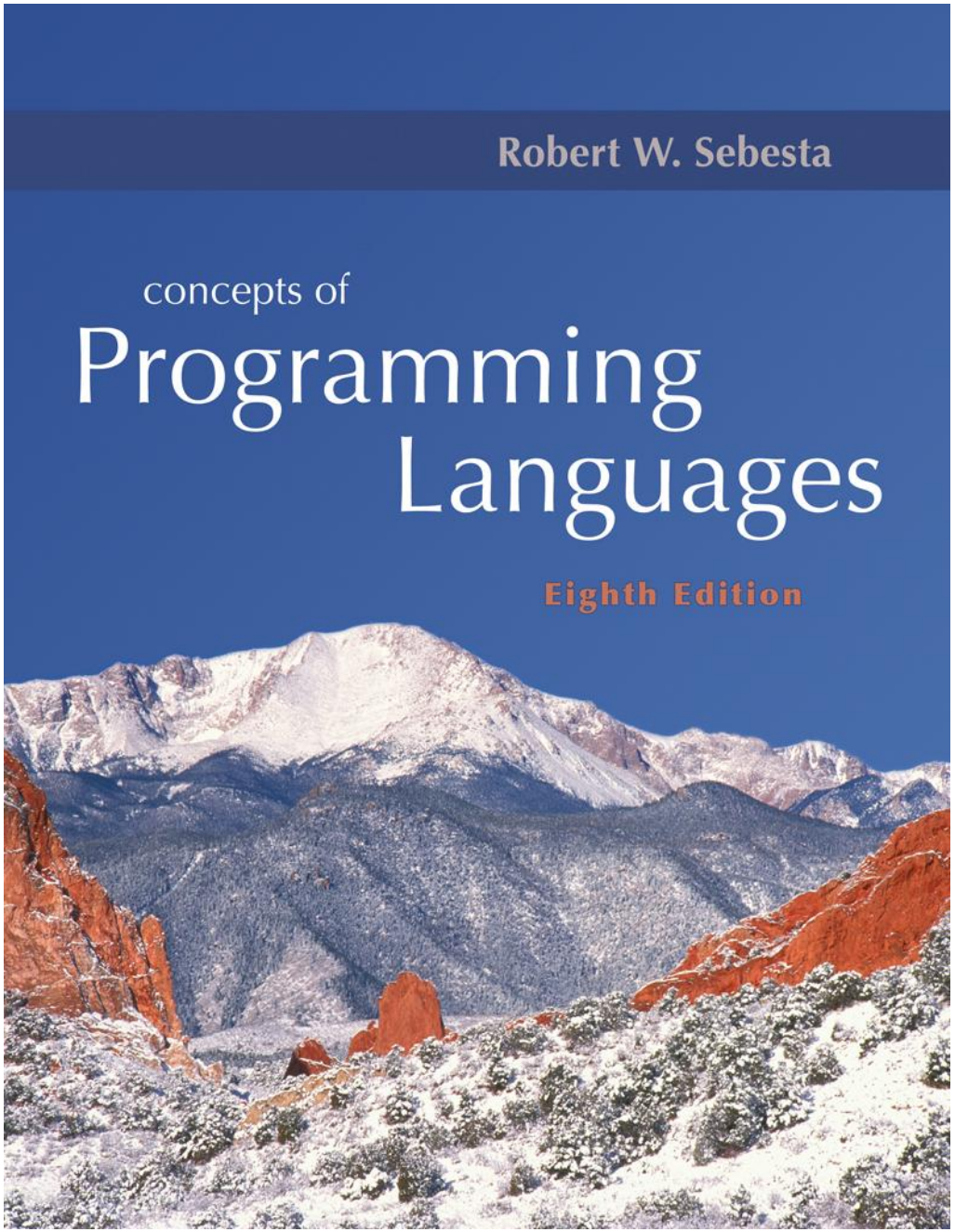
Chapter 6

Data Types



ISBN 0-321-49362-1

Chapter 6 Topics

- Introduction
- Primitive Data Types
- Character String Types
- User-Defined Ordinal Types
- Array Types
- Associative Arrays
- Record Types
- Union Types
- Pointer and Reference Types

Introduction

- A *data type* defines **a collection of data objects and a set of predefined operations on those objects**
- A *descriptor* is the collection of the attributes of a variable
- An *object* represents an instance of a user-defined (abstract data) type
- **One design issue for all data types:**
  What operations are defined and how are they specified?

Primitive Data Types

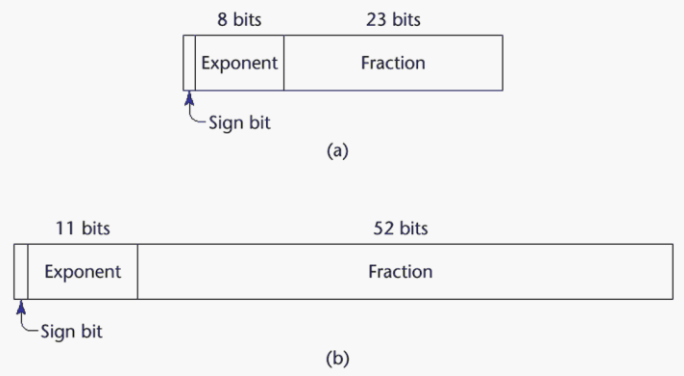
- Almost all programming languages provide **a set of *primitive data types***
- Primitive data types: Those not defined in terms of other data types
- **Some primitive data types are merely reflections of the hardware**
- Others require little non-hardware support

Primitive Data Types: Integer

- Almost always an exact reflection of the hardware so the mapping is trivial
- **There may be as many as eight different integer types in a language**
- **Java's signed integer sizes: `byte`, `short`, `int`, `long`**

Primitive Data Types: Floating Point

- Model real numbers, but only as approximations
- **Languages for scientific use support at least two floating-point types (e.g., `float` and `double`; sometimes more**
- Usually exactly like the hardware, but not always
- IEEE Floating-Point Standard 754



Primitive Data Types: Decimal

- For business applications (money)
  - Essential to COBOL
  - C# offers a decimal data type
- Store a fixed number of decimal digits
- *Advantage*: accuracy
- *Disadvantages*: limited range, wastes memory

Primitive Data Types: 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**
  - **Advantage: readability**

Primitive Data Types: Character

- Stored as numeric codings
- **Most commonly used coding: ASCII**
- **An alternative, 16-bit coding: Unicode**
  - **Includes characters from most natural languages**
  - **Originally used in Java**
  - **C# and JavaScript also support Unicode**

