

Introdução à Linguagem C

Laboratório de Sistemas Operacionais

Prof. MSc. João Tavares



JESUÍTAS BRASIL

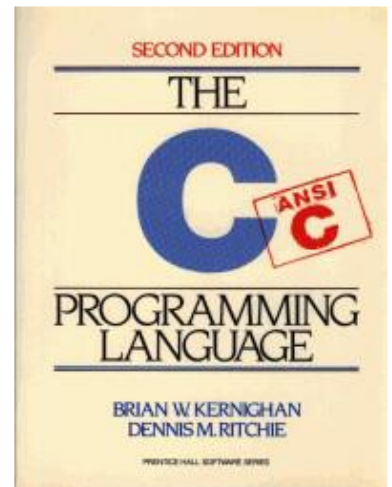


Somos infinitas possibilidades

História

- Desenvolvimento inicial AT&T Bell Labs por Dennis Ritchie em 1972
 - Derivada da Linguagem B (versão de BCPL)
 - Criada para implementar o Unix
- Livro Clássico: (Primeira edição 1978)

The C Programming Language, Second Edition
Brian W. Kernighan and Dennis M. Ritchie.
Prentice Hall, Inc., 1988.



ANSI C / ISO C

- 1983: American National Standards Institute definiu uma especificação padrão para C
- 1990: o ANSI C com pequenas modificações foi padronizada pela ISO
- Versão anterior é denominada de K&R C (baseada na descrição do livro de 1978)

Características de Linguagem C

- É uma linguagem
 - de proposito geral
 - estruturada
 - procedural / imperativa
 - concebida (originalmente) para uso com o Unix
- Muito usada para desenvolver sistemas e aplicativos
- Disponível para muitas plataformas

Características de Linguagem C

- Considerada de nível médio
 - C permite manipular bits, bytes e endereços de memória como uma linguagem de baixo nível
 - C possui construções e rotinas como linguagens de alto nível
- C é para profissionais
 - Uso primário para desenvolvimento de sistemas

Primeiro programa em C

- Ciclo básico de desenvolvimento:

1) Editar o arquivo fonte utilizando o editor de textos de sua preferência (ex.: HelloWorld.c)

```
#include <stdio.h>
int main(void)
{
    printf("hello, world\n");
    return 0;
}
```

2) Compilar o fonte, gerando o programa executável


```
$ gcc -Wall HelloWorld.c -o helloworld
```

3) Executar o programa resultante

```
$ ./helloworld [parâmetros, se houverem]
```

Primeiro programa em C

- Primeira linha instrui o pré-processador a expandir, naquele ponto, o conteúdo do arquivo **header *stdio.h***
 - **pré-processador** → modifica o fonte antes da compilação, eliminando as linhas iniciadas por **#**
 - **header** é o arquivo que contém apenas protótipos (assinatura) de funções, mas não suas implementações
 - **stdio.h** → Funções padrão para entrada e saída



```
#include <stdio.h>
int main(void)
{
    printf("hello, world\n");
    return 0;
}
```

Primeiro programa em C

- A seguir, define uma função de nome main que retorna um inteiro e não recebe parâmetros
 - função **main()** é **especial** → marca o ponto de início de execução de um programa escrito em C
 - main() pode receber os parâmetros da linha de comando: **int main(int argc, char *argv[])**

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



```
int main(void)
```

```
{
```

```
    printf("hello, world\n");
```

```
    return 0;
```

```
}
```


Primeiro programa em C

- { e } são delimitadores de bloco de comandos
- Neste caso em particular, o bloco corresponde ao corpo (implementação) da função **main()**

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

```
int main(void)
```

```
{
```

```
    printf("hello, world\n");
```

```
    return 0;
```

```
}
```

Primeiro programa em C

- Chama a função **printf()**, passando uma constante **char*** a ser escrita na saída padrão
 - **declaração** está no header `stdio.h`;
 - **implementação** em uma biblioteca do sistema (`libc`) a ser ligada ao programa pelo compilador
 - **char*** → ponteiro de caracter, aponta para a primeira posição de um array de caracteres na memória.
- * Em C, strings são arrays de caracteres onde a última posição contém o caractere `'\0'` (NULL)

