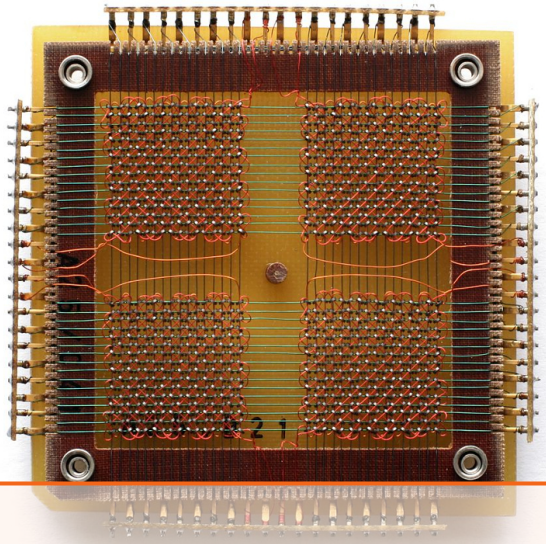
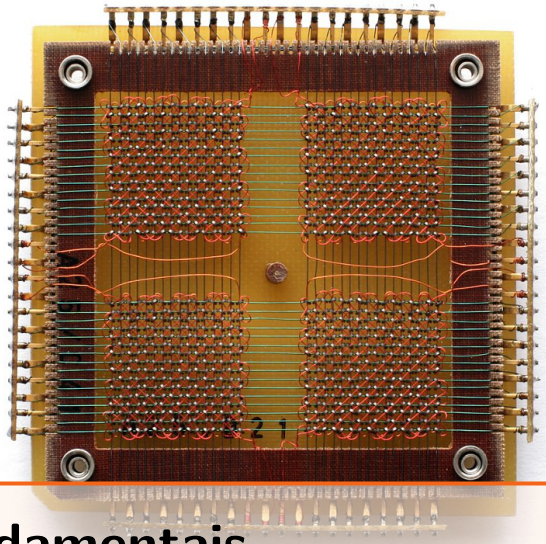


Sumário



1. Conceitos fundamentais

Antes de começarmos a falar sobre C devemos nos perguntar: o que é o C?

C é uma linguagem de programação, cara!

Ok, ok. Você está certo. Mas entender o que é uma linguagem de programação é o primeiro passo. Se você já sabe o que é uma, não tenha pressa. Quem sabe você aprenda algo novo ou melhore seu conceito. Ou quem sabe contribua com este livro e melhore esta seção.

Bem, uma linguagem de programação é uma forma estruturada (e organizada) criada com a intenção de comunicar a uma máquina que ela deve realizar certas instruções e comandos. Essa máquina, não por acaso, é um computador (pelo menos no nosso caso).

A forma que estas instruções e comandos são ordenados irá alterar o que o computador entenderá e consequentemente o que ele executará. Este passo-a-passo é conhecido como *algoritmo*.

Porém, como nem tudo são flores, existem várias maneiras de se passar estas instruções para o computador. Provavelmente não existe limite para o desejo humano de criar coisas novas; e assim uma nova linguagem é criada com mais frequência do que se ganha na loteria. Até o Google já estudou a possibilidade de criar a sua!

E o que eu ganho com isso? Só mais uma linguagem para aprender?

Bom..você não precisa aprender todas, correto? A não ser que realmente queira, o que não recomendo!

Então é preciso escolher a linguagem (ou algumas delas) que atenda esse seu desejo ”estranho”de programar. Como este é um livro sobre linguagem C, não preciso nem dizer que você começou muito bem, preciso? ;)

A seguir, vamos falar um pouco mais sobre algumas categorias de linguagens e suas diferenças.

1.1 Linguagens de programação

1.1.1 Assembly

É considerada uma linguagem de baixo nível que se aproxima bastante do código executado máquina. Suas instruções são o que há de mais próximo da linguagem da máquina mas ainda assim legíveis pelos humanos (a não ser que você seja o Neo da Matrix ler o código da máquina não vai ser problema). Suas instruções estão intimamente relacionadas com algo chamado de "Conjunto de instruções"(Instruction Set Architecture, em inglês).

