Linguagens de Programação
Conceitos e Técnicas

Modularização

Prof.

Programação em Bloco Monolítico

- Inviabiliza grandes sistemas de programação
  - Um único programador pois não há divisão do programa
  - Indução a erros por causa da visibilidade de variáveis e fluxo de controle irrestrito
  - Dificulta a reutilização de código
- Eficiência de programação passa a ser gargalo

Processo de Resolução de Problemas Complexos

- Uso de Dividir para Conquistar
  - Resolução de vários problemas menos complexos
  - Aumenta as possibilidades de reutilização
- Técnicas de Modularização objetivam Dividir para Conquistar
  - Tornam mais fácil o entendimento do programa
  - Segmentam o programa
  - Encapsulam os dados – agrupam dados e processos logicamente relacionados

Sistemas de Grande Porte

- Características
  - Grande número de entidades de computação e linhas de código
  - Equipe de programadores
  - Código distribuído em vários arquivos fonte
  - Conveniente não recompilar partes não alteradas do programa

Sistemas de Grande Porte

- Módulo
  - Unidade que pode ser compilada separadamente
  - Propósito único
  - Interface apropriada com outros módulos
  - Reutilizáveis e Modificáveis
  - Pode conter um ou mais tipos, variáveis, constantes, funções, procedimentos
  - Deve identificar claramente seu objetivo e como o atinge

Abstração

- Fundamental para a Modularização
- Seleção do que deve ser representado
- Possibilita o trabalho em níveis de implementação e uso
- Uso Disseminado na Computação

Abstração

- Exemplos de uso na computação
  - Comandos do SO
  - Assemblers
  - LPs
  - Programa de Reservas
- Modos
  - LP é abstração sobre o hardware
  - LP oferece mecanismos para o programador criar suas abstrações
  - O segundo modo fundamenta a Modularização

Abstração e Modularização

- Foco na distinção entre
  - O que uma parte do programa faz
    - foco do programador que usa a abstração
  - Como isso é implementado
    - Foco do programador que implementa a abstração

Tipos de Abstrações

- Abstrações de Processos
  - Abstrações sobre o fluxo de controle do programa
  - Suprogramas – funções da biblioteca padrão de C (*printf*)
- Abstrações de Dados
  - Abstrações sobre as estruturas de dados do programa
  - Tipos de Dados – tipos da biblioteca padrão de C (*FILE*)

Abstrações de Processos

- Subprogramas
  - Permitem segmentar o programa em vários blocos logicamente relacionados
  - Servem para reusar trechos de código que operam sobre dados diferenciados
  - Modularizações efetuadas com base no tamanho do código possuem baixa qualidade
  - Propósito único e claro facilita legibilidade, depuração, manutenção e reutilização

Perspectivas do Usuário e do Implementador do Subprograma

- Usuário
  - Interessa o que o subprograma faz
  - Como usar é importante
  - Como faz é pouco importante ou não é importante
- Implementador
  - Importante é como o subprograma realiza a funcionalidade

Perspectivas do Usuário e do Implementador Sobre Função

```
int fatorial(int n) {
   if (n<2) {
      return 1;
   } else {
      return n * fatorial (n – 1);
   }
}
```

- Usuário
  - Função fatorial é mapeamento de n para n!
- Implementador
  - Uso de algoritmo recursivo

Perspectivas do Usuário e do Implementador Sobre Procedimento

```
void ordena (int numeros[50]) {
   int j, k, aux ;
   for (k = 0; k < 50; k++) {
     for (j = 0; j < 50; j++) {
        if (numeros[j] < numeros[j+1]) {
           aux = numeros[j];
           numeros[j] = numeros[j+1];
           numeros[j+1] = aux;
        }
     }
   }
}
```

- Usuário
  - Ordenação de vetor de inteiros
- Implementador
  - Método da bolha

Parâmetros

```
int altura, largura, comprimento;
int volume () { return altura * largura * comprimento; }
main() {
   int a1 = 1, l1 = 2, c1 = 3, a2 = 4, l2 = 5, c2 = 6;
   int v1, v2;
   altura = a1;
   largura = l1;
   comprimento = c1;
   v1 = volume();
   altura = a2;
   largura = l2;
   comprimento = c2;
   v2 = volume();
   printf ("v1: %d\nv2: %d\n", v1, v2);
}
```

Parâmetros

- Ausência reduz
  - Redigibilidade
    - Necessário incluir operações para atribuir os valores desejados às variáveis globais
  - Legibilidade
    - Na chamada de volume não existe qualquer menção à necessidade de uso dos valores das variáveis altura, largura e comprimento
  - Confiabilidade
    - Não exige que sejam atribuídos valores a todas as variáveis globais utilizadas em volume

Parâmetros

- Resolvem esses problemas

```
int volume (int altura, int largura, int comprimento) {
   return altura * largura * comprimento;
}
main() {
   int a1 = 1, l1 = 2, c1 = 3, a2 = 4, c2 = 5, l2 = 6;
   int v1, v2;
   v1 = volume(a1, l1, c1);
   v2 = volume(a2, l2, c2);
   printf ("v1: %d\nv2: %d\n", v1, v2);
}
```

