UNIT-2

Variables

A variable is an abstraction of a memory cell **or** variable is a name given to a memory location

- **Variables can be characterized as a sixtuple of attributes:**
  - Name
  - Address
  - Value
  - Type
  - Lifetime
  - Scope

5.3.1 Name

Variable names are the most common names in programs. One of the fundamental attributes of variables is names. Names are also associated with subprograms, formal parameters, and other program constructs. The term *identifier* is often used interchangeably with *name*.

5.2.1 Design Issues

The following are the primary design issues for names:

• Are names case sensitive?
• Are the special words of the language reserved words or keywords?

5.2.2 Name Forms

A **name** is a string of characters used to identify some entity in a program

- Case sensitivity
  - Disadvantage: readability (names that look alike are different ex: Rose, rose)
    - Names in the C-based languages are case sensitive i.e uppercase and lowercase letters in names are distinct
    - Names in others are not

- Length
  - Language examples:
    - FORTRAN 95: maximum of 31 characters in it's name
    - C89 has no length limitation on its internal names, but only the first 31 are significant; external names are limited to a maximum of 6 characters
    - 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 in most programming languages have the same form: a letter followed by a string consisting of letters, digits, and underscore characters (_). Although the use of underscore characters to form names was widely used in the 1970s and 1980s, that practice is now far less popular.
        - All variable names in PHP must begin with a dollar sign.
        - In Perl, any name that begins with $ is a scalar that can store either a string or a numeric value. If name begins with @, it is an array.

- 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!)

5.3.2 **Address (l – value)**

- Address of a variable is the machine memory address with which it is associated
- .**The address of a variable is sometimes called** its **l-value**, because the address is what is required when the name of a variable appears in the left side of an assignment
    - A variable may have different addresses at different times during execution
    - A variable may have different addresses at different places in a program

Aliases: 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)

Example in C

```
int x, *p1, *p2;
p1 = &x;
p2 = &x;
```

p1, p2 are aliases because both are points to x.

5.3.3 Type

The **type** of a variable determines the range of values the variable can store and the set of operations that are defined for values of the type. For example, the **int** type in Java specifies a value range of -2147483648 to 2147483647 and arithmetic operations for addition, subtraction, multiplication, division,and modulus. C int takes the values between the range -32768 to + 32767

5.3.4 Value (r- value)

- *Value* of a variable is the contents of memory cell or cells associated with the variable

- A variable’s value is sometimes called its **r-value** because it is what is required when the name of the variable appears in the right side of an assignment statement.
- To access the *r*-value, the *l*-value must be determined first.

5.3.5 Scope

- The *scope* of a variable is the range of statements in which it is visible

5.4 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.
- The values of attributes must be set before they can be used

  Binding can occur in various ways

- **Language design time binding --** bind operator symbols to operations For example,the asterisk symbol (*) is usually bound to the multiplication operation at language design time.
- **Language implementation time binding--** bind floating point type to a representation
- **Compile time(translation time) binding --** variable type is bound in C or Java at compile time
- **load time binding --** bind a C or C++ static variable to a memory cell at load time
- **Runtime binding --** bind a non static local variable to a memory cell at runtime and binding can be modified repeatedly during execution

First four categories refer to binding before runtime. These refer to static binding.
Last category refers to dynamic binding

Consider the following Java assignment statement:

```
count = count + 5;
```

Some of the bindings and their binding times for the parts of this assignment statement are as follows:
• The type of count is bound at compile time.
• The set of possible values of count is bound at compiler design time.
• The meaning of the operator symbol + is bound at compile time, when the types of its operands have been determined.
• The internal representation of the literal 5 is bound at compiler design time.
• The value of count is bound at execution time with this statement.

5.4.1 Binding of Attributes to Variables

- 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

5.4.2 Type Bindings

- Before a variable can be referenced in a program, it must be bound to a data type.

-The two important aspects of this binding are

- how the type is specified and when the binding takes place?
- If static, the type may be specified by either an explicit or an implicit declaration

5.4.2.1 Static Type Binding

Before a variable can be referenced in a program, it must be bound to a data type. They can be specified statically using explicit declaration like in C or implicit declaration like in FORTRAN. It can also be specified dynamically by associating a value during runtime like in Java Script.

- An *explicit declaration* is a program statement used for declaring the types of variables
- An *implicit declaration* is a default mechanism for specifying types of variables (the first appearance of the variable in the program)
- In Fortran,If the identifier begins with one of the letters I, J, K, L, M, or N, or their lowercase versions, it is implicitly declared to be Integer type; otherwise, it is implicitly declared to be Real type.
- For example, in Perl any name that begins with $ is a scalar, which can store either a string or a numeric value. If a name begins with @, it is an array.If it begins with a %, It a hash structure
- FORTRAN, BASIC, and Perl provide implicit declarations (Fortran has both explicit and implicit)
  - Advantage: writability
  - Disadvantage: reliability

- Another kind of implicit type declarations called **type inference**. For example, in C# a **var** declaration of a variable must include an initial value, whose type is made the type of the variable.

Consider the following declarations:

var sum = 0;
var total = 0.0;
var name = "Fred";
The types of sum, total, and name are **int**, **float**, and **string**, respectively. These are statically typed variables—their types are fixed for lifetime of the unit in which they are declared.

5.4.2.2 Dynamic Type Binding

- In dynamic type binding, the type of a variable is not specified by a declaration statement.
- The variable is bound to a type when it is assigned a value in an assignment statement. When the assignment statement is executed, the variable being assigned is bound to the type of the value of the expression on the right side of the assignment.

- In Python, Ruby, JavaScript, and PHP, type binding is dynamic.

For example, a JavaScript script may contain the following statement:

```
list = [2, 4.33, 6, 8];

list = 17.3;
```

The variable list is bound to a type when it is assigned a value in an assignment statement.

- Advantage: flexibility (generic program units)
- Disadvantages:
  - The cost of implementing dynamic attribute binding is high
  - Type checking must be done at run time. Furthermore, every variable must have a run-time descriptor associated with it to maintain the current type. The storage used for the value of a variable must be of varying size, because different type values require different amounts of storage
  - Type error detection by the compiler is missing
  - Dynamic type binding allows any variable to be assigned a value of any type.

5.4.3 Storage Bindings and Lifetime

The memory cell to which a variable is taken from a pool of available memory. This process is called **allocation (** getting a cell from some pool of available cells). **Deallocation** is the process of placing a memory cell that has been unbound from a variable back into the pool of available memory (putting a cell back into the pool)

The **lifetime** of a variable is the time during which the variable is bound to a specific memory location. So, the lifetime of a variable begins when it is bound to a specific cell and ends when it is unbound from that cell.

Categories of Variables by Lifetimes

These categories are named static, stack-dynamic, explicit heap-dynamic, and implicit heap-dynamic. variables that are defined inside subprograms are called **local variables.**
local variables can be either static or stack-dynamic.

5.4.3.1 **Static Variables**

Static variables are those that are bound to memory cells before program execution begins and remain bound to those same memory cells until program execution terminates.

e.g., C and C++ static variables, all FORTRAN 77 variables.

Advantages of static variables

- Efficiency (direct addressing),

-History sensitive : variables retain their values between separate executions of the subprogram

Disadvantages:

- **lack of flexibility** (a language that has only static variables **cannot support recursive subprograms**.)

-storage cannot be shared among variables.

For example, suppose a program has two subprograms, both of which require large arrays. Furthermore, suppose that the two subprograms are never active at the same time. If the arrays are static, they cannot share the same storage for their arrays.

5.4.3.2 Stack-Dynamic Variables

- **Stack-dynamic variables** are those whose storage bindings are created when their declaration statements are elaborated, but whose types are statically bound.
- Variable declarations of a function are elaborated when the method is called. Therefore, elaboration occurs during run time.
- For example, the variable declarations that appear at the beginning of a Java method are elaborated when the method is called and the variables defined by those declarations are deallocated when the method completes its execution.
- As their name indicates, stack-dynamic variables are allocated from the run-time stack.
- All attributes other than storage are statically bound to stack-dynamic scalar variables.

  Advantage: allows recursion;

  Disadvantages:
  - **Overhead of allocation and deallocation**
  - **Inefficient references (indirect addressing)**

Ex:
In Java, C++, and C#, variables defined in methods are by default stack dynamic.
Local variables of C and C++ functions are stack dynamic unless specifically declared to be **static**
For example, in the following C (or C++) function, the variable sum is static and count is stack dynamic.

```
int adder ( int list[ ], int listlen )
{
   static int sum = 0;
   int count;
   for (count = 0; count < listlen; count ++)
   sum += list [count];
   return sum;
 }
```

5.4.3.3 Explicit Heap-Dynamic Variables

- **Explicit heap-dynamic variables** are nameless (abstract) memory cells that are allocated and deallocated by explicit run-time instructions written by the programmer.

- These variables, which are allocated from and deallocated to the heap,

- Referenced only through pointer or reference variables, e.g. dynamic objects in C++ (via new and delete), all objects in Java

- The heap is a collection of storage cells whose organization is highly disorganized because of the unpredictability of its use

- The pointer or reference variable that is used to access an explicit heap-dynamic variable

- In C++, the allocation operator, named **new**, uses a type name as its operand. When executed, an explicit heap-dynamic variable of the operand type is created and its address is returned.
-
- Because an explicit heap-dynamic variable is bound to a type at compile time, that binding is static. However, such variables are bound to storage at the time they are created, which is during run time.