1.1.2 CLI

Apesar de não serem comumente encaradas como linguagens de programação, as "linguagens de linha de comando"(Command Line Interface¡ em inglês) são linguagens bastante utilizadas. Quer apostar?

Quem já trabalhou com sistemas operacionais Unix, Linux e Windows com certeza já viu estas linguagens: Windows batch (aquela tela preta do command/cmd), Windows PowerShell (administradores de Windows Server conhecem bem), bash, sh, zsh, ksh (todas presente em sistemas Unix e Linux).

1.1.3 Linguagens compiladas

É aqui que o C cai! Uma linguagem compilada! "Compilada? Explica melhor.". Bom, elas passam por compiladores. "Não me diga!". Teoricamente, qualquer linguagem pode ser do tipo compilada (ou interpretada). A diferença é que uma linguagem que está sendo compilada é traduzida do seu código fonte para instruções de máquina quase que diretamente. E este é um assunto polêmico.

A princípio, linguagens de baixo nível (mais próximas da máquina) são compiladas, pois se busca uma maior "eficiência". Como o código é traduzido completamente antes de ser exposto à máquina, não há necessidade de traduzir parte do código no momento execução. Por outro lado, se desejarmos executar este código em uma máquina que possui uma arquitetura diferente, teremos que realizar uma nova tradução (no nosso caso, uma nova compilação).

Exemplos de linguagens compiladas: C, ALGOL, BASIC, C, C++, C Sharp, Delphi, Erlang, Go, Haskell, Objective-C, Smalltalk, etc.

1.1.4 Linguagens interpretadas

Eis uns dos tipos mais conhecidos. Neste caso, a linguagem é interpretada a partir do seu código fonte por um agente ("a gente"não, agente) externo: o interpretador. Este agente lê o código fonte e, dependendo da arquitetura de computador no qual ele reside, faz uma tradução de um jeito ou de outro.

A grande vantagem aqui é a "portabilidade". Seu código fonte executa, teoricamente, da mesma forma em máquinas diferentes. É uma vantagem muito boa, desde que você tenha um interpretador para aquela arquitetura. A desvantagem aqui é que temos uma camada "extra"de execução: o interpretador. Em ambientes de tamanho reduzido e performance limitada, isto pode ser um problema.

Assim como mencionado anteriormente, nada impede que uma linguagem compilada tenha que passar por um interpretador antes e vice-versa.

> *"Linguagens compiladas e interpretadas são termos que separam apenas as linguagens comumente compiladas das que são comumente interpretadas"*

1.2 Sistemas operacionais

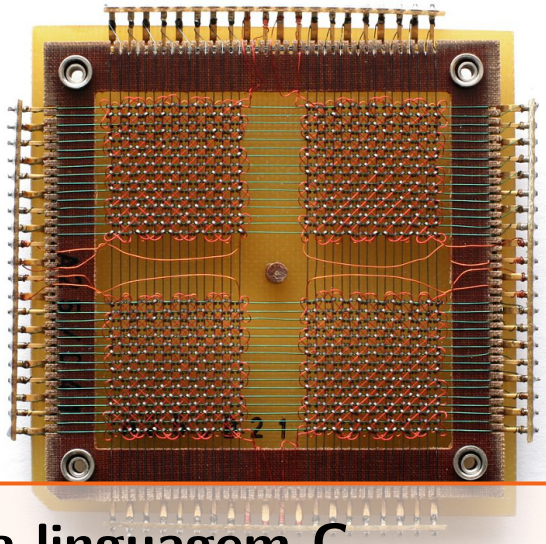
Linux.

1.3 O que são compiladores

Compiladores.

1.4 Ambientes de desenvolvimento

Ambientes de desenvolvimento.