Parâmetros Reais, Formais e Argumentos

- Parâmetro formal
  - Identificadores listados no cabeçalho do subprograma e usados no seu corpo
- Parâmetro real
  - Valores, identificadores ou expressões utilizados na chamada do subprograma
- Argumento
  - Valor passado do parâmetro real para o parâmetro formal

Parâmetros Reais, Formais e Argumentos

```
float area (float r) {
   return 3.1416 * r * r;
}
main() {
   float diametro, resultado;
   diametro = 2.8;
   resultado = area (diametro/2);
}
```

- Correspondência entre parâmetros reais e formais
  - Posicional
  - Por palavras chave

Correspondência Por Palavras Chave Entre Parâmetros Reais e Formais

```
procedure palavrasChave is
   a: integer := 2;
   b: integer := 3;
   c: integer := 5;
   res: integer;
   function multiplica(x, y, z: integer) return integer is
   begin
      return x * y * z;
   end multiplica;
begin
   res := multiplica(z=>b, x=>c, y=>a);
end palavrasChave;
```

Valores Default de Parâmetros

- Em C++

```
int soma (int a[], int inicio = 0, int fim = 7, int incr = 1){
   int soma = 0;
   for (int i = inicio; i < fim; i+=incr) soma+=a[i];
   return soma;
}
main() {
   int [] pontuacao = { 9, 4, 8, 9, 5, 6, 2};
   int ptotal, pQuaSab, pTerQui, pSegQuaSex;
   ptotal = soma(pontuacao);
   pQuaSab = soma(pontuacao, 3);
   pTerQui = soma(pontuacao, 2, 5);
   pSegQuaSex = soma(pontuacao, 1, 6, 2);
}
```

Valores Default de Parâmetros

- Em Python

```
 def dividir(num=0.0, den=1.0):
print num/den

dividir()
dividir(5,2)          # posicional
dividir(den=4,  num=16)    # palavra chave
dividir(5)

#apos valor padrao nao pode haver valor sem padrao
#apos palavra chave nao pode haver posicional
```

Lista de Parâmetros Variável

- Em C

```
#include <stdarg.h>
int ou (int n, ...) {
   va_list vl;
   int i;
   va_start (vl, n);
   for (i = 0; i < n; i++)
      if (va_arg (vl, int)) return 1;
   va_end (vl);
   return 0;
}
```

Lista de Parâmetros Variável

```
main() {
   printf ("%d\n", ou (1, 3 < 2));
   printf ("%d\n", ou (2, 3 > 2, 7 > 5));
   printf ("%d\n", ou (3, 1 != 1, 2 != 2, 3 != 3));
   printf ("%d\n", ou (3, 1 != 1, 2 != 2, 3 == 3));
}
```

- Oferece maior flexibilidade à LP
- Reduz a confiabilidade pois não é possível verificar os tipos dos parâmetros em tempo de compilação

Lista de Parâmetros Variável

- Em Python

```
def fexemplo(n, *opc, **opc_com_nome):
    print "Tupla de argumentos opcionais:", opc
    print "Dicionario de argumentos opcionais:", opc_com_nome

fexemplo(1,2,3,4,a=95,c=43,texto="Texto", d=(1,3))

Tupla de argumentos opcionais: (2, 3, 4)
Dicionario de argumentos opcionais: {'a': 95, 'c': 43, 'texto': 'T
  extos ', 'd': (1, 3)}

# apos palavra chave nao pode haver posicional
```

Lista de Parâmetros Variável

- Em Python

```
def ou (*conds):
    for cond in conds:
        if cond: return True
    return False
```

- Em Java

```
int ou (boolean ... cond) {
    for (i = 0; i < cond.length; i++)
        if (cond[i]) return 1;
    return 0;
}
```

Passagem de Parâmetros

- Processo no qual os parâmetros formais assumem seus respectivos valores durante a execução de um subprograma
- Faz parte do processo de passagem de parâmetros a eventual atualização de valores dos parâmetros reais durante a execução do subprograma
- Três Aspectos Importantes
  - Direção da passagem
  - Mecanismo de implementação
  - Momento no qual a passagem é realizada

Direção da Passagem

Direção da Passagem	Forma do Parâmetro Real (R)	Atrib. do Parâm. Formal (F)	Fluxo
Entrada Variável	Variável, Constante ou Expressão	Sim	R → F
Entrada Constante	Variável Constante ou Expressão	Não	R →F
Saída	Variável	Sim	R ← F
Entrada e Saída	Variável	Sim	R ↔F

Direção da Passagem

- C usa passagem unidirecional de entrada variável

```
void naoTroca (int x, int y) {
   int aux;
   aux = x;
   x = y;
   y = aux;
}
void troca (int* x, int* y) {
   int aux;
   aux = *x;
   *x = *y;
   *y = aux;
}
```

```
main() {
   int a = 10, b = 20;
   naoTroca (a, b);
   troca (&a, &b);
}
```

Direção da Passagem

- C++ também usa unidirecional de entrada constante e bidirecional

```
int triplica (const int x) {
   // x = 23;
   return 3*x;
}
void troca (int& x, int& y) {
   int aux;
   aux = x;
   x = y;
   y = aux;
}
```

```
main() {
   int a = 10, b = 20;
   b = triplica (a);
   troca (a, b);
   // troca (a, a + b);
}
```

