

Notas de Aula de Algoritmos e Programação de Computadores

FLÁVIO KEIDI MIYAZAWA

com a colaboração de

TOMASZ KOWALTOWSKI

Instituto de Computação - UNICAMP

Versão 2000.1

Estas notas de aula não devem ser usadas como única fonte de estudo. O aluno deve ler outros livros disponíveis na literatura.

Nenhuma parte destas notas pode ser reproduzida, qualquer que seja a forma ou o meio, sem a permissão dos autores.

Os autores concedem a permissão explícita para a utilização e reprodução deste material no contexto do ensino de disciplinas regulares dos cursos de graduação sob a responsabilidade do Instituto de Computação da UNICAMP.

© Copyright 2000

Instituto de Computação
UNICAMP
Caixa Postal 6176
13083-970 Campinas-SP
{fkm,tomasz}@ic.unicamp.br

9 Variáveis Compostas Heterogêneas - Registros

A linguagem Pascal nos permite especificar objetos formados por diversos dados associados a eles, possivelmente de tipos diferentes. Assim, em apenas um objeto (variável) poderemos ter vários outros dados como por exemplo do tipo string, integer, real, Vamos chamar este tipo de objeto por *registro*. Poderemos usar dois tipos de registros: um onde os diferentes tipos de dados estão armazenados em memória distinta e outro quando parte dos dados estão armazenados em um mesmo endereço de memória.

9.1 Registros Fixos

Para se especificar um registro onde cada campo está definido em uma determinada memória (sem interseção com os demais campos), usamos a seguinte sintaxe (que pode ser usada para definir tanto variáveis como tipos):

```
record
    Lista_de_Identificadores_do_Tipo_1 : Tipo_1;
    Lista_de_Identificadores_do_Tipo_2 : Tipo_2;
    :
    Lista_de_Identificadores_do_Tipo_K : Tipo_K;
end
```

Na sintaxe acima, cada identificador é chamado de *Campo* do registro. Além disso, cada um dos tipos (Tipo_i) também pode ser a especificação de outro registro.

Para acessar um campo chamado **Campo1** de um registro chamado **Reg1**, usamos a seguinte sintaxe:

Reg1.Campo1

Caso Campo1 seja também um registro e contenha o campo **SubCampo11**, acessamos este último da seguinte maneira:

Reg1.Campo1.SubCampo11

Números Complexos

Como tipos numéricos a linguagem Pascal oferece os tipos **integer** e **real**, mas a maioria dos compiladores Pascal não oferece um tipo especial para tratar números complexos. Como um número complexo é dividido em duas partes na forma $n = a + b \cdot i$, onde a e b são números reais, precisamos de um tipo que contemple estes dois números em apenas um tipo. Com isso, podemos definir um tipo chamado *complexo* que contém as duas partes de um número complexo. No seguinte quadro, definimos um tipo complexo usando **record** e apresentamos funções para fazer soma, subtração e multiplicação de números complexos:

É importante observar que a sintaxe da linguagem Pascal não permite acessar os campos da própria variável de retorno de função. Devemos usar uma variável para receber um valor de retorno e em seguida atribuir esta variável para o retorno de função, sendo este o motivo de usarmos a variável z nas rotinas *SumComplex*, *SubComplex* e *MulComplex*.

```

program ProgComplex;
type complex = record
    a,b : real; {Número na forma (a+b.i) }
end;
procedure LeComplex(var x : complex); { Le um número complexo }
begin
    writeln('Entre com um número complexo (a+b.i) entrando com a e b: ');
    readln(x.a,x.b);
end; { LeComplex }
procedure ImpComplex(x : complex); { Imprime um número complexo }
begin
    writeln(' ( ',x.a:5:2,' + ',x.b:5:2,' i ) ');
end; { ImpComplex }
function SomComplex(x,y:complex):complex; { Retorna a soma de dois números complexos }
var z : complex;
begin
    z.a := x.a+y.a;   z.b:=x.b+y.b;
    SomComplex := z;
end; { SomComplex }
function SubComplex(x,y:complex):complex; { Retorna a subtração de dois números complexos }
var z : complex;
begin
    z.a := x.a-y.a;   z.b:=x.b-y.b;
    SubComplex := z;
end; { SubComplex }
function MulComplex(x,y:complex):complex; { Retorna a multiplicação de dois números complexos }
var z : complex;
begin
    z.a := x.a*y.a-x.b*y.b;   z.b:=x.a*y.b+x.b*y.a;
    MulComplex := z;
end; { MulComplex }
var x,y      : complex;
begin
    LeComplex(x); LeComplex(y);
    write('A soma dos números complexos lidos é '); ImpComplex(SomComplex(x,y));
    write('A subtração dos números complexos lidos é '); ImpComplex(SubComplex(x,y));
    write('A multiplicação dos números complexos lidos é '); ImpComplex(MulComplex(x,y));
end.

