BCC202 - Estruturas de Dados I

Aula 02: Princípios da Programação em C

Pedro Silva

Universidade Federal de Ouro Preto, UFOP
Departamento de Computação, DECOM
Email: **silvap@ufop.edu.br**

Conteúdo

Conteúdo

Construções Fundamentais

A linguagem C provê as construções fundamentais de fluxo de controle necessárias para programas bem estruturados:

- agrupamentos de comandos;
- tomadas de decisão (*if-else*);
- laços com testes de encerramento no início (*while*, *for*)
- laços com testes de encerramento no fim (*do-while*);
- seleção de um dentre um conjunto de possíveis casos (*switch*);
- acesso a apontadores e a habilidade de fazer aritmética com endereços.

Por outro lado

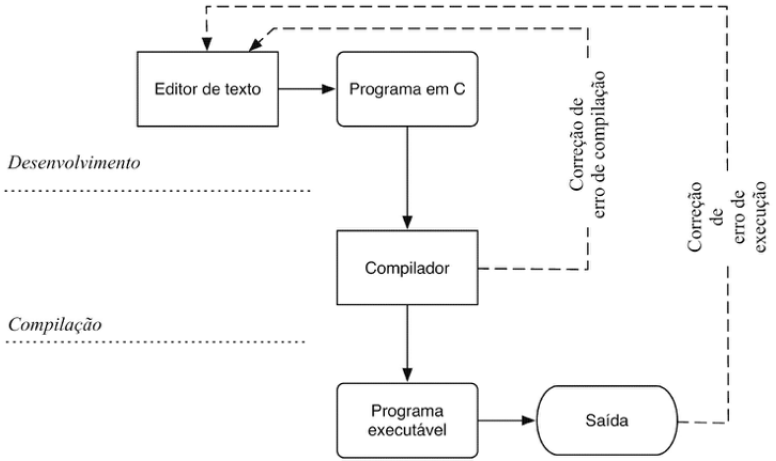
A linguagem C não provê operações para:

- manipular diretamente objetos compostos, tais como cadeias de caracteres;
- nem facilidades de entrada e saída: não há comandos *READ* e *WRITE*.

Execução de um Programa em C

- Compilação: Para executar um programa em C, devemos antes converter o programa em C para um programa em linguagem de máquina.

Principais Etapas



Conteúdo

Principais conceitos

- Variável representa um espaço na memória para armazenar determinado tipo de dados.
- Na linguagem C, todas as variáveis devem ser explicitamente declaradas.

  $< tipo\_variavel > nome\_variavel;$

Impressão no Terminal

A linguagem C, tem a função *printf*(...), que como o próprio nome diz, imprime.

Lendo do Teclado

A função *scanf* permite ler valores do teclado.

Tipo de Dado

- A função *scanf* recebe dois parâmetros: o tipo do dado que ela vai ler, e em qual variável ela deverá colocar o valor lido.
- A função *printf* usualmente também precisa saber o tipo de dado que será impresso.

Especificador de Formato

% c especifica um char
% d especifica um int
% u especifica um unsigned int
% f especifica um double (ou float)
% e especifica um double (ou float) no formato científico
% g especifica um double (ou float) no formato mais apropriado (% f ou % e)
% s especifica uma cadeia de caracteres
% c especifica um char
% d especifica um int
% u especifica um unsigned int
% f especifica um double (ou float)
% e especifica um double (ou float) no formato científico
% g especifica um double (ou float) no formato mais apropriado (% f ou % e)
% s especifica uma cadeia de caracteres

Exemplo

```
1  // stdio: uma biblioteca: standard I/O, ou seja, entrada e saída padrão.
2  #include <stdio.h>
3
4  int main()
5  {
6    ....
7    int idade;
8    printf("Qual sua idade?\n");
9    scanf("%d", &idade);
10   printf("Idade informada: %d", idade);
11   ...
12   return 0; /*A função main devolve um inteiro para informar o sistema
     operacional sobre o fim da execução do programa. */
13 }
14
```

Principais Conceitos

- A Linguagem C oferece uma gama variada de operadores, entre unários[1] e binários. [2]
- Em C, um operador binário é escrito entre dois operandos e operador unário precede seu único operando.

