Question Bank Programming Language

CSE-312-B

Unit-1

Q1. What is Programming Language? What are the Characteristics of good Programming Language?

Q2. What are syntactic and Semantic Rules of Programming Language?

Q3. What is compiler and Interpreter? What is difference between Compiler and Interpreter?

Q4.What are Virtual Computers and Binding times?

Q5. What is Procedural and non- Procedural Language?

Q6. What is Functional Structured and Object Oriented Language? Q7. What is Comparison between C and C++ Programming Language?

Unit-2

Q1. What are Elementary data types? Q2. What are data Objects?

Q3. What are variable and constants?

Q4. What are type checking and type Conversions? Q5. What are assignment and initialization?

Q6. What are numeric data types?

Q7. What are Enumerations?

Q8. What are Boolean Data Types?

Q9. What are Vector and Arrays?

Q10.what is Union And Pointer?

Q11. What is Programmer defined data objects?

Q12. What are sets and files?

Unit-3

Q1.What is Implicit and Explicit sequence control?

Q2.What is sequence control within Statements?

Q3. What is subprogram sequence Control?

Q4. What is recursive subprograms?

Q5.What is Exception and Exception handlers? Q6. What is Co routines?

Q7.what is static and dynamic scope?

Q8. What is Local data and Local referencing environment? Q9. What is shared Data?

Q10.what is Parameter and Parameter Transmission scheme?

Unit-4

Q1.What is static storage Management?

Q2.What is stack based storage management?

Q3.What is heap based storage management?

Q4.What is variable and fixed size element?

Q5. What is Abstraction?

Q6. What is Encapsulation?

Q7. What is Information Hiding?

Q8.Whta is type definitions?

Q9. What are Abstract data types?

Unit-1

SYNTACTIC AND SEMANTIC RULES OF A PROGRAMMING LANGUAGE

Definition of Syntax

The **Syntax** of a programming language is used to signify the structure of programs without considering their meaning. It basically emphasizes the structure, layout of a program with their appearance. It involves a collection of rules which validates the sequence of symbols and instruction used in a program. The pragmatic and computation model figures these syntactic components of a programming language. The tools evolved for the specification of the syntax of the programming languages are regular, context-free and attribute grammars.

However, what is the use of grammar in this aspect? The **Grammars** generally are the rewriting rules whose purpose is to recognize and generate the programs. Grammar does not rely on the computation model instead used in the description of the structure of the language. The grammar contains a finite set of grammatical categories (such as noun phrase, verb phrase, article, noun, etc), solitary words (elements of the alphabets) and the well-formed rules to specify the order within which components of the grammatical categories should appear.

Syntax analysis is a task performed by a compiler which examines whether the program has a proper associated derivation tree or not.

The syntax of a programming language can be interpreted using the following formal and informal techniques:

- **Lexical syntax** for defining the rules for basic symbols involving identifiers, literals, punctuators and operators.
- **Concrete syntax** specifies the real representation of the programs with the help of lexical symbols like its alphabet.
- **Abstract syntax** conveys only the vital program information.

Types of grammars

- **Context-free grammar** is prevalently used to figure out the whole language structure.
- **Regular expressions** describe the lexical units (tokens) of a programming language.
- **Attribute grammars** specify the context-sensitive part of the language.

Definition of Semantics

Semantics term in a programming language is used to figure out the relationship among the syntax and the model of computation. It emphasizes the interpretation of a program so that the programmer could understand it in an easy way or predict the outcome of program execution. An approach known as **syntax-directed semantics** is used to map syntactical constructs to the computational model with the help of a function.

The programming language semantics can be described by the various techniques – Algebraic semantics, Axiomatic semantics, Operational semantics, Denotational semantics, and Translation semantics.

- **Algebraic semantics** interprets the program by defining an algebra.
- **Axiomatic semantics** determine the meaning of a program by building assertions about an association that detain at each point in the execution of the program (i.e. implicitly).
- **Operational semantics** compares the languages to the abstract machine, and the program is then evaluated as a sequence of the state transitions.
- **Denotational semantics** expresses the meaning of the program in the form of a set of functions operating on the program state.
- **Translational semantics** focuses on the methods used for translating a program into another language.

CHARACTERISTICS OF GOOD PROGRAMMING LANGUAGE:-

Characteristics Of A Good Programming Language?

There are some popular high-level programming languages,while there are others that could not become so popular in-spite of being very powerful.There might be reasons for the success of a language but one obvious reason is its characteristics.Several characteristics believed to be important for making it good:

A good programming language must be simple and easy to learn and use.It should provide a programmer with a clear,simple and unified set of concepts that can be grasped easily.The overall simplicity of a this strongly affects the readability of the programs written in that language and programs that are easier to read and understand are easier to maintain.It is also easy to develop and implement a compiler or an interpreter for a simple language.However,the power needed for the language should not be sacrificed for simplicity.For Example,BASIC is liked by many programmers because of its simplicity.

1-Naturalness:

A good language should be natural for the application area for which it is designed.That is,it should provide appropriate operators,data structures,control structures and a natural syntax to facilitate programmers to code their problems easily and efficiently.FORTRAN and COBOL are good examples of languages possessing high degree of naturalness in scientific and business application areas,respectively.

2-Abstraction:
Abstraction means ability to define and then use complicated structures or operations in ways that allow many of the details to be ignored.The degree of abstraction allowed by a language directly affects its ease of programming.For Example,object-oriented languages support high degree of abstraction.Hence,writing programs in object-oriented languages is much easier.Object-oriented also support re usability of program segments due to this feature.

3-Efficiency:
Programs written in a good language are translated into machine code efficiently,are executed and require relatively less space in memory.That is,a good programming language is supported with a good language translator (a compiler or an interpreter) that gives due consideration to space and time efficiency.

4-Structured Programming Support:
A good language should have necessary features to allow programmers to write their programs based on the concepts of structured programming.This property greatly affects the ease with which a program may be written.,tested and maintained.More over,it forces a programmer to look at a problem in a logical way so that fewer errors are created while writing a program for the problem.

5-Compactness:
In a good language,programmers should be able to express the intended operations concisely without losing readability.Programmers generally do not like a verbose language because they need to write too much.Many programmers dislike COBOL,because it is verbose in nature (Lacks Compactness)

6-Locality:
A good language should be such that while writing a program,a programmer need not jump around the visually as the text of a program is prepared.This allows the programmer to concentrate almost solely on the part of the program around the statement currently being worked with.COBOL and to some extent C and Pascal lack locality because data definitions are separated from processing statements,perhaps by many pages of code,or have to appear before any processing statement in the function/procedure.

7-Extensibility:
A good language should also allow extensions through a simply,natural and elegant mechanism.Almost all languages provide subprogram definition mechanisms for the purpose,but some languages are weak in this aspect.

8-Suitability to its Environment:
Depending upon the type of application for which a programming language has been designed,the language must also be made suitable to its environment.For Example, a language designed for a real-time applications must be interactive in nature.On the other hand,languages used for data-processing jobs like payroll,stores accounting etc may be designed to operative in batch mode.

Programming language translators compiler & interpreters

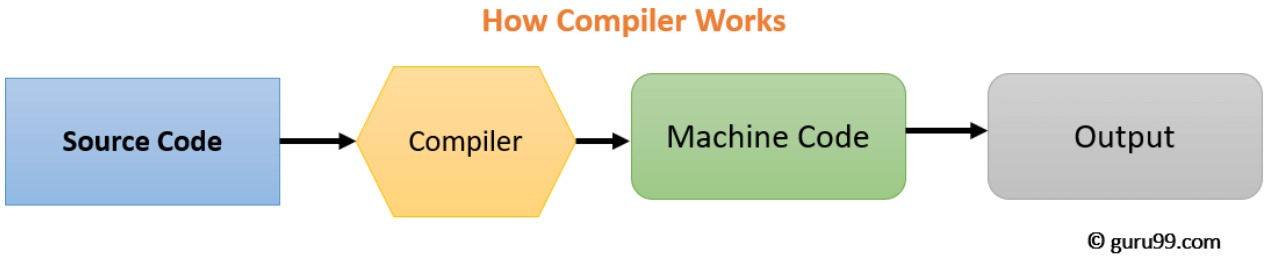
What is Compiler?

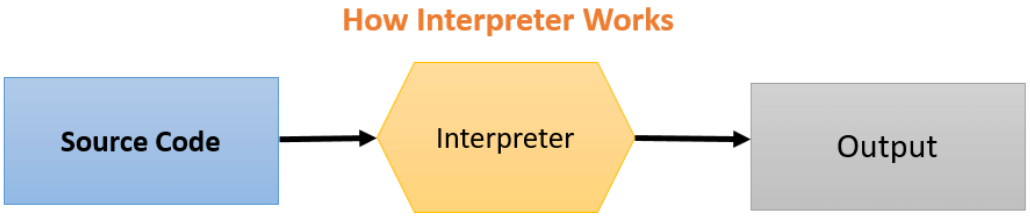
A compiler is a computer program that transforms code written in a high-level programming language into the machine code. It is a program which translates the human-readable code to a language a computer processor understands (binary 1 and 0 bits). The computer processes the machine code to perform the corresponding tasks.

A compiler should comply with the syntax rule of that programming language in which it is written. However, the compiler is only a program and cannot fix errors found in that program. So, if you make a mistake, you need to make changes in the syntax of your program. Otherwise, it will not compile.

What is Interpreter?

An interpreter is a computer program, which coverts each high-level program statement into the machine code. This includes source code, pre-compiled code, and scripts. Both compiler and interpreters do the same job which is converting higher level programming language to machine code. However, a compiler will convert the code into machine code (create an exe) before program run. Interpreters convert code into machine code when the program is run.





Difference Between Compiler and Interpreter

Basis of difference	Compiler	Interpreter
Programming Steps	• Create the program. • Compile will parse or analyses all of the language statements for its correctness. If incorrect, throws an error • If no error, the compiler will convert source code to machine code. • It links different code files into a runnable program(know as exe) • Run the Program	• Create the Program • No linking of files or machine code generation • Source statements executed line by line DURING Execution
Advantage	The program code is already translated into machine code. Thus, it code execution time is less.	Interpreters are easier to use, especially for beginners.

Basis of difference	Compiler	Interpreter
Disadvantage	You can't change the program without going back to the source code.	Interpreted programs can run on computers that have the corresponding interpreter.
Machine code	Store machine language as machine code on the disk	Not saving machine code at all.
Running time	Compiled code run faster	Interpreted code run slower
Model	It is based on language translationlinking-loading model.	It is based on Interpretation Method.
Program generation	Generates output program (in the form of exe) which can be run independently from the original program.	Do not generate output program. So they evaluate the source program at every time during execution.
Execution	Program execution is separate from the compilation. It performed only after the entire output program is compiled.	Program Execution is a part ofInterpretation process, so it is performed line by line.
Memory requirement	Target program executeindependently and do not require the compiler in the memory.	The interpreter exists in the memory during interpretation.
Best suited for	Bounded to the specific target machine and cannot be ported. C and C++ are a most popular a programming language which uses compilation model.	For web environments, where load times are important. Due to all the exhaustive analysis is done, compiles take relatively larger time to compile even small code that may not be run multiple times. In such cases, interpreters are better.
Code Optimization	The compiler sees the entire code upfront. Hence, they perform lots of	Interpreters see code line by line, and thus optimizations are not as robust as compilers

Basis of difference	Compiler	Interpreter
	optimizations that make code run faster	
Dynamic Typing	Difficult to implement as compilers cannot predict what happens at turn time.	Interpreted languages support Dynamic Typing
Usage	It is best suited for the Production Environment	It is best suited for the program and developmentenvironment.
Error execution	Compiler displays all errors and warning at the compilation time. Therefore, you can't run the program without fixing errors	The interpreter reads a single statement and shows the error if any. You must correct the error to interpret next line.
Input	It takes an entire program	It takes a single line of code.
Output	Compliers generates intermediate machnie code.	Interpreter never generate any intermediate machnie code.
Errors	Display all errors after, compilation, all at the same time.	Displays all errors of each line one by one.
Pertaining Programming languages	C,C++,C#, Scala, Java all use complier.	PHP, Perl, Ruby uses an interpreter.

Role of Compiler

- Compliers reads the source code, outputs executable code
- Translates software written in a higher-level language into instructions that computer can understand. It converts the text that a programmer writes into a format the CPU can understand.

- The process of compilation is relatively complicated. It spends a lot of time analyzing and processing the program.
- The executable result is some form of machine-specific binary code.

Role of Interpreter

- The interpreter converts the source code line-by-line during RUN Time.
- Interpret completely translates a program written in a high-level language into machine level language.
- Interpreter allows evaluation and modification of the program while it is executing.
- Relatively less time spent for analyzing and processing the program
- Program execution is relatively slow compared to compiler

HIGH-LEVEL LANGUAGES

High-level languages, like C, C++, JAVA, etc., are very near to English. It makes programming process easy. However, it must be translated into machine language before execution. This translation process is either conducted by either a compiler or an interpreter. Also known as source code.

MACHINE CODE

Machine languages are very close to the hardware. Every computer has its machine language. A machine language programs are made up of series of binary pattern. (Eg. 110110) It represents the simple operations which should be performed by the computer. Machine language programs are executable so that they can be run directly.

OBJECT CODE

On compilation of source code, the machine code generated for different processors like Intel, AMD, an ARM is different. tTo make code portable, the source code is first converted to Object Code. It is an intermediary code (similar to machine code) that no processor will understand. At run time, the object code is converted to the machine code of the underlying platform.

Java is both Compiled and Interpreted.

