UNIT-4

1. **Explain the specification of a simple type checker**

A type checker for a simple language checks the type of each identifier. The type checker is a translation scheme that synthesizes the type of each expression from the types of its subexpressions. The type checker can handle arrays, pointers, statements and functions.

**A Simple Language**

Consider the following grammar:

P → D ; E

D → D ; D | id : T

T → char | integer | array [ num ] of T | ↑ T

E → literal | num | id | E mod E | E [ E ] | E ↑

Translation scheme:

P → D ; E

D → D ; D

D → id : T { addtype (id.entry , T.type) }

T → char { T.type : = char }

T → integer { T.type : = integer }

T → ↑ T1 { T.type : = pointer(T1.type) }

T → array [ num ] of T1 { T.type : = array ( 1… num.val , T1.type) }

In the above language,

→ There are two basic types : char and integer ; → type\_error is used to signal errors;

→ the prefix operator ↑ builds a pointer type. Example , ↑ integer leads to the type expression

pointer ( integer ).

**2) Define a type expression?** **Explain the equivalence of type expressions with an appropriate example?**

TYPE EXPRESSION. The idea is to associate each language construct with an expression describing its type and so called type expression. Type expressions are defined inductively from basic types and constants using type constructors. Here are the type constructors that we shall consider in the remaining of this chapter. They are close to type constructors of languages like C or PASCAL.

Equivalence of Type Expressions

* If two type expressions are equal then **return** a certain type else return type\_error.
* **Key Ideas:**
  + The main difficulty arises from the fact that most modern languages allow the naming of user-defined types.
  + For instance, in C and C++ this is achieved by the typedef statement.
  + When checking equivalence of named types, we have two possibilities.
    - Structural Equivalence
    - Names Equivalence

**3) Explain about reusing the storage space for names?**

Symbol table is an important data structure created and maintained by compilers in order to store information about the occurrence of various entities such as variable names, function names, objects, classes, interfaces, etc. Symbol table is used by both the analysis and the synthesis parts of a compiler.

A symbol table may serve the following purposes depending upon the language in hand:

* To store the names of all entities in a structured form at one place.
* To verify if a variable has been declared.
* To implement type checking, by verifying assignments and expressions in the source code are semantically correct.
* To determine the scope of a name (scope resolution).

A symbol table is simply a table which can be either linear or a hash table. It maintains an entry for each name in the following format:



For example, if a symbol table has to store information about the following variable declaration:



then it should store the entry such as:



The attribute clause contains the entries related to the name.

**4) Discuss about all allocation strategies in run-time storage environment?**

**Storage Allocation**

Runtime environment manages runtime memory requirements for the following entities:

* **Code :** It is known as the text part of a program that does not change at runtime. Its memory requirements are known at the compile time.
* **Procedures** : Their text part is static but they are called in a random manner. That is why, stack storage is used to manage procedure calls and activations.
* **Variables :** Variables are known at the runtime only, unless they are global or constant. Heap memory allocation scheme is used for managing allocation and de-allocation of memory for variables in runtime.

**Static Allocation**

In this allocation scheme, the compilation data is bound to a fixed location in the memory and it does not change when the program executes. As the memory requirement and storage locations are known in advance, runtime support package for memory allocation and de-allocation is not required.

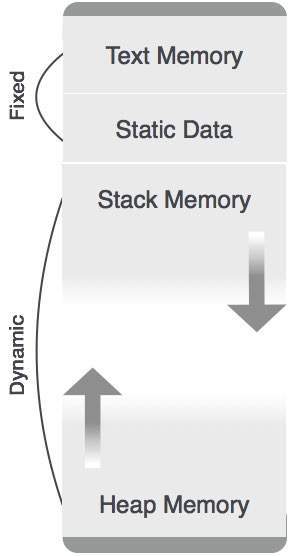
**Stack Allocation**

Procedure calls and their activations are managed by means of stack memory allocation. It works in last-in-first-out (LIFO) method and this allocation strategy is very useful for recursive procedure calls.

