Curso Rápido de C

Prof. Vítor Souza
Linguagens de Programação

Departamento de Informática
Univ. Federal do Espírito Santo

11/04/2006

Sobre o Curso

- Estes slides foram criados no Departamento de Informática da Universidade Federal do Espírito Santo (UFES) e estão disponível no seguinte endereço:

  http://www.inf.ufes.br/~vsouza/

- O material usado como base* foi uma apostila de C montada pela Universidade Federal de Minas Gerais (UFMG), disponível no seguinte endereço:

  http://www.ead.eee.ufmg.br/cursos/C/

- Este curso é voltado para pessoas que já possuem conhecimento de lógica de programação.

* Exceto parte 9

Conteúdo do Curso

- Primeiros passos;
- Variáveis, constantes, operadores e expressões;
- Vetores e strings;
- Ponteiros;
- Funções;
- Diretivas de compilação;
- Entrada e saída;
- Tipos de dados avançados;
- Tipos abstratos de dados – TADs.

Curso Rápido de C

Parte 1: Primeiros Passos

Prof. Vítor Souza
Linguagens de Programação

Departamento de Informática
Univ. Federal do Espírito Santo

11/04/2006

Primeiros Passos

- C foi criado na década de 70 por Dennis Ritchie;
- C é Case Sensitive;
- Hello World:

```
#include <stdio.h> // Para usar printf().

/* Um primeiro programa. */
main() {
    printf("Hello, World!\n");
}
```

Explicando...

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

- **Inclusão de cabeçalho de biblioteca de sistema;**

```
// Para usar printf().
/* Um primeiro programa. */
```

- **Comentários. Ignorados pelo compilador.**

```
main() {
```

- **Definição da função principal do programa.**

```
  printf("Hello, World!\n");
```

- **Chamada de função para impressão na tela. Todo comando em C deve terminar com “;”.**

Compilando...

- Escreva o código num arquivo com a extensão .c;
- Utilize o compilador GCC;
- Ex.: suponha que o programa chama-se hello.c:

```
$ gcc -o hello hello.c
```

Arquivo de saída.

Arquivo(s) fonte.

```
$ ./hello
Hello World!
```

Outro Exemplo

```
main() {
   int dias; // Declaração de variáveis.
   float anos;

   /* Entrada de dados. */
   printf("Entre com o número de dias: ");
   scanf("%d", &dias);

   /* Conversão dias -> anos. */
   anos = dias / 365.25;
   printf("\n\n%d dias equivalem a %f
  anos.\n", dias, anos);
}
```

Introdução a Funções

- Bloco de código que pode ser usado diversas vezes;
- Um programa C é um conjunto de funções.

```
#include <stdio.h>

mensagem() {
  printf("Hello ");
}

main() {
  mensagem();
  printf("World!\n");
}
```

Argumentos

- Parâmetros (entrada) que as funções recebem.

```
// Cálculo do quadrado de um número.
square(int x) {
  printf ("O quadrado é %d",(x * x));
}
main() {
  int num;
  printf("Entre com um número: ");
  scanf("%d", &num);
  printf("\n\n");
  square(num);
}
```

Vários Argumentos

```
mult (float a, float b, float c) {
   printf("%f", a * b * c);
}

main () {
   float x, y;
   x = 23.5;
   y = 12.9;
   mult(x, y, 3.87);
}
```

Retornando Valores

- O retorno “padrão” é inteiro;
- Usa-se `return` para indicar o valor a ser retornado.

```
prod(int x, int y) {
   return (x * y);
}

main() {
   int saida;
   saida = prod(12, 7);
   printf("A saída é: %d\n", saida);
}
```

Especificando o Retorno

- Para retornos diferentes de `int`, precisamos especificar o tipo de retorno da função.

```
float prod(float x, float y) {
   return (x * y);
}

int main() {
   float saida;
   saida = prod(45.2, 0.0067);
   printf("A saída é: %f\n", saida);
}
```

Forma Geral das Funções

- Declaração de funções obedece a seguinte sintaxe:

```
<tipo> <nome>(<argumentos>) {
  <conjunto de comandos>
}
```

- Sendo que:

```
<argumentos> = <arg1>, <arg2>, ..., <argN>

<arg> = <tipo> <nome>
```

Introdução a Entrada e Saída de Dados

- Existem várias funções para entrada e saída de dados, porém utilizaremos as duas mais poderosas:
  - **scanf()** para entrada de dados;
  - **printf()** para saída de dados.

- Existem versões delas para I/O em arquivos e em strings (veremos mais adiante).

- Sintaxe:
  - `int printf(const char *formato, ...);`
  - `int scanf(const char *formato, ...);`

N parâmetros

Formatos

- Formatos são strings com códigos de controle embutidos. Veja alguns códigos:

Código	Tipo de dado
%d	Número inteiro (int)
%f	Número real (float)
%s	String (vetor de caracteres)
%u	Número inteiro sem sinal (unsigned int)
%x	Valor hexadecimal (unsigned int)
%c	Caractere (char) ou sequência de caracteres (vetor)

- Mais detalhes no manual dos comandos printf / scanf.

Exemplos

```
printf ("Teste %% %%"); // "Teste % %"
printf ("%f", 40.345); // "40.345"

// "Um char D e um int 120"
printf ("Um char %c e um int %d", 'D', 120);

// "Este é um exemplo"
printf ("%s é um exemplo", "Este");

// "Juros de 10%"
printf ("%s%d%%", "Juros de ", 10);
```

Exemplos

```
main() {
   char ch;
   scanf("%c", &ch);
   printf("Voce pressionou %c.\n", ch);
}

main() {
   int idade;
   scanf("%d", &idade);
   printf("Você tem %d anos.\n", idade);
}
```

Exemplos

```
main() {
   unsigned int x;
   scanf("%x", &x);
   printf("Valor hexadecimal: %x.\n", x);
   printf("Equivalente inteiro: %d.\n", x);
}

main() {
   int dia, mes, ano;
   scanf("%d/%d/%d", &dia, &mes, &ano);
}
```

Lendo Strings

```
main() {
   char string[100];
   printf ("Digite uma string: ");
   scanf("%s", string); // Sem o &!
   printf("\nVocê digitou: %s\n\n", string);
}
```

```
Digite uma string: Engenharia de Software

Você digitou: Engenharia
```

Lendo Linhas Inteiras

```
main() {
   char string[100];
   printf ("Digite uma string: ");
   scanf("%[^\n]", string); // Sem o &!
   printf("\nVocê digitou: %s\n\n", string);
}
```

```
Digite uma string: Engenharia de Software

Você digitou: Engenharia de Software
```

Introdução ao Controle de Fluxo

- C possui diversos comandos de controle de fluxo: desvios condicionais, desvios incondicionais, repetições, etc:
  - `if` / `else`;
  - `switch`;
  - `for`;
  - `while`;
  - `do` / `while`;
  - `goto`.

Desvio Condicional com if

- Sintaxe:

```
if (<condição>) <comandos>
else <comandos>
```

- Exemplo:

```
if (num < 10) printf("Menor que 10.\n");
else {
  printf("Maior ou igual a 10.\n");
  num = num % 10
  printf("Normalizado: %d.\n", num);
}
```

Desvio Condicional com `if`

- else é opcional:

```
if (idade > 18) {
  printf("Maior de idade.");
}
```

- Várias condições podem ser testadas com `else if`

```
if (num < 10) printf("Menor que 10.\n");
else if (num == 10) printf("Igual a 10.\n");
else printf("Maior que 10.\n");
```

Desvio Condicional com switch

- Sintaxe:

```
switch (<variavel inteira>) {
   case <val1>:
     <comandos>
   case <val2>:
     <comandos>
   ...
   case <valN>:
     <comandos>
   default:
     <comandos>
}
```

Desvio Condicional com switch

- Exemplo:

```
switch (num) {
   case 10:
     printf("Igual a 10.\n");
     break;
   case 5:
     printf("Igual a 5.\n");
     break;
   default:
     printf("Nem 10 nem 5.\n");
}
```

Desvio Condicional com switch

```
printf("Digite uma letra do alfabeto: ");
scanf("%c", &c);

switch (c) {
  case 'a': case 'A':
  case 'e': case 'E':
  ...
  case 'u': case 'U':
    printf("É vogal.\n");
    break;
  default:
    printf("É consoante.\n");
}
```

Repetição com for

- Sintaxe:

```
for (<inic>; <cond>; <incr>) <comandos>
```

- Exemplo:

```
for (num = 10; num > 0; num--) {
  printf("%d, ", num);
}
printf("FOGO!\n");
```

Repetição com for

```
// Procura um elemento num vetor.
for (i = 0; (i < 10) && (vet[i] != x); i++);

// Soma números até ler o número 0.
for (s = 0, scanf("%d", &n); n != 0; s += n,
  scanf("%d", &n));
```

Repetição com while e do / while

- Sintaxe:

```
while (<condição>) <comandos>
do <comandos> while (<condição>);
```

- Exemplo:

```
int num = 10;
while (num > 0) printf("%d, ", num--);
printf("FOGO!\n");

num = 10;
do printf("%d, ", num--); while (num > 0);
printf("FOGO!\n");
```

Repetição com while e do / while

```
// Procura um elemento num vetor.
i = 0;
while ((i < 10) && (vet[i] != x)) i++;

// Soma números até ler o número 0.
s = 0;
scanf("%d", &n);
while (n != 0) {
  s += n;
  scanf("%d", &n));
}
```

break e continue

- Utilizados respectivamente para sair de um bloco e ir para a próxima iteração:

```
char ch;
for (;;) {
  printf("Digite uma letra, X para sair: ");
  scanf("%c", &ch);
  if (ch == 'X') break;
}
```

break e continue

```
int opcao = 0;
while (opcao != 5) {
  printf("Escolha a opção (1-5): ");
  scanf("%d", &opcao);

  // Verifica opção inválida.
  if ((opcao < 1) || (opcao > 5)) continue;

  switch (opcao) {
    // ...
  }
}
```

Desvio Incondicional com goto

- Sintaxe:

```
goto <rótulo>;
```

- Exemplo:

```
   scanf("%d", &x);
   if (x > 10) goto maior;
   printf("Menor ou igual a 10!");
   goto fim;
maior:
   printf("Maior que 10!");
fim:
   printf("\n");
```

Palavras Reservadas do C

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

Curso Rápido de C

Parte 2: Variáveis, Constantes, Operadores e Expressões

Prof. Vítor Souza
Linguagens de Programação

Departamento de Informática
Univ. Federal do Espírito Santo

11/04/2006

Variáveis

- Nomes:
  - Começar com letras ou “_” (underline);
  - Seguido por letras, números ou underline;
  - Não pode ser igual a uma palavra reservada;
  - Não pode ser igual a uma função já declarada (pelo programador ou das bibliotecas importadas);
  - São aceitos nomes de até 32 caracteres.
- Tipos:
  - Básicos: `char`, `int`, `float`, `double`, `void`;
  - Modificadores: `signed`, `unsigned`, `long`, `short`;
  - Nenhum modificador pode ser aplicado a `float`, somente `long` pode ser aplicado a `double`.

