Tipos Abstratos de Dados

Estrutura de Dados

Universidade Federal de Juiz de Fora
Departamento de Ciência da Computação

Conteúdo

Introdução

- Um tipo de dados é constituído de um conjunto de objetos (domínio de dados) e de um conjunto de operações aplicáveis aos objetos do domínio.
- Toda linguagem de programação tem embutido um conjunto de tipos de dados, também chamados de implícitos, primitivos ou básicos.
- As linguagens disponibilizam mecanismos que permitem construir combinações dos tipos primitivos (vetor, registro etc.).

Introdução

Abstração

- Abstrair significa desconsiderar detalhes.
- Refinamentos sucessivos
- A abstração no método dos refinamentos sucessivos

N_1	nível conceitual
$N_2, N_3, \ldots$	níveis intermediários
N_i	nível de implementação (programa)

Introdução

Abstração

Nível	Código	Dados
N_1	nível conceitual (programa abstrato)	Trata dos objetos do mundo real do problema e de suas respectivas operações. Exemplo: **Objetos**: aluno, disciplina, livro, mercadoria etc. **Operações**: matricular aluno em disciplina, cadastrar livro, comprar mercadoria, etc.
N_2 N_3 ... N_{i-1}	níveis intermediários	Detalhamento gradativo dos objetos e de suas operações.
N_i	nível de implementação (programa na linguagem de programação)	Trata dos objetos como **representações** na linguagem de programação e as operações que manipulam essas representações.

Introdução

Abstração

- **Exemplo:** Dados 3 valores x, y e z verificar se eles podem ser os comprimentos dos lados de um triângulo e, se forem, verificar se é um triângulo equilátero, isósceles ou escaleno. Se eles não formarem um triângulo, escrever uma mensagem.

Introdução

Abstração

- Nível 1
  - Programa abstrato (Algoritmo em alto nível de abstração)

```
inicio
  "declarar as variaveis";
  "leia os 3 numeros";
  se "existe triangulo" entao
    "verifique o tipo de triangulo";
  senao
    "escreva mensagem";
  fim-se;
fim
```

Introdução

Abstração

- Nível 2
  - Programa abstrato (Algoritmo em alto nível de abstração)

```
inicio
  // declaração das variáveis
  int X, Y, Z;

  // leitura dos 3 números
  leia(X, Y, Z);

  se "existe triangulo" entao
    "verifique o tipo de triângulo";
  senao
    imprima("Os lados", X, Y, "e", Z,
            "não formam triângulo");
  fim-se;
fim
```

Introdução

Abstração

- Nível 3: refinamento de ”Existe triangulo”e outros comandos de alto nível.
- Nível 4: nível de implementação

```
#include <iostream>

using namespace std;

void verifica_escaleno(int x, int y, int z)
{
  if ((x == y) || (x == z) || (y == z))
    cout << "Triangulo isosceles" << endl;
  else
    cout << "Triangulo escaleno" << endl;
}
```

Introdução

Abstração

- Nível 4: Nível de implementação

```
int main()
{
  int x, y, z;
  cout << "Digite tres valores inteiros\n";
  cin >> x >> y >> z;
  if ((x < y+z) && (y < x+z) && (z < x+y))
  {
    // verifica tipo de triangulo
    if ((x == y) && (x == z))
      cout << "Triangulo equilatero";
    else
      verifica_escaleno(x, y, z);
  }
  else
    cout << x << ", " << y << " e"
         << z << " nao formam triangulo";
}
```

Domínio de Dados

- É um conjunto determinado de objetos D.
- Exemplos:
  - Domínio de (dados) inteiros: $D = \{0, \pm 1, \pm 2, \ldots\}$
  - Domínio de sequência de caracteres alfabéticos maiúsculos de comprimento inferior a trinta e um: $D = \{'A', 'B', 'C', \ldots, 'Z', 'AA', 'AB', \ldots\}$
- Observações:
  - O domínio pode ser finito ou infinito
  - Muitas vezes é necessário utilizar mecanismos especiais em um programa para representar os objetos de um determinado domínio.

Domínio de Dados

- Classificação de Domínios
  - Um domínio pode ser **simples** ou **estruturado**.
- **Simples**: objetos indivisíveis, atômicos, não acessíveis ao programador. Podem ser:
  - **Primitivo**: Mecanismo: embutido na linguagem.
  - **Definido**: Mecanismo: enumeração ou restrição.
- **Estruturado** (Construído ou Agregado): objetos construídos por componentes acessíveis ao programador:
  - **Homogêneo**: componentes pertencem ao mesmo domínio. Mecanismo: array, string e ponteiro.
  - **Heterogêneo**: componentes não são necessariamente do mesmo domínio. Mecanismo: `struct` e `class`.

Domínio de Dados

Simples Primitivo

- Embutido na linguagem.
- Alguns exemplos em C++
  - `char`, `int`, `float`, `double`, `bool`
- Não há necessidade de serem definidos pelo programador, pois esses domínios (**tipos de dados** primitivos já estão definidos na própria linguagem.
- Propriedades importantes dos tipos primitivos:
  - **Invisibilidade**: A representação interna de um objeto como cadeia de bits em palavras da memória é invisível e inacessível ao programador
  - **Proteção**: a manipulação dos objetos por suas operações garante que não serão gerados objetos inválidos
- Observação: Mesmo nos tipos primitivos pode-se notar abstração de dados, mas ainda muito próximo da máquina, longe do problema.

Domínio de Dados

Simples Definido

- Enumeração

```
enum nomeTipo = { lista de valores };
```

- Exemplo de definição:

```
enum DiaUtil = {SEG, TER, QUA, QUI, SEX};
```

- Exemplo de uso:

```
DiaUtil niver;
// atribui algum valor a niver

if(niver == SEG) {
  // processamento
}
else if (niver == SEX) {
  // outro processamento
}
else { ... }
```

Domínio de Dados

Simples Definido

- Restrição: Em C++ não há uma forma genérica de construir subdomínios a partir da restrição de outros domínios.
- Entretanto, pode-se utilizar as seguintes opções para restringir o intervalo de valores de alguns tipos primitivos:
  - `short`, `signed` e `unsigned`
- Exemplo:

```
unsigned int x;
```

- Pode-se também definir outro tipo de dados com o comando `typedef`:

```
typedef unsigned int Uint;
```

