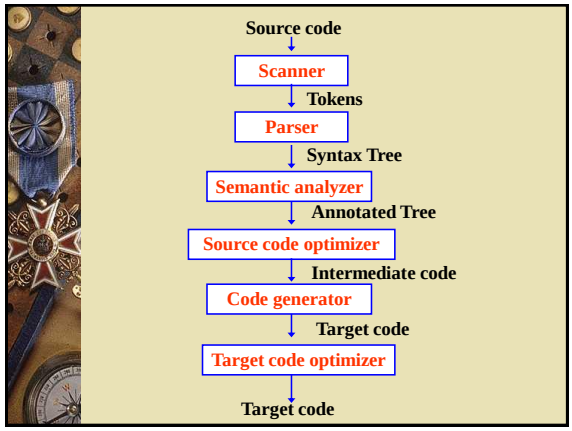
Chapter 7 Runtime Environments

Study Goals：

- **Master**
  Parameter Passing Mechanisms
- **Understand**
  The main idea of Fully static, Stack-Based, and Fully dynamic runtime environments
- **Know**



- **Scanning, Parsing and Static semantic analysis are all the phases of a compiler that perform static analysis of the source language, this analysis depends only on the properties of the source language**
- **The task of code generation is dependent on the details of the target machine.**

Nevertheless, the general characteristics of code generation remain the same across a wide variety of architectures, such as runtime environment

- **Runtime Environment**
  is the structure of the target computer's registers and memory that serves to manage memory and maintain the information needed to guide the execution process
  - **Registers and memory allocation is performed during execution**
  - **Designing the runtime environment during compilation can maintain an environment only indirectly**
  - **It must generate code to perform the necessary maintenance operations during program execution**

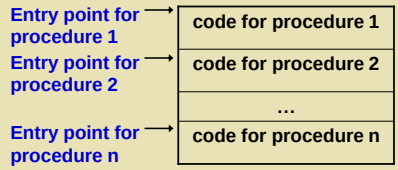
- **Three kinds of runtime environment**
  - **Fully static environment**
  - **Stack-based environment**
  - **Fully dynamic environment**

7.1 Memory Organization During Program Execution

- The memory of a typical computer may be divided into:
  - A register area
  - A slower directly addressable random access memory (RAM)

- The RAM area may be further divided into a code area and a data area
  - **Code area**

    Code area is fixed prior to execution, all code address are computable at compiler time



Code memory

  - **Data area**

    The memory area used for the allocation of data that code will access during execution
    - Global/static area
    - Stack area
    - Heap area

    } dynamic data area

- **Classes of Data Area**

1) Global/Static Area

- Data that can be fixed in memory prior to execution and that comprises the global and static data of a program
- Such data are usually allocated separately in a fixed area in a similar fashion to the code

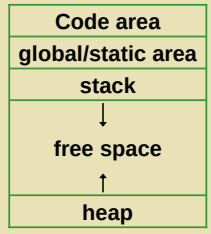
2) Stack Area

Stack area is used for data whose location occurs in LIFO (last-in, first-out) fashion

3) Heap Area

- Heap area is used for dynamic allocation that does not conform to a LIFO protocol
- Such as pointer allocation and deallocation: **new** and **delete** in C++, **malloc** and **free** in C

- The general organization of Runtime storage



- Stack and heap occupy the same area
- The arrows indicate the direction of growth of the stack and heap

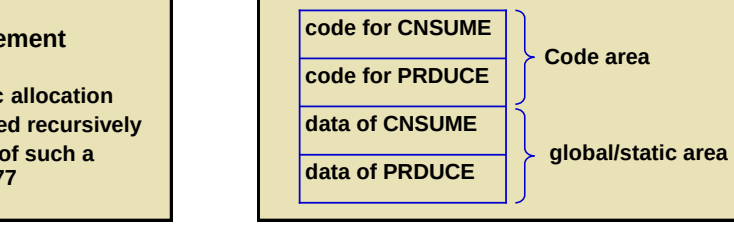
7.2 Three Kinds of Runtime Environment

1. Fully Static Runtime Environments

- All data are static, remaining fixed in memory for the duration of program execution
- It can be used to implement language in which:
  - No pointers or dynamic allocation
  - Procedures not be called recursively
  - The standard example of such a language is FORTRAN77

Example: FORTRAN program

```
Program CNSUME      //main program
……
Character function PRDUCE()     //sub function
……
```

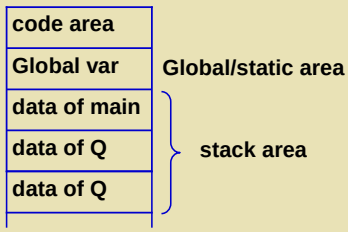


2. Stack-based Runtime Environments

- Data space is allocated at the top of the stack as a new procedure call is made and deallocated again when the call exits
- It is the most common form of runtime environment among the standard imperative languages such as C, Pascal, where:
  - Recursive calls are allowed
  - Local variables are newly allocated at each call

```
Program main;
global variables;
Procedure R;
…
End(R);
Procedure Q;
…
End(Q);
body of main
End.(main)
```

Suppose that Q is called in the body of main, Q recursively calls itself, the runtime environment when Q is called the second time is as following



3. Fully Dynamic Runtime Environments

- Fully Dynamic Runtime Environments
  - Data space is allocated and deallocated at arbitrary times during execution. Data space is dynamically freed only when all references to it have disappeared
  - A fully dynamic runtime environment is significantly more complicated than a stack-based environment, since it involves the tracking of references during execution, and the ability to find and deallocate inaccessible areas of memory at arbitrary times during execution (this process is called garbage collection)