Variáveis

- Tamanhos:
  - Definidos pelo compilador.
  - `sizeof()` informa o número de bytes de um tipo.

```
// gcc version 4.0.3
printf("%d, %d, %d, %d, %d, %d, %d\n",
  sizeof(char), sizeof(short int),
  sizeof(int), sizeof(long int),
  sizeof(long long int), sizeof(float),
  sizeof(double));

$ ./teste
1, 2, 4, 4, 8, 4, 8
```

Variáveis

- Declaração:
  - É obrigatória a declaração antes do uso;
  - É possível declarar mais de uma variável na mesma linha.
  - Podemos atribuir um valor à variável logo na declaração (chama-se inicialização);
  - C não inicializa (zera) variáveis automaticamente.

```
char ch, letra;
long count = 0;
float pi = 3.14, x, y = 4.44;
```

Variáveis

- Escopos:
  - Globais: declaradas fora das funções, disponível em todo o programa, vive enquanto o programa executar;
  - Locais: declaradas dentro de blocos, disponível para aquele bloco, vive durante a execução do bloco;
  - Parâmetros: declaradas como argumentos para funções, disponível para aquela função, vive durante a execução da função.
- Exemplo:

```
int global;
funcao (int parametro) {
  int local;
}
```

Variáveis

- Espaço de nomes:
  - Duas variáveis globais não podem ter o mesmo nome;
  - Idem para duas variáveis locais do mesmo bloco;
  - Duas variáveis locais de blocos diferentes podem ter o mesmo nome.
- Ocultamento:
  - Uma variável local pode ter o mesmo nome que uma global, ocultando esta última durante seu escopo.

```
int numero = 10;
main() {
  int numero = 1;
  printf("%d\n", numero); // Imprime 1.
}
```

Constantes

Tipo	Exemplos
char	'b', '\n', '\0', '1'
int	2, 3200, -130
float	0.0, 23.7, -12.3e-10
double	125463544334.0
Em hexadecimal	0x0F, 0x12A4
Em octal	03212, 034215432

Variáveis Constantes

- Constantes declaradas pelo programador:

```
const float pi = 3.141592;
```

- Alguns compiladores C permitem alteração de variáveis constantes, emitindo apenas um aviso (*warning*);

```
pi = pi * 2;
```

- No GCC 4.0.3: assignment of read-only variable ‘pi’.

Constantes Strings

- Apesar de não existir o tipo string (somente vetor de caracteres), C facilita a construção de vetores de caracteres constantes:

```
char *nome = "Fulano da Silva";
```

- Observações:
  - `'a' é diferente de "a".`
  - nome aponta para um vetor constante de caracteres (`nome[0] = 'X';` causa *Segmentation Fault*).
  - Para construir Strings manipuláveis:

```
char *nome = (char *)malloc(20 * sizeof(char));
strcpy(nome, "Fulano da Silva");
```

Caracteres Especiais

- Também chamados “constantes de barra invertida”:

Código	Significado
\b	Retrocesso (“back”)
\f	Alimentação de formulário (“form feed”)
\n	Nova linha (“new line”)
\r	Retorno de carro (“carriage return”)
\t	Tabulação horizontal (“tab”)
\"	Aspas
\'	Aspas simples (apóstrofo)
\0	Nulo (fim de string)
\\	Barra invertida (“\”)
\v	Tabulação vertical
\a	Sinal sonoro (“beep”)

Ausência do tipo booleano

- C não possui o tipo booleano. Utiliza qualquer tipo inteiro para “simular” o tipo booleano:
  - O valor 0 (zero) representa FALSO;
  - Qualquer outro valor representa VERDADEIRO.

- Por isso, é possível fazer verificações do tipo:

```
int escolha;
printf("Tecle 0 para sair. Opção: ");
scanf("%d", &escolha);
while (escolha) {
   // ...
}
```

Operadores

- Atribuição:
  - Pode ser simples ou múltipla;
  - Retorna o valor atribuído.

```
a = 10;
x = y = z = 1.5;

// Erro comum. Não confunda com “==”.
if (a = 15) {
  // ...
}
```

Operadores

- Aritméticos:

+	Soma
-	Subtração ou troca de sinal
*	Multiplicação
/	Divisão
%	Resto da divisão inteira
++	Incremento
- -	Decremento

- Soma e subtração podem ser unários ou binários;
- Incremento e decremento são unários e podem ser pré-fixados (incrementa e retorna) ou pós-fixados (retorna e incrementa).

Operadores

- Relacionais:

>	Maior que
>=	Maior ou igual a
<	Menor que
<=	Menor ou igual a
==	Igual a
!=	Diferente de

- Retornam valor booleano: 1 (verdadeiro) ou 0 (falso).

Operadores

- Lógicos:

&&	AND (E)
||	OR (OU)
!	NOT (NÃO)

  - Operam sobre valores booleanos e retornam um valor booleano.

Operadores

- Binários:

&	AND (E)
|	OR (OU)
^	XOR (OU Exclusivo)
~	NOT (NÃO)
>>	Deslocamento de bits para a esquerda
<<	Deslocamento de bits para a direita

- Realizam operações lógicas bit a bit nos tipos inteiros.

Operadores

- Atribuição composta:

Expressão	Equivale a
x += y	x = x + y
x -= y	x = x - y
x *= y	x = x * y
x /= y	x = x / y
x %= y	x = x % y

Expressão	Equivale a
x &= y	x = x & y
x |= y	x = x | y
x ^= y	x = x ^ y
x >>= y	x = x >> y
x <<= y	x = x << y

  - Todos os operadores binários podem fazer composição com a atribuição, como mostrado acima.

Expressões

- Variáveis, constantes e operadores podem formar expressões se dispostos corretamente;
- Exemplo: `anos = dias / 365.25;`
- É preciso conhecer as regras de precedência:

```
1º) ( ) [ ] ->
2º) ! ~ ++ -- . - (unário)
3º) (cast) * (unário)
4º) & (unário) sizeof
5º) * / %
6º) + -
7º) << >>
8º) <<= >>=
...
```

Expressões Encadeadas

- Várias expressões podem ser encadeadas em uma mesma linha, usando vírgulas;

```
a = 2, x = 10, y = x;
```

- No caso de uma atribuição, a expressão retorna a última expressão da lista;

```
x = (a = 2, x = 10, y = x, a + 3);
printf("%d, %d, %d\n", a, x, y); // 2, 5, 10
```

Expressão Condicional Ternária

- Permite retornar um valor dependendo de uma condição:

```
int b = (a > 0) ? -150 : 150;
```

- Este código equivale à:

```
int b;
if (a > 0) b = -150;
else b = 150;
```

Conversão Automática de Tipos

- O compilador C realiza conversões automáticas de tipo quando necessário e possível;
- A conversão é sempre do tipo menor para o tipo maior: `char` para `int`, `float` para `double`;
- Também são feitas conversões de signed para unsigned e vice-versa (pode ter resultados estranhos);
- Conversões impossíveis geram erros de compilação: de `float` para `int`, de `double` para `float`, etc.

Coerção (*cast*)

- Quando a conversão automática não é possível, você deve informar ao compilador que quer forçá-la;
- Forçando a conversão, possivelmente haverá perda de dados e você deve estar ciente disso;
- A sintaxe é: `(tipo) expressão`

```
int num = (int)10.5;

float f = num / 7;
printf("%f\n", f); // 1.000000

f = ((float) num) / 7;
printf("%f\n", f); // 1.428571
```

Curso Rápido de C

Parte 3: Vetores e Strings

Prof. Vítor Souza
Linguagens de Programação

Departamento de Informática
Univ. Federal do Espírito Santo

11/04/2006

Vetores

- Sequência contígua de elementos de mesmo tipo:
- Sintaxe:

```
tipo nome[tamanho]
```

- Exemplo:

```
float v1[20]; // Separa memória p/ 20 floats
int v2[40];   // Separa memória p/ 40 ints
```

Operador de Indexação

- Para acessar um elemento do vetor, usamos o operador de indexação: [ ]
- Exemplo:

```
int i = v2[15];
```

- A indexação começa de 0 e vai até tamanho – 1:

```
for (i = 0; i < 20; i++) v2[i]++;
```

- C não faz verificação da posição acessada:

```
v2[46] = v2[100] + v2[1039]; // Compila!
```

Exemplo de Utilização

```
int num[100];
int count = 0;
int total;
do {
  printf("Digite um nº, 0 para sair: ");
  scanf("%d", &num[count]);
} while (num[count++] != 0);
total = count – 1;
printf("Números que você digitou: ");
for (count = 0; count < total; count++) {
  printf("%d, ", num[count]);
}
printf("\n");
```

Strings

- São vetores de chars, que recebem tratamento especial em determinadas ocasiões (ex.: printf);
- Declaração da string é igual a um vetor:

```
char nome[tamanho];
```

- Ao final da string é colocado um caractere de fim de string: `'\0'`;
- Quando declarar o tamanho da string, levar isso em consideração – funções que manipulam strings dependem dele!

Manipulando Strings

- Por serem vetores, não podem ser tratadas como variáveis primitivas, como inteiros ou floats;
- Por exemplo, você não deve fazer isso, pois não tem o mesmo efeito quando feitas com tipos primitivos:

```
string1 = string2;
if (string1 == string2) { /* ... */ }
```

- Ambas os comandos acima operam sobre a referência, e não valores!
- Veremos mais sobre isso quando discutirmos ponteiros...