As an example of explicit heap-dynamic variables, consider the following C++ code segment:

```
int  *p; // Create a pointer
p = new int; // Create the heap-dynamic variable
…….
…….
delete p; // Deallocate the heap-dynamic variable
// to which intnode points
```

In this example

an explicit heap-dynamic variable of **int** type is created by the **new** operator.
This variable can then be referenced through the pointer, p.
Later, the variable is deallocated by the **delete** operator.
C++ requires the explicit deallocation operator **delete**
Java uses implicit garbage collection

advantages: **provides dynamic storage management**

The disadvantages

-difficulty of using pointer and reference variables correctly,
-Inefficient " cost of allocation and deallocation"

5.4.3.4 Implicit Heap-Dynamic Variables

Implicit **heap-dynamic variables** are bound to heap storage only when they are assigned values. Allocation and de-allocation caused by assignment statements

– all variables in APL; all strings and arrays in Perl, JavaScript, and PHP

eg: highs = [74, 84, 86, 90, 71];

it is an array of five numeric values in javascript

advantage flexibility allowing generic code to be written.

disadvantages

run-time overhead of maintaining all the dynamic attributes, which could include array subscript types and ranges, among
Loss of some error detection by the compiler

5.5 Scope

The *scope* of a variable is the range of statements in which the variable is visible. A variable is **visible** in a statement if it can be referenced in that statement

A variable is **local** in a program unit or block if it is declared there

The nonlocal variables of a program unit or block are those that are visible within the program unit or block but are not declared there

Global variables are a special category of nonlocal variables.

The scope rules of a language determine how a particular occurrence of a name is associated with a variable or The scope rules of a language determine how references to names are associated with variables

SCOPE RULES: 1. Static scoping 2. Dynamic scoping

5.5.1 Static Scope

ALGOL 60 introduced the method of binding names to nonlocal variables called **static scoping**, **Static scoping is so named because the scope of a variable can be statically determined—that is, prior to execution**. This permits program reader (and a compiler) to determine the type of every variable in the program simply by examining its source code.
There are two types of static- scoped languages

- Languages have nested sub programs
- Languages does not have nested sub programs

To connect a name reference to a variable, you (or the compiler) must find the declaration by searching

Search process:

- search declarations first locally, then in increasingly larger enclosing scopes, until correct declaration 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, C does not allows nested subprograms)

Consider the following JavaScript function, big, in which the two functions sub1 and sub2 are nested:

```
function big()
{
    function sub1()
       {
        var x = 7;
         sub2();
       }
    function sub2()
       {
       var y = x;
       }
 var x = 3;
 sub1();
}
```

Under static scoping, the reference to the variable x in sub2 is to the x declared in the procedure big. This is true because the search for x begins in the procedure in which the reference occurs, sub2, but no declaration for x is found there. The search continues in the static parent of sub2, big, where the declaration of x is found. The x declared in sub1 is ignored, because it is not in the static ancestry of sub2. In some languages that use static scoping, regardless of whether nested subprograms are allowed, some variable declarations can be hidden from some other code segments. For example, consider again the JavaScript function big. The variable x is declared in both big and in sub1, which is nested inside big. Within sub1, every simple reference to x is to the local x. Therefore, the outer x is hidden from sub1.

5.5.5 Evaluation of Static Scoping

Static scoping provides a method of nonlocal access that works well in many situations.

Problems with Static Scoping

Static scoping provides a method of nonlocal access that works well in many situations. however, it has problems.

- First, in most cases it allows more (too much) access to both variables and subprograms than is necessary.

- Second problem related to program evolution. Software is highly dynamic—programs that are used regularly continually change. These changes often result in restructuring, thereby destroying the initial structure that restricted variable and subprogram access.

5.5.6 Dynamic Scope

Dynamic scoping is based on calling sequence of subprograms. The scope of variables in APL, SNOBOL4, and the early versions of LISP is dynamic scoped languages . Perl and Common LISP also allow variables to be declared to have dynamic scope, although the default scoping mechanism in these languages is static.

Dynamic scoping is based on the calling sequence of subprograms, not on their spatial relationship to each other. Thus, the scope can be determined only at run time.

```
function big()
{
      function sub1()
          {
           var x = 7;
            sub2();
          }
     function sub2()
          {
          var y = x;
          }
 var x = 3;
 sub1();
}
```

Dynamic-scoping rules apply to nonlocal references. When a variable is referredin a subprogram is searched for type definitions. When the search of local declarations fails, the declarations of the dynamic parent, or calling function, are searched. If a declaration for `x` is not found there, the search continues in that function's dynamic parent(calling sub program),and so forth, until a declaration for `x` is found. If none is found in any dynamic ancestor, it is a run-time error.

Consider the two different call sequences for `sub2` in the earlier example. First, `big` calls `sub1`, which calls `sub2`. In this case, the search proceeds from the local procedure, `sub2`, to its caller, `sub1`, where a declaration for `x` is found. So, the reference to `x` in `sub2` in this case is to the `x` declared in `sub1`. Next, `sub2` is called directly from `big`. In this case, the dynamic parent of `sub2` is `big`, and the reference is to the `x` declared in `big`.

5.5.7 Evaluation of Dynamic Scoping (advantage and disadvantage of dynamic scoping)

When dynamic scoping is used, the correct attributes of nonlocal variables visible to a program statement cannot be determined statically. A statement in a subprogram that contains a reference to a nonlocal variable can refer to different nonlocal variables during different executions of the subprogram

Advantage: convenience

problems with dynamic scoping

- While a subprogram is executing, its variables are visible to all subprograms it calls
- Impossible to statically type check
- Poor readability- it is not possible to statically determine the type of a variable

5.6 **Scope and Lifetime**

Sometimes the scope and lifetime of a variable appear to be related. For example, consider a variable that is declared in a Java method that contains no method calls. The scope of such a variable is from its declaration to the end of the method.

The lifetime of that variable is the period of time beginning when the method is entered and ending when execution of the method terminates. Although the scope and lifetime of the variable are clearly not the same, because static scope is a textual, or spatial, concept whereas lifetime is a temporal concept, they at least appear to be related in this case. This apparent relationship between scope and lifetime does not hold in other situations. In C and C++, for example, a variable that is declared in a function using the specifier **static** is statically bound to the scope of that function and is also statically bound to storage. So, its scope is static and local to the function, but its lifetime extends over the entire execution of the program of which it is a part.

Consider the following C++ functions:

```
void printheader() {
. . .
} /* end of printheader */
void compute() {
int sum;
. . .
printheader();
} /* end of compute */
```

The scope of the variable `sum` is completely contained within the `compute` function. It does not extend to the body of the function `printheader`, although `printheader` executes in the midst of the execution of `compute`. However, the lifetime of `sum` extends over the time during which `printheader` executes. Whatever storage location `sum` is bound to before the call to `printheader`, that binding will continue during and after the execution of `printheader`.

Referencing environment

-
- The *referencing environment* of a statement is the collection of all names that are visible in the statement
- In a static-scoped language, it is the local variables plus all of the visible variables in all of the enclosing scopes
- A subprogram is active if its execution has begun but has not yet terminated
- In a dynamic-scoped language, the referencing environment is the local variables plus all visible variables in all active subprograms

Type checking

- *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
- If all type bindings are static, nearly all type checking can be static
- If type bindings are dynamic, type checking must be dynamic

Strong Typing

- 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
  - FORTRAN 95 is not strongly typed language
  - C and C++ are not not strongly typed because parameter type checking can be avoided; unions are not type checked
  - Ada , Java , C# are strongly typed languages

Primitive Data Types

Data types that are not defined in terms of other types are called **primitive data types**. Nearly all programming languages provide a set of primitive data types.

6.2.1 Numeric Types

Many early programming languages had only numeric primitive types. Numeric types still play a central role among the collections of types supported by modren languages.

6.2.1.1 Integer

The most common primitive numeric data type is **integer**. Many computers now support several sizes of integers..For example, Java includes four signed integer sizes: **byte**, **short**, **int**, and **long**. Some languages, for example, C++ and C#, include unsigned integer types, which are simply types for integer values without signs. Unsigned types are often used for binary data. A signed integer value is represented in a computer by a string of bits, with one of the bits (typically the leftmost) representing the sign. Most integer types are supported directly by the hardware. Long integer values can be specified as literals, as in the following example:
`243725839182756281923L`
A negative integer could be stored in sign-magnitude notation, in which the sign bit is set to indicate negative and the remainder of the bit string represents the absolute value of the number. however, does not lend itself to computer arithmetic. Most computers now use a notation called **twos complement** to store negative integers, which is convenient for addition and subtraction. In twos-complement notation, the representation of a negative integer is formed by taking the logical complement of the positive version of the number and adding one. Ones-complement notation is still used by some computers. In ones-complement notation, the negative of an integer is stored as the logical complement of its absolute value. Ones-complement notation has the disadvantage that it has two representations of zero.

6.2.1.2 Floating-Point

Floating-point data types model real numbers, but the representations are only approximations for many real values. For example, neither of the fundamental numbers _ or *e* (the base for the natural logarithms) can be correctly represented in floating-point notation. On most computers, floating point numbers are stored in binary. Another problem with floating-point types is the loss of accuracy through arithmetic operations.

Floating-point values are represented as fractions and exponents, a form that is borrowed from scientific notation. Most newer machines use the IEEE Floating-Point Standard 754 format. Most languages include two floating-point types, often called **float** and **double**. The float type is the standard size, usually being stored in four bytes of memory. The double type is provided for situations where larger fractional parts and/or a larger range of exponents is needed. Double-precision variables usually occupy twice as much storage as float variables and provide at least twice the number of bits of fraction. The collection of values that can be represented by a floating-point type is defined in terms of precision and range. **Precision** is the accuracy of the fractional part of a value, measured as the number of bits. **Range** is a combination of the range of fractions and, more important, the range of exponents. Figure 6.1 shows the IEEE Floating-Point Standard 754 format for single and double-precision representation (IEEE, 1985).

6.2.1.3 Complex

Some languages support a complex type, e.g. Fortran, and Python

- Complex values are represented as ordered pairs of floating-point values
- *Each value consists of two floats, the real part and the imaginary part*
- In Python, the imaginary part of a complex literal is specified
  by following it with a j or J—for example,
  (7 + 3j), where 7 is the real part and 3 is the imaginary part
- Languages that support a complex type include operations for arithmetic on complex values.

