UNIT-II

DATA, DATA TYPES, AND BASIC STATEMENTS

NAMES

- The design of one of the fundamental attributes of variables, names, must be covered.
- Names are also associated with subprograms, formal parameters, and other program constructs.
- The term identifier is often used interchangeably with name.
- A name is a string of characters used to identify some entity in a program.

VARIABLE

- A variable is an abstraction of a memory cell or collection of cells.
- The move from machine languages to assembly languages was largely one of replacing absolute numeric memory addresses for data with names, making programs far more readable and therefore easier to write and maintain.
- **Address:** The address of a variable is the machine memory address with which it is associated. (also called *l*-value). A variable may have different addresses at different times during execution. A variable may have different addresses at different places in a program. If two variable names can be used to access the same memory location, they are called aliases. Aliases are harmful to readability (program readers must remember all of them). Aliases can be created in programs in several different ways. One common way in C and C++ is with their union types.
- **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.
- **Value**: 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.

BINDING

- A **binding** is an association between an attribute and an entity, such as between a variable and its type or value, or between an operation and a symbol.
- The time at which a binding takes place is called binding time.
- For example, the asterisk symbol (*) is usually bound to the multiplication operation at language design time. A data type, such as int in C, is bound to a range of possible values at language implementation time.

- At compile time, a variable in a Java program is bound to a particular data type. A variable may be bound to a storage cell when the program is loaded into memory. That same binding does not happen until run time in some cases, as with variables declared in Java methods.
- A call to a library subprogram is bound to the subprogram code at link time.

  Example: 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.
- Possible binding times:
  - Language design time--e.g., bind operator symbols to operations.
  - Language implementation time--e.g., bind floating point type to a representation.
  - Compile time--e.g., bind a variable to a type in C or Java.
  - Load time--e.g., bind a FORTRAN 77 variable to a memory cell (or a C static variable)
  - Runtime--e.g., bind a nonstatic local variable to a memory cell.
- A binding is **static** if it first occurs before run time and remains unchanged throughout program execution.
- A binding is if it first occurs during execution or can change during execution of the program.

Type Bindings:

- 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).
- FORTRAN, PL/I, BASIC, and Perl provide implicit declarations

  Advantage: writability

  Disadvantage: reliability (less trouble with Perl)
- Dynamic Type Binding (JavaScript and PHP)

  Specified through an assignment statement e.g., JavaScript

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

  list = 17.3;
- **Type Inferencing:**(ML, Miranda, and Haskell) Rather than by assignment statement, types are determined from the context of the reference.

- **Storage Bindings & Lifetime:**

  Allocation - getting a cell from some pool of available cells.

  Deallocation - putting a cell back into the pool.

  The **lifetime** of a variable is the time during which it is bound to a particular memory cell.

Categories of variables by lifetimes:

- **Static**--bound to memory cells before execution begins and remains bound to the same memory cell throughout execution.

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

  Advantages: efficiency (direct addressing), history-sensitive subprogram support.

  Disadvantage: lack of flexibility (no recursion).

- **Stack-dynamic**--Storage bindings are created for variables when their declaration statements are elaborated. If scalar, all attributes except address are statically bound

  e.g. local variables in C subprograms and Java methods.

  Advantage: allows recursion; conserves storage.

  Disadvantages:
  - Overhead of allocation and deallocation.
  - Subprograms cannot be history sensitive.
  - Inefficient references (indirect addressing).

- **Explicit heap-dynamic**--Allocated and deallocated by explicit directives, specified by the programmer, which take effect during execution.

  e.g. dynamic objects in C++ (via new and delete)all objects in Java.

  Advantage: provides for dynamic storage management.

  Disadvantage: inefficient and unreliable.

- **Implicit heap-dynamic**--Allocation and deallocation caused by assignment statements.

  e.g. all variables in APL; all strings and arrays in Perl and JavaScript.

  Advantage: flexibility.

  Disadvantages:
  - Inefficient, because all attributes are dynamic.
  - Loss of error detection.

SCOPE

- The scope of a variable is the range of statements in which the variable is visible.
- A variableis visible in a statement if it can be referenced in that statement.
- The scope rules of a language determine how a particular occurrence of a name is associated with a variable, or in the case of a functional language, how a name is associated with an expression.
- 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.