Direção da Passagem

- JAVA usa
  - passagem unidirecional de entrada para tipos primitivos
  - unidirecional de entrada ou bidirecional para tipos não primitivos

```
void preencheVet (final int[] a, int i, final int j) {
        while (i <= j) a[i] = i++;
        // j = 15;
        // a = new int [j];
}
```

Direção da Passagem

- Python usa unidirecional de entrada e bidirecional

```
def f (x, y):
        print x
        print y
        x = 10
        y[0] = 17
        print x
        print y
        y = [1, 2, 3]
        print y
```

```
a = 20
b = [10, 20, 30]
f(a,b)
print a
print b
```

Direção da Passagem

- ADA usa unidirecional de entrada constante, unidirecional de saída e bidirecional

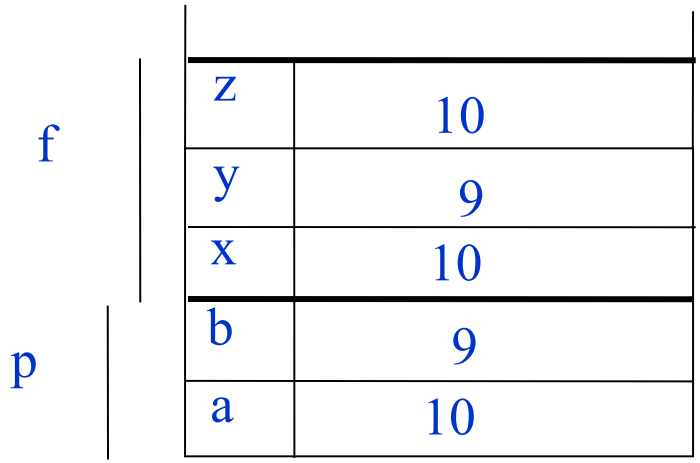
  function triplica (x: in integer; out erro: integer) return integer;

  procedure incrementa (x: in out integer; out erro: integer);

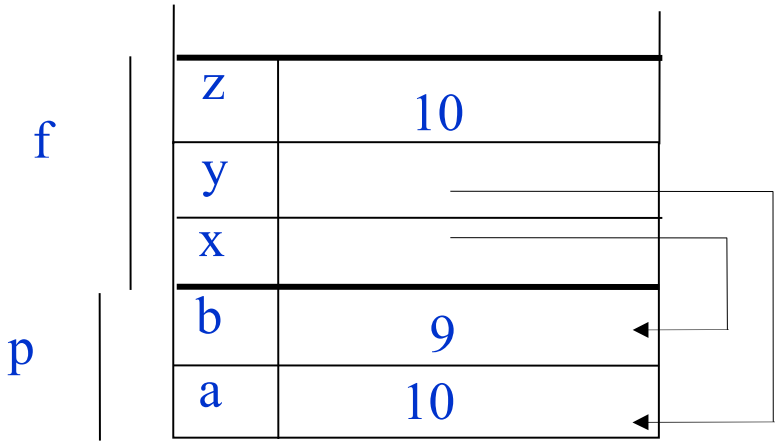
- Cuidado com colisões nas passagens unidirecional de saída e bidirecional

  incrementa (i, i);

Mecanismos de Passagem



Cópia



Referência

Mecanismos de Passagem

- Cópia
  - Viabiliza a passagem unidirecional de entrada variável
  - Facilita a recuperação do estado do programa em interrupções inesperadas
- Referência
  - Proporciona semântica simples e uniforme na passagem de todos os tipos
  - Mais eficiente por não envolver cópia de dados
  - Pode ser ineficiente em implementação distribuída
  - Permite a ocorrência de sinonímia

```
void incr (int& k, int& l) {
  k = k + 1;
  l = l + 1;
}
```

  - Dificil entender o que ocorre em `incr (a[i], a[j])` quando `i == j`

Mecanismos de Passagem

- C oferece apenas passagem por cópia
- C++ e PASCAL oferecem mecanismos de cópia e referência
- ADA usa cópia para primitivos e referência para alguns tipos
  - Outros tipos são definidos pelo compilador e podem ser cópia ou referência

Mecanismos de Passagem

- JAVA adota
  - passagem por cópia para tipos primitivos
  - há cópia de referência para tipos não primitivos
    - Alguns consideram passagem por cópia e outros por referência -> passagem por compartilhamento
    - Diferente da passagem por referência de C++

```
void f (T t1, T t2) {
      t1 = t2;
}
```

```
void f(T& t1, T& t2) {
      t1 = t2;
}
```

- Python adota cópia de referência

Mecanismos de Passagem

- Passagem unidirecional de entrada por cópia é conhecida como passagem por valor
- Passagem unidirecional de entrada constante por referência equivale a por valor
  - Tem como vantagem de não demandar cópias de grandes volumes de dados

