

Tipos de Dados

Prof. Dr. Eduardo Takeo Ueda
eduardo.tueda@sp.senac.br

Introdução (1/2)

- 1956:
 - FORTRAN I
 - INTEGER, REAL, arrays.
- 1983:
 - ADA
 - Usuário pode criar um único tipo para cada categoria de variáveis no espaço de problemas e deixar o sistema verificar os tipos.
- É conveniente pensar nas variáveis em termos de descritores:
 - Um descriptor é o conjunto dos atributos de uma variável.

Introdução (2/2)

- Questões de projeto para todos os tipos de dados:
 - Qual é a sintaxe de referência a variáveis?
 - Que operadores são definidos e como eles são especificados?

Tipos de Dados Primitivos

- Tipos de dados não definidos em termos de outros do mesmo nome são chamados **tipos de dados primitivos**:
 - Tipo numérico:
 - Inteiro.
 - Ponto flutuante.
 - Decimal.
 - Tipo lógico.
 - Tipo caractere.

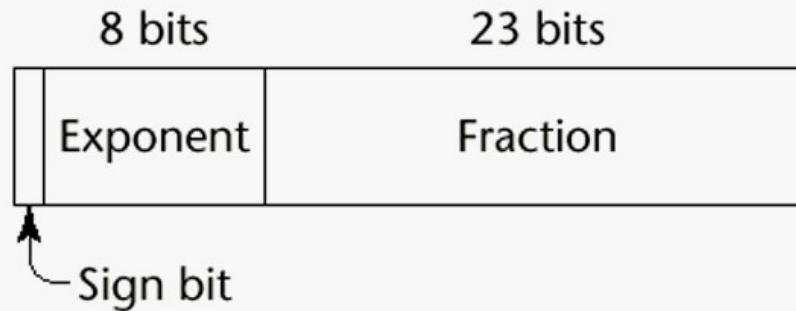
Tipo Numérico – Inteiro

- Quase sempre uma reflexão exata do *hardware*, de modo que o mapeamento é trivial:
 - **C** → seis tipos diferentes de inteiros:
 - **int**, **short int**, **long int**, **unsigned int**, **unsigned short int**, **unsigned long int**
 - **Java** → quatro tipos diferentes de inteiros:
 - **byte**, **short**, **int** e **long**
- Representação de números negativos:
 - Complemento de dois:
 - Conveniente para as operações de adição e subtração.

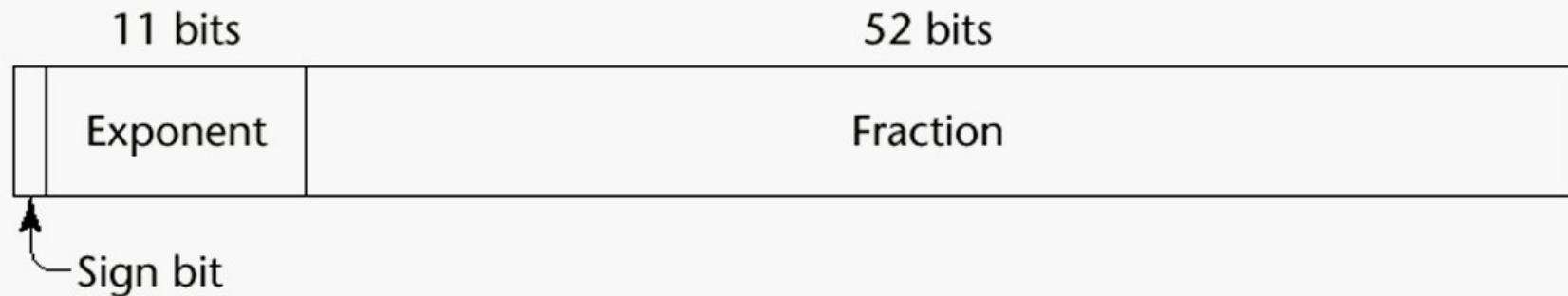
Tipo Numérico – Ponto Flutuante (1/2)

- Modela números reais, mas somente como aproximações:
 - Linguagens para uso científico suportam ao menos dois tipos de ponto flutuante, algumas vezes mais:
 - `float` e `double`.
 - Em geral, uma reflexão exata do *hardware*, mas não sempre.
 - Algumas linguagens permitem a especificação da precisão no código:
 - `ADA`.