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

```
int main(void)
```

```
{
```



```
    printf("hello, world\n");
```

```
    return 0;
```

```
}
```

Primeiro programa em C

- O comando **return** finaliza a execução da função, retornando ao chamador o valor passado como parâmetro
 - No caso do `main()` esse valor é retornado ao sistema como status de terminação do processo
 - = 0 → sucesso
 - ≠ 0 → falha

```
#include <stdio.h>
int main(void)
{
    printf("hello, world\n");
    return 0;
}
```



Mais sobre a compilação

- Programa resultante já tem permissão de execução
- gcc é um dos compiladores mais usados devido à ampla disponibilidade e por ser livre
 - Opções básicas do **gcc**:
 - Wall → instrui o compilador para alertar sobre todas as construções potencialmente perigosas no fonte
 - o → informa o nome do executável (default: *a.out*)
 - Para conhecer mais opções → *man* e *info*
 - * Ex.: nível de otimização, ligação a bibliotecas, informações para depuração etc.
- Existem outros compiladores C: cc (sun/spark), icc (intel/x86), etc.

Variáveis

- Devem ser declaradas antes de usadas
 - Declaração começa com o tipo, seguido do nome de uma ou mais variáveis.

Ex.: `int max, min, *p, resultados[20]`

- Podem ser declaradas no início de cada bloco
 - Variáveis locais são criadas (na pilha) quando uma função é chamada e destruídas no retorno da função
 - Variáveis declaradas fora de funções são globais

- Podem ser inicializadas e documentadas na declaração

Ex.: `int max=1000; // numero máximo de itens`

Variáveis

- C não garante inicialização automática
 - se a inicialização for omitida, a variável pode conter valor indeterminado;
- Tipos mais comuns em C:
 - int** → Número inteiro com sinal
Ocupa 1 palavra em memória
 - char** → Número inteiro com sinal / caracter ASCII
Ocupa 1 byte em memória
 - float** → Numero real (ponto flutuante)
 - double** → Numero real, precisão dupla

Nomes das variáveis

- Devem iniciar por uma letra, sendo os demais caracteres letras, números ou sublinhado
 - Caracteres minúsculos e maiúsculos são diferentes
 - Por convenção, começam com letra minúscula, assim como nomes de funções
- Nomes completamente em maiúsculas são reservados para constantes/macros do pre-processador
- Não pode ser uma palavra-reservada:

auto	break	case	char	continue	default
do	double	else	enum	extern	float
for	goto	if	int	long	register
return	short	sizeof	static	struct	switch
typedef	union	unsigned	void	while	

Modificadores de tipos

- Permitem alterar a capacidade de representação e o consumo de memória dos tipos básicos inteiros
 - short → reduz consumo, reduz capacidade
 - long → amplia capacidade, ampliando consumo
 - unsigned → amplia capacidade descartando intervalo de números negativos, não altera consumo
- Implementação é variável!
 - Depende do compilador utilizado e das características do processador/da arquitetura alvo
 - Manipulação de tipos derivados podem ser mais lentos (maior custo de processamento) que de tipos básicos

Modificadores de tipos

- Caso o tipo básico seja omitido ao lado do modificador, assume-se que seja int
- Tipos derivados mais usuais
 - unsigned int, ou unsigned
 - short int, ou short
 - long int, ou long
 - long long int, ou long long
 - unsigned long int, ou unsigned long
 - unsigned char

Tipos estruturados

- struct → possibilita agrupar variáveis de tipos preexistentes originando um tipo de dado estruturado
 - Semelhante a um objeto, porém sem métodos
 - Por conveniência, pode-se atribuir um novo nome a esse tipo utilizando a diretiva **typedef**, Ex.:

```
typedef struct Ponto {  
    float coord_x;  
    float coord_y;  
} TPonto;  
  
...  
struct Ponto p1;  
TPonto p2;  
  
...  
p1.coord_x = 7.9;  
p2.coord_x = 85.37;
```

