1. Type Checking

Type checking is a static semantic analysis that ensures that every operation and expression in a program is applied to the correct type of operands and the result is of a valid type.

1. Introduction :

In any programming language, every variable, constant, expression, function, etc. has a type (like int, float, char, bool, etc.). The type checker verifies that all operations are performed on compatible types. It detects type errors at compile time before execution. This helps in preventing runtime errors and ensures the correctness of programs.

2. Need for Type Checking :

- Detects type errors early (at compile time).
- Prevents illegal operations (e.g., int + char*).
- Ensures type safety and program reliability.
- Helps in optimization and code generation.

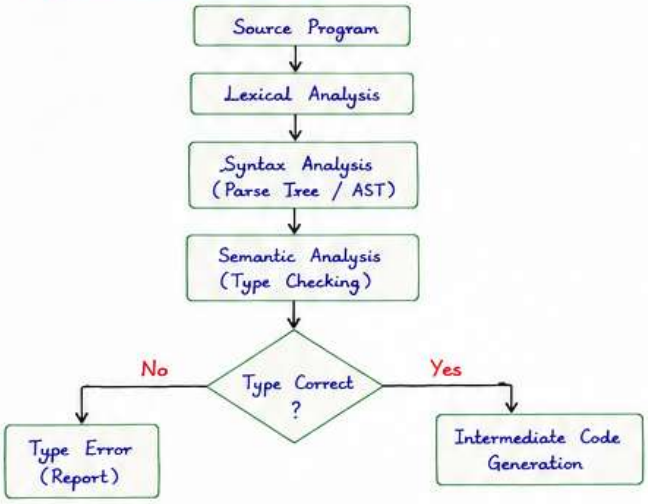
3. Type System :

A type system is a set of rules that assigns a type to each construct in a program and defines the legal operations on those types.

Examples of basic types :

int, float, char, bool, array, pointer, record, function

4. Type Checking Process :



5. Rules for Type Checking :

① Each identifier must be declared before use.
② The type of operands in an operation must be compatible.
③ The result of an operation must be of a valid type.
④ Function arguments must match the parameter types.
⑤ The type of return value must match the function's return type.

6. Type Checking in Expressions :

Expression	Type Rules	Result Type
$E_1 + E_2$	E_1 and E_2 must be numeric	numeric
$E_1 - E_2$	E_1 and E_2 must be numeric	numeric
$E_1 * E_2$	E_1 and E_2 must be numeric	numeric
E_1 / E_2	E_1 and E_2 must be numeric, $E_2 \neq 0$	numeric
$E_1 == E_2$	E_1 and E_2 must be compatible	bool
$E_1 < E_2$	E_1 and E_2 must be compatible and ordered	bool
E && E_2	E_1 and E_2 must be bool	bool
E || E_2	E_1 and E_2 must be bool	bool
! E	E must be bool	bool

7. Example :

Consider the declarations :

```
int a, b;   float x;   char ch;   bool flag;
```

Expression	Valid / Invalid	Reason
a + b	Valid	Both are int
a + x	Valid	int and float → numeric
a + ch	Invalid	int and char not compatible
a / b	Valid	Both are int
a / ch	Invalid	char not numeric
flag && (a < b)	Valid	a < b gives bool
flag + b	Invalid	bool and int not compatible

8. Type Checking in Assignments :

If T_1 and T_2 are the types of E_1 and E_2 in the statement

$$E_1 = E_2$$

then assignment is valid if T_1 and T_2 are compatible.

Example :

- int a; float x; a = x; → Valid (implicit conversion)
- float y; int b; y = b; → Valid (implicit conversion)
- int a; char *p; a = p; → Invalid (incompatible)

9. Type Errors Detected :

- Undeclared variable used.
- Incompatible types in operations.
- Wrong number or type of function arguments.
- Return type mismatch in functions.
- Invalid type conversions.

Important Points for Exams

★ Type checking is a static semantic check.
★ It ensures type safety and prevents runtime errors.
★ Performed after parsing and before code generation.
★ Helps in optimization and efficient code generation.
★ Different languages have different type systems (static / dynamic).

Summary

Type checking verifies that all operations in a program are applied to compatible types. It is done during semantic analysis of the compiler and helps in generating correct and efficient intermediate code.

2. Type Systems

A type system is a collection of rules that associates a type with various constructs of a programming language and defines the legal operations on those types.

1. Introduction :

A type system assigns a type to all language constructs (variables, constants, expressions, functions, etc.) and specifies the set of operations that are permissible on values of those types.
It helps in detecting type errors at compile time and ensures type safety, consistency and correctness of programs.

2. Need for Type Systems :

- Detects type errors at compile time.
- Prevents illegal operations (e.g., int + char*).
- Ensures program reliability and safety.
- Helps in optimization and code generation.
- Makes the language semantics precise and unambiguous.

3. Components of a Type System :

① Type Set : The set of all types supported by the language.
② Operation Set : The set of operations permitted on types.
③ Type Rules : Rules that assign a type to each construct.
④ Type Equivalence : Rules to determine compatibility between types.
⑤ Type Checking Rules : Rules that verify type correctness.

4. Classification of Type Systems :

(A) Based on Value Set :

Type System	Description	Example
Static Type System	Types are checked at compile time.	C, C++, Java, Pascal
Dynamic Type System	Types are checked at run time.	Python, JavaScript, Smalltalk
Hybrid Type System	Combination of static and dynamic checking.	C#, Ruby

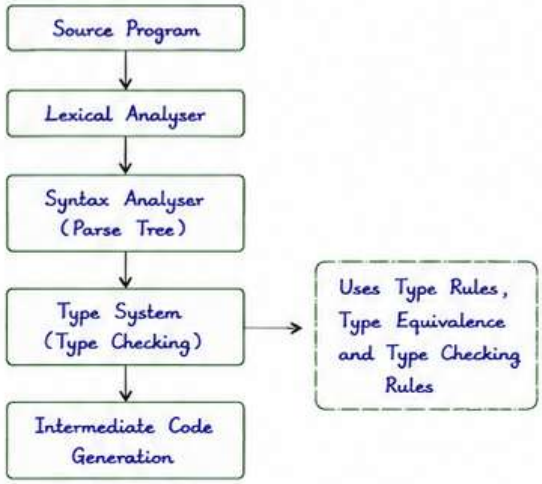
(B) Based on Type Checking Strictness :

Type System	Description	Example
Strong (Strict)	No implicit type conversion. All operations must be explicitly defined.	Java, Haskell
Weak (Lenient)	Implicit type conversions are allowed.	C (old style), JavaScript
Strong with Type Inference	Compiler infers types when possible.	ML, Scala, Kotlin

Important Points for Exams

★ A type system defines the set of types and legal operations.
★ Static type systems detect errors at compile time.
★ Dynamic type systems detect errors at run time.
★ Strong type systems do not allow implicit conversions.
★ Weak type systems allow implicit conversions.
★ Type systems ensure program correctness and safety.

5. Structure of a Type System :



6. Examples of Types in Languages :

Language	Type Categories	Examples
C	Basic, Derived, User-defined	int, float, array, struct
C++	Basic, Derived, User-defined, Class types	int, float, pointer, class, enum
Java	Primitive, Reference	int, float, boolean, String, Object
Python	Dynamic (All are objects)	int, float, str, list, dict

7. Type System Rules (General) :

① Every constant has a predefined type.
② Every variable must be declared with a type.
③ Every operator is defined for specific type combinations.
④ The result of an expression has a type.
⑤ Function arguments and return values have types.
⑥ Type compatibility must be checked in assignments and expressions.
⑦ Array elements are of the same type.

8. Example :

Consider the expression : a + b * c
If a, b, c are of type integer.

Type System Steps :

① b * c → operands are integers → result is integer
② a + (b * c) → operands are integers → result is integer
③ Hence, the expression has type integer.

Summary

Type systems assign types to language constructs and define legal operations on them. They help in detecting errors early, ensuring program correctness, and enabling efficient compilation. Types may be checked at compile time (static), run time (dynamic) or both (hybrid).

3. Type Expressions

A type expression is a syntactic construct used in a programming language to denote a type. It describes the type of objects and is used by the compiler to perform type checking and other semantic analysis.

1. Introduction :

Type expressions are the way to specify types in a program. They are built using basic type names and constructors (such as array, pointer, function, etc.). The compiler evaluates these expressions and determines the actual type.

2. Need for Type Expressions :

- To represent complex types using simple building blocks.
- To allow user-defined types and derived types.
- To support type checking and type equivalence.
- To make the language expressive and flexible.

3. Basic Type Symbols (Primitive Types) :

int, float, char, bool, void, double, long, etc.

4. Type Constructors (Operators on Types) :

① [] → Array constructor
② * → Pointer constructor
③ → → Function constructor
④ × (or ,) → Product (structure / pair) constructor
⑤ union → Union constructor
⑥ ( ) → Parentheses for grouping

5. Formation of Type Expressions :

Type expressions are formed by applying type constructors to type names or to other type expressions.