To exploit relative advantages of compilers are interpreters some programming language like Java are both compiled and interpreted. The Java code itself is compiled into Object Code. At run time, the JVM interprets the Object code into machine code of the target computer.

KEY DIFFERENCE

- Compiler transforms code written in a high-level programming language into the machine code, at once, before program runs, whereas an Interpreter coverts each high-level program statement, one by one, into the machine code, during program run.
- Compiled code runs faster while interpreted code runs slower.
- Compiler displays all errors after compilation, on the other hand, the Interpreter displays errors of each line one by one.
- Compiler is based on translation linking-loading model, whereas Interpreter is based on Interpretation Method.
- Compiler takes an entire program whereas the Interpreter takes a single line of code.

VIRTUAL COMPUTERS & BINDING TIMES

Binding Time

As we have just seen, operating systems use various kinds of names to refer to objects. Sometimes the mapping between a name and an object is fixed, but sometimes it is not. In the latter case, it may matter when the name is bound to the object. In general, **early binding** is simple, but is not flexible, whereas **late binding** is more complicated but often more flexible.

To clarify the concept of binding time, let us look at some real-world examples. An example of early binding is the practice of some colleges to allow parents to enroll a baby at birth and prepay the current tuition. When the student shows up 18 years later, the tuition is fully paid up, no matter how high it may be at that moment.

In manufacturing, ordering parts in advance and maintaining an inventory of them is early binding. In contrast, just-in-time manufacturing requires suppliers to be able to provide parts on the spot, with no advance notice required. This is late binding.

Programming languages often support multiple binding times for variables. Global variables are bound to a particular virtual address by the compiler. This exemplifies early binding. Variables local to a procedure are assigned a virtual address (on the stack) at the time the procedure is invoked. This is intermediate binding. Variables stored on the heap (those allocated by *malloc* in C or *new* in Java) are assigned virtual addresses only at the time they are actually used. Here we have late binding.

Operating systems often use early binding for most data structures, but occasionally use late binding for flexibility. Memory allocation is a case in point. Early multiprogramming systems on machines lacking address relocation hardware had to load a program at some memory address and relocate it to run there. If it was ever swapped out, it had to be brought back at the same

memory address or it would fail. In contrast, paged virtual memory is a form of late binding. The actual physical address corresponding to a given virtual address is not known until the page is touched and actually brought into memory.

Another example of late binding is window placement in a GUI.

INTRODUCTION TO PROCEDURAL, NON-PROCEDURAL:-

Difference between Procedural and Non-Procedural language

Procedural Language:
In procedural languages, the program code is written as a sequence of instructions. User has to specify "what to do" and also "how to do" (step by step procedure). These instructions are executed in the sequential order. These instructions are written to solve specific problems.

Examples of Procedural languages:
FORTRAN, COBOL, ALGOL, BASIC, C and Pascal.

Non-Procedural Language:
In the non-procedural languages, the user has to specify only "what to do" and not "how to do". It is also known as an applicative or functional language. It involves the development of the functions from other functions to construct more complex functions.

Examples of Non-Procedural languages:
SQL, PROLOG, LISP.

Difference between Procedural and Non-Procedural language:

PROCEDURAL LANGUAGE	NON-PROCEDURAL LANGUAGE
It is command-driven language.	It is a function-driven language
It works through the state of machine.	It works through the mathematical functions.

PROCEDURAL LANGUAGE	NON-PROCEDURAL LANGUAGE
Its semantics are quite tough.	Its semantics are very simple.
It returns only restricted data types and allowed values.	It can return any datatype or value
Overall efficiency is very high.	Overall efficiency is low as compared to Procedural Language.
Size of the program written in Procedural language is large.	Size of the Non-Procedural language programs are small.
It is not suitable for time critical applications.	It is suitable for time critical applications.
Iterative loops and Recursive calls both are used in the Procedural languages.	Recursive calls are used in Non-Procedural languages.

STRUCTURED, FUNCTIONAL AND OBJECT ORIENTED PROGRAMMING LANGUAGE:-

the most basic difference between object oriented and structured programming language is the capabilty of certain language to follow the object oriented principles using various syntax and methodology while procedure or structured language though

have certain capability to follow the oops principle but it takes a lot of line of codes as well as combination of basic syntax to implement such.

The OOPS oriented language such as Java follow object oriented principles but can not be said as fully OOPS oriented.

whereas python is fully oop oriented and c is procedure oriented

Structured Programming

1. Structured Programming is designed which focuses on **process.**
2. Structured programming follows **top-down approach**.
3. In Structured Programming, Programs are divided into small self contained **functions**
4. Structured Programming provides **less reusability**, more function dependency.
5. Less abstraction and less flexibility.

Object Oriented Programming

1. Object Oriented Programming is designed which focuses on **data**.
2. Object oriented programming follows **bottom-up approach**.
3. In Object Oriented Programming, Programs are divided into small entities called **objects**
4. Object Oriented Programming provides more reusability, less function **dependency**.
5. More abstraction and more **flexibility**.

Structured Programming is designed which focuses on **process**/ logical structure and then data required for that process.

Object Oriented Programming is designed which focuses on **data**.

Structured programming follows **top-down approach**.

Object oriented programming follows **bottom-up approach**.

Structured Programming is also known as **Modular Programming** and a subset of **procedural programming language**.

Object Oriented Programming supports **inheritance, encapsulation, abstraction**, **polymorphism**, etc.

In Structured Programming, Programs are divided into small self contained **functions**.

In Object Oriented Programming, Programs are divided into small entities called **objects**.

Structured Programming is **less** secure as there is no way of **data hiding**.

Object Oriented Programming is more secure as having data hiding feature.

Structured Programming can solve **moderately** complex programs.

Object Oriented Programming can solve any **complex** programs.

Structured Programming provides **less reusability**, more function dependency.

Object Oriented Programming provides more reusability, less function **dependency**.

Less abstraction and less flexibility.

More abstraction and more **flexibility**.

COMPARISON OF C & C++ PROGRAMMING LANGUAGES.

As we know both C and C++ are programming languages and used for application development. The main difference between both these languages is C is a procedural programming language and does not support classes and objects, while C++ is a combination of both procedural and object-oriented programming languages.

The following are the important differences between C and C++.

Sr. No.	Key	C	C++
1	Introduction	C was developed by Dennis Ritchie in around 1969 at	C++ was developed by Bjarne Stroustrup in 1979.

Sr. No.	Key	C	C++
		AT&T Bell Labs.	
2	Language Type	As mentioned before C is procedural programming.	On the other hand, C++ supports both procedural and object-oriented programming paradigms.
3	OOPs feature Support	As C does not support the OOPs concept so it has no support for polymorphism, encapsulation, and inheritance.	C++ has support for polymorphism, encapsulation, and inheritance as it is being an object-oriented programming language
4	Data Security	As C does not support encapsulation so data behave as a free entity and can be manipulated by outside code.	On another hand in the case of C++ encapsulation hides the data to ensure that data structures and operators are used as intended.
5	Driven type	C in general known as function-driven language.	On the other hand, C++ is known as object driven language.
6	Feature supported	C does not support function and operator overloading also do not have namespace feature and reference variable functionality.	On the other hand, C++ supports both function and operator overloading also have namespace feature and reference variable functionality.

UNIT-2

Elementary data types

Basic differences among programming languages:

- types of data allowed
- types of operations available
- mechanisms for controlling the sequence of operations

Elementary data types: built upon the available hardware features
Structured data types: software simulated

1. **Data objects, variables, and constants**

1. 1. Data object:
a run-time grouping of one or more pieces of data in a virtual computer.
a location in memory with an assigned name in the actual computer.

Types of data objects:

- Programmer defined data objects - variables, arrays, constants, files, etc.
- System defined data objects - set up for housekeeping during program execution, not directly accessible by the program. E.g. run-time storage stacks.

Data value: a bit pattern that is recognized by the computer.

Elementary data object: contains a data value that is manipulated as a unit.
Data structure: a combination of data objects.

Attributes: determine how the location may be used. Most important attribute - the data type.

Attributes and Bindings

- **Type:** determines the set of data values that the object may take and the applicable operations.
- **Name:** the binding of a name to a data object.
- **Component:** the binding of a data object to one or more data objects.
- **Location:** the storage location in memory assigned by the system.
- **Value:** the assignment of a bit pattern to a name.

Type, *name* and *component* are bound at translation, *location* is bound at loading, *value* is bound at execution

1. 2. Data objects in programs

In programs, data objects are represented as variables and constants

Variables: Data objects defined and named by the programmer explicitly.

Constants: a data object with a name that is permanently bound to a value for its lifetime.

- **Literals:** constants whose name is the written representation of their value.
- **A programmer-defined constant**: the name is chosen by the programmer in a definition of the data object.

1. 4. Persistence Data objects are created and exist during the execution of the program. Some data objects exist only while the program is running. They are called **transient data objects**. Other data objects continue to exist after the program terminates, e.g. data files. They are called **persistent data objects.** In certain applications, e.g. transaction-based systems the data and the programs coexist practically indefinitely, and they need a mechanism to indicate that an object is persistent. Languages that provide such mechanisms are called **persistent languages**.

2. Data types

A data type is a class of data objects with a set of operations for creating and manipulating them.

Examples of elementary data types:

integer, real, character, Boolean, enumeration, pointer.

2. 1. Specification of elementary data types

1. **Attributes** that distinguish data objects of that type

Data type, name - invariant during the lifetime of the object

- stored in a descriptor and used during the program execution
- used only to determine the storage representation, not used explicitly during execution

2. **Values** that data object of that type may have

   Determined by the type of the object

   Usually an ordered set, i.e. it has a least and a greatest value

3. **Operations** that define the possible manipulations of data objects of that type.

Primitive - specified as part of the language definition
Programmer-defined (as subprograms, or class methods)

An operation is defined by:

- Domain - set of possible input arguments
- Range - set of possible results
- Action - how the result is produced

The domain and the range are specified by the **operation signature**

- the number, order, and data types of the arguments in the domain,
- the number, order, and data type of the resulting range

mathematical notation for the specification:

op name: arg type x arg type x … x arg type ® result type

The action is specified in the operation implementation

Sources of ambiguity in the definition of programming language operations

- Operations that are undefined for certain inputs.
- Implicit arguments, e.g. use of global variables
- Implicit results - the operation may modify its arguments

(HW 01 - the value of **a** changed in x = a + b)

- Self-modification - usually through change of local data between calls,

i.e. random number generators change the seed.

Subtypes : a data type that is part of a larger class.
Examples: in C, C++ int, short, long and char are variations of integers.
The operations available to the larger class are available to the subtype.
This can be implemented using inheritance.

2. 2. Implementation of a data type

4. Storage representation

Influenced by the hardware

Described in terms of:

Size of the memory blocks required
Layout of attributes and data values within the block

Two methods to treat attributes:

a. determined by the compiler and not stored in descriptors during execution - C
b. stored in a descriptor as part of the data object at run time - LISP Prolog

5. Implementation of operations

- Directly as a hardware operation. E.g. integer addition

- Subprogram/function, e.g. square root operation
- In-line code. Instead of using a subprogram, the code is copied into the program at the point where the subprogram would have been invoked.

Declarations

Declarations provide information about the name and type of data objects needed during program execution.

- Explicit – programmer defined
- Implicit – system defined

e.g. in FORTRAN - the first letter in the name of the variable determines the type
Perl - the variable is declared by assigning a value
$abc = 'a string' $abc is a string variable
$abc = 7 $abc is an integer variable

Operation declarations: prototypes of the functions or subroutines that are programmer-defined.
Examples:
declaration: **float Sub(int, float)**
signature: **Sub: int x float --> float**

Purpose of declaration

- Choice of storage representation
- Storage management
- Declaration determines the lifetime of a variable, and allowes for more efficient memory usage.
- Specifying polymorphic operations.

Depending on the data types operations having same name may have different meaning, e.g. integer addition and float addition
In most language +, -. *, / are overloaded
Ada - aloows the programmer to overload subprograms
ML - full polymorphism

Declarations provide for static type checking

Type checking and type conversion

Type checking: checking that each operation executed by a program receives the proper number of arguments of the proper data types.
Static type checking is done at compilation.
Dynamic type checking is done at run-time.
Dynamic type checking – Perl and Prolog
Implemented by storing a type tag in each data object

Advantages: Flexibility

Disadvantages:

- Difficult to debug
- Type information must be kept during execution
- Software implementation required as most hardware does not provide support

Concern for static type checking affects language aspects:

Declarations, data-control structures, provisions for separate compilation of subprograms

Strong typing: all type errors can be statically checked

Type inference: implicit data types, used if the interpretation is unambiguous. Used in ML

Type Conversion and Coercion

Explicit type conversion : routines to change from one data type to another.

Pascal: the function round - converts a real type into integer

C - cast, e.g. (int)X for float X converts the value of X to type integer

Coercion: implicit type conversion, performed by the system.

Pascal: + integer and real, integer is converted to real

Java - permits implicit coercions if the operation is widening

C++ - and explicit cast must be given.

Two opposite approaches to type coercions:

- No coercions, any type mismatch is considered an error : Pascal, Ada
- Coercions are the rule. Only if no conversion is possible, error is reported.

Advantages of coercions: free the programmer from some low level concerns, as adding real numbers and integers.

Disadvantages: may hide serious programming errors.

Assignment and Initialization

Different ways to initialize a variable in C/C++

Variables are arbitrary names given to a memory location in the system. These memory locations addresses in the memory. Suppose we want to save our marks in memory. Now, these marks will get saved at a particular address in the memory. Now, whenever these marks will be updated, they will be stored at a different memory address. Thus, to facilitate the fetching of these memory addresses, variables are used. Variables are names given to these memory locations. The memory location referred to by this variable holds a value of our interest. Now, these variables once declared, are assigned some value. This assignment of value to these variables is called initialization of variables.