Entrada e Saída de dados

- `stdio.h` → inclui funções para interação através da E/S padrão (STDIN, STDOUT, STDERR) do processo
Ex.: *soma.c*

```
#include <stdio.h>
int main(){
    int a, b, c;
    printf("Entre o primeiro valor:");
    scanf("%d", &a);
    printf("Entre o segundo valor:");
    scanf("%d", &b);
    c = a + b;
    printf("%d + %d = %d\n", a, b, c);
    return 0;
}
```

Saída formatada

- `printf()` → imprime para a saída padrão
 - Primeiro parâmetro é a string de formatação
 - * Define como apresentar os demais parâmetros na saída
 - * Marcações iniciadas por % indicam uma especificação de formato
 - * Serão substituídas pelo valor do parâmetro correspondente após formatado
- Demais caracteres são copiados diretamente para a saída padrão

Input: `printf("Color %s, Number %d, Float %5.2f", "red", 123456, 3.14);`

Output: Color red, Number 123456, Float 3.14

Saída formatada

- Alguns especificadores de formato:

Cód.	Efeito / Formatação
%d	Número decimal inteiro (int). Também pode ser usado %i como equivalente a %d.
%u	Número decimal natural (unsigned int), ou seja, sem sinal.
%o	Número inteiro representado na base octal. Exemplo: 41367 (corresponde ao decimal 17143).
%x	Número inteiro representado na base hexadecimal. Exemplo: 42f7 (corresponde ao decimal 17143). Se usarmos %X, as letras serão maiúsculas: 42F7.
%f	Número decimal de ponto flutuante (float). Se quisermos usar um número do tipo double, usamos %lf em vez de %f.
%e	Número em notação científica, por exemplo 5.97e-12. Podemos usar %E para exibir o E maiúsculo (5.97E-12).
%g	Escolhe automaticamente o mais apropriado entre %f e %e. Novamente, podemos usar %G para escolher entre %f e %E
%p	Ponteiro: exibe o endereço de memória do ponteiro em notação hexadecimal.
%c	Caractere: imprime o caractere que tem o código ASCII correspondente ao valor dado.
%s	Seqüência de caracteres (string, em inglês).

Sequências de escape

- São combinações de caracteres iniciadas por \
- São tratadas de forma especial pelo compilador
 - Utilizadas para representar em uma string:
 - * caracteres sem representação visual (ex.: beep)
 - * caracteres de difícil acesso pelo teclado
 - * modificar o significado de um caracter usado pela sintaxe da linguagem C (ex.: aspas)

Sequências de escape

Seq.	Significado
\n	Quebra de linha (line feed ou LF)
\t	Tabulação horizontal
\b	Retrocede o cursor em um caractere (backspace)
\r	CR: volta o cursor para o começo da linha sem mudar de linha
\a	Emite um sinal sonoro
\f	Alimentação de formulário (form feed ou FF)
\v	Tabulação vertical (em impressoras)
\"	Aspa dupla
'	Aspa simples
\\	Barra invertida
\0	NULL (byte zero, usado como terminador de strings)
\N	O caractere cuja representação octal é N (dígitos de 0 a 7)
\xN	O caractere cuja representação hexadecimal é N (dígitos 0-9A-F)

Entrada formatada

- **scanf()** → entrada de dados formatada
 - Lê caracteres da entrada padrão
 - * Dados somente são processados quando usuário pressiona a tecla enter
 - Converte para tipo de dados especificado na string de formatação
 - * Interpreta apenas os caracteres válidos para o formato especificado
 - Atribui o valor resultante à região de memória indicada no parâmetro
 - &var** → retorna o endereço de memória da variável var
 - * variáveis do tipo ponteiro (arrays ou variáveis declaradas com *) são usadas diretamente

