Universidade de Brasília

Funções

Computação para Engenharia — Tópico 7

Daniel Guerreiro e Silva

Departamento de Engenharia Elétrica (ENE), Faculdade de Tecnologia (FT)

Roteiro

Funções

A função `main`

Variáveis locais, globais e escopo

Argumentos por valor e por referência

Vetores em funções

Funções

Procedimentos e funções

Funções permitem organizar o programa em “blocos” de comandos que cumprem tarefas específicas.

Uma função é um grupo de comandos que recebe um determinado nome e que pode ser invocada a partir de algum ponto do programa.

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

Exemplo: função `sqrt` da biblioteca cmath

```
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.

Exemplo de função

A função abaixo recebe dois valores passados como parâmetro, realiza a soma destes e responde (retorna) o resultado a quem a invocou:

```
int soma (int a, int b) {
  int r;
  r = a + b; //cálculo da soma
  return r; //retorno do resultado e fim da função
}
```

Declarando uma função

Uma função é declarada da seguinte forma:

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

- Toda função deve ter um tipo. Esse tipo determina qual será o tipo de seu valor de retorno, isto é, o tipo de dado da resposta entregue a quem invoca a função, quando ela termina.
- Os parâmetros de uma função determinam qual será o seu comportamento e atuam como variáveis locais, que são iniciadas somente quando a função é chamada.

Declarando uma função

- Uma função pode não ter parâmetros, basta não declará-los (parênteses vazios).
- A expressão contida dentro do comando `return` é chamada de valor de retorno e corresponde a resposta de uma determinada função.
  - Esse comando encerra a execução de uma função: um comando após ele nunca é executado.
- As funções só podem ser declaradas fora de outras funções.
- O corpo do programa principal — `int main()` — é uma função.

Invocando uma função

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

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

No entanto, o resultado da chamada de uma função é uma expressão e pode ser usada em qualquer lugar que aceite uma expressão:

Exemplo

```
cin >> a >> b;
cout << "Soma de a e b: " << soma(a, b) << endl;
c = soma(a,b) / 2;
```

Exemplo: soma.cpp

```cpp
#include <iostream>

using namespace std;

//funcao soma: recebe dois valores inteiros e retorna a soma deles
int soma (int a, int b) {
  int r; //variavel local
  r = a + b;
  return r;
}

int main () {
  int c, d;

  cout << "Digite o valor 1: ";
  cin >> c;

  cout << "Digite o valor 2: ";
  cin >> d;

  cout << "Soma: " << soma(c, d) << endl;
  cout << "Soma de 5 e 10: " << soma(5, 10) << endl;

  return 0;
}
```

Invocando uma função

Para cada um dos parâmetros da função, devemos fornecer uma expressão de mesmo tipo, chamada de argumento ou parâmetro real. O valor destas expressões são copiados para os parâmetros da função.

- Ao usar variáveis como argumentos, estamos usando apenas os seus valores para avaliar a expressão.
  - Os argumentos passados pela função não são obrigados a ter os mesmos nomes que os parâmetros que a função espera.
- O valor das expressões que fornecem os argumentos não é afetado, a princípio, por alterações nos parâmetros dentro da função.

Veja um exemplo em `parametros.cpp`.

Procedimentos

O tipo `void` é um tipo especial que representa um conteúdo indefinido e uma função desse tipo retorna um conteúdo indeterminado.

- Este tipo é utilizado quando queremos que uma função não retorne nenhum valor, isto é, que seja só um procedimento.
- Procedimentos em C++ são simplesmente funções do tipo `void`.

Procedimentos

Exemplo

Procedimento que imprime o número que for passado como parâmetro:

```
void imprime (int numero) {
  std::cout << "Número " << numero << "\n";
}
```

- Note que, em um procedimento, podemos ignorar o comando `return`.

Invocando um procedimento

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

```
procedimento(parametros);
```

Invocando um procedimento

Exemplo: imprime.cpp

```cpp
#include <iostream>
using namespace std;

//exemplo de procedimento
void imprime (int numero) {
  cout << "Numero " << numero << "\n";
}

int main () {
  imprime(10);
  imprime(20);

  return 0;
}
```

A função `main`

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() {
  std::cout << "Hello, World!\n" << std::endl;
  return 0;
}
```

Invocando funções antes de defini-las

Uma função só pode ser chamada / invocada no código se ela tiver sido declarada anteriormente.