Initialization of a variable is of two types:

- **Static Initialization:** Here, the variable is assigned a value in advance. This variable then acts as a constant.
- **Dynamic Initialization:** Here, the variable is assigned a value at the run time. The value of this variable can be altered every time the program is being run.

Different ways of initializing a variable in C

Method 1 (Declaring the variable and then initializing it)

```
int a;
a = 5;
```

Method 2 (Declaring and Initializing the variable together):

```
int a = 5;
```

Method 3 (Declaring multiple variables simultaneously and then initializing them separately)

```
int a, b;
a = 5;
b = 10;
```

Method 4 (Declaring multiple variables simultaneously and then initializing them simultaneously)

```
int a, b;
a = b = 10;
int a, b = 10, c = 20;
```

Method 5 (Dynamic Initialization : Value is being assigned to variable at run time.)

```
int a;
printf("Enter the value of a");
scanf("%d", &a);
```

Different ways of initializing a variable in C++

Method 1 (Declaring and Initializing a variable)

```
int a = 5;
```

Method 2 (Initializing a variable using parenthesis)

```
int a (5) ;
```

Yes, they're the same. On the other hand, for a class type they're different. For example :

```
struct A {
   A(int);
};
A a(5);
// This statement is to construct a;
```

Method 3 (Initializing a variable using braces)

```
int a{5} ;
```

Method 4 (Declaring a variable using auto class)

```
auto a = 5;
```

'auto' is a keyword which tells the compiler the type of the variable upon its initialization.

Method 5 (Declaring and Initializing a variable through 'auto' keyword with parenthesis)

```
auto a (5);
```

Method 6 (Declaring and Initializing a variable through 'auto' keyword with braces)

```
auto a{5};
```

Method 7 (Dynamic Initialization)

```
int a;
cin>>a;
```

These are all the different ways in which a variable can be defined in C or C++. The ways are similar for all fundamental variable but the way to initialize a variable of derived data type changes accordingly. Different derived data types have an altogether different way of getting their variable initialized and hence can be explored in detail while diving in the all about of that particular data type.

Numeric Data Types

Numeric data types are numbers stored in database columns. These data types are typically grouped by:

- **Exact** numeric types, values where the precision and scale need to be preserved. The exact numeric types are INTEGER, BIGINT, DECIMAL, NUMERIC, NUMBER, and MONEY.

- **Approximate** numeric types, values where the precision needs to be preserved and the scale can be floating. The approximate numeric types are DOUBLE PRECISION, FLOAT, and REAL.

Implicit casts from INTEGER, FLOAT, and NUMERIC to VARCHAR are not supported. If you need that functionality, write an explicit cast using one of the following forms:

CAST(*numeric-expression* AS *data-type*)
numeric-expression::*data-type*
For example, you can cast a float to an integer as follows:

```
=> SELECT(FLOAT '123.5')::INT;
 ?column?
----------
      124
(1 row)
```

String-to-numeric data type conversions accept formats of quoted constants for scientific notation, binary scaling, hexadecimal, and combinations of numeric-type literals:

- Scientific notation:

- => SELECT FLOAT '1e10';
- ?column?
- ----------
- 10000000000
- (1 row)
- BINARY scaling:

- => SELECT NUMERIC '1p10';
- ?column?
- ----------
- 1024
- (1 row)
- hexadecimal:

- => SELECT NUMERIC '0x0abc';
- ?column?

- ----------
- 2748

(1 row)

Enumeration

.

An **enumeration** is a complete, ordered listing of all the items in a collection. The term is commonly used in mathematics and computer science to refer to a listing of all of the elements of a set. The precise requirements for an enumeration (for example, whether the set must be finite, or whether the list is allowed to contain repetitions) depend on the discipline of study and the context of a given problem.

Some sets can be enumerated by means of a **natural ordering** (such as 1, 2, 3, 4, ... for the set of positive integers), but in other cases it may be necessary to impose a (perhaps arbitrary) ordering. In some contexts, such as enumerative combinatorics, the term *enumeration* is used more in the sense of *counting* – with emphasis on determination of the number of elements that a set contains, rather than the production of an explicit listing of those elements.

Combinatorics

In combinatorics, enumeration means counting, i.e., determining the exact number of elements of finite sets, usually grouped into infinite families, such as the family of sets each consisting of all permutations of some finite set. There are flourishing subareas in many branches of mathematics concerned with enumerating in this sense objects of special kinds. For instance, in *partition enumeration* and *graph enumeration* the objective is to count partitions or graphs that meet certain conditions.

Listing

When an enumeration is used in an ordered list context, we impose some sort of ordering structure requirement on the index set. While we can make the requirements on the ordering quite lax in order to allow for great generality, the most natural and common prerequisite is that the index set be well-ordered. According to this characterization, an ordered enumeration is defined to be a surjection (an onto relationship) with a well-ordered domain. This definition is natural in the sense that a given well-ordering on the index set provides a unique way to list the next element given a partial enumeration.

Countable vs. uncountable

The most common use of enumeration in set theory occurs in the context where infinite sets are separated into those that are countable and those that are not. In this case, an enumeration is merely an enumeration with domain ω, the ordinal of the natural numbers. This definition can also be stated as follows:

- As a surjective mapping from (the natural numbers) to *S* (i.e., every element of *S* is the image of at least one natural number). This definition is especially suitable to questions of computability and elementary set theory.

We may also define it differently when working with finite sets. In this case an enumeration may be defined as follows:

- As a bijective mapping from *S* to an initial segment of the natural numbers. This definition is especially suitable to combinatorial questions and finite sets; then the initial segment is {1,2,...,*n*} for some *n* which is the cardinality of *S*.

In the first definition it varies whether the mapping is also required to be injective (i.e., every element of *S* is the image of *exactly one* natural number), and/or allowed to be partial (i.e., the mapping is defined only for some natural numbers). In some applications (especially those concerned with computability of the set *S*), these differences are of little importance, because one is concerned only with the mere existence of some enumeration, and an enumeration according to a liberal definition will generally imply that enumerations satisfying stricter requirements also exist.

Enumeration of finite sets obviously requires that either non-injectivity or partiality is accepted, and in contexts where finite sets may appear one or both of these are inevitably present.

Boolean data type

In computer science, the **Boolean data type** is a data type that has one of two possible values (usually denoted *true* and *false*) which is intended to represent the two truth values of logic and Boolean algebra. It is named after George Boole, who first defined an algebraic system of logic in the mid 19th century. The Boolean data type is primarily associated with conditional statements, which allow different actions by changing control flow depending on whether a programmer-specified Boolean *condition* evaluates to true or false. It is a special case of a more general *logical data type (see probabilistic logic)*—logic doesn't always need to be Boolean.

In programming languages with a built-in Boolean data type, such as Pascal and Java, the comparison operators such as `>` and `≠` are usually defined to return a Boolean value. Conditional and iterative commands may be defined to test Boolean-valued expressions.

Languages with no explicit Boolean data type, like C90 and Lisp, may still represent truth values by some other data type. Common Lisp uses an empty list for false, and any other value for true. The C programming language uses an integer type, where relational expressions like `i > j` and logical expressions connected by `&&` and `||` are defined to have value 1 if true and 0 if false, whereas the test parts of `if`, `while`, `for`, etc., treat any non-zero value as true.[1][2] Indeed, a Boolean variable may be regarded (and implemented) as a numerical variable with one binary digit (bit), which can store only two values. The implementation of Booleans in computers are most likely represented as a full word, rather than a bit; this is usually due to the ways computers transfer blocks of information.

Most programming languages, even those with no explicit Boolean type, have support for Boolean algebraic operations such as conjunction (AND, &, *), disjunction (OR, |, +), equivalence (EQV, =, ==), exclusive or/non-equivalence (XOR, NEQV, ^, !=), and negation (NOT, ~, !).

In some languages, like Ruby, Smalltalk, and Alice the *true* and *false* values belong to separate classes, i.e., True and False, respectively, so there is no one Boolean *type*.

In SQL, which uses a three-valued logic for explicit comparisons because of its special treatment of Nulls, the Boolean data type (introduced in SQL:1999) is also defined to include more than two truth values, so that SQL *Booleans* can store all logical values resulting from the evaluation of predicates in SQL. A column of Boolean type can also be restricted to just TRUE and FALSE though.

SEQUENCE CONTROL

SEQUENCE CONTROL

Control Structure in a PL provides the basic framework within which operations and data are combined into a program and sets of programs.

Sequence Control -> Control of the order of execution of the operations Data Control -> Control of transmission of data among subprograms of program

Sequence Control may be categorized into four groups:

1) Expressions – They form the building blocks for statements.

An expression is a combination of variable constants and operators according to syntax of language. Properties as precedence rules and parentheses determine how expressions are evaluated

1) Statements – The statements (conditional & iterative) determine how control flows from one part of program to another.

2) Declarative Programming – This is an execution model of program which is independent of the program statements.

Logic programming model of PROLOG.

3) Subprograms – In structured programming, program is divided into small sections and each section is called subprogram. Subprogram calls and co- routines, can be invoked repeatedly and transfer control from one part of program to another.

IMPLICIT AND EXPLICIT SEQUENCE CONTROL

Implicit Sequence Control

Implicit or default sequence-control structures are those defined by the programming language itself. These structures can be modified explicitly by the programmer.

eg. Most languages define physical sequence as the sequence in which statements are executed.

Explicit Sequence Control

Explicit sequence-control structures are those that programmer may optionally use to modify the implicit sequence of operations defined by the language.

eg. Use parentheses within expressions, or goto statements and labels

Sequence Control Within Expressions

Expression is a formula which uses operators and operands to give the output value.

i) Arithmetic Expression –

An expression consisting of numerical values (any number, variable or function call) together with some arithmetic operator is called “Arithmetic Expression”.

Evaluation of Arithmetic Expression

Arithmetic Expressions are evaluated from left to right and using the rules of precedence of operators.
If expression involves parentheses, the expression inside parentheses is evaluated first

ii) Relational Expressions –

An expression involving a relational operator is known as “Relational Expression”. A relational expression can be defined as a meaningful combination of operands and relational operators.

(a + b) > c c < b

Evaluation of Relational Expression

The relational operators <, >, <=, >= are given the first priority and other operators (== and !=) are given the second priority

The arithmetic operators have higher priority over relational operators. The resulting expression will be of integer type, true = 1, false = 0

Sequence Control Within Expressions

iii) Logical Expression –

An expression involving logical operators is called ‘Logical expression”. The expression formed with two or more relational expression is called logical expression.

Ex. a > b && b < c Evaluation of Logical Expression

The result of a logical expression is either true or false.

For expression involving AND (&&), OR (||) and NOT(!) operations, expression involving NOT is evaluated first, then the expression with AND and finally the expression having OR is evaluated.

Sequence Control Within Expressions

1. Controlling the evaluation of expressions

a) Precedence (Priority)

If expression involving more than one operator is evaluated, the operator at higher level of precedence is evaluated first

b) Associativity

The operators of the same precedence are evaluated either from left to right or from right to left depending on the level

Most operators are evaluated from left to right except

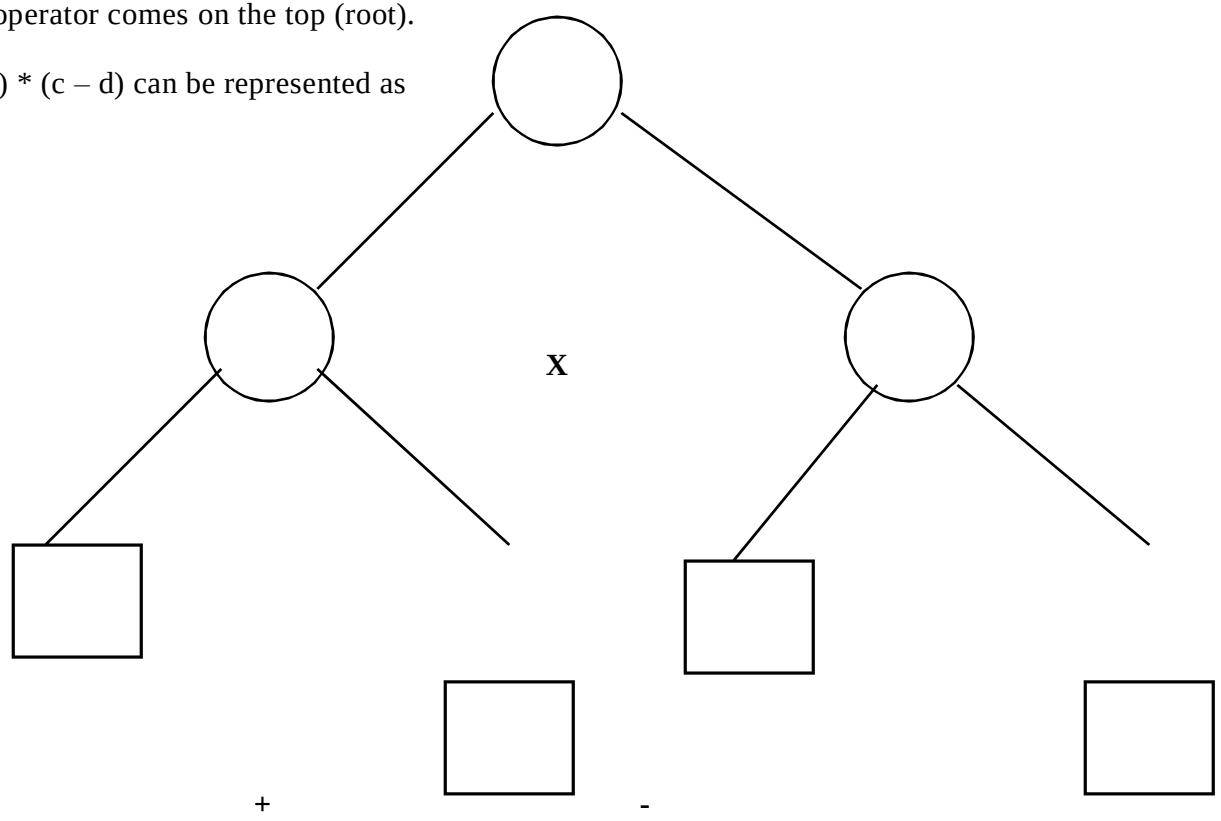
+ (unary plus),	- (unary minus)	++, --, !,
& Assignment operators	= , +=, *=, /=, %=	

Sequence Control Within Expressions

2. Expression Tree

An expression (Arithmetic, relational or logical) can be represented in the form of an "expression tree". The last or main operator comes on the top (root).

