Data Types

Dr. Nguyen Hua Phung

HCMC University of Technology, Viet Nam

09, 2014

Outline

Introduction

A data type is

- a homogeneous collection of values and
- a set of operations which manipulate these values

Uses of type system:

- Conceptual organization
- Error detection
- Implementation

Type System

A type system consists of:

- The set of predefined types
- The mechanisms to define a new type
- The mechanisms for the control of types:
  - Type equivalence
  - Type compatibility
  - Type inference
- The specification which type constraints are statically or dynamically checked

Scalar Types

Scalar Types are

- **atomic** không thể chia nhỏ được nữa, thao tác như thực thể hoàn chỉnh
- used to compose another types
- sometimes supported directly by hardware
- booleans, characters, integers, floating-point, fixed-point, complex, void, enumerations, intervals,...

▸ Skip Scalar Types

Integer

Languages may support several sizes of integer

Java's signed integer sizes: byte, short, int, long

Some languages include unsigned integers

Supported directly by hardware: a string of bits

To represent negative numbers: two's complement

kiểu nguyên với kích thước khác nhau

bù 2 (Biểu diễn số âm):
Lấy giá trị tuyệt đối của số âm và chuyển nó sang hệ nhị phân.
Đảo ngược tất cả các bit (0 thành 1, 1 thành 0). Cái này gọi là "one's complement" (bù một).
Cộng 1 vào kết quả vừa đảo ngược. Cái này là "two's complement" (bù hai).

Floating-Point

sấp xỉ

Model real numbers, but only as approximations

Languages for scientific use support at least two floating-point types (e.g., float and double)

Precision and range

float và double: Khác nhau ở tính chính xác và range

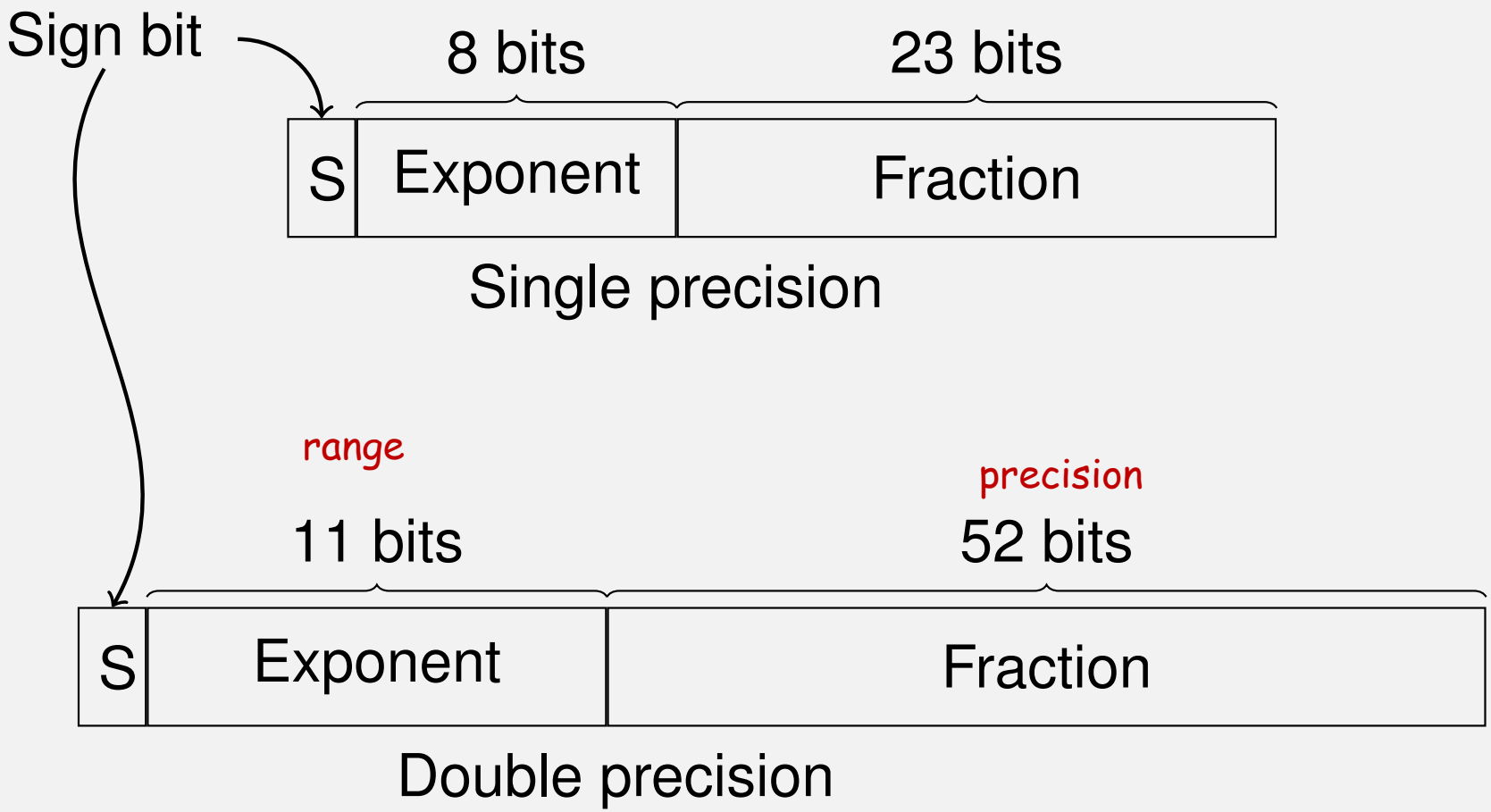
IEEE Floating-Point Standard 754

Ví dụ: convert -126 qua chuỗi 8 bits:
Converse 126 into binary form: 01111110 (make sure that there are 8 bits)

Converse it into one's complement: 10000001 (0->1 and 1->0)

Converse it into two's complement 10000010 (plus 1, keep 8 bits)

IEEE-754


Sign bit
8 bits
23 bits
S
Exponent
Fraction
Single precision
range
precision
11 bits
52 bits
S
Exponent
Fraction
Double precision

-25.45
The sign bit is 1 (vì số âm).
The sequence of bits of integral part (25) is 11001 (5 bits).
Để tìm sequence of bits của fractional part (.45), mình làm như sau:
0.45 * 2 = 0.9 (0)
0.9 * 2 = 1.8 (1)
0.8 * 2 = 1.6 (1)
0.6 * 2 = 1.2 (1)
0.2 * 2 = 0.4 (0)
0.4 * 2 = 0.8 (0)
0.8 * 2 = 1.6 (1)
0.6 * 2 = 1.2 (1)
0.2 * 2 = 0.4 (0)
0.4 * 2 = 0.8 (0)
0.8 * 2 = 1.6 (1)
0.6 * 2 = 1.2 (1)
0.2 * 2 = 0.4 (0)
0.4 * 2 = 0.8 (0)
0.8 * 2 = 1.6 (1)
0.6 * 2 = 1.2 (1)
0.2 * 2 = 0.4 (0)
0.4 * 2 = 0.8 (0)
0.8 * 2 = 1.6 (1)
The sequence of bits of fractional part (.45) is 01110011001100110011001 (23 bits).
The binary form of the above number is 11001.01110011001100110011001...
The standardized binary form of the above number is 1.10010111001100110011001... * 2 ^ 4.
The exponent is 4. Trong single precision, mình cần bias exponent bằng cách cộng thêm 127. Vậy exponent sẽ là 4 + 127 = 131.
Đổi 131 ra binary:
131 chia 2 dư 1
65 chia 2 dư 1
32 chia 2 dư 0
16 chia 2 dư 0
8 chia 2 dư 0
4 chia 2 dư 0
2 chia 2 dư 0
1 chia 2 dư 1
Đọc ngược lại ta được 10000011 (8 bits).
The sequence of bits of the exponent part is 10000011 (8 bits).
The sequence of bits of the above number is 1 10000011 10010111001100110011001 (32 bits).

Decimal

kiểu decimal được sử dụng khi độ chính xác là ưu tiên hàng đầu, đặc biệt là trong các ứng dụng liên quan đến tiền tệ. Nó đảm bảo rằng các phép tính luôn chính xác, nhưng đổi lại là phạm vi biểu diễn số nhỏ hơn và tốn nhiều bộ nhớ hơn.

For business applications (money)

- Essential to COBOL
- C# offers a decimal data type

Store a fixed number of decimal digits

Advantage: accuracy

Disadvantage: limited range, wastes memory

Dành cho ứng dụng kinh doanh (tiền tệ):
Trong các ứng dụng tài chính, độ chính xác là cực kỳ quan trọng. Ví dụ, khi tính toán tiền tệ, bạn không thể chấp nhận sai số nhỏ nhất.
Kiểu decimal được thiết kế để lưu trữ số thập phân một cách chính xác, không bị sai số làm tròn như kiểu float hoặc double.
Quan trọng trong COBOL:
COBOL là một ngôn ngữ lập trình cổ điển, được sử dụng rộng rãi trong các hệ thống tài chính lớn.
COBOL có hỗ trợ mạnh mẽ cho kiểu decimal, vì vậy nó rất phù hợp cho các ứng dụng xử lý tiền tệ.
C# cung cấp kiểu dữ liệu decimal:
Các ngôn ngữ lập trình hiện đại như C# cũng cung cấp kiểu decimal để đáp ứng nhu cầu của các ứng dụng tài chính.
Ưu điểm: Độ chính xác:
Decimal lưu trữ số thập phân một cách chính xác, không bị sai số làm tròn. Điều này đảm bảo rằng các phép tính tiền tệ luôn chính xác.
Nhược điểm:
Phạm vi giới hạn: Decimal có phạm vi biểu diễn số nhỏ hơn so với float hoặc double.
Tốn bộ nhớ: Decimal thường tốn nhiều bộ nhớ hơn so với float hoặc double.

Boolean

Simplest of all

Range of values: two elements, one for “true” and one for “false”

Could be implemented as bits, but often as bytes

Character

Stored as numeric codings

Most commonly used coding: ASCII 8 bits

An alternative, 16-bit coding: Unicode

- Includes characters from most natural languages
- Originally used in Java
- C# and JavaScript also support Unicode

User-Defined Ordinal Types

Kiểu dữ liệu có thứ tự do người dùng định nghĩa

Kiểu dữ liệu thứ tự là gì?
Đơn giản là kiểu dữ liệu mà các giá trị có thể có của nó có thể dễ dàng liên hệ với tập hợp các số nguyên dương (1, 2, 3,...). Nghĩa là mày có thể đánh số thứ tự cho từng giá trị một cách tự nhiên.
Ví dụ trong Java:
int (integer - số nguyên):Các số nguyên như 1, 2, 3, -5, 0,... đều có thứ tự và có thể liên hệ với số nguyên dương (ví dụ: có thể coi số nguyên dương là chính nó, số 0 có thể coi là một điểm bắt đầu, số âm có thể liên hệ một cách gián tiếp).
char (character - ký tự): Mỗi ký tự (ví dụ: 'a', 'b', 'A', '$', '9') đều có một mã số tương ứng (thường là theo bảng mã ASCII hoặc Unicode). Vì có mã số nên chúng ta có thể sắp xếp và đánh thứ tự cho các ký tự. Ví dụ, 'a' có mã 97, 'b' có mã 98, nên 'a' đứng trước 'b'.
boolean (logic - đúng/sai): Kiểu boolean chỉ có hai giá trị là true (đúng) và false (sai). Chúng ta có thể dễ dàng gán cho false giá trị 0 và true giá trị 1 (hoặc ngược lại). Như vậy, nó cũng có thể liên hệ với tập hợp số nguyên dương (hoặc số nguyên không âm).

An ordinal type is one in which the range of possible values can be easily associated with the set of positive integers

Examples of primitive ordinal types in Java

- integer
- char
- boolean

Enumeration Types - Kiểu Liệt Kê

Kiểu liệt kê là một kiểu dữ liệu đặc biệt mà tất cả các giá trị có thể có của nó đã được định nghĩa sẵn và được đặt tên như là các hằng số (named constants).
Ví dụ trong C#:
enum days {Mon, Tue, Wed, Thu, Fri, Sat, Sun};
days myDay = Mon, yourDay = Tue;

All possible values, which are named constants, are provided in the definition

C# example
enum days {Mon, Tue, Wed, Thu, Fri, Sat, Sun};
days myDay = Mon, yourDay = Tue;

Design issues:

- Is an enumeration constant allowed to appear in more than one type definition?
- Are enumeration values coerced to integer?
- Are any other types coerced to an enumeration type?

Các vấn đề thiết kế (Design issues):
Đây là những câu hỏi mà những người thiết kế ngôn ngữ lập trình phải cân nhắc khi tạo ra tính năng kiểu liệt kê:

Is an enumeration constant allowed to appear in more than one type definition? (Một hằng số liệt kê có được phép xuất hiện trong định nghĩa của nhiều kiểu liệt kê khác nhau không?)

Ví dụ: Mày có thể có enum colors {Red, Green, Blue}; và enum trafficLights {Red, Yellow, Green};. Trong trường hợp này, hằng số Red và Green xuất hiện trong cả hai định nghĩa. Một số ngôn ngữ cho phép điều này, một số thì không. Nếu cho phép, cần phải có cách để phân biệt hằng số Red nào đang được nhắc đến (của colors hay trafficLights).
Are enumeration values coerced to integer? (Các giá trị liệt kê có được tự động chuyển đổi thành số nguyên không?) ví dụ: days tomorrow = monday + 1

Trong nhiều ngôn ngữ, các giá trị liệt kê thường được gán một giá trị số nguyên ngầm định (thường bắt đầu từ 0). Ví dụ, trong C#, Mon có thể là 0, Tue là 1, và cứ thế tiếp tục. Việc có cho phép tự động chuyển đổi từ giá trị liệt kê sang số nguyên hay không là một quyết định thiết kế. Nếu có, nó có thể hữu ích trong một số trường hợp (ví dụ, để sử dụng trong mảng hoặc so sánh), nhưng cũng có thể gây ra lỗi nếu không cẩn thận.
Are any other types coerced to an enumeration type? (Có kiểu dữ liệu nào khác được tự động chuyển đổi thành kiểu liệt kê không?) ví dụ: int dayNumber = (int)monday

Câu hỏi này ngược lại với câu trên. Liệu có trường hợp nào một giá trị của kiểu dữ liệu khác (ví dụ, một số nguyên) có thể tự động được coi như là một giá trị của kiểu liệt kê hay không? Điều này thường ít phổ biến hơn việc chuyển đổi từ liệt kê sang số nguyên, vì nó có thể dẫn đến các giá trị không hợp lệ cho kiểu liệt kê. Ví dụ, nếu kiểu days chỉ có giá trị từ Mon đến Sun, thì việc tự động chuyển một số nguyên 10 thành một giá trị của kiểu days sẽ không có ý nghĩa.

