Unit - 3

* Sequence Control *

Control structures in a programming language provide the basic framework within which operations and data are combined into programs. This involves two aspects: control of the order of execution of the operations, both primitive and user defined, which we term sequence control and control of the transmission of data among the subprograms of a program, which we term data control.

Sequence-control structures may be either (default) implicit or explicit. Implicit sequence-control structures are those defined by the language to be in effect unless modified by the programmer through some explicit structure. For example, most languages define the physical sequence of statements in a program as controlling the sequence in which statements are executed unless modified by an explicit sequence-control statement. Explicit sequence-control structures are those that the programmer may optionally use to modify the implicit sequence of operations defined by the language; eg) using parenthesis within expressions or goto statements and statement labels.

Sequence-control structures may be conveniently categorized into four groups:

1) Expressions, form the basic building blocks for statements and express how data are manipulated and changed by a program. Properties such as precedence

rules and parentheses determine how expression become evaluated.

2) Statements or groups of statements, such as conditional and iteration statements, determine how control flows from one statements or segment of a program to another.

3) Declarative programming, is an execution model that does not depend on statements, but nevertheless causes execution to proceed through a program. The logic programming model of prolog is an example of this.

4) Subprograms, such as subprogram calls and coroutines, form a way to transfer control from one segment of a program to another.

(Arithmetic statement)

A) Sequence Control within Expression:

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

i) Arithmetic Expression:- An expression consisting of numerical values (number, or 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 parenthesis, the expression inside parenthesis is evaluated first.

ii) Relational Expressions:- An expressions involving a relational operator is known as "Relational Expression".

A relational expression can be defined as a meaningful combination of operands and relational operators.

* 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 rational operators. The resulting expression will be of integer type, true=1, false=0.

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. Eg: a>b && b<c.

* Evaluation of Logical Expression:- The result of logical expression is either true or false. For expression involving AND(&&), OR(||) and NOT(!) operations, expression involving NOT is evaluated first, then expression with AND and finally expression having OR is evaluated.

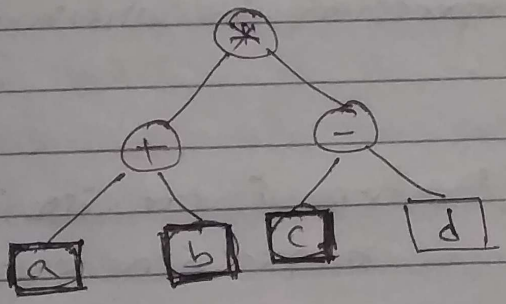
⇒ Controlling the evaluation of expressions:-

a) Precedence (Priority):- If expression involving more than one operator is evaluated, the operator at higher level in a hierarchy or precendence order is evaluated first.

b) Associativity:- An expression involving operations

at the same level in the hierarchy, an additional implicit rule for associativity is needed to completely define the order of operations. The operators of the same precedence are evaluated either from left to right or from right to left. Left-to-right associativity is the most common implicit rule, so that a-b-c is treated as (a-b)-c. However, common mathematical conventions say that exponentiation works from right to left: $a \uparrow b \uparrow c = a \uparrow (b \uparrow c)$.

* Expression Tree: An expression (Arithmetic, logical, relational) 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:



* Syntax for Expressions:

a) Prefix or Polish notation :- Named after polish mathematician Jan Lukasiewicz, refers to notation in which operator is prefixed to operands. i.e operator is written ahead of operands.

b) Postfix or reverse polish notation:- in which the operator symbol is placed after its two operands.

c) Infix notation:- It is most suitable for binary (dyadic) operation. The operator symbol is placed between the two operands.

Examples:

	Infix Notation	Prefix Notation	Postfix Notation
1.	a + b	+ ab	ab +
2.	(a + b) * c	* + abc	ab + c *
3.	a * (b + c)	* a + bc	abc + *
4.	a/b + c/d	+ / ab / cd	ab / cd / +
5.	(a + b) * (c + d)	* + ab + cd	ab + cd + *
6.	((a + b) * c) - d	- * + abcd	ab + c * d -
7.			

Semantics for Expressions:- Sementics 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 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, then operator is n-ary operator).

ii) If 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 and replace the operator and it's n operand

by the result.

b) Evaluation of Postfix Expression:
If P is an expression evaluate using stack

i) If 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.

c) Evaluation of Infix Expression → Infix notation is common but its use in expression cause the problems:
~~Infix notation is common but its use in expression~~
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.