Strings Constantes

- C permite a representação de Strings constantes com o uso de aspas duplas:

```
char *nome = "Dennis Ritchie";
printf("%s\n", nome); // Dennis Ritchie

// Segmentation fault! É uma constante!
nome[6] = '_';
```

Inicialização de Strings

- Se declararmos a variável como um vetor, o espaço de memória é separado para ela, e é feita a inicialização da string com o valor da constante:

```
char nome[15] = "Dennis Ritchie";
printf("%s\n", nome); // Dennis Ritchie

// Não é mais uma constante, posso alterar.
nome[6] = '_';
printf("%s\n", nome); // Dennis_Ritchie
```

Funções de Manipulação de Strings

- Leitura e escrita: `scanf` e `printf` – já apresentados;
- Cópia de strings: `strcpy(destino, origem)`:

```
#include <stdio.h>
#include <string.h> // Para usar strcpy()!

int main() {
  char *nome = "Dennis Ritchie", string[15];
  strcpy(string, nome);

  string[6] = '_';
  printf("%s\n", string); // Dennis_Ritchie
}
```

Funções de Manipulação de Strings

- Concatenação de strings:
  `strcat(destino, origem)`:

```
char *nome = "Dennis Ritchie", string[50];

strcpy(string, "O criador do C se chama ");
strcat(string, nome);

// O criador do C se chama Dennis Ritchie
printf("%s\n", string);
```

Funções de Manipulação de Strings

- Tamanho de uma string: `strlen(string)`:

```
char *nome = "Dennis Ritchie", string[50];

strcpy(string, "O criador do C se chama ");
strcat(string, nome);

printf("%d\n", strlen(nome)); // 14
printf("%d\n", strlen(string)); // 38

// Note que o '\0' não conta:
string[0] = '\0';
printf("%d\n", strlen(string)); // 0
```

Funções de Manipulação de Strings

- Comparação de strings: `strcmp(s1, s2)` – retorna 0 se iguais, outro valor se diferentes:

```
char *nome = "Dennis Ritchie", string[50];

strcpy(string, "Dennis");
strcat(string, " Ritchie");

if (strcmp(nome, string) == 0) {
  printf("nome e string são iguais!\n");
}
```

Funções de Manipulação de Strings

- `strcmp(s1, s2)` pode ser usado para determinar ordem alfabética, pois retorna
  - valor positivo (1), se s1 > s2
  - valor negativo (-1), se s1 < s2:

```
char *nome = "Dennis Ritchie", string[50];
printf("Digite seu nome: ");
scanf("%[^\n]", string);

printf("\nEm ordem alfabética:\n");
if (strcmp(nome, string) < 0)
  printf("1. %s\n2. %s\n", nome, string);
else printf("1. %s\n2. %s\n", string, nome);
```

Matrizes e Vetores Multidimensionais

- Podemos declarar vetores com mais de uma dimensão, especificando os tamanhos de cada uma:

```
int matriz[20][10], i, j;

// Preenche uma matriz com linha * coluna.
for (i = 0; i < 20; i++)
  for (j = 0; j < 10; j++)
    matriz[i][j] = i * j;
```

Matriz de char

- Você consegue perceber que uma matriz de char é equivalente a um vetor de strings?

```
char nomes[3][50];
int i;

strcpy(nomes[0], "Dennis Ritchie");
strcpy(nomes[1], "Brian Kernighan");
strcpy(nomes[2], "Richard Stallman");

for (i = 0; i < 3; i++)
  printf("%s\n", nomes[i]);
```

Inicialização de Vetores

- Assim como podemos inicializar tipos primitivos, podemos também inicializar vetores:

```
float vetor[4] = {0.0, 0.5, 1.0, 1.5};

int matriz[2][3] = {1, 2, 3, 4, 5, 6};
int matIgual[2][3] = {{1, 2, 3}, {4, 5, 6}};

char str[10] = {'J', 'o', 'a', 'o', '\0'};
char strIgual[10] = "Joao";

char vetorStrings[3][10] = {"Joao", "Maria",
  "Jose"};
```

Inicialização sem Tamanho

- Podemos não especificar o tamanho do vetor que estamos incializando. O compilador automaticamente atribuirá o valor de acordo com o número de itens na inicialização:

```
int vetor[] = {1, 2, 3, 4, 5};
char nome[] = "Dennis Ritchie";

printf("%d\n", strlen(nome)); // 14
```

Curso Rápido de C

Parte 4: Ponteiros

Prof. Vítor Souza
Linguagens de Programação

Departamento de Informática
Univ. Federal do Espírito Santo

11/04/2006

Ponteiros ou Apontadores

- C é altamente baseado em ponteiros;
  - Isso é tão verdade que já os utilizamos e talvez vocês não tenham percebido!
  - Para ser um bom programador C, é preciso dominar as técnicas de manipulação de ponteiros;

- Advertência: o uso descuidado de ponteiros pode causar sérias dores de cabeça! Não nos responsabilizamos por danos causados pelo mal uso desta técnica – use por sua própria conta e risco!

Ponteiros ou Apontadores

- Ponteiros guardam endereços de memória que fazem referência a variáveis ou funções do seu programa;
- Ponteiros possuem um tipo: um ponteiro para um inteiro é diferente de um ponteiro para um float;
- Declaração de ponteiros – usa o operador *:

```
int *p; // Um ponteiro para inteiro
float *q; // Um ponteiro para float
char *s; // Um ponteiro para char (string)
```

  - Os ponteiros acima não foram inicializados e, portanto, apontam para algum lugar aleatório na memória. Usá-lo pode causar erros!

Atribuindo Valores a Ponteiros

- Ponteiros podem receber endereços de variáveis (usando o operador &) ou valores de outros ponteiros:

```
int contador = 10;
int *p;

p = &contador; // & retorna o endereço!
int *q = p;
```

  - Os ponteiros p e q apontam para o mesmo endereço de memória, onde está situada a variável `contador`.

Acessando Valores

- Se imprimirmos o valor de um ponteiro, obteremos um endereço de memória:

```
// -1075062180, -1075062180
printf("%d, %d\n", p, q);
// 0xbf973d7c, 0xbf973d7c
printf("%p, %p\n", p, q);
```

- Se utilizarmos o operador de derreferenciamento *, obtemos o valor da variável para a qual ele aponta:

```
// 10, 10, 10
printf("%d, %d, %d\n", *p, *q, contador);
```

Alterando Valores

- O operador de derreferenciamento também pode ser usado para alterar o valor de uma variável apontada por um ponteiro:

```
// p e q apontam para a mesma variável!
*p = 15;
*q = 20;

// 20, 20, 20
printf("%d, %d, %d\n", *p, *q, contador);
```

Aritmética de Ponteiros

- O que fizemos até agora com ponteiros não apresenta muitos perigos;
- Já a aritmética de ponteiros potencialmente causa grandes dores de cabeça;
- Aritmética de ponteiros consiste em fazer com ponteiros tudo o que se faz com números inteiros:
  - Atribuição;
  - Soma, subtração, multiplicação, divisão;
  - Incremento (++), decremento (--);
  - Comparação (==, !=, etc.).

Aritmética de Ponteiros

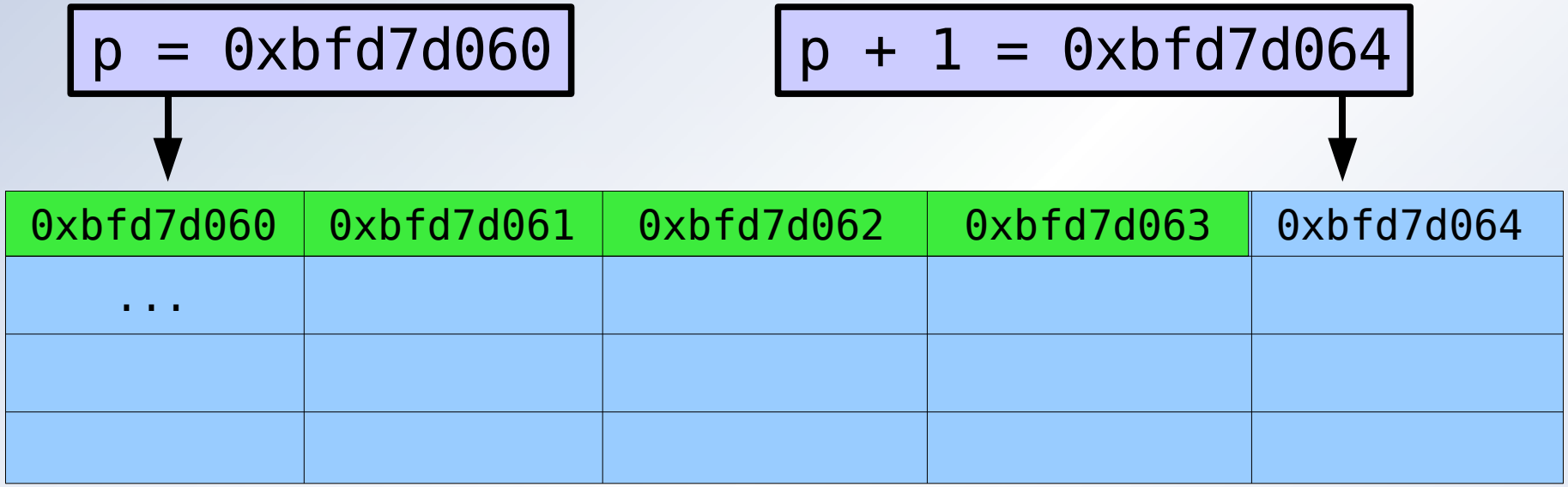
- Todas as operações aritméticas em ponteiros tem efeitos que dependem do tipo do ponteiro;
- Ao somar 1 ao valor de um ponteiro estamos, na verdade, aumentando seu valor de forma que aponte para a próxima posição de memória, considerando o tamanho de um inteiro:

```
printf("%p\n", p); // 0xbfd7d060

p++;

printf("%p\n", p); // 0xbfd7d064
```

Aritmética de Ponteiros



- O que há na posição de memória `0xbfd7d064`?
  - Pode ser um inteiro (OK!);
  - Pode não ser um inteiro... (Problemas!).

= Memória

= Variável `contador`

Exemplo de Aritmética de Ponteiros

```
int vetor[5] = {1, 2, 3, 4, 5};
int *p;

// 1, 4, 9, 16, 25,
for (p = &vetor[0]; p <= &vetor[4]; p++)
  printf("%d, ", (*p *= *p));
printf("\n");
```

* = Referência ao endereço apontado por p
* = Operação de multiplicação
* = Referência ao valor da memória no endereço apontado por p

Trívia

- Qual é a diferença entre:

```
p++;        (*p)++;        *(p++);
```

Trívia

- Qual é o valor de y ao final do programa?

```
int y, *p, x;
y = 0;
p = &y;
x = *p;
x = 4;
(*p)++;
x--;
(*p) += x;
printf ("y = %d\n", y);
```

y = 4

?