Character String Types

- **Values are sequences of characters**
- Design issues:
  - **Is it a primitive type or just a special kind of array?**
  - **Should the length of strings be static or dynamic?**

Character String Types Operations

- **Typical operations:**
  - Assignment and copying
  - Comparison (=, >, etc.)
  - Catenation
  - Substring reference
  - Pattern matching
  - …

Character String Type in Certain Languages

- C and C++
  - Not primitive
  - **Use `char` arrays and a library of functions that provide operations**

Character String Type in Certain Languages

- SNOBOL4 (a string manipulation language)
  - **Primitive**
  - **Many operations, including elaborate pattern matching**

Character String Type in Certain Languages

- Java
  - **Primitive via the `String` class**

Character String Length Options

- Static: COBOL, Java's `String` class

Character String Length Options

- ***Limited Dynamic Length*: C and C++**
  - In C-based language, a special character is used to indicate the end of a string's characters, rather than maintaining the length

Character String Length Options

- ***Dynamic* (no maximum)**: SNOBOL4, Perl, JavaScript

Character String Length Options

- **Ada supports all three string length options**

Character String Type Evaluation

- **Aid to writability**
- **As a primitive type with static length, they are inexpensive to provide**--why not have them?
- **Dynamic length is nice**, but is it worth the expense?

Character String Implementation

- Static length: **compile-time** descriptor

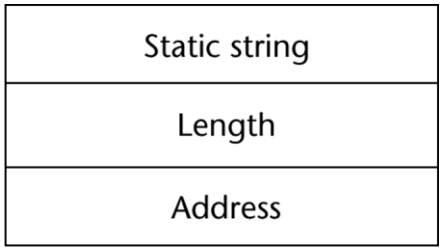
Character String Implementation

- Limited dynamic length: **may need a run-time descriptor for length** (but not in C and C++)

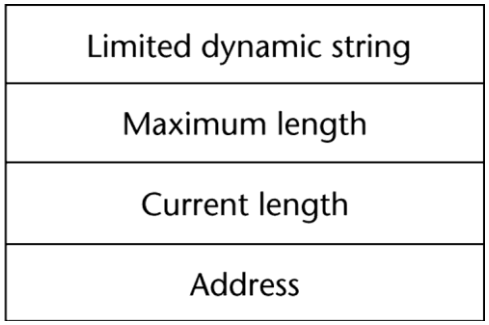
Character String Implementation

- Dynamic length: **need run-time descriptor; allocation/de-allocation is the biggest implementation problem**

Compile- and Run-Time Descriptors



Compile-time descriptor for static strings



Run-time descriptor for limited dynamic strings

Enumeration Types

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

- **C# example**

```
enum days {mon, tue, wed, thu, fri, sat, sun};
```

Evaluation of Enumerated Type

- Aid to readability, e.g., **no need to code a color as a number**

Evaluation of Enumerated Type

- Aid to reliability, e.g., **compiler can check**:
  - **operations** (don't allow colors to be added)
  - **No enumeration variable can be assigned a value outside its defined range**
  - Ada, C#, and Java 5.0 provide better support for enumeration than C++ because enumeration type variables in these languages are not coerced into integer types

Subrange Types

- An ordered contiguous subsequence of an ordinal type
  - Example: 12..18 is a subrange of integer type

Subrange Types

- Ada's design

```
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: Weekday;
Day2 := Day1;
```

Subrange Evaluation

- Aid to readability
  - Make it clear to the readers that **variables of subrange can store only certain range of values**
- Reliability
  - **Assigning a value to a subrange variable that is outside the specified range is detected as an error**

Implementation of User-Defined Ordinal Types

- Enumeration types are **implemented as integers**
- Subrange types are **implemented like the parent types with code inserted (by the compiler) to restrict assignments to subrange variables**

Kiểu tập hợp (Type)

- Là tập hợp các giá trị được định nghĩa sẳn
- Chỉ có pascal hỗ trợ
- Số giá trị tối đa<100
- Hỗ trợ các toán tử : hội, giao

Kiểu tập hợp (Type)

Sets – Pascal

```
type
  colors = (red, blue, green, yellow, orange,
  white, black);
  colorset = set of colors;
var
  set1, set2: colorset;
…
  set1 := [red, blue, green];
  set2 := [white, black];
```

Kiểu tập hợp (Type)

- Dùng một chuỗi bit để biểu diễn
- Ví dụ dùng một chuỗi có 16 bit để biểu diễn một tập hợp các ký tự từ [‘a’..’p’] với qui định
  - 1 :có
  - 0 ; không có
  - Ví dụ tập hợp [‘a’, ‘c’, ‘h’, ‘o’] được biểu diễn bằng chuỗi 1010000010000001
  - Phép hội :or
  - Phép giao :and

Array Types

- An array is an aggregate of homogeneous data elements in which **an individual element is identified by its position in the aggregate, relative to the first element.**

Array Design Issues

- Các kiểu dữ liệu của chỉ mục ?
- Có kiểm tra miền giá trị của chỉ mục không ?
- Có bao nhiêu chiều trong mảng ?
- Khi nào cấp pháp vùng nhớ cho mảng ?
- Có thể cấp phát giá trị ban đầu cho mảng?
- Vấn đề tách một phần của mảng

Array Indexing

- *Indexing* (or subscripting) is **a mapping from indices to elements**

  `array_name (index_value_list)` → `an element`

- Index Syntax
  - FORTRAN, PL/I, Ada use parentheses
    - Ada explicitly uses parentheses to show uniformity between array references and function calls because both are *mappings*
  - Most other languages use brackets

Arrays Index (Subscript) Types

- FORTRAN, C: integer only
- Pascal: any ordinal type (integer, Boolean, char, enumeration)
- Ada: integer or enumeration (includes Boolean and char)
- Java: integer types only
- C, C++, Perl, and Fortran do not specify range checking
- Java, ML, C# specify range checking

Array Categories

- ***Static***: kích thước được xác định tại thời điểm biên dịch và được cấp phát tại thời điểm load ứng dụng vào vùng nhớ data segments,và địa chỉ không đổi trong suốt thời gian thực thi của chương trình

  - Advantage: efficiency (no dynamic allocation)

Array Categories

- ***Fixed stack-dynamic*** : mảng được cấp phát động trên ngăn xếp nhưng kích thước cố định, đây là loại mảng khai báo cục bộ trong chương trình con,cấp phát tại thời điểm thực thi câu lệnh khai báo
  - Advantage: space efficiency