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.
- There are two categories of static-scoped languages: those in which subprograms can be nested, which create nested static scopes, and those in which subprograms cannot be nested.
- Ada, JavaScript, Common LISP, Scheme, Fortran 2003+, F#, and Pythonallow nested subprograms, but the C-based languages do not.
- Enclosing static scopes (to a specific scope) are called its static ancestors; the nearest static ancestor is called a static parent.
- Variables can be hidden from a unit by having a "closer" variable with the same name.
- C++ and Ada allow access to these "hidden" variables
    - In Ada: unit.name
    - In C++: class_name::name
- A method of creating static scopes inside program units--from ALGOL 60

  Examples:

```
C and C++: for (...)
{
int index;
...
}
Ada: declare LCL : FLOAT;
begin
...
End
```

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.
- Static scope is a textual, or spatial, concept whereas lifetime is a temporal concept, they at least appear to be related in this case.
- Scope and lifetime are also unrelated when subprogram calls are involved.

  Consider the following C++ functions:

```
voidprintheader() {
 . . .
 } /* 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 Environments

- Def: 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.

Named Constants

- Def: A named constant is a variable that is bound to a value only when it is bound to storage.
- Advantages: readability and modifiability
- Used to parameterize programs
- The binding of values to named constants can be either static (called manifest constants) or dynamic
- Languages:
  Pascal: literals only.
  FORTRAN 90: constant-valued expressions.
  Ada, C++, and Java: expressions of any kind.

- **Variable Initialization**
  - Def: The binding of a variable to a value at the time it is bound to storage is called initialization.
  - Initialization is often done on the declaration statement

e.g., Java
int sum = 0;

Data Type

- A **data type** defines a collection of data values and a set of predefined operations on those values.
- Computer programs produce results by manipulating data.

Primitive Data Types

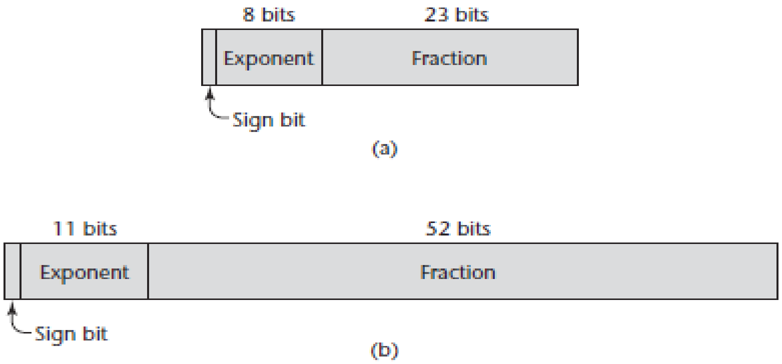
- Data types that are not defined in terms of other types are called primitive data types.
- Nearly all programming languages provide a set of primitive data types.
- Some of the primitive types are merely reflections of the hardware—for example, most integer types. Others require only a little non hardware support for their implementation.

Integer:

- Many computers now support several sizes of integers. These sizes of integers, and often a few others, are supported by some programming languages.
- 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.
- 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.
- Most computers now use a notation called twos complement to store negative integers, which is convenient for addition and subtraction.

Float:

- Model real numbers, but only as approximations.
- Languages for scientific use support at least two floating-point types (e.g., 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.
- 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.
- IEEE Floating-Point Standard 754.



IEEE floating-point formats: (a) single precision, (b) double precision

Complex:

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

Decimal:

- Most larger computers that are designed to support business systems applications have hardware support for decimal data types.
- Decimal data types store a fixed number of decimal digits, with the decimal point at a fixed position in the value. These are the primary data types for business data processing and are therefore essential to COBOL. C# and F# also have decimal data types.
- Store a fixed number of decimal digits, in coded form (BCD).
- Advantage: accuracy
- Disadvantages: limited range, wastes memory.

Boolean:

- 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 vdesigned since 1960.
- Could be implemented as bits, but often as bytes
- Advantage: readability.
- Boolean types are often used to represent switches or flags in programs.

Character Type:

- Character data are stored in computers as numeric codings.
- Most commonly used coding: ASCII
- An alternative, 16-bit coding: Unicode

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

Array Types

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

  Array Design Issues:

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

Array Indexing:

- Indexing (or subscripting) is a mapping from indices to elements array_name (index_value_list) an element.
- Index Syntax

  –FORTRAN, PL/I, Ada use parentheses

  •Ada explicitly uses parentheses to show uniformity between array references and function calls because both are mappings

  –Most other languages use brackets

Arrays Index (Subscript) Types:

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

Subscript Binding and Array Categories:

•**Static:** subscript ranges are statically bound and storage allocation is static (before run-time).

–Advantage: efficiency (no dynamic allocation)

•**Fixed stack-dynamic:** subscript ranges are statically bound, but the allocation is done at declaration time.

–Advantage: space efficiency

•**Stack-dynamic:** subscript ranges are dynamically bound and the storage allocation is dynamic (done at run-time)

–Advantage: flexibility (the size of an array need not be known until the array is to be used)

•**Fixed heap-dynamic:** similar to fixed stack-dynamic: storage binding is dynamic but fixed after allocation (i.e., binding is done when requested and storage is allocated from heap, not stack)

•**Heap-dynamic:** binding of subscript ranges and storage allocation is dynamic and can change any number of times

–Advantage: flexibility (arrays can grow or shrink during program execution).

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

Array Initialization:

•Some languages allow initialization at the time of storage allocation.

–C, C++, Java, C# example
```
int list [] = {4, 5, 7, 83}
```
–Character strings in C and C++
```
char name [ ] = “Freddie”;
```
–Arrays of strings in C and C++
```
char *names [ ] = {“Bob”, “Jake‖”, “Joe‖”};
```
–Java initialization of String objects
```
String[ ] names = {“Bob”, “Jake”, “Joe‖”};
```

Heterogeneous Arrays:

•A heterogeneous array is one in which the elements need not be of the same type

•Supported by Perl, Python, JavaScript, and Ruby.

Arrays Operations:

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

•Ada allows array assignment but also catenation.

•Python‘s array assignments, but they are only reference changes. Python also supports array catenation and element membership operations.

•Ruby also provides array catenation.

•Fortran provides elemental operations because they are between pairs of array elements

–For example, + operator between two arrays results in an array of the sums of the element pairs of the two arrays.

Rectangular and Jagged Arrays:

•A rectangular array is a multi-dimensioned array in which all of the rows have the same number of elements and all columns have the same number of elements.

•A jagged matrix has rows with varying number of elements.

  –Possible when multi-dimensioned arrays actually appear as arrays of arrays.

•C, C++, and Java support jagged arrays.

•Fortran, Ada, and C# support rectangular arrays (C# also supports jagged arrays).

Slices:

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

•Slices are only useful in languages that have array operations.

Slice Examples

Implementation of Arrays:

•Access function maps subscript expressions to an address in the array.

•Access function for single-dimensioned arrays:

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

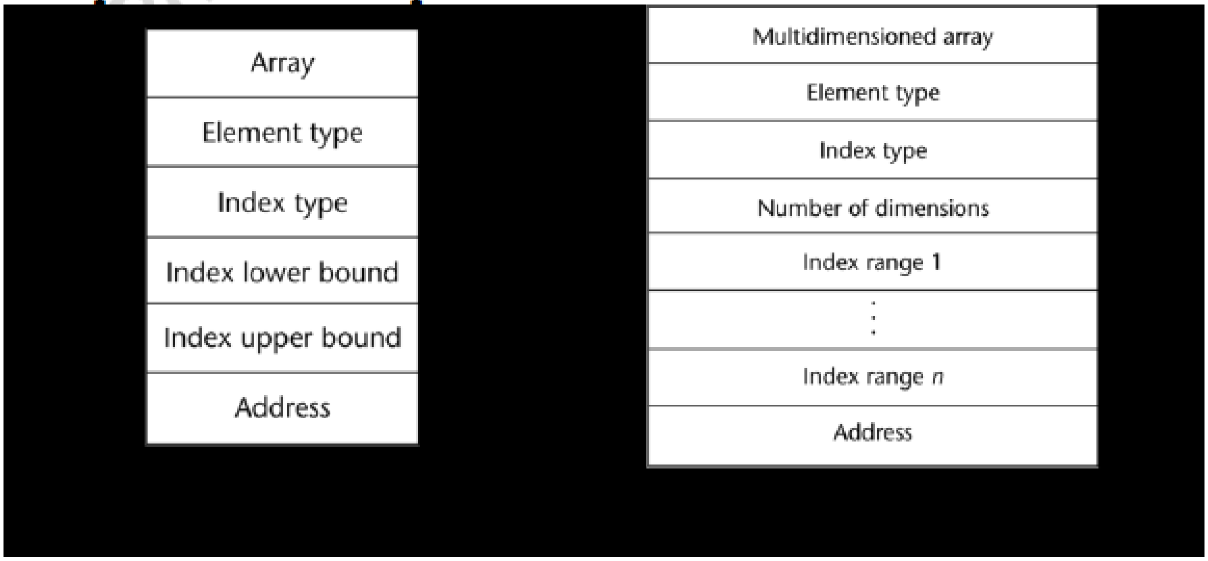
Accessing Multi-dimensioned Arrays:

•Two common ways:

–Row major order (by rows) – used in most languages.

–column major order (by columns) – used in Fortran.

Compile-Time Descriptors



Associative Arrays

- An associative array is an unordered collection of data elements that are indexed by an equal number of values called keys. User-defined keys must be stored

  •Design issues:

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

Associative Arrays in Perl:

•Names begin with %; literals are delimited by parentheses