Ponteiros & Vetores

- Vetores são ponteiros!

Vetores	Ponteiros
`int vetor[10];`	`int *vetor; // + alocação`
`vetor[0]`	`*vetor`
`vetor[5]`	`*(vetor + 5)`

- A notação de vetores é só uma facilidade sintática para o programador. “Por baixo dos panos”, o compilador trata ambos da mesma forma!

Códigos Melhores em Alguns Casos

- Usando índices (2.500 cálculos de deslocamento):

```
float matriz[50][50];
int i, j;
for (i = 0; i < 50; i++)
  for (j = 0; j < 50; j++)
    matriz[i][j] = 0.0;
```

- Usando ponteiro (2.500 incrementos):

```
float matriz[50][50], *p = matriz[0];
int i;
for (i = 0; i < 2500; i++) *p++ = 0.0;
```

Porém, um Vetor não é uma Variável

```
// Algumas coisas eu consigo fazer:
int vetor[5] = {5, 4, 3, 2, 1};
int *ponteiro = vetor;
printf("%d\n", (*vetor *= *vetor)); // 25
printf("%d\n", *(vetor + 1));       // 4

// Outras o compilador não deixa:
vetor = vetor + 1; // Erro!
vetor = ponteiro;  // Erro!

// Ponteiros podem funcionar como vetores:
printf("%d\n", ponteiro[2]); // 3
```

Ponteiros e Strings

```
// Nossa própria versão de strcpy!
void copstr(char *orig, char *dest) {
  while (*orig) *dest++ = *orig++;
  *dest = '\0';
}

int main() {
  char nome[15] = "Dennis Ritchie";
  char string[15];
  copstr(nome, string);
  printf("%s\n", string); // Dennis Ritchie
}
```

Vetores de Ponteiros

- Claro que é perfeitamente possível construir vetores de ponteiros:

```
// 10 ponteiros para inteiro.
int *vetorPonteiros[10];
```

Inicialização de Ponteiros

- Também é possível inicializar ponteiros, da mesma forma que inicializamos qualquer outra variável;
- Na verdade, já vimos isso acontecer em vários exemplos passados:

```
float matriz[50][50], *p = matriz[0];
char *nome = "Dennis Ritchie";

/* Outros exemplos: */
int x = 100, *q = &x, *r = q;
short bs[2] = {1, 2}, *b = &bs[1];
```

Ponteiros para Ponteiros

- É um ponteiro que armazena o endereço de outro ponteiro;
- Também pode existir ponteiro para ponteiro para ponteiro e assim por diante...
- Basta usarmos * várias vezes:

```
float f = 3.1415, *p, **pp;
p = &f;
pp = &p; // &&f ou &(&f) produzem erro!

printf("%f\n", *p);
printf("%f\n", **pp);
```

Regra de Ouro dos Ponteiros

- Se você não sabe para onde o ponteiro está apontando, não use o ponteiro!

```
int x, *p;
x = 13;
*p = x; // Resultado imprevisível!
```

Curso Rápido de C

Parte 5: Funções

Prof. Vítor Souza
Linguagens de Programação

Departamento de Informática
Univ. Federal do Espírito Santo

11/04/2006

Funções

- Um programa em C é um conjunto de funções, incluindo a função main();
- Funções são as estruturas que nos permitem organizar o código. Sem elas, somente os programas muito pequenos não ficariam complexos demais;
- Funções possuem a seguinte forma geral:

```
<tipo> <nome>(<lista de argumentos>) {
  <conjunto de comandos>
}
```

- O tipo de retorno padrão é `int`. Uma função que não retorna dados pode ser declarada como do tipo `void`.

O Comando return

- O comando `return` é usado para sair da função e, opcionalmente, retornar algum valor:

```
float quadrado(float x) {
  return x * x;
}
void imprimePositivo(float x) {
  if (x <= 0) return;
  printf("%f\n", x);
}
int main() {
  imprimePositivo(quadrado(-4.7));
}
```

A Função `main()`

- A função main() é uma função como qualquer outra;
- Seu tipo de retorno é inteiro;
- O valor retornado por ela é disponibilizado ao sistema operacional (são chamados de *exit codes*);
- Há uma convenção na qual 0 (zero) significa término normal e qualquer número diferente de zero (de 1 a 256) significa algum erro;
- Alguns comandos especificam em seus manuais os diferentes códigos de erro existentes. Ao construir scripts, você pode usá-los para dar respostas amigáveis ao usuário;
- Exemplo: no Linux, veja a página de manual do comando `grep`, na seção "*Diagnostics*".

A Função exit()

- Um programa C pode terminar normalmente (sem erros) de duas formas:
  - Em algum ponto da função `main()` há um comando `return <codigo de saída>;`
  - Em qualquer ponto do código há uma chamada à função `exit(<código de saída>)`.

```
#include <stdlib.h> // Para usar exit()!

void verificaCondicoesIniciais() {
  if (erro25()) exit(25);
}
int erro25() { /* ... */ }
```

Declaração x Definição

- Declarar uma função é definir seu nome, tipo de retorno e argumentos (“protótipo” da função);
- Definir uma função é declará-la e ainda fornecer um corpo de código;
- Funções precisam ser declaradas antes de usadas, ou o compilador emitirá um erro em tempo de compilação;
- Funções precisam ser definidas antes da fase de ligação *(linkage)* ou o compilador também emitirá um erro (em tempo de ligação);
- É útil para quando queremos separar nossas funções em arquivos que podem ser “importados”, assim como já fizemos com `stdio.h` e `string.h`.

Declaração x Definição – Exemplo

```
// Declaração (protótipo).
float quadrado(float x);

int main() {
  // Uso da função, só permitido porque
  // ela já foi declarada.
  float f = quadrado(1.44);
}

// Definição necessária em tempo de ligação.
float quadrado(float x) {
  return x * x;
}
```

Arquivos Cabeçalho (*Header Files*)

- Arquivos cabeçalho são arquivos que contém somente declarações (protótipos) de funções;
- Possuem a extensão “`.h`” e já conhecemos dois deles: `stdio.h` e `string.h`;
- As definições das funções das bibliotecas do C, como `stdio` e `string`, estão disponíveis automaticamente para o compilador em tempo de ligação;
- Podemos criar nossas próprias bibliotecas de função, neste caso é preciso especificar ao compilador onde encontrar sua definição depois;
- Para importar arquivos de cabeçalho definidos pelo programador, usamos "" ao invés de <>:

```
#include "minhabiblioteca.h"
```

Usando Arquivos Cabeçalho

- funcoes.h:

```
float quadrado(float x);
```

- funcoes.c:

```
#include "funcoes.h"

float quadrado(float x) {
  return x * x;
}
```

Usando Arquivos Cabeçalho

- programa.c:

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

int main() {
   float f = quadrado(1.44);
   printf("%f\n", f);
}
```

Usando Arquivos Cabeçalho

- Compilação da biblioteca de funções:

```
$ gcc -c -o funcoes.o funcoes.c
```

Compilação de biblioteca de funções: não faz ligação, não exige uma função `main()` e gera arquivo objeto (binário não-executável).

- Compilação do programa principal:

```
$ gcc -o programa funcoes.o programa.c
```

O arquivo objeto é necessário em tempo de ligação!

Usando Arquivos Cabeçalho

- Pode ser compilado de forma mais direta:

```
$ gcc -o programa funcoes.c programa.c
```

- Ou ainda mais indireta:

```
$ gcc -c funcoes.c
$ gcc -c programa.c
$ gcc funcoes.o programa.o
$ ./a.out
```

Gera `funcoes.o`

Gera `programa.o`

Gera `a.out`

Relembrando Escopo de Variáveis

- Variáveis declaradas dentro de uma função (ou bloco qualquer) só existe dentro da mesma e não interfere com variáveis de outras funções:

```
void imprimeValor() {
   int x = 20;
   printf("%d\n", x);
}
int main() {
   int x = 10;
   imprimeValor();      // 20
   printf("%d\n", x);   // 10
}
```

Relembrando Escopo de Variáveis

- Já as variáveis globais são ocultadas se for definida uma variável de mesmo nome numa função:

```
int x = 10;
void imprimeValor() {
   int x = 20;
   printf("%d\n", x);
}
int main() {
   imprimeValor();      // 20
   printf("%d\n", x);   // 10
}
```

Parâmetros de Funções

- Parâmetros (argumentos) de funções são variáveis locais e comportam-se como tal;
- C faz passagem de parâmetro por valor. Alterações no parâmetro não afetam a variável externa:

```
int x = 10;
void imprimeValor(int x) {
  printf("%d\n", ++x);
}
int main() {
  imprimeValor(x);    // 11
  printf("%d\n", x);  // 10
}
```

Simulando Passagem por Referência

- É possível por meio de ponteiros;
- Já vimos isso quando apresentamos `scanf`!
- Ainda consiste em passagem por valor, só que é o valor do ponteiro, que é uma referência à variável:

```
int x = 10;
void imprimeValor(int *x) {
  printf("%d\n", ++(*x));
}
int main() {
  imprimeValor(&x);   // 11
  printf("%d\n", x);  // 11
}
```

Vetores como Argumentos

- Podemos declarar vetores como argumentos de funções de três maneiras diferentes:

```
void funcao(int vetor[10]);
void funcao(int vetor[]);
void funcao(int *vetor);
```

- O resultado é o mesmo: é passado o valor da referência do vetor (ou seja, não é alocada memória local para ele);
- Desta maneira, alterações nos elementos do vetor são vistas por elementos externos à função!

Vetores como Argumentos

- Equivalente ao exemplo com ponteiros:

```
void imprimeValor(int x[]) {
  printf("%d\n", ++x[0]);
}
int main() {
  int x[] = {10};
  imprimeValor(x);       // 11
  printf("%d\n", x[0]); // 11
}
```

Parâmetros da Função `main()`

- A função main() pode ter parâmetros, mas só fará algum sentido a seguinte declaração:

```
int main(int argc, char *argv[]);
```

- Tais parâmetros permitem acesso aos argumentos passados pela linha de comando:
  - `argc` = *argument count* = contador de argumentos;
  - `argv` = *argument values* = valores dos argumentos.

Parâmetros da Função main()

```
// Note que *argv[] equivale a **argv!
int main(int argc, char **argv) {
  int i;
  for (i = 0; i < argc; i++)
    printf("argv[%d] = %s\n", i, argv[i]);
}

$ ./programa 100 slides "sobre C"
argv[0] = ./programa
argv[1] = 100
argv[2] = slides
argv[3] = sobre C
```

O nome do programa também é considerado um argumento!

Funções Recursivas

- São perfeitamente possíveis em C;
- Muito cuidado ao definir o critério de parada para não criar uma recursão infinita (*stack overflow*).

