Unit 3

Chapter 4

Runtime Environments

Structure of the Unit

1. Run-Time Environments:
2. Source Language Issues
3. Storage Organization
4. Storage-allocation strategies
5. Storage-allocation in C
6. Parameter passing

4.1 Course Outcomes

After the successful completion of this unit, the student will be able to:

1. Define Run-Time Environments.
2. Discuss the role of Run-Time Environments in program execution.
3. Apply appropriate storage-allocation strategies to allocate memory in a programming language like C.
4. Analyze the impact of different Source Language Issues on Run-Time Environments.

4.2 Runtime Environments

A runtime environment, in the context of compiler design, refers to the infrastructure and support systems necessary for executing a compiled program. When a high-level programming language source code is compiled, the resulting machine code often requires certain resources and services to run correctly. The runtime environment provides these resources and services to ensure that the compiled program can execute as expected. It plays a crucial role in managing memory, handling I/O operations, and providing various system services. The key components of the runtime environment are:

Memory Management: The runtime environment is responsible for allocating and managing memory for variables and data structures used by the program. This includes managing the program's stack (for function call and local variable storage) and heap (for dynamic memory allocation).

Type Checking and Type Conversion: The runtime environment may perform type checking and type conversion as needed during program execution to ensure that operations are performed on compatible data types.

Exception Handling: It provides mechanisms to handle runtime exceptions or errors. These mechanisms allow the program to gracefully handle issues like divide-by-zero errors, null pointer dereferences, and other unexpected situations.

Dynamic Linking and Loading: In some languages, the runtime environment handles dynamic linking and loading of libraries and modules. This allows the program to use external functions and libraries that may not be known at compile time.

I/O Operations: The runtime environment facilitates input and output operations, including file I/O, console I/O, network communication, and more. It provides APIs and system calls to interact with external devices and resources.

Garbage Collection: In languages with automatic memory management (e.g., Java, C#, Python), the runtime environment includes a garbage collector that reclaims memory occupied by objects that are no longer in use, preventing memory leaks.

Runtime Libraries: The runtime environment includes standard libraries that provide common functions and routines used by programs. These libraries can include mathematical functions, data structures, and utility functions.

Concurrency and Multithreading Support: Many modern runtime environments offer support for creating and managing concurrent processes or threads, allowing for parallel execution of code.

Environment Setup and Initialization: The runtime environment sets up the execution environment, initializes variables, and performs any necessary setup before the program starts executing.

Interpreter or Just-In-Time Compilation: In some cases, the runtime environment includes an interpreter or a Just-In-Time (JIT) compiler that translates high-level code into machine code during execution, rather than ahead of time in a traditional compiler.

Security and Permissions: The runtime environment enforces security and permission settings, ensuring that a program can only access resources and perform operations it is authorized to do.

The specific features and capabilities of a runtime environment can vary depending on the programming language and the target platform (e.g., operating system and hardware architecture). Compiler designers must carefully design the runtime environment to support the language's features and ensure efficient and reliable program execution.

4.3 Source Language Issues

4.3.1 Procedures

A procedure definition associates an identifier with a statement(set of statements). The identifier is the *procedure name* and the statement(set of statements) is the *procedure body*. Procedures that return values are called *functions*. When procedure name appears in an executable statement, we say that the procedure is *called*. The procedure call executes the procedure body.

Formal parameters:

Formal parameters, also known as formal arguments or parameters, are placeholders in the definition of a procedure(function or method) in computer programming. These parameters are used to specify the inputs that a procedure expects to receive when it is called. Formal parameters are defined within the function's signature and serve as references to the actual values that will be provided when the function is invoked.
In most programming languages, formal parameters are typically declared with a name and a data type. The data type specifies the kind of value that the parameter should accept, and the name provides a reference to that value within the function's body.

Actual Parameters:

Actual parameters, also known as arguments, are the values or expressions that are passed to a function or method when it is called. These values are provided to match the formal parameters of the function, and they determine what values the formal parameters will take on within the function's execution. Actual parameters are used to supply the necessary data for the function to perform its intended operation.

Actual parameters allow you to customize the behavior of a function and provide the specific data that the function needs to operate on. They can be constants, variables, expressions, or any valid data that matches the data type and order expected by the function's formal parameters.