Example: (a + b) * (c – d) can be represented as



Sequence Control Within Expressions

a

c

b

d

Sequence Control Within Expressions

3. Syntax for Expressions

a) Prefix or Polish notation

Named after polish mathematician Jan Lukasiewicz, refers to notation in which operator symbol is placed before its operands.

*XY, -AB, /*ab-cd

Cambridge Polish - variant of notation used in LISP, parentheses surround an operator and its arguments.

(/(*ab)(-cd))

b) Postfix or reverse polish

Postfix refers to notation in which the operator symbol is placed after its two operands.

AB*, XY-

c) Infix notation

It is most suitable for binary (dyadic) operation. The operator symbol is placed between the two operands.

Sequence Control Within Expressions

4. Semantics for Expressions

Semantics determine the order of expression in which they are evaluated.

a) Evaluation of Prefix Expression

If P is an expression evaluate using stack

i) If the next item in P is an operator, push it on the stack. set the arguments count to be number of operands needed by operator.

(if number is n, operator is n-ary operator).

ii) If the next item in P is an operand, push it on the stack

iii) If the top n entries in the stack are operand entries needed for the last n-ary operator on the stack, apply the operator on those operands. Replace the operator and its n operands by the result of applying that operation on the n operands.

b) Evaluation of Postfix Expression

If P is an expression evaluate using stack

i) If the next item in P is an operand, push it on the stack.

ii) If the next item in P is an n-ary operator, its n arguments must be top n items on the stack. Replace these n items by the result of applying this operation using the n items as arguments.

Sequence Control Within Expressions

c) Evaluation of Infix Expression

Infix notation is common but its use in expression cause the problems:

i) Infix notation is suitable only for binary operations. A language cannot use only infix notation but must combine infix and postfix (or prefix) notations. The mixture makes translation complex.

ii) If more than one infix operator is in an expression, the notation is ambiguous unless parentheses are used.

Sequence Control Within Expressions

5. Execution-Time Representation:

Translators evaluate the expression using a method so as to getefficient result (optimum value at optimum time with optimum use of memory and processor). Translation is done in two phases –

In first phase the basic tree control structure for expression is established. In next stage whole evaluation process takes place.

The following methods are used for translation of expression –

a) Machine code sequences

Expression can be translated into machine code directly performing the two stages (control structure establishment and evaluation) in one step.

The ordering of m/c code instructions reflect the control sequence of original expression.

b) Tree Structure

The expressions may be executed directly in tree structure representation using a software interpreter.

This kind of evaluation used in SW interpreted languages like LISP where programs are represented in the form of tree during execution

c) Prefix or postfix form

Problems with Evaluation of Expressions

1. Uniform Evaluation Code

Eager Evaluation Rule – For each operation node, first evaluate each of the operands, then apply the operation to the evaluated operands.

The order of evaluations shouldn't matter.

In C: A + (B = 0 ? C : C/B) ---------- Problem

Lazy Evaluation Rule – Never evaluate operands before applying the operation. Pass the operands unevaluated and let the operation decide if evaluation is needed.

It is impractical to implement the same in many cases as it requires substantial software simulation.
LISP, Prolog use lazy rule.

In general, implementations use a mixture of two techniques. LISP – functions split into two categories

SNOBOL – programmer-defined operations always receive evaluated operands language-defined operations receive unevaluated operands

2. Side Effects

The use of operations may have side effects inexpressions c / func(y) + c

r-value of c must be fetched and func(y) must be evaluated before division.

If fun(y) has the side effect of modifying the value of c, the order of evaluation is critical.

Problems with Evaluation of Expressions

3. Short-circuit Boolean Expression

If ((X == 0) || (Y/X < Z) { ..... }

do {……} while ((I > UB) && (A[I] < B))

Evaluation of second operand of Boolean expression may lead to an error condition (division by zero, subscript range error).

In C -- The left expression is evaluated first and second expression is evaluated only when needed.

In many languages, both operands are evaluated before boolean expression is Evaluated

ADA includes two special Boolean operations and then , or else

if (X = 0) or else (Y/X > Z) then can't fail

Sequential Control within Statement

i) Assignment Statement

Assignment operator (=), compound assignment operator (+=)

MOVE A TO B. - COBOL

ii) Input and Output Statement

printf, scanf

iii) Declaration Statement

int age;

iv) GoTo statement

Explicit sequence control statement. Used to branch conditionally from one point to another in the program

```
int a, b;


scanf ("%d", &a);

if (a == 0) goto Read; y =
sqrt(x); prinf("%d", y);
Read:   goto Read;
```

Sequential Control within Statement

v) Break Statement

An early exit from a loop can be accomplished by using break statement.

2. Statement Level Sequence Control

i) Implicit Sequence Control

The natural or default programming sequence of a PL is called implicit sequence. They are of 3 types.

a) Composition Type

Standard form of implicit sequence. Statements placed in order of execution.

b) Alternation Type

There are two alternate statement sequence in the program, the program chooses any of the sequence but not both at same type

c) Iteration Type

Here normal sequence is given to statements but the sequence repeats itself for more than one time.

ii) Explicit Sequence Control

The default sequence is altered by some special statements

a) Use of Goto statement b) Use of Break Statement

Sequential Control within Statement

3. Structured Sequence Control

a) Compound Statement

Collection of two or more statements may be treated as single statement. begin /* ----- Pascal { /* C

…….. ……. end

}

b) Conditional Statements

if (conditional exp) then …….statements endif

if (conditional exp) then …….statements else …..statements endif if (conditional exp) then

…….statements

elseif (conditional exp) then … statements else …. statements

…endif

switch (exp) { case val1: …statements break;

val2: ….statetments break; default:

statements break;}

c) Iteration Statements

do {…….} while (conditional exp) while

(conditional exp) {..... }

for (initialization; test condition; increment) { ..... }

Subprogram Sequence Control

Subprogram sequence control is related to concept:

How one subprogram *invokes* another and called subprogram returns to the first.

Simple Call-Return Subprograms

- Program is composed of single main program.
- During execution It calls various subprograms which maycall other subprograms and so on to any depth
- Each subprogram returns the control to the program/subprogram after execution
- The execution of calling program is temporarily stopped during execution of thesubprogram
- After the subprogram is completed, execution of the calling program resumes at the point immediately following thecall

Copy Rule

The effect of the call statement is the same as would be if the call statement is replaced by body of subprogram (with suitable substitution of parameters)

We use subprograms to avoid writing the same structure in program again and again.

Subprogram Sequence Control

Simple Call-Return Subprograms

The followingassumptions are made for simple call return structure

i) Subprogram can not be recursive

ii) Explicit call statements are required

iii) Subprograms must execute completely at call

iv) Immediate transfer of control at point of call orreturn

v) Single execution sequence for each subprogram Implementation

1. There is a distinction between a subprogram definition and subprogram activation.
Subprogram definition – The written program which is translated into a template.
Subprogram activation – Created each time a subprogram is called using the

template created from the definition

2. An activation is implemented as twoparts

Code Segment – contains executable code and constants

Activation record – contains local data, parameters & other dataitems

3. The code segment is invariant during execution. It is created by translator and stored statically in memory. They are never modified. Each activation uses the same code segment.

4. A new activation record is created each time the subprogram is called and is destroyed when the subprogram returns. The contents keep on changing while subprogram is executing

Subprogram Sequence Control

Two system-defined pointer variables keep track of the point at which program is being executed.

Current Instruction Pointer (CIP)

The pointer which points to the instruction in the code segment that is currently being executed (or just about to be)

by the hardware or software interpreter.

Current Environment Pointer (CEP)

Each activation record contains its set of local variables. The activation record represents the "referencing environment" of the subprogram.

The pointer to current activation record is Current Execution Pointer.

Execution of Program

First an activation for the main program is created and CEP is assigned to it. CIP is assigned to a pointer to the first instruction of the code segment for the subprogram.

When a subprogram is called, new assignments are set to the CIP and CEP for the first instruction of the code segment of the subprogram and the activation of the subprogram.

To return correctly from the subprogram, values of CEP and CIP are stored before calling the subprogram. When return instruction is reached, it terminates the activation of subprogram, the old values of CEP and CIP that were saved at the time of subprogram call are retrieved and reinstated.

Recursive Subprograms

Recursive Subprograms

Recursion is a powerful technique for simplifying the design of algorithms.

Recursive subprogram is one that calls itself (directly or indirectly) repeatedly having two properties

a) It has a terminating condition or base criteria for which it doesn't call itself

b) Every time it calls itself, it brings closer to the terminating condition

In Recursive subprogram calls A subprogram may call any other subprogram including A itself, a subprogram B that calls A or so on.

The only difference between a recursive call and an ordinary call is that the recursive call creates a second activation of the subprogram during the lifetime of the first activation.

If execution of program results in chain such that 'k' recursive calls of subprogram occur before any return is made. Thus 'k+1' activation of subprogram exist before the return from k^{th} recursive call.

Both CIP and CEP are used to implement recursive subprogram.

Exception and Exception Handlers

Type of Bugs -

Logic Errors – Errors in program logic due to poor understanding of the problem and solution procedure.
Syntax Errors – Errors arise due to poor understanding of the language.

Exceptions are runtime anomalies or unusual conditions that a program may encounter while executing.
eg. Divide by zero, access to an array out of bounds, running out of memory or disk space

When a program encounters an exceptional condition, it should be Identified and dealt with effectively.

Exception and Exception Handlers

Exception Handling –

It is a mechanism to detect and report an 'exceptional circumstance" so that appropriate action can be taken. It involves the following tasks.

- Find the problem (Hit the exception)
- Inform that an error has occurred (Throw the exception)
- Receive the error information (catch the expression)
- Take corrective action (Handle the exception) main()

```
{ int x, y;
    cout << "Enter values of x and y"; cin >>x>>y;

  try {
        if (x != 0)
            cout "y/x is ="<<y/x; else

            throw(x);
      }
  catch (int i) {
     cout << "Divide by zero exception caught";
    }
}
```

Exception and Exception Handlers

try – Block contains sequence of statements which may generate exception. throw – When an exception is detected, it is thrown using throw statement

catch – It's a block that catches the exception thrown by throw statement and handles it appropriately.

catch block immediately follows the try block.

The same exception may be thrown multiple times in the try block. There may be many different exceptions thrown from the same try block.

There can be multiple catch blocks following the same try block handling different exceptions thrown.

The same block can handle all possible types of exceptions. catch(…)

```
{
        // Statements for processing all exceptions
}
```

Exception and Exception Handlers

```
procedure sub1() divide_zero
   exception;
   wrong_array_sub exception;
 ---------------- other exceptions begin
    ------------
    if x = 0 then raise divide_zero;
    ------------
exception
     when divide_zero =>
           ----------------- handler for divide-zero when array_sub =>
          ------------------ handler for array sub
end;
```

Exception and Exception Handlers

Propagating an Exception –

If an handler for an exception is not defined at the place where an exception occurs then it is propagated so it could be handled in the calling subprogram. If not handled there it is propagated further.

If no subprogram/program provides a handler, the entire program is terminated and standard language-defined handler is invoked.

After an exception is handled –

What to do after exception is handled? Where the control should be transferred?

Should it be transferred at point where exception was raised?

Should control return to statement in subprogram containing handler after it was propagated?

