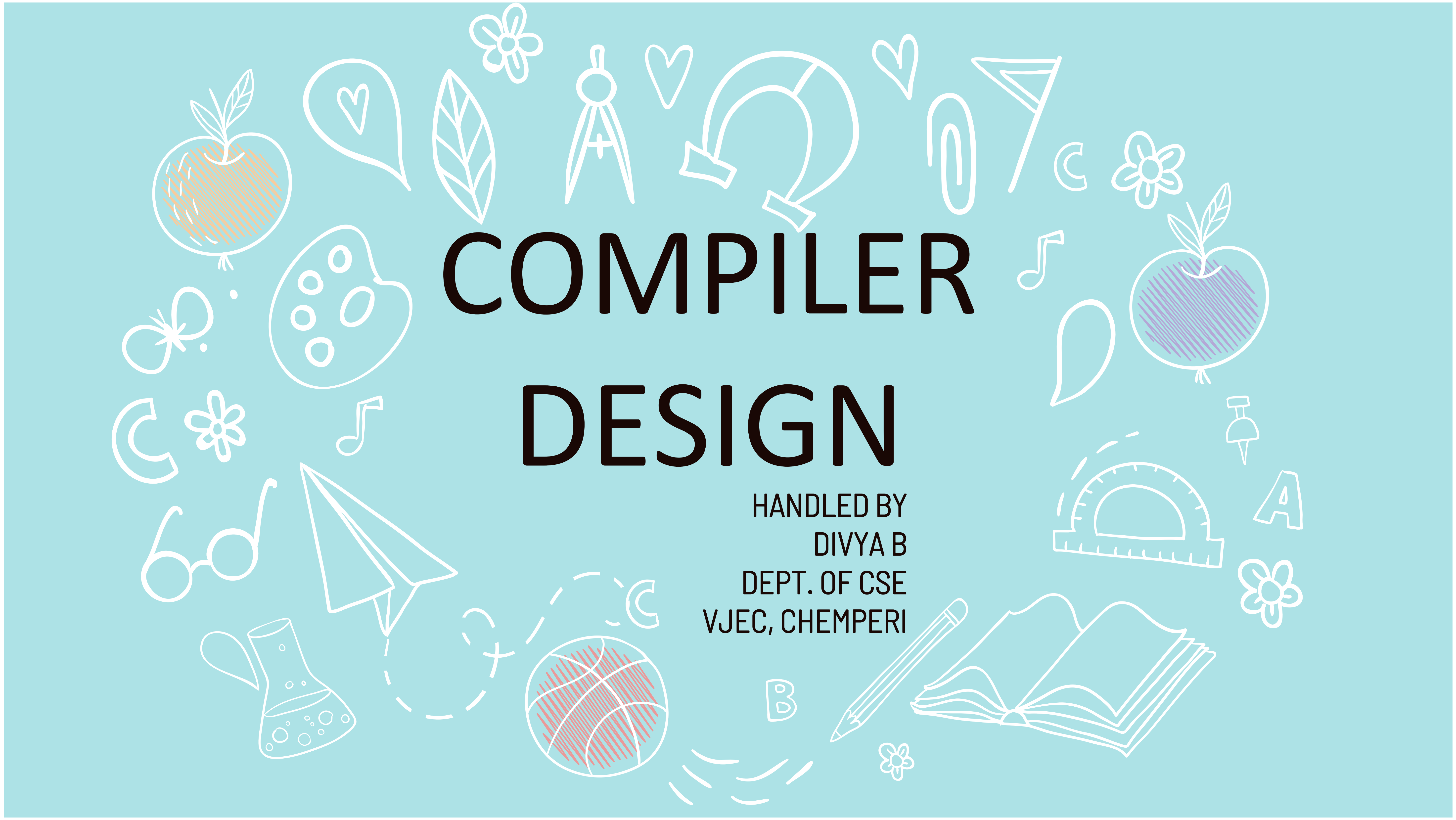


COMPILER DESIGN

HANDLED BY
DIVYA B
DEPT. OF CSE
VJEC, CHEMPERI

RUN-TIME ENVIRONMENTS

- A compiler must accurately implement the abstractions embodied in the source language definition.
- The abstractions include names, scopes, bindings, data types, operators, procedures, parameters and flow-of-control constructs.
- The compiler must cooperate with the OS and other systems to support these abstractions on the target machine.

RUN-TIME ENVIRONMENTS

- To support these abstractions, the compiler creates and manages a **run-time environment** in which it assumes its target programs are being executed.

RUN-TIME ENVIRONMENTS

Issues

- The layout and allocation of storage locations for the objects named in the source program.
- The mechanisms used by the target program to access variables.
- The linkages between procedures.
- The mechanisms for passing parameters.
- The interfaces to the OS, input/output devices and other programs.