4.3.2 Activation Trees

Each execution of a procedure body is called as an activation of the procedure. In the context of compiler design, "activation trees" typically refer to a data structure used to represent the activation records (also known as stack frames) of functions during the execution of a program. Flow of control between procedures can be depicted using “activation trees”. Activation records are used to manage the state of function calls, including parameters, local variables, and return addresses. Activation trees are a way to visualize and manage the nested structure of these activation records, which correspond to the call stack during program execution.

An activation tree represents the call hierarchy of functions within a program. Each node in the tree corresponds to an activation record for a specific function call, and child nodes represent function calls made within the context of their parent functions. The root of the tree usually represents the initial function call.

Each time control enters the procedure $\boldsymbol{q}$ from procedure $\boldsymbol{p}$, it returns to $\boldsymbol{p}$. Precisely each time the control flows from an activation of procedure $\boldsymbol{p}$ to an activation of procedure $\boldsymbol{q}$, it returns to the same activation of $\boldsymbol{p}$. If $\boldsymbol{a}$ and $\boldsymbol{b}$ are procedure activations, then their life times are either *non-overlapping* or *nested*. That is if $\boldsymbol{b}$ is entered before is $\boldsymbol{a}$ left, then control must leave $\boldsymbol{b}$ before it leaves $\boldsymbol{a}$. For *recursive procedures* a new activation begins before an earlier activation of the same procedure has ended.

4.3.3 Control Stacks

The flow of control in a program corresponds to a depth first traversal of the activation tree. The depth first traversal of a tree starts at the root, visits a node before its children, and recursively visits children at each node from left to right. A stack, called a *control stack* is used to keep track of the live procedure activations. When the activation of a procedure begins push a node on the stack and pop the node when the activation ends.

There may be multiple declarations of the same name in different parts of the program. The *scope rules* of the language determine which declaration of a name is applicable in a particular part of a program. The portion of the program to which the declaration applies is called the scope of the declaration.

4.3.4 The Scope of a declaration

A declaration(variable) in a programming language associates information with a name. In some languages the declarations are *explicit*.

For example, in **PASCAL** a declaration is of the type

```
    var name: type;
```

In **C** it is of the type

```
    type list of variables;
```

In some languages the declarations are *implicit*. For example, in **FORTRAN** the variables that start with one of the letters I, J, K, L, M, N are of the type *integers* whereas other variables are of the type *real*. If a name is declared in a procedure it is said to be local to the procedure; otherwise, it is a non-local variable. To find the declaration that applies to a particular occurrence of a name the compiler uses the symbol table. The compiler creates an entry for a name in the symbol table when it sees its declaration.

4.3.5 Bindings of Name

The name declared once may denote different data objects at runtime. Data object refers to the memory(storage) location associated with the name and it holds the value. In the programming language context, the term "environment" refers to a function which maps a name to a memory location. The term "state" refers to a function that maps a storage location to the value held in the location. Fig 4.1 depicts these mappings.

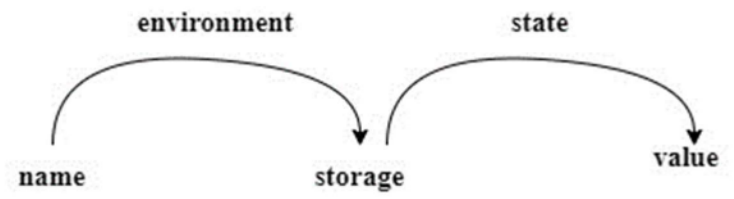

Fig 4.1 Two stage mapping from names to values

An assignment changes the state but not the environment. For example, the name x is mapped to the location 100. The assignment $x = 23$ places the value 23 in the location 100 and the assignment $x = x + 5$ changes the value in the location 100 to 28.

When an environment associates a storage location to a name, we say that the name is "bound" to the location. In the above example the name x is associated with the location 100, i.e., the name x is "bound" to the location 100.

4.4 Storage Organization

In this section we shall discuss the run-time storage organization used in languages like FOTRAN, PASCAL and C.

4.4.1 Subdivision of Run-Time Memory

The storage obtained from the operating system for the compiled program can be subdivided to hold:

1. The generated Target Code,
2. Data Objects, and
3. Control stack to keep track of the procedure activations

The size of the generated target code is fixed, hence the it can be placed in a statically determined area, normally the low end of the memory. Similarly, the size of some data objects may be known at compile time, and these also can be placed in a statically determined area. The reason for statically allocating the storage for data objects is the addresses for these objects can be compiled into the target code. In FORTRAN all data objects can be allocated statically.