Erros frequentes

- Situações de erro comuns (a serem evitadas):
 - Esquecer que C é *case-sensitive* e ignorar maiúsculas e minúsculas
 - * Ex.: funções C são sempre com minúsculas
 - Esquecer de usar o & no scanf
 - * Ex.: passar variáveis que não são ponteiros
 - Parâmetros em excesso ou falta deles
 - Esquecer de declarar uma variável antes de utilizar
 - Declarações no meio dos comandos não é permitida
 - Vários problemas que não geram erro de compilação
 - * Wall no gcc pode ajudar em alguns casos

Erros x avisos

- Programas com erro de sintaxe não compilam
 - Outros problemas de codificação não impedem a compilação!
- Nos compiladores modernos pode-se habilitar a busca por problemas que vão além da sintaxe
 - Alertas (warnings) são emitidos pelo compilador avisando sobre potenciais problemas identificados
 - * warnings não impedem a compilação!
 - Por padrão, isso não é habilitado pois torna a compilação mais lenta
- Outros problemas não geram nem avisos
 - É preciso depurar o programa!

Expressões aritméticas

- Atribuição (=): avalia uma expressão e armazena o resultado em uma variável.

Ex.: `var = 123 + 912;`

- Operadores aritmaritméticos básicos são: +, -, *, /, e % (resto de divisao inteira)
 - Podemos usar mais de um operador na mesma expressão.
 - A precedência é igual à usada na matemática
 - É possível usar parenteses.
 - Ex.:

`a = 2 + 4 * 10; /* retornara 42 (e nao 60) */`

`a = 2 + 40 / 2 + 5; /* retornara 27 (e nao 6) */`

Expressões aritméticas

- Incremento / Decremento em uma Unidade

Ex.:

```
valor++; /* valor = valor + 1 */
```

```
++valor;
```

```
valor--; /* valor = valor -1 */
```

- Notação abreviada

Ex.:

```
x *= 12; /* x = x * 12; */
```

```
x /= 10; /* x = x / 10; */
```

```
x += 2; /* x = x + 2; */
```

```
x -= 2; /* x = x - 2; */
```

```
x %= 11; /* x = x % 11; */
```

Expressões aritméticas

- Exemplo:

```
#include <stdio.h>
int main(){
    int a, b;
    a = b = 5;
    printf("%d\n", ++a + 5);
    printf("%d\n", a);
    printf("%d\n", b++ + 5);
    printf("%d\n", b);
    return 0;
}
```

- Qual o resultado?

Seleção: comando if

- Sintaxe:

```
if (expressão_de_condição) {  
    bloco de comandos  
} else {  
    bloco de comandos alternativo  
}
```

- Comportamento

- Avalia a expressão de condição e caso ela resulte em um valor diferente de zero, executa o bloco de comandos do *if*, caso contrário, o do *else*

- Não existe tipo booleano em C! → int

- = 0 → falso

- ≠ 0 → verdadeiro

Expressões de condição

- Operadores Relacionais
 - $>$ → maior que
 - $>=$ → maior ou igual a
 - $<$ → menor que
 - $<=$ → menor ou igual a
 - $==$ → igual
 - $!=$ → diferente
- Operadores Lógicos
 - $\&\&$ → AND/conjunção (“e” lógico)
 - $||$ → OR/disjunção (“ou” lógico)
 - $!$ → NOT/negação

Seleção: alternativas ao if

- Considere as situações a seguir:

Ex.1: Atribuição de valores alternativos

```
if (teste) { var = expr1; }  
else { var = expr2; }
```

Ex.2: Comparações == sobre uma variável inteira

```
if (var == valor1) { comandos... }  
else if (var == valor2) { comandos... }  
else if (var == valor3) { comandos... }  
else { comandos... }
```

- A linguagem C oferece construções mais concisas para essas situações particulares (frequentes) do comando if
→ ?: e switch

Seleção: operador ternário ?:

- Sintaxe:

(expressão_de_condição) ? valor : valor_alternativo

- Comportamento

- Avalia a expressão de condição e caso ela resulte em um valor diferente de zero (verdadeiro), gera o valor padrão; caso contrário gera o valor alternativo