Momento da Passagem

- Normal (`eager`)
  - Avaliação na chamada do subprograma
- Por nome (`by name`)
  - Avaliação quando parâmetro formal é usado
- Preguiçosa (`lazy`)
  - Avaliação quando parâmetro formal é usado pela primeira vez
- Maioria das LPs (tais como, C, PASCAL, JAVA e ADA) adota modo normal

Momento da Passagem

```
int caso (int x, int w, int y, int z) {
    if (x < 0) return w;
    if (x > 0) return y;
    return z;
}
caso(p(), q(), r(), s());
```

Momento da Passagem

- Avaliação normal
  - Avaliação desnecessária de funções em caso
  - Pode reduzir eficiência e flexibilidade
- Avaliação por nome
  - Somente uma de q, r ou s seria avaliada
  - Problemas
    - p poderia ser avaliada duas vezes
    - Se p produzisse efeitos colaterais (como em um iterador)
- Avaliação Preguiçosa
  - Única execução de p e somente uma de q, r ou s

Momento da Passagem

- Haskell usa preguiçosa

```
fib :: Int -> Int -> [Int]
fib 0 _ = []
fib m n = m : (fib n (m+n))

obtenha :: [Int] -> Int -> Int
obtenha [] _ = 0
obtenha (x:xs) 1 = x
obtenha (x:xs) n = obtenha xs (n-1)

obtenha (fib 1 1) 3
```

Momento da Passagem

- Haskell usa preguiçosa

1) obtenha (fib 1 1) 3 => obtenha (x:xs) n = obtenha xs (n-1)

(fib 1 1) => 1: (fib 1 (1 + 1))

x = 1 / xs = (fib 1 2) / n = 3 => obtenha (fib 1 2) 2

2) obtenha (fib 1 2) 2 => obtenha (x:xs) n = obtenha xs (n-1)

(fib 1 2) => 1: (fib 2 (1 + 2))

x = 1 / xs = (fib 2 3) / n = 2 => obtenha (fib 2 3) 1

3) obtenha (fib 2 3) 1 => obtenha (x:xs) 1 = x

(fib 2 3) => 2: (fib 2 (2 + 3))

x = 2 / xs = (fib 2 5) / n = 1 => 2

Verificação de Tipos

- Ausência de verificação estática de tipos dos parâmetros
  - Retarda a identificação de erros para a execução
  - Produz programas menos robustos
- Maioria das LPs ALGOL-like (PASCAL, ADA e JAVA) fazem verificação estática
- Python não faz verificação estática nem dinâmica

Verificação de Tipos

- Versão original de C não requer verificação estática de tipos
- ANSI C fornece duas maneiras de especificar os parâmetros de uma função
- É papel do programador ANSI C definir se deve ou não haver verificação estática

Verificação de Tipos em C

```
#include <math.h>
typedef struct coordenadas {
   int x, y, z;
} coord;
int origem (c)
    coord c;
{
    return c.x == 0 && c.y == 0 && c.z == 0;
}
float distancia (coord c) {
   return sqrt(c.x*c.x + c.y*c.y +c.z*c.z);
}
```

Verificação de Tipos em C

```
main() {
        coord c = { 1, 2, 3 };
        printf("%d\n", origem(2));
        printf("%d\n", origem(1, 2, 3, 4));
        printf("%d\n", origem(c));
        // printf("%f\n", distancia(2));
        // printf("%f\n", distancia (1, 2, 3));
        printf("%f\n", distancia (c));
}
```

- Existe diferença entre
  - f() – Aceita qualquer lista de parâmetros
  - f(void) – Lista de parâmetros vazia

Tipos de Dados

- Forma de modularização usada para implementar abstrações de dados
- Agrupam dados correlacionados em uma entidade computacional
- Usuários enxergam o grupo de dados como um todo
  - Não se precisa saber como entidade é implementada ou armazenada

Tipos de Dados

- Tipos Anônimos
  - Definidos exclusivamente durante a criação de variáveis e parâmetros

```
struct {
        int elem[100];
        int topo;
} pilhaNumeros;
```

Tipos de Dados

- Tipos Simples
  - Agrupam dados relacionados em uma única entidade nomeada
  - Aumentam reusabilidade, redigibilidade, legibilidade e confiabilidade

```
#define max 100
typedef struct pilha {
   int elem[max];
   int topo;
} tPilha;
tPilha global;
```

Tipos Simples

```
void preenche (tPilha *p, int n) {
   for (p->topo=0; p->topo < n && p->topo < max; p->topo++)
      p->elem[p->topo] = 0;
   p->topo--;
}
main( ) {
  tPilha a, b;
   preenche(&a, 17);
   preenche(&b, 29);
   preenche(&global, 23);
}
```

Tipos Simples

- Operações são definidas pela LP
- Não possibilitam ocultamento de informação
  - Isso prejudica legibilidade, confiabilidade e modificabilidade

```
main( ) {
   tPilha a;
   preenche(&a, 10);
   a.elem[++a.topo] = 11;
   a.topo= 321;
}
```

  - Mistura código da implementação com o de uso do tipo
  - Programador pode alterar errônea e inadvertidamente a estrutura interna do tipo
  - Alteração na estrutura interna implica em alteração do código fonte usuário

