `strcat()`, que concatena duas strings, `strcpy()`, que faz cópia de strings, `strlen()`, que retorna o tamanho da string, `strcmp()` que compara duas strings, `gets()`, que lê uma string do teclado. Todas estas funções e muitas outras estão na biblioteca string.h. Caso alguma destas funções for usada, deve-se adicionar no topo do programa a diretiva:

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

As seguintes funções fazem o mesmo que `strlen()` da biblioteca string.h faz. Essa função é usada para contar quantos caracteres uma string possui. Observe que o caractere `'\0'` não é contado.

```
int conta(char *s)
{
   int cont = 0;
   while( *(s+cont)!='\0')
   {
      cont ++;
   }
   return cont;
}
```

```
int conta(char *s)
{
   int cont = 0;
   for( ;s[cont]!='\0'; cont++);
   return cont;
}
```

```
int conta(char *s)
{
   char *aux;
   for(aux=s;*aux!='\0';aux++);
   return (int)(aux-s);
}
```

Outra função muito útil é a função `strcpy()`, que faz a cópia de uma string para outra. Deve-se lembrar que não se pode atribuir uma string a uma variável do tipo vetor usando o operador =, exceto se for na declaração. Sua implementação pode ser feita de várias formas. Deve-se lembrar de fazer a cópia do caractere terminador (`'\0'`), e garantir que a string de destino tenha espaço suficiente.

```
char * strcpy(char *dest, char *orig)
{
   int i;
   if( dest == NULL || orig == NULL)
      return NULL;
   for(i=0; i < strlen(orig); i++)
   {
      dest[i] = orig[i];
   }
   dest[i] = '\0'; //nunca esquecer o '\0'
   return dest;
}
```

```
char * strcpy(char *dest, char *orig)
{
   char *tmp;
   if( dest == NULL || orig == NULL)
      return NULL;
   for( tmp = dest; *orig != '\0'; orig++, tmp++)
   {
      *tmp = *orig;
   }
   *tmp = *orig; //nunca esquecer o '\0'
   return dest;
}
```

Outra forma de implementar a função `strcpy()` é fazendo a alocação do espaço dentro da própria função, como mostrado no seguinte exemplo. Deve-se observar que na função `malloc()` foi alocado um byte a mais, para armazenar o caractere `'\0'`.

```
char * strcpy(char *orig)
{
   int i, tam = strlen(orig);
   char *dest = (char *) malloc( (tam + 1) * sizeof(char) );

   for(i=0; i < tam; i++)
   {
      dest[i] = orig[i];
```

```
    }
    dest[i] = '\0'; //nunca esquecer o '\0'
    return dest;
}
```

Observe que a string retornada foi alocada com malloc, ou seja, alocação dinâmica. Esse espaço de memória continua válido mesmo após a finalização da função. O mesmo não ocorre com o retorno de uma string com alocação estática, visto que o espaço alocado é perdido quando a função é finalizada, como no seguinte exemplo, podendo causar erros aleatórios.

```
char * retornaStr()
{
    int arr[4] = "abc";
    return arr; //fonte de erros aleatorios
}
```

Outra função semelhate é a função para concatenar duas strings e retornar uma nova string. Essa versão é um pouco diferente da funcao strcat() da biblioteca string.h. A ideia é descobrir o tamanho das duas strings e alocar somente o espaço necessário para comportar as duas. Fazendo-se uso das funções da biblioteca string.h, tem-se a seguite solução. Fica como exercício a versão sem uso das funções de string. Não esquecer também o espaço para o caractere`'\0'`.

```
char * concatena(char *s1, char *s2)
{
    int tam = strlen(s1) + strlen(s2) + 1;
    char *dest = (char *) malloc( tam * sizeof(char) );
    strcpy(dest, s1);
    strcat(dest, s2);
    return dest;
}
```

Outra função muito útil é a `strcmp()`, que faz a comparação de duas strings, retornando 0 (strings iguais), maior que zero para indicar que o primeiro caractere diferente é maior na primeira string, e menor que zero caso contrário. A implementação desta função é dada como **exercício**. Fica também como exercício a implementação da função `strstr()`, que faz a procura de uma substring dentro de uma string.

Para fazer a leitura de strings pode-se utilizar a função `scanf()` ou `gets()`. Deve-se garantir que a string de destino tenha espaço suficiente para armazenar a string lida.

```
scanf("%s", a);      //lê ate que seja encontrado um espaço ou <enter>
scanf("%[^\n]", a);  //lê ate que seja pressionado um enter (lê inclusive espaços)
scanf("%6[^\n]", a); //lê ate que seja pressionado um enter, limitado a 6 caracteres
gets(a);             //lê até que seja pressionado um enter
```

Para fazer a conversão entre tipos pode-se utilizar as funcoes `itoa()` e `atoi()`. A função `itoa()` converte um inteiro para string e `atoi()` converte de string para inteiro. Como exemplo, é muito comum em programação a geração de mensagens textuais que devem ser processadas, como por exemplo, no caso de um jogo: “Voce tem 5 vidas restantes”. Neste exemplo, deve-se fazer a concatenação de strings com números. Existem duas forma de produzir essa string final. A solução mais trabalhosa faz uso das funções `itoa()`, `strcpy()` e `strcat()`, como mostrado no seguinte exemplo.

```
int vidas = 5;
char num[2];
char string[100];
char prefix[10] = "Voce tem ";
char sufix[20] = " vidas restantes";
itoa(vidas, num, 10); //numero, string, base de conversao
strcpy(string, prefix);
```

```
strcat(string, num);
strcat(string, sufix);
printf("%s", string); //Imprime a mensagem “Voce tem 5 vidas restantes”
```

Outra forma mais compacta de implementação é com o uso da função `sprintf()`. Essa função é muito semelhante a função `printf()`, com a **única** diferença que o resultado é guardado em uma string, ao invés de ser impresso na tela. A versão com `sprintf()` do exemplo anterior fica:

```
int vidas = 7;
char string[100];
sprintf(string, "Voce tem %d vidas restantes", vidas);
printf("%s", string); //Imprime a mensagem “Voce tem 7 vidas restantes”
```

Nos dois exemplos, cabe ao programador garantir que a string de destino tenha espaço suficiente para armazenar todos os caracteres, caso contrário ocorrerá um overflow, que é um erro difícil de ser detectado e que pode levar a aplicação a travar de forma imprevisível.

Se fosse usado o comando `sprintf(string, "Voce tem %03d vidas restantes", vidas)`, seria gerada a string “Voce tem 007 vidas restantes”. Este comando também pode ser usado para converter qualquer tipo primitivo para string, como no seguinte exemplo.

```
float x = 3.14159265358979323846264338327795;
sprintf(string, "X vale %0.5f", x); //gera a string “X vale 3.14159”
```

Pode-se criar também matrizes de caracteres, como no seguinte exemplo:

```
char mat[10][100];  //guarda um vetor de strings (ou uma matriz de caracteres)

char str1[] = "Primeira linha";
char str2[] = "Segunda linha";

strcpy(mat[0], str1);
strcpy(mat[1], str2);
strcpy(mat[2], "Terceira linha");

printf("%s", mat   ); //imprime "Primeira linha"
printf("%s", mat[0]); //imprime "Primeira linha"
printf("%s", mat[1]); //imprime "Segunda linha"
```

A linguagem C é uma das mais complexas e trabalhosas para se tratar strings, pois ela não possui um tipo de dado para isso. Linguagens como C# e Java possuem o tipo `string`, e com isso, por exemplo, a concatenação de duas strings pode ser feita com o operador +, da mesma forma como se somam dois números. O tipo `string` tem tamanho ilimitado, ou seja, o programador **não precisa se preocupar** com alocação de memória, adição de `'\0'`, indexação fora dos limites do vetor e nem liberação de memória.