RUN-TIME ENVIRONMENTS

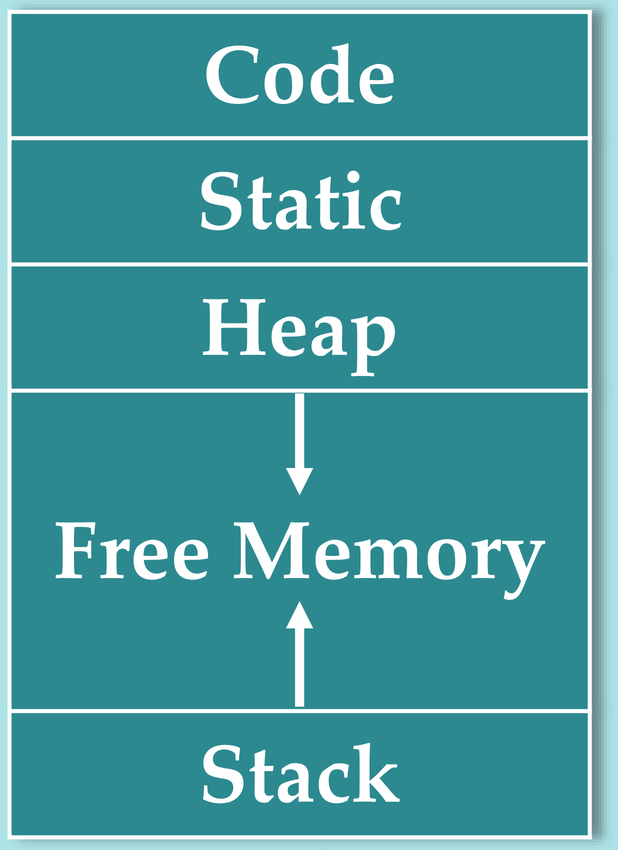
Storage Organization

- From the perspective of the compiler writer, the executing target program runs in its own logical address space in which each program value has a location.
- The management and organization of this logical address space is shared between the compiler, operating system, and target machine.
- The operating system maps the logical addresses into physical addresses, which are usually spread throughout memory.

RUN-TIME ENVIRONMENTS

Storage Organization

- The run-time representation of an object program in the logical address space consists of data and program areas.



Divys-Compiler Design PPT

A compiler for a language like C + + on an operating system like Linux might subdivide memory in this way.

RUN-TIME ENVIRONMENTS

Storage Organization

- It is assumed that the run-time storage comes in blocks of contiguous bytes, where a byte is the smallest unit of addressable memory.
- A byte is eight bits and four bytes form a machine word.
- Multibyte objects are stored in consecutive bytes and given the address of the first byte.

RUN-TIME ENVIRONMENTS

Storage Organization

- The amount of storage needed for a name is determined from its type.
- An elementary data type, such as a character, integer, or float, can be stored in an integral number of bytes.
- Storage for an aggregate type, such as an array or structure, must be large enough to hold all its components.

RUN-TIME ENVIRONMENTS

Storage Organization

- The storage layout for data objects is strongly influenced by the addressing constraints of the target machine.
- On many machines, instructions to add integers may expect integers to be **aligned**, that is, placed at an address divisible by 4.
- Although an array of ten characters needs only enough bytes to hold ten characters, a compiler may allocate 12 bytes to get the proper alignment, leaving 2 bytes unused.

RUN-TIME ENVIRONMENTS

Storage Organization

- Space left unused due to alignment considerations is referred to as **padding**.
- When space is at a premium, a compiler may pack data so that no padding is left; additional instructions may then need to be executed at run time to position packed data so that it can be operated on as if it were properly aligned.

RUN-TIME ENVIRONMENTS

Storage Organization

- The size of the generated target code is fixed at compile time, so the compiler can place the executable target code in a statically determined area called **code**, usually in the low end of memory.

RUN-TIME ENVIRONMENTS

Storage Organization

- The size of some program data objects, such as global constants, and data generated by the compiler, such as information to support garbage collection, may be known at compile time, and these data objects can be placed in another statically determined area called **static**.

RUN-TIME ENVIRONMENTS

Storage Organization

- One reason for statically allocating as many data objects as possible is that the addresses of these objects can be compiled into the target code.
- In early versions of Fortran, all data objects could be allocated statically.

RUN-TIME ENVIRONMENTS

Storage Organization

- To maximize the utilization of space at run time, the other two areas, **stack** and **heap**, are at the opposite ends of the remainder of the address space.
- These areas are dynamic; their size can change as the program executes.
- These areas grow towards each other as needed.
- The stack is used to store data structures called **activation records** that get generated during procedure calls.

RUN-TIME ENVIRONMENTS

Storage Organization

- In practice, the stack grows towards lower addresses, the heap towards higher addresses.
- But, we assume that the stack grow towards higher addresses so that positive offsets can be used for notational convenience.

RUN-TIME ENVIRONMENTS

Storage Organization

- An activation record is used to store information about the status of the machine, such as the value of the program counter and machine registers, when a procedure call occurs.
- When control returns from the call, the activation of the calling procedure can be restarted after restoring the values of relevant registers and setting the program counter to the point immediately after the call.