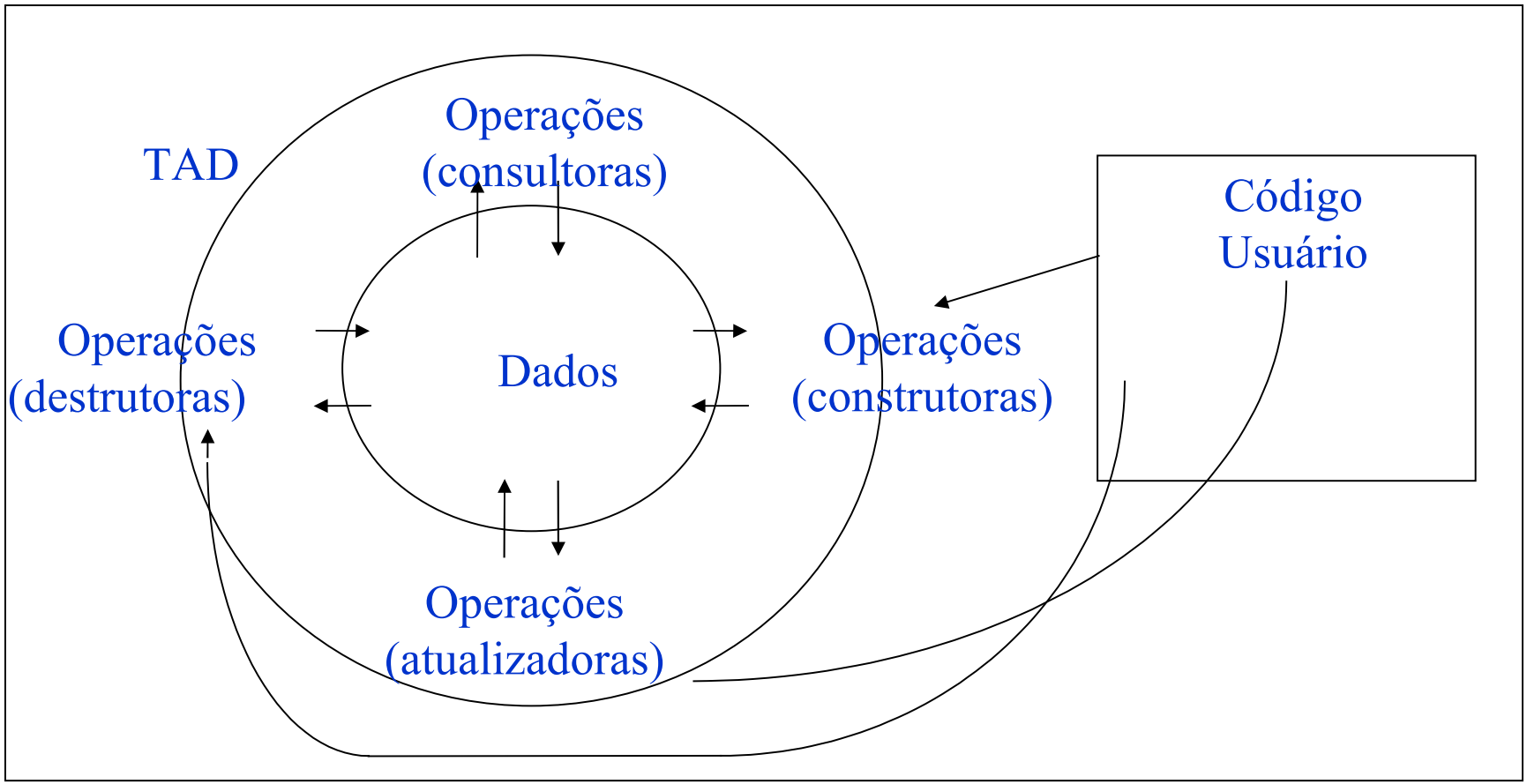
Tipos Abstratos de Dados (TADs)

- Conjuntos de valores com comportamento uniforme definido por operações
- Em LPs, TADs possuem representação e operações especificadas pelo programador
- O usuário do TAD utiliza sua representação e operações como uma caixa preta
- Essencial haver ocultamento da informação para tornar invisível a implementação
- Interface são os componentes públicos do TAD (tipicamente, operações)

Tipos Abstratos de Dados

- Encapsulam e protegem os dados
- Resolvem os problemas existentes com tipos simples
- Quatro tipos diferentes de operações
  - Construtoras
  - Consultoras
  - Atualizadoras
  - Destrutoras

Tipos Abstratos de Dados



Simulação de TADs em C

- Assinaturas das Operações

  cria: void → tPilha

  vazia: tPilha → int

  empilha: tPilha x int → tPilha

  desempilha: tPilha → tPilha

  obtemTopo: tPilha → int

Simulação de TADs em C

```
#define max 100
typedef struct pilha {
        int elem[max];
        int topo;
} tPilha;
tPilha cria () {
   tPilha p;
   p.topo = -1;
   return p;
}
int vazia (tPilha p) {
   return p.topo == -1;
}
```