6. Estruturas

Para definir tipos de dados complexos de tipos variados, C define o tipo struct, que permite aglomerar em um único elemento diversos tipos básicos ou complexos. A sintaxe de definição é a seguinte:

```
struct nome{
   /* colocar aqui os tipos desejados */
   /* colocar aqui os tipos desejados */
};
```

Um exemplo muito comum em jogos é a definição de primitivas geométricas. A definição de um ponto em 3D pode ser da seguinte maneira:

```
struct ponto3D
{
   float x, y, z;
   int *p;
};
```

Com esta definição, o tipo "struct ponto3D" pode ser usado para armazenar pontos do espaço 3D. O exemplo seguinte mostra a utilização de estruturas.

```
struct ponto3D p1, p2; /*define duas variáveis do tipo struct ponto3D */
struct ponto3D *ponteiro;
p1.x = 20;              /* atribui 20 ao campo x de p1 */
p1.y = 1.002;
p1.z = 70;
p2.x = p1.x – p1.z;
ponteiro->x = 3; // quando a variável eh do tipo ponteiro, deve-se utilizar ->
ponteiro->p = &a; // quando a variável eh do tipo ponteiro, deve-se utilizar ->
p1.p = &a;
```

Para acessar os elementos da estrutura deve-se usar ponto. Caso a estrutura for alocada dinamicamente, deve-se usa o operador seta (->).

```
struct ponto3D *p1;
p1 = (struct ponto3D *) malloc( sizeof(struct ponto3D) );
p1->x = 20;
```

Outra característica do C é permitir criar nomes de tipos. No seguinte exemplo:

```
typedef struct
{
   float x, y, z;
}Ponto3D;
```

define-se `Ponto3D` como um tipo, o que facilita a futura utilização. Costuma-se utilizar letra inicial maiúscula:

```
Ponto3D *p1;
p1 = (Ponto3D *) malloc( sizeof(Ponto3D) );
p1->x = 20;
```

Estruturas podem conter outras estruturas bem como pode também definir vetores de estruturas:

```
Ponto3D vet[100]; /*vetor com 100 posições do tipo ponto3D */
/*definição da estrutura quadrado compostas por 4 ponto3D
  e mais uma string para conter o nome do elemento */
typedef struct{
   char nome[100];
   Ponto3D pontos[4];
}Quadrado;
```

Quando se aloca um vetor de estruturas de forma dinâmica, pode-se utilizar o operador [] para fazer a indexação, seguido do operador ".".

```
struct ponto3D *p;
struct ponto3D var;

//variavel estática. Usar o operador “.”
var.p = &a;
var.x = 100;

//uma única estrutura dinâmica. Usar o operador ->
p = (struct ponto3D*) malloc(sizeof(struct ponto3D));
p->p = &a;
p->x = 10;

//vetor de estruturas. Usar o operador [] seguido do operador “.”
p = (struct ponto3D*) malloc(10*sizeof(struct ponto3D));
p[0].p = &a;
p[0].x = 10;
printf(" %d", p->x);
printf(" %d", p[0].x);
```

A passagem de estruturas para funções é feita por ***valor***, e é semelhante aos outros tipos de dados. Caso for passado um vetor de estruturas, por ser vetor, a passagem é por referência. Neste exemplo a variável `quadro1` é passada por referência para a função imprime. A função imprime recebe somente o endereço da localização da variável `quadro1`.

```
/* a definição da estrutura quadrado e ponto3D deve ser feita fora de
   main( ) pois a função imprime também usa esta estrutura */
void imprime(Quadrado *q)
{
   int cont;
   printf("Nome da variável: %s", q->nome );
   for( cont=0; cont<4; cont++)
   {
      printf("%f ", q->pontos[cont].x);
      printf("%f ", q->pontos[cont].y);
      printf("%f ", q->pontos[cont].z);
   }
}

int main(void)
{
   /*declaração da variável quadro1 */
   Quadrado quadro1;
   /* eh passado para a função imprime o endereço (&) da variável
      quadrado, que deve ser inicializada */
   imprime( &quadro1 );
   return 0;
}
```

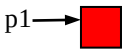
A cópia de uma variável struct para outra ocorre como qualquer outra variável. Todo o conteúdo de um elemento é copiado para o outro. O seguinte exemplo ilustra uma atribuição t2 = t1 (ambos do tipo struct teste). Deve-se observar que um elemento da estrutura é um ponteiro, que aponta para um espaço alocado de forma dinâmica. Nesse caso, t2 aponta para o mesmo endereço apontado por t1 (**não é alocado um novo espaço para t2 apontar**).

```
typedef struct teste
{
   int i;
   int  v[10];
   char str[10];
   float *pi;
}Teste;
void main(void)
{
   Teste t1, t2; //tem-se alocadas duas estruturas Teste de forma estatica

   //inicializa alguns campos de t1
   t1.i = 10;
   strcpy(t1.str, "abc");
   t1.v[1] = 1; //ou *(t1.v+1) = 1
   t1.pi = (float*)malloc(sizeof(float));
   *t1.pi = 3.1415;

   t2 = t1; //copia tudo de t1 para t2, nesse caso os 60 bytes da estrutura
}
```

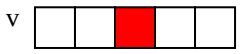
Segue um exemplo completo com diversas formas da utilização de matrizes, com alocação estática e dinâmica. Ao final do exemplo, ilustra-se a alocação de matrizes dinâmicas de estruturas.

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

struct aluno
{
   int a;
   float f, *p;
   char str[100];
};
typedef struct aluno Aluno;

//passagem por copia
void printAlunoCopia(Aluno a)
{
   printf("\n%d %s", a.a, a.str);
}
//passagem por referencia
void printAlunoRef(Aluno *p)
{
   printf("\n%d %s", p->a, p->str);
}

void main(void)
{
   Aluno a1, *p1, v[5], m[3][3], *p2, *v2[5], **pp, ***ppp;

   //1) variavel estatica - ja esta alocada
   a1.a = 10;
   a1.f = 3.1415;
   a1.p = (float*)malloc(10 * sizeof(float)); //aloca espaco para o atributo da estrutura
   strcpy(a1.str, "Isso eh uma string");
   printAlunoCopia(a1);
   printAlunoRef(&a1);
```

```c
//2) variavel ponteiro - deve alocar
p1 = (Aluno*)malloc(sizeof(Aluno));
p1->a = 10;
p1->f = 3.1415;
p1->p = (float*)malloc(10 * sizeof(float));
strcpy(p1->str, "Isso eh uma string");
printAlunoRef(p1);
```

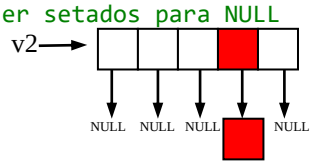



```c
//3) vetor estatico - ja esta alocado
v[2].a = 10;
v[2].f = 3.1415;
v[2].p = (float*)malloc(10 * sizeof(float));
strcpy(v[2].str, "Isso eh uma string");
printAlunoCopia(v[2]);
printAlunoRef(&v[2]);
```

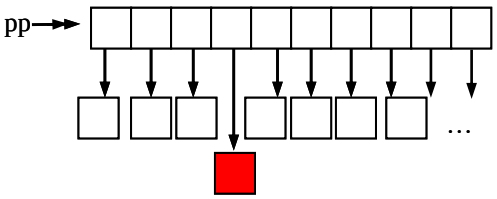



```c
//4) Ponteiro para um vetor dinamico - deve alocar um vetor de alunos
p2 = (Aluno*)malloc(10 * sizeof(Aluno)); //aloca 10 alunos
p2[3].a = 10;
p2[3].f = 3.1415;
p2[3].p = (float*)malloc(10 * sizeof(float));
strcpy(  p2[3].str, "Isso eh uma string");
strcpy( (p2+3)->str, "Isso eh uma string"); //igual ao comando anterior
printAlunoCopia(p2[3]);
printAlunoRef(&p2[3]);
```

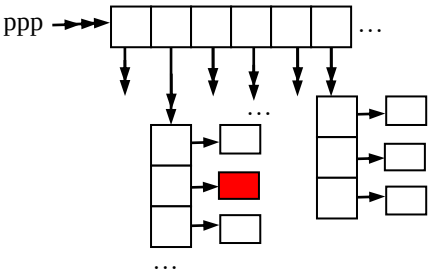

```c
//5) vetor de ponteiros - o vetor ja esta alocado, soh nao existem os alunos.
v2[3] = (Aluno*)malloc(sizeof(Aluno)); //aloca 1 aluno na posicao 3 do vetor.
v2[3]->a = 10;                         //Os outros devem ser setados para NULL
v2[3]->f = 3.1415;
v2[3]->p = (float*)malloc(10 * sizeof(float));
strcpy(v2[3]->str, "Isso eh uma string");
printAlunoRef(v2[3]);
```




```c
//6) ponteiro de ponteiro - deve alocar o vetor de ponteiros e apos os alunos
pp = (Aluno**)malloc(20 * sizeof(Aluno*)); //aloca vetor com 20 ponteiros para aluno
for (int i = 0; i < 20; i++)
   pp[i] = (Aluno*)malloc(sizeof(Aluno)); //aloca 1 aluno
pp[3]->a = 10;
pp[3]->f = 3.1415;
pp[3]->p = (float*)malloc(10 * sizeof(float));
strcpy(pp[3]->str, "Isso eh uma string");
printAlunoRef(pp[3]);
```