2. Visão Geral da linguagem C

blz

2.1 Origens da linguagem

blz

- oi

2.2 Porque aprender C?

blz

2.3 Onde C é usado?

blz

2.4 Estrutura de um programa em C

blz

2.5 Meu primeiro programa em C

blz

2.6 Compilando meu programa

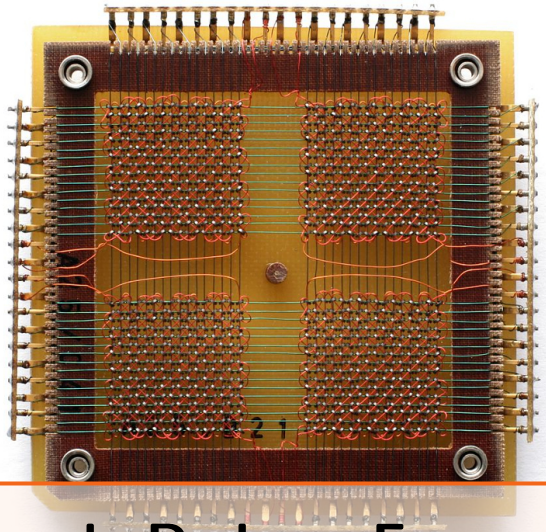
blz

2.7 Executando

blz

2.8 Entendendo meu programa

blz



3. Variaveis, Tipos de Dados e Expressões Arit

Conceitos

- item

3.1 Tipo de dados básicos

3.2 Variáveis

3.3 Modificadores

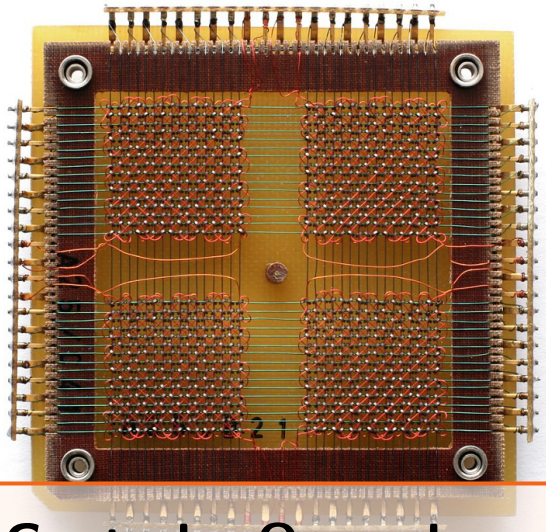
3.4 Constantes

3.5 Expressões Aritméticas

3.5.1 Operação de resto (%)

3.6 Conversão de tipo de dados

Conversao.



4. Estruturas de Controle, Operadores Logicos

Conceitos

- item

4.1 Operadores relacionais

4.2 Operadores lógicos

4.3 O comando if

4.3.1 O comando else

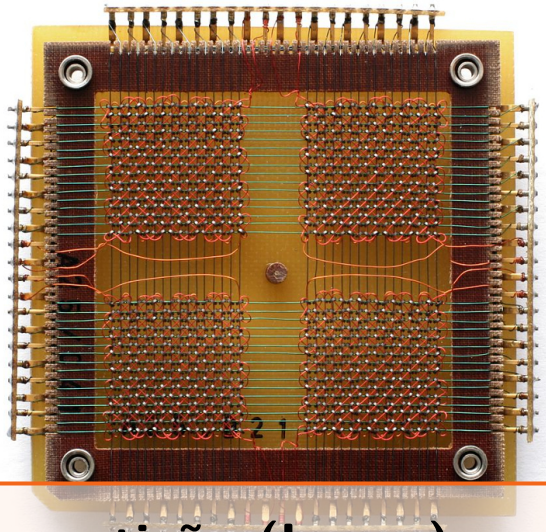
4.3.2 O comando if-else-if

4.3.3 Ifs aninhados

4.4 O comando switch

4.5 Operador ternário(?)

Conversao.