- Assim, pode-se declarar variáveis da seguinte forma:

```
Uint x, y, z;
```

Domínio de Dados

Simples Definido

- Restrição

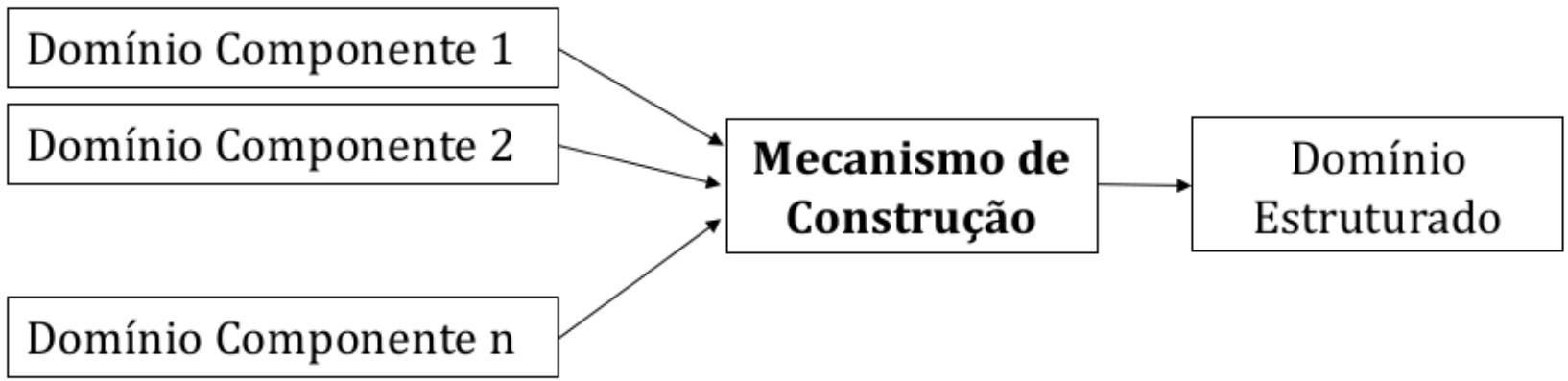
Nome	Bytes	Equivalência	Intervalo
int	4	signed	–2,147,483,648 to 2,147,483,647
unsigned int	4	unsigned	0 to 4,294,967,295
bool	1	none	false or true
char	1	none	–128 to 127
signed char	1	none	–128 to 127
unsigned char	1	none	0 to 255

Nome	Bytes	Equivalência	Intervalo
short	2	short int, signed short int	–32,768 to 32,767
unsigned short	2	unsigned short int	0 to 65,535
long	4	long int, signed long int	–2,147,483,648 to 2,147,483,647
unsigned long	4	unsigned long int	0 to 4,294,967,295
float	4	none	3.4E +/- 38 (7 dígitos)
double	8	none	1.7E +/- 308 (15 dígitos)

Domínio de Dados

Estruturado

- Um objeto **estruturado** é formado por uma ou mais componentes que são objetos de domínios menos complexos (simples ou estruturados).



- Um mecanismo de construção é especificado por:
  - **definição**
  - **funções de transferência**

Domínio de Dados

Estruturado

- Mecanismo de construção: **definição**
  - Identifica o mecanismo e os domínios componentes.
  - Formato geral:

```
typedef nomeAntigo nomeTipo;
```

- Mecanismo de construção: **funções de transferência**
  - Construtora: operação para criar um objeto estruturado a partir de objetos dos domínios componentes.
  - Seletora: operação para decompor um objeto estruturado em seus objetos componentes.
- Alguns mecanismos de construção em C++:
  - vetor (array)
  - registro (struct)
  - sequência (string)
  - referência (ponteiro)

Domínio de Dados

Estruturado homogêneo - Vetor

- Exemplo de vetor (array):

```
typedef float ValorMes[12];
```

  - Uma variável do tipo ValorMes é um vetor com 12 componentes reais, numeradas de 0 a 11.

- Exemplo

```
// definicao
typedef int X[6];

// declaracao de uma variavel
X var;

// acesso
var[1] = 100;
```

- Assim, a variável `var` do tipo `X` possui 6 valores inteiros que podem ser acessados como `var[0]`, ... , até `var[5]`.

Domínio de Dados

Estruturado heterogêneo - Registro

- Exemplos de criação de registros:

```
typedef struct { int num;
                 int salario; } Funcionario;

typedef struct { int num;
                 char turno;
                 bool licenca; } Situacao;

typedef struct { int disciplina;
                 char turma; } Inscricao;
```

- Ex. declaração de variável

```
Inscricao matricula;
```

- Se a variavel `matricula` assumir a constante (49, 'C') então o valor de `matricula.disciplina` é 49 e de `matricula.turma` é 'C'.

Domínio de Dados

Estruturado

- **Aplicação**: desenvolver um programa para ler um vetor de 50 elementos inteiros, calcular e imprimir:
  - A soma dos elementos
  - A soma dos elementos pares
  - A soma dos elementos de índice par
  - O maior elemento
  - O índice do menor elemento
  - O vetor em ordem crescente de seus elementos juntamente com os índices originais

Domínio de Dados

Estruturado

```
typedef int VetN[50];

int main()
{
  VetN V;
  int maior, menor;

  // leitura do vetor V
  for(int i=0; i<50; i++)
    cin >> V[i];

  // calculo e impressao da soma, spar e sipar
  calcSomas(V);
  // calculo e impressao: maior e indice do menor
  calcMaiorMenorImp(V);
  // ordena e imprime c/ indices originais
  ordenaImprime(V);
}
```

Domínio de Dados

Estruturado

```
void calcSomas(VetN V)
{
  int i, soma=0, spar=0, sipar=0;
  for(i=0; i<50; i++)
  {
    soma += V[i];
    if(V[i] % 2 == 0)
      spar += V[i];
    if(i % 2 == 0)
      sipar += V[i];
  }
  cout << "Soma = " << soma << endl;
  cout << "Soma pares = " << spar << endl;
  cout << "Soma indices pares = " << sipar << endl;
}
```

Domínio de Dados

Estruturado

```
void calcMaiorMenorImp(VetN V)
{
  int maior=V[0], menor=V[0], i;
  for(i=1; i<50; i++){
    if(V[i] > maior) maior = V[i];
    else if(V[i] < menor) menor = V[i];
  }
  cout << "Maior = " << maior << endl;
  impMenor(V, menor);
}

void impMenor(VetN V, int menor)
{
  int i;
  cout << "Indice(s) do menor:" << endl;
  for(i=0; i<50; i++)
    if(V[i] == menor)
      cout << i;
}
```