Tipo Numérico – Ponto Flutuante (2/2)



(a)



(b)

Formatos de ponto flutuante IEEE: (a) precisão simples, (b) precisão dupla.

Tipo Numérico – Decimal

- Para aplicações de sistemas comerciais:
 - Armazena um número fixo de dígitos decimais, com o ponto decimal em uma posição fixa no valor.
 - Vantagem:
 - Capacidade de armazenar, com precisão, valores decimais.
 - Desvantagens:
 - Faixa de valores é restrita porque não se permite nenhum expoente.
 - Representação dos mesmos na memória é um desperdício:
 - *Binary Coded Decimal* (BCD) – strings de dígitos decimais.
 - Exemplos de linguagens:
 - COBOL.

Tipo Lógico

- Sua faixa de valores tem somente dois elementos:
 - Um para verdadeiro e um para falso.
- Foram introduzidos no **ALGOL 60** e têm sido incluídos na maioria das linguagens a partir de então.
 - Exceção popular é o **C**, que utiliza expressões numéricas.
- Poderia ser implementada como *bits*, mas usualmente como *bytes*.
- Vantagem:
 - Legibilidade.

Tipo Caractere

- Armazenados como codificações numéricas:
 - ASCII:
 - *American Standard Code for Information Interchange.*
 - Codificação em oito *bits*.
 - UNICODE:
 - Codificação em dezesseis *bits*.
 - Suporte para a maioria das linguagens naturais internacionais.
 - O **Java** foi a primeira linguagem a introduzir o conjunto de caracteres UNICODE.

Tipos String de Caracteres (1/6)

- Um **tipo string de caracteres** é aquele cujos valores consistem em seqüências de caracteres:
 - Questões de projeto:
 - É um tipo primitivo ou apenas um tipo especial de *array*?
 - O tamanho dos objetos é estático ou dinâmico?
 - Operações:
 - Atribuição.
 - Comparação.
 - Concatenação.
 - Referência a *substrings*.
 - Casamento de padrões (*pattern matching*).

Tipos String de Caracteres (2/6)

- Exemplos em linguagens de programação:
 - **ADA:**
 - **STRING** é um tipo que é predefinido como um *array* unidimensional de elementos **CHARACTER**.
 - Referência a *substrings*, concatenação, operadores relacionais e atribuição são oferecidos aos tipos **STRING**.
 - **C e C++:**
 - Usam *arrays* de **char** para armazenar *strings* de caracteres.
 - Biblioteca **string.h** fornece uma coleção de operações sobre *strings*.
 - **Java:**
 - **String** é um objeto responsável pela representação e operações sobre *strings*.

Tipos String de Caracteres (3/6)

- Opções de tamanho da *string*:
 - string de tamanho estático:
 - O tamanho da *string* é especificado na declaração.
 - FORTRAN 77, COBOL, PASCAL e ADA.
 - string de tamanho dinâmico limitado:
 - O tamanho máximo da *string* é especificado na declaração.
 - C e C++.
 - string de tamanho dinâmico:
 - Exige o *overhead* da alocação e desalocação dinâmica do armazenamento.
 - SNOBOL4 e PERL.

Tipos String de Caracteres (4/6)

- Avaliação:
 - Ajudam a redigibilidade.
 - Como um tipo primitivo com tamanho estático, eles são fáceis de implementar – então porque não?
 - Tamanho dinâmico é bom, mas vale a pena (custo)?

Tipos String de Caracteres (5/6)

- Implementação:
 - Tamanho estático:
 - Descrição em tempo de compilação.
 - Tamanho dinâmico limitado:
 - Pode necessitar uma descrição em tempo de execução para tamanho:
 - Não em C e C++, onde o final de um *string* é marcado com o caractere nulo.
 - Tamanho dinâmico:
 - Necessita de uma descrição em tempo de execução.
 - Alocação é o maior problema da implementação.