```
// Fatorial é um bom exemplo para recursão.
int fatorial(int n) {
  if (n) return n * fatorial(n – 1);
  else return 1;
}
```

Dicas com Relação à Funções

- Implemente funções da maneira mais genérica possível, para poder reutilizá-la mais vezes;
- Evite o uso de variáveis globais. Em programas com muitas funções você pode perder o controle;
- Se houver uma preocupação exagerada com relação ao desempenho, lembre-se que cada chamada a função consome um pequeno tempo (no entanto, muitos otimizadores realizam substituição *in-line*);
- Use nomes coerentes para suas funções. Muitas vezes podemos entender o que uma função faz somente olhando para seu protótipo, caso seu nome e de seus argumentos forem amigáveis.

Curso Rápido de C

Parte 6: Diretivas de Compilação

Prof. Vítor Souza
Linguagens de Programação

Departamento de Informática
Univ. Federal do Espírito Santo

11/04/2006

Diretivas de Compilação

- Em C, a compilação é dividida em três fases:
  - Pré-processamento;
  - Compilação;
  - Ligação.

- Na primeira fase, o pré-processador C examina o código-fonte e executa certas alterações baseadas em diretivas de compilação:

`#if`	`#ifdef`	`#ifndef`
`#else`	`#elif`	`#endif`
`#include`	`#define`	`#undef`

A Diretiva #include

- Já usamos esta diretiva várias vezes;
- Indica ao compilador que inclua arquivos de cabeçalho para a compilação de um código-fonte;
- Usamos <> para especificar bibliotecas do sistema (arquivos presentes nos caminhos de procura pré-especificados pelo compilador);
- Usamos "" para especificar bibliotecas criadas pelo programador (arquivos no mesmo diretório ou especificação de caminho completo).

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

As Diretivas #define e #undef

- Permite a criação de macros e constantes;
- Onde o pré-processador encontrar a macro, ele substituirá pelo texto especificado:

```
#define PI 3.141592
#define MAX(A, B) ((A > B) ? (A) : (B))

int main() {
  float area1 = PI * (5.0 * 5.0);  // 78.5
  float area2 = PI * (7.5 * 7.5);  // 176.7
  float maior = MAX(area1, area2); // 176.7
}
```

As Diretivas #define e #undef

- Este é o código gerado para compilação:

```
int main() {
   float area1 = 3.141592 * (5.0 * 5.0);
   float area2 = 3.141592 * (7.5 * 7.5);
   float maior = ((area1 > area2) ? (area1) :
  (area2));
}
```

As Diretivas #define e #undef

- As ocorrências da macro no código são substituídas pelo conteúdo da mesma e não é feita nenhuma verificação;
- Portanto, temos que ter cuidado para não gerarmos erros de compilação ou de lógica, como a seguir:

```
#define QUADRADO(X) X * X

int main() {
  int a = 5, b = 3;
  // Substituído por: int c = A + B * A + B!
  int c = QUADRADO(a + b);
}
```

As Diretivas #define e #undef

- Convenção de código:
  - Constantes e macros devem ser escritas com todas as letras maiúsculas e com "_" no lugar de espaços, para diferenciá-las das variáveis normais:
  - Ex.: `#define VALOR_MAXIMO 50`

- A diretiva `#undef`:
  - Serve para remover uma definição:

```
#define PI 3.141592
#undef PI
```

As Diretivas #if, #else e #endif

- Diretiva de compilação condicional – funciona como o comando `if`:
  - Executa as diretivas entre `#if` e `#else` caso a condição seja verdadeira, entre `#else` e `#endif` caso contrário;
  - A condição é uma expressão constante, não pode conter variáveis. O `#else` é opcional.

```
#if SISTEMA == DOS
  #define CABECALHO biblioteca_dos.h
#else
  #define CABECALHO biblioteca_unix.h
#endif
#include CABECALHO
```

As Diretivas #ifdef e #endif

- Caso específico de diretiva condicional;
- Executa a seqüência de declarações entre `#ifdef` e `#endif`, caso a expressão em `#ifdef` esteja definida;
- A diretiva `#else` pode ser usada.

```
#define PI 3.141592

/* ... */

#ifdef PI
  #undef PI
  #include "trigonometria.h"
#endif
```

A Diretiva #ifndef

- Contrário de `#ifdef`, funciona exatamente como ele:

```
#ifndef PI
  #define PI 3.141592
#endif
```

A Diretiva #elif

- Conjunção de #else e #if, funciona para implementar uma estrutra de muitos casos:

```
#if SISTEMA == DOS
  #define CABECALHO biblioteca_dos.h
#elif SISTEMA == UNIX
  #define CABECALHO biblioteca_unix.h
#elif SISTEMA = BSD
  #deinfe CABECALHO biblioteca_bsd.h
#else
  #define CABECALHO biblioteca_generica.h
#endif
```

Curso Rápido de C

Parte 7: Entrada e Saída

Prof. Vítor Souza
Linguagens de Programação

Departamento de Informática
Univ. Federal do Espírito Santo

11/04/2006

Fluxos

- I/O em C é realizada por meio de fluxos, independente do meio (teclado/tela, arquivo, impressora, etc.);
- Para abrir/fechar e operar fluxos, utilizamos funções específicas das bibliotecas do C;
- Para cada meio, pode haver diversas funções, estudaremos neste curso somente as principais;
- Fluxos padrão:
  - Entrada = teclado, saída = tela;
  - Abertos automaticamente;
  - Leitura feita com `scanf`, escrita com `printf` (já vistos).

Formatos de Impressão

- Já vimos os formatos mais comuns para impressão com `printf`. Vejamos algumas possibilidades de formatação de números inteiros (%d) e reais (%f):

- %<t>d = inteiro com tamanho mínimo t:

```
printf("[%5d]\n", 12);       // [   12]
printf("[%5d]\n", 12345678); // [12345678]
```

- %0<t>d = idem, preenchendo com 0 os espaços:

```
printf("[%05d]\n", 12);       // [00012]
printf("[%05d]\n", 12345678); // [12345678]
```

Formatos de Impressão

- %-<t>d = alinhamento à esquerda:

```
printf("[%-5d]\n", 12);       // [12   ]
printf("[%-5d]\n", 12345678); // [12345678]
```

- %<t>.<d>f = real, com tamanho mínimo t e d casas decimais (o número é arredondado):

```
printf("[%8.8f]\n", PI); // [3.14159200]
printf("[%8.4f]\n", PI); // [  3.1416]
printf("[%8.2f]\n", PI); // [    3.14]
printf("[%8.0f]\n", PI); // [       3]
```

- Funcionam para toda a “família printf/scanf”. Mais informações em seus manuais.

Strings como Fluxos: `sscanf` e `sprintf`

- Assim como teclado/tela, strings podem ser usadas como fluxos de I/O;
- Usaremos as funções `sscanf` e `sprintf` para leitura/escrita em strings;
- São idênticas a `scanf` e `printf`, só que especificando a string (`char *`) de entrada/saída:

```
int sprintf(char *s, const char *formato, ...);
int sscanf(char *s, const char *formato, ...);
```

- São muito utilizadas para conversão de números para strings e vice-versa;
- Assim como suas irmãs, estão definidas no `stdio.h`.

Strings como Fluxos: sscanf e sprintf

```
char string[100];
float f;

// Escreve um número numa string:
f = 3.141592;
sprintf(string, "f = %1.1f!", f);
printf("%s\n", string); // f = 3.1!

// Converte de string para número:
strcpy(string, "3.141592");
sscanf(string, "%f", &f);
printf("%1.4f\n", f); // 3.1416
```

I/O em Arquivos

- Funções e tipos para manipulação de arquivos também são definidos em `stdio.h`;
- C define o tipo (não-primitivo) FILE. Uma variável FILE é um descritor de arquivo (ou fluxo de arquivo);
- As funções de manipulação trabalham com ponteiros para FILE;
- Há funções para abrir, fechar, escrever, ler e se movimentar em arquivos;
- Portanto, para trabalhar com arquivos, precisamos usar:

```
FILE *arquivo;
```

Abrir Arquivos: fopen

- A função `fopen()` abre um arquivo:

```
FILE *fopen(char *nomeArquivo, char *modo);
```

  - `nomeArquivo` contém o nome ou caminho completo para o arquivo a ser aberto;
  - `modo` define o tipo de uso que vai se fazer do arquivo.

- `fopen()` retorna o ponteiro para o descritor do arquivo aberto, ou `0` (nulo) caso algum erro tenha ocorrido.

Abrir Arquivos – Modos de Uso

“r”	Abre um arquivo texto para leitura. O arquivo deve existir antes de ser aberto.
“w”	Abre um arquivo texto para gravação. Se o arquivo não existir, ele será criado; se já existir, o conteúdo anterior será destruído.
“a”	Abre um arquivo texto para gravação. Os dados serão adicionados no fim do arquivo ("append") se ele já existir. Um novo arquivo será criado no caso dele não existir anteriormente.
“rb”	Abre um arquivo binário para leitura. Igual ao modo "r" anterior, só que o arquivo é binário.
“wb”	Cria um arquivo binário para escrita, como no modo "w" anterior, só que o arquivo é binário.
“ab”	Acrescenta dados binários no fim do arquivo, como no modo "a" anterior, só que o arquivo é binário.
“r+”	Abre um arquivo texto para leitura e gravação. O arquivo deve existir e pode ser modificado.
“w+”	Cria um arquivo texto para leitura e gravação. Se o arquivo existir, o conteúdo anterior será destruído. Se não existir, será criado.
“a+”	Abre um arquivo texto para gravação e leitura. Os dados serão adicionados no fim do arquivo se ele já existir, ou um novo arquivo será criado, no caso do arquivo não existir anteriormente.
“r+b”	O mesmo que "r+", só que o arquivo é binário.
“w+b”	O mesmo que "w+", só que o arquivo é binário.
“a+b”	O mesmo que "a+", só que o arquivo é binário.

Abrir Arquivos – Exemplo

```
// Declara o ponteiro para o descritor.
FILE *arquivo;

// Abre o arquivo exemplo.txt, que
// encontra-se no diretório corrente:
arquivo = fopen("exemplo.txt", "w");