```
%hi_temps = ("Mon" => 77, "Tue" => 79, ―Wed‖ =>65, …);
```

•Subscripting is done using braces and keys

```
$hi_temps{"Wed"} = 83;
```

–Elements can be removed with delete

```
delete $hi_temps{"Tue"};
```

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.

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.

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

References to Records:

•Record field references

1. COBOL

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

2. Others (dot notation)

record_name_1.record_name_2. ... record_name_n.field_name

•Fully qualified references must include all record names.

Operations on Records:

•Assignment is very common if the types are identical

•Ada allows record comparison

•Ada records can be initialized with aggregate literals

•COBOL provides MOVE CORRESPONDING

–Copies a field of the source record to the corresponding field in the target record

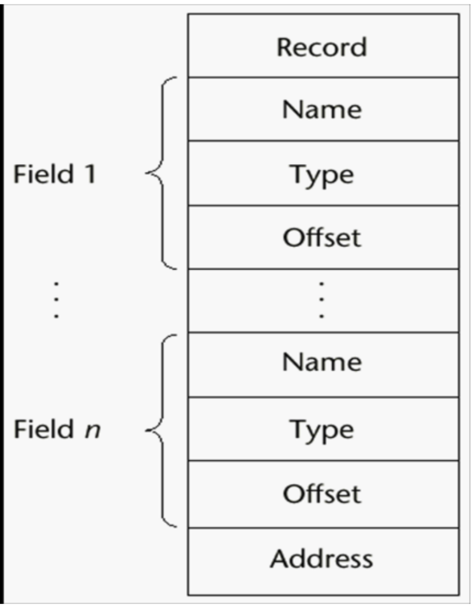
Evaluation and Comparison to Arrays

•Records are used when collection of data values is heterogeneous

•Access to array elements is much slower than access to record fields, because subscripts are dynamic (field names are static)

•Dynamic subscripts could be used with record field access, but it would disallow type checking and it would be much slower.

Implementation of Record Type:



Unions Types

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

•Design issues

–Should type checking be required?

–Should unions be embedded in records?

Discriminated vs. Free Unions:

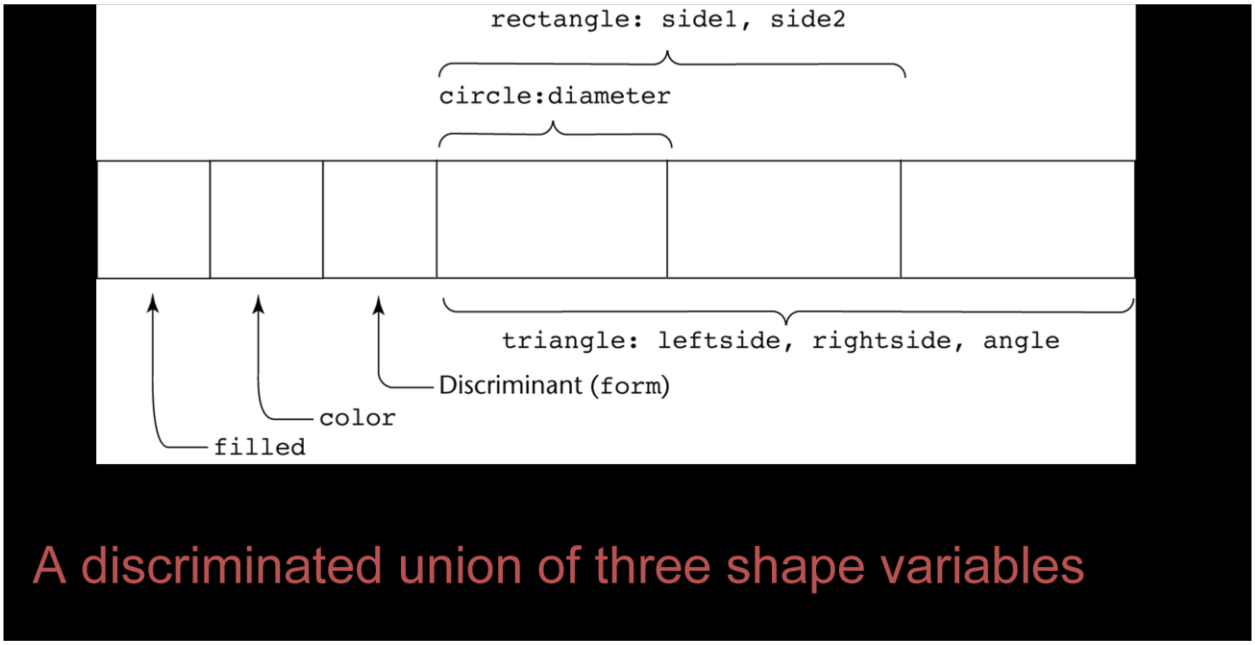
- Fortran, C, and C++ provide union constructs in which there is no language support for type checking; the union in these languages is called free union.
- Type checking of unions require that each union include a type indicator called a discriminant.

–Supported by Ada

Ada Union Types

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

Ada Union Type Illustrated



A discriminated union of three shape variables

Pointer and Reference Types

- A pointer type variable has a range of values that consists of memory addresses and a special value, nil.
- Provide the power of indirect addressing.
- Provide a way to manage dynamic memory
- A pointer can be used to access a location in the area where storage is dynamically created (usually called a heap).

Design Issues of Pointers

•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?

Pointer Operations:

- Two fundamental operations: assignment and dereferencing.