T ::= B	(B is a basic type)
| T [n]	(Array of n elements of type T)
| T *	(Pointer to T)
| $T_1 \rightarrow T_2$	(Function from T_1 to T_2)
| (T_1 , T_2 , … , T_n)	(Product / Structure type)
| union { T_1 , T_2 , … , T_n }	(Union type)

6. Examples of Type Expressions :

① int → basic type
② float * → pointer to float
③ char [20] → array of 20 chars
④ int * [10] → array of 10 pointers to int
⑤ float → int → function taking float and returning int
⑥ (int , float) → pair (product) of int and float
⑦ union { int , float } → union of int and float
⑧ int (*[5]) (float) → array (size 5) of pointers to functions taking float and returning int

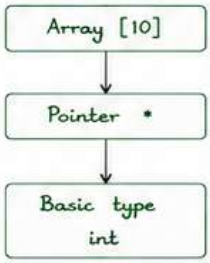
7. Classification of Type Expressions :

Category	Description	Examples
Simple (Basic)	Consist of basic type names.	int, float, char, bool
Derived	Built using constructors on basic types.	int *, float [10], (int, float), etc.
User-defined	Defined by the programmer using type declarations.	struct student, enum color, typedef
Complex	Combination of derived and user-defined types.	struct node *, int (*) (float), etc.

8. Representation (Abstractly) :

Type expressions can be represented as a type tree.

Example : int * [10]



This tree shows that we have an array of 10 elements where each element is a pointer to an int.

9. Equality of Type Expressions :

Two type expressions are equivalent if they denote the same type.

Examples :

① int * and int * → Equivalent
② int [10] and int [10] → Equivalent
③ int *[10] and (int *) [10] → Equivalent
④ int [10] * and int * [10] → Not Equivalent

10. Uses of Type Expressions :

- Used in variable declarations, function declarations.
- Used for parameter passing and return types.
- Used in type checking and type equivalence algorithms.
- Used in storage allocation and memory management.

Important Points for Exams

★ Type expressions describe types using constructors.
★ They are evaluated by the compiler during semantic analysis.
★ Constructors: array, pointer, function, product, union.
★ They help in building complex types from basic types.
★ Used in type checking, storage allocation and code generation.

Summary

Type expressions are syntactic forms used to represent types in a program. They are built from basic types using constructors like array, pointer, function, etc. The compiler interprets these expressions to perform semantic analysis such as type checking and type equivalence.

4. Type Equivalence

Two types are said to be equivalent if values of one type can be used in place of values of the other type without requiring an explicit type conversion.

1. Introduction :

Type equivalence is a relation used by the compiler to determine whether two types are compatible. If two types are equivalent, a value of one type can be assigned to a variable of the other type without any explicit conversion.

2. Need for Type Equivalence :

- Ensures type compatibility in assignments, expressions and function calls.
- Helps in detecting type errors at compile time.
- Avoids unnecessary type conversions.
- Supports user-defined types and complex structures.

3. Basic Rules of Type Equivalence :

① Two basic types are equivalent only if they are the same.
② Two derived types are equivalent if they are constructed using equivalent component types and the same structure.
③ The names of user-defined types must be identical.
④ The order of fields in structures and elements in arrays must be the same.
⑤ Type equivalence is reflexive, symmetric and transitive.

4. Equivalence of Basic Types :

Type 1	Type 2	Equivalent ?	Reason
int	int	Yes	Same basic type
float	float	Yes	Same basic type
char	char	Yes	Same basic type
int	float	No	Different basic types
int	char	No	Different basic types

5. Equivalence of Derived Types :

(A) Arrays :

Two arrays are equivalent if
→ they have the same number of dimensions
→ the size in each dimension is the same
→ the element types are equivalent

- `int A[10]` and `int B[10]` → Equivalent
- `float A[5][20]` and `float B[5][20]` → Equivalent
- `int A[10]` and `float B[10]` → Not Equivalent
- `int A[10][20]` and `int B[20][10]` → Not Equivalent

(B) Pointers :

Two pointer types are equivalent if their pointed-to types are equivalent.

- `int *p` and `int *q` → Equivalent
- `float *p` and `float *q` → Equivalent
- `int *p` and `float *q` → Not Equivalent

(C) Functions :

Two function types are equivalent if
→ they have the same return type
→ they have the same number of parameters
→ corresponding parameter types are equivalent

6. Equivalence of Structures (Records) :

Two structures are equivalent if
(i) they have the same number of fields
(ii) the corresponding fields have equivalent types
(iii) the fields are declared in the same order.

Example :

```
struct S1
{
   int a;
   float b;
   char c;
};
```

$S1 \equiv S2$ (Equivalent)

```
struct S2
{
   int a;
   float b;
   char c;
};
```

```
struct S3
{
   int a;
   char c;
   float b;
};
```

$S3 \neq S4$ (Not Equivalent)

```
struct S4
{
   int a;
   float b;
   char c;
};
```

(Order of fields is different)

7. Equivalence Summary :

Type Category	Conditions for Equivalence	Example (Equivalent)	Example (Not Equivalent)
Basic	Both must be the same basic type	int and int	int and float
Arrays	Same dimensions, same size in each dimension, same element type	int A[10] and int B[10]	int A[10] and float B[10]
Pointers	Pointed-to types must be equivalent	int *p and int *q	int *p and float *q
Structures	Same number of fields, same order, corresponding fields equivalent	S1 and S2 (same order)	S1 and S3 (different order)
Functions	Same return type, same number of parameters, equivalent parameter types	f(int,float) and f(int,float)	f(int,float) and f(float,int)
Unions	Same number of fields, corresponding fields equivalent	U1 and U2 (same fields)	U1 and U3 (different fields)

Important Points for Exams

★ Type equivalence is used for compatibility checking.
★ Names of user-defined types must match exactly.
★ For structures, order of fields is important.
★ Arrays must have same size in every dimension.
★ Pointers are equivalent if and only if their pointed-to types are equivalent.

Summary

Type equivalence determines whether two types can be used interchangeably without explicit type conversion. It plays a vital role in assignments, operations and function calls. The compiler uses equivalence rules to ensure type safety and correct program behavior.

5. Type Conversion

Type conversion is the process of converting a value of one type into another type. It is done either implicitly by the compiler or explicitly by the programmer.

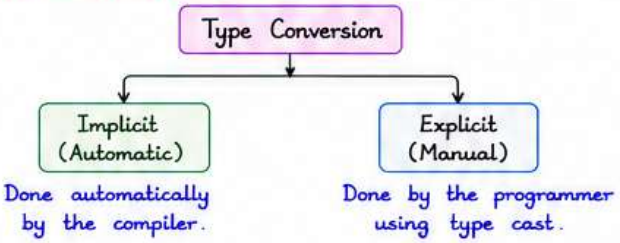
1. Introduction :

Type conversion allows values of one type to be used where values of another type are expected. This helps in expression evaluation, function calls, assignments, etc., without causing type errors.

2. Need for Type Conversion :

- To perform operations between different types.
- To match function parameter types.
- To assign values of one type to another.
- To ensure program flexibility and correctness.
- To support type compatibility rules.

3. Types of Type Conversion :



4. Implicit (Automatic) Type Conversion :

- Also called implicit conversion or coercion.
- Done automatically by the compiler.
- Occurs when no data is lost.
- Example :

```
int a = 10;
float b = 3.5;
float c = a + b; // int 'a' implicitly converted to float
```

Here, int 'a' is automatically converted to float.

5. Explicit (Manual) Type Conversion :

- Also called explicit conversion or type casting.
- Done by the programmer.
- Syntax : `(type) expression`
- Example :

```
float f = 10.75;
int x = (int) f; // float explicitly converted to int
```

6. Type Conversion Rules (General) :

① Smaller type can be converted to larger type (safe). e.g., int → float
② Larger type to smaller type may cause data loss. e.g., float → int
③ Conversion between incompatible types gives error. e.g., int → string (not allowed implicitly)
④ User-defined types require explicit conversion.

Important Points for Exams

★ Implicit conversion is automatic and safe.
★ Explicit conversion is done using cast operator.
★ Widening conversion (smaller → larger) is safe.
★ Narrowing conversion (larger → smaller) may lose data.
★ Type conversion is important for expressions, assignments, and function calls.

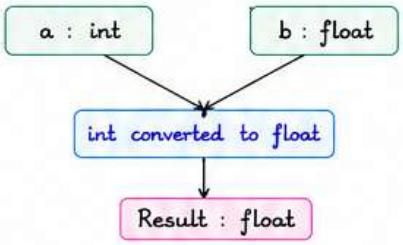
7. Classification of Type Conversion :

Type	Description	Example
Widening Conversion	Converting smaller type to larger type.	int → float char → int
Narrowing Conversion	Converting larger type to smaller type.	float → int double → float
User-defined Conversion	Conversion between user-defined types.	class A → class B (using constructor)
Implicit Conversion	Done automatically by compiler.	int → float char → int
Explicit Conversion	Done by programmer using cast.	(int) 3.14 (float) 10

8. Type Conversion in Expressions :

When mixed types are used in an expression, the compiler converts the smaller type to larger type.

Example : a + b



Rule :
Result type = Larger type

9. Example Programs :

(A) Implicit Conversion

```
int a = 10;
float b = 2.5;
float c = a + b; // int → float
printf("%f", c);
```

(B) Explicit Conversion

```
float f = 9.87;
int x = (int) f; // float → int
printf("%d", x);
```

10. Conversion Hierarchy (from low → high)

char ⇨ short ⇨ int ⇨ long ⇨ float ⇨ double

(Compiler prefers converting to higher type in expressions)

11. Conversion Between User-defined Types :

- Done using constructors, operator overloading, or explicit cast functions.
- Example : class A → class B (using constructor of B)

Summary

Type conversion helps in using one type in place of another type. It can be implicit (automatic) or explicit (manual). It ensures correct evaluation of expressions, function calls, assignments and improves flexibility of programs.

6. Polymorphic Functions

Polymorphic functions are functions that can operate on arguments of different types. The same function name can work for multiple types, making the language more flexible and reusable.

