UNIT – 4 : RUNTIME ENVIRONMENTS

Introduction:

A program as a source code is merely a collection of text (code, statements etc.) and to make it alive, it requires actions to be performed on the target machine. A program needs memory resources to execute instructions. A program contains names for procedures, identifiers etc., that require mapping with the actual memory location at runtime.

By runtime, we mean a program in execution. **Runtime environment** is a state of the target machine, which may include software libraries, environment variables, etc., to provide services to the processes running in the system.

Runtime deals with the layout, allocation, and deallocation of storage locations, linkages between procedures, and passing parameters among other concerns.

Issues Dealt with Runtime Environments:

• How to pass parameters when a procedure is called?

• What happens to locals when procedures return from an activation?

• How to support recursive procedures?

• Can a procedure refer to nonlocal names? If yes, then how?

Importance of Runtime Environments:

The runtime environment is essential for:

- **Program Execution:** It ensures that the compiled program can execute correctly on the target machine.
- **Portability:** By abstracting the underlying hardware, runtime environments enable programs to run on different platforms without modification.
- **Resource Management:** It manages resources like memory and CPU time, ensuring efficient use during program execution.
- **Security and Stability:** Runtime environments enforce security measures and handle exceptions, reducing the likelihood of program crashes or unauthorized access.

Storage Allocation:

A compiler is a program that converts HLL (High-Level Language) to LLL (Low-Level Language) like machine language. In a compiler, there is a need for storage allocation strategies in Compiler design because it is very important to use the right strategy for storage allocation as it can directly affect the performance of the software.

Storage Allocation Strategies:

There are mainly three types of Storage Allocation Strategies:

1. Static Allocation
2. Heap Allocation
3. Stack Allocation

1. Static Allocation:

In Static Allocation, All variables that are created will be **assigned to memory locations at compile time**, and the addresses of these variables will remain the same throughout the program's execution. As the size of variables does not vary much so this allocation is easy to understand and efficient, but this allocation is not flexible and scalable.

C, C++, and Java support static allocation through the use of the "static" keyword.

Advantages:

- Static Allocation is simple and easy to understand.
- As the memory is being allocated at compile time, so there will be no additional time needed in run time.
- Debugging can be easier for developers as the memory is allocated at compile time.

Disadvantages:

- Variables that are dynamic cannot be handled using static allocation.
- No Flexibility and Scalability.

Here's an example of static allocation:

```
int a = 10;

static int b = 1;

const int x = 9;

// In all the above three examples, the memory will be assigned to each of them at the program startup. and the addresses of these variables will remain the same in the memory throughout the lifetime of the program.
```

2. Heap Allocation:

Heap allocation is used where the Stack allocation lacks if we want to retain the values of the local variable after the activation record ends, which we cannot do in stack allocation, here LIFO scheme does not work for the allocation and de-allocation of the activation record. Heap is the most flexible storage allocation strategy we can dynamically allocate and de-allocate local variables whenever the user wants according to the user needs at run-time. The variables in heap allocation can be changed according to the user's requirement.

C, C++, Python, and Java all of these support Heap Allocation.

For example: int* ans = new int[5];

Advantages of Heap Allocation:

1. Heap allocation is useful when we have data whose size is not fixed and can change during the run time.

2. We can retain the values of variables even if the activation records end.

3. Heap allocation is the most flexible allocation scheme.

Disadvantages of Heap Allocation:

1. Heap allocation is slower as compared to stack allocation.

2. There is a chance of memory leaks.

3. Stack Allocation:

Stack is commonly known as **Dynamic allocation**. Dynamic allocation means the allocation of memory at run-time. Stack is a data structure that follows the LIFO principle so whenever there is multiple activation record created it will be pushed or popped in the stack as activations begin and ends. Local variables are bound to new storage each time whenever the activation record begins because the storage is allocated at runtime every time a procedure or function call is made. When the activation record gets popped out, the local variable values get erased because the storage allocated for the activation record is removed.

