A repetição deixa sua marca até nas pedras.

provérbio árabe

1 Trabalhando com escopos - a teoria

O escopo de uma variável é o alcance que ela tem, de onde pode ser acessada. O escopo de uma variável é simplesmente a região do programa em que ela é utilizável, ou seja, entende-se como escopo de variáveis a área onde o valor e o nome dela tem significado.

Pode-se ter dois tipos de variáveis na linguagem C++:

O primeiro tipo é a *variável global* : uma variável é global quando a mesma é definida fora de qualquer função. Esta variável pode ser usada em qualquer função e o significado dela abrange todo o programa fonte.

Uma *variáveis local* : é definida dentro de funções e o seu significado é somente válido dentro da função. Assim têm-se duas variáveis com o mesmo nome em funções diferentes.

2 Trabalhando com escopos - a prática com variáveis locais

Quando uma variável é definida dentro de uma função, está sendo definindo uma variável local à função. Esta variável utiliza a pilha interna da função como memória, portanto ao final da função este espaço de memória é liberado e a variável não existe mais. Portanto a definição da variável só é valida dentro da função. Os parâmetros de uma função também são considerados variáveis locais e também utilizam a pilha interna para a sua alocação.

O programa 1 apresenta várias situações no uso de variáveis locais e escopos. Lembre-se que o escopo de uma variável é o contexto onde ela foi definida e este programa apresenta vários contextos diferentes.

Programa 1: Trabalhando com escopos - variáveis locais.

```
1  // trabalhando com escopos
2  // programa_001.cpp
3  #include "biblaureano.h"
4
5  void semUtilidade();
6
7  int main()
8  {
9     int qualquer = 5; //variável local de main
10
11    cout << "Em main() qualquer é:" << qualquer << endl;
12
13    semUtilidade();
14
15    cout << "Voltando para main() e qualquer continua com:" << qualquer << endl;
16
17    //sempre que abro chaves, estou iniciando um novo bloco
18    // e um novo escopo
19    {
20       cout << "Em main() o novo escopo de qualquer é:" << qualquer << endl;
21       cout << "Criando um novo qualquer neste novo escopo." << endl;
22
23       int qualquer = 37;
24       cout << "Em main(), no novo escopo, qualquer é:" << qualquer << endl;
25    }
26
```

```
27     cout << "Chegando ao fim de main() e qualquer criado no novo escopo não existe mais."
28          << endl;
29
30     cout << "Fim de main() e qualquer é:" << qualquer << endl;
31     cout << "Game over!!!" << endl;
32     return 0;
33 }
34
35 void semUtilidade()
36 {
37     int qualquer = -24; //variável local de semUtilidade;
38     cout << "Em semUtilidade() qualquer é:" << qualquer << endl;
39     return;
40 }
```

2.1 Entendendo o programa

Na primeira parte do programa principal (*main()*) declaramos uma variável *qualquer*, esta variável é local e é válido apenas o contexto do *main()*.

```
1     int qualquer = 5; //variável local de main
```

Até o momento, não existe nada de novo a ser acrescentando. Na sequência, o valor de *qualquer* é impresso e ocorre uma chamada para a função *semUtilidade()*.

```
1     cout << "Em main() qualquer é:" << qualquer << endl;
2
3     semUtilidade();
```

A função *semUtilidade()* também define uma variável *qualquer*, mas com valor diferente, e realiza a impressão do conteúdo desta variável. Bom, o que ocorre na prática ?

```
1 void semUtilidade()
2 {
3     int qualquer = -5;  //variável local de semUtilidade;
4     cout << "Em semUtilidade() qualquer é:" << qualquer << endl;
5     return;
6 }
```

Observando a figura 1 percebemos que, na prática, todo programa ocupa uma espaço de memória durante sua execução e que esta memória é dividida entre as diversas funções (neste caso, *main()* e *semUtilidade()*) que compõem todo o programa. Logo, uma função não *enxerga* a área de memória de outra função.

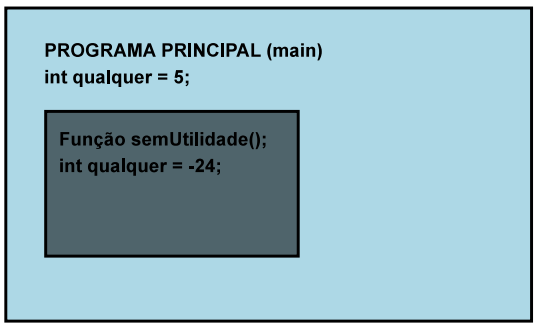


Figura 1: Representação da memória do programa 1.