[1] Operam sobre um operando
[2] Operam sobre dois operandos

Aritméticos

Os operadores binários aritméticos são:

- adição ($+$)
- subtração ($-$)
- multiplicação ($*$)
- divisão ($/$)

Atribuição

Uma atribuição é uma expressão cujo valor resultante corresponde ao valor atribuído. Exemplo:

```
1    int x, y;
2    x = y = 5; /*a ordem de avaliação é da direita para esquerda*/
3
```

Operadores de Atribuição Compostos

A variável à esquerda do sinal da atribuição também aparece à direita. Exemplo:

```
1 int x = 10;
2 x = x + 1;
```

O que equivale a:

```
1 int x = 10;
2 x += 1;
```

Analogicamente existem: $-=$; $*=$; $/=$

Incremento e Decremento

São dois operadores não convencionais que possuem precedência ao operador unário - e são utilizados para incrementar ou decrementar uma unidade nos valores armazenados em variáveis. Exemplo:

```
1 /*incrementa de uma unidade o valor de n*/
2 x = n ++;
3 /*análogo para o decremento em n--*/
```

Prefixados e Pós-fixados

- Antes da variável $(++n)$
- Após a variável $(n++)$

```
1 int n = 5;
2 x = n++; // atribui 5 a x.
3 n = 5;
4 x = ++n; //atribui 6 a x.
```

Relacionais

São usados para comparar dois valores (*e.g.*, $==; != ; <, \leq$)

Lógicos

São usados para combinar expressões booleanas .

- && operador binário `AND`.
- || operador binário `OR`.
- ! operador unário de negação.

if

Avalia uma expressão *booleana* e redireciona o fluxo de execução baseado no resultado avaliado (verdadeiro ou falso).

```
1  if(expressao_booleana) {
2    bloco_comandos_1;
3  }
4  bloco_comandos_2;
```

if/else

Outra possível construção com o comando *if* acrescenta o bloco *else*, a ser executado apenas se a expressão *booleana* resultar em falso.

```
1  if(expressao_booleana) {
2    bloco_comandos_1;
3  } else {
4    bloco_comandos_2;
5  }
6  bloco_comandos_3;
```

Seleção

Comando para selecionar um dentre um conjunto de possíveis casos.

```
1  switch(expr) {
2      case op1:
3        .../*comandos executados para expr == op1*/
4      break;
5      case op2:
6        .../*comandos executados para expr == op2*/
7      break;
8      case op3:
9        .../*comandos executados para expr == op3*/
10     break;
11     default:
12       .../*comandos executados se expr for diferente dos valores listados*/
13     break;
14 }
15
```

Repetições iterativas são programadas mediante a construção de laços (ou ciclos).

While

Repete a execução de um bloco de comandos enquanto determinada condição for satisfeita. A sintaxe deste comando é:

```
1     while(expressao_booleana) {
2       bloco_comandos_1;   //
3     }
4     bloco_comandos_2;
5
```

Loop "infinito"pode ser útil?

```
1 while (true) {
2   ...
3   if (expressao_booleana)
4     break;
5 }
```

for

Equivalente ao comando *while*, porém com uma sintaxe mais concisa e compacta.

```
for(expressao_inicializacao; expressao_booleana; expressao_de_atualizacao) {
  bloco_comandos_1;
}
bloco_comandos_2;
```

```
for(int x = 0; x <100; x++) {
  printf("%d \n", x);
}
```

A "*expressao_inicializacao*"é executada uma única vez e usualmente **declara** e inicializa uma variável, a ser incrementada ou decrementada nas iterações do laço.

