MC-102 — Aula 12
Funções e Procedimentos I

Instituto de Computação – Unicamp

Segundo Semestre de 2009

Roteiro

Registros = structs

Motivação:

- Unir variáveis inter-relacionadas
- Organizar código
  - Reduzir número de variáveis
  - Usar um único nome de variável

- Exemplo: Um aluno tem as seguintes informações: nome, RA, média de provas, média de labs...
  - Como ter um tipo que represente este aluno?
  - Com registros! Vou criar um tipo "meu" para representar este aluno!

Declaração do tipo registro

```
struct nome_do_tipo_do_registro {
  tipo_1 nome_1;
  tipo_2 nome_2;
  tipo_3 nome_3;
  ...
  tipo_n nome_n;
};
```

```
struct TipoRegistroAluno {
  char nome[50];
  int ra;
  float media;
};
```

Declarando as variáveis

```
struct TipoRegistroAluno {
    char nome[50];
    int ra;
    float media;     ...
};
```

struct TipoRegistroAluno aluno1;
struct TipoRegistroAluno aluno2;
struct TipoRegistroAluno aluno_selecionado;

Perceba que da mesma forma que na declaração **int soma**, **int** é o tipo da **variável soma**, struct TipoRegistroAluno é o tipo das variáveis aluno1, aluno2, aluno_selecionado.

Usando registros

```
printf("Digite informaçãoes do Aluno 1: ");
scanf("%s",aluno1.nome);
scanf("%d",&aluno1.ra);
scanf("%f",&aluno1.media);

printf("Digite informaçãoes do Aluno 2: ");
scanf("%s",aluno2.nome);
scanf("%d",&aluno2.ra);
scanf("%f",&aluno2.media);
```

Usando registros

Determinando o aluno de maior RA:

```
if (aluno1.ra > aluno2.ra){
    aluno_selecionado = aluno1;
 }else{
    aluno_selecionad = aluno2;
 }

printf("Nome: %s \n",aluno_selecionado.nome);
printf("RA: %d \n",aluno_selecionao.ra);
printf("Media: %f \n",aluno_selecionado.media);
```

O comando typedef

Definir novos nomes para tipos existentes

```
struct TipoRegistroAluno {
    char nome[50];
    int ra;
    float media;     ...
};
```

struct TipoRegistroAluno aluno1;
struct TipoRegistroAluno aluno2;
struct TipoRegistroAluno aluno_selecionado;

O comando typedef

```
struct TipoRegistroAluno aluno1;
struct TipoRegistroAluno aluno2;
struct TipoRegistroAluno aluno_selecionado;
```

Denifinir novo nome para um tipo existente

```
typedef struct TipoRegistroAluno Aluno;
```

Perceba que Aluno é um sinônimo para o tipo **struct TipoRegistroAluno**. E assim podemos declarar:

```
Aluno aluno1;
Aluno aluno2;
Aluno aluno_selecionado;
```

ou seja, aluno1, aluno2, aluno_selecionado são variáveis do tipo Aluno

O comando typedef

typedef *int* **Inteiro**;

```
Inteiro soma;
scanf("%d",&soma);
```

typedef *double* **Real**;

```
Real media;
scanf("%f",&media);
```

typedef *char* **String[30]**;

```
String nome;
scanf("%s",nome);
```

Registros aninhados

```
struct TipoRegistroAluno {
  char nome[50];
  int ra;
  float media;
  int dia;
  int mes;
  int ano;
};

struct TipoRegistroAluno {
  char nome[50];
  int ra;
  float media;
  data  data_ingresso;
};
```

Registros aninhados

```
struct TipoRegistroData {
  int dia;
  int mes;
  int ano;
};

typedef struct TipoRegistroData data;

struct TipoRegistroAluno {
  char nome[50];
  int ra;
  float media;
  data  data_ingresso;
};
```

Registros aninhados

```
struct TipoRegistroAluno {
  char nome[50];
  int ra;
  float media;
  data  data_ingresso;
};

typedef struct TipoRegistroAluno Aluno;

Aluno aluno1;

printf("%d",aluno1.data_ingresso.dia);
printf("%d",aluno1.data_ingresso.mes);
printf("%d",aluno1.data_ingresso.ano);
```

Registros aninhados

```
struct TipoRegistroFuncionario {
  char nome[50];
  int cargo;
  data  data_contratacao;
};

typedef struct TipoRegistroFuncionario Funcionario;

Funcionario funcionario1;
Aluno aluno1;
```

Perceba que reutilizamos o tipo data! Sempre que precisarmos de uma data no nosso programa podemos reutilizar o resgistro que criamos!