Simulação de TADs em C

```
tPilha empilha (tPilha p, int el) {
   if (p.topo < max-1 && el >= 0)
       p.elem[++p.topo] = el;
   return p;
}
tPilha desempilha (tPilha p) {
   if (!vazia(p)) p.topo--;
   return p;
}
int obtemTopo (tPilha p) {
   if (!vazia(p)) return p.elem[p.topo];
   return -1;
}
```

Simulação de TADs em C

```
tPilha global;
void preenche (tPilha *p, int n) {
    int i;
    for (i = 0; i < n; i++) *p = empilha (*p, 0);
}
main( ) {
    tPilha a, b;
    global = cria();
    a = cria();
    b = cria();
    preenche(&a, 17);
    preenche(&b, 29);
    preenche(&global, 23);
    a = empilha(a, 11);
}
```

Simulação de TADs em C

```
    // a.elem[++a.topo] = 11;
    // a.topo= 321;
    // global = a;
}
```

- Uso Disciplinado da Simulação
  - Maior legibilidade
    - Compare preenche
  - Maior redigibilidade
    - Usuário não implementa operações
  - Maior modificabilidade
    - Alterações no TAD não provocam alterações no usuário

Simulação de TADs em C

- Não promove encapsulamento das operações e dados em uma única unidade sintática
- Uso Indisciplinado da Simulação
  - Programador pode não chamar a operação de inicialização
  - Programador pode usar outras operações para acessar o TAD
    - Isso acaba com todas as vantagens do uso de TADs

TADs em C

- Uso de módulos de interface e implementação
- Uso de tipo opaco
  - Interface contém definição de ponteiro para declaração do TAD e declarações de funções do TAD
  - Implementação contém definição do TAD e das funções

TADs em C - Interface

```
typedef struct pilha * tPilha;

tPilha cria ();
int vazia (tPilha p);
tPilha empilha (tPilha p, int el);
tPilha desempilha (tPilha p);
int obtemTopo (tPilha p);
```

TADs em C - Implementação

```
#define max 100
struct pilha {
        int elem[max];
        int topo;
};
tPilha cria () {
   tPilha p = (tPilha) malloc (sizeof(tPilha));
   p->topo = -1;
   return p;
}
int vazia (tPilha p) {
   return p->topo == -1;
}
```

TADs em C - Implementação

```
tPilha empilha (tPilha p, int el) {
   if (p->topo < max-1 && el >= 0)
       p->elem[++p->topo] = el;
   return p;
}
tPilha desempilha (tPilha p) {
   if (!vazia(p)) p->topo--;
   return p;
}
int obtemTopo (tPilha p) {
   if (!vazia(p)) return p->elem[p.topo];
   return -1;
}
```

Uso de TADs em C

```
tPilha global;
void preenche (tPilha p, int n) {
    int i;
    for (i = 0; i < n; i++) p = empilha (p, 0);
}
main( ) {
    tPilha a, b;
    global = cria();
    a = cria();
    b = cria();
    preenche(a, 17);
    preenche(b, 29);
    preenche(global, 23);
    a = empilha(a, 11);
}
```

Uso de Interface e Implementação nos TADs em ADA

```
package pilha_naturais is
   type tPilha is limited private;
   procedure cria (p: out tPilha);
   function vazia (p: in tPilha) return boolean;
   procedure empilha (p: in out tPilha; el: in integer);
   procedure desempilha (p: in out tPilha);
   function obtemTopo (p: in tPilha) return integer;
private
   max: constant integer := 100;
   type tPilha is record
      elem: array (1 .. max) of integer;
     topo: integer;                    -- topo: integer := 0;
   end record;
end pilha_naturais;
```

Uso de Interface e Implementação nos TADs em ADA

```
package body pilha_naturais is
   procedure cria (p: out tPilha) is
   begin
      p.topo := 0;
   end cria;
   function vazia (p: in tPilha) return boolean is
   begin
      return (p.topo = 0);
   end vazia;
   procedure empilha (p: in out tPilha; el: in integer) is
   begin
      if p.topo < max and then el >= 0 then
         p.topo := p.topo + 1;
         p.elem(p.topo) := el;
      end if;
```

Uso de Interface e Implementação nos TADs em ADA

```
  end empilha;
  procedure desempilha (p: in out tPilha) is
  begin
    if not vazia(p) then
      p.topo = p.topo – 1;
    end if;
  end desempilha;
  function obtemTopo (p: in tPilha) return integer is
  begin
    if not vazia(p) then
      return p.elem(p.topo);
    return –1;
  end obtemTopo;
end pilha_naturais;
```

Uso de Interface e Implementação nos TADs em ADA

```
use pilha_naturais;
procedure main is
   pilha: tPilha;
   numero: integer;
   cria (pilha);
   empilha (pilha, 1);
   empilha (pilha, 2);
   while not vazia(pilha) loop
      numero := obtemTopo(pilha);
      desempilha(pilha);
   end loop;
end main;
```

Uso de Interface e Implementação nos TADs em ADA

- Declaração de tPilha como limited garante não aplicação de atribuição e comparação
- Garante
  - Legibilidade
  - Redigibilidade
  - Confiabilidade
  - Modificabilidade
- Problema da falta de inicialização continua