```c
//7) ponteiro de ponteiro de ponteiro - alocacao de uma matriz de ponteiros para aluno
ppp = (Aluno***)malloc(20 * sizeof(Aluno**)); //aloca vetor com 20 ponteiros para ponteiro para aluno
for (int i = 0; i < 20; i++)
{
   ppp[i] = (Aluno**)malloc(20 * sizeof(Aluno*)); //aloca vetor com 20 ponteiros para aluno
   for (int j = 0; j < 20; j++)
   {
      ppp[i][j] = (Aluno*)malloc(sizeof(Aluno)); //aloca 1 aluno
   }
}
ppp[1][1]->a = 10;
ppp[1][1]->f = 3.1415;
ppp[1][1]->p = (float*)malloc(10 * sizeof(float));
strcpy(ppp[1][1]->str, "Isso eh uma string");
printAlunoRef(ppp[1][1]);
```



```
}
```

Devido ao **alinhamento de bytes** (veja http://0xc0de.wordpress.com/2008/10/31/alinhamento-alignment-e-preenchimento-padding/), o espaço retornado pelo operador `sizeof` pode ser maior que o esperado (ao se alocar structs), quando se usam tipos que sejam menores que uma *word* (que representa o número de bits que são processados a cada iteração de uma instrução. Para máquinas de 32 bits, a word ocupa 32 bits), como no caso do char e short.

```
struct qualquer  //sizeof = 12 bytes
{
   int x, *p;
   char c;
};

struct qualquer //sizeof = 12 bytes
{
   int x, y;
   char c, d, e, f;
};

struct qualquer //sizeof = 16 bytes
{
   int x;
   char c;
   int z;
   char d;
};

struct qualquer //sizeof = 8 bytes
{
   int x, y;
};

struct qualquer //sizeof = 16 bytes
{
   long long x, y;
};

struct qualquer //sizeof = 16 bytes
{
   long long x, *y, *c;
};
```

Funções e Retorno de parâmetros

Uma função em C pode receber um número qualquer de argumentos e pode retornar no máximo uma única variável (exceto se for por referência). Funções podem também não receber e nem retornar argumentos (void). Seguem exemplos de protótipos de funções.

```
void teste(void);
int calcula(void);
void soma(int a, int b, int *resp);
struct x * media(char **p, int v[]);
```

O retorno de tipos primitivos em uma função é uma operação elementar. Isso ocorre pois a passagem é feita por valor, e não por referência. Isso se aplica tanto para variaveis do tipo int, float, bem como para estruturas, como no seguinte exemplo.

```c
#include <stdio.h>

typedef struct x
{
    int i;
    int b;
}X;
```

```c
X init()
{
    X a;
    a.i = 3;
    a.b = 77;
    return a;
}
```

```c
int main()
{
    X ret;
    ret = init();
    printf("%d %d", ret.i, ret.b );
    return 0;
}
```

O tratamento de vetores na linguagem C não é feito por valor e sim por referência, o que garante maior eficiência, porém ao mesmo tempo dificulta o retorno. Isso vale para vetores de qualquer tipo, incluindo strings. Desta forma, o seguinte código está incorreto, apesar de até poder funcionar em algumas situações. O compilador Microsoft Visual Studio gera a seguinte warning de compilação: "`returning address of local variable or temporary`".

```c
int * init()
{
    int v[10]; //vetor local a função init()
    int i;
    for(i = 0; i < 10; i++)
        v[i] = i;
    return v; // = &v[0]
}
```

```c
int main()
{
    int *p, i;
    p = init();

    for(i=0; i < 10; i++)
        printf("%d ", p[i]) ;

    return 0;
}
```

Isso ocorre porque todas as variáveis locais de uma função são liberadas quando a mesma é finalizada. Desta forma, o seguinte exemplo é valido visto que na chamdada da funcao `teste()`, a funcao `init()` ainda continua ativa.

```c
void teste(int *p)
{
    int i;
    for(i=0; i < 10; i++)
        printf("%d ", p[i]) ;
}

void init()
{
    int v[10], i; //vetor local a funcao init()
    for(i = 0; i < 10; i++)
        v[i] = i;
    teste( &v[0] );
}
```

```c
int main()
{
    init();
    return 0;
}
```

Para poder retornar um vetor, este deve ser alocado de forma dinâmica, visto que variáveis dinâmicas somente são liberadas com uma chamada explicita da função `free()`.

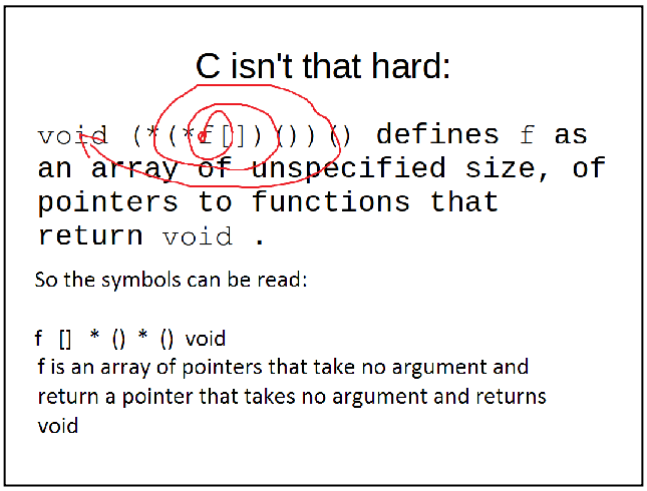
```c
int * init()
{
    int *v, i;
    v = (int *) malloc( 10*sizeof(int) );
    for(i = 0; i < 10; i++)
        v[i] = i;
    return v;
}
```

```c
int main()
{
    int *p, i;
    p = init();

    for(i=0; i < 10; i++)
        printf("%d ", p[i]) ;

    return 0;
}
```

Ponteiro para Funções

Pode-se criar ponteiros para funções na linguagem C. Esse recurso é útil em diversas situações e é utilizado, por exemplo, pela biblioteca Glut/Opengl (como call-backs) e pela função `qsort()`, disponível na linguagem C para fazer a ordenação de vetores. O seguinte exemplo ilustra alguns usos desse recurso. O vetor de ponteiros para função pode ser muito útil para tratar entrada do usuário. Como exemplo, vamos considerar uma calculadora. Se o usuário digitar 0, deve somar, se digitar 1, deve subtrair e assim por diante. Sem o uso de vetor de ponteiros, seria necessário implementar um switch/case.

```c
void funcaoVoid(void)
{
    printf("\nSou funcao void/void");
}

int funcaoInt(int a)
{
    printf("\nSou funcao int/int : %d", a);
}

void funcao1()
{
    printf("\nSou funcao 1");
}

void funcao2()
{
    printf("\nSou funcao 2");
}

void funcao3()
{
    printf("\nSou funcao 3");
}


int main(void)
{
    void(*f1)();
    f1 = funcaoVoid;
    f1();
```



```
    int(*f2)(int);
    f2 = funcaoInt;
    f2(222);

    //vetor de pointeiros para funcao.
    void(*f[3])();
    f[0] = funcao1;
    f[1] = funcao2;
    f[2] = funcao3;

    //chama a função indexada no índice i
    for (int i = 0; i < 3; i++)
        f[i]();
}
```

Erros típicos de programação envolvendo Funções (para alunos de programação)

Em diversos problemas é solicitado para que uma função faça uma dada tarefa, como por exemplo, encontrar o menor valor de um vetor. Para facilitar a solução, algumas soluções erroneamente colocam, dentro da função `main()`, um algoritmo para ordenar o vetor, e assim facilitar a implementação da função. Se o enunciado diz para a função realizar uma tarefa, ela é quem deve fazer tudo, e não preprocessar a informação em outras partes do código.

Outro erro comum é quando é solicitado para que a função gere algum valor, que deve ser retornado. Neste caso, o valor deve ser retornado com o comando `return` e não deve ser usado `printf()` dentro da função para imprimir a resposta.

Dentro de uma função não se deve alterar o valor dos dados a serem consultados. Como exemplo, suponha encontrar o maior valor de um vetor. A função que processar o vetor não pode alterar os dados do vetor. Deve somente encontrar o maior e retornar.

Também relativo a funções, existem o problema de quando a função deve ser finalizada. Como exemplo, considere o problema de criar uma função que recebe um vetor com números aleatórios e deve dizer se existem dois valores iguais. Obviamente, o vetor não pode ser ordenado antes de ser passado para a função. O erro mais comum neste caso consiste em, caso no primeiro teste os valores sejam diferentes, a função para e já retorna um valor falso. Neste tipo de solução, somente pode-se dizer que não existem valores iguais após testar todos, ou seja, o `return` false deve ser colocado após o laço de repetição.

Variáveis Globais

Toda variável que não for definida dentro de uma função é considerada global, e pode ser usada em qualquer parte do código. Seu uso não é recomendado, visto que pode tornar o código pouco legível e suceptível a erros, como no seguinte exemplo. Costuma-se usar o prefixo g_ para descrever variáveis globais, como em `int g_cont;`