6.2.1.4 Decimal

- Decimal types Essential to COBO for business applications (money).C# offers a decimal data type. Decimal types have the advantage of being able to precisely store decimal values, at least those within a restricted range, which cannot be done with floating-point.
- Decimal data types store a fixed number of decimal digits, with the decimal point at a fixed position in the value and store a fixed number of decimal digits, in coded form (BCD)
- For example, the number 0.1 (in decimal) can be exactly represented in a decimal type, but not in a floating-point type,
- *Advantage*: accuracy
- . Disadvantage: range of values is restricted because no exponents are allowed and wastes memory

6.2.1.5 Boolean Types

Boolean types are perhaps the simplest of all types. Their range of values has only two elements: one for true and one for false. They were introduced in ALGOL 60 and have been included in most general-purpose languages designed since 1960. One popular exception is C89, in which numeric expressions are used as conditionals. In such expressions, all operands with nonzero values are considered true, and zero is considered false. Although C99 and C++ have a Boolean type, they also allow numeric expressions to be used as if they were Boolean. This is not the case in the subsequent languages, Java and C#. Boolean types are often used to represent switches or flags in programs.
A Boolean value could be represented by a single bit, but because a single bit of memory cannot be accessed efficiently on many machines, they are often stored in the smallest efficiently addressable cell of memory, typically a byte.

6.2.1.6 Character Types

Character data are stored in computers as numeric codings. Traditionally, the most commonly used coding was the 8-bit code ASCII (American Standard Code for Information Interchange), which uses the values 0 to 127 to code 128 different characters. ISO 8859-1 is another 8-bit character code, but it allows 256 different characters. Because of the globalization of business and the need for computers to communicate with other computers around the world, the ASCII character set became inadequate. In response, in 1991, the Unicode Consortium published the UCS-2 standard, a 16-bit character set. This character code is often called **Unicode**. Unicode includes the characters from most of the world's naturallanguages. The first 128 characters of Unicode are identical to those of ASCII. Java was the first widely used language to use the Unicode character set.

NON PRIMITIVE DATA TYPES AS FOLLOWS

6.3 **Character String Types**

A **character string type** is one in which the values consist of sequences of characters.

6.3.1 Design Issues

• Should strings be simply a special kind of character array or a primitive type?
• Should strings have static or dynamic length?

6.3.2 Strings and Their Operations

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

Some of the most commonly used library functions for character strings in C and C++ are `strcpy`, which moves strings; `strcat`, which catenates one given string onto another; `strcmp`, which lexicographically compares (by the order of their character codes) two given strings; and `strlen`, which returns the number of characters, not counting the null, in the given string.
A **substring reference** is a reference to a substring of a given string. substring references are called **slices**.

Character String Type in Certain Languages

- C and C++
  - Not primitive
  - Use **char** arrays and a library of functions that provide operations
- SNOBOL4 (a string manipulation language)
  - Primitive
  - Many operations, including elaborate pattern matching
- Fortran and Python
  - Primitive type with assignment and several operations
- Java

    – Primitive via the String class
- Perl, JavaScript, Ruby, and PHP
    - Provide built-in pattern matching, using regular expressions

Consider the following pattern expression:

```
/[A-Za-z][A-Za-z\d]+/
```

This pattern matches (or describes) the typical name form in programming languages. The brackets enclose character classes. The first character class specifies all letters; the second specifies all letters and digits (a digit is specified with the abbreviation `\d`). If only the second character class were included, we could not prevent a name from beginning with a digit. The plus operator following the second category specifies that there must be one or more of what is in the category. So, the whole pattern matches strings that begin with a letter, followed by one or more letters or digits.

6.3.3 String Length Options

- Static: COBOL, Java's String class
- *Limited Dynamic Length*: C and C++

2. In these languages, a special character is used to indicate the end of a string's characters, rather than maintaining the length. C and C++ actual length is indicated by a null character
    –

- *Dynamic* (no maximum): SNOBOL4, Perl, JavaScript
- Ada supports all three string length options

Static string length means length of the string is set when the string is created. We cannot change the length of the static strings.

Ex: String str = "ppl" ;

The second option is to allow strings to have varying length up to a declared and fixed maximum set by the variable's definition, as exemplified by the strings in C and the C-style strings of C++. These are called **limited dynamic length strings**. Such string variables can store any number of characters between zero and the maximum

Ex: char name[100]

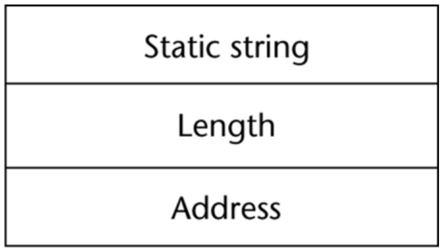
The third option is to allow strings to have varying length with no maximum, as in JavaScript, Perl, and the standard C++ library. These are called **dynamic length strings**.

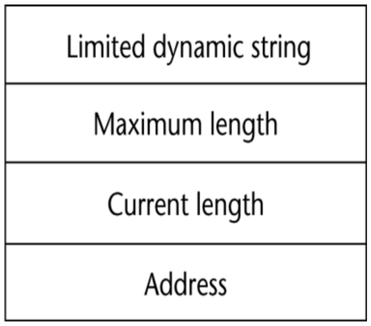
```
JavaScript Ex:  var str = "principles of " ;
                var str = str + "programming" ;
                alert(str);
```

```
O/P:    principles of programming
```

6.3.5 Implementation of Character String Types

- Static length: compile-time descriptor
- Limited dynamic length: may need a run-time descriptor for length (but not in C and C++)
- Dynamic length: need run-time descriptor; allocation/de-allocation is the biggest implementation problem





Compile-time descriptor for static strings   Run-time descriptor for limited dynamic strings

6.4 User-Defined Ordinal Types

- An ordinal type is one in which the range of possible values can be easily associated with the set of positive integers
- There are two user defined ordinal ordinal types

1. enumeration
2. subrange

6.4.1 **Enumeration Types**

An **enumeration type** is one in which all of the possible values, which are named constants, are provided, or enumerated, in the definition. Enumeration types provide a way of defining and grouping collections of named constants, which are called **enumeration constants**. The definition of a typical enumeration type is shown in the following C# example:

```
enum days {Mon, Tue, Wed, Thu, Fri, Sat, Sun};
```

The enumeration constants are typically implicitly assigned the integer values, 0, 1, . . . but can be explicitly assigned any integer literal in the type's definition.

The design issues for enumeration types are as follows:

• Is an enumeration constant allowed to appear in more than one type definition, and if so, how is the type of an occurrence of that constant in the program checked?
• Are enumeration values coerced to integer?

• Are any other types coerced to an enumeration type?

Syntax of Declaring an Enumerated Type in C

```
enum  typename  { identifier list };

int main( )

{
      enum  colors  { red, green, blue, yellow } ;
      printf( " red = %d\n", red) ;
      printf( " green = %d\n", green) ;
      printf( " blue = %d\n", blue) ;
      printf( " yellow = %d\n", yellow) ;
      return 0 ;
}
```

O/P is

```
red =0
green=1
blue = 2
yellow = 3
```

6.4.2 **Subrange Types**

- An ordered contiguous subsequence of an ordinal type
  - Example: 12..18 is a subrange of integer type
  - Subrange types were introduced by Pascal and are included in Ada.

6.4.2.1 Ada's subrange types

Ada' s **subtype** is a range constrained version of an existing type. Subtype is compatible with its parent type

```
type Days is (mon, tue, wed, thu, fri, sat, sun);
subtype Weekdays is Days range mon..fri;
subtype Index is Integer range 1..100;
Day1: Days;
Day2: Weekdays;
Day2 := Day1;
```

The assignment is legal unless the value of Day1 is Sat or Sun

6.4.2.2 Evaluation (**merits of sub range types)**

- Aid to readability
  - variables of sub range can store only certain range of values
- Reliability

– Assigning a value to a sub range variable that is outside the specified range is detected as an error

6.4.3 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

6.5 **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 subscripts: An element of an array is referred using the name of the array followed by variable known as **subscript or index.**

array-name(index) = element

6.5.1 **Design Issues**

The primary design issues specific to arrays are the following:

• What types are legal for subscripts?
• Are subscripting expressions in element references range checked?
• When are subscript ranges bound?
• When does array allocation take place?
• Can arrays be initialized when they have their storage allocated?
• What kinds of slices are allowed, if any?

6.5.2 **Arrays and Indices**

- 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
Ada allows any ordinal type to be used as subscripts, such as Boolean,character, and enumeration. For example, in Ada one could have the following:

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

Subscripting in Perl is a bit unusual in that although the names of all arrays begin with at signs (@), because array elements are always scalars and the names of scalars always begin with dollar signs ($), references to array elements use dollar signs rather than at signs in their names. For example, for the array @list, the second element is referenced with $list[1].

6.5.3 **Subscript Bindings and Array Categories**

The binding of the subscript type to an array variable is usually static, but the subscript value ranges are sometimes dynamically bound.

- In some languages, the lower bound of the subscript range is implicit. For example, in the C-based languages, the lower bound of all subscript ranges is fixed at 0; in Fortran 95+ it defaults to 1 but can be set to any integer literal.
- In some other languages, the lower bounds of the subscript ranges must be specified by the programmer.

Array Categories (based on subscript binding and binding to storage)

There are five categories of arrays, based on the binding to subscript ranges, the binding to storage, and from where the storage is allocated. The category names indicate the design choices of these three. In the first four of these categories, once the subscript ranges are bound and the storage is allocated, they remain fixed for the lifetime of the variable.When the subscript ranges are fixed, the array cannot change size.

A **static array** is one in which the subscript ranges are statically bound and storage allocation is static (done before run time). The advantage of static arrays is execution efficiency: No dynamic allocation or deallocation is required. The disadvantage is that the storage for the array is fixed for the entire execution time of the program.