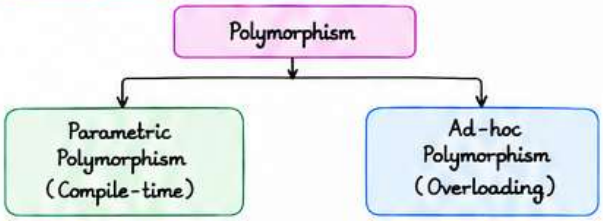
1. Introduction :

Polymorphism means "many forms". A polymorphic function behaves differently for different types of arguments. It allows the same function to be used with different data types without writing separate functions for each type.

2. Need for Polymorphic Functions :

- Reduces code duplication.
- Increases code reusability and readability.
- Makes programs compact and flexible.
- Works well with user-defined and built-in types.
- Essential for large programs and libraries.

3. Types of Polymorphism :



Same function works for all types.

Same function name with different parameter types.

4. Parametric Polymorphism (Generic Functions) :

- The function is written in a generic form.
- Works for any data type.
- Type is determined at the time of use.
- Supported in languages like C++ (templates), ML, Haskell.

Example (C++ Template) :

```
template <typename T>
T maxValue(T a, T b) {
    return (a > b) ? a : b;
}
// Works for int, float, char, etc.
```

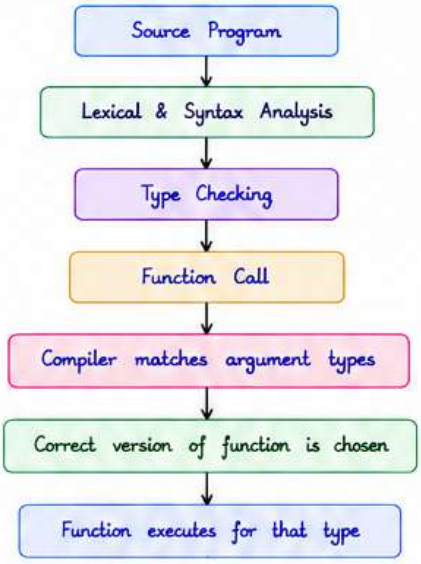
5. Ad-hoc Polymorphism (Function Overloading) :

- Same function name, different parameter types.
- Compiler chooses the correct version based on arguments.
- It is a form of polymorphism often called overloading.

Example (C++ Overloading) :

```
int add(int a, int b) { return a + b; }          // for int
float add(float a, float b) { return a + b; }    // for float
```

6. Working of Polymorphic Functions :



7. Example in Different Languages :

Language	Feature Used	Example
C++	Templates / Overloading	template <class T> / int add(), float add()
Java	Method Overloading	add(int, int) add(float, float)
Python	Dynamic Typing	def add(a, b): return a + b # works for int, float, str
Haskell	Type Classes	+ :: Num a => a -> a -> a

8. Advantages :

- ✓ Single function works for multiple types.
- ✓ Reduces redundant code.
- ✓ Easier maintenance and updates.
- ✓ Supports user-defined and built-in types.
- ✓ Makes the program more flexible and readable.

9. Disadvantages / Limitations :

- ✗ Can make compiler design more complex.
- ✗ Error messages may become difficult to understand.
- ✗ Overuse of overloading may reduce program clarity.
- ✗ Some languages (like C) do not directly support polymorphism.

Important Points for Exams

- ★ Polymorphic functions improve code reusability.
- ★ There are two types → Parametric and Ad-hoc polymorphism.
- ★ Generic functions use type parameters (templates).
- ★ Overloading allows same function name with different types.
- ★ Compiler selects the correct version based on argument types.

Summary

Polymorphic functions allow the same function to operate on arguments of different types. They help in code reusability, flexibility and compactness. Parametric polymorphism uses generic functions, while ad-hoc polymorphism uses function overloading. It is an important concept in modern programming languages and is widely used in compiler design.

7. Run Time Environment

Run Time Environment (RTE) is the set of components and facilities that are needed to support the execution of a program. It provides services to the running program such as memory management, storage allocation, procedure calls, exception handling, I/O operations etc.

1. Introduction :

After the program is compiled and loaded in memory, it executes with the help of run time environment. It acts as an interface between the compiled code and the system resources.

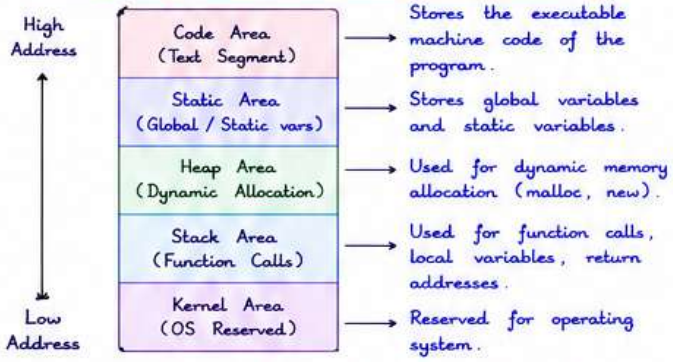
2. Need for Run Time Environment :

- To manage memory during program execution.
- To support function calls and parameter passing.
- To handle dynamic storage allocation and deallocation.
- To perform input-output operations.
- To handle run time errors and exceptions.
- To maintain program state and control information.

3. Components of Run Time Environment :

Component	Description
Memory Management	Allocates and deallocates memory for variables, arrays, objects, stack, heap etc.
Storage Organization	Organizes data in memory (stack, heap, static area, code area).
Procedure Call Mechanism	Handles function calls, parameter passing, return addresses and activation records.
I/O Management	Provides run time support for input-output operations.
Run Time Error Handling	Detects and handles errors like divide by zero, overflow, null pointer etc.
System Interface	Provides interface to OS services like file handling, process management etc.

4. Memory Organization in Run Time Environment :



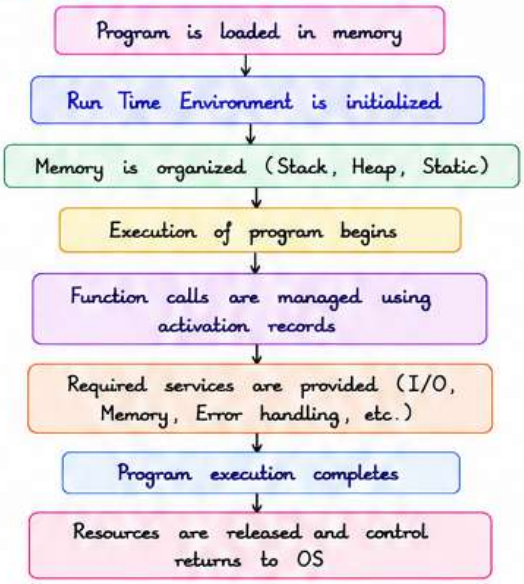
5. Activation Records (Run Time Stack Frames) :

When a function is called, an activation record (also called stack frame) is created on the stack. It stores all information needed for that function.

Typical fields in an activation record :

- Actual parameters
- Return address
- Control link (previous frame pointer)
- Access link (for nested functions)
- Local variables
- Temporary data

6. Working of Run Time Environment :



7. Services Provided by Run Time Environment :

- Memory allocation and deallocation.
- Procedure call and return handling.
- Parameter passing and value returning.
- Input and output operations.
- Dynamic storage allocation (heap management).
- Run time error detection and handling.
- Program termination and resource release.

8. Example :

Consider a C program with a function call :

```
int add(int a, int b);
int main() {
   int x = 10, y = 20, z;
   z = add(x, y);
   return 0;
}

int add(int a, int b) {
   int c = a + b;
   return c;
}
```

When add(x, y) is called :

- An activation record for add() is created on the stack.
- Actual parameters x and y are passed to a and b.
- Local variable c is allocated in the stack frame of add().
- After computation, value of c is returned to main().
- Activation record is removed and control returns to main().

9. Importance :

- Ensures efficient and correct execution of programs.
- Manages memory and resources effectively.
- Provides flexibility through dynamic memory and polymorphism.
- Supports modularity and code reusability.
- Handles errors gracefully during execution.

Important Points for Exams

- Run Time Environment supports program execution.
- Main components include memory management, storage organization, procedure calls, I/O and error handling.
- Stack is used for function calls and local variables.
- Heap is used for dynamic memory allocation.
- Activation record is created for every function call.
- RTE provides interface between program and OS.

Summary

Run Time Environment is essential for executing a program. It provides services like memory management, storage allocation, I/O, procedure calls and error handling. It organizes memory into code, static, heap and stack areas. Without RTE, the compiled program cannot execute properly.

8. Storage Organization

Storage Organization refers to the way in which memory is organized and managed during the execution of a program. It decides how the storage is allocated, used and released for various program components such as variables, functions, parameters, etc.

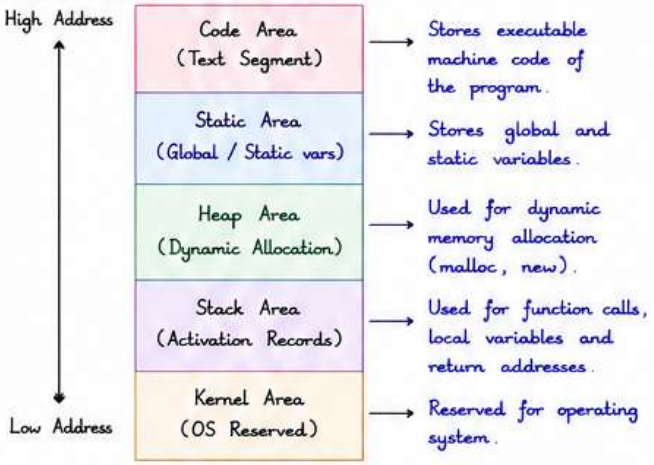
1. Introduction :

The compiler and run time environment together manage memory for storing code, data, variables, intermediate results, activation records etc. Storage is usually divided into different areas with specific purposes.