```
#include <stdio.h>

int global;  //variável global. Deve ser declarada como g_global

void imprime(void)
{
   int global = 20;    //variável local. Tem prioridade sobre a global.
   printf(" %d", global); //imprime o valor 20 da variável local
}
```

```c
int main(void)
{
   global = 10;
   imprime( );
   printf(" %d", global); //imprime o valor 10 da variável global
   return 0;
}
```

Estruturação de Programas em C em Módulos

Programas em linguagem C são divididos em arquivos com extensão .c e .h. Os arquivos .c contém os códigos e os arquivos .h contém os protótipos das funções. Pode-se fazer toda a programação somente em um único arquivo .c, porém, por questões de estruturação, recomenda-se criar arquivos que englobem funções de um mesmo assunto.

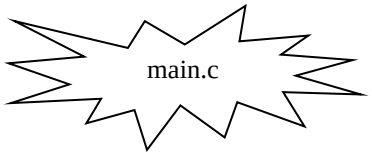
O seguinte exemplo ilustra a definição de uma função. Se essa função vier antes da função main() não é necessário definir seu protótipo, porém se ela vier, é necessário.

```c
#include <stdio.h>

//função imprime antes da main()
void imprime(int i)
{
   printf("%d", i);
}

int main(void)
{
   imprime( 1 );
   return 0;
}
```

```c
#include <stdio.h>

void imprime(int i); //prototipo da funcao
int main(void)
{
   imprime( 1 );
   return 0;
}
//função imprime após a main()
void imprime(int i)
{
   printf("%d", i);
}
```

Porém, por questões de legibilidade (e em casos de referência cruzada onde uma função chama a outra e vice-versa), recomenda-se sempre colocar os protótipos. Para programas grandes, recomenda-se colocar os protótipos sempre em um arquivo .h, como no seguinte exemplo:

```c
#include <stdio.h>

#include "global.h"
#include "modulo_1.h"
#include "modulo_2.h"

//use o prefixo g_ para indicar variavel global.
int g_variavel_da_main = 20;

int main(void)
{
   printf("\n Main: %d", g_variavel_do_modulo_2 );
```



```
    imprime_modulo_1();
    imprime_modulo_2();

    getchar();
    return 0;
}
```

global.h

```
#ifndef ___GLOBAL___H__
#define ___GLOBAL___H__

#define GLOBAL 0

//variaveis globais definidas em main.c e modulo_2.c
//use o prefixo g_ para indicar variavel global.
extern int g_variavel_do_modulo_2;
extern int g_variavel_da_main;

//IMPORTANTE
//Nao é aconselhado a definição de variavel global no arquivo .h pois
//podem haver definicoes multiplas desta variavel em varios arquivos.
//Use extern, como no exemplo de cima.
//int variavel_global = 4; //NAO USE

#endif
```

modulo_1.h

```
#ifndef ___MODULO__1__H__
#define ___MODULO__1__H__

#include "modulo_2.h"
#include "global.h"

void imprime_modulo_1();


#endif
```

modulo_1.c

```
#include "modulo_1.h"
#include <stdio.h>

void imprime_modulo_1()
{
    printf("\nModulo 1. A variavel_do_modulo_2 vale %d", g_variavel_do_modulo_2);
    printf("\nModulo 1. A variavel_da_main vale %d", g_variavel_da_main);
}
```

modulo_2.h

```
#ifndef ___MODULO__2__H__
#define ___MODULO__2__H__

#include "global.h"
#include "modulo_1.h"

void imprime_modulo_2();


#endif
```

modulo_2.c

```
#include "modulo_2.h"
#include <stdio.h>

int g_variavel_do_modulo_2 = 10;
```

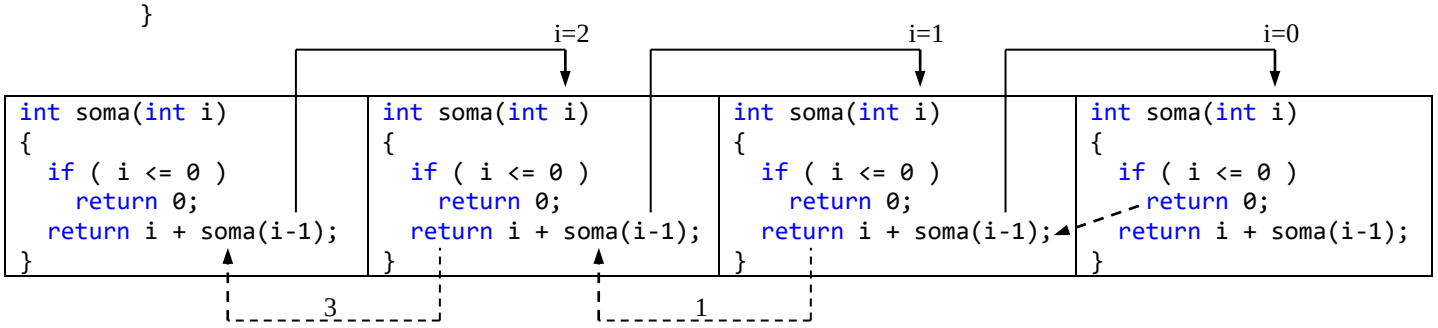
```
void imprime_modulo_2()
{
   g_variavel_da_main = 111;
   printf("\n Eu sou o modulo 2. O define GLOBAL vale %d", GLOBAL);
   printf("\n Eu sou o modulo 2. A variavel_da_main vale %d", g_variavel_da_main);
}
```

Uma das grandes vantagens de C# e Java é a não existência de arquivos de cabeçalhos (arquivos .h). Não é necessário fazer a especificação dos protótipos em um arquivo e a implementação em outro arquivo. Em C# é muito mais fácil fazer um programa com vários arquivos de código fonte.

Recursividade

Recursividade ocorre quando uma função chama ela mesma, direta ou indiretamente. Deve-se tomar cuidado para garantir a parada. No seguinte exemplo, a função `soma()` chama ela mesma até que uma condição de parada seja verificada. A condição de parada é <= 0, dessa forma caso seja passado como argumento -1, a função pára e retorna 0. Deve-se atentar que deve-se obrigatoriamente tratar o retorno quando a função recursiva não tiver retorno void.

```
int soma(int i)
{
   if ( i <= 0 )
      return 0;
   return i + soma(i-1);
}

int main(void)
{
   printf("%d", soma(3));
   return 0;
}
```



O seguinte exemplo apresenta outra forma de fazer a soma recursiva, porém a solução não é tão boa quanto a primeira pois tem uma condição de parada explicita dentro da função recursiva, o que torna a função não genérica.

```
int soma(int i)
{
    if (i == 3)
        return i;
    return i + soma(i + 1);
}
```

Pode-se também ter várias chamada recursivas dentro de uma função. Nesse exemplo, a função recursiva nao tem retorno, o que torna a implementação mais simplificada.

```
void print(int i)
{
    if (i == 10)
        return;
    print(i+1);
    print(i+1);
    printf(" %d ", i);
}
```

Na linguagem C, usando o compilador gcc, um programa conseguiu realizar 65.000 chamadas recursivas em uma função de teste, que nao possuia nenhuma variável local, rodando em Windows 10. Utilizando-se a engine Unity, em linguagem C#, pode-se chamar a função recursiva apenas 11.000 vezes antes de gerar o erro de stack overflow.

Exercícios:

1. Função recursiva para somar apenas os números pares, utilizando-se incrementos de 1 em 1, como no exemplo anterior.
2. Função para o cálculo do fatorial de um número
3. Função int strlen(char *) de forma recursiva
4. Função void strcpy(char *dest, char *src) de forma recursiva
5. Função void char * strcpy(char *src, int cont) recursiva que aloca uma nova string e faz a cópia. A função deve determinar o tamanho da string src.
6. Função para somar os números inteiros positivos de 1 a N enquanto a somatória for menor que 1000.
7. Função para imprimir a sequencia 12345...N...54321.

