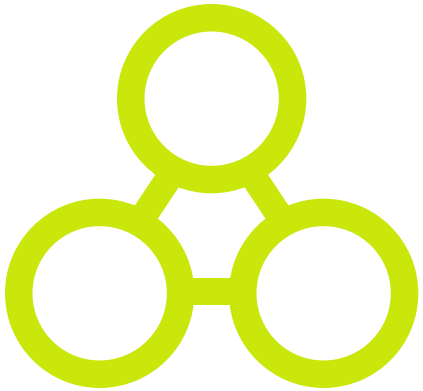


nemo

ontology & conceptual modeling research group

Linguagens de Programação
6 – Modularização

Vítor E. Silva Souza

(vitor.souza@ufes.br)

http://www.inf.ufes.br/~vitorsouza

Departamento de Informática

Centro Tecnológico

Universidade Federal do Espírito Santo

Conteúdo do curso

- Introdução;
- Amarrações;
- Valores e tipos de dados;
- Variáveis e constantes;
- Expressões e comandos;

- Modularização;
- Polimorfismo;
- Exceções;
- Concorrência;
- Avaliação de linguagens.

- Estes slides foram baseados em:
  - Slides do prof. Flávio M. Varejão;
  - Livro “Linguagens de Programação – Conceitos e Técnicas” (Varejão);
  - Livro “Linguagens de Programação – Princípios e Paradigmas, 2a edição” (Tucker & Noonan).

Programação em bloco monolítico

- Poucas variáveis representando diferentes coisas;
- Uso extensivo de desvio incondicional (goto);
- Inviabiliza grandes sistemas de programação:
  - Um único programador, não há divisão do programa;
  - Indução a erros por causa da visibilidade de variáveis e fluxo de controle irrestrito;
  - Reutilização de código muito difícil.
- Eficiência de programação passa a ser gargalo.

Resolução de problemas complexos

- 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

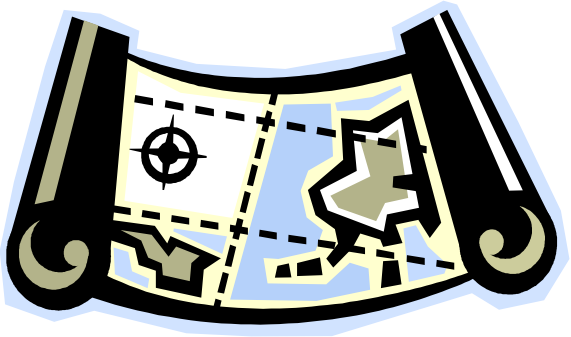
- Características:
  - Grande número de entidades de computação e linhas de código;
  - Equipe de programadores;
  - Código distribuído em vários arquivos fonte;
  - Conveniente não recompilar partes não alteradas do programa.

Sistemas de grande porte

- Módulo
  - Unidade que pode ser compilada separadamente;
  - Propósito único;
  - Interface apropriada com outros módulos;
  - Reutilizável e modificável;
  - Pode conter um ou mais tipos, variáveis, constantes, funções, procedimentos;
  - Deve identificar claramente seu objetivo e como o atinge;
  - Deve indicar que outros módulos ele afeta;
  - Ex.: `java.util.regex`, `javax.swing`, etc.

Abstração

- Fundamental para a modularização;
- Seleção do que deve ser representado;
- Possibilita o trabalho em níveis de implementação e uso;
- Uso disseminado na computação.

