Aquele que pergunta, pode ser um tolo por cinco minutos. Aquele que deixa de perguntar, será um tolo para o resto da vida.

provérbio chinês

1 Funções e parâmetros padrões

Em alguns casos as funções que criamos sempre recebem um mesmo valor como parâmetro quando chamadas. Um exemplo seria uma função para definir a resolução do vídeo no momento do jogo, supondo que os valores padrões sejam 1024×768 você não precisaria a todo instante passar este valores, somente alterando a chamada da função nos casos que os valores passados nos parâmetros não sejam os padrões.

A linguagem C++ permite criar funções com parâmetros que podem assumir um valor padrão. Sempre que você não informar um valor para aquele parâmetro, o parâmetro receberá o valor padrão definido no momento da sua criação. O programa 1 ilustra este conceito.

Programa 1: Funções e valores padrões.

```
1  // funções e valores padrões
2  // programa_001.cpp
3  #include "biblaureano.h"
4
5  int digaUmNumero(int maior, int menor = 1);
6
7  int main()
8  {
9     int numero;
10
11    numero = digaUmNumero(10);
12    cout << "Você informou o número " << numero << endl;
13
14    numero = digaUmNumero(15,10);
15    cout << "Você informou o número " << numero << endl;
16
17    cout << "Game over!!!" << endl;
18    return 0;
19 }
20
21 int digaUmNumero(int maior, int menor)
22 {
23    int numero;
24    do
25    {
26       cout << "Por favor entre com um número entre "
27            << menor
28            << " e "
29            << maior
30            << " :";
31       numero = readInt();
32    } while( numero > maior || numero < menor);
33    return numero;
34 }
```

1.1 Entendendo o programa

Na definição do protótipo da função informamos o valor padrão a ser assumido por um parâmetro no caso de um valor não ser passado pelo programador. Neste caso o parâmetro *maior* sempre deve ser informado e o parâmetro

menor é opcional e assume o valor padrão 1 caso não seja informado outro valor:

```
1 int digaUmNumero(int maior, int menor = 1);
```

No momento da chamada de cada função são passados os valores (parâmetros) para a função. Na primeira chamada a função *digaUmNumero()* é passado apenas o valor para o parâmetro *maior*. O parâmetro *menor* assume o valor padrão definido:

```
1     numero = digaUmNumero(10);
2     cout << "Você informou o número " << numero << endl;
```

Na segunda chamada a função *digaUmNumero()* são passados os valores para os dois parâmetros, neste caso o parâmetro *menor* assume o valor informado pelo programador e não o valor padrão:

```
1     numero = digaUmNumero(15,10);
2     cout << "Você informou o número " << numero << endl;
```

Devemos tomar cuidado na declaração da função *digaUmNumero()*, pois o valor padrão é informado **apenas** na declaração do protótipo. No corpo da função **não** é informado o valor padrão:

```
1  int digaUmNumero(int maior, int menor)
2  {
3     int numero;
4     do
5     {
6        cout << "Por favor entre com um número entre "
7             << menor
8             << " e "
9             << maior
10            << " :";
11       numero = readInt();
12    } while( numero > maior || numero < menor);
13    return numero;
14 }
```

2 Criando funções com diferentes assinaturas

Você viu como especificar uma lista de parâmetros e um tipo de retorno único para cada função escrita. Mas e se você quiser uma função mais versátil que possa aceitar diferentes conjuntos de argumentos? Por exemplo, suponha que você queira escrever uma função que executa uma transformação em 3D em um conjunto de vértices que estão representados como *float*, mas você deseja que a função trabalhe com *int* também. Em vez de escrever duas funções separadas com dois nomes diferentes, você pode utilizar uma a função de *sobrecarga* de modo que uma única função possa manipular diferentes listas de parâmetros. Dessa forma, você poderia chamar uma função e passar como vértices ou *float* ou *int*.

Em orientação à objetos esta característica chama-se polimorfismo. Polimorfismo é a capacidade de se definir duas ou mais funções com o mesmo nome, ou seja, uma função definida de várias formas mas com o mesmo nome. O que muda são os valores de retorno ou os parâmetros de entrada.

Vejamos o programa 2:

Programa 2: Polimorfismo de funções.

```
1 // funções com múltiplas assinaturas
2 // programa_002.cpp
3 #include "biblaureano.h"
4
```

```
5  //protótipos da função
6  int retornaTriplo(int numero);
7  string retornaTriplo(string palavra);
8
9  float calculaBonus( int nivel, int placar );
10 float calculaBonus( int nivel, int placar, int dificuldade );
11
12 enum dificuldade { FACIL, INTERMEDIARIO, DIFICIL };
13
14 int main()
15 {
16    cout << "Triplo de 3 = " << retornaTriplo(3) << endl;
17
18    cout << "Triplo de xpto = " << retornaTriplo("xpto") << endl;
19
20    cout << "Bônus = " << calculaBonus(10,20) << endl;
21    cout << "Bônus = " << calculaBonus(10,20,FACIL)<< endl;
22    cout << "Bônus = " << calculaBonus(10,20,INTERMEDIARIO)<< endl;
23    cout << "Bônus = " << calculaBonus(10,20,DIFICIL)<< endl;
24
25    cout << "Game over!!!" << endl;
26    return 0;
27 }
28
29 //recebe inteiro e retorna inteiro
30 int retornaTriplo(int numero)
31 {
32    return (numero*3);
33 }
34
35 //recebe string e retorna string
36 string retornaTriplo(string palavra)
37 {
38    return (palavra+palavra+palavra);
39 }
40
41 //recebe apenas dois parâmetros
42 float calculaBonus( int nivel, int placar )
43 {
44    return (nivel+placar)/10.0;
45 }
46
47 //recebe três parâmetros
48 float calculaBonus( int nivel, int placar, int dificuldade )
49 {
50    if( dificuldade == FACIL)
51    {
52       return (nivel+placar)/10.0;
53    }
54    else if( dificuldade == INTERMEDIARIO )
55    {
56       return (nivel+placar)/9.0;
57    }
58    else if( dificuldade == DIFICIL )
59    {
60       return (nivel+placar)/8.0;
61    }
62 }
```