Enumeration Type (2)

Readability (Tính dễ đọc):
no need to code a color as a number (không cần mã hóa màu sắc bằng số): Thay vì phải dùng các con số khó nhớ để biểu diễn màu sắc (ví dụ: 1 là đỏ, 2 là xanh), mày có thể dùng các tên có ý nghĩa như Red, Green, Blue trong một kiểu liệt kê colors. Code sẽ dễ đọc và dễ hiểu hơn rất nhiều.

Readability

no need to code a color as a number

Reliability

operations (don't allow colors to be added)

No enumeration variable can be assigned a value outside its defined range ví dụ: const day = "hello" => sai

Better support for enumeration than C++:

enumeration type variables are not coerced into integer types không thể so sánh days[0] và month[0]

Implemented as integers

Reliability (Tính tin cậy):

operations (don't allow colors to be added) (các phép toán - không cho phép cộng màu sắc): Với kiểu liệt kê, ngôn ngữ lập trình thường sẽ hạn chế các phép toán không có nghĩa. Ví dụ, mày không thể thực hiện phép cộng giữa hai giá trị của kiểu colors (ví dụ: Red + Green không có nghĩa). Điều này giúp tránh được những lỗi logic không đáng có.

No enumeration variable can be assigned a value outside its defined range (Không biến liệt kê nào có thể được gán một giá trị nằm ngoài phạm vi đã định nghĩa): Khi mày khai báo một biến có kiểu liệt kê days, mày chỉ có thể gán cho nó các giá trị đã được định nghĩa trong enum days (ví dụ: Mon, Tue, ..., Sun). Mày không thể gán cho nó một giá trị lung tung khác (ví dụ: một số 10 hay một chuỗi "Hello"). Điều này giúp đảm bảo tính hợp lệ của dữ liệu.

Better support for enumeration than C++ (Hỗ trợ kiểu liệt kê tốt hơn C++):

enumeration type variables are not coerced into integer types (các biến kiểu liệt kê không bị ép kiểu ngầm định sang kiểu số nguyên): Trong C++, các biến kiểu liệt kê thường có thể được sử dụng một cách tự do như là các số nguyên. Điều này có thể dẫn đến những lỗi khó phát hiện. Ví dụ, mày có thể vô tình so sánh một giá trị của kiểu colors với một giá trị của kiểu days mà trình biên dịch không báo lỗi. Các ngôn ngữ hiện đại hơn (như C# mà ví dụ ở slide trước đã dùng) thường có sự kiểm tra kiểu chặt chẽ hơn, không cho phép ép kiểu ngầm định như vậy, giúp code an toàn hơn.

Implemented as integers (Được triển khai như là số nguyên):

Mặc dù kiểu liệt kê mang lại nhiều lợi ích về mặt đọc hiểu và độ tin cậy, nhưng ở mức độ bên dưới (khi chương trình chạy), các giá trị liệt kê thường được biểu diễn bằng các số nguyên. Ví dụ, Mon có thể được gán giá trị 0, Tue là 1, và cứ thế tiếp tục. Việc này giúp cho việc lưu trữ và xử lý các giá trị liệt kê được hiệu quả hơn về mặt bộ nhớ và tốc độ. Tuy nhiên, lập trình viên thường không cần quan tâm đến chi tiết triển khai này mà chỉ cần làm việc với các tên hằng số đã được định nghĩa.

Subrange Type - Kiểu con

Kiểu Dữ Liệu Con là gì?

Nó là một dãy các giá trị liên tục và có thứ tự nằm trong một kiểu dữ liệu thứ tự (ordinal type) đã có sẵn.

an ordered contiguous subsequence of an ordinal type

type pos = 0 .. MAXINT;

Subrange types behave as their parent types; can be used as *for* variables and array indices

type sv = array[1 .. 50] of string;

Subrange types are the parent types with code inserted (by the compiler) to restrict assignments to subrange variables

Ví dụ 1: type pos = 0 .. MAXINT;
Khi đó: pos bao gồm tất cả số nguyên từ 0 -> MAXINT, nên nó là một subrange của integer

Hành vi của kiểu dữ liệu con:
Subrange types behave as their parent types; can be used as for variables and array indices (Kiểu dữ liệu con hoạt động giống như kiểu dữ liệu cha của chúng; có thể được sử dụng làm biến và chỉ số mảng):
Điều này có nghĩa là mày có thể khai báo biến có kiểu pos và sử dụng nó như một biến kiểu integer thông thường (nhưng giá trị của nó sẽ bị giới hạn trong khoảng từ 0 đến MAXINT). Tương tự, mày cũng có thể dùng một subrange để định nghĩa kích thước của một mảng.

Ví dụ 2:
type sv = array[1 .. 50] of string;
Trong ví dụ này, 1 .. 50 là một subrange của kiểu số nguyên. Nó được sử dụng để định nghĩa chỉ số của mảng sv. Mảng sv sẽ có 50 phần tử, với chỉ số chạy từ 1 đến 50.

Cách triển khai:
Subrange types are the parent types with code inserted (by the compiler) to restrict assignments to subrange variables (Kiểu dữ liệu con thực chất là kiểu dữ liệu cha với đoạn code được chèn vào (bởi trình biên dịch) để giới hạn việc gán giá trị cho các biến thuộc kiểu dữ liệu con): Về cơ bản, khi mày định nghĩa một subrange type, trình biên dịch sẽ "ghi nhớ" cái giới hạn đó. Khi mày cố gắng gán một giá trị cho một biến có kiểu subrange, trình biên dịch sẽ tự động chèn thêm code để kiểm tra xem giá trị đó có nằm trong phạm vi cho phép hay không. Nếu không, nó sẽ báo lỗi (thường là lỗi runtime).

> Nghĩa là nó vẫn là thằng cha nhưng bị giới hạn, khi gán một kiểu subrange ở ngoài tầm của nó, compiler sẽ check xem nó có ở trong range hợp lệ không, nếu không -> lỗi runtime

Composite Types - Kiểu dữ liệu phức hợp

Đơn giản là một kiểu dữ liệu mà một đối tượng (object) thuộc kiểu đó chứa nhiều thành phần (components) bên trong. Mày có thể truy cập từng thành phần này một cách riêng lẻ.

An object in composite type contains many components which can be accessed individually

component's type may be the same (homogeneous) or different (heterogeneous)

the number of components may be fixed or changed

there may be operations on structured-type object or its components

there may be component insertion/removal operations

there may be creation/destruction operations

Các thành phần bên trong:

component's type may be the same (homogeneous) or different (heterogeneous) (kiểu dữ liệu của thành phần có thể giống nhau (đồng nhất) hoặc khác nhau (dị biệt)): Nghĩa là, một kiểu dữ liệu phức hợp có thể chứa các thành phần có cùng kiểu dữ liệu (ví dụ, một mảng toàn số nguyên) hoặc các thành phần có kiểu dữ liệu khác nhau (ví dụ, một đối tượng chứa tên là chuỗi, tuổi là số nguyên, và địa chỉ là một đối tượng khác).
the number of components may be fixed or changed (số lượng thành phần có thể cố định hoặc thay đổi): Có những kiểu phức hợp mà số lượng thành phần được xác định ngay từ đầu và không thay đổi (ví dụ, một bản ghi (record) với các trường cố định). Cũng có những kiểu phức hợp mà số lượng thành phần có thể tăng hoặc giảm trong quá trình chạy chương trình (ví dụ, một danh sách liên kết hoặc một mảng động).

Các thao tác:

there may be operations on structured-type object or its components (có thể có các thao tác trên đối tượng kiểu cấu trúc hoặc trên các thành phần của nó): Mày có thể thực hiện các thao tác trên toàn bộ đối tượng phức hợp (ví dụ, gán một đối tượng cho một đối tượng khác) hoặc trên từng thành phần riêng lẻ (ví dụ, truy cập vào một phần tử cụ thể trong mảng).
there may be component insertion/removal operations (có thể có các thao tác thêm/xóa thành phần): Đối với các kiểu phức hợp mà số lượng thành phần có thể thay đổi, thường sẽ có các thao tác để thêm một thành phần mới hoặc xóa một thành phần hiện có.
there may be creation/destruction operations (có thể có các thao tác tạo/hủy đối tượng): Giống như bất kỳ kiểu dữ liệu nào khác, các đối tượng thuộc kiểu phức hợp cũng cần được tạo ra (cấp phát bộ nhớ) và hủy bỏ (giải phóng bộ nhớ) khi không còn sử dụng nữa.

một vài ví dụ về kiểu dữ liệu phức hợp trong lập trình:

Mảng (Array): Chứa nhiều phần tử có cùng kiểu dữ liệu (homogeneous), số lượng thường cố định (static array) hoặc có thể thay đổi (dynamic array).

Bản ghi/Struct/Object: Chứa nhiều thành phần có thể có kiểu dữ liệu khác nhau (heterogeneous), số lượng thường cố định.

Danh sách liên kết (Linked List): Chứa các node, mỗi node có thể chứa dữ liệu và một con trỏ đến node tiếp theo. Số lượng node có thể thay đổi.

Tập hợp (Set): Chứa một tập các phần tử duy nhất (thường là cùng kiểu dữ liệu), số lượng có thể thay đổi.

Từ điển/Map/Hash Table: Chứa các cặp khóa-giá trị, các khóa thường có cùng kiểu dữ liệu, còn giá trị có thể khác. Số lượng cặp khóa-giá trị có thể thay đổi.

Array Types

một tập hợp các phần tử dữ liệu mà tất cả các phần tử này phải có cùng kiểu dữ liệu (homogeneous).

Collection of homogeneous data elements

Each element is identified by its position relative to the first element and referenced using subscript expression

array_name (index expression list) $\rightarrow$ *an element*

- What type are legal for subscripts?
  - Pascal, Ada: any ordinal type (integer, boolean, char, enumeration)
  - Others: subrange of integers
- Are subscripting expressions range checked?
  - Most contemporary languages do not specify range checking but Java, ML, C#
  - Unusual case: Perl

▸ Skip Array Type

Truy cập phần tử:
Mỗi phần tử trong mảng được xác định bởi vị trí của nó so với phần tử đầu tiên. Để truy cập một phần tử, người ta thường dùng một biểu thức chỉ số (subscript expression).
Cú pháp truy cập:
array_name (index expression list) → an element
Ví dụ, nếu mày có một mảng tên là myArray, để lấy phần tử ở vị trí thứ 3 (thường chỉ số bắt đầu từ 0), mày có thể viết kiểu như myArray[2] (trong nhiều ngôn ngữ). Cái [2] chính là index expression.
Kiểu dữ liệu nào được phép dùng làm chỉ số?
Đây là chỗ các ngôn ngữ lập trình có thể khác nhau:
Pascal, Ada: Cho phép dùng bất kỳ kiểu dữ liệu thứ tự nào (ordinal type) làm chỉ số. Ví dụ, mày có thể dùng số nguyên, boolean, ký tự, hoặc thậm chí là kiểu liệt kê.
Các ngôn ngữ khác: Thường chỉ cho phép dùng một subrange của kiểu số nguyên làm chỉ số. Ví dụ, chỉ số có thể chạy từ 0 đến n-1, hoặc từ 1 đến n.
Biểu thức chỉ số có được kiểm tra phạm vi không?
Đây là một vấn đề quan trọng liên quan đến tính an toàn của chương trình:
Hầu hết các ngôn ngữ hiện đại không quy định việc kiểm tra phạm vi: Điều này có nghĩa là nếu mày cố gắng truy cập một phần tử nằm ngoài phạm vi chỉ số hợp lệ của mảng (ví dụ, một mảng có 10 phần tử nhưng mày lại cố gắng truy cập phần tử thứ 15), thì ngôn ngữ đó có thể không phát hiện ra lỗi này tại thời gian chạy. Điều này có thể dẫn đến những hành vi không mong muốn hoặc thậm chí là crash chương trình.
Java, ML, C#: Đây là những ngoại lệ. Các ngôn ngữ này thường có cơ chế kiểm tra phạm vi chỉ số tại thời gian chạy. Nếu mày cố gắng truy cập một chỉ số không hợp lệ, nó sẽ quăng ra một ngoại lệ (exception) để báo cho mày biết.
Trường hợp đặc biệt: Perl: Perl có cách xử lý mảng khá linh hoạt và có thể tự động điều chỉnh kích thước mảng khi mày cố gắng truy cập một chỉ số nằm ngoài phạm vi hiện tại.