2. Need for Storage Organization :

- Efficient use of memory.
- Support for recursive and nested function calls.
- Fast allocation and deallocation of memory.
- Avoid memory wastage and fragmentation.
- Provide scope rules for variables.

3. Memory Areas in a Program :



4. Description of Memory Areas :

① **Code Area (Text Segment) :**
Contains the compiled machine instructions of the program. Usually read-only. Loaded when the program starts.

② **Static Area :**
Stores global variables and static variables. These exist throughout the program execution. Memory is allocated at compile/load time.

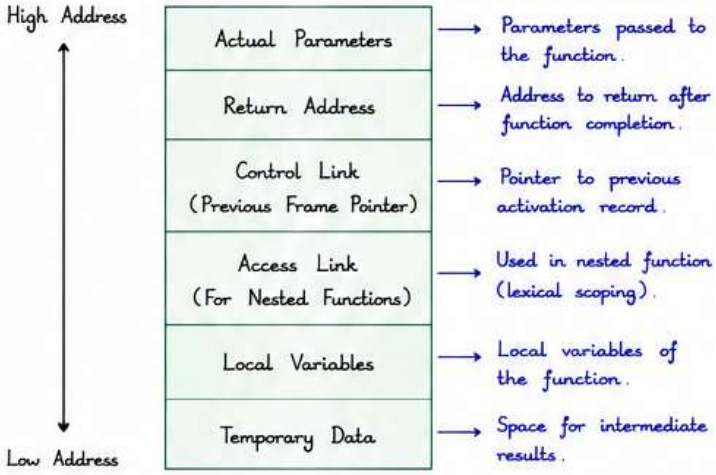
③ **Heap Area :**
Used for dynamic memory allocation during run time. Memory can be allocated and deallocated explicitly by programmer (using malloc(), free() in C or new, delete in C++). Heap grows upwards (towards higher addresses).

④ **Stack Area :**
Used for function calls and local variables. Each time a function is called, an activation record (stack frame) is pushed onto the stack. It is automatically removed when function returns. Stack grows downwards (towards lower addresses).

⑤ **Kernel Area :**
Lowest part of memory reserved for OS and system processes.

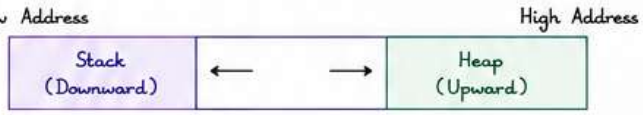
5. Stack Organization (Activation Records) :

When a function is called, an activation record (or stack frame) is created on the stack. It contains all information needed for that function.



6. Heap Organization :

- Heap is used for dynamic memory allocation.
- Memory is allocated using malloc(), new etc.
- Memory must be explicitly deallocated using free(), delete etc.
- Heap grows towards higher addresses.



7. Comparison :

Feature	Stack	Heap
Allocation	Automatic (by compiler)	Dynamic (by programmer)
Deallocation	Automatic	Explicit
Access Speed	Fast	Relatively slow
Size	Limited	Large (depends on memory)
Used For	Function calls, local vars	Dynamic memory allocation
Growth Direction	Downward (low address)	Upward (high address)

8. Design Issues in Storage Organization :

- How much storage to allocate?
- When to allocate and deallocate?
- How to handle nested scopes?
- How to avoid fragmentation?
- How to manage run time stack and heap efficiently?

Important Points for Exams

- ★ Memory is divided into code, static, heap, stack and kernel areas.
- ★ Stack is used for function calls and local variables.
- ★ Heap is used for dynamic memory allocation.
- ★ Stack grows downward, heap grows upward.
- ★ Static variables exist throughout program execution.

Summary

Storage organization defines how memory is divided and managed in different areas such as code, static, heap, stack and kernel. Stack is used for function calls and grows downward while heap is used for dynamic allocation and grows upward. Efficient storage organization is essential for correct and fast execution of a program.

9. Activation Record (Stack Frame)

An Activation Record (AR) is a block of data created on the run time stack when a function is called. It stores all the information needed for the function's execution and is removed when the function returns.

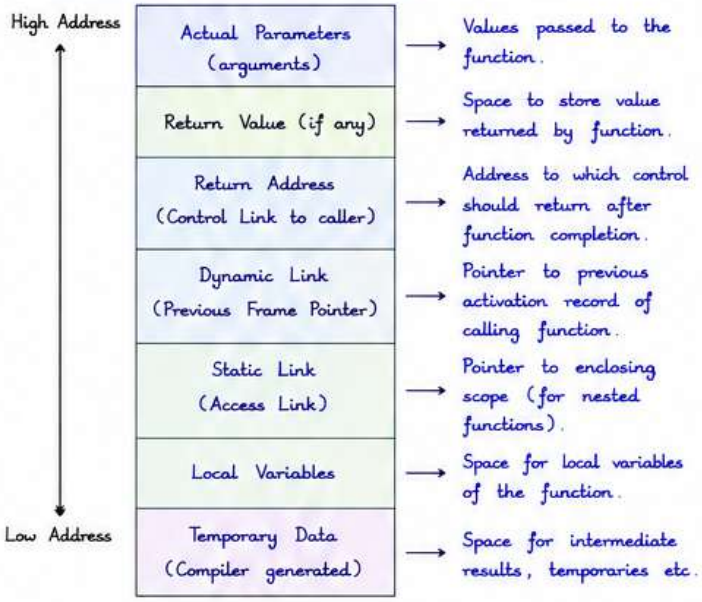
1. Introduction :

When a function is called, control transfers to that function. All details required for its execution are stored in an activation record (or stack frame) on the run time stack. When the function returns, the activation record is removed and control returns to the calling function.

2. Need for Activation Record :

- To support function calls and returns.
- To store parameters, local variables and return address.
- To save previous environment information (dynamic link).
- To support recursion and nested function calls.

3. Structure of Activation Record :



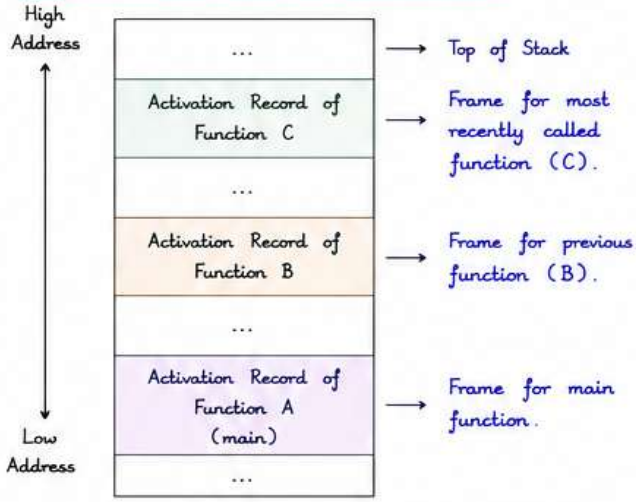
4. Explanation of Fields :

① Actual Parameters : Stores the values (or addresses) passed by the caller.
② Return Value : If the function returns a value, it is stored here before control returns to caller.
③ Return Address : The address of the next instruction in the calling function. After return, control goes to this address.
④ Dynamic Link : Points to the activation record of the calling function. Used to restore the previous environment.
⑤ Static Link : Points to the activation record of the lexically enclosing function. Used for non-local variable access.
⑥ Local Variables : Stores all local variables declared in the function.
⑦ Temporary Data : Used by compiler for evaluation of expressions, intermediate results etc.

Important Points for Exams

★ Activation record is also called stack frame.
★ It is created at function call time and destroyed at return time.
★ Stack grows from high address to low address.
★ Dynamic link is used for run time nesting (calling sequence).
★ Static link is used for lexical (compile time) nesting.
★ Activation records are essential for recursion and nested calls.

5. Run Time Stack with Activation Records :



6. Working of Activation Record :

① When a function is called, space is allocated on stack for its activation record.
↓
② Parameters are passed and copied into the frame.
↓
③ Return address, dynamic link and static link are saved.
↓
④ Local variables and temporaries are allocated.
↓
⑤ Function executes.
↓
⑥ On return, the activation record is removed and control returns to the address in return address field.

7. Example (Function Call) :

Consider : main() calls fun1(), which calls fun2().

(a) After call to fun1()

AR of fun1
Return Address (to main)
Dynamic Link (to main)
Parameters of fun1
Local Variables

(b) After call to fun2()

AR of fun2
Return Address (to fun1)
Dynamic Link (to fun1)
Parameters of fun2
Local Variables
AR of fun1
Return Address (to main)
Dynamic Link (to main)
Parameters of fun1
Local Variables

(c) After return from fun2()

AR of fun1
Return Address (to main)
Dynamic Link (to main)
Parameters of fun1
Local Variables

After return from fun1(), stack becomes empty and control goes back to main().

8. Importance of Activation Records :

- Helps in managing function calls and returns.
- Allows recursion.
- Supports nested functions.
- Maintains proper access to variables.
- Enables memory to be allocated and deallocated systematically.

Summary

An activation record is a run time data structure stored on the stack for each function call. It contains parameters, return address, dynamic link, static link, local variables and temporary data. It is created at call time and removed at return time, helping in proper execution and management of functions.

10. Stack Allocation

Stack Allocation is a method of memory management in which memory is allocated and deallocated from a stack for storing activation records, local variables, parameters, return addresses, etc. It follows LIFO (Last In First Out) order.

