Tipos Abstratos de Dados (TAD)

Módulos e Compilação em Separado

- Um programa em linguagem C/C++ pode ser dividido em vários arquivos fontes com extensão c/cpp.
- **Módulo**: arquivo que implementa funções que representam apenas parte da implementação do programa, ou seja um programa pode ser composto por um ou mais módulos.
- Cada módulo é compilado separadamente, gerando, para cada módulo, um arquivo objeto.
- Após a compilação individual de cada módulo, esses objetos são *ligados* ou ***linkados*** pelo *ligador* ou ***link-editor***.

Exemplo

str.h

```
int comp(char* s);
void cop(char* d, char* o);
void conc(char* d, char* o);
```

str.cpp

```
/* implementação das fçs
```

prg.cpp

```
#include  <stdio.h>
#include  "str.h"
int main( void )
{
   // programa que usa as fçs
}
```

```
> g++ –c str.cpp
> g++ –c prg.cpp
> g++ –o prg.exe str.o prg.o
```

- O mesmo arquivo str.c pode ser utilizado por outros programas ou módulos de programação que queiram utilizar funções utilitárias para a manipulação de strings

Exemplo Polígono

```
#ifndef _POINT
#define _POINT

struct ponto
{
  float x;
  float y;
};

void   attribPonto    ( ponto *p, float x, float y );
float  distanciaPonto ( ponto* p, ponto* q );

#endif
```

```
#include <math.h>

#include "point.h"

void attribPonto( ponto *p, float x, float y )
{
  p->x = x;
  p->y = y;
}

float distanciaPonto ( ponto* p, ponto* q )
{
  float x2 = (q->x-p->x)*(q->x-p->x);
  float y2 = (q->y-p->y)*(q->y-p->y);

  return sqrt( x2+y2 );
}
```

Exemplo Polígono

```
#ifndef _POLIGON
#define _POLIGON

#include "point.h"

struct poligono
{
  int    n;
  ponto *vp;
};

poligono  *criaPol      ( int n, ponto *vp );
void       destroiPol   ( poligono *pol    );
void       imprimePol   ( poligono *pol    );
float      perimetroPol ( poligono *pol    );

#endif
```

```
#include <iostream>
#include <math.h>

#include "poligon.h"

using namespace std;

poligono *criaPol( int n, ponto *vp )
{
  poligono* pol = new poligono;

  pol->n  = n;
  pol->vp = new ponto[n];

  for( int i=0; i<n; i++ )
    pol->vp[i] = vp[i];

  return pol;
}

void destroiPol( poligono* pol )
{
  delete [] pol->vp;
  delete pol;
}

void imprimePol( poligono *pol )
{
  cout << "Número de Vértices do Polígono: " << pol->n << endl;

  for( int i=0; i< pol->n; i++ )
  {
    float x = pol->vp[i].x;
    float y = pol->vp[i].y;

    cout << "ponto " << i+1 << ": (" << x << "," << y << ")" << endl;
  }
}

float perimetroPol( poligono *pol )
{
  float per = 0.0;

  for(int i=0; i<pol->n; i++)
  {
    int it = (i+1)-((i+1)/pol->n)*pol->n;

    per += distanciaPonto( &(pol->vp[it]), &(pol->vp[i]) );
  }
}
```

Exemplo: Polígono

```
#include <iostream>
#include <math.h>

using namespace std;

#include "point.h"
#include "poligon.h"

int main( )
{
  cout << "Entre com o numero de pontos: ";
  int n; cin  >> n;

  ponto *vp = new ponto[n];

  for( int i=0; i<n; i++ )
  {
    cout << "Entre com o ponto " << i+1 << "(x,y): ";
    float x, y; cin >> x >> y;

    attribPonto( &vp[i], x, y );
  }

  poligono *pol = criaPol( n, vp );

  delete [] vp;

  imprimePol( pol );

  cout << "Perimetro = " << perimetroPol( pol ) << endl;

  destroiPol( pol );

  return 1;
}
```

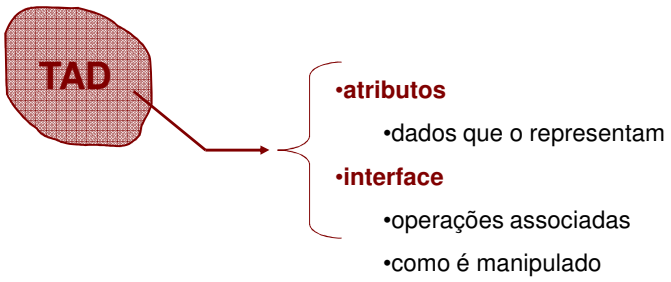
```
#ifndef _POINT
#define _POINT

struct ponto
{
  float x;
  float y;
};

void   attribPonto    ( ponto *p, float x, float y );
float  distanciaPonto ( ponto* p, ponto* q );

#endif
```

```
#ifndef _POLIGON
#define _POLIGON

#include "point.h"

struct poligono
{
  int    n;
  ponto *vp;
};

poligono  *criaPol      ( int n, ponto *vp );
void       destroiPol   ( poligono *pol    );
void       imprimePol   ( poligono *pol    );
float      perimetroPol ( poligono *pol    );

#endif
```