* Execution-Time Representation:- Translators evaluate the expression using a method to get efficient result. 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 by performing the two stages in one step. The ordering of machine code instructions reflect the control sequence of original expression.

b) Tree structures :- The expressions may be executed directly in tree structure representation using a software interpreter.

c) Prefix or postfix form :- Expression is prefix or postfix form may be executed by the simple interpretation algorithms.

⇨ Evaluation of tree representation:

→ Eager evaluation :- evaluates all operands before applying operators.

→ Lazy evaluation :- first evaluates all operands and then apply operations.

Problems: (Issues in evaluating expressions)

1. Side effects :- Some operations may change operands of other operations
2. Error conditions :- may depend on the evaluation strategy (eager or lazy evaluation)
3. Boolean expressions :- results may differ depending on the evaluation strategy.

1) Sequential control (between) within statement:
The basic mechanism for controlling the sequence in which the individual statements are executed within a program.

2) Basic Statements:- The results from any program are determined by its basic statements that apply operations to data objects.

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;

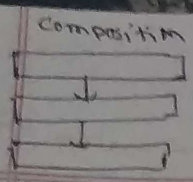
iv) GoTo statement
Explicit sequence control statement. Used to branch conditionally from one point to another in the program

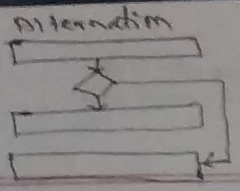
v) Break statement:- An early exit from a loop can be accomplished by using break statement.

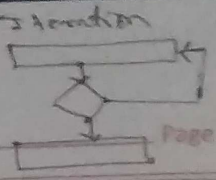
2) Statement Level Sequence Control

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







Date Page 18

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 time.

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 → unconditional goto → conditional goto

b) Use of Break statement

3) Structured Sequence Control:

a) Compound Statement:- collection of two or more statements may be treated as single statement. In C we have

{

- - -

. - -

. - -

}

b) Conditional Statements:

→ if(conditional exp) then ... statements

→ if(conditional exp) then ... statements else ... statements

→ if(conditional exp) then ... statements else if (conditional exp) then ... statements else ... statements

```
switch (exp)
{
  case val1: ---- Statements break;
  case val2: ---- Statements break;
  default : Statements break;
```

c) Iteration Statements:

→ do { --- } while (conditional exp)
→ While (conditional exp) { ---- }
→ for (initialization; test condition; increment) {---- }

c) Sequencing With Nonarithmetic Expressions:

We discussed the execution sequence in the evaluation of arithmetic expressions. However, other expression formats are present in languages. Languages like Prolog, designed for processing character data, include other forms of expression evaluation.

Prolog is not a general-purpose programming language, but instead is oriented toward solving problems using the predicate calculus. A Prolog program consists of a series of facts (data objects), concrete relationships among data objects and a set of rules (pattern of relationship among the database objects). These facts and rules are entered into the database via a consult operation. A program executes by the user entering a query, a set of terms that must all be true. The facts and rules of the database are used to determine which substitutions for variables in the

query are consistent with the information in the database.

Prolog, as an interpreter, prompt the user for input. The user types in a query or function name. The truth (Yes) or falsity (no) of that query is output, as well as an assignment to the variables of the query that make the query true.

Prolog has a simple syntax and semantics. Because it is looking for relationships among series of objects. The variable and the list are the basic data structures that are used.

* Pattern Matching:- A crucial operation in languages like Prolog is pattern matching. An operation succeeds by matching and assigning a set of variables to a predefined template. The recognition of parse tree in BNF grammar is representative of this operation.
For example, the following grammar recognizes odd-length palindromes over the alphabet 0 and 1.

$$A \rightarrow 0A0 \mid 1A1 \mid 0 \mid 1$$

* Term rewriting:- A restricted form of pattern match.
eg: given string $a_1a_2a_3a_4$ and $B \rightarrow a$
if $a = a_3$ then $a_1a_2Ba_4$
we say $a_1a_2Ba_4$ is term re-write of $a_1a_2a_3a_4$

* Unification and substitution:- Prolog consists

of facts and rules and query.

Substitution → replacement of a string to another one.

Unification → A pattern match to determine if the query has a valid substitution consistent with the rules and facts in the database.

