Universidade Federal do Amazonas
Faculdade de Tecnologia
Departamento de Eletrônica e Computação

Programação de Pequenos Sistemas

Lucas Cordeiro
lucascordeiro@ufam.edu.br

Programação de Pequenos Sistemas (1)

- Objetivo: lembrá-los de como ler e escrever programas em ADA95, Java e ANSI-C
- Ada é importante por causa do seu crescente uso em **sistemas críticos de segurança**
- Java tem se tornado a linguagem padrão para programar **aplicações baseadas em internet**
- C (e seu derivativo C++) é talvez a **linguagem de programação mais popular** usada atualmente
- As práticas desta disciplinas serão todas implementadas em Ada 95, C e Java

Programação de Pequenos Sistemas (2)

- Para a prova, vocês deverão ser capazes de esboçar a **solução nas três linguagens**. Porém, eu não espero que os programas estejam sintaticamente corretos!

- Espero uma **alta afinidade** com Ada 95, Java e ANSI-C

Exemplo do Fortran (Análise Léxica)

DO 20 I = 1, 100

ler o arquivo fonte em busca de tokens (agrupar caracteres com um papel bem definido)

Em um certo projeto, o programador escreveu

DO 20 I = 1. 100

O compilador interpretou como um comando de atribuição e ignorou os espaços

DO20I = 1.100

Variáveis não precisam ser declaradas em Fortran, e aquelas que começãm com D são assumidas para ser do tipo real. 1.100 é um literal real!

Uma Visão Geral de Ada

Um programa em ADA consiste de um ou mais unidades de programa:

- um sub-programa (procedimento ou função) — pode ser genérico
- um pacote (possivelmente genérico) — usado para encapsulamento ou modularidade
- uma tarefa — usado para concorrência
- Um unidade protegida — mecanismo de sincronização orientado a dado

Unidades de biblioteca: pacote, sub-programa

Sub-unidades: sub-programa, pacote, tarefa, unidade protegida

Bloco (1)

Ada é uma linguagem estruturada por bloco

```
declare
   -- definição de tipos, objetos,
   -- sub-programas etc.
begin
   -- seqüência de comandos
exception
   -- tratadores de exceção
end;
```

Um bloco pode ser colocado em um programa em qualquer lugar em que um comando pode ser escrito

Bloco (2)

Uma variável interia *Temp* é introduzida para trocar os valores contidos pelos dois inteiros *A* e *B*

```
declare
   Temp: Integer := A; -- initial value given to
                       -- a temporary variable.
begin
   A := B;             -- := é o operador de atribuição
   B := Temp;
end;                   -- sem a parte da exceção
```

Um bloco pode ser colocado em um programa em qualquer lugar em que um comando pode ser escrito

Observações sobre Bloco

- Objetos usados em um bloco podem ser somente usados dentro daquele bloco **(escopo)**
- **Tratadores de exceções** podem ser usados para capturar erros oriundos da execução da seqüência de comandos (eles podem também ser omitidos)
- Ada é fortemente classificada em termos de tipo, isto é, atribuições e expressões devem envolver **objetos do mesmo tipo**

A Linguagem C (1)

- É uma linguagem **seqüêncial**
- Unidades de estruturação principal são as **funções** (embora arquivos possam ser usados para ajudar a separar a compilação)
- Bloco estruturado (chamado de comandos compostos) – delimitado por um { e um }

```
{
  < declarative part >

  < sequence of statement >
}
```

- Parte declarativa não pode conter funções
- Seqüência de comandos pode conter comando composto

Exemplo em C (1)

```
{
  int temp = A; /* declaração e
                   inicialização*/
 /* note que em C e Java o nome do tipo
  aparece primeiro enquanto que em ADA aparece
  depois do nome da variável */
  A = B;
  B = Temp;
}
```

Ambos ADA e C permitem tipos de inteiro básico ser sinalizados e não sinalizados

Exemplo de C (2)

```
int largest(vector X, int len)
{
  int max = 0;
  int i;

  for (i = 0; i < len; i++) {
    // bounds start at 0
    if(X[i] > max) max = X[i];
  }
  return max;
}
```

Comando de atribuição é =

Igualdade é = =

Não tão seguro como o ADA em termos de tipo (um tipo ***int*** pode ser atribuido a um ***short*** sem conversão de tipo explícito). Porém, as inseguranças em C são bem conhecidas.