```

Exercício 9.1 Um número racional é definido por duas partes inteiras, o numerador e o denominador. Defina um tipo chamado **racional** usando a estrutura **record** para contemplar estas duas partes. Além disso, faça funções para ler, somar, subtrair, dividir e simplificar números racionais. Por simplificar, queremos dizer que o número racional $\frac{a}{b}$ tem seu numerador e denominador divididos pelo máximo divisor comum entre eles (veja programa do exemplo 7.10).

Cadastro de Alunos

Suponha que você tenha um cadastro de alunos onde cada aluno contém as seguintes características: Nome, Data de Nascimento (dia, mês e ano), RG, Sexo, Endereço (Rua, Cidade, Estado, CEP), RA (Registro do Aluno) e CR (Coeficiente de Rendimento: número real no intervalo $[0, 1]$).

Note que um aluno deve ter as informações da data de nascimento que podem ser divididas em três partes (dia, mês e ano). Sendo assim, definiremos um tipo chamado TipoData que representará uma data. Note que o mesmo ocorre com endereço assim, usaremos um tipo chamado TipoEndereco que representará um endereço. Obs.: Não necessariamente precisávamos definir tipos separados para data e endereço, mas sempre que temos informações com um tipo independente e com certa “vida própria”, é razoável se definir um tipo particular a ele.

O seguinte quadro apresenta a declaração do tipo aluno e os demais tipos necessários para defini-lo.

```
type
TipoNome      = string[50];
TipoRG        = string[10];
TipoDia       = 1..31;
TipoMes       = 1..12;
TipoAno       = integer;
TipoRua       = string[50];
TipoEstado    = string[2];
TipoCep       = string[9];
TipoRA        = string[6];
TipoCR        = real;
TipoData      = record
    Dia : TipoDia;
    Mes : TipoMes;
    Ano : TipoAno;
end;
TipoEndereco  = record
    Rua   : TipoRua;
    Cidade : TipoCidade;
    Estado : TipoEstado;
    CEP   : TipoCep;
end;
TipoAluno     = record
    Nome      : TipoNome;
    RG        : TipoRG;
    DataNascimento : TipoData;
    Endereco  : TipoEndereco;
    RA        : TipoRA;
    CR        : TipoCR;
end;
```

Usando as declarações dos tipos acima, exemplificamos, nos dois quadros seguinte, o acesso e uso destes tipos com variáveis. Os dois programas apresentados são equivalentes.

<pre> { Supondo feita as declarações } { dos tipos do quadro acima } var Aluno : TipoAluno; Data : TipoData; Endereco : TipoEndereco; begin Aluno.Nome := 'Fulano de Tal'; Aluno.RG := '9999999999'; Data.Dia := 1; Data.Mes := 1; Data.Ano := 2000; Aluno.DataNascimento := Data; Endereco.Rua := 'R. Xxx Yyy, 999'; Endereco.Cidade := 'Campinas'; Endereco.Estado := 'SP'; Aluno.Endereco := Endereco; Aluno.RA := '999999'; Aluno.CR := 0.99; Writeln('O aluno ', Aluno.Nome, ' mora em ', Aluno.Endereco.Cidade, '- ', Aluno.Endereco.Estado, '. '); end. </pre>	<pre> { Supondo feita as declarações } { dos tipos do quadro acima } var Aluno : TipoAluno; begin Aluno.Nome := 'Fulano de Tal'; Aluno.RG := '9999999999'; Aluno.DataNascimento.Dia := 1; Aluno.DataNascimento.Mes := 1; Aluno.DataNascimento.Ano := 2000; Aluno.Endereco.Rua := 'R. Xxx Yyy, 999'; Aluno.Endereco.Cidade := 'Campinas'; Aluno.Endereco.Estado := 'SP'; Aluno.RA := '999999'; Aluno.CR := 0.99; Writeln('O aluno ', Aluno.Nome, ' mora em ', Aluno.Endereco.Cidade, '- ', Aluno.Endereco.Estado, '. '); end. </pre>
--	---