do-while

Ao contrário dos comandos *for* e *while*, a expressão *booleana* é avaliada no final do laço.

```
1    do {
2      bloco_comandos_1;
3    } while(expressao_booleana);
4    bloco_comandos_2;
5
```

A construção dos laços com testes no final é especialmente útil quando precisamos executar um passo da iteração antes de avaliar a condição de repetição.

break

- Quando utilizado dentro de *loop*, interrompe e termina a execução deste.
- A execução prossegue com os comandos subsequentes ao bloco.

continue

- Interrompe a execução dos comandos de um laço.
- O laço não é automaticamente finalizado.
- A execução de uma interação do *loop* é interrompida, passando para a próxima iteração.

Em programas estruturados, é recomendado:

- dividir as grandes tarefas de computação em tarefas menores;
- utilizar os resultados parciais das tarefas menores para compor o resultado final.

Bons programas em geral são compostos por diversas funções, sendo que cada função implementa uma funcionalidade.

Sintaxe

```
1    tipo_do_retorno nome_da_funcao(lista_de_parametros) {
2      corpo_da_funcao /*bloco de comandos, incluindo return, se cabível*/
3    }
4
```

Conceitos importantes: interface/especificação, implementação e escopo de variáveis.

Uso de Memória por um Programa

Informalmente, podemos dizer que existem três maneiras de reservar espaço de memória para o armazenamento de informações.

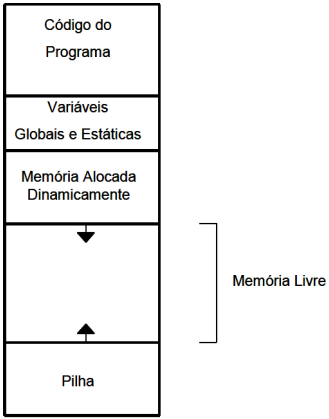
- Uso de variáveis globais e estáticas o espaço existe enquanto o programa estiver sendo executado.
- Variáveis locais (Pilha) - o espaço existe enquanto a função que declarou a variável estiver sendo executado, sendo liberado para outros usos quando a execução da função termina.
- Requisitando ao sistema em tempo de execução um espaço de memória de determinado tamanho - o espaço alocado dinamicamente permanece reservado até que seja explicitamente liberado,

Em geral, a memória utilizada por um programa de computador é dividida em:

- Segmento de Código
- Segmento de Dados
- *Heap*
- Pilha (*stack*)

Ilustração: Alocação de Memória

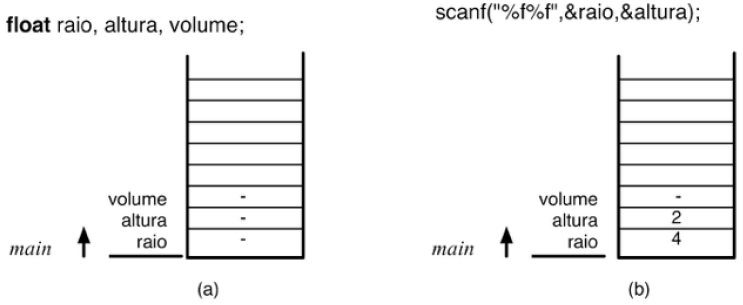
A seguir, um esquema didático que ilustra a distribuição de memória, feita pelo sistema operacional, para um programa e



Pilha de Variáveis Durante a Execução de um Programa

Exemplo de alocação de variáveis na pilha durante a execução de um programa.