The subdivision of run-time memory into different areas is depicted in Fig 4.2

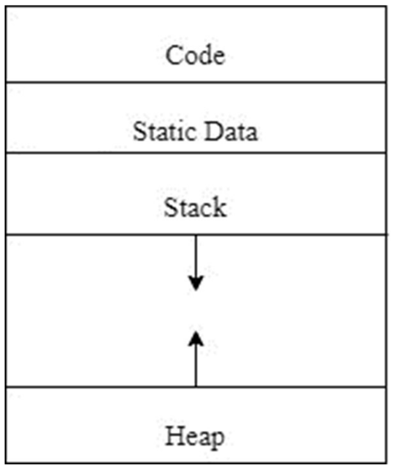

Fig 4.2 Subdivision of run-time memory into different data areas

Languages like C and PASCAL use control stack to manage activations of procedures. When a procedure is called, the status of the machine such as the program counter value and the values of the registers, is placed on the control stack. When the control returns from the call, these values are restored. Values of the data objects whose life times are contained in the activation are also allocated on the stack.

A separate area of memory called *heap* holds other information. It includes the space allocated under program control, i.e., dynamic memory allocation. The sizes of the *stack* and *heap* can change during the execution of the program; hence these areas are shown at the opposite ends of memory in Fig 4.2.

4.4.2 Activation Records

Information required for the execution of a procedure is managed using a contiguous block of storage called an *activation record* or *frame*. The *activation record* consists of the fields shown in Fig 4.3.

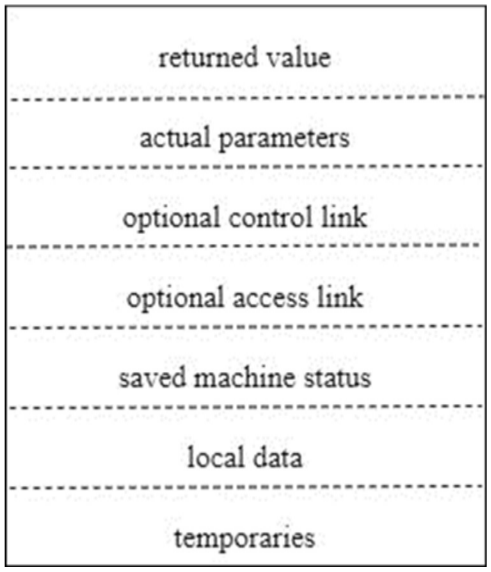

Fig 4.3 The general Activation Record

The activation record fields serve the following purpose:

1. **Temporaries:** This field is utilized for storing intermediate values that emerge during the evaluation of an expression.
2. **Local Data:** This field holds the data that is local to a procedure.
3. **Saved Machine Status:** It holds the information about the state of the machine just before the procedure is called. It includes the values of the program counter and machine registers. That have to be restored when the control returns from the procedure.
4. **Optional Access Link:** Refers to the non-local data held in other activation records.
5. **Optional Control Link:** points to the activation of the caller.
6. **Actual Parameters:** Used by the calling procedure to pass the pass parameters to the called procedure. In practice often parameters are passed in machine registers.
7. **Returned Value:** Used by the called procedure to a value to the calling procedure. In practice this value is often returned in a machine registers.

Compile-Time Layout of Local Data:

Byte is the smallest unit of addressable memory. Number of bytes form a *machine word*. Multibyte objects are stored in consecutive locations(bytes) and given the address of the first byte. The type of a name determines the amount of storage required for it. Names of the elementary data types like integer, real, character require an integral number of bytes. Storage for an aggregate like array or record, must be large enough to hold all its components. To facilitate easy access storage for aggregates is typically allocated in one contiguous block of bytes.

The storage for local data is laid out as the declarations in a procedure are examined during compilation. This does not include variable length data. A count for the previously allocated memory locations is maintained. This facilitates determination of relative address(offset) for a local with respect to some position.

The storage layout for the data objects is influenced by the addressing constraints of the target machine. For example, instructions to perform arithmetic operations on integers may expect integers to be aligned, i.e., placed at an address divisible by 4. An array of 10 characters needs 10 bytes to be allocated, but for the alignment considerations 12 bytes may be allocated. This leaves two unused bytes. Space left unused due to alignment is called as *padding*.

4.5 Storage Allocation Strategies

A different storage allocation strategy is used for the static data, heap and stack.