Exercício 9.2 Usando as declarações de tipo apresentadas no quadro da página 9.1, faça um cadastro de alunos em um vetor com 100 posições, cada posição do tipo *TipoAluno*, i.e.,

type *TipoCadastro* = **array**[1..100] **of** *TipoAluno*;

O programa deve manipular este cadastro com as seguintes opções:

1. Iniciar cadastro vazio (inicialmente sem elementos).
2. Inserir um novo aluno no cadastro (se o cadastro estiver cheio, avise que não há memória disponível).
3. Ordenar o cadastro por nome em ordem alfabética.
4. Ordenar o cadastro por CR, maiores primeiro.
5. Ler o valor de um RA e imprimir os dados do aluno no cadastro com mesmo RA.
6. Imprimir o cadastro na ordem atual.
7. Sair do programa.

9.2 Registros Variantes

A linguagem Pascal permite que possamos definir um registro com uma parte variante. Nesta parte variante, podemos ter diferentes conjuntos de campos. A idéia aqui envolve a situação onde o valor da característica de um objeto pode implicar em conjuntos de características diferentes.

Por exemplo, suponha que temos registros com informações dos móveis de uma casa. Vamos supor que temos apenas três tipos de móveis: (*Armário, Mesa, Sofá*). Todos os móveis considerados têm atributos em comum, que são: (*cor, material, comprimento, largura, altura*). Mas se o móvel for um armário, então ele tem um atributo particular que é o *número de gavetas*. Se o móvel for uma mesa, ela tem outro atributo que é a *forma*, se *circular* ou *retangular* e o *número de cadeiras* que usa. Se o móvel for um sofá, então ele tem um atributo especial que é a *quantidade de lugares* para se sentar. Além disso, um atributo particular de um tipo de móvel não faz sentido para outro tipo de móvel.

Se formos usar o tipo de registro fixo, então deveríamos colocar todos os atributos no registro que especifica o móvel, cada um usando uma parte da memória. Mas note que se o móvel for um Armário, então o campo que indica a quantidade de cadeiras de uma mesa é um desperdício de memória e não será usado. A idéia de usar registros variantes é que cada conjunto de atributos particulares esteja começando na mesma posição de memória.

Uma restrição da linguagem Pascal é que podemos ter apenas uma parte que é definida como parte variante. Sua sintaxe é definida da seguinte maneira:

```
record
    Lista_de_Identificadores_do_Tipo_1 : Tipo_1;
        :
    Lista_de_Identificadores_do_Tipo_K : Tipo_K;
case [CampoSeletor:] Tipo_Selecionador of
    Escalar_1 : (Lista_de_Declaração_de_Campos_1);
        :
    Escalar_P : (Lista_de_Declaração_de_Campos_P);
end;
```

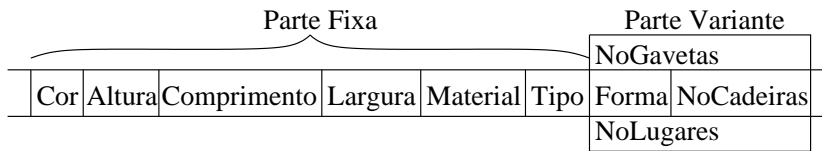
A lista de identificadores dos tipos *Tipo_1...Tipo_K* formam a parte fixa do registro. A parte variante começa depois da palavra **case**. O *CampoSeletor* é opcional. A idéia é que caso o *CampoSeletor* seja igual a *Escalar_1*, então os campos válidos serão os que estão na *Lista_de_Declaração_de_Campos_1*. Caso o *CampoSeletor* seja igual a *Escalar_2*, então os campos válidos serão os que estão na *Lista_de_Declaração_de_Campos_2*, e assim por diante. O tipo do *CampoSeletor* deve ser necessariamente um escalar. Note também que temos apenas uma palavra **end**, i.e., a palavra **end** termina a definição de todo o registro.

