

## 7. Subprogram Control

Subprogram sequence control is related to concept: How one subprogram invokes another and called subprogram returns to the first.

### Simple Call-Return Subprograms

- Program is composed of single main program.
- During execution It calls various subprograms which may call other subprograms and so on to any depth.
- Each subprogram returns the control to the program/subprogram after execution.
- The execution of calling program is temporarily stopped during execution of the subprogram.
- After the subprogram is completed, execution of the calling program resumes at the point immediately following the call.

### Copy Rule

The effect of the call statement is the same as would be if the call statement is replaced by body of subprogram (with suitable substitution of parameters)

We use subprograms to avoid writing the same structure in program again and again.

### Simple Call-Return Subprograms

The following assumptions are made for simple call return structure:

1. Subprogram cannot be recursive
2. Explicit call statements are required
3. Subprograms must execute completely at call
4. Immediate transfer of control at point of call or return
5. Single execution sequence for each subprogram

### Implementation

1. There is a distinction between a subprogram definition and subprogram activation.

**Subprogram definition:** The written program which is translated into a template.

**Subprogram activation:** Created each time a subprogram is called using the template created from the definition

2. An activation is implemented as two parts:

**Code Segment:** contains executable code and constants

**Activation record:** contains local data, parameters & other data items

3. The code segment is invariant during execution. It is created by translator and stored statically in memory. They are never modified. Each of the activation uses the same code segment.

4. A new activation record is created each time the subprogram is called and is destroyed when the subprogram returns. While the subprogram is executing,

the contents of the activation record are constantly changing as assignments are made to local variables and other data objects.

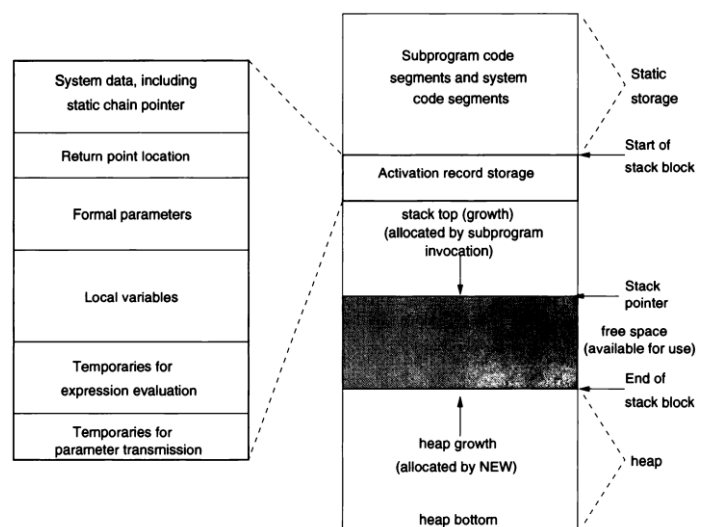
There are two system-defined pointer variables which keeps track of the point at which program is being executed.

**Current Instruction Pointer (CIP):** The pointer which points to the instruction in the code segment that is currently being executed (or just about to be) by the hardware or software interpreter.

**Current Environment Pointer (CEP):** Each activation record contains its set of local variables. The activation record represents the “referencing environment” of the subprogram. The pointer to current activation record is called Current Execution Pointer.

### Execution of Program

- First of all activation for the main program is created and CEP is assigned to it. CIP is assigned to a pointer to the first instruction of the code segment for the subprogram.
- When a subprogram is called, new assignments are set to the CIP and CEP for the first instruction of the code segment of the subprogram and the activation of the subprogram.
- To return correctly from the subprogram, values of CEP and CIP are stored before calling the subprogram. When return instruction is reached, it terminates the activation of subprogram, the old values of CEP and CIP that were saved at the time of subprogram call are retrieved and reinstated.



(a) Activation record for one procedure (b) Memory organization during execution

Figure 9.3. C memory organization.

## Recursive Subprograms

Recursion is a powerful technique for simplifying the design of algorithms. Recursive subprogram is one that calls itself (directly or indirectly) repeatedly having two properties:

1. It has a terminating condition or base criteria for which it doesn't call itself
2. Every time it calls itself, it brings closer to the terminating condition

If Recursive subprogram calls 'A', subprogram may call any other subprogram including 'A' itself, a subprogram 'B' that calls 'A' or so on. The only difference between a recursive call and an ordinary call is that the recursive call creates a second activation of the subprogram during the lifetime of the first activation.