Tipos String de Caracteres (6/6)

Static string
Length
Address

Descritor em tempo de compilação
(*compile-time*) para *strings* estáticas.

Limited dynamic string
Maximum length
Current length
Address

Descritor em tempo de compilação (*compile-time*)
para *strings* dinâmicas limitadas.

Tipos Ordinais

- Um **tipo ordinal** é aquele cuja faixa de valores possíveis pode ser facilmente associada ao conjunto dos números inteiros positivos:
 - Tipo enumeração.
 - Tipo subfaixa.

Tipo Enumeração (1/2)

- Um **tipo enumeração** é aquele em que todos os valores possíveis, os quais se tornam constantes simbólicas, são enumerados na definição:
 - Questão de projeto:
 - Uma constante simbólica deveria ser possível estar em mais de uma definição de tipo?
 - Exemplo em **ADA**:
 - `type DIAS is (Seg, Ter, Qua, Qui, Sex, Sab, Dom);`
 - **C** e **C++**:
 - Podem ser usadas como índices de *arrays*, como variáveis num **for**, seletores em um **case**.
 - **Java**:
 - Os tipos **enum** do **C** e **C++** não estão incluídos.

Tipo Enumeração (2/2)

- Avaliação:
 - Auxiliam a legibilidade:
 - Não é necessário codificar uma cor como um número.
 - Auxiliam a confiabilidade:
 - Compilador pode verificar operações e faixa de valores.

Tipo Subfaixa

- Um **tipo subfaixa** é uma subsequencia de um ordinal:
 - Exemplo em Pascal:
 - `type maiuscula = 'A'..'Z';`
 - `type indice = 1..100;`
 - Avaliação de tipo sub-faixa:
 - Auxilia a legibilidade.
 - Confiabilidade:
 - Aumenta pois atribuição de valores fora da faixa é detectada como erro de execução.

Tipos Ordinais

- Implementação de tipos ordinais:
 - Tipos enumerados são implementados como inteiros (positivos).
 - Tipos sub-faixa são implementados da mesma forma que seus tipos pai, com código inserido (pelo compilador) para restringir atribuições a variáveis sub-faixa.

Tipos Array (1/12)

- Um array é um agregado de elementos de dados homogêneos no qual um indivíduo é identificado por sua posição no agregado, relativo ao primeiro elemento:
 - Questões de projeto:
 - Quais tipos são legais (legítimos) para índices?
 - As expressões nos índices de uma faixa de elementos são verificadas?
 - Em que momento os índices sofrem vinculações?
 - Quando a alocação se dá?
 - Qual é o número máximo de índices?
 - Um objeto *array* pode ser inicializado?
 - O fatiamento de um *array* é permitido?

Tipos Array (2/12)

- Elementos específicos de um *array* são referenciados por meio de um mecanismo sintático de dois níveis, cuja primeira parte é o nome do agregado e a segunda é um seletor possivelmente dinâmico que consiste em um ou mais itens conhecido como subscritos ou índices:
 - Sintaxe:
 - FORTRAN, PL/I, ADA usam parênteses.
 - Maioria das outras linguagens usa colchetes.
 - Tipos de índices:
 - FORTRAN, C e Java – somente inteiros.
 - Pascal – qualquer tipo ordinal (integer, boolean, char ou enum).
 - ADA – int ou enum (incluindo boolean e char).

Tipos Array (3/12)

- A vinculação do tipo subscrito a uma variável de *array* normalmente é estática, mas as faixas de valor de subscrito, às vezes, são vinculadas dinamicamente:
 - O limite inferior da faixa do subscrito é implícito:
 - C, C++ e Java → fixado em zero.
 - FORTRAN e PASCAL → fixado em um.
 - Os *arrays* ocorrem em quatro categorias:
 - Baseiam-se na vinculação às faixas de valor de subscritos e nas vinculações ao armazenamento.