Uma Visão Geral de Java

- Uma linguagem baseada em **classe**
- Programação de pequenos componentes é similar ao C porém sem valores de ponteiro explícito
- Tem mais segurança em termos de tipo de dados do que em C
- Como em Ada, Java pode ter **tratadores de exceção** no fim de um bloco (mas somente se o bloco tiver um bloco *try*)
- Métodos podem ser declarados no contexto de uma classe

Um Exemplo de Java

```
class SomeArrayManipulationClass
{
  public int largest(vector X)
  {
    int max = 0;
    int i;

    for (i = 0; i < X.length; i++) {
      // bounds start at 0
      // length is an instance variable of array objects
      if(X[i] > max) max = X[i];
    }
    return max;
  }
}
```

Todos os arrays são objetos

Java e Tipos de Referências

- Todos os objetos em Java são representados como **valores de referência**
- A comparação de dois objetos comparará a referência deles e não os valores deles:

```
Node Ref1 = new Node();
Node Ref2 = new Node();
. . .
if(Ref1 == Ref2) { . . . }
```

  irá comparar a localização dos objetos e não os valores dele; é necessário implementar um método `compareTo`

- Situação similar ocorre com a atribuição de objetos; é necessário fornecer um método `clone`

Tipos Discretos

Ada	Java	C
Integer	int	int
	short	short
	long	long
	byte	
Boolean	boolean	
Character		char
Wide_Character	char	wchar_t
Enumeration types	*enum*	*typedef enum*

Enumeração em C

```
/* C */
{
  typedef enum {xplane, yplane, zplane } dimension;
  dimension line, force;
  line = xplane;
  force = line + 1;
}
```

Qual é o valor da variável **force**?

force tem agora o valor yplane, pois o compilador gera literais de interno para *xplane=0*, *yplane=1* e *zplane=2*

Enumeração em Ada

```
-- Ada
type Dimension is (Xplane,Yplane,Zplane);
type Map is (Xplane,Yplane);
Line, Force : Dimension;
Grid : Map;
begin
  Line  := Xplane;
  Force := Dimension'Succ(Xplane);
-- Force tem agora o valor de Yplane irrespectivo da
técnica de implementação
  Grid := Yplane;  -- o nome 'Yplane' é ambíguo pois
                   -- Grid é do tipo 'map'
  Grid := Line;    -- ilegal? – confronto de tipo
end
```

Sub-intervalo ou Sub-tipo

- Ada suporta o uso de sub-intervalo ou sub-tipo para restringir os valores de um objeto

```
-- Ada
subtype Surface is Dimension range Xplane .. Yplane;
```

- Todos os tipos em Ada e C podem ser duplicados através da definição de um tipo para ser uma nova versão de um tipo anterior:

```
-- in Ada
type New_Int is new Integer;
type Projection is new Dimension range Xplane..Yplane;

/* in C*/
typedef int newint;
typedef dimension projection;
```

Conversão de Tipos

- Enquanto em ADA objetos de um **tipo e seus sub-tipos podem ser misturados** (em expressões), objetos de um **tipo e um tipo derivado não podem**:

```
-- Ada
D : Dimension;
S : Surface;
P : Projection;
begin
  D := S  -- É permitido?   sim
  S := D; -- É permitido mas pode causar erro em tempo de
          -- execução se D tiver o valor 'Zplane'
  P := D; -- ilegal?   – confronto de tipo
  P := Projection(D); -- legal, conversão de tipo
                      -- explícito
end;
```

Em C, **typedefs** não fornecem este nível de segurança

Números Reais (1)

- Várias aplicações de tempo real (processamento de sinal, controle de processo e simulação) requerem computação numérica
- Existem dois maneiras de representar número real: **ponto flutuante** e **inteiro com escala**
- Um número de ponto flutuante é representado por três valores:
  - **m**antissa M; **e**xpoente E; e uma **r**aiz R
  - tem a forma M x R^E onde R geralmente tem o valor 2
  - como M é limitado em comprimento, a representação tem **precisão limitada**
  - **error relativo** é a diferença entre o número de ponto flutuante e o número real

