Run-time Environments - Part 1

Y.N. Srikant
Computer Science and Automation
Indian Institute of Science
Bangalore 560 012

NPTEL Course on Compiler Design

Outline of the Lecture – Part 1

- What is run-time support?
- Parameter passing methods
- Storage allocation
- Activation records
- Static scope and dynamic scope
- Passing functions as parameters
- Heap memory management
- Garbage Collection

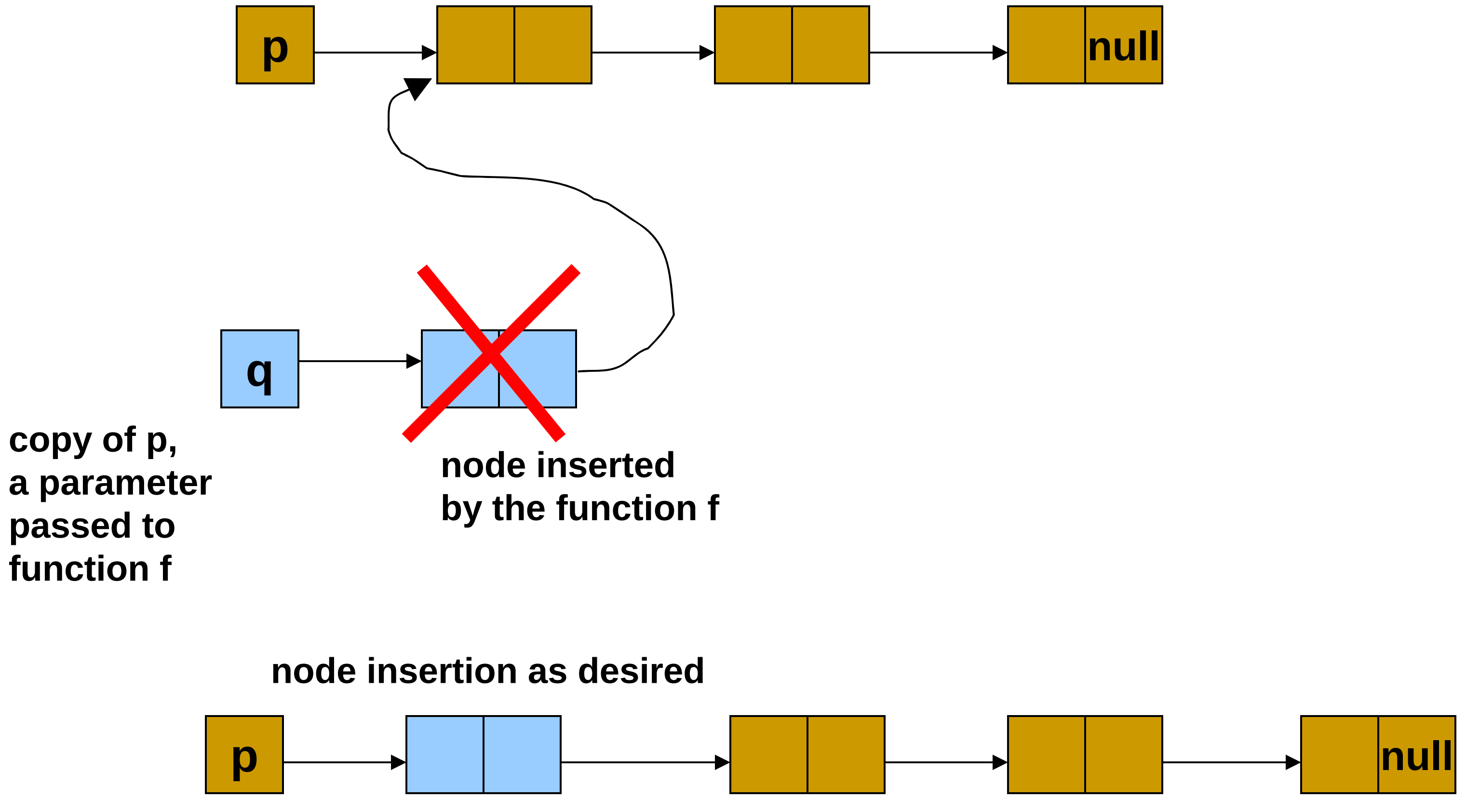
What is Run-time Support?

- It is not enough if we generate machine code from intermediate code
- Interfaces between the program and computer system resources are needed
  - There is a need to manage memory when a program is running
    - This memory management must connect to the data objects of programs
    - Programs request for memory blocks and release memory blocks
    - Passing parameters to fucntions needs attention
  - Other resources such as printers, file systems, etc., also need to be accessed
- These are the main tasks of run-time support
- In this lecture, we focus on memory management

Parameter Passing Methods - Call-by-value

- At runtime, prior to the call, the parameter is evaluated, and its actual value is put in a location private to the called procedure
  - Thus, there is no way to change the actual parameters.
  - Found in C and C++
  - C has only call-by-value method available
    - Passing pointers does not constitute call-by-reference
    - Pointers are also copied to another location
    - Hence in C, there is no way to write a function to insert a node at the front of a linked list (just after the header) without using pointers to pointers

Problem with Call-by-Value



Parameter Passing Methods - Call-by-Reference

- At runtime, prior to the call, the parameter is evaluated and put in a temporary location, if it is not a variable
- The **address** of the variable (or the temporary) is passed to the called procedure
- Thus, the actual parameter may get changed due to changes to the parameter in the called procedure
- Found in C++ and Java

Call-by-Value-Result

- ***Call-by-value-result*** is a hybrid of Call-by-value and Call-by-reference
- Actual parameter is calculated by the calling procedure and is copied to a local location of the called procedure
- Actual parameter's value is not affected during execution of the called procedure
- At return, the value of the formal parameter is copied to the actual parameter, if the actual parameter is a variable
- Becomes different from call-by-reference method
  - when global variables are passed as parameters to the called procedure and
  - the same global variables are also updated in another procedure invoked by the called procedure
- Found in Ada

Difference between Call-by-Value, Call-by-Reference, and Call-by-Value-Result

```
program RTST;
  var a: integer;
  procedure Q;
    begin a:= a+1; end
  procedure R(x:integer);
    begin x:= x+10; Q; end
begin a:= 1; R(a); print(a); end
```

call-by-value	call-by-reference	call-by-value-result
2	**12**	**11**

Value of a printed

Note: In Call-by-V-R, value of x is copied into a, when proc R returns. Hence a=11.