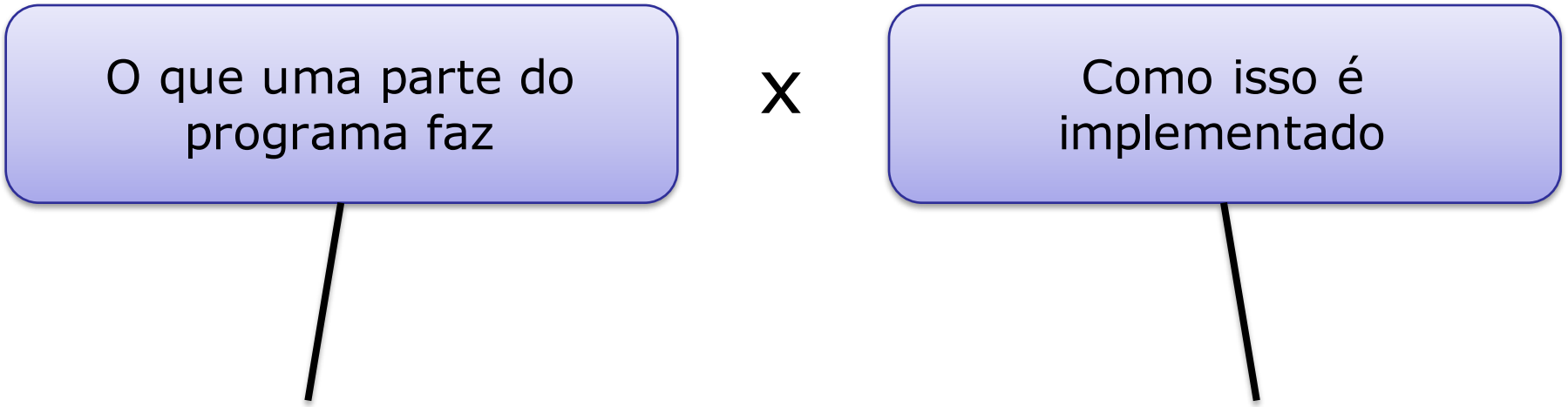


Abstração

- Exemplos de uso na computação:
  - Comandos do Sistema Operacional;
  - Assemblers (montadores);
  - Linguagens de Programação;
  - Um programa de reserva de passagens aéreas.
- 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:



Tipos de abstrações

- Abstrações de Processos:
  - Abstrações sobre o fluxo de controle do programa;
  - Suprogramas: funções da biblioteca padrão de C (ex.: `printf`);
- Abstrações de Dados:
  - Abstrações sobre as estruturas de dados do programa;
  - Tipos de Dados: tipos da biblioteca padrão de C (ex.: `FILE`).

Modularização

ABSTRAÇÕES DE PROCESSO

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 (mito antigo) possuem baixa qualidade;
  - Propósito único e claro facilita legibilidade, depuração, manutenção e reutilização;
  - Nome: sub1 vs. imprimeContrachequeFuncionarios.

Função = abstração de uma expressão.

Procedimento = abstração de um comando.

Perspectivas: usuário vs. implementador

- 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: usuário vs. implementador

- Usuário:
  - Função fatorial é mapeamento de n para (n!);
- Implementador:
  - Uso de algoritmo recursivo.

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

Cite um exemplo de uso de API em que você foi usuário e não se importou com a implementação.

Perspectivas: usuário vs. implementador

- Usuário:
  - Ordenação de vetor de inteiros;
- Implementador:
  - Método da bolha.

```
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;
        }
     }
  }
}
```

Parâmetros

```
int altura, largura, comprimento;

int volume () { return altura * largura * comprimento; }

int 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);
}
```

Qual o impacto da ausência de parâmetros em subprogramas?

Ausência de parâmetros 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

```
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);
}
```

O uso de parâmetros resolve os problemas.

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

Parâmetro formal.

```
float area (float r) {
  return 3.1416 * r * r;
}

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

Parâmetro real = diametro / 2
Argumento = 1.4

Correspondência entre parâmetros reais e formais pode ser: (a) posicional; ou (b) por palavras-chave

Parâmetros: correspondência real-formal

- Ada suporta as duas formas:

```
procedure palavrasChave is
   a: integer := 2;
   b: integer := 3;
   c: integer := 5;
   res: integer;

   function multipl(x, y, z: integer) return integer is
   begin
      return x * y * z;
   end multipl;

begin
   res := multipl(z=>b, x=>c, y=>a);
end palavrasChave;
```

Parâmetros: valores default

- Devem ser os últimos da lista. Exemplo 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;
}

int 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);
}
```

Parâmetros: *varargs*

- Lista de parâmetros variáveis em C:

```
#include <stdarg.h>
// Faz um OU lógico entre vários valores "booleanos".
int ou(int n, ...) {
  va_list vl;      // Estrutura varargs.
  int i;
  // Inicializa a lista, espec. último arg. antes dela.
  va_start (vl, n);

  for (i = 0; i < n; i++)
    // Obtém o próximo arg., convertendo ao tipo int.
    if (va_arg (vl, int)) return 1;

  // Invalida a estrutura(uso obrigatório).
  va_end (vl);
  return 0;
}
```