- *Assignment* is used to set a pointer variable‘s value to some useful address.
- *Dereferencing* yields the value stored at the location represented by the pointer‘s value

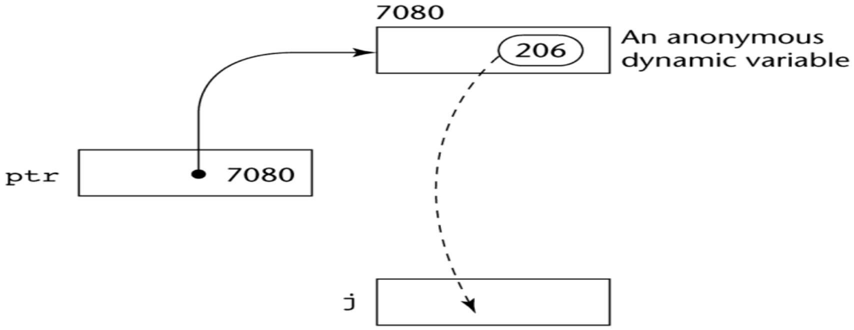
–Dereferencing can be explicit or implicit

–C++ uses an explicit operation via *

j = *ptr

sets j to the value located at ptr.

Pointer Assignment Illustrated



The assignment operation j = *ptr

Pointers in C and C++:

- Extremely flexible but must be used with care.
- Pointers can point at any variable regardless of when or where it was allocated.
- Used for dynamic storage management and addressing.
- Pointer arithmetic is possible.
- Explicit dereferencing and address-of operators.
- Domain type need not be fixed (void *)

void * can point to any type and can be type

checked (cannot be de-referenced)

Pointer Arithmetic in C and C++:

```
float stuff[100];
float *p;
p = stuff;
*(p+5) is equivalent to stuff[5] and p[5].
*(p+i) is equivalent to stuff[i] and p[i].
```

Reference Types

- C++ includes a special kind of pointer type called a reference type that is used primarily for formal parameters.
  –Advantages of both pass-by-reference and pass-by-value
- Java extends C++‘s reference variables and allows them to replace pointers entirely.
- References are references to objects, rather than being addresses.
- C# includes both the references of Java and the pointers of C++.

Heap Management:

- A very complex run-time process.
- Single-size cells vs. variable-size cells.
- 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 Counter

- 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.

Mark-Sweep

- 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.

Arithmetic Expressions

- ➢ Expressions are the fundamental means of specifying computations in a programming language.
- ➢ To understand expression evaluation, need to be familiar with the orders of operator and operand evaluation.
- ➢ 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:

–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)
– *, /
– +, -

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).

•Precedence and associativity rules can be overridden with parentheses

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
```

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 Type. Conversions: Mixed Mode - is one that has operands of different types.
- Type Conversions: Errors in Expressions
  •Causes:
  –Inherent limitations of arithmetic e.g., division by zero
  –Limitations of computer arithmetic e.g. overflow
  • Often ignored by the run-time system.