5. Estruturas de repetição (Loops)

Conceitos

- item

5.1 Comando for

5.2 Comando while

5.3 Comando do-while

5.4 Controle de loops

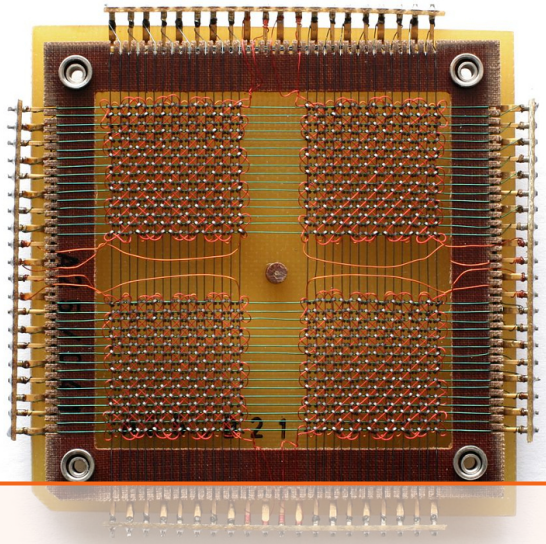
5.4.1 O comando break

5.4.2 O comando continue

5.5 Loops aninhados

5.6 Loops infinitos

Conversao.



6. Arrays

Quem nunca precisou organizar dados de tipo semelhante num único local? Eu sim, você não? Então quando você, por exemplo, escreve um texto num pedaço de papel, o que acha que está fazendo? "*Organizando ideias?*". Sim, também! Contudo, o que você está organizando materialmente são caracteres.

Imagine agora que, usando o mesmo texto do papel no computador, para cada caractere organizado tenhamos que definir uma variável do tipo *char* e atribuir-lhe o valor respectivo.

```
char c1, c2, c3, c4, c5, c6, c7, c8, c9, c10,
     c11, c12, c13, c14, c15, c16, c17;

c1 = 'A';
c2 = 'u';
c3 = 't';
c4 = 'h';
c5 = 'o';
c6 = 'r';
c7 = ' ';
...
```

Tal exemplo é deveras trabalho do capiroto, e só será utilizado na vida real se o programador tiver algum pacto com esse tinhoso. Para os que seguem o caminho da luz e dos ensinamentos jedi há uma alternativa feliz.

Arrays, em C, são tipos de dados de alocação contíngua derivados dos tipos simples. Em outras palavras, são estruturas de dados homogêneas, compostas de elementos de mesmo tipo, e posicionados em sequência na memória. Arrays também são conhecidos como *vetores*, quando a sequência dos dados é unidimensional (linear), e como *matrizes*, quando essa sequência se dá multimensionalmente (linhas e colunas). Divagaremos a respeito dos arrays do tipo matriz mais a frente, abordando, por enquanto, o array do tipo vetor. Seguem dois exemplos de arrays.

Smug Author!											
S	m	u	g		A	u	t	h	o	r	!

Nesse primeiro, a frase "Smug Author!" é a organização de doze caracteres um atrás do outro. Isso mesmo, doze, pois o espaço também é um dado. Nunca se esqueça que, quando tratamos dados, até um "0" ocupa espaço na memória do computador.

Abaixo outra organização de dados. Desta vez de inteiros que serão o resultado do próximo sorteio da mega-sena. Duvida?

9	20	11	7	43	31

Obs.: Em algumas fontes esparsas em língua portuguesa, você, leitor, encontrará referências a arrays como sendo ***arranjos***. Ressaltamos que um array, numa tradução mais fiel, é um ***conjunto*** de dados. Já um arranjo:

Figura 6.1: Array não é Arranjo

6.1 Usando Arrays

6.1.1 Definindo Arrays

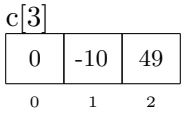
Ao definirmos um array, precisamos determinar a respectiva quantidade de elementos.

```
int i[6];
char c[3];
```

Os colchetes, *operadores* de índice [], informam ao compilador que as variáveis **i** e **c** são do tipo array. Já o número presente entre esses operadores define a quantidade de elementos a serem alocados para o array em questão. No código acima, **i** é definido como um array de *int* com seis elementos e **c** como um array de *char* com três elementos.

i[6]

-57	3	791431480	0	32767	0
0	1	2	3	4	5



Uma boa prática de programação é inicializar os arrays, assim como os tipos simples, pois, tanto em i[] quanto em c[], vemos claramente que os elementos têm valores aleatórios. Isso se dá pelo lixo encontrado na área da memória usada por esses recém-definidos arrays num escopo local.