Subscript Binding and Array Categories

Static

static int x[10];

Fixed Stack-dynamic

int x[10]; //inside a function

Stack-dynamic

cin »n;
int x[n];

Fixed Heap-dynamic

int[] x = new int[10];

Heap-dynamic

cin »n;
int[] x = new int[n];

Array Initialization

Some language allow initialization at the time of storage allocation

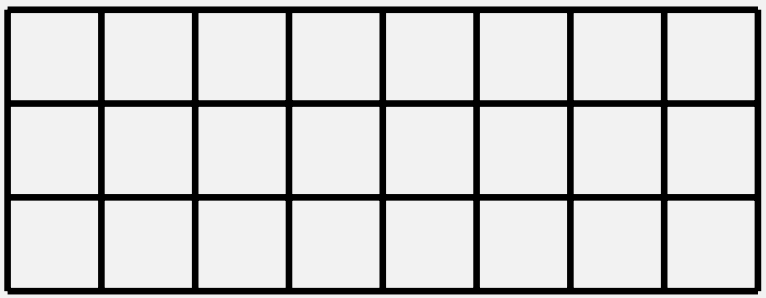
- C, C++, Java, C# example
- *int list [] = {4, 5, 7, 83}*
- Character strings in C and C++
- *char name [] = "freddie";*
- Arrays of strings in C and C++
- *char *names [] = {"Bob", "Jake", "Joe"};*
- Java initialization of String objects
- *String[] names = {"Bob", "Jake", "Joe"};*

Rectangular and Jagged Arrays

C, C++, Java, C#: jagged arrays
myArray[3][7]

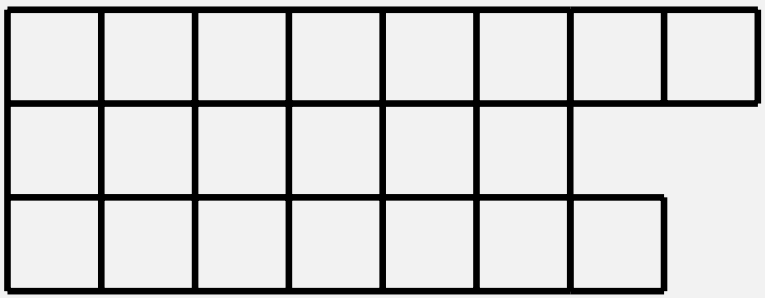
Fortran, Ada, C#: rectangular array
myArray[3,7]

rectangular



Các hàng có kích thước cố định cùng chiều dài, tạo thành hình chữ nhật.
Truy cập: myArrray[3, 7]

jagged



giống như một mảng mà mỗi phần tử của nó lại là một mảng khác, và mấy cái mảng con bên trong đó có thể có độ dài khác nhau (nên mới gọi là "răng cưa").
Truy cập: myArray[3][7]

Slices

Slices là gì?

Đơn giản là một phần (substructure) của một mảng. Nó không phải là một bản sao dữ liệu mới, mà chỉ là một cách để tham chiếu (referencing mechanism) đến một đoạn các phần tử trong mảng gốc thôi.

Khi nào thì Slices hữu ích?

Slices đặc biệt hữu dụng trong các ngôn ngữ mà cho phép mày thực hiện các phép toán trên cả mảng hoặc trên các phần của mảng một cách dễ dàng.

A slice is some substructure of an array; nothing more than a referencing mechanism

Slices are only useful in languages that have array operations

E.g. Python

vector = [2, 4, 6, 8, 10, 12, 14, 16]

mat = [[1, 2, 3],[4, 5, 6],[7, 8, 9]]

vector[3:6], mat[1], mat[0][0:2]

vector[3:6]: Cái này sẽ tạo ra một "lát cắt" của mảng vector, bao gồm các phần tử từ chỉ số 3 (là số 8) đến chỉ số 5 (là số 12). Chỉ số kết thúc (6) không được bao gồm. Vậy kết quả của vector[3:6] sẽ là [8, 10, 12].

Implementation of Arrays

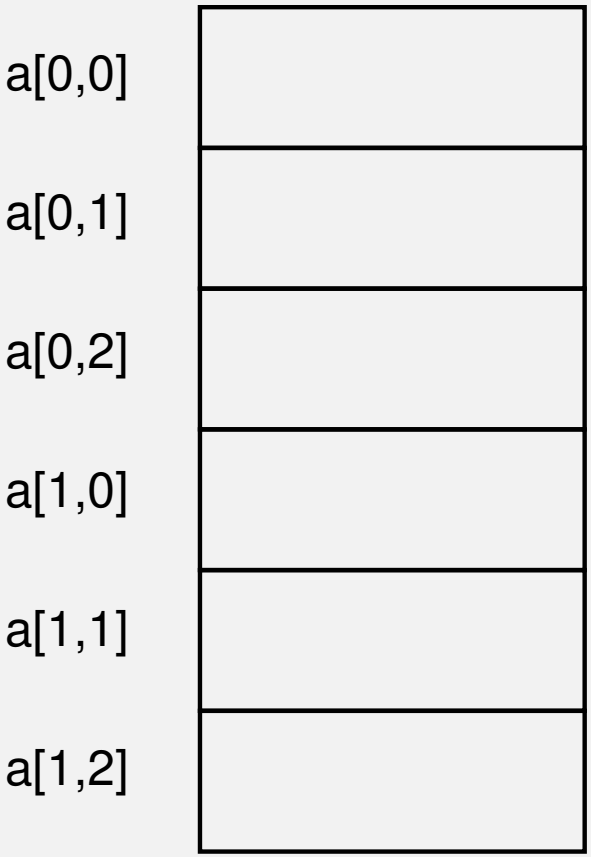
Hàm Truy Cập (Access Function):
Cái hàm này giống như một cái bản đồ chỉ đường cho máy tính biết chính xác cái phần tử mà mày muốn lấy trong mảng nó nằm ở đâu trong bộ nhớ. Nó nhận cái chỉ số (subscript expression) mà mày đưa vào và trả về cái địa chỉ (address) thực tế của ô nhớ chứa cái phần tử đó.
Mảng Một Chiều (Single-dimensioned):
Mảng một chiều thì dễ hình dung nhất. Nó giống như một hàng dài các ô nhớ nằm cạnh nhau, mỗi ô chứa một phần tử của mảng.
Công thức tính địa chỉ:
address(list[k]) = address(list[lower_bound]) + ((k - lower_bound) * element_size)

Access function maps subscript expressions to an address in the array

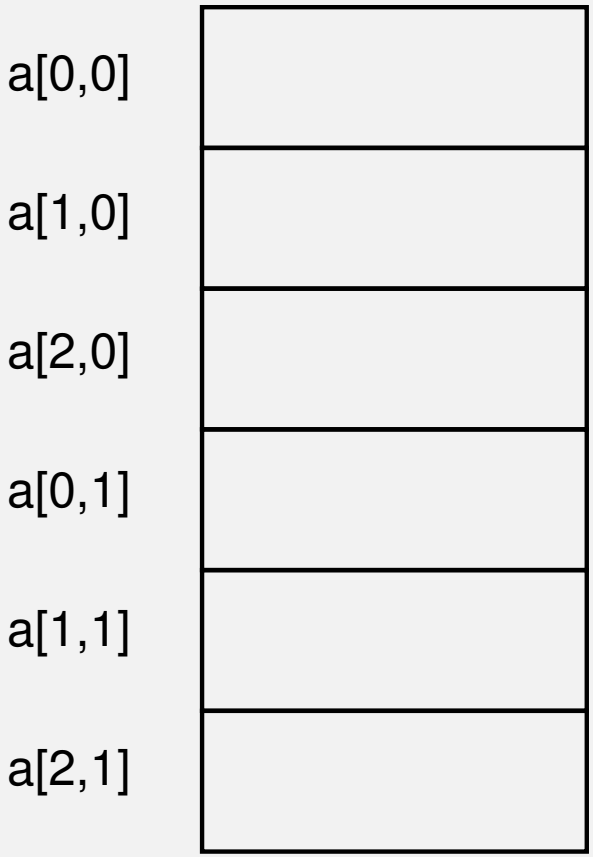
Single-dimensioned: list of adjacent memory cells

Access function for single-dimensioned arrays:
address(list[k]) = address(list[lower_bound]) + ((k-lower_bound) * element_size)

Accessing Two-dimensional Arrays



Row-major order
used in most languages



Column-major order
used in Fortran

Accessing Two-dimensional Arrays

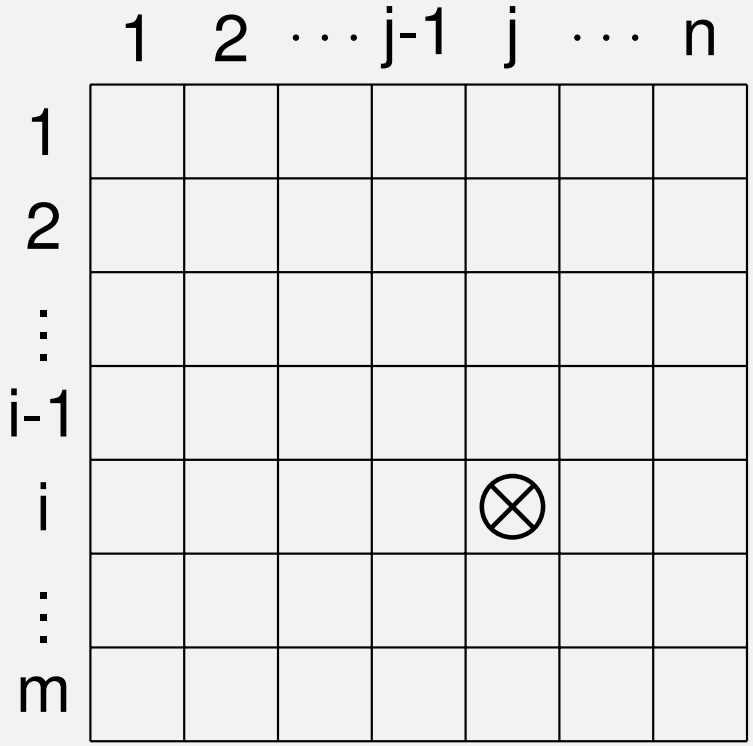
Row-major order:

số lượng phần tử nằm trong tất cả các hàng trước hàng muốn đến

vị trí tương đối của cột muốn đến trong hàng hiện tại

Location (a[i,j]) = α + (((i - row_lb) * n) + (j - col_lb)) * E

where α is address of a[row_lb,col_lb] and E is element size



Location (a[i,j]): Đây là địa chỉ bộ nhớ của phần tử mà mày muốn tìm, nó nằm ở hàng thứ i và cột thứ j trong mảng a.
a: Đây là địa chỉ bộ nhớ bắt đầu của toàn bộ mảng a. Nó chính là địa chỉ của phần tử đầu tiên trong mảng, thường là ở hàng có chỉ số row_lb (lower bound của hàng) và cột có chỉ số col_lb (lower bound của cột).
i - row_lb: Cái này tính ra số hàng mà mày cần "bỏ qua" để đến được hàng thứ i. Ví dụ, nếu hàng bắt đầu từ chỉ số 0 và mày muốn đến hàng 2, thì mày cần bỏ qua 2 hàng (hàng 0 và hàng 1).
n: Đây là số lượng cột trong mỗi hàng của mảng. Trong row-major order, các phần tử của cùng một hàng sẽ nằm liên tiếp nhau trong bộ nhớ.
(i - row_lb) * n: Cái này tính ra số lượng phần tử nằm trong tất cả các hàng trước hàng mà mày muốn đến. Ví dụ, nếu mày muốn đến hàng 2 và mỗi hàng có 5 cột, thì sẽ có (2 - 0) * 5 = 10 phần tử nằm ở hàng 0 và hàng 1.
j - col_lb: Cái này tính ra vị trí tương đối của cột mà mày muốn đến trong hàng hiện tại (hàng thứ i). Ví dụ, nếu cột bắt đầu từ chỉ số 0 và mày muốn đến cột 3, thì nó sẽ cách cột đầu tiên 3 vị trí.
((i - row_lb) * n) + (j - col_lb): Cái này là tổng số phần tử nằm trước phần tử a[i,j] trong bộ nhớ. Nó bao gồm tất cả các phần tử ở các hàng trước và tất cả các phần tử ở các cột trước trong hàng hiện tại
.
E: Đây là kích thước của mỗi phần tử trong mảng (tính bằng byte). Ví dụ, nếu mảng chứa các số nguyên (mỗi số 4 byte), thì E sẽ là 4.
(((i - row_lb) * n) + (j - col_lb)) * E: Cái này tính ra tổng khoảng cách (tính bằng byte) từ địa chỉ bắt đầu của mảng (a) đến địa chỉ của phần tử a[i,j].
Cuối cùng, mày cộng cái địa chỉ bắt đầu (a) với cái tổng khoảng cách vừa tính được, là sẽ ra địa chỉ bộ nhớ chính xác của phần tử a[i,j].

Given the following array declaration:

int x[5][7]; //the lower bound is 0

Assume that the size of an int element is 4, the elements of the array are allocated in row-major order and the starting address of the array is 1000, what is the address of the element **x[3][4]**?

The address of the element x[3][4] is

Location (a[i,j]) = α + (((i - row_lb) * n) + (j - col_lb)) * E

1000 + (((3 - 0)*7 + (4 - 0)) * 4
= 1100

Cách 2:
1000 + (3 * 7 + 4) * 4

1 1 1 1 1 1 1
1 1 1 1 1 1 1
1 1 1 1 1 1 1
1 1 1 1 1 1 1
1 1 1 1 1 1 1

Mảng 3 chiều:
Với mảng 3 chiều x[D][R][C] (D là số lớp/khối, R là số hàng, C là số cột), địa chỉ của phần tử x[i][j][k] (với index bắt đầu từ 0) trong bộ nhớ theo thứ tự hàng (row-major order) sẽ là:

địa_chỉ(x[i][j][k]) = địa_chỉ_bắt_đầu(x) + (i * R * C + j * C + k) * kích_thước_phần_tử

Given the following array declaration:

int x[4][6][5]; // the lower bound is 0

Assume that the size of. an int element is 4, the elements of an array are allocated in row-major order, and the starting address of the variable x is 1000, what is the address of the element x[2][4][3]?

the address of the element x[2][4][3] is

1000 + (2 * 6 * 5 + 4 * 5 + 3) * 4 = 1332

Compile-time Descriptors

(bộ mô tả tại thời điểm biên dịch)

Array
Element type
Index type
Index lower bound
Index upper bound
Address

thường là số nguyên

Địa chỉ bộ nhớ nơi mảng bắt đầu được lưu trữ.

Single dimensional array

Multidimensional array
Element type
Index type
Number of dimensions
Index range 1
:
Index range n
Address

Phạm vi chỉ số cho chiều thứ nhất (từ chỉ số bắt đầu đến chỉ số kết thúc)

Multi-dimensional array

Đối với mảng một chiều (Single dimensional array):

Trình biên dịch sẽ lưu trữ mấy thông tin sau:

Array: Để biết đây là một mảng.
Element type: Kiểu dữ liệu của các phần tử trong mảng (ví dụ: số nguyên, số thực, ký tự,...).
Index type: Kiểu dữ liệu của chỉ số dùng để truy cập các phần tử (thường là số nguyên).
Index lower bound: Chỉ số bắt đầu của mảng (ví dụ: có ngôn ngữ bắt đầu từ 0, có ngôn ngữ bắt đầu từ 1).
Index upper bound: Chỉ số kết thúc của mảng.
Address: Địa chỉ bộ nhớ nơi mảng bắt đầu được lưu trữ.