C and C++ both have support for Stack allocation.

For example:

```
void sum(int a, int b){int ans = a+b;cout<<ans;}

// when we call the sum function in the example above, memory will be allotted for the variable ans
```

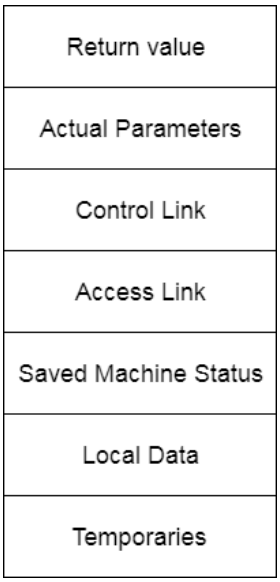
Activation Records:

An activation record, also known as **stack frames**, is a contiguous block of storage that manages information required by a single execution of a procedure. When you enter a procedure, you allocate an activation record, and when you exit that procedure, you de-allocate it.

Basically, it stores the status of the current activation function. So, whenever a function call occurs, then a new activation record is created and it will be pushed onto the top of the stack. It will remain in stack till the execution of that function. So, once the procedure is completed and it is returned to the calling function, this activation function will be popped out of the stack.

If a procedure is called, an activation record is pushed into the stack, and it is popped when the control returns to the calling function.

Activation Record includes some fields which are –



Temporaries: The temporary values, such as those arising in the evaluation of expressions, are stored in the field for temporaries.

Local data: The field for local data holds data that is local to an execution of a procedure.

Saved Machine States: The field for Saved Machine Status holds information about the state of the machine just before the procedure is called. This information includes the value of the program counter and machine registers that have to be restored when control returns from the procedure.

Access Link: It refers to information stored in other activation records that is non-local. The access link is a static link and the main purpose of the access link is to access the data which is not present in the local scope of the activation record. It is a static link.

Let’s take an example to understand this –

```
#include <stdio.h>
int g=12;
void Geeks()
{
  printf("%d", g);
}
void main()
{
 Geeks();
}
```

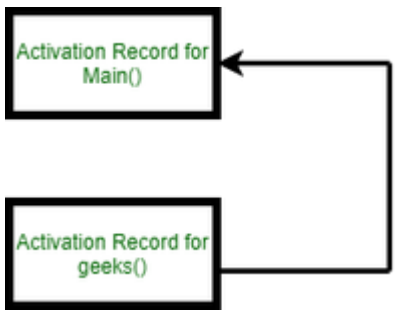
Now, In this example, when Geeks() is called in a main(), the task of Geeks() in main() is to print(g), but g is not defined within its scope(local scope of Geeks()); in this case, Geeks() would use the access link to access ‘g’ from Global Scope and then print its value (g=12).

As a chain of access links (think of scopes), the program traces its static structure.

Control Links: In this case, it refers to an activation record of the caller. They are generally used for links and saved status. It is a dynamic link in nature. When a function calls another function, then the control link points to the activation record of the caller. Record A contains a control link pointing to the previous record on the stack. Dynamically executed programs are traced by the chain of control links.

Example –

```
#include<stdio.h>
int geeks(int x)
{
  printf("value of x is: %d", x);
}
int main()
{
  geeks(10);
}
```



Parameter List: The field for parameters list is used by the calling procedure to supply parameters to the called procedure. We show space for parameters in the activation record, but in practice, parameters are often passed in machine registers for greater efficiency.

Return Value: The field for the return value is used by the called procedure to return a value to the calling procedure. Again, in practice, this value is often returned in a register for greater efficiency.

Accessing Local and Non-Local Names in a Block Structured Language:

In block-structured languages, such as C, Pascal, and Python, accessing local and non-local names (variables) is an essential part of how functions and blocks of code interact with each other. Understanding how these names are accessed helps in grasping the scope rules and lifetime of variables, as well as the runtime mechanisms used to manage these variables.

