Tipos de dados

Marco A L Barbosa
malbarbo.pro.br

Departamento de Informática
Universidade Estadual de Maringá

Conteúdo

Introdução

Introdução

- Um **tipo de dado** define a coleção de valores e um conjunto de operações nestes valores
- É importante que uma linguagem de programação forneça uma coleção apropriada de tipos de dados

Introdução

- Utilidade
  - Legibilidade e facilidade de escrita
  - Modularização e abstração
  - Detecção de erros
  - Documentação
  - Criação de ferramental

Introdução

- Sistema de tipos
  - Define como um tipo é associada a uma expressão
  - Incluí regras para equivalência e compatibilidade de tipos
- Entender o sistema de tipos de uma linguagem é um dos aspectos mais importante para entender a sua semântica
- Questão de projeto relativa a todos os tipos de dados
  - Quais operações podem ser utilizadas em variáveis de um determinado tipo e como elas são especificadas?

Tipos primitivos

Tipos primitivos

- **Tipos primitivos** são aqueles que não são definidos em termos de outros tipos
- Quase todas as linguagens de programação fornecem um conjunto de tipos primitivos
- Usados com construções de tipo para fornecer os tipos estruturados

Tipos primitivos

- Os mais comuns são
  - Tipos numéricos
  - Tipos booleanos
  - Tipos caracteres

Tipos primitivos

- Tipos numéricos
  - Inteiro
    - Vários tamanhos
    - Várias representações
    - Com ou sem sinal
    - Tamanho ilimitado
  - Ponto flutuante
    - Modelo (aproximado) dos números reais
    - Padronização IEEE 754 (`float` e `double`)

Tipos primitivos

- Tipos numéricos
  - Número complexo
    - Par de números com ponto flutuante
    - Literal em Python: `7 + 3j`
  - Decimal
    - Utilizado em aplicações financeiras
    - Armazenado com um número fixo de dígitos em forma codificada (BCD)
    - Cobol, C#
    - Vantagem: precisão
    - Desvantagem: intervalo limitado, desperdício de memória

Tipos primitivos

- Tipo booleano
  - O mais simples de todos
  - Dois valores: `true` e `false`
  - Pode ser implementado com 1 bit, mas em geral é implementado com uma palavra da memória
  - Algumas linguagens (C89) não tem tipo booleano, valores inteiros iguais a zero são considerados falsos e valores diferentes de zero verdadeiros
  - O tipo booleano é mais legível que inteiros quando usados na representação de flags e switches

Tipos primitivos

- Tipo caractere
  - Caracteres são armazenados no computador como valores numéricos
  - Os valores numéricos são mapeados para caracteres através de tabelas
    - ASCII (7 bits - 127 valores)
    - ISO 8859-1 (8 bits - 256 valores)
    - Unicode (16 - não é suficiente, e 32 bits - suficiente para representar todos os caracteres utilizados pelos humanos)
  - Python não tem um tipo caractere, um caractere é representado como um cadeia de tamanho 1

Cadeia de caracteres

Cadeia de caracteres

- O tipo **cadeia de caractere** tem como valores sequências de caracteres
- Questões de projeto
  - As cadeias são simplesmente um tipo especial de arranjo de caractere ou um tipo primitivo?
  - As cadeias tem um tamanho estático ou dinâmico?

Cadeia de caracteres

- Operações comuns
  - Atribuição
  - Concatenação
  - Referência a subcadeia
  - Comparação
  - Casamento de padrão

Cadeia de caracteres - exemplos

- C e C++
  - Não é primitivo
  - Uso de arranjo de caracteres
  - A cadeia é terminada com um caractere especial nulo (`'\0'`)
  - Biblioteca de funções (`strcpy`, `strcat`, `strcmp`, `strlen`)
  - Problemas de confiabilidade
  - O que acontece no exemplo a seguir se `src` tiver tamanho 30 e `dest` tamanho 20?