```
void print(char *s)
{
    if( *s == '\0')
    {
        return;
    }
    printf("%c", *s);

    print( &s[1] ); //passa o endereço do vetor na posição 1.
}

void main(void)
{
    char v[] = "abcde";
    print( v );
}
```

Manipulação de Arquivos Texto

Em diversas aplicações é necessário fazer a leitura de dados de arquivos texto ou binários. Um arquivo texto é caracterizado pela presença unicamente de caracteres imprimíveis, e toda informação é tradada byte a byte individualmente. São exemplos de extensões de arquivos texto o txt, C, HTML, Java, etc.

Antes de fazer a leitura/escrita em um arquivo deve-se abri-lo. Para isso usa-se a função `fopen()`. Após o processamento deve-se fechar o arquivo com a função `fclose()`.

A função fopen() inicia a struct FILE, que tem a seguinte definição:

```
typedef struct _iobuf
{
    char* _ptr;
    int _cnt;
    char* _base;
    int _flag;
    int _file;
    int _charbuf;
    int _bufsiz;
    char* _tmpfname;
}FILE;
```

Para arquivos texto, as funções mais comuns são `fscanf()`, `fgetc()`, `fgets()`, `fprintf()`, `fputs()` e `fputc()`. As funções `fteel()` e `fseek()` também são muito úteis para acessos aleatórios no arquivo. O seguinte programa lê um arquivo texto, um caractere por vez e imprime na tela.

```
#include <stdio.h>

int main(void)
{
   FILE *fp;
   char ch;

   fp = fopen("a.txt", "rt"); //abre arquivo para leitura em texto. Opcoes: r, w, a, t, b, +
   if( fp == NULL )           //"wt" é usado para abrir para gravação em modo texto.
   {
      printf("Erro para abrir arquivo");
      exit(0);
   }
   do
   {
      ch = fgetc(fp); //fputc(char c, fp) eh usado para gravar um caractere no arquivo
                      //fscanf(fp, "%c", &ch); → igual a ch = fgetc(fp);
      printf("%c", ch);
   }while (!feof(fp));
   fclose(fp);
   return 0;
}
```

Outra forma de fazer a mesma operação:

```
   while( (c=fgetc(fp)) != EOF )
   {
      printf("%c", ch);
   }
```

O seguinte programa faz a leitura de um arquivo texto estruturado em 3 colunas, sendo uma de valores inteiros, uma com valores reais e uma com texto.

```
#include <stdio.h>

int main(void)
{
   FILE *fp;
   char ch;
   int id1;
   float id2;
```

```
1   55.890   teste
2  124.44566  ola mundo
55 -0.999    xxxx teste
```

```
    char vet[100];

    fp = fopen("tabela.txt", "rt"); //abre arquivo para leitura (read/text)
    do
    {
       int cont = fscanf(fp, "%d %f %[^\n]s", &id1, &id2, vet );
       if( cont == 3 )
          printf("%d %f %s\n", id1, id2, vet);
    }while (!feof(fp));
    fclose(fp);
    return 0;
}
```

Outra forma de fazer a mesma operação:

```
char linha[200];
do
{
   fgets(linha, 200, fp); //le uma linha inteira do arquivo, incluindo o '\n'
   int cont = sscanf(linha, "%d %f %[^\n]s", &id1, &id2, vet ); //parse na linha
   if( cont == 3 )
      printf("%d %f %s\n", id1, id2, vet);
}while (!feof(fp));
```

Manipulação de Arquivos Binários

Arquivos binários são arquivos que tem uma estrutura mais elaborada e a informação não é tratada unicamente byte a byte, mas também em blocos. Como exemplo, um número inteiro gasta 4 bytes, que devem ser processados de uma única vez. São exemplos arquivos com extensão bmp, jpg, doc, exe, pdf, gif, mp3, mpg, etc.

Para leitura de arquivos binários deve-se saber de antemão o formado dos dados. Neste formado de arquivo, pode-se gravar/ler diretamente tipos básicos ou estruturas complexas. Observe que nao faz sentido armazenar ponteiros em arquivos.

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

struct x{
   int a;
   char c;
   float d;
};

int main(void)
{
   FILE *fp;
   struct x v1, v2, vet[20];
   v1.a = 10;
   v1.c = 'X';
   v1.d = 24.9980;

   fp = fopen("a.bin", "wb"); //escrita em binário (write/binary). Cria o arquivo se não existir
   if( fp == NULL )
   {
      printf("Erro para criar arquivo");
      exit(0);
```

```
    }
    fwrite(&v1, sizeof(struct x), 1, fp);
    //fwrite(vet, sizeof(struct x), 20, fp); → para gravar todo vetor vet de 20 posicoes.
    fclose(fp);

    fp = fopen("a.bin", "rb"); //leitura em binário (read/binary)
    if( fp == NULL )
    {
       printf("Erro para abrir arquivo");
       exit(0);
    }
    fread(&v2, sizeof(struct x), 1, fp);
    printf("%f", v2.d);
    flose(fp);
    return 0;
}
```

sizeof(struct x) = 12 bytes

12 * 20 = 240 bytes

A função fwrite() vai gravar em disco 240 bytes lidos a partir do endereço inicial do vetor vet.

Em arquivos binários, principalmente, constuma-se deslocar o ponteiro fp para um local específico do arquivo para, por exemplo, ler um registro. Para isso utiliza-se a função `fseek()`. Outra função relacionada é o `ftell()`, que diz em que posição do arquivo o ponteiro fp está apontando .

A linguagem C oferece muito mais recursos para manipulação de arquivos binários que a linguagem C#. C# somente permite a gravação em disco de arrays de bytes, ou seja, para gravar uma variável `float` em um arquivo, deve-se inicialmente transformá-la em um array de bytes para então chamar a função que grava o array em disco. O mesmo vale para vetores e estruturas.

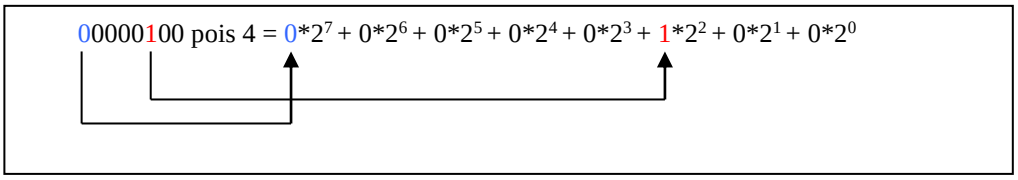
Manipulação de Bits

O computador utiliza uma representação binária para fazer o armazenamento e manipulação dos dados. Dados neste caso podem ser programas armazenados, imagens, sons, textos, vídeos, dentre outros. Qualquer informação ou dado dentro de um computador é representado em números binários (zeros ou uns). Eles são a menor unidade de informação possível de ser representada digitalmente.

Para facilitar o processamento e gerenciamento dos dados, eles são agrupados em bytes (conjunto de 8 bits). Cada byte pode representar um intervalo de 0 a 255, ou seja, pode representar até 256 combinações diferentes de dados, pois $2^8 = 256$. Para melhor compreender a aritmética de números binários, basta fazer todas as combinações possíveis destes 8 bits:

```
00000000 = 0
00000001 = 1
00000010 = 2
00000011 = 3
00000100 = 4
....
11111111 = 255 = 128+64+32+16+8+4+2+1
```

Para compreender a conversão, veja o seguinte exemplo. O número 4 em binário vale



Ou seja, 0 + 0 + 0 + 0 + 0 + 4 + 0 + 0 = 4. Para mostrar que isso não tem mistério nenhum, podemos fazer uma analogia ao sistema decimal e mostrar que o número 2501 é formado pela seguinte expressão:

$2*10^3 + 5*10^2 + 0*10^1 + 1*10^0 = 2000 + 500 + 0 + 1 = 2501.$

Números inteiros positivos menores que 255 podem ser representados com um único byte, como mostrado anteriormente. Caso o número for maior, deve-se utilizar uma quantidade de bytes maior. Com 16 bits, pode-se representar até o número 65.535, com 24 até o número 16.777.216 e com 32 bits (4 bytes) até o número 4.294.967.295.