// Verifica se retornou nulo (erro).
if (! arquivo) {
  printf("Erro: ... \n");
  exit(1);
}
```

Fechar Arquivos: `fclose`

- Devemos fechar os arquivos que abrimos:
  - Por questões de eficiência, os dados escritos são armazenados num buffer de memória e levados ao arquivo somente de tempos em tempos;
  - Fechar um arquivo escreve o restante do buffer no arquivo. Se não o fizermos, podemos perder dados!
- A função `fclose()` fecha um arquivo:

```
int fclose(FILE *arquivo);
```

- Exemplo:

```
fclose(arquivo); // Retorna 0 se tudo OK!
```

I/O em Arquivos: `fscanf` e `fprintf`

- Mais dois membros da família scanf/printf;
- Funcionam iguais aos seus irmãos, só que lêem/escrevem em arquivos:

```
int fprintf(FILE *f, const char *formato, ...);
int fscanf(FILE *f, const char *formato, ...);
```

Fim de Arquivos: `feof`

- Quando lemos de arquivos, é interessante saber quando o mesmo terminou (*end of file*);
- Para isso, usamos a função `feof()`:

```
int feof(FILE *f);
```

- A função `feof()` retorna `0` (falso) se não é o fim do arquivo e não-zero (verdadeiro) se o fim do arquivo foi encontrado.

Exemplo: Leitura Linha por Linha

```
FILE *arquivo; char linha[1000]; int num;
arquivo = fopen("exemplo.txt", "r");
if (! arquivo) {printf("Erro!\n"); exit(1);}

num = 0;
while (! feof(arquivo)) {
  // Note: não usamos o 2º \n com scanf()!
  // Obs.: não lê linhas vazias!
  fscanf(arquivo, "%[^\n]\n", linha);
  printf("(%3d) %s\n", ++num, linha);
}
fclose(arquivo);
```

Leitura/Escrita Binária: `fread` e `fwrite`

- Se abrirmos um arquivo para I/O binário, podemos ler blocos de dados (bytes) com `fread()`:

```
unsigned fread(void *buffer, int numbytes,
  int qtd, FILE *arquivo);
```

  - `buffer` = onde serão armazenados os dados lidos;
  - `numbytes` = tamanho da estrutura a ser lida;
  - `qtd` = indica quantas estruturas serão lidas;
  - `arquivo` = descritor do arquivo de onde leremos os dados.

Leitura/Escrita Binária: `fread` e `fwrite`

- `fwrite()` funciona exatamente como `fread()`, só que escreve no arquivo:

```
unsigned fwrite(void *buffer, int numbytes,
  int qtd, FILE *arquivo);
```

- Tanto `fwrite()` como `fread()` retornam 1 se tudo ocorreu sem problemas.

Leitura/Escrita Binária: fread e fwrite

```
FILE *pf; float pi = 3.1415, float pi2;

pf = fopen("arq.bin", "wb");
if(fwrite(&pi, sizeof(float), 1, pf) != 1)
  printf("Erro na escrita do arquivo");
fclose(pf);

pf = fopen("arq.bin", "rb")
if(fread(&pi2, sizeof(float), 1, pf) != 1)
  printf("Erro na leitura do arquivo");
printf("\npi2 = %f", pi2);
fclose(pf);
```

Movimentar-se: `fseek` e `rewind`

- Imagine um SGBD:
  - Que armazene em arquivos tabelas de dados muito grandes (centenas ou milhares de MegaBytes);
  - Que precise ler numa destas tabelas um dado que está no final do arquivo, depois um no começo, etc.;
  - A leitura sequencial atende?
- Nestes casos, precisamos de funções que mude a "posição atual" do cursor (fluxo de leitura/escrita):
  - `fseek()`: posiciona o cursor em um ponto específico;
  - `rewind()` posiciona o cursor no início do arquivo.

```
int fseek(FILE *arq, long qtd, int origem);
void rewind(FILE *arq);
```

Movimentar-se: fseek e rewind

- `fseek(arq, qtd, origem)`:
  - `arq` = descritor do arquivo;
  - `qtd` = número de bytes representando o tamanho do deslocamento;
  - `origem` = de onde iremos nos delocar. Os valores aceitos encontram-se na tabela abaixo:

Nome	Valor	Significado
SEEK_SET	0	Início do arquivo
SEEK_CUR	1	Ponto corrente no arquivo
SEEK_END	2	Fim do arquivo

* Constantes definidas em `stdio.h`

Tratar Erros: `ferror`, `perror` e `clearerr`

- Quando acessamos arquivos, há uma quantidade enorme de situações de erro possíveis:
  - O usuário não possui permissões;
  - O espaço em disco é insuficiente;
  - Há um erro físico no disco;
  - Etc...
- A função `ferror(FILE *arq)` retorna 0 (zero) se nenhum erro ocorreu durante o acesso a um arquivo;
- A função `clearerr(FILE *arq)` limpa os erros relativos a um determinado arquivo;
- A função `perror(char *msg)` imprime uma mensagem junto com o erro reportado pelo programa.

Tratar Erros: ferror, perror e clearerr

```
arquivo = fopen("exemplo.txt", "rw");
if (! arquivo) printf("Erro!\n");

fprintf(arquivo, "Arquivo é read-only!");
if (ferror(arquivo))
  perror("O seguinte erro ocorreu");


$ chmod 444 exemplo.txt
$ ./programa
O seguinte erro ocorreu: Bad file descriptor
```

Excluir Arquivos: `remove`

- Para excluir um arquivo, usamos a função `remove()`:

```
int remove(char *nomeArquivo)
```

- A função especifica o nome do arquivo, e não um descritor.

Fluxos Pré-Definidos

- Existem alguns fluxos pré-definidos pelo C:
  - `stdin`: dispositivo de entrada padrão (*standard input*, geralmente o teclado);
  - `stdout`: dispositivo de saída padrão (*standard output*, geralmente o vídeo);
  - `stderr`: dispositivo de saída de erros padrão (*standard error*, geralmente o vídeo).
- São como descritores de arquivo e podem ser usados em funções como `fscanf()` e `fprintf()`;
- No Linux, podemos configurar saídas padrão e erro quando executamos um programa:

```
./programa 1>padrao.txt 2>erro.txt
```

Curso Rápido de C

Parte 8: Tipos de Dados Avançados

Prof. Vítor Souza
Linguagens de Programação

Departamento de Informática
Univ. Federal do Espírito Santo

11/04/2006

Recapitulando...

- Já vimos vários conceitos sobre tipos de dados:
  - Tipos de dados primitivos (`int`, `float`, ...);
  - Alguns modificadores (`short`, `long`, `unsigned`, ...);
  - Tamanho;
  - Declaração;
  - Escopo;
  - Espaços de nomes;
  - Constantes;
  - Vetores e Strings;
  - Ponteiros para tipos de dados primitivos.

- Veremos alguns conceitos mais avançados sobre tipos de dados...

Outros Modificadores de Primitivos

- Além de `short`, `long`, `signed` e `unsigned`, existem outros quatro modificadores de variáveis de tipos primitivos:
  - `const`;
  - `volatile`;
  - `static`;
  - `register`.

O Modificador const

- Usado para definir variáveis que não podem ser alteradas depois de inicializadas (constantes):

```
const float pi = 3.141592;
```

- Constantes podem ser definidas com `#define`;
- `const` é mais usado na definição de parâmetros:

```
int sqr(const int *num) {
  // *num = 10; geraria erro de compilação!
  return ((*num) * (*num));
}
```

O Modificador `volatile`

- Indica ao compilador que uma determinada variável é volátil, ou seja, pode ser alterada por processos externos sem que isso seja avisado;
- Declarar uma variável volátil como `volatile` evita vários problemas.

O Modificador `static`

- Variáveis globais marcadas como `static` não podem ser usadas fora de seu módulo;
- Variáveis locais marcadas como `static` mantém seu valor mesmo depois que a função termina:

```
int count() {
  static int num = 0;
  return ++num;
}
int main() {
  printf("%d\n", count()); // 1
  printf("%d\n", count()); // 2
}
```

O Modificador register

- Diz ao compilador que a variável em questão deve ser, se possível, armazenada em um registrador:
  - Aumenta a velocidade de acesso à variável;
  - É um pedido que pode não ser atendido;
  - Cuidado ao usar (ex.: variáveis globais);
  - Muitas vezes é melhor deixar o otimizador fazer este tipo de escolha.

```
register int count;
for (count = 0; count < 1000000; count++) {
  /* ... */
}
```

Ponteiros para Função

- Além de ponteiros para tipos primitivos, é possível também criar ponteiros para função;
- Desta forma, podemos passar funções como parâmetros e executá-las (cidadãs de 1ª classe);
- Declaração:

```
<tipo> (*<nome>)();

// Declaração com especificação dos
// parâmetros (não é obrigatória).
<tipo> (*<nome>)(<tipo>, <tipo>, ...);
```

Ponteiros para Função – Exemplo

```
void executa(char * (*f)(), char *p1, char *p2) {
  f(p1, p2);
}
int main() {
  char *nome = "Dennis Ritchie";
  char string[50];
  char * (*funcao)(); // (char *, const char *) opc.
  funcao = strcpy;
  executa(funcao, string, nome);
  printf("%s\n", string); // Dennis Ritchie
  funcao = strcat;
  executa(funcao, string, " criou C!");
  printf("%s\n", string); // Dennis Ritchie criou C!
}
```

Ponteiros para Função

- São como ponteiros para variáveis;
- Os nomes das funções são referências aos endereços de memória onde estão suas implementações:

```
// 0x8048350 0x8048330
printf("%p %p\n", strcpy, strcat);
```

- Temos que ter os mesmos cuidados que temos com ponteiros para variáveis:

```
funcao = strcpy;
funcao += 100;
funcao(string, nome); // Segmentaton fault.
```

Alocação Dinâmica

- Quando declaramos variáveis, uma quantidade de memória suficiente é automaticamente alocada;
- No entanto, podemos não saber, de antemão, quanto espaço precisaremos (ex.: vetor);
- C permite que façamos alocação dinâmica: solicitação de memória em tempo de execução;
- Para isso usamos ponteiros e três funções da biblioteca `stdlib.h`:
  - `malloc`: alocação (separação da memória);
  - `realloc`: realocação (aumento da memória alocada);
  - `free`: desalocação (liberação da memória alocada).

A Função malloc()

- Aloca o número de bytes informados e retorna um ponteiro para o início da memória alocada:

```
void *malloc(unsigned int qtd);
```

- Para calcular o número de bytes que desejamos, usaremos sizeof(). Depois, precisamos fazer o *cast*:

```
int *vet = (int *)malloc(10 * sizeof(int));
```

A coerção é obrigatória.

Tamanho de 1 inteiro

Tamanho de 10 inteiros

A Função malloc()

```
int *vetor, qtd, i;

printf("Quantidade: ");
scanf("%d", &qtd);
vetor = (int *)malloc(qtd * sizeof(int));

printf("Digite %d número(s):\n", qtd);
for (i = 0; i < qtd; i++)
  scanf("%d", &vetor[i]);

printf("\nNúmero(s) ao contrário:\n");
for (i = qtd - 1; i >= 0; i--)
  printf("%d\n", vetor[i]);
```

A Função realloc()

- Se alocamos uma quantidade de espaço e depois percebemos que não foi o suficiente, podemos alocar mais com `realloc()`:
  - Se houver mais espaço em seqüência, este é alocado e o endereço se mantém;
  - Se não houver, é alocado outro espaço e os dados são copiados automaticamente.