    ```
    strcpy(dest, src);
    ```

  - C++ oferece uma classe `string`, semelhante a classe `String` do Java, que é mais confiável que as cadeias do C

Cadeia de caracteres - exemplos

- Fortran
  - Tipo primitivo com diversas operações
- Python
  - Tipo primitivo com diversas operações
  - Imutável
  - Algumas operações são como as de arranjos

Cadeia de caracteres - exemplos

- Java
  - Classe `String` - imutável
  - Classe `StringBuilder` e `StringBuffer` - mutável
  - Os caracteres individuais são acessado com o método `charAt(index)`
  - C# é semelhante
- Perl, JavaScript, Ruby e PHP
  - Primitivo com diversas operações, inclusive de casamento de padrão

Cadeia de caracteres - opções de tamanho

- Tamanho estático: Java (classe `String`), Python
  - Como é possível realizar um operação de concatenação (ou outra operação destrutiva qualquer) em uma cadeia de tamanho estático e imutável?
  - Criando uma nova cadeia que o resultado da operação
- Tamanho dinâmico limitado: C
- Tamanho dinâmico: C++, Perl
- Ada suporta os três tipos
- Como estes tipos de cadeias podem ser implementados?

Cadeia de caracteres

- Avaliação
  - Ajuda na facilidade de escrita
  - Trabalhar com tipos primitivos é em geral mais conveniente do que com arranjos de caracteres
  - Operações como concatenação e busca são essenciais e devem ser fornecidas
  - Cadeias com tamanho dinâmicos são interessantes, mas o custo de implementação deve ser levado em consideração

Tipo ordinal definido pelo usuário

Tipo ordinal definido pelo usuário

- Um **tipo ordinal** é aquele em que o intervalo de valores pode facilmente ser associado com o conjunto dos inteiros positivos
- Exemplos de tipos primitivos ordinal do Java
  - `boolean`
  - `char`
  - `int`
- Definidos pelo usuário
  - Tipos enumerados
  - Tipos subintervalo

Tipo ordinal definido pelo usuário - tipos enumerados

- Um **tipo enumerado** é aquele que todos os possíveis valores, que são constantes nomeadas, são enumerados na definição
- Exemplo em C#

```
enum Dia { Seg, Ter, Qua, Qui, Sex, Sab, Dom };
```

- Em geral, os valores $0, 1, 2, ...$ são atribuídos implicitamente as constantes enumeradas, mas outros valores podem ser explicitamente atribuídos

Tipo ordinal definido pelo usuário - tipos enumerados

- Questões de projeto
  - Uma constante enumerada pode aparecer em mais de uma definição de tipo, e se pode, como o tipo de uma ocorrência da constante é verificada?
  - Os valores enumerados podem ser convertidos para inteiros?
  - Algum outro tipo pode ser convertido para um tipo enumerado?

Tipo ordinal definido pelo usuário - tipos enumerados

- Linguagens sem suporte a tipos enumerados

```
type Cor = int;
Cor vermelho = 0, azul = 1; // Possíveis cores
Cor x = vermelho;
```

- Limitações? Qualquer valor inteiro, mesmo que não seja uma cor válida, pode ser atribuída a valores do "tipo" `Cor`. Todas as operações do tipo `int` pode ser feita em variáveis do tipo `Cor`, etc...

```
x = 2;
x = x + 3;
```

Tipo ordinal definido pelo usuário - tipos enumerados

- C / C++

```
enum Cor { vermelho, azul, verde, preto };
Cor c = azul;
int i = c; // válido em C e C++
c++;       // ilegal em C++, legal em C
c = 4;     // ilegal em C++, legal em C
```

- As constantes enumeradas só podem aparecer em um tipo

Tipo ordinal definido pelo usuário - tipos enumerados

- Ada
  - As constantes enumeradas podem aparecer em mais do que um declaração de tipo
  - O contexto é utilizado para distinguir entre constantes com o mesmo nome
  - As vezes o usuário tem que indicar o tipo
  - As constantes enumeradas não podem ser convertidas para inteiros
  - Outros tipos não podem ser convertidos para tipos enumerados

Tipo ordinal definido pelo usuário - tipos enumerados

- Java
  - Adicionado ao Java 5.0 em 2004
  - Todos os tipos enumerados são subclasses da classe `Enum`
  - Métodos herdados de `Enum` (`ordinal`, `name`, etc)
  - É possível adicionar métodos e atributos
  - Sem conversões automáticas
  - É possível utilizar o mesmo nome da constante enumerada em diferentes tipos

Tipo ordinal definido pelo usuário - tipos enumerados

- Outras linguagens
  - Interessantemente, nenhum linguagem de script recente suporta tipos enumerados. Por quê?