- * Tipo do valor gerado pode ser qualquer

- * Tipicamente usado na atribuição a variáveis

Ex.:

```
int horaAbertura = (diaSemana == DOMINGO) ? 11 : 9;  
printf ("Abrimos às %d horas", horaAbertura);
```

Seleção: comando switch

- Sintaxe:

```
switch (expressão_tipo_int) {  
  case valor1:  
    lista de comandos; (Não é necessário usar a notação de blocos { })  
    break;  
  case valor2:  
    lista de comandos; (Não é necessário usar a notação de blocos { })  
    break;  
  default:  
    lista de comandos; (Não é necessário usar a notação de blocos { })  
}
```

- Comportamento

- Compara sequencialmente o valor da expressão (int) com cada um dos valores (int) nos case
 - * Em caso de *match* executa todos os comandos a partir daquele case até um break ou o fim do switch.
 - * Senão, executa o bloco default se houver

Seleção: comando switch

- Exemplo:

```
int opcao;
printf ("[1] Cadastrar cliente\n"
        "[2] Procurar cliente\n"
        "[3] Inserir pedido\n"
        "[0] Sair\n\n"
        "Digite sua escolha: ");
scanf ("%d", &opcao);
switch (opcao) {
case 1:
    cadastra_cliente(); break;
case 2:
    procura_cliente(); break;
case 3:
    insere_pedido(); break;
case 0:
    return 0;
default:
    printf ("Opção inválida!\n");
}
```

Repetição: comando while

- Sintaxe:

```
while (expressão_de_condição) {  
    bloco de comandos;  
}
```

- Comportamento

- Laço de repetição com execução condicionada a avaliação do resultado da expressão de condição

- * Se for diferente de zero (verdadeiro) executa o bloco de comandos e retorna ao início do laço

- * Caso contrário, sai do laço

Ex.:

```
while (a < b) {  
    printf ("%d é menor que %d\n", a, b);  
    a++;  
}
```

Repetição: comando do ... while

- Sintaxe:

```
do {  
    bloco de comandos;  
} while (expressão_de_condição);    (não esquecer do ‘;’)
```

- Comportamento

- Semelhante ao while, porém a expressão de condição é avaliada somente após a primeira execução do bloco de comandos

- * O bloco de comandos é executado ao menos uma vez!

Ex.:

```
do {  
    printf ("%d\n", a);  
    a++;  
} while (a < b);
```

Repetição: comando for

- Sintaxe:

(qualquer componente do for dentro dos '(...)' pode ser omitido)

```
for (inicialização; condição; incremento) {  
    bloco de comandos;  
}
```

- Comportamento

- Notação abreviada para o seguinte caso particular (frequente) de uso do comando while:

```
inicialização;  
while (condição) {  
    bloco de comandos;  
    incremento;  
}
```

- Diferentemente de C++ e Java, em C “puro”, variáveis têm que ser declaradas antes das instruções de inicialização

Repetição: comando for

- Exemplo 1 – utilizando incremento inteiro

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

```
int main() {
```

```
    int i, soma;
```

```
    int n, x;
```

```
    float media;
```

```
    printf("Quantos números para cálculo da média?");
```

```
    scanf("%d", &n);
```

```
    for (soma=0, i=0; i<n; i++) {
```

```
        printf("Digite %d. número : ", i);
```

```
        scanf("%d", &x);
```

```
        soma += x;
```

```
    }
```

```
    media = (float) soma / n;
```

```
    printf("A média é %f\n", media);
```

```
    return 0;
```

```
}
```

Repetição: comando for

- Exemplo 2 – incremento com ponto flutuante

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

```
int main() {
```

```
    int max;
```

```
    float c;
```

```
    printf("Até que número você deseja a contagem? ");
```

```
    scanf("%d", &max);
```

```
    for (c=0; c<=max; c=c+0.1) {
```

```
        printf("%f\n", c);
```

```
    }
```

```
    return 0;
```

```
}
```