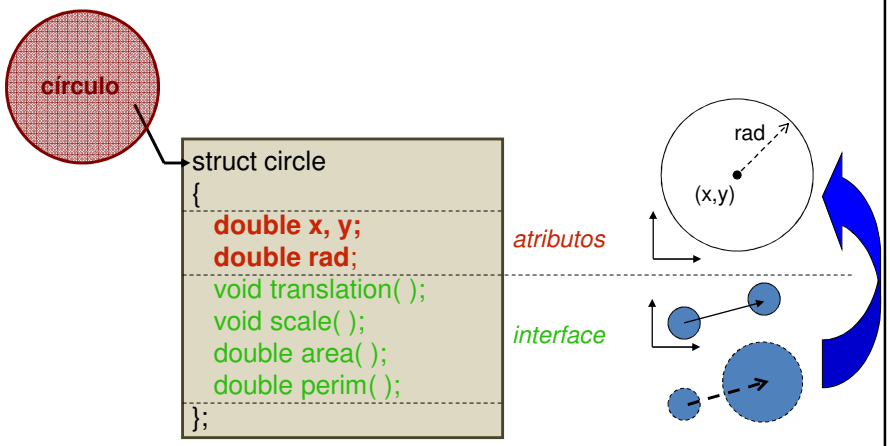
Tipos Abstratos de Dados (TAD)

- Em geral um módulo agrupa vários tipos e funções com funcionalidades relacionadas caracterizando uma finalidade bem definida.
- Quando um módulo define um novo tipo de dado e o conjunto de funções para manipular dados desse tipo, diz-se que esse módulo representa um **TAD** (tipo abstrato de dados).
- Abstrato nesse contexto quer dizer: *"esquecida a forma de implementação"*.
- Assim, um TAD é definido pela finalidade do tipo e das operações associadas a este tipo (funções que manipulam variáveis do tipo) e **não pela forma como está implementado (abstrato)**.

Tipos Abstratos de Dados



Exemplo: Editor Gráfico (classe círculo)



Exemplo: *Stack*

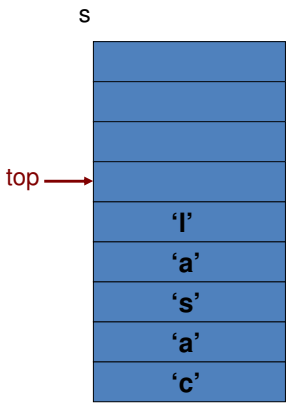
push: insere um elemento na pilha

pop: retira um elemento na pilha

FILO: first in, last out

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

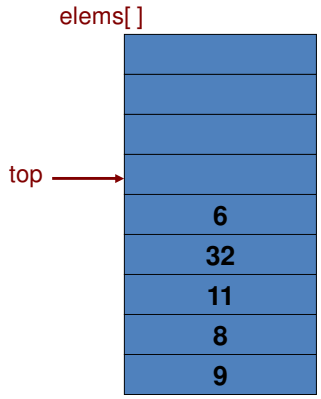
Sempre no topo



Exemplo: pilha de inteiros (*header file*)

stk.h

```
#ifndef stack_h
#define stack_h

#define MAX 50

struct Stack {
  int top;
  int elems[MAX];
};

void push(Stack* s, int i);
int  pop(Stack* s);
int  empty(Stack* s);
Stack* createStack(void);

#endif
```



TAD: Pilha de Números Reais

stk.h

```
#ifndef _STACK_H
#define _STACK_H

const unsigned int STACK_MAX = 50;

struct Stack
{
   int    top;
   float *elems;
};

Stack *createStack  ( int n );
void   destroyStack ( Stack *s );

void  push  ( Stack *s, float val );
float pop   ( Stack *s );
bool  empty ( Stack *s );
void  show  ( Stack *s );

#endif
```

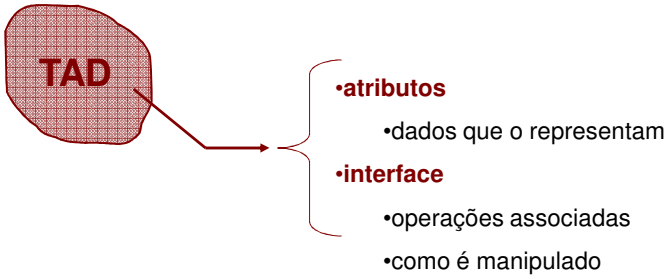
stk.cpp

```
using namespace std;

#include "stack.h"

Stack *createStack( int n )
{
  Stack *stk = new Stack;
  stk->top = 0;
  stk->elems = new float[n];

  return stk;
}

void destroyStack( Stack *s )
{
  delete s;
}

void  push( Stack *s, float val )
{
  s->elems[s->top++] = val;
}

float pop( Stack *s )
{
  return s->elems[--s->top];
}

bool empty( Stack *s )
{
  return s->top == 0;
}

void  show( Stack *s )
{
  cout.precision( 2 );

  for( int i=0; i<s->top; i++ )
  {
    cout << i << ": ";
    cout << fixed << s->elems[i] << endl;
  }
}
```