1. Local Names:

Local names are variables that are defined within the current block or function. They have the following characteristics:

- **Scope:** The scope of a local variable is limited to the block or function in which it is declared. This means that it can only be accessed within that specific block or function.
- **Lifetime:** The lifetime of a local variable usually starts when the block is entered and ends when the block is exited.
- **Access:** Local variables are directly accessible in the block where they are declared, often through the activation record (stack frame) associated with that block or function.

Example of Local Names:

```
void functionA() {
    int localVar = 10; // local variable
    printf("%d", localVar); // localVar is accessible here
}
// localVar is not accessible here, outside of functionA
```

2. Non-Local Names:

Non-local names refer to variables that are not defined within the current block or function but can still be accessed from it. These include:

- **Global Variables:** Variables defined outside of any function or block, accessible from any part of the program.
- **Variables from Enclosing Blocks:** In languages that support nested functions (e.g., Python, Pascal), a function can access variables defined in its enclosing block or function.

Example of Non-Local Names

```
int globalVar = 20; // global variable


void functionB() {
   int localVarB = 30;


   void nestedFunction() {
      int localVarNested = 40;
      printf("%d", globalVar); // Accessing global variable
      printf("%d", localVarB); // Accessing local variable of the enclosing function
      // localVarNested is accessible only within nestedFunction
   }
}
```

Mechanisms for Accessing Local and Non-Local Names:

1. Accessing Local Variables:

- **Direct Access via Activation Records:** Local variables are stored in the activation record of the current function or block, which is located on the stack. They can be accessed directly using the frame pointer or stack pointer that points to the current activation record.

2. Accessing Non-Local Variables:

- **Global Variables:** These are usually stored in a separate data segment, and access is done via a global symbol table that maps variable names to their memory addresses.

- **Variables from Enclosing Blocks:**
  - In languages with nested functions, non-local variables from enclosing blocks are accessed using a chain of activation records (static or dynamic links).
  - **Static Links:** These point to the activation record of the immediate enclosing function. The chain of static links is followed to reach the activation record that contains the non-local variable.
  - **Dynamic Links:** These point to the caller's activation record and are more commonly used for managing the return address rather than non-local variable access.

Parameters Passing:

Parameter passing is an essential step in compiler design. Parameter passing refers to the exchange of information between methods, functions, and procedures. Some mechanism transfers the values from a variable procedure to the called procedure.

Parameters are of two types: **Actual Parameters** and **Formal Parameters**.

Actual Parameter:

Actual parameters are variables that accept data given by the calling process. These variables are listed in the called function's definition. They accept the calling process's data. The called function's definition contains a list of these variables.

There is no need to specify the datatype in the actual arguments. They could be expressions, constants, or variables without regard to data types. The term "actual parameters" refers to the parameters handled during a function call.

Formal Parameter:

Formal parameters are variables whose values and functions are given to the called function. These variables are supplied as parameters in the function call. They must include the data type.

They are the numbers listed in a subprogram's parameter index. Specifying the data type of the receiving value for formal parameters is necessary. Formal parameters are data-type-related variables.

Now let us see an example of actual and formal parameters to understand them in a better way.

Basic Terminology in Parameter Passing:

R-value:

The value of an expression is its **R-value**. An R-value is a temporary object or literal without a permanent place in memory. They are basically placed on the right-hand side of the assignment operator.

Some examples of R-values are given below:

Examples:

a = 1;

b = a * 7;

c = b * 12;

Here, all variables, including a, b, and c, have **R-values** like **1, a * 7, and b * 12**. Here, the variables do not have a permanent place in memory.

L-value:

The **L-Value** of the expression refers to the location in memory where the expression is kept. Here the expression has a permanent memory location assigned. L-values are placed on the left side of an assignment operator.

Some examples of L-value are given below:

Examples:

x = 10;

a = 20;

In the above-mentioned examples, x and a are **L-values,** and the **R-values** are 10 and 20.

Methods for Parameter Passing:

1. Pass by Value: A copy of the actual parameter (the argument) is made and passed to the called function. The function operates on this copy, and changes to the parameter do not affect the original argument.