Domínio de Dados

Estruturado

```
void ordenaImprime(VetN V){
  VetN ind;
  for(int i=0; i<50; i++)
    ind[i] = i;

  // ordenacao de V e de ind
  ordena(V, ind);

  for(int i=0; i<50; i++){
    cout << "Elemento " << V[i] << endl;
    cout << "Indice original " << ind[i] << endl;
  }
}

void ordena(VetN V, VetN Ind){
  int i, j, indMenor, Menor;
  for(i=0; i<49; i++){
    indMenor = i;
    Menor = V[i];
    for(j=i+1; j<50; J++){
      if(V[j] < Menor){
        indMenor = j;
        Menor = V[j];
      }
    }
    V[indMenor] = V[i];
    V[i] = Menor;
    Ind[indMenor] = Ind[i];
    Ind[i] = indMenor;
  }
}
```

Domínio de Dados

Estruturado - `typedef`

- Seja a definição do domínio `Vet` e a declaração das variáveis `A`, `B` e `C`:

```
typedef float Vet[11];
Vet A, B, C;
```

- Pode-se obter o mesmo resultado com a seguinte declaração:

```
float A[11], B[11], C[11];
```

Domínio de Dados

Estruturado - `typedef`

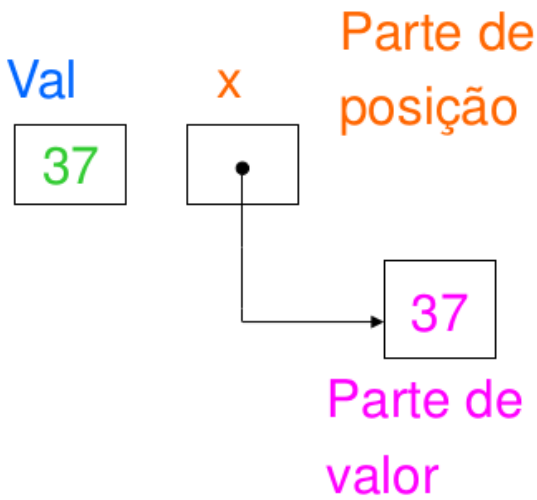
- Vantagens da definição de domínios com `typedef`:
  - **Legibilidade**: Usando-se nome de domínios apropriados pode-se melhorar a legibilidade de programas.
  - **Modificabilidade**: Uma alteração nos parâmetros do mecanismo de construção requer modificação apenas na definição do domínio e não na declaração de todas as variáveis (ver, variáveis A, B e C).
  - **Fatoração**: A definição de uma estrutura de dados protótipo complicada é escrita uma só vez e então usada quantas vezes forem necessárias para declarar variáveis. Isto reduz a quantidade de codificação necessária para copiar a mesma definição para cada variável e diminui a possibilidade de erros por distração.

Domínio de Dados

Estruturado - Ponteiro

- Permite a alocação dinâmica de memória (em tempo de execução) e para isto, uma variável do tipo ponteiro necessita de duas partes na memória, chamadas de: parte de **posição** e parte de **valor**.

```
typedef int* R;

int val;
R x;

cin >> val;
x = new int;
*x = val;
cout << *x;

delete x;
x = NULL;
```



Invisibilidade, Encapsulamento e Proteção

Tipos de Dados e Abstração

Invisibilidade, Encapsulamento e Proteção

- No desenvolvimento de um programa, deve-se organizar as definições de dados envolvidas, de modo que:
  - O texto do programa reflita as abstrações usadas no projeto (ex.: Aluno, Curso, Turma,..)
  - Seja feita distinção clara entre os níveis de abstração

- Assim, usa-se pelo menos dois níveis de abstração:
  - Abstrações do projeto (entidades como Aluno e Curso + operações sobre estas entidades: matricular Aluno em Curso)
  - Abstrações da implementação (objetos concretos disponíveis na linguagem, ver domínios)

Tipos de Dados e Abstração

Invisibilidade, Encapsulamento e Proteção

- Uma forma de se obter essa organização é através do princípio da invisibilidade da informação usado no nível de abstração do projeto:
  - As estruturas de dados que representam os objetos abstratos (Aluno, Curso, Turma, ...) são invisíveis
  - As informação desejadas só são fornecidas através de um elenco de funções de acesso, protegendo, desta forma, as informações internas

Tipos de Dados e Abstração

Invisibilidade, Encapsulamento e Proteção

- Juntamente com a invisibilidade, deve-se levar em conta os seguintes conceitos de programação:
  - **Proteção**: os objetos só podem ser manipulados por meio das operações de um elenco pré-estabelecido, permitindo que a representação dos objetos permaneça invisível e protegendo contra manipulações indisciplinadas.
  - **Encapsulamento**: a representação dos objetos e implementação das operações abstratas ficam reunidas numa única unidade sintática;

Tipos Abstratos de Dados

- **Definição**: tipos de dados (domínio + operações) definidos pelo usuário que satisfazem as propriedades de invisibilidade, proteção e encapsulamento são chamados de **Tipos Abstratos de Dados (TAD)**.
  - O encapsulamento faz com que a versão final do programa reflita as abstrações encontradas durante o projeto do programa (a estrutura fica auto explicativa).
  - A proteção compele à distinção entre níveis de abstração e favorece a modificabilidade do programa.
- Uma característica importante no desenvolvimento de programas usando TAD é a fatoração do programa em:
  - Programa abstrato (PA) envolvendo operações abstratas sobre objetos abstratos; e
  - Módulo de implementação (MI) encapsulando a representação dos objetos e a codificação das operações.

Tipos Abstratos de Dados

Invisibilidade, Encapsulamento e Proteção

- Ao escrever o PA, o programador (abstrato) emprega informação sobre o que fazem as operações abstratas;
- O MI contém a representação dos objetos abstratos e as operações que manipulam esta representação. Assim, ao escrever o MI, o implementador se preocupa em como codificar as operações para manipular as representações concretas dos objetos abstratos.
- Há uma independência entre PA e MI. Vantagens:
  - legibilidade: o PA e o MI podem ser lidos e entendidos isoladamente e com mais facilidade.
  - modificabilidade: alterações na estrutura de representação ficam localizadas dentro do MI, não afetando o PA, e, analogamente o mesmo MI pode suportar vários PA's;
  - portabilidade: para se transportar o programa de uma instalação (SO + hardware) para outra, basta alterar as partes do MI que dependem da instalação, o PA não é afetado.