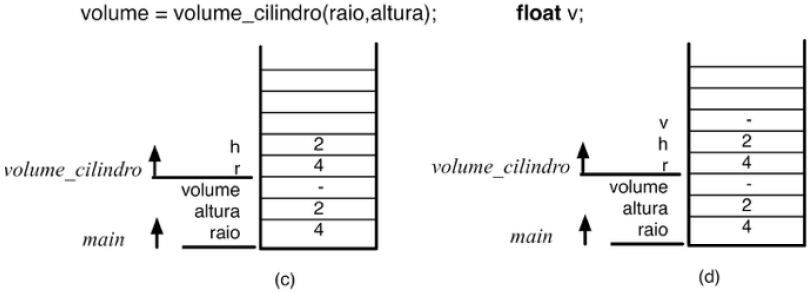
Etapas: *(a)* e *(b)*



Pilha de Variáveis Durante a Execução de um Programa

Exemplo de alocação de variáveis na pilha durante a execução de um programa.

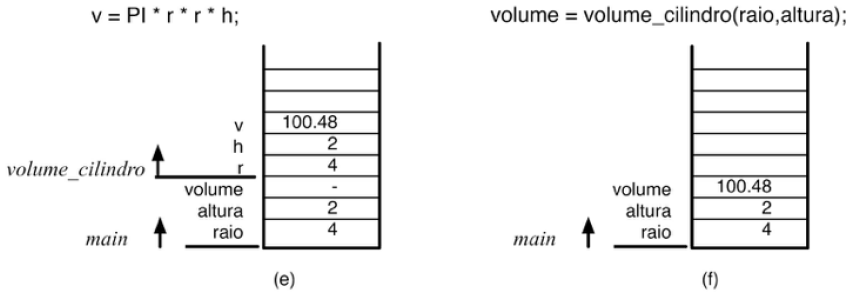
Etapas: *(c)* e *(d)*



Pilha de Variáveis Durante a Execução de um Programa

Exemplo de alocação de variáveis na pilha durante a execução de um programa.

Etapas: *(e)* e *(f)*



Variáveis Globais

Podem ser utilizadas para fazer a comunicação entre funções. Sobre variáveis globais:

- Declaradas fora do corpo das funções.
- Não são armazenadas na pilha de execução.
- Existem enquanto o programa estiver sendo executado.

Considerações...

Variáveis Estáticas

Não são armazenadas na pilha.

- Visível somente dentro do arquivo que a declara.
- Variáveis estáticas dentro de funções:
  - A variável também é armazenada numa área de memória estática que existe enquanto o programa está sendo executado.
  - Continuam existindo antes e depois de a função ser executada.
  - Existem enquanto o programa estiver sendo executado.
  - São visíveis dentro da função que as declarou.

Variável Estática dentro de Função

Exemplo:

```
1       void imprime ( float a ) {
2         static int n = 1;
3         printf(" %f ", a);
4         if ((n % 5) == 0) printf(" \n ");
5         n++;
6       }
7
```

- A linguagem C permite o armazenamento e a manipulação de valores de endereços de memória.
- Para cada tipo existente, há um tipo ponteiro que pode armazenar endereços de memória onde existem valores do tipo correspondente armazenado.
- Uma variável do tipo ponteiro tem a seguinte sintaxe geral:

```
1
2   ...
3   <tipo_variavel>* nome_variavel;
4   ...
```

Operadores Para Atribuir e Acessar Endereços de Memória

- O operador unário &, **endereço de**, aplicado a variáveis, resulta no endereço da posição da memória reservada para a variável.
- O operador unário $*$, **conteúdo de**, aplicado a variáveis do tipo ponteiro, acessa o conteúdo do endereço de memória armazenado pela variável ponteiro.

Exemplo

- Qual valor será impresso ao executar o código a seguir?

```
1  int main ( void ) {
2    int a, b, *p;
3    a = 2;
4    *p = 3;
5    b = a + (*p);
6    printf(" %d ", b);
7    return 0;
8  }
```

Exemplo de Código

Considere o trecho de código mostrado na figura abaixo.