TADs Como Classes em C++

- Operações construtoras e destrutoras especiais

```
class tPilha {
   static const int max = 100;
   int elem[max];
   int topo;
public:
   tPilha () {
       topo = -1;
   }
   int vazia () {
       return topo == -1;
   }
```

TADs Como Classes em C++

```
        void empilha (int el);
        void desempilha (void);
        int obtemTopo (void);
};
void tPilha::empilha (int el) {
   if (topo < max-1 && el >= 0)
       elem[++topo] = el;
}
void tPilha::desempilha (void) {
   if (!vazia()) topo--;
}
```

TADs Como Classes em C++

```
int  tPilha::obtemTopo (void) {
   if (!this->vazia()) return elem[topo];
   return -1;
}
main () {
   tPilha p;
   int n;
   p.empilha (1);
   p.empilha (2);
   while (! p.vazia ()) {
      n = p.obtemTopo ();
      p.desempilha ();
   }
}
```

TADs Como Classes em C++

- Estrutura de dados privada e operações da interface públicas
- Uso do operador de resolução de escopo
- Sintaxe diferenciada com parâmetro implícito na definição e chamada
- Podem haver vários construtores sempre chamados antes de qualquer outra operação

```
tPilha* i = new tPilha;
```

TADs Como Classes em C++

- Construtores default e por cópia e a operação de atribuição

```
main() {
   tPilha p1, p2;
   tPilha p3 = p1;      // construtor de copia
   p3 = p2;             // operador de atribuicao
}
```

- Construtor de cópia é importante em C++
  - Uso implícito na passagem por cópia e retorno de função
- Função destrutora única em C++ (~tPilha)

TADs Como Classes

- Membros de classe
  - Palavra reservada static precedendo método ou atributo
  - Atributo de classe
    - Compartilhado por todos objetos da classe
  - Método de classe
    - Pode ser chamado independentemente da existência de um objeto da classe
    - Pode ser chamado pela própria classe ou pelos objetos da classe

TADs Como Classes

- Em JAVA
  - Não ha' necessidade de definição de construtor de cópia e operador de atribuição
  - Não precisa de função destrutora
    - Em alguns casos pode ser necessária uma função de finalização chamada explicitamente após o fim do uso do objeto

TADs Como Classes

- Em Python
  - Classes podem ser usadas para simular TADs
  - Não existe ocultamento de informação
    - Name Mangling: nomes precedidos por __ aparentam ser privados porque são automaticamente prefixados com _NomeDaClasse

TADs Como Classes

- Em Python
  - Classes são abertas
    - Novos atributos de instância podem ser inseridos a qualquer momento, inclusive no programa cliente
    - Não há garantia de inicialização apropriada: método pode usar atributo ainda não criado
  - Método único de inicialização (init)
    - Trata internamente os diferentes contextos
    - Pode implicar na passagem de parâmetros desnecessários

Classes em Python

```
class Classe:
   clsvar = []
   def __init__(self, args):
     ...
   def metodo(self, params):
     ...
   @classmethod
   def cls_metodo(cls, params):
     ...
   @staticmethod
   def est_metodo(params):
     ...
```

Classes em Python

```
class Classe:
    i = 50                      # variavel de classe
    def __init__(self):
        Classe.i+=1
        Classe.l = 200          # variavel de classe
        i = 300                 # variavel local
        self.__j = 120          # variavel de instancia
        self.k = 150            # variavel de instancia
        print str(Classe.i) + " - " + str(self.__j)
    def incrJ(self):
        self.__x = 10           # variavel de instancia
        self.__j += 30
```

Classes em Python

```
o1 = Classe()      # 51 - 120
o2 = Classe()      # 52 - 120
print o1.i         # 52
o1.i += 1
print o2.i         # 52
print o1.i         # 53
del o1.i
print o1.i         # 52
o2.j = o1._Classe__j + 100
print o1._Classe__j       # 120
print o2.j                # 220
o1.incrJ()
print o1._Classe__x       # 10
```

Classes em Python

- É possível incluir métodos dinamicamente

```
class Usuario:
    def __init__(self, nome):
        self.nome = nome
def def_senha(self, senha):
    self.senha = senha
Usuario.def_senha = def_senha
usuario = Usuario('novo')
usuario.def_senha('xpto')
print 'Senha:', usuario.senha
```

Pacotes

- Só subprogramas e tipos não são suficientes para sistemas de grande porte
  - Código com baixa granularidade
  - Possibilidade de conflito entre fontes de código
- Fontes de código são coleções de entidades reutilizáveis de computação
  - Bibliotecas
  - Aplicações utilitárias
  - Frameworks
  - Aplicações completas

Pacotes

- Pacotes agrupam entidades de computação (exportáveis ou não)
- Usados para organizar as fontes de informação e para resolver os conflitos de nomes
- ADA possui package e package body
- Pacotes em C++
  - Conjunto de definições embutidas em uma entidade nomeada

Pacotes em C++

```
namespace umaBiblioteca {
   int x = 10;
   void f() {};
   class tC {};
}
```

- Nova definição adiciona mais entidades a umaBiblioteca

```
namespace umaBiblioteca {     // nao eh redefinicao
   int y = 15;
   void g() {};
   // int x = 13;
}
```