Tipos Abstratos de Dados

Invisibilidade, Encapsulamento e Proteção

- Todo tipo de dados possui as características de invisibilidade e proteção.
- Usando apenas os mecanismos de definição e construção:
  - Não há invisibilidade: um objeto estruturado é manipulado indiretamente através da manipulação de seus objetos componentes.
  - Não há proteção: o programador não dispõe de operações que manipulem diretamente os objetos do domínio construído.
- Os mecanismos de definição e construção não criam tipos de dados, mas somente domínios de dados.
- Com os domínios definidos/construídos, passa a existir uma clara mistura de níveis de abstração de dados, isto é, nível de abstração do problema e da linguagem.

Tipos Abstratos de Dados

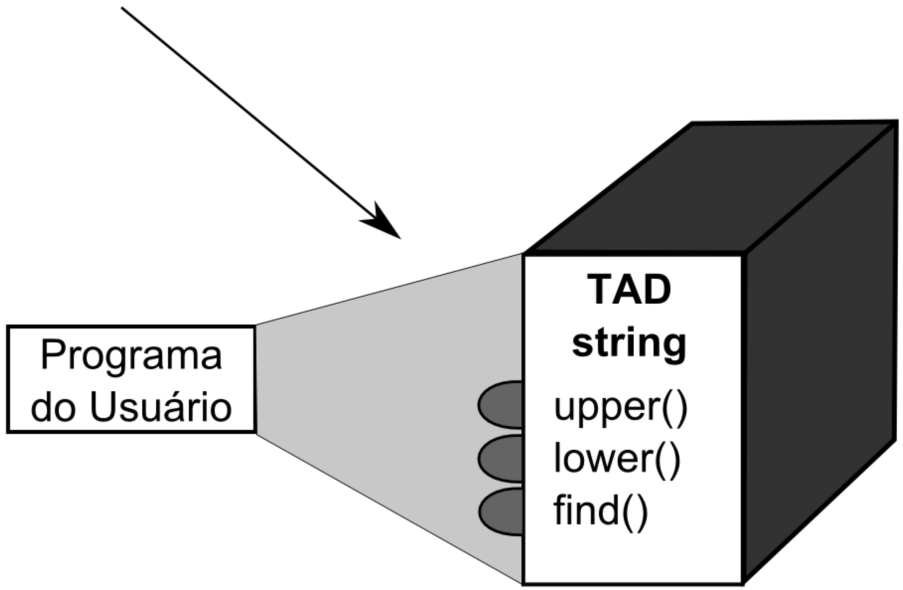
Invisibilidade, Encapsulamento e Proteção

- Para se ter os benefícios do princípio da invisibilidade, é crucial a independência entre PA e MI e, para tanto, precisamos de recursos da linguagem que suportam o encapsulamento e a proteção.
- A linguagem de programação C++ fornece tal recurso através das **classes**.
- Desenvolvimento em três passos:
  1. Projetar um TAD considerando disponíveis operações abstratas do nível do problema em questão. (Ex.: "matricular estudante em disciplina";).
  2. Desenvolver um programa abstrato (PA), usando os TAD's.
  3. Tornar o PA executável, desenvolvendo um módulo de implementação (MI), onde os objetos abstratos devem ser representados em termos dos objetos concretos disponíveis na linguagem e, em seguida, cada operação abstrata é descrita por um procedimento ou função que manipula os objetos assim representados.

Tipos Abstratos de Dados

Invisibilidade, Encapsulamento e Proteção

Os programas do usuário interagem
com um TAD através da sua interface



Os detalhes da implementação
estão escondidos, como se fosse
uma caixa preta.

Tipos Abstratos de Dados

Classes

- Será usado o conceito de classe para criar o TAD.
- Uma classe em C++ é uma forma especial de definição de domínios, adicionando-se a possibilidade de definir componentes que são funções (métodos).
- A definição da classe começa com a palavra chave **class**.
- O corpo da classe fica dentro de um par de chaves, { } **;** (finalizando com um ponto e vírgula).
- Sintaxe:

```
class nomeDaClasse
{
  // ... corpo da classe ...
};
```

- As classes determinam quais são os métodos e os domínios de um objeto.

Tipos Abstratos de Dados

Classes

- O corpo da classe é constituído de membros, que são:
  - Variáveis (ou atributos); formam o domínio.
  - Funções (ou métodos); operações.
- Categorias de permissão: os membros (variáveis ou funções) de uma classe podem ser:
  - `private`: só podem ser acessados pelos membros da própria classe.
  - `public`: podem ser acessados em qualquer lugar (na própria classe ou, por exemplo, no programa principal)
- A seguir tem-se um exemplo de uma classe para representar um polígono regular, isto é, um polígono que tem todos os seus lados e ângulos iguais.

Tipos Abstratos de Dados

Classes - TAD Poligono Regular

- TAD Poligono Regular

```
class Poligono
{
 private:
  int numLados;  // numero de lados do poligono
  float tamLado; // tamanho de cada lado

 public:
  float area();         // calcula area
  float perimetro();    // calcula perimetro
  float angInterno(); // calcula angulo interno
};
```

- A definição da classe `Poligono` deve ser feita no arquivo **Poligono.h**

Tipos Abstratos de Dados

Classes

- A implementação das funções membro (operações) `area()`, `perimetro()` e `angInterno()` pode ser feita dentro da classe ou fora da classe.
- Será adotado nesse curso o padrão de implementar as operações fora da classe no Módulo de Implementação (MI).
- Logo, a implementação das funções membros (operações) devem ser feitas no arquivo **Poligono.cpp** e o operador `::` deve ser usado.
- A classe passa a possuir apenas o protótipo do método.
- Sintaxe:

```
NomeDaClasse::nomeDaFuncao(...)
{
  // implementacao
}
```

Tipos Abstratos de Dados

Classes

- Exemplo do arquivo **Poligono.cpp** com as implementações das operações

```
float Poligono::area()
{
  return numLados*pow(tamLado,2) / (4*tan
      (3.1416/numLados));
};

float Poligono::perimetro()
{
  return numLados*tamLado;
};

float Poligono::angInterno()
{
  return 180*(numLados - 2)/numLados;
};
```

Tipos Abstratos de Dados

Classes

NomeDoTad.cpp

NomeDoTad.h

- Em resumo:
  - Arquivo **.h**: é onde a definição da classe é feita, isto é, quais são as suas variáveis e funções membro.
  - Arquivo **.cpp**: é onde é feita a implementação das funções membro da classe, ou seja, onde de fato os algoritmos são implementados.