No programa do quadro seguinte apresentamos as declarações e algumas atribuições usando registro variante no exemplo dos móveis, apresentado acima.

```
program ProgMobilia;
type
    TipoMaterial: (Madeira, Metal, Couro, Sintetico);
    TipoCor:      (Branco, Preto, Azul, Amarelo, Vermelho, Verde, Cinza);
    TipoMovel = record
        Cor       : TipoCor;
        Altura,Comprimento,Largura : real;
        Material  : TipoMaterial;
        case Tipo : (Mesa,Sofa,Armario) of
            Armario : (NoGavetas:integer);
            Mesa    : (Forma:(Circular,Retangular); NoCadeiras:integer);
            Sofa    : (NoLugares:integer);
        end;
var M1,M2 : TipoMovel;
begin
    M1.Tipo := Sofa; {Definindo o atributo selecionador}
    M1.Cor := Amarelo; M1.Material := Couro;
    M1.Largura:=1.0; M1.Altura:=0.9; M1.Comprimento :=3.0;
    M1.NoLugares := 3; {Atributo particular de sofá}

    M2.Tipo := Mesa; {Definindo o atributo selecionador}
    M2.Cor := Azul; M2.Material := Madeira;
    M2.Largura:=1.5; M2.Altura:=1.2; M2.Comprimento :=2.0;
    M2.NoCadeiras := 6; {Atributo particular de mesa}
    M2.Forma := Retangular; {Atributo particular de mesa}
    {...}
end.
```

O acesso aos campos do registro é feita da mesma forma como nos registros fixos. A seguinte figura apresenta a configuração da memória para o registro do tipo *TipoMovel* apresentado no quadro acima.



Mesmo que um campo selecionador esteja indicado com um determinado tipo, todos os campos da parte variante do registro são passíveis de serem usados. I.e., a responsabilidade de manter a coerência dos campos da parte variante é do programador.

Outro uso interessante deste tipo de estrutura é a não necessidade do campo selecionador. Neste caso, podemos ter campos que nos permitem acessar uma mesma memória de diferentes maneiras, sem precisar gastar memória com um campo selecionador.

```

type    Tipo32bits = record
            B1,B2,B3,B4 : byte;
        end;
    TipoConjuntoElementos = record
        case integer of
            1      : VetorByte: array[1..4*1000] of byte;
            2      : Vetor32Bits: array[1..1000] of Tipo32Bits;
        end;

```

Com o tipo *TipoConjuntoElementos* podemos estar usando no programa o campo *VetorByte*, mas em situações onde precisamos copiar um vetor deste tipo para outro será em muitos computadores mais rápido fazer 1000 atribuições do tipo *Tipo32Bits* do que 4000 atribuições do tipo *byte*. Nestes casos, usaremos o campo *Vetor32Bits* em vez do vetor *VetorByte*, já que o resultado será o mesmo. Isto se deve a arquitetura do computador, que caso seja de 32 bits, pode fazer transferência de 32 bits por vez.

Exemplo 9.1 *Uma universidade mantém registros de todos os seus alunos, que inclui os alunos de graduação e de pós-graduação. Muitos tipos de dados são iguais para estas duas classes de alunos, mas alguns dados são diferentes de um para outro. Por exemplo: Todos os alunos possuem nome, data de nascimento, RG, e RA. Entretanto existem algumas diferenças para cada tipo de aluno. As disciplinas feitas pelos alunos de graduação são numéricas. Para as disciplinas de pós-graduação temos conceitos. Além disso, alunos de pós-graduação fazem tese, possuem orientador e fazem exame de qualificação. Assim, podemos ter a seguinte definição de tipo para um aluno desta universidade:*

```

type
    NivelAluno    = (Grad, Pos);
    TipoData      = record
        dia : 1..31; mes:1..12; ano:1900..2100;
    end;
    TipoAlunoUni  = record
        nome      : string[50];
        RG        : string[15];
        DataNascimento : TipoData;
        {... demais campos ...}
    case nivel: NivelAluno of
        Grad :
            (Notas: ListaNotas);
        Pos :
            (Conceitos: ListaConceitos;
             ExamQualif, Defesa : TipoData;
             Titulo: string[100],
             Orientador: string[50]);
    end;

