Introdução a POO

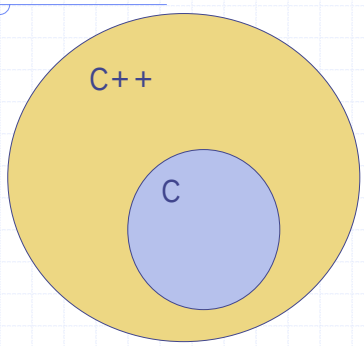
Marcio Santi

Linguagem C++

Introdução a Linguagem C++ e POO

- Programação Orientada a Objetos (POO) e C++
- Recursos C++ não relacionados às classes
- Incompatibilidades entre C e C++
- Classes em C++
- Encapsulamento
- Construtores e destrutores
- Variáveis e métodos *static*
- Classes e métodos *friend*
- Sobrecarga de Operadores
- Herança
- Polimorfismo
- Visão de modelagem em POO

C++: Definição



- Todo código C pode ser compilado, *a priori*, em C++
- O nome original dado por Stroustrup para a sua linguagem foi "C with Classes"
- Principal motivação: *desenvolvimento de uma linguagem de programação que suportasse o paradigma da programação orientada a objetos*
- POO:
  - classes
  - encapsulamento
  - herança
  - polimorfismo

Exemplo: Pilha de caracteres

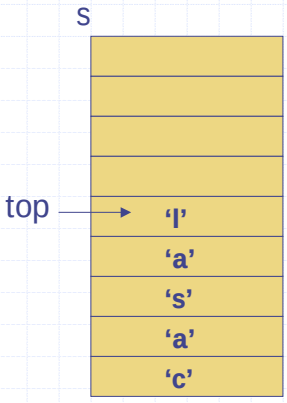
push: insere um elemento na pilha

pop: retira um elemento na pilha

FILO: first in, last out

```
struct stack
{
    int top;
    char s[max_len];
}
```

Sempre no topo



Recursos C++ não relacionados às classes

- Comentário
- Métodos ou mensagens
- Declaração de variáveis
- Declaração de tipos
- Protótipos de funções
- Funções que não recebem parâmetros
- Funções *inline*
- Referências
- Alocação dinâmica de memória
- Valores *default* para parâmetros de funções
- Sobrecarga de nomes de funções
- Operador de escopo

Comentários

- Em C++, os caracteres **//** iniciam um comentário que termina no fim da linha na qual estão estes caracteres
- Por exemplo:

```
int main(void)
{
   return -1;    // retorna o valor -1 para o Sistema Operacional
}
```

Declaração de variáveis

- Em C++ é possível declarar variáveis em qualquer trecho do código, e não apenas no início dos blocos
- O objetivo deste recurso é minimizar declarações de variáveis não inicializadas. Se a variável pode ser declarada em qualquer ponto, ela pode ser sempre inicializada na própria declaração.

```
void main(void)
{
   int a;
   a = 1;
   printf("%d\n", a);
   // ...
   char b[] = "teste";
   printf("%s\n", b);
}
```

```
for (int i=0; i<20; i++)
{
   printf("%d\n", i);
}
```

Declaração de tipos

- Em C++, as declarações abaixo são equivalentes:

```
struct a {
   // ...
};

typedef struct a {
   // ...
} a;
```

- Não é mais necessário o uso de *typedef* neste caso.
- A simples declaração de uma estrutura já permite que se use o nome sem a necessidade da palavra reservada *struct*.

Protótipos de funções

- Em C++ uma função só pode ser usada se esta já foi declarada.
- Chamadas de funções não declaradas em C causam um *warning*, mas em C++ geram erro de símbolo desconhecido.
- Os protótipos de C++ incluem não só o tipo de retorno da função, mas também os tipos dos parâmetros.

Funções que não recebem parâmetros

- Em C puro, um protótipo pode especificar apenas o tipo de retorno de uma função, sem dizer nada sobre seus parâmetros
- Por exemplo:
  - `float f(); // em C: prototype incompleto`
- Um compilador de C++ interpretará a linha acima como o protótipo de uma função que retorna um float e não recebe nenhum parâmetro:

```
float f(); // em C++ é o mesmo que float f(void);
```

Funções *inline*

- Têm como objetivo tornar mais eficiente, em relação à velocidade, o código que chama estas funções.
- São tratadas pelo compilador quase como uma macro definição.
- A chamada da função é substituída pelo corpo da função.
- Extremamente eficiente para funções pequenas, já que evita geração de código para a chamada e o retorno da função.
- O corpo de funções *inline* podem ser idênticos a uma função normal.
- A semântica de uma função e sua chamada é a mesma seja ela *inline* ou não.
- Utilizado em funções pequenas, pois funções *inline* grandes podem aumentar muito o tamanho do código gerado.

Funções *inline vs.* macros

```
inline double quadrado(double x)
{
   return x * x;
}

{
   double c = quadrado( 7 );
   double d = quadrado( c );
}

// ...

{
   double c = 7 * 7;
   double d = c * c;
}
```

```
#define quadrado(x) ((x)*(x))

// ...

void main(void)
{
   double a = 4;
   double b = quadrado(a++);
}

// ...

{

   double b = ((a++)*(a++));

}
```

Funções *inline*: vantagens

- Uma macro é uma simples substituição de texto, enquanto que funções *inline* são elementos da linguagem.
- Macros não podem ser usadas exatamente como funções, como mostrou o exemplo anterior.