Parâmetros: *varargs*

- Oferece maior flexibilidade à LP;
- Em C, reduz a confiabilidade pois não é possível verificar os tipos dos parâmetros em tempo de compilação.

```
int 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));
}
```

Parâmetros: *varargs*

- A partir da versão 5, Java também oferece varags;
- Em Java, porém, há verificação do tipo;
- Parâmetro variável funciona como vetor simples.

```
public class Teste {
  private static boolean ou(boolean ... valores) {
     for (int i = 0; i < valores.length; i++)
        if (valores[i]) return true;
     return false;
  }
  public static void main(String ... args) {
     System.out.println(ou(3 < 2));
     System.out.println(ou(3 > 2, 7 > 5));
     System.out.println(ou(1 != 1, 2 != 2, 3 != 3));
     System.out.println(ou(1 != 1, 2 != 2, 3 == 3));
  }
}
```

Parâmetros: passagem

- 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.

Parâmetros: direção da passagem

Direção da passagem	Forma do parâmetro real (R)	Atribuição do parâmetro 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

Parâmetros: 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;
}

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

Parâmetros: direção da passagem

- C++ tem 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;
}

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

Parâmetros: direção da passagem

- Java usa passagem unidirecional de entrada;
- Se os parâmetros são objetos, porém, pode-se considerar passagem bidirecional do objeto ou passagem unidirecional da referência.

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

Parâmetros: 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);
```

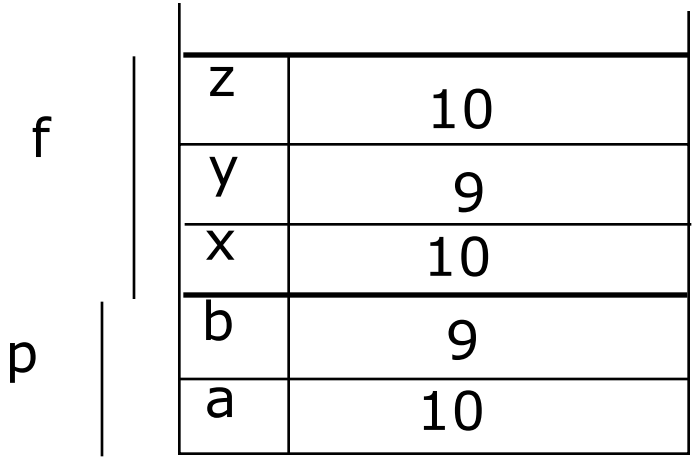
- Cuidado com colisões nas passagens unidirecional de saída e bidirecional:

```
incrementa (i, i);
```

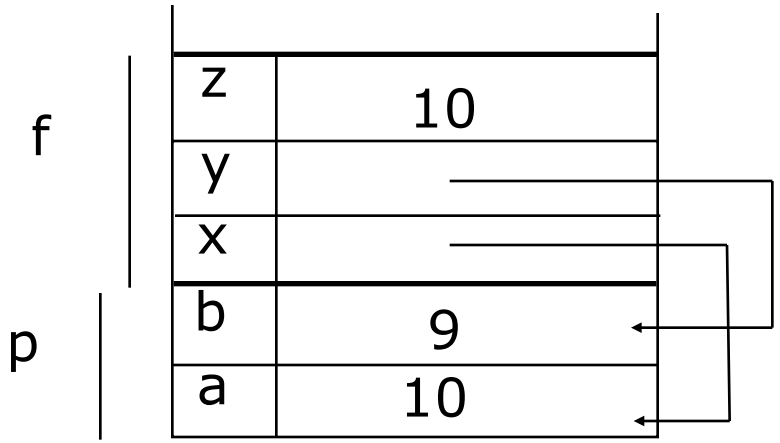
Parâmetros: mecanismos de passagem

```
void f(int x, int y) {
  int z = 10;
}

void p() {
  int a = 10, b = 9;
  f(a, b);
}
```



Cópia



Referência

Parâmetros: 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 (ex.: passar função como parâmetro);
  - 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;
}
// Imagine: incr(a[i], a[j]), com i == j. Incr. duplo.
```

Parâmetros: 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;
- Java adota passagem por cópia. Quando os parâmetros são objetos, pode-se considerar:
  - Passagem por referência dos objetos; OU
  - Passagem por cópia das referências (endereço de memória).

