Subject Name: **Principles of Programming Languages**

Subject Code: **CS-6002**

Semester: **6th**

Subject Notes

Department of Computer Science and Engineering

Subject Name: PPL **Subject Code:CS6002**

Unit-1

Language Evaluation Criteria

The following factors influences Language evaluation criteria

1) Readability 2) Simplicity 3) Orthogonality 4) Writability 5) Reliability 6) Cost

1. Readability

• One of the most important criteria for judging a programming language is the ease with which programs can be read and understood.

• Language constructs were designed more from the point of view of the computer than of computer users

2. Overall simplicity

• Language with too many features is more difficult to learn

• Feature multiplicity is bad. For example: In Java, increment can be performed if four ways as:

```
Count= count+1
Count+=1
Count++
++count
```

• Next problem is operator overloading, in which single operator symbol has more than one meaning.

3. Orthogonality

• A relatively small set of primitive constructs that can be combined in a relatively small number of ways

• Consistent set of rules for combining constructs (simplicity)

• Every possible combination is legal

• For example, pointers should be able to point to any type of variable or data structure

• Makes the language easy to learn and read

• Meaning is context independent

• Lack of orthogonality leads to exceptions to rules

• C is littered with special cases

Orthogonality is closely related to simplicity. The more orthogonal the design of a language, the fewer exceptions the language rules require. Fewer exceptions mean a higher degree of regularity in the design, which makes the language easier to learn, read, and understand.

– Useful control statements

– Ability to define data types and structures

– Syntax considerations

4. Writability - Writability is a measure of how easily a language can be used to create programs for a chosen problem domain and can be directly related to:

– Most readability factors also apply to writability

– Simplicity and orthogonality

– Control statements, data types and structures

– Support for abstraction

5. Reliability-

A program is said to be reliable if performs to its specifications under all conditions.

Type checking

• Type checking is simply testing for type errors in a given program, either by the compiler or during the program execution

- Because run time type checking is expensive, compile time type checking is more desirable
- Famous failure of space shuttle experiment due to int / float mix-up in parameter passing.

Exception handling

- Ability to intercept run-time errors
- Ability to use different names to reference the same memory
- A dangerous feature

6. Cost-

The cost is also an important language evaluation criteria. The total cost of a programming is a function of cost of training to programmers, cost of writing programs, cost of compiling programs, cost of executing programs, cost of maintaining programs.

Influence on Language design

1. **Computer Architecture**
2. **Programming Methodologies**

Computer Architecture Influence

– Data and programs stored in memory
– Memory is separate from CPU
– Instructions and data are piped from memory to CPU
– Basis for imperative languages
- Variables model memory cells
- Assignment statements model piping
- Iteration is efficient

Programming Methodologies Influences

- structured programming
  - top-down design and step-wise refinement
- Process-oriented to data-oriented
  - Data abstraction
- Object-oriented programming
  - Data abstraction + inheritance + polymorphism

Programming Domain

- **Scientific applications-**– Large number of floating point computations. The most common data structures are arrays and matrices; the most common control structures are counting loops and selections. The first language for scientific applications was FORTRAN, ALGOL 60 and most of its descendants. Examples of languages best suited: Mathematica and Maple.
- **Business applications-** Business languages are characterized by facilities for producing reports, precise ways of describing and storing decimal numbers and character data, and ability to specify decimal arithmetic operations. Its uses for decimal numbers and characters. COBOL is the first successful high-level language for those applications**.**
- **Artificial intelligence-** In AI Symbols rather than numbers are typically manipulated. Symbolic computation is more conveniently done with linked lists of data rather than arrays. This kind of programming sometimes requires more flexibility than other programming domains. The first AI language was LISP and is still most widely used
- **Systems programming-** The operating system and all of the programming support tools of a computer system are collectively known as systems software. It needs for efficiency because of continuous use and low-level features for interfaces to external devices

• **Special-purpose languages**
– RPG (Report Program Generator) : business reports
– APT (Automatically Programmed Tool) programmable machine tools
– GPSS (General Purpose Simulation System) : simulation

Language Categories-
The four categories usually recognized are imperative, object-oriented, functional, and logic. Each Language category has some special features.

Programming Paradigms
1. Imperative- imperative programming is a programming paradigm that uses statements that change a program's state.
In much the same way that the imperative mood in natural languages expresses commands, an imperative program consists of commands for the computer to perform. Imperative programming focuses on describing how a program operates.
Examples: C, Pascal
2. Object-oriented-Stands for "Object-Oriented Programming." OOP refers to a programming methodology based on objects, instead of just functions and procedures. These objects are organized into classes, which allow individual objects to be group together. Most modern programming languages including Java, C/C++, and PHP, are object-oriented languages, and many older programming languages now have object-oriented versions.
An "object" in an OOP language refers to a specific type, or "instance," of a class. Each object has a structure similar to other objects in the class, but can be assigned individual characteristics. An object can also call functions, or methods, specific to that object. For example, the source code of a video game may include a class that defines the structure of characters in the game. Individual characters may be defined as objects, which allow them to have different appearances, skills, and abilities. They may also perform different tasks in the game, which are run using each object's specific methods.
Examples: Java, C++
3. Functional-Functional programming languages are specially designed to handle symbolic computation and list processing applications. Functional programming is based on mathematical functions. Some of the popular functional programming languages include: Lisp, Python etc.
Examples: LISP, Scheme
Functional programming languages are categorized into two groups, i.e. −
• **Pure Functional Languages** − these types of functional languages support only the functional paradigms. For example − Haskell.
• **Impure Functional Languages** − these types of functional languages support the functional paradigms and imperative style programming. For example − LISP.

Functional Programming – Characteristics-
• Functional programming languages are designed on the concept of mathematical functions that use conditional expressions and recursion to perform computation.
• Functional programming supports higher-order functions and lazy evaluation features.
• Functional programming languages don't support flow Controls like loop statements and conditional statements like If-Else and Switch Statements. They directly use the functions and functional calls.
• Like OOP, functional programming languages support popular concepts such as Abstraction, Encapsulation, Inheritance, and Polymorphism.
4. Logic (declarative)- A logic program consists of a set of axioms and a goal statement. The rules of inference are applied to determine whether the axioms are sufficient to ensure the truth of the goal statement. The

execution of a logic program corresponds to the construction of a proof of the goal statement from the axioms.
Example: Prolog

Programming Language Implementation
We have three different types of implementation methods. They are

- Compilation -Programs are translated into machine language
- Pure Interpretation - Programs are interpreted by another program known as an interpreter
- Hybrid Implementation Systems - A compromise between compilers and pure interpreters

Compiler- The compilation process is a sequence of various phases. Each phase takes input from its previous stage, has its own representation of source program, and feeds its output to the next phase of the compiler.
The structure of compiler consists of two parts:
Lexical Analysis
The first phase of scanner works as a text scanner. This phase scans the source code as a stream of characters and converts it into meaningful lexemes.
Syntax Analysis
The next phase is called the syntax analysis or **parsing**. It takes the token produced by lexical analysis as input and generates a parse tree (or syntax tree). In this phase, token arrangements are checked against the source code grammar, i.e. the parser checks if the expression made by the tokens is syntactically correct.
Semantic Analysis
Semantic analysis checks whether the parse tree constructed follows the rules of language. For example, assignment of values is between compatible data types, and adding string to an integer. Also, the semantic analyzer keeps track of identifiers, their types and expressions; whether identifiers are declared before use or not etc. The semantic analyzer produces an annotated syntax tree as an output.
Intermediate Code Generation
After semantic analysis the compiler generates an intermediate code of the source code for the target machine. It represents a program for some abstract machine. It is in between the high-level language and the machine language. This intermediate code should be generated in such a way that it makes it easier to be translated into the target machine code.
Code Optimization
The next phase does code optimization of the intermediate code. Optimization can be assumed as something that removes unnecessary code lines, and arranges the sequence of statements in order to speed up the program execution without wasting resources (CPU, memory).

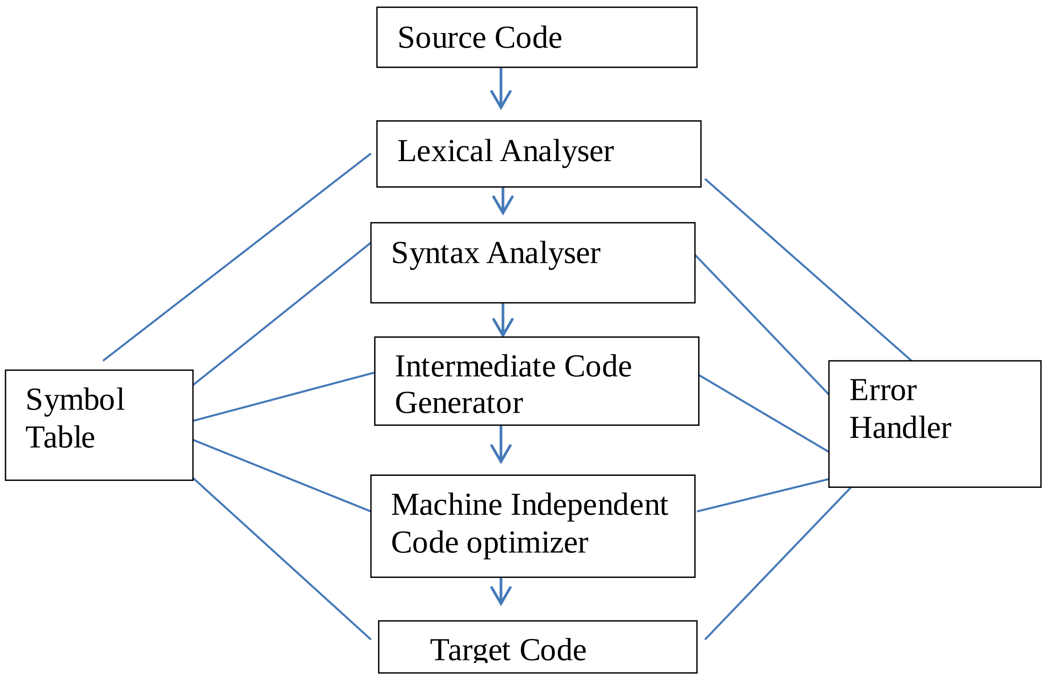


Figure 1.1 Phases of Compiler

The Compilation Process

Translate high-level program (source language) into machine code (machine language)

• Slow translation, but fast execution.

Translates whole source code into object code at a time and generate errors at the end. If no errors, generates object code. The object code after linking with libraries generates exe code. User input will be given with execution.

Ex: C++ is a compiler.

Pure Interpretation Process - Translation of source code line by line so that generates errors line by line. If no errors in the code generates object code. While interpretation itself user input will be given.

• Immediate feedback about errors

• Slower execution

• Often requires more space

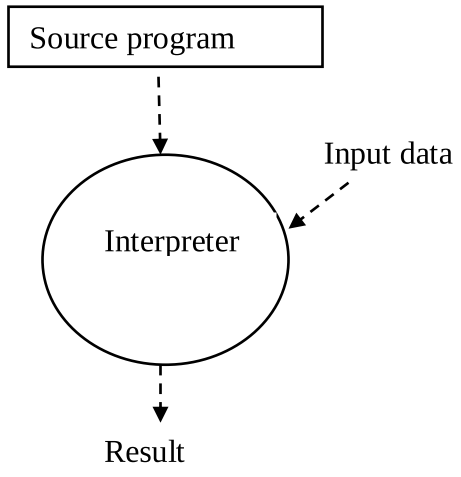


Figure 1.2 Pure Interpretation Process

• Used mainly for scripting languages

Example for interpreter is Dbase III plus

Hybrid Implementation Process

A compromise between compilers and pure interpreters

• A high-level language program is translated to an intermediate language that allows easy interpretation

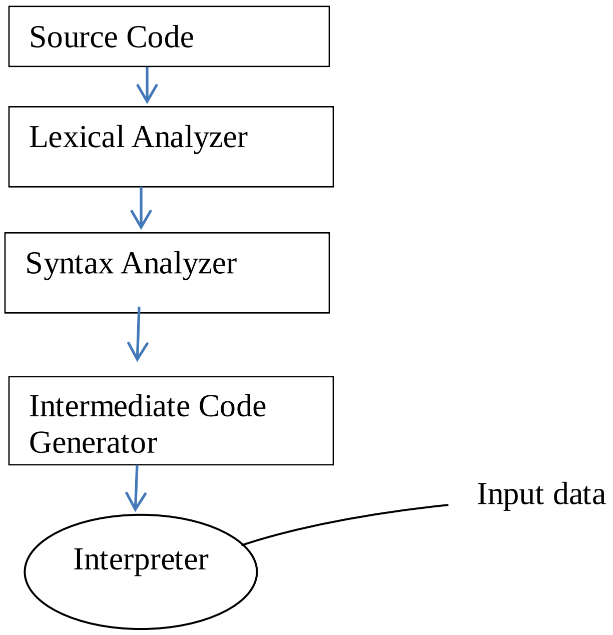


Figure 1.3 Hybrid Implementation Process

Example for hybrid implementation is Java. Java is a compiled interpreted language.

Examples - Perl programs are partially compiled to detect errors before interpretation

– Initial implementations of Java were hybrid; the intermediate form, byte code, provides portability to any machine that has a byte code interpreter and a run- time system (together, these are called Java Virtual Machine)

Just-in-Time Implementation Systems

•Initially translate programs to an intermediate language

• Then compile intermediate language into machine code

• Machine code version is kept for subsequent calls

• JIT systems are widely used for Java programs

• .NET languages are implemented with a JIT system

Preprocessors

• Pre-processor macros (instructions) are commonly used to specify that code from another file is to be included

• A pre-processor processes a program immediately before the program is compiled to expand embedded pre-processor macros

• A well-known example: C pre-processor

– expands #include, #define, and similar macros

Programming Environments-

Integrated Development Environments

An Integrated Development Environment (IDE) is a software application that provides comprehensive facilities to computer programmers for software development. An IDE normally consists of a source code editor, build automation tools and a debugger. Most modern IDEs have intelligent code completion. Some IDEs, such as NetBeans and Eclipse, contain a compiler, interpreter, or both. The boundary between an integrated development environment and other parts of the broader software development environment is not well-defined. Sometimes a version control system and various tools to simplify the construction of a Graphical User Interface (GUI) are integrated. Many modern IDEs also have a class browser, an object browser, and a class hierarchy diagram, for use in object-oriented software development.

Issues in Language Translation:

Language or Programming Language in computer science is an artificial language used to write a sequence of instructions (a computer program) that can be run by a computer. Similar to natural languages, such as English, programming languages have a vocabulary, grammar, and syntax. The languages used to program computers must have simple logical structures, and the rules for their grammar, spelling, and punctuation must be precise.

Programming Language Syntax- Syntax is defined as "the arrangement of words as elements in a sentence to show their relationship"; it also describes the sequence of symbols that make up valid programs. In other words syntax is the study of how words combine to make sentences. The order of words in sentences varies from language to language. Syntax provides significant information needed for understanding a program and provides much needed information toward the translation of the source program into an object program. There are some other attributes under the semantics that are not always determined in syntax rules such as the use of declarations, operations, sequence control and referencing environments. In other words syntax is a "solved problem".

General Syntactic criteria- The primary purpose of syntax is to provide notation for communication between the programmer and the programming languages processor. The choice of particular syntactic structures is constrained only slightly by the necessity to communicate particular information. General Syntactic criteria composed has general goals in making programs. These are readability (easy to read), Writability (easy to write), ease of verifiability, ease of translation (easy to translate), and lack of ambiguity (unambiguous). The concept of ease of verifiability or program correctness is related to readability and Writability.

Syntactic Elements of a Language

Character set. The choice of character set is one the first to be made in designing language syntax.