If execution of program results in chain such that 'k' recursive calls of subprogram occur before any return is made. Thus 'k+1' activation of subprogram exist before the return from k<sup>th</sup> recursive call. Both CIP and CEP are used to implement recursive subprogram.

## Attributes of data control

**Data control features:** We need to determine the accessibility of data at different points during program execution. But the central problem is in the meaning of variable names, i.e. the correspondence between names and memory locations.

**Names and referencing environments:** There are two ways to make a data object available as an operand for an operation.

1. **Direct transmission** – A data object computed at one point as the result of an operation may be directly transmitted to another operation as an operand.

Example:  $x = y + 2 * z;$

The result of multiplication is transmitted directly as an operand of the addition operation.

2. **Referencing through a named data object** – A data object may be given a name when it is created, and the name may then be used to designate it as an operand of an operation.

Program elements that may be named

1. Variables
2. Formal parameters
3. Subprograms
4. Defined types
5. Defined constants
6. Labels
7. Exception names
8. Primitive operations
9. Literal constants

Here the names from 4 to 9 are resolved at translation time and names 1 to 3 are - discussed below.

Simple names can be like identifiers, e.g. var1, age etc.

Composite names can be names for data structure components like e.g. student[4].last\_name.

## Associations and Referencing Environments

Data control is concerned in large part with the binding of identifiers (simple names) to particular data objects and subprograms. Such a binding is termed an association and may be represented as a pair consisting of the identifier and its associated data object or subprogram.

**Referencing environment:** Referencing environments. Each program or subprogram has a set of identifier associations available for use in referencing during its execution. This set of identifier associations is termed the referencing environment of the subprogram (or program). The referencing environment of a subprogram is ordinarily invariant during its execution. It is set up when the subprogram activation is created, and it remains unchanged during the lifetime of the activation. The values contained in the various data objects may change, but the associations of names with data objects and subprograms do not. The referencing environment of a subprogram may have several components:

1. **Local referencing environment:** The set of associations created on entry to a subprogram that represents formal parameters, local variables, and subprograms defined only within that subprogram
2. **Nonlocal referencing environment:** The set of associations for identifiers that may be used within a subprogram but that are not created on entry to it. It can be global or predefined.
3. **Global referencing environment:** associations created at the start of execution of the main program, available to be used in a subprogram,
4. **Predefined referencing environments:** It is defined directly in the language definition.

**Visibility of associations:** Associations are visible if they are part of the referencing environment, otherwise associations are hidden.

**Dynamic Scope:** The dynamic scope of an association for an identifier, as defined in the preceding section, is that set of subprogram activations in which the association is visible during execution. The dynamic scope of an association always includes the subprogram activation in which that association is created as part of the local

environment. It may also be visible as a nonlocal association in other subprogram activations. A dynamic scope rule defines the dynamic scope of each association in terms of the dynamic course of program execution. For example, a typical dynamic scope rule states that the scope of an association created during an activation of subprogram P includes not only that activation but also any activation of a subprogram called by P, or called by a subprogram called by P, and so on, unless that later subprogram activation defines a new local association for the identifier that hides the original association. With this rule, the dynamic scope of an association is tied to the dynamic chain of subprogram activations.

The static scope of a declaration is that part of the program text where a use of the identifier is a reference to that particular declaration of the identifier.

A static scope rule is a rule for determining the static scope of a declaration. Static scope rules relate references with declarations of names in the program text; dynamic scope rules relate references with associations for names during program execution.

### Parameter Transmission

A data object that is strictly local is used by operations only within a single local referencing environment (e.g., within a single subprogram). Data objects, however, are often shared among several subprograms so that operations in each of the subprograms may use the data. A data object may be transmitted as an explicit parameter between subprograms, but there are numerous occasions in which use of explicit parameters is cumbersome.

Explicitly transmitted parameters and results are the major alternative method for sharing data objects among subprograms.

Four basic approaches to nonlocal environments are in use in programming languages:

1. Explicit common environments and implicit nonlocal environments based on
2. Dynamic scope,
3. Static scope, and
4. Inheritance

The terms **argument** and **result** apply to data sent to and returned from the subprogram through a variety of language mechanisms. In narrowing our focus to parameters and parameter transmission, the terms actual parameter and formal parameter become central.