**Heap Allocation**

Variables local to a procedure are allocated and de-allocated only at runtime. Heap allocation is used to dynamically allocate memory to the variables and claim it back when the variables are no more required.

Except statically allocated memory area, both stack and heap memory can grow and shrink dynamically and unexpectedly. Therefore, they cannot be provided with a fixed amount of memory in the system.



**5) Explain the data structures used for implementing Symbol Table?**

Following are commonly used data structure for implementing symbol table :-

**1. List**

* In this method, an array is used to store names and associated information.
* A pointer​ “available”​ is maintained at end of all stored records and new names are added in the order as they arrive
* To search for a name we start from beginning of list till available pointer and if not found we get an error ​“use of undeclared name”
* While inserting a new name we must ensure that it is not already present otherwise error occurs i.e. “Multiple defined name”
* Insertion is fast O(1), but lookup is slow for large tables – O(n) on average
* Advantage is that it takes minimum amount of space.

**2. Linked List​**

* This implementation is using linked list. A link field is added to each record.
* Searching of names is done in order pointed by link of link field.
* A pointer ​“First”​ is maintained to point to first record of symbol table.
* Insertion is fast O(1), but lookup is slow for large tables – O(n) on average

**3. Hash Table​**

* In hashing scheme two tables are maintained – a hash table and symbol table and is the most commonly used method to implement symbol tables..
* A hash table is an array with index range: 0 to tablesize – 1.These entries are pointer pointing to names of symbol table.
* To search for a name we use hash function that will result in any integer between 0 to tablesize – 1.
* Insertion and lookup can be made very fast – O(1).
* Advantage is quick search is possible and disadvantage is that hashing is complicated to implement.

4. **Binary Search Tree**​

* Another approach to implement symbol table is to use binary search tree i.e. we add two link fields i.e. left and right child.
* All names are created as child of root node that always follow the property of binary search tree.
* Insertion and lookup are O(log​2​ n) on average.

**6) Explain Static and Dynamic Checking of types with examples?**

**Static Type Checking**

Static type checking checks the input program against the type rules defined in the language after the program parses successfully (that is there are no syntax errors in it). It is done without giving any input to the program being checked.  
The aim of static type checking is to reject programs early, that may cause an error if they will execute. If the program does not parse, It will also result in an error. This error is different from what we get from the type checker. This error is called parser error or Syntax error. The static type checker would typically give a type error — something like operation on incompatible types etc.

Whatever the static checker performs is part of the language definition and different languages may define different rules.

Static type checking is achieved typically through a type system. The main purpose of a type system is to reduce possibilities for bugs in programs by defining interfaces between different parts of a program, and then checking that the parts have been connected in a consistent way.

# **Dynamic type checking**:

In a dynamic language, type checking occurs at run-time. Many languages like python, ruby etc check the type safety of a program at runtime.  
Typically, It is done by tagging each run-time object with a tag (that is a reference to a type) which has all the type information. This information is used to check for type errors.  
Many languages implement some form of dynamic type checking even if they have static type checking because many useful features and non-trivial properties are difficult to verify statically. A good example of this is downcasting. A dynamic check is needed to verify that the operation is safe or not during downcasting.