Tipo ordinal definido pelo usuário - tipos enumerados

- Avaliação
  - Ajuda na legibilidade
  - Confiabilidade (o compilador pode detectar erros)
  - Java e Ada oferecem mais confiabilidade do que C++, e C++ mais do C
  - Java tem um melhor encapsulamento (como adicionar um atributo a um tipo enumerado em C/C++ ou Ada?)

Tipo ordinal definido pelo usuário - tipos subintervalo

- Um **tipo subintervalo** é uma subsequência contínua de um tipo ordinal
- Em geral,são utilizados para índices de arranjos

Tipo ordinal definido pelo usuário - tipos subintervalo

- Exemplo em Ada

```
subtype Indice is Integer range 1..100;
type Dia is (Seg, Ter, Qua, Qui, Sex, Sab, Dom);
subtype Dia_Semana is Dia range Seg..Sex;
Dia1 : Dia;
Dia2 : Dia_Semana;
...
Dia2 := Dia1;
```

- A atribuição `Dia2 := Dia1` é verificada em tempo de execução e só será válida se `Dia1` estiver no intervalo `Seg..Sex`.

Tipo ordinal definido pelo usuário - tipos subintervalo

- Avaliação
  - Ajuda na legibilidade e confiabilidade
  - Segundo Sebesta, é estranho que nenhum outra linguagem contemporânea além do Ada 95 suporte subintervalos
  - Uso restrito (e se os valores permitidos não formam um intervalo contínuo?)

Arranjos

Arranjos

- Um **arranjo** é um agregado homogêneo de elementos, onde cada elemento é identificado pela sua posição relativa ao primeiro elemento do agregado

Arranjos

- Questões de projeto
  - Quais são os tipos permitidos para os índices?
  - Os índices são checados?
  - Quando o intervalo de índice é vinculado?
  - Quando a alocação acontece?
  - Os arranjos podem ser inicializados quando a sua memória é alocada?
  - Arranjos multidimensionais irregulares ou regulares são permitidos?
  - Quais os tipos de fatias (slices) são permitidos?

Arranjos

- Índices
  - Em um operação de seleção, é especificado o nome do arranjo e o(s) índice(s), e o elemento correspondente é obtido

    `nome_arranjo(indices) -> elemento`
  - Sintaxe da seleção
    - Fortran e Ada: `Soma := Soma + B(I)`
    - Maioria das linguagens: `Soma := Soma + B[I]`

Arranjos

- Índices
  - Tipos dos índices
    - Maioria da linguagens: inteiros
    - Ada: qualquer tipo ordinal

```
type Dia_Semana is (Seg, Ter, Qua, Qui, Sex);
type Vendas is array (Dia_Semana) of Float;
```

Arranjos

- Índices
  - Verificação do intervalo do índice
    - Java, C#, ML, Python: sim
    - C, C++, Perl, Fortran: não
    - Ada: por padrão sim, mas o programador pode optar por não
  - Índices negativos: Perl, Python, Ruby, Lua

Arranjos

Categorização segundo a vinculação do intervalo de índices, da vinculação da memória e de onde a memória é alocada

- Estático
  - Intervalo de índices e memória vinculados estaticamente

Arranjos

- Dinâmico fixo na pilha
  - Intervalo de índices vinculado estaticamente, mas a alocação é feita em tempo de execução (na elaboração da variável)
- Dinâmico na pilha
  - Intervalo de índices e memória são vinculados em tempo de execução
  - O intervalo de índices não pode ser alterado depois de vinculado

Arranjos

- Dinâmico fixo no heap
  - Semelhante ao dinâmico fixo na pilha
  - Intervalo de índices é vinculado em tempo de execução, mas não pode ser alterado
- Dinâmico no heap
  - Intervalo de índices e memória são vinculados dinamicamente
  - Podem ser alterados a qualquer momento

Arranjos - exemplos

- C++ suporta os 5 tipos

```
void f(int n) {
    static int a[10];     // Estático
    int b[10];            // Dinâmico fixo na pilha
    int c[n];             // Dinâmico na pilha
    int *d = new int[n]; // Dinâmico fixo no heap
    vector<int> e;        // Dinâmico no heap
    e.push_back(52);      // aumenta o tamanho em 1
    delete[] d;
}
```