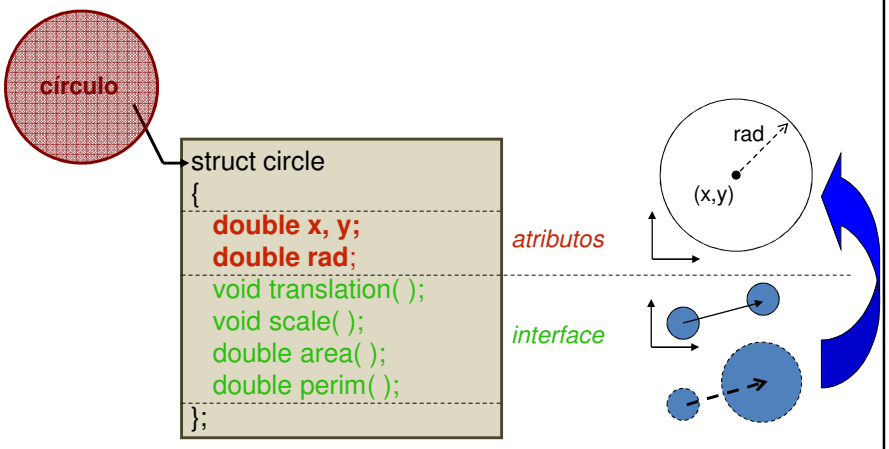
Cliente: Calculadora RPN

Classes e Encapsulamento

Tipos Abstratos de Dados



Exemplo: Editor Gráfico (classe círculo)



Funções Membro de Estruturas

- Conceito de estruturas em C++ permite que funções sejam definidas como membros.
- Declaração da função é incluída na definição da estrutura.
- Define-se operações diretamente relacionadas a um TAD:

```
#ifndef _STACK_H
#define _STACK_H

const unsigned int STACK_MAX = 50;

struct Stack
{
   int    top;
   float *elems;
};

Stack *createStack  ( int n );
void   destroyStack ( Stack *s );

void  push  ( Stack *s, float val );
float pop   ( Stack *s );
bool  empty ( Stack *s );
void  show  ( Stack *s );

#endif
```

```
#ifndef _STACK_H
#define _STACK_H

const unsigned int STK_MAX   = 50;
const unsigned int STK_EMPTY = 0;

struct Stack
{
   int    top;
   float *elems;

   void  push  ( float val );
   float pop   ( );
   bool  empty ( );
   void  show  ( );

};

Stack *createStack  ( int n );
void   destroyStack ( Stack *s );

#endif
```

Funções Membro de Estruturas

- Para compreender-se melhor esse conceito, considera-se a implementação de um novo método (função membro) à *Stack*;
- Para se criar uma nova *variável Stack*, ou melhor, *instanciar um novo objeto da classe Stack*, é necessário atribuir logo em seguida zero ao seu campo tops, para dar consistência a esse objeto;
- Para isso o ideal é criar mais um método para fazer essa atribuição:

Funções Membro de Estruturas

```
#ifndef _STACK_H
#define _STACK_H

const unsigned int STK_MAX   = 50;
const unsigned int STK_EMPTY = 0;

struct Stack
{
   int    top;
   float *elems;

   void  reset( ) { top = STK_EMPTY; }

   // ...
};
```

```
Stack *createStack( int n )
{
  Stack *stk = new Stack;
  stk->top = 0;
  ...[n];
Stack *createStack( int n )
{
  Stack *stk = new Stack;
  stk->reset( );
  stk->elems = new float[n];

  return stk;
}
```

- Ao ser executada a chamada ***stk->reset()***, o objeto ***stk*** passa a ter o seu atributo ***top*** com o valor ***STK_EMPTY***;

Funções Membro de Estruturas

- Analisa-se agora então a função push também como método de Stack, só que dessa vez implementa-se o método fora da estrutura:

```
struct Stack
{
   int    top;
   float *elems;
};
// ...
void  push  ( Stack *s, float val );
// ...
```

```
void push( Stack *s, float val )
{
  s->elems[s->top++] = val;
}
```

```
Stack s;
Stack *ps;
// ...
push( &s,14 );
push( ps,14 );
// ...
```

```
struct Stack
{
   int    top;
   float *elems;

   // ...
   void  push  ( float val );
   // ...
};
```

```
void Stack :: push( float val )
{
  elems[top++] = val;
}
```

```
Stack s;
Stack *ps;
// ...
s.push( 14 );
ps->push( 14 );
// ...
```

Funções Membro de Estruturas

- Para o objeto ***Stack s***, a chamada do método ***push*** incrementa ***s.top*** e atribui a ***s.elems[top]*** o valor ***14***;
- Para o ponteiro para objeto ***Stack *sp***, ocorre a mesma coisa, mas para o objeto para o qual ***sp*** está apontando;
- Para o caso de *sp* estar apontando para *s* (*sp = &s;*), nesse caso, os atributos alterados serão os do objeto *s*;
- Funções membro podem ser definidas dentro e fora da estrutura, conforme observa-se abaixo:

Operador de Escopo

- Funções membro, ou métodos, definidos dentro da estrutura são otimizadas pelo compilador e tratadas implicitamente como métodos ***inline***:

```
struct Stack
{
   // ...
   void  push  ( float val ) { elems[top++] = val; }
};
```

- Para definir funções membro fora da estrutura, é necessário utilizar-se o **operador de escopo**:

```
struct Stack
{
   // ...
   void  push  ( float val );
};

void Stack::push( float val )
{
  elems[top++] = val;
}
```

operador de escopo

Operador de Escopo

- o operador de escopo permite o acesso a nomes declarados em escopos que não sejam o corrente:

```
char *a;       // escopo global

void main(void)
{
   int a;      // escopo local (main)
   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:

```
char *a;

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

Sintaxe:

escopo :: nome

Facilidades C++ Para Funções

- C++ provê algumas facilidades de implementação para funções em geral, ou seja, métodos ou funções de escopo global:
  - Funções *inline*;
  - Parâmetros default;
  - Sobrecarga;

Funções *inline*

- Tornam a execução do código mais eficiente computacionalmente em relação às funções convencionais;
- 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.