Vetor de registros

Pode ser declarado quando necessitarmos de diversas cópias de um registro (por exemplo, para cadastrar todos os alunos de uma mesma turma).

- Para declarar: `Aluno turmaMN[66];`
- Para usar: `turmaMN[indice].nome;`

O vetor turmaMN tem capacidade para armazenar os 66 alunos do tipo Aluno que definimos anteriormente! E se temos um vetor, então acessamos cada um dos alunos através das posições/índices deste vetor!

Veja o exemplo em `vetor.c` (aula 11)

Tipos enumerados

Para que serve?

Para representar um conjunto de opções através de identificadores.

```
switch (forma_pagamento){
    case 'd': //dinheiro
    case 'c': //cartão
    case 'v': //vale
}
```

```
switch (forma_pagamento){
    case dinheiro: ...
    case cheque: ...
    case vale: ...
    case cartão: ...
}
```

Tipos enumerados

Para que serve?

```
enum Tipo_pagamento(dinheiro, cheque, vale, cartão);

enum Tipo_pagamento forma_pagamento;

forma_pagamento = dinheiro;
switch (forma_pagamento){
   case dinheiro: ...
   case cheque: ...
   case vale: ...
   case cartão: ...
}
```

Tipos enumerados - declaração

```
enum  < nome > { < constante_1 > , < constante_2 > , ··· ,
                 < constante_N > , }
```

Associa a cada uma das constantes um valor inteiro!

Para que serve? Para facilitar sua vida!

Quando usar? Quando achar que isso pode ser útil na programação!

Tipos enumerados - outro exemplo

Queremos definir um **booleano** = variável que aceita o valor "true" ou "false"

```
enum   tipoBooleano  { false,true; }
```

```
typedef enum tipoBooleano booleano;
```

```
boolean escolha;

scanf("%d",&escolha);
if (escolha==true){
comandos;
}else{
comandos;
}
```

Procedimentos e funções

Procedimentos

São estruturas que agrupam um conjunto de comandos, que são executados quando o procedimento é chamado.

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

Funções

São procedimentos que retornam um único valor ao final de sua execução.

```
x = sqrt(4);
```

Porque utilizar funções?

- Evitar que os blocos do programa fiquem grandes demais e, por consequência, mais difíceis de ler e entender.
- Separar o programa em partes que possam ser logicamente compreendidos de forma isolada.
- Permitir o reaproveitamento de código já construído (por você ou por outros programadores).
- Evitar que um trecho de código seja repetido várias vezes dentro de um mesmo programa, minimizando erros e facilitando alterações.

Declarando uma função

```
tipo nome (tipo parâmetro1, tipo parâmetro2, ...,
           tipo parâmetroN) {
      comandos;
      return valor de retorno;
}
```

- nome: identifica a função; é usado na chamada/invocação da função; define da mesma forma que definimos variáveis.
- Toda função deve ter um tipo. Esse tipo determina qual será o tipo de seu valor de retorno
- Os parâmetros de uma função determinam os dados de entrada da fução; são variáveis locais declaradas e atribuídas na chamada da função; lista de tipos e nomes
- No corpo da função podemos escrever qualquer código que escreveriamos no "main", inclusive declarar variáveis

Declarando uma função

- Uma função pode não ter parâmetros, basta não informá-los.
- A expressão contida dentro do comando `return` é chamado de valor de retorno, e corresponde a resposta de uma determinada função. Esse comando é sempre o último a ser executado por uma função, e nada após ele será executado.
- As funções só podem ser declaradas fora de outras funções. Lembre-se que o corpo do programa principal (`main()`) é uma função.

Exemplo de função

A função abaixo soma dois valores, passados como parâmetros:

```
int soma (int a, int b) {
  return (a + b);
}
```

Invocando uma função

Uma forma clássica de realizarmos a invocação (ou chamada) de uma função é atribuindo o seu valor a uma variável:

```
x = soma(4, 2);
```

Na verdade, o resultado da chamada de uma função é uma expressão e pode ser usada em qualquer lugar que aceite uma expressão:

Exemplo

```
printf("Soma de a e b: %d\n", soma(a, b));
```

Veja um exemplo em `soma.c`.

