Nada se obtém sem esforço; e tudo se pode conseguir com ele.

Ralph Waldo Emerson

1 Agora, um pouco mais... do mesmo - trabalhando com referências

Uma referência fornece um outro nome para uma variável. Qualquer operação de alteração numa referência também ocorrerá na variável referida.Você pode pensar em uma referência como um apelido para um nome de outra variável.

O programa 1 declara variável de referência *meuPlacarTambem* para a variável *meuPlacar*.

Programa 1: *Apelidos* para variáveis.

```
1  // referências – começando o abacaxi
2  // programa_001.cpp
3  #include "biblaureano.h"
4
5  int main()
6  {
7     int meuPlacar = 100;
8     int& meuPlacarTambem = meuPlacar;
9
10    cout << "meuPlacar = " << meuPlacar << endl;
11    cout << "meuPlacarTambem = " << meuPlacarTambem << endl;
12
13    cout << "Hackeando o jogo... subindo o meu placar..." << endl;
14
15    meuPlacar += 500;
16
17    cout << "meuPlacarTambem = " << meuPlacarTambem << endl;
18
19    cout << "Sacaneando vocês... estou roubando no placar... " << endl;
20    meuPlacarTambem += 1000;
21
22    cout << "meuPlacar = " << meuPlacar << endl;
23
24    cout << "Game over!!!" << endl;
25    return 0;
26 }
```

1.1 Entendendo o programa

Inicialmente, a variável *meuPlacar* é declarada normalmente (como qualquer outra variável que já utilizamos em vários outros programas):

```
1     int meuPlacar = 100;
```

Na sequência, é criada a variável *meuPlacarTambem* que é uma referência para a variável *meuPlacar*. Repare que existe um & (letra E comercial) antes do nome da variável:

```
1     int& meuPlacarTambem = meuPlacar;
```

Cuidado: sempre que você declarar uma variável referência, você deve atribuir algum valor a ela (nome da variável que será referenciada. Uma chamada a:

```
1   int& meuPlacarTambem;
```

Causará um erro de compilação, pois a variável não foi inicializada.

Com a referência criada, as duas variáveis terão o mesmo valor na memória:

```
1   cout << "meuPlacar = " << meuPlacar << endl;
2   cout << "meuPlacarTambem = " << meuPlacarTambem << endl;
```

A partir deste momento, qualquer alteração feita na variável original (*meuPlacar*) também refletirá na variável apelido (*meuPlacarTambem*:

```
1   cout << "Hackeando o jogo... subindo o meu placar..." << endl;
2
3   meuPlacar += 500;
4
5   cout << "meuPlacarTambem = " << meuPlacarTambem << endl;
```

E o inverso também ocorre, sempre que ocorrer qualquer alteração realizada na variável referência (*meuPlacarTambem*) a variável original (*meuPlacar*) também será alterada:

```
1   cout << "Sacaneando vocês... estou roubando no placar... " << endl;
2
3   meuPlacarTambem += 1000;
4
5   cout << "meuPlacar = " << meuPlacar << endl;
```

Lembre-se: pense sempre na variável referência como um *apelido* para uma variável normal.

2 Referências para funções - alterando os parâmetros de entrada

Agora que você já entendeu como funcionam as referências, você deve estar se perguntando: *Quando irei utilizá-los?* ou *Qual a utilidade?*.

Bem, as referências são úteis quando você está passando as variáveis para as funções, pois quando você passa uma variável (parâmetro de entrada), a função recebe uma *cópia* da variável. Isto significa que a variável original que você passou não pode ser modificada.