Arranjos - exemplos

- Java e C#
  - Todos os arranjos (primitivos) são dinâmicos fixo no heap
  - A classe `ArrayList` fornecem suporte a arranjos dinâmicos no heap
- Perl, JavaScript, Ruby, Lua, Python: dinâmicos no heap

Arranjos - inicialização

- Fortran

```
Integer, Dimension (3) :: List = (/0, 5, 5/)
```

- C, C++, Java, C#

```
int list[] = {4, 5, 7, 83};
```

Arranjos - inicialização

- Strings em C

```
char name[] = "freddie";
char *name[] = {"Bob", "Jake", "Darcie"};
```

- Ada

```
List : array (1..5) of Integer := (1, 3, 5, 7, 9);
Bunch : array (1..5) of Integer :=
        (1 => 17, 3 => 34, others => 0);
```

Arranjos - inicialização

- Python suporta *list comprehensions* (criação de listas com uma notação semelhante a de conjuntos)

```
> a = [4, 10, -6]
> [x ** 2 for x in range(12) if x % 3 == 0]
[0, 9, 36, 81]
```

Arranjos - operações

- Atribuição
- Comparação de igualdade
- Fatias

Arranjos - operações

- APL fornece o mais poderoso conjunto de operações com arranjos
- Ada: atribuição, comparação de igualdade e desigualdade, concatenação (`&`)
- Python: atribuição (referência), igualdade, pertinência (`in`), concatenação (+)
- Fortran 95: atribuição, operações aritméticas, relacionais e lógicas
- Linguagens baseadas no C: através de funções de biblioteca

Tipos de arranjos

- Um **arranjo regular** é um arranjo multidimensional onde todas as linhas tem o mesmo número de elementos, todas as colunas tem o mesmo números de elementos...
- Um **arranjo irregular** tem linhas (colunas, etc), com um número variado de elementos
    - Em geral quando arranjos multidimensionais são arranjos de arranjos

Tipos de arranjos - exemplos

- Arranjo irregular (Java)

  ```
  myArray[3][7]
  ```

- Arranjo regular (Julia, Python, Fortran, Ada, C#)

  ```
  myArray[3, 7]
  ```

- Ambos (C/C++, Julia, C#)

Tipos de arranjos - exemplos

- Arranjo regular em C/C++

```
int matrix[2][3] = {
    {1, 2, 3},
    {4, 5, 6},
};
```

- Arranjo irregular em C/C++

```
int a[3] = {1, 2, 3};
int b[2] = {4, 5};
int *matrix[2] = {
    a,
    b,
};
```

Arranjos

- Uma **fatia** de um arranjo é uma subestrutura deste arranjo
- Algumas linguagens oferecem maneiras de se referir a fatias
- Exemplos
  - Python

```
vector = [2, 4, 6, 8, 10, 12, 14, 16]
mat = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
vector[3:6]   # [8, 10, 12]
mat[1]        # [4, 5, 6] é uma fatia?
vector[0:7:2] # [2, 6, 10, 14]
```

  - Fortran 95: mais elaborado, se `mat` é uma matriz, a coluna 2 pode ser referenciada por `mat(:,2)`

Arranjos

- Avaliação
  - Praticamente todas as linguagens fornecem suporte a arranjos
  - Avanços em relação ao Fortran 1
    - Possibilidade de usar tipos ordinais como índices
    - Fatias
    - Arranjos dinâmicos

Arranjos - implementação

- A função de acesso mapeia os índices para um endereço do arranjo
- Arranjos unidimensionais
  - `endereco(arranjo[k]) = endereco(arranjo[limite_inferior]) + ((k - limite_inferior) * tamanho_elemento)`
  - Se `limite_inferior` é 0, `endereco(arranjo[k]) = endereco(arranjo[0]) + k * tamanho_elemento`

Arranjos - implementação

- Arranjos multidimensionais regulares
  - Ordenados por linhas ou colunas
  - O tipo da implementação tem alguma influência na execução do programa?