Invocando uma função

Parâmetros na chamada da função:

- Ao chamar a função, o valor de cada parâmetro deve ser informado
- Um valor para cada parâmetro
- Valores do mesmo tipo do parâmetro
- Valores na mesma ordem que na decaração

O que acontece quando invocando uma função

1. Interrompe o programa principal (aquele que está no "main")
2. Passa dados para a função
3. Desvia a execução para a função
4. Executa o código da função
5. Retorna o resultado
6. Continua a execução do programa principal

Invocando uma função

- Para cada um dos parâmetros da função, devemos fornecer uma expressão de mesmo tipo, chamada de parâmetro real. O valor destas expressões são copiados para os parâmetros da função.
- Ao usar variáveis como parâmetros reais, estamos usando apenas os seus valores para avaliar a expressão.
- Se forem variáveis, os parâmetros reais passados pela função não necessariamente possuem os mesmos nomes que os parâmetros que a função espera.
- O valor das expressões que fornecem os parâmetros reais não é afetado por alterações nos parâmetros dentro da função.

Veja um exemplo em `parametros.c`.

O tipo `void`

- O tipo `void` é um tipo especial, utilizado principalmente em funções.
- Ele é um tipo que representa o "nada", ou seja, uma variável desse tipo armazena conteúdo indeterminado, e uma função desse tipo retorna um conteúdo indeterminado.
- Este tipo é utilizado para indicar que uma função não retorna nenhum valor.

Procedimentos em C

- Procedimentos em linguagem C nada mais são que funções do tipo `void`. Por exemplo, o procedimento abaixo imprime o número que for passado para ele como parâmetro:

```
void imprime (int numero) {
  printf ("Número %d\n", numero);
}
```

- Podemos ignorar o valor de retorno de uma função e, para esta chamada, ela será equivalente a um procedimento.

Invocando um procedimento

- Para invocarmos um procedimento, devemos utilizá-lo como utilizaríamos qualquer outro comando, ou seja:

```
procedimento (parametros);
```

- Esta é a forma como chamamos usualmente as funções `printf` e `scanf`.

Veja um exemplo em `imprime.c`.

A função `main`

- O programa principal é uma função especial, que possui um tipo fixo (`int`) e é invocada automaticamente pelo sistema operacional quando este inicia a execução do programa.
- Quando utilizado, o comando `return` informa ao sistema operacional se o programa funcionou corretamente ou não. O padrão é que um programa retorne zero caso tenha funcionado corretamente ou qualquer outro valor caso contrário.

Exemplo

```
int main() {
  printf("Hello, World!\n");
  return 0;
}
```

Variáveis locais e variáveis globais

- Uma variável é chamada local se ela foi declarada dentro de uma função. Nesse caso, ela existe somente dentro daquela função e após o término da execução da mesma, a variável deixa de existir.
- Uma variável é chamada global se ela for declarada fora de qualquer função (ou seja, no mesmo lugar onde registros, tipos enumerados e funções são declarados). Essa variável é visível em todas as funções, qualquer função pode alterá-la e ele existe durante toda a execução do programa.

Variáveis globais

```
#include <stdio.h>
int variavel_global;
int main () {
  variavel_global = 0;
  printf ("%d", variavel_global);
}
```

Veja outro exemplo em `global.c`

Escopo de variáveis

- O escopo de uma variável determina de que partes do código ela pode ser acessada.
- A regra de escopo em C é bem simples:
- As variáveis globais são visíveis por todas as funções.
- As variáveis locais são visíveis apenas na função onde foram declaradas.

Escopo de variáveis

```
int global;
void a() {
  int local_a;
  /* Neste ponto são visíveis global e local_a */
}
int main() {
  int local_main;
  a();
  /* Neste ponto são visíveis global e local_main */
}
```

Veja outro exemplo em `parametros.c`.

Escopo de variáveis

- É possível declarar variáveis locais com o mesmo nome de variáveis globais.
- Nesta situação, a variável local "esconde" a variável global.

```
int nota;
void a() {
  int nota;
  /* Neste ponto nota é a variável local. */
}
```

Veja mais detalhes em `escopo.c`

Exercício

Reestruture a atividade de laboratório em termos de funções e inclua a opção "todas as verificações". Este programa testa um número de acordo com as opções abaixo:

```
1 - Número primo
2 - Número par
....
5 - Todas as verificações
6 - Sair
```