

Tipos de dados

Marco A L Barbosa
malbarbo.pro.br

Departamento de Informática
Universidade Estadual de Maringá



Este trabalho está licenciado com uma Licença Creative Commons - Atribuição-CompartilhaIgual 4.0 Internacional.
<http://github.com/malbarbo/na-lp-copl>

Conteúdo

Introdução

Tipos primitivos

Cadeia de caracteres

Tipo ordinal definido pelo usuário

Arranjos e registros

Tuplas e listas

Uniões

Ponteiros e referências

Verificação de tipos, tipificação forte e equivalência de tipos

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

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

- Sistema de tipos
 - Define como um tipo é associada a uma expressão
 - Inclui 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

- 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 boolean 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 string de tamanho 1

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?

- 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 null
 - 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?

- 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 Days {Mon, Tue, Wed, Thu, Fri, Sat, Sun};
```

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

- 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

- `int red = 0, blue = 1;`

- Problemas?

- C / C++

- As constantes enumeradas só podem aparecer em um tipo

```
enum colors {red, blue, green, yellow, black};
```

```
colors myColor = blue;
```

```
int c = myColor; // válido em C e C++
```

```
myColor++;      // ilegal em C++, legal em C
```

```
myColor = 4;    // ilegal em C++, legal em C
```

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

- 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
- Exemplo em Ada

```
type Days is (Mon, Tue, Wed, Thu, Fri, Sat, Sun);
subtype Weekdays is Days range Mon..Fri;
subtype Index is Integer range 1..100;
Day1 : Days;
Day2 : Weekdays;
...
Day2 := Day1;
```

- A atribuição `Day2 := Day1` é verificada em tempo de execução e só será válida se `Day1` estiver no intervalo `Mon..Fri`.

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

Arranjos e registros

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

- 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?

- Índices

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

`array_name(indices) -> element`

- Sintaxe da seleção

- Fortran e Ada: `Sum := Sum + B(I)`

- Maioria das linguagens: `Sum := Sum + B[I]`

- Índices

- Tipos dos índices

- Maioria da linguagens: inteiros
 - Ada: qualquer tipo ordinal

```
type Week_Day_Type is (Monday, Tuesday, Wednesday,  
                      Thursday, Friday);  
type Sales is array (Week_Day_Type) of Float;
```

- 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

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
- 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
- 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

- C++ suporta os 5 tipos

```
void f(int n) {  
    static int a[10];  
    int b[10];  
    int c[n];  
    int *d = new int[n];  
    vector<int> e;  
    e.push_back(52); // aumenta o tamanho em 1  
    delete[] d;  
}
```


- 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

- Inicialização

- Fortran

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

- C, C++, Java, C#

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

- 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

- Operações
 - Atribuição
 - Comparação de igualdade
 - Fatias
- Exemplos
 - 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

Arranjos

- Um **arranjo regular** é um arranjo multidimensional onde todas as linhas tem o mesmo número de elementos, todas as colunas tem o mesmo número de elementos...
- Um **arranjo irregular** tem linhas (colunas, etc), com um número variado de elementos
 - Em geral quando arranjos multidimensionais são arranjos de arranjos
- Exemplos
 - Arranjo irregular (C, C++, Java)
`myArray[3][7]`
 - Arranjo regular (Fortran, Ada, C#)
`myArray[3, 7]`

- 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 matrix, a coluna 2 pode ser referenciada por `mat(:,2)`

- 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 associativos

- Implementação
 - A função de acesso mapeia os índices para um endereço do arranjo
 - Arranjos unidimensionais
 - $\text{address}(\text{list}[k]) = \text{address}(\text{list}[\text{lower_bound}]) + ((k - \text{lower_bound}) * \text{element_size})$
 - Se o limite inferior for 0, $\text{address}(\text{list}[k]) = \text{address}(\text{list}[0]) + k * \text{element_size}$
 - Arranjos multidimensionais regulares
 - Ordenados por linhas
 - Ordenados por colunas
 - O tipo da implementação tem alguma influência na execução do programa?

Arranjos associativos

- Um **arranjo associativo** é um agregado de elementos indexados através de uma chave. Normalmente conhecido como tipo hash

- Exemplo Perl

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

- Outras linguagens que possuem arranjos associativos
 - PHP, Lua, Python, Javascript (primitivo)
 - C# e C++ (bibliotecas)
- Como os arranjos associativos podem ser implementados?

- 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?

- Definição de registros
 - 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.
```

- Definição de registros
 - 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: Employee_Name_Type;
  Hourly_Rate: Float;
end record;
```

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

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

- 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, desde que não haja ambiguidade
 - Cobol: FIRST, FIRST of EMPLOYEE-NAME, FIRST of EMPLOYEE-RECORD

- Operações
 - Atribuição (cópia) - C/C++, Ada
 - Comparação por igualdade - Ada

- 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

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

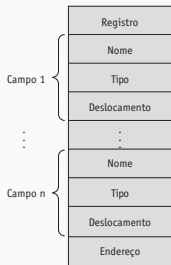


Figura 6.7 Um descritor em tempo de compilação para um registro.

Tuplas e listas

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

- Python

