

Concepts

Introduction

Primitive Data Types

Array Types



CONCEPTS

- Introduction
- Names
- Variables
- The concept of binding
- Scope
- Scope and lifetime
- Referencing Environments
- Named constants

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

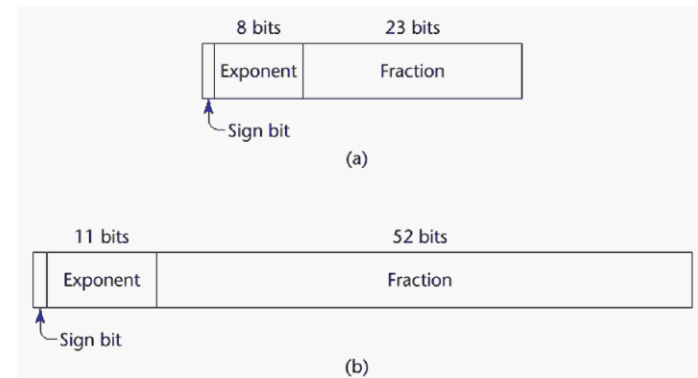
- Almost all programming languages provide a set of *primitive data types*
-  Primitive data types: Those not defined in terms of other data types
- Some primitive data types are merely reflections of the hardware 
- Others require only a little non-hardware support for their implementation

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: Complex

- Some languages support a complex type, e.g., C99, Fortran, and Python
- Each value consists of two floats, the real part and the imaginary part
- Literal form (in Python):
`(7 + 3j)`, where 7 is the real part and 3 is the imaginary part


Primitive Data Types: Decimal

- For business applications (money)
 - Essential to COBOL
 - C# offers a decimal data type
- Store a fixed number of decimal digits, in coded form (BCD)
- *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 (UCS-2)
 - Includes characters from most natural languages
 - Originally used in Java
 - C# and JavaScript also support Unicode
- 32-bit Unicode (UCS-4)
 - Supported by Fortran, starting with 2003

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

- What types are legal for subscripts?
- Are subscripting expressions in element references range checked?
- When are subscript ranges bound?
- When does allocation take place?
- What is the maximum number of subscripts?
- Can array objects be initialized?
- Are any kind of slices supported?

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
- Ada: integer or enumeration (includes Boolean and char)
- Java: integer types only
- Index range checking
 - C, C++, Perl, and Fortran do not specify range checking
 - Java, ML, C# specify range checking
 - In Ada, the default is to require range checking, but it can be turned off

Subscript Binding and Array Categories

- *Static*: subscript ranges are statically bound and storage allocation is static (before run-time)
 - Advantage: efficiency (no dynamic allocation)
- *Fixed stack-dynamic*: subscript ranges are statically bound, but the allocation is done at declaration time
 - Advantage: space efficiency

Subscript Binding and Array Categories (continued)

- *Stack-dynamic*: subscript ranges are dynamically bound and the storage allocation is dynamic (done at run-time)
 - Advantage: flexibility (the size of an array need not be known until the array is to be used)
- *Fixed heap-dynamic*: similar to fixed stack-dynamic: storage binding is dynamic but fixed after allocation (i.e., binding is done when requested and storage is allocated from heap, not stack)

Subscript Binding and Array Categories (continued)

- **Heap-dynamic:** binding of subscript ranges and storage allocation is dynamic and can change any number of times
 - **Advantage:** flexibility (arrays can grow or shrink during program execution)

Subscript Binding and Array Categories (continued)