Arrays declared in C and C++ functions that include the **static** modifier are static

A **fixed stack-dynamic array** is one in which the subscript ranges are statically bound, but the allocation is done at declaration elaboration time during execution. **The advantage of fixed stack-dynamic arrays over static arrays is space efficiency**. A large array in one subprogram can use the same space as a large array in a different subprogram, as long as both subprograms are not active at the same time. The same is true if the two arrays are in different blocks that are not active at the same time. **The disadvantage is the required allocation and deallocation time.**

Ex: Arrays that are declared in C and C++ functions (without the **static** specifier) are examples of fixed stack-dynamic arrays

A **stack-dynamic array** is one in which both the subscript ranges and the storage allocation are dynamically bound at elaboration time. Once the subscript ranges are bound and the storage is allocated, however, they remain fixed during the lifetime of the variable. The advantage of stack-dynamic arrays over static and fixed stack-dynamic arrays is flexibility. The size of an array need not be known until the array is about to be used.

Ada arrays can be stack dynamic, as in the following:

```
Get(List_Len);
declare
List : array (1..List_Len) of Integer;
begin
. . .
end;
```

In this example, the user inputs the number of desired elements for the array `List`. The elements are then dynamically allocated when execution reaches the **declare** block. When execution reaches the end of the block, the `List` array is de-allocated. C and C++ also provide fixed heap-dynamic arrays

A **fixed heap-dynamic array** is similar to a fixed stack-dynamic array, in that the subscript ranges and the storage binding are both fixed after storage is allocated. The differences are that both the subscript ranges and storage bindings are done when the user program requests them during execution, and the storage is allocated from the heap, rather than the stack. **The advantage** of fixed heap-dynamic arrays is flexibility—the array's size always fits the problem. The **disadvantage is** allocation time from the heap, which is longer than allocation time from the stack.

Ex. int *ptr = new int[100] ;

A **heap-dynamic array** is one in which the binding of subscript ranges and storage allocation is dynamic and can change any number of times during the array's lifetime. **The advantage of heap-dynamic arrays over the others is flexibility:** Arrays can grow and shrink during program execution as the need for space changes. The disadvantage is that allocation and deallocation take longer and may happen many times during execution of the program.
Ex :

```
      cin >> n;
 .    int *ptr
     ptr = new int[ n ]
```

The standard C library functions `malloc` and `free`, which are general heap allocation and deallocation operations, respectively, can be used for C arrays. C++ uses the operators **new** and **delete** to manage heap storage.

6.5.4 Array Initialization

- Some language allow initialization at the time of storage allocation

C, C++, Java, C# example

Ex: int list [ ] = {4, 5, 7, 83}

Character strings in C and C++

Ex: char name [ ] = “hello”;

Arrays of strings in C and C++

Ex: char *names [ ] = {“Bob”, “Jake”, “Joe”};

Java initialization of String objects

Ex: String[ ] names = {“Bob”, “Jake”, “Joe”};

Ada

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

6.5.5 **Array Operations**

- An array operation is one that operates on an array as a unit.
- The most common array operations are assignment, catenation, comparison for equality and inequality, and slices.
- The C-based languages do not provide any array operations, except through the methods of Java, C++, and C#.
- Perl supports array assignments but does not support comparisons.
- Ada allows array assignments, including those where the right side is an aggregate value rather than an array name.

.

- Arrays and their operations are the heart of APL; it is the most powerful array-processing language ever devised. In APL, the four basic arithmetic operations are defined for vectors (single-dimensioned arrays) and matrices, as well as scalar operands.

For example,
A + B
is a valid expression, whether A and B are scalar variables, vectors, or matrices.
APL includes a collection of unary operators for vectors and matrices,
some of which are as follows (where V is a vector and M is a matrix):

øV reverses the elements of V
øM reverses the columns of M

6.5.6 **Rectangular and Jagged Arrays**

A **rectangular array** is a multidimensioned array in which all of the rows have the same number of elements and all of the columns have the same number of elements. Rectangular arrays model rectangular tables exactly.
A **jagged array** is one in which the lengths of the rows need not be the same. For example, a jagged matrix may consist of three rows, one with 5 elements, one with 7 elements, and one with 12 elements. This also applies to the columns and higher dimensions. So, if there is a third dimension (layers), each
layer can have a different number of elements. Jagged arrays are made possible when multidimensioned arrays are actually arrays of arrays.
For example, a matrix would appear as an array of single-dimensioned arrays.
C, C++, and Java support jagged arrays but not rectangular arrays. In those languages, a reference to an element of a multidimensioned array uses a separate pair of brackets for each dimension. For example,
`myArray[3][7]`

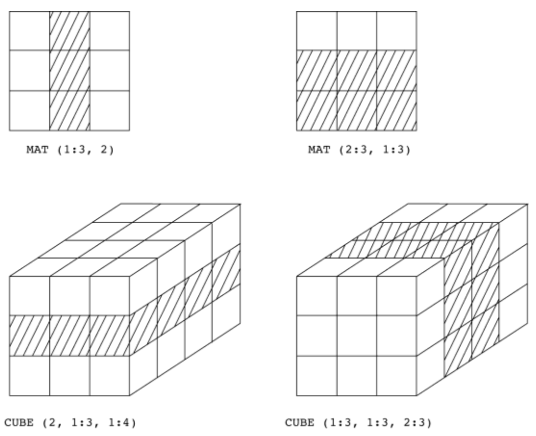
Fortran, Ada, C#, support rectangular arrays. (C# also support jagged arrays.) In these cases, all subscript expressions in references to elements are placed in a single pair of brackets. For example, `myArray[3, 7]`

6.5.7 **Slices**

A **slice** of an array is some substructure of that array. For example, if `A` is a matrix, then the first row of `A` is one possible slice, as are the last row and the first column. It is important to realize that a slice is not a new data type. Rather, it is a mechanism for referencing part of an array as a unit.

- Fortran 95

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

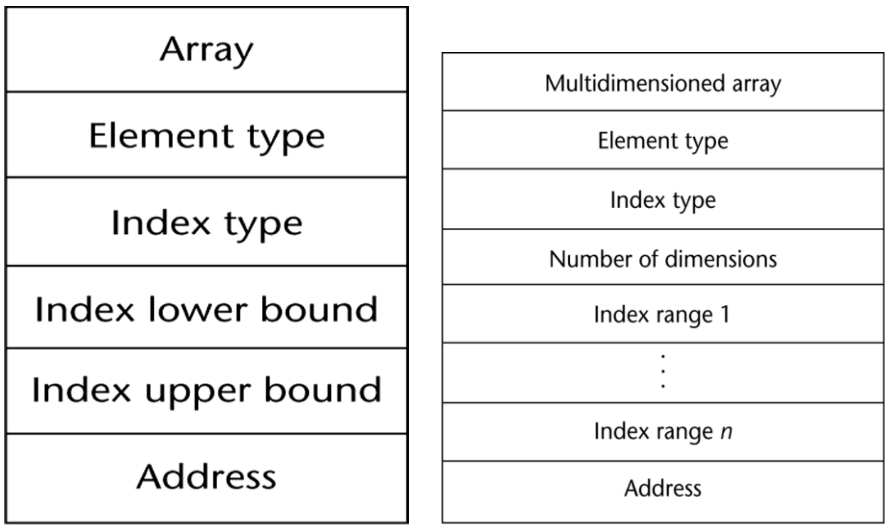


6.5.8 Implementation of Arrays

- Access function maps subscript expressions to an address in the array
- Access function for single-dimensioned arrays:

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

The compile-time descriptor for single-dimensioned arrays can have the form shown in Figure 6.4. The descriptor includes information required to construct the access function.



Single-dimensioned array Multi-dimensional array
figure 6.4 compiler time descriptors

Accessing Multi-dimensioned Arrays

There are two ways in which multidimensional arrays can be mapped to one dimension: row major order and column major order. **row major order** used in most languages

In **row major order**, the elements of the array that have as their first subscript the lower bound value of that subscript are stored first, followed by the elements of the second value of the first subscript, and so forth. If the array is a matrix, it is stored by rows.

If the array is a matrix, it is stored by rows. For example, if the matrix had the values

3 4 7
6 2 5
1 3 8

it would be stored in row major order, it would have the following order in memory:

3, 4, 7, 6, 2, 5, 1, 3, 8

In **column major order**, the elements of an array that have as their last subscript the lower bound value of that subscript are stored first, followed by the elements of the second value of the last subscript, and so forth. If the array is a matrix, it is stored by columns.

If the example matrix were stored in column major order, it would have the following order in memory:

3, 6, 1, 4, 2, 3, 7, 5, 8

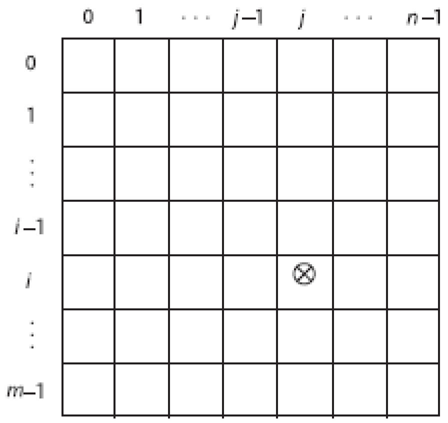
Column major order is used in Fortran, but other languages that have true multidimensional arrays use row major order.

The access function for a multidimensional array is the mapping of its base address and a set of index values to the address in memory of the element specified by the index values. The access function for two-dimensional arrays stored in row major order can be developed as follows.

To get an actual address value, the number of elements that precede the desired element must be multiplied by the element size. Now, the access function can be written as

Figure 6.5

The location of the `[i,j]` element in a matrix



$$\begin{aligned}\text{location}(\texttt{a[i, j]}) = {} & \text{address of a[row\_lb, col\_lb]} \\ & - (((\text{row\_lb} * \text{n}) + \text{col\_lb}) * \text{element\_size}) \\ & + (((\texttt{i} * \text{n}) + \texttt{j}) * \text{element\_size})\end{aligned}$$

Associative arrays

- An *associative array* is an unordered collection of data elements that are indexed by an equal number of values called *keys*
  In associative array, user-defined keys must be stored in the structure. so each element in *associative array is a pair of entities, a key and a value*

- **Design issues:**

  - What is the form of references to elements?
  - Is the size static or dynamic?