**7) Compare the call by value and call by name with examples?**

Text

Description automatically generated with medium confidence

**8) Distinguish between static and dynamic storage allocation**

|  |  |  |
| --- | --- | --- |
| S.No | Static Memory Allocation | Dynamic Memory Allocation |
| 1 | In the static memory allocation, variables get allocated permanently, till the program executes or function call finishes. | In the Dynamic memory allocation, variables get allocated only if your program unit gets active. |
| 2 | Static Memory Allocation is done before program execution. | Dynamic Memory Allocation is done during program execution. |
| 3 | It uses [stack](https://www.geeksforgeeks.org/stack-data-structure/) for managing the static allocation of memory | It uses [heap](https://www.geeksforgeeks.org/heap-data-structure/) for managing the dynamic allocation of memory |
| 4 | It is less efficient | It is more efficient |
| 5 | In Static Memory Allocation, there is no memory re-usability | In Dynamic Memory Allocation, there is memory re-usability and memory can be freed when not required |
| 6 | In static memory allocation, once the memory is allocated, the memory size can not change. | In dynamic memory allocation, when memory is allocated the memory size can be changed. |
| 7 | In this memory allocation scheme, we cannot reuse the unused memory. | This allows reusing the memory. The user can allocate more memory when required. Also, the user can release the memory when the user needs it. |
| 8 | In this memory allocation scheme, execution is faster than dynamic memory allocation. | In this memory allocation scheme, execution is slower than static memory allocation. |
| 9 | In this memory is allocated at compile time. | In this memory is allocated at run time. |
| 10 | In this allocated memory remains from start to end of the program. | In this allocated memory can be released at any time during the program. |
| 11 | **Example:** This static memory allocation is generally used for [array](https://www.geeksforgeeks.org/introduction-to-arrays/). | **Example:** This dynamic memory allocation is generally used for [linked list](https://www.geeksforgeeks.org/data-structures/linked-list/). |

**9) Explain the type checking of expressions?**

**Type Expressions**

The type of a language construct will be denoted by a “type expression.” A type expression is either a basic type or is formed by applying an operator called a type constructor to other type expressions. The sets of basic types and constructors depend on the language to be checked. The following are the definitions of type expressions:

1. Basic types such as boolean, char, integer, real are type expressions.

A special basic type, type\_error , will signal an error during type checking; void denoting “the absence of a value” allows statements to be checked.

1. Since type expressions may be named, a type name is a type expression.
2. A type constructor applied to type expressions is a type expression.

**Constructors include:**

**Arrays :** If T is a type expression then array (I,T) is a type expression denoting the type of an array with elements of type T and index set I.

**Products :** If T1 and T2 are type expressions, then their Cartesian product T1 X T2 is a type expression. **Records :** The difference between a record and a product is that the names. The record type constructor will be applied to a tuple formed from field names and field types.

**For example:**

type row = record

address: integer;

lexeme: array[1..15] of char

end;

var table: array[1...101] of row;

declares the type name row representing the type expression record((address X integer) X (lexeme X array(1..15,char))) and the variable table to be an array of records of this type.

**Pointers :** If T is a type expression, then pointer(T) is a type expression denoting the type “pointer to an object of type T”.

For example, var p: ↑ row declares variable p to have type pointer(row).

**Functions :** A function in programming languages maps a domain type D to a range type R. The type of such function is denoted by the type expression D → R

4. Type expressions may contain variables whose values are type expressions.

**10) Explain storage organization in runtime environment?**

**Types of Runtime Environments –**

**Fully Static :**  
Fully static runtime environment may be useful for the languages in which pointers or dynamic allocation is not possible in addition to no support for recursive function calls.

* Every procedure will have only one activation record which is allocated before execution.
* Variables are accessed directly via fixed address.
* Little bookkeeping overhead; i.e., at most return address may have to be stored in activation record.
* The calling sequence involves the calculation of each argument address and storing into its appropriate parameter location and saving the return address and then a jump is made.

**Stack Based :**  
In this, activation records are allocated (push of the activation record) whenever a function call is made. The necessary memory is taken from the stack portion of the program. When program execution return from the function the memory used by the activation record is deallocated (pop of the activation record). Thus, the stack grows and shrinks with the chain of function calls.

**Fully Dynamic :**  
Functional language such as Lisp, ML, etc. use this style of call stack management. Silently, here activation record is deallocated only when all references to them have disappeared, and this requires the activation records to dynamically freed at arbitrary times during execution. Memory manager (garbage collector) is needed.

The data structure that handles such management is heap an this is also called as Heap Management.

**11) Explain the types of storage allocations?**

**SAME AS PART-B6**

**12) Describe the name and structure equivalence in type expressions?**

**SAME AS PART-B2**

**13) Explain the type checking of control flow statements?**

**1. Type checks -** A compiler should report an error if an operator is applied to an incompatible operand. Example: If an array variable and function variable are added together.