Claro, isso pode ser exatamente o que você deseja, pois a função mantém a variável de argumento seguro e imutável. Mas outras vezes você pode querer alterar uma variável argumento dentro da função e você pode fazer isso usando referências. O programa 2 demonstra uma possibilidade de uso para referências de variáveis nos argumentos/parâmetros de entrada de uma função:

Programa 2: Parâmetros como referências.

```
// trocando variáveis
// programa_002.cpp
#include "biblaureano.h"

void naoFuncionaTroca( int a, int b);
void aquiFuncionaTroca( int& a, int& b);

int main()
{
   int umValor, outroValor;

   umValor = 30072010;
   outroValor = 27121975;

   cout << "Valores antes da troca.." << endl;
   cout << "umValor = " << umValor << endl;
   cout << "outroValor = " << outroValor << endl;

   cout << "Trocando as variáveis..." << endl;
   naoFuncionaTroca( umValor, outroValor );

   cout << "umValor = " << umValor << endl;
   cout << "outroValor = " << outroValor << endl;

   cout << "Trocando as variáveis... agora vai!!!" << endl;
   aquiFuncionaTroca( umValor, outroValor );

   cout << "umValor = " << umValor << endl;
   cout << "outroValor = " << outroValor << endl;

   cout << "Game over!!!" << endl;
   return 0;
}

void aquiFuncionaTroca( int& a, int& b)
{
   int temporario = a;
   a = b;
   b = temporario;
   return;
}

void naoFuncionaTroca( int a, int b)
{
   int temporario = a;
   a = b;
   b = temporario;
   return;
}
```

2.1 Entendendo o programa

O objetivo do programa é simples, realizar a troca (*swap*,permuta) dos valores entre duas variáveis. Inicialmente são declarado os protótipos das funções que realizam a troca, repare que os argumentos de entrada da função *aquiFuncionaTroca* são declarados como referência.

```
void naoFuncionaTroca( int a, int b);
void aquiFuncionaTroca( int& a, int& b);
```

Após a declaração das variáveis, é chamada a função *naoFuncionaTroca* e mostrado o resultado da troca:

```
   cout << "Trocando as variáveis..." << endl;
   naoFuncionaTroca( umValor, outroValor );

   cout << "umValor = " << umValor << endl;
```

```
5     cout << "outroValor = " << outroValor << endl;
```

Neste caso, o seguinte código será executado:

```
1  void naoFuncionaTroca( int a, int b)
2  {
3     int temporario = a;
4     a = b;
5     b = temporario;
6     return;
7  }
```

Na sequência, o programa tenta novamente realizar a troca das variáveis. Desta vez é chamada a função *aquiFuncionaTroca* e novamente é impresso o resultado da permuta:

```
1     cout << "Trocando as variáveis... agora vai!!!" << endl;
2     aquiFuncionaTroca( umValor, outroValor );
3
4     cout << "umValor = " << umValor << endl;
5     cout << "outroValor = " << outroValor << endl;
```

Neste caso, o seguinte código será executado:

```
1  void aquiFuncionaTroca( int& a, int& b)
2  {
3     int temporario = a;
4     a = b;
5     b = temporario;
6     return;
7  }
```

2.2 O que ocorreu ?

Quando passamos argumentos para um função sem utilizar referência, em termos de programação dizemos que estamos passando uma variável por *valor*. Neste caso é criada uma *cópia* da variável em memória e qualquer alteração será realizada na cópia e não na variável original.

Quando passamos argumentos para uma função utilizando uma referência, em termos de programação dizemos que estamos passando uma variável por *referência*. Neste caso não é criada nenhuma cópia da variável em memória e qualquer alteração será realizada na variável original.

Estes recursos estão disponíveis em qualquer linguagem de programação moderna.

3 Vamos entender um pouco como a memória funciona

O operador & é um operador unário que devolve endereço na memória de seu operando. Observe o programa 3:

Programa 3: Verificando os endereços de memória utilizados.

```
1  // trocando variáveis
2  // programa_003.cpp
3  #include "biblaureano.h"
4
5  void mostraValores( int a, int b);
6  void mostraValoresDeNovo( int& a, int& b);
7
8  int main()
```

```
9  {
10    int umValor, outroValor;
11
12    cout << "Endereço de umValor:" << &umValor << endl;
13    cout << "Endereço de outroValor:" << &outroValor << endl;
14
15    umValor = readInt("Entre com um valor:");
16    outroValor = readInt("Entre com outro valor:");
17
18    //chamando a função e passando variáveis por valor
19    mostraValores(umValor, outroValor);
20
21    //chamando a função e passando variáveis por referência
22    mostraValoresDeNovo(umValor, outroValor);
23
24    return 0;
25 }
26
27 void mostraValores( int a, int b)
28 {
29    cout << "A:" << a << endl;
30    cout << "Endereço de A:" << &a << endl;
31    cout << "B:" << b << endl;
32    cout << "Endereço de B:" << &b << endl;
33    return;
34 }
35
36 void mostraValoresDeNovo( int& a, int& b)
37 {
38    cout << "A:" << a << endl;
39    cout << "Endereço de A:" << &a << endl;
40    cout << "B:" << b << endl;
41    cout << "Endereço de B:" << &b << endl;
42    return;
43 }
```