- C and C++ arrays that include `static` modifier are static
- C and C++ arrays without `static` modifier are fixed stack-dynamic
- C and C++ provide fixed heap-dynamic arrays
- C# includes a second array class `ArrayList` that provides fixed heap-dynamic
- Perl, JavaScript, Python, and Ruby 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";
```

- Arrays of strings in C and C++

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

- Java initialization of String objects

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

Heterogeneous Arrays

- A *heterogeneous array* is one in which the elements need not be of the same type
- Supported by Perl, Python, JavaScript, and Ruby

Array Initialization

- C-based languages

- `int list [] = {1, 3, 5, 7}`
- `char *names [] = {"Mike", "Fred", "Mary Lou"};`

- Ada

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

- Python

- List comprehensions

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


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)
- Ada allows array assignment but also catenation
- Python's array assignments, but they are only reference changes. Python also supports array catenation and element membership operations
- Ruby also provides array 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

Type Checking

- Generalize the concept of operands and operators to include subprograms and assignments
- *Type checking* is the activity of ensuring that the operands of an operator are of compatible types
- A *compatible type* is one that is either legal for the operator, or is allowed under language rules to be implicitly converted, by compiler-generated code, to a legal type
 - This automatic conversion is called a *coercion*.
- A *type error* is the application of an operator to an operand of an inappropriate type

Type Checking (continued)

- If all type bindings are static, nearly all type checking can be static
- If type bindings  are dynamic, type checking must be dynamic
- A programming language is *strongly typed* if type errors are always detected
- Advantage of strong typing: allows the detection of the misuses of variables that result in type errors

Strong Typing

Language examples:

- FORTRAN 95 is not: parameters, EQUIVALENCE
- C and C++ are not: parameter type checking can be avoided; unions are not type checked
- Ada is, almost (UNCHECKED CONVERSION is loophole)
(Java and C# are similar to Ada)

Strong Typing (continued)

- Coercion rules strongly affect strong typing--they can weaken it considerably (C++ versus Ada)
- Although Java has just half the assignment coercions of C++, its strong typing is still far less effective than that of Ada

Names

- Design issues for names:
 - Are names case sensitive?
 - Are special words reserved words or keywords?

Names (continued)

- Length
 - If too short, they cannot be connotative
 - Language examples:
 - FORTRAN 95: maximum of 31
 - C99: no limit but only the first 63 are significant; also, external names are limited to a maximum of 31
 - C#, Ada, and Java: no limit, and all are significant
 - C++: no limit, but implementers often impose one

Names (continued)

- Special characters
 - PHP: all variable names must begin with dollar signs
 - Perl: all variable names begin with special characters, which specify the variable's type
 - Ruby: variable names that begin with @ are instance variables; those that begin with @@ are class variables

Names (continued)

- Case sensitivity
 - Disadvantage: readability (names that look alike are different)
 - Names in the C-based languages are case sensitive
 - Names in others are not
 - Worse in C++, Java, and C# because predefined names are mixed case (e.g. `IndexOutOfBoundsException`)

Names (continued)

- Special words
 - An aid to readability; used to delimit or separate statement clauses
 - A *keyword* is a word that is special only in certain contexts, e.g., in Fortran
 - `Real VarName` (*Real is a data type followed with a name, therefore Real is a keyword*)
 - `Real = 3.4` (*Real is a variable*)
 - A *reserved word* is a special word that cannot be used as a user-defined name
 - Potential problem with reserved words: If there are too many, many collisions occur (e.g., COBOL has 300 reserved words!)

Variables

- A variable is an abstraction of a memory cell
- Variables can be characterized as a sextuple of attributes:
 - Name
 - Address
 - Value
 - Type
 - Lifetime
 - Scope

Variables Attributes

- Name - not all variables have them
- Address - the memory address with which it is associated
 - A variable may have different addresses at different times during execution
 - A variable may have different addresses at different places in a program
 - If two variable names can be used to access the same memory location, they are called **aliases**
 - Aliases are created via pointers, reference variables, C and C++ unions
 - Aliases are harmful to readability (program readers must remember all of them)

Variables Attributes (continued)

- *Type* - determines the range of values of variables and the set of operations that are defined for values of that type; in the case of floating point, type also determines the precision
- *Value* - the contents of the location with which the variable is associated
 - The l-value of a variable is its address
 - The r-value of a variable is its value
- *Abstract memory cell* - the physical cell or collection of cells associated with a variable

The Concept of Binding

A *binding* is an association, such as between an attribute and an entity, or between an operation and a symbol

- *Binding time* is the time at which a binding takes place.

Possible Binding Times

- Language design time -- bind operator symbols to operations
- Language implementation time-- bind floating point type to a representation
- Compile time -- bind a variable to a type in C or Java
- Load time -- bind a C or C++ `static` variable to a memory cell)
- Runtime -- bind a nonstatic local variable to a memory cell

Static and Dynamic Binding

- A binding is *static* if it first occurs before run time and remains unchanged throughout program execution.
- A binding is *dynamic* if it first occurs during execution or can change during execution of the program

Categories of Variables by Lifetimes

- Static--bound to memory cells before execution begins and remains bound to the same memory cell throughout execution, e.g., C and C++ `static` variables
 - Advantages: efficiency (direct addressing), history-sensitive subprogram support
 - Disadvantage: lack of flexibility (no recursion)

Variable Attributes: Scope

- The *scope* of a variable is the range of statements over which it is visible
- The *nonlocal variables* of a program unit are those that are visible but not declared there
- The scope rules of a language determine how references to names are associated with variables

Static Scope

- Based on program text
- To connect a name reference to a variable, you (or the compiler) must find the declaration
- *Search process*: search declarations, first locally, then in increasingly larger enclosing scopes, until one is found for the given name
- Enclosing static scopes (to a specific scope) are called its *static ancestors*; the nearest static ancestor is called a *static parent*
- Some languages allow nested subprogram definitions, which create nested static scopes (e.g., Ada, JavaScript, Fortran 2003, and PHP)

Scope (continued)

- Variables can be hidden from a unit by having a "closer" variable with the same name
- Ada allows access to these "hidden" variables
 - E.g., `unit.name`

Blocks

- A method of creating static scopes inside program units--from ALGOL 60
- Example in C:

```
void sub() {  
    int count;  
    while (...) {  
        int count;  
        count++;  
        ...  
    }  
    ...  
}
```

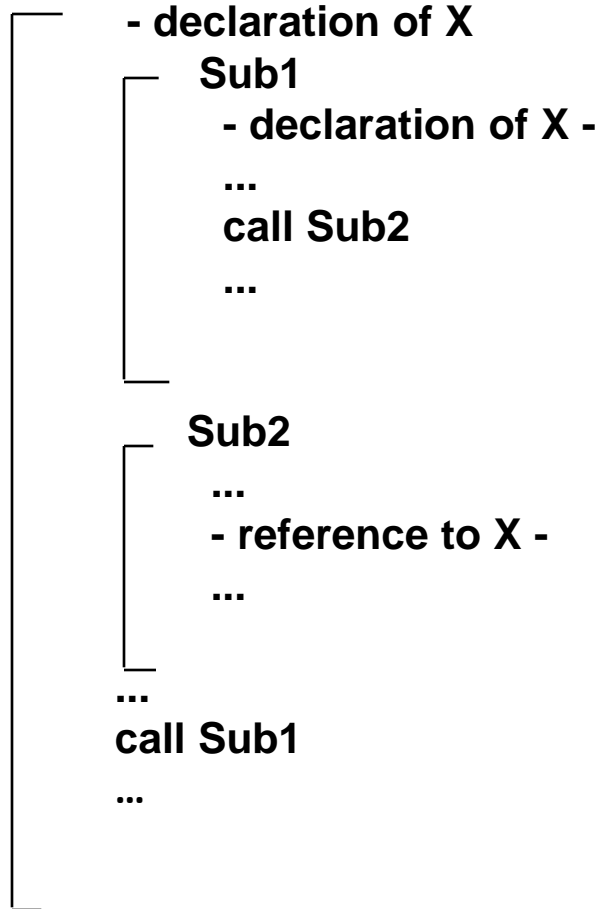
- Note: legal in C and C++, but not in Java and C# - too error-prone

Global Scope

- C, C++, PHP, and Python support a program structure that consists of a sequence of function definitions in a file
 - These languages allow variable declarations to appear outside function definitions
- C and C++ have both declarations (just attributes) and definitions (attributes and storage)
 - A declaration outside a function definition specifies that it is defined in another file

Scope Example

Big



Big calls Sub1
Sub1 calls
Sub2
Sub2 uses X

Scope Example

- Static scoping
 - Reference to X is to Big's X
- Dynamic scoping
 - Reference to X is to Sub1's X
- Evaluation of Dynamic Scoping:
 - Advantage: convenience
 - *Disadvantages:*
 1. While a subprogram is executing, its variables are visible to all subprograms it calls
 2. Impossible to statically type check
 3. Poor readability- it is not possible to statically determine the type of a variable

Scope and Lifetime

- Scope and lifetime are sometimes closely related, but are different concepts
- Consider a `static` variable in a C or C++ function

Array Programs

An array is defined as the collection of similar type of data items stored at contiguous memory locations.

C program to sort the array of elements in ascending order using bubble sort method.