Valores Default

- 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

- 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 Funções

- Esse recurso permite que um nome de função possa ter mais de um significado, ou seja, 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 faz sobrecarga, porém de forma implícita, ex: operadores aritméticos

Sobrecarga 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 Funções

- O uso misturado de sobrecarga e valores default para parâmetros também 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)???
}
```

Sobrecarga de métodos

- É possível também sobrecarregar métodos em uma estrutura:

```
struct stack {
      // …
      char pop( );
      char pop( int n );          // sobrecarga à pop( );
};
char stack::pop( int n )          // retira n elementos da pilha.
{
      while( (n-- > 1) && (top == 0) ) top--;
      return  s[top--];
}
```

Acesso público e privado

- O conceito de estruturas em C++ também permite a definição das formas de acesso a seus dados.
- IDÉIA: esconder o máximo de informação possível.
- Para tal são impostas certas regras para acesso a atributos e métodos de uma classe.
- São estabelecidos três níveis de permissão de acordo com o contexto de uso dos atributos:
  - Privado: acesso restrito aos métodos da classe.
  - Público: acesso a partir de objetos da classe.
  - Protegido: acesso liberado a métodos de classes derivadas
- Cada um tem uma palavra reservada associada, ***private***, ***public*** e ***protected*** respectivamente.

Acesso público e privado

```
struct controle {
 private:
   int a;
   int f1( char* b );
 protected:
   int b;
   int f2( float );
 public:
   int c;
   float d;
   void f3( controle* );
};
```

- As seções podem ser declaradas em qualquer ordem, inclusive podem aparecer mais de uma vez.

Membros de classes ***private***

- Somente a própria classe pode acessar os atributos privados, ou seja, somente os métodos da própria classe têm acesso a esses atributos.

```
struct SemUso {
 private:
   int valor;
   void f1() { valor = 0; }
   int  f2() { return valor; }
};

int main()
{
   SemUso p;              // cria um objeto do tipo SemUso
   p.f1();                // erro, f1 é private!
   cout << p.f2();        // erro, f2 é private!

   return 0;
}
```

Membros de classes ***public***

- É possível fazer chamadas e manipular respectivamente métodos e atributos da classe.
- Para poder chamar os métodos *f1* e *f2* no exemplo anterior é necessário defini-los como ***public***:

```
struct SemUso {
 private:
   int valor;
 public:
   void f1() { valor = 0; }
   int  f2() { return valor; }
};

    // …

     p.f1();                  // ok, f1 é public!
     printf("%d", p.f2() );   // ok, f2 é public!
```

Conceito de classes em C++

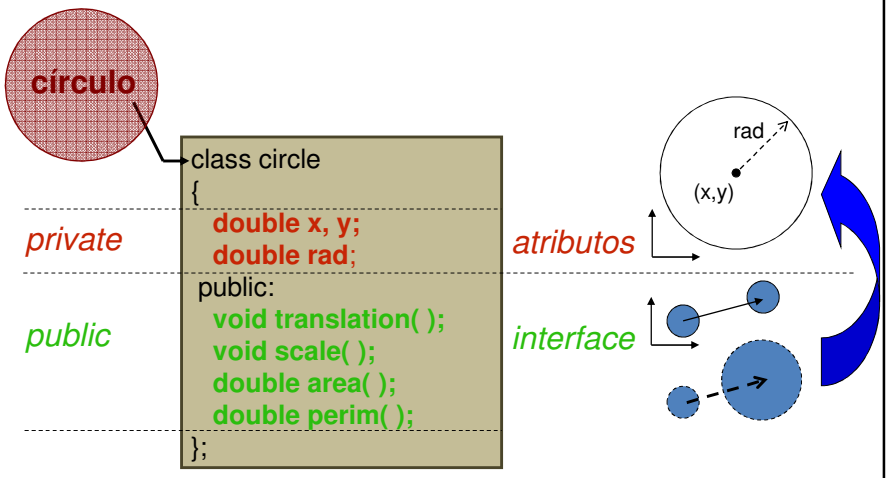
- O conceito de classes em C++ pode ser visto como uma extensão à idéia de estruturas encontrada em C puro.
- É uma forma de se implementar um tipo e associar a ele funções e operadores (TAD).
- Uma classe em C++ é o elemento básico sobre o qual toda a orientação a objetos está apoiada.
- Classes,ou TADs não são definidos pela sua representação interna, mas sim pelas operações sobre esse tipo (interface).
- Estas operações só fazem sentido quando associadas às suas representações.

struct e *class* em C++

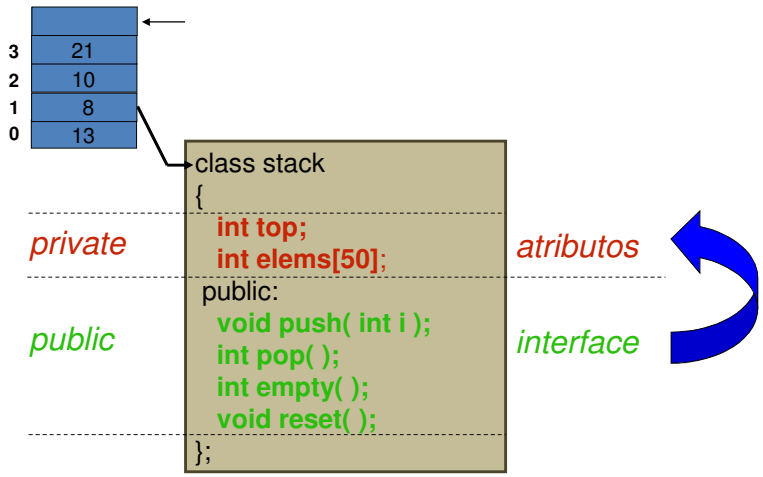
- Ambas podem representar, um **TAD** em C++;
- A diferença está no nível de proteção, caso nenhum dos especificadores de acesso seja usado:
  - ***class:*** por *default* é ***private***
  - ***struct:*** por *default* é ***public***