Arrays também podem ser definidos com a quantidade de elementos informada por outra variável, quando em escopo local. Mas atente sempre para inicializar a variável que é responsável pela quantidade de elementos acima de zero, do contrário comportamentos indesejados acontecerão, tais como pipocas voando na tela ou suco de tangerina escorrendo pelos cantos do monitor.

Inicialize sempre a variável com a quantidade de elementos.

```
int elements = 10;
...
int i[elements];
```

Nunca se esqueça de inicializar! NUNCA! Ou o bicho vem pegar você...

6.1.2 Declarando Arrays

Faz bem relembrarmos que declarar não é o mesmo que definir, portanto, ao contrário do que se pensa comumente, arrays podem normalmente ser declarados. Para tanto, basta fazermos uso do modificador *extern* já visto no item 3.3.

Forçando a quantidade de elementos **ext1[32]** ou não **ext2[]**.

array.h - Declaração

```
extern int ext1[32];
extern int ext2[];
```

E posteriormente definindo os mesmos arrays.

main.c - Definição

```
#include "array.h"
...
int ext1[32];
int ext2[32];
...
```

6.1.3 Inicializando Arrays

Recordar é viver, então viva mais uma vez: "inicializar é atribuir valor no ato da declaração". No caso de arrays podemos inicializá-los de várias formas, vejamos.

Inicializando os elementos um a um.

```
int arr[3] = {1, 5, 10};
```

Inicializando os elementos um a um e forçando o compilador a reservar a quantidade de elementos informados na inicialização. Neste caso, o array é declarado e inicializado com 4 elementos.

```
int arr[] = {3, 2, 9, 1};
```

Inicializando todos os elementos com zero.

```
int arr[3] = {0};
```

Inicializando todos os elementos com zero (modo "preguiçoso suvina", economiza um byte no código fonte por inicialização).

```
int arr[3] = {};
```

Inicializando o primeiro elemento com 5 e os demais com zero.

```
int arr[3] = {5};
```

Quando um array é definido num escopo global *e/ou* com a classe de armazenamento *static* e não há inicialização explícita, de acordo com o *C Standard*, todos os elementos são inicializados com zero.

Inicializando todos os elementos com zero por definiçao estática.

```
static int array[6];
```

Inicializando por escopo global.

```
int arr[6];  /* Os elementos de arr iniciam
                com zero, automaticamente.
                "static" subentendido */

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

Há, também, um array especial chamado array flexível. Ele só pode ser usado como último membro de uma estrutura que já contém pelo menos outro membro. Esse array se destina ao uso em estruturas com tamanho dinâmico que precisam ser compartilhadas integralmente e/ou acessadas linearmente.

Veja abaixo exemplo de uso de um array flexível.

```
struct str {
  int body_len;
  double body[];
};

int length = sizeof(double [10]);

struct str *s = malloc(sizeof(struct str) + length);
s->body_len = length;
...
```

E há o array "sou brasileiro, não desisto nunca" que, tal qual o mote, não tem sentido na vida real. Esse array aloca apenas um elemento.

```
int brazilian_i[] = {1};
float brazilian_f[1] = {2};
```

6.1.4 Acessando Arrays

Os elementos dos arrays em C são acessados através do índice 0 até a quantidade de elementos menos um (n-1), ou seja, num array com três elementos podemos acessá-los da seguinte maneira.

```
float att[3] = {10.5, 25.3, 9.0};