```
#include<stdio.h>
void main ()
{
    int i, j,temp;
    int a[10] = { 10, 9, 7, 101, 23, 44, 12, 78, 34, 23};
    for(i = 0; i<10; i++)
    {
        for(j = i+1; j<10; j++)
        {
            if(a[j] > a[i])
            {
                temp = a[i];
                a[i] = a[j];
                a[j] = temp;
            }
        }
    }
}
```

Array Programs

```
printf("Printing Sorted Element List ...\n");  
for(i = 0; i<10; i++)  
{  
    printf("%d\n",a[i]);  
}  
}
```


Array Programs

Java Program to demonstrate the addition of two matrices in Java

```
class Testarray5
{
    public static void main(String args[])
    {
        //creating two matrices
        int a[][]={{ 1,3,4},{3,4,5}};
        int b[][]={{ 1,3,4},{3,4,5}};
        //creating another matrix to store the sum of two matrices
        int c[][]=new int[2][3];
        //adding and printing addition of 2 matrices
        for(int i=0;i<2;i++)
        {
            for(int j=0;j<3;j++)
            {
                c[i][j]=a[i][j]+b[i][j];
                System.out.print(c[i][j]+" ");
            }
            System.out.println();//new line
        }
    }
}
```

Scope

Scope of Variable

Each variable is defined and can be used within its scope and determines that wherein the program this variable is available to use. The scope means the lifetime of that variable. It means the variable can only be accessed or visible within its scope.

Scope of goto

The goto statement is a jump statement which is sometimes also referred to as unconditional jump statement. The goto statement can be used to jump from anywhere to anywhere within a function.

// C program to check if a number is even or odd using goto statement

Scope

```
#include <stdio.h>

void checkEvenOrNot(int num) // function to check even or not
{
    if (num % 2 == 0)
        goto even; // jump to even
    else
        goto odd; // jump to odd
even:
    printf("%d is even", num);
    return; // return if even
odd:
    printf("%d is odd", num);
}

int main()
{
    int num = 26;

    checkEvenOrNot(num);
    return 0;
}
```

Scope

Local Vs. Global in Python

```
v1 = "Hey, I am Global Variable." #globalvariable
def func1():
    v2="Hey, I am Local Variable." #localvariable
    print(v2)
func1()    #calling func1

def func2():
    print(v1)
func2()    #callin func2
```

Output:

Hey, I am a Local Variable

Hey, I am Global Variable

In the above program, we have taken one global variable v1 and one local variable v2. Since v1 is global, it can be easily accessed in any function, and v2 is local; it is used only within its declared function. But if we try to use v1 in func1, it will give an error.

Scope

Scope in Java

Local Vs. Global Variable in Java

In Java, there is no concept of global variables; since Java is an Object-oriented programming language, everything is a part of the Class. But if we want to make a variable globally accessible, we can make it static by using a **static** Keyword.

```
class Demo
{
    // static variable
    static int a = 10;
    // non-static or local variable
    int b = 20;
}
```

Scope

```
public class Main
{
    public static void main(String[] args)
    {
        Demo obj = new Demo();
        // accessing the non-static variable
        System.out.println("Value of non-
static variable is: " + (obj.b));
        // accessing the static variable

        System.out.println("Value of static variable is:" + (Demo.a));
    }
}
```

Scope

Output:

Value of non-static variable is: 20

Value of static variable is:10

In the above program, we have used one local variable or non-static variable and one static variable. The local variable can be accessed by using the object of the Demo class, whereas the static variable can be accessed using the name of the class.