* Backtracking:- Backup to the previous sub-goal that matched and try another possible goal for it to match when current sub-goal is failed.

D) Subprogram Control:

A subprogram (also called a procedure, subroutine or function) is a program inside any larger program that can be reused any number of times.

Characteristics:

1) A subprogram is implemented using 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) Main program is suspended during the execution of any subprogram.
4) It avoids repetition of code and allow us to reuse the same code again and again.

* Subprogram Sequence Control:- It is related to the concept how one subprogram invokes another and called subprogram returns to the first.

Before looking at the implementation of the simple call-return structure used for copy-rule view of subprograms, let us list out some of the implicit assumptions present in this view to get more general subprogram control structures.

v.i

1. Subprograms cannot be recursive.
2. Explicit call statements are required.
   - For a subprogram used as an exception handler, no explicit call may be present.
3. Subprograms must execute completely at each call.
   - Not like a coroutine which continues execution from the point of its last termination each time it is called.
4. Immediate transfer of control at point of call.
5. Single execution sequence.
   - Not like tasks which may execute concurrently so that several are in execution at once.

v.ii

⇒ Simple Call-Return Subprograms:

→ Program is composed of single main program.
→ During execution it calls various subprograms which may call 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 the subprogram.
→ After the subprogram is completed, execution of the calling program resumes at the point immediately following the call.

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.

* IMPLEMENTATION:

1) There is a distinction between a subprogram definition and subprogram activation.
-) Subprogram definition → is what we see in written program which is translated into a template. An activation is created each time a subprogram is called using the template created from the definition.
-) Subprogram activation → Created each time a subprogram is called.

2) An activation is implemented as two parts:
i) Code segment :- contains executable code and constants. It is created by translator and stored statically in memory. During execution, it is used but never be modified. Every activation of the subprogram uses the same code segment.

ii) Activation record :- Contains local data, parameters and other data items. It is created new each time the subprogram is called and destroyed when the subprogram returns. While execution of subprogram, the contents of activation record are constantly changing as assignments are made to local variables and other data objects.

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

ii) 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.

iii) Current Environment pointer:- 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 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.