Números Reais (2)

- Inteiro com escala é usado para computação numérica exata
  - é o produto de um inteiro e uma escala (qualquer valor pode ser representado)
  - a escala deve ser conhecida em tempo de compilação (se o valor da escala não for conhecido, representação por ponto flutuante deve ser usado)
  - nem todos os números no domínio matemático podem ser representados exatamente (1/3 não pode ser visto como inteiro decimal com escala finita)
  - **erro absoluto** é a diferença entre um inteiro de escala e seu valor real
- Inteiros de escala são mais difíceis de usar especialmente se expressões com diferentes escalas precisam ser avaliadas

Números Reais em Ada (1)

- Além do tipo pré-definido do tipo *Float*, Ada fornece as facilidades de criar números de ponto flutuante (com diferente precisão) e números de ponto fixo

```
type New_Float is digits 10 range -1.0E18..1.0E18
```

- Um sub-tipo deste tipo pode restringir o **intervalo** ou a **precisão**

```
subtype Crude_Float is New_Float digits 2;
subtype Pos_New_Float is New_Float range 0.0..1000.0
```

Números Reais em Ada (2)

- A construção de um tipo de **ponto fixo** solicita informações de **intervalo** e um limite de **erro absoluto** chamado *delta*:

```
type Scaled_Int is delta 0.05 range -100.00..100.00
```

- Para representar todos os inteiros decimais (-100.00,-99.95,...,99.95,100.00) são necessários 13 bits:
  - 5 bits para a parte fracionária (2^{-5} é a potência de 2 mais próximo do delta 1/20 = 0.05)
  - 8 bits para a parte inteira (incluindo o bit de sinal)

    `sbbbbbbb.fffff`
  - **s** denota o bit do sinal, **b** denota o bit do inteiro enquanto **f** denota o bit da fração (pode facilmente ser implementado em uma máquina de 16 bits)

Tipos de Dados Estruturados (Vetores)

- Ada e C suportam vetores e registros enquanto que Java suporta somente vetor

```
/* C */

#define MAX 10 /* define MAX to be 10 */
typedef float reading_t[MAX]; /* index is 0 .. max-1 */
typedef short int switches_t[MAX][MAX]; /*no boolean in C*/
reading_t reading;
switches_t switches;

//Java
static final int max = 10 // a constant
float reading[] = new float[max]; //index is 0..max-1
boolean switches[] [] = new boolean[max][max];
```

Tipos de Dados Estruturados (Vetores)

```
-- Ada
Max: Constant Integer := 10;
type Reading_T is array(0..Max-1) of Float
Size: Const Integer := Max – 1;
type Switches_T is array(0 .. Size, 0.. Size ) of Boolean
Reading: Reading_T;
Switches: Switches_T;
```

- Note que Java e C usam colchetes para declarar vetores enquanto que Ada usa parênteses

- Os vetores em Ada podem começar em qualquer índice equanto que Java e C sempre começam em 0

Tipos de Dados Estruturados (Registro)

```
-- Ada
type Day_T is new Integer range 1 .. 31;
type Month_T is new Integer range 1 .. 12;
type Year_T is new Integer range 1900 .. 2050;
type Date_T is
  record
    Day : Day_T := 1;
    Month: Month_T := 1;
    Year: Year_T;
  end record;

/* C */
typedef short int day_t;
typedef short int month_t;
Typedef int year_t;
```

Tipos de Dados Estruturados (Registro)

```
struct date_t {
  day_t day;
  month_t month;
  year_t year; };
/* como data_t não tem sido introduzido por um typedef, */
/* seu nome é ‘struct date_t’ */

typedef struct {
  day_t day;
  month_t month;
  year_t year; } date2_t;
/* aqui o nome de tipo ‘date2_t’ pode ser usado */
```

Tipos de Dados Estruturados (Registro)