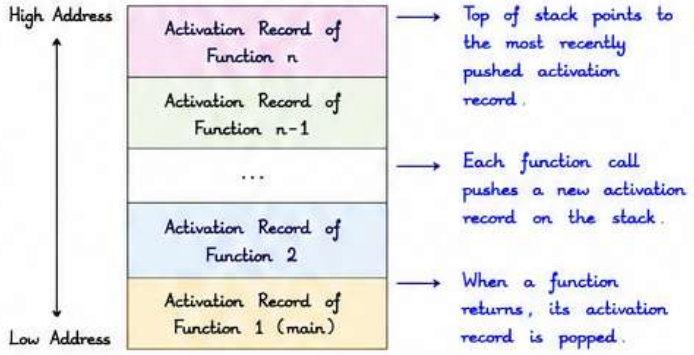
1. Introduction :

During program execution, functions are called and returned in a nested manner. Stack allocation uses a run time stack to manage memory for each function call. Memory for an activation record is pushed on the stack when a function is called and popped when the function returns.

2. Need for Stack Allocation :

- Efficient memory management for function calls and returns.
- Automatic allocation and deallocation of memory.
- Supports recursion and nested function calls.
- Requires less time compared to heap allocation.
- Provides data in LIFO order which matches function call order.

3. Run Time Stack Representation :

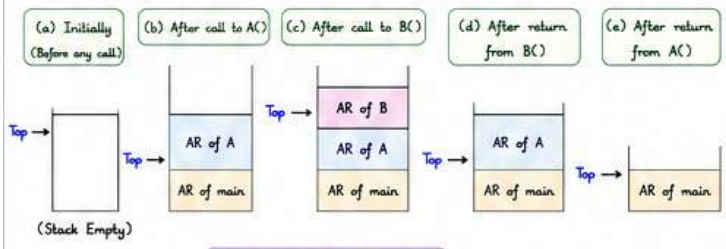


4. Working of Stack Allocation :

① When a function is called, space is allocated on the stack for its activation record.
② Parameters are passed and stored in the activation record.
③ Local variables and temporary data are allocated.
④ Control information like return address and dynamic link is saved.
⑤ Function executes using this activation record.
⑥ On return, the activation record is removed (popped) and control returns to the calling function.

5. Example (Function Calls) :

Consider the following sequence : main() calls A(), A() calls B().



AR = Activation Record

Important Points for Exams

★ Stack allocation uses run time stack.
★ Memory is allocated when a function is called and deallocated when it returns.
★ It follows LIFO (Last In First Out) order.
★ Supports recursion, nested calls and local variables.
★ Activation record contains all necessary information for a function.
★ Faster than heap allocation.

6. Structure of Activation Record (Using Stack Allocation) :

Actual Parameters (or Arguments)	→ Values passed to the function.
Return Value (if any)	→ Space for value returned.
Return Address	→ Address to which control returns after function completion.
Dynamic Link (Previous Frame Pointer)	→ Pointer to previous activation record of calling function.
Static Link (Access Link)	→ Pointer to lexically enclosing function (for nested functions).
Local Variables	→ Storage for local variables declared in the function.
Temporary Data	→ Space for intermediate results, temporaries, etc.

7. Stack Allocation vs Heap Allocation :

Feature	Stack Allocation	Heap Allocation
Memory Area	Run time stack	Heap
Alteration	Automatic (by compiler)	Dynamic (by programmer)
Deallocation	Automatic	Explicit (free / delete)
Order	LIFO (Last In First Out)	No specific order
Speed	Fast	Relatively slow
Fragmentation	No fragmentation	May cause fragmentation
Best Use	Function calls, local vars, parameters	Dynamic data structures, large objects

8. Advantages :

✓ Simple and efficient memory management.
✓ Automatic allocation and deallocation.
✓ Supports recursion and nested function calls.
✓ Less memory overhead.
✓ Fast allocation and deallocation.

9. Disadvantages :

✗ Limited size (stack overflow may occur).
✗ Suitable only for data with limited lifetime.
✗ Cannot store large or long-term data structures.

10. Applications :

- Storing activation records for function calls.
- Managing local variables and parameters.
- Implementing recursion and nested functions.
- Used in almost all modern programming languages.

Summary

Stack allocation is a memory management technique in which memory for activation records is taken from a run time stack. It allocates memory when a function is called and releases it when the function returns. It follows LIFO order, is fast, and supports recursion and nested function calls.

11. Heap Allocation

Heap Allocation is a memory management technique in which memory is allocated and deallocated from a heap during run time. The programmer requests memory explicitly using functions like malloc(), new, calloc(), etc. and releases it using free(), delete, etc. Heap does not follow any strict order like stack.

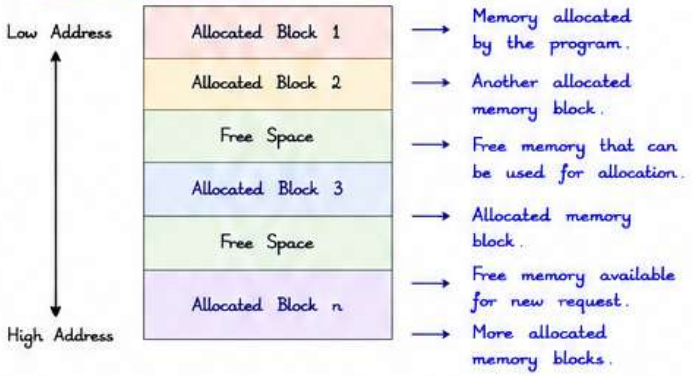
1. Introduction :

Heap is a region of memory in the data segment used for dynamic memory allocation. It is larger than stack and growth is towards higher addresses. Memory remains allocated until it is explicitly released by the program.

2. Need for Heap Allocation :

- To allocate memory whose size is known only at run time.
- To create large data structures (arrays, lists, trees, graphs, etc.).
- Memory remains allocated until explicitly deallocated.
- Supports dynamic data structures.
- Less chance of stack overflow.

3. Heap Organization :



4. Working of Heap Allocation :

① When memory is required, program requests memory from heap using malloc(), new, calloc(), etc.
② Memory manager searches for a free block of sufficient size.
③ If available, memory is allocated and a pointer to that block is returned to the program.
④ If not available, heap may be expanded (if possible) and new memory is allocated.
⑤ The allocated memory is used by the program.
⑥ When memory is no longer needed, it is released explicitly using free(), delete, etc. The memory becomes available for future use.

5. Example (C Program) :

```c
#include <stdio.h>
#include <stdlib.h>

int main() {
    int *arr, n, i;

    printf("Enter size of array: ");
    scanf("%d", &n);

    // Heap allocation
    arr = (int *)malloc(n * sizeof(int));

    if (arr == NULL) {
        printf("Memory allocation failed!\n");
        return 1;
    }
    printf("Enter elements:\n");
    for (i = 0; i < n; i++)
        scanf("%d", &arr[i]);

    printf("\nElements are: \n");
    for (i = 0; i < n; i++)
        printf("%d ", arr[i]);

    free(arr);          // Release memory
    return 0;
}
```

Explanation:
Memory is taken from heap using malloc() and released using free().

6. Heap vs Stack Allocation :

Feature	Heap Allocation	Stack Allocation
Memory Area	Heap	Run time stack
Allocation	Dynamic (by programmer)	Automatic (by compiler)
Deallocation	Explicit (free/delete)	Automatic (on return)
Order	No strict order	LIFO (Last In First Out)
Speed	Relatively slow	Fast
Size	Large (limited by system memory)	Limited (defined by system)
Fragmentation	May cause fragmentation	No fragmentation
Access	Indirect (through pointer)	Direct
Best Use	Dynamic data structures, large objects	Function calls, local vars, parameters

7. Advantages of Heap Allocation :

- ✓ Allows dynamic memory allocation.
- ✓ Supports large and complex data structures.
- ✓ More flexible (memory can be allocated/deallocated anytime).
- ✓ Memory remains until explicitly freed.
- ✓ Less risk of stack overflow.

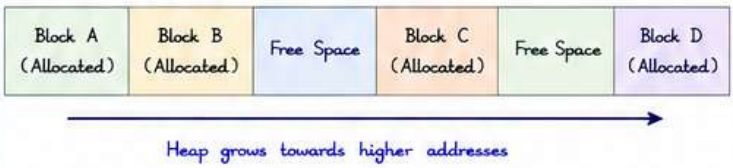
8. Disadvantages of Heap Allocation :

- ✗ Slower than stack allocation.
- ✗ Memory may get fragmented.
- ✗ Programmer must manage memory manually (free/delete).
- ✗ Memory leaks possible if memory is not freed.
- ✗ More complex to use.

9. Applications :

- Dynamic data structures (linked lists, trees, graphs, queues, stacks).
- Handling large data objects.
- Memory used throughout program execution.
- Database systems, compilers, operating systems.
- Any situation where size is not known until run time.

10. Heap Memory Example (Diagram) :



Important Points for Exams

- ★ Heap is used for dynamic memory allocation.
- ★ Memory is allocated using malloc(), new, calloc() etc.
- ★ Memory is released using free(), delete etc.
- ★ Heap does not follow LIFO order.
- ★ Heap is slower than stack but more flexible.

Summary

Heap allocation is a dynamic memory management technique used to allocate large and variable size memory blocks during run time. Memory is allocated explicitly by the programmer and must be released manually. It provides flexibility but is slower than stack.

12. Parameter Passing

Parameter Passing is the mechanism of passing values (actual parameters / arguments) from the calling function to the called function during a function call.
The way parameters are passed affects the function's behaviour and the values in the calling function.