Repetição: comando for

- Exemplo 3 – laço infinito

```
for (;;) {  
    comandos...  
}
```

Desvio: comandos break e continue

- São utilizados para implementação de desvios incondicionais
 - break → aborta o último laço ou switch iniciado
 - * Salta para o comando imediatamente subsequente ao laço (for, while ou do-while) ou switch que contém o break
 - continue → salta para a próxima iteração do laço

Ex.:

```
while (1) {  
    printf("Valor de a? ", &a); scanf("%d", &a);  
    if (a < 0) break;  
    if (a<MIN || a>MAX) continue;  
    processar_valor(a);  
}
```

Desvio: comandos break e continue

- Exemplo 2 – utilizando while e switch

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

```
int main() {
```

```
    int opcao = 0;
```

```
    while (opcao != 5){
```

```
        printf("Escolha uma opção entre 1 e 5: ");
```

```
        scanf("%d", &opcao);
```

```
        if (opcao > 5 || opcao < 1) continue;
```

```
        switch (opcao){
```

```
            case 1: printf("\n --> 1a. opcao.."); break;
```

```
            case 2: printf("\n --> 2a. opcao.."); break;
```

```
            case 3: printf("\n --> 3a. opcao.."); break;
```

```
            case 4: printf("\n --> 4a. opcao.."); break;
```

```
            case 5: printf("\n --> Abandonando.."); break;
```

```
        }
```

```
    }
```

```
    return 0;
```

```
}
```

Arrays em C

- Armazenam uma coleção de elementos de um mesmo tipo em posições contíguas de memória
 - Índices do array iniciam em 0 (zero)
 - Posições acessadas fornecendo o índice do elemento entre colchetes

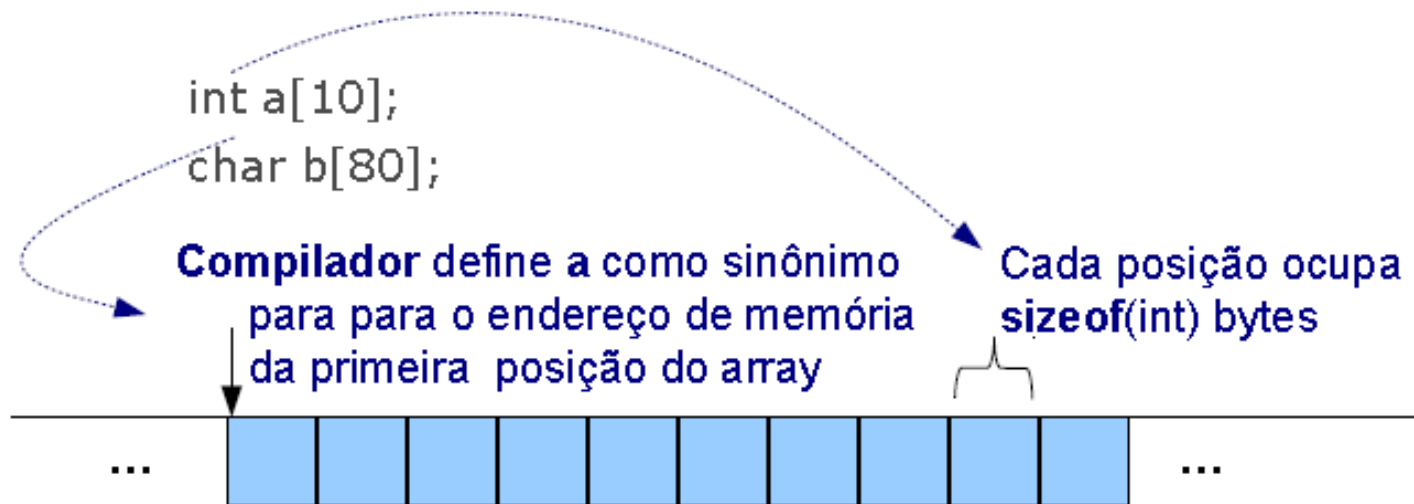
Ex.: `s[59] = 'C';`

- Não há verificação de limites de índices em C!
- Abordagens de alocação de Memória
 - estática → realizada pelo compilador
 - dinâmica → realizada durante a execução