```
void *realloc(void *ptr, unsigned int num);
```

- Tanto `malloc()` quanto `realloc()` retornam `0` se não há memória suficiente para a alocação.

A Função realloc()

```
int *vetor, qtd = 3, num = 1, i = 0;
vetor = (int *)malloc(qtd * sizeof(int));
printf("Digite positivos, 0 para sair.\n");

while (num != 0) {
  scanf("%d", &num);
  vetor[i++] = num;
  if (i == qtd) {
    qtd *= 2;
    vetor = realloc(vetor, qtd * sizeof(int));
  }
}
```

A Função free()

- Serve para liberar memória alocada previamente;
- Toda área de memória que não é mais útil deve ser liberada com free():

```
void free(void *ptr);
```

- O constante “esquecimento” desta regra poderá causar ao programador diversos erros por falta de memória.

```
// Como fazer seu sistema suar:
while (1) malloc(100000);
```

A Função free()

```
int *vetor, qtd, i;

printf("Quantidade: ");
scanf("%d", &qtd);
vetor = (int *)malloc(qtd * sizeof(int));

/* Mesmo exemplo do malloc()... */

// Ao final, liberar a memória.
free(vetor);
```

Alocação de Matrizes

```
int i;
float **matriz, L = 4, C = 5;

// Aloca "L" vetores.
matriz = (float **)malloc(L *
  sizeof(float *));

// Aloca "C" floats em cada vetor.
for (i = 0; i < L; i++)
   matriz[i] = (float *)malloc(C *
    sizeof(float));

// Libera-se a memória de forma semelhante.
```

Tipos de Dados Compostos

- O programador C pode:
  - Criar produtos cartesianos (estruturas) com `struct`;
  - Criar uniões com `union`;
  - Criar enumerações com `enum`;
  - Definir um novo tipo de dado com `typedef`.

Definindo Estruturas

- Para produtos cartesianos, usa-se `struct`:

```
struct <nome> {
  <declaração de variáveis>
  ...
  <declaração de variáveis>
} <var>, <var>, ...;
```

- Onde:

```
<declaração de variáveis> =
  <tipo> <nome>, <nome>, ...;
```

Definindo Estruturas

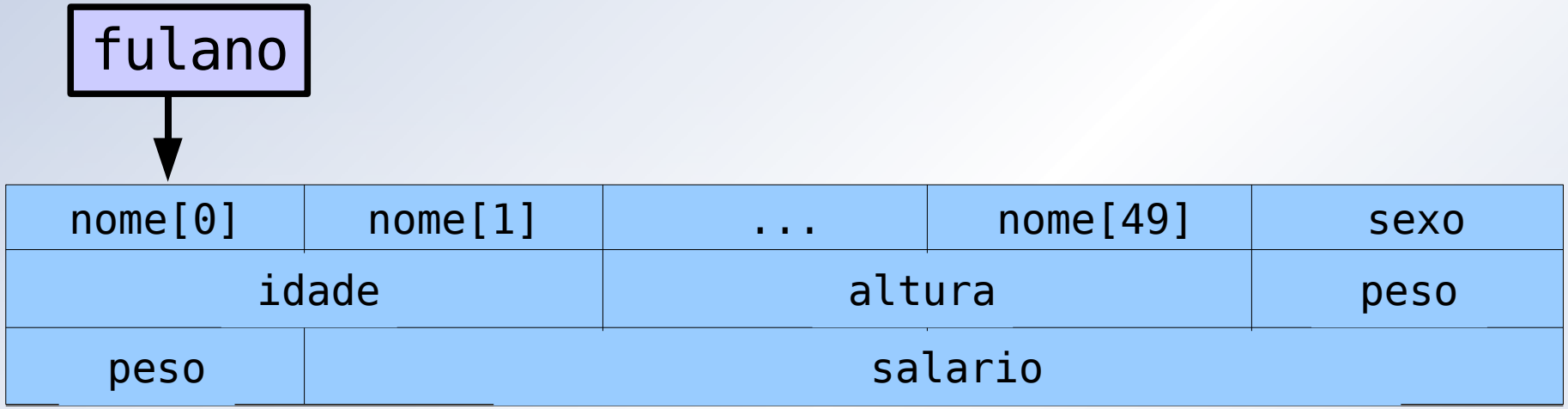
Nome do novo tipo de dados

```
struct pessoa {
   char nome[50], sexo;
   short int idade, altura, peso;
   float salario;
} fulano, beltrano, cicrano;
```

Componentes da estrutura

Declaração de variáveis deste novo tipo (opcional).

Representação em Memória



- Cada variável declarada como do novo tipo `pessoa` ocupa 61 bytes:
  - 50 bytes para o vetor de 50 chars;
  - 1 byte para uma variável do tipo char;
  - 6 bytes para 3 variáveis do tipo short int;
  - 4 bytes para uma variável do tipo float.

Declarando Variáveis `struct`

- Os novos tipos de dados criados são declarados da mesma forma que os antigos, usando seu nome;
- Seu nome é composto pela palavra reservada `struct` e o nome dado à estrutura:

```
struct pessoa p1, p2, p3;
struct pessoa *ponteiroPessoa;
struct pessoa vetorPessoa[50];
```

Estruturas Compostas por Estruturas

- Estruturas podem ser compostas por quaisquer tipos de dados, sejam eles primitivos ou compostos:

```
struct competidor {
   struct pessoa pessoa;
   char equipe[50];
   float potenciaMotor;
};
```

Usando Estruturas

- Uma vez declaradas variáveis de um tipo estrutura, acessamos seus componentes internos com `.`:

```
strcpy(p1.nome, "Vítor Souza");
p1.idade = 24;
p1.altura = 180;
p1.peso = 60;
p1.salario = 100000.0;

// Acesso à estrutura dentro de estrutura:
struct competidor comp;
comp.pessoa.idade = 30;
```

Atribuindo Estruturas

- Atribuição de estruturas funciona da mesma forma que tipos primitivos:

```
strcpy(p1.nome, "Vitor Souza");
p1.idade = 24;
/* Igual ao exemplo anterior... */

p2 = p1; // Os valores de p1 são copiados!
p1.idade = 25;

// 25, 24
printf("%d, %d\n", p1.idade, p2.idade);
```

Cuidado com Estruturas com Ponteiros

```
struct palavra { char *str; int tam; } p1, p2;

p1.str = (char *)malloc(50 * sizeof(char));
strcpy(p1.str, "Dennis Ritchie");
p1.tam = 14;

p2 = p1;
strcat(p2.str, " criou C!");
p2.tam = 21;

// Dennis Ritchie criou C!, 14
printf("%s, %d\n", p1.str, p1.tam);
// Dennis Ritchie criou C!, 21
printf("%s, %d\n", p2.str, p2.tam);
```

Passando Estruturas como Parâmetros

- Também da mesma forma que fazemos com tipos primitivos:

```
void imprime(struct pessoa p) {
  printf("%s tem %d anos de idade.\n",
    p.nome, p.idade);
}

int main() {
  /* Igual aos exemplos anteriores... */
  imprime(p1);
}
```

Passagem de Parâmetros é por Valor!

```
void aniversario(struct pessoa p) {
  p.idade++;
}

int main() {
  /* Igual aos exemplos anteriores... */

  printf("%d\n", p1.idade); // 24
  aniversario(p1);
  printf("%d\n", p1.idade); // 24 !!!
}
```

Ponteiros para Estruturas

- Na maioria das vezes, trabalhamos não com as estruturas, mas com ponteiros para elas:

```
struct pessoa p1;
struct pessoa *p;

strcpy(p1.nome, "Vitor Souza");
p1.idade = 24;
/* Igual aos exemplos anteriores... */

p = &p1;
```

Usando Ponteiros para Estruturas

- O acesso aos componentes da estrutura pode ser feito de forma tradicional ou com ->:

```
// Tradicional: derreferenciando.
printf("%s\n", (*p).nome);

// Com o operador ->.
printf("%s\n", p->nome);
```

Alocação Dinâmica

- Feita da mesma forma:

```
p = (struct pessoa *)malloc(sizeof(struct
  pessoa));
```

Definindo Novo Tipo de Dados

- O uso de ponteiros para estrutura é tão mais comum do que a estrutura em si, que normalmente definimos o ponteiro para a estrutura como um novo tipo:

```
struct TPessoa {
   char nome[50];
   int idade;
};
typedef struct TPessoa *Pessoa;
```

Definindo Novo Tipo de Dados

- O novo tipo Pessoa é um ponteiro para a estrutura:

```
void imprime(Pessoa p) {
  printf("%s: %d anos.\n", p->nome, p->idade);
}

int main() {
  Pessoa p;
  p = (Pessoa)malloc(sizeof(struct TPessoa));
  strcpy(p->nome, "Vítor Souza");
  p->idade = 24;
  imprime(p); // Vítor Souza: 24 anos.
}
```

O Operador typedef

- Como pudemos perceber, `typedef` cria um novo tipo de dados baseado em outro:

```
typedef <tipo existente> <novo tipo>;
```

- Pode servir simplesmente para dar um novo nome a um tipo que já existe:

```
typedef int inteiro;
typedef Pessoa SerHumano;
```

Uso Conjunto de `struct` e `typedef`

- Podemos combinar a declaração da `struct` com o operador `typedef`, desta forma:

```
typedef struct TPessoa {
   char nome[50];
   int idade;
} *Pessoa;
```

Definindo Uniões

- Para uniões, usa-se `union`:

```
union <nome> {
  <declaração de variáveis>
  ...
  <declaração de variáveis>
} <var>, <var>, ...;
```

- Onde:

```
<declaração de variáveis> =
  <tipo> <nome>, <nome>, ...;
```

Como Funciona uma União?

- É separado um único espaço de memória para todos os componentes da união (do tamanho do maior);
- Devemos usar somente um de cada vez:

```
union angulo {
   float graus;
   float radianos;
} a1;

a1.graus = 90;
printf("%f\n", a1.graus); // 90.000000
```

Cuidado com Uniões

```
union angulo {
   int graus;
   float radianos;
} a1;

a1.graus = 90;
printf("%f\n", a1.radianos); // 00.000000
```

- A representação de um inteiro é bem diferente da representação de um ponto-flutuante. Os resultados são imprevisíveis...

Definindo Enumerações

- Para enumerações, usa-se enum:

```
enum <nome> {
   <valor>
   ...
   <valor>
} <var>, <var>, ...;
```

- Dentro das chaves define-se os valores possíveis do novo tipo de dados.

Usando Enumerações

```
enum dias { dom, seg, ter, qua, qui, sex,
  sab } dia;

int main() {
  dia = seg;
  if ((dia == dom) || (dia == sab))
    printf("Não trabalhar\n");
  else
    printf("Trabalhar\n");
}
```

Correspondência Numérica

- Os valores dados ao tipo enumerado são correspondentes aos inteiros 0, 1, 2, ...;

```
dia = dom;
printf("%d\n", dia); // 0

// Podemos fazer comparações:
if (dia < qua)
  printf("Ainda não chegou na quarta.\n");
else
  printf("Já passou da quarta.\n");
```

Curso Rápido de C

Parte 9: Tipos Abstratos de Dados

Prof. Vítor Souza
Linguagens de Programação

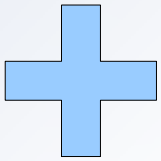
Departamento de Informática
Univ. Federal do Espírito Santo

11/04/2006

Definição

- Um TAD é um tipo composto definido pelo usuário que satisfaz às seguintes condições:
  - Está completamente definido em uma única unidade sintática (podendo ser baseado em outros TADs);
  - A representação interna do TAD não é visível às outras unidades sintáticas do programa;
  - As operações sobre os objetos são aquelas oferecidas na declaração do TAD.

TADs em C



Arquivo cabeçalho

Declarações

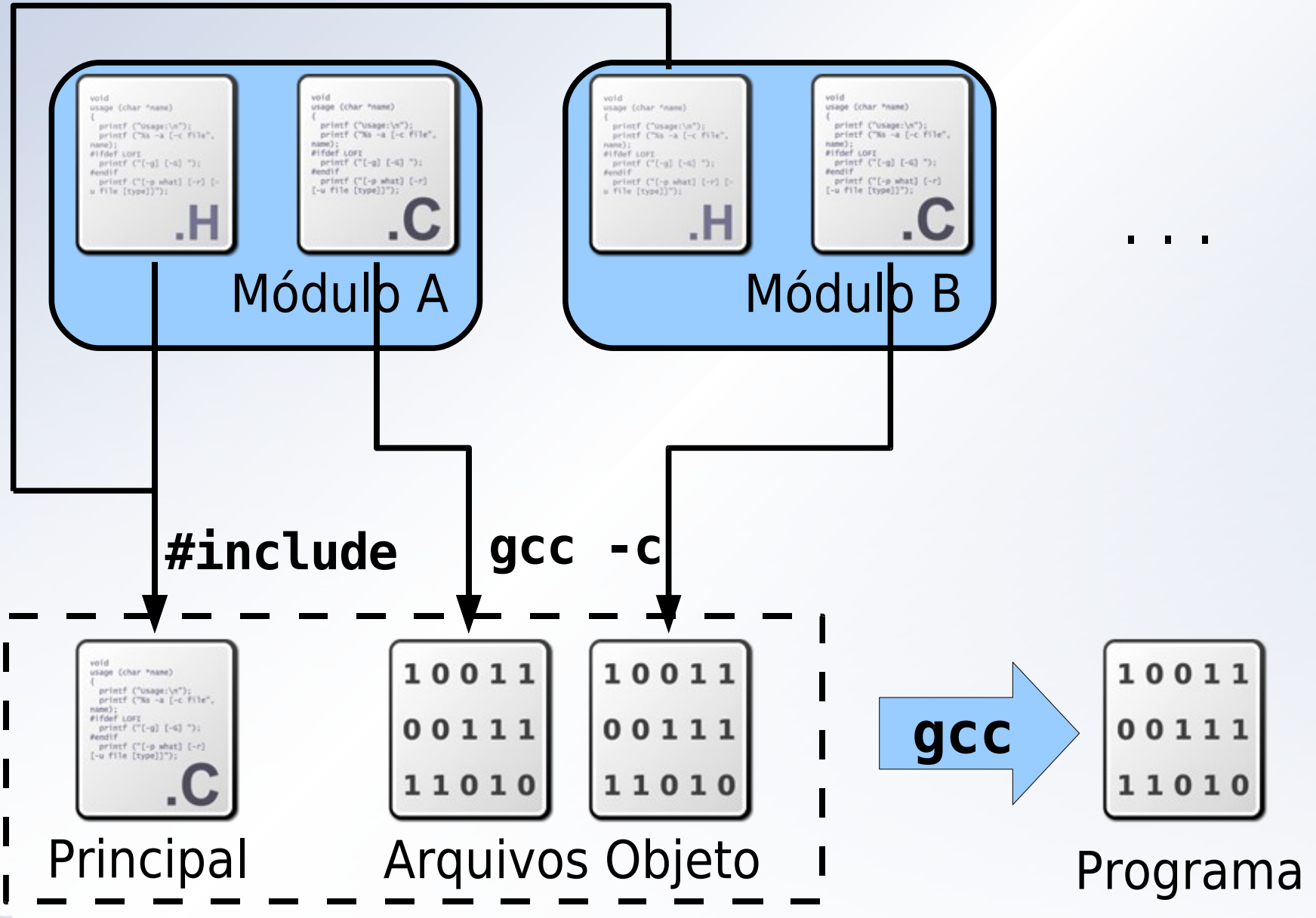
Importado por outros módulos

Arquivo de código-fonte

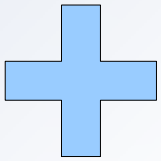
Definições

Compilado e ligado a outros módulos compilados

TADs em C


.H
.C
Módulo A
.H
.C
Módulo B
. . .
#include
gcc -c
.C
10011
00111
11010
10011
00111
11010
gcc
10011
00111
11010
Principal
Arquivos Objeto
Programa

Exemplo



`PilhaVetor.h`

Declarações:
Tipo Pilha
criarPilha
empilharPilha
desempilharPilha
destruirPilha

`PilhaVetor.c`

Definição do tipo Pilha

Implementação de todas as operações.

Exemplo - PilhaVetor.h

```
#ifndef __PilhaVetor_h
#define __PilhaVetor_h

typedef void * PilhaVetor;

PilhaVetor criarPilhaVetor(int);
void empilharPilhaVetor(PilhaVetor, char *);
char * desempilharPilhaVetor(PilhaVetor);
void destruirPilhaVetor(PilhaVetor);

#endif
```

Evita repetição.

Oculta o tipo.

Assinaturas (protótipos) das funções do TAD.

Exemplo - PilhaVetor.c

```c
#include <stdlib.h>
#include "PilhaVetor.h"

typedef struct TPilhaVetor {
   char ** elems;
   int topo;
} * PilhaVetorImpl;

PilhaVetor criarPilhaVetor(int tamanho) {
   // ...
}
```

Garante definição correta das funções.

Definição oculta da estrutura interna do tipo.

Exemplo - PilhaVetor.c

```
void empilharPilhaVetor(PilhaVetor pilha,
  char * elem) {
  // ...
}

char * desempilharPilhaVetor(PilhaVetor
  pilha) {
  // ...
}

void destruirPilhaVetor(PilhaVetor pilha) {
  // ...
}
```

Nomes das funções possuem o nome do tipo para não haver conflito.

Objeto do tipo é sempre passado como parâmetro.

Exemplo – criação e destruição

```
PilhaVetor criarPilhaVetor(int tamanho) {
  PilhaVetorImpl pilha = (PilhaVetorImpl)
    malloc(sizeof(struct TPilhaVetor));
  pilha->elems = (char **)malloc(tamanho *
    sizeof(char *));
  pilha->topo = 0;
  return pilha;
}
void destruirPilhaVetor(PilhaVetor pilha) {
  PilhaVetorImpl pilhaImpl =
    (PilhaVetorImpl)pilha;
  free(pilhaImpl->elems);
  free(pilhaImpl);
}
```

Conversão implícita, automática.

Conversão explícita para acesso interno.

Exemplo – empilhar e desempilhar

```
void empilharPilhaVetor(PilhaVetor pilha,
  char * elem) {
 PilhaVetorImpl pilhaImpl =
                          (PilhaVetorImpl)pilha;
 pilhaImpl->elems[pilhaImpl->topo++] = elem;
}
char * desempilharPilhaVetor(PilhaVetor
  pilha) {
  PilhaVetorImpl pilhaImpl =
                          (PilhaVetorImpl)pilha;
  if (pilhaImpl->topo > 0) return pilhaImpl
                    ->elems[--pilhaImpl->topo];
  else return NULL;
}
```

Compilação automática com `make`

- `make` é uma ferramenta de compilação automática;
- Configura-se os passos de compilação num arquivo chamado `Makefile`;
- Características:
  - Definição de alvos;
  - Dependências entre alvos e arquivos;
  - Compilação seletiva dos arquivos alterados;
  - Sintaxe bastante inflexível.

```
$ make
gcc -c PilhaVetor.c
gcc -o TestePilhaVetor PilhaVetor.o TestePilhaVetor.c
```

Compilação automática com make

```
all: TestePilhaVetor

TestePilhaVetor: PilhaVetor.o TestePilhaVetor.c
	gcc -o TestePilhaVetor PilhaVetor.o
	  TestePilhaVetor.c

PilhaVetor.o: PilhaVetor.c
	gcc -c PilhaVetor.c

clean:
	@rm -f *.o TestePilhaVetor
```

Dependências.

Comandos.

Tabulação.

Exercício

- Escreva um novo TAD ListaSimples, implementando uma lista com encadeamento simples;
- Escreva um novo TAD Pilha, implementando novamente a pilha usando a ListaSimples como infra-estrutura.

Obs.: Não é necessário implementar os métodos!