Exemplo: `depois.cpp`

```cpp
#include <iostream>

int main () {
  int a = 0, b = 5;
  std::cout << soma (a, b) << "\n";
  return 0;
}

int soma (int op1, int op2) {
  return (op1 + op2);
}
```

```
In function `int main()':
depois.cpp:5:26: [Error] 'soma'
was not declared in this scope
```

Declarando uma função sem defini-la

Para organizar melhor um programa ou para escrever um programa em vários arquivos podemos declarar uma função sem implementá-la (defini-la).

- Para declarar uma função sem a sua implementação nós substituímos as chaves e seu conteúdo por ponto-e-virgula.

```
tipo nome (tipo parâmetro1, tipo parâmetro2, …, tipo parâmetroN);
```

- A declaração de uma função deve vir sempre antes do seu uso. A sua definição pode aparecer em qualquer lugar do programa.

Exemplo: depoiscorrigido.cpp

```cpp
#include <iostream>

//declaracao da funcao
int soma (int op1, int op2);

int main () {
  int a = 0, b = 5;
  std::cout << soma (a, b) << "\n";
  return 0;
}

//definicao / implementacao da funcao
int soma (int op1, int op2) {
  return (op1 + op2);
}
```

Exemplo: depoiscorrigido2.cpp

A declaração da função pode ser feita sem os nomes dos parâmetros. Na definição, naturalmente, os nomes dos parâmetros devem estar presentes.

```cpp
#include <iostream>

//declaracao da funcao
int soma (int, int);

int main () {
  int a = 0, b = 5;
  std::cout << soma (a, b) << "\n";
  return 0;
}

//definicao / implementacao da funcao
int soma (int op1, int op2) {
  return (op1 + op2);
}
```

Variáveis locais, globais e escopo

Variáveis locais e variáveis globais

Variável Local

Uma variável é denominada 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.

Variáveis locais e variáveis globais

Variável Local
Uma variável é denominada 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.

Variável Global
Uma variável é denominada global se ela for declarada fora de qualquer função. Essa variável é visível em todas as funções, qualquer função pode alterá-la e ela existe durante toda a execução do programa.

Variáveis globais: `global.cpp`

```cpp
#include <iostream>

int global;

//procedimento imprime conteudo de global
void imprime_global () {
  std::cout << global << "\n";
}
//procedimento que escreve na var. global
void le_global () {
  std::cout << "Digite o valor da variavel global: ";
  std::cin >> global;
}
//funcao principal
int main () {
  global = 0;
  le_global();
  imprime_global();
  std::cout << global << std::endl;
  return 0;
}
```

Escopo de variáveis

O escopo de uma variável determina em quais partes do código ela pode ser acessada.

A regra de escopo em C++ é:

- As variáveis globais são visíveis por todas as funções.
- As variáveis locais são visíveis apenas na função onde foram declaradas.

Escopo de variáveis

```c
int global;
void rotina1() {
  int local_a;
  /* Neste ponto sao visiveis global e local_a */
}
int main() {
  int local_main;
  rotina1();
  /* Neste ponto sao visiveis global e local_main */
}
```

Escopo de variáveis

É possível declarar variáveis locais com o mesmo nome de variáveis globais.

- Nesta situação, a variável local “suprime” a variável global.

```
1 int nota;
2 void a() {
3   int nota;
4   /* Neste ponto nota eh a variavel local. */
5 }
```

Escopo de variáveis

Um determinado nome só pode representar uma entidade, dentro de um determinado escopo.

Exemplo

```
1 int uma_funcao() {
2   int x;
3   x = 0;
4   double x;   // errado: nome ja usado nesse escopo
5   x = 0.0;
6 }
```

Namespaces

A linguagem C++ oferece um nível intermediário de escopo, que são os *namespaces*.

- Nomes de variáveis, tipos e funções podem daí ser agrupados em diferentes escopos lógicos referenciados por um mesmo nome.