Parâmetros: 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.

Parâmetros: 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.
- Uso:
  - Maioria das LPs (C, Pascal, Java, Ada): modo normal;
  - ALGOL-60 permite escolher: normal e por nome;
  - SML entre normal e preguiçoso;
  - Haskell e Miranda usam preguiçoso;
  - Python: pacotes lazypy e objproxies.

Parâmetros: 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());
```

O que acontece em cada caso de momento da passagem?

Parâmetros: 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());
```

- Avaliação normal:
  - Avaliação desnecessária de funções (dependendo do valor de x);
  - Pode reduzir eficiência e flexibilidade.

Parâmetros: 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());
```

- Avaliação por nome:
  - Somente uma de `q()`, `r()` ou `s()` seria avaliada;
  - Porém, `p()` poderia ser avaliada duas vezes;
  - Problema mais grave se `p()` produzir efeitos colaterais!

Parâmetros: 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());
```

- Avaliação preguiçosa:
  - Única execução de `p()` e somente uma de `q()`, `r()` ou `s()`;
  - Porém não é intuitivo que as funções não sejam avaliadas.

Modularização

ABSTRAÇÕES DE DADOS

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 (ex.: Pascal, Ada e Java) fazem verificação estática;
- Versão original de C não requeria verificação estática. Outras versões permitiam que o programador escolhesse. Atualmente, *varargs* "desativa" verificação.

```
int origem (c) coord c; {
  return c.x*c.y*c.z;
}
```

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 Anônimos: definidos exclusivamente durante a criação de variáveis e parâmetros.

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

Tipos simples

- Agrupam dados relacionados em uma única entidade nomeada;
- Aumentam:
  - Reusabilidade: define o tipo apenas uma vez e o reutiliza em diversas variáveis;
  - Redigibilidade: idem;
  - Legibilidade: mais fácil de entender que variáveis são de um “tipo X” ao invés de ter que analisar estrutura;
  - Confiabilidade: não se pode atribuir a variável do “tipo X” algo que não seja desse tipo, mesmo que a estrutura seja idêntica.

Tipos simples

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

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--;
}
int 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:

```
tPilha a;
preenche(&a, 10);          // Uso recomendado.
a.elem[++a.topo] = 11;     // Usos não recomendados.
a.topo = 321;
```

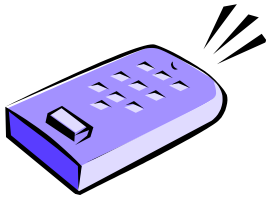
- Mistura código da implementação com o de uso do tipo;
- Programador pode alterar errônea e inadvertidamente a estrutura interna do tipo;
- Alteração na estrutura interna implica em alteração do código fonte usuário.

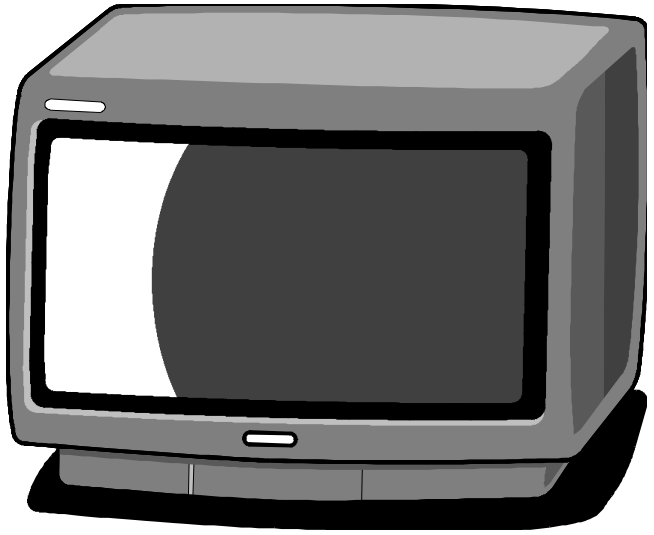
Tipos abstratos de dados (TADs)

