Funções[1][2]

C++ é uma linguagem híbrida, podendo ser classificada como pertencente a diferentes paradigmas de programação. Por ser um super conjunto de C, C++ oferece suporte ao paradigma procedural, isto é, linguagens onde o modelo conceitual de programação se baseia em procedimentos (conhecidas em C++ como funções). Assim, é fundamental compreender os conceitos e inovações relacionados a funções que C++ apresenta.

1 Protótipos

Protótipos (ou assinaturas) são uma exigência de C++ para a declaração de funções em projetos modulares (com múltiplos arquivos). Em geral, a regra básica de declaração de um protótipo em C++ segue o modelo:

```
1 <tipo_retorno> <nome_funcao> ([<tipo_parametro1>, <tipo_parametro2>, ...]);
```

Em particular, C++ trata protótipos de forma diferente de C. O código abaixo, por exemplo, é tratado pelo C como uma função que pode ou não ter parâmetros. Por sua vez, o mesmo código em C++ pode ser usado apenas para o caso de uma função sem parâmetros.

```
1 <tipo_retorno> <nome_funcao> ();
```

A declaração de protótipos é fundamental em C++, uma vez que o `g++` é mais criterioso que o `gcc`. Especificamente, um código em C organizado em múltiplos arquivos que não apresente declarações de protótipos será compilado pelo `gcc` sem warnings, a menos que flags específicas sejam ativadas. Por sua vez, um código em C++ organizado em múltiplos arquivos sem declarações não será compilado pelo `g++`.

Exemplo

- Protótipo correto: `http://cpp.sh/4sw3`
- Protótipo incorreto: `http://cpp.sh/5z2g`

2 Parâmetros *default*

Uma conveniência oferecida por C++ é o uso de valores padrão (*default*) para parâmetros de funções, usados quando o usuário invoca a função sem especificar aqueles argumentos. No entanto, o uso de parâmetros *default* apresenta algumas restrições para permitir que o compilador identifique corretamente a função invocada e a qual parâmetro atribuir cada valor.

1. **Protótipos:** valores *default* não fazem parte do protótipo de funções, devendo ser especificados apenas em sua definição.
2. **Ordem de definição:** parâmetros *default* devem vir, obrigatoriamente, após parâmetros tradicionais.

[1] Roteiro de estudo fornecido na disciplina de Linguagem de Programação 1 (LP1) da Universidade Federal do Rio Grande do Norte (UFRN). Disponível em `https://leobezerra.github.io/LP1-2017-1-T05`.

[2] Autor: Leonardo Bezerra (`leobezerra@imd.ufrn.br`).

3. **Invocação:** a omissão de um parâmetro *default* na invocação assegura que o valor padrão para aquele parâmetro seja usado. No entanto, caso uma função apresente múltiplos parâmetros *default*, o primeiro parâmetro *default* omitido implicará na omissão obrigatória dos parâmetros *default* restantes. Por exemplo, se o penúltimo parâmetro *default* for omitido na invocação, o último parâmetro (que, pela definição 2, também é obrigatoriamente *default*) terá de ser omitido.

Exemplos

- Protótipos (correto): `http://cpp.sh/7pqz6`
- Protótipos (incorreto): `http://cpp.sh/67btr`
- Ordem de definição (correto): `http://cpp.sh/77ixy`
- Ordem de definição (incorreto): `http://cpp.sh/6fx5`
- Múltiplos parâmetros *default* (correto): `http://cpp.sh/6dbl`
- Múltiplos parâmetros *default* (incorreto): `http://cpp.sh/64ncv`

3 Ponteiros para funções

Assim como no caso de variáveis, tanto C como C++ permitem que funções sejam acessadas de maneira indireta. Especificamente, funções são tratadas pelo compilador como trechos de código armazenados na memória (segmento de código). Assim, funções podem ser referenciadas pelo seu endereço. Analogamente ao caso das variáveis, o endereço de uma função é obtido através do operador &, podendo ser armazenado em um ponteiro. Para declarar um ponteiro para função em C++, a notação é bastante similar à notação da declaração de protótipos:

```
<tipo_retorno> (*<nome_funcao>) ([<tipo_parametro1>, <tipo_parametro2>, ...]);
```

A principal aplicação de ponteiros de função é o polimorfismo. Mais precisamente, ponteiros para função são utilizados como argumentos de funções mais amplas, permitindo que estas apresentem diferentes comportamentos dependendo da função passada como argumento. Uma restrição do acesso indireto a funções, no entanto, é que este método não ativa os parâmetros default de funções. Assim, quando se quer invocar uma função através de ponteiros, é preciso especificar todos os parâmetros apresentados em seu protótipo.