Pacotes em C++

- Pacotes podem usar nomes já definidos em outro

```
namespace outraBiblioteca {
   int x = 13;
   void h(){};
}
```

- Operador de resolução de escopo resolve conflitos

```
main() {
   umaBiblioteca::y = 20;
   umaBiblioteca::f();
   umaBiblioteca::x = 5;
   outraBiblioteca::x = 5;
   outraBiblioteca::h();
}
```

Pacotes em C++

- É possível renomear o pacote

```
namespace bib1 = umaBiblioteca;
namespace bib2 = outraBiblioteca;
main() {
   bib1::y = 20;
   bib1::x = 5;
   bib2::x = 5;
   bib2::h(){};
}
```

Pacotes em C++

- É possível evitar o uso repetitivo do operador de resolução de escopo

```
using namespace umaBiblioteca;
using namespace outraBiblioteca;
main() {
   y = 20;
   f();
   h(){};
   // x = 5;
   umaBiblioteca::x = 5;
   outraBiblioteca::x = 5;
}
```

- Biblioteca padrão de C++ usa namespace std

Pacotes em JAVA

- Conjunto de classes relacionadas

```
package umPacote;
public class umaClasse {}
class outraClasse {}
```

- Uso das classes do pacote

```
umPacote.umaClasse m = new umPacote.umaClasse();

import umPacote.*;
umaClasse m = new umaClasse();

import umPacote.umaClasse;
```

Pacotes em JAVA

- Relação de pacotes com a organização de arquivos em diretórios
- Novo especificador de acesso relacionado com pacotes

```
package umPacote;
public class umaClasse {
   int x;
   private int y;
   public int z;
}
```

Pacotes em JAVA

```
class outraClasse {
   void f() {
      umaClasse a = new umaClasse();
      // a.y = 10;
      a.z = 15;
      a.x = 20;
   }
}
```

- Evita o uso de operações públicas quando só construtor do pacote precisa ter acesso

Modularização, Arquivos e Compilação Separada

- Uso de arquivo único causa problemas
  - Redação e modificação se tornam mais difíceis
  - Reutilização apenas com processo de copiar e colar
- Divisão do código em arquivos separados com entidades relacionadas
  - Biblioteca de arquivos .c
  - Arquivos funcionam como índices
  - Reutilização através de inclusão
  - Necessidade de recompilação de todo o código

Modularização, Arquivos e Compilação Separada

- Compilação Separada de Arquivos
  - Ligação para gerar executável
  - Perda da verificação de tipos
    - Na chamada de funções
    - Nas operações sobre variáveis externas
  - Alternativa é extern de C
    - Não é suficientemente genérica pois não contempla tipos

Modularização, Arquivos e Compilação Separada

- Interface e implementação
  - Interface define o que é exportável (arquivo .h)
  - Implementação define o específico (arquivo .c)
  - Arquivos de implementação contêm a parte pesada da compilação
  - Novo meio de ocultamento de informação
  - Muito usado para construir TADs
    - Interface: declaração do tipo e das operações
    - Implementação: definição dos tipos e das operações, além de entidades auxiliares

Modularização, Arquivos e Compilação Separada

- TADs
  - ADA e C++ requerem a definição do tipo na interface
  - ADA e C++ possibilitam o ocultamento da estrutura interna, mesmo na interface
  - Necessidade de recompilar o código usuário após alteração da estrutura interna do TAD

Modularização, Arquivos e Compilação Separada

- Tipo Opaco em C
  - Ponteiros utilizados na interface para designar tipo na implementação
  - Protótipos das operações do TAD se referem apenas ao tipo opaco
    - Impede o uso inconsistente do TAD
    - Alteração somente quando protótipos das operações são modificados
      - Espaço para um ponteiro no código usuário
      - Uso exclusivo de operações do TAD no código usuário

Modularização, Arquivos e Compilação Separada

- Problema com Tipo Opaco em C
  - Força o uso de ponteiros e alocação dinâmica
    - Reduz redigibilidade e legibilidade
    - Perde eficiência de execução
  - Poderia ter implementação mais transparente
- JAVA não requer recompilação de código usuário quando interface permanece
  - Uso de referências para objetos
  - Coletor de Lixo
- JAVA usa arquivo único com interface e implementação

Vantagens da Modularização

- Melhoria da legibilidade
  - Divisão lógica do programa em unidades funcionais
  - Separação do código de implementação do código de uso da abstração
- Aprimoramento da redigibilidade
  - Mais fácil escrever código em vários módulos do que em um módulo único
- Aumento da modificabilidade
  - Alteração no módulo é localizada e não impacta código usuário

Vantagens da Modularização

- Incremento da reusabilidade
  - Módulo pode ser usado sempre que sua funcionalidade é requerida
- Aumento da produtividade de programação
  - Compilação separada
  - Divisão em equipes
- Maior confiabilidade
  - Verificação independente e extensiva dos módulos antes do uso

Vantagens da Modularização

- Suporte a técnicas de desenvolvimento de software
  - Orientadas a funcionalidades (top-down)
    - Uso de subprogramas
  - Orientadas a dados (bottom-up)
    - Uso de tipos
  - Complementaridade dessas técnicas