Associative arrays in Perl

In Perl, associative arrays are called **hashes**, because in the implementation their elements are stored and retrieved with hash functions. The namespace for Perl hashes is distinct: Every hash variable name must begin with a percent sign (`%`). Each hash element consists of two parts: a key, which is a string, anda value, which is a scalar (number, string, or reference). Hashes can be set to literal values with the assignment statement, as

%hi_temps = ("Mon" => 77, "Tue" => 79, “Wed” => 65, …);

To reference the value 79 , $high_temp(Tue) can be used

- To add the element to hash the following notation is used
  $hi_temps{"thurs"} = 83;

- To delete an element from a hash a **delete** operator can be used as follows

  delete $hi_temps{"Tue"};

  The entire hash can be emptied by assigning the empty literal to it, as in
  @hi_temps = ( );

Perl hash can grow and shrink dynamically. A hash is efficient when element searches are required and an array is useful when all elements are to be processed

6.7 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?
  - Are elliptical references allowed

6.7.1 Definition of Records in COBOL

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

```
01 EMP-REC.

      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_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;
```

For example, consider the following declaration:

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

These assignment statements create a table (record) named employee with two elements (fields) named name and hourlyRate, both initialized.

6.7.2 **References to Record Fields**

References to the individual fields of records are syntactically specified by several different methods, two of which name the desired field and its enclosing records. COBOL field references have the form

field_name `OF` record_name_1 `OF` ... `OF` record_name_n

COBOL record example above can be referenced with

```
MIDDLE OF EMPLOYEE-NAME OF EMPLOYEE-RECORD
```

Most of the other languages use **dot notation** for field references,

Others (dot notation) record_name_1.record_name_2. ....record_name_n.field_name

Ada record example:

```
Employee_Record.Employee_Name.Middle
```

Fully qualified references must include all record names

Elliptical references allow leaving out record names as long as the reference is unambiguous

Operations on Records

COBOL provides MOVE CORRESPONDING statement for moving records. It copies a field from the source record to the destination record only when the same field is defined in the target record.

For example

```
 01 A-REC
          02 NAME  PIC X(10)
          02 SALARY  PIC 9(5)
```

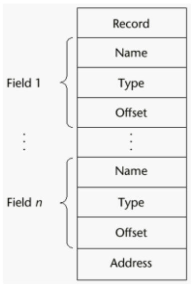
```
            02 STREET PIC X(15)
01 B-REC
            02 CITY  PIC X(10)
            02 STREET PIC X(15)
```

MOVE CORRESPONDING A-REC TO B-REC

STREET of the A-REC will be copied into B-REC

6.7.3 **Implementation of Record Type**

figure is compiler time descriptor for a record



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

Comparing records and arrays

1. Access to array elements is much slower than access to record fields, because subscripts are dynamic (field names are static)
2. Dynamic subscripts could be used with record field access, but it would disallow type checking and it would be much slower

6.10 **Union 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?

Discriminated vs. Free Unions

- 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

For example, consider the following C union:

```
union Data
{
        int  intEl;
        float  floatEl;
};

void main()
{
    union Data e;
        float x;
        e.intEl = 27;
        x = e.floatEl;
        printf("%f",x);
}
```

- This last assignment is not type checked, because the system cannot determine the current type of the current value of e. So output of x is 0.0000. generally we may think that the x value is 27.0000, but it is not

- Type checking of unions requires that each union construct include a type indicator. Such an indicator is called a **tag**, or **discriminant**, and a union with a discriminant is called a **discriminated union** and it include a tag to record the current type value. A free union is one without the tag to record the type value.

  - Use a hidden tag to maintain the current type
  - Tag is implicitly set by assignment
  - References are legal only in conformity clauses (see book example p. 231)
  - This runtime type selection is a safe method of accessing union objects

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;
```

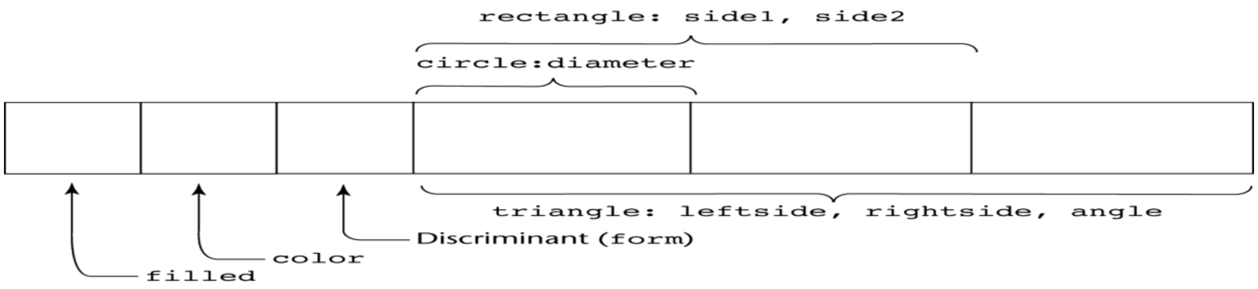


Fig : A discriminated union of three shape variables

```
Example

Figure_1 : Figure(Form => Triangle);
Figure_1 := (Filled => True,
Color => Blue,
Form => Rectangle,
Side_1 => 12,
Side_2 => 3);
```

Evaluation of Unions

- Free unions are unsafe
  - Do not allow type checking
- Java and C# do not support unions
- Ada's descriminated unions are safe

6.11 Pointer Types

- A *pointer* type variable has a range of values that consists of memory addresses and a special value NULL. The value NULL is not a valid address and is used to indicate that a pointer cannot currently be used to reference a memory cell
- 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*)
- 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?

Uses of pointers

1. Pointers used for indirect addressing (Provides indirect reference to variable)

2. Dynamic storage management

Pointer Operations

- Two fundamental operations: **assignment and dereferencing**

- Assignment is used to set a pointer variable's value to some useful address (Assignment of an address to a pointer)

- Dereferencing (*) gives the value stored at the location and it takes a reference through one level of indirection

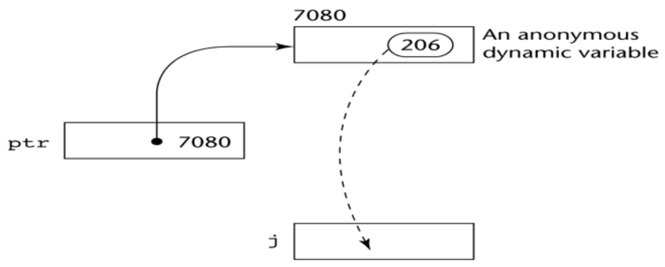
  - Dereferencing can be explicit or implicit
  - C++ uses an explicit Dereferencing via *
  - C++ uses & as referencing operator. It displays address of a variable
  - C++ uses * operator as Dereferencing operator. It displays value stored at address and

Example of dereferencing:

```
int *ptr = new int;

j = *ptr
```

sets j to the value located at ptr i.e. j = 206



Problems with Pointers

- **Dangling pointers (dangerous)**
  A pointer points to a heap-dynamic variable that has been de-allocated
- **Lost heap-dynamic variable**
  A lost heap-dynamic variable that is no longer referenced by any program pointer
  An allocated heap-dynamic variable that is no longer accessible to the user program (often called *garbage*)
  - Pointer p1 is set to point to a newly created heap-dynamic variable

- Pointer p1 is later set to point to another newly created heap-dynamic variable
- The process of losing heap-dynamic variables is called *memory leakage*
-

Ex of lost heap dynamic variable

```
int *p1;
p1 = new int;
p1 = new float
```
now p1 can not access previous heap dynamic memory. This problem is called memory leakage

C++ **example1**   illustrating dangling pointer

```
        int * p1;
        int * p2 = new int[100];
        p1 = p2;
        delete [ ] p2;
// Now, p1 is dangling, because the heap storage to which it was pointing has been deallocated.
```
Ex2:
```
        int * p1;
        int * p2 = new int
        p1 = p2;
        delete  p2;
// Now, p1 is dangling, because the heap storage to which it was pointing has been deallocated.
```

- ♦ Avoid dangling pointers by
  - ♦ Assign pointer to NULL after delete:
    ```
    delete p;
    p = NULL
    ```

Pointers in C and C++

- Used for dynamic storage management and addressing
- Explicit dereferencing is possible

```
int  *ptr;
 int count , init;
 …..
 ptr = &init;
 count = *ptr;
```

Assignment to count dereferences ptr to produce the value at init, which is then assigned to count
Two assignment stmts is to assign the value of init to count

Java - Only references no pointers in java

No pointer arithmetic
Can only point at objects (which are all on the heap)
No explicit deallocator (garbage collection is used) means there can be no dangling references
Dereferencing is always implicit

Pointers in Ada:

- Some dangling pointers are disallowed because dynamic objects can be automatically deallocated at the end of pointer's scope . dangling pointers are partially eliminated in this language.

- In Ada, all pointers are initialized to **null** . This prevents random access of memory

Pointer Arithmetic in C and C++

```
int list[100];
int *ptr;
pt r= list
which assigns address of list[0]  to  ptr

*(ptr + 1) is equivalent to  list[1]
*(ptr + index) is equivalent to  list[index]
  ptr[index] is equivalent to list[index]
```

this pointer in C++

- Each object maintains a pointer to itself which is called the “this” pointer.
- Every object in C++ has access to its own address through an important pointer called **this** pointer. The **this** pointer is an implicit parameter to all member functions. Therefore, inside a member function, this may be used to refer to the invoking object.

- Friend functions do not have a **this** pointer, because friends are not members of a class. Only member functions have a **this** pointer.

Ex:

```
Class Simple
{
     public:
          void showStuff() const;
     private:
        int stuff;
};
```

Two ways for member functions to access:

```
cout << stuff;
cout << this->stuff;
```

Evaluation of Pointers

- Pointers or references are necessary for dynamic data structures--so we can't design a language without them

Solutions to dangling pointer Problem

Tombstones: in which every heap-dynamic variable includes a special cell called tombstone that itself a pointer to the heap dynamic variable

- The actual pointer variable points only at tombstones never to heap dynamic variables
- When heap-dynamic variable de-allocated, tombstone remains but set to null ,indicating that heap dynamic variable no longer exists. This approach prevents a pointer from ever pointing to a deallocated variable
- Costly in time and space

Locks-and-keys: Pointer values are represented as (key, address) pairs where the key is an integer value .

- Heap-dynamic variables are represented as the storage for the variable plus a header cell for integer lock value
- When heap-dynamic variable allocated, lock value is created and placed in lock cell of the heap dynamic variable and key cell of the pointer that is specified in the call to new.
- Every access to the dereferenced pointer compares the key value of the pointer to the lock value in the heap-dynamic variable. If they match, the access is legal; otherwise the access is treated as a run-time error.

Heap management

The variables which are created dynamically are allocated storage from heap. It is very complex run-time process

Heap has Single-size cells and variable-size cells

Single-size cells

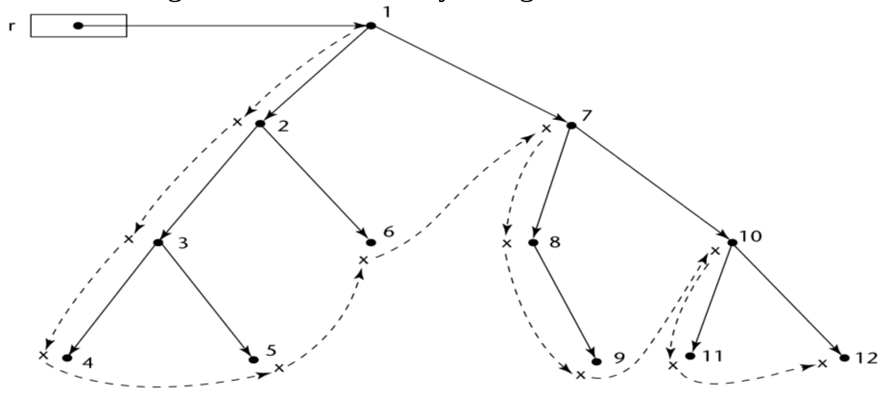
In a single size allocation heap, all available cells are linked together using the pointers in the cells, forming a list of available space. Allocation is a simple matter of taking the required no. of cells from the list when they are needed. Deallocation is much more complex process.

A heap dynamic variable can be pointed to by more than one pointer , making it difficult to determine when the variable is no longer useful to the program. Simply because one pointer is disconnected from a cell does not make it **garbage**; there could be several other pointers still pointing to the cell.

Two approaches to reclaim garbage

- **Reference counters** (*eager approach*): reclamation is gradual
- **Mark-sweep** (*lazy approach*): reclamation occurs when the list of variable space becomes empty

- Reference counters: maintain a counter in every cell that store the number of pointers currently pointing at the cell
  - *Disadvantages*: space required, execution time required, complications for cells connected circularly
  - *Advantage*: it is intrinsically incremental, so significant delays in the application execution are avoided
- The run-time system allocates storage cells as requested and disconnects pointers from cells as necessary; mark-sweep then begins
  - Every heap cell has an extra bit used by collection algorithm
  - All cells initially set to garbage
  - All pointers traced into heap, and reachable cells marked as not garbage
  - All garbage cells returned to list of available cells
  - Disadvantages: in its original form, it was done too infrequently. When done, it caused significant delays in application execution. Contemporary mark-sweep algorithms avoid this by doing it more often—called incremental mark-sweep



Dashed lines show the order of node_marking

Variable-Size Cells

- Heap with Variable-Size Cells have a ll the difficulties of single-size cells plus more
- Variable-Size Cells are required by most programming languages
- If mark-sweep is used, additional problems occur
  - The initial setting of the indicators of all cells in the heap is difficult
  - The marking process in nontrivial

– Maintaining the list of available space is another source of overhead

Reference Types

Referencing operator(&) is used to define reference variable and it is act as alias(alternative name) for the other value variable. the variable ref_result is declared as reference variable for variable result. We can use variable ref_result to access the value of result. Any changes made in one of the variable changes the content of both. The contents of both variables and addresses are always same.

Syntax: datatype & reference variable = variable name

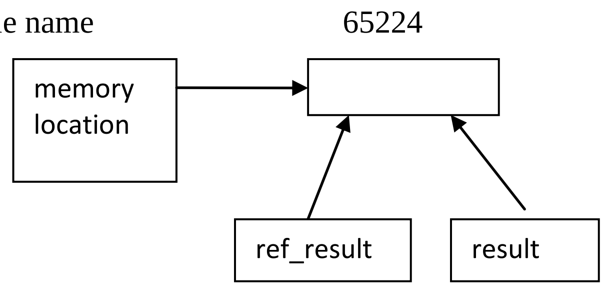


Fig: reference variable

Ex1:

```
int result = 0;
int  &ref_result  =  result;
cout<<ref_result;   // prints 0
ref_result = 100;
cout<<ref_result;   // prints 100
cout<<result;       // prints 100
```

Ex2:
```
int i;
int &j = i;
i= 2;
j = 3;
cout << i     // prints 3
```

Arithmetic Expressions

- Arithmetic evaluation was one of the motivations for the development of the first programming languages
- Arithmetic expressions consist of operators, operands, parentheses, and function calls

- **Design issues for arithmetic expressions**
  - Operator precedence rules?
  - Operator associativity rules?
  - Order of operand evaluation?
  - Operand evaluation side effects?
  - Operator overloading?
  - Type mixing in expressions?

Arithmetic Expressions: Operators

- A unary operator has one operand
- A binary operator has two operands
- A ternary operator has three operands

Arithmetic Expressions: Operator Precedence Rules

- The *operator precedence rules* for expression evaluation define the order in which "adjacent" operators of different precedence levels are evaluated
- Typical precedence levels
  - parentheses
  - unary operators
  - ** (if the language supports it)
  - *, /
  - +, -

operator precedence of various languages

	C-based languages	Ada	Ruby
Highest	postfix ++, --	**, abs	**
	prefix ++, -- , unary +, -	* , / , mod	unary +,-
	*, /, %	unary + , -	*, / , %
Lowest	binary + , -	binary + , -	binary + , -

The ** operator is exponentiation. The % operator of C exactly like the mod operator of Ada. The precedence rules of C++ and those of C are same. Java's precedence rules are those of C++. When operator with the same precedence occur in an expression , associativity rules affect it

Arithmetic Expressions: Operator Associativity Rule

- The *operator associativity rules* for expression evaluation define the order in which adjacent operators with the same precedence level are evaluated
- Typical associativity rules
  - Left to right, except **, which is right to left
  - Sometimes unary operators associate right to left (e.g., in FORTRAN)
- APL is different; all operators have equal precedence and all operators associate right to left
- Precedence and associativity rules can be overriden with parentheses
-

Associative rules of few common imperative languages

Language	**Associativity Rule**
Ruby	Left : *, / , + , -
	Right: **

C- Based Languages	Left: *, / ,%, binary + , binary - Right: ++ , - -, unary - , unary +
Ada	Left : all except * * nonassociative : * *

Ruby Expressions

- All arithmetic, relational, and assignment operators, as well as array indexing, shifts, and bit-wise logic operators, are implemented as methods

Arithmetic Expressions: Conditional Expressions

- Conditional Expressions
  - C-based languages (e.g., C, C++)
  - An example:
    average = (count == 0)? 0 : sum / count
  - Evaluates as if written like
    if (count == 0)
    average = 0
    else
    average = sum /count

Arithmetic Expressions: Operand Evaluation Order

1. Variables: fetch the value from memory
2. Constants: sometimes a fetch from memory; sometimes the constant is in the machine language instruction
3. Parenthesized expressions: evaluate all operands and operators first
4. The most interesting case is when an operand is a function call

Functional side effects: A side effect of a function is called a functional side effect, occurs when the function changes either one of its parameters or a global variable.(a global variable is declared outside the function but is accessible in the function)

Ex: a + fun (a)
If A is evaluated first, no effect on the result of the expression. If FUN is evaluated first there is an effect.

The following C program illustrates a function changes a global variable that appears in an expression (**functional side effects**)

```
int a = 5;
int fun1( )
{
a = 17;
return 3;
} /* end of fun1 */
```

```
void main( )
{
a = a + fun1( );
} /* end of main */
```

The value computed for a in main depends on the order of evaluation of the operands in the expression a + fun1( ). The value of a will be either 8 (if a is evaluated first) or 20 (if the function call is evaluated first).

Problem with functional side effects:

- When a function referenced in an expression alters another operand of the expression; e.g., for a parameter change:

There are two possible solutions to the problem of operand evaluation order and side effects.

First, the language designer could disallow function evaluation from affecting the value of expressions by simply disallowing functional side effects. (Write the language definition to disallow functional side effects

Second, the language definition could state that operands in expressions are to be evaluated in a particular order and demand that implementors guarantee that order. Java requires that operands appear to be evaluated in left-to-right order

Overloaded Operators

- Use of an operator for more than one purpose is called *operator overloading*
- Arithmetic operators are often used for more than one purpose. For example , + is used for addition of any numeric type operands. Some languages , Java for example ,also use + for string concatenation. There is too much operator overloading in APL and SNOBOL, where most operators are used for both unary and binary operations.
- As an example let us consider an operator ampersand (&) in C. As binary operator it denotes Bitwise logical AND operation . As unary operator it specifies the address of a variable
- Ex: x = &y causes address of y to be placed in x
- C++ and C# allow user-defined overloaded operators. Operator overloading was one of the C++ features that was not copied into Java. A few operators that cannot be overloaded in C++ are scope resolution operator(: :)

- Potential problems:
  - Users can define nonsense operations
  - Readability may suffer, even when the operators make sense

Type Conversions

- A *narrowing conversion* is one that converts an object to a type that cannot include all of the values of the original type e.g., float to int

- A *widening conversion* is one in which an object is converted to a type that can include at least approximations to all of the values of the original type
- e.g., int to float

Mixed Mode *expressions*

- A *mixed-mode expression* is one that has operands of different types