Identifiers- The basic syntax for identifiers—a string of letters and digits beginning with a letter—is widely accepted.

Operator symbols. – are special characters that most language used to represent the two basic arithmetic operations.

Keywords and reserved words- keyword is an identifier used as a fixed part of the syntax statement. It is also a reserved word if it may also be used as a programmer –chosen identifier.

***Noise* words-** These are optional words that are inserted in statements to improve readability. COBOL provides many options.

Comments- In relation to computers also called *remark*. Text embedded in a computer program for documentation purposes. Comments usually describe what the program does, who wrote it, why it was changed, and so on. Most programming languages have syntax for creating comments so that the comments will be ignored by the compiler or assembler.

Overall Program –Subprogram Structure- The overall syntactic organization of main program and subprogram definition is as varied as the other aspects of language syntax. Under of this are the following: separate subprogram definitions, separate data definitions, nested subprogram definitions, separate interface definitions, data descriptions separated from executable statements and unseparated subprogram definitions.

Stages in translation- The process of translation of a program from its original syntax into executable form is central in every programming language implementation. Translation may divide into two major parts: the analysis of the input source program and the synthesis of the executable object program.

Analysis of the Source Program- To a translator, the source program appears initially as one long undifferentiated sequence of symbols composed of thousands or tens of thousands characters. Under of this are the following: Lexical analysis (scanning), syntactic analysis (parsing), and Semantic analysis. There are four functions in semantic analysis. These as follows:

1. Symbol-table maintenance.
2. Insertion of implicit information.
3. Error detection
4. Macro processing and compile-time operations.

Macro is a set of keystrokes and instructions recorded and saved under a short key code. When the key code is typed, the program carries out the instructions of the macro. Program users create macros to save time by replacing often-used, sometimes lengthy, series of strokes with shorter versions. A compile-time operation is an operation to be performed during translation to control the translation of the source program.

Synthesis of the Object Program- The final stages of translation are concerned with the construction of the executable program from the outputs produced by the semantic analyser. It is composed of

Optimization- The semantic analyser ordinarily produces as output the executable translated program represented in some intermediate code.

Code generation- After the translated program in the internal representation has been optimized; it must be formed into the assembly language statements, machine code, or other object program form that is to be the output of the translation.

Linking and loading- In the optional final stage of translation, the pieces of code resulting from separate translations of subprograms are coalesced into the final executable program.

Formal Translation models- The formal definition of the syntax of a programming language is usually called *grammar*. Grammar is a branch of linguistics dealing with the form and structure of words (morphology), and their interrelation in sentences (syntax). The study of grammar reveals how language works.

Context Free Grammar- A context-free grammar (CFG) consisting of a finite set of grammar rules is a quadruple (N, T, P, S) where

- N is a set of non-terminal symbols.
- T is a set of terminals where N ∩ T = NULL.
- P is a set of rules, P: N → (N ∪ T)*, i.e., the left-hand side of the production rule P does have any right context or left context.
- S is the start symbol.

Example

The grammar ({A}, {a, b, c}, P, A), P : A → aA, A → abc.
The grammar ({S, a, b}, {a, b}, P, S), P: S → aSa, S → bSb, S → ε
The grammar ({S, F}, {0, 1}, P, S), P: S → 00S | 11F, F → 00F | ε

Bacus-Naur form is a notation which can be used to give inductive specifications for the syntactic elements of a language. The problem we face when we wish to define the rules of syntax (grammar) of a language is that we have to use notation (often involving the same alphabet as used in the language) to describe the rules for forming strings of symbols. We need to be able to distinguish symbols appearing in the syntax rule which are part of the rule itself from symbols which are part of a properly formed string of the language. Bacus and Naur developed a notational scheme, called BNF, for such syntax descriptions.

BNF can be used to inductively define a number of sets of syntactic elements of a language at once. These sets are called syntactic categories, or sometimes nonterminals and we write the names of these sets by enclosing them in "<" ">" . For example, the list_of_numbers J data type would be a nonterminal and would be written as <list_of_numbers>. Each syntactic category is defined by a finite set of rules, or productions. Each rule asserts that certain values must be in the syntactic category.

Example 1

Here is the BNF definition of the list-of-numbers syntax. This description has two rules.

<list_of_numbers> ::= ''
<list_of_numbers> ::= <number> , <list_of_numbers>
The first rule says that the empty list is in <list_of_numbers> and the second rule says that if n is in <number> and l is in <list_of_numbers>, the n , l which is n append l is in <list_of_numbers>. Each BNF rule starts with a syntactic category followed by ::= which is read as "is". The right hand side of the rule or production specifies how a member of the syntactic category is to be constructed from other syntactic categories and terminal symbols.
In the above rules the comma "," (the J word for append) and single quote "'" are terminal symbols and <number> and <list_of_numbers> are nonterminals. Sometimes we use the symbol "|" to shorten or simplify rules. "|" is read as or. The above two rules can be combined as:
<list_of_numbers> ::= '' | <number> , <list_of_numbers>
Shorthand is the Kleene star {...}* which means zero or more repetitions of whatever is between the braces. And the Kleene plus {...}+ which means one or more repetitions of whatever is between the braces. For example, the <list_of_numbers> category might be defined as:
<list_of_numbers> ::= {<number>}*
The shorthand items "|", Kleene star and plus are not really necessary, but are used with BNF to help shorten the descriptions.
As an example, consider the following specifications of Scheme data:
Example 2
<list> ::= ({<datum>}*)
<dotted-datum> ::= ({<datum>}+ . <datum>)
<vector> ::= #({<datum>}*)
<datum> ::= <number> | <symbol> | <boolean> | <string> | <list> | <dotted-datum> | <vector>
This means that
(1 abc 2 10) is a valid Scheme datum.
We are discussing how one formulates the syntax rules of a programming language. The properties of a program which can be determined by analyzing just the text of a program are said to be static properties. Similarly, the dynamic properties of a program are those which are determined by run-time inputs. Analysis of static properties is important because a translator can use this information to catch certain kinds of program errors and perhaps write a more efficient program.
Parse Tree
Parse tree is a hierarchical structure which represents the derivation of the grammar to yield input strings.
Root node of parse tree has the start symbol of the given grammar from where the derivation proceeds.
• Leaves of parse tree represent terminals.
• Each interior node represents productions of grammar.
• If A -> xyz is a production, then the parse tree will have A as interior node whose children are x, y and z from its left to right.

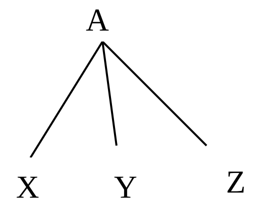


Figure 1.4 Parse Tree

Construct parse tree for E --> E + E I E * E I id

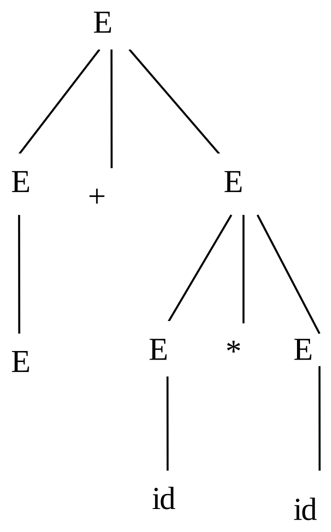


Figure 1.5 Example of Parse Tree

Example: Given the following grammar, find a parse tree for the string 1 + 2 * 3:

<E> --> <D>
<E> --> (<E>)
<E> --> <E> + <E>
<E> --> <E> - <E>
<E> --> <E> * <E>
<E> --> <E> / <E>
<D> --> 0 | 1 | 2 | ... 9

The parse tree is:

```
  E --> E --> N --> 1
        +
        E --> E --> N --> 2
              *
              E --> N --> 3
```

Extended BNF (EBNF) notation

Extended Backus-Naur form (EBNF) is a collection of extensions to Backus-Naur form.

Not all of these are strictly a superset, as some change the rule-definition relation ::= to =, while others remove the angled brackets from non-terminals.

More important than the minor syntactic differences between the forms of EBNF are the additional operations it allows in expansions.

Option

In EBNF, square brackets around an expansion, [expansion], indicates that this expansion is optional.

For example, the rule:

<term> ::= ["-"] <factor>

allows factors to be negated.

Repetition

In EBNF, curly braces indicate that the expression may be repeated zero or more times.

For example, the rule:

<args> ::= <arg> { "," <arg> }

defines a conventional comma-separated argument list.

Grouping

To indicate precedence, EBNF grammars may use parentheses, (), to explictly define the order of expansion.

For example, the rule:

```
<expr> ::= <term> ("+" | "-") <expr>
```

defines an expression form that allows both addition and subtraction.

Concatenation

In some forms of EBNF, the , operator explicitly denotes concatenation, rather than relying on juxtaposition.

Subject Name: **Advanced Computer Architecture**

Subject Code: **CS-6001**

Semester: **6th**

Subject Notes

Department of Computer Science and Engineering

Subject Name: PPL **Subject Code:CS6002**

Unit-2

Data type:

A data type defines a collection of data values and a set of predefined operations on those values.

Design issues for all data types

i. What operations are defined and how are they specified.

ii. It is convenient, both logically and concurently, to think of variables in terms of **descriptors**.

Descriptor:

A descriptor is the collection of the attributes of a variable

i. A descriptor is used for type checking, allocation and, de-allocation.

ii. Static attributes need only be available at compile-time; dynamic attributes need to be available at run-time.

Primitive data types

Primitive data types are those that are not defined in terms of other data types

Common primitive types:

1. Numeric types

Early PLs had only numeric primitive types, and still play a central role among the collections of types supported by contemporary languages.

A. Integers

i. For example, C, Ada, java .. allows these: short integer, integer and long integer.

ii. An integer is represented by a string of bits, with the leftmost representing the sign bit.

B. Floating point numbers

i. Model real numbers but only as approximations.

ii. Languages for scientific use support at least two floating-point types; sometimes more.

iii. usually exactly like the hardware, but not always; some languages allow accuracy specs in code e.g. (Ada).

IEEE (The Institute of Electrical and Electronics Engineers) floating-point formats: (a) Single precision, (b) Double precision

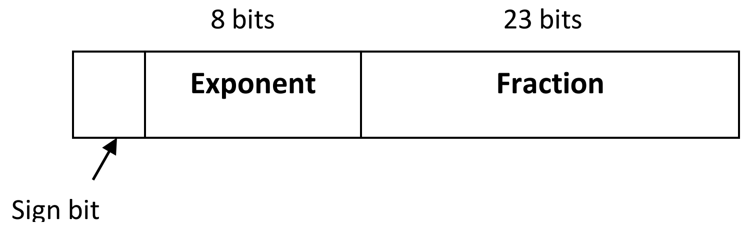


Figure 2.1 (a) Single Precision floating point format

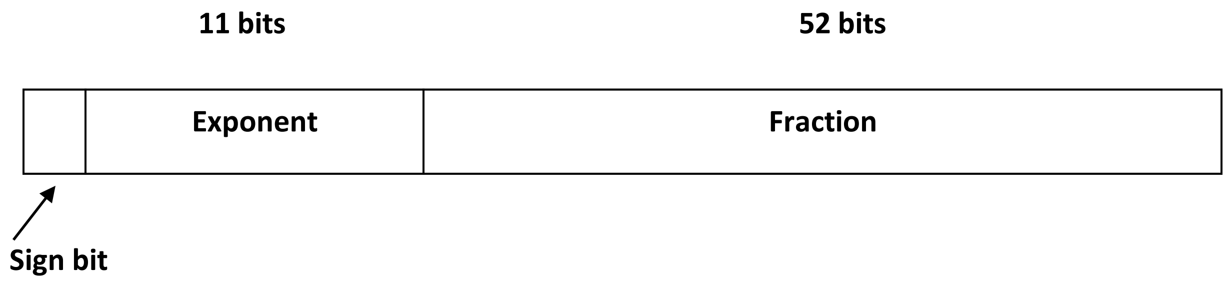


Figure 2.1 (b) Double Precision floating point format

C. Decimal

i. for business applications.
ii. store a fixed number of decimal digits (coded)

Advantage: accuracy

Disadvantages: limited range, wastes memory

D. Boolean types

i. The range of values has only two elements TRUE or FALSE.
ii. Booleans types are often used to represent switches or flags in programs

Character- Character is a symbol in programming language that has meaning. A character can be any letter, number, punctuation marks, symbols or whitespace. For example, the word "character" consists of eight characters and the phrase "Hello World!" consists of 12 characters including the whitespace and exclamation mark. In programming character is a datatype. We can declare a variable as of a character type and store characters in the variable. For example, in C and Java, we write,

Characters when storing in a variable must be inserted between single quotes. String is another datatype in programming, which is a modified version of character datatype. Strings are used to store more than one character

Character String Types

- Character string type is one in which the values consist of sequences of characters

Design issues with the string types

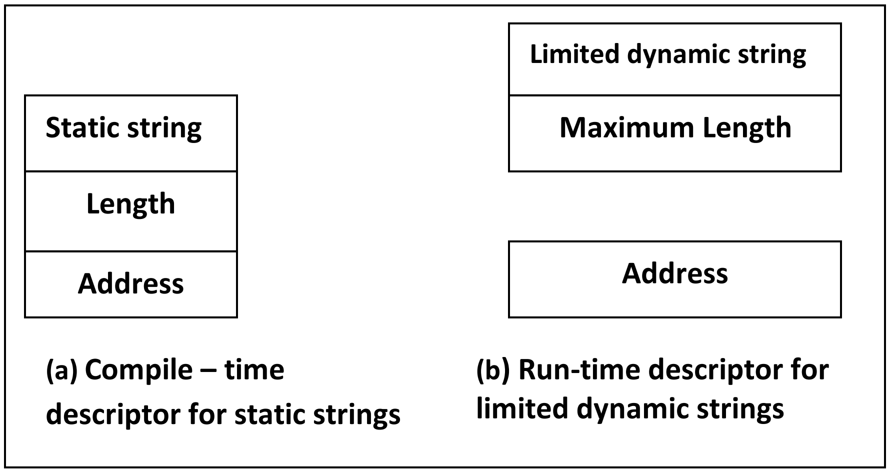
i. Should strings be simply a special kind of character array or a primitive type?
ii. Should strings have static or dynamic length?

String Operations

i. Assignment (Java: str1 = str2;) (C: strcpy(pstr1, pstr2);
ii. Comparison (=, >, etc.) BASIC: str1 < str2
iii. Concatenation, C: strcat (str1,str2), (Java : str2 + str3;)
iv. Substring reference
v. Pattern matching, C: strcmp(str1,str2);

Implementation

i. **Static length** - compile-time descriptor
ii. **Limited dynamic length** - may need a run-time descriptor for length (but not in C and C++ because the end of a string is marked with the null character)
iii. **Dynamic length** - need run-time descriptor; allocation/deallocation is the biggest implementation problem



(a) Compile – time descriptor for static strings

(b) Run-time descriptor for limited dynamic strings

Figure 2.2 Compile and Run time Descriptor for static and dynamic string

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
Design issue:
Should a symbolic constant be allowed to be in more than one type definition?
Examples

i. Java does not include an enumeration type, but provides the **Enumeration** interface

C# example

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

ii. Evaluation of enumeration types
iii. aid to readability e.g. no need to code a color as number.
iv. aid to reliability e.g. compiler can check

Array
Array is a data structure, which provides the facility to store a collection of data of same type under single variable name. Just like the ordinary variable, the array should also be declared properly. The declaration of array includes the type of array that is the type of value we are going to store in it, the array name and maximum number of elements.
Design Issues

i. What types are legal for subscripts?
ii. Are subscripting expressions in element references range checked?
iii. When are subscript ranges bound?
iv. When does allocation take place?
v. Are Jagged or rectangular multidimensioned arrays allowed, or both?
vi. Can array objects be initialized?
vii. Are any kind of slices allowed?