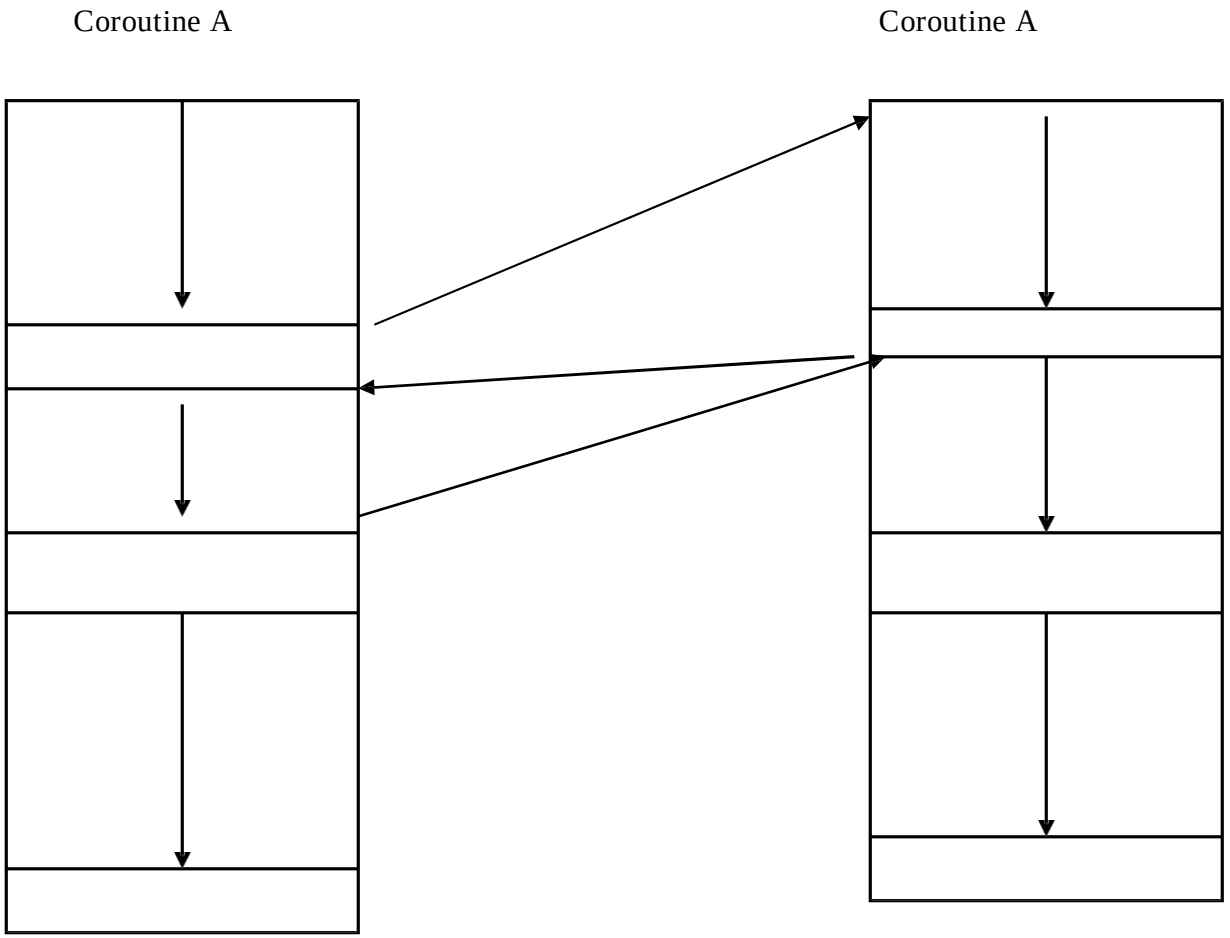
Should subprogram containing the handler be terminated normally and control transferred to calling subprogram? – ADA

Depends on language to language

COROUTINES

COROUTINES –

Coroutines are subprogram components that generalize subroutines to allow multiple entry points and suspending and resuming of execution at certain locations.



COROUTINES

Comparison with Subroutines

1. The lifespan of subroutines is dictated by last in, first out (the last subroutine called is the first to return); lifespan of coroutines is dictated by their use and need,
2. The start of the subroutine is the only point entry. There might be multiple entries in coroutines.
3. The subroutine has to complete execution before it returns the control. Coroutines may suspend execution and return control to caller.

Example: Let there be a consumer-producer relationship where one routine

creates items and adds to the queue and the other removes from the queue and uses them.

```
var q := new queue

coroutine produce
  loop
    while q is not full
      create some new items
      add item to q
      yield to consume
```

```
coroutine consume
  loop
    while q is not empty
      remove some items from q
      use the items
      yield to produce
```

COROUTINES

Implementation of Coroutine

Only one activation of each coroutine exists at a time.

A single location, called *resume point* is reserved in the activation record to save the old ip value of CIP when a resume instruction transfer control to another subroutine.

Execution of resume B in coroutine A will involve the following steps:

- The current value of CIP is saved in the resume point location of activation record for A.

The ip value in the resume point location is fetched from B's activation record and assigned to CIP so that subprogram B

resume at proper location

SCHEDULED SUBPROGRAMS

Subprogram Scheduling

Normally execution of subprogram is assumed to be initiated immediately upon its call

Subprogram scheduling

relaxes the above condition.

Scheduling Techniques:

1. Schedule subprogram to be executed before or after other subprograms. call B after A

2.Schedule subprogram to be executed when given Boolean expression is true call X when Y = 7 and Z > 0

3. Schedule subprograms on basis of a simulated time scale. call B at time = Currenttime + 50

4. Schedule subprograms according to a priority designation call B with priority 5

Languages : GPSS, SIMULA

DATA CONTROL

- **Names and referencing environments**

Two ways to make a data object available as an operand for an operation.

1. **Direct transmission** – A data object computed at one point as the result of an operation may be directly transmitted to another operation as an operand

   Example: x = y + 2*z;

   The result of multiplication is transmitted directly as an operand of the addition operation.

2. **Referencing through a named data object** –
   A data object may be given a name when it is created, and the name may then be used to designate it as an operand of an operation.

Program elements that may be named

1. Variables
2. Formal parameters
3. Subprograms
4. Defined types
5. Defined constants
6. Labels
7. Exception names
8. Primitive operations
9. Literal constants

Names from 4 thru 9 - resolved at translation time. Names 1 thru 3 - discussed below.

Simple names: identifiers, e.g. *var1*.
Composite names: names for data structure components, e.g. *student[4].last_name*.

Associations and Referencing Environments

Association: binding identifiers to particular data objects and subprograms
Referencing environment: the set of identifier associations for a given

subprogram. **Referencing operations** during program execution: determine the particular data object or subprogram associated with an identifier.

Local referencing environment:

The set of associations created on entry to a subprogram that represent formal parameters, local variables, and subprograms defined only within that subprogram

Nonlocal referencing environment:

The set of associations for identifiers that may be used within a subprogram but that are not created on entry to it. Can be global or predefined.

Global referencing environment: associations created at the start of execution of the main program, available to be used in a subprogram,

Predefined referencing environments: predefined association in the language definition.

Visibility of associations

Associations are visible if they are part of the referencing environment. Otherwise associations are hidden

Dynamic scope of associations

The set of subprogram activations within which the association is visible

Aliases for data objects: Multiple names of a data object

- separate environments - no problem
- in a single referencing environment - called aliases.

Problems with aliasing

- Can make code difficult to understand for the programmer.
- Implementation difficulties at the optimization step - difficult to spot interdependent statements - not to reorder them

- **Static and dynamic scope**

The **dynamic scope of an association** for an identifier is that set of subprogram activations in which the association is visible during execution.

Dynamic scope rules

relate references with associations for names during program execution.

The **static scope of a declaration** is that part of the program text where a use of the identifier is a reference to that particular declaration of the identifier.

Static scope rules

relate references with declarations of names in the program text.

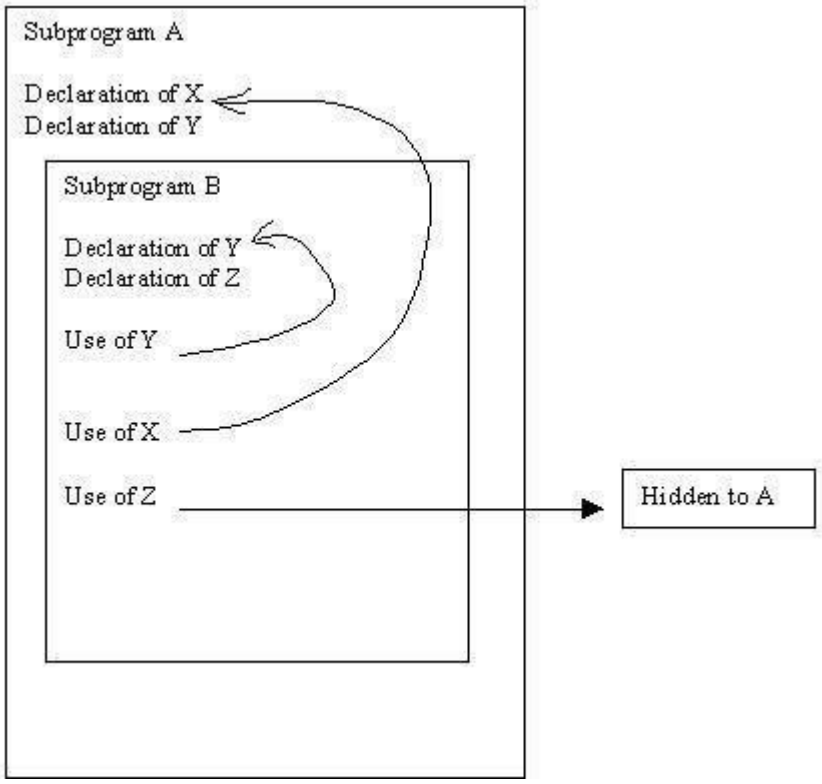
Importance of static scope rules - recording information about a variable during translation.

- **Block structure**

Block-structured languages (Pascal):

- Each program or subprogram is organized as a set of nested blocks.
- The chief characteristic of a block is that it introduces a new local referencing environment.

Static scope rules for block-structured programs



- **Local data and local referencing environments**

Local environment of a subprogram: various identifiers declared in the subprogram - variable names, parameters, subprogram names.

Static scope rules: implemented by means of a table of the local declarations

Dynamic scope rules: two methods:

- **Retention** - associations and the bound values are retained after execution.
- **Deletion** - associations are deleted.
  (For further explanation and example see Figure 9.9 on p. 369)

Implementation of dynamic scope rules in local referencing environments: by means of a local environment table to associate names, types and values.

Retention: the table is kept as part of the code segment
Deletion: the table is kept as part of the activation record, destroyed after each execution.

- **Parameter transmission**

Subprograms need mechanisms to exchange data.

Arguments - data objects sent to a subprogram to be processed

Obtained through

- parameters
- non-local references

Results - data object or values delivered by the subprogram

Returned through

- parameters
- assignments to non-local variables
- explicit function values

1. **Actual and Formal Parameters**

A formal parameter is a particular kind of local data object within a subprogram. It has a name, the declaration specifies its attributes.

An actual parameter is a data object that is shared with the caller subprogram. Might be:

- a local data object belonging to the caller,
- a formal parameter of the caller,
- a nonlocal data object visible to the caller,
- a result returned by a function invoked by the caller and immediately transmitted to the called subprogram.

Establishing a Correspondence

Positional correspondence – pairing actual and formal parameters based on their respective positions in the actual- and formal- parameter lists.

Correspondence by explicit name – the name is paired explicitly by the caller.

2. **Methods for transmitting parameters**

Call by name – the actual parameter is substituted in the subprogram.

Call by reference – a pointer to the location of the data object is made available to the subprogram. The data object does not change position in memory.

Call by value – the value of the actual parameter is copied in the location of the formal parameter.

Call by value-result – same as call by value, however at the end of execution the result is copied into the actual parameter.

Call by constant value – if a parameter is transmitted by constant value, then no change in the value of the formal parameter is allowed during program execution.

Call by result – a parameter transmitted by result is used only to transmit a result back from a subprogram. The initial value of the actual-parameter data object makes no difference and cannot be used by the subprogram.

Note: Often "pass by" is used instead of "call by" .

Examples:

Pass by name in Algol

```
procedure S (el, k);
integer el, k;
begin
        k:=2; el := 0
end;
A[1] := A[2] := 1; i :=
1; S(A[i],i);
```

Pass by reference in FORTRAN

```
SUBROUTINE S (EL,
K) K = 2
EL = 0
RETU
RN
END
A(1) = A(2) = 1 I
= 1
CALL S (A(I), I)
```

Pass by name:

After calling S(A[i],i), the effect is as if the procedure were

i := 2;

A[i] := 0;

As a result A[2]becomes 0.

On exit we have

i = 2, A[1] = 1, A[2] = 0.

Pass by reference:

Since at the time of call **i** is 1, the formal parameter **el** is linked to the address of A(1). Thus it is A(1)that becomes 0.

On exit we have: i = 2, A(1) = 0, A(2) = 1

3. Transmission semantics

Types of parameters:

input information
output information (the
result) both input and output

The three types can be accomplished by copying or using pass-by-

reference Return results:

Using parameters
Using functions with a return value

4. Implementation of parameter transmission

Implementing formal parameters:

Storage - in the activation
record Type:
Local data object of type T in case of pass by value, pass by value-result, pass by result
Local data object of type pointer to T in case of pass by reference
Call by name implementation: the formal parameters are subprograms that evaluate the actual parameters.

Actions for parameter transmission:

- associated with the point of call of the subprogram

  each actual parameter is evaluated in the referencing environment of the calling program, and list of pointers is set up.

- associated with the entry and exit in the

  subprogram on entry:
  copying the entire contents of the actual parameter in the formal parameter, or copying the pointer to the actual parameter
  on exit:
  copying result values into actual parameters or copying function values into registers

These actions are performed by *prologue* and *epilogue* code generated by the compiler and stored in the segment code part of the activation record of the subprogram.

Thus the compiler has two main tasks in the implementation of parameter transmission

3. It must generate the correct executable code for transmission of parameters, return of results, and each reference to a formal-parameter name.
4. It must perform the necessary static type checking to ensure that the type of each actual- parameter data object matches that declared for the corresponding formal parameter

- **Explicit common environment**

This method of sharing data objects is straightforward.

Specification: A common environment that is similar to a local environment, however it is not a part of any single subprogram.

It may contain: definitions of variables, constants, types. It cannot contain: subprograms, formal parameters.

Implementation: as a separate block of memory

storage. Special keywords are used to specify variables

to be shared.