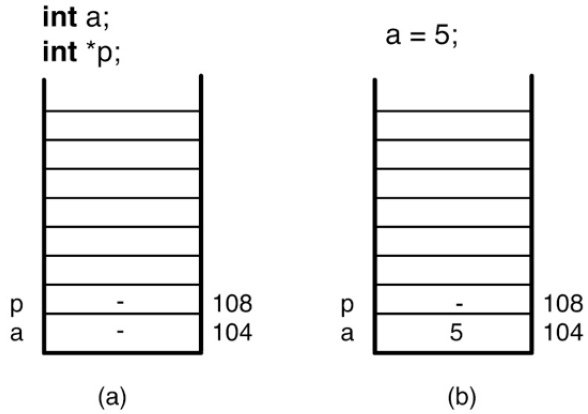
```
1  ...
2  int a; //variável do tipo inteiro
3  int* p; // variável do tipo ponteiro para inteiro
4  p = & a; //p recebe o endereço de a
5  *p = 8; //o conteúdo de p é alterado para 8
6  ...
```

O que ocorre na pilha de execução?

Pilha de Execução: código anterior

Nas figuras a seguir, os números à direita são valores fictícios dos espaços na memória.

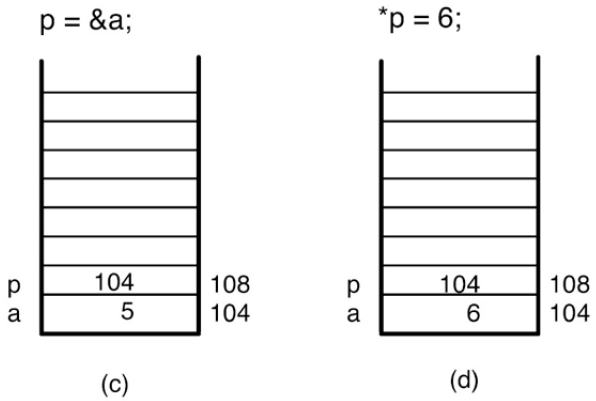
Etapas: *a* e *b*



Pilha de Execução: código anterior

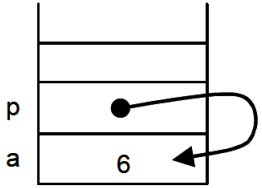
Nas figuras a seguir, os números à direita são valores fictícios dos espaços na memória.

Etapas: *c* e *d*



Representação Gráfica do Valor de um Ponteiro

Em vez de criarmos valores fictícios para os endereços de memória no nosso esquema ilustrativo da memória, podemos desenhar setas graficamente, sinalizando que um ponteiro aponta para uma determinada variável.



Ainda sobre ponteiros

Vantagem

A possibilidade de manipular ponteiros de variáveis é uma das maiores potencialidades de C.

Desvantagem

O uso **indevido** desta manipulação é o maior causador de programas que não só não funcionam como, pior ainda, podem gerar efeitos colaterais não previstos.

Os ponteiros oferecem meios de alterarmos valores de variáveis acessando-as indiretamente.

- As funções não podem alterar diretamente valores de variáveis da função que fez a chamada.
- E se passarmos para uma função os valores dos endereços de memória onde suas variáveis estão armazenadas...

Tentativa 1

```c
#include <stdio.h>

/* funcao troca */
void troca (int x, int y ) {
  int temp;
  temp = x;
  x = y;
  y = temp;
}

int main ( void ) {
  int a = 5, b = 7;
  troca(a, b);
  printf("%d %d \n", a, b);
  return 0;
}

```

Quais valores serão impressos?

Tentativa 2

```
1  #include <stdio.h>
2
3  /* funcao troca */
4  void troca (int *px, int *py ) {
5    int temp;
6    temp = *px;
7    x = *py;
8    *py = temp;
9  }
10
11 int main ( void ) {
12   int a = 5, b = 7;
13   troca(&a, &b);
14   printf("%d %d \n", a, b);
15   return 0;
16 }
17
```

Quais valores serão impressos?

Curiosidade

Agora fica explicado por que passamos o endereço das variáveis para a função *scanf*, pois, caso contrário, a função não conseguiria devolver os valores lidos.