1. Indexing is a mapping from indices to elements map(array_name, index_value_list) $\rightarrow$ an element
2. Index Syntax
   i. FORTRAN, PL/I, Ada use parentheses
   ii. Most other languages use brackets
3. Subscript Types:
   i. FORTRAN, C - integer only
   ii. Java - integer types only
4. Array Initialization

Some languages allow initialization at the time of storage allocation

i. C, C++, Java, C# example
   int list [] = {4, 5, 7, 83} ;
ii. Character strings in C and C++
   char name [] = “freddie”;
iii. Arrays of strings in C and C++
   char *names [] = {“Bob”, “Jake”, “Joe”};
iv. Java initialization of String objects
   String[] names = {“Bob”, “Jake”, “Joe”};

Associative

i. An associative array is an unordered collection of data elements that are indexed by an equal number of values called keys.
ii. Also known as Hash tables
  a. Index by key (part of data) rather than value.
  b. Store both key and value (take more space).
  c. Best when access is by data rather than index.
iii. Examples:
  Lisp alist: ((key1 . data1) (key2 . data2) (key3 . data3)
  $age = array("Peter"=>"35", "Ben"=>"37", "Joe"=>"43");

Associative arrays have an index that is not necessarily an integer, and can be sparsely populated. The index for an associative array is called the key, and its type is called the Key Type.

Design Issues

i. What is the form of references to elements?
ii. Is the size static or dynamic?

Record

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

Design Issues

i. What is the form of references?
ii. What unit operations are defined? (Assignment, equality, assign corresponding filed)

Some Important Points

1. Implementation method

i. Simple and efficient, because field name references are literals bound at compile-time.
ii. Use offsets to determine address.

2. Record Definition Syntax

COBOL uses level numbers to show nested records; others use recursive definitions.

3. Record Field References

i. COBOL

field_name OF record_name_1 OF ... OF record_name_n

ii. Others (dot notation)

record_name_1.record_name_2. ... .record_name_n.field_name

4. **Record Operations**

i. Assignment

a. Pascal, Ada, and C allow it if the types are identical
b. In Ada, the RHS can be an aggregate constant

ii. Initialization

Allowed in Ada, using an aggregate constant

iii. Comparison

In Ada, = and /=; one operand can be an aggregate constant

iv. Move Corresponding

a. In COBOL - it moves all fields in the source record to fields with the same names in the destination record

b. Useful operation in data processing application, where input records are moved to output files after same modification.

Comparing records and arrays

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

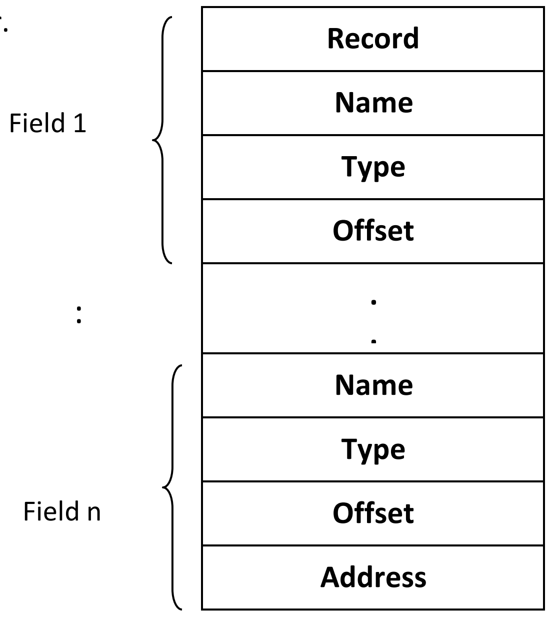


Figure 2.3 Record

Union

Union is a data type with two or more member similar to structure but in this case all the members share a common memory location. The size of the union corresponds to the length of the largest member. Since the member share a common location they have the same starting address.The real purpose of unions is to prevent memory fragmentation by arranging for a standard size for data in the memory. Java has neither records nor unions.

The syntax of union declaration is

```
union union_name
{
type element 1;
type element 2;
…………….
type element n;
};
```

This declares a type template. Variables are then declared as:

union union_name x,y,z;

For example, the following code declares a union data type called Student and a union variable called stud:

```
union student
{
int rollno;
float totalmark;
};
```

Design Issues for unions
i. Should type checking be required? Note that any such type checking must be dynamic.
ii. Should unions be integrated with records?

Pointer
Pointer is a variable that represents the location of a data item, such as variable or an array element. Within the computer's memory, every stored data item occupies one or more contiguous memory cells. The number of memory cells required to store a data item depends on the type of the data item. For example, a single character will typically be stored in one byte of memory; an integer usually requires two contiguous bytes, a floating-point number usually requires four contiguous bytes, and a double precision usually requires eight contiguous bytes.
For example, a C program contains the following declarations.
int i,*ptri;
float f,*ptrf;
The first line declares i to be an integer type variable and ptri to be a pointer variable whose object is an integer quantity. The second line declares f to be a floating-point type variable and ptrf to be a pointer variable whose object is a floating point quantity.

Uses
i. Addressing flexibility (support indirect addressing)
ii. Dynamic storage management (scoping)
Design Issues
What is the scope and lifetime of pointer variables?
i. What is the lifetime of heap-dynamic variables?
ii. Are pointers restricted to pointing at a particular type?
iii. Are pointers used for dynamic storage management, indirect addressing, or both?
iv. Should a language support pointer types, reference types, or both?

Variables
A variable is an abstraction of a memory cell. They have the following characteristics:
i. Name The identifier that refers to the variable. Variables created dynamically with new or malloc can be anonymous.
ii. Address The location in memory where the variable is stored. A single variable can have multiple different addresses as a program runs. It is also possible that multiple variables can refer to the same address. An address is sometimes called an "L-Value" since it is needed for the left-hand side of an assignment.
iii. Value The contents of the variable. A value is sometimes called an "R-Value" since it is needed for the right-hand side of an assignment.
iv. Type Determines what possible values can be stored in the variable and what operations are permitted on it. We will discuss types more in the future.
v. Lifetime The period of time when a variable exists.
vi Scope The portion of code which can access the variable.

```
int main() {
  int a;
  int b;
}
```

The above program creates two variables to reserve two memory locations with names a and b. We created these variables using int keyword to specify variable data type which means we want to store integer values in these two variables. Similarly, you can create variables to store long, float, char or any other data type

Binding

A binding is an association between an entity and an attribute, such as between a variable and its type or value, or between a function and its code.

Binding time is the point at which a binding takes place. There are different times a binding can happen:

i. Design Time Some binding decisions are made when a language is designed such as the binding of + to addition in C, the operations of the String class in Java and so on.

ii. Compile Time Bindings can also be done while the program is compiled such as binding variables to types in C++ or Java.

iii. Link Time aaasLink time is when compiled code is combined into a full program for C and C++. At this time, global and static variables are bound to addresses.

iv. Run Time Many bindings happen at run time including most values being bound to variables. In dynamically typed languages like Python, types are bound at run time as well.

Any binding that happens at run time is called dynamic, and any that happens before run time is called static. Things that are statically bound can not change as a program runs, and those that are dynamically bound can change.

What bindings take place in the following C code?

count = count + 5;

- The type of count is bound at compile time
- The set of possible values of count is bound at 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 design time.
- The value of count is bound at execution time with this statement

Static Type Binding

Static type binding can be done either implicitly or explicitly. Explicit declaration means that the programmer must specify the types of variables. This is done in languages like C, C++, and Java.

The code below has a few type annotations:

```
int fact(int x) {
  if(x == 0) {
    return 1;
  } else {
    return x * fact(x - 1);
  }
}
```

Implicit static typing means that the programmer does not need to declare types. In languages with type inferencing, the types are still statically bound, but it is done by the compiler. These languages also normally provide the ability to declare types if necessary. Type inferencing can lead to difficult compiler errors if the programmer makes a type error.

Below is the factorial function in Haskell which uses type inference:

```
fact x =
  if x == 0 then
    1
  else
    x * fact (x - 1)
```

What are some benefits and drawbacks of type inference?

Dynamic Type Binding

Dynamic type binding is when the type of a variable is not decided until the program runs. Dynamically-bound types are always implicit.

Below is the factorial function in Python which uses dynamic binding:

```
def fact(x):
  if x == 0:
    return 1
  else:
    return x * fact(x - 1)
```

Dynamic typing can lead to run time errors that a compiler would have caught. Consider the following Python code that attempts to add the numbers from from 1 to 100:

```
total = 0
number = 1

while number <= 100:
  toal = total + number
  number = number + 1

print(total)
```

This code prints “0” instead of “5050". This type of error can be difficult to debug.

TYPE CHECKING

Type checking is the process of verifying and enforcing the constraints of types, and it can occur either at compile time (i.e. statically) or at runtime (i.e. dynamically). Type checking is all about ensuring that the program is type-safe, meaning that the possibility of type errors is kept to a minimum. A type error is an erroneous program behavior in which an operation occurs (or trys to occur) on a particular data type that it’s not meant to occur on. This could be a situation where an operation is performed on an integer with the intent that it is a float, or even something such as adding a string and an integer together:

Static Type Checking

A language is statically-typed if the type of a variable is known at compile time instead of at runtime. Common examples of statically-typed languages include Ada, C, C++, C#, JADE, Java, Fortran, Haskell, ML, Pascal etc. The big benefit of static type checking is that it allows many type errors to be caught early in the development cycle. Static typing usually results in compiled code that executes more quickly because when the compiler knows the exact data types that are in use, it can produce optimized machine code (i.e. faster and/or using less memory). Static type checkers evaluate only the type information that can be determined at compile time, but are able to verify that the checked conditions hold for all possible executions of the program, which eliminates the need to repeat type checks every time the program is executed.

For example consider a statement in c++

```
int a=10;
```

here compiler needs to know the datatype of variable "a" before using it.

Dynamic Type Checking

Dynamic type checking is the process of verifying the type safety of a program at **runtime**.

In Dynamic type checking our need not specify or declare the type of variable instead compiler itself figures out what type a variable is when you first assign it a value.

Now consider some statements in python:

```
str="Python"
str2=10
```

Here you need not declare the data type. The compiler itself will know which type the variable belongs to when you first assign it a value (str1 is of "String" data type and str2 is of type "int"). Common dynamically-typed languages include JavaScript, Lisp, PHP, Prolog, Python, Ruby, Smalltalk.

Strong Typing
A strongly typed language is one in which each name in a program in the language has a single type associated with it, and that type is known as compile time. The essence of this definition is that all types are statically bound. The weakness of this definition is that it ignores the possibility that the storage location to which it is bound may store values of different types at different times. We define a programming language to be **strongly typed** if type errors are always detected.

Type Compatibility
There are two types of compatibility methods: name compatibility and structure compatibility.
Name type compatibility means that 2 variables have compatible types only if they are in either the same declaration or in declarations that use the same type name.
Example: int x; int y; // x and y are name type compatible (if the variables are declared using the same declaration or the same type)

Structure type compatibility means that 2 variables have compatible types if either type has identical structures.
Example struct foo { int x; float y; };
struct bar { int x; float y;};
foo a; bar b; // a and b are structure type compatible (if the variables have the same structure even though they are of differently named type)

1. C uses compatibility by structure while C++ uses name compatibility
2. Ada uses name compatibility except for anonymous arrays which use structure compatibility

Named constants- a constant is a value that cannot be altered by the program during normal execution, i.e., the value is constant. When associated with an identifier, a constant is said to be "named," although the terms "constant" and "named constant" are often used interchangeably. This is contrasted with a variable, which is an identifier with a value that can be changed during normal execution, i.e., the value is variable. Constants are useful for both programmers and compilers: for programmers they are a form of self-documenting code and allow reasoning about correctness; while for compilers they allow compile-time and run-time checks that constancy assumptions are not violated, and allow or simplify some compiler optimizations.

Variable initialization- A variable provides us with named storage that our programs can manipulate. Each variable in Java has a specific type, which determines the size and layout of the variable's memory; the range of values that can be stored within that memory; and the set of operations that can be applied to the variable.
Initialization is often done on the declaration statement, e.g., in Java

```
int sum = 0;
```

There are three kinds of variables in Java −
(a) Local variables
(b) Instance variables
(c) Class/Static variables

Local Variables
1. Local variables are declared in methods, constructors, or blocks.
2. Local variables are created when the method, constructor or block is entered and the variable will be destroyed once it exits the method, constructor, or block.
3. Access modifiers cannot be used for local variables.
4. Local variables are visible only within the declared method, constructor, or block.
5. Local variables are implemented at stack level internally.

Instance Variables

1. Instance variables are declared in a class, but outside a method, constructor or any block.
2. When a space is allocated for an object in the heap, a slot for each instance variable value is created.
3. Instance variables are created when an object is created with the use of the keyword 'new' and destroyed when the object is destroyed.
4. Instance variables hold values that must be referenced by more than one method, constructor or block, or essential parts of an object's state that must be present throughout the class.

Class/Static Variables

1. Class variables also known as static variables are declared with the static keyword in a class, but outside a method, constructor or a block.
2. There would only be one copy of each class variable per class, regardless of how many objects are created from it.
3. Static variables are rarely used other than being declared as constants. Constants are variables that are declared as public/private, final, and static. Constant variables never change from their initial value.
4. Static variables are stored in the static memory. It is rare to use static variables other than declared final and used as either public or private constants.

Sequence control with Expressions: -The control of the order of execution of the operations both primitive and user defined.

Implicit: determined by the order of the statements in the source program or by the built-in execution model

Explicit: the programmer uses statements to change the order of execution (e.g. uses If statement)

Conditional Statements-

In the programs that we have examined to this point, each of the statements is executed once, in the order given. Most programs are more complicated because the sequence of statements and the number of times each is executed can vary. We use the term *control flow* to refer to statement sequencing in a program.

If Statement

The simplest if structure involves a single executable statement. Execution of the statement occurs only if the condition is true.

Syntax:

```
if (condition)
 statement;
```

If-else statement

In if-else statement if the condition is true, then the true statement(s), immediately following the if-statement are executed otherwise the false statement(s) are executed. The use of else basically allows an alternative set of statements to be executed if the condition is false.

Syntax:

```
 If (condition)
{
        Statement(s);
 }
else
{
        statement(s);
 }
```

IF -else if statement

It can be used to choose one block of statements from many blocks of statements. The condition which is true only its block of statements is executed and remaining are skipped.

Syntax:

```
 if (condition)
 {
     statement(s);
 }
else if (condition)
{
      statement(s);
 }
 else
 {
     (statement);
 }
```

Switch Statement

Switch statement is alternative of nested if-else.it is executed when there are many choices and only one is to be executed.

Syntax:

```
switch(expression)
{
case 1:
statement;
break;
case 2:
statement;
break;
.
.
.
.
case N:
statement;
break;
default:
statement;
}
```

Loops

Looping statement are the statements execute one or more statement repeatedly several number of times. In C programming language there are three types of loops; while, for and do-while.

Advantage with looping statement

i. Reduce length of Code

ii. Take less memory space.

iii. Burden on the developer is reducing.

iv. Time consuming process to execute the program is reduced.

Types of Loops.

There are three types of Loops available in 'C' programming language.

while loop-

Syntax-

```
While(expression)
{   Block of statements;        }
```

for loop

Syntax

```
for ( expression1; expression2; expression3)
{
Single statement
or
Block of statements;
}
```

do..while

Syntax-

```
do
{
  Single statement
  or
  Block of statements;
}while(expression);
```

Exception handling-

An exception is a problem that arises during the execution of a program. A C++ exception is a response to an exceptional circumstance that arises while a program is running, such as an attempt to divide by zero.