- 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 (TADs)

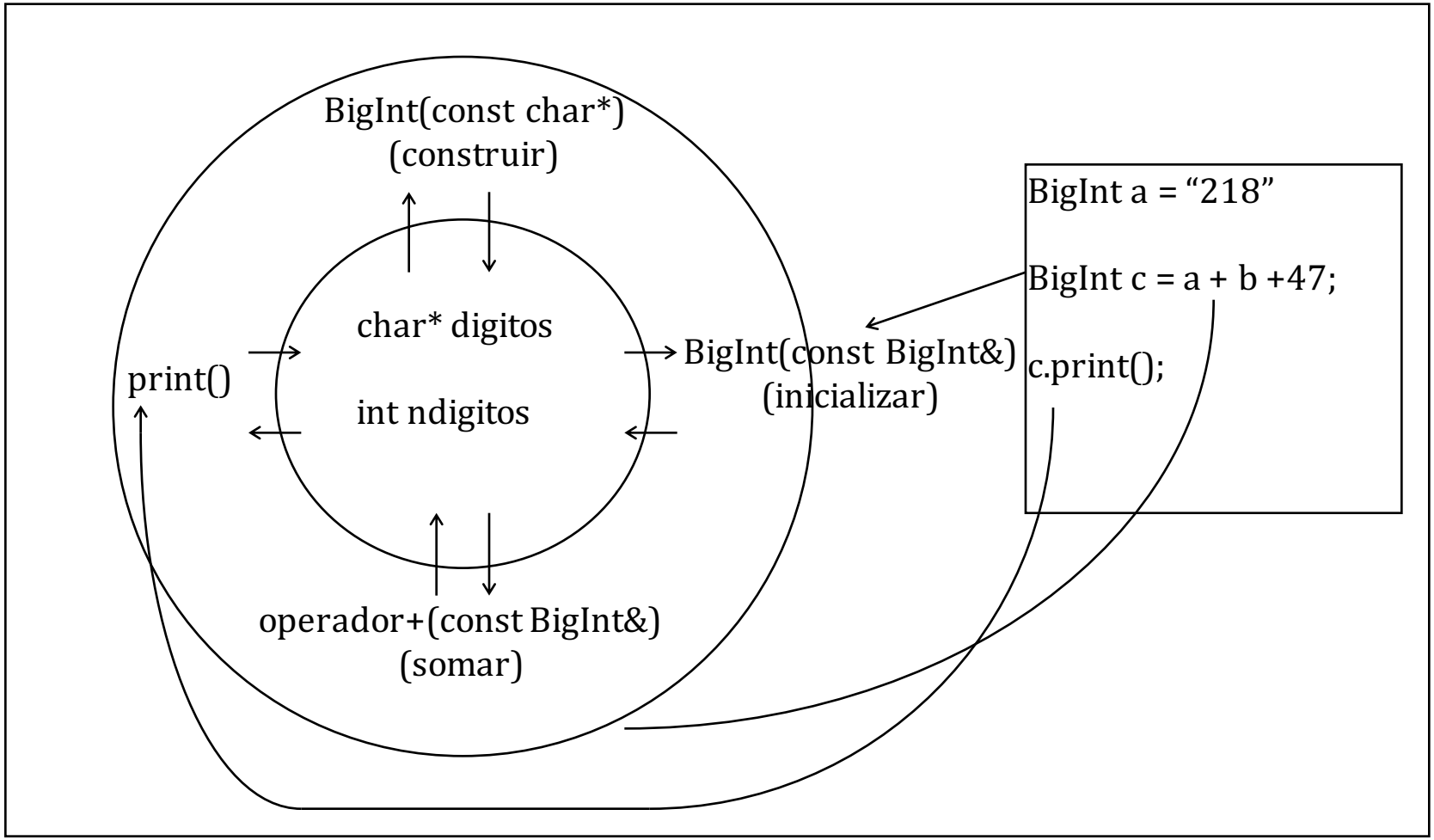
- Encapsulam e protegem os dados;
- Resolvem os problemas existentes com tipos simples;
- Quatro tipos diferentes de operações:
  - Construtoras;
  - Consultoras;
  - Atualizadoras;
  - Destrutoras.





