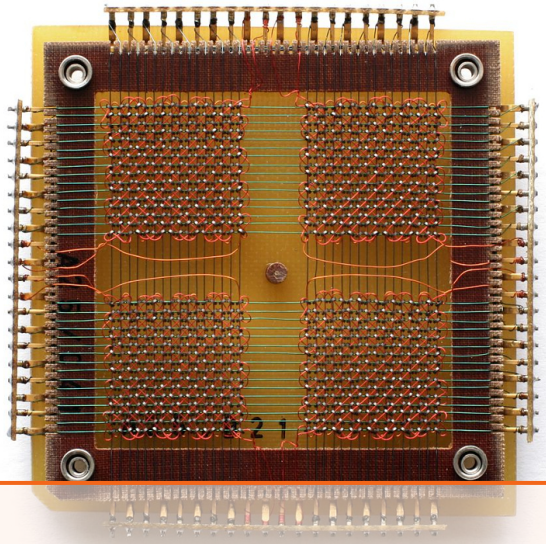
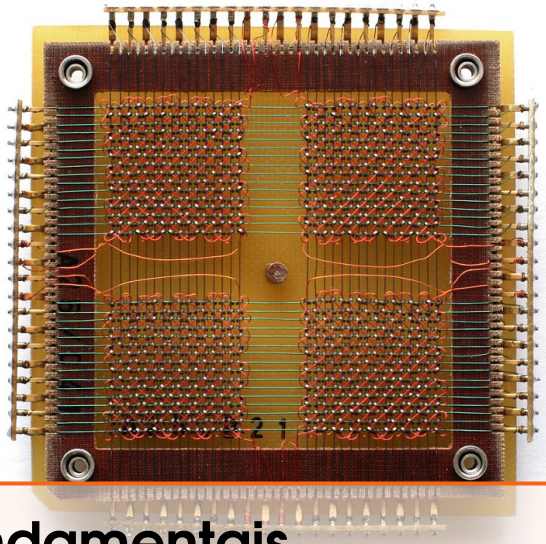


Sumário



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

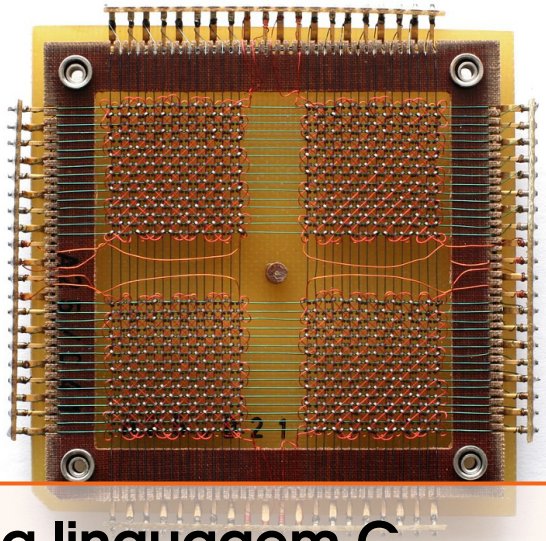
Linux.