Tipos Abstratos de Dados

Classes

- O usuário só tem acesso à parte `public`: que lhe fornece informações somente para fazer chamadas de funções sintaticamente corretas.
- De acordo com o princípio da **invisibilidade**, as partes de representação e implementação, encapsuladas na classe `Poligono`, não são visíveis ao usuário do TAD.
  - Não tem acesso para alterar as variáveis membro `numLado` e `tamLado` que são privadas.
  - Embora as operações `area()`, `perimetro()`, etc sejam públicas e possam ser chamadas, o usuário não sabe como estas são implementadas. Este só precisa saber que essas operações calculam as quantidades desejadas.

Tipos Abstratos de Dados

Classes

- **Construtor**
  - É uma função (método) especial que é chamado quando um novo objeto é criado.
  - Deve possuir o mesmo nome da classe.
  - Não possui valor de retorno.
  - É utilizado para inicializar as variáveis (atributos) da classe e realizar algum outro processamento, se necessário.
- **Destrutor**
  - Método especial que é chamado automaticamente quando um objeto está prestes a ser apagado da memória.
  - Deve ter o mesmo nome da classe mas precedido pelo caractere˜(til).
  - Assim como o construtor ele não possui valor de retorno.
  - Além disso, o destrutor não pode ter parâmetros.

Tipos Abstratos de Dados

Classes - TAD Poligono Regular

- TAD Poligono Regular

```
class Poligono
{
 private:
  int numLados;  // numero de lados do poligono
  float tamLado; // tamanho de cada lado

 public:
  Poligono(int n, float l); // construtor
  ~Poligono();              // destrutor
  // ...
  void setNumLados(int n);
  void setTamLado(int l);
};
```

Tipos Abstratos de Dados

Classes - TAD Poligono Regular

```
Poligono::Poligono(int n, float l)
{
  setNumLados(n);
  setTamLado(l);
}

Poligono::~Poligono() { }

void Poligono::setNumLados(int n)
{
  if (n >= 3) numLados = n;
}

void Poligono::setTamLado(int l)
{
  if(l > 0) tamLado = l;
}
```

TAD Aluno

- Exemplo de um TAD para representar um aluno da UFJF.
- Aluno.h

```
class Aluno
{
  public:
    Aluno(string n, string m);          // construtor
    ~Aluno();                           // destrutor

    void info();                        // operacoes
    float getNota();
    string getNome();
    void setNota(float valor);
    bool verificaAprovado();

  private:
    string nome;                        // dados
    string matricula;
    float nota;
};
```

TAD Aluno

- Aluno.cpp (1/3)

```
Aluno::Aluno(string n, string m) {
  nome = n;
  matricula = m;
}

Aluno::~Aluno() {
  cout << "Destruindo aluno: " << nome << endl;
}

float Aluno::getNota() {
  return nota;
}

void Aluno::setNota(float valor) {
  cout << "Alterando nota do aluno" << endl;
  nota = valor;
}
```

TAD Aluno

- Aluno.cpp (2/3)

```
bool Aluno::verificaAprovado() {
  if (nota >= 60.0)
    return true;
  else
    return false;
}

string Aluno::getNome() {
  return nome;
}
```

TAD Aluno

- Aluno.cpp (3/3)

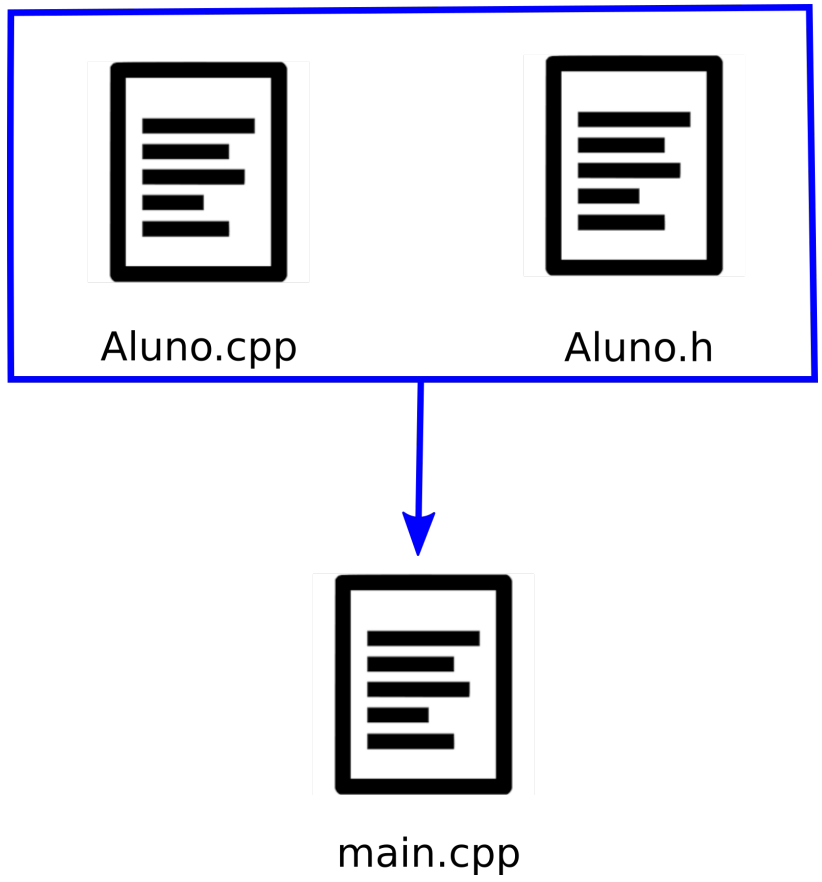
```
void Aluno::info()
{
  cout << "Nome:       " << nome << endl;
  cout << "Matricula:  " << matricula << endl;
  cout << "Nota:       " << nota << endl;

  if ( verificaAprovado() )
    cout << "Situacao: aprovado" << endl;
  else
    cout << "Situacao: reprovado" << endl;
}
```

TAD Aluno

- Exemplo de programa que usa o tipo de dados `Aluno`
- main.cpp

```
#include <iostream>
#include "Aluno.h"

int main()
{
  Aluno a("Fulano", "20154565AB");

  cout << "Digite a nota de "
       << a.getNome() << endl;
  float n;
  cin >> n;
  a.setNota(n);
  a.info();

  return 0;
}
```

TAD Aluno



Exemplos

Exemplo

TAD Relogio

- Desenvolver um TAD para representar um relógio. Você deve fornecer as seguintes operações:
  - consultar as horas;
  - consultar os minutos;
  - consultar os segundos;
  - acertar o relógio;
  - adiantar o relógio um segundo.