Arranjos associativos

- Um **arranjo associativo** ou **dicionário** é um agregado de elementos indexados através de chaves
- Exemplo Perl

```
%salarios = ("Gary" => 75000, "Perry" => 57000
             "Mary" => 55750, "Cedric" => 47850);
$salarios{"Perry"} = 58850;
delete $salarios{"Garry"};
%salaries = ();
```

Arranjos associativos

- Outras linguagens que possuem arranjos associativos
  - PHP, Lua, Python, JavaScript (primitivo)
  - C#, C++, Java (bibliotecas)
- Como os arranjos associativos podem ser implementados?
  - Tabelas hash
  - Árvores de busca (AVL, vermelho e preto, etc)

Registros e tuplas

Registros

- Um **registro** é um agregado heterogêneo de elementos identificados pelo nome e acessados pelo deslocamento em relação ao início do registro
- Questões de projeto
  - Qual é sintaxe para referenciar os campos?
  - Referências elípticas são permitidas?

Registros - definição

- Cobol utiliza números para mostrar o nível de registros aninhados

```
01  EMPLOYEE-RECORD.
    02  EMPLOYEE-NAME.
        05  FIRST  PICTURE IS X(20).
        05  MIDDLE PICTURE IS X(10).
        05  LAST PICTURE IS x(20).
    02  HOURLY-RATE PICURE IS 99V99.
```

Registros - definição

- Ada, os registros são definidos de uma forma mais ortogonal

```
type Employee_Name_Type is record
  First: String(1..20);
  Middle: String(1..10);
  Last: String(1..20);
end record;

type Employee_Record_Type is record
  Employee_Name: Employe_Name_Type;
  Hourly_Rate: Float;
end record;
```

Registros - definição

- Em Lua, os registros podem ser simulados com o tipo table

```
employee = {}
employee.name = "Freddie"
employee.hourlyRate = 13.20
```

Registros - referência aos campos

- Cobol: `nome_do_campo of nome_do_registro_1 ... nome_do_registro_n`
- Demais linguagens: `registro.campo` (Fortran usa %)
- Referência totalmente especificada: inclui todos os nomes dos registros
- Referências elípticas: nem todos os nomes dos registros precisam ser especificados, deste que não haja ambiguidade
  - Cobol: `FIRST`, `FIRST of EMPLOYEE-NAME`, `FIRST of EMPLOYEE-RECORD`

Registros - operações

- Atribuição (cópia) - C/C++, Ada
- Comparação por igualdade - Ada
- Outras?
  - Exibição para depuração
  - Cálculo de hash
  - Ordem relativa

Registros - avaliação

- As referências elípticas do Cobol são ruins para a legibilidade
- Usado quando uma coleção de valores heterogêneos são necessários
- Acesso a um elemento de um registro é rápido, pois o deslocamento do início do registro é estático

Registros - implementação

- Os campos do registro são armazenados em células adjacentes de memória
- A cada campo é associado o deslocamento relativo ao início do registro

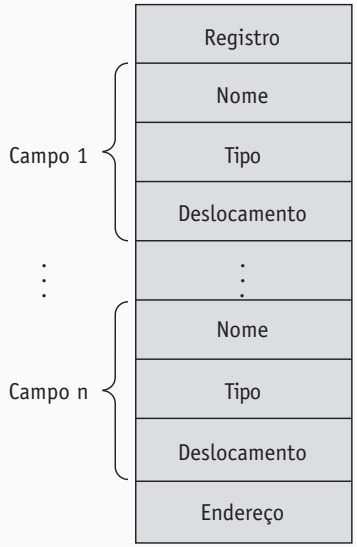

Figura 6.7 Um descritor em tempo de compilação para um registro.

Tuplas

- Um tupla é semelhante a um registro, mas os elementos não são nomeados
- Em geral usados para retornar múltiplos valores

Tuplas - exemplo Python

```
> x = (1, 3.4, 'casa')
> x[0]
1
> x[2]
'casa'
> a, b, c = x
> y = zip([1, 2, 3], [4, 5, 6])
> next(y)
(1, 2)
> for a, b in y: print(a, b)
2 5
3 6
```

Tuplas - exemplo ML

```
val myTuple = (3, 5.8, 'apple');
#1(myTuple);
```

Tuplas - exemplo Rust

```rust
let p = ("casa", 20);
assert_eq!(p.0, "casa");
assert_eq!(p.1, 20);
```

Uniões

Uniões

- **União** é um tipo cujo as variáveis podem armazena valores com diferentes tipos durante a execução do programa
- Aplicações
  - Programação de sistemas: o mesmo conjunto de bytes pode ser interpretado de maneiras diferentes em momentos diferentes
  - Representar conjuntos alternativos de campos em um registro
- Questões de projeto
  - Deve haver checagem de tipo?
  - As uniões devem ficar dentro de registros?