Relational and Boolean Expressions

Relational Expressions:

- A relational operator is an operator that compares the values of its two operands.
- A relational expression has two operands and one relational operator.
- The value of a relational expression is Boolean, except when Boolean is not atype included in the language.
- The operation that determines the truth or falsehoodof a relational expression depends on the operand types.
- Operator symbols used vary somewhat among languages (!=, /=, .NE., <>, #).
- JavaScript and PHP have two additional relational operator, === and !==
  - Similar to their cousins, == and !=, except that they do not coerce their operands.

Boolean Expressions:

- Boolean expressions consist of Boolean variables, Boolean constants, relational expressions, and Boolean operators.
- A short-circuit evaluation of an expression is one in which the result is determinedwithout evaluating all of the operands and/or operators.
- –Example operators

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

- One odd characteristic of C‘s expressions: a < b < c is a legal expression, but the result is not what you might expect.
- The evaluation result is then compared with the third operand (i.e., c)Short Circuit Evaluation.
- An expression 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).

Assignment Statements

- The assignment statement is one of the central constructs in imperative languages. It provides the mechanism by which the user can dynamically change the bindings of values to variables.
- The general syntax

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

  = FORTRAN, BASIC, the C-based languages

  := ALGOLs, Pascal, Ada

Assignment Statements: Conditional Targets:

- Conditional targets (Perl) ($flag ? $total : $subtotal) = 0

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

Assignment Statements: Compound 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

Assignment Statements: 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)

Mixed-Mode Assignment:

- Assignment statements can also be mixed-mode, for example

```
int a, b;
float c;
c = a / b;
```

- 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.
- In Ada, there is no assignment coercion.

Control Structure

- 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 Statements:

•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?

The Control Expression

- If the then reserved word or some other syntactic marker is not used to introduce the then clause, the control expression is placed in parentheses.
- In C89, C99, Python, and C++, the control expression can be arithmetic.
- In languages such as Ada, Java, Ruby, and C#, the control expression must be Boolean.

Clause Form

- In many contemporary languages, the then and else clauses can be single statements or compound statements.
- In Perl, all clauses must be delimited by braces (they must be compound).
- In Fortran 95, Ada, and Ruby, clauses are statement sequences.
- Python uses indentation to define clauses

```
if x > y :
x = y
print "case 1"
```

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.
- Statement sequences as clauses: **Ruby**

```
if sum == 0 then
if count == 0 then
result = 0
else
result = 1
end
end
```

- **Python**

```
if sum == 0 :
if count == 0 :
result = 0
else :
result = 1
```

Multiple-Way Selection Statements:

- Allow the selection of one of any number of statements or statement groups

  Design Issues:

  - What is the form and type of the control expression?
  - How are the selectable segments specified?
  - Is execution flow through the structure restricted to include just a single selectable segment?

- How are case values specified?
- What is done about unrepresented expression values?

Multiple-Way Selection: Examples

- C, C++, and Java

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

- ➢ Design choices for C's switch statement
  - Control expression can be only an integer type
  - Selectable segments can be statement sequences, blocks, or compound statements
  - Any number of segments can be executed in one execution of the construct (there is no implicit branch at the end of selectable segments)
  - **default** clause is for unrepresented values (if there is no **default**, the whole.

Multiple-Way Selection: Examples

- C#
  - Differs from C in that it has a static semantics rule that disallows the implicit execution of more than one segment
  - Each selectable segment must end with an unconditional branch (goto or break)
- Ada

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

- Ada 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
  4. Unrepresented values are not allowed
- Constant List Forms:
  1. A list of constants
  2. Can include:
     - Subranges
     - Boolean OR operators (|)

Multiple-Way Selection Using if:

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

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

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?

Counter-Controlled Loops:

- A counting iterative statement has a loop variable, and a means of specifying the initial and terminal, and step size values

•*Design Issues:*

•What are the type and scope of the loop variable?

•What is the value of the loop variable at loop termination?