Por convenção, bytes são representados pela letra B e bits pela letra b. Essa terminologia é muito usada para descrever taxas de transferência de dados, como no caso de redes, modems e dispositivos de armazenamento. Como exemplo, um modem pode trafegar dados a uma taxa de 54kbps, ou seja, 54 kilo bits por segundo. Um HD pode ler dados a uma taxa de 10 MB/s, ou seja, 10 Mega Bytes por segundo. Muitas fontes de informação de baixa credibilidade (Internet, por exemplo) usam B e b como sinônimos. Deve-se também observar que em informática as convenções de grandezas não são iguais às conhecidas habitualmente, como mostrado na seguinte tabela.

Tabela 1: Grandezas utilizadas em Informática

Símbolo	Tamanho	Comentários
Bit (b)	1	$2^0 = 1$. Menor unidade de informação: vale 0 ou 1.
Byte (B)	8 bits	$2^3 = 8$. Por convenção, e por ser potência de 2
Kilo (K)	1024 Bytes	$2^{10} = 1.024$
Mega (M)	1024 Kilo	$2^{20} = 1.048.576$
Giga (G)	1024 Mega	$2^{30} = 1.073.741.824$. Unidade dos HDs atuais
Tera (T)	1024 Giga	$2^{40} = 1.099.511.627.776$
Peta (P)	1024 Tera	$2^{50} = 1.125.899.906.842.624$
Exa (E)	1024 Peta	2^{60}. Talvez vá ser usado pelos seus netos ☺
Zetta (Z)	1024 Exa	2^{70}.
Yotta (Y)	1024 Zetta	2^{80}. É algo muito grande.

1 - Manipulação de Bits em Linguagem C

A ideia de manipular bis é usar o menor espaço para guardar uma informação, como por exemplo, em um sistema de reserva de passagens, indicar se uma poltrona de um ônibus está ocupada ou não. Neste caso, um único bit pode ser suficiente. Sem esse recurso, a unidade mínima de armazenamento em memória é 1 byte, ou seja, um tipo char, que gasta 8 bits.

- Vantagens:
  - Economia de memória
  - Armazenamento de várias informações em uma área de memória atribuída a um tipo básico da linguagem

- Usos:
  - representar conjuntos: um elemento pertence ao conjunto se o respectivo bit for 1
  - Manipulação de cabeçalhos de arquivos binários
  - Manipulação de Imagens
  - Protocolos de rede

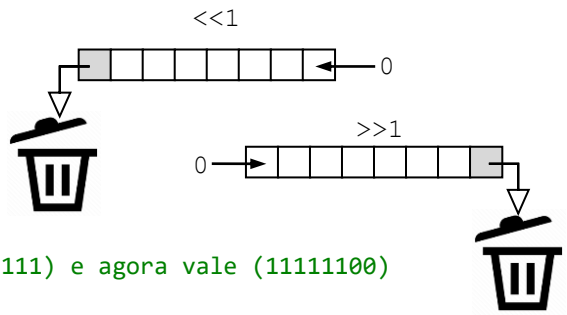
- Multiplicações e divisões inteiras com maior velocidade
- Passagem de parâmetros variados para funções: glutInitDisplayMode(...)

Existem 6 operadores para se trabalhar com bits:

1. Operador >>: Usado para deslocar todos os bits para a direita. Se for deslocado apenas 1 bit, esta operação equivale a uma divisão por 2. Ao se deslocar para a direita, bits 0s são adicionados a esquerda por default.
2. Operador <<: Usado para deslocar todos os bits para a esquerda. Se for deslocado apenas 1 bit, esta operação equivale a uma multiplicação por 2. Ao se deslocar para a esquerda, bits 0s são adicionados a direita por default.
3. Operador |: o operador OR bitwise é usado para fazer operação lógica OR entre bits que estão na mesma posição dentro do tipo primitivo. Não confundir com o operador lógico ||.
4. Operador &: o operador AND bitwise é usado para fazer operação lógica AND entre bits que estão na mesma posição dentro do tipo. Não confundir com o operador lógico &&.
5. Operador ~: usado para negar todos os bits.
6. Operador ^: usado para fazer uma operação de XOR (ou exclusivo).

Seguem alguns exemplos da utilização destes operadores. Será usado o tipo uchar (unsigned char) nos exemplos. Deve-se lembrar que o tipo char tem 8 bits.

```
uchar a = 1; //a vale 1          (00000001)
a = a << 1;  //a passa a valer 2 (00000010)
a = a << 1;  //a passa a valer 4 (00000100)
a <<= 2;     //a passa a valer 16(00010000)
a >>= 1;     // a passa a valer 8
a = a >> 5;  // a passa a valer 0
a = a >> 5;  // a continua valendo 0
a = 255;
a = a << 2;  //a passa a valer 252 pois valia (11111111) e agora vale (11111100)
a = 1;
a = a | 2;  //a passa a valer 3 pois a valia 1 (00000001) e 2 vale (000000010)
a = a & 2;  //a passa a valer 2 pois a valia 3 (00000011) e 2 vale (000000010)
a = 0;
a = ~a;      //a passa a valer 255 (11111111) pois a valia 0 (00000000)
a = a ^ 3;   //a passa a valer 252 (11111100) pois 3 vale (00000011)
```



Em uma operação OU, segue-se a regra: 1 OU 1 = 1, 1 OU 0 = 1, 0 OU 1 = 1 e 0 OU 0 = 0

```
  01010101
  11101001
----------------
  11111101
```

Em uma operação E, segue-se a regra: 1 E 1 = 1, 1 E 0 = 0, 0 E 1 = 0 e 0 E 0 = 0

```
  01010101
  11101001
----------------
  01000001
```

Duas funções são muito úteis para se trabalhar com bits. A função get( ) é usada para ler um único bit, enquanto que a função set( ) é usada para setar um único bit. Estas duas funções podem ser usadas sobre tipos primitivos. Costuma-se utilizar tipos não sinalizados, pois tipos sinalizados utilizam o bit mais significativo para representar o sinal.

Em uma operação para ler um bit, deve-se primeiro criar uma máscara com um único bit valendo 1 na posição desejada. Para isso, basta deslocar o bit 1 N casas para a esquerda. Por exemplo, para gerar a máscara (10000000), pode-se utilizar o seguinte código: 1<<7. Deve-se observar que o bit foi deslocado apenas 7 casas, e não 8. Caso fosse deslocado 8 casas, o valor da máscara seria 0.

```
uchar a = 1;
a = a << 7;  //a passa a valer 128 (100000000)
a = 1
a = a << 8;  //a passa a valer 0 (000000000)
```

Uma vez tendo-se a máscara configurada, para setar um bit 1 basta realizar uma operação OU do valor com a máscara. Para setar um bit 0, deve-se negar a máscara e realizar uma operação de E. Faça experimentos e desenhe num papel para entender o processo.

```
typedef unsigned char uchar;

int get(uchar c, int bit)
{
    uchar mask = 1 << bit;
    c &= mask;
    return (c>0);
}

uchar set(uchar c, int bit, int val)
{
   uchar mask = 1 << bit;
   if( val == 1)
   {
      c |= mask;
   }
   if( val == 0)
   {
      c &= ~mask;
   }
   return c;
}
```

```
int main(void)
{
    uchar c = 0;

    c = set(c, 1, 1); //00000010 = 2
    c = set(c, 4, 1); //00010000 = 16
    //                = 00010010 = 18
    printf("%d %d", get(c, 1), c);

    return 0;
}
```

Como regra, em uma operação de OU, se a máscara for 1, o valor do bit final é 1. Se a máscara for 0, tem-se o valor original. Em uma operação de E, se a máscara for 1, tem-se o valor original, e se a máscara for 0, tem-se 0.