Đối với mảng đa chiều (Multidimensional array):

Nó cũng tương tự, nhưng có thêm thông tin về số chiều và phạm vi chỉ số cho từng chiều:

Multidimensional array: Để biết đây là mảng đa chiều.
Element type: Kiểu dữ liệu của các phần tử.
Index type: Kiểu dữ liệu của chỉ số cho mỗi chiều.
Number of dimensions: Số lượng chiều của mảng (ví dụ: 2 chiều, 3 chiều,...).
Index range 1: Phạm vi chỉ số cho chiều thứ nhất (từ chỉ số bắt đầu đến chỉ số kết thúc).
... (và tương tự cho các chiều khác)
Index range n: Phạm vi chỉ số cho chiều thứ n.
Address: Địa chỉ bộ nhớ bắt đầu của mảng.

Associative Arrays - Mảng kết hợp

Đây là một kiểu tập hợp dữ liệu (collection) mà nó không quan tâm đến thứ tự của các phần tử (unordered). Điểm đặc biệt là mỗi phần tử trong mảng kết hợp được xác định và truy cập thông qua một giá trị khác, gọi là key (khóa), thay vì chỉ số số nguyên như mảng bình thường. Số lượng keys sẽ bằng với số lượng phần tử dữ liệu.

An *associative array* is an unordered collection of data elements that are indexed by an equal number of values called *keys*

For example,

dt = [("name","John");("age","28");("address","1 John st.")]

dt["name"] $\Rightarrow$ "John"

dt["address"] $\Rightarrow$ "1 John st."

User defined keys must be stored

Similar to Map in Scala

Design issues: What is the form of references to elements

User defined keys must be stored (Các khóa do người dùng định nghĩa phải được lưu trữ): Vì mày tự đặt ra các khóa (ví dụ: "name", "age"), nên hệ thống cần có cách để lưu trữ và quản lý các khóa này cùng với các giá trị tương ứng.

What is the form of references to elements (Hình thức tham chiếu đến các phần tử là gì)? Câu hỏi này đề cập đến cú pháp mà mày dùng để truy cập các phần tử trong associative array. Ví dụ, trong ví dụ trên, người ta dùng cú pháp dt["name"]. Các ngôn ngữ khác nhau có thể có cú pháp khác nhau để truy cập các phần tử bằng khóa.

String Types

Một dãy các ký tự (sequence of characters)

Values are sequences of characters

Design issues:

Một số ngôn ngữ coi chuỗi là một kiểu dữ liệu cơ bản, giống như số nguyên hay số thực. Số khác lại coi nó như là một mảng các ký tự, nhưng có thêm một số tính năng đặc biệt.

- Is it a primitive type or just a special kind of array?
- Should the length of strings be static or dynamic?

Độ dài của chuỗi nên cố định hay thay đổi được?

Typical operations

- Assignment
- Comparison (=, >, etc.)
- Concatenation
- Substring reference
- Pattern matching (regular expression)

Assignment (Gán): Gán một chuỗi cho một biến.
Comparison (=, >, etc.) (So sánh): So sánh hai chuỗi với nhau (bằng nhau, lớn hơn, nhỏ hơn,...).
Concatenation (Nối chuỗi): Ghép hai hay nhiều chuỗi lại thành một chuỗi lớn hơn.
Substring reference (Tham chiếu chuỗi con): Lấy ra một phần của chuỗi (ví dụ: từ ký tự thứ 3 đến ký tự thứ 7).
Pattern matching (regular expression) (So khớp mẫu - biểu thức chính quy): Tìm kiếm một mẫu nhất định trong chuỗi.
String Length (Độ dài chuỗi): Lấy độ dài của chuỗi.

▸ Skip String Type

String Length Options

Static (Cố định): Độ dài của chuỗi được xác định ngay tại thời điểm biên dịch và không thể thay đổi sau đó.
Thường dùng "compile-time descriptor" để lưu thông tin về chuỗi (ví dụ: địa chỉ bắt đầu và độ dài cố định).

Static: String length is fixed at compiling time

- Python, Java String class
- compile-time descriptor

Độ dài của chuỗi có thể thay đổi trong quá trình chạy chương trình, nhưng không được vượt quá một giới hạn nhất định.
Thường dùng "run-time descriptor" để lưu thông tin (ví dụ: địa chỉ bắt đầu, độ dài hiện tại, và dung lượng tối đa).

Limited Dynamic: String length may be changed but less than a limit

- C, C++
- run-time descriptor

Độ dài của chuỗi có thể thay đổi thoải mái trong quá trình chạy chương trình, không có giới hạn nào cả (ngoài giới hạn bộ nhớ).
Cơ chế: Thường dùng "run-time descriptor" và đôi khi có thể dùng cấu trúc dữ liệu phức tạp hơn như linked list để quản lý chuỗi dài
.

Dynamic: String length may be changed without any limit

- Perl, JavaScript
- run-time descriptor; linked list

Ada supports all three string length options

Descriptor

Static string
String length
Address

Compile-time descriptor for static length strings

Limited dynamic string
Maximum length
Current length
Address

Run-time descriptor for limited dynamic length strings

Record Types

Một bản ghi (record) là một tập hợp các phần tử dữ liệu dị biệt (heterogeneous aggregate), nghĩa là các phần tử bên trong nó có thể có kiểu dữ liệu khác nhau. Mỗi phần tử riêng lẻ được xác định bằng một tên (name).

A record:

- heterogeneous aggregate of data elements
- individual elements are identified by names

Popular in most languages, OO languages use objects as records

Kiểu bản ghi rất phổ biến trong hầu hết các ngôn ngữ lập trình. Trong các ngôn ngữ hướng đối tượng (OO), người ta thường dùng đối tượng (objects) theo cách tương tự như bản ghi để lưu trữ dữ liệu.

Design issues:

- What is the syntactic form of references to the field?
- Are elliptical references allowed

▸ Skip Record Type

Các vấn đề thiết kế (Design issues):

What is the syntactic form of references to the field? (Cú pháp để tham chiếu đến một trường (field) là gì?): Ví dụ, mày dùng dấu chấm (.) hay một ký hiệu nào khác để truy cập vào một thành phần bên trong bản ghi?

Are elliptical references allowed (Có cho phép tham chiếu rút gọn không?): Ví dụ, nếu có các bản ghi lồng nhau, mày có cần phải viết đầy đủ đường dẫn để đến một trường bên trong hay có thể bỏ bớt một vài tên nếu nó không gây nhầm lẫn?

Definition of Records in Ada

Record structures are indicated in an orthogonal way

```
type Emp_Name_Type is record
        First: String (1..20);
        Mid: String (1..10);
        Last: String (1..20);
end record;
type Emp_Rec_Type is record
        Emp_Name: Emp_Name_Type;
        Hourly_Rate: Float;
end record;
Emp_Rec: Emp_Rec_Type;
```

Emp_Name_Type: Để lưu trữ tên nhân viên, gồm tên đầu (First), tên đệm (Mid), và tên cuối (Last), mỗi cái là một chuỗi có độ dài tối đa.
Emp_Rec_Type: Để lưu trữ thông tin về một nhân viên, bao gồm tên (sử dụng kiểu Emp_Name_Type vừa định nghĩa) và mức lương theo giờ (Hourly_Rate).
Sau đó, nó khai báo một biến Emp_Rec có kiểu Emp_Rec_ Type.

References to Records

Notation:

Dot-notation (Ký hiệu dấu chấm): Đây là cách phổ biến nhất, giống như trong nhiều ngôn ngữ khác. Ví dụ: Emp_Rec.Emp_Name.Mid để truy cập vào tên đệm của nhân viên trong bản ghi Emp_Rec.

- Dot-notation: *Emp_Rec.Emp_Name.Mid*
- Keyword-based:
  Mid OF Emp_Name OF Emp_Rec

Format:

- **Fully qualified references**: include all record names
- **Elliptical references**: may leave out some record names as long as reference is unambiguous
  Mid, Mid OF Emp_Name, Mid OF Emp_Rec

Ada còn cho phép dùng từ khóa OF. Ví dụ: Mid OF Emp_Name OF Emp_Rec. Cách này có thể giúp code dễ đọc hơn trong một số trường hợp.

Operations in Records

Assignment is very common if the types are identical

Ada allows record comparison

Ada records can be initialized with aggregate literals

COBOL provides MOVE CORRESPONDING
Copies fields which have the same name

Assignment (Gán): Rất phổ biến, thường được cho phép nếu hai bản ghi có cùng kiểu.
Comparison (So sánh): Ada cho phép so sánh trực tiếp hai bản ghi với nhau (nếu các kiểu dữ liệu bên trong hỗ trợ so sánh).
Initialization (Khởi tạo): Ada cho phép khởi tạo bản ghi bằng cách cung cấp các giá trị cho từng trường một cách tường minh (aggregate literals).
COBOL's MOVE CORRESPONDING: Đây là một thao tác đặc biệt có trong ngôn ngữ COBOL. Nó sẽ tự động sao chép các trường có cùng tên giữa hai bản ghi (ngay cả khi chúng có thể nằm ở các vị trí khác nhau trong định nghĩa của hai bản ghi).

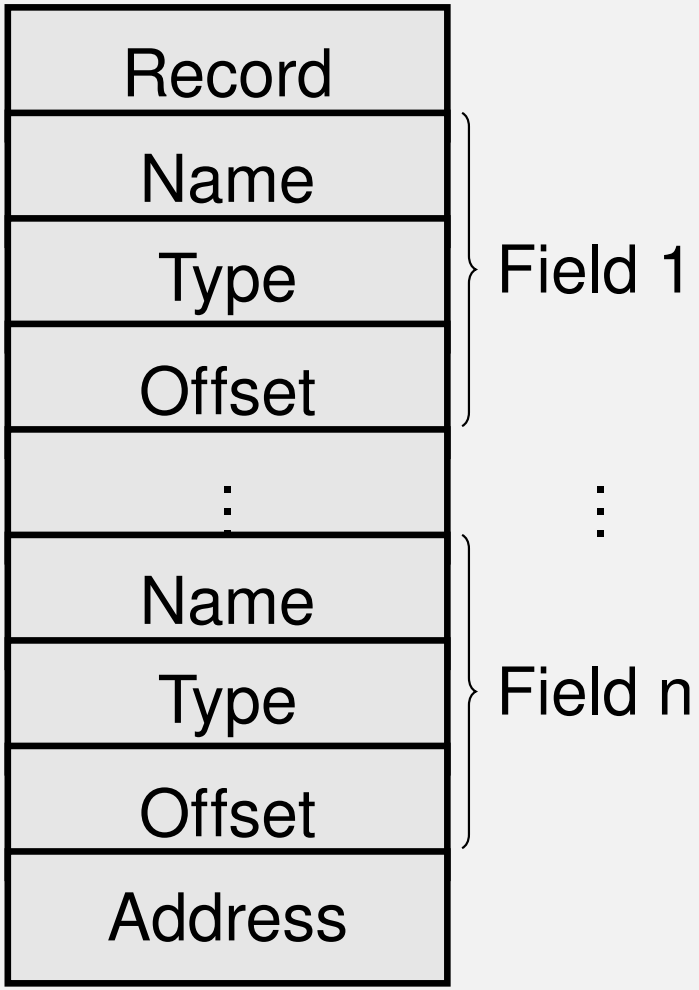
Straight forward and safe design

Comparison of arrays and records

Arrays	Records
homogenous	heterogeneous
elements are processed in the same way	elements are processed in different way
dynamic subscripting	static subscripting

Đồng nhất vs. Dị biệt: Mảng giống như một dãy các ngăn tủ, mỗi ngăn chứa cùng một loại đồ (ví dụ, toàn là sách). Bản ghi giống như một cái hồ sơ nhân viên, có thể chứa tên (chuỗi), tuổi (số nguyên), địa chỉ (chuỗi),... mỗi thông tin có một kiểu khác nhau.
Cách xử lý: Khi làm việc với mảng, mày thường duyệt qua từng phần tử và thực hiện cùng một thao tác (ví dụ, tăng giá trị của tất cả các phần tử lên 1). Với bản ghi, mày thường truy cập và xử lý từng trường riêng lẻ theo mục đích của nó (ví dụ, in ra tên nhân viên, tính lương dựa trên mức lương theo giờ).
Chỉ số động vs. Tĩnh: Với mảng, mày có thể dùng một biến để chỉ định phần tử muốn truy cập (ví dụ, myArray[i], giá trị của i có thể thay đổi khi chương trình chạy). Với bản ghi, mày thường dùng tên cố định của trường để truy cập (ví dụ, employee.name, employee.age). Cái tên trường đó thường đã được xác định tại thời điểm viết code (tĩnh).

Implementation of Record Type



Data Alignment

Căn chỉnh: Lý do chính là để tăng tốc độ truy cập dữ liệu. CPU thường truy cập bộ nhớ theo các khối có kích thước nhất định (ví dụ: 4 byte, 8 byte). Nếu dữ liệu không được canh chỉnh đúng cách, CPU có thể phải thực hiện nhiều lần truy cập bộ nhớ để lấy một biến, làm chậm chương trình.

b-byte aligned

A b-byte aligned object has an address that is a multiple of b bytes.

b-byte aligned (canh chỉnh theo b byte): Nghĩa là một đối tượng (ví dụ một biến) có địa chỉ bộ nhớ là một bội số của b byte.

Example

1. A **char** (one byte) will be 1-byte aligned.
2. A **short** (two bytes) will be 2-byte aligned.
3. A **int** (four bytes) will be 4-byte aligned.
4. A **long** (four bytes) will be 4-byte aligned.
5. A **float** (four bytes) will be 4-byte aligned.

LẤY KÍCH THƯỚC CỦA THẰNG LỚN NHẤT

Padding

when a structure member is

- followed by a member with a larger alignment requirement, or
- at the end of the structure to make the structure size be multiple of the biggest member size.

```
struct MyStruct {
    char data1;
    int data2;
    char data3;
    short data4;
    char data5;
};
```

What is the size of the above struct?

Data structure Padding (Đệm Dữ Liệu Cấu Trúc):

Padding (Đệm): Là việc trình biên dịch (compiler) tự động chèn thêm các khoảng trống (byte "rác") vào giữa các thành viên của một cấu trúc (struct) hoặc ở cuối cấu trúc.