Array Categories (continued)

- ***Stack-dynamic***: mảng có kích thước động, được lưu trữ trong ngăn xếp:mảng được khai báo trong chương trình con và kích thước sẽ tự động cấp phát tại thời điểm thực thi chương trình
  - Advantage: flexibility (the size of an array need not be known until the array is to be used)

Array Categories (continued)

- ***Fixed heap-dynamic***: Loại mảng có kích thước động được cấp phát trên vùng nhớ Heap, nhưng sau khi cấp phát thì vị trí của nó sẽ cố định trên vùng nhớ Heap

- Vùng nhớ sẽ được cấp phát khi có yêu cầu bằng các lệnh cấp phát như malloc hay alloc trong c và vùng nhớ được cấp phát trong Heap

Array Categories (continued)

- **Heap-dynamic:** có kích thước động được cấp phát trên Heap nhưng kích thước và địa chỉ của mảng có thể thay đổi nhiều lần trong chương trình
  - Advantage: flexibility (arrays can grow or shrink during program execution)

Array Categories (continued)

- C and C++ arrays that include `static` modifier are static
- C and C++ arrays without `static` modifier are fixed stack-dynamic
- Ada arrays can be stack-dynamic

Array Categories (continued)

- C and C++ provide fixed heap-dynamic arrays
- C# includes a second array class `ArrayList` that provides fixed heap-dynamic
- Perl and JavaScript support heap-dynamic arrays

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";
```

Array Initialization

- Some language allow initialization at the time of storage allocation
  - Arrays of strings in C and C++

```
char *names [] = {"Bob", "Jake", "Joe"];
```

  - Java initialization of String objects

```
String[] names = {"Bob", "Jake", "Joe"};
```

Arrays Operations

- APL provides the most powerful array processing operations for vectors and matrixes as well as unary operators (for example, to reverse column elements)

Arrays Operations

- Ada **allows array assignment but also catenation**

- Fortran **provides *elemental* operations because they are between pairs of array elements**
  - For example, + operator between two arrays results in an array of the sums of the element pairs of the two arrays

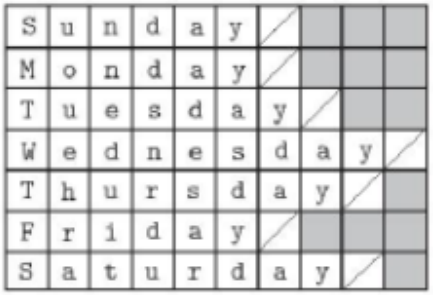
Rectangular and Jagged Arrays

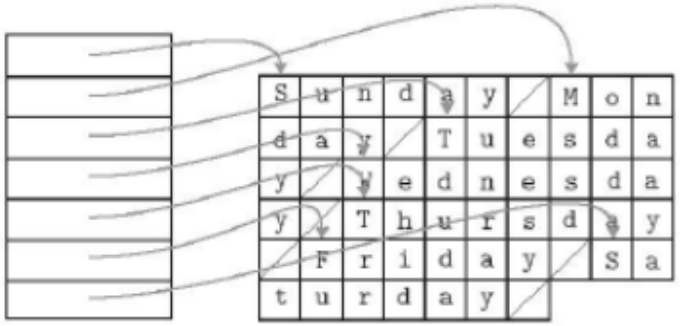
- **Mảng răng cưa**
- Là mảng nhiều chiều (2 chiều) nhưng có số phần tử của các dòng không bằng nhau
- Hướng tiếp cận là xem mảng hai chiều là một mảng một chiều mà mỗi phần tử của nó là một mảng một chiều

- Ưu điểm :tiết kiệm không gian vùng nhớ

Rectangular and Jagged Arrays

An example in C/C++





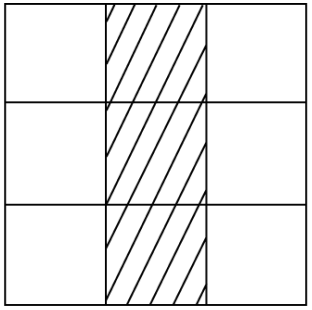

```
char days[][10] =
{"Sunday", "Monday",
"Tuesday", "Wednesday",
"Thursday", "Friday",
"Saturday"};

…

days[2][3] == 's';
```

```
char *days[] =
{"Sunday", "Monday",
"Tuesday", "Wednesday",
"Thursday", "Friday",
"Saturday"};