```
struct A {
   int a;        // a é público
};

class B {
   int a;        // a é privado
};
```

Classes em C++



Classes em C++



Classes em C++

```
#ifndef _STACK_H
#define _STACK_H

const unsigned int STK_MAX   = 50;
const unsigned int STK_EMPTY = 0;

class Stack
{
  private:
    int    top;
    float  elems[STK_MAX];

  public:
    void  push  ( float val );
    float pop   ( );

    bool  empty ( );
    void  show  ( );
};

Stack *createStack  ( int n );
void   destroyStack ( Stack *s );
```

```
void Stack :: push( float val )
{
  elems[top++] = val;
}
```

```
Stack *createStack( int n )
{
  Stack *stk = new Stack;
  stk->top = STK_EMPTY;

  return stk;
}
```

Classes em C++

```
#ifndef _STACK_H
#define _STACK_H

const unsigned int STK_MAX   = 50;
const unsigned int STK_EMPTY = 0;

class Stack
{
  private:
    int    top;
    float  elems[STK_MAX];

  public:
    void  reset ( );

    void  push  ( float val );
    float pop   ( );

    bool  empty ( );
    void  show  ( );
};

Stack *createStack  ( int n );
void   destroyStack ( Stack *s );

#endif
```

```
inline void Stack :: reset ( )
{
  top = STK_EMPTY;
}
```

```
Stack *createStack( int n )
{
  Stack *stk = new Stack;
  stk->reset( );

  return stk;
}
```

Funções e classes *friend*

- Algumas vezes duas classes são tão próximas conceitualmente que seria desejável que uma delas tivesse acesso irrestrito aos membros da outra.

```
class point
{
  friend class circle;
  private:  double x, y;
  public:   // …
};
class circle
{
    double rad;
    point  cnt;
  public:
    // …
    void print( )
    {
      cout << "x = " << cnt.x << endl;       // x é private.
      cout << "y = " << cnt.y << endl;       // y é private.
    }
};
```

Funções e classes *friend*

- Em C++ funções também podem ser ***friend*** de uma classe:

```
class Stack {
  friend Stack* createStack(void);
  int top;
  int elems[MAX];
 public:
  void push(int i) { elems[++top]=i; }
  int  pop(void)   { return elems[top--]; }
  int  empty(void) { return top == EMPTY; }
};
// …
Stack* createStack(void)
{
   Stack* s = new Stack;
   s->top = STK_EMPTY;        // top é private
   return s;
}
```

Funções e classes *friend*

- A idéia de métodos e classes *friend*, no entanto, quebra um dos conceitos básicos de POO, o encapsulamento.
- No exemplo da classe ***stack*** seria mais aconselhável utilizar-se do método *reset* ao invéz do uso do recurso de funções ***friend***.

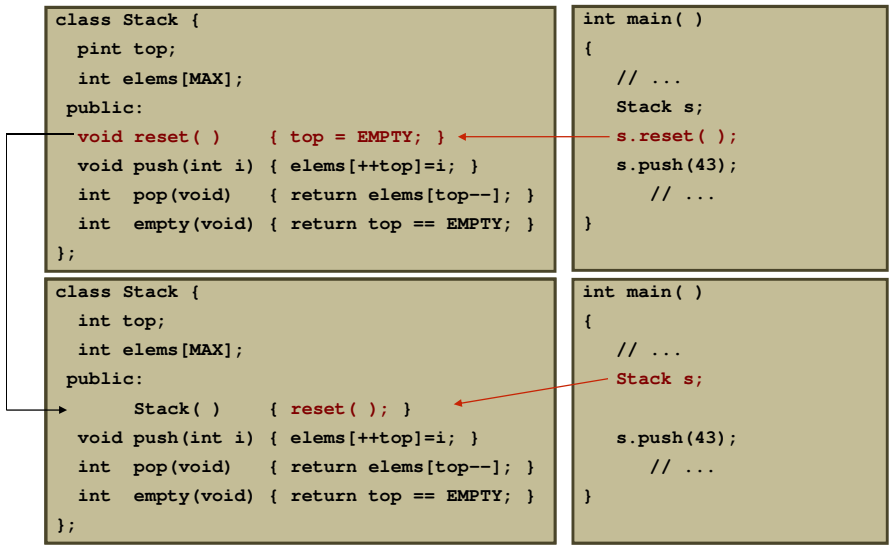
```
class Stack {
  int top;
  int elems[MAX];
 public:
  void push(int i) { elems[++top]=i; }
  int  pop(void)   { return elems[top--]; }
  void reset( )    { top = EMPTY; }
  int  empty(void) { return top == EMPTY; }
};
```