Exceptions provide a way to transfer control from one part of a program to another. C++ exception handling is built upon three keywords: **try, catch,** and **throw**.

Throw − A program throws an exception when a problem shows up. This is done using a **throw** keyword.

Catch − A program catches an exception with an exception handler at the place in a program where you want to handle the problem. The **catch** keyword indicates the catching of an exception.

Try − A **try** block identifies a block of code for which particular exceptions will be activated. It's followed by one or more catch blocks.

Subject Name: **Principles of Programming Languages**

Subject Code: **CS-6002**

Semester: **6th**

Unit-3

General Subprogram Characteristics

i. Each subprogram has a single entry point.
ii. The calling program unit is suspended during the execution of the called subprogram, which implies that there is only one subprogram in execution at any given time.
iii. Control always returns to the caller when the subprogram execution terminates.

Although FORTRAN subprograms can have multiple entries, that particular kind of entry is relatively unimportant because it doesn't provide any fundamentally different capabilities. Other alternatives to the above assumptions results in co routines and concurrent units.

Basic Definitions

A subprogram definition describes the actions of the subprograms abstraction. It is the explicit request that the subprogram be executed. It said to be active if, after having been called, it has begun execution but has not yet completed that execution.

A subprogram header, which is the first line of the definition, serves several purposes. First, specifies that the following syntactic unit is a subprogram definition of some particular kind. This specification is often accomplished with a special word. Second, it provides a name for the subprogram. Third, it may optionally specify a list of parameters. A subprogram header can itself serve as subprogram declaration.

SUBROUTINES ADDER (parameters)

This is the header of a FORTRAN subroutine subprogram named ADDER. In ADA, the header for ADDER would be

procedure ADDER (parameters)

No special word appears in the header of a C subprograms. C has only one kind of subprogram, the function, and the header of a function is recognized by context rather than by a special word. For example,

adder (parameters)

would serve as the header of a function named adder.

The parameters profile of a subprogram is the number, order, and types of its formal parameters. The protocol of a subprogram is its parameter profile plus, if it is a function, its return type. In languages in which subprograms have types, those types are defined by the subprogram's protocol.

Subprograms can have declarations as well as definitions. This parallels the variable declarations and definitions in C, in which the declarations are used to provide type information but not to define variables. They are necessary when a variable must be referenced before the compiler has seen its definition. Subprogram declarations provide interface information, which is primarily parameters types, but do not include subprogram bodies. They are necessary when the compiler must translate a call to a subprogram before it has seen that subprogram's definition. In both the cases of variables and subprograms, declarations are needed for static type checking. Subprogram declarations are common in C programs, where they are called **prototypes.** They are also used in ADA and Pascal, where they are sometimes called forward or external declarations.

1. Int cube(int);   // prototype
2. Int main(){
3. Int y=5;   //actual parameters
4. Cout<<cube(y);   //Subprogram call
5. Int x=3;
6. Int cube (int x);   //subprogram heade

```
7. {                // in line 6 int x is formal parameter
8. return x*x;
9. }
10. }
```

Scope and Lifetime of a variable

Scope – The scope of any variable is actually a subset of life time. A variable may be in the memory but may not be accessible though. So, the area of our program where we can actually access our entity (variable in this case) is the scope of that variable.

The scope of any variable can be broadly categorized into three categories :

Global scope : When variable is defined outside all functions. It is then available to all the functions of the program and all the blocks program contains.

Local scope : When variable is defined inside a function or a block, then it is locally accessible within the block and hence it is a local variable.

Function scope : When variable is passed as formal arguments, it is said to have function scope. **Life Time** – Life time of any variable is the time for which the particular variable outlives in memory during running of the program.

Static: A static variable is stored in the data segment of the "object file" of a program. Its lifetime is the entire duration of the program's execution.

Automatic: An automatic variable has a lifetime that begins when program execution enters the function or statement block or compound and ends when execution leaves the block. Automatic variables are stored in a "function call stack".

Dynamic: The lifetime of a dynamic object begins when memory is allocated for the object (e.g., by a call to malloc() or using new) and ends when memory is de-allocated (e.g., by a call to free() or using delete). Dynamic objects are stored in "the heap".

Static and Dynamic Scope

Static Scoping: Static scoping is also called lexical scoping. In this scoping a variable always refers to its top level environment. This is a property of the program text and unrelated to the run time call stack. Static scoping also makes it much easier to make a modular code as programmer can figure out the scope just by looking at the code. In contrast, dynamic scope requires the programmer to anticipate all possible dynamic contexts.

In most of the programming languages including C, C++ and Java, variables are always statically (or lexically) scoped i.e., binding of a variable can be determined by program text and is independent of the run-time function call stack.

Example- A C Program to demonstrate static scoping.

```
#include<stdio.h>
int x = 10;
// Called by g()
int f()
{
  return x;
}
```

```
// g() has its own variable
// named as x and calls f()
int g()
{
  int x = 20;
  return f();
}

int main()
{
 printf("%d", g());
 printf("\n");
 return 0;
}
```

Output :

```
10
```

Dynamic Scoping: With dynamic scope, a global identifier refers to the identifier associated with the most recent environment, and is uncommon in modern languages. In technical terms, this means that each identifier has a global stack of bindings and the occurrence of a identifier is searched in the most recent binding.

In simpler terms, in dynamic scoping the compiler first searches the current block and then successively all the calling functions.

```
// Since dynamic scoping is very uncommon in
// the familiar languages, we consider the
// following pseudo code as our example. It
// prints 20 in a language that uses dynamic
// scoping.
int x = 10;
// Called by g()
int f()
{
  return x;
}
// g() has its own variable
// named as x and calls f()
int g()
{
  int x = 20;
  return f();
}
main()
```

```
{
  printf(g());
}
```

Output

```
20
```

Design Issues for Subprograms

1. What parameter passing Methods are Provided
2. Are parameter types checked?
3. Are local variables static or dynamic?
4. Can subprogram definitions appear in other program sub definition?
5. What is the referencing environment of a passed subprogram?
6. Can subprograms be overloaded?
7. Are subprograms allowed to be generic?

1. Parameter Passing Methods

Parameter passing methods are the ways in which parameters are transmitted to and/or from called subprograms

Semantics Models of Parameter Passing

Formal parameters are characterized by one these distinct semantics models (1) They can receive data from the corresponding actual parameter, (2) They can transmit data to the actual parameter (3) They can do both. These three semantics models are called in mode, out mode, and inout mode, respectively. There are two conceptual model of how data transfers take place in parameter transmission: Either an actual value is physically moved (to the caller, to the callee, or both ways, and an access path is transmitted.

(i) pass-by-value (In-mode)

The value of the actual parameter is used to initialize the corresponding formal parameter. This formal parameter is then used as a local variable in the subprogram. Since the subprogram is receiving data from the actual parameter, this is a model of in-mode semantics. In most cases, pass-by-value is implemented using copy when an actual value is copied then transmitted. However, it can be implemented by passing an access path to the value of the actual parameter.

Advantage - Speed.

Example

```
#include <iostream.h>
int square (int x)
{
return x*x;
}
int main ( )
{
int num = 10;
int answer;
answer = square(num);
cout<<"Answer is "<<answer; // answer is 100
cout<<" Value of a is "<<num; // num will be 10
return 0;
}
```

ii.Pass by Result

With pass-by-result, no value is transmitted to the subprogram. Instead, the formal parameter acts like a local variable, and before control is transferred back to the caller the variables value is transmitted back to the actual parameter, because no data is transferred to the subprogram, but it transmits data back to the actual parameter it is an out-mode semantic. Most typically pass-by-result uses a copy conceptual model.

iii.Pass-by-value-result

This passing method is actually a combination of pass-by-value and pass-by-result. The value of the actual parameter is used to initialize the corresponding formal parameter, which then acts as a local variable. The formal parameters must have local storage associated with the called subprogram. At termination, the subprogram transmits the value of the formal parameter back to the actual parameter. As such, it uses in out-mode semantics and copy passing conceptual model. Also, pass-by-value-result has the same advantages and disadvantages as pass-by-value and pass-by-result with some more advantages. The largest extra advantage of pass-by-value-result is that it solves pass-by-reference's aliasing problems.

iv.Pass-by-reference

With pass-by-reference an address (reference to the memory location of the actual parameter) is passed to the subprogram. It is another example of an inout-mode semantic.

Advantage- It is efficient in both time and space. This is no duplicate space required or copying.

Disadvantage – It increase the time to access formal parameters because of the additional level of indirect addressing.

```
// Illustration of pass by reference
#include <iostream.h>
void square (int *x)
{
*x = (*x) * (*x);
}
int main ( )
{
int num = 10;
square(&num);
cout<<" Value of num is "<<num; // Value of num is 100
return 0;
}
```

As you can see the result will be that the value of a is 100. The idea is simple: the argument passed is the address of the variable 'num'. The parameter of 'square' function is a pointer pointing to type integer. The address of 'num' is assigned to this pointer. You can analyze it as follows: &num is passed to int *x, therefore it is the same as:

int *x = &num;

This means that 'x' is a pointer to an integer and has the address of the variable num.

Within the function we have: *x = (*x) * (*x);
* when used before a pointer will give the value stored at that particular address. Hence we find the product of 'num' and store it in 'num' itself. i.e. the value of 100 is stored in the address of 'num' instead of 10 which was originally present there. The diagram below illustrates the difference between pass by value and pass by reference. Now when we dereference 'x' we are actually manipulating the value stored in 'num'.

Difference between Pass by value and pass by reference

S.No	Pass by Value	Pass by Reference
1	Passes an argument by value.	Passes an argument by reference.
2	Specifying the ByVal keyword.	Specifying the ByRef keyword.
3	The procedure code does not have any access to the underlying element in the calling code.	The procedure code gives a direct reference to the programming element in the calling code.
4	In this, you are sending a copy of the data.	In this, you are passing the memory address of the data that is stored.
5	A change does not affect the actual value.	Changes to the value effect the original data.

2. Type-Checking Parameters

It is now widely accepted that software reliability demands that the types of actual parameters be checked for consistency with the types of the corresponding formal parameters.

Example

Result = sub1 (1)

- The actual parameter is an integer constant. If the formal parameter of sub1 is a floating-point type, no error will be detected without parameter type checking.
- Early languages, such as Fortran 77 and the original version of C, did not require parameter type checking.
- Pascal, FORTRAN 90, Java, and ADA: it is always required
- Perl, PHP, and JavaScript do not have type checking.

3. Local Referencing Environments

- Variables that are defined inside subprograms are called local variables.
- Local variables can be either static or stack dynamic "bound to storage when the program begins execution and are unbound when execution terminates."

Advantages of using stack dynamic

a. Support for recursion.

b. Storage for locals is shared among some subprograms.

Disadvantages

a. Allocation/deallocation time.

b. Indirect addressing "only determined during execution."

c. Subprograms cannot be history sensitive "can't retain data values of local variables between calls."

Advantages of using static variables

a. Static local variables can be accessed faster because there is no indirection.

b. No run-time overhead for allocation and deallocation.

c. Allow subprograms to be history sensitive.

Disadvantages:

a. Inability to support recursion.

b. Their storage can't be shared with the local vars of other inactive Subprograms.

In C functions, locals are stack-dynamic unless specifically declared to be static.

Example

```
int adder(int list[ ], int listlen) {
static int sum = 0; //sum is static variable
int count; //count is stack-dynamic
for (count = 0; count < listlen; count++)
sum += list[count];
return sum;
}
```

Ada subprograms and the methods of C++, Java, and C# have only stack dynamic local variables.

4. Parameters that are Subprogram Names

- In languages that allow nested subprograms, such as JavaScript, there is another issue related to subprogram names that are passed as parameters.
- The question is what referencing environment for executing the past subprogram should be used.
- The three choices are:
  (i) It is the environment of the call statement that enacts the past subprogram "Shallow binding."
  (ii) It is the environment of the definition of the passed subprogram "Deep binding."
  (iii) It is the environment of the call statement that passed the subprogram as an actual parameter "Ad hoc binding; has never been used"

Example "written in the syntax of Java"

```
function sub1( ) {
var x;
function sub2( ) {
alert(x); // Creates a dialog box with the value of x
};
function sub3( ) {
var x;
x = 3;
sub4(sub2);
};
function sub4(subx ) {
var x;
x = 4;
subx( );
};
x = 1;
sub3( );
};
```

Shallow Binding: The referencing environment of that execution is that of sub4, so the reference to x in sub2 is bound to the local x in sub4, and the output of the program is 4.

Deep Binding: The referencing environment of sub2's execution is that of sub1, so the reference so the reference to x in sub2 is bound to the local x in sub1 and the output is 1.

Ad hoc: The binding is to the local x in sub3, and the output is 3.

Shallow binding is not appropriate for static-scoped languages with nested subprograms.

5. Overloaded Subprograms

- An overloaded operator is one that has multiple meanings. The types of its operands determine the meaning of a particular instance of an overloaded operator.
- For example, if the * operator has two floating-point operands in a Java program, it specifies floating-point multiplication.
- But if the same operator has two integer operands, it specifies integer multiplication.
- An overloaded subprogram is a subprogram that has the same name as another subprogram in the same referencing environment.
- Every version of an overloaded subprogram must have a unique protocol; that is, it must be different from the others in the number, order, or types of its parameters, or in its return if it is a function.
- The meaning of a call to an overloaded subprogram is determined by the actual parameter list.
- Users are also allowed to write multiple versions of subprograms with the same name in ADA, Java, C++, and C#.
- Overloaded subprograms that have default parameters can lead to ambiguous subprogram calls.

Function Overloading

If any class has multiple functions with same names but different parameters then they are said to be overloaded. Function overloading allows you to use the same name for different functions, to perform, either same or different functions in the same class.

Function overloading is usually used to enhance the readability of the program. If you have to perform one single operation but with different number or types of arguments, then you can simply overload the function.

Example - int sum (int x, int y)
```
{
cout<<x+y;
}

int sum(int x, int y, int z)
{
cout<<x+y+z;
}
```
Here sum() function is overloaded, to have two and three arguments. Which sum() function will be called, depends on the number of arguments.
```
int main()
{
sum (10,20);  // sum() with 2 parameter will be called
sum(10,20,30);  //sum() with 3 parameter will be called
}
```
6. Generic Subprograms

- A programmer should not need to write four different sort subprograms to sort four arrays that differ only in element type.
- A generic or polymorphic subprogram takes parameters of different types on different activations.
- Overloaded subprograms provide a particular kind of polymorphism called ad hoc polymorphism.
- Parametric polymorphism is provided by a subprogram that takes a generic parameter that is used in a type expression that describes the types of the parameters of the subprogram.

Generic Functions in C++

- Generic functions in C++ have the descriptive name of template functions.
- The following is the C++ version of the generic sort subprogram.