…

days[2][3] == 's';
```

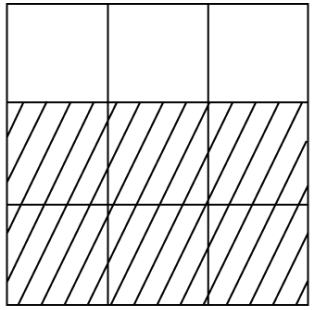
Slices

- A slice is some substructure of an array; nothing more than a referencing mechanism
- Slices are only useful in languages that have array operations

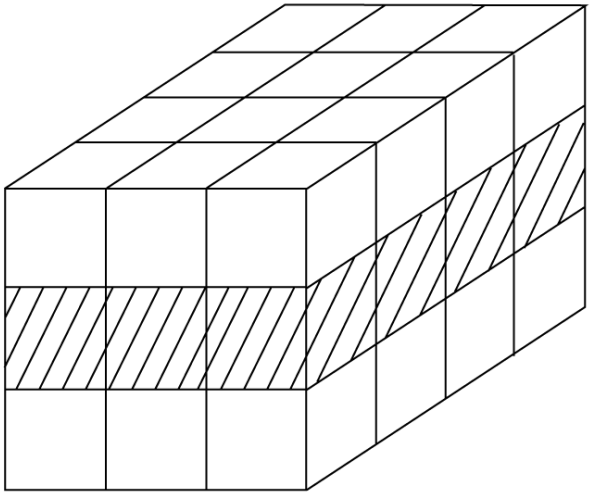
Slices Examples in Fortran 95



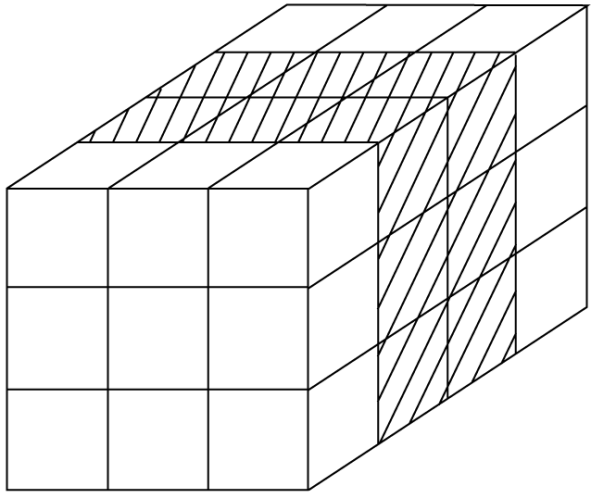
MAT (1:3, 2)



MAT (2:3, 1:3)



CUBE (2, 1:3, 1:4)



CUBE (1:3, 1:3, 2:3)

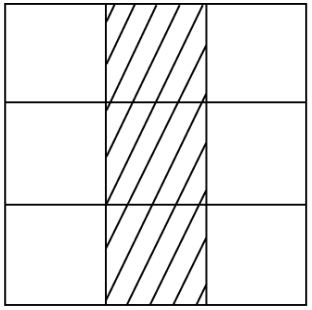
Slice Examples

- Fortran 95

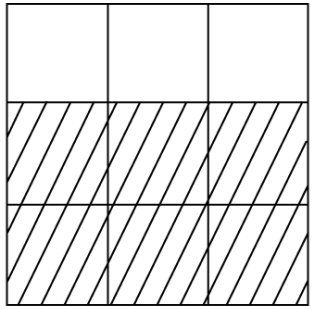
```
Integer, Dimension (10) :: Vector
Integer, Dimension (3, 3) :: Mat
Integer, Dimension (3, 3) :: Cube
```

`Vector (3:6)` is a four element array

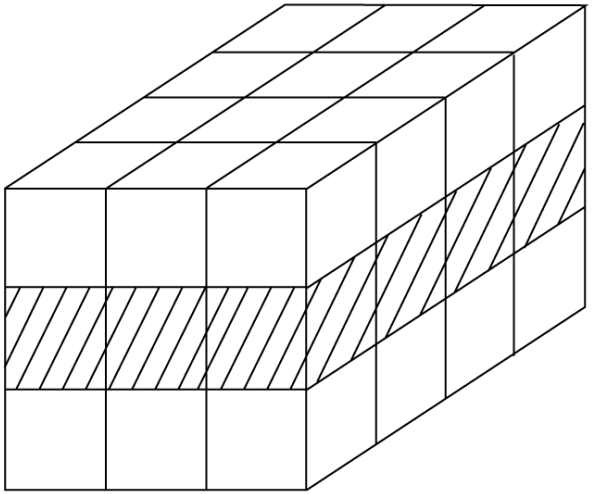
Slices Examples in Fortran 95



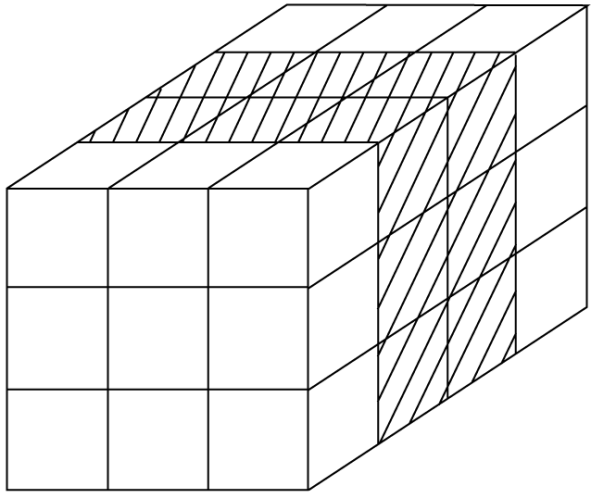
MAT (1:3, 2)



MAT (2:3, 1:3)



CUBE (2, 1:3, 1:4)



CUBE (1:3, 1:3, 2:3)

Implementation of Arrays

- **Access function maps subscript expressions to an address in the array**

- **Access function for single-dimensioned arrays:**

  address(list[k]) = address (list[lower_bound])
  + ((k-lower_bound) * element_size)

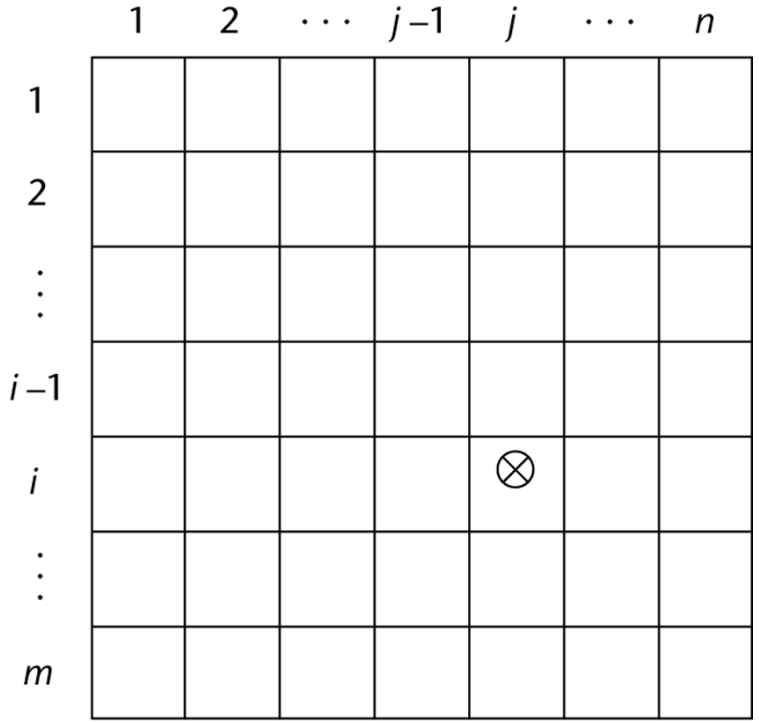
Accessing Multi-dimensioned Arrays

- Two common ways:
  - **Row major order (by rows) – used in most languages**
  - column major order (by columns) – used in Fortran

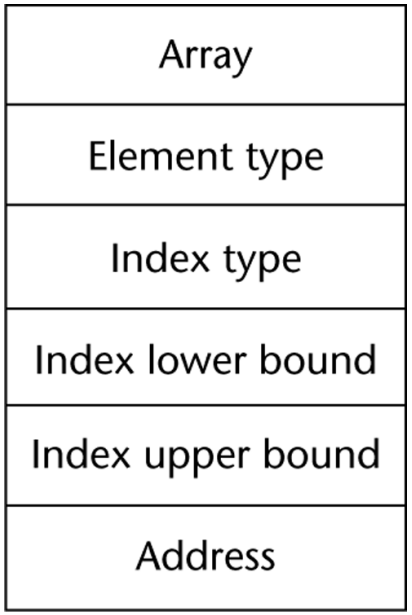
Locating an Element in a Multi-dimensioned Array