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

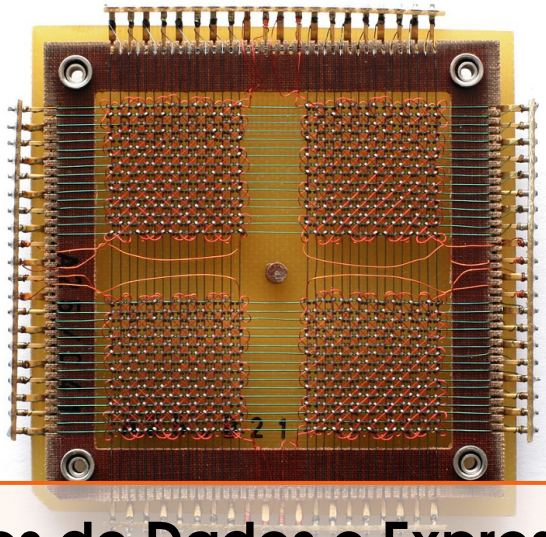
blz

2.7 Executando

blz

2.8 Entendendo meu programa

blz



3. Variaveis, Tipos de Dados e Expressões Ar

Conceitos

- item

3.1 Tipo de dados básicos

3.2 Variáveis

3.3 Modificadores

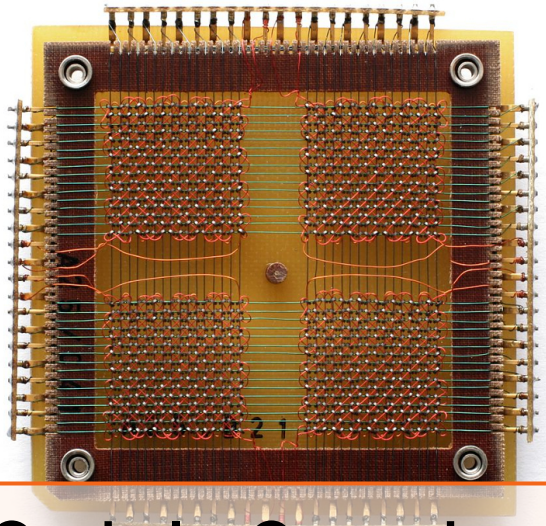
3.4 Constantes

3.5 Expressões Aritméticas

3.5.1 Operação de resto (%)

3.6 Conversão de tipo de dados

Conversao.



4. Estruturas de Controle, Operadores Logico

Conceitos

- item

4.1 Operadores relacionais

4.2 Operadores lógicos

4.3 O comando if

4.3.1 O comando else

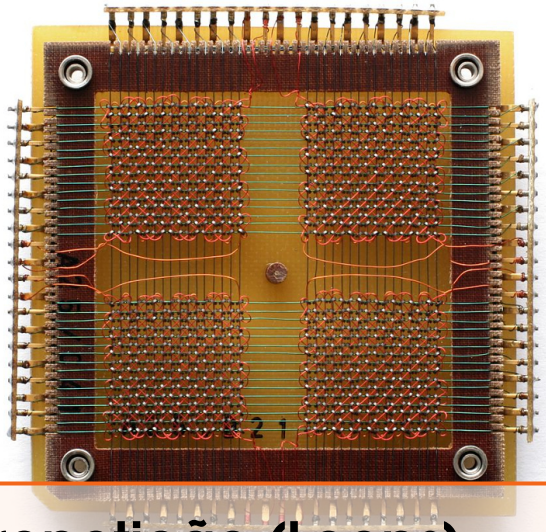
4.3.2 O comando if-else-if

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

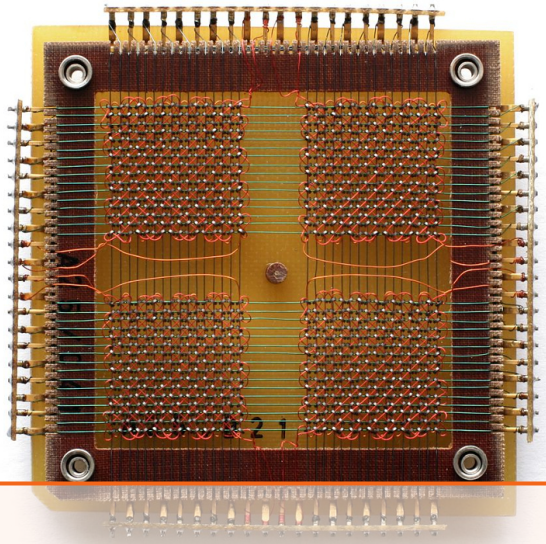
5.4.1 O comando break

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, o que acha que está fazendo? "*Organizando ideias?*". Sim, também! Contudo, o que você está organizando materialmente são caracteres.

Imagine agora termos que, para cada caractere organizado, 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';
c1 = 'u';
c1 = 't';
c1 = 'o';
c1 = 'r';
c1 = ' ';
...
```

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. Divagaremos a respeito dos arrays do tipo matriz mais a frente, por enquanto a abordagem se dará com o conceito de array do tipo vetor. Abaixo, dois exemplos de arrays.

Autor presunçoso!																
A	u	t	o	r		p	r	e	s	u	n	ç	o	s	o	!

A frase “Autor presunçoso!” é a organização de dezessete caracteres um atrás do outro. Isso mesmo, dezessete, pois o espaço também é um dado. Nunca se esqueça que, quando tratamos dados, até um “0” ocupa espaço na memória do computador.

Segue 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 de idioma português, 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 Trabalhando com Arrays

6.1.1 Definindo Arrays

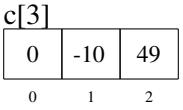
Ao definirmos um array é necessário determinarmos o respectivo tamanho.

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

O uso do *operador* de índice **[]** é que informa ao compilador que as variáveis **i** e **c** são do tipo array. Já o número entre os colchetes 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** 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 contêm valores aleatórios. Isso se dá pelo lixo encontrado na área da memória usada por esses recém-definidos arrays num escopo local.

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

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

main.c

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

6.1.3 Inicializando Arrays

Recordar é viver, então cito-lhes o já discursado noutra parte desta obra: "inicializar é atribuir valor no ato da declaração". No caso de arrays podemos inicializá-los de várias formas, vejamos.

Inicializando todos os elementos um a um.

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

Inicializando todos os elementos com zero.

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

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

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

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