A **formal parameter** is a particular kind of local data object within a subprogram. The subprogram definition ordinarily lists the names and declarations for formal parameters as part of the specification part (heading). A formal parameter name is a simple identifier, and the declaration ordinarily gives the type and other attributes,

as for an ordinary local variable declaration. For example, the C procedure heading

```
SUB(int X; charY);
```

defines two formal parameters named X and Y and declares the type of each. The declaration of a formal parameter, however, does not mean the same thing as a declaration for a variable.

An **actual parameter** is a data object that is shared with the caller subprogram. An actual parameter may be a local data object belonging to the caller, it may be a formal parameter of the caller, it may be a nonlocal data object visible to the caller, or it may be a result returned by a function invoked by the caller and immediately transmitted to the called subprogram. An actual parameter is represented at the point of call of the subprogram by an expression, termed an actual-parameter expression, that ordinarily has the same form as any other expression in the language (e.g., such as an expression that might appear in an assignment statement). For example, the subprogram SUB specified earlier might be called with any of the following types of actual parameter expressions:

```
SUB(5,10);
```

### Methods for Transmitting Parameters

When a subprogram transfers control to another subprogram, there must be an association of the actual parameter of the calling subprogram with the formal parameter of the called program. Two approaches are often used: The actual parameter may be evaluated and that value passed to the formal parameter, or the actual data object may be passed to the formal parameter.

Several methods have generally been devised for passing actual parameters as formal parameters. There are four that are most common:

1. call by name,
2. call by reference,
3. call by value, and
4. call by value-result

### Static and Dynamic Scoping

The scope of a variable x is the region of the program in which uses of x refers to its declaration. One of the basic reasons of scoping is to keep variables in different parts of program distinct from one another. Since there are only a small number of short variable names, and programmers share habits about naming of variables (e.g., i for an array index), in any program of moderate size the same variable name will be used in multiple different scopes.

Scoping is generally divided into two classes:

1. Static Scoping
2. Dynamic Scoping

Static Scoping:

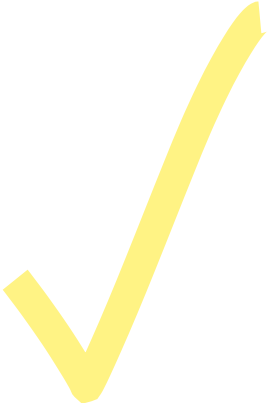
Static scoping is also called lexical scoping. In this scoping a variable always refers to its top level environment. This is a property of the program text and unrelated to the run time call stack. Static scoping also makes it much easier to make a modular code as programmer can figure out the scope just by looking at the code. In contrast, dynamic scope requires the programmer to anticipate all possible dynamic contexts.

In most of the programming languages including C, C++ and Java, variables are always statically (or lexically) scoped i.e., binding of a variable can be determined by program text and is independent of the run-time function call stack.

For example, output for the below program is 10, i.e., the value returned by f() is not dependent on who is calling it (Like g() calls it and has a x with value 20). f() always returns the value of global variable x.

// A C program to demonstrate static scoping.

```
#include<stdio.h>
int x = 10;
// Called by g()
int f()
{
    return x;
}
// g() has its own variable
// named as x and calls f()
int g()
{
    int x = 20;
    return f();
}
int main()
{
    printf("%d", g());
    printf("\n");
    return 0;
}
```




Dynamic Scoping:

With dynamic scope, a global identifier refers to the identifier associated with the most recent environment, and is uncommon in modern languages. In technical terms, this means that each identifier has a global stack of bindings and the occurrence of an identifier is searched in the most recent binding. In simpler terms, in dynamic scoping the compiler first searches the current block and then successively all the calling functions. Since dynamic scoping is very uncommon in the familiar languages, we consider the following pseudo code as our example. It prints 20 in a language that uses dynamic scoping.

```
int x = 10;
// Called by g()
int f()
```

```
{
    return x;
}
// g() has its own variable
// named as x and calls f()
int g()
{
    int x = 20;
    return f();
}
main()
{
    printf(g());
}
```



### Static Vs Dynamic Scoping

In most of the programming languages static scoping is dominant. This is simply because in static scoping it's easy to reason about and understand just by looking at code. We can see what variables are in the scope just by looking at the text in the editor.

Dynamic scoping does not care how the code is written, but instead how it executes. Each time a new function is executed, a new scope is pushed onto the stack.

Perl supports both dynamic and static scoping. Perl's keyword "my" defines a statically scoped local variable, while the keyword "local" defines dynamically scoped local variable.