Khi nào thì cần padding?

Khi một thành viên có yêu cầu canh chỉnh lớn hơn đứng sau một thành viên có yêu cầu canh chỉnh nhỏ hơn. Trình biên dịch sẽ chèn padding để đảm bảo thành viên sau bắt đầu ở một địa chỉ thỏa mãn yêu cầu canh chỉnh của nó.

Ở cuối cấu trúc để làm cho kích thước của toàn bộ cấu trúc là một bội số của kích thước của thành viên lớn nhất trong cấu trúc. Điều này giúp cho việc tạo mảng các cấu trúc được hiệu quả hơn.

Để tính kích thước, mình phải xem xét yêu cầu canh chỉnh của từng thành viên và padding cần thiết:

char data1: Kích thước 1 byte, canh chỉnh 1 byte.
int data2: Kích thước 4 byte, canh chỉnh 4 byte. Để data2 bắt đầu ở địa chỉ là bội số của 4, cần thêm 3 byte padding sau data1.
char data3: Kích thước 1 byte, canh chỉnh 1 byte.
short data4: Kích thước 2 byte, canh chỉnh 2 byte. Để data4 bắt đầu ở địa chỉ là bội số của 2, cần thêm 1 byte padding sau data3.
char data5: Kích thước 1 byte, canh chỉnh 1 byte.
Tổng kích thước hiện tại là: 1 (data1) + 3 (padding) + 4 (data2) + 1 (data3) + 1 (padding) + 2 (data4) + 1 (data5) = 13 byte.

Union Types - Kiểu Hợp Nhất

A union is a type whose variables are allowed to store different type values at different times during execution

Kiểu hợp nhất là một kiểu dữ liệu mà biến của nó có thể lưu trữ các giá trị thuộc các kiểu liệu khác nhau tại các thời điểm khác nhau trong quá trình chạy chương trình.

```
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 =>
            Leftside, Rightside: Integer;
            Angle: Float;
        when Rectangle => Side1, Side2: Integer;
    end case;
end record;
```

liệt kê định nghĩa các hình dạng có thể có.

liệt kê định nghĩa các màu sắc có thể có.

Đây là định nghĩa của kiểu hợp nhất Figure. Cái (Form: Shape) này gọi là discriminant (bộ phân biệt), nó cho biết hình dạng hiện tại của Figure.

Đây là phần quan trọng của union. Tùy thuộc vào giá trị của Form, bản ghi Figure sẽ có các trường khác nhau:
Nếu Form là Circle, nó sẽ có thêm trường Diameter (đường kính).
Nếu Form là Triangle, nó sẽ có thêm các trường Leftside, Rightside (độ dài hai cạnh), và Angle (góc).
Nếu Form là Rectangle, nó sẽ có thêm các trường Side1 và Side2 (độ dài hai cạnh).

▸ Skip Union Type

Given the following record declaration in Ada:

```
                                1 enumeration
type Shape is (Circle, Triangle, Rectangle);
type Colors is (Red, Green, Blue);1 enumeration
type Figure (Form: Shape) is record  2
 Filled: Boolean;        1
 Color: Colors;    2
case Form is
  when Circle => Diameter: Float;
  when Triangle =>
     Leftside, Rightside: Integer;    2+2
     Angle: Float;     4
   when Rectangle => Side1, Side2: Integer;
 end case;
end record;
```

Assume that the size of Boolean, enumeration, Integer, and Float are 1, 2, 2 and 4, respectively. What is the size of an object in type Figure without padding?

The size of an object in type Figure without padding is

13

Given the following declaration of a set:

x: set of 10..73;

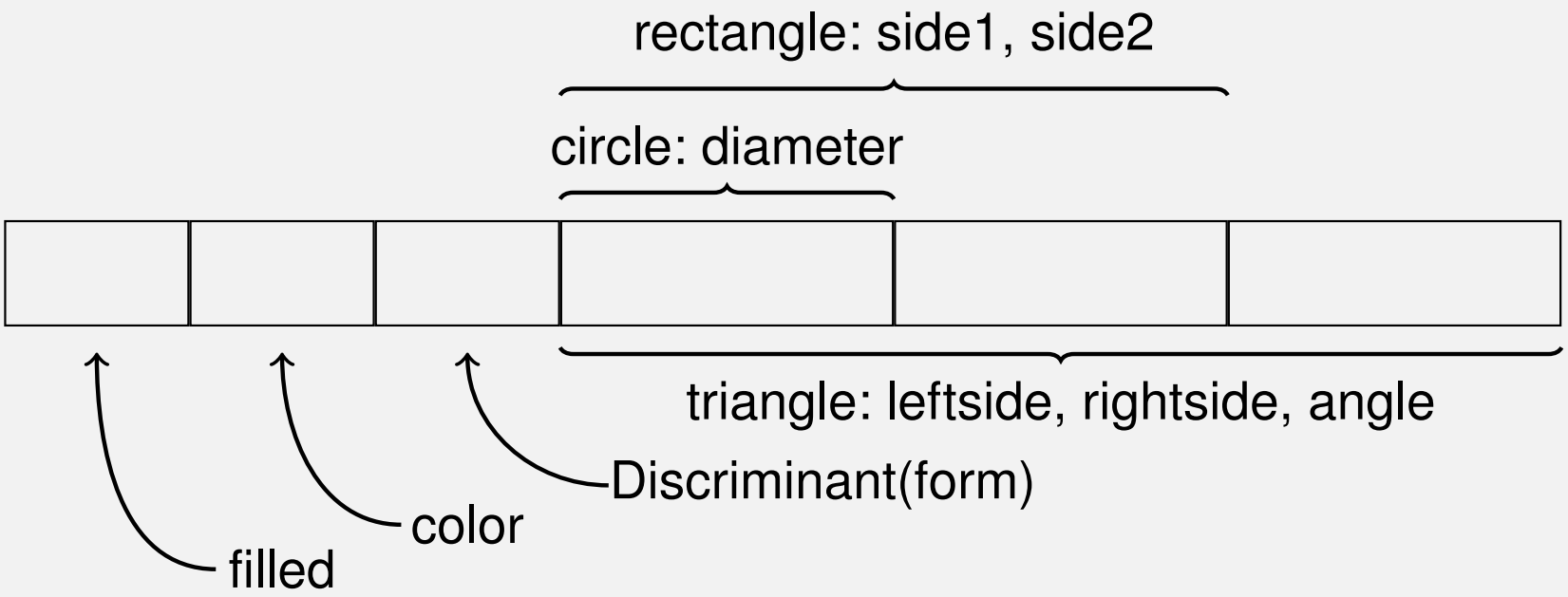
Assume that an object in set type is implemented by a bit chain, what is the size of x in byte?

The size of x is 64 bits = 8 bytes.

bytes

Ada Union Type Illustrated

Kiểu hợp nhất cho phép mày tiết kiệm bộ nhớ bằng cách sử dụng cùng một vùng nhớ để lưu trữ các dữ liệu có kiểu khác nhau tại những thời điểm khác nhau. Tuy nhiên, việc sử dụng union cần cẩn thận để tránh các lỗi liên quan đến kiểu dữ liệu. Discriminated unions là một cách tiếp cận an toàn hơn so với free unions vì chúng cung cấp cơ chế kiểm tra kiểu.



Design issues

Should type checking be required? (Có nên yêu cầu kiểm tra kiểu không?): Đây là một câu hỏi quan trọng. Nếu không có kiểm tra kiểu, mày có thể truy cập vào một trường không phù hợp với kiểu dữ liệu hiện tại được lưu trữ trong union, dẫn đến lỗi khó lường.

- Should type checking be required?
- Discriminated vs. Free Union
  - Fortran, C, and C++ provide union constructs in which there is no language support for type checking; the union in these languages is called **free union**
  - Type checking of unions require that each union include a type indicator called a **discriminant**
    - Supported by Ada
- Should unions be embedded in records?

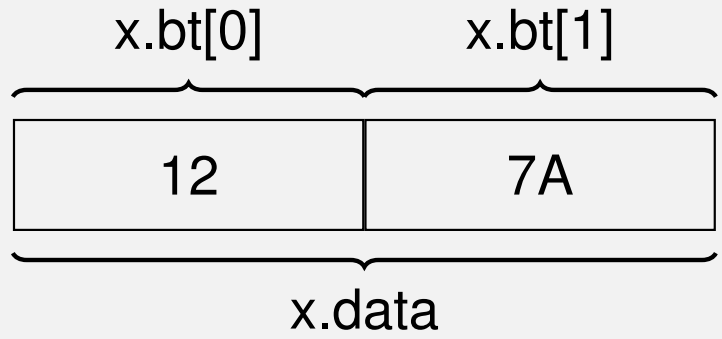
Discriminated vs. Free Union (Union có phân biệt vs. Union tự do):
Free Union: Như trong Fortran, C, và C++, không có sự hỗ trợ ngôn ngữ nào cho việc kiểm tra kiểu. Trình lập trình phải tự biết kiểu dữ liệu nào đang được lưu trữ trong union. Điều này rất dễ gây ra lỗi.
Discriminated Union: Như trong Ada (và nhiều ngôn ngữ hiện đại khác), mỗi union bao gồm một bộ chỉ thị kiểu (type indicator) gọi là discriminant. Cái discriminant này cho biết kiểu dữ liệu hiện tại đang được lưu trữ trong union, cho phép ngôn ngữ thực hiện kiểm tra kiểu.

Should unions be embedded in records? (Có nên nhúng union vào trong bản ghi không?): Câu trả lời thường là có, đặc biệt là đối với discriminated unions. Việc nhúng union vào trong một bản ghi cùng với một trường discriminant giúp quản lý kiểu dữ liệu được lưu trữ một cách an toàn và có cấu trúc hơn.

Example

Cái union này có hai thành viên:
int data: Một biến kiểu số nguyên.
char bt[2]: Một mảng gồm 2 ký tự (byte).
Điểm mấu chốt của union là tất cả các thành viên của nó chia sẻ cùng một vùng nhớ. Điều này có nghĩa là khi mày gán giá trị cho x.data, thì giá trị đó cũng sẽ ảnh hưởng đến x.bt (và ngược lại). Kích thước của union sẽ bằng kích thước của thành viên lớn nhất của nó. Trong trường hợp này, giả sử int chiếm 2 byte (dù thông thường là 4 byte), thì cả data và mảng bt sẽ cùng chiếm 2 byte trong bộ nhớ.

```
union {
    int data;
    char bt[2];
} x;
x.data = 0x7A12;
cout << x.bt[0] << endl; //18
cout << x.bt[1] << endl; //122
```



Giá trị 0x7A12 này trong hệ thập phân tương ứng với 31250. Nếu ta coi nó là một số nguyên 2 byte, thì byte thấp (least significant byte) là 0x12 (tương đương 18 trong hệ thập phân) và byte cao (most significant byte) là 0x7A (tương đương 122 trong hệ thập phân).

Bây giờ, vì x.bt là một mảng 2 ký tự (byte) và nó dùng chung vùng nhớ với x.data, nên mỗi phần tử của mảng bt sẽ tương ứng với một byte của x.data. Thông thường trên các hệ thống little-endian (byte thấp được lưu trữ ở địa chỉ thấp hơn), thì:

[ĐỌC Ở DƯỚI ĐỂ DỄ HIỂU HƠN]

x.bt[0] sẽ chứa byte thấp của x.data, tức là 0x12, có giá trị là 18 trong hệ thập phân.
x.bt[1] sẽ chứa byte cao của x.data, tức là 0x7A, có giá trị là 122 trong hệ thập phân.

Cùng một chỗ: Cái union x này giống như là mày có một cái hộp.
Hai cách gọi: Mày có thể gọi cái hộp này bằng hai tên khác nhau:
data: Nếu mày muốn coi cái hộp này chứa một con số nguyên (kiểu int).
bt: Nếu mày muốn coi cái hộp này chứa hai ký tự (kiểu char). Thực ra là hai byte liên tiếp nhau.
Gán giá trị: Khi mày gán giá trị 0x7A12 cho x.data, thì mày đang bỏ cái giá trị đó vào cái hộp. Cái giá trị 0x7A12 này là một con số (trong hệ thập lục phân).
Xem kiểu ký tự: Bây giờ, mày lại muốn "nhìn" vào cái hộp đó nhưng theo kiểu hai ký tự. Thì cái mảng bt sẽ giúp mày làm điều đó. Nó sẽ chia cái nội dung (giá trị 0x7A12) trong cái hộp ra thành hai phần, mỗi phần là một byte (vì char là 1 byte).
Thứ tự byte: Cái hình nó vẽ x.bt[0] ở bên trái và x.bt[1] ở bên phải, tương ứng với cách người ta thường biểu diễn byte thấp (ít quan trọng hơn) và byte cao (quan trọng hơn) của một số nguyên.
0x12 là byte thấp (nằm ở địa chỉ thấp hơn trong bộ nhớ, tương ứng với x.bt[0]). Giá trị thập phân của 0x12 là 18.
0x7A là byte cao (nằm ở địa chỉ cao hơn trong bộ nhớ, tương ứng với x.bt[1]). Giá trị thập phân của 0x7A là 122.
In ra: Khi mày dùng cout để in x.bt[0], nó sẽ lấy cái byte đầu tiên (byte thấp) trong cái hộp đó ra và in giá trị thập phân của nó là 18. Tương tự, cout << x.bt[1] sẽ lấy cái byte thứ hai (byte cao) và in ra 122.

Cái union cho phép mày sử dụng cùng một vùng nhớ để lưu trữ các kiểu dữ liệu khác nhau. Trong ví dụ này, mày bỏ một số nguyên vào, rồi mày lại "nhìn" vào từng byte của cái số nguyên đó như là các ký tự riêng biệt. Quan trọng là phải nhớ là khi mày gán giá trị cho một thành viên của union, thì giá trị của các thành viên khác cũng sẽ bị ảnh hưởng vì chúng dùng chung bộ nhớ.