- Heap Management and Fully Dynamic Runtime Environments
  - Heap management uses allocate and free operations to perform dynamic allocation and deallocation of pointers, it is a manual method, since the programmer must write explicit calls to allocate and free memory
  - In a language that needs a fully dynamic runtime environment, the heap must be managed automatically

7.3 Parameter Passing Mechanisms

- **Procedure activation record**
  - It is the memory allocated for the procedure or function when it is called
  - An activation record must contain the following sections in minimum

space for parameters
space for bookkeeping information, including return address
space for local data
space for local temporaries

- **Parameter Passing**
  - In a procedure call, parameters correspond to locations in the activation record, which are filled in with the arguments by the caller, prior to jumping to the code for the called procedure
  - How the argument values are interpreted by the procedure code depends on the particular parameter passing mechanisms adopted by the source language

- **Parameter passing mechanisms**
  - pass by value (call by value)
  - pass by reference (call by reference)
  - pass by value-result (copy-restore)
  - pass by name (delayed evaluation)

1 Pass by Value

- **Arguments**

  The arguments are expressions
- **Parameters passing**
  - Arguments are evaluated at the time of the call, and their values are stored in locations corresponding to parameters in the activation record
  - During the execution of the called procedure, code will access these eventual value directly

- **Result**

  Value parameters are viewed as initialized local variables, but changes to them never cause any nonlocal changes to take place.

Example

①Parameter passing

```
//Sub procedure definition
procedure  SWAP(n, m: real);
 var  j: real;
begin j:=n; n:=m; m:=j
end;
//The body of main
int i;
int k[10];
i:=5;
k[i]:=6;
SWAP(i, k[i]);
```

(add_i represent the address of i)

data area of caller

add_i	5
add_K[i]	6

data area of callee

add_n	5
add_m	6
add_j	

①Parameter passing(add_i represent the address of i)

data area of caller

add_i	5
add_K[i]	6

data area of callee

add_n	5
add_m	6
add_j	

②After the code of called procedure is performed "j:=n;n:=m;m:=j"

data area of caller

add_i	5
add_K[i]	6

data area of callee

add_n	6
add_m	5
add_j	5

Result :i=5，k[5]=6 nonlocal variable is not changed

2 Pass by Reference

- Arguments

  The arguments must be variables with allocated locations
- Parameters passing
  - Pass by reference passes the location of the variable (the address of the argument), which is stored in the local activation record
  - During the execution of the called procedure, the compiler must turn local access to a reference parameter into indirect access, since the local "value" is actually the address elsewhere in the environment

- Result

  The parameter becomes an alias for the argument (it does not require a copy to be made of the passed value), and any changes made to the parameter occur to the argument as well

Example

①Parameter passing

```
//Sub procedure definition
procedure  SWAP(n, m: real);
 var  j: real;
begin j:=n; n:=m; m:=j
end;

//The body of main
int i;
int k[10];


i:=5;
k[i]:=6;
SWAP(i, k[i]);
```

(add_i represent the address of i)

data area of caller

add_i	5
add_K[i]	6

data area of callee

add_n	add_I
add_m	add_K[i]
add_j	

①Parameter passing(add_i represent the address of i)

data area of caller

add_i	5
add_K[i]	6

data area of callee

add_n	add_i
add_m	add_K[i]
add_j	

②After the code of called procedure is performed "j:=n;n:=m;m:=j"

data area of caller

add_i	6
add_K[i]	5

data area of callee

add_n	add_i
add_m	add_k[i]
add_j	5

Result :i=6，k[5]=5 ,changes occur to the arguments

3 Pass by Value-Result

- This mechanism achieves a similar result to pass by reference, except that no actual alias is established
- Parameters passing

  The value of the argument is copied and used in the procedure, and then the final value of the parameter is copied back out to the location of the argument when the procedure exits.
- This method is known as copy-in copy-out (or copy-restore)

Example

//Sub procedure definition

```
procedure  SWAP(n, m: real);
 var  j: real;
begin j:=n; n:=m; m:=j
end;
```

//The body of main

```
int i;
int k[10];

i:=5;
k[i]:=6;
SWAP(i, k[i]);
```

①Parameter passing

(add_i represent the address of i)

data area of caller

add_i	5
add_K[i]	6

data area of callee

add_n	5
add_m	6
add_j	

①Parameter passing(add_i represent the address of i)

data area of caller

add_i	5
add_K[i]	6

data area of callee

add_n	5
add_m	6
add_j	

②After the code of called procedure is performed "j:=n;n:=m;m:=j"

data area of caller

add_i	6
add_K[i]	5

data area of callee

add_n	6
add_m	5
add_j	5

Result :i=6，k[5]=5 , changes occur to the arguments

4 Pass by Name

- It is also called delayed evaluation

  The idea of pass by name is that the argument is not evaluated until its actual use in the called program.
- Parameters passing

  The text of an argument at the point of call is views as a function, which is evaluated every time the corresponding parameter name is reached in the code of the called procedure.

Example

```
procedure  SWAP(n, m: real);
var  j: real;
begin      j:=n; n:=m; m:=j
end;
```

//The body of main

```
i:=5;
k[i]:=6;
SWAP(i, k[i]);
…
```

Pass by name

In the body of main SWAP(i, k[i]) is replaced with

j:=i;	j=i=5
i:=k[i];	i=k[5]=6
k[i]:=j;	k[6]=j=5

The result is i=6,k[6]=5 and k[5] is left unchanged

- Explanation

  Pass by name can be viewed as copy the code of called procedure to where it is called in the caller, replacing the parameter names by corresponding argument names