```

A seguir apresentamos um esboço de programa que usa esta estrutura para ler os dados de um aluno desta universidade.

```

var Aluno : TipoAlunoUni;
begin
  { ... }
  write('Entre com o nome do aluno: '); readln(Aluno.nome);
  LeDataNascimento(Aluno.DataNascimento);
  { ... leitura dos demais dados iguais tanto para aluno de graduação como de pos ... }
  case Aluno.Nivel of
    Grad : begin
      LeNotasAlunoGraduacao(Aluno.NotasGrad);
    end;
    Pos : begin
      LeConceitosAlunoPos(Aluno.Notas);
      LeDataQualificacao(Aluno.ExamQualif);
      LeDataDefesa(Aluno.Defesa);
      writeln('Entre com o título da tese/dissertação: '); readln(Aluno.Titulo);
      writeln('Entre com o nome do orientador: '); readln(Aluno.Orientador);
    end;
  end; { case }
  { ... }
end.

```

9.3 Comando With

A linguagem Pascal oferece uma maneira fácil para acessar os campos de um registro. Isto é feito usando-se o comando **with** que permite referenciar, no escopo do comando, os campos dos registros sem referência aos identificadores dos registros. A sintaxe do comando with é a seguinte:

```
with Lista_de_Variáveis_de_Registro do Comando_ou_Bloco_de_Comandos;
```

Obs.: na lista de variáveis não deve haver variáveis de mesmo tipo, ou variáveis que tenham campos com o mesmo nome.

Exemplo 9.2 Vamos escrever a função *MulComplex* apresentada no quadro anterior usando o comando *with*.

```

function MulComplex(x,y:complex):complex; { Retorna a multiplicação de dois números complexos }
var z : complex;
begin
  with z do begin
    a := x.a*y.a-x.b*y.b;
    b := x.a*y.b+x.b*y.a;
  end;
  MulComplex := z;
end; { MulComplex }

```

9.4 Exercícios

1. Suponha que você tenha um cadastro de alunos onde cada aluno contém as seguintes características: Nome, Data de Nascimento (dia, mês e ano), RG, Sexo, Endereço (Rua, Cidade, Estado, CEP), RA (Registro do Aluno) e CR (Coeficiente de Rendimento: número real no intervalo $[0, 1]$). O tipo para um aluno, que chamaremos de *TipoAluno* é especificado na linguagem Pascal no quadro seguinte:

type

```
TipoNome=string[50]; TipoRG=string[10];
TipoDia=1..31; TipoMes=1..12; TipoAno=integer;
TipoRua=string[50]; TipoEstado=string[2]; TipoCep=string[9];
TipoRA=string[6]; TipoCR=real;
TipoData = record
    Dia : TipoDia; Mes:TipoMes; Ano:TipoAno;
end;
TipoEndereco = record
    Rua : TipoRua; Cidade:TipoCidade; Estado:TipoEstado; CEP:TipoCep;
end;
TipoAluno = record
    Nome : TipoNome; RG:TipoRG; DataNascimento:TipoData;
    Endereco : TipoEndereco; RA: TipoRA; CR:TipoCR;
end;
```

Faça um cadastro de alunos em um vetor definido com 100 posições posições, cada posição do tipo *TipoAluno*, i.e.,

type *TipoCadastro* = **array**[1..100] **of** *TipoAluno*;

O programa deve manipular este cadastro com os seguintes comandos:

- Iniciar cadastro vazio (inicialmente sem elementos).
- Inserir um novo aluno no cadastro (se o cadastro estiver cheio, avise que não há memória disponível).
- Ordenar o cadastro por nome em ordem alfabética.
- Ordenar o cadastro por idade, mais velhos primeiro.
- Ordenar o cadastro por RA, ordem crescente.
- Ordenar o cadastro por CR, maiores primeiro.
- Imprimir o cadastro na ordem atual.
- Sair do programa.

Um item que o aluno pode fazer, opcionalmente (não precisa entregar com este item) é fazer uma rotina que insere n alunos aleatórios (nome aleatório, idade aleatório, ...).