Vetores

Usados para armazenar um conjunto de valores na memória do computador.

- Os valores são armazenados em sequência, um após o outro.
- É possível acessar qualquer valor do conjunto diretamente.
- Ao declarar um vetor, devemos informar o número máximo de elementos que poderá ser armazenado.
- Tipo **homogêneo** de dado: num vetor, só podemos armazenar valores de um mesmo tipo.

Sintaxe para declaração de um vetor

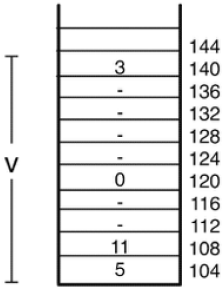
```
1  ...
2  <tipo_do_vetor> nome_do_vetor_[tamanho_do_vetor];
3  ...
4
```

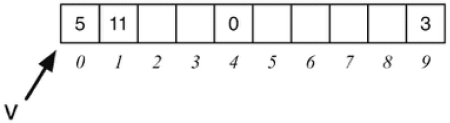
Exemplo

```
1  ...
2  int v[4];
3  v[0] = 1;
4  v[1] = 2;
5  v[2] = 4;
6  v[3] = 8;
7  /*ou ainda*/
8  int valores[] = {3,5,7};
9  ...
```

Armazenamento do vetor

A figura a seguir ilustra o armazenamento de vetores na memória do computador (pilha no lado direito, representação gráfica no lado esquerdo).





Considerando *int v*[10];: *v* é uma constante com o valor do **endereço inicial do vetor**.

Relação entre Vetor e Ponteiro

O nome de um vetor é um ponteiro para o tipo do elemento do vetor. No exemplo anterior:

- *v* é *int*∗.

Exemplo

```
1  ...
2  int v[10];
3  ... /*valores são inseridos no vetor*/
4  int* u = v;
5  u[0] = 4;
6  u[1] = v[0] + 2;
```

Quais os valores armazenados em *u*[1] e *v*[1]?

Aritmética de ponteiros para vetores

A linguagem C também suporta aritmética de ponteiros. Podemos somar e subtrair ponteiros, desde que o valor do ponteiro resultante aponte para dentro da área reservada para o vetor.

Com isso, um vetor tem as seguintes equivalências, dado $v[10]$:

`v+0`	$\rightarrow$	*aponta para o primeiro elemento do vetor*
`v+1`	$\rightarrow$	*aponta para o segundo elemento do vetor*
`v+2`	$\rightarrow$	*aponta para o terceiro elemento do vetor*
$\cdots$		
`v+9`	$\rightarrow$	*aponta para o último elemento do vetor*

Equivalências

- $\&v[i]$ é equivalente a escrever $(v + i)$.
- $v[i]$ é equivalente a escrever $*(v + i)$

A forma indexada é mais clara e adequada.

Passagem de Vetores para Funções

Passar um vetor para uma função consiste em passar o endereço da primeira posição do vetor.

- Se passarmos um valor de endereço, a função chamada deve ter um parâmetro do **tipo ponteiro** para armazenar este valor.
- Assim, se passarmos para uma função um vetor de *int*, devemos ter um parâmetro do tipo $int*$, capaz de armazenar endereços de inteiros.
- **Importante:** A expressão "*passar um vetor para uma função*" deve ser interpretada como "*passar o endereço inicial do vetor*".

  - **Os elementos do vetor não são copiados para a função, o argumento copiado é apenas o endereço do primeiro elemento.**

Passagem de Vetores para Funções

É passado para a função o endereço do primeiro elemento do vetor (e não os elementos propriamente ditos).