No retorno para a função *main()* ocorre novamente a impressão da variável *qualquer*, que como pode ter sido observado na execução do programa, manteve o seu valor inalterado (a variável *qualquer* da função *semUtilidade()* não afetou o conteúdo da variável *qualquer* da função *main()*).

```
1    cout << "Voltando para main() e qualquer continua com:" << qualquer << endl;
```

Faça a seguinte analogia. Se numa sala de aula pode ter duas alunas chamadas de Maria, que se vestem, agem, falam de forma diferentes, porque não poderíamos ter duas variáveis com o mesmo nome ? Reforçando novamente que toda variável tem um escopo/contexto de atuação.

Na sequência, é criado um novo escopo chamado de *escopo aninhado* (*nested scope*). Um escopo aninhado é um escopo que é lexicalmente (textualmente) encapsulado dentro de outro escopo. Em outras palavras, o escopo aninhada é limitado pelo delimitador de escopo ({ }).

```
1    //sempre que abro chaves, estou iniciando um novo bloco
2    // e um novo escopo
3    {
4        cout << "Em main() o novo escopo de qualquer é:" << qualquer << endl;
5        cout << "Criando um novo qualquer neste novo escopo." << endl;
6
7        int qualquer = 37;
8
9        cout << "Em main(), no novo escopo, qualquer é:" << qualquer << endl;
10   }
```

Ao entrar no novo escopo, a variável *qualquer* esta disponível para uso normalmente, sendo possível alterar o seu conteúdo. Mas ao declararmos uma nova variável *qualquer* dentro deste novo escopo, estamos *dizendo* para o programa que não utilizaremos a variável *qualquer* que já está disponível, ou seja, estamos criando uma nova entidade (variável) que por *coincidência* tem o mesmo nome.

Trabalhar com escopos aninhados significa trabalhar com outros contextos de execução, logo, a memória é diferente para cada escopo.

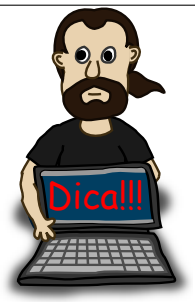
Ao término do escopo aninhado (ou seja, logo após encontrar o }) a variável *qualquer* retorna ao seu valor original.

```
1    cout << "Chegando ao fim de main() e qualquer criado no novo escopo não existe mais."
2         << endl;
3
4    cout << "Fim de main() e qualquer é:" << qualquer << endl;
```

Ao declarar variáveis dentro de outras estruturas (tais como *for*, *if* e outras) estas variáveis não estarão mais disponíveis para uso após o término da estrutura. Como ocorre no próximo exemplo:

```
1  for( int i=0 ; i<10; ++i)
2  {
3     cout << i << endl;
4  }
5  //a variável i não existe mais
```

Cuidado: não é uma boa ideia ficar criando variáveis em escopos aninhados dentro do seu programa. Uma série variáveis criadas em escopos aninhados pode levar a falta de controle e causar confusão.



Os parâmetros de uma função podem ser acessados da mesma maneira que variáveis locais. Eles na verdade funcionam exatamente como variáveis locais, e modificar um argumento não modifica o valor original no contexto da chamada de função, pois, ao dar um argumento numa chamada de função, ele é copiado como uma variável local da função.

3 Variáveis globais - a tentação e o descontrole

As variáveis globais são definidas fora de qualquer função e o seu nome é válido para todo o programa. Qualquer função que alterar o seu conteúdo estará alterando para todo o programa pois estas variáveis ficam em uma área de dados disponível para todo o programa.

O programa 2 demonstra a utilização de uma variável global. A vantagem no uso de variáveis globais é a facilidade com que sem tem acesso ao seu conteúdo, pois não é necessário passar o conteúdo da variável como parâmetro nas funções que a usam. Esta também é a sua desvantagem, pois o uso de variáveis globais não é recomendado em projetos de desenvolvimento de qualquer tipo, pois *perde-se* o controle sobre o conteúdo da variável.