```
Stack* createStack(void)
{
   Stack* s = new Stack;
   s->reset( );
   return s;
};
```

Construtores e destrutores

- **CONSTRUTOR**: É uma função membro, ou método, usado para construir um objeto de uma dada classe e é chamado automaticamente quando um objeto é instanciado.
- **DESTRUTOR**: Analogamente, é um método chamado automaticamente quando um objeto é destruído.
- Os construtores são importantes para manter a consistência interna do objeto. O construtor deve garantir que o objeto recém criado seja consistente.
- No exemplo da classe ***Stack*** era necessário antes de *usar* o objeto, ou seja, carregar a pilha, inicializá-lo através do método ***reset()***.
- Mas algum cliente incauto da classe ***Stack*** poderia inserir e eliminar elementos na pilha sem antes inicializá-la o que provocaria invasão de memória.
- O uso de construtores elimina essa possibilidade.

Construtores: classe ***Stack***



Construtores: Declaração

- Os construtores e destrutores são métodos especiais. Ambos não apresentam um tipo de retorno.
- O destrutor não pode receber parâmetros, ao contrário dos construtores, que podem receber parâmetros.
- A declaração de um construtor é feita definindo-se um método com o mesmo nome da classe.
- O nome do destrutor é o nome da classe precedido de ~ (til).
- Um construtor de uma classe ***X*** é declarado como um método de nome ***X***, o nome do destrutor é ***~X*** .
- Pelo mecanismo de sobrecarga, classes podem ter mais de um construtor, ao contrário dos destrutores.

Chamada de construtores e destrutores

- Os construtores são chamados automaticamente sempre que um objeto da classe for criado:
  - Quando a variável é declarada (alocado na pilha)
  - Quando o objeto é alocado com ***new*** (alocado do *heap*)
- Os destrutores de objetos alocados na pilha são chamados quando o objeto sai do seu escopo. Os destrutores de objetos criados com ***new*** são chamados com o operador ***delete***.

```
class A {
  public:
    A( ) { cout << "construtor" << endl; }
   ~A( ) { cout << "destrutor"  << endl; }
 };
```

```
void main()
{
   A a1;              // chama construtor de a1
   A *a3;
   {
      A  a2;          // chama construtor de a2
      a3 = new A;     // chama construtor de a3
   }                  // chama destrutor  de a2
   delete a3;         // chama destrutor  de a3
}                     // chama destrutor  de a1
```

Construtores: classe ***Stack***

```
class Stack {
  int top;
  int elems[MAX];
 public:
       Stack( )     { top = EMPTY; }
  void push(int i) { elems[++top]=i; }
  int  pop(void)   { return elems[top--]; }
  int  empty(void) { return top == EMPTY; }
};
```

```
int main( )
{
   Stack s;
   s.push(43);
   // ...
}
```

```
int main( )
{
   Stack* s;
   s = new Stack;
   s->push(43);
   // ...
}
```

Construtores com Parâmetros

- No exemplo da classe stack o cliente da classe poderia querer definir um objeto pilha com uma dimensão maior do que 50:

```
class Stack {
   int   top;
   int   size;
   float *s;
  public:
   Stack( ) { top = STK_EMPTY; size = STK_MAX; s = new float[size]; }
   Stack( int n )  // sobrecarga do construtor
   {
     top = STK_EMPTY; size = n;
     s = new float[size];
   }
   ~Stack( ) { delete [] s; }
};
```

```
int main( )
{
  Stack s1;
  Stack s2(2000);
  Stack *s3 = new Stack;
  Stack *s4 = new Stack(2000);
}
```

Construtores: Classe *Stack*

```
class Stack
{
  private:
    int    top;
    float  elems[STK_MAX];

  public:
    void  reset ( );

    void  push  ( float val );
    float pop   ( );

    bool  empty ( );
    void  show  ( );
};

Stack *createStack  ( int n );
void   destroyStack ( Stack *s );
```

```
Stack *createStack( int n )
{
  Stack *stk = new Stack;
  stk->reset( );

  return stk;
}

void destroyStack( Stack *s )
{
  delete s;
}
```

```
class Stack
{
  public:
    Stack( int s );
   ~Stack( );

  private:
    int    top;
    float  *elems;

  public:
    void  push  ( float val );
    float pop   ( );
    void  reset ( );

    bool  empty ( ) const;
};
```

```
Stack :: Stack( int n )
{
  elems = new float[n];
  reset( );
}

Stack :: ~Stack( )
{
  delete [] elems;
}
```

Construtores Automaticos

- O fato de algumas classes não declararem construtores não significa que elas não tenham construtores. Na realidade o compilador gera dois construtores automaticamente, um ***default*** e um de **cópia**.
- Considere uma classe ***X***:

```
class X
{
  int a;
};
```

```
class X
{
  int a;
  // public: X() {}
  // public: X(const X&);
};
```

construtor vazio

construtor cópia

```
void main()
{
   X a1;       // usa construtor vazio
   X a2(a1);   // usa construtor de cópia
   X a3 = a2;  // usa construtor de cópia
}
```

Construtor de Cópia

- Dependendo dos tipos dos atributos de uma classe, é necessário implementar explicitamente um construtor de cópia. Exemplo:

```
class Stack {
  int   top;
  int   size;
  float *s;
 public:
  Stack( );
  Stack( int n );
  // ...
};
```

```
int main( )
{
  stack s1(50);
  // ...
  stack s2 = s1;
}
```

Requer construtor de cópia !!