Arrays

- Alocação estática → realizada pelo compilador
 - Tamanho do array definido na declaração da variável
 - Não é necessário liberar explicitamente a região de memória reservada ao array

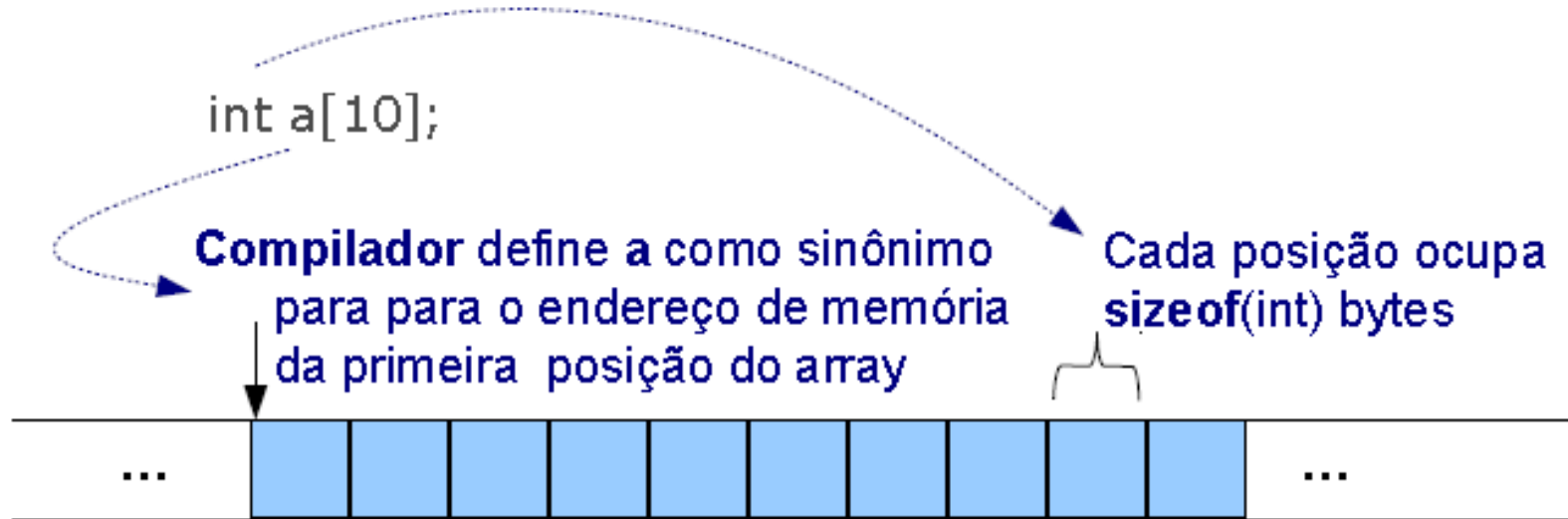
Ex.:



Arrays

- Alocação dinâmica → realizada durante a execução
 - Variável declarada como um ponteiro;
 - Posteriormente uma região de memória é alocada e seu endereço (posição 0 do array) atribuído ao ponteiro
- Ex.: `int *a; ... a=calloc(sizeof(int), 10);`
- Quando não for mais usado, o espaço deve ser explicitamente liberado com `free()`
- Ex.: `free(a);`

Acessando arrays



- `v = a[i]`
 - valor **int** da posição de memória cujo endereço é $(a + i * \text{sizeof}(\text{int}))$
- `a[i] = ...`
 - posição de memória $(a + i * \text{sizeof}(\text{int}))$ é tratada como contendo um **int** e é sobrescrita com novo valor **int**

Strings em C

- String em C → é uma sequência de caracteres terminada pelo caractere `'\0'` (null)
 - Armazenada como um **vetor de caracteres**
 - `'\0'` marca o fim da “parte útil” do vetor
 - `char*` → endereço do vetor que armazena a string
 - O tamanho da string é determinado pela posição do primeiro caractere `'\0'` no vetor

`char* s = "abc";`



Strings em C

- Em C, na prática, as três definições seguintes representam o mesmo string “abc”.