Uniões

- Uniões livres não fornecem suporte a checagem de tipo
  - Fortran, C, C++, Rust

Uniões - exemplo C

```
union Integer {
    int small;
    BigInteger *large;
};
union Integer x;
x.small = 10;
...
// sem verifcação
// o programador tem que garantir que x.large é valido
BigInteger *y = x.large;
```

Uniões

- Uniões discriminadas fornecem suporte a checagem de tipo
  - Algol 68, Pascal, Ada, ML, F#, Rust, etc

Uniões - exemplo Ada

```
type Shape is (Circle, Triangle,
               Rectangle);
type Colors is (Red, Green, Blue);
type Figure (Form : Shape) is record
    Filled : Boolean;
    Color : Colors;
    case Form is
    when Circle =>
        Diameter : Float;
    when Triangle =>
        Left_Side : Integer;
        Right_Side : Integer;
        Angle : Float;
    when Rectangle =>
        Side_1 : Integer;
        Side_2 : Integer;
    end case;
end record;
```

```
-- pode assumir qualquer forma
Figure_1 : Figure;

-- só pode ser um Triangle
Figure_2 : Figure(Form => Triangle);
...
Figure_1 := (Filled => True,
             Color => Blue,
             Form => Rectangle,
             Side_1 => 12,
             Side_2 => 3);

...
-- checado em tempo de execução
-- se o Firgure_1.Form não
-- for Circle, um erro é gerado
if (Figure_1.Diameter > 3.0) ...
```

Uniões - exemplo Rust

```rust
enum Shape {
    Circle {
        diameter: f64,
    },
    Triangle {
        left_side: f64,
        righ_side: f64,
        angle: f64,
    },
    Rectangle {
        side_1: f64,
        side_2: f64,
    },
}
enum Color { Red, Green, Blue }
struct Figure {
    filled: bool,
    color: Color,
    shape: Shape,
}
```

```rust
let fig1 = Figure {
    filled: false,
    color: Color::Blue,
    shape: Shape::Rectangle {
        side_1: 12.0,
        side_2: 3.0,
    },
};

// ...

let area = match &fig1.shape {
    Shape::Circle { .. } =>
        /* código */,
    Shape::Triangle { .. } =>
        /* código */,
    Shape::Rectangle { side_1,
                       side_2 } =>
        side_1 * side_2,
};
```