Example

```
template <class Type>
void generic_sort (Type list [ ], int len) {
int top, bottom;
Type temp;
for (top = 0, top < len –2; top ++)
for (bottom = top + 1; bottom < len – 1; bottom++)
if (list [top] > list [bottom]) {
temp = list [top];
list[top] = list[bottom];
} // end for bottom
} // end for generic
```

The instantiation of this template function is:

```
float flt_list [100];
…
generic_sort (flt_list, 100);
```

Example : Function Template to find the largest number

Program to display largest among two numbers using function templates.

```
// If two characters are passed to function template, character with larger ASCII value is displayed.
#include <iostream>
using namespace std;
// template function
template <class T>
T Large(T n1, T n2)
{
        return (n1 > n2) ? n1 : n2;
}
int main()
{       int i1, i2;
        float f1, f2;
        char c1, c2;
        cout << "Enter two integers:\n";
        cin >> i1 >> i2;
        cout << Large(i1, i2) <<" is larger." << endl;
        cout << "\nEnter two floating-point numbers:\n";
        cin >> f1 >> f2;
        cout << Large(f1, f2) <<" is larger." << endl;
        cout << "\nEnter two characters:\n";
        cin >> c1 >> c2;
        cout << Large(c1, c2) << " has larger ASCII value.";
        return 0;
}
```

Output

```
Enter two integers:
5   10
10 is larger.
Enter two floating-point numbers:
12.4   10.2
12.4 is larger.
```

Design Issues for Functions

The two design issues specific to functions are

1. **Functions Side effects (Are side effect allowed ?)** :- To avoid the problem caused by functional side effects, the parameters passed to functions should always be inmode. In ADA, functions can have only inmode parameters. This avoids functional side effects caused because of its parameters or through aliasing of parameters. In most other languages, functions can have either pass by value or pass by reference parameters, thus allowing functions that cause side effects and aliasing.
2. **Types of returned value (What types of values can be returned ?):-** Many imperative languages restrict the type that can be returned by their functions.
   - FORTRAN 77, Pascal and Modula-2 functions allow only unstructured types to be returned.
   - C allows any type to be returned by its functions except arrays and functions.
   - C++ also allows any type to be returned by functions including class.

Co-routines

- A co-routine is a subprogram that has multiple entries and controls them itself
- Also called symmetric control: caller and called co-routines are on a more equal basis
- A co-routine call is named a resume
- The first resume of a co-routine is to its beginning, but subsequent calls enter at the point just after the last executed statement in the co-routine
- Co-routines repeatedly resume each other, possibly forever
- Co-routines provide quasi-concurrent execution of program units (the co-routines); their execution is interleaved, but not overlapped

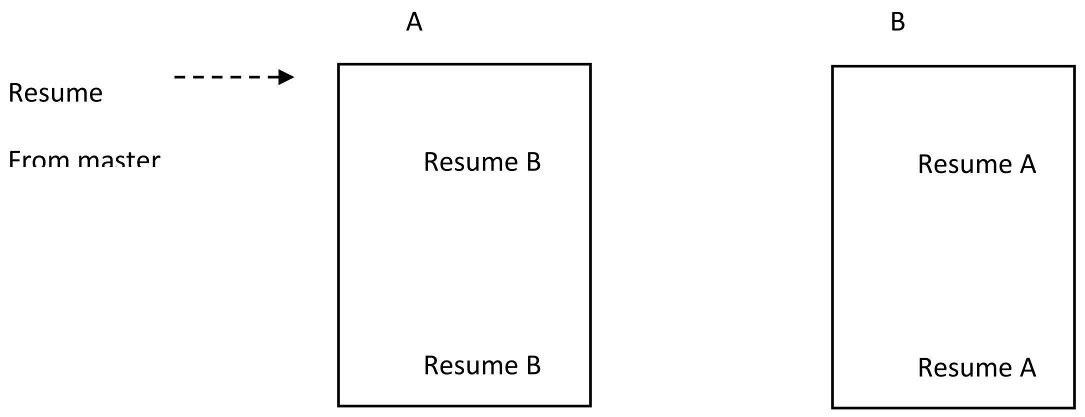


Fig-3.1 (a) Co-routine Proess

Co-routines Illustrated: Possible Execution Controls

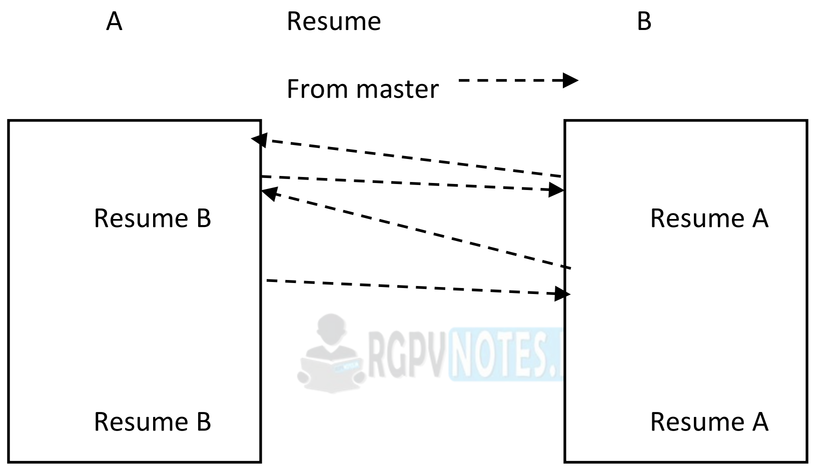


Fig-3.1 (b) Co-routine Possible Execution Controls

Co-routines Illustrated: Possible Execution Controls with Loops

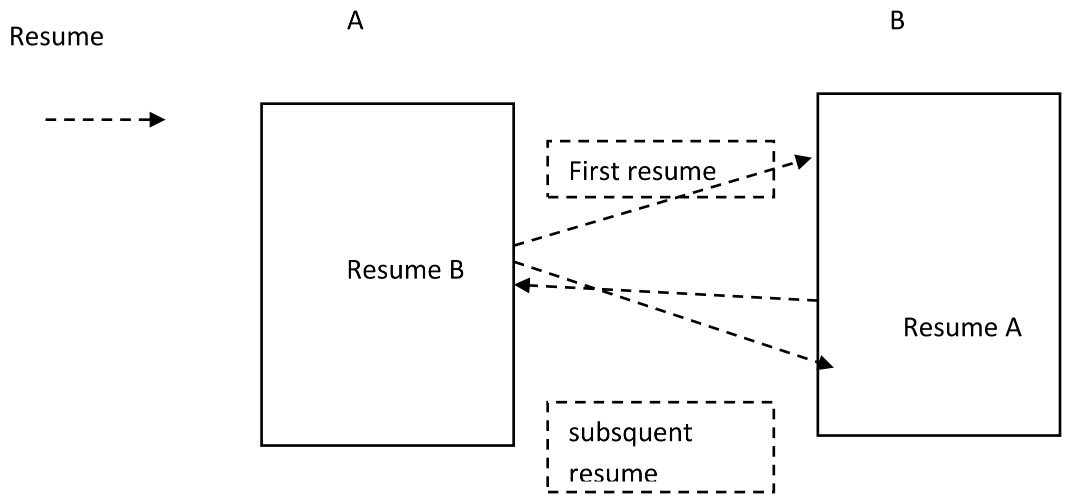


Fig-3.1 (c) Co-routine Possible Execution Controls with Loops

Subject Name: **Principles of Programming Languages**

Subject Code: **CS-6002**

Semester: **6th**

Unit 4

Abstract Data Type

Abstract data types are mathematical models of a set of data values or information that share similar behaviour or qualities and that can be specified and identified independent of specific implementations. Abstract data types, or ADTs, are typically used in algorithms. It is defined in term of its data items or its associated operations rather than by its implementation.
An abstract data structure or type "is defined indirectly, only by the operations that may be performed on it and by mathematical constraints on the effects (and possibly cost) of those operations."

Abstraction

Abstraction is a process where you show only “relevant” data and “hide” unnecessary details of an object from the user. Consider your mobile phone, you just need to know what buttons are to be pressed to send a message or make a call, what happens when you press a button, how your messages are sent, how your calls are connected is all abstracted away from the user.

Encapsulation

Encapsulation is the process of combining data and functions into a single unit called class. In Encapsulation, the data is not accessed directly; it is accessed through the functions present inside the class. In simpler words, attributes of the class are kept private and public getter and setter methods are provided to manipulate these attributes. Thus, it makes the concept of data hiding possible.

C++ supports the properties of encapsulation and data hiding through the creation of user-defined types, called classes. We already have studied that a class can contain

```
  public:
    double getVolume(void) {
      return length * breadth * height;
    }

  private:
    double length;      // Length of a box
    double breadth;     // Breadth of a box
    double height;      // Height of a box
};
```

The variables length, breadth, and height are private. This means that they can be accessed only by other members of the Box class, and not by any other part of your program

Data Encapsulation Example

Any C++ program where we implement a class with public and private members is an example of data encapsulation and data abstraction. Consider the following example:

```
#include <iostream>
using namespace std;

class Adder{
  public:
    // constructor
    Adder(int i = 0) {
      total = i;
    }
```

```
      // interface to outside world
      void addNum(int number) {
        total += number;
      }

      // interface to outside world
      int getTotal() {
        return total;
      };
             private:
      // hidden data from outside world
      int total;
  };
  int main( ) {
    Adder a;
    a.addNum(10);
    a.addNum(20);
    a.addNum(30);
    cout << "Total " << a.getTotal() <<endl;
    return 0;
  }
```

When the above code is compiled and executed, it produces the following result:

```
Total 60
```

Introduction to Data abstraction

Data abstraction refers to providing only essential information to the outside world and hiding their background details, i.e., to represent the needed information in program without presenting the details.

Data abstraction is a programming (and design) technique that relies on the separation of interface and implementation.

Let's take one real life example of a TV, which you can turn on and off, change the channel, adjust the volume, and add external components such as speakers, VCRs, and DVD players, BUT you do not know its internal details, that is, you do not know how it receives signals over the air or through a cable, how it translates them, and finally displays them on the screen.

Thus, we can say a television clearly separates its internal implementation from its external interface and you can play with its interfaces like the power button, channel changer, and volume control without having zero knowledge of its internals.

Benefits of Data Abstraction

Data abstraction provides two important advantages –

- Class internals are protected from inadvertent user-level errors, which might corrupt the state of the object.
- The class implementation may evolve over time in response to changing requirements or bug reports without requiring change in user-level code.

Data Abstraction Example

```
#include <iostream>
using namespace std;
class Adder {
   public:
    // constructor
    Adder(int i = 0) {
      total = i;
    }
      // interface to outside world
    void addNum(int number) {
      total += number;
    }
    // interface to outside world
    int getTotal() {
      return total;
    };
   private:
    // hidden data from outside world
    int total;
};
int main() {
  Adder a;
   a.addNum(10);
  a.addNum(20);
  a.addNum(30);
  cout << "Total " << a.getTotal() <<endl;
  return 0;
}
```

Storage Management

Static Storage Management

It is the simplest form of allocation i.e an allocation which remain fixed through out the execution.

Properties of static storage management

1. Storage for all variables allocated in static block.
2. Allocation can be done by translator.
3. Memory reference can be calculated during run time.
4. Subprogram variables uses space even subprogram never called.
5. Recursion is not possible.
6. All storage known at translator time, and memory reference are calculated.
7. Activation records are directly associated with code segment.
8. Procedure call and return straight forward.

Advantage

- Time or space is not expanded for storage management during execution.
- The translator can directly generate values address for all data item.

Disadvantage

- In compatible with recursive subprograms or any data structure whose size, is dependent on computed or input data.

Stack Based Storage Management

Stacks are used to hold information about procedures calling and arrange them in such a manner that when a procedure terminate, it will move to the context of the main program. They follow a run time protocol between caller and callee to save arrangements and return value on the stack. They also support recursive

subprogram and that is a system controlled storage management. Some language use stack to create local referencing environment which will vanish on the exit of the procedure.
A stack has two basic operations : Push and Pop.
Push : adds a node to the top.
Pop : Removes and return current top (LIFO).
The stack is implemented with a little more than just push and pop. The length of a stack can also returned as a parameter top[1] can return the top element without removing it.

Advantages

- As data is added and removed by LIFO. So, allocation is simpler and faster.
- Memory on the stack is automatically reclaimed with the function exits.

Disadvantage

- A stack can be very small and is dynamically grow and shrink in size. So, sometimes allocating more memory on the stack than available can result in crash due to overflow.

Heap Storage Management
A heap is a block of storage with in which pieces are allocated and freed in any manner. Here the problems of storage allocation, recovery, compaction and reuse may be serve. The heap storage management is a collection of techniques, which needs to be run side by side. There are two types of dynamic storage which needs to be co-exist in memory. They both grow and shrink through out program execution.
Properties of Heap Storage Management

- Generally used as storage for object with unrestricted lifetime.
- Maintains a list of free space (free list).
- An allocation memory allocation finds space according to any of the method and mark it as used :
  1. best fit 2. worst fit 3. first fit.
- On deallocation-memory manager marks it as free.
- Memory fragmentation - The memory is divided in small blocks during lifetime because for eg. when a block is allocated memory to then the size may be small as compared to free block that may again break.
- Garbage collection.

The allocation and deallocation can be done at arbitrary points. This techniques is divided into two categories depending on whether the elements allocated are always of the same fixed size or of variable size.

Heap Storage Management For Fixed Size Elements
In fixed size elements all the elements stored are of same size thus allocation can be done easily. Let suppose an element of heap is 'N' words long and in total there are 'K' elements. Thus size of heap is K*N words long. If allocation is done then generally is done on the basis of first fit. Initially all the 'K' elements are linked together in the form of a free space list. The free element will point to the next free element and allocated element is removed from the list. When we free the allocated storage it will take that as a part of the free list. In the free space of identification and of freed storage elements in fixed sized heap, many problems occurs and they are :-
1. **Explicit return by programmer or system :** This is the simplest method, when an element available for reuse, it will be identified as free and return to free space list but there are some problems in that :

i. Dangling reference :- It occurs when an object is deleted or deallocated, without being modified the value of the pointers. Such that the pointer is still pointing towards the deallocated memory references. This cause unpredictable results because that memory space is now allocated to any other element such that the pointer can now modify the value.

ii. Garbage :- It is opposite to dangling reference, if the access path is destroyed with the object still allocated to the memory and storage is recovered but it is not a part of the free list and not available for reuse. If garbage accumulate, available storage is reduced until the program halts due to lack of free space.

2. Reference counts :- It is an easy way to recognize garbage and makes its space reusable for new cells. In that, each heap cell is argument by count field, which record the total number of pointer which points to the particular cell. when an item is initially allocated, the reference count is set to 1, and every time a new pointer points to the object the counter is increased by 1 and each time, a pointer is destroyed the reference counts is decreased by 1. If counter sets to 0, we can reuse the cell by placing it on a free list and non-zero reference counts indicate that the structure still accessible and a free command cannot access that.

Garbage collection

The basic principle of how a garbage collector works are :

- determine what data objects in a program will not be accessed in future.
- and after determination, will reclaim the resources used by those objects.

We sat that dangling reference is bigger problem of two, they both are associated with the freeing of storage. In dangling reference, the memory is freed too early and in garbage memory is freed too late and it is better to have garbage in a program rather than dangling reference. So, we avoid dangling and allow garbage to be created. Which creates problem when the heap is full so garbage collection occurs once heap is full and process is called rarely. So it is allowable for the procedure to be fairly costly. Two stages are involved.

1. Mark :- Initially the garbage collection bit is set to 1 and elements are either active or garbage and after checking it will change the garbage collection bit of active element to 'O'. Thus every garbage is now marked as "on".
2. Sweep :- After a sequential scan, the garbage collection bit is checked and the elements whose garbage collection bit is "on" has been added to the free space list.

The marking procedure has to check all outside pointers and pointers from active members that means they are not garbage. For marking we have three critical assumption :-

- Any active element must be reachable by a chain of pointers beginning outside the heap.
- It must be possible to identify every pointer outside the heap that points to an element inside the heap.
- It must be possible to identify within any active heap element the fields that contain pointers to other heap elements.

Advantages

1. Dangling pointer bugs are reduced.
2. Double free bugs are reduced : which attempt to free a region that already freed.

Disadvantage

It adds overhead that can affect program performance. Thus more CPU time is used to free memory space.