1. Static allocation lays out storage for all data objects at compile time.
2. Stack allocation manages the run-time storage as a stack.
3. Heap allocation allocates and deallocates storage as needed at run time from heap.

4.5.1 Static Allocation:

In this scheme names are assigned addresses during compilation. Bindings do not change at run time. Hence the values of locals do not change across different activations of the procedures. That is when the control returns to the procedure the values of the locals are the same as they were before the procedure call.

The amount of storage required for a name is determined by its type. The address of this consists of an offset from the end of an activation record for the procedure.

Limitations on the Static Allocation:

1. The size of the data object and constraints on its position in memory must be known at compile time.
2. Recursive procedures are restricted, as all activations use the same bindings for the local data.
3. Dynamic memory allocation cannot be used to create data objects, as there is no mechanism for runtime storage allocation.

4.5.2 Stack Allocation

Storage is organized as a stack. Activation records are pushed and popped as activations begin and end respectively. Locals in each call of the procedure are stored in the activation record of the call. Thus, locals are bound to fresh storage in each activation. The locals are deleted when procedure ends, i.e., the values are lost when the activation ends. The register *top* marks the top of the stack. For each activation the *top* is updated by the size of the activation record.

Calling Sequences:

Procedure calls are implemented by generating a *calling sequence*. A *call sequence* allocates an activation record and enters information into its fields. A *return sequence* restores the state of the machine.

The call sequence is:

1. The *caller* evaluates actuals.
2. The *caller* stores the return address and the old value of *top* into the *callee's* activation record. The *caller* then increments the *top* to the new position.
3. The *callee* saves register values and other status information.
4. The *callee* initializes its local data and begins its execution.

A possible return sequence is:

1. The *callee* places a return value next to the activation record of the *caller*.
2. The *callee* restores the *top* and other registers and branches to a return address in the caller's code.

Variable Length Data:

A common strategy for handling the variable length data is to store the base address in the activation record and the data itself outside. The data is accessed by using the base pointer and the relative addresses of the individual elements.

Dangling References:

A dangling reference is a pointer(reference) to a storage that has been deallocated. For example, suppose p and q are references to the same storage and the storage is deallocated using the reference q. The reference p is a dangling reference in this scenario.

4.5.3 Heap Allocation

Heap allocation gives out chunks of continuous storage as required for storing information like activation records or other objects. The chunks may be deallocated in any order.

Small activation records may be handled as follows for efficiency reasons;

1. For each size of interest, keep a linked list of free blocks of that size.
2. If possible, fill a request for size s, with a block of size s', where s' is the smallest size greater than or equal to s. When the block is deallocated, it is returned back to the linked list.
3. For large blocks of storage use the heap manager.

4.6 Storage Allocation in C

In C language, storage allocation is the process of assigning memory for variables and data structures during program execution. There are two main types of storage allocation: *stack allocation*, where memory is managed in a last-in, first-out fashion, and *heap allocation*, which involves dynamic memory allocation during runtime. The scope rules of a language determine how the non-local names are treated.

Lexical or Static Scope Rule:

Figures out which declarations relate to a name just by looking at the program text. Pascal and C are among the languages that use the lexical scope.

Dynamic Scope Rule:

Determines a declaration applicable to a name at run time. Lisp, APL and Snobol are among the languages using dynamic scope rule.

Blocks:

A block is a statement containing its own declarations. In C a block has the following syntax

```
{ declarations  statements  }
```

Delimiters mark the beginning and end of block. In C **{** and **}** used to mark the beginning and end of blocks respectively. Blocks may be nested. The scope of a declaration in a block structured language is given by the *most closely nested rule*.

1. The scope of a declaration in a block is within that block.
2. If a name x is not declared in the block B, then x can be accessed in B if and only if
    a. x is declared in an enclosing block B' and
    b. B' is more closely nested around B than any other block with a declaration for x.

The block structure can be implemented using stack allocation. Since the scope of a name is within the block of declaration, the block may be treated as a "parameterless procedure".

4.7 Parameter Passing

When you're working with programming languages, parameter passing is how information is sent to a function or method. It's like giving instructions to a function about what data it should use. There are different ways this can happen, like passing values directly or giving the memory address where the data is stored. Understanding parameter passing is crucial for making sure your functions get the right information to work with.

When a procedure calls another procedure, usually they communicate with each other through non-local names and parameters in the called procedure. There are various methods for associating actual and formal parameters.