```
int array[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 array[6];    /* Os elementos de array são inicializados
                    com zero, automaticamente. ‘‘static’’ implícito */

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

6.1.4 Atribuindo Valores em Arrays

Aqui falar sobre índice e demonstrar como fica o array na memória após a atribuição.

```
  int att[2];
  att[0] = 10;
  att[1] = 20;
```

6.1.5 Acessando Arrays

Discorrer brevemente sobre o acesso aos elementos de um array.

6.2 Array de Caracteres

6.3 Arrays na Vida Real?!

6.3.1 Imprimir Elementos

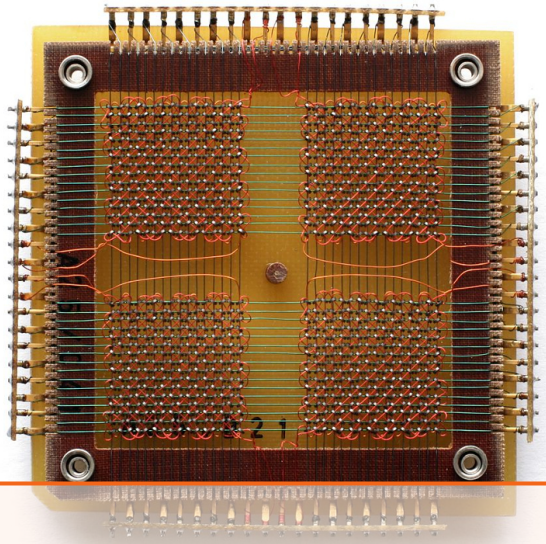
6.3.2 Somar Elementos

6.3.3 Inverter o Array

6.3.4 Ordenar o Array

6.4 Arrays como Parâmetros

6.5 Arrays Multidimensionais (Matrizes)



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).

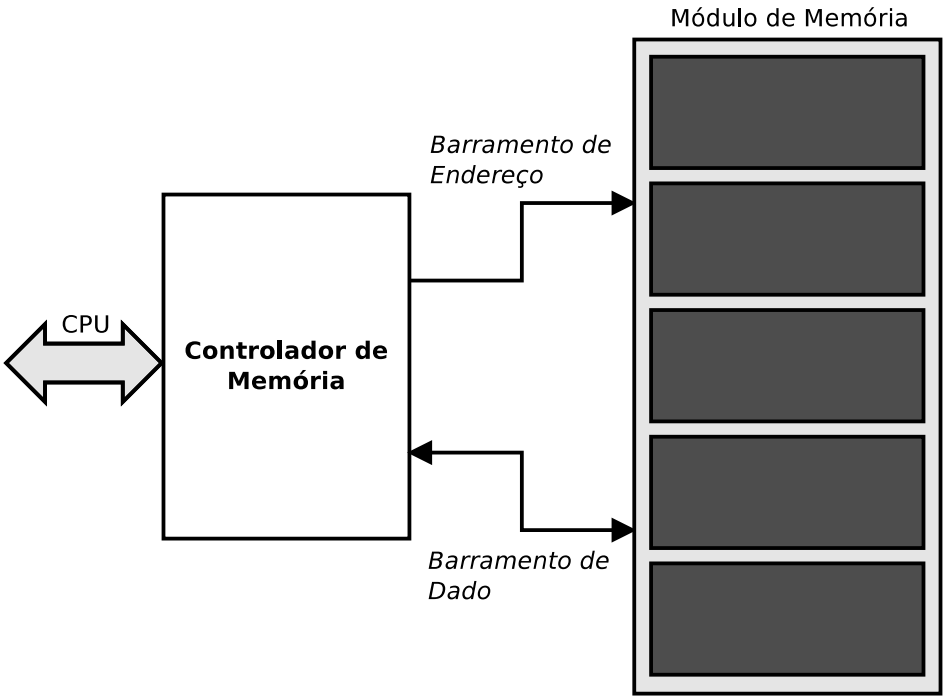


Figura 7.1: Barramentos de Endereço e de Dados

7.1.1 Memória Principal

Sobre a memória principal.

7.2 Usando Ponteiros

Como usar ponteiros.

7.3 Usando Vetores

Como usar vetores.

7.4 Vetores NÃO são Ponteiros

Vetores não são ponteiros.