Evaluation of Unions

cái ví dụ này cho thấy cách mà một union cho phép mày "nhìn" vào cùng một vùng nhớ dưới các dạng kiểu dữ liệu khác nhau. Ở đây, mày gán một giá trị số nguyên cho x.data, nhưng sau đó mày lại có thể truy cập từng byte của số nguyên đó thông qua mảng ký tự x.bt. Điều này rất hữu ích trong một số trường hợp, ví dụ như khi mày muốn thao tác trực tiếp với các byte của một kiểu dữ liệu lớn hơn. Tuy nhiên, mày phải tự quản lý kiểu dữ liệu nào đang được lưu trữ trong union để tránh đọc dữ liệu một cách sai lệch.

- Potentially unsafe construct in some languages
  - Do not allow type checking
- Java and C# do not support unions
  - Reflective of growing concerns for safety in programming language

Potentially unsafe construct in some languages (Có khả năng là một cấu trúc không an toàn trong một số ngôn ngữ): Đúng vậy, như mình đã thấy ở ví dụ C++, việc sử dụng union có thể hơi nguy hiểm nếu không cẩn thận.
Do not allow type checking (Không cho phép kiểm tra kiểu): Trong các ngôn ngữ như C và C++, trình biên dịch thường không kiểm tra xem mày đang truy cập vào union với kiểu dữ liệu nào. Nếu mày lưu một kiểu dữ liệu vào union nhưng lại cố gắng đọc nó ra với một kiểu dữ liệu khác, thì có thể sẽ nhận được kết quả không mong muốn hoặc thậm chí là crash chương trình.
Java and C# do not support unions (Java và C# không hỗ trợ unions): Hai ngôn ngữ lập trình phổ biến và hiện đại này đã quyết định không đưa unions vào ngôn ngữ của họ.
Reflective of growing concerns for safety in programming language (Phản ánh mối quan tâm ngày càng tăng về tính an toàn trong ngôn ngữ lập trình): Việc Java và C# bỏ qua unions cho thấy rằng các nhà thiết kế ngôn ngữ ngày càng chú trọng đến việc ngăn chặn các lỗi có thể xảy ra do việc sử dụng không an toàn các cấu trúc như unions. Họ thà hy sinh một chút tính linh hoạt để đổi lấy sự an toàn và ổn định cho chương trình.

Set Types

Nó dùng để biểu diễn cái khái niệm tập hợp trong toán học. Một tập hợp là một bộ sưu tập các phần tử duy nhất (không có phần tử nào trùng lặp) và không có thứ tự cụ thể.

```
x: set of 1..10;
y: set of char;
```

x: set of 1..10;: Đây là một tập hợp các số nguyên từ 1 đến 10. Ví dụ, nó có thể chứa các số {1, 3, 5, 7, 9} hoặc {2, 4, 6, 8, 10} hoặc tất cả các số từ 1 đến 10. Quan trọng là mỗi số chỉ xuất hiện một lần.
y: set of char;: Đây là một tập hợp các ký tự. Ví dụ, nó có thể chứa {'a', 'b', 'c'} hoặc {'$', '#', '@'}.

represent the concept of set

has operators: membership, union, intersection, different,...

implemented by bit chain or hash table.

membership (thuộc về): Kiểm tra xem một phần tử có nằm trong tập hợp hay không (ví dụ: "5 có thuộc tập hợp x không?").
union (hợp): Tạo một tập hợp mới chứa tất cả các phần tử từ cả hai tập hợp (loại bỏ các phần tử trùng lặp). Ví dụ, hợp của {1, 2, 3} và {3, 4, 5} là {1, 2, 3, 4, 5}.
intersection (giao): Tạo một tập hợp mới chỉ chứa các phần tử chung của cả hai tập hợp. Ví dụ, giao của {1, 2, 3} và {3, 4, 5} là {3}.
difference (hiệu): Tạo một tập hợp mới chứa các phần tử chỉ có trong tập hợp thứ nhất mà không có trong tập hợp thứ hai. Ví dụ, hiệu của {1, 2, 3} và {3, 4, 5} là {1, 2}.

bit chain (chuỗi bit): Cách này thường hiệu quả khi các phần tử của tập hợp thuộc một kiểu dữ liệu thứ tự có phạm vi nhỏ (ví dụ, tập hợp các số từ 1 đến 10). Mỗi bit trong chuỗi bit sẽ tương ứng với một phần tử có thể có. Nếu bit đó là 1, nghĩa là phần tử đó có trong tập hợp, ngược lại nếu là 0 thì không.
hash table (bảng băm): Cách này linh hoạt hơn và có thể xử lý các tập hợp có số lượng phần tử lớn hơn hoặc các phần tử không thuộc kiểu dữ liệu thứ tự có phạm vi nhỏ. Mỗi phần tử trong tập hợp sẽ được băm (hashed) để xác định vị trí của nó trong bảng băm.

Pointer Types

Kiểu Con Trỏ là gì?

int *ptr;: Đây là cách khai báo một biến con trỏ tên là ptr, nó sẽ trỏ đến một vùng nhớ chứa một giá trị kiểu int.
Biến con trỏ có thể chứa các giá trị là địa chỉ bộ nhớ và một giá trị đặc biệt là nil (thường có nghĩa là con trỏ không trỏ đến đâu cả).

int *ptr;

A *pointer* type variable has a range of values that consists of memory addresses and a special value, *nil*

Provide the power of indirect addressing (Cung cấp khả năng định địa chỉ gián tiếp): Thay vì trực tiếp làm việc với một biến, mày có thể làm việc với địa chỉ của nó thông qua con trỏ.

Provide the power of indirect addressing

Provide a way to manage dynamic memory

- A pointer can be used to access a location in the area where storage is dynamically created (usually called a *heap*)

Provide a way to manage dynamic memory (Cung cấp cách quản lý bộ nhớ động): Con trỏ cho phép mày cấp phát và thu hồi bộ nhớ trong quá trình chạy chương trình (ở vùng nhớ gọi là heap).

▸ Skip Pointer Type

Pointer Operations

Có hai thao tác cơ bản:
Assignment (Gán): Gán một địa chỉ bộ nhớ cho biến con trỏ:
int *p, *q;
p = q; // Bây giờ p trỏ đến cùng địa chỉ mà q đang trỏ đến

Two fundamental operations: assignment and dereferencing

Assignment is used to set a pointer variable's value to some useful address

int *p,*q;

p = q

Dereferencing yields the value stored at the location represented by the pointer's value

- Dereferencing can be explicit or implicit
- C++ uses an explicit operation via *

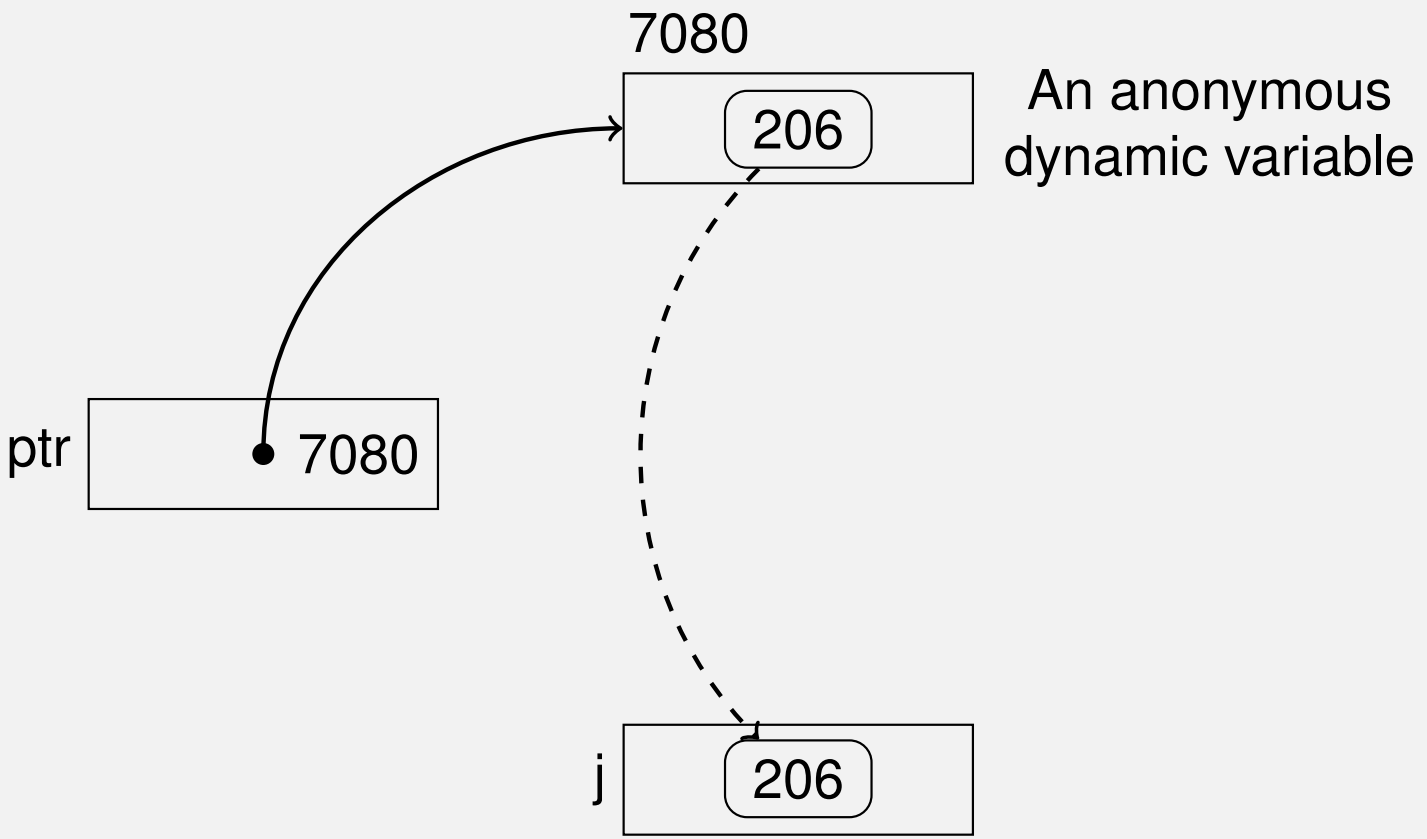
j = *ptr

sets j to the value located at ptr

Dereferencing (Giải tham chiếu): Lấy giá trị được lưu trữ tại địa chỉ mà con trỏ đang trỏ đến.
Trong C++, người ta dùng dấu * để giải tham chiếu.

Pointer Deferencing Illustrated

Biến ptr đang giữ địa chỉ 7080, và tại địa chỉ đó có giá trị là 206. Khi mày viết j = *ptr;, thì giá trị của j sẽ là 206.



The deferencing operation j = *ptr

Problems with Pointers

Dangling pointers (Con trỏ lơ lửng - nguy hiểm): Xảy ra khi con trỏ trỏ đến một vùng nhớ đã được giải phóng (de-allocated). Lúc này, con trỏ trở nên vô nghĩa và việc truy cập nó có thể gây ra lỗi.

Lost heap-dynamic variable (Biến động trên heap bị mất - garbage): Xảy ra khi một vùng nhớ động đã được cấp phát nhưng không còn con trỏ nào trỏ đến nó nữa. Vùng nhớ này không thể được sử dụng lại và gây lãng phí bộ nhớ (gọi là "rác").

- Dangling pointers (dangerous)
  - A pointer points to a heap-dynamic variable that has been de-allocated
- Lost heap-dynamic variable
  - An allocated heap-dynamic variable that is no longer accessible to the user program (often called *garbage*)

Rất linh hoạt nhưng phải sử dụng cẩn thận. Con trỏ có thể trỏ đến bất kỳ biến nào, bất kể khi nào nó được cấp phát.
Được dùng để quản lý bộ nhớ động và định địa chỉ.

```
int *ptr;
int count, init;
...
ptr = &init;  // Gán địa chỉ của biến init cho con trỏ ptr (dấu & là "address-of operator")
count = *ptr;  // Giải tham chiếu ptr để lấy giá trị (10) và gán cho count
```

- Extremely flexible but must be used with care
- Pointers can point at any variable regardless of when it was allocated
- Used for dynamic storage management and addressing

Trong C/C++, mày có thể thực hiện một số phép toán số học trên con trỏ (thường dùng khi làm việc với mảng):

Pointer arithmetic is possible

```
int list[10];

int *ptr = list; // Con trỏ ptr trỏ đến phần tử đầu tiên của mảng list


*(ptr + 1) // Truy cập phần tử thứ hai của mảng (tương đương list[1])

*(ptr + index) // Truy cập phần tử ở vị trí index (tương đương list[index
])

ptr[index] // Đây là cách viết gọn của *(ptr + index)
```

void *: Một kiểu con trỏ đặc biệt có thể trỏ đến bất kỳ kiểu dữ liệu nào. Tuy nhiên, mày không thể trực tiếp giải tham chiếu void * mà cần phải ép kiểu nó về kiểu con trỏ cụ thể trước.

Trong C/C++, khi con trỏ trỏ đến một bản ghi (struct), mày có thể truy cập các thành viên của bản ghi theo hai cách:

Explicit: (*p).name (dùng dấu ngoặc để đảm bảo giải tham chiếu p trước khi truy cập thành viên name).
Implicit: p->name (cách viết gọn và thường dùng hơn).

Pointer Operations

Pointer points to a record in C/C

++ Explicit: (*p).name
Implicit: p -> name

Management of heap use explicit

allocation C: function **malloc**
C++: **new** and **delete** operators

Trong C, dùng hàm malloc để cấp phát bộ nhớ động và hàm free để giải phóng.
Trong C++, dùng toán tử new để cấp phát và delete (hoặc delete cho mảng) để giải phóng.

Design Issues of Pointers

Phạm vi (scope) và thời gian tồn tại (lifetime) của biến con trỏ là gì?
Thời gian tồn tại của biến động trên heap là bao lâu?
Con trỏ có bị giới hạn về kiểu dữ liệu mà nó có thể trỏ đến không?
Con trỏ được dùng để quản lý bộ nhớ động, định địa chỉ gián tiếp, hay cả hai?
Ngôn ngữ nên hỗ trợ kiểu con trỏ, kiểu tham chiếu (reference types), hay cả hai?

What are the scope of and lifetime of a pointer variable?

What is the lifetime of a heap-dynamic variable?

Are pointers restricted as to the type of value to which they can point?

Are pointers used for dynamic storage management, indirect addressing, or both?

Should the language support pointer types, reference types, or both?