printf("%f\n", att[0]);
printf("%f\n", att[1]);
printf("%f\n", att[2]);
```

O código acima imprime na tela os valores dos elementos att[0] que é 10.5, att[1] com 25.3, e o att[2] respectivamente 9.0. O array como qualquer variável se encontra em locais específicos da memória. Antes do primeiro e logo após seu último elemento há dados referentes a outros controles ou estruturas do próprio programa, por isso, acessá-lo, inadvertidamente, além da área definida na declaração/inicialização ocasionará seguramente comportamentos inesperados no software (bugs), até porque o compilador não o avisará de plano sobre limites extrapolados se você não estiver utilizando flags de warning (-Warray-bounds, neste caso). Portanto, atenção redobrada e vergonha na cara.

Atribuindo Valores a Arrays

Até então vimos apenas como informar ao compilador o quanto de espaço precisamos que ele reserve na memória para nossos dados e como atribuir valores através da inicialização. *Vamos bagunçar o coreto, então?*

"Pensei que não iria falar nada sobre atribuição..."

Foi mesmo, bateu uma preguiça... /o\ A verdade é que discorrer brevemente o tema índices, também conhecido como subscritores, foi crucial para mostrar a você, aspirante a programador C, que nesta bela linguagem, assim como em grande parte das demais, o índice inicial de conjunto de dados é o 0 e não o 1. Sem esse conhecimento básico, talvez você já tivesse desistido de C e iniciado a programar em Pascal.

"Mais falatório e não explicou como atribuo valores num array!"

Nervosinho, você, hein? Calma que é simples.

```
float att[3];

att[0] = 1.1;
att[1] = 1.2;
att[2] = 1.3;
```

Pronto! Esse último código não inicializa o array deixando essa função para a atribuição singular de cada elemento. Doeu tanto esperar?

6.2 Array de Caracteres

Este tipo de array costuma causar arrepios, mas ele, na realidade, não tem nada de espetacular.

"E que arrepios são esses?"

Hmmm... Aquele arrepio de medo de coisas pipocarem na memória e possivelmente no fluxo do programa. Mas calma, antes de você, ávido e inquieto leitor, perguntar que pipocos são esses, respondo-lhe: portais do submundo por onde *hackers* do mal vêm à superfície calma do seu mundo virtual, caro leitor. Mas isso é assunto para outro capítulo. Continuemos.

"Maas..."

Eu já disse rapá! Deixa de ser teimoso. Nunca ouviu falar que o apressado come cru? Vôte!

Posso?

"..."

Vejamos. Um caractere tem uma representação visual cujo valor real está atrelado a um standard de caracteres. O mais comum desses standards é o ***ASCII*** - *American Standard Code for Information Interchange*.

Um caractere na liguagem C deve ser representado entre duas aspas simples e é chamado de *one integer character constant*.

```
char c = 'A';
```

Acima vemos que *c* é uma variável do tipo *char* com o valor *A*, que não é nada mais que:

```
char c = 65;
```

Há também o *multibyte integer character constant*.

```
int micc = 'Hi';
```

Cá entre nós, se não for realmente necessário, evite o uso dele uma vez que seu valor dependerá da implementação definida (ambiente).

"Ok, anotado. Mas decimal sessenta e cinco? Como isso?"

Eu também ficava confuso nos meus primórdios informáticos, mas basta você lembrar que, num computador, tudo são bits, estados 0 ou 1, que por sua vez, aglutinam-se matematicamente e corroboram, numa concepção mais chula, com a afirmação de que todos os dados são números. A esse respeito reveja os primeiros capítulos, se houver alguma dúvida.

"-.-"

Esse valor 65 é o correspondente decimal ao caractere ASCII *A*. Já a conversão desse valor para a imagem bonita que conhecemos do caractere *A* se dá pelo processo de renderização por hardware/software o qual se vale da própria tabela do standard envolvido. Mas voltemos ao C pois um só caractere ASCII não faz palavra (em outros standards um só caractere poderá sim significar palavra).

Conseguiremos esmiuçar melhor o que é um array de caracteres com o exemplo abaixo.

```
char word1[] = {'A', 'u', 't', 'h', 'o', 'r', '\0'};
```

"Hmm... Eu só não entendi a barra invertida e o zero como último elemento do array."

Esse elemento é necessário para construção de *strings*. Em C não há um tipo para strings como em outras linguagens, isso se dá por conta da proximidade que C tem do hardware e pelo fato de na memória uma string não ser nada mais que uma sequência de números que representam singularmente um caractere em particular (ou parte dele em standards mais modernos). Assim esse caractere é conhecido como o caractere NUL (de valor 0) na tabela ASCII e significa, para qualquer código que interpreta strings nesse padrão, o fim do texto a ser tratado.

O código acima poderia ser reescrito da seguinte forma sem prejuízo da funcionalidade.