Exemplo

TAD Relogio

```
class Relogio
{
 public:
   void acertar(int h, int m, int s);
   int getHora();
   int getMinuto();
   int getSegundo();
   void tique();

 private:
   int horas, minutos, segundos;
};
```

Exemplo

TAD Relogio 1/2

```
void Relogio::acertar(int h, int m, int s)
{
  horas    = h;
  minutos  = m;
  segundos = s;
}

int Relogio::getHora() {
  return horas;
}

int Relogio::getMinuto() {
  return minutos;
}

int Relogio::getSegundo()) {
  return segundos;
}
```

Exemplo

TAD Relogio 2/2

```
void Relogio::tique()
{
  segundos++;

  if(segundos >= 60) {
    minutos++;
    segundos -= 60;
  }
  if(minutos >= 60) {
    horas++;
    minutos -= 60;
  }
  if(horas > 12) {
    horas -= 12;
  }

}
```

Exemplo

Aplicação com TAD Relogio

```
#include "Relogio.h"

int main()
{
  Relogio r;

  r.acertar(10,59,58);
  r.tique();
  r.tique();
  r.tique();
  r.tique();
  cout << "Hora certa: ";
  cout << r.getHora() << ":"
       << r.getMinuto() << ":"
       << r.getSegundo() << "\n";

  return 0;
}
```

Construtor e Destrutor

- Quando um construtor (destrutor) não é especificado, a linguagem C++ automaticamente define um construtor (destrutor) padrão de forma implícita.
- O construtor (destrutor) padrão implícito tem corpo vazio.

```
Relogio::Relogio()
{
  // nao faz nada
}

Relogio::~Relogio()
{
  // nao faz nada
}
```

Exemplo

TAD Relogio

- Modifique o TAD `Relogio` e inclua um construtor que ao criar um objeto acerte a hora, isto é, faça um construtor que receba como parâmetros a hora, os minutos e os segundos.

Exemplo

TAD Relogio com Construtor

```
class Relogio {
 public:
   Relogio(int h, int m, int s);
   // ...
};
```

- Implementação do construtor

```
Relogio::Relogio(int h, int m, int s) {
  horas = h;
  minutos = m;
  segundos = s;
}

// ou

Relogio::Relogio(int h, int m, int s)
{
  acertar(h,m,s);
}
```

Construtores

- É possível definir vários construtores ao mesmo tempo.
- Construtor sem parâmetro, com parâmetros e ainda diferentes listas de parâmetros.
- Não se pode definir 2 construtores com o mesmo protótipo.

```
class Relogio
{
 public:
   // inicializa tudo com zero
   Relogio();

   // acerta hora
   Relogio(int h, int m, int s);

   // inicializa na "hora em ponto"
   Relogio(int h);

   //...
};
```

Funções de Modificação e de Acesso

- Como visto no TAD `Relogio` e em outros exemplos, quando se define atributos com `private` pode ser preciso fornecer funções para modificar ou acessar esses atributos.
  - Ex: `getHora()`, `getMinuto()` e etc...
- É muito comum nomear essas funções com *get* e *set*.
  - `getHora()`: retorna hora
  - `setHora(int h)`: altera hora
- Isso é questão de **estilo**.
- Poderia ser:
  - `obtemHora()`: retorna hora
  - `fixaHora(int h)`: altera hora
- Vamos adotar o padrão com *get* e *set* durante o curso de Estrutura de Dados.

Exemplo

TAD Venda

- Definir e implementar um TAD para representar as vendas de uma determinada loja.
- O TAD deve considerar os seguintes dados:
  - número de itens N vendidos;
  - valor (número real) de venda de cada item (vetor).
- além das seguintes operações:
  - construtor (alocar memória de forma dinâmica para o vetor de itens e fazer a leitura do valor dos itens);
  - destrutor (desalocar a memória alocada no construtor);
  - calcular o total vendido;
  - retornar o valor do item mais caro vendido.

Exemplo

TAD Venda

```
class Venda
{
 public:
   Venda(int n);
   ~Venda();
   double calculaTotal();
   double itemMaisCaro();

 private:
   int numItensVendidos;
   double * valorItens;
};
```

Exemplo

TAD Venda (Construtor)

```
Venda::Venda(int n)
{
  numItensVendidos = n;

  // aloca o vetor de forma dinamica
  valorItens = new double[numItensVendidos];

  // leitura dos itens
  for(int i=0; i<numItensVendidos; i++)
  {
    cout << "Digite o valor do item=";
    double x;
    cin >> x;
    valorItens[i] = x;
  }
}
```

Exemplo

TAD Venda (Destrutor)

- Nesse exemplo, como o construtor alocou memória de forma dinâmica, é preciso desalocar essa memória no destrutor.

```
Venda::~Venda()
{
  delete [] valorItens;
}

double Venda::calculaTotal()
{
  // exercicio
}

double Venda::itemMaisCaro()
{
  // exercicio
}
```

Exemplo

TAD Ponto

- Definir e implementar um TAD para representar um ponto no espaço bidimensional.
- O TAD deve considerar os seguintes dados:
  - coordenadas x e y.
- e as seguintes operações (além de construtor e destrutor):
  - modificar/acessar as coordenadas;
  - calcular a distância de um ponto a outro.

Exemplo

TAD Ponto

```
class Ponto
{
 public:
   Ponto();
   Ponto(float a, float b);
   ~Ponto();
   float getX();
   float getY();
   void setX(float xx);
   void setY(float yy);
   float distancia(Ponto op);

 private:
   float x, y;
};
```

Exemplo

TAD Ponto

```
Ponto::Ponto(float a, float b)
{
  x = a;
  y = b;
}

// etc ...

float Ponto::distancia(Ponto outroPt)
{
  float dx = x - outroPt.x;
  float dy = y - outroPt.y;
  float d = sqrt(dx*dx + dy*dy);
  return d;
}
```

TADs com outros TADs

- Também pode-se utilizar TAD como atributos de um TAD.
- Exemplo:

```
class MinhaClasseA
{
 public:
   // ...
 private:
   // ...
};

class MinhaClasseB
{
 public:
   // ...
 private:
   MinhaClasseA atributoX;
}
```

Exemplo

TAD Retangulo

- Definir e implementar um TAD para representar um retângulo definido por 2 pontos, o ponto inferior à esquerda e o superior à direita.
- Usar o TAD Ponto.
- O TAD deve considerar os seguintes dados:
  - ponto inferior esquerdo;
  - ponto superior direito;
- e as seguintes operações (além de construtor e destrutor):
  - calcular área;
  - verificar se um determinado ponto está dentro do retângulo.