Referências

- Uma referência para um objeto qualquer é, internamente, um ponteiro para o objeto.
- Uma variável que é uma referência é utilizada como se fosse o próprio objeto.
- Uma referência deve ser sempre inicializada.
- Referências se aplicam a objetos. A inicialização não pode ser feita com valores constantes.

Referências como variáveis locais

```
{
   int  a;          // ok, variável normal
   int& b = a;   // ok, b é uma referencia para a
   int& c;          // erro! não foi inicializada
   int& d = 12; // erro! inicialização inválida
}
```

```
{
   int  a = 10;
   int& b = a;
   printf("a=%d, b=%d\n", a, b); // produz a=10, b=10
   a = 3;
   printf("a=%d, b=%d\n", a, b); // produz a=3, b=3
   b = 7;
   printf("a=%d, b=%d\n", a, b); // produz a=7, b=7
}
```

Referência como tipo de parâmetro

```
void f(int a1, int &a2, int *a3)
{
   a1 = 1;  // altera cópia local
   a2 = 2;  // altera a variável passada (b2 de main)
  *a3 = 3;  // altera o conteúdo do endereço de b3
}
```

```
void main()
{
   int b1 = 10, b2 = 20, b3 = 30;
   f(b1, b2, &b3);
   printf("b1=%d, b2=%d, b3=%d\n", b1, b2, b3);
   // imprime b1=10, b2=2, b3=3
}
```

Referência como tipo de retorno de função

```
int& f()
{
   static int global;
   return global;      // retorna uma referência para a variável
}
```

```
void main()
{
   int a = 3;
   f() = a;              // altera a variável global
}
```

Alocação dinâmica de memória

- Programas C++ não precisam usar as funções calloc, malloc ou free para fazer alocação dinâmica.
- Para o gerenciamento da memória, existem dois operadores: new e delete.
- New aloca memória (substitui malloc).
- Delete libera memória (substitui free).

Alocação dinâmica: exemplo

```
{
   int * i1 = (int*)malloc(sizeof(int));         // C
   int * i2 = new int;                           // C++

   int * i3 = (int*)malloc(sizeof(int) * 10);    // C
   int * i4 = new int[10];                       // C++
}
```

```
{
   free(i1);          // alocado com malloc   ( C  )

   delete i2;         // alocado com new      ( C++)

   free(i3);          // alocado com malloc   ( C  )

   delete [] i4;      // alocado com new[]    ( C++ )
}
```

Valores *default* para parâmetros de função

- Em C++ existe a possibilidade de definir valores *default* para parâmetros de uma função, exemplo:

```
void impr( char* str, int x = -1, int y = -1)
{
   if (x == -1) x = wherex();
   if (y == -1) y = wherey();
   gotoxy( x, y );
   cputs( str );
}

// ...

impr( "especificando a posição", 10, 10 ); // x=10, y=10
impr( "só x", 20 );                        // x=20, y=-1
impr( "nem x nem y" );                     // x=-1, y=-1
```

Valores *default*: continuação

- A declaração do valor *default* só pode aparecer uma vez:
  - na implementação da função
  - No protótipo da função (melhor)

```
void impr( char* str, int x = -1, int y = -1 );
// ...
void impr( char* str, int x, int y )
{
   // ...
}
```

Sobrecarga de nomes de funções

- Permite que um nome de função possa ter mais de um *significado* (várias implementações diferentes).
- Cada implementação deverá apresentar também diferentes assinaturas (protótipos).
- A função a ser executada dependerá da forma como foi chamada.
- Em C puro também se fazia sobrecarga, porém implicitamente, ex: operadores aritméticos.

Sobrecarga de nomes de funções

```
void display( char *v )   { printf("%s", v); }
void display( int   v   )   { printf("%d", v); }
void display( float v   )   { printf("%f", v); }
```

```
display( "string" );
display( 123 );
display( 3.14159 );
```

Sobrecarga de nomes de funções

- O uso misturado de sobrecarga e valores *default* para parâmetros pode causar erros.

```
void f();
void f(int a = 0);

void main()
{
   f( 12 ); // ok, chamando f(int)
   f();      // erro!! chamada ambígua: f() ou f(int = 0)???
}
```

Operador de escopo

- C++ possui um novo operador que permite o acesso a nomes declarados em escopos que não sejam o corrente. Exemplo em C:

```
char *a;

void main(void)
{
   int a;
   a = 23;
   /* como acessar a variável global a??? */
}
```

- A declaração da variável local *a* esconde a global, tornando-a impossível de ser acessada.

Operador de escopo

- O operador de escopo possibilita o uso de nomes que não estão no escopo corrente, o que pode ser usado neste caso.
- Exemplo em C++:

```
char *a;

void main(void)
{
   int a;
   a = 23;
   ::a = “abc”;
}
```

Sintaxe:

escopo::nome

Exemplo:

- Reimplementação da pilha utilizando os novos recursos C++.
- Arquivo *stack3.cpp*.