C++
Básico ao Avançado
Constante

Heitor Rodrigues Savegnago

UFABC Rocket Design

2017.3

Modificando mais

- Vocês ainda lembram os modificadores?
  1. `signed`
  2. `unsigned`
  3. `short`
  4. `long`
- Vocês ainda lembram o que cada um faz?
  1. Números com sinal
  2. Números sem sinal
  3. Diminui a faixa de valores
  4. Aumenta a faixa de valores
- Lembram de mais algum?
- Prontos para mais modificadores?

Para todo ($\forall$)

- Qual o problema de utilizar uma variável global?
- *Qualquer um pode alterar seu valor*
- E se ela fosse emcapsulada?
- Podemos definir variáveis com comportamento global, porém são locais
- *Isso não faz sentido...*
- Vamos pensar numa função que tenha um contador (variável)
- E é chamada várias vezes
- Seu contador não pode ser alterado entre chamadas
- A variável tem que ser *invariável* entre chamadas
- Deve ser a mesma variável para toda chamada da função

Uma solução possível

```
int contador = 0;

int contadorMaisUm()
{
  return ++contador;
}
```

- Mas variável global é ruim...
- O novo modificador faz com que a variável não seja redefinida em toda chamada
- Sua palavra-chave é `static`
- Sua função é manter a variável armazenada mesmo depois da saída do escopo
- Não é necessário desalocar memória

Com static

```
int contadorMaisUm()
{
  static int contador = 0;
  return ++contador;
}
```

- *Só isso?*
- Sim, ela é bem direta

Em classes

- Para classes, todo os membros podem receber este argumento, inclusive métodos
- O que acontece com membros `static` em classes?
- *Atributos são comuns à todas as instâncias da classe*
- Ou seja, você tem um valor que se comporta como global para qualquer objeto da classe
- *Isso faz sentido?*
- Pensa na gravidade, para todos nós é a mesma
- Se muda pra um, muda pra todos

Outro operador

- Utilizaramos um novo operador para acessar membros estáticos
- O operador de acesso a membro estático, ou operador de escopo
- Utiliza um par de dois-pontos (::)
- É binário e de consulta

```
<escopo>::<membro>;
```

- E para o caso das classes, não é necessária a utilização de um objeto
- Para atributos, é necessária a inicialização externa

Vários detalhes

```
class A
{
  public:
    static int a;
    static int getA(){return a;}
} alpha, beta;

int A::a = 10;

int main()
{
  A::a=10;
  int B = A::getA();
  return 0;
}
```

- *Ah, então aquele `std::` é isso?*
- Não... Mas é quase

Não é literal

- Qual a utilidade de uma variável que não varia?
- *Nenhuma*
- E se o valor dela for passado pelo usuário e não puder ser alterado?
- *Aí eu simplesmente não altero ele...*
- Quem te garante?
- E se existisse uma varável que você não pode alterar o valor, apenas inicializar?
- *E aquele* `#define`*, não é isso?*
- Não, nele você define uma palavra que será substiituída por um literal
- Aqui inicializamos uma variável utilizando um valor do programa

Constante

- Existe um modificador especial para estes casos
- O modificador `const` impede que o valor de uma variável seja alterado
- *Mas qual a utilidade de uma variável que não varia?*
- Não variar!
- *Dã?*
- Em geral, utilizamos esse modificador em classes que declaram vetores
- Os objetos dessas classes precisam saber o tamanho do vetor, e este tamanho não pode variar
- Também é possível utilizar este modificador para evitar cópias de variáveis, e garantir integridade de dados
- *É o que?*
- Vamos lá...

Para funções

- Lembra do operador de endereço? &
- Lembra que eu fiz um breve comentário sobre ele fazer outra coisa?
- Ele consegue criar variáveis com comportamnto fantasmal!
- *Agora esse cara endoidou de vez...*
- Com ele, você pode declarar uma variável que é apenas um símbolo novo para uma variável existente
- Ou seja, um ponteiro para o mesmo endereço de meória, porém sem precisar de operadores especiais

O fantasma

```
//...
int A(5);
//...
int &B(A);
//...
B=10;
//...
A==5;          //Falso
//...
```

- *Legal, e pra que isso serve?*
- Com essa sintaxe, criamos funções que recebem a variável real
- Estas funções podem alterar o valor da variável original, não é só uma cópia

Função fantasma

```
void kill(int &a)
{
  a = 0;
}
```

- Há algo errado aqui?
- O valor da variável é destruído
- Por isso é normal colocar uma coisinha a mais nesse código, para proteger a variável

Protegendo

```
void kill(const int &a)
{
  a = 0;
}
```

- **Esse código tem um erro**
- Se você tentar compilá-lo, ele apontará tentativa de alteração em Constante
- Além deste caso, temos o caso de objetos constantes

Objeto constante

- Não pode sofrar alteração
- Não pode acessar métodos, pois eles poderiam fazer alterações
- *Mas e os `get`? Eles não alteram*
- Para estes casos, temos novidade...

Constante acessível

```
class alpha
{
  int x;
public:
  void setX(int x){this->x=x;}
  int getX() const {return x;}
};
```

- Colocamos a palavra-chave `const` após a assinatura, antes da declaração
- Ela informa ao compilador que este método não altera os atributos do objeto

Vamos testar!