- Comandos de atribuição de registros

```
-- Ada
D: Data
begin
  D.Year := 1989;       -- notação de ponto
  -- D tem agora o valor 1-1-1989 devido a initialização
  D:= (3, 1, 1953);     -- atribuição completa
  D:= (Year => 1974, Day => 4, Month => 7);
  -- atribuição completa usando a notação de nome
  ...
end;

/* C */
struct date_t D = {1,1,1};
```

Tipos de Dados Estruturados (Registro)

- Java não possue a estrutura de registro, mas o mesmo efeito pode ser alcançado usando classes

```
// Java
class Date
{
  int day, month, year;
}

Date birthday = new Date();

birthday.day = 31;
birthday.month = 1;
birthday.year = 2000;
```

Tipos de Dados Dinâmicos e Ponteiros (1)

- Existem situações na qual o **tamanho exato** ou **organização dos objetos** não podem ser determinados antes da execução
- A implementação de tipos dinâmicos representa uma **sobrecarga** para o sistema (durante o tempo de execução)

```
{
 typedef struct node {
   int value;
   struct node *next; /*ponteiro para uma estrutura*/
 } node_t;
 int V;
 node_t *Ptr;
 Ptr = malloc(sizeof(node_t)); /*aloca mem. dinamicamente*/
 Ptr->value = V; /*”dereference” do ponteiro*/
 Ptr->next = 0;
}
```

Tipos de Dados Dinâmicos e Ponteiros (2)

- Em Ada, um tipo de acesso é usado em vez de um ponteiro

```
type Node;          --declaração incompleta
type Ac is access Node;
type Node is
  record
    Value: Integer;
    Next: Ac;
  end record;
 V: Integer;
 A1: Ac;
 begin
   A1:= new(Node); --constrói o primeiro nó
   A1.Value := V; --a variável de acesso é “dereferenciada”
   A1.Next := null;
 ...
 end;
```

Nem Ada e nem C suportam coletor de lixo (*garbage collector*)

Tipos de Dados Dinâmicos e Ponteiros (3)

- Ponteiros podem também apontar para objetos estáticos

```
{
  typedef date_t events_t[MAX], *next_event_t;

  events_t history;
  next_event_t next_event;

  next_event = &history[0];
  /* recebe o endereço do primeiro elemento do vetor */

  next_event++;
 /* incrementa o ponteiro next_event que aprontará */
 /* para o próximo elemento do vetor */
}
```

O ponteiro pode apontar para objetos que estão fora do escopo

Tipos de Dados Dinâmicos e Ponteiros (4)

- Ada fornece uma solução segura para o problema do ponteiro fora dos limites

```
Object : aliased Some_Type;
-- aliased para dizer que ‘Object’ pode ser referenciado
-- por um tipo de acesso

type General_Ptr is access all Some_Type;
-- access all indica que uma variável de acesso deste tipo
-- pode apontar para objetos estáticos ou dinâmicos

Gp : General_Ptr := Object’Access;
-- atribui referência de ‘Object’ para ‘Gp’
```

Tipos de Dados Dinâmicos e Ponteiros (5)

- Uma forma final de definição de tipo de acesso, em Ada, permite uma **restrição somente de leitura** para ser implementado com o uso de acessos

```
Object1 : aliased Some_Type;
Object2 : aliased constant Some_Type := ...;

type General_Ptr is access constant Some_Type;

Gp1 : General_Ptr := Object1'Acess;
-- 'Gp1' pode agora somente ler o valor do Object1

Gp2 : General_Ptr := Object2'Acess;
-- 'Gp2' é uma referência para uma constante
```

Tipos de Dados Dinâmicos e Ponteiros (6)

- Todos os objetos em Java são referenciados para o objeto atual contendo o dado (nenhum acesso adicional ou tipo de ponteiro é necessário)

```
//Java
{
  class Node
  {
    int value;
    Node next;
  }
  Node Ref1 = new Node();
  Node Ref2 = new Node();
  ...
  if (Ref1 == Ref2) {...}
}
```

Comparação de dois objetos em Java irá comparar a localização dos objetos e não o valor encapsulado pelos objetos