```
char word1[] = {'A', 'u', 't', 'h', 'o', 'r', 0};
```

Você verá em alguns códigos o uso de *NUL* em vez do zero, mas não se engane assim como eu me enganei por muito tempo, NUL não é uma keyword da linguagem C, e sim apenas uma implementação acessória.

```
#define NUL '\0'
/* or */
#define NUL 0
...
char word1[] = {'A', 'u', 't', 'h', 'o', 'r', NUL};
```

"Pelo que percebi, em C, usar um texto, melhor dizendo, uma string, é um baita pé no saco!"

Sim se o leitor adotar o método acima como regra. Em realidade, há outra forma mais inteligente de se trabalhar com strings.

```
char word2[] = "Smug Author";
```

Basta inicializar o array com as famosas strings literais usando aspas duplas e não

mais a sintaxe de inicialização de elementos de arrays. E ainda dá para usar concatenação ou mais de uma linha, se for necessário.

```
char word2[] = "Smug " "Author";
```

```
char word2[] = "Smug "
               "Author";
```

```
char word2[] =
"Smug \
Author";
```

É saudável aconselhá-lo a nunca, NUNCA MESMO, definir o tamanho de um array quando for usá-lo como string. Deixe esse serviço para o compilador, usando apenas os colchetes vazios [], pois ele calculará com exatidão o tamanho a ser reservado na memória. Para mais sobre o tema, consulte a regra do CERT ARR02-C.

E aí o que achou?

"Gostei, mas você se esqueceu, nesses últimos exemplos, do caractere que finaliza a string. O zero."

Olha que leitor astuto! Está de parabéns, contudo enganado. E a culpa é minha, devo assumir, de tê-lo induzido até essa armadilha.

Há dois aspectos interessantes quando utilizamos strings literais:

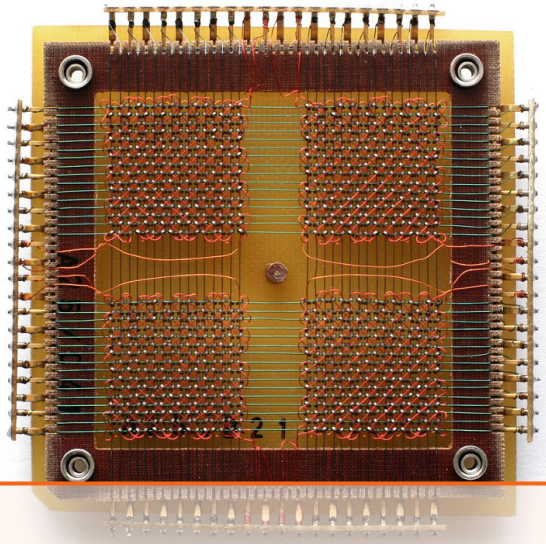
1. O próprio compilador armazenará corretamente o caractere NUL após a string, e isso pode causar dores de cabeça aos desavisados que reservarem buffers menores do que o necessário.
2. Se estivermos usando *ponteiros* o texto passa a ser armazenado no segmento *read-only-data*, portanto, dependendo da arquitetura, não poderá ser alterado.

Sem desmerecer o primeiro aspecto, ressalto que o segundo é bem capcioso, uma vez que algumas arquiteturas podem não suportar segmentos *read-only*, ou seja, permitirão a alteração dessa região da memória mesmo inicializada através de ponteiro. Em suma, mais um comportamento indefinido da linguagem.

"Curioso... E o que são esses ponteiros mesmo?"

Calma jovem padawan, o capiroto o aguarda faz tempo e não se reclama disso.

"Vôte!"



7. Ponteiros

7.1 Armazenamento Primário