Unit-4

Storage Management

Memory management is the functionality of an operating system which handles or manages primary memory and moves processes back and forth between main memory and disk during execution. Memory management keeps track of each and every memory location, regardless of either it is allocated to some process or it is free. It checks how much memory is to be allocated to processes. It decides which process will get memory at what time. It tracks whenever some memory gets freed or unallocated and correspondingly it updates the status.

This tutorial will teach you basic concepts related to Memory Management.

Process Address Space

The process address space is the set of logical addresses that a process references in its code. For example, when 32-bit addressing is in use, addresses can range from 0 to 0x7fffffff; that is, 2^31 possible numbers, for a total theoretical size of 2 gigabytes.

The operating system takes care of mapping the logical addresses to physical addresses at the time of memory allocation to the program. There are three types of addresses used in a program before and after memory is allocated −

S.N.	Memory Addresses & Description
1	**Symbolic addresses** The addresses used in a source code. The variable names, constants, and instruction labels are the basic elements of the symbolic address space.
2	**Relative addresses** At the time of compilation, a compiler converts symbolic addresses into relative addresses.
3	**Physical addresses** The loader generates these addresses at the time when a program is loaded into main memory.

Virtual and physical addresses are the same in compile-time and load-time address-binding schemes. Virtual and physical addresses differ in execution-time address-binding scheme.

The set of all logical addresses generated by a program is referred to as a **logical address space**. The set of all physical addresses corresponding to these logical addresses is referred to as a **physical address space.**

The runtime mapping from virtual to physical address is done by the memory management unit (MMU) which is a hardware device. MMU uses following mechanism to convert virtual address to physical address.

- The value in the base register is added to every address generated by a user process, which is treated as offset at the time it is sent to memory. For example, if the base register value is 10000, then an attempt by the user to use address location 100 will be dynamically reallocated to location 10100.
- The user program deals with virtual addresses; it never sees the real physical addresses.

Static vs Dynamic Loading

The choice between Static or Dynamic Loading is to be made at the time of computer program being developed. If you have to load your program statically, then at the time of compilation, the complete programs will be compiled and linked without leaving any external program or module dependency. The linker combines the object program with other necessary object modules into an absolute program, which also includes logical addresses.

If you are writing a Dynamically loaded program, then your compiler will compile the program and for all the modules which you want to include dynamically, only references will be provided and rest of the work will be done at the time of execution.

At the time of loading, with **static loading**, the absolute program (and data) is loaded into memory in order for execution to start.

If you are using **dynamic loading**, dynamic routines of the library are stored on a disk in relocatable form and are loaded into memory only when they are needed by the program.

Static vs Dynamic Linking

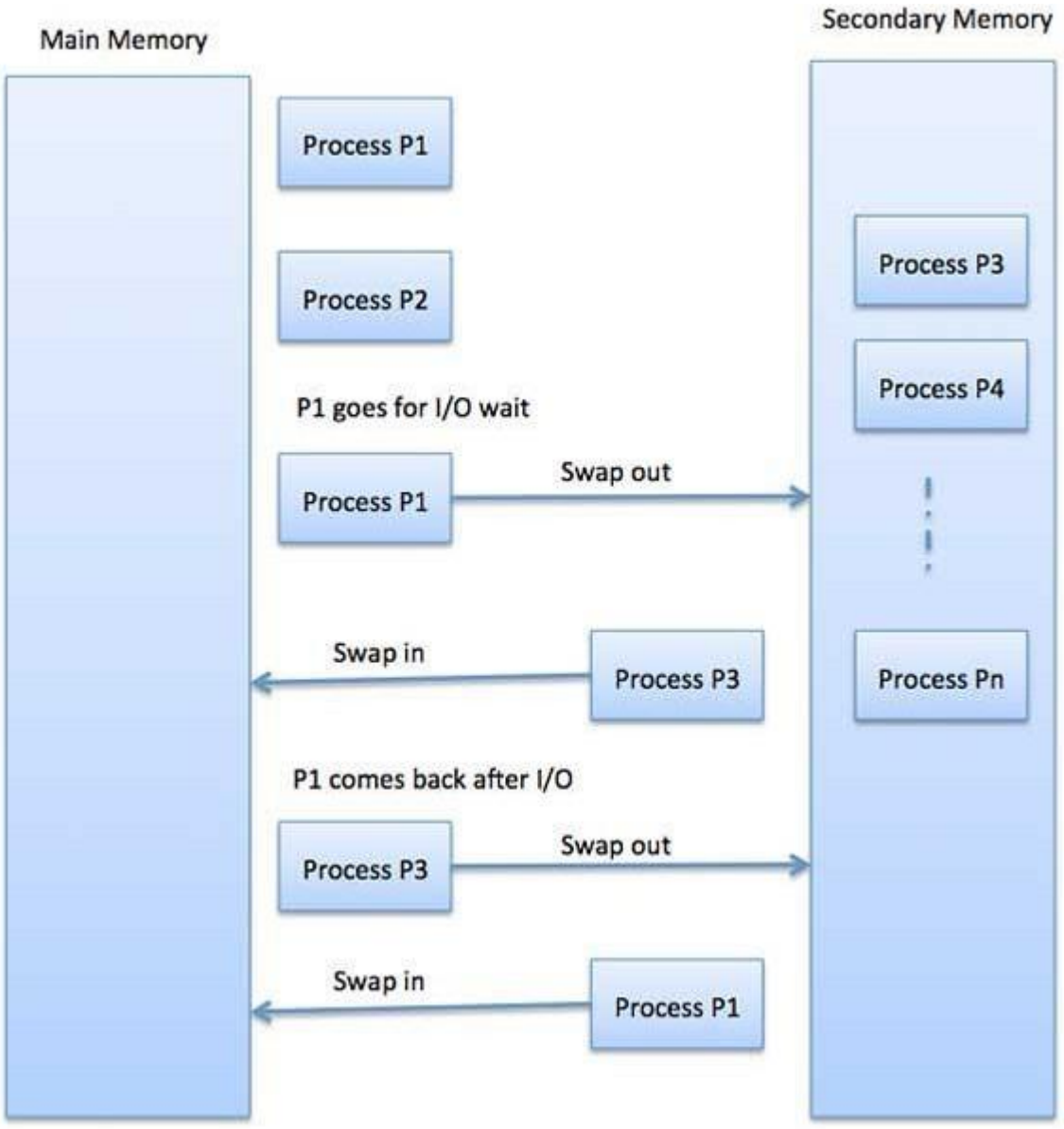
As explained above, when static linking is used, the linker combines all other modules needed by a program into a single executable program to avoid any runtime dependency.

When dynamic linking is used, it is not required to link the actual module or library with the program, rather a reference to the dynamic module is provided at the time of compilation and linking. Dynamic Link Libraries (DLL) in Windows and Shared Objects in Unix are good examples of dynamic libraries.

Swapping

Swapping is a mechanism in which a process can be swapped temporarily out of main memory (or move) to secondary storage (disk) and make that memory available to other processes. At some later time, the system swaps back the process from the secondary storage to main memory.

Though performance is usually affected by swapping process but it helps in running multiple and big processes in parallel and that's the reason **Swapping is also known as a technique for memory compaction**.



The total time taken by swapping process includes the time it takes to move the entire process to a secondary disk and then to copy the process back to memory, as well as the time the process takes to regain main memory.

Let us assume that the user process is of size 2048KB and on a standard hard disk where swapping will take place has a data transfer rate around 1 MB per second. The actual transfer of the 1000K process to or from memory will take

2048KB / 1024KB per second

= 2 seconds

= 2000 milliseconds

Now considering in and out time, it will take complete 4000 milliseconds plus other overhead where the process competes to regain main memory.

Memory Allocation

Main memory usually has two partitions −

- **Low Memory** − Operating system resides in this memory.
- **High Memory** − User processes are held in high memory.

Operating system uses the following memory allocation mechanism.

S.N.	Memory Allocation & Description
1	**Single-partition allocation** In this type of allocation, relocation-register scheme is used to protect user processes from each other, and from changing operating-system code and data. Relocation register contains value of smallest physical address whereas limit register contains range of logical addresses. Each logical address must be less than the limit register.
2	**Multiple-partition allocation** In this type of allocation, main memory is divided into a number of fixed-sized partitions where each partition should contain only one process. When a partition is free, a process is selected from the input queue and is loaded into the free partition. When the process terminates, the partition becomes available for another process.

Fragmentation

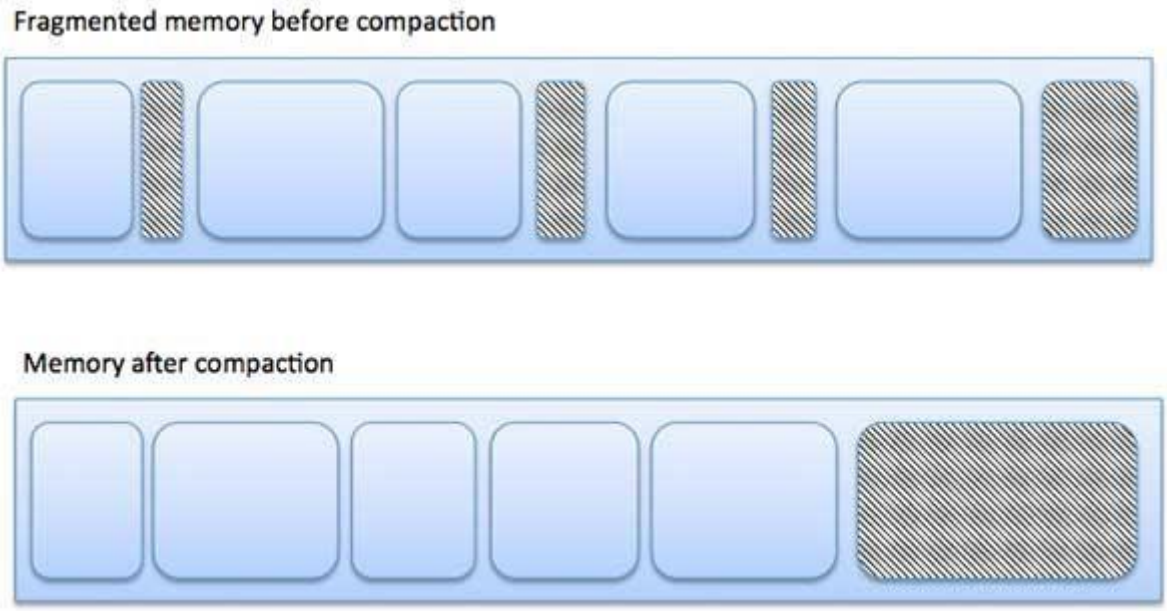
As processes are loaded and removed from memory, the free memory space is broken into little pieces. It happens after sometimes that processes cannot be allocated to memory blocks considering their small size and memory blocks remains unused. This problem is known as Fragmentation.

Fragmentation is of two types −

S.N.	Fragmentation & Description

1	**External fragmentation** Total memory space is enough to satisfy a request or to reside a process in it, but it is not contiguous, so it cannot be used.
2	**Internal fragmentation** Memory block assigned to process is bigger. Some portion of memory is left unused, as it cannot be used by another process.

The following diagram shows how fragmentation can cause waste of memory and a compaction technique can be used to create more free memory out of fragmented memory −



External fragmentation can be reduced by compaction or shuffle memory contents to place all free memory together in one large block. To make compaction feasible, relocation should be dynamic.

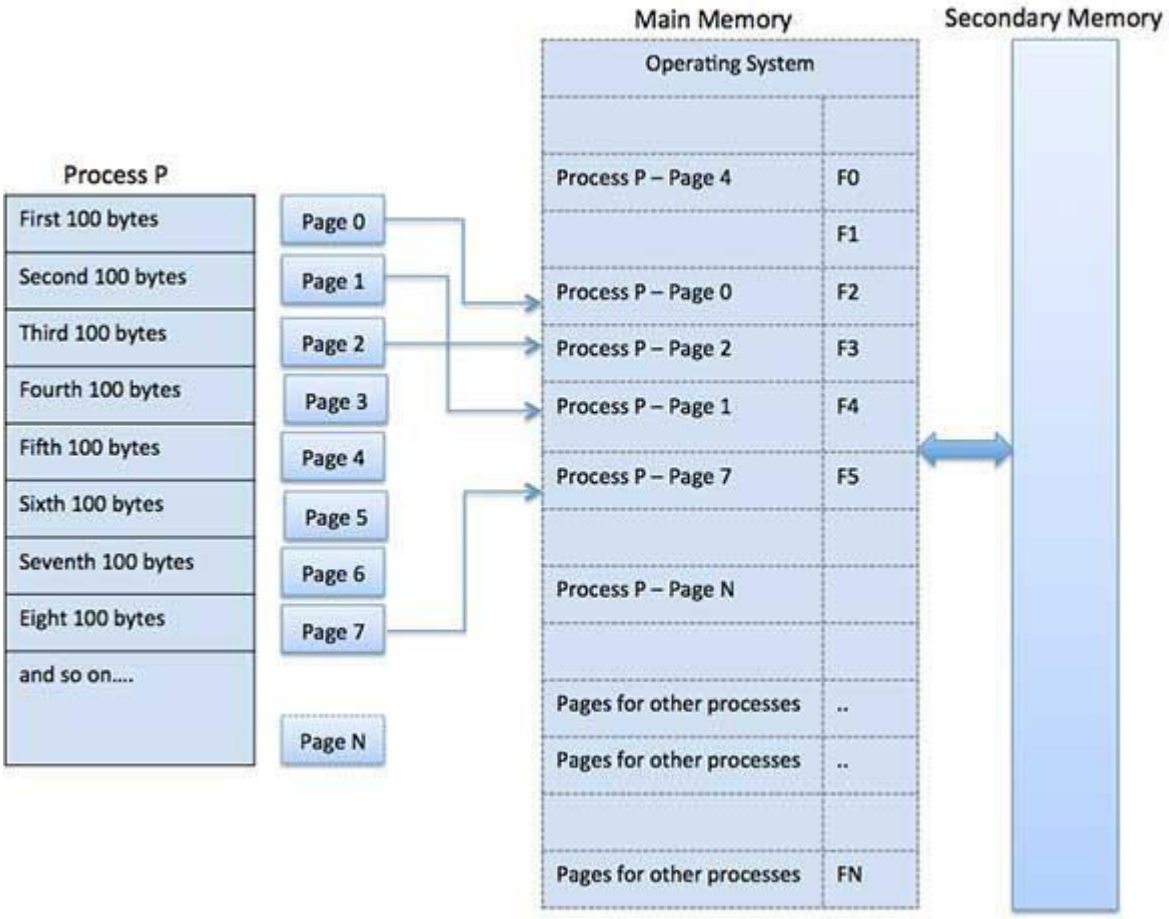
The internal fragmentation can be reduced by effectively assigning the smallest partition but large enough for the process.