```
Stack::Stack( const Stack& t )
{
  size = t.size;
  top = t.top;
  s = new char[size];
  for(int i=0; i<top; ++i) s[i] = t.s[i];
}
```

Métodos *const*

- Métodos ***const*** são métodos que não alteram o estado interno de um objeto. A especificação *const* faz parte da declaração do método:

```
class A {
   public: int value;
   public: int get ( )       { return value; }
   public: void put(int v)   { value = v; }
};

void f(const A a)
{
  // parâmetro a não pode ser modificado.
  // quais métodos de a podem ser chamados???
  int b = a.get();    // Erro! método não é const
}
```

- Para eliminar o erro seria necessário declarar o método *get* como sendo ***const***:

```
class A {
   // ...

   public: int get ( ) const  { return value; }
};
```

Parâmetro Implícito *This*

- Em todo método não ***static*** a palavra reservada ***this*** é um ponteiro para o objeto sobre o qual o método está sendo executado.
- Todos os métodos de uma classe são sempre chamados associados a um objeto.
- Durante a execução de um método, os campos do objeto são manipulados normalmente, sem necessidade de referência ao objeto.
- A palavra ***this*** é utilizada quando se necessita manipular o próprio objeto sobre o qual o método foi chamado.

This: Exemplo

```
class A {
   public: int i;
   public: A& inc( );
};

// Este método incrementa o valor interno
// e retorna o próprio objeto
A& A::inc()
{
   // estamos executando este código para obj1 ou obj2?
   // como retornar o próprio objeto?
   i++;
   return *this;  // this aponta para o próprio objeto
}

void main()
{
   A obj1, obj2;
   obj1.i = 0;
   obj2.i = 100;
   for (j=0; j<10; j++)
      cout << obj1.inc().i << endl;
}
```

This: Classe *Stack*

```
Stack :: Stack( int n )
{
  this->elems = new float[n];
  this->reset( );
}
```

```
void Stack :: reset ( )
{
  this->top = STK_EMPTY;
}
```

```
void Stack :: push( float val )
{
  this->elems[this->top++] = val;
}
```

this

```
// ...

Stack *s = new Stack(100);
Stack  t;

s->reset();

s->push(3.14159);
t->push(1.00);
```

Campos de classe *static*

- Quando uma variável é declarada ***static*** dentro de uma classe, todas as instâncias de objetos desta classe compartilham a mesma variável.
- Uma variável ***static*** é uma variável global, só que com escopo limitado à classe.
- A declaração de um campo ***static*** aparece na declaração da classe, junto com todos os outros campos.
- Como a declaração de uma classe é normalmente incluída em vários módulos de uma mesma aplicação via *header files* (arquivos .h), a declaração dentro da classe é equivalente a uma declaração de uma variável global ***extern***.
- A declaração apenas diz que a variável existe; algum módulo precisa defini-la.

Campos *static*: Exemplo

```
class A {
   public:
     int a;
     static int b; // declara a variável,
                   // equivalente ao uso de extern para
                   // variáveis globais
};

int A::b = 0;  // define a variável criando o seu espaço
               // esta é a hora de inicializar

void main(void)
{
   A a1, a2;
   a1.a = 0;  // modifica o campo a de a1
   a1.b = 1;  // modifica o campo b compartilhado por a1 e a2
   a2.a = 2;  // modifica o campo a de a1
   a2.b = 3;  // modifica o campo b compartilhado por a1 e a2
   cout << a1.a << a1.b << a2.a << a2.b;
   // imprime 0 3 2 3
}
```

- Como uma variável estática é única para todos os objetos da classe, não é necessário um objeto para referenciar este campo:

```
A::b = 4;
```

Métodos *static*

- Métodos ***static*** são como funções globais com escopo reduzido à classe.
- Isto significa que métodos ***static*** não apresentam o parâmetro ***this***, ou seja, apenas os campos *static* podem ser acessados.

```
class A {
   public:  int a;
   public:  static int b;
   public:  static void f();
};

int A::b = 10;

void A::f()
{
   a = 10; // errado, a só faz sentido com um objeto
   b = 10; // ok, b declarado static
}
```

Calculadora Versão III: *TAD Stack*

```
#ifndef _STACK_H
#define _STACK_H

const unsigned int STK_MAX   = 50;
const unsigned int STK_EMPTY = 0;

class Stack
{
  friend class StackIterator;
  public:
    Stack( );
    Stack( int s );
   ~Stack( );

  private:
    int    top;
    float  *elems;

  public:
    void  push  ( float val );
    float pop   ( );
    void  reset ( );

    bool  empty ( ) const;
};


class StackIterator {
 private:
   int current;
   Stack* st;
 public:
   StackIterator(Stack* s) { st = s; current = 0;         }
   bool end()              { return st->top == current;   }
   float next()            { return st->elems[current++]; }
};

#endif
```

```
#include <iostream>

using namespace std;

#include "stack.h"

Stack :: Stack( )
{
  elems = new float[STK_MAX];
  reset( );
}

Stack :: Stack( int n )
{
  elems = new float[n];
  reset( );
}

Stack :: ~Stack( )
{
  delete [] elems;
}

void Stack :: reset ( )
{
  top = STK_EMPTY;
}

void Stack :: push( float val )
{
  elems[top++] = val;
}

float Stack :: pop( )
{
  return elems[--top];
}

bool Stack :: empty ( ) const
{
  return top == 0;
}
```

Calculadora Versão III: ***Cliente***