1. Introduction :

When a function is called, some information must be passed to it so that it can perform the required task. This information is passed through parameters. There are mainly two methods of parameter passing.

- Call by Value
- Call by Reference

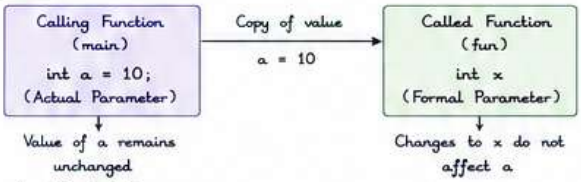
2. Call by Value (Pass by Value) :

- In this method, a copy of the actual parameter is passed to the called function.
- Any changes made to the parameter inside the called function do not affect the actual parameter in the calling function.
- It is the default method in C.

Working :

① Value of actual parameter is copied to formal parameter.
② Changes made to formal parameter affect only the local copy.
③ Actual parameter remains unchanged.

Diagram :



Example (C Code) :

```c
#include <stdio.h>
void fun(int x)
{
    x = x + 5;      // changes only local copy
    printf("Inside fun(), x = %d\n", x);
}
int main()
{
    int a = 10;
    printf("Before call, a = %d\n", a);
    fun(a);          // call by value
    printf("After call, a = %d\n", a);
    return 0;
}
```

Output :
Before call, a = 10
Inside fun(), x = 15
After call, a = 10

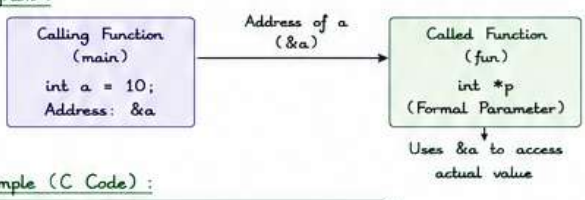
3. Call by Reference (Pass by Reference / Call by Address) :

- In this method, the address of the actual parameter is passed to the called function.
- Any changes made to the parameter inside the called function affect the actual parameter in the calling function.
- It is achieved in C using pointers.

Working :

① Address of actual parameter is passed to formal parameter (pointer).
② Formal parameter uses this address to access the actual value.
③ Changes made inside the called function affect the original value.

Diagram :



Example (C Code) :

```c
#include <stdio.h>
void fun(int *p)
{
    (*p) = (*p) + 5;     // changes actual value
    printf("Inside fun(), *p = %d\n", *p);
}
int main()
{
    int a = 10;
    printf("Before call, a = %d\n", a);
    fun(&a);          // call by reference
    printf("After call, a = %d\n", a);
    return 0;
}
```

Output :
Before call, a = 10
Inside fun(), *p = 15
After call, a = 15

Important Points for Exams

- ★ Define parameter passing and its types.
- ★ Draw diagrams for call by value and call by reference.
- ★ Write differences between call by value and call by reference.
- ★ Write C programs for both methods.

4. Comparison between Call by Value and Call by Reference :

Feature	Call by Value	Call by Reference
What is passed	Copy of actual value	Address of actual value
Memory used	More (for copy)	Less
Speed	Slower	Faster
Effect on actual parameter	No change	Changes are reflected
Safety	Safer	Less safe (can modify original data)
Use	When actual data should not be changed	When function needs to modify actual data

5. Example to Understand :

Call by Value

```c
void swap(int x, int y)
{
   int temp = x;
   x = y;
   y = temp;
}

int main()
{
   int a = 5, b = 10;
   swap(a, b);
   printf("a = %d, b = %d", a, b);
}
```

Output : a = 5, b = 10
(No change in a and b)

Call by Reference

```c
void swap(int *x, int *y)
{
   int temp = *x;
   *x = *y;
   *y = temp;
}

int main()
{
   int a = 5, b = 10;
   swap(&a, &b);
   printf("a = %d, b = %d", a, b);
}
```

Output : a = 10, b = 5
(Values are swapped)

6. Parameter Passing in Other Languages :

- C / C++ – Pass by value (default), Pass by reference (using pointers / references)
- Java – Always pass by value (object reference is also passed by value)
- Python – Pass by object reference (immutable objects behave like value)
- Pascal – Both are supported

7. Advantages and Disadvantages :

Method	Advantages	Disadvantages
Call by Value	• Simple to implement • Safer (actual data is not modified) • Easy to understand	• Extra memory is used • Large data copying makes it slower • Cannot modify actual data
Call by Reference	• No extra copying • Faster execution • Can modify actual data inside function	• Less safe • Less easy to understand • Wrong operations may change original data

8. Important Points :

- ★ Parameter passing is essential for communication between functions.
- ★ Default in C is Call by Value.
- ★ Call by Reference is achieved using pointers.
- ★ Choose method based on requirement (safety vs efficiency).

Summary

Parameter passing allows data to be transferred from the calling function to the called function. In Call by Value, a copy of the value is passed, so changes do not affect the original value. In Call by Reference, the address is passed, so changes affect the original value. Call by reference is faster and more efficient, while call by value is safer and easier.

13. Dynamic Storage Allocation

Dynamic Storage Allocation (DSA) is a run time memory management technique in which memory is allocated from a heap during program execution as and when required, and returned back to the heap when it is no longer needed. This memory allocation and deallocation is performed explicitly by the program.

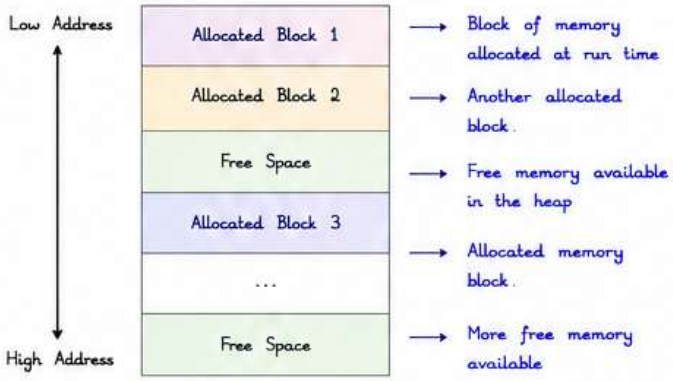
1. Introduction :

In some cases, the size of data structures (arrays, lists, trees, graphs, etc.) is not known at compile time. In such situations, memory is allocated dynamically at run time. Dynamic storage allocation provides flexibility and efficient memory utilization.

2. Need for Dynamic Storage Allocation :

- Size of data is not known until run time.
- Efficient utilization of memory (allocate only as much as needed).
- Supports dynamic data structures (linked lists, trees, stacks, queues).
- Reduces memory wastage.

3. Heap Organization :



4. Working of Dynamic Storage Allocation :

① When memory is required, the program requests heap memory using functions like malloc(), calloc(), etc.
② Memory manager searches for a free block of sufficient size.
③ If available, memory is allocated and a pointer to that block is returned.
④ If not available, heap may be expanded (if possible) and memory is allocated.
⑤ When memory is no longer needed, it is released using free().
⑥ The released memory becomes available for future use.

5. Functions for Dynamic Storage Allocation (C) :

Function	Purpose	Syntax	Header File
malloc()	Allocates a block of memory of specified size	void *malloc(size_t size);	<stdlib.h>
calloc()	Allocates memory for an array and initializes to zero	void *calloc(size_t n, size_t size);	<stdlib.h>
realloc()	Changes the size of previously allocated memory	void *realloc(void *ptr, size_t new_size);	<stdlib.h>
free()	Deallocates the previously allocated memory	void free(void *ptr);	<stdlib.h>

Important Points :

★ malloc() returns a pointer to the allocated memory.
★ If allocation fails, NULL is returned.
★ Always check the returned pointer against NULL.
★ Memory allocated using malloc/calloc must be released using free().
★ After free(), the pointer becomes a dangling pointer.
★ Use realloc() carefully; if it fails, original block remains unchanged.

6. Example Program (C) :

```c
#include <stdio.h>
#include <stdlib.h>

int main()
{
   int *arr, n, i;

   printf("Enter number of elements: ");
   scanf("%d", &n);

   arr = (int *)malloc(n * sizeof(int)); // allocate memory
   if (arr == NULL)
   {
      printf("Memory allocation failed!\n");
      return 1;
   }

   printf("Enter elements:\n");
   for (i = 0; i < n; i++)
      scanf("%d", &arr[i]);

   printf("Elements are:\n");
   for (i = 0; i < n; i++)
      printf("%d ", arr[i]);

   free(arr);   // deallocate memory
   return 0;
}
```

Output :

```
Enter number of elements: 5
Enter elements:
10 20 30 40 50
Elements are:
10 20 30 40 50
```

7. Comparison: Static vs Dynamic Allocation :

Feature	Static Allocation	Dynamic Allocation
Memory	Allocated at compile time	Allocated at run time
Size	Fixed size	Can grow or shrink
Flexibility	Less flexible	More flexible
Memory Use	May waste memory	Efficient memory usage
Allocation	Automatic	Explicit (by programmer)
Deallocation	Automatic	Explicit (using free())
Implementation	Simple	More complex
Best Used For	Simple, fixed size data	Dynamic data structures

8. Advantages :

✓ Memory is allocated only when required.
✓ Efficient memory utilization (no large unused blocks).
✓ Supports dynamic data structures.
✓ Allows programs to handle large data.
✓ Provides more flexibility to the programmer.

9. Disadvantages :

✗ Slower than static allocation (overhead of allocation/deallocation).
✗ Memory leaks possible if free() is not used.
✗ Fragmentation of heap memory may occur.
✗ Programmer must manage memory carefully.