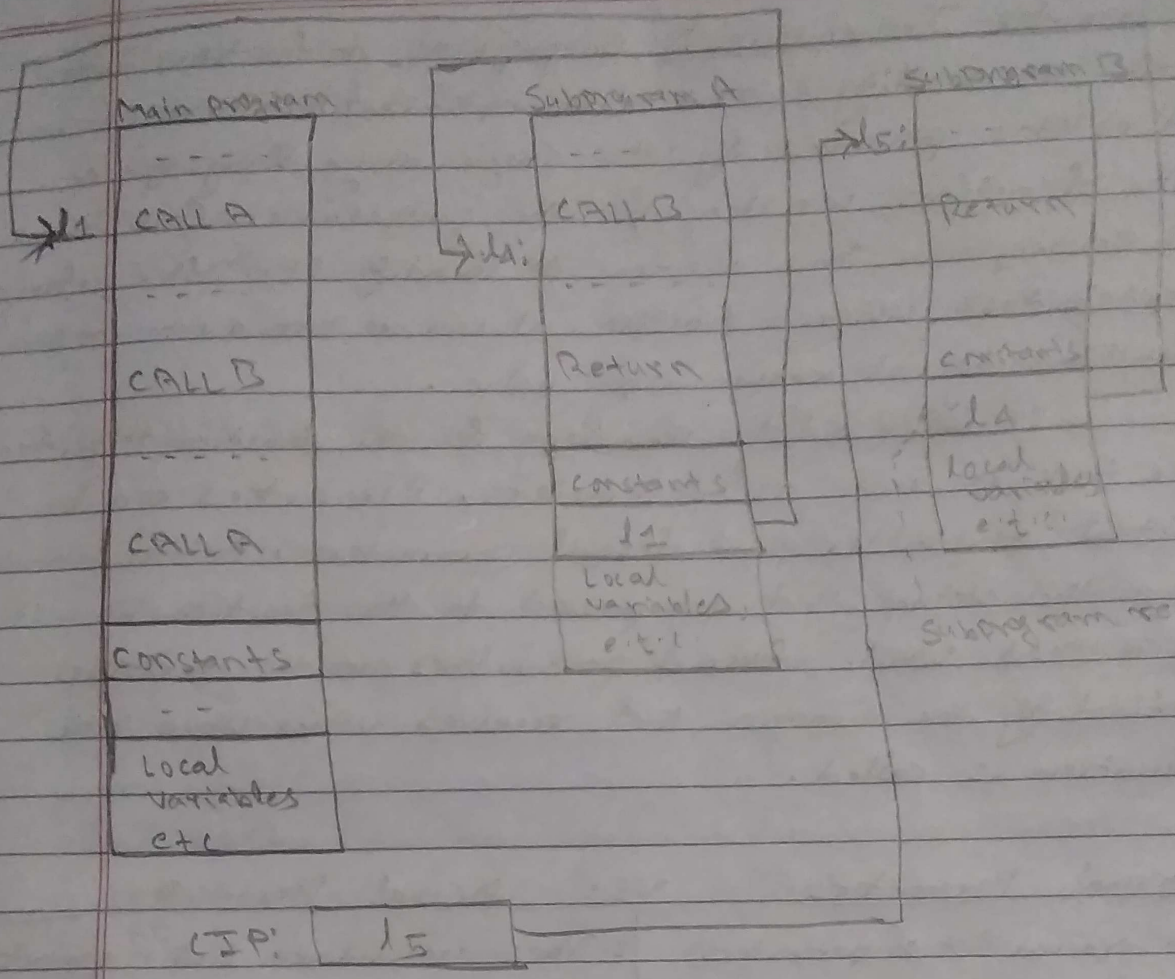


figure 9.2: Subprogram call-return ~~Statement~~ Structure.

⇒ Parameter Transmission :- Subprograms or functions improve code reusability and code optimization. There can be functions provided by the programming language or the functions written by the programmer. Each function or subprogram has a name to identify it. After performing certain task using a function, it can return a value or some functions do not return any value.

The data necessary for the function to perform the task is sent as parameters. Parameters can be actual parameters or formal parameters. Actual parameters are the values that are passed to the function when it is invoked while formal parameters are the variables defined by the function that receives values when the function is called.

S.N	Formal Parameters	vs. Actual Parameters
1)	An ordered list of parameters which are included at the time of definition of a function or subprogram.	1. These are ordered list of parameters which are included at the time of function call.
2)	Data types needs to be mentioned.	2) Data type not required, but they should match with corresponding data type of formal parameters.
3)	These are variables with their data type.	3) These can be variables, expressions and constant without data types.

Eg:

```
int sum(a, b)
{
    int s;
    s = a+b;
    return(s)
}
// Here a and b are formal parameters.
```

Eg:

```
void main()
{
    int s1, s2;
    // function call
    s1 = sum(A, 5);
    s2 = sum(3*g, 25);
}
```

these are actual parameters

3) These are only definition of input data types. These are to shows which type of data should be given as input while calling the function	4) These are actual values which are given as input to the called function.

⇒ Methods for Transmitting Parameters:

When a subprogram transfers control to another subprogram, there must be an association of the actual parameter of the calling subprogram with the formal parameter of the called program.

Modes:

- IN :- Passes into from caller to calle
- OUT :- Callee writes values in caller
- IN/OUT :- Caller tells callee value of variable, which may be updated by calloe.

1) Call by value :- This method uses in-mode semantics. Changes made to formal parameter do not get transmitted back to the caller. Any modifications to the formal parameter variable inside the called function will no be reflected in the actual parameter in the calling environment.

Example:

```
void func(int a, int b)
{
    a = a+b;
    printf("In func, a= %d  b= %d \n", a, b);
}

int main(void)
{
  int x=5 , y=7;
```

```
// Passing parameters
func(x, y)
printf("In main, x=%d y=%d \n", x, y)
return 0;
}
```

Output:

In func, a=12 b=7

In main, x=5 y=7

2) Call by reference :- This technique uses in/out-mode semantics. Changes made to formal parameter do get transmitted back to the caller through parameter passing. Any changes to the formal parameter are reflected in the actual parameter in the calling environment as formal parameter receives a reference (or pointer) to the actual data.

eg:

```
#include <stdio.h>
void change(int *num) {
    printf("Before adding value inside function num = %d \n", *num);
    (*num) += 100;
    printf("After adding value inside function num = %d \n", *num);
}
int main() {
    int x = 100;
    printf("Before function call x = %d \n", x);
    change(&x); // passing reference in function
    printf("After function call x = %d \n", x);
return 0;
}
```

Output:

Before function call x = 100

Before adding value inside function num = 100

After adding value inside function num = 200

After function call x = 200

3) Call by name :- This model of parameter transmission views a subprogram call as a substitution for the entire body of the subprogram. This technique is used in programming language such as Algol. In this technique, symbolic "name" of a variable is passed, which allows it both to be accessed and update.