Tipos Array (4/12)

- Um **array estático** é aquele em que as faixas de subscritos estão estaticamente vinculadas e a alocação de armazenamento é estática:
 - Vantagem:
 - Eficiência, pois nenhuma alocação ou desalocação dinâmica é exigida.
 - Linguagens de Programação:
 - **FORTAN 77**.

Tipos Array (5/12)

- Um array stack-dinâmico fixo é aquele em que as faixas de subscritos estão estaticamente vinculadas, mas a alocação é feita no momento de elaboração da declaração durante a execução:
 - Vantagens:
 - Eficiência de espaço.
 - Linguagens de Programação:
 - Arrays locais em Pascal.

Tipos Array (6/12)

- Um array stack-dinâmico é aquele em que as faixas de subscrito estão dinamicamente vinculadas e a alocação de armazenamento é dinâmica:
 - Porém permanecem fixos durante o tempo de vida da variável.
 - Vantagem:
 - Flexibilidade, pois o tamanho do *array* não precisa ser conhecido até que ele esteja prestes a ser usado.
 - Linguagens de Programação:
 - C e C++.

Tipos Array (7/12)

- Um array heap-dinâmico é aquele em que a vinculação das faixas de subscrito e de alocação de armazenamento é dinâmica e pode mudar qualquer número de vezes durante o seu tempo de vida:
 - Vantagem:
 - Flexibilidade, pois os *arrays* podem crescer ou reduzir durante a execução do programa.
 - Linguagens de Programação:
 - C e C++ → utilização dos operadores **new** e **delete** para gerenciar o armazenamento do *heap*.
 - Java → todos os arrays são dinâmicos.

Tipos Array (8/12)

- Número de índices:
 - **FORTAN I** permitia até 3.
 - **FORTAN 77** permite até 7.
 - **C**, **C++** e **Java** permite apenas um, mas os elementos podem também ser *arrays*.
 - Outras – sem limite.

Tipos Array (9/12)

- Inicialização de *arrays*:
 - Usualmente só uma lista de valores que são colocados no *array* na ordem na qual os elementos do *array* estão armazenados (dispostos) na memória.
 - Exemplos:
 - **FORTRAN** – usa o comando **DATA**, ou coloca os valores entre **/ /** na declaração.
 - **C**, **C++** e **Java** – coloca os valores entre chaves. Pode-se deixar o compilador contá-lo:
 - **Ex.: `int stuff [] = {2, 4, 6, 8};`**
 - **ADA** – posições para os valores podem ser especificados:
 - **`SCORE : ARRAY (1..14, 1..2) := (1 => (24, 10), 2 => (10, 7), 3 => (12, 30), others => (0, 0));`**
 - **Pascal** e **Modula-2** não permitem inicialização.

Tipos Array (10/12)

- Operações com arrays:
 - APL – muitas.
 - ADA:
 - Atribuição (assinalamento).
 - Concatenação – para todos os *arrays* unidimensionais.
 - Operadores relacionais (apenas = e !=).
 - FORTRAN 90
 - Intrínsecos (subprogramas) para uma grande variedade de operações com *arrays*.
 - Ex.: multiplicação de matrizes, produto escalar.

Tipos Array (11/12)

- Fatiamento:
 - Uma **fatia** é uma sub-estrutura de um array, nada mais que um mecanismo de referência.
 - Exemplo de fatiamento:
 - **FORTRAN 90**
 - `INTEGER MAT (1 : 4, 1 : 4)`
 - `MAT (1 : 4, 1)` – a primeira coluna
 - `MAT (2, 1 : 4)` – a segunda linha
 - **ADA** – Somente arrays unidimensionais
 - `LIST (4..10)`

Tipos Array (12/12)

- Implementação de *arrays*:
 - Funções de acesso mapeiam expressões de índices para um endereço no *array*.
 - Ordem de armazenamento:
 - Ordenação por maior linha ou maior coluna.