Paging

A computer can address more memory than the amount physically installed on the system. This extra memory is actually called virtual memory and it is a section of a hard that's set up to emulate the computer's RAM. Paging technique plays an important role in implementing virtual memory.

Paging is a memory management technique in which process address space is broken into blocks of the same size called **pages** (size is power of 2, between 512 bytes and 8192 bytes). The size of the process is measured in the number of pages.

Similarly, main memory is divided into small fixed-sized blocks of (physical) memory called **frames** and the size of a frame is kept the same as that of a page to have optimum utilization of the main memory and to avoid external fragmentation.



Address Translation

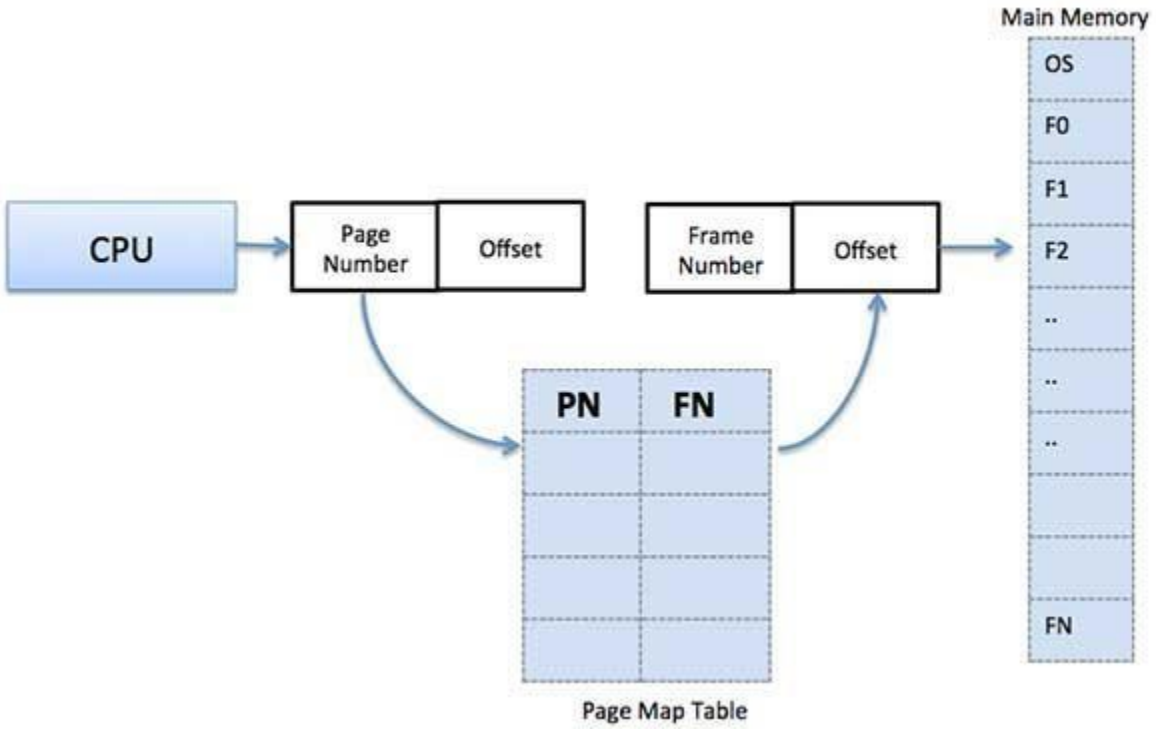
Page address is called **logical address** and represented by **page number** and the **offset**.

```
Logical Address = Page number + page offset
```

Frame address is called **physical address** and represented by a **frame number** and the **offset**.

```
Physical Address = Frame number + page offset
```

A data structure called **page map table** is used to keep track of the relation between a page of a process to a frame in physical memory.



When the system allocates a frame to any page, it translates this logical address into a physical address and create entry into the page table to be used throughout execution of the program.

When a process is to be executed, its corresponding pages are loaded into any available memory frames. Suppose you have a program of 8Kb but your memory can accommodate only 5Kb at a given point in time, then the paging concept will come into picture. When a computer runs out of RAM, the operating system (OS) will move idle or unwanted pages of memory to secondary memory to free up RAM for other processes and brings them back when needed by the program.

This process continues during the whole execution of the program where the OS keeps removing idle pages from the main memory and write them onto the secondary memory and bring them back when required by the program.

Advantages and Disadvantages of Paging

Here is a list of advantages and disadvantages of paging −

- Paging reduces external fragmentation, but still suffer from internal fragmentation.
- Paging is simple to implement and assumed as an efficient memory management technique.
- Due to equal size of the pages and frames, swapping becomes very easy.

- Page table requires extra memory space, so may not be good for a system having small RAM.

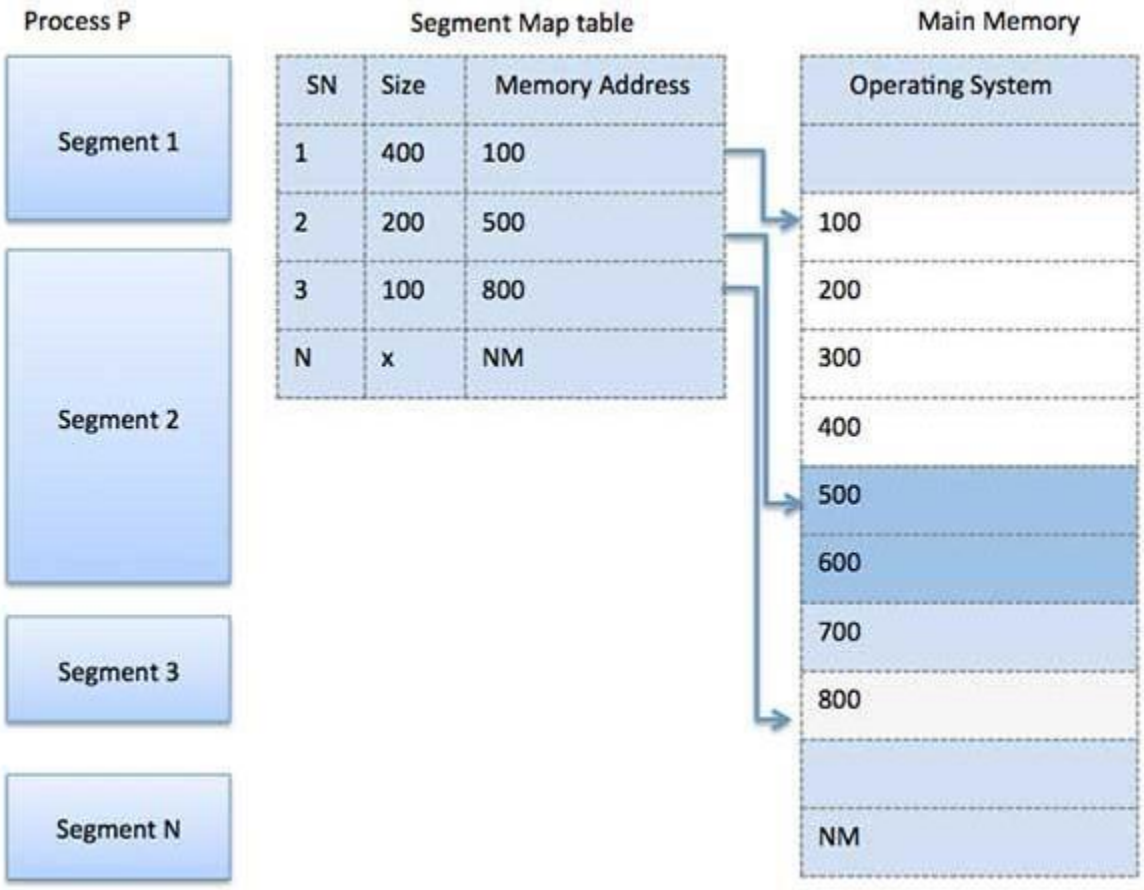
Segmentation

Segmentation is a memory management technique in which each job is divided into several segments of different sizes, one for each module that contains pieces that perform related functions. Each segment is actually a different logical address space of the program.

When a process is to be executed, its corresponding segmentation are loaded into non-contiguous memory though every segment is loaded into a contiguous block ofavailable memory.

Segmentation memory management works very similar to paging but here segments are of variable-length where as in paging pages are of fixed size.

A program segment contains the program's main function, utility functions, data structures, and so on. The operating system maintains a **segment map table**for every process and a list of free memory blocks along with segment numbers, their size and corresponding memory locations in main memory. For each segment, the table stores the starting address of the segment and the length of the segment. A reference to a memory location includes a value that identifies a segment and an offset.



Stack vs Heap Memory Allocation

Memory in a C/C++ program can either be allocated on stack or heap.

Stack Allocation : The allocation happens on contiguous blocks of memory. We call it stack memory allocation because the allocation happens in function call stack. The size of memory to be allocated is known to compiler and whenever a function is called, its variables get memory allocated on the stack. And whenever the function call is over, the memory for the variables is deallocated. This all happens using some predefined routines in compiler. Programmer does not have to worry about memory allocation and deallocation of stack variables.

```
int main()

{

  // All these variables get memory

  // allocated on stack

  int a;

  int b[10];

  int n = 20;

  int c[n];

}
```

Heap Allocation : The memory is allocated during execution of instructions written by programmers. Note that the name heap has nothing to do with heap data structure. It is called heap because it is a pile of memory space available to programmers to allocated and de-allocate. If a programmer does not handle this memory well, memory leak can happen in the program.

```
int main()

{

  // This memory for 10 integers

  // is allocated on heap.

  int *ptr = new int[10];

}
```

Key Differences Between Stack and Heap Allocations

1. In a stack, the allocation and deallocation is automatically done by whereas, in heap, it needs to be done by the programmer manually.
2. Handling of Heap frame is costlier than handling of stack frame.
3. Memory shortage problem is more likely to happen in stack whereas the main issue in heap memory is fragmentation.
4. Stack frame access is easier than the heap frame as the stack have small region of memory and is cache friendly, but in case of heap frames which are dispersed throughout the memory so it cause more cache misses.
5. Stack is not flexible, the memory size allotted cannot be changed whereas a heap is flexible, and the allotted memory can be altered.
6. Accessing time of heap takes is more than a stack.

Comparison Chart:

PARAMETER	STACK	HEAP
Basic	Memory is allocated in a contiguous block.	Memory is allocated in any random order.
Allocation and Deallocation	Automatic by compiler instructions.	Manual by programmer.
Cost	Less	More
Implementation	Hard	Easy
Access time	Faster	Slower
Main Issue	Shortage of memory	Memory fragmentation
Locality of reference	Excellent	Adequate
Flexibility	Fixed size	Resizing is possible

Heap Storage Management

It is based on the heap data structure.Heap is a block of storage within which data are allocated/freed in an arbitrary manner. In this the problem of allocation, recovery, compaction and reuse may be complicated.Heap storage management can be of two types:

✓ Fixed Size Elements:

This is quite simple.Compaction is not a problem as all the elements are of same size.

✓ Variable Size Elements:

This storage technique is programmer control allocation more difficult than the fixed size elements.The major difficulties with variable size element are reuse of recover space.

Abstraction

abstraction is a technique for arranging complexity of computer systems. It works by establishing a level of complexity on which a person interacts with the system, suppressing the more complex details below the current level. The programmer works with an idealized interface (usually well defined) and can add additional levels of functionality that would otherwise be too complex to handle. For example, a programmer writing code that involves numerical operations may not be interested in the way numbers are represented in the underlying hardware (e.g. whether they're *16 bit* or *32 bit integers*), and where those details have been suppressed it can be said that they were *abstracted away*, leaving simply *numbers* with which the programmer can work. In addition, a task of sending an email message across continents would be extremely complex if the programmer had to start with a piece of fiber optic cable and basic hardware components. By using layers of complexity that have been created to abstract away the physical cables and network layout, and presenting the programmer with a virtual data channel, this task is manageable.

Abstraction can apply to control or to data: **Control abstraction** is the abstraction of actions while **data abstraction** is that of data structures.