```
Ex:  int a;
      float b,c;
      c = a +b;
```

A *coercion* is an implicit type conversion will be performed in Mixed Mode *expressions*

- Disadvantage of coercions:
  - They decrease in the type error detection ability of the compiler
- In most languages, all numeric types are coerced in expressions, using widening conversions
- In Ada, there are virtually no coercions in expressions

Explicit Type Conversions

- Called *casting* in C-based languages
- Examples
  - C: (int)angle
  - Ada: Float (Sum)
- Note that Ada's syntax is similar to that of function calls

Errors in Expressions

- Inherent limitations of arithmetic   e.g., division by zero
- Limitations of computer arithmetic   e.g. overflow

Relational and Boolean Expressions

Relational Expressions

Relational Expressions use relational operators and operands of various types. Relational operator is an operator that compares the values of its two operands. Relational Expression has two operands and one relational operator. The value of relational expression is Boolean except when Boolean is not a type in the language. Operator symbols used vary somewhat among languages (!=, /=, ~=, .NE., <>, #). Relational operators always have lower precedence than arithmetic operators. JavaScript and PHP have two additional relational operator, === and !==

Ex: a+1 > 2 * b

In the above ex arithmetic expressions are evaluated first.

The syntax of relational operators in some common languages

Operation	Ada	C- Based Languages	Fortran 95
Equal	=	= =	.EQ. or = =
Not equal	/=	! =	.NE. or < >
Greater than	>	>	.GT. or >
Less than	<	<	.LT. or <
Greater than or equal	>=	>=	.GE. or >=
Less than or equal	<=	<=	.LE. or ,<=

Boolean Expressions

- Boolean Expressions consist of Boolean variables, Boolean constants, relational expressions and Boolean operators. The operators include those for the AND , OR , NOT operations.
  - Operands are Boolean and the result is Boolean

Boolean operators of various languages as follows

FORTRAN 77	**FORTRAN 90**	**C**	**Ada**
.AND.	and	&&	and
.OR.	or	||	or
.NOT.	not	!	not
			xor

Ex: (a>b) && (b<c)

No Boolean Type in C89

- C89 has no Boolean type--it uses int type with 0 for false and nonzero for true
- One odd characteristic of C's expressions: **a < b < c** is a legal expression, but the result is not what you might expect:
    - – Left operator is evaluated, producing 0 or 1
    - – The evaluation result is then compared with the third operand (i.e., **c**)

  C99 has Boolean type with a keyword **bool**

Short Circuit Evaluation

- Short circuit evaluation of an expression is one in which the result is determined without evaluating all of the operands and/or operators

- Example: **(13*a) * (b/13–1)**
  If a is zero, there is no need to evaluate **(b/13-1)**

  short circuit evaluation of boolean expression

  if((a >= 0) && (b < 10))

  when a<0 there is no need to evaluate b <10. The value of above boolean expression is independent of second relational expression if a < 0, because (FALSE & (b<10) is FALSE. Un like the case of arthmatic expressions , this short circuit can be easily discovered during execution.

- Problem with non-short-circuit evaluation
  index = 1;
  while (index <= length) && (LIST[index] != value)
  index++;
    - – When **index=length**, **LIST [index]** will cause an indexing problem (assuming **LIST** has **length -1** elements)

– C, C++, and Java use short-circuit evaluation for the usual Boolean operators (**&&** and **||**), but also provide bitwise Boolean operators that are not short circuit (**&** and **|**)
– Ada: programmer can specify either (short-circuit is specified with **and then** and **or else**)

Assignment Statements

- The general syntax
  <target_var> <assign_operator> <expression>
- The assignment operator

  = FORTRAN, BASIC, the C-based languages

:= ALGOLs, Pascal, Ada

= can be bad when it is overloaded for the relational operator for equality (that's why the C-based languages use == as the relational operator)

Conditionaltargets

```
($flag ? $total : $subtotal) = 0
  Which is equivalent to
  if ($flag){
          $total = 0
  } else {
          $subtotal = 0
  }
```

Compound assignment Operators

- A shorthand method of specifying a commonly needed form of assignment
- Introduced in ALGOL; adopted by C
- Example
  a = a + b
  is written as
  a += b

Unary Assignment Operators

Unary assignment operators in C-based languages combine increment and decrement operations with assignment

- Examples

sum = ++count (count incremented, added to sum)
sum = count++ (count incremented, added to sum)
count++ (count incremented)
-count++ (count incremented then negated)

Assignment as an Expression

- In C, C++, and Java, the assignment statement produces a result and can be used as operands
- An example:
  while ((ch = getchar())!= EOF){…}
  ch = getchar() is carried out; the result (assigned to ch) is used as a conditional value for the while statement

List Assignments

- Perl and Ruby support list assignments

e.g.,($first, $second, $third) = (20, 30, 40);

Mixed-Mode Assignment

- Assignment statements can also be mixed-mode
- In Fortran, C, and C++, any numeric type value can be assigned to any numeric type variable
- In Java, only widening assignment coercions are done
- Ada does not allow mixed-mode assignment.
- Fortran, C, C++, and Perl use coercion rules for mixed-mode assignment

- C++, Java and C# allow mixed-mode assignment only if the required coercion is widening.8 So, an **int** value can be assigned to a **float** variable, but not vice versa

For example

```
int  a , b;
float c;
….
…..
c = a / b;            // result of a/b coerced from int to float
```

In all languages that allow mixed-mode assignment, the coercion takes place only after the right side expressions has been evaluated.

Control Structures

- A *control structure* is a control statement and the statements whose execution it controls
- Design question
  - Should a control structure have multiple entries?

Selection Statements

A *selection statement* provides the means of choosing between two or more paths of execution

- Two general categories:
  - Two-way selectors
  - Multiple-way selectors
  -

Two-way selection

- General form:
  ```
  if control_expression
          then clause
          else clause
  ```
- Design Issues:
  - What is the form and type of the control expression?
  - How are the **then** and **else** clauses specified?
  - How should the meaning of nested selectors be specified?

Nesting Selectors

- Java example

```
 if (sum == 0)
    if (count == 0)
```

```
        result = 0;
        else result = 1;
```

- Which if gets the else?
- Java's static semantics rule: else matches with the nearest if
- To force an alternative semantics, compound statements may be used:

```
if (sum == 0) {
 if (count == 0)
        result = 0;
       }
else result = 1;
```

- The above solution is used in C, C++, and C#
- Perl requires that all then and else clauses to be compound

Multiple-Way Selection Statements

- Allow the selection of one of any number of statements or statement groups
- Design Issues:
    1. What is the form and type of the control expression?
    2. How are the selectable segments specified?
    3. Is execution flow through the structure restricted to include just a single selectable segment?
    4. How are case values specified?
    5. What is done about unrepresented expression values?

Examples:

- **C, C++, and Java switch statement**

```
switch (expression) {
        case const_expr_1: stmt_1;
        …
        case const_expr_n: stmt_n;
        [default: stmt_n+1]
}
```

- Design choices for C's **switch** statement
    1. Control expression can be only an integer type
    2. Selectable segments can be statement sequences, blocks, or compound statements
    3. Any number of segments can be executed in one execution of the construct (there is no implicit branch at the end of selectable segments)
    4. **default** clause is for unrepresented values (if there is no **default**, the whole statement does nothing)

case statement inAda

```
case expression is
        when choice list => stmt_sequence;
        …
        when choice list => stmt_sequence;
        when others => stmt_sequence;]
    end case;
```

- More reliable than C's switch (once a stmt_sequence execution is completed, control is passed to the first statement after the case statement

Ada case design choices:

1. Expression can be any ordinal type
2. Segments can be single or compound
3. Only one segment can be executed per execution of the construct

Multiple Selectors can appear as direct extensions to two-way selectors, using else-if clauses, for example in Python:

```
if count < 10 :
  bag1 = True
elif count < 100 :
  bag2 = True
elif count < 1000 :
  bag3 = True
```

The Python example can be written as a Ruby case

```
 case
  when count < 10 then bag1 = true
  when count < 100 then bag2 = true
  when count < 1000 then bag3 = true
end
```

Iterative Statements

- The repeated execution of a statement or compound statement is accomplished either by iteration or recursion

- General **design issues for iteration control statements**:
    1. How is iteration controlled?
    2. Where is the control mechanism in the loop?

\

Iterative Statements : Counter-Controlled Loops

1. Do loop of FORTRAN
2. for stmt of Ada ad C language

A counting iterative statement has a loop variable, and a means of specifying the *initial* and *terminal*, and *stepsize* values

- Design Issues:
    1. What are the type and scope of the loop variable?
    2. Should it be legal for the loop variable or loop parameters to be changed in the loop body, and if so, does the change affect loop control?
    3. Should the loop parameters be evaluated only once, or once for every iteration?

- FORTRAN 95 DO LOOP

```
        DO label var = start, finish [, stepsize]
 Ex:    DO  10  index  =  1 , 10
        C(I)  =  A(I)  +  B(I)

     10  CONTINUE
```

- stepsize can be any value but not zero
- Parameters can be expressions
- Design choices:
    1. Loop variable must be **INTEGER**
    2. The loop variable cannot be changed in the loop, but the parameters can; because they are evaluated only once, it does not affect loop control
    3. Loop parameters are evaluated only once

- FORTRAN 95 : a second form:

```
[name:] Do variable = initial, terminal [,stepsize]
        …
End Do [name]
```

- Cannot branch into either of Fortran's Do statements

for statement in Ada