Sintaxe:

```
namespace identificador
{
  entidades_nomeadas
}
```

Exemplo

```
namespace meuNamespace{
  int a, b;
}
```

`a` e `b` são acessadas normalmente dentro de `meuNamespace`, mas se desejarmos acessar fora é necessário que elas sejam qualificadas apropriadamente, por meio do operador de escopo `::`

```
meuNamespace::a
meuNamespace::b
```

Namespaces: `namespaces.cpp`

```cpp
// namespaces
#include <iostream>
using namespace std;

namespace foo{
        int value() {
                return 5;
        }
}

namespace bar{
        //modificador const impede mudanca na var. pi
        const double pi = 3.1416;
        double value() {
                return 2*pi;
        }
}

int main () {
        //acessamos as funcoes pelos nomes qualificados
        cout << foo::value() << '\n';
        cout << bar::value() << '\n';
        cout << bar::pi << '\n';
        return 0;
}
```

Comando *using*

O comando *using* introduz um nome na região/bloco em que for declarado, daí desobrigando que se use nomes qualificados.

```
1  #include <iostream>
2  using namespace std;
3
4  namespace first{
5    int x = 5;
6    int y = 10;
7  }
8  namespace second{
9    double x = 3.1416;
10   double y = 2.7183;
11 }
12
13 int main(){
14   using first::x;
15   using second::y;
16   cout << x << '\n';
17   cout << y << '\n';
18   cout << first::y << '\n';
19   cout << second::x << '\n';
20   return 0;
21 }
```

Ele pode ser usado para um namespace completo, como `using namespace std;`

Argumentos por valor e por referência

Passagem de argumentos por valor

Quando passamos argumentos a uma função, os valores fornecidos são copiados para os parâmetros formais da função. Este processo é idêntico a uma atribuição.

- Desta forma, alterações nos parâmetros dentro da função não alteram, a princípio, as variáveis cujos valores foram passados:

```
void nao_troca(int x, int y) {
  int aux;
  aux = x;
  x = y;
  y = aux;
}
```

Veja o exemplo completo em `nao_troca.cpp`.

Passagem de argumentos por referência

Existe em C++ uma forma de alterarmos a variável passada como argumento, ao invés de usarmos apenas o seu valor.

- Para isso devemos declarar os parâmetros de uma função como referências, usando o operador `&` ao lado do tipo do parâmetro.

Passagem de argumentos por referência

Para indicarmos que será passado um argumento por referência, usamos o tipo seguido do operador `&`:

```
tipo nome (tipo& parâmetro1, tipo& parâmetro2, …, tipo& parâmetroN) {
      comandos;
}
```

Na passagem por referência, o parâmetro da função torna-se “vinculado” com o argumento passado à função no momento da invocação.

- Qualquer modificação nos parâmetros correspondentes dentro da função reflete-se nas variáveis passadas como argumento na chamada.

Exemplo: troca.cpp

```cpp
#include <iostream>
using namespace std;

void troca(int& x, int& y) {
  int aux;
  //troca de valores entre var.
  aux = x;
  x = y;
  y = aux;
}

int main() {
  int a = 100, b = 200;

  troca(a, b);
  cout << "a = " << a << ", b = " << b << endl;
  return 0;
}
```

Vetores em funções

Vetores em funções

Ao contrário dos outros tipos, vetores têm um comportamento diferente quando usados como parâmetros ou valores de retorno de funções.

- Por padrão, ao se declarar um vetor como parâmetro, este sempre é interpretado pelo compilador como o endereço do seu primeiro elemento.
- Por isso, sem precisarmos usar uma notação especial, os vetores são sempre passados por referência.
  - Exceção: o tipo `std::string` continua sendo passado por valor.

Veja exemplos em `vetor_parametro.cpp` e `vetor_vs_variavel.cpp`.

Vetores em funções

Devemos ficar atentos às implicações do fato dos vetores serem sempre passados por referência.

- Ao passar um vetor como argumento, se este for alterado dentro da função, as alterações ocorrerão no próprio vetor e não em uma cópia.
- Ao retornar um vetor como valor de retorno, não é feita uma cópia deste vetor, como no caso de tipos “elementares” (int, double, char, float...). Assim, o vetor “retornado” pode desaparecer se ele foi declarado no corpo da função.

Vetores multi-dimensionais e funções

Ao declarar um vetor como parâmetro não é necessário fornecer o seu tamanho no cabeçalho da função. Porém, é importante lembrar que o vetor tem um tamanho que deve ser considerado, como no exemplo em `vetor_parametro.cpp`.

- Quando o vetor é multi-dimensional a possibilidade de não informar o tamanho na declaração se restringe à primeira dimensão, apenas.

Exemplo

```
void mostra_matriz(int mat[][10], int n_linhas) {
  ...
}
```

Veja o exemplo completo em `matriz.cpp`.

Até o próximo tópico...