- General format

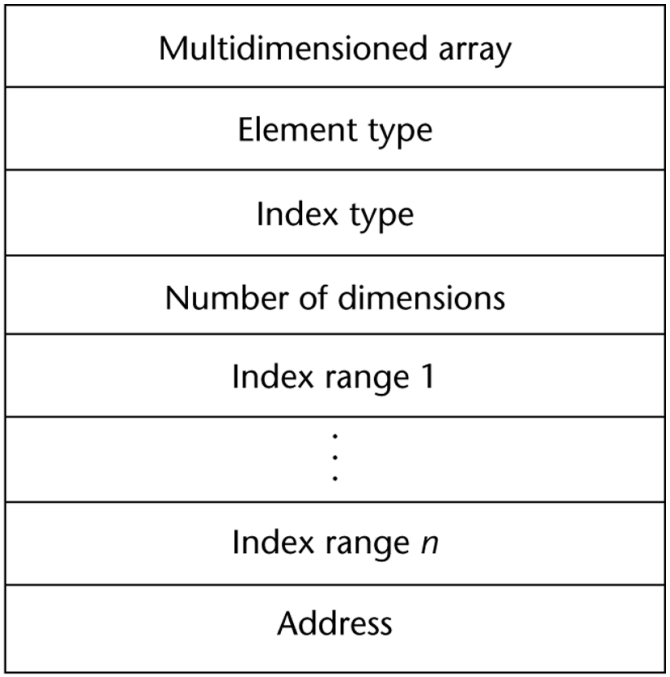
  Location (a[I,j]) = address of a [row_lb,col_lb] + (((I – row_lb) * n) + (j – col_lb)) * element_size



Compile-Time Descriptors



Single-dimensioned array



Multi-dimensional array

Associative Arrays

- **Mảng liên hợp :**
- Là một mảng không có thứ tự, các phần tử của mảng được truy xuất thông qua khóa
- Mỗi phần tử của mảng liên hợp bao gồm hai thành phần <Key,Value>
- Một số vấn đề gặp phải khi thiết kế mảng liên hợp
  - Cách thức tham chiếu đến một phần tử của mảng
  - Kích thước của mảng liên hợp là tĩnh hay động

Associative Arrays

- Trong ngôn ngữ C++ mảng liên hợp được gọi là **HashTable**
- Kích thước của **HashTable** là động
  - **HashTable danhsach=new HashTable();**
  - **danhsach.Add(“apple”,”Trai tao”);**
  - **danhsach[“orange”]=“Trai cam”;**

Record Types

- A *record* is a possibly heterogeneous aggregate of **data elements in which the individual elements are identified by names**

- Design issues:
  - **What is the syntactic form of references to the field?**

Definition of Records

- COBOL uses level numbers to show nested records; others use recursive definition

- Record Field References

  1. COBOL

  field_name OF record_name_1 OF ... OF record_name_n

  2. Others (dot notation)

  record_name_1.record_name_2. ... record_name_n.field_name

Definition of Records in COBOL