- Declare um tipo chamado *tiporeg*, definido como um tipo de registro contendo os seguintes campos: *Nome*, *RG*, *Salario*, *Idade*, *Sexo*, *DataNascimento*; onde *Nome* e *RG* são strings, *Salario* é real, *Idade* é inteiro, *sexo* é char e *DataNascimento* é um registro contendo três inteiros, dia, mes e ano. Declare um tipo de registro chamado *TipoCadastro* que contém dois campos: Um campo, *Funcionario*, contendo um vetor com 100 posições do tipo *tiporeg* e outro campo inteiro, *Quant*, que indica a quantidade de funcionários no cadastro.

Todos os exercícios seguintes fazem uso do tipo *TipoCadastro*.

- Faça uma rotina, *InicializaCadastro*, que inicializa uma variável do tipo *TipoCadastro*. A rotina atribui a quantidade de funcionários como zero.
- Faça um procedimento, *LeFuncionarios*, com parâmetro uma variável do tipo *TipoCadastro*. A rotina deve ler os dados de vários funcionários e colocar no vetor do cadastro, atualizando a quantidade de elementos não nulos. Caso o nome de um funcionário seja vazio, a rotina deve parar de ler novos funcionários. A rotina deve retornar com o cadastro atualizado. Lembre que o cadastro não suporta mais funcionários que os definidos no vetor de funcionários.
- Faça uma rotina, chamada *ListaFuncionários*, que imprime os dados de todos os funcionários.
- Faça duas rotinas para ordenar os funcionários no cadastro. Uma que ordena pelo nome, *OrdenaNome*, e outra que ordena pelo salário, *OrdenaSalario*.
- Faça uma rotina, *SalarioIntervalo*, que tem como parâmetros: um parâmetro do tipo *TipoCadastro* e dois valores reais v_1 e v_2 , $v_1 \leq v_2$. A rotina lista os funcionários com salário entre v_1 e v_2 . Depois de imprimir os funcionários, imprime a média dos salários dos funcionários listados.
- Faça uma rotina que dado um cadastro, imprime o nome do funcionário e o imposto que é retido na fonte. Um funcionário que recebe até R\$1000,00 é isento de imposto. Para quem recebe mais que R\$1000,00 e até R\$2000,00 tem 10% do salário retido na fonte. Para quem recebe mais que R\$2000,00 e até R\$3500,00 tem 15% do salário retido na fonte. Para quem recebe mais que R\$3500,00 tem 25% do salário retido na fonte.

9. Faça uma função, BuscaNome, que tem como entrada o cadastro e mais um parâmetro que é o um nome de um funcionário. O procedimento deve retornar um registro (tipo tiporeg) contendo todas as informações do funcionário que tem o mesmo nome. Caso a função não encontre um elemento no vetor contendo o mesmo nome que o dado como parâmetro, o registro deve ser retornado com nome igual a vazio.
10. Faça uma rotina, AtualizaSalario, que tem como parâmetros o cadastro de funcionários. A rotina deve ler do teclado o RG do funcionário a atualizar. Em seguida a rotina lê o novo salário do funcionário. Por fim, a rotina atualiza no cadastro o salário do funcionário com o RG especificado.
11. Faça uma função, chamada ListaMaraja, que tem como parâmetro o cadastro e devolve um registro contendo os dados de um funcionário que tem o maior salário.
12. Faça uma rotina que tem como parâmetros o cadastro e o RG de um funcionário. A rotina deve remover do cadastro o funcionário que contém o RG especificado. Lembre-se que os elementos não nulos no vetor do cadastro devem estar contíguos. Além disso, caso um elemento seja removido, a variável que indica a quantidade de elementos deve ser decrementada de uma unidade. Caso não exista nenhum elemento no vetor com o RG fornecido, a rotina não modifica nem os dados do vetor nem sua quantidade.
13. Faça uma rotina, ListaAniversarioSexo, que tem como entrada o cadastro, três inteiros: dia, mes e ano, que correspondem a uma data e um caracter (sexo) com valor 'F' ou 'M'. A rotina deve imprimir o nome dos funcionários que nasceram nesta data e com sexo igual ao definido pelo parâmetro.