10. Applications :

- Dynamic data structures (linked lists, trees, graphs).
- Arrays whose size is not known at compile time.
- Database systems, compilers, operating systems.
- Memory management in modern programming languages.

Summary

Dynamic storage allocation allocates memory from the heap at run time using functions like malloc(), calloc(), realloc() and free(). It provides flexibility and efficient memory usage but requires careful management by the programmer.

14. Symbol Table

A Symbol Table is a data structure used by the compiler to store information about identifiers (variables, constants, functions, arrays, etc.) encountered in the source program. It is created in the lexical analysis phase and used throughout compilation.

1. Introduction :

During compilation, many identifiers appear in the source program. The compiler needs to store information about each identifier such as its name, type, scope, value, memory location, etc. This information is stored in a data structure called Symbol Table.

2. Need for Symbol Table :

- Stores information about all identifiers in the program.
- Helps in detecting errors (undeclared, multiply declared, type mismatch).
- Provides information for code generation.
- Supports scope handling in block structured languages.
- Improves efficiency of compilation.

3. Information Stored in Symbol Table :

Field	Description
Name	Name of the identifier
Type	Data type (int, float, char, etc.)
Kind	Variable, constant, function, array, etc.
Scope	Scope level where it is declared
Address	Memory location or offset
Size	Size in bytes
Value	Initial value (for constants)
Other Info.	Parameter list (for functions), return type, dimensions (for arrays), etc.

4. Example :

Consider the following program snippet :

```
int x, y;
float area;
int add(int a, int b)
{
   int sum;
   sum = a + b;
   return sum;
}
```

Symbol Table

Name	Type	Kind	Scope	Address
x	int	variable	1	1000
y	int	variable	1	1004
area	float	variable	1	1008
add	function		1	1012
a	int	parameter	2	-
b	int	parameter	2	--
sum	int	variable	2	2000

5. Operations on Symbol Table :

- → Insert() : Add new entry when an identifier is encountered.
- → Lookup() : Search for an identifier in the table.
- → Delete() : Remove an entry (if scope ends).
- → Update() : Modify existing entry (if needed).

6. Implementation of Symbol Table :

Symbol Table can be implemented using the following data structures :

1. Linear List (Array) – Simple but slow search (O(n)).
2. Linked List – Easy insertion/deletion, search is O(n).
3. Hash Table – Fast search, average time O(1).
4. Binary Search Tree – Search, insert and delete in O(logn).
5. Balanced Tree (AVL, Red-Black Tree) – Efficient operations.

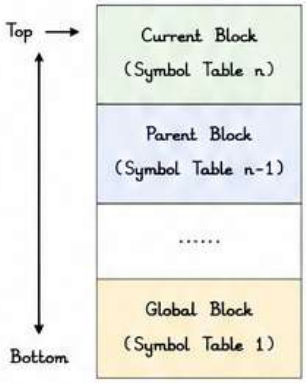
Important Points for Exams

- ★ Created during lexical analysis phase.
- ★ One symbol table may be used for entire program or multiple symbol tables for different scopes.
- ★ Helps in semantic analysis and intermediate code generation.
- ★ Efficient implementation improves compiler performance.

7. Handling Scope with Symbol Table :

Block structured languages use nested scopes. This can be handled using Stack of Symbol Tables.

- When a new block is entered, a new symbol table is pushed on stack.
- When a block is exited, the top symbol table is popped.
- Lookup starts from top and moves down the stack.



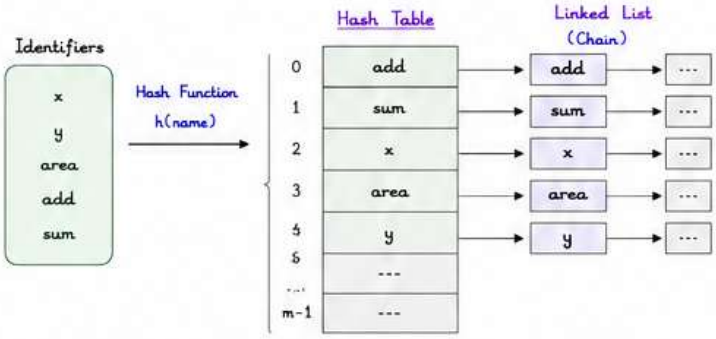
Example

```
{
   int a;
   {
      float b;
      {
         int c;
      }
   }
}
```

3 Symbol Tables will be created and managed using stack (Bottom → Top).

8. Symbol Table Implementation using Hash Table :

- Hash function h(name) is used to compute index.
- If two names have same index, collision occurs.
- Collision can be handled using chaining or open addressing.



9. Advantages and Disadvantages :

Advantages	Disadvantages
✓ Fast access to identifier information.	✗ Extra memory is required.
✓ Helps in error detection.	✗ Complex to implement for nested scopes.
✓ Supports efficient code generation.	✗ Performance depends on data structure used.
✓ Manages scopes in block structured languages.	
✓ Improves overall compiler efficiency.	

10. Applications :

- Used in lexical analysis, syntax analysis and semantic analysis.
- Stores variable, constant, function, array and type information.
- Used in intermediate code generation and optimization.
- Essential for block structured programming languages.

Summary

Symbol Table is an important data structure in compiler design that stores information about identifiers. It is used throughout compilation for error checking, scope handling and code generation. Efficient implementation (using hash table, tree, etc.) is essential for building a good compiler.

15. Symbol Table Operations

A Symbol Table supports a set of basic operations that allow the compiler to store, retrieve, update and delete information about identifiers efficiently. These operations are used repeatedly during all phases of compilation.

1. Introduction :

A Symbol Table stores information about identifiers (name, type, scope, address, value, etc.). The compiler performs various operations on the symbol table during compilation.
The main operations are :

- Insert, Lookup, Delete, Update
- BeginScope, EndScope

2. Information Stored in Symbol Table (Example) :

Field	Description
Name	Name of identifier
Type	Data type (int, float, char, array, etc.)
Kind	Variable, constant, function, array, etc.
Scope	Scope level at which it is declared
Address	Memory location or offset
Size	Size in bytes
Value	Initial value (for constants)
Other Info.	Parameter list, return type, dimensions, etc.

3. Symbol Table Operations :

Operation	Purpose	Input	Output
Insert()	Adds a new entry into the table	Identifier and its attributes	Success / Failure
Lookup()	Searches an identifier in the table	Identifier name	Pointer to entry / Not found
Delete()	Removes an entry from the table	Identifier name	Success / Failure
Update()	Modifies attributes of an existing entry	Identifier name and new info.	Success / Failure
BeginScope()	Starts a new scope (enter a block)	–	–
EndScope()	Ends current scope (exit a block)	–	–

4. Description of Operations :

- Insert() : Adds a new identifier with its attributes to the current symbol table.
- Lookup() : Searches for an identifier; returns its entry if present, otherwise reports error.
- Delete() : Removes an identifier from the symbol table.
- Update() : Changes an attribute (e.g., type, value, address) of an existing entry.
- BeginScope() : Creates a new scope level. New entries will be inserted in the current (inner) scope.
- EndScope() : Deletes all entries of the current scope and restores the previous scope.

5. Example :

Program :

```
int x;
void fun()
{
  int y;
  float z;
  y = x + 1;
}
float x;
```

Symbol Table (During Execution)

Step	Operation	Symbol Table Content (top → bottom)	Comment
1	Insert(x)	x (int)	Global x
2	BeginScope()	– x (int)	Enter fun() (global)
3	Insert(y)	y (int) x (int)	Insert y
4	Insert(z)	z (float) y (int) x (int)	Insert z
5	EndScope()	x (int)	Remove y and z
6	Insert(x)	x (float)	New x in global (hides old x)

Note :
When a scope ends, its entries are removed and control returns to the previous scope.

6. Implementation Techniques :

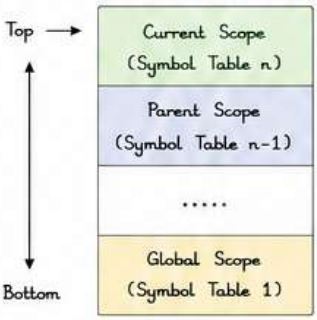
- Linear List / Array – Simple but slow search (O(n)).
- Linked List – Easy insertion/deletion, search is O(n).
- Hash Table – Fast search, average time O(1).
- Binary Search Tree – Search, insert and delete in O(log n).
- AVL / Red-Black Tree – Balanced tree for efficient operations.

7. Operations with Example (Summary Table) :

Operation	Example Call	Action Performed
Insert	Insert("x", int, scope 1, addr 1000)	Add x to current scope
Lookup	Lookup("x")	Search x and return info.
Update	Update("x", type = float)	Change type of x to float
Delete	Delete("x")	Remove x from table
BeginScope	BeginScope()	Create new scope level
EndScope	EndScope()	Remove entries of current scope

8. Stack of Symbol Tables (for Scoped Languages) :

- Each scope has its own symbol table.
- All symbol tables are maintained in a stack structure.
- Top of stack represents current scope.



Working :

1. Insert and Lookup are performed in the top table.
2. If not found, search moves down the stack.
3. When a scope ends, top table is popped.

9. Advantages and Disadvantages :

Advantages	Disadvantages
✓ Efficient storage and retrieval of identifier information.	✗ Extra memory is required.
✓ Helps in error detection (undeclared, multiple declared).	✗ Implementation can be complex (especially for nested scopes).
✓ Supports scope handling in block structured languages.	✗ Performance depends on the data structure used.
✓ Essential for code generation and optimization.	