Example:

```
void function(int x) {
    x = x + 10; // modifies the copy, not the original argument
```

```
}

int main() {
    int a = 5;
    function(a);
    // a remains 5
}
```

2. Pass by Reference: Instead of passing a copy, a reference (or pointer) to the actual parameter is passed. The function can modify the original argument through this reference.

Example:

```
void function(int &x) {
    x = x + 10; // modifies the original argument
}

int main() {
    int a = 5;
    function(a);
    // a is now 15
}
```

3. Pass by Value-Result (Copy-In/Copy-Out): A copy of the actual parameter is passed to the function (like pass by value). However, at the end of the function execution, the modified copy is copied back to the original parameter.

Example:

```
procedure function(x: in out integer);
begin
    x := x + 10;
end;
```

```
var
   a: integer;
begin
   a := 5;
   function(a);
   // a is now 15
end.
```

4. Pass by Name: The actual parameter is not evaluated or copied before the function call. Instead, the expression (or parameter) is substituted directly into the function body wherever it is used. This is like "textual substitution" and is mainly theoretical, rarely used in practice.

Example: (pseudo-code for conceptual understanding)

```
procedure function(x: name integer);
begin
   y := x + x; // x is evaluated each time it's used
end;


function(a + b); // function sees x as (a + b) + (a + b)
```

5. Pass by Result: Similar to pass by value-result, but only the final result is copied back to the actual parameter. The original value of the parameter is ignored by the function.

Example:

```
procedure function(x: out integer);
begin
   x := 10;
end;
```

```
var
    a: integer;
begin
    function(a);
    // a is now 10, original value of a is ignored
end.
```

6. Pass by Constant Reference: A reference to the actual parameter is passed to the function, but the function is not allowed to modify the parameter. This method is used to avoid copying large data structures while ensuring the data remains unaltered.

```
void function(const int &x) {
    // x cannot be modified
    cout << x;
}

int main() {
    int a = 5;
    function(a);
    // a remains 5
}
```

Symbol Table Organization:

The symbol table is defined as the set of Name and Value pairs.

Symbol Table is an important data structure created and maintained by the compiler in order to keep track of semantics of variables i.e. it stores information about the scope and binding information about names, information about instances of various entities such as variable and function names, classes, objects, etc.

Features Of Symbol Table:

- It is built-in lexical and syntax analysis phases.
- The information is collected by the analysis phases of the compiler and is used by the synthesis phases of the compiler to generate code.
- It is used by the compiler to achieve compile-time efficiency.
- It is used by various phases of the compiler as follows:

1. **Lexical Analysis:** Creates new table entries in the table, for example like entries about tokens.
2. **Syntax Analysis:** Adds information regarding attribute type, scope, dimension, line of reference, use, etc in the table.
3. **Semantic Analysis:** Uses available information in the table to check for semantics i.e. to verify that expressions and assignments are semantically correct (type checking) and update it accordingly.
4. **Intermediate Code generation:** Refers symbol table for knowing how much and what type of run-time is allocated and table helps in adding temporary variable information.
5. **Code Optimization:** Uses information present in the symbol table for machine-dependent optimization.
6. **Target Code generation:** Generates code by using address information of identifier present in the table.

Symbol Table Entries:

Each item in the symbol table is linked to a set of properties that assist the compiler at various stages.

Items stored in Symbol table:

- Labels in source languages
- Procedure and function names
- Variable names and constants
- Compiler generated temporaries
- Literal constants and strings

Operations of Symbol table – The basic operations defined on a symbol table include:

Operation	Function
allocate	to allocate a new empty symbol table
free	to remove all entries and free storage of symbol table
lookup	to search for a name and return pointer to its entry
insert	to insert a name in a symbol table and return a pointer to its entry
set_attribute	to associate an attribute with a given entry
get_attribute	to get an attribute associated with a given entry

Operations on Symbol Table:

Following operations can be performed on symbol table-

1. Insertion of an item in the symbol table.

2. Deletion of any item from the symbol table.

3. Searching of desired item from symbol table.

Information used by the compiler from Symbol table:

Symbol Identification: The symbol table enables the compiler to apprehend and keep track of variables, functions, and different symbols in your application.