Programa 2: Trabalhando com escopos - variáveis globais.

```
1 // trabalhando com escopos
2 // programa_002.cpp
```

```
3  #include "biblaureano.h"
4
5  //declaração dos protótipos das funções
6  void acessoGlobal();
7  void brincandoEscondeEsconde();
8  void mudancaGlobal();
9
10 //declaração da variável pública
11 int global=5;
12
13 int main()
14 {
15    cout << "Estou em main() e global vale " << global << endl;
16
17    acessoGlobal();
18
19    brincandoEscondeEsconde();
20
21    cout << "Estou em main() e global vale " << global << endl;
22
23    mudancaGlobal();
24    cout << "Estou em main() e global vale " << global << endl;
25
26    acessoGlobal();
27
28    cout << "Game over!!!" << endl;
29    return 0;
30 }
31
32 void acessoGlobal()
33 {
34    cout << "Estou em acessoGlobal() e global vale "
35         << global
36         << endl;
37    return;
38 }
39
40 void brincandoEscondeEsconde()
41 {
42    int global = 99;
43    cout << "Estou em brincandoEscondeEsconde() e global vale "
44         << global
45         << endl;
46    return;
47 }
48
49 void mudancaGlobal()
50 {
51    cout << "Estou em MudancaGlobal() e global vale "
52         << global
53         << endl;
54    cout << "Resolvi alterar a variável global!!"
55         << endl;
56
57    //alterando o conteúdo da variável
58    ++global;
59
60    cout << "Estou em mudancaGlobal() e agora global vale "
61         << global
62         << endl;
63
64    return;
65 }
```

3.1 Entendendo o programa

Inicialmente declaramos a variável *global*. Repare que a declaração desta variável ocorre antes do início da função *main()*.

```
//declaração da variável pública
int global = 5;

int main()
{
    ...
}
```

Na função *main()* o acesso a variável *global* é normal, pois como a variável está disponível para acesso em todos os escopos de execução.

```
int main()
{
    cout << "Estou em main() e global vale " << global << endl;

    acessoGlobal();

    brincandoEscondeEsconde();

    cout << "Estou em main() e global vale " << global << endl;

    mudancaGlobal();
    cout << "Estou em main() e global vale " << global << endl;

    acessoGlobal();

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

A função *acessoGlobal()* também faz referência ao conteúdo da variável *global*. Repare que não ocorre nenhuma declaração de nova variável nesta função:

```
void acessoGlobal()
{
    cout << "Estou em acessoGlobal() e global vale "
         << global
         << endl;
    return;
}
```

Já a função *mudancaGlobal()* além de fazer referência ao conteúdo da variável *global* também altera o seu conteúdo, ou seja, no retorno para a função *main()* a variável *global* já terá o novo valor com o incremento:

```
void mudancaGlobal()
{
    cout << "Estou em mudancaGlobal() e global vale "
         << global
         << endl;
    cout << "Resolvi alterar a variável global!!"
         << endl;

    //alterando o conteúdo da variável
    ++global;

    cout << "Estou em mudancaGlobal() e agora global vale "
         << global
         << endl;

    return;
}
```

Devemos tomar cuidado com a função *brincandoEscondeEsconde()*, pois neste função ocorre a declaração de uma nova variável *global*, só que o escopo desta variável é **local** a função onde ela foi declarada:

```
1 void brincandoEscondeEsconde()
2 {
3     int global = 99;
4     cout << "Estou em brincandoEscondeEsconde() e global vale "
5          << global
6          << endl;
7     return;
8 }
```

A figura 2 ilustra o que ocorre na memória do programa

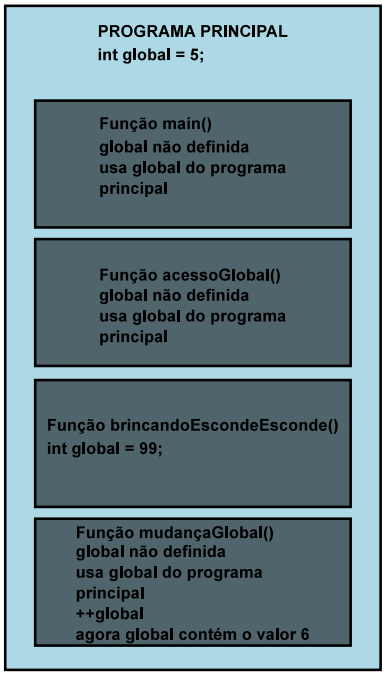


Figura 2: Representação da memória do programa 2.

Cuidado: não é uma boa ideia utilizar variáveis globais no seu programa. Mas se utilizar, tome cuidado para não criar variáveis locais com o mesmo nome de uma variável global pois pode causar confusão.

Quando declaramos as variáveis, nós podemos fazê-lo:

- Dentro de uma função;
- Fora de todas as funções inclusive a *main()*.

As primeiras são as designadas como locais: só têm validade dentro do bloco no qual são declaradas. As últimas são as globais, elas estão vigentes em qualquer uma das funções.

Quando uma função tem uma variável local com o mesmo nome de uma variável global a função dará preferência à variável local.

Daqui conclui-se que podemos ter variáveis com o mesmo nome. Apenas na situação em que temos 2 variáveis locais no mesmo escopo é que é colocada a restrição de termos nomes diferentes caso contrário não conseguiríamos distinguir uma da outra.

4 Utilizando constantes globais - um bom uso para variáveis globais

No programa 3 podemos observar um bom uso para as variáveis globais, ou seja, declarando constantes globais.

Programa 3: Constantes globais.