10. Applications :

- Used in lexical analysis, syntax analysis and semantic analysis.
- Stores information about variables, constants, functions, arrays, types and labels.
- Essential for intermediate code generation and optimization.
- Used in all compilers for block structured programming languages.

Summary

Symbol Table Operations are the basic actions performed on the symbol table to manage identifier information. The main operations are Insert, Lookup, Update, Delete, BeginScope and EndScope. Efficient implementation (using hash table, tree, etc.) is crucial for building a fast and reliable compiler.

16. Error Detection

Error Detection is the process of finding errors in the source program. The compiler must detect as many errors as possible and report them in a meaningful way so that the programmer can correct them.

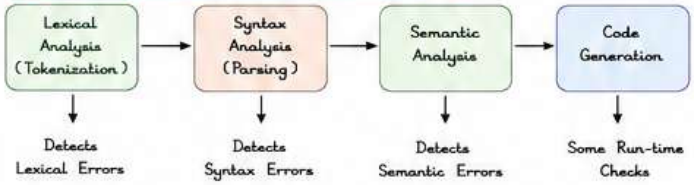
1. Introduction :

Errors may occur in a program due to nenparmk syntax, semantic violations or logical mistakes. The compiler detects errors and reports them with a suitable message along with the line number to help the programmer.

2. Types of Errors :

Type of Error	Description
Lexical Errors	Errors in character sequence (tokens) such as invalid tokens, invalid identifiers, etc.
Syntax Errors	Violations of the grammar rules of the programming language.
Semantic Errors	Meaning of the program is incorrect even though it is syntactically correct.
Logical Errors	Program runs but gives incorrect output.
Run-time Errors	Errors occur during execution such as divide by zero, array out of bounds, etc.

3. Phases of Error Detection in Compiler :



4. Lexical Error Detection :

- Lexical errors occur during scanning of characters to form tokens.
- Examples :
  1. Invalid character in a token, e.g., @, $, #
  2. Identifier starts with a digit, e.g., 2abc
  3. Unrecognized escape sequence in a string

Example :

```
int $a = 10;          → '$' is not a valid character
Char name = 'A;       → Missing closing single quote
float 2value;         → Identifier cannot start with digit
```

5. Syntax Error Detection :

- Syntax errors are detected by the parser (syntax analyzer).
- These errors violate the grammar rules of the language.
- Examples :
  1. Missing semicolon
  2. Unmatched parentheses or braces
  3. Incorrect order of tokens

Example :

```
int a = 10;           → Missing semicolon
if (a > 5 {           → Missing closing parenthesis ')'
int b = a + * 5;      → Operator used incorrectly
```

6. Semantic Error Detection :

- Semantic errors occur when the meaning of the program is incorrect.
- These cannot be detected by the parser.
- Examples :
  1. Using undeclared variable
  2. Type mismatch in expressions
  3. Invalid function call (wrong number/type of arguments)
  4. Return type mismatch

Example :

```
int a;
b = a + 1;            → 'b' is not declared
int c = "hello";      → Type mismatch (string to int)
sum(10);              → Function 'sum()' expects two arguments
```

Important Points for Exams

- ★ Errors are detected in lexical, syntax and semantic analysis phases.
- ★ Good error messages help the programmer to correct errors easily.
- ★ Compiler should detect as many errors as possible in one run.
- ★ Error recovery allows compiler to continue after an error.

7. Strategies for Error Detection :

- Detect errors as early as possible.
- Detect as many errors as possible in one compilation.
- Report errors with meaningful messages.
- Avoid cascading error messages (one error causing many errors).
- Continue compilation after detecting an error (error recovery).

8. Error Reporting :

- The compiler should provide the following information in an error message.
- ✓ The type of error.
- ✓ The location of error (line number and position).
- ✓ A meaningful description of the error.
- ✓ Suggestion (if possible) to remove the error.

Example of Error Message :

```
Error: Syntax Error
Line No.: 15     Position: 8
Message   : Missing semicolon at the end of statement.
Code      : int a = 10
                    ↑
Hint      : Add a ';' at the end of the statement.
```

9. Error Recovery Techniques :

- After detecting an error, compiler should resume compilation.
- Main techniques :

① Panic Mode Recovery : Skip symbols until a synchronizing token (;, },),],), etc.) is found.

② Phrase Level Recovery : Make small changes to the program and continue parsing.

③ Error Productions : Add productions in grammar to handle common mistakes.

Example (Panic Mode) :

```
Input : int a = 10   int b = 20 ;
                           ↑ Missing semicolon
Recovery : Skip tokens until ';' is found and continue parsing.
```

10. Error Detection vs. Error Correction :

Error Detection	Error Correction
Finds and reports errors.	Finds and fixes errors automatically.
Does not modify the source program.	Modifies the source program.
Simpler to implement.	Complex to implement.
Used in most compilers.	Rarely used in compilers.

11. Importance of Error Detection :

- Helps in writing correct programs.
- Saves time of programmer in debugging.
- Increases reliability and robustness of compiler.
- Improves user experience.

Summary

Error Detection is an essential task of compiler to find errors in different phases. It includes lexical, syntax, semantic and run-time error detection. Compiler reports errors with useful messages and uses error recovery techniques to continue compilation and report more errors in a single run.

17. Error Recovery

Error Recovery is the process of allowing the compiler to continue compilation after detecting an error, so that more errors can be found in the same run and meaningful output (if possible) can still be generated.

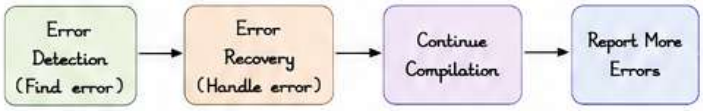
1. Introduction :

When an error is detected, the compiler reports it and tries to recover from it. Recovery helps in finding more errors in one compilation and improves the efficiency of the compiler. Without error recovery, the compiler stops at the first error.

2. Need for Error Recovery :

- Helps to report multiple errors in a single compilation.
- Saves time for the programmer (no need to recompile again and again).
- Increases the reliability and robustness of the compiler.
- Allows the compiler to generate partial or complete code.

3. Phases Involved :



4. Error Recovery Techniques :

Technique	Idea
Panic Mode Recovery	Discard input symbols until a synchronizing token is found.
Phrase Level Recovery	Make small local changes to the input (insert, delete, replace) to make it syntactically correct.
Error Production	Modify grammar by adding special productions that handle common errors.
Global Correction	Make minimum changes to the entire program using more complex algorithms.

5. Panic Mode Recovery :

- The parser discards input symbols one by one until a synchronizing token (like ;, }, or)) is found.
- Synchronizing tokens are chosen so that parser can continue with correct parsing.
- Simple and easy to implement.

Example :

```
Input    :  int  a  =  10     print  (  a  )  ;
                       ↑
Error    :  Missing semicolon after 10
Recovery :  Discard tokens until ';' is found.
            After skipping 'print', parser resumes from '('.
```

6. Phrase Level Recovery :

- Make small changes (insert, delete, replace) in the program to correct the error.
- Allows the parser to continue without discarding large portion.
- 3 basic actions :
  ① Insertion ② Deletion ③ Replacement

Example :

```
Input     :  if  (  a  <  5     a  =  10  ;
Error     :  Missing ')'
Recovery (Insertion)  :  Insert ')' after 5
Corrected :  if  (  a  <  5  )  a  =  10  ;
```

7. Error Productions :

- Grammar is modified by adding special productions to handle common errors.
- These productions generate specific error messages and allow parsing to continue.

Example :

```
Original Grammar  :  S  →  if  (  E  )  S
Modified Grammar  :  S  →  if  (  E  )  S
                        |  if  (  E     error
Where  'error'  handles missing ')'.
```

8. Global Correction :

- Computes minimum number of changes to the entire input to make it syntactically correct.
- Uses algorithms like Levenshtein distance, dynamic programming etc.
- Complex and costly, used in advanced compilers.

9. Comparison of Error Recovery Techniques :

Technique	Advantage	Disadvantage
Panic Mode	Simple and easy to implement	May skip important part of the program
Phrase Level	More precise, corrects locally	More complex than panic mode
Error Production	Can handle common mistakes well	Grammar becomes large and complicated
Global Correction	Finds optimal correction	Very complex and high overhead

10. Example - Panic Mode Recovery :

```
Input  :  id  =  id  +  *  id  ;  id  =  id  -  id  ;
Step 1 :  Parser finds '*' where an operand is expected → error
Step 2 :  Discard '*' and continue till next synchronizing token ';'
Step 3 :  Parsing resumes from next statement "id = id - id ;"
```

11. Example - Phrase Level Recovery :

```
Input     :  if  (  a  <  5     b  =  10  ;
Error     :  Missing ')'
Action    :  Insert ')' after 5
Corrected Input  :  if  (  a  <  5  )  b  =  10  ;
```

12. Important Points :

- ★ Error recovery is essential for practical compilers.
- ★ It improves user experience by reporting multiple errors.
- ★ Recovery should introduce minimum additional errors.
- ★ The choice of technique depends on the language and compiler design.

13. Applications :

- Used in all modern compilers (C, C++, Java, etc.).
- Useful in IDEs to highlight multiple errors at once.
- Help in syntax analysis phase during parsing.

Summary

Error Recovery allows the compiler to continue compilation after an error is found. It helps to detect more errors in one run and saves the programmer's time. Main techniques include Panic Mode Recovery, Phrase Level Recovery, Error Productions and Global Correction.

- Panic Mode → discard tokens till synchronizing token.
- Phrase Level → make small changes (insert/delete/replace).
- Error Production → add special grammar rules.
- Global Correction → find minimum changes for entire input.