2.1 Entendendo o programa

Como pode ser observado, temos duas definições da função *retornaTriplo*. Uma que recebe e retorna um valor inteiro e outra que recebe e retorna um valor *string*.

```
1 //protótipos da função
2 int retornaTriplo(int numero);
3 string retornaTriplo(string palavra);
```

Será no momento da chamada a função que a decisão ocorrerá, caso seja passado um valor inteiro:

```
1    cout << "Triplo de 3 = " << retornaTriplo(3) << endl;
```

Será executada a função:

```
1  //recebe inteiro e retorna inteiro
2  int retornaTriplo(int numero)
3  {
4     return (numero*3);
5  }
```

E caso seja passado um valor *string*:

```
1    cout << "Triplo de xpto = " << retornaTriplo("xpto") << endl;
```

Será executada a função:

```
1  //recebe string e retorna string
2  string retornaTriplo(string palavra)
3  {
4     return (palavra+palavra+palavra);
5  }
```

O mesmo ocorre para a função *calculaBonus()*, que neste caso a diferença está no número de parâmetros passados:

```
1  float calculaBonus( int nivel, int placar );
2  float calculaBonus( int nivel, int placar, int dificuldade );
```

Caso sejam passados dois parâmetros na chamada da função:

```
1    cout << "Bônus = " << calculaBonus(10,20) << endl;
```

Será executada a função:

```
1  //recebe apenas dois parâmetros
2  float calculaBonus( int nivel, int placar )
3  {
4     return (nivel+placar)/10.0;
5  }
```

E caso sejam passados três parâmetros na chamada da função:

```
1     cout << "Bônus = " << calculaBonus(10,20,FACIL)<< endl;
2     cout << "Bônus = " << calculaBonus(10,20,INTERMEDIARIO)<< endl;
3     cout << "Bônus = " << calculaBonus(10,20,DIFICIL)<< endl;
```

Será executada a função:

```
1  //recebe três parâmetros
2  float calculaBonus( int nivel, int placar, int dificuldade )
3  {
4     if( dificuldade == FACIL)
5     {
6        return (nivel+placar)/10.0;
7     }
8     else if( dificuldade == INTERMEDIARIO )
9     {
10       return (nivel+placar)/9.0;
11    }
12    else if( dificuldade == DIFICIL )
13    {
14       return (nivel+placar)/8.0;
15    }
16 }
```

Como você deve ter reparado, a decisão de qual função será executada é dado pelo tipo do parâmetro ou a quantidade de parâmetros informados. Logo, não é possível criar funções que recebem o mesmo tipo de parâmetro mas com retorno diferente, pois não é possível determinar em tempo de compilação qual a função que será chamada. Por exemplo, não seria possível criar funções como:

```
1    int bonus(int valor);
2    float bonus(int valor);
```

3 Em busca da velocidade - funções *inline*

A cada chamada de função ocorre uma pequena perda de tempo, pois o programa precisa desviar a execução do programa para a chamada da função. Uma forma de evitar esta perda é criarmos funções *inline* (em linha). Um função *inline* é idêntica as funções normais em funcionamento, a diferença ocorre no momento da compilação já que o compilador copia o código da função *inline* para o local onde ela é chamada. Por exemplo:

```
1  inline void mensagem()
2  {
3     cout << "Game over!!" << endl;
4     return;
5  }
6
7  int main()
8  {
9     mensagem();
10    return 0;
11 }
```

Quando o compilador compilar este código, ele se tornará apenas:

```
1  int main()
2  {
3     cout << "Game over!!" << endl;
4     return 0;
5  }
```

Para declarar uma função em linha basta acrescentar o comando *inline* antes do nome da função. Como pode ser observado no programa 3, a função *radiacao()* é declarada como uma função *inline*.

Programa 3: Funções *inline*.

```
1  // Deixando o seu programa mais rápido
2  // programa_003.cpp
3  #include "biblaureano.h"
4
5  //protótipos da função
6  inline float radiacao(float vida);
7
8  int main()
9  {
10    float vida = 100;
11
12    cout << "Sua vida está " << vida << "%!" << endl;
13
14    vida = radiacao(vida);
15    cout << "Após exposição a radiação, sua vida está em "
16         << vida << "%!" << endl;
17
```

```
18     vida = radiacao(vida);
19     cout << "Após exposição a radiação, sua vida está em "
20          << vida << "%!" << endl;
21
22     vida = radiacao(vida);
23     cout << "Após exposição a radiação, sua vida está em "
24          << vida << "%!" << endl;
25
26     cout << "Game over!!!" << endl;
27     return 0;
28 }
29
30 inline float radiacao( float vida)
31 {
32     return vida * 0.9;
33 }
```

A utilização de funções *inline* aumentam o tamanho do código executável final e o seu uso deve ocorrer apenas para funções *pequenas* (uma ou duas linhas de código). É possível declarar funções *inline* com mais linhas, mas o tamanho do executável e o consumo de memória adicional não compensará o ganho de velocidade economizado nas chamadas a função.

4 Atividades

1. No exemplo da função *calculaBonus()* (2) é possível escrever apenas uma função para atender os dois casos ?
2. Existe um comando chamado *assert*, pesquise sobre o seu funcionamento e responda como ele poderia ser utilizado na validação de parâmetros em funções.