Parameter Passing Methods - Call-by-Name

- Use of a call-by-name parameter implies a **textual** substitution of the formal parameter name by the **actual** parameter
- For example, if the procedure

  procedure *R (X,I :* ***integer****);*

  begin *I := 2; X := 5; I := 3; X := 1;* ***end****;*

  is called by *R(B[J*2], J)*

  this would result in (effectively) changing the body to

  begin *J :=2; B[J*2] := 5; J :=5; B[J*2] := 1;* ***end****;*

  just before executing it

Parameter Passing Methods - Call by Name

- Note that the actual parameter corresponding to ***X*** changes whenever ***J*** changes
  - Hence, we cannot evaluate the address of the actual parameter just once and use it
  - It must be recomputed every time we reference the formal parameter within the procedure
- A separate routine (called ***thunk***) is used to evaluate the parameters whenever they are used
- Found in Algol and functional languages

Example of Using the Four Parameter Passing Methods

1. **procedure swap**
   (x, y : integer);
2. **var temp : integer;**
3. **begin**
4. **temp := x;**
5. **x := y;**
6. **y := temp;**
7. **end (*swap*);**
8. **...**
9. **i := 1;**
10. **a[i]:=10; (* a: array[1..5] of integer *)**
11. **print(i,a[i]);**
12. **swap(i,a[i]);**
13. **print(i,a[1]);**

- Results from the 4 parameter passing methods (print statements)

call-by-value	call-by-reference	call-by-val-result	call-by-name
1 10	**1 10**	**1 10**	**1 10**
1 10	**10 1**	**10 1**	**error!**

Reason for the error in the Call-by-name Example

The problem is in the swap routine

temp := i; (* => temp:=1 *)
i := a[i]; (* => i:=10 since a[i]=10 *)
a[i] := temp; (* => a[10]:=1 => index out of bounds *)

Code and Data Area in Memory

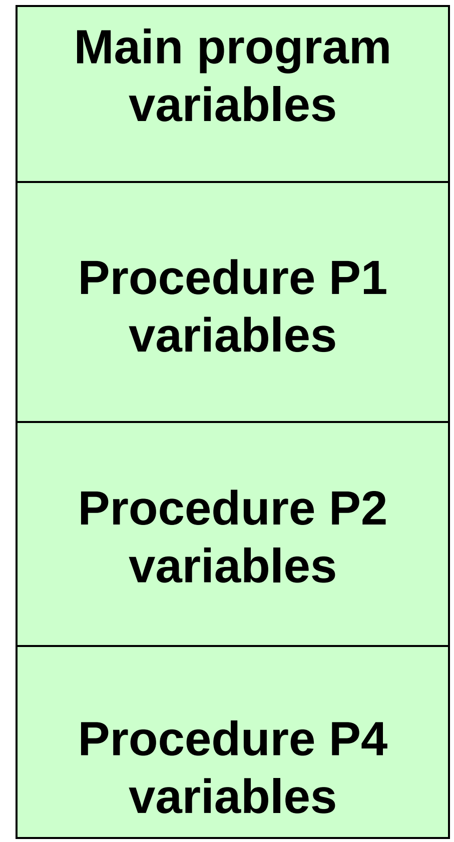
- Most programming languages distinguish between code and data
- Code consists of only machine instructions and normally does not have embedded data
  - Code area normally does not grow or shrink in size as execution proceeds
    - Unless code is loaded dynamically or code is produced dynamically
      - As in Java – dynamic loading of classes or producing classes and instantiating them dynamically through reflection
  - Memory area can be allocated to code statically
    - We will not consider Java further in this lecture
- Data area of a program may grow or shrink in size during execution

Static Versus Dynamic Storage Allocation

- Static allocation
  - Compiler makes the decision regarding storage allocation by looking only at the program text
- Dynamic allocation
  - Storage allocation decisions are made only while the program is running
  - Stack allocation
    - Names local to a procedure are allocated space on a stack
  - Heap allocation
    - Used for data that may live even after a procedure call returns
    - Ex: dynamic data structures such as symbol tables
    - Requires memory manager with garbage collection

Static Data Storage Allocation

- Compiler allocates space for all variables (local and global) of all procedures at compile time
  - No stack/heap allocation; no overheads
  - Ex: Fortran IV and Fortran 77
  - Variable access is fast since addresses are known at compile time
  - No recursion



Main memory

Dynamic Data Storage Allocation

- Compiler allocates space only for golbal variables at compile time
- Space for variables of procedures will be allocated at run-time
  - Stack/heap allocation
  - Ex: C, C++, Java, Fortran 8/9
  - Variable access is slow (compared to static allocation) since addresses are accessed through the stack/heap pointer
  - Recursion can be implemened

Activation Record Structure

Return address
Static and Dynamic links (also called Access and Control link resp.)
(Address of) function result
Actual parameters
Local variables
Temporaries
Saved machine status
Space for local arrays

Note:

The position of the fields of the act. record as shown are only notional.

Implementations can choose different orders; e.g., function result could be at the top of the act. record.