**2. Flow-of-control checks** - Statements that cause flow of control to leave a construct must have some place to which to transfer the flow of control. Example: An enclosing statement, such as break, does not exist in switch statement.

Diagram

Description automatically generated

A typechecker verifies that the type of a construct matches that expected by its context. For example : arithmetic operator mod in Pascal requires integer operands, so a type checker verifies that the operands of mod have type integer. Type information gathered by a type checker may be needed when code is generated.

**14) Explain briefly about storage allocation strategies?**

**SAME AS PART-B4**

**15) Describe the basic implementation techniques for symbol table?**

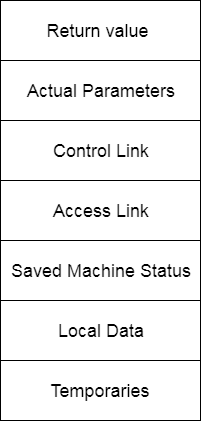
**SAME AS PART-B5**

**16) Explain the calling sequences of activation record?**

# **Activation Record**

* Control stack is a run time stack which is used to keep track of the live procedure activations i.e. it is used to find out the procedures whose execution have not been completed.
* When it is called (activation begins) then the procedure name will push on to the stack and when it returns (activation ends) then it will popped.
* Activation record is used to manage the information needed by a single execution of a procedure.
* An activation record is pushed into the stack when a procedure is called and it is popped when the control returns to the caller function.

The diagram below shows the contents of activation records:



**Return Value:** It is used by calling procedure to return a value to calling procedure.

**Actual Parameter:** It is used by calling procedures to supply parameters to the called procedures.

**Control Link:** It points to activation record of the caller.

**Access Link:** It is used to refer to non-local data held in other activation records.

**Saved Machine Status:** It holds the information about status of machine before the procedure is called.

**Local Data:** It holds the data that is local to the execution of the procedure.

**Temporaries:** It stores the value that arises in the evaluation of an expression.

**17)** **Differentiate ordered, unordered and binary search tree in symbol table?**

Text

Description automatically generated

**18) Explain briefly about static storage allocation with block diagram?**

## Static storage allocation

* In static allocation, names are bound to storage locations.
* If memory is created at compile time then the memory will be created in static area and only once.
* Static allocation supports the dynamic data structure that means memory is created only at compile time and deallocated after program completion.
* The drawback with static storage allocation is that the size and position of data objects should be known at compile time.
* Another drawback is restriction of the recursion procedure.

Diagram

Description automatically generated

**19) Differentiate explicit and implicit allocation of memory to variables?**

**Implicit Memory**

The implicit memory definition is also sometimes referred to as unconscious memory or automatic memory. When you are asked to define implicit memory, you could say that it is the long-term memory of things that you do automatically every day, without conscious thought. You don't have to try to recall the things that are in implicit memory. They simply come to you automatically.

There are some subsets of implicit memory. One subset is referred to as procedural memory. This is the memory that allows us to walk, hold a fork, ride a bike, and other processes that we don't have to think about consciously. It is all of the things that you learn to do as you grow as a small child and henceforth come to you naturally.

Another smaller subset of implicit memory is priming. Priming is the act of associating a picture, word, or other stimuli to assist in recognizing another word. An example would be to think of the color green to remember the word grass or the color red to remember the word apple.

**Explicit Memory**

The explicit memory definition is much simpler. This is the type of memory responsible for your long-term memories, such as where you went on vacation last year. Recollection of previous events or knowledge gained over time but not used on a regular basis are part of explicit memory. It is also often referred to as declarative memory because you must consciously recall and declare the information.

Explicit memory is the subset of long-term memory that most people think of when they are thinking about memory loss or how well they remember things. It can deteriorate naturally over time or be accelerated by a disorder. However, most people can remember things from explicit memory long after they experience short-term memory loss.