Array Associativo (1/2)

- Um **array associativo** é uma coleção não ordenada de elementos de dados que são indexados por um número igual de valores chamados chaves:
 - Questão de projeto:
 - Qual é a forma de referência a elementos?
 - O tamanho é estático ou dinâmico?

Array Associativo (2/2)

- Estrutura e operações em Perl:
 - Nomes começam com %.
 - Literais são delimitados por parênteses:
 - `%salarios = ("Cedric" => 75000, "Perry" => 57000, "Mary" => 55750, "Gary" => 47850);`
 - Indexação é feita usando {} e chaves:
 - `$salarios{"Perry"} = 58850;`
 - Elementos podem ser removidos com delete:
 - `delete $salarios{"Gary"};`

Tipo Registro (1/5)

- Um **registro** é um agregado possivelmente heterogêneo de elementos cujos elementos individuais são identificados por nomes:
 - Questão de projeto:
 - Qual a forma das referências?
 - Quais operadores unitários são definidos?

Tipo Registro (2/5)

- Sintaxe da definição de um registro:
 - COBOL usa números de nível para mostrar registros aninhados.
 - Outras usam definições recursivas:
 - Exemplo em ADA:

```
REGISTRO_EMPREGADO: record
  NOME_EMPREGADO: record
    PRIMEIRO: STRING(1..20);
    MEIO      : STRING(1..10);
    ULTIMO    : STRING(1..20);
  end record;
  TAXA_HORARIA: FLOAT;
end record;
```

Tipo Registro (3/5)

- Referência a campos de um registro:
 - **COBOL**:
 - nome_do_campo **OF** nome_do_registro_1 **OF** ... **OF** nome_do_registro_n
 - Outras (notação com ponto):
 - nome_registro_1.nome_registro_2.....nome_registro_n.nome_do_campo
 - Referências a campos amplamente qualificadas (*fully qualified*) devem incluir todos os nomes de registro.
 - **Pascal** e **Modula-2** possuem uma cláusula **with** para abreviar referências.

Tipo Registro (4/5)

- Operadores de registro:
 - Atribuição (assinalamento):
 - **Pascal**, **ADA** e **C** permitem-no se as variáveis são idênticas.
 - Inicialização:
 - Permitida em **ADA**, usando um agregado constante.
 - Comparação:
 - Em **ADA**, **=** e **/=**, sendo que um dos operandos pode ser um agregado constante.

Tipo Registro (5/5)

- Comparando *records* e *arrays*:
 - Acesso a elementos de um *array* é muito mais lento que acesso a campos de um registro, pois índices são dinâmicos (nomes de campos são estáticos).
 - Índices dinâmicos poderiam ser usados com acesso a campos de registros, mas isso poderia desativar a verificação de tipos e seria muito mais lento.

Tipos União (1/2)

- Uma união é um tipo que pode armazenar diferentes valores de tipo durante a execução do programa:
 - Questões de projeto para uniões:
 - Que tipo de verificação de tipo (se é que há alguma), deve ser feita?
 - Uniões deveriam ser integradas a registros?

Tipos União (2/2)

- **C** e **C++** - uniões livres com a construção **union**:
 - Não pode ser parte dos registros.
 - Não existe verificação de tipos de referência.
- **Java** não possui registros nem uniões.
- Avaliação:
 - Potencialmente insegura na maioria das linguagens.

Tipos Conjuntos (1/3)

- Um tipo **conjunto** é aquele cujas variáveis podem armazenar coleções não-ordenadas de valores distintos de algum tipo ordinal chamado de seu **tipo básico**:
 - Questões de projeto:
 - Qual o número máximo de elementos em qualquer tipo básico de conjunto?