Example:

To double the value of c[j], you can pass its name (not its value) into following procedure.

```
Procedure double(x);
  real x;
begin
   x := x*2
end;
```

In general, the effect of pass-by-name is to textually substitute the argument in a procedure call for the corresponding parameter in the body of the procedure.

⇒ Referencing environments :- Each program or subprogram has a set of identifier associations available for use in referencing during its execution. This set of identifier associations is termed as the referencing environment of the subprogram (or program).

The referencing environment of a subprogram is ordinarily invariant during its execution. It is set up when the subprogram activation is created, and it remains unchanged during the lifetime of the activation. The values contained in the various data objects may change, but the associations of names with data objects and subprograms

do not. The referencing environment of a subprogram may have several components:

1) 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 forms the local referencing environment of that activation of the subprogram. The meaning of a reference to a name in the local environment may be determined without going outside the subprogram activation.

2) 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 is termed as the nonlocal referencing environment of the subprogram.

3) Global referencing environment :- If the association created at the start of execution of the main program are available to be used in a subprogram then these associations form the global referencing environment of that subprogram. The global environment is a part of the non-local environment.

4) Predefined referencing environment :- Some identifiers have a predefined association that is defined directly in the language definition. Any program or subprogram may use these associations without explicitly creating them.

⇒ Static and Dynamic Scope:

In computer programming, a scope is the context within a computer program in which a variable name or identifier is valid and can be used or within which a declaration has effect. Outside of the scope of a variable name, the variable's value may still be stored, and may even be accessible in some way, but the name does not refer to it, i.e the name is not bound to the variable's storage.

1) Static Scope:- Also called lexical scoping. If a variable name's scope is a certain function, then it's scope is the program text of the function definition, within that text, the variable name exists, and is bound to its variable, but outside that text, the variable name does not exist. In this scoping a variable always refers to it's top level environment.

For example, output of below program is 10, i.e the value returned by f() is not dependent on who is calling it. f() always returns the value of global variable x.

```
# include <stdio.h>
 int x = 10;
 int f()      // called by g()
 {
   return x;
 }
 int g()      // g() has it's own variable and calls f()
 {
   int x = 20;
   return f();
 }
 int main()
 {
   printf("%d", g());
```

```
    printf("\n");
    return 0;
}
```

output:
10.

To sum up in static scoping the compiler first searches in the current block, then in global variables, then in successively smaller scopes.

2) Dynamic Scoping:- A global identifier refers to the identifier associated with the most recent environment, and is uncommon in modern languages. Each identifier has a global stack of bindings and the occurrence of a identifier is searched in the most recent binding. In simpler terms, in dynamic scoping, compiler first searches the current block and then successively all the calling functions.

→ If a variable name's scope is a certain function, then its scope is the time-period during which the function is executing. While the function is running, the variable name exists, and is bound to its variable, but after the function returns, the variable name does not exist.

eg:

```
int x;
int main() {
    x = 2;
    f();
    g();
}
void f() {
    int x = 3;
    h();
}
```

- At runtime one searches for the declaration
- Search in local variables
- If not found, search in local variable of the caller
- If not found, search in global variables
- Otherwise error

```
void g() {
    int x = 4;
    h();
}
void h() {
    printf("%d\n", x);
}
```

Output:

Under static scoping: prints out
2
2

Under dynamic scoping: Prints out
3
4

⇒ Exception and Exception Handlers:

Types of Bugs:

i) Logic Errors :- Errors in program logic due to poor understanding of the problem and solution procedure.

ii) 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.

→ Whenever a program encounters an exceptional condition, it should be identified and dealt with effectively.

* 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)

eg:
```
main()
{
  int x,y;
  cout << "enter the values of x and y";
  cin >> x >> y;
  try {
    if(x!=0)
    cout "y/x is =" << y/x;
```

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

→ 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.