Reference Types

```
int A;
int &rA = A;
A = 1;                  // rA là một tham chiếu đến biến A
cout << rA << endl;     // In ra 1 (vì rA đang tham chiếu đến A)
rA++;
cout << A << endl       // In ra 2 (vì việc tăng rA cũng làm tăng giá trị của A)
```

Một biến tham chiếu (reference) giống như một bí danh (alias) hoặc một tên gọi khác cho một biến đã tồn tại. Khi mày khai báo một biến tham chiếu, mày đang tạo ra một cách khác để truy cập vào cùng một vùng nhớ.

- Pointers refer to an address, references refer to object or value (Con trỏ tham chiếu đến một địa chỉ, tham chiếu tham chiếu đến đối tượng hoặc giá trị).
- C++ includes a special kind of pointer type called a reference type that is used primarily for formal parameters
- Java extends C++'s reference variables and allows them to replace pointers entirely
- C# includes both the references of Java and the pointers of C++

Đặc điểm	Kiểu Tham Chiếu (`&`)	Con Trỏ (`*`)
Khai báo	`int &rA = A;`	`int *pA = &A;`
Truy cập giá trị	`rA`	`*pA`
Tăng giá trị	`rA++`	`(*pA)++` hoặc `pA++` (tăng địa chỉ)
Gán lại tham chiếu	Không (cannot reseated)	`pA = &B;` (có thể trỏ đến biến khác)
Giá trị null	Không (cannot be null)	`pA = nullptr;` (có thể là null)
Khởi tạo	Phải khởi tạo khi khai báo (cannot be uninitialized)	Không bắt buộc (`int *pA;`)

References vs. Pointers in C++

Reference Type	Pointer
`int A;` `int& rA = A;`	`int A;` `int* pA = &A;`
rA $\Rightarrow$ A rA sẽ là A	*pA $\Rightarrow$ A giải tham chiếu của pA sẽ là A
N/A	pA++
cannot reseated	pA = &B
cannot be null	pA = null
cannot be uninitialized	int* pA

Evaluation of Pointers

Dangling pointers and garbage are big problems (Con trỏ lơ lửng và rác là những vấn đề lớn): Như đã nói ở slide trước, đây là những nguy cơ tiềm ẩn khi sử dụng con trỏ.

Pointers are like goto's – they widen the range of cells that can be accessed by a variable (Con trỏ giống như lệnh goto – chúng mở rộng phạm vi các ô nhớ mà một biến có thể truy cập): Điều này có nghĩa là con trỏ có thể làm cho code trở nên khó hiểu và khó bảo trì hơn nếu không được quản lý tốt.

- Dangling pointers and garbage are big problems
- Pointers are like goto's–they widen the range of cells that can be accessed by a variable
- Essential in some kinds of programming applications, e.g. device drivers
- Using references provide some of the flexibility and capabilities of pointers, without the hazards

Essential in some kinds of programming applications, e.g. device drivers (Thiết yếu trong một số loại ứng dụng lập trình, ví dụ như trình điều khiển thiết bị): Trong một số trường hợp, việc thao tác trực tiếp với bộ nhớ thông qua con trỏ là cần thiết.

Using references provide some of the flexibility and capabilities of pointers, without the hazards (Sử dụng tham chiếu cung cấp một số tính linh hoạt và khả năng của con trỏ, mà không có những rủi ro): Tham chiếu an toàn hơn con trỏ vì chúng luôn trỏ đến một đối tượng hợp lệ sau khi được khởi tạo và không thể bị "lạc lối" như con trỏ lơ lửng.

Representations of Pointers

Biểu Diễn của Con Trỏ (Representations of Pointers):

Hầu hết các máy tính sử dụng một giá trị duy nhất (địa chỉ bộ nhớ).
Vi xử lý Intel sử dụng segment và offset để định địa chỉ.

Most computers use single values

Intel microprocessors use segment and offset

Dangling Pointer Problem

- Tombstone: extra heap cell that is a pointer to the heap-dynamic variable
  - The actual pointer variable points only at tombstones
  - When heap-dynamic variable de-allocated, tombstone remains but set to nil
  - Costly in time and space
- Locks-and-keys: Pointer values are represented as (key, address) pairs
  - Heap-dynamic variables are represented as variable plus cell for integer lock value
  - When heap-dynamic variable allocated, lock value is created and placed in lock cell and key cell of pointer

Giải Pháp 1: Tombstone (Bia Mộ):

Thêm một bước trung gian: Thay vì con trỏ trỏ thẳng đến vùng nhớ động (trên heap), người ta tạo ra một cái "bia mộ" (tombstone). Cái bia mộ này là một ô nhớ đặc biệt trên heap, và nó chứa địa chỉ thực sự của cái vùng nhớ động mà mày muốn dùng.

Con trỏ chỉ trỏ vào bia mộ: Cái biến con trỏ của mày bây giờ chỉ lưu địa chỉ của cái bia mộ thôi, chứ không trực tiếp lưu địa chỉ của dữ liệu thật.

Khi vùng nhớ bị giải phóng: Khi cái vùng nhớ động mà mày dùng xong rồi và được giải phóng, thì cái bia mộ vẫn còn đó, nhưng địa chỉ mà nó trỏ tới sẽ được đặt thành một giá trị đặc biệt, thường là nil (giống như số điện thoại cũ bị hủy).

Kiểm tra qua bia mộ: Khi con trỏ của mày cố gắng truy cập dữ liệu, nó sẽ đi qua cái bia mộ trước. Nếu địa chỉ trong bia mộ là nil, hệ thống sẽ biết là vùng nhớ đó không còn hợp lệ nữa và có thể báo lỗi.

Ưu điểm: Có thể phát hiện ra con trỏ lơ lửng.

Nhược điểm: Tốn thêm bộ nhớ để lưu trữ các bia mộ và tốn thêm thời gian để truy cập gián tiếp qua bia mộ.

Giải Pháp 2: Locks-and-keys (Khóa và Chìa Khóa):

Cái này thì giống như là bảo mật hơn một chút:

Mỗi vùng nhớ có một cái khóa: Khi một vùng nhớ động được cấp phát trên heap, nó sẽ được gắn một cái "khóa" (lock) là một giá trị số nguyên.

Con trỏ giữ chìa khóa: Khi mày tạo một con trỏ trỏ đến vùng nhớ đó, con trỏ sẽ được cấp một cái "chìa khóa" (key) có giá trị tương ứng với cái khóa của vùng nhớ. Con trỏ sẽ lưu cả cái chìa khóa này và địa chỉ của vùng nhớ.

Kiểm tra khi truy cập: Mỗi khi con trỏ muốn truy cập vào vùng nhớ, hệ thống sẽ kiểm tra xem cái chìa khóa mà con trỏ đang giữ có khớp với cái khóa của vùng nhớ đó hay không.

Khi vùng nhớ bị giải phóng: Khi vùng nhớ bị giải phóng, cái khóa của nó có thể bị thay đổi hoặc hủy bỏ. Những con trỏ vẫn còn giữ cái chìa khóa cũ sẽ không thể truy cập hợp lệ vào vùng nhớ đó nữa.

Ưu điểm: Cũng giúp phát hiện con trỏ lơ lửng.

Nhược điểm: Cần thêm bộ nhớ để lưu trữ giá trị khóa cho mỗi vùng nhớ và cần thêm thời gian để kiểm tra khóa mỗi khi truy cập.

Recursive Type

Kiểu đệ quy là một kiểu dữ liệu mà định nghĩa của nó có chứa một (tham chiếu đến) giá trị của chính nó.

A value of a *recursive type* can contain a (reference to) value of the same type.

```
type int_list = {int val;
                 int_list next;}
{3,{43,{-1,{6,null}}}}
```

```
type char_tree = {char val;
                  char_tree left;
                  char_tree right;}
{A,{B,{C,null,null},{D,{E,null,null},null}},
   {F,null,null}}
```

Example on Ocaml

nghĩa là một nút của cây chứa một ký tự và hai cây con (trái và phải), mà mỗi cây con này cũng có thể là một char_btree hoặc là Null (để chỉ nút lá hoặc cây rỗng).

```
type char_btree = Tree of char * char_btree * char_btree
                | Null

Tree('A',Tree('B',Tree('C',Null,Null),
                  Tree('D',
                           Tree('E',Null,Null),
                           Null)),
         Tree('F',Null,Null))

type 'a btree = Tree of 'a * 'a btree * 'a btree
              | Null

Tree(4,Tree(3,Null,Null),Tree(6,Null,Null))
```

Type Expression: Motivation Example

x là một mảng 10 phần tử, mỗi phần tử là một bản ghi. Bản ghi này lại chứa một mảng số nguyên a, một bản ghi b (chứa số thực c và mảng số thực d), và một chuỗi d. Để mô tả chính xác kiểu dữ liệu phức tạp như vậy, người ta cần đến các biểu thức kiểu.

```
x: array [1..10] of record
a: array [5..10] of integer;
     b: record
          c: real;
          d: array[1..3] of real;
     end;
     d: string[3];
end;
```

Hàm foo định nghĩa là def foo(x, f) = f(f(x))

Mình suy luận kiểu của nó như sau:

foo là một hàm, nên kiểu của nó sẽ có dạng input_type -> output_type.
foo nhận hai tham số là x và f. Vậy input_type sẽ là kiểu của x kết hợp với kiểu của f.
Mình sẽ dùng * để biểu thị kết hợp này.
Giả sử kiểu của x là T (theo yêu cầu đề bài).
Trong phần thân hàm, có biểu thức f(x). Vì x có kiểu T, và f được gọi với x, nên f phải là một hàm nhận vào một đối số có kiểu T.
Tiếp theo, có biểu thức f(f(x)). Kết quả của f(x) lại được truyền vào f. Điều này có nghĩa là kiểu trả về của f(x) phải giống với kiểu đầu vào của f (là T).
Vậy, f là một hàm nhận vào T và trả về T, tức là kiểu của f là T -> T.
Cuối cùng, kết quả của f(f(x)) sẽ là kiểu trả về của hàm foo. Vì f có kiểu T -> T, và f(x) có kiểu T, thì f(f(x)) cũng sẽ có kiểu T.
Vậy, kiểu của hàm foo là:

(kiểu của x * kiểu của f) -> kiểu trả về

(T * (T -> T)) -> T

Type Expressions

Biểu thức kiểu là một cách hình thức để mô tả một kiểu dữ liệu. Nó được xây dựng từ các thành phần cơ bản theo một số quy tắc:

A **basic type** is a type expression.
boolean, char, integer, float, void, subrange.

A **type name** is a type expression.

A **type constructor** applied to type expressions is a type expression. Including:

Arrays: array(I,T) where I: index type, T:element type
Products: T1 $\times$ T2
Records: record((name1 $\times$ T1) $\times$ (name2 $\times$ T2) $\times \ldots$)
Pointers: pointer(T)
Functions: T1 $\rightarrow$ T2

A **type variable** is a type expression.

Biểu thức kiểu là một cách chính xác và hình thức để mô tả các kiểu dữ liệu, từ đơn giản đến phức tạp.

Type Expressions (Biểu Thức Kiểu):

Biểu thức kiểu là một cách hình thức để mô tả một kiểu dữ liệu. Nó được xây dựng từ các thành phần cơ bản theo một số quy tắc:

A basic type is a type expression (Kiểu cơ bản là một biểu thức kiểu): Ví dụ: boolean, char, integer, float, void, subrange.
A type name is a type expression (Tên kiểu là một biểu thức kiểu): Ví dụ, nếu mày định nghĩa typedef int siso;, thì siso cũng là một biểu thức kiểu.
A type constructor applied to type expressions is a type expression (Một bộ xây dựng kiểu áp dụng cho các biểu thức kiểu là một biểu thức kiểu): Cái này hơi trừu tượng, nhưng ý là mày có thể dùng các "bộ xây dựng" để tạo ra các kiểu phức tạp hơn từ các kiểu đơn giản:
Arrays: array(I, T) trong đó I là kiểu chỉ số và T là kiểu phần tử. Ví dụ: int t[10]; có thể được biểu diễn là array(0..9, int).
Products: T1 × T2 (thường dùng để biểu diễn tuples hoặc danh sách tham số của hàm). Ví dụ: một hàm nhận vào một số nguyên và một số thực có thể có kiểu là int × float.
Records: record((name1 × T1) × (name2 × T2) × ...) Ví dụ: struct { int a; int b; } có thể được biểu diễn là record((a × int) × (b × int)).
Pointers: pointer(T) trong đó T là kiểu dữ liệu mà con trỏ trỏ đến. Ví dụ: int *p có thể được biểu diễn là pointer(int).
Functions: T1 → T2 trong đó T1 là kiểu của tham số và T2 là kiểu trả về. Ví dụ: int foo(int a, float b) có thể có kiểu là (int × float) → int.
A type variable is a type expression (Một biến kiểu là một biểu thức kiểu): Thường dùng trong các ngôn ngữ hỗ trợ generic programming (lập trình đa hình). Ví dụ, trong template <class T> struct vd { T a; T b[3]; };, T là một biến kiểu. Kiểu của struct vd có thể được biểu diễn là record((a × T) × (b × array(0..2, T))).

Example

int $\Rightarrow$ int

typedef int siso; $\Rightarrow$ siso

int t[10]; $\Rightarrow$ array(0..9,int)

int foo(int a,float b) $\Rightarrow$ (int $\times$ float) $\rightarrow$ int

struct int a;int b $\Rightarrow$ record((a $\times$ int) $\times$ (b $\times$ int))

int *p $\Rightarrow$ pointer(int)

template <class T> struct vd T a; T b[3];
$\Rightarrow$ record((a $\times$ T) $\times$ (b $\times$ array(0..2,T)))

Type Checking

Definition (Định nghĩa): Là quá trình đảm bảo rằng chương trình tuân thủ các quy tắc của hệ thống kiểu dữ liệu. Nói đơn giản là kiểm tra xem mày có đang dùng dữ liệu đúng kiểu ở đúng chỗ không.

Definition

Type checking is the activity of ensuring that a program respects the rules imposed by the type system

Static type checking is performed in **compiling** time. It is often applied for static type binding languages.

Được thực hiện trong quá trình biên dịch (compile time). Thường được áp dụng cho các ngôn ngữ có liên kết kiểu tĩnh (static type binding), ví dụ như C++, Java.