Tipos Conjuntos (2/3)

- Exemplo em Pascal:

```
type cores = (vermelho, azul, verde, amarelo, laranja,  
branco, preto);  
    conjcores = set of cores;  
var conj1, conj2 : conjcores;  
begin  
    conj1 := [vermelho, azul, verde, amarelo, branco];  
    conj2 := [preto, azul];  
end;
```

Tipos Conjuntos (3/3)

- Avaliação:
 - Se uma linguagem não possui conjuntos, eles podem ser simulados, seja como tipos enumerados ou como *arrays*.
 - *Arrays* são mais flexíveis que conjuntos, mas operações são muito mais lentas.
- Implementação:
 - Normalmente armazenados como pedaços de cadeias (strings) e usam operações lógicas para seu conjunto de operações.

Tipos Ponteiros (1/8)

- Um tipo ponteiro é aquele em que as variáveis têm uma faixa de valores que consiste em endereços de memória e um valor especial, **nil** ou **null**:
 - **nil** ou **null** não é um endereço válido e serve para indicar que um ponteiro não pode ser usado atualmente para referenciar qualquer célula de memória.
 - Usos:
 - Flexibilidade no endereçamento.
 - Gerenciamento do armazenamento dinâmico.
 - Operações fundamentais sobre ponteiros:
 - Atribuição de um endereço a um ponteiro.
 - Referências (explícita versus referência implícita).

Tipos Ponteiros (2/8)

- Questão de projeto:
 - Quais são o escopo e o tempo de vida de uma variável de ponteiro?
 - Qual é o tempo de vida de uma variável heap-dinâmica?
 - Ponteiros são restritos a um tipo em particular?
 - Ponteiros são usados para gerenciamento do armazenamento dinâmico, endereçamento indireto, ou ambos?
 - Uma linguagem deve suportar tipos ponteiro, tipos referência ou ambos?

Tipos Ponteiros (3/8)

- Problemas com ponteiros:
 - Ponteiros “soltos” (perigoso):
 - Um ponteiro aponta para uma variável heap-dinâmica que foi desalocada;
 - Criando um:
 - Alocar uma variável heap-dinâmica e “setar” um ponteiro apontando para ela.
 - Setando um segundo ponteiro para o valor do primeiro.
 - Desalocando a variável heap-dinâmica, usando o primeiro ponteiro.

Tipos Ponteiros (4/8)

- Problemas com ponteiros:
 - Variáveis Heap-Dinâmicas perdidas (desperdício):
 - Uma variável heap-dinâmica que não é mais referenciada por qualquer ponteiro.
 - Criando uma:
 - O ponteiro p1 é setado para uma variável heap-dinâmica recém criada.
 - p1 é, posteriormente, levado a apontar para outra variável heap-dinâmica recém criada.
 - O processo de perder variável heap-dinâmica é chamado de vazamento de memória.

Tipos Ponteiros (5/8)

- Exemplos – C e C++:
 - Usados para gerenciamento de armazenamento dinâmico e endereçamento.
 - Deferência explícita e operador “endereço-de”.
 - Podem apontar para, virtualmente, qualquer variável em qualquer lugar da memória.
 - Pode fazer aritmética de endereços em formas restritas.

```
float stuff[100];
```

```
float *p;
```

```
p = stuff;
```

- `*(p+5)` é equivalente a `stuff[5]` e `p[5]`;
- `*(p+i)` é equivalente a `stuff[i]` e `p[i]`;

Tipos Ponteiros (6/8)

- Tipos referência do C++:
 - Ponteiros constantes são implicitamente deferenciados.
 - Usados para parâmetros.
 - Vantagens da passagem por referência e da passagem por valor.

Tipos Ponteiros (7/8)

- **Java** – Somente referência:
 - Sem aritméticas de ponteiros.
 - Pode somente apontar a objetos (todos *heap*).
 - Sem desalocador explícito (*garbage collection* usado):
 - Significa que não há referências soltas.
 - Dereferencia é sempre implícita.

Tipos Ponteiros (8/8)

- Avaliação de ponteiros:
 - Ponteiros e objetos soltos são um problema, como também o gerenciamento do *heap*.
 - Ponteiros são como **goto** – aumentam a faixa de células que pode ser acessada por uma variável.
 - Ponteiros são necessários – para que possamos projetar uma linguagem sem eles.