```
• 02 EMP-NAME.
       05 FIRST PIC X(20).
       05 MID   PIC X(10).
       05 LAST  PIC X(20).
   02 HOURLY-RATE PIC 99V99.
```

Definition of Records in Ada

- Record structures are indicated in an orthogonal way

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

References to Records

- Most language use dot notation

```
Emp_Rec.Name
```

- Fully qualified references must include all record names

Operations on Records

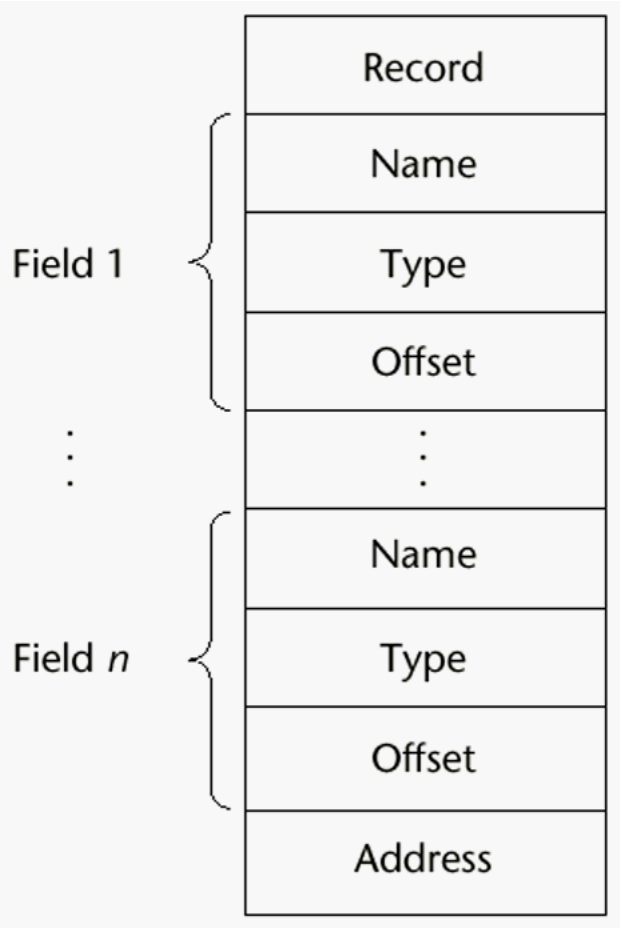
- **Assignment is very common** if the types are identical
- **Ada allows record comparison**
- Ada records can be initialized with aggregate literals
- COBOL provides `MOVE CORRESPONDING`
  - **Copies a field of the source record to the corresponding field in the target record**

Evaluation and Comparison to Arrays

- Straight forward and safe design
- **Records are used when collection of data values is heterogeneous**
- **Access to array elements is much slower than access to record fields, because subscripts are dynamic (field names are static)**

Implementation of Record Type

Offset address relative to the beginning of the records is associated with each field



Unions Types

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

- Design issues
  - Should type checking be required?
  - Should unions be embedded in records?

Ví dụ

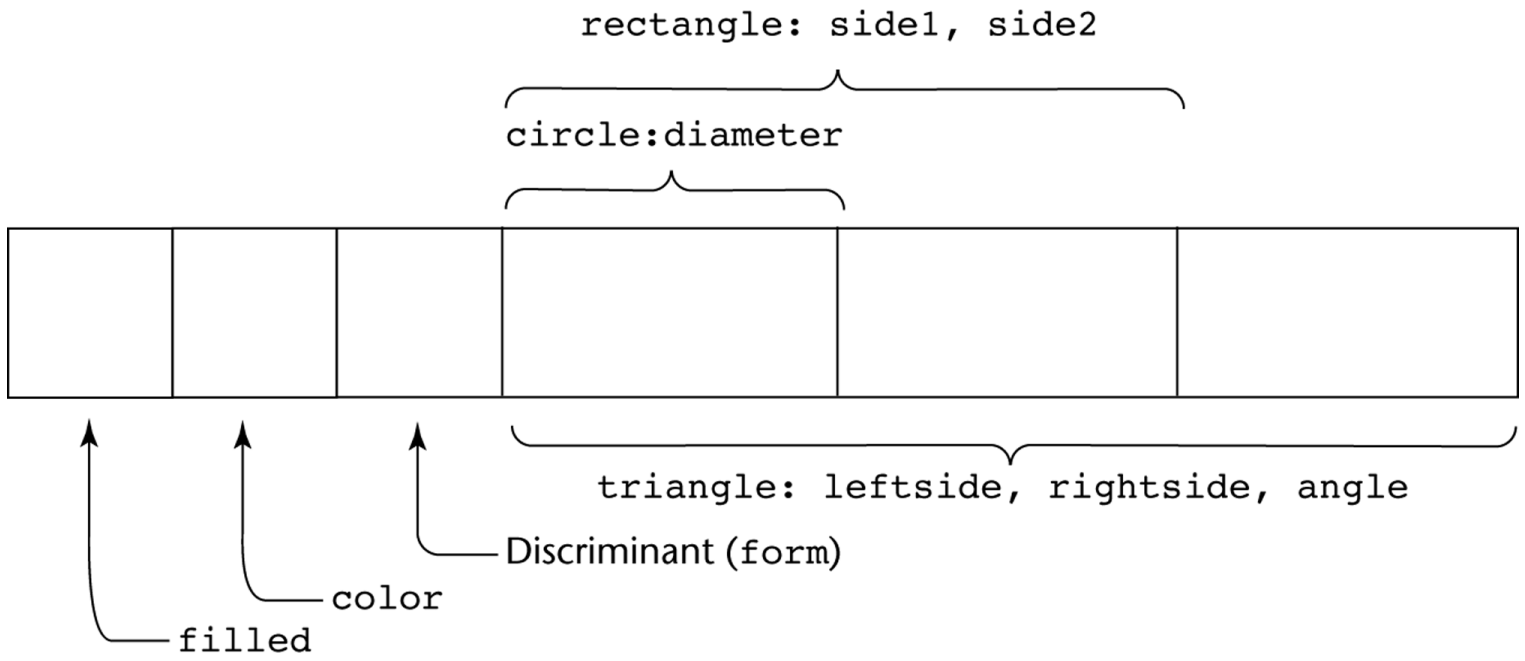
trong pascal khai báo bảng ghi có sử dụng bộ chỉ định kiểu như sau :

```
type rec_var =
   record tag : Boolean of
      true : (blint : integer);
      false : (blreal : real);
   end;

var x: rec_var; y: real;
x.tag := true;   { it is an integer }
x.blint := 47;   { ok }
x.tag := false;  { it is a real }
y := x.blreal;
```

Ada Union Types

```
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;
```

Ada Union Type Illustrated



A discriminated union of three shape variables

Evaluation of Unions

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

Pointer and Reference Types

- 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**
- **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*)

Design Issues of Pointers

- Phạm vi và thời gian sống của biến con trỏ ?
- Thời gian sống của bộ nhớ Heap ?
- Có hạn chế kiểu dữ liệu mà con trỏ mà con trỏ có thể trỏ đến hay không ?
- Con trỏ dùng quản lý bộ nhớ động (truy xuất trực tiếp) hay bộ nhớ truy xuất gián tiếp hay cả hai ?
- Hỗ trợ kiểu con trỏ hay kiểu tham chiếu hay cả hai ?

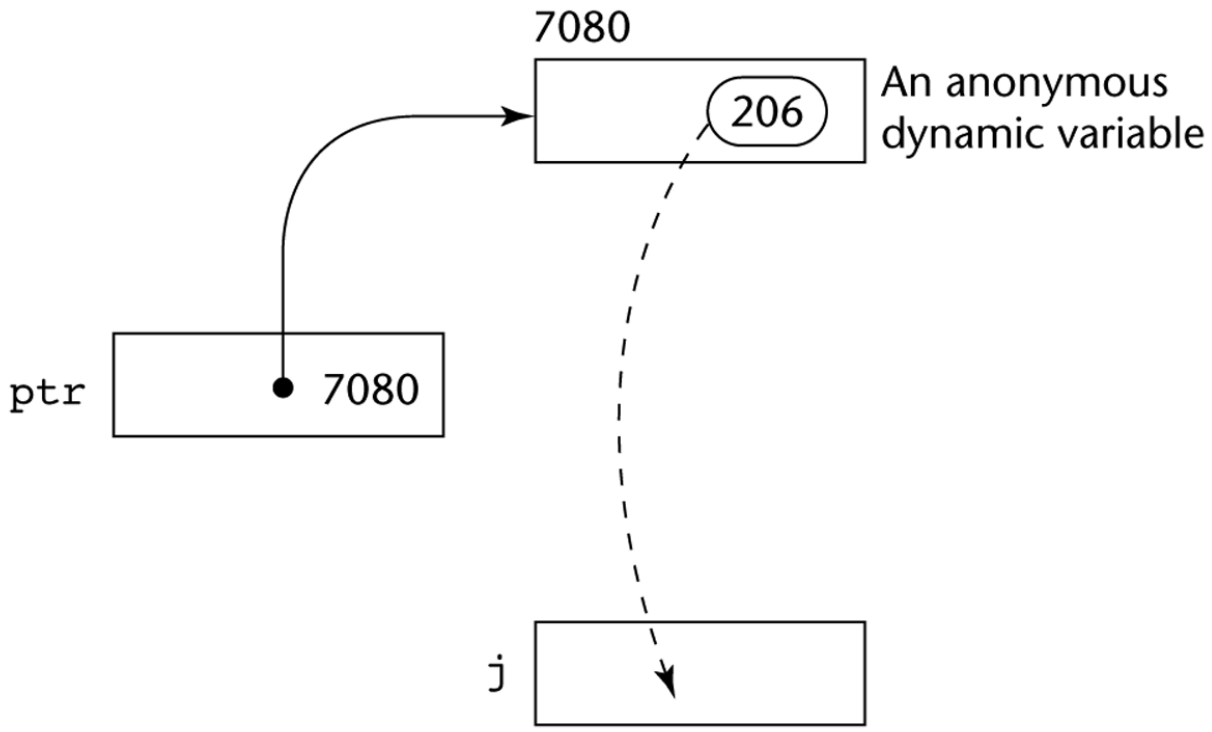
Pointer Operations

- **Two fundamental operations: assignment and dereferencing**
- Assignment is used to set a pointer variable's value to some useful address
- 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`

Pointer Assignment Illustrated



The assignment operation j = *ptr

Problems with Pointers

- **Con trỏ treo (Dangling pointers)**
  - Con trỏ trỏ đến một vùng nhớ động của một biến đã bị xóa
  - Ví dụ