3.1 Exemplos

- Como funções independentes: `http://cpp.sh/35dj`
- Como parâmetro: `http://cpp.sh/4qays`
- Com parâmetros default (incorreto): `http://cpp.sh/2mwhj`
- Com parâmetros default (correto): `http://cpp.sh/46vfk`

3.2 Laboratório

O arquivo `modelo-parser.zip` contém a implementação parcial deste laboratório. Especificamente, vocês deverão implementar um interpretador para um subconjunto da linguagem de máquina MIPS. Este subconjunto da linguagem apresenta apenas instruções ternárias, isto é, instruções que recebem três operandos (índices de registradores do processador). Em nossa aplicação, utilizaremos um vetor de tamanho 10 para simular os registradores do processador. A lista de instruções e suas definições podem ser vistas abaixo:

add a b c Soma os valores contidos nos registradores **b** e **c** e armazena o resultado no registrador **a**.

sub a b c Subtrai o valor contido no registrador **c** do valor contido no registrador **b** e armazena o resultado no registrador **a** (isto é, $a = b - c$).

and a b c Realiza um E binário entre os valores contidos nos registradores **b** e **c** e armazena o resultado em **a**.

or a b c Realiza um OU binário entre os valores contidos nos registradores **b** e **c** e armazena o resultado em **a**.

slt a b c Armazena 1 no registrador **a** caso o valor contido no registrador **b** seja menor que o valor contido no registrador **c**. Caso contrário, armazena 1 no registrador **a**.

Uma pasta contendo os testes de aceitação para este exercício está disponível no SIGAA (`aceitacao.zip`). Mais precisamente, cada caso de teste seguirá o modelo abaixo (desconsidere as linhas iniciadas com #):

```
# o estado inicial dos registradores
R0 R1 ... R9
# instruções
I1 a1 b1 c1
I2 a2 b2 c2
...
In an bn cn
```

A saída da sua aplicação será composta de três linhas. As duas primeiras linhas de saída serão impressas logo após a leitura do valor inicial dos registradores, seguindo o modelo abaixo. Em ambas as linhas, deve-se utilizar separação por tabulação.

```
[0]  [1]  [2]  [3]  [4]  [5]  [6]  [7]  [8]  [9]
R0   R1   R2   R3   R4   R5   R6   R7   R8   R9
```

A terceira linha de saída desta aplicação será impressa ao final do parsing, contendo os valores de cada registrador, em ordem, separados por um espaço simples. Note que os dois tipos de impressão deverão ser efetuados com o uso da função `estado_registradores`. Para executar um caso de teste, use as ferramentas e características dos ambientes Unix:

1. Dado um caso de teste `input/entrada.in`, seu conteúdo pode ser lido por seu programa como se um usuário estivesse digitando. Isto se chama redirecionamento de fluxo, e para redirecionamento de entrada o operador é o <

   ```
   bin/lab03 < input/entrada.in
   ```

2. Seguindo o mesmo raciocínio, é possível redirecionar o fluxo de saída do programa, escrevendo a saída do terminal em um arquivo de texto. Para isto, use o operador >

   ```
   bin/lab03 < input/entrada.in > saida.out
   ```

3. Para comparar a saída do seu programa com a saída esperada, use a ferramenta `diff`:

   ```
   bin/lab03 < input/entrada.in > saida.out
   diff saida.out output/saida.out
   ```

Notem que existem vários casos de entrada e saída nas pastas `input` e `output`. Assim, tenha o cuidado de usar o par de arquivos corretos (ex.: `entrada1.in` e `saida1.out`).

4 Sobrecarga de funções

Uma inovação introduzida pelo C++ é a possibilidade de que um mesmo nome possa ser utilizado para se referir a diferentes funções, diferindo apenas em seu protótipo.

1. **Argumentos:** é necessário que as funções difiram em seus protótipos, isto é, na quantidade ordem dos tipos dos argumentos. Note que, como o nome dos argumentos não faz parte do protótipo, não é permitido duas declarações que difiram somente pelo nome dos argumentos.

2. **Argumentos *default*:** duas funções podem apresentar diferentes valores padrão para argumentos *default*. No entanto, para que o compilador não se confunda em ambiguidades, isto só é possível quando as listas de argumentos destas funções diferem.

3. **Tipo de retorno:** duas funções podem apresentar diferentes tipos de retorno. No entanto, para que o compilador não se confunda em ambiguidades, isto só é possível quando as listas de argumentos destas funções diferem.

A principal aplicação da sobrecarga de funções é permitir que uma mesma função apresente diferentes comportamentos em função do tipo de entrada. No entanto, deve-se respeitar a noção de semântica da aplicação, isto é, utilizar o mesmo nome apenas para funções que sirvam para o mesmo propósito.

4.1 Exemplos

- Argumentos (correto): `http://cpp.sh/6xci`
- Argumentos (incorreto): `http://cpp.sh/32ptb`
- Argumentos *default* (correto): `http://cpp.sh/6jr7y`
- Argumentos *default* (incorreto): `http://cpp.sh/6penp`
- Tipo de retorno (correto): `http://cpp.sh/6c77`
- Tipo de retorno (incorreto): `http://cpp.sh/8ix2s`

4.2 Laboratório

Neste exercício, você deverá ampliar as funcionalidades implementadas no arquivo `src/registradores.cpp`. Especificamente, você deve escrever uma nova versão da função `estado_registradores` que aceite como parâmetros o vetor de registradores e uma função para computar estatísticas. As subrotinas de estatísticas estão descritas abaixo.

min – índice do registrador com valor mínimo não-nulo.

max – índice do registrador com valor máximo.

zeros – quantidade de registradores com valores nulos.

ones – quantidade de registradores com valor igual a 1.