Data Type Information: It stores data about the data types of variables, like whether they're numbers, text, or something else.

Variable Scope: It is information wherein variables are declared and where they may be used within your code.

Function Details: For functions, it stores their names, parameters, and what they do.

Error Detection: The symbol desk aids in detecting mistakes for your code by way of making sure that variables and functions are used correctly and consistently.

Code Generation: It assists in generating machine code or executable programs primarily based on the statistics saved, making your code run effectively.

Advantages of Symbol Table:

- **Efficient identifier management:** Symbol tables provide an efficient mechanism for managing identifiers such as variables, functions, and data types used in a program.

- **Error detection:** Symbol tables can help detect errors such as duplicate declarations, undeclared variables, and type mismatches in a program.
- **Code optimization:** Symbol tables can be used to track the usage of variables and functions in a program. This information can be used during code optimization to eliminate unused variables and optimize function calls.
- **Language extensions:** Symbol tables can be extended to support new language features or extensions.

Disadvantages of Symbol Table:

- Symbol tables can take up a lot of memory because they store a lot of information about the names used in a program.
- Creating and maintaining a symbol table can be slow and can make the compilation process take longer, especially for large programs.
- Symbol tables can be difficult to implement correctly because they have to handle different kinds of names used in different parts of the program.
- It's possible that symbol tables don't offer all the features a developer needs, and therefore more tools or libraries will be needed to round out their capabilities.

Applications of Symbol Table:

1. **Resolution of variable and function names:** Symbol tables are used to identify the data types and memory locations of variables and functions as well as to resolve their names.
2. **Resolution of scope issues:** To resolve naming conflicts and ascertain the range of variables and functions, symbol tables are utilized.
3. Symbol tables, which offer quick access to information such as memory locations, are used to optimize code execution.
4. **Code generation:** By giving details like memory locations and data kinds, symbol tables are utilized to create machine code from source code.
5. **Error checking and code debugging:** By supplying details about the status of a program during execution, symbol tables are used to check for faults and debug code.
6. **Code organization and documentation:** By supplying details about a program's structure, symbol tables can be used to organize code and make it simpler to understand.

Data Structures Used in Symbol Tables / Implementation of Symbol table:

If a compiler only needs to process a small quantity of data, the symbol table can be implemented as an unordered list, which is simple to code but only works for small tables. The following methods can be used to create a symbol table:

List: A List is a collection of elements in which each element has a position or an index. This is one of the way to create a symbol table is to use a List to store the entries for each identifier. This method is simple and easy to implement, but it may not be efficient for large symbol tables because searching for a specific identifier requires iterating through the entire List.

Linked List: A Linked List is a data structure in which each element contains a reference to the next element in the list. One way to create a symbol table is to use a Linked List to store the entries for each identifier. This method can be more efficient than a List for searching because it allows for faster traversal of the entries. However, it can be slower than other methods for inserting or deleting entries because it requires modifying the references between elements.

Binary Search Tree: A Binary Search Tree is a data structure in which each node has at most two children, and the values in the left subtree are less than the value in the node, and the values in the right subtree are greater than the value in the node. One way to create a symbol table is to use a Binary Search Tree to store the entries for each identifier. This method is efficient for searching because it allows for a binary search to quickly find a specific identifier. However, it can be slower than other methods for inserting or deleting entries because it may require rebalancing the tree.

Hash Table: A Hash Table is a data structure that uses a hash function to map keys to values. One way to create a symbol table is to use a Hash Table to store the entries for each identifier. This method is efficient for searching, inserting, and deleting entries because it allows for constant-time access to entries. However, it may require more memory than other methods because it needs to store the hash table and handle collisions between entries.