Uniões - implementação

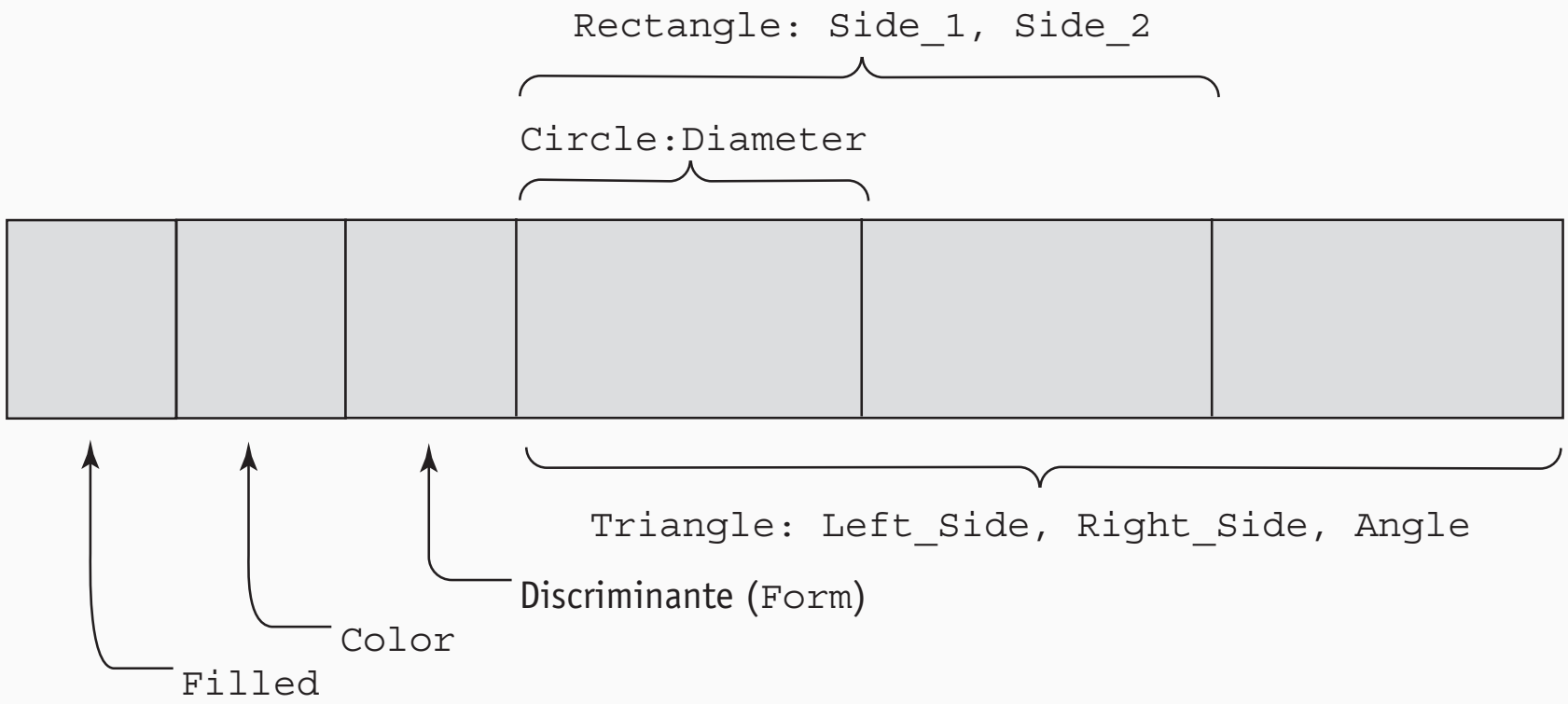

Figura 6.8 Uma união discriminada de três variáveis do tipo Shape (assuma que todas as variáveis são do mesmo tamanho).

Uniões - avaliação

- Uniões livres não são seguras, mas são necessárias para programação de sistemas
- Uniões discriminadas do Ada são mais seguras
- Uniões discriminadas de Rust, Haskell, ML e F# são completamente seguras
- Java e C# não oferecem suporte a uniões
- Em linguagens com suporte a programação orientada a objetos, a herança com polimorfismo é uma alternativa (para conjuntos de campos alternativos)

Ponteiros e referências

Ponteiros e referências

- Um **ponteiro** é um tipo cuja as variáveis podem armazenar valores de memória ou um valor especial nil
- Usos
  - Endereçamento indireto
  - Alocação dinâmica de memória
- Categorias de variáveis
  - Tipos referência
  - Tipos valor

Ponteiros e referências

- Questões de projeto
  - Qual é o escopo e o tempo de vida de um ponteiro?
  - Qual é o tempo de vida das variável dinâmicas no heap?
  - Os ponteiros são restritos ao tipo de valores que eles podem apontar?
  - Os ponteiros são usados para alocação dinâmica, endereçamento indireto ou ambos?
  - A linguagem deve suportar tipos ponteiros, tipos referências, os ambos?

Ponteiros e referências

- Operações
  - Atribuição: define um valor de endereço útil
  - Desreferenciamento
    - Retorna o valor armazenado no local indicado pelo valor do ponteiro
    - Implícito
    - Explícito
  - Aritmética (C/C++)

Ponteiros e referências

- Problemas com ponteiros
  - Ponteiro pendente: O valor do ponteiro aponta para uma variável dinâmica no heap que foi desalocada
    - Como criar um ponteiro pendente?
  - Vazamento de memória: Uma variável dinâmica na pilha não é mais acessível no programa (lixo)
    - Como criar lixo?

Ponteiros e referências

- Exemplo de C/C++
  - Bastante flexível, usando para endereçamento indireto e para gerenciamento de memória
  - Ponteiros podem apontar para qualquer variável, independente de quando ou de onde ela foi alocada
  - Desreferenciamento explícito (operador *)
  - Ponteiros "sem tipo" (`void *`)

```
int a = 10
int *p = &a; // aponta para variável alocada na pilha
p = malloc(sizeof(int)); // aponta para varíavel dinâmica na pilha
...
int list[10];
p = list;          // equivalente a &list[0]
int x = *(p + 3); // equivalente a p[3] e list[3]
```