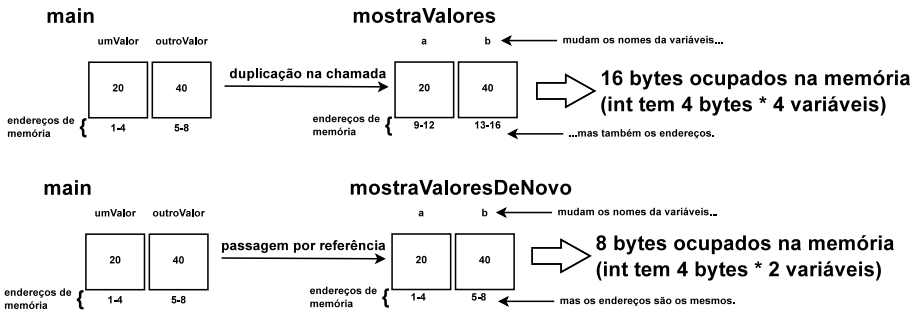

Figura 1: O que acontece na memória.

Como pode ser observado na figura 1, o programa 3 define duas funções que fazem praticamente a mesma coisa, a única diferença é que uma função está definida para trabalhar com as referências e outra não:

```
1 void mostraValores( int a, int b);
2 void mostraValoresDeNovo( int& a, int& b);
```

Logo na sequência, utilizamos o operador & para descobrir o endereço de memória das variáveis *umValor* e *outroValor*:

```
1    int umValor, outroValor;
2
3    cout << "Endereço de umValor:" << &umValor << endl;
4    cout << "Endereço de outroValor:" << &outroValor << endl;
```

Após a leitura dos dados, ocorre uma chamada a função *mostraValores* (que não está setado para receber os dados por referência, ou seja, recebe os dados por valor):

```
1    //chamando a função e passando variáveis por valor
2    mostraValores(umValor, outroValor);
```

O conteúdo da variável *umValor* é *copiado* para a variável *a* e o conteúdo da variável *outroValor* é *copiado* para a variável *b*. A função *mostraValores* também mostra os endereços de memória das variáveis *a* e *b*. Como os valores foram copiados, houve uma *duplicação* dos valores na memória e logo *a* e *b* ocupam áreas de memória diferentes de *umValor* e *outroValor*:

```
1  void mostraValores( int a, int b)
2  {
3     cout << "A:" << a << endl;
4     cout << "Endereço de A:" << &a << endl;
5     cout << "B:" << b << endl;
6     cout << "Endereço de B:" << &b << endl;
7     return;
8  }
```

Mas com a chamada da função *mostraValoresDeNovo* (que está setado para receber os dados por referência), não existe a duplicação dos dados, pois as variáveis *a* e *b* *compartilham* o mesmo endereço de memória das variáveis *umValor* e *outroValor*:

```
1    //chamando a função e passando variáveis por referência
2    mostraValoresDeNovo(umValor, outroValor);
```

Como as variáveis *compartilham* a mesma memória, os endereços de memória de *a* e *b* serão os mesmos de *umValor* e *outroValor*:

```
1  void mostraValoresDeNovo( int& a, int& b)
2  {
3     cout << "A:" << a << endl;
4     cout << "Endereço de A:" << &a << endl;
5     cout << "B:" << b << endl;
6     cout << "Endereço de B:" << &a << endl;
7     return;
8  }
```

4 Atividades

1. Faça um programa simples que contenham as seguintes funções:
   - Uma para ler três valores inteiros do teclado;
   - Outra para retornar o maior valor.
2. Quais as vantagens e desvantagens de passar um argumento por valor?
3. Quais as vantagens e desvantagens de passar um argumento por referência?