```
int main( )
{
  Stack *s = new Stack( 100 );

  cout.precision( 2 );

  while (1)
  {
    string str;

    cout << "> ";
    cin  >> str;

    float val;
    if( StrToFloat( str, &val ) )
    {
      s->push( val );
    }
    else
    {
      char c;
      float n1, n2;
      StrToChar( str, &c );

      switch(c)
      {
        case '+':
          if( getop( s, &n1, &n2 ) ) s->push(n1+n2); break;
        case '-':
          if( getop( s, &n1, &n2 ) ) s->push(n1-n2); break;
        case '/':
          if( getop( s, &n1, &n2 ) ) s->push(n1/n2); break;
        case '*':
          if( getop( s, &n1, &n2 ) ) s->push(n1*n2); break;
        case 'q':
          delete s;
          return 1;
        default:
          cout << "Invalid Parameter !" << endl;
      }
    }

    StackIterator si(s);

    int i=0;
    while( !si.end( ) )
    {
      cout << i++ << ": ";
      cout << fixed << si.next() << endl;
    }
  }

  return 1;
}
```

```
#include "stack.h"

#include <iostream>
#include <sstream>
#include <string>

using namespace std;

bool getop( Stack *s, float *n1, float *n2 );

bool StrToFloat ( const string &str, float *v );
bool StrToChar  ( const string &str, char *c  );
```

```
bool getop( Stack *s, float *n1, float *n2 )
{
  if( s->empty( ) )
  {
    cout << "Empty Stack !" << endl;
    return false;
  }

  *n2 = s->pop( );
  if( s->empty( ) )
  {
    s->push( *n2 );
    cout << "Two operands needed !" << endl;

    return false;
  }
  *n1 = s->pop( );

  return true;
}

bool StrToFloat( const string &str, float *v )
{
  string *s = new string(str);

  istringstream buffer(*s, istringstream::in);

  return buffer >> *v;

}

bool StrToChar( const string &str, char *c )
{
  string *s = new string(str);

  istringstream buffer(*s, istringstream::in);

  return buffer >> *c;
}
```

Exemplo Polígono

```
#ifndef _POINT
#define _POINT

class ponto
{
  private:
    float _x;
    float _y;
  public:
    void   attrXY    ( float x, float y );

    float  x( ) const { return _x; }
    float  y( ) const { return _y; }
    float  distancia (  const ponto* q ) const;
};
```

```
#include <math.h>

#include "point.h"

void ponto :: attrXY( float x, float y )
{
  this->_x = x;
  this->_y = y;
}

float ponto :: distancia( const ponto* q ) const
{
  float x2 = (q->_x - this->_x)*(q->_x - this->_x);
  float y2 = (q->_y - this->_y)*(q->_y - this->_y);

  return sqrt( x2+y2 );
}
```

Exemplo Polígono

```
#ifndef _POLIGON
#define _POLIGON

#include "point.h"

class poligono
{
  private:
    int     n;
    ponto *vp;
  public:
    poligono  ( int num, ponto *vp );
   ~poligono  ( );

    void  imprime  ( );
    float perimetro( );
};

#endif
```

```
#include <iostream>
#include <math.h>

#include "poligon.h"

using namespace std;

poligono :: poligono ( int num, ponto *v )
{
  n  = num;
  this->vp = new ponto[n];

  for( int i=0; i<n; i++ )
    this->vp[i] = v[i];
}

poligono :: ~poligono( )
{
  delete [] vp;
}

void poligono :: imprime( )
{
  cout << "Número de Vértices do Polígono: " << n << endl;

  for( int i=0; i< n; i++ )
  {
    cout << "ponto " << i+1
         << ": (" << vp[i].x() << "," << vp[i].y() << ")"
         << endl;
  }
}

float poligono :: perimetro( )
{
  float per = 0.0;

  for(int i=0; i<n; i++)
  {
    int it = (i+1)-((i+1)/n)*n;

    per += vp[it].distancia( &(vp[i]) );
  }
}
```

Exemplo: Polígono

```
#include <iostream>
#include <math.h>

using namespace std;

#include "point.h"
#include "poligon.h"

int main( )
{
  cout << "Entre com o numero de pontos: ";
  int n; cin  >> n;

  ponto *vp = new ponto[n];

  for( int i=0; i<n; i++ )
  {
    cout << "Entre com o ponto " << i+1 << "(x,y): ";
    float x, y; cin >> x >> y;

    vp[i].attrXY( x, y );
  }

  poligono *pol = new poligono( n, vp );

  delete [] vp;

  pol->imprime( );

  cout << "Perimetro = " << pol->perimetro( ) << endl;

  delete pol;

  return 1;
}
```

```
#ifndef _POINT
#define _POINT

class ponto
{
  private:
    float _x;
    float _y;
  public:
    void   attrXY    ( float x, float y );

    float  x( ) const { return _x; }
    float  y( ) const { return _y; }
    float  distancia (  const ponto* q ) const;
};
```

```
#ifndef _POLIGON
#define _POLIGON

#include "point.h"

class poligono
{
  private:
    int     n;
    ponto *vp;
  public:
    poligono  ( int num, ponto *vp );
   ~poligono  ( );

    void  imprime  ( );
    float perimetro( );
};

#endif
```