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

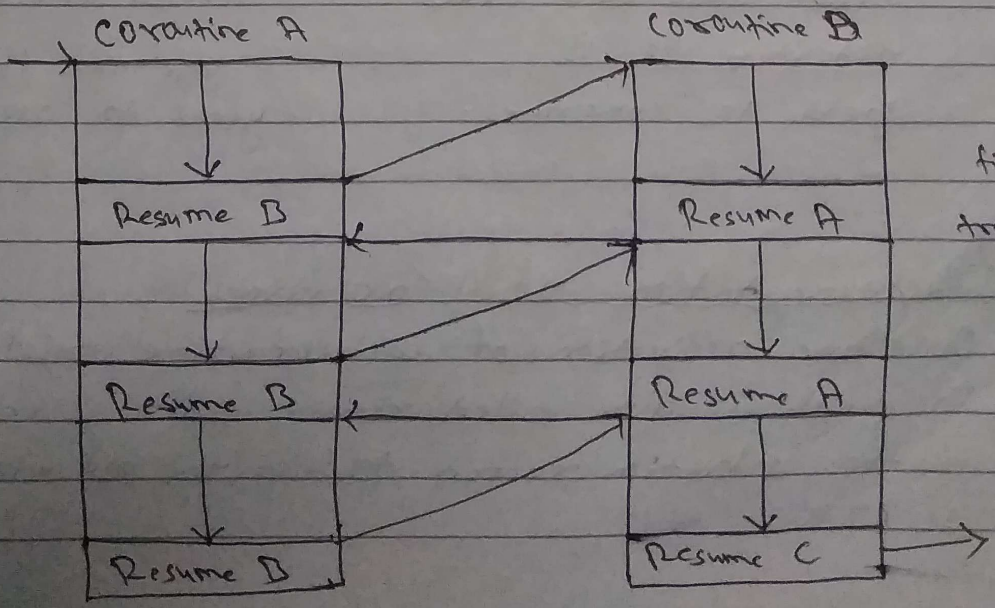


fig: Control transfer between coroutines.

* 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 used and need.
2. The start of subroutine is the only entry point. 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.

q: new queue

Coroutine produce	Coroutine consume
while q is not full	while q is not empty
create some new items	remove some items from q
add item to q	use the items
yield to consume	yield to produce

* Implementation of coroutines:

→ 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 value of ip of CIP

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 it's call.
→ Subprogram scheduling relaxes the above conditions.

* Scheduling Techniques:

1. Schedule subprogram to be executed before or after other subprograms.
   eg: Call B after A
2. Schedule subprogram to be executed when given boolean expression is true.
   eg: 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

⇒ Parallel Programming :- When there is a single execution sequence, the program is termed as a sequential program because execution proceeds in a predefined sequence. A program is termed as a concurrent or parallel program, when each subprogram or task or process can execute concurrently with other subprograms / tasks / process.

Computer systems capable of executing several programs concurrently are now quite common. A multiprocessor system has several central processing units (CPU) sharing a common memory. A distributed or parallel computer system has several computers, each with its own memory and CPU, connected with communication links into a network. In such systems, many tasks may execute concurrently.

Parallel programming constructs add complexity to the language design because several processors may be accessing the same data simultaneously. To consider parallelism in programming languages, the following five concepts must be addressed:

1) Variable definitions: variables may be either mutable or definitional. Mutable variables are the common variables declared in most sequential languages. Values may be assigned to the variables and changed during program execution. A definitional variable may be assigned a value only once. Once assigned a value, any task may access the variable and obtain the correct value.

2. Parallel Composition :- In addition to the sequential and conditional statements of sequential programming, we need to add the parallel statement, which causes additional threads of control to begin executing.

3) Program structure :- Parallel programs generally follow one of the two execution models : a) They may be transformational, where the goal is to transform input data into an appropriate output value. parallelism is applied to speed up the process. b) They may be reactive, where the program reacts to external stimuli called events. Real-time and command and control systems are examples of reactive systems.

4) Communication :- Parallel programs must communicate with one another. Such communications will typically be via shared memory with common data objects or via message passing, Each parallel program has its own copy of the data object and passes data values among the other parallel programs.

5) Synchronization :- Parallel programs must be able to order the execution of its various threads of control. Although nondeterministic behavior is appropriate for many applications, in some an ordering must be imposed.