```
1  // trabalhando com escopos
2  // programa_003.cpp
3  #include "biblaureano.h"
4
5  //protótipo da função
6  void criaEspaconave(int identificacao);
7
8  //declaração da variável CONSTANTE pública
9  const int MAX_ESPACONAVES = 15;
10
11 int main()
12 {
13    for(int i=1; i<= MAX_ESPACONAVES; ++i)
14    {
15       criaEspaconave(i);
16    }
17    cout << "Game over!!!" << endl;
18    return 0;
19 }
20
21 void criaEspaconave(int identificacao)
22 {
23    cout << "Criando a espaçonave "
24         << identificacao
25         << " de "
26         << MAX_ESPACONAVES
27         << " possíveis..."
28         << endl;
29    return;
30 }
```

5 Atividades

Para todos os exercícios você terá que montar um programa para validar a função.

1. Faça uma função que verifique se um número é primo ou não. Ela deve retornar *true* se o número informado for primo e *false* se não for.
2. Faça uma função que retorne quantos números primos existem entre o intervalo 1 e N. Para N=12, a função deveria imprimir o valor 6, já que os números 1,2,3,5,7,11 são primos e menores que 12.
3. Implemente uma função que recebe uma data no formato 'DD/MM/AAAA' e retorne a mesma por extenso. Por exemplo: a data *27/12/1975* deve retornar *27 de dezembro de 1975*.
4. Escrever uma função que recebe número do mês e retorna seu nome por extenso.
5. Faça uma função que retorne o n-ésimo termo da sequência de Fibonacci. Sendo os primeiros 8 termos da sequência: 1,1,2,3,5,8,13,21 uma chamada a função passando o valor 3 deve retornar 2, uma chamada passando o valor 8 deve retornar 21.
6. A fórmula da permutação é $P(n,r) = \frac{n!}{(n-r)!}$ para $0 \leq r \leq n$. Faça um programa para ler os valores de n e r e crie uma função chamada *Permutacao* que recebe *n* e *r* e retorna o resultado.
7. Faça um programa para ler a quantidade de termos t, um valor inicial n e um quociente q e crie uma função chamada *geometrica*. Seu programa deve calcular a progressão até a quantidade de termos. Considere $0 \leq t \leq n$.
8. A fórmula da combinação é $C(n,r) = \frac{n!}{r! \times (n-r)!}$ para $0 \leq r \leq n$. Faça um programa para ler os valores de n e r e crie uma função chamada *Combinacao* que recebe *n* e *r* e retorna o resultado.
9. Faça um programa para ler a quantidade de termos t, um valor inicial n e uma constante r e crie uma função chamada *aritmetica*. Seu programa deve calcular a progressão até a quantidade de termos. Considere $0 \leq t \leq n$.
10. A fórmula do arranjo simples é $A_r^n = \frac{n!}{(n-r)!}$ para $1 \leq r \leq n$. Faça um programa para ler os valores de n e r e crie uma função chamada *Arranjo* que recebe *n* e *r* e retorna o resultado.
11. Faça uma função que retorna o maior valor entre 2 valores passados.
12. Faça uma função que retorna o menor valor entre 2 valores passados.
13. Faça uma função que retorna o maior valor entre 3 valores passados.
14. Faça uma função que retorna o menor valor entre 3 valores passados.
15. É possível implementar os 2 exercícios anteriores utilizando as funções que retornam maior e menor recebendo apenas 2 valores?
16. Faça uma função que retorne o MDC (máximo divisor comum) entre 2 números naturais.
17. Faça uma função que retorne o MMC (mínimo múltiplo comum) entre 2 números naturais.
18. Faça uma função que monte um triângulo isósceles de largura e altura N, para N = 6 o triângulo seria:

```
*
**
***
****
```

```
*****
******
```

19. Faça uma função que monte um triângulo com altura $2N - 1$ e largura N, para N = 4 o triângulo seria:

```
*
**
***
****
***
**
*
```

20. Faça uma função que monte um triângulo com altura N e largura $2N - 1$, para N=6 o triângulo seria:

```
     *
    ***
   *****
  *******
 *********
***********
```