Tipos abstratos de dados (TADs)

- Exemplo do curso de C++:



TADs em C

- Não há abstrações diretas na linguagem para TADs;
- O programador deve simulá-lo com os recursos que a linguagem oferece:
  - Declaração e definição;
  - Compilação separada.
- Exemplo: um TAD pilha com
  - cria: void → Pilha;
  - vazia: Pilha → int;
  - empilha: Pilha x int → Pilha;
  - desempilha: Pilha → int.

TADs em C: arquivo cabeçalho

```
#ifndef __TPilhaInt_H_
#define __TPilhaInt_H_

typedef struct TPilhaInt *PilhaInt;

PilhaInt criaPilhaInt(void);
int vaziaPilhaInt(PilhaInt);
PilhaInt empilhaPilhaInt(PilhaInt, int);
int desempilhaPilhaInt(PilhaInt);
void destroiPilhaInt(PilhaInt);

#endif
```

Um exemplo mais completo e complexo encontra-se no meu blog: http://www.inf.ufes.br/~vitorsouza/blog/aulas-sobre-listas-estruturas-de-dados-20151/

TADs em C: arquivo implementação

```
#include "Pilha.h"
#define max 100
struct TPilhaInt {
  int elem[max];
  int topo;
};

PilhaInt criaPilhaInt() {
  /* ... */
}

int vaziaPilhaInt(PilhaInt pilha) {
  /* ... */
}

PilhaInt empilhaPilhaInt(PilhaInt pilha, int elem) {
  /* ... */
}
                    /* Etc. */
```

TADs em C: código cliente

```
#include <stdio.h>
#include "Pilha.h"

PilhaInt global;
int main() {
  PilhaInt a, b;
  global = criaPilhaInt();
  a = criaPilhaInt();
  empilhaPilhaInt(a, 17);
  empilhaPilhaInt(a, 29);
  empilhaPilhaInt(global, 23);
  printf("%d\n", desempilhaPilhaInt(a));
  destroiPilhaInt(a);
  destroiPilhaInt(global);

  // a->elem[++a->topo] = 11;
  // a->topo= 321;
}
```

Uso de TADs promove

- Legibilidade:
  - Código cliente não se mistura com código de manipulação de estrutura interna do tipo;
- Redigibilidade:
  - Usuário do TAD não implementa as operações;
- Modificabilidade:
  - Modificar implementação do TAD não implica modificação no código cliente.

TADs em C

- Não promove encapsulamento das operações e dados em uma única unidade sintática;
- Criador do TAD deve ser disciplinado:
  - Estrutura (struct) deve estar no arquivo de implementação, do contrário será visível ao cliente;
  - Condições de guarda para evitar inclusão duplicada;
  - Repetição do nome do TAD nos nomes das operações para evitar colisão de nomes;
  - Compilação separada (Makefile) para evitar recompilação completa do código-fonte.
- Usuário também deve ser disciplinado: criando e destruindo a pilha adequadamente.

TADs em Ada: interface

```
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;
```

TADs em Ada: implementação

```
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;
  end empilha;
                                        -- Continua...
```

TADs em Ada: implementação

```
  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;
```

TADs em Ada: código cliente

```
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;
```

TADs em Ada

- Declaração de `tPilha` como `limited` garante não aplicação de atribuição e comparação;
- Promove:
  - Legibilidade;
  - Redigibilidade;
  - Confiabilidade;
  - Modificabilidade;
- Problema da falta de inicialização continua.

TADs como classes em C++: cabeçalho

- Operações construtoras e destrutoras especiais:

```
#ifndef __PilhaInt_h_
#define __PilhaInt_h_

class PilhaInt {
  int *elem;
  int topo;

public:
  PilhaInt(unsigned tamanho);
  bool vazia() { return topo == -1; }
  void empilha(int elem);
  int desempilha(void);
  ~PilhaInt() { delete elem; }
};

#endif
```

TADs como classes em C++: implement.

```
#include "Pilha.hpp"

PilhaInt::PilhaInt(unsigned tamanho) {
  /* ... */
}

void PilhaInt::empilha(int elem) {
  /* ... */
}

int PilhaInt::desempilha(void) {
  /* ... */
}
```