Em um computador atual (2014), existem três tipos principais de armazenamento primário: **registradores do processador**; **cache do processador**, e; **memória principal**.

Registradores são pequenos locais de armazenamento, com tamanho estático, contidos no processador. Por fazerem parte do *ISA (Instruction Set Architecture)*, variam com a arquitetura (x86, x86_64, MIPS etc). Um exemplo de registradores de uso geral da arquitetura x86_64 é: *rax, rbx, rcx, rdx.*

A cache do processador é um contêiner de dados intermediário e de acesso aleatório com maior capacidade que os registradores. Ela se situa entre a memória principal e o próprio processador com o intuito de diminuir o tempo médio de acesso às informações; podendo ser subdividida em cache de instruções (lida apenas com a leitura da memória), cache de dados (trata da leitura e escrita da memória) e *TLB - Translation lookaside buffer* (com a finalidade de agilizar a tradução da memória virtual x física).

O termo "Memória Principal" é comumente usado como referência à memória RAM *(Random Access Memory)*, uma vez que nesta reside a grande porção de dados utilizados antes e/ou após o processamento. Essa referência se dá, também, pela transparência que os registradores e a cache do processador tem em comparação à memória principal, tendo em vista a utilizaçao direta dos dados contidos nela quando no desenvolvimento de software.

Há alguns detalhes importantes a serem ressaltados a respeito desses três repositórios primários de dados.

1. A memória principal (RAM), distintamente dos registradores e da cache, não se encontra no processador mas sim conectada a ele através do barramento de memória que, por sua vez, subdivide-se em dois: barramento de endereço e barramento de dados [Figura 7.1].
2. O gerenciamento da memória cache, mesmo que possível através de instruções como INVD e WBINVD na arquitetura x86, é e deve, por segurança, ser deixado

ao encargo do próprio processador, salvo em casos realmente justificáveis.

3. Os dados de todos eles são obtidos por endereçamento, ou seja, por indexação, mas o que os torna diferentes, à visão do programador, é o fato de os registradores e a cache só serem acessados dessa forma pelo microcódigo da arquitetura, enquanto a memória principal pode o ser por endereçamento via código de máquina (programa residente na própria memória).

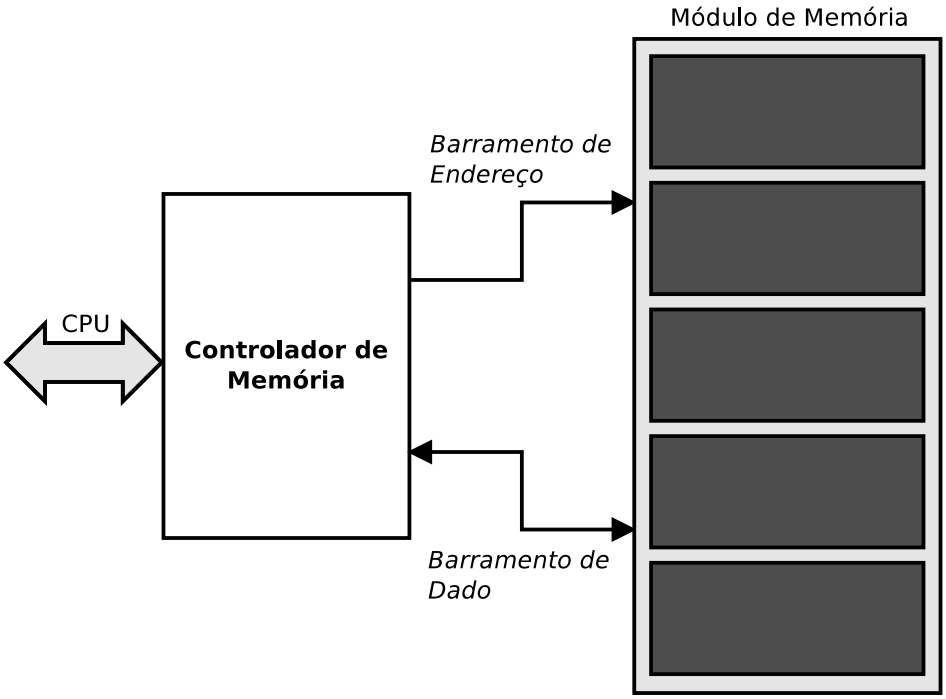


Figura 7.1: Barramentos de Endereço e de Dados

7.1.1 Memória Principal

Sobre a memória principal.

7.2 Usando Ponteiros

Como usar ponteiros.

7.3 Arrays NÃO são Ponteiros

Arrays não são ponteiros.