Concepts of Object Oriented Programming in Different Languages

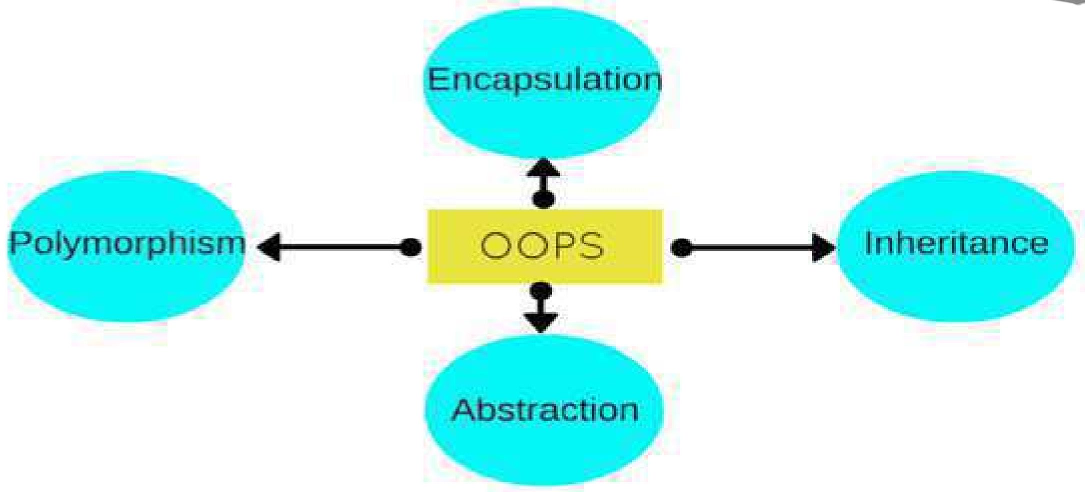


Figure 4.1 Concept of Object oriented programming

Smalltalk-

Smalltalk was the first language to include complete support for the object-oriented programming paradigm.

General Characteristics

- A program in Smalltalk consists entirely of objects.
- All objects are treated uniformly. They all have local memory, inherent processing ability, the capability to communicate with other objects, and the possibility of inheriting methods and instance variables from ancestors.
- All Smalltalk objects are allocated from the heap and are referenced through reference variables, which are implicitly de-referenced.
- The Smalltalk system integrates a program editor, compiler, the usual features of an operating system, and a virtual machine into a single system.
- Smalltalk methods are constructed from expressions. An expression specifies an object, which happens to be the value of the expression.
- The most common literals are numbers, strings and keywords.
- Smalltalk variables come in two varieties: private, which means they are local to an object, and shared, which means they are visible outside the object in which they are declared
- All Smalltalk variables are references; they can only refer to objects or classes.
- Instance variables are either named or indexed.
- Messages have the form of expressions. They provide the means of communicating among objects and are the way operations of an object are requested.

C++

C++ supports the object-oriented programming, the four major pillar of object oriented programming used in C++ are:

1. Inheritance
2. Polymorphism
3. Encapsulation
4. Abstraction

Example of Inheritance

```
#include <iostream>
using namespace std;
```

```
//Base class
class Parent
{
   public:
     int id_p;
};

// Sub class inheriting from Base Class(Parent)
class Child : public Parent
{
   public:
     int id_c;
};

//main function
int main()
  {

        Child obj1;

        // An object of class child has all data members
        // and member functions of class parent
        obj1.id_c = 7;
        obj1.id_p = 91;
        cout << "Child id is " <<  obj1.id_c << endl;
        cout << "Parent id is " <<  obj1.id_p << endl;

        return 0;
  }
```

Output:

```
Child id is 7
Parent id is 91
```

Usage of C++

By the help of C++ programming language, we can develop different types of secured and robust applications:

- Window application
- Client-Server application
- Device drivers
- Embedded firmware etc.

Java

Java is a high level, platform independent, robust, secured and object-oriented programming language.

Types of Java Applications

There are mainly 4 types of applications that can be created using java programming:

1) Standalone Application

It is also known as desktop application or window-based application. An application that we need to install on every machine such as media player, antivirus etc. AWT and Swing are used in java for creating standalone applications.

2) Web Application
An application that runs on the server side and creates dynamic page, is called web application. Currently, servlet, jsp, struts, jsf etc. technologies are used for creating web applications in java.
3) Enterprise Application
An application that is distributed in nature, such as banking applications etc. It has the advantage of high level security, load balancing and clustering. In java, EJB is used for creating enterprise applications.
4) Mobile Application
An application that is created for mobile devices. Currently Android and Java ME are used for creating mobile applications.

Example of Polymorphism
```
public Class BowlerClass{
void bowlingMethod()
{
System.out.println(" bowler ");
}
public Class FastPacer{
void bowlingMethod()
{
System.out.println(" fast bowler ");
}
Public static void main(String[] args)
{
FastPacer obj= new FastPacer();
obj.bowlingMethod();
}
}
```

C#
C# is a modern, general-purpose, object-oriented programming language developed by Microsoft and approved by European Computer Manufacturers Association (ECMA) and International Standards Organization (ISO). It is designed for Common Language Infrastructure (CLI), which consists of the executable code and runtime environment that allows use of various high-level languages on different computer platforms and architectures.
C# offers full support for OOP including inheritance, encapsulation, abstraction, and polymorphism:
Encapsulation is when a group of related methods, properties, and other members are treated as a single object.
Inheritance is the ability to receive (“inherit”) methods and properties from an existing class.
Polymorphism is when each class implements the same methods in varying ways, but you can still have several classes that can be utilized interchangeably.
Abstraction is the process by which a developer hides everything other than the relevant data about an object in order to simplify and increase efficiency.
Example of Inheritance in C##
```
using System;

namespace RectangleApplication {
  class Rectangle {

    //member variables
```

```
      protected double length;
      protected double width;

      public Rectangle(double l, double w) {
        length = l;
        width = w;
      }
      public double GetArea() {
        return length * width;
      }
      public void Display() {
        Console.WriteLine("Length: {0}", length);
        Console.WriteLine("Width: {0}", width);
        Console.WriteLine("Area: {0}", GetArea());
      }
    }//end class Rectangle
    class Tabletop : Rectangle {
      private double cost;
      public Tabletop(double l, double w) : base(l, w) { }

      public double GetCost() {
        double cost;
        cost = GetArea() * 70;
        return cost;
      }
      public void Display() {
        base.Display();
        Console.WriteLine("Cost: {0}", GetCost());
      }
    }
    class ExecuteRectangle {
      static void Main(string[] args) {
        Tabletop t = new Tabletop(4.5, 7.5);
        t.Display();
        Console.ReadLine();
      }
    }
  }
   }
```

Output

```
Length: 4.5
Width: 7.5
Area: 33.75
Cost: 2362.5
```

PHP

- PHP is a recursive acronym for "PHP: Hypertext Preprocessor".
- PHP is a server side scripting language that is embedded in HTML. It is used to manage dynamic content, databases, session tracking, even build entire e-commerce sites.

- It is integrated with a number of popular databases, including MySQL, PostgreSQL, Oracle, Sybase, Informix, and Microsoft SQL Server.

Example of Inheritance in PHP

```
<?php
class Animal
{
   private $family;
   private $food;
   public function __construct($family, $food)
   {
      $this->family = $family;
      $this->food   = $food;
   }
   public function get_family()
   {
      return $this->family;
   }
   public function set_family($family)
   {
      $this->family = $family;
   }
   public function get_food()
   {
      return $this->food;
   }
   public function set_food($food)
   {
      $this->food = $food;
   }
}
?>
<?php
class Cow extends Animal
{
   private $owner;
   public function __construct($family, $food)
   {
      parent::__construct($family, $food);
   }
   public function set_owner($owner)
   {
      $this->owner = $owner;
   }
   public function get_owner()
   {
      return $this->owner;
   }
}
?>
```

```
<?php
class Lion extends Animal
{
   public function __construct($family, $food)
   {
      parent::__construct($family, $food);
   }
}
?>
```

Perl

Perl is a general-purpose programming language originally developed for text manipulation and now used for a wide range of tasks including system administration, web development, network programming, GUI development, and more. It is a stable, cross platform programming language.

Example of Inheritance in Perl

```
#!/usr/bin/perl

package Employee;
use Person;
use strict;
our @ISA = qw(Person);    # inherits from Person

#!/usr/bin/perl

use Employee;

$object = new Employee( "Mohammad", "Saleem", 23234345);
# Get first name which is set using constructor.
$firstName = $object->getFirstName();

print "Before Setting First Name is : $firstName\n";

# Now Set first name using helper function.
$object->setFirstName( "Mohd." );

# Now get first name set by helper function.
$firstName = $object->getFirstName();
print "After Setting First Name is : $firstName\n";
```

Output

```
First Name is Mohammad
Last Name is Saleem
SSN is 23234345
Before Setting First Name is : Mohammad
Before Setting First Name is : Mohd.
```

Concurrency

Concurrency can be divided into different levels

- **Instruction level** is the execution of two or more machine instructions simultaneously.

- **Statement level** is the execution of two or more statements simultaneously.
- **Unit level** is the execution of two or more subprogram units simultaneously.
- **Program level** is the execution of two or more programs simultaneously.

Concurrent control methods increase programming flexibility

Categories of Concurrency

There are two distinct categories of concurrent unit control, **physical concurrency** and **logical concurrency**.

- **Physical concurrency** happens when several program units from the same program execute simultaneously on more than one processor.
- **Logical concurrency**, happens when the execution of several programs takes place in an interleaving fashion on a single processor

Introduction to Subprogram level Concurrency

- A task or process is a program unit that can be in concurrent execution with other program units
- Tasks differ from ordinary subprograms in that:
  1. A task may be implicitly started
  2. When a program unit starts the execution of a task, it is not necessarily suspended
  3. When a task's execution is completed, control may not return to the caller
- Tasks usually work together

Categories of Tasks

- **Heavyweight tasks** execute in their own address space
- **Lightweight tasks** all run in the same address space

If a task does not communicate with or affect another task it is considered **disjoint**

Task Synchronization

Synchronization is a mechanism that controls the order in which tasks execute.

- **Cooperation synchronization** is required between two tasks that when the second task must wait for the first task to finish executing before it may proceed.
- **Competition synchronization** is required between two tasks when both require the use of the same resource that cannot be simultaneously used. To provide competition synchronization, mutually exclusive access to the shared resource must be guaranteed.

Semaphores

Semaphores are used to restrict the number of threads than can access some (physical or logical) resource. It is devices used to help with synchronization. If multiple processes share a common resource, they need a way to be able to use that resource without disrupting each other. You want each process to be able to read from and write to that resource uninterrupted. A semaphore will either allow or disallow access to the resource, depending on how it is set up. One example setup would be a semaphore which allowed any number of processes to read from the resource, but only one could ever be in the process of writing to that resource at a time.

It is used to share a common memory space and to share access to files

Type of Semaphores

Counting semaphores are used when you might have multiple devices (like 3 printers or multiple memory buffers).

Binary semaphores are used to gain exclusive access to a single resource (like the serial port, a non-reentrant library routine, or a hard disk drive). A counting semaphore that has a maximum value of 1 is equivalent to a binary semaphore (because the semaphore's value can only be 0 or 1).

Mutex semaphores are optimized for use in controlling mutually exclusive access to a resource. There are several implementations of this type of semaphore.

Monitors

Monitors provide a structured concurrent programming, which is used by processes to ensure exclusive access to resources, and for synchronizing and communicating among users. A monitor module encapsulates both a resource definition and operations/ procedures that exclusively manipulate it. Those procedures are the gateway to the shared resource and called by the processes to access the resource. Only one call to a monitor procedure can be active at a time and this protects data inside the monitor from simultaneous access by multiple users. Thus mutual exclusion is enforced among tasks using a monitor. Processes that attempt monitor entry while the monitor is occupied are blocked on a monitor entry queue.

To synchronize tasks within the monitor, a condition variable is used to delay processes executing in a monitor. It may be declared only within a monitor and has no numeric value like semaphores do. Two operations wait and signal are defined on condition variables. The wait operation suspends/blocks execution of the calling process if a certain condition is true. Then the monitor is unlocked and allows another task to use the monitor. When the same condition becomes false, then the signal operation resumes execution of some process suspended after a wait on that condition, by placing it in the processor ready queue. If there are several such processes, choose one of them; if there are no waiting processes, the signal operator is ignored. Therefore, the introduction of condition variables allows more than one process to be in the same monitor at the same time, although only one of them will be actually active within that monitor.

A condition variable is associated with a queue of the processes that are currently waiting on that condition. First-in-first-out (FIFO) discipline is generally used with queues, but priority queues can also be implemented by specifying the priority of the process to be delayed as a parameter in the wait operation. (Condition variables are assumed to be fair in the sense that a process will not remain suspended forever on a condition variable that is signaled infinitely often.) Therefore, monitors allow flexibility in scheduling of the processes waiting in queues.

General Structure of a Monitor

```
< Monitor-Name > : monitor
begin
   Declaration of data local to the monitor.
   .
   .
   procedure < Name > ( < formal parameters > );
      begin
       procedure body
      end;

   Declaration of other procedures
   .
   .
   begin
    Initialization of local data of the monitor
   end;
end.
```

Note that a monitor is not a process, but a static module of data and procedure declarations. The actual processes which use the monitor need to be programmed separately.

Message Passing

Message passing is a type of communication between processes or objects in computer science. In this model, processes or objects can send and receive messages (signals, functions, complex data structures, or data packets) to other processes or objects.

Definition

Message passing is a form of communication between objects, processes or other resources used in object-oriented programming, inter-process communication and parallel computing. It can be synchronous or asynchronous. Synchronous message passing systems require the sender and receiver to wait for each other while transferring the message. In asynchronous communication the sender and receiver do not wait for each other and can carry on their own computations while transfer of messages is being done.

The concept of message passing makes it easier to build systems that model or simulate real-world problems.

Java Threads

Java is a multi-threaded programming language which means we can develop multi-threaded program using Java. A multi-threaded program contains two or more parts that can run concurrently and each part can handle a different task at the same time making optimal use of the available resources especially when your computer has multiple CPUs.

By definition, multitasking is when multiple processes share common processing resources such as a CPU. Multi-threading extends the idea of multitasking into applications where you can subdivide specific operations within a single application into individual threads. Each of the threads can run in parallel. The OS divides processing time not only among different applications, but also among each thread within an application.

Life Cycle of a Thread

A thread goes through various stages in its life cycle. For example, a thread is born, started, runs, and then dies. The following diagram shows the complete life cycle of a thread.