Iterative Statements: Examples

•FORTRAN 95 syntax

```
DO label var = start, finish [, step size]
```

•Step size can be any value but zero

•Parameters can be expressions

•Design choices:

1. Loop variable must be INTEGER.
2. Loop variable always has its last value.
3. 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.
4. Loop parameters are evaluated only once.

•FORTRAN 95 : a second form:

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

- Cannot branch into either of Fortran‘s Do statements

•Ada

```
forvar in [reverse] discrete_range loop ...
```

```
    end loop
```

•C-based languages

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

Iterative Statements: Logically-Controlled Loops: 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

Iterative Statements: Logically-Controlled Loops: Examples

Guarded Commands

- Designed by Dijkstra.
- Purpose: to support a new programming methodology that supported verification (correctness) during development.
- Basis for two linguistic mechanisms for concurrent programming (in CSP and Ada).
- **Selection Guarded Command**

  Form:

```
        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

- **Loop Guarded Command**

  Form:

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

Guarded Commands: Rationale

•Connection between control statements and program verification is intimate

•Verification is impossible with goto statements

•Verification is possible with only selection and logical pretest loops

•Verification is relatively simple with only guarded commands.

Type Checking

It is the activity of ensuring that the operands of an operator are of compatible types. A **compatible** type is one that either is legal for the operator or is allowed under language rules to be implicitly converted by compiler-generated code (or the interpreter) to a legal type. This automatic conversion is called a **coercion**.

For example, if an **int** variable and a **float** variable are added in Java, the value of the **int** variable is coerced to **float** and a floating-point add is done.

A **type error** is the application of an operator to an operand of an inappropriate type.

If all bindings of variables to types are static in a language, then type checking can nearly always be done statically. Dynamic type binding requires type checking at run time, which is called **dynamic type checking**.

Garbage Collection in JAVA

Java Memory Management, with its built-in garbage collection, is one of the language's finest achievements. It allows developers to create new objects without worrying explicitly about memory allocation and deallocation, because the garbage collector automatically reclaims memory for reuse. This enables faster development with less boilerplate code, while eliminating memory leaks and other memory-related problems. At least in theory.

Ironically, Java garbage collection seems to work too well, creating and removing too many objects. Most memory-management issues are solved, but often at the cost of creating serious performance problems. Making garbage collection adaptable to all kinds of situations has led to a complex and hard-to-optimize system. In order to wrap your head around garbage collection, you need first to understand how memory management works in a Java Virtual Machine (JVM).

How Garbage Collection Really Works

Many people think garbage collection collects and discards dead objects. In reality, Java garbage collection is doing the opposite! Live objects are tracked and everything else designated garbage. As you'll see, this fundamental misunderstanding can lead to many performance problems.

Let's start with the heap, which is the area of memory used for dynamic allocation. In most configurations the operating system allocates the heap in advance to be managed by the JVM while the program is running. This has a couple of important ramifications:

Object creation is faster because global synchronization with the operating system is not needed for every single object. An allocation simply claims some portion of a memory array and moves the offset pointer forward (see Figure 2.1). The next allocation starts at this offset and claims the next portion of the array.

When an object is no longer used, the garbage collector reclaims the underlying memory and reuses it for future object allocation. This means there is no explicit deletion and no memory is given back to the operating system.

Garbage Collection - C

Garbage Collection (GC) is a mechanism that provides automatic memory reclamation for unused memory blocks. Programmers dynamically allocate memory, but when a block is no longer needed, they do not have to return it to the system explicitly with a free() call. The GC engine takes care of recognizing that a particular block of allocated memory (heap) is not used anymore and puts it back into the free memory area. GC was introduced by John McCarthy in 1958, as the memory management mechanism of the LISP language. Many different approaches to garbage collection exist, resulting in some families of algorithms that include reference counting, mark and sweep and copying GCs. Hybrid algorithms, as well as generational and conservative variants, complete the picture.

Important Questions

1.) Explain about Type Checking and Type Conversion in detail?
2.) Explain about primitive data types in detail?
3.) Define scope and Lifetime?
4.) Explain Categories of Arrays and Array Operations in detail?
5.) Explain the following in detail
    a.) Arithmetic Operations.
    b.) Relational and Boolean Expressions.
    c.) Assignment Statements.
    d.) Guarded Command.
6.) Explain Control Structures in detail?
7.) Explain binding in detail?
8.) Explain structure of an associative array?