Dynamic type checking is performed in **running** time. It is often applied for dynamic type binding languages

Được thực hiện trong quá trình chạy chương trình (running time). Thường được áp dụng cho các ngôn ngữ có liên kết kiểu động (dynamic type binding), ví dụ như Python, JavaScript.

dome features in static type binding language that cannot be type checked during compiling time.

Một số tính năng trong ngôn ngữ liên kết kiểu tĩnh không thể kiểm tra kiểu trong quá trình biên dịch. (Ví dụ: downcasting trong OOP có thể cần kiểm tra ở runtime).

Type Inference - Suy Luận Kiểu Dữ Liệu

Definition (Định nghĩa): Là khả năng của trình biên dịch tự động suy ra kiểu dữ liệu của một thành phần chương trình (ví dụ: biến, hàm) mà không cần lập trình viên phải khai báo rõ ràng.

Definition

Type inference is the ability of a compiler to deduce type information of program unit.

Example on Scala

```
def add(x:Int) = x + 1
```

Trong ví dụ này, Scala có thể tự động suy ra rằng kiểu trả về của hàm add là Int vì nó nhận một Int và cộng thêm 1 (kết quả chắc chắn là Int).

Return type of function add is inferred to be Int

Mechanism

- Assign type (built-in or variable type)to leaf nodes in AST. Gán kiểu (kiểu sẵn có hoặc biến kiểu) cho các nút lá trong cây cú pháp trừu tượng (AST).
- Generate type constraints in each internal node in AST. Tạo các ràng buộc kiểu ở mỗi nút bên trong của AST.
- Resolve these type constraints Giải quyết các ràng buộc kiểu này để xác định kiểu cuối cùng.

Type Equivalence - Tương Đương Kiểu Dữ Liệu

an operand of one type can be substituted for one of

the other type without coercion.

Two approaches:

Equivalence by name: same type name

```
type Celsius = Float;
type Fahrenheit = Float;
```

Structural equivalence: same structure

```
type A = record
    field1: integer;
    field2: real;
end
```

```
type B = record
    field1: integer;
    field2: real;
end
```

Type Equivalence (Tương Đương Kiểu Dữ Liệu): Cái này có nghĩa là hai kiểu dữ liệu phải hoàn toàn giống nhau. Giống như hai giọt nước vậy đó. Nếu hai kiểu tương đương, thì mày có thể dùng kiểu này ở bất kỳ chỗ nào mà kiểu kia được dùng mà không cần phải làm gì thêm. Ví dụ, nếu mày định nghĩa type MyInt is Integer; thì MyInt và Integer là tương đương.

Static Type Checking for Structural Equivalence

Hàm sequiv này là một ví dụ về cách kiểm tra xem hai kiểu dữ liệu s và t có tương đương về mặt cấu trúc hay không. Nó nhận vào hai kiểu (được biểu diễn dưới dạng Type) và trả về true nếu chúng tương đương, false nếu không. Nó hoạt động như sau:

```
function sequiv(Type s, Type t):boolean
begin
    if (s and t are the same basic type) then
        return true;
    else if (s = array(s1,s2) and t = array(t1,t2) then
        return sequiv(s1,t1) and sequiv(s2,t2);
    else if (s = s1 × s2 and t = t1 × t2) then
        return sequiv(s1,t1) and sequiv(s2,t2);
    else if (s = pointer(s1) and t = pointer(t1)) then
        return sequiv(s1,t1);
    else if (s = s1 → s2 and t = t1 → t2) then
        return sequiv(s1,t1);
    else
        return false;
```

Cái hàm sequiv này là một ví dụ về cách kiểm tra xem hai kiểu dữ liệu s và t có tương đương về mặt cấu trúc hay không. Nó nhận vào hai kiểu (được biểu diễn dưới dạng Type) và trả về true nếu chúng tương đương, false nếu không. Nó hoạt động như sau:

if (s and t are the same basic type) then return true;: Nếu cả s và t đều là cùng một kiểu dữ liệu cơ bản (ví dụ: cả hai đều là integer, hoặc cả hai đều là boolean), thì chúng tương đương.
else if (s = array (s1, s2) and t = array (t1, t2) then return sequiv (s1, t1) and sequiv (s2, t2) ;: Nếu s là một mảng có kiểu phần tử là s1 và kiểu chỉ số là s2, và t cũng là một mảng có kiểu phần tử là t1 và kiểu chỉ số là t2, thì s và t tương đương nếu kiểu phần tử s1 tương đương với t1 và kiểu chỉ số s2 tương đương với t2. Nó gọi lại chính nó (sequiv) để kiểm tra sự tương đương của các kiểu bên trong.
else if (s = s1 × s2 and t = t1 × t2) then return sequiv (s1, t1) and sequiv (s2, t2) ;: Nếu s là một " product" (thường đại diện cho một tuple hoặc danh sách tham số) của kiểu s1 và s2, và t cũng là một product của kiểu t1 và t2, thì s và t tương đương nếu s1 tương đương với t1 và s2 tương đương với t2.
else if (s = pointer (s1) and t = pointer (t1)) then return sequiv (s1, t1) ;: Nếu s là một con trỏ trỏ đến kiểu s1, và t là một con trỏ trỏ đến kiểu t1, thì s và t tương đương nếu kiểu mà chúng trỏ đến (s1 và t1) tương đương.
else if (s = s1 → s2 and t = t1 → t2) then return sequiv (s1, t1) ;: (Lưu ý: Có vẻ như có một lỗi nhỏ ở đây, nó nên là return sequiv (s1, t1) and sequiv (s2, t2) ; giống như trường hợp array và product) Nếu s là một hàm có kiểu tham số là s1 và kiểu trả về là s2, và t là một hàm có kiểu tham số là t1 và kiểu trả về là t2, thì s và t tương đương nếu kiểu tham số s1 tương đương với t1 và kiểu trả về s2 tương đương với t2.
else return false;: Nếu không có trường hợp nào ở trên đúng, thì hai kiểu s và t không tương đương về cấu trúc.
Nói tóm lại, hàm này so sánh cấu trúc của hai biểu thức kiểu một cách đệ quy để xem chúng có giống nhau hay không. Nếu chúng có cùng "hình dạng" và các thành phần bên trong cũng tương đương, thì chúng được coi là tương đương về cấu trúc.

Type Compatibility

Definition (Định nghĩa): Kiểu T được gọi là tương thích với kiểu S nếu một giá trị thuộc kiểu T được phép sử dụng ở bất kỳ chỗ nào mà một giá trị thuộc kiểu S có thể được dùng. Nói nôm na là mày có thể "nhét" một giá trị kiểu T vào chỗ cần một giá trị kiểu S mà không gây ra lỗi (hoặc ít nhất là không gây ra lỗi nghiêm trọng).
Example (Ví dụ): int (số nguyên) và float (số thực) thường là tương thích với nhau. Ví dụ, mày có thể gán một giá trị int cho một biến float.

Definition

Type T is compatible with type S if a value of type T is permitted in any context where a value of type S is admissible

Example, *int* and *float*
A type T is compatible with type S when:

- T is equivalence to S
- Values of T form a subset of values of S
- All operations on S are permitted on T
- Values of T correspond *in a canonical fashion* to values of S. (*int* and *float*)
- Values of T can transform to some values of S.

A type T is compatible with type S when (Kiểu T tương thích với kiểu S khi):

T is equivalence to S (T tương đương với S): Nếu kiểu T và kiểu S là cùng một kiểu. Cái này thì rõ ràng rồi, cùng kiểu thì chắc chắn là dùng được ở mọi chỗ của nhau.
Values of T form a subset of values of S (Các giá trị của T là một tập con của các giá trị của S): Ví dụ, một kiểu subrange (ví dụ, số từ 1 đến 10) có thể tương thích với kiểu số nguyên lớn hơn (ví dụ, số nguyên từ 1 đến 100). Tất cả các giá trị trong subrange cũng đều là giá trị hợp lệ trong kiểu lớn hơn.
All operations on S are permitted on T (Tất cả các phép toán trên S đều được phép thực hiện trên T): Nếu kiểu S có một số phép toán nhất định, và kiểu T cũng hỗ trợ tất cả các phép toán đó (hoặc có thể được sử dụng một cách an toàn với các phép toán đó), thì T có thể tương thích với S.
Values of T correspond in a canonical fashion to values of S. (int and float) (Các giá trị của T tương ứng một cách chính tắc với các giá trị của S. (ví dụ: int và float)): Có một cách chuyển đổi tự nhiên và rõ ràng giữa các giá trị của hai kiểu. Ví dụ, một số nguyên có thể được biểu diễn một cách tự nhiên dưới dạng số thực (có thể có phần thập phân là 0).
Values of T can transform to some values of S (Các giá trị của T có thể chuyển đổi thành một số giá trị của S): Có thể có một quy tắc chuyển đổi (ngầm định hoặc tường minh) để biến một giá trị kiểu T thành một giá trị kiểu S.

Type Compatibility (Tính Tương Thích Kiểu Dữ Liệu): Cái này thì rộng hơn. Nó có nghĩa là một giá trị của kiểu này có thể được sử dụng ở một chỗ mà người ta mong đợi một kiểu khác. Nhưng hai kiểu này không nhất thiết phải giống nhau hoàn toàn. Có thể cần có một sự chuyển đổi ngầm định (coercion) hoặc một quy tắc nào đó để giá trị của kiểu này "vừa vặn" với chỗ cần kiểu kia.

Type Conversion - Chuyển Đổi Kiểu Dữ Liệu

Definition (Định nghĩa): Là quá trình chuyển đổi một giá trị từ kiểu dữ liệu này sang kiểu dữ liệu khác.
Implicit conversion - coercion (Chuyển đổi ngầm định - ép kiểu tự động): Trình biên dịch tự động thực hiện việc chuyển đổi (ví dụ: khi gán một int cho một biến float).
Explicit conversion - cast (Chuyển đổi tường minh - ép kiểu): Lập trình viên phải chỉ định rõ ràng việc chuyển đổi (ví dụ: dùng (int) myFloat trong C++).

Definition

Type conversion is conversing a value of this type to a value of another type

Implicit conversion - coercion

Explicit conversion - cast

Polymorphism

Polymorphism (Tính Đa Hình):
Definition (Định nghĩa):
Monomorphic: Bất kỳ đối tượng ngôn ngữ nào cũng có một kiểu duy nhất.
Polymorphic: Cùng một đối tượng có thể có nhiều hơn một kiểu.

Definition

- *Monomorphic*: any language object has a unique type
- *Polymorphic*: the same object can have more than one type

Example, +: $int \times int \rightarrow int$ or $float \times float \rightarrow float$

Kind of Polymorphism

- *Ad hoc polymorphism* - Overloading
- *Universal Polymorphism*
  - Parametric polymorphism (swap(T& x,T& y))
  - Subtyping polymorphism (in OOP)

Ad hoc polymorphism - Overloading (Đa hình ad hoc - Nạp chồng): Cùng một tên (ví dụ: tên hàm) có thể có nhiều định nghĩa khác nhau dựa trên kiểu của các tham số.

Universal Polymorphism (Đa hình phổ quát):

Parametric polymorphism (Đa hình tham số): Cho phép định nghĩa các hàm hoặc kiểu dữ liệu có thể làm việc với nhiều kiểu dữ liệu khác nhau mà không cần biết trước kiểu cụ thể (ví dụ: dùng templates trong C++ hoặc generics trong Java/C#). Cái ví dụ về hàm swap là một ví dụ
.
Subtyping polymorphism (Đa hình kiểu con - trong OOP): Cho phép một biến của kiểu cha có thể tham chiếu đến một đối tượng của kiểu con. Cái ví dụ về Polygon, Rectangle, và Triangle là một ví dụ.

Example of Parametric Polymorphism

```
template<typename T>
void swap (T& x, T& y){
     T tmp = x;
     x = y;
     y = tmp;
}
int a = 5, b = 3;
swap(a,b);
cout << a << " " << b << endl;
```

Example of Subtyping Polymorphism

```
class Polygon
    public:
        virtual float getArea() = 0;
class Rectangle : public Polygon
    public:
        float getArea()
            return height * width;
    private:
        float height,width;
class Triangle : public Polygon
    public:
        float getArea()
            float p = (a + b + c) / 2;
            return sqrt(p*(p-a)*(p-b)*(p-c));
    private:
        float a,b,c;
Shape *s;
s = (...) ? new Rectangle(3,4) : new Triangle(3,4,5);
s->getArea();
```

Example on Scala

```
abstract class Stack[A]
  def push(x: A): Stack[A] =
                new NonEmptyStack[A](x, this)
  def isEmpty: Boolean
  def top: A
  def pop: Stack[A]
class EmptyStack[A] extends Stack[A]
  def isEmpty = true
  def top = error("EmptyStack.top")
  def pop = error("EmptyStack.pop")
class NonEmptyStack[A](elem:A,rest:Stack[A])
                                    extends Stack[A]
  def isEmpty = false
  def top = elem
  def pop = rest
val x = new EmptyStack[Int]
val y = x.push(1).push(2)
println(y.pop.top)
```

Example on Scala

```
def isPrefix[A](p: Stack[A], s: Stack[A]): Boolean = {
    p.isEmpty ||
    p.top == s.top && isPrefix[A](p.pop, s.pop)
}
```

Summary [1]

Type system is mainly used to error detection

Primitive type

Structure type

Type checking

Type system is mainly used to error detection (Hệ thống kiểu dữ liệu chủ yếu được dùng để phát hiện lỗi): Đúng vậy, mục đích chính của hệ thống kiểu dữ liệu là giúp tìm ra các lỗi liên quan đến việc sử dụng sai kiểu dữ liệu trong chương trình ngay từ giai đoạn biên dịch hoặc runtime.

Primitive type (Kiểu nguyên thủy): Các kiểu dữ liệu cơ bản như số nguyên, số thực, ký tự, boolean.

Structure type (Kiểu cấu trúc): Các kiểu dữ liệu phức tạp hơn được xây dựng từ các kiểu khác, ví dụ như mảng, bản ghi, union, tập hợp, con trỏ, tham chiếu.

Type checking (Kiểm tra kiểu dữ liệu): Quá trình đảm bảo chương trình tuân thủ các quy tắc của hệ thống kiểu.

References


Maurizio Gabbrielli and Simone Martini, Programming Languages: Principles and Paradigms, Chapter 8, Springer, 2010.