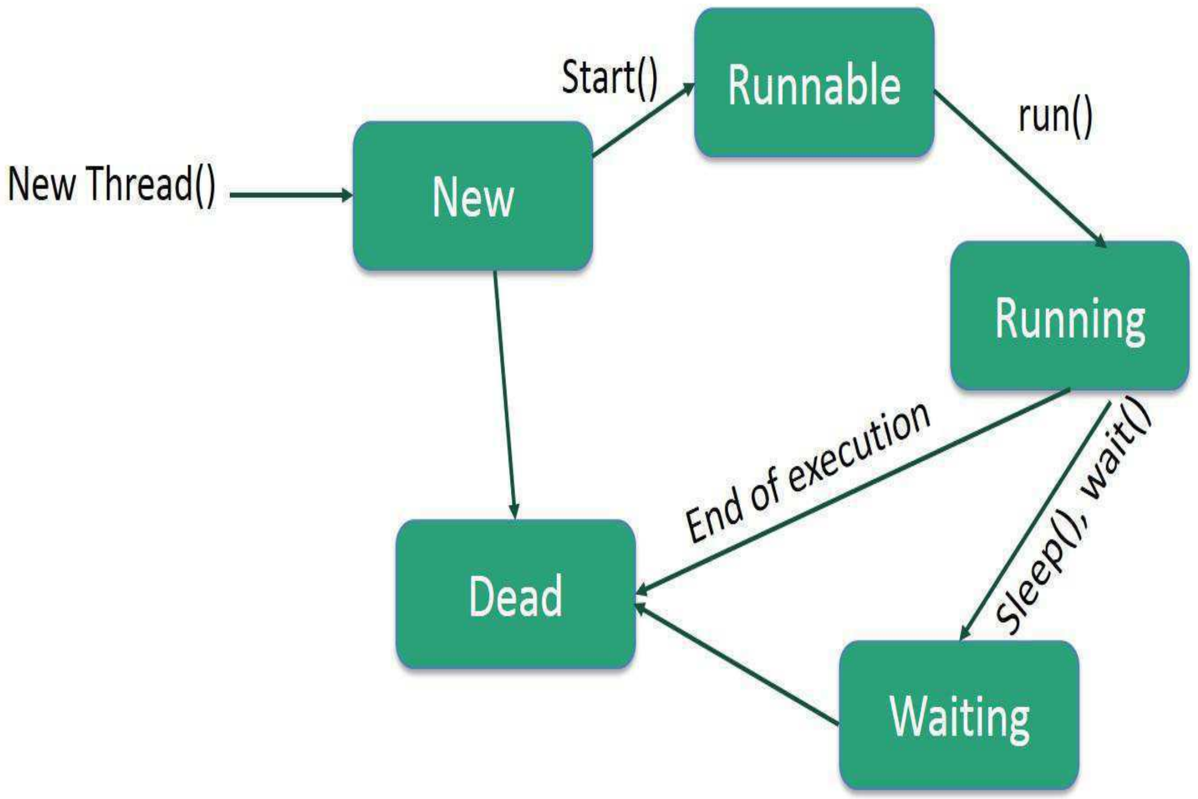


Figure 4.2 Thread Life cycle

Following are the stages of the life cycle −

- New − A new thread begins its life cycle in the new state. It remains in this state until the program starts the thread. It is also referred to as a born thread.
- Runnable − after a newly born thread is started, the thread becomes runnable. A thread in this state is considered to be executing its task.
- Waiting − Sometimes, a thread transitions to the waiting state while the thread waits for another thread to perform a task. A thread transitions back to the runnable state only when another thread signals the waiting thread to continue executing.
- Timed Waiting − A runnable thread can enter the timed waiting state for a specified interval of time. A thread in these state transitions back to the runnable state when that time interval expires or when the event it is waiting for occurs.
- Terminated (Dead) − A runnable thread enters the terminated state when it completes its task or otherwise terminates.

Thread Priorities

Every Java thread has a priority that helps the operating system determine the order in which threads are scheduled.

Java thread priorities are in the range between MIN_PRIORITY (a constant of 1) and MAX_PRIORITY (a constant of 10). By default, every thread is given priority NORM_PRIORITY (a constant of 5).

Threads with higher priority are more important to a program and should be allocated processor time before lower-priority threads. However, thread priorities cannot guarantee the order in which threads execute and are very much platform dependent.

Create a Thread by Implementing a Runnable Interface

If your class is intended to be executed as a thread then you can achieve this by implementing a Runnable interface. You will need to follow three basic steps −

Step 1

As a first step, you need to implement a run () method provided by a Runnable interface. This method provides an entry point for the thread and you will put your complete business logic inside this method. Following is a simple syntax of the run () method −

```
public void run( )
```

Step 2

As a second step, you will instantiate a Thread object using the following constructor −

```
Thread (Runnable threadObj, String threadName);
```

Where, threadObj is an instance of a class that implements the Runnable interface and threadName is the name given to the new thread.

Step 3

Once a Thread object is created, you can start it by calling start() method, which executes a call to run( ) method. Following is a simple syntax of start() method −

```
void start();
```

Example

Here is an example that creates a new thread and starts running it −

```
class RunnableDemo implements Runnable {
   private Thread t;
   private String threadName;

   RunnableDemo( String name) {
      threadName = name;
      System.out.println("Creating " +  threadName );
   }
```

```
  public void run() {
}
    System.out.println("Running " +  threadName );
    try {
    }
  }
      for(int i = 4; i > 0; i--) {
        System.out.println("Thread: " + threadName + ", " + i);
        // Let the thread sleep for a while.
        Thread.sleep(50);
      }
    }catch (InterruptedException e) {
      System.out.println("Thread " +  threadName + " interrupted.");
    }
    System.out.println("Thread " +  threadName + " exiting.");
  }
   public void start () {
    System.out.println("Starting " +  threadName );
    if (t == null) {
      t = new Thread (this, threadName);
      t.start ();
```

Result –

```
Output public class TestThread {

  public static void main(String args[]) {
    RunnableDemo R1 = new RunnableDemo( "Thread-1");
    R1.start();

    RunnableDemo R2 = new RunnableDemo( "Thread-2");
    R2.start();
  }
}
```

This will produce the following

```
Creating Thread-1
Starting Thread-1
Creating Thread-2
Starting Thread-2
Running Thread-1
Thread: Thread-1, 4
Running Thread-2
Thread: Thread-2, 4
Thread: Thread-1, 3
Thread: Thread-2, 3
Thread: Thread-1, 2
Thread: Thread-2, 2
Thread: Thread-1, 1
Thread: Thread-2, 1
Thread Thread-1 exiting.
Thread Thread-2 exiting.
```

Create a Thread by Extending a Thread Class

The second way to create a thread is to create a new class that extends Thread class using the following two simple steps. This approach provides more flexibility in handling multiple threads created using available methods in Thread class.

Step 1

You will need to override run( ) method available in Thread class. This method provides an entry point for the thread and you will put your complete business logic inside this method. Following is a simple syntax of run() method −

```
public void run( )
```

Step 2

Once Thread object is created, you can start it by calling start() method, which executes a call to run( ) method. Following is a simple syntax of start() method −

```
void start( );
```

Example

Here is the preceding program rewritten to extend the Thread −

```
class ThreadDemo extends Thread {
  private Thread t;
  private String threadName;

  ThreadDemo( String name) {
    threadName = name;
    System.out.println("Creating " +  threadName );
  }
    public void run() {
    System.out.println("Running " +  threadName );
    try {
      for(int i = 4; i > 0; i--) {
        System.out.println("Thread: " + threadName + ", " + i);
        // Let the thread sleep for a while.
        Thread.sleep(50);
      }
    }catch (InterruptedException e) {
      System.out.println("Thread " +  threadName + " interrupted.");
    }
    System.out.println("Thread " +  threadName + " exiting.");
  }

  public void start () {
    System.out.println("Starting " +  threadName );
    if (t == null) {
      t = new Thread (this, threadName);
      t.start ();
    }
  }
}

public class TestThread {
  public static void main(String args[]) {
    ThreadDemo T1 = new ThreadDemo( "Thread-1");
    T1.start();

    ThreadDemo T2 = new ThreadDemo( "Thread-2");
    T2.start();
  }
}
```

This will produce the following result –

Output

```
Creating Thread-1
Starting Thread-1
Creating Thread-2
Starting Thread-2
Running Thread-1
Thread: Thread-1, 4
Running Thread-2
Thread: Thread-2, 4
Thread: Thread-1, 3
Thread: Thread-2, 3
Thread: Thread-1, 2
Thread: Thread-2, 2
Thread: Thread-1, 1
Thread: Thread-2, 1
Thread Thread-1 exiting.
Thread Thread-2 exiting.
```

C# Thread

A thread is defined as the execution path of a program. Each thread defines a unique flow of control. If your application involves complicated and time consuming operations, then it is often helpful to set different execution paths or threads, with each thread performing a particular job.

Threads are lightweight processes. One common example of use of thread is implementation of concurrent programming by modern operating systems. Use of threads saves wastage of CPU cycle and increase efficiency of an application

Thread Life Cycle

The life cycle of a thread starts when an object of the System.Threading. Thread class is created and ends when the thread is terminated or completes execution.

Following are the various states in the life cycle of a thread –

- **The Unstarted State** – It is the situation when the instance of the thread is created but the Start method is not called.
- **The Ready State** – It is the situation when the thread is ready to run and waiting CPU cycle.
- **The Not Runnable State** – A thread is not executable, when
  - Sleep method has been called
  - Wait method has been called
  - Blocked by I/O operations
- **The Dead State** – It is the situation when the thread completes execution or is aborted.

The Main Thread

In C#, the **System.Threading.Thread** class is used for working with threads. It allows creating and accessing individual threads in a multithreaded application. The first thread to be executed in a process is called the **main** thread.

When a C# program starts execution, the main thread is automatically created. The threads created using the **Thread** class are called the child threads of the main thread. You can access a thread using the **CurrentThread** property of the Thread class.

The following program demonstrates main thread execution −

```
using System;
using System.Threading;

namespace MultithreadingApplication {
  class MainThreadProgram {
    static void Main(string[] args) {
      Thread th = Thread.CurrentThread;
      th.Name = "MainThread";

      Console.WriteLine("This is {0}", th.Name);
      Console.ReadKey();
    }
  }
}
```

Subject Name: **Principles of Programming Languages**

Subject Code: **CS-6002**

Semester: **6th**

Unit 5

Exception Handling

Exceptions - An exception (or exceptional event) is a problem that arises during the execution of a program. When an exception occurs the normal flow of the program is disrupted and the program/application terminates abnormally, which is not recommended, therefore, these exceptions are to be handled.

An exception can occur for many different reasons. Following are some scenarios where an exception occurs.

- A user has entered an invalid data.
- A file that needs to be opened cannot be found.
- A network connection has been lost in the middle of communications or the JVM has run out of memory.

Some of these exceptions are caused by user error, others by programmer error, and others by physical resources that have failed in some manner.

Exception Propagation

An exception is first thrown from the top of the stack and if it is not caught, it drops down the call stack to the previous method, If not caught there, the exception again drops down to the previous method, and so on until they are caught or until they reach the very bottom of the call stack. This is called exception propagation.

Example

```
class ExceptionPropagation{
  void m(){
    int data = 10/0;
  }
  void n(){
    m();
  }
  void p(){
   try{
    n();
   }catch(Exception e){
      System.out.println("exception handled");
   }
  }
  public static void main(String args[]){
   ExceptionPropagation obj = new ExceptionPropagation();
   obj.p();
   System.out.println("normal flow...");
  }
}
```

Output:

```
 exception handled
  normal flow...
```

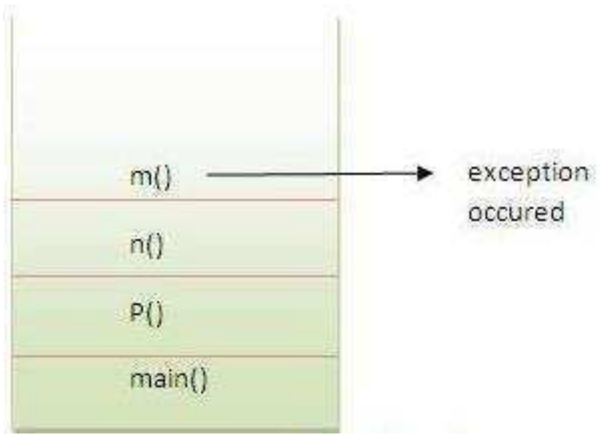


Figure 5.1 Call Stack

In the above example exception occurs in m() method where it is not handled, so it is propagated to previous n() method where it is not handled, again it is propagated to p() method where exception is handled. Exception can be handled in any method in call stack either in main() method,p() method,n() method or m() method.

Exception handling in C++

Exceptions are runtime anomalies that a program encounters during execution. It is a situation where a program has an unusual condition and the section of code containing it can't handle the problem. Exception includes condition such as division by zero, accessing an array outside its bound, running out of memory, etc. In order to handle these exceptions, exception handling mechanism is used which identifies and deal with such condition. Exception handling mechanism consists of following parts:

1. Find the problem (Hit the exception)
2. Inform about its occurrence (Throw the exception)
3. Receive error information (Catch the exception)
4. Take proper action (Handle the exception)

C++ consists of 3 keywords for handling the exception. They are

1. **try:** Try block consists of the code that may generate exception. Exceptions are thrown from inside the try block.
2. **throw:** Throw keyword is used to throw an exception encountered inside try block. After the exception is thrown, the control is transferred to catch block.
3. **catch:** Catch block catches the exception thrown by throw statement from try block. Then, exceptions are handled inside catch block.

Syntax

```
try
{
    statements;
    ... ... ...
    throw exception;
}

catch (type argument)
{
    statements;
    ... ... ...
}
```

Multiple catch exception

Multiple catch exception statements are used when a user wants to handle different exceptions differently. For this, a user must include catch statements with different declaration.

Syntax

```
try
{
   body of try block
}

catch (type1 argument1)
{
   statements;
   ... ... ...
}

catch (type2 argument2)
{
   statements;
   ... ... ...
}
... ... ...
... ... ...
catch (typeN argumentN)
{
   statements;
   ... ... ...
}
```

Example of exception handling

C++ program to divide two numbers using try catch block.

```
#include <iostream>
using namespace std;
int main()
{
   int a,b;
   cout << "Enter 2 numbers: ";
   cin >> a >> b;
   try
   {
      if (b != 0)
      {
         float div = (float)a/b;
         if (div < 0)
            throw 'e';
         cout << "a/b = " << div;
      }
      else
         throw b;
```

```
    }
    catch (int e)
    {
       cout << "Exception: Division by zero";
    }
    catch (char st)
    {
       cout << "Exception: Division is less than 1";
    }
    catch(...)
    {
       cout << "Exception: Unknown";
    }
    getch();
    return 0;
}
```

This program demonstrates how exceptions are handled in C++. This program performs division operation. Two numbers are entered by user for division operation. If the dividend is zero, then division by zero will cause exception which is thrown into catch block. If the answer is less than 0, then exception "Division is less than 1" is thrown. All other exceptions are handled by the last catch block throwing "Unknown" exception.

Output

```
Enter 2 numbers: 8 5
a/b = 1.6
Enter 2 numbers: 9 0
Exception: Division by zero
Enter 2 numbers: -1 10
Exception: Division is less than 1
```

C++ Standard Exceptions

C++ provides a list of standard exceptions defined in **<exception>** which we can use in our programs. These are arranged in a parent-child class hierarchy shown below –

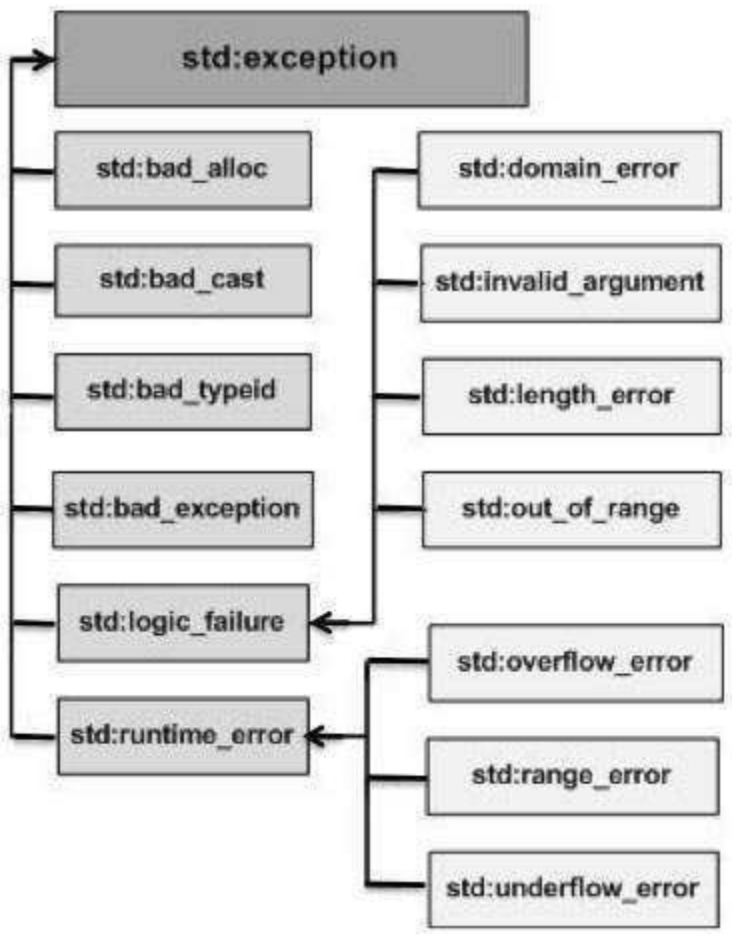


Figure 5.2 C++ Standard Exceptions

Advantage of Exception Handling
The core advantage of exception handling is **to maintain the normal flow of the application**. Exception normally disrupts the normal flow of the application that is why we use exception handling.