There are two main subsets of explicit memory. These are a semantic and episodic memory. Semantic memory is your general knowledge of the world around you but is not tied to specific events. For example, if you can recall the names and dates associated with the civil war but not the time or place you learned this information, you are using semantic memory.

**20) Differentiate stack and heap storage allocation strategies?**

| Parameter | STACK | HEAP |
| --- | --- | --- |
| Basic | Memory is allocated in a contiguous block. | Memory is allocated in any random order. |
| Allocation and De-allocation | Automatic by compiler instructions. | Manual by the programmer. |
| Cost | Less | More |
| Implementation | Easy | Hard |
| Access time | Faster | Slower |
| Main Issue | Shortage of memory | Memory fragmentation |
| Locality of reference | Excellent | Adequate |
| Safety | Thread safe, data stored can only be accessed by owner | Not Thread safe, data stored visible to all threads |
| Flexibility | Fixed-size | Resizing is possible |
| Data type structure | Linear | Hierarchical |

**PART-C**

**1) Suppose that the type of each identifier is a sub range of integers, for expressions with operators +, -, \*, div and mod, as in Pascal. Explain type- checking rules that assign to each sub expression the sub range its value must lie in?**

**2) Explain briefly about Source language issues?**

**SOURCE LANGUAGE ISSUES**

**Activation Tree**

A program consist of procedures, a procedure definition is a declaration that, in its simplest form, associates an identifier (procedure name) with a statement (body of the procedure). Each execution of procedure is referred to as an activation of the procedure. Lifetime of an activation is the sequence of steps present in the execution of the procedure. If ‘a’ and ‘b’ be two procedures then their activations will be non-overlapping (when one is called after other) or nested (nested procedures). A procedure is recursive if a new activation begins before an earlier activation of the same procedure has ended. An activation tree shows the way control enters and leaves activations. Properties of activation trees are :-

* Each node represents an activation of a procedure.
* The root shows the activation of the main function.
* The node for procedure ‘x’ is the parent of node for procedure ‘y’ if and only if the control flows from procedure x to procedure y.

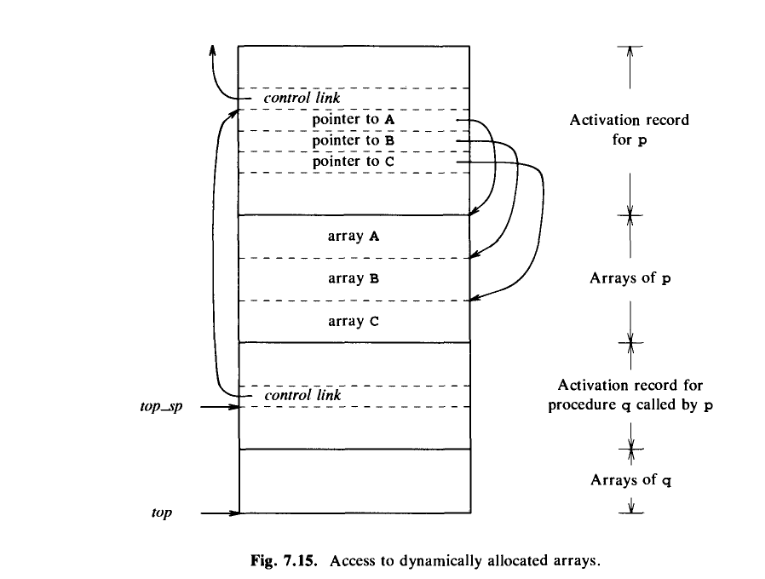
**3) Explain briefly about Activation record with block diagram?**

**SAME AS PART-B16**

**4)** **Discuss about variable length data on stack with neat diagram?**

**Variable length data on stack:**

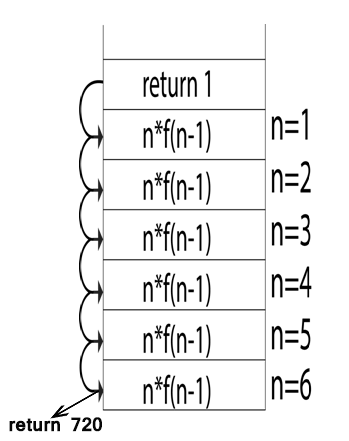
The run-time memory management system must deal frequently with the allocation of space for objects, the sizes of which are not known at the compile time, but which are local to a procedure and thus may be allocated on the stack. The reason to prefer placing objects on the stack is that we avoid the expense of garbage collecting their space. The same scheme works for objects of any type if they are local to the procedure called and have a size that depends on the parameters of the call.