TAD Retangulo

```
#include "Ponto.h"

class Retangulo
{
 public:
   Retangulo();
   Retangulo(float a, float b, float c, float d);
   Retangulo(Ponto pa, Ponto pb);
   ~Retangulo();

   float calculaArea();
   bool dentro(Ponto p);

 private:
   Ponto p1;
   Ponto p2;
};
```

TAD Retangulo

```
Retangulo::Retangulo()
{
  p1.setX(0); p1.setY(0);
  p2.setX(0); p2.setY(0);
}

Retangulo::Retangulo(float a, float b,
                     float c, float d)
{
  p1.setX(a); p1.setY(b);
  p2.setX(c); p2.setY(d);
}

Retangulo::Retangulo(Ponto pa, Ponto pb)
{
  p1.setX( pa.getX() );
  p1.setY( pa.getY() );
  p2.setX( pb.getX() );
  p2.setY( pb.getY() );
}
```

TAD Retangulo

- Implementação alternativa usando o construtor de `Ponto`

```
Retangulo::Retangulo()
{
  p1 = Ponto(0.0, 0.0);
  p2 = Ponto(0.0, 0.0);
}

Retangulo::Retangulo(float a, float b,
                     float c, float d)
{
  p1 = Ponto(a,b);
  p2 = Ponot(c,d);
}

Retangulo::Retangulo(Ponto pa, Ponto pb)
{
  p1 = Ponto( pa.getX(), pa.getY() );
  p2 = Ponto( pb.getX(), pb.getY() );
}
```

TAD Retangulo

```
float Retangulo::calculaArea()
{
  float largura = p2.getX() - p1.getX();
  float altura  = p2.getY() - p1.getY();
  return (altura*largura);
}

bool Retangulo::dentro(Ponto p)
{
  bool condA = (p1.getX() <= p.getX() &&
                p.getX()  <= p2.getX());
  bool condB = (p1.getY() <= p.getY() &&
                p.getY() <= p2.getY());
  if(condA && condB)
    return true;
  else
    return false;
}
```

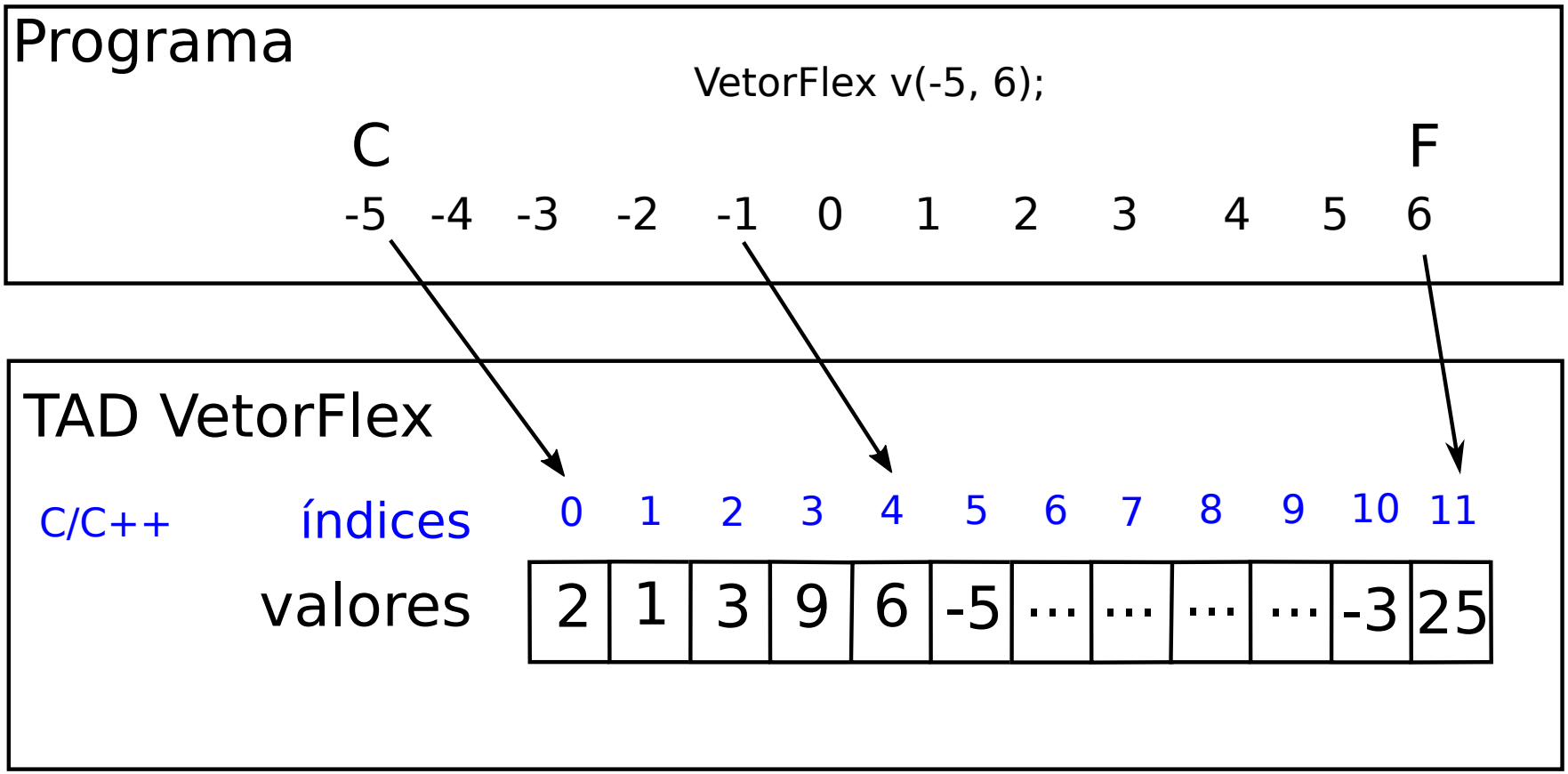
Aplicação com o TAD Retangulo

```
#include "Ponto.h"
#include "Retangulo.h"

int main() {
  float xx, yy;
  cin >> xx >> yy;

  Ponto p(xx,yy);
  Retangulo r(5,5,20,20);

  cout << "Area de r=" << r.calculaArea();
  cout << "Ponto p=" << p.getX() << " "
                     << p.getY() << endl;
  cout << "Ponto dentro do retangulo=";

  if ( r.dentro(p) ) cout << "Sim" << endl;
  else               cout << "Nao" << endl;

  return 0;
}
```

Vetor com índices flexíveis

- Sabendo que na linguagem C/C++, o índice de um vetor de tamanho n é um valor inteiro entre 0 e $n - 1$, pede-se:
- Desenvolver um **TAD** para possibilite criar vetores cujos índices podem assumir seus valores em intervalos inteiros e quaisquer, como por exemplo entre -10 e 45.
- Desenvolver uma **aplicação** para testar o TAD anterior, criando um vetor de 60 elementos reais numerados de –29 (limite inferior) a 30 (limite superior).
- Os valores limites (inferior e superior) do intervalo do índice devem ser definidos na aplicação, em **tempo de execução**. Assim, o construtor deve alocar memória dinamicamente para o vetor, de acordo com a definição desses limites.
- Verificar a validade do índice, quando necessário.