Some of the common methods are

1. Call-by-Value
2. Call-by-Reference
3. Copy-Restore
4. Call-by-Name
5. Macro Expansion

Call by Value:

This is the simplest possible method of passing parameters. The actual parameters are evaluated and their values are passed to the corresponding formal parameters. Call by value can be implemented as follows

1. Formal parameter is treated like a local name. Hence the storage is in the activation record for the called procedure.
2. The caller evaluates the actual parameters and places their *r-values* in the storage for the formals.

Call by-Reference:

The address(reference) of the parameters is passed. This method is also known as *call-by-address* or *call-by-location*. It is implemented as follows

1. If an actual parameter is a name or an expression having an *l-value*, then that *l-value* itself is passed.
2. If the actual parameter is an expression having an *r-value,* then the expression is evaluated and its value is stored in a new location and the address of the new location is passed.

Copy-Restore:

This method is a hybrid between call-by-value and call-by-reference. It is also known as *copy-restore-linkage* or *copy-in-copy-out*, or *value-result*). It is implemented as follows

1. The actual parameters are evaluated before the procedure call. The *r-values* of the actuals are passed as in call-by-value. However, if an actual parameter has an *l-value* it is determined before the call(*copy-in*).
2. When the control returns, the *r-values* of the formal parameters are copied back to the *l-values* of the corresponding actual parameters using the *l-values* computed before the call(*copy-out*).

Call-by-Name:

It is referred to as *copy-rule* in the language "Algol". It is implemented as follows

1. The procedure is treated as if it is a *macro*, i.e., its body is substituted for the call in the caller. The actual parameters replace the formal parameters. This is called *macro-expansion* or *in-line expansion*.
2. The local names of the called procedure are kept distinct from the locals of the calling procedure.

4.8 Multiple Choice Questions

Q1. The purpose of a runtime environment in compiler design is to:

a. Translate source code into machine code
b. Manage the execution of programs at runtime
c. Optimize the compiler's performance
d. Check for syntax errors in the code

Q2. Which of the following components is NOT typically part of the runtime environment?

a. Symbol table
b. Code generator
c. Stack
d. Heap

Q3. What does the term "activation record" refer to in the context of runtime environments?

a. A record that contains information about a program's source code
b. A data structure that represents a function's local variables and control flow information
c. A table that stores information about the symbols used in the program
d. A component of the lexer in the compiler

Q4. Which memory area is used for storing dynamically allocated variables in the runtime environment?

a. Code segment
b. Data segment
c. Stack
d. Heap

Q5. What is the role of the symbol table in the runtime environment?

a. It stores information about the program's instructions.
b. It maintains a list of reserved keywords in the programming language.
c. It stores information about the names and attributes of variables and functions.
d. It is responsible for managing the execution stack.

Q6. In a runtime environment, which component is responsible for managing function calls and returns?

a. Heap
b. Code generator
c. Activation record
d. Stack

Q7. Which of the following is true about the runtime stack in a runtime environment?

a. It only stores global variables.
b. It is used for dynamic memory allocation.
c. It manages function calls and local variables.
d. It is part of the code segment.

Q8. The purpose of the heap in the runtime environment is to:

a. Store local variables.
b. Manage function calls.
c. Handle dynamic memory allocation.
d. Store the program's source code.

Q9. Which phase of the compiler is responsible for generating code that interacts with the runtime environment?

a. Lexical analysis
b. Syntax analysis
c. Code generation
d. Optimization

Q10. What information is typically stored in an activation record on the runtime stack?

a. Function names only
b. Global variables only
c. Local variables and control flow information
d. Reserved keywords in the programming language

4.9 Summary

The runtime environment in compiler design is the execution environment where a program runs after being translated from source to machine code. It comprises components like the symbol table, activation records, memory allocation, and parameter passing mechanisms. Source language issues involve considerations related to programming language syntax, semantics, and language-specific features affecting runtime behavior. Storage organization manages memory segmentation into sections like code, data, stack, and heap. Storage-allocation strategies, such as static, dynamic, and stack-based allocation, influence memory efficiency. In C, storage allocation combines static and dynamic methods. Parameter passing mechanisms, like pass by value or reference, impact how values move between functions. This understanding is critical for efficient and correct program execution.

4.10 Keywords

- Scope
- Binding
- Activation Trees
- Activation Records
- Control Stack
- Lexical Scope
- Dynamic Scope