Ponteiros e referências

- Uma **referência** é um tipo cuja as variáveis referem-se a objetos e valores
- Diferença em relação a ponteiros: como ponteiros armazenam endereços de memória, é possível fazer operações aritméticas

Ponteiros e referências

- Referências em C++
  - Uma referência em C++ é um ponteiro constante que é desreferenciado implicitamente
  - Usado principalmente na passagem de parâmetros

```
int x = 0;
int &ref = x;
ref = 10; // altera o valor de x
```

Ponteiros e referências

- Referências em Java
  - Versão estendida das referências em C++
  - Substitui o uso de ponteiros
  - Todas as instâncias de classe em Java são referenciados por variáveis do tipo referência
  - Gerência automática de memória
- Referências em C#
  - Inclui as referências do Java e os ponteiros do C++

Ponteiros e referências

- Smalltalk, Ruby, Lua
  - Todos os valores são acessados através de referências
  - Não é possível acessar o valor diretamente

Ponteiros e referências

- Implementação
  - Soluções para o problema de ponteiro pendente
    - Tombstones (lápides)
    - Travas e chaves
    - Não permitir desalocação explícita
  - Soluções para o problema de vazamento de memória
    - Contagem de referências
    - Coletor de lixo (marcação e varredura - mark-swep)

Ponteiros e referências

- Avaliação
  - Ponteiros pendentes e vazamento de memória são problemas
  - O gerenciamento do heap é difícil
  - Ponteiros e referências são necessários para estrutura de dados dinâmicas
  - As referências de Java e C# oferecem algumas das capacidades dos ponteiros, mas sem os problemas

Verificação de tipos, tipificação forte e equivalência de tipos

Verificação de tipos, tipificação forte e equivalência de tipos

- Vamos generalizar o conceito de operadores e operandos para incluir atribuição e subprogramas
- **Verificação de tipo** é a atividade de garantir que os operandos de um operador são de tipos compatíveis
- Um **tipo compatível** ou é um tipo legal para o operador, ou pode ser convertido implicitamente pelas regras da linguagem, para o tipo legal
  - Conversão automática é chamada de coerção
- Um **erro de tipo** é a aplicação de um operador a operandos de tipos inapropriados

Verificação de tipos, tipificação forte e equivalência de tipos

- Uma linguagem é **fortemente tipada** se os erros de tipos são sempre detectados
  - Se a vinculação do tipo é estática, quase todas as verificações de tipo podem ser estáticas
  - Se a vinculação de tipo é dinâmica, a verificação de tipo deve ser feita em tempo de execução
- Vantagens da verificação de tipos
  - Permite a detecção do uso incorreto de variáveis que resultaria em erro de tipo

Verificação de tipos, tipificação forte e equivalência de tipos

- Exemplos
  - Fortran 95 menos fortemente tipada: parâmetros, `equivalence`
  - C/C++ menos fortemente tipada: uniões
  - Ada é quase fortemente tipada
  - Java e C# são semelhantes a Ada
  - ML e Haskell são fortemente tipadas
- As regras de coerção afetam o valor da verificação de tipos

Verificação de tipos, tipificação forte e equivalência de tipos

- Equivalência de tipos
  - Dois tipos são equivalentes se um operando de um tipo em uma expressão puder ser substituído por um de outro tipo, sem coerção
  - Por nome
  - Por estrutura
  - Linguagens orientadas a objetos serão discutidas em outro momento

Verificação de tipos, tipificação forte e equivalência de tipos

- Por nome
  - Duas variáveis tem tipos equivalentes se elas foram declaradas na mesma sentença ou em declarações que usam o mesmo nome do tipo
  - Fácil de implementar, mas oferece restrições
    - Subintervalos de inteiros não são equivalentes a inteiros

Verificação de tipos, tipificação forte e equivalência de tipos

- Por estrutura
  - Duas variáveis tem tipos equivalentes se os seus tipos tem a mesma estrutura
  - Mais flexível, mas difícil de implementar
  - Questões
    - Dois registros com a mesma estrutura mas com nomes dos campos diferentes, são equivalentes?
    - Arranjos com o mesmo tamanho, mas faixa de índice diferentes são equivalentes?
  - Não permite a diferenciação entre tipos com a mesma estrutura mas nomes diferentes

Referências

- Robert Sebesta, Concepts of programming languages, 9ª edição. Capítulo 6.