Pode-se utilizar laços de repetição para iterar sobre os bits. No seguinte exemplo, vamos alterar um único bit de v de forma a tornar o valor de v o maior possível. Para isso precisamos primeiro encontrar qual bit mais significativo tem valor 0, para então trocá-lo por 1.

```
uchar v = 20, mask;
for (int i = 7; i >= 0; i--)
{
    mask = (1 << i);
    if ((v & mask) == 0) //achei o bit mais significativo que vale 0
    {
        v |= mask;
        break;
    }
}
```

O seginte exemplo mostra como verificar se um número é uma potência de 2, da forma mais rápida e compacta possível.

```
bool IsPowerOfTwo(ulong x)
{
    if( x == 0 )
       return 0;
    return (x & (x - 1)) == 0;
}
```

2 - Dicas para geração de máscaras

1. Use `~0` para gerar uma máscara com todos os valores 1's. Como se sabe, 0 em decimal corresponde a 00000000. Negando-se todos os bits, temos 11111111.
2. Use `((1<<N)-1)` para gerar máscaras com N bits 1's consecutivos. Isso se explica pelo fato de `(1<<N)` gerar uma máscara com apenas 1 bit 1, ou seja, `1<<4 = 10000`. Esse valor representa 16. Subtraindo-se 1, tem-se o valor 15, que equivale a 1111, ou seja, uma máscara com 4 bits 1, que era a máscara desejada.
3. Use `((v<<N)>>M)` para isolar sequências de K bits nos bits menos significativos. A ideia aqui é "limpar" os bits que não são de interesse. Por exemplo, caso deseja-se saber o valor de uma sequencia de 4 bits centrados em um byte (ex: 10**1111**00), pode-se realizar a seguinte operação para extrair a sequencia 1111, que em decimal corresponde a 15:
    a. empurra-se todos os bits 2 casas para a esquerda. Isso faz com que os dois bits mais significativos (esquerda) sejam removidos. Nesta operação, foram adicionados dois bits 0 a direita. Como resultado, tem-se **1111**0000.
    b. Após, desloca-se todos os bits 4 casas a esquerda (2 que já tinha e as duas da operação anterior). Com isso, tem-se 00001111, que corresponde ao valor decimal 15.

3 - Aplicações: Manipulação de Imagens

Caso uma imagem seja representada por vetor de unsigned int com 32 bits/pixel, ou seja, formato (alpha, R, G, B), conhecido como formato 888, pois cada componente usa 8 bits, o seguinte código é usado para extrair cada componente. Faz-se uso da estratégia 3, apresentada na seção anterior. Há uma pequena modificação na estratégia pois ao invés de deslocar para esquerda e a após para a direita, desloca-se apenas para a direita e após aplica-se uma máscara contendo bits 1s apenas nas posições de interesse, neste caso a máscara 255 = 11111111.

```
unsigned int pixel = v[i];
A = (pixel >> 24) & 255; //nessa linha não é necessário fazer a operação & 255
R = (pixel >> 16) & 255;
G = (pixel >> 8)  & 255;
B = (pixel >> 0)  & 255;  //nessa linha não é necessário fazer >> 0
```

Uma mesma solução poderia ser obtida com o seguinte código

```
unsigned int pixel = v[i];
A = ((pixel << 0)  >> 24);
R = ((pixel << 8)  >> 24);
G = ((pixel << 16) >> 24);
B = ((pixel << 24) >> 24);
```

Caso uma imagem seja representada por vetor de unsigned char com 16 bits/pixel, ou seja, no formato (R, G, B), sendo 5 bits para o R, 6 para o G e 5 bits para o B – formato 565, o seguinte código é usado para extrair cada componente. Faz-se uso da estratégia 3 (criação de máscaras), apresentada na seção anterior. Neste caso, como as componentes RGB não estão no intervalo [0, 255], deve-se aplicar mais um shift para a esquerda para se obter o

resultado final. Como exemplo, se o pixel R vale 00011111=31, o mesmo deve ser transformado em 11111000, que resultaria em um valor 248, que é o valor máximo que a componente pode atingir. Para o caso do G, a sequência 00111111 deve ser mapeada para 11111100, que corresponde ao valor 252.

```
unsigned char v[SIZE];
unsigned int pixel = v[i] | (v[i + 1] << 8);

R = ((pixel >> 0)  & 31) << 3;     //armazenado em formato BGR (a comp. R está mais a direita)
G = ((pixel >> 5)  & 63) << 2;
B = ((pixel >> 11) & 31) << 3;
```

4 - Aplicações: Parâmetro de retorno múltiplo

Uma forma de uma função retornar mais de um valor é colocando-os dentro de um tipo primitivo de forma organizada. No seguinte exemplo, coloca-se as coordenadas do mouse (x, y) em um tipo inteiro, sendo a coordenada x nos 16 bits menos significativos e a coordenada y nos 16 mais significativos. Deve-se tomar cuidado que o valor armazenado, como neste caso, caiba em 16 bits, caso contrário será gerado um "overflow", que irá corromper a informação.

No segundo exemplo, temos a função `glutInitDisplayMode()`, da API OpenGL, que tem como argumento um único inteiro. Ao ser chamada a função, o usuário deve especificar parâmetros como no exemplo. Os parâmetros seguem um padrão de potência de 2 previamente definidos, que para este caso são:

```
#define GLUT_RGB    0   //00000000
#define GLUT_INDEX  1   //00000001
#define GLUT_DOUBLE 2   //00000010
#define GLUT_ACCUM  4   //00000100
#define GLUT_ALPHA  8   //00001000
#define GLUT_DEPTH  16  //00010000
```

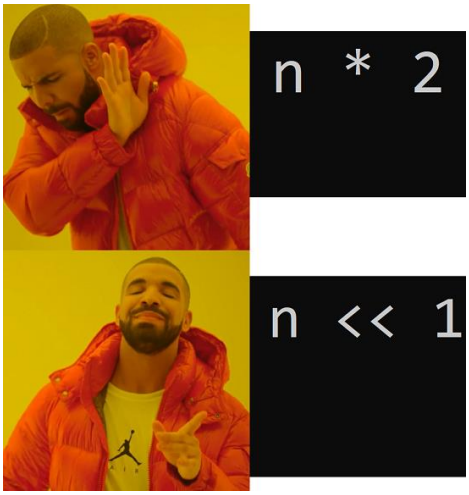
Ou seja, ao se aplicar o operador OU, temos (`GLUT_DOUBLE | GLUT_ACCUM`), que corresponde a (2 | 4) = 6, ou seja, 00000110. Com este tipo estrutura, as funções podem receber um número variável de argumentos.

```
int main(void)
{
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_ACCUM);
}
```

5 - Cálculo do inverso da raiz quadrada de um número real

O seguinte código é uma versão otimizada do cálculo do inverso da raiz quadrada de um número real, que é uma operação típica na normalização de vetores, muito comum em jogos. Para mais detalhes do seu funcionamento, bem como investigação na descoberta de seu criador, consulte http://www.beyond3d.com/content/articles/8/. A análise do funcionamento deste algoritmo foi desenvolvido por Chris Lomont em http://www.lomont.org/Math/Papers/2003/InvSqrt.pdf. Outra dedução da formulação utilizada por ser vista em "*The Mathematics Behind the Fast Inverse Square Root Function Code*", por Charles McEniry (2007). Este código foi divulgado globalmente com a abertura do código fonte do jogo Quake3.

```
float InvSqrt(float x)
{
    float xhalf = 0.5f*x;
    int i = *(int*)&x;
    i = 0x5f3759df - (i >> 1);
    x = *(float*)&i;
    x = x*(1.5f - xhalf*x*x);
    return x;
}
```



6 - Protocolos de rede

Ao se trabalhar com camadas de rede baixas (próximas do hardware), pode ser necessário interpretar a informação que chegou em um pacote de dados. Uma forma de fazer isso (senão a única) é usando operadores de bits, como mostrados nos exemplos anteriores. Seguem exemplos de cabeçalhos de protocolos de rede conhecidos.

- IP / TCP Headers (http://www.networksorcery.com/enp/protocol/tcp.htm)

00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
Version				IHL				TOS								Total length															
Identification																Flags			Fragment offset											
TTL								Protocol								Header checksum															
Source IP address																															
Destination IP address																															
Options and padding :::																															

00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
Source Port																Destination Port															
Sequence Number																															
Acknowledgment Number																															
Data Offset				reserved				ECN			Control Bits					Window															
Checksum																Urgent Pointer															
Options and padding :::																															
Data :::																															

- RTP Headers (http://www.networksorcery.com/enp/protocol/tcp.htm)

00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
Ver		P	X	CC				M	PT							Sequence Number															
Timestamp																															
SSRC																															
CSRC [0..15] :::																															

Referências Bibliográficas

[1] Kernighan, B., Ritchie, D. C, A linguagem de Programação Padrão Ansi. Ed. Campus, 1990.
[2] Schildt, H. C completo e total. Ed. Makron Books, 1991.
[3] Waldemar Celes, Renato Cerqueira, Jose Rangel. Introdução a Estruturas de Dados. Ed. Campus, 2004.

Apêndices

Nesta seção são apresentados 5 programas exemplo que mostram as principais características da linguagem C. No cabeçalho de cada exemplo existe uma descrição dos assuntos apresentados.