Estruturas de Decisão - *if*

- Uma estrutura de decisão fornece uma escolha para a rota de execução que vai de algum ponto na seqüência de um programa até um ponto mais adiante daquela seqüência

```
--Ada
if A /= 0 then    --primeiro checa se A é diferente de zero
  if B/A > 10 then
    High := True;
  else
    High := False;
  end if;
end if;    --Se A=0 então nenhum valor é atribuído a High
```

- O C não precisa ter explicitamente a plavara chave **‘end’**

```
if (A!=0) {
  if (B/A > 10) high = 1;
  else high=0; }
```

Estruturas de Decisão – *switch/case*

- Decisão de múltiplos caminhos pode ser declarada e eficientemente implementada usando **case** (ou **switch**)

```
-- Ada
case Command is
     when 'A' | 'a'        => Action1;    --A ou a
     when 't'              => Action2;
     when 'e'              => Action3;
     when 'x' .. 'z'       => Action4;    --x, y ou z
     when others           => null;       --nenhuma ação
/* C e Java */
switch(command) {
     case 'A'       :
     case 'a'       : action1; break;     /*A ou a*/
     case 't'       : action2; break;
     …
     default: break;
}
```

Sem o break o controle continua para a próxima opção

Estruturas de Loop – *for*

- Existem dois tipos de loop: iteração e recursão
  - iteração: cada execução do loop é completada antes da próxima iniciar
  - recursão: o primeiro loop é interrompido para iniciar o segundo loop, o qual interrompe para iniciar um terceiro loop e assim por diante
- A iteração vem em duas formas:
  - um loop no qual o número de iterações é fixado antes do início da execução
  - um loop no qual um teste para a finalização é feito durante a iteração

```
-- Ada
for I in 0 .. 9 loop     --I é definido pelo loop
    A(I) := I;           --I é lido somente no loop
end loop;          --I está fora do escopo depois do loop

/*C e Java*/
for(i=0; i<=9; i++) {
    a[i] = i;
}
```

Estruturas de Loop – *while (1)*

- A principal variação com *while* está relacionada ao ponto no qual o teste de saída do loop é feito

```
-- Ada
while <Boolean Expression> loop
    <Statements>
end loop;
/*C e Java*/
while(<expression>) {
    /*avaliação da expressão – 0 indica o término do loop*/
    <statement>
}
/*C e Java*/
do {
    <statement>
} while(<expression>)
```

Estruturas de Loop – *while (2)*

- Ada, Java e C permitem **controlar o término do loop** a partir de qualquer ponto dentro do loop

```
-- Ada
loop
...
exit when <Boolean Expression>
...
end loop;

/*C e Java*/
while(1) {
    if (<expression>) break;
}
```

Loop infinito é alcançado removendo a linha de comando **exit when**

Mecanismo de Passagem de Parâmetro (1)

- Modos de parâmetro para expressar a transferência de dados

```
-- Ada
procedure Quadratic (A, B, C   : in Float;
                     R1, R2    : out Float;
                     Ok        : out Boolean;)
```

- Um parâmetro `in` atua como uma constante local dentro de um procedimento
  - um valor é atribuído para o parâmetro formal uma vez que o procedimento ou função seja chamado
- Dentro de procedimentos, um parâmetro `out` pode ser de escrita ou de leitura
  - um valor é atribuído para o parâmetro uma vez que o proced. seja finalizado
- Um parâmetro `in out` atua como uma variável no procedimento
  - na chamada um valor é atribuído para o parâmetro; no término um valor é retornado

Mecanismo de Passagem de Parâmetro (2)

- C passa parâmetros por valor (se resultados devem ser retornados, ponteiros devem ser usados)
  - C e Java só suportam funções/métodos (procedimentos são implementados com `void`)

```
void quadratic (float A, float B, float C,
                float *R1, float *R2, int *OK);
```

- Em Java, os argumentos primitivos são passados por cópia
  - variáveis de tipos de classe são variáveis de referência (quando são passados como argumentos eles são copiados)

```
public class Roots {
    float R1, R2;
}
boolean quadratic(final float A, final float B, final float C, Roots R);
```