Vetor com índices flexíveis



Vetor com índices flexíveis

```
class VetorFlex
{
 private:
   int n;       // tamanho do vetor
   float *vet; // array que armazena n floats
   int c, f    // c: limite inferior do indice
               // f: limite superior do indice

   int detInd(int i); // operador privado

 public:
   VetorFlex(int a, int b);
   ~VetorFlex();
   float get(int i);
   void set(int i, float val);
};
```

Vetor com índices flexíveis

```
// construtor
VetorFlex::VetorFlex(int cc, int ff)
{
  c = cc;
  f = ff;
  n = f - c + 1;
  vet = new double[n];
}

// destrutor
VetorFlex::~VetorFlex()
{
  delete [] vet;
}
```

Vetor com índices flexíveis

- A função `detInd(int i)` é privada, isto é, só pode ser utilizada dentro da classe `VetorFlex`.
- A função `detInd(int i)` verifica a validade do índice de vet, isto é, se $c \leq i \leq f$.
- Se for válido, retorna o valor do índice de acordo com o padrão C/C++, isto é, o valor correspondente a índice dentro do intervalo de 0 a n–1.
- Senão (se for inválido), retorna -1.

Vetor com índices flexíveis

```
int VetorFlex::detInd(int i)
{
  if(c <= i && i <= f)
    return (i - c);
  else
    return -1;
};
```

- Até então só foram utilizados atributos (variáveis membro) como membros privados.
- Por que criar uma função privada em uma classe?
  - Para realizar alguma tarefa que só é de interesse da classe.
  - Nesse exemplo, o usuário do TAD `VetorFlex` não precisa saber se essa verificação é feita ou como ela é feita antes de acessar ou modificar um elemento do vetor.

Vetor com índices flexíveis

```
float VetorFlex::get(int i) {
  int k = detInd(i);
  if(k != -1)
    return vet[k];
  else {
    cout << "Indice invalido: get\n";
    exit(1);
  }
}

float VetorFlex::set(int i, double val) {
  int k = detInd(i);
  if(k != -1)
    vet[k] = val;
  else {
    cout << "Indice invalido: set\n";
    exit(1);
  }
}
```

Vetor com índices flexíveis

Aplicação

```
#include "VetorFlex.h"

int main()
{
  VetorFlex v(-29,30);
  for(int i=-29; i<=30; i++)
  {
    // valores no intervalo 1...60
    double val = i - cc + 1;
    v.set(i,val);
  }
  for(int i = cc; i <= ff; i++)
  {
    double val = v.get(i);
    cout << val << endl;
  }
  return 0;
}
```

Criando objetos

- Após a definição e implementação de uma classe, pode-se criar objetos no programa da mesma forma como qualquer outra variável de tipo primitivo, isto é, de forma **estática** ou **dinâmica**.
- Estática

```
Circulo c(0,0,5);
float a1 = c.calcArea();
c.setRaio(6);
float a2 = c.calcArea();
```

- Dinâmica

```
Circulo * c = new Circulo(0,0,5);
float a1 = c->calcArea();
c->setRaio(6);
float a2 = c->calcArea();
```

Criação e acesso a objetos

- Quando se tem uma variável (objeto) de uma classe usa-se o operador **.** para acessar seus membros.

```
Estudante obj();
obj.funcao();
```

- Quando se tem um **ponteiro** para um objeto de uma classe usa-se o operador $\rightarrow$ para acessar seus membros.

```
Estudante * obj = new Estudante();
obj->funcao();
```

Classes e Funções

- Assim como em C, na linguagem C++ todo argumento é passado **por valor** como padrão.
- A menos que se especifique o contrário, parâmetros de funções são inicializados com **cópias** dos argumentos.
- Quando uma função retorna, quem a chamou recebe uma **cópia** do valor retornado pela função.
- Para objetos mais complexos com muitos dados, fica claro que essa abordagem se torna extremamente cara.

Classes e Funções

- Exemplo

```
class Aluno {
 public:
   Aluno();
   ~Aluno();
   void imprime(); // imprime os dados
 private:
   string nomeUni, enderecoUni;
};
```

- Função (simples) que recebe e retorna um estudante:

```
void funcaoTeste(Aluno e) {
  e.imprime();
}
```

- Cópias:
  - Chamadas ao construtor e destrutor de `Aluno`.
  - Similar para os atributos `string` da classe.

Classes e Funções

- Para evitar esses incovenientes, basta utilizar **passagem de argumentos por referência**.
- Dessa forma não serão feitas cópias e, portanto, não haverá chamadas ao construtor/destrutor daquele objeto.

```
void funcaoTeste(Aluno * e)
{
  e->imprime();
}
```

Exercícios

1. Implemente um TAD para trabalhar com os **números racionais**. Implemente as operações aritméticas básicas: adição, subtração, multiplicação e divisão.
   Lembre-se:
   $$\mathbb{Q} = \{\frac{a}{b} \mid a \in \mathbb{Z};\ b \in \mathbb{Z}^*\}$$
   onde:
   - $\mathbb{Z}$ conjunto dos números inteiros;
   - $\mathbb{Z}^*$ conjunto dos números inteiros exceto o zero;

Exercícios

2. Desenvolver o TAD círculo, que deve ser representado pelo seu centro - do tipo `Ponto` - e pelo raio (um número real). O TAD deve ser capaz de realizar as seguintes operações:
   - Construtor que recebe o ponto representando o centro e o raio do círculo;
   - Imprimir as coordenadas do centro;
   - Imprimir o raio do centro;
   - Calcular a área do círculo;
   - Calcular o perímetro do círculo.

Exercícios

3. Implemente um TAD para trabalhar com vetores de números reais que serão indexados de 1 até N, onde N é o número de elementos do vetor. As seguintes operações devem ser implementadas:
   - adição de vetores;
   - multiplicação por um valor escalar (real);
   - produto escalar.