Exemplo

```
1  #include <stdio.h>
2
3  void incr_vetor ( int n, int *v ) {
4    int i;
5    for (i = 0; i < n; i++)
6    v[i]++;
7  }
8
9  int main ( void ) {
10   int a[ ] = {1, 3, 5};
11   incr_vetor(3, a);
12   printf("%d %d %d \n", a[0], a[1], a[2]);
13     return 0;
14 }
```

Matrizes

Usadas para conjuntos bidimensionais de dados.

- Matrizes devem ser declaradas para que o espaço apropriado de memória seja reservado.
- As duas dimensões da matriz devem ser especificadas: número de linhas e número de colunas.
- Tipo homogêneo de dados: numa matriz, só podemos alocar valores de um mesmo tipo.

Sintaxe para declaração de uma matriz

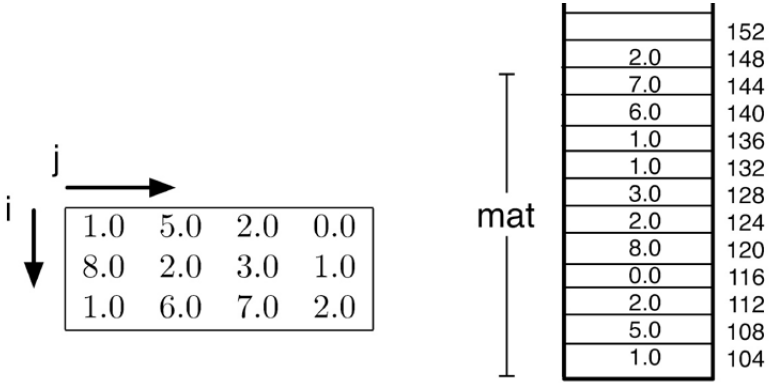
```
...
<tipo_da_matriz> nome_matriz [num_linhas][num_colunas];
...
```

Exemplo

```
...
int v[2][2] = {{0,1},
               {2,3}};
/*
 * os valores da matriz poderiam também ser
 * especificados usando um laço de repetições
 */
...
```

Armazenamento da matriz

A figura a seguir ilustra o armazenamento de matrizes na memória do computador (representação gráfica no lado direito, pilha no lado esquerdo).



Uma estrutura em C permite definir um tipo de dado cujos campos são compostos de vários valores de diversos tipos (tipo heterogêneo de dado).

Exemplo de `struct`

```
1  #include <stdio.h>
2  struct ponto {
3    float x;
4    float y;
5  };
6
7  int main (void) {
8    struct ponto p;
9    printf("Digite as coordenadas do ponto(x y): ");
10   scanf("%f %f", &p.x, &p.y);
11   printf("O ponto fornecido foi: (%.2f,%.2f)\n", p.x, p.y);
12   return 0;
13 }
```

A linguagem C permite criar nomes de tipos. Em geral, definimos nomes de tipos para as estruturas. Exemplo:

```
1  typedef struct ponto Ponto;
2  struct ponto {
3    float x;
4    float y;
5  };
6
7  /* forma alternativa
8   * typedef struct {
9   *     int x;
10  *     int y;
11  * } Ponto;
12  */
```

Assim *Ponto* passa a representar a *struct ponto*. Após essas definições, podemos declarar variáveis da seguinte maneira:

```
1  Ponto p; /*não precisa ser struct ponto p*/
```

Modularização

É um conceito bastante antigo que consiste em **dividir** um programa em **componentes individuais**, chamados módulos, que, uma vez integrados, atendem aos requisitos do problema.

- Reúso
- Legibilidade
- Manutenibilidade
- Dependabilidade

Formas de modularização em C e granularidade dos módulos.