```
int *p1,*p2;
p1= new int();
*p1=5;
p2=p1;
delete(p1);
printf ("%d",*p2);
```

Problems with Pointers

- **Tạo rác trong bộ nhớ Heap**
  - Ví dụ con trỏ p1 được thiết lập chỉ đến một vùng nhớ trong bộ nhớ Heap
  - Gán p1 cho một vùng nhớ khác ,vùng nhớ cũ sẽ tạo thành rác, không sử dụng được

- Int *p1,*p2;
- P1= new int();
- P2= new int();
- P1=p2;

Pointers in Ada

- **Some dangling pointers are disallowed**

Pointers in C and C++

- Vô cùng linh hoạt nhưng phải được sử dụng cẩn thận
- Con trỏ có thể trỏ bất kỳ biến nào bất kể khi nào nó được phân bổ
- “Dereferencing” rõ ràng
- void * có thể trỏ đến bất kỳ loại nào và có thể kiểm tra kiểu

Pointer Arithmetic in C and C++

```
float stuff[100];
float *p;
p = stuff;
```

`*(p+5)` is equivalent to `stuff[5]` and `p[5]`

`*(p+i)` is equivalent to `stuff[i]` and `p[i]`

Con trỏ trong ngôn ngữ Pascal

- Con trỏ dùng quản lý bộ nhớ động (truy xuất trực tiếp) hay bộ nhớ truy xuất gián tiếp
- Truy xuất tường minh
- Bị con trỏ treo

Reference Types

- C++ includes a special kind of pointer type called a ***reference type*** **that is used primarily for formal parameters**
  - Advantages of both pass-by-reference and pass-by-value

Reference Types

- Java **extends C++'s reference variables and allows them to replace pointers entirely**
  - References refer to call instances

C# includes both the references of Java and the pointers of C++

Representations of Pointers

- Hầu hết các máy tính biểu diễn con trỏ bằng một giá trị đơn
- Bộ xử lý biểu diễn địa chỉ bằng hai giá trị segment và offset

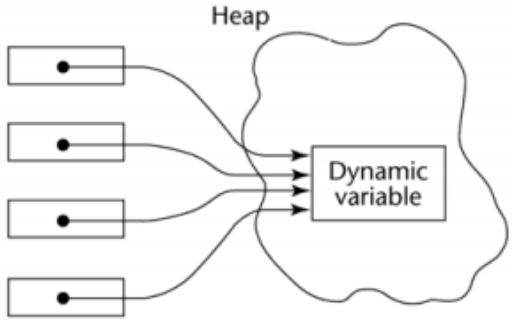
Dangling Pointer Problem

Tombstone (bia mộ)

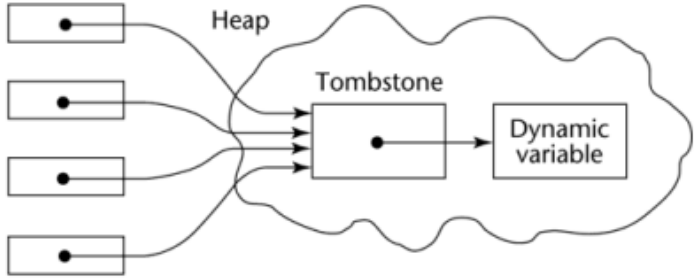
- Thêm một con trỏ trung gian trỏ đến địa chỉ của biến,con trỏ này là Tombstone
- Các con trỏ trỏ đến biến thông qua Tombstone