Procedimentos em Ada (1)

```
--Ada
procedure Quadratic (A, B, C : in Float;
                     R1, R2 : out Float;
                     Ok : out Boolean) is
  Z : Float;
begin
  Z := B*B – 4.0*A*C;
  if Z < 0.0 or A = 0.0 then
    Ok := False;
    R1 := 0.0;        --valor arbitrário
    R2 := 0.0;
    return;           --retorna de um procedimento antes de
                      --alcançar o end
  end if;
  Ok := True;
  R1 := (-B + Sqrt(Z)) / (2.0*A);
  R2 := (-B - Sqrt(Z)) / (2.0*A);
end Quadratic;
```

Procedimentos em Ada (2)

- Existem duas facilidade em Ada que melhoram a visualização dos programas

```
type Setting is (Open, Closed);

type Valve is new Integer range 1 .. 10;
```

- A seguinte especificação de componente fornece um sub-programa para alterar o valor de uma válvula

```
procedure Change_Setting (Valve_Number : Valve;
                          Position : Setting := Closed; );
```

- Chamadas para este poderia ter um número de formas:

```
Change_Setting (6, Open) -- chamada normal
Change_Setting(3);        -- valor default usado ‘Closed’
Change_Setting(Position => Open,
               Valve_Number => 9); -- notação de nome
```

Procedimentos em C

```
/* C */
void quadratic (float A, float B, float C, float *R1,
                float *R2, int *OK)
{
  float Z;

  Z = B*B – 4.0*A*C;
  if (Z < 0.0 || A == 0.0) {
    *OK = 0;
    *R1 = 0.0;       /* valor arbitrário */
    *R2 = 0.0;
    return;
  }
  *OK = 1;
  *R1 = (-B + SQRT(Z)) / (2.0*A);
  *R2 = (-B – SQRT(Z)) / (2.0*A);
}
```

Procedimentos em Java

```
// Java
public class Roots {
  float R1, R2;
}
boolean quadratic(final float A, final float B, final
                  float C, Roots R) {

float Z;
Z = (float) (B*B – 4.0*A*C);
if (Z < 0.0 || A == 0.0) {
 R.R1 = 0f;
 R.R2 = 0f;   /* valor arbitrário */
  return false;
}
  R.R1 = (float)(-B + Math.sqrt(Z)) / (2.0*A);
  R.R2 = (float)(-B – Math.sqrt(Z)) / (2.0*A);
  return true;
}
```

Funções em Ada

- Ada, C e Java suportam funções em uma maneira parecida com os procedimentos

```
-- Ada
function Minimum(X, Y : in Integer) return Integer is
begin
  if X > Y then
    return Y;
  else
    return X;
  end if;
end Minimum;

/* C e Java */
int minimum(int X, int Y)
{
 if (X>Y) return Y;
 else return X;
}
```

Exercícios

- Ada termina cada construtor com **end** <nome do construtor>; C não usa um marcador final. Quais são as vantagens e desvantagens dos projetos destas linguagens?
- Java e C são *case-sensitive*; Ada não é. Quais são os argumentos a favor e contra do *case-sensitive*?
- Uma linguagem deve sempre solicitar que sejam dados valores iniciais para as variáveis?
- O uso do comando **exit** em Ada leva a programas legíveis e confiáveis?
- Escreva um método/função para buscar um dado elemento em uma lista encandeada usando Ada, C e Java
- Escrever a seqüência de Fibonacci dos termos inferiores a um número inteiro L qualquer usando Ada, C e Java

Lista Encandeada - Busca

```
typedef struct list {
  int key;
  struct list *next;
} mlist;

mlist *head;

mlist* search_list(mlist *l, int k){
  l = head;
  while(l!=NULL && l->key!=k) {
    l = l->next;
  }
  return l;
}
```

Seqüência de Fibonacci

```
int fib(int n) {
      if (n <= 2) return 1
      else return fib(n-1) + fib(n-2) }
```

Recursão cria chamadas desnecessárias que ficam na pilha

```
int fib(int n) {
  int f[n+1];
  f[1] = f[2] = 1;
  for (int i = 3; i <= n; i++)
    f[i] = f[i-1] + f[i-2];
  return f[n];
}
```