Exemplo 1

```
/* TOPICOS
- impressao de mensagens na tela
- leitura de valores inteiros e reais
- definicao de vetores de inteiros e caracteres
- manipulacao de vetores
- passagem de vetores para funções
- lacos de repeticao
*/

int main(void)
{
   int   a;
   float b;
   char  str[20]; /*define um vetor com 19 posicoes uteis */
   int   vet[4];  /*define um vetor de inteiros com 4 posicoes */
   int   cont;

   /* o caractere '\n' faz pular uma linha na tela*/
   printf("\nDigite um numero inteiro e um float, separados por espaco e tecle <enter>:\n");

   /*"%i" indica que a variavel a sera lida como inteira. */
   /*"%f" indica que a variavel a sera lida como float. */
   /* a funcao scanf recebe o endereco das variaveis a serem lidas */
   scanf("%i %f", &a, &b);

   /* para formatar a saida do tipo float com 3 casas decimais, usar 0.3f */
   printf("\nOs numeros foram: \ninteiro: %i \nfloat:   %0.3f\n", a, b);

   /* limpa o buffer de teclado */
   fflush(stdin);

   printf("\nDigite uma string com no maximo 19 caracteres\n");
   gets(str);
   /* %s indica que a variavel str sera lida como uma string */
   printf("\nA string digitada foi: %s\n", str);

   /* limpa o buffer de teclado */
   fflush(stdin);
   printf("\n\nDigite 4 numeros inteiros:   ");
   /*faz a leitura de um vetor com 4 posicoes de inteiro */
   for(cont=0; cont<4; cont++)
   {
      scanf("%d", &vet[cont] );
   }
   /* chama funcao para imprimir o vetor */
   imprime_vetor(vet);
   return 0;
}
```

Exemplo 2

/* TOPICOS
- utilizacao de #defines
- leitura de valores char
- uso do comando switch
*/

```
#define KEY_UP    72
#define KEY_DOWN  80
#define KEY_LEFT  75
#define KEY_RIGHT 77
#define KEY_EXIT  27

#include <stdio.h>
#include <conio.h>

int main(void)
{
   char  opcao; /*define uma variavel char. */

   /* inicializa a variavel opcao */
   opcao = 0;
   printf("\nPressione alguma seta ou ESC para sair: ");
   while(opcao != KEY_EXIT )
   {
      /* limpa o buffer de teclado */
      fflush(stdin);

      /*le uma tecla */
      opcao = getch();
      //printf("%d ", opcao);
      switch(opcao)
      {
         case KEY_UP:
            printf("\n Seta para cima");
         break;

         case KEY_DOWN:
            printf("\n Seta para baixo");
         break;

         case KEY_LEFT:
            printf("\n Seta para esquerda");
         break;

         case KEY_RIGHT:
            printf("\n Seta para direita");
         break;
      }
   }
   return 0;
}
```

Exemplo 3

/* TOPICOS
- uso de funcoes da biblioteca string.h
- representacao numérica de caracteres
- conversao de caracteres para maiúsculo
- manipulação de caracteres dentro de vetores (strings)
*/

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

/* Cada letra tem um numero decimal associado. Para descobrir o valor,
   pode-se usar printf("%d", variavel)*/
void valor_decimal(void)
{
   char letra1='a', letra2='A', letra3='5';
   printf("\nO valor da letra a eh: %d", letra1);
   printf("\nO valor da letra A eh: %d", letra2);
   printf("\nO valor da diferenca eh %d", letra1-letra2);
   printf("\nO valor da letra 5 eh: %d", letra3);
}

void maiusculo(char *str)
{
   int i=0;
   /*enquanto nao encontrar o final da string, troca as letras para maiusculo.*/
   while( str[i] != '\0' )
   {
       str[i] = str[i]-32;
       /* outra solucao eh
       str[i] = str[i]-('a'-'A');*/
       i++;
   }
}

int main(void)
{
   int   a, tam;
   char str[100];

   /*copia a string "teste" em str */
   strcpy(str, "teste");

   /*verifica o tamanho de str */
   tam = strlen(str);
   printf("\n O tamanho de str eh %d", tam);

   /*Varre a string e troca as letras 'e' por 'a'
     Observar a diferenca entre o operador de atribuicao e comparacao*/
   for(a=0; a<tam; a++)
   {
      if(str[a]=='e') /* "==" -> coparacao logica */
      str[a] = 'a'; /* "=" -> atribuicao */
   }

   /*imprime a nova string alterada*/
```

```
   printf("\nA nova string eh %s", str);

   /*adiciona uma nova string ao final de str */
   strcat(str, ".exe");
   printf("\nA nova string eh %s", str);

   /*trunca a string para 2 caracteres colocando o caractere '\0' na posicao 2*/
   str[2]='\0';
   printf("\nA nova string truncada eh %s", str);

   /*imprime o valor decimal de algumas letras. Tambem vale para numeros */
   valor_decimal();

   /*chama a funcao maiusculo passando uma referencia da string.
     Nao precisa colocar o & antes de str pois somente str ja é
     igual ao endereco da primeira posicao do vetor = &str[0].*/
   maiusculo(str);
   printf("\nA nova string maiuscula eh %s", str);
   return 0;
}
```

Exemplo 4

```
/* TOPICOS
- definição de tipos de dados
- passagem de tipos por valor
- passagem de tipos por referência
- passgem de strings.
- indexação de estruturas por ponteiros
- leitura de valores reais
- ponteiros para tipos definidos
*/

#include <stdio.h>

/*como a definicao da estrutura foi feita fora da funcao main, pode ser acessada
  de qualquer parte do programa */
typedef struct {
   float x;
   float y;
}ponto2D;

/*funcao que recebe uma referência de uma variavel do tipo ponto2D
  Como a passagem eh por referencia, qualquer alteracao nesta funcao
  ira alterar o valor de p1 na função main. p eh um ponteiro  */
void altera_referencia(ponto2D *p)
{
   p->x = 10;
   p->y = 10;
}

/*neste caso, p eh possado por valor. Para alterar p1 de main deve-se retornar o novo valor */
ponto2D altera_valor(ponto2D p)
{
   p.x = 1000;
   p.y = 1000;
   return p;
```

```
}

void imprime(ponto2D p, char *msg)
{
   /* o argumento %0.2f indica que a variavel sera impressa com duas casas decimais */
   printf("%s = (%.2f, %.2f)\n", msg, p.x, p.y);
}

int main (void)
{
   /*definicao de p1 do tipo struct ponto2D*/
   ponto2D p1;

   printf("Digite as coordenadas do ponto(x y): ");

   /* na funcao scanf, deve-se passar o endereco "&" das variaveis que serao lidas */
   scanf("%f %f", &p1.x, &p1.y);

   /* chama a funcao imprime passando uma copia de p1 e uma mensagem*/
   imprime(p1, "O ponto lido foi ");
   altera_referencia(&p1);  /* eh passado o endereco da varival p1 */
   imprime(p1, "O ponto alterado eh ");
   p1 = altera_valor(p1);   /* eh passado uma copia da varival p1 */
   imprime(p1, "O ponto alterado eh ");

   return 0;
}
```

Exemplo 5

/* TOPICOS
- definição de tipos de dados
- alocacao dinâmica
- ponteiros para tipos definidos
- vetores de ponteiros para estruturas
- funções matemáticas
*/

```
#include <stdio.h>   /*funcao printf() */
#include <math.h>    /*funcao seno e rand() */
#include <stdlib.h>  /*funcao malloc() */

#define NUM_PONTOS 8

/*como a definicao da estrutura foi feita fora da funcao main, pode ser acessada
  de qualquer parte do programa */
typedef struct {
   float x;
   float y;
}ponto2D;

void imprime(ponto2D *p, int indice)
{
   /* o argumento %0.2f indica que a variavel sera impressa com duas casas decimais */
   printf("\n Ponto %d = (%.2f, %.2f)\n", indice, p->x, p->y);
}

int main (void)
```

```
{
   int i;

   /*definicao de um vetor de ponteiros para ponto2D*/
   ponto2D *p1[NUM_PONTOS];

   /*agora deve-se fazer a alocacao dos pontos */
   for(i=0; i< NUM_PONTOS; i++)
   {
      p1[i] = (ponto2D*)malloc(sizeof(ponto2D));
   }

   /*faz a inicializacao dos pontos com valores randomicos */
   for(i=0; i< NUM_PONTOS; i++)
   {
      p1[i]->x = (float)sin(i);
      p1[i]->y = (float)(rand()%100); /* valor randomico entre 0 e 100 */
   }
   /*faz a impressao dos pontos */
   for(i=0; i< NUM_PONTOS; i++)
   {
      imprime(p1[i], i);
   }
   return 0;
}
```