- Khi xóa hết tất cả các con trỏ trỏ đến Tombstone, thì con trỏ Tombstone sẽ gán= null

Dangling Pointer Problem

Implementation of Pointer and Reference Types (cont.)



(a) Without tombstones



(b) With tombstones

Dangling Pointer Problem

Dùng khóa và chìa khóa

- khi cấp phát một vùng nhớ cho con trỏ thì trên con trỏ và vùng nhớ được cấp phát một giá trị gọi là <chìa khóa, địa chỉ> trên con trỏ và một giá trị là "ổ khóa" trên ô nhớ
- khi truy xuất ô nhớ, hệ thống sẽ kiểm tra giá trị khóa và ổ khóa có trùng không, nếu trùng thì cho truy xuất
- khi một con trỏ xóa thì thuộc tính khóa của nó sẽ xóa

Heap Management

- **Thu dọn rác trong bộ nhớ**
  - Nhiều ngôn ngữ lập trình cố gắng không tạo rác trên heap khi lập trình nhưng điều này rất khó
  - Có nhiều ngôn ngữ lập trình xây dựng hệ thống dọn rác tự động

Reference Counter

- Khi có một con trỏ trỏ đến một vùng nhớ thì tăng biến đếm lên một
- Khi xóa một con trỏ trỏ đến vùng nhớ sẽ giảm biến đếm xuống 1
- Và chỉ cho phép giải phóng vùng nhớ khi biến đếm =0

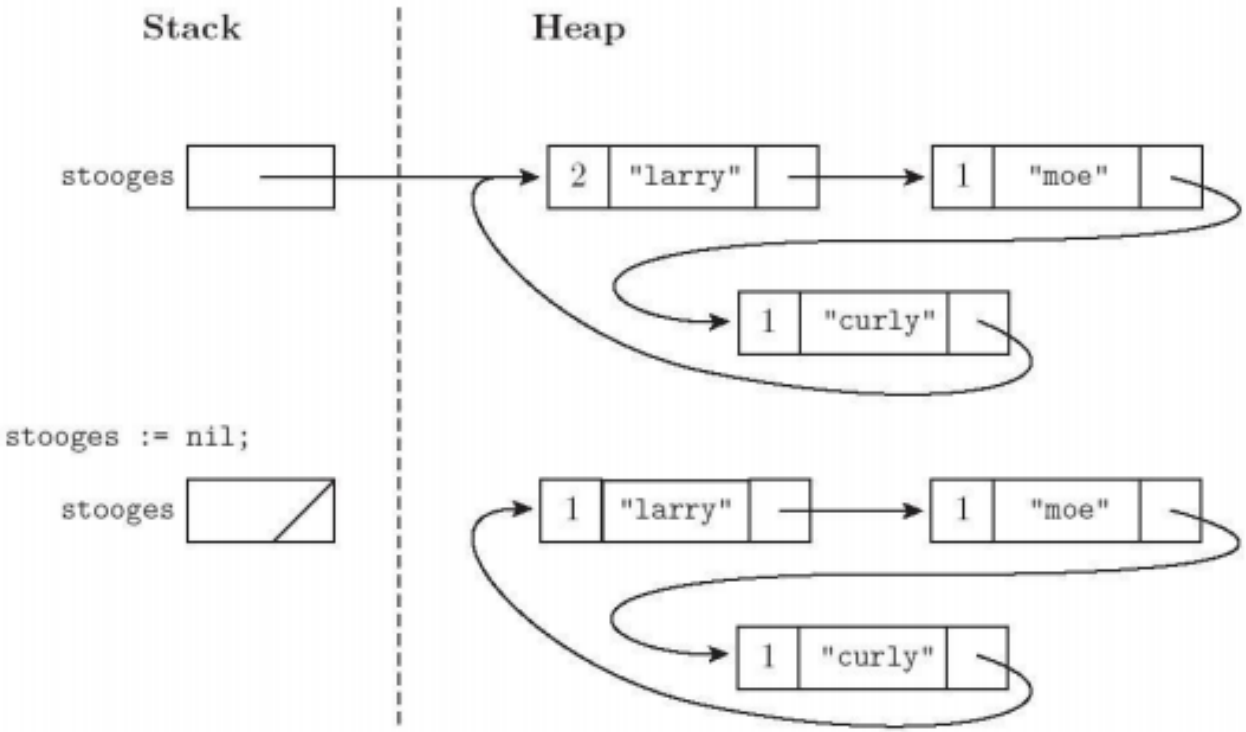
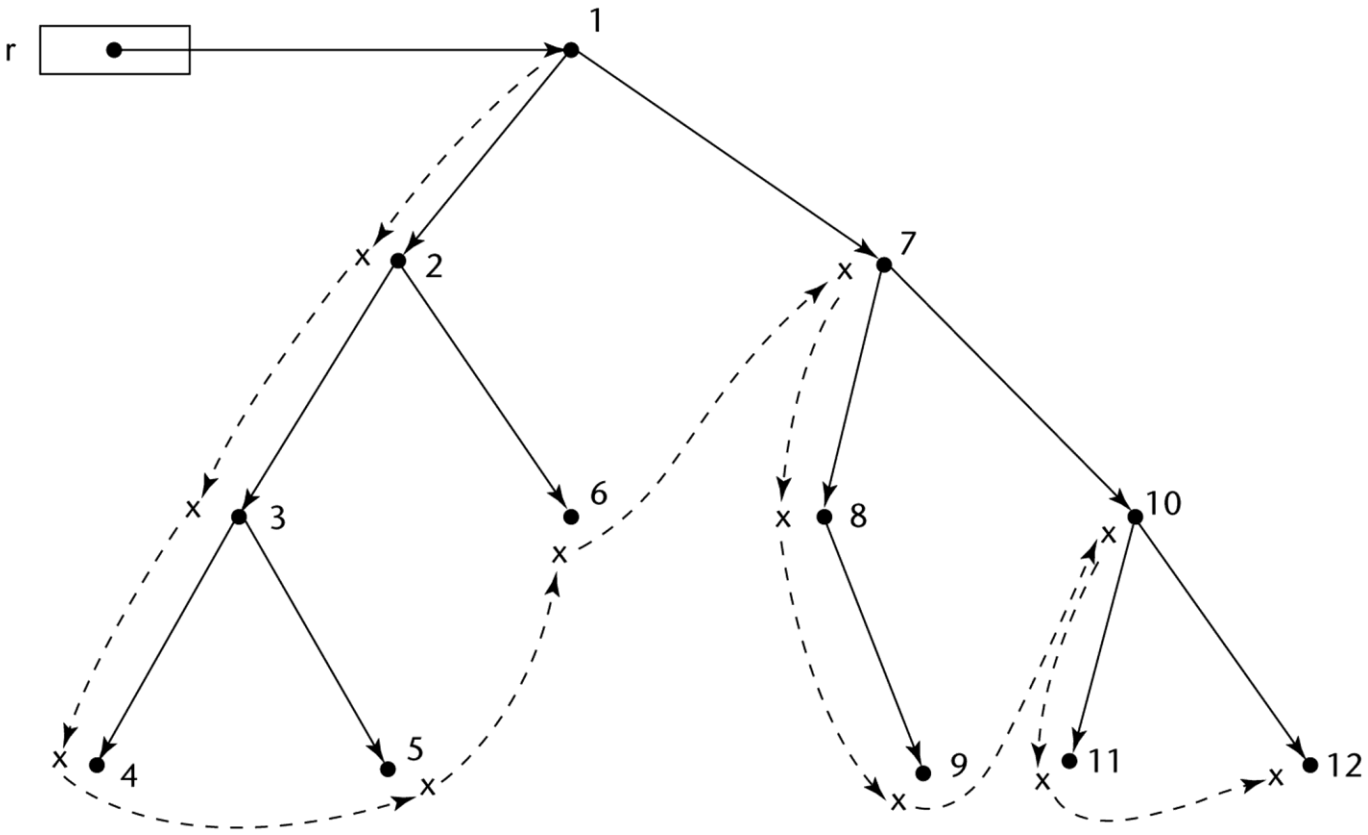


Figure 7.17: **Reference counts and circular lists.** The list shown here cannot be found via any program variable, but because it is circular, every cell contains a nonzero count.

Marking Algorithm



Dashed lines show the order of node_marking

Summary

- The data types of a language are a large part of what determines that language's style and usefulness
- The primitive data types of most imperative languages include numeric, character, and Boolean types
- The user-defined enumeration and subrange types are convenient and add to the readability and reliability of programs
- Arrays and records are included in most languages
- Pointers are used for addressing flexibility and to control dynamic storage management