- Control abstraction involves the use of subroutines and control flow abstractions
- Data abstraction allows handling pieces of data in meaningful ways. For example, it is the basic motivation behind the datatype.

encapsulation

encapsulation is used to refer to one of two related but distinct notions, and sometimes to the combination[1][2] thereof:

- A language mechanism for restricting direct access to some of the object's components.[3][4]
- A language construct that facilitates the bundling of data with the methods (or other functions) operating on that data.[5][6]

Some programming language researchers and academics use the first meaning alone or in combination with the second as a distinguishing feature of object-oriented programming, while some programming languages which provide lexical closures view encapsulation as a feature of the language orthogonal to object orientation.

The second definition is motivated by the fact that in many of the OOP languages hiding of components is not automatic or can be overridden; thus, information hiding is defined as a separate notion by those who prefer the second definition.

The features of encapsulation are supported using classes in most object-oriented programming languages, although other alternatives also exist.

An information-hiding mechanism

Encapsulation can be used to hide data members and members function. Under this definition, encapsulation means that the internal representation of an object is generally hidden from view outside of the object's definition. Typically, only the object's own methods can directly inspect or manipulate its fields. Some languages like Smalltalk and Ruby only allow access via object methods, but most others (e.g. C++, C#, Delphi or Java) offer the programmer a degree of control over what is hidden, typically via keywords like public and private.[4] It should be noted that the ISO C++ standard refers to protected, private and public as "access specifiers" and that they do not "hide any information". Information hiding is accomplished by furnishing a compiled version of the source code that is interfaced via a header file.

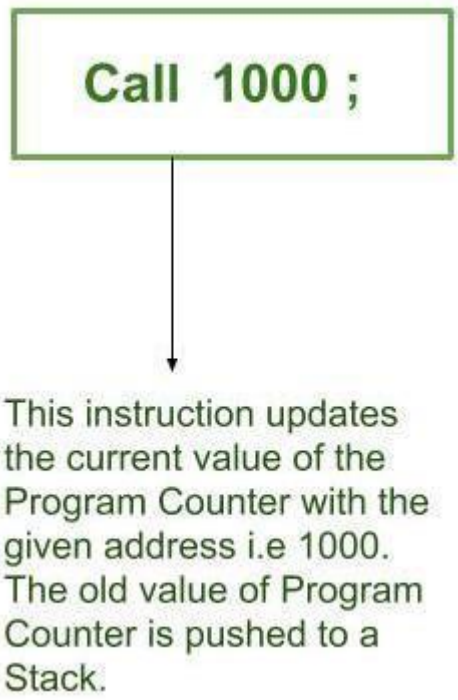
Hiding the internals of the object protects its integrity by preventing users from setting the internal data of the component into an invalid or inconsistent state. A supposed benefit of encapsulation is that it can reduce system complexity, and thus increase robustness, by allowing the developer to limit the inter-dependencies between software components

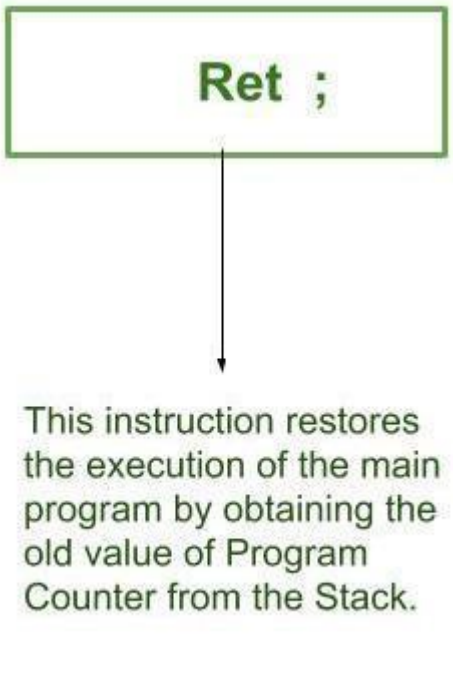
Almost always, there is a way to override such protection – usually via reflection API (Ruby, Java, C#, etc.), sometimes by mechanism like name mangling (Python), or special keyword usage like friend in C++.

Subprogram and its Characteristics

A **Subprogram** is a program inside any larger program that can be reused any number of times.
Characteristics of a Subprogram:
(1) A Subprogram is implemented using the **Call & Return** instructions in Assembly Language.

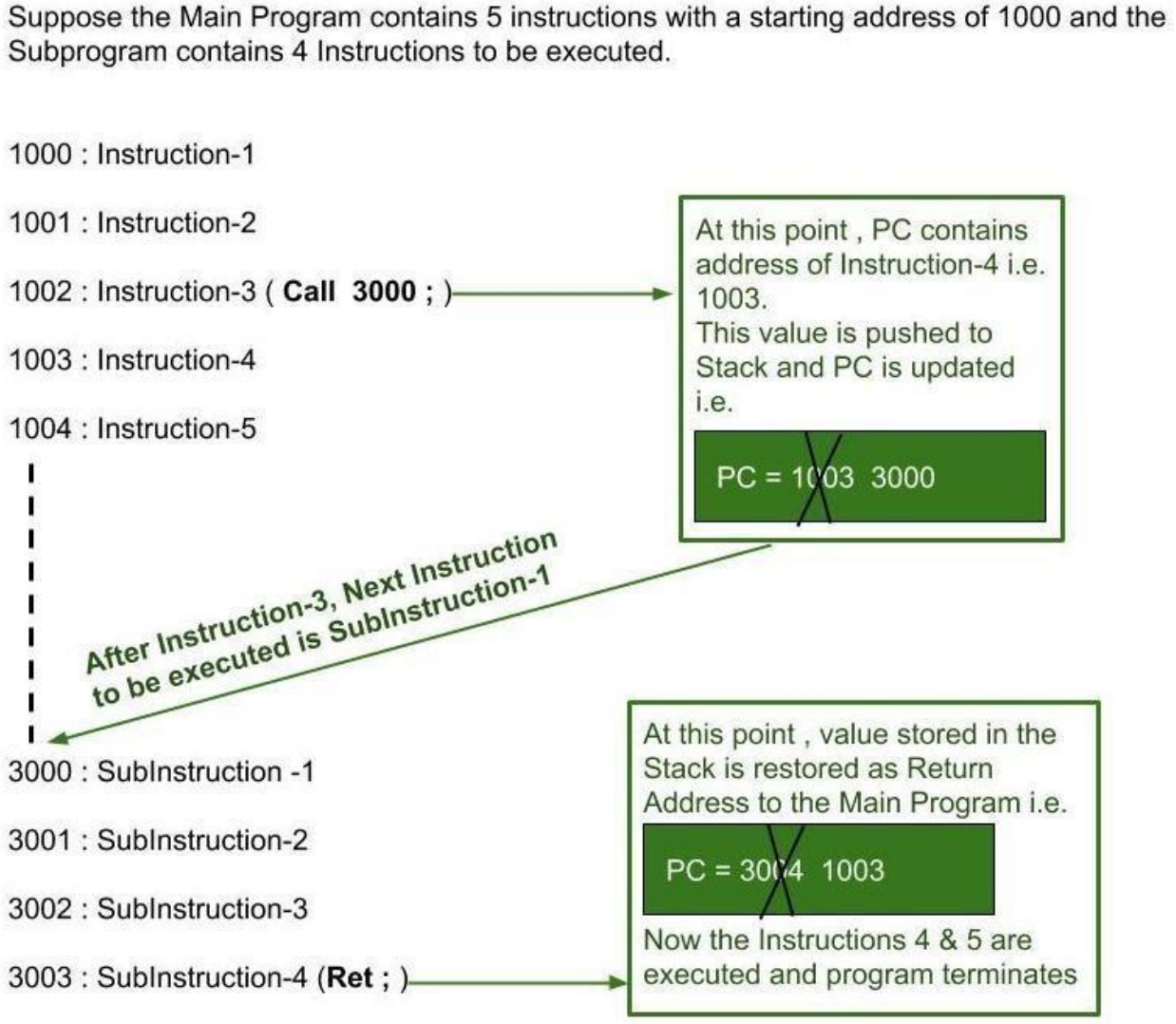
(2) The Call Instruction is present in the Main Program and the Return(Ret) Instruction is present in the subprogram itself.





(3) It is important to note that the Main Program is suspended during the execution of any subprogram. Moreover, after the completion of the subprogram the main program executes from the next sequential address present in the Program Counter .

(4) For the implementation of any subprogram, a **"Stack"** is used to store the **"Return Address"** to the Main Program . Here, Return Address means the immediately next instruction address after the Call Instruction in the Main program. This Return Address is present inside the Program Counter . Thus during the execution of the Call Instruction, the Program Counter value is first pushed to the Stack as the Return Address and then the Program Counter value is updated to the given address in the Call Instruction . Similarly, during the execution of Return(Ret) Instruction, the value present in the stack is popped and the Program Counter value is restored for further execution of the Main Program .
(5) The Main advantage of Subprogram is that it avoids repetition of Code and allows us to reuse the same code again and again.



Introduction to Programming Languages/Type Definition

Data Types

The vast majority of the programming languages deal with typed values, i.e., integers, booleans, real numbers, people, vehicles, etc. There are however, programming languages that have no types at all. These programming languages tend to be very simple. Good examples in this category are the core lambda calculus, and Brain Fuc*. There exist programming languages that have some very primitive typing systems. For instance, the x86 assembly allows to store floating point numbers, integers and addresses into the same registers. In this case, the particular instruction used to process the register determines which data type is being taken into consideration. For instance, the x86 assembly has a subl instruction to perform integer subtraction, and another instruction, fsubl, to subtract floating point values. As another

example, BCPL has only one data type, a word. Different operations treat each word as a different type. Nevertheless, most of the programming languages have more complex types, and we shall be talking about these typing systems in this chapter.

The most important question that we should answer now is "what is a data type". We can describe a data type by combining two notions:

- Values: a type is, in essence, a set of values. For instance, the boolean data type, seen in many programming languages, is a set with two elements: true and false. Some of these sets have a finite number of elements. Others are infinite. In Java, the integer data type is a set with 2^{32} elements; however, the string data type is a set with an infinite number of elements.
- Operations: not every operation can be applied on every data type. For instance, we can sum up two numeric types; however, in most of the programming languages, it does not make sense to sum up two booleans. In the x86 assembly, and in BCPL, the operations distinguish the type of a memory location from the type of others.

Types exist so that developers can represent entities from the real world in their programs. However, types are not the entities that they represent. For instance, the integer type, in Java, represents numbers ranging from -2^{31} to $2^{31} - 1$. Larger numbers cannot be represented. If we try to assign, say, 2^{31} to an integer in Java, then we get back -2^{31}. This happens because Java only allows us to represent the 31 least bits of any binary integer.

Types are useful in many different ways. Testimony of this importance is the fact that today virtually every programming language uses types, be it statically, be it at runtime. Among the many facts that contribute to make types so important, we mention:

- Efficiency: because different types can be represented in different ways, the runtime environment can choose the most efficient alternative for each representations.
- Correctness: types prevent the program from entering into undefined states. For instance, if the result of adding an integer and a floating point number is undefined, then the runtime environment can trigger an exception whenever this operation might happen.
- Documentation: types are a form of documentation. For instance, if a programmer knows that a given variable is an integer, then he or she knows a lot about it. The programmer knows, for example, that this variable can be the target of arithmetic operations. The programmer also knows much memory is necessary to allocate that variable. Furthermore, contrary to simple comments, that mean nothing to the compiler, types are a form of documentation that the compiler can check.

Types are a fascinating subject, because they classify programming languages along many different dimensions. Three of the most important dimensions are:

- Statically vs Dynamically typed.
- Strongly vs Weakly typed.
- Structurally vs Nominally typed.

In any programming language there are two main categories of types: primitive and constructed. Primitive types are atomic, i.e., they are not formed by the combination of other types. Constructed, or composite types, as the name already says, are made of other types, either

primitive or also composite. In the rest of this chapter we will be showing examples of each family of types.

Abstract data types

Earlier, we referred to procedural abstraction as a process that hides the details of a particular function to allow the user or client to view it at a very high level. We now turn our attention to a similar idea, that of**data abstraction**. An **abstract data type**, sometimes abbreviated **ADT**, is a logical description of how we view the data and the operations that are allowed without regard to how they will be implemented. This means that we are concerned only with what the data is representing and not with how it will eventually be constructed. By providing this level of abstraction, we are creating an **encapsulation**around the data. The idea is that by encapsulating the details of the implementation, we are hiding them from the user's view. This is called **information hiding**.

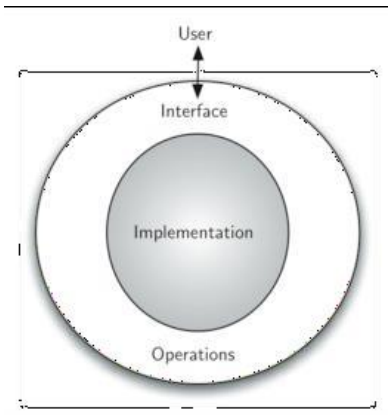


Figure shows a picture of what an abstract data type is and how it operates. The user interacts with the interface, using the operations that have been specified by the abstract data type. The abstract data type is the shell that the user interacts with. The implementation is hidden one level deeper. The user is not concerned with the details of the implementation.

The implementation of an abstract data type, often referred to as a **data structure**, will require that we provide a physical view of the data using some collection of programming constructs and primitive data types. As we discussed earlier, the separation of these two perspectives will allow us to define the complex data models for our problems without giving any indication as to the details of how the model will actually be built. This provides an **implementation-independent** view of the data. Since there will usually be many different ways to implement an abstract data type, this implementation independence allows the programmer to switch the details of the implementation without changing the way the user of the data interacts with it. The user can remain focused on the problem-solving process.