TADs como classes em C++: cliente

```
#include <iostream>
#include "Pilha.hpp"

using namespace std;

int main() {
  PilhaInt *p = new PilhaInt(10);
  p->empilha(1);
  p->empilha(2);
  while (! p->vazia()) cout << p->desempilha() << endl;
  delete p;
}
```

TADs como classes em C++

- Estrutura de dados privada e operações da interface públicas;
- Uso do operador de resolução de escopo;
- Podem haver vários construtores sempre chamados antes de qualquer outra operação;
  - Instâncias diretas: chamada implícita;
  - Ponteiros: chamada com new;
- Função destrutora é única (`~PilhaInt()`):
  - Chamada automaticamente para instâncias diretas;
  - Para ponteiros: chamada explicita de `delete`. Programadores devem ainda ser disciplinados!

TADs como classes em C++ (e Java)

- Exemplos de instâncias diretas:

```
PilhaInt p1, p2;
PilhaInt p3 = p1;       // Construtor de cópia.
p3 = p2;                // Operador de atribuição.
```

  - Construtor de cópia vs. operador de atribuição.
- Em Java:
  - Não há instâncias diretas;
  - Portanto, não há necessidade de construtor de cópia ou sobrescrita do operador de atribuição;
  - Não há função destrutora, coletor de lixo toma conta. `finalize()` útil em alguns casos, desaconselhado em outros.

Modularização

PACOTES & MODULARIZAÇÃO

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: agrupam tipos, variáveis, constantes e subprogramas úteis;
  - Aplicações utilitárias: usados como caixas pretas: entrada, processamento, saída;
  - Frameworks: implementações parciais, deve ser complementada por código específico.;
  - 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;
- Modula 2 possui `DEFINITION MODULE` e `IMPLEMENTATION MODULE`;
- Python possui módulos;
- Pacotes em C++:
  - Conjunto de definições embutidas em uma entidade nomeada;
  - Uso do conceito de espaço de nomes (*namespace*).

Espaços de nome em C++

- À medida que os códigos crescem em tamanho, aumenta a chance de colisão de nomes;
- Para evitá-lo, C++ oferece o mecanismo de *namespaces*;
  - Dois nomes iguais em espaços de nomes diferentes não colidem;
  - A API do C++ é declarada no espaço de nome `std`;
- Define-se um *namespace* como se define classes:

```
namespace mylib {
  /* ... */
}
```

Espaços de nome em C++

- Ao contrário de classes, a redefinição de um *namespace* (ex.: em outro arquivo) continua a definição anterior;
- É possível criar apelidos para *namespaces*:

```
namespace VitorSouzaLibrary {
  void hello() { cout << "Hello, world!\n"; }
}
namespace vsl = VitorSouzaLibrary;

int main () {
  VitorSouzaLibrary::hello();
  vsl::hello();
}
```

- Pode-se também usar também a diretiva:
  `using namespace <nome-do-namespace>;`

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

- Diferente dos *namespaces* em C++, os pacotes Java:
  - Possuem uma relação com a organização dos arquivos de código-fonte/classes em diretórios;
  - Determinam quem pode acessar um elemento sem declaração de visibilidade (*package-private*).

Modularização dos arquivos

- Inicialmente se usava um arquivo único para o programa, o que 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.

Compilação separada de código .c

- Uso de ligação para gerar executável;
- Porém, ainda sem arquivo cabeçalho;
- 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;
  - Gera repetição de declaração/definição da variável.

Uso de interface (.h) e implementação (.c)

- 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.

Interface e implementação em TADs

- Ada e versões antigas de C requerem definição do tipo na interface (prejudica o ocultamento);
  - Se usava `void*` para ocultar o tipo. Hoje usa-se declaração no .h e definição no .c;
  - Modula 2 tem solução parecida com `void*`, chamada Tipo Opaco.
- Ada e C++ possibilitam o ocultamento da estrutura interna, mesmo na interface (`private`);
- Necessidade de recompilar o código usuário após alteração da estrutura interna do TAD;
- Java usa arquivo único com interface e implementação. Não requer recompilação de código cliente se interface permanece inalterada.

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.

http://nemo.inf.ufes.br/