Programa para Conversão de Temperatura

Seja um programa que tem a finalidade de converter valores de temperatura dados em Celsius para Fahrenheit.

converte.h

```
1  /* especificação do módulo de conversão */
2  float converte_celsius_fahrenheit (float c);
3
4  /*especificação de outras funções para conversão de temperatura*/
```

Programa para Conversão de Temperatura

Implementação da especificação.

converte.c

```
1  /* Implementação do módulo de conversão */
2  #include "converte.h"/*inclusão da especificação do módulo*/
3  float converte_celsius_fahrenheit (float c) {
4    float f;
5    f = 1.8*c + 32;
6    return f;
7  }
8
9  /*implementação de outras funções para conversão de temperatura*/
```

Módulo Principal

principal.c

```c
#include <stdio.h>
#include "converte.h" // inclusão do módulo de conversão de temperaturas

int main (void) {
  float t1; /*espaço para armazenar temperatura em Celsius*/
  float t2; /*espaço para armazenar temperatura em Fahrenheit*/
  printf("Entre com temperatura em Celsius: ");
  /* captura valor entrado via teclado */
  scanf("%f",&t1);
  /* faz a conversão */
  t2 = converte_celsius_fahrenheit(t1);
  /* exibe resultado */
  printf("Fahrenheit: %f\n", t2);
  return 0;
}
```

Compilando tudo junto

No Terminal

```
> gcc -o exe converte.c principal.c
```

Melhor Estratégia

- Se alterarmos a implementação de um determinado módulo precisaremos re-compilar os outros.
- ALTERNATIVA: Compilar os módulos separadamente e depois ligar os diversos módulos-objetos gerados para criar um executável.

No terminal

```
> gcc -c converte.c

> gcc -c principal.c

> gcc -o exe converte.o principal.o
```

Conteúdo

Conclusão

- Nesta aula, foi realizada uma revisão de programação em C.
- Foram apresentados recursos de programação e estruturação de dados fundamentais da linguagem.

Próxima Aula

- Esquema de memória.
- Alocação estática de memória vs Alocação dinâmica de memória.

Conteúdo

Exercício 1

Implemente o jogo conhecido *pedra, papel, tesoura*. Neste jogo, o usuário e o computador escolhem entre *pedra, papel* ou *tesoura*. Sabendo que *pedra* ganha de *tesoura*, *papel* ganha de *pedra* e *tesoura* ganha de *papel*, exiba na tela o ganhador de cinco rodadas (melhor de cinco): usuário ou computador. Para essa implementação, assuma que o número 0 representa a *pedra*, 1 representa *papel* e 2 representa *tesoura*. Observação: seu algoritmo deve ser honesto: a escolha feita pelo computador não pode ser realizada baseada na escolha do usuário e vice-versa. Dica: reutilizar funções para gerar números aleatórios num intervalo pré-definido.

Exercício 2

Implemente uma função iterativa para calcular o máximo divisor comum de dois números inteiros positivos, $MDC(x, y)$. Utilize o algoritmo de *Euclides*. Esse algoritmo é baseado no fato de que se o resto da divisão x por y, representado por r, for igual a zero, y é o MDC. Se o resto r for diferente de zero, o MDC de x e y é igual ao MDC de y e r. O processo se repete até que o valor da divisão seja igual a zero. Sua função para cálculo do MDC deverá estar num módulo separado, que posteriormente será refatorado para incluir outras funções matemáticas de interesse.

Exercício 3

Reutilizando o tipo *Ponto*, previamente descrito, implemente um tipo estruturado para representar um círculo: ponto central (*Ponto*) e raio (*float*). Implemente também uma função para verificar se, dado um ponto, ele está ou não dentro de um círculo, conforme protótipo a seguir. Implemente a função *main*() para testar sua solução.

```
1  /*a função retorna 1 se ponto pertencer ao círculo, zero caso contrário*/
2  int circ_interior (Circulo* c, Ponto* p);
```

Conteúdo

Bibliografia

Os conteúdos deste material, incluindo figuras, textos e códigos, foram extraídos ou adaptados do livro-texto indicado a seguir:


Celes, Waldemar and Cerqueira, Renato and Rangel, José
Introdução a Estruturas de Dados com Técnicas de Programação em C.
Elsevier Brasil, 2016.
ISBN 978-85-352-8345-7.