**5) Explain briefly about heap storage allocation with block diagram?**

## Heap Storage Allocation

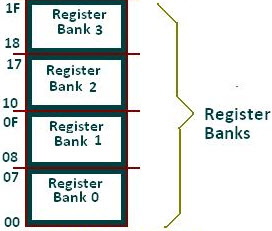
* Heap allocation is the most flexible allocation scheme.
* Allocation and deallocation of memory can be done at any time and at any place depending upon the user's requirement.
* Heap allocation is used to allocate memory to the variables dynamically and when the variables are no more used then claim it back.
* Heap storage allocation supports the recursion process.



**6) Explain briefly about stack storage allocation with block diagram?**

## Stack Storage Allocation

* In static storage allocation, storage is organized as a stack.
* An activation record is pushed into the stack when activation begins and it is popped when the activation end.
* Activation record contains the locals so that they are bound to fresh storage in each activation record. The value of locals is deleted when the activation ends.
* It works on the basis of last-in-first-out (LIFO) and this allocation supports the recursion process.



**7)** **Explain briefly about language facilities for dynamic storage allocation?**

Text, letter

Description automatically generated

**8****) Describe the parameter passing methods with examples?**

**Parameter Passing**

The communication medium among procedures is known as parameter passing. The values of the variables from a calling procedure are transferred to the called procedure by some mechanism. Before moving ahead, first go through some basic terminologies pertaining to the values in a program.

**r-value**

The value of an expression is called its r-value. The value contained in a single variable also becomes an r-value if it appears on the right-hand side of the assignment operator. r-values can always be assigned to some other variable.

**l-value**

The location of memory (address) where an expression is stored is known as the l-value of that expression. It always appears at the left hand side of an assignment operator.

For example:

day = 1;

week = day \* 7;

month = 1;

year = month \* 12;

From this example, we understand that constant values like 1, 7, 12, and variables like day, week, month and year, all have r-values. Only variables have l-values as they also represent the memory location assigned to them.

For example:

7 = x + y; is an l-value error, as the constant 7 does not represent any memory location.

**Formal Parameters**

Variables that take the information passed by the caller procedure are called formal parameters. These variables are declared in the definition of the called function.

**Actual Parameters**

Variables whose values or addresses are being passed to the called procedure are called actual parameters. These variables are specified in the function call as arguments.

**9)** **Explain Over loading of Operators & Functions with examples?**

Graphical user interface, text, application

Description automatically generated

Text

Description automatically generated

**10) Differentiate the call by reference and call by copy restore with examples?**

**Call by copy-restore**​ is similar to call by reference except that the values of actual parameters are changed when the called function ends, unlike call by reference. When a function call is made the values of actual parameters are copied to the activation record of the called function. The formal parameters are manipulated and changed in the function, leaving the actual parameters intact still. But as soon as the called function is executed completely, the values of formal parameters are copied to the actual parameters in the activation record and finally the values of the actual parameters are changed. So basically the difference between call by reference and call by copy-restore is that in call by reference changes are made in real-time (reflected simultaneously) and in call by copy-restore changes are reflected after completion of the called function.

**example**

int a;

void unsafe(int x) {

x= 2; //a is still 1

a= 0; //a is now 0

}

//function ends so the value of x is now stored in a -> value of a is now 2

int main() {

a= 1; unsafe(a); //when this ends the value of a will be 2

printf("%d\n", a); //prints 2

}