```
> x = (1, 3.4, 'casa')
```

```
> x[0]
```

```
1
```

```
> x[2]
```

```
'casa'
```

```
> a, b, c = x
```

- ML

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

```
#1(myTuple); (* primeiro elemento *)
```

```
type intReal = int * real;
```

- Suportado primeiramente em Lisp
- Comum nas linguagens imperativas atuais

- Lisp (imutável)
 - Elementos podem ser de tipos diferentes
 - Literal: `'(1 2 5 8)`, vazio `null`
 - Construção: `(cons 0 '(1 2))` resulta em `'(0 1 2)`
 - Cabeça: `(car '(1 2 3))` resulta em `1`
 - Cauda: `(cdr '(1 2 3))` resulta em `'(2 3)`

- ML (imutável)
 - Elementos precisam ser do mesmo tipo
 - Literal: `[1, 2, 5, 8]`, vazio `[]`
 - Construção: `0 :: [1, 2]` resulta em `[0, 1, 2]`
 - Cabeça: `hd [1, 2, 3]` resulta em `1`
 - Cauda: `tl [1, 2, 3]` resulta em `[2, 3]`

- Python (mutável)
 - Arranjo dinâmico
 - *List comprehensions*, usado para criar listas usando uma notação semelhante a de conjuntos

```
> a = [4, 10, -6]
```

```
> a[0]
```

```
4
```

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

Uniões

- **União** é um tipo cujo as variáveis podem armazenar 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?

Unões

- Unões livres não fornecem suporte a checagem de tipo.
Exemplos: Fortran, C, C++
- Exemplo C

```
union flexType {  
    int a;  
    float b;  
};  
union flexType x;  
float y;  
...  
x.a = 10;  
y = x.b;
```

- Uniãoes discriminadas fornecem suporte a checagem de tipo.
Exemplos: Algol 68, Pascal, Ada
- Exemplo Ada

```
type 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 Figure_1.Form não
// for Circle, um erro é gerado
if (Figure_1.Diameter > 3.0) ...
```

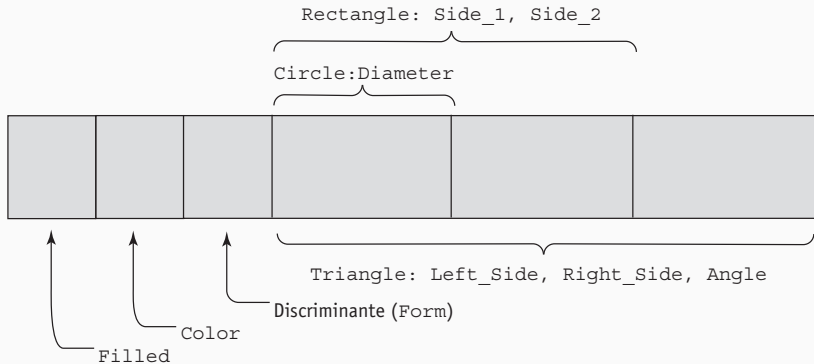


Figura 6.8 Uma união discriminada de três variáveis do tipo `Shape` (assuma que todas as variáveis são do mesmo tamanho).

- F# (semelhante a Haskell e ML)

```
type intReal =  
  | IntValue of int  
  | RealValue of float;;
```

```
let ir1 = IntValue 17;;  
let ir2 = RealValue 3.4;;
```

```
let printType value =  
  match value with  
  | IntValue value -> printfn "It is an integer"  
  | RealValue value -> printfn "It is a float";;
```

```
printType ir1;;  
It is an integer  
printType ir2;;  
It is a float
```

- Avaliação
 - Unões livres não são seguras, mas são necessárias para programação de sistemas
 - Unões discriminadas (como o do Ada) são mais seguras
 - Linguagens funcionais como 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

- Questões de projeto
 - Qual é o escopo e o tempo de vida de um ponteiro?
 - Qual é o tempo de vida das variáveis 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?

- 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++)

- 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 variável dinâmica na pilha
...
int list[10];
p = list; // equivalente a &list[0]
int x = *(p + 3); // equivalente a p[3] e list[3]
```

- 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

- 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
```

- 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++

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

- 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-sweep)

- 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

- 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

- 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 Haskell são fortemente tipadas
- As regras de coerção afetam o valor da verificação de tipos

- Equivalência de tipos
 - Dois tipos são equivalentes se um operando de 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

- 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

- Muitas linguagens usam uma combinação dos dois
- Linguagens de script: duck type

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