```
for var in [reverse] discrete_range loop          ...
        end loop
```

Design choices:
- Type of the loop variable is that of the discrete range (A discrete range is a sub-range of an integer or enumeration type).
- Loop variable does not exist outside the loop
- The loop variable cannot be changed in the loop, but the discrete range can;
- The discrete range is evaluated just once.

A discrete range is a subrange of an integer or enumeration type, such as 1..10 or monday..Friday. The **reverse** reserved word, when present, indicates that the values of the discrete range are assigned to the loop variable in reverse order.
The most interesting new feature of the Ada **for** statement is the scope of the loop variable, which is the range of the loop. The variable is implicitly declared at the **for** statement and implicitly undeclared after loop termination

For example,

Count : Float := 1.35;
for Count **in** 1..10 **loop**
Sum := Sum + Count;
end loop;

the Float variable Count is unaffected by the **for** loop. Upon loop termination, the variable Count is still Float type with the value of 1.35. Also, the Float-type variable Count is hidden from the code in the body of the loop, being masked by the loop counter Count, which is implicitly declared to be the type of the discrete range, Integer.

Iterative Statements of C-based languages

for ([expr_1] ; [expr_2] ; [expr_3]) statement

- The expressions can be whole statements, or even statement sequences, with the statements separated by commas
  – The value of a multiple-statement expression is the value of the last statement in the expression
  – If the second expression is absent, it is an infinite loop
- Design choices:

- There is no explicit loop variable
- Everything can be changed in the loop
- The first expression is evaluated once, but the other two are evaluated with each iteration
- The value of a multiple-statement expression is the value of the last statement in the expression e.g.,`for (i = 0, j = 10; j == i;  i++) ...`

The operational semantics of for statement is follows
expression_1
loop:
if expression_2 = 0 **goto** out
[loop body]
expression_3
goto loop
out: . . .

Following is an example of a skeletal C **for** statement:

for (count = 1; count <= 10; count++)
{
. .
.}

C++ differs from C in two ways:

1. The control expression can also be Boolean

2. The initial expression can include variable definitions (scope is from the definition to the end of the loop body)

Java and C#

Differs from C++ in that the control expression must be Boolean

2. Scope of variables defined in the initial expression is only the loop body

Python for statement:

```
for loop_variable in object:
    - loop body
```

```
for count in [2, 4, 6]:
print count
```

produces

```
2
4
6
```

The object is often a range, which is either a list of values in brackets ([2, 4, 6]), or a call to the range function (range(5), which returns 0, 1, 2, 3, 4

- The loop variable takes on the values specified in the given range, one for each iteration
- The else clause, which is optional, is executed if the loop terminates normally

for statement of python including range

```
for count in range (5, 11, 2):
print count
```

produces

```
5
7
9
```

Perl has a built-in iterator for arrays and hashes e.g.,

```
@names = ( "anu","rony","alex")
foreach $name (@names) { print $name }
```

Iterative Statements: Logically-Controlled Loops

- Repetition control is based on a Boolean expression
- Design issues:
  - Pretest or posttest?
  - Should the logically controlled loop be a special case of the counting loop statement or a separate statement?

Examples:

- C and C++ have both pretest and posttest forms, in which the control expression can be arithmetic:

```
        while (ctrl_expr)                    do
                loop body                      loop body
                                             while (ctrl_expr)
```

- Java is like C and C++, except the control expression must be Boolean (and the body can only be entered at the beginning -- Java has no **goto**
- Ada has a pretest version, but no posttest
- FORTRAN 95 has neither
- Perl and Ruby have two pretest logical loops, while and until. Perl also has two posttest loops

Iterative Statements: User-Located Loop Control Mechanisms

- Sometimes it is convenient for the programmers to decide a location for loop control (other than top or bottom of the loop)
- Simple design for single loops (e.g., break)
- Design issues for nested loops
    1. Should the conditional be part of the exit?
    2. Should control be transferable out of more than one loop?

User-Located Loop Control Mechanisms : **break and continue**

- C , C++, Python, Ruby, and C# have unconditional unlabeled exits (break)
- Java and Perl have unconditional labeled exits (break in Java, last in Perl)
- C, C++, and Python have an unlabeled control statement, continue, that skips the remainder of the current iteration, but does not exit the loop
- Java and Perl have labeled versions of continue

Example of continue statement

```
int main( )
{
   for( int i = 1; i<=10;i++)
  {   if(x%2==0)
 continue;
cout<<x;
}}    o/p      1 3 5 7 9
```

Example for break statement

```
int main( )
{
    int x =1;
    while(1)
    {
            cout<< x;
            if( x ==8)
            break;
            x++;
     }
return 0;
}
```

output:

```
1 2 3 4 5 6 7 8
```

Iterative Statements: Iteration Based on Data Structures

- Number of elements of in a data structure control loop iteration
- Control mechanism is a call to an *iterator* function that returns the next element in some chosen order, if there is one; else loop is terminate
- C's for can be used to build a user-defined iterator:

```
for (p=root; p==NULL; traverse(p)){
}
```

C#’s foreach statement iterates on the elements of arrays and
other collections:

```
Strings[ ]  strList = {"Bob", "Carol", "Ted"};
foreach (Strings name in strList)
        Console.WriteLine ("Name: {0}", name);
```

- The notation {0} indicates the position in the string to be displayed

Unconditional Branching

- Transfers execution control to a specified place in the program (example goto statement in C language)
- Major concern: Readability
- Some languages do not support goto statement (e.g., Java)
- C# offers goto statement (can be used in switch statements)

  An **unconditional branch statement** transfers execution control to a specified location in the program. The most heated debate in language design in the late 1960s was over the issue of whether unconditional branching should be part of any high-level language, and if so, whether its use should be restricted. The unconditional branch, or goto, is the most powerful statement for controlling the flow of execution of a program’s statements. However, using the goto carelessly can lead to serious problems. The goto has stunning power and great flexibility (all other control structures can be built with goto and a selector), but it is this power that makes its use dangerous. Without restrictions on use, imposed by either language design or programming standards, goto statements can make programs very difficult to read, and as a result, highly unreliable and costly to maintain.

Guarded Commands (**Dijkstra's selection and loop statements)**

- Designed by Dijkstra
- Purpose: to support a new programming methodology that supported verification (correctness) during development
- Guarded Commands s for concurrent programming in Ada
- Basic Idea: if the order of evaluation is not important, the program should not specify one

Selection Guarded Command

```
if <Boolean exp> -> <statement>
[ ] <Boolean exp> -> <statement>
 ...
[ ] <Boolean exp> -> <statement>
fi
```

- semantics: when construct is reached,
  - Evaluate all Boolean expressions
  - If more than one are true, choose one non-deterministically
  - If none are true, it is a runtime error

- The closing reserved word, **`fi`**, is the opening reserved word spelled backward.This form of closing reserved word is taken from ALGOL 68. The small blocks, called *fatbars*,are used to separate the guarded clauses and allow the clauses to be statement sequences. Each line in the selection statement, consisting of a Boolean expression (a guard) and a statement or statement sequence ,is called a **guarded command**.

```
Ex : if    i= 0 -> sum := sum+i
      [ ]   i > j -> sum := sum+j
      [ ]   j > i  -> sum := sum+i
       fi
```

If `i = 0` and `j > i`, this statement chooses nondeterministically between the first and third assignment tatements. If `i` is equal to `j` and is not zero, a runtime error occurs because none of the conditions is true.

```
Ex:  if  x>=y  - > max:=x;
      [ ]  y>=x  -> max := y
      fi
```

Loop Guarded Command

```
do <Boolean> -> <statement>
[ ] <Boolean> -> <statement>
 ...
[ ] <Boolean> -> <statement>
od
```

- Semantics: for each iteration
  - Evaluate all Boolean expressions
  - If more than one are true, choose one non-deterministically; then start loop again
  - When all expressions are simultaneously false, the loop terminates.
  - If none are true, exit loop

```
do   i= 0 -> sum := sum+i
[ ]  i > j -> sum := sum+j
[ ]  j > i  -> sum := sum+i
od
```

Consider the following problem: Given four integer variables, `q1`, `q2`, `q3`, and `q4`, rearrange the values of the four so that `q1<= q2<= q3 <= q4`. Without guarded commands, one straightforward solution is to put the four values into an array, sort the array, and then assign the values from the array back into the scalar variables `q1`, `q2`, `q3`, and `q4`. While this solution is not difficult, it requires a good deal of code, especially if the sort process must be included. Now, consider the following code, which uses guarded commands to solve the same problem but in a more concise and elegant way.

```
do q1 > q2 -> temp := q1; q1 := q2; q2 := temp;
[] q2 > q3 -> temp := q2; q2 := q3; q3 := temp;
[] q3 > q4 -> temp := q3; q3 := q4; q4 := temp;
od
```

Type Checking

- *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
- 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

  Name Type Equivalence

- *Name type equivalence* means the two variables have equivalent types if they are in either the same declaration or in declarations that use the same type name
- Easy to implement but highly restrictive:
  - Subranges of integer types are not equivalent with integer types
  - Formal parameters must be the same type as their corresponding actual parameters

Structure Type Equivalence

- *Structure type equivalence* means that two variables have equivalent types if their types have identical structures
- More flexible, but harder to implement

Example

type vector = array [1..10] of real
type weight = array [1..10] of real
var x, y: vector; z: weight

Name type Equivalence: When they have the same name.

x, y have the same type; z has a different type. so, x and y are name type quivalent

Structural type Equivalence: When they have the same structure.

x, y, z have the same structure

Dangling Else

The **dangling else** is a problem in computer programming in which an optional else clause in an if–then(–else) statement results in nested conditionals being ambiguous. Formally, the reference context-free grammar of the language is ambiguous, meaning there is more than one correct parse tree.

In many programming languages one may write conditionally executed code in two forms: the if-then form, and the if-then-else form – the else clause is optional:

```
if a then s
if b then s1 else s2
```

This gives rise to an ambiguity in interpretation when there are nested statements, specifically whenever an if-then form appears as `s1` in an if-then-else form:

```
if a then if b then s else s2
```

In this example, `s` is unambiguously executed when `a` is true and `b` is true, but one may interpret `s2` as being executed when `a` is false (thus attaching the else to the first if) or when `a` is true and `b` is false (thus attaching the else to the second if). In other words, one may see the previous statement as either of the following expressions:

```
if a then (if b then s) else s2
      or
if a then (if b then s else s2)
```