- Declaração usual (compilador faz o serviço)

```
char *s1 = "abc";
```

- Declarando e inicializando o array explicitamente

```
char s2[4] = { 'a', 'b', 'c', '\0' };
```

- **s3** até comportaria mais caracteres, mas 'd' e 'e' não são relevantes para o string...

```
char s3[6] = { 'a', 'b', 'c', '\0', 'd', 'e' };
```

(Irrelevantes para o string pois estão após o '\0')

Manipulando Strings em C

- O header padrão <string.h> contém várias funções para manipular strings
 - **strcat (s1,s2)**: Adiciona uma cópia de s2 (incluindo o terminador null) ao final de s1. Retorna s1.
 - **strcmp (s1,s2)**: Realiza comparação ASCII (+/- alfabética) entre as strings. Retorna zero se as strings são iguais, negativo se s1 é anterior a s2 ou positivo caso contrário
 - **strcpy (s1,s2)**: Copia conteúdo de s2 para s1 (incluindo '\0'). Retorna s1.
 - **strlen (s)**: Retorna o número de bytes de s (sem contar o '\0')

Manipulando Strings em C

- Outras funções definidas em `<string.h>`
 - **strtok()**, **strstr()**, **strchr()**, etc.
- O header `<string.h>` também inclui funções para cópia e manipulação de arrays
 - **memcpy()**, **bcopy()**, **memset()**, etc.
 - Diferença principal é que não incluem tratamento especial para o `'\0'`
- Para mais informações → [man string](#)

Ponteiros

- Ponteiros → armazenam o endereço de memória de uma dada informação
 - Tem tipo (ex.: ponteiro para int, ponteiro para char)
- Origens de ponteiros
 - Estruturas de dados alocadas dinamicamente com `malloc()` / `calloc()`
 - Arrays
 - * São de fato ponteiros para a região de memória onde estão os elementos do array
 - * Idem para strings
 - Operador `&var` retorna o endereço de memória da variável *var*

Expressões com Ponteiros

- Seja a uma variável do tipo int
 - $p \leftarrow$ endereço da variável a
`int* p = &a;`
 - $b \leftarrow$ valor int da posição apontada por p
`int b = *p;`
- Seja s uma variável do tipo `struct T {int x; int y;}`
 - $d \leftarrow$ endereço da variável s
`struct T* d = &s;`
 - $b \leftarrow$ valor do campo x da struct cujo endereço de memória esta em d
`int b = d->x;`

Expressões com Ponteiros

- Seja v um vetor de int
 - v é naturalmente um ponteiro para o primeiro elemento do vetor

```
int* p = v;
```

```
int b = *v; /* equivalente a acessar v[0] */
```

- p ← endereço do **10o elemento** do vetor v

```
int* p = &v[9];
```

```
int b = *p; /* equivalente a acessar v[9] */
```

- p ← endereço do próximo elemento do array (aritmética de ponteiros)

```
p++;
```

Leituras complementares

- Alguns recursos em formato digital...
 - Páginas *info* da **libc** e
 - Páginas *man* do **gcc** e da biblioteca de **strings**
 - Programar em C (Wikibooks.org) *
http://pt.wikibooks.org/wiki/Programar_em_C/Indice
 - C Programming
<http://www2.its.strath.ac.uk/courses/c/>
 - Como funciona a programacao em C
<http://informatica.hsw.uol.com.br/programacao-em-c.htm>
 - C (programming language) na Wikipedia
[http://en.wikipedia.org/wiki/C_\(programming_language\)](http://en.wikipedia.org/wiki/C_(programming_language))

Referências Bibliográficas

- Material originalmente elaborado por Prof. Cristiano Costa. Material autorizado e cedido pelo autor. Revisado e atualizado por Prof. Luciano Cavalheiro e posteriormente pelo Prof. João Tavares.