Exception Handling in Java
Exception Hierarchy
All exception classes are subtypes of the java.lang.Exception class. The exception class is a subclass of the Throwable class. Other than the exception class there is another subclass called Error which is derived from the Throwable class.
Errors are abnormal conditions that happen in case of severe failures, these are not handled by the Java programs. Errors are generated to indicate errors generated by the runtime environment. Example: JVM is out of memory. Normally, programs cannot recover from errors.
The Exception class has two main subclasses: IOException class and RuntimeException Class.

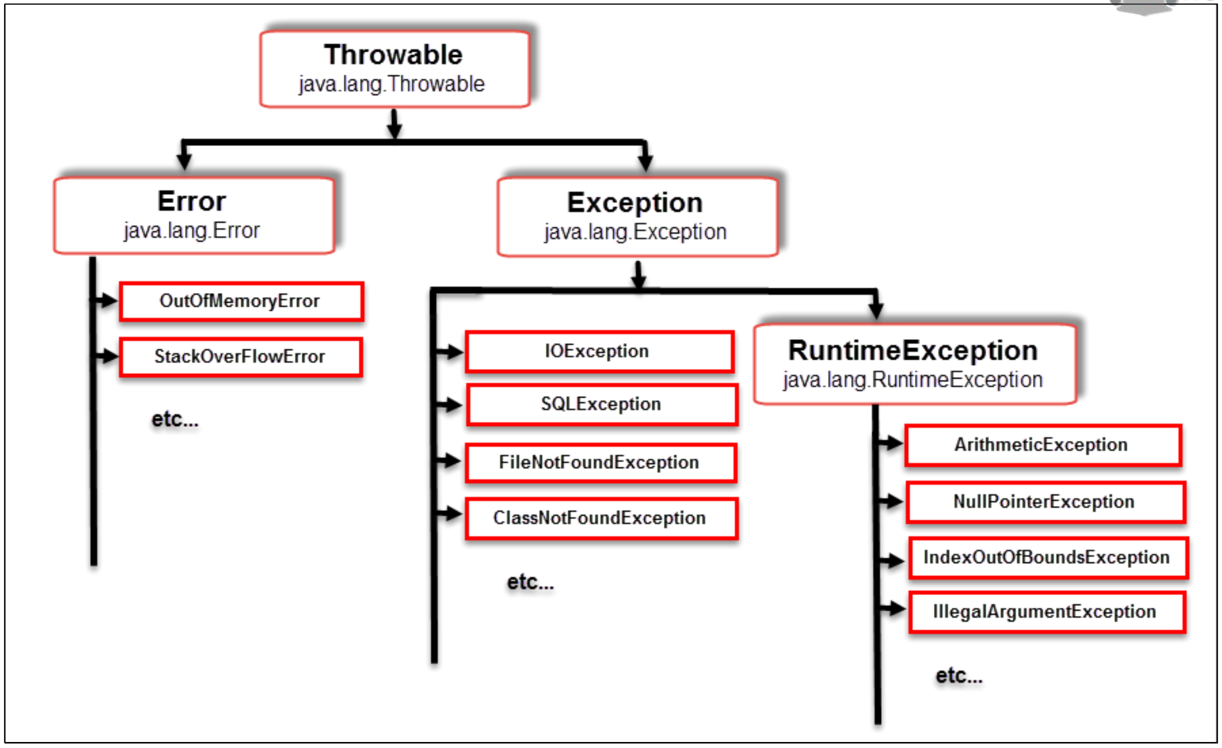


Figure 5.3 Exceptions in Java

Catching Exceptions

A method catches an exception using a combination of the try and catch keywords. A try/catch block is placed around the code that might generate an exception. Code within a try/catch block is referred to as protected code, and the syntax for using try/catch looks like the following –

Syntax

```
try {
  // Protected code
}catch(ExceptionName e1) {
  // Catch block
}
```

The code which is prone to exceptions is placed in the try block. When an exception occurs, that exception occurred is handled by catch block associated with it. Every try block should be immediately followed either by a catch block or finally block.

A catch statement involves declaring the type of exception you are trying to catch. If an exception occurs in protected code, the catch block (or blocks) that follows the try is checked. If the type of exception that occurred is listed in a catch block, the exception is passed to the catch block much as an argument is passed into a method parameter.

Example

The following is an array declared with 2 elements. Then the code tries to access the 3rd element of the array which throws an exception.

```
// File Name : ExcepTest.java
import java.io.*;

public class ExcepTest {

  public static void main(String args[]) {
    try {
      int a[] = new int[2];
      System.out.println("Access element three :" + a[3]);
    }catch(ArrayIndexOutOfBoundsException e) {
      System.out.println("Exception thrown  :" + e);
    }
    System.out.println("Out of the block");
  }
}
```

This will produce the following result −

Output

```
Exception thrown: java.lang.ArrayIndexOutOfBoundsException: 3
Out of the block
```

Multiple Catch Blocks

A try block can be followed by multiple catch blocks. The syntax for multiple catch blocks looks like the following −

Syntax

```
try {
  // Protected code
}catch(ExceptionType1 e1) {
  // Catch block
}catch(ExceptionType2 e2) {
  // Catch block
}catch(ExceptionType3 e3) {
  // Catch block
}
```

The previous statements demonstrate three catch blocks, but you can have any number of them after a single try. If an exception occurs in the protected code, the exception is thrown to the first catch block in the list. If the data type of the exception thrown matches ExceptionType1, it gets caught there. If not, the exception passes down to the second catch statement. This continues until the exception either is caught or falls through all catches, in which case the current method stops execution and the exception is thrown down to the previous method on the call stack.

Example

Here is code segment showing how to use multiple try/catch statements.

```
try {
  file = new FileInputStream(fileName);
  x = (byte) file.read();
}catch(IOException i) {
  i.printStackTrace();
  return -1;
}catch(FileNotFoundException f) // Not valid! {
  f.printStackTrace();
  return -1;
}
```

Throws/Throw Keywords

If a method does not handle a checked exception, the method must declare it using the throws keyword. The throws keyword appears at the end of a method's signature.

You can throw an exception, either a newly instantiated one or an exception that you just caught, by using the throw keyword.

throws is used to postpone the handling of a checked exception and throw is used to invoke an exception explicitly.

The following method declares that it throws a RemoteException –

Example

```
import java.io.*;
public class className {

  public void deposit(double amount) throws RemoteException {
    // Method implementation
    throw new RemoteException();
  }
  // Remainder of class definition
}
```

Finally Block

The finally block follows a try block or a catch block. A finally block of code always executes, irrespective of occurrence of an Exception.

Using a finally block allows you to run any cleanup-type statements that you want to execute, no matter what happens in the protected code.

A finally block appears at the end of the catch blocks and has the following syntax –

Syntax

```
try {
  // Protected code
}catch(ExceptionType1 e1) {
  // Catch block
}catch(ExceptionType2 e2) {
  // Catch block
}catch(ExceptionType3 e3) {
  // Catch block
}finally {
  // The finally block always executes.
}
```

Example

```
public class ExcepTest {

  public static void main(String args[]) {
    int a[] = new int[2];
    try {
      System.out.println("Access element three :" + a[3]);
    }catch(ArrayIndexOutOfBoundsException e) {
      System.out.println("Exception thrown  :" + e);
    }finally {
      a[0] = 6;
      System.out.println("First element value: " + a[0]);
      System.out.println("The finally statement is executed");
    }
  }
}
```

This will produce the following result –

Output

```
Exception thrown: java.lang.ArrayIndexOutOfBoundsException: 3
First element value: 6
The finally statement is executed
```

Important points

- A catch clause cannot exist without a try statement.
- It is not compulsory to have finally clauses whenever a try/catch block is present.
- The try block cannot be present without either catch clause or finally clause.
- Any code cannot be present in between the try, catch, finally blocks.

Logic Programming

Logic programming is a computer programming paradigm in which program statements expresses facts and rules about problems within a system of formal logic. Rules are written as logical clauses with a head and a body; for instance, "H is true if B1, B2, and B3 are true." Facts are expressed similar to rules, but without a body; for instance, "H is true."

Some logic programming languages such as Data log and Answer Set Programming (ASP) are purely declarative they allow for statements about what the program should accomplish, with no explicit step-by-step instructions about how to do so. Others, such as Prolog, are a combination of declarative and imperative — they may also include procedural statements such as "To solve H, solve B1, B2, and B3."

- Programs are written in the language of some logic.
- Execution of a logic program is a theorem proving process.
- Prolog, Programming in Logics, is a representative LP language, based on a subset of first order predicate logic. However, logic programming does not equal programming in Prolog, there can be different logic programming languages based on different logics.

Introduction and overview of Logic programming

- A logic program is a specification of a solution to a problem; in addition, it is an executable specification.
- Like LISP, LP is about manipulation of symbols, and thus has potential in AI applications.
- Unlike LISP, computations in LP are reasoning processes.

Logic programming is widely used in parsing, both in natural languages and programming languages. Since the creator of logic programming is also an linguist, it once was widely used in natural language processing (NLP), mostly in parsing natural language processing and generating natural language. Prolog and Natural language analysis is a book about this topic. But nowadays, NLP is mostly occupied by statistic approach. It is really easy to write a parser in Prolog, it is also quite often used to implement new programming. The first interpreter of Erlang Programming Language is written in Prolog. It is also used widely when there are a lot of relations, like in semantic web's RDF manipulation (which you can view it as the root of Knowledge Graph). There are also some libraries to do constrained logic programming which is quite interesting and exciting.

Basic Elements of prolog

Prolog is a logic language that is particularly suited to programs that involve symbolic or non-numeric computation. For this reason it is a frequently used language in Artificial Intelligence where manipulation of symbols and inference about them is a common task.

Prolog consists of a series of rules and facts. A program is run by presenting some query and seeing if this can be proved against these known rules and facts.

Simple Facts

In Prolog we can make some statements by using facts. Facts either consist of a particular item or a relation between items. For example we can represent the fact that it is sunny by writing the program:

sunny.

We can now ask a query of Prolog by asking

?- sunny.

?- is the Prolog prompt. To this query, Prolog will answer yes. sunny is true because (from above) Prolog matches it in its database of facts.

Facts have some simple rules of syntax. Facts should always begin with a lowercase letter and end with a full stop. The facts themselves can consist of any letter or number combination, as well as the underscore _ character. However, names containing the characters -,+,*,/, or other mathematical operators should be avoided.

Examples of Simple Facts

Here are some simple facts about an imaginary world. /* and */ are comment delimiters

```
john_is_cold.                    /* john is cold */
raining.                         /* it is raining */
john_Forgot_His_Raincoat.        /*  john forgot his raincoat */
```

Rule Statements

- Used for the hypotheses
- Headed Horn clause
- Right side: antecedent (if part)
  - – May be single term or conjunction
- Left side: consequent (then part)
  - – Must be single term
- Conjunction: multiple terms separated by logical AND operations (implied)

Example Rules

Ancestor (mary,shelley):- mother(mary,shelley).

- Can use variables (universal objects) to generalize meaning:

```
parent(X,Y):- mother(X,Y).
 parent(X,Y):- father(X,Y).
 grandparent(X,Z):- parent(X,Y), parent(Y,Z).
 sibling(X,Y):- mother(M,X), mother(M,Y),
 father(F,X), father(F,Y).
```

Application of Logic Programming

1. Relational database management system.
2. Expert System.
3. Natural language processing.
4. Symbolic Equation solving.
5. Planning
6. Prototyping.
7. Simulation.
8. Programming Language Implementation.

Functional Programming Language Introduction

Functional programming languages are specially designed to handle symbolic computation and list processing applications. Functional programming is based on mathematical functions. Some of the popular functional programming languages include: LISP, Python etc. Functional programming languages are categorized into two groups, i.e. −

- **Pure Functional Languages** − These types of functional languages support only the functional paradigms. For example − Haskell.
- **Impure Functional Languages** − These types of functional languages support the functional paradigms and imperative style programming. For example − LISP.

Functional Programming – Characteristics

The most prominent characteristics of functional programming are as follows −

- Functional programming languages are designed on the concept of mathematical functions that use conditional expressions and recursion to perform computation.
- Functional programming supports **higher-order functions** and **lazy evaluation** features.
- Functional programming languages don't support flow Controls like loop statements and conditional statements like If-Else and Switch Statements. They directly use the functions and functional calls.
- Like OOP, functional programming languages support popular concepts such as Abstraction, Encapsulation, Inheritance, and Polymorphism.

Functional Programming – Advantages

Functional programming offers the following advantages −

- **Bugs-Free Code** − Functional programming does not support **state**, so there are no side-effect results and we can write error-free codes.
- **Efficient Parallel Programming** − Functional programming languages have NO Mutable state, so there are no state-change issues. One can program "Functions" to work parallel as "instructions". Such codes support easy reusability and testability.
- **Efficiency** − Functional programs consist of independent units that can run concurrently. As a result, such programs are more efficient.
- **Supports Nested Functions** − Functional programming supports Nested Functions.
- **Lazy Evaluation** − Functional programming supports Lazy Functional Constructs like Lazy Lists, Lazy Maps, etc.

Fundamentals of Functional Programming Languages

- The objective of the design of a FPL is to mimic mathematical functions to the greatest extent possible.
- The basic process of computation is fundamentally different in a FPL than in an imperative language.
  – In an imperative language, operations are done and the results are stored in variables for later use.
  – Management of variables is a constant concern and source of complexity for imperative programming.
- In an FPL, variables are not necessary, as is the case in mathematics.

- Referential Transparency - In an FPL, the evaluation of a function always produces the same result given the same parameters.

Introduction to 4GL

A fourth-generation programming language (4GL) is any computer programming language that belongs to a class of languages envisioned as advancement upon third-generation programming languages (3GL). Each of the programming language generations aims to provide a higher level of abstraction of the internal computer hardware details, making the language more programmer-friendly, powerful and versatile. While the definition of 4GL has changed over time, it can be typified by operating more with large collections of information at once rather than focusing on just bits and bytes. Languages claimed to be 4GL may include support for database management, report generation, mathematical optimization, GUI development, or web development. Some researchers state that 4GLs are a subset of domain-specific languages

Advantages of 4GL

1. Programming productivity is increased. One line of 4GL code is equivalent to several lines of 3GL code.
2. System development is faster.
3. Program maintenance is easier.
4. The finished system is more likely to be what the user envisaged, if a prototype is used and the user is involved throughout the development.
5. End user can often develop their own applications.
6. Programs developed in 4GLs are more portable than those developed in other generation of languages.
7. Documentation is improved because many 4GLs are self documenting.

Disadvantages of 4GL

1. The programs developed in the 4GLs are executed at a slower speed by the CPU.
2. The programs developed in these programming languages need more space in the memory of the computer system.