RUN-TIME ENVIRONMENTS

Storage Organization

- Data objects whose lifetimes are contained in that of an activation can be allocated on the stack along with other information associated with the activation.
- Many programming languages allow the programmer to allocate and deallocate data under program control.
- E.g., C has the functions **malloc** and **free** that can be used to obtain and give back arbitrary chunks of storage. Heap is used to manage this kind of long-life data.

RUN-TIME ENVIRONMENTS

Storage Organization

- *Static Vs Dynamic Storage Allocation*
  - ✓ The layout and allocation of data to memory locations in the run-time environment are key issues in storage management. These issues are tricky because the same name in a program text can refer to multiple locations at run time.
  - ✓ **Static** and **dynamic** distinguish between compile time and run time, respectively.

RUN-TIME ENVIRONMENTS

Storage Organization

- *Static Vs Dynamic Storage Allocation*
  - ✓ A storage-allocation decision is static, if it can be made by the compiler looking only at the text of the program, not at what the program does when it executes.
  - ✓ A decision is dynamic if it can be decided only while the program is running.
  - ✓ Many compilers use some combination of the following two strategies for dynamic storage allocation.

RUN-TIME ENVIRONMENTS

Storage Organization

- *Static Vs Dynamic Storage Allocation*
  - ✓ **Stack storage**
    - Names local to a procedure are allocated space on a stack.
    - The stack supports the normal call/return policy for procedures.

RUN-TIME ENVIRONMENTS

Storage Organization

- *Static Vs Dynamic Storage Allocation*
  - ✓ **Heap storage**
    - Data that may outlive the call to the procedure that created it is usually allocated on a "**heap**" of reusable storage.
    - The heap is an area of virtual memory that allows objects or other data elements to obtain storage when they are created and to return that storage when they are invalidated.

RUN-TIME ENVIRONMENTS

Storage Organization

- *Static Vs Dynamic Storage Allocation*
  - ✓ To support heap management, "**garbage collection**" enables the run-time system to detect useless data elements and reuse their storage, even if the programmer does not return their space explicitly.
  - ✓ Automatic garbage collection is an essential feature of many modern languages, despite it being a difficult operation to do efficiently; it may not even be possible for some languages.

RUN-TIME ENVIRONMENTS

Storage Organization

- *Static Vs Dynamic Storage Allocation*
  - ✓ To support heap management, "**garbage collection**" enables the run-time system to detect useless data elements and reuse their storage, even if the programmer does not return their space explicitly.
  - ✓ Automatic garbage collection is an essential feature of many modern languages, despite it being a difficult operation to do efficiently; it may not even be possible for some languages.

RUN-TIME ENVIRONMENTS

Storage Allocation Strategies

- Static Allocation
  - ✓ Lays out storage for all data objects at compile time.
- Stack Allocation
  - ✓ Manages the run-time storage as a stack.
- Heap Allocation
  - ✓ Allocates and deallocates storage as needed at run time from a data area known as heap.

RUN-TIME ENVIRONMENTS

Storage Allocation Strategies

- Static Allocation
  - ✓ The size of data objects is known at compile time.
  - ✓ The names of these objects are bound to storage at compile time itself.
  - ✓ The binding of name with the amount of storage allocated do not change at run time. Hence the name of this allocation is static allocation.

RUN-TIME ENVIRONMENTS

Storage Allocation Strategies

- Static Allocation
  - ✓ The compiler can determine the amount of storage required by each data object and therefore it becomes easy for a compiler to find the address of these data in the activation record.
  - ✓ At compile time itself, addresses of these objects can be compiled into the target code.

RUN-TIME ENVIRONMENTS

Storage Allocation Strategies

- Static Allocation - Limitations
  - ✓ The static allocation can be done only if the size of data object is known at compile time.
  - ✓ The data structures cannot be created dynamically. In the sense that, the static allocation cannot manage the allocation of memory at run time.

RUN-TIME ENVIRONMENTS

Storage Allocation Strategies

- Stack Allocation
  - ✓ The storage is organized as stack.
  - ✓ It is also called as **control stack**.
  - ✓ As activation begins the activation records are pushed onto the stack and on completion of this activation the corresponding activation records can be popped.

RUN-TIME ENVIRONMENTS

Storage Allocation Strategies

- Stack Allocation
  - ✓ The local data are stored in the each activation record. Hence these are bound to corresponding activation record on each fresh activation.
  - ✓ The data structures can be created dynamically for stack allocation.

RUN-TIME ENVIRONMENTS

Storage Allocation Strategies

- Stack Allocation – Calling Sequences
  - ✓Procedures called is implemented in what is called as **calling sequence**, which consists of code that allocates an activation record on the stack and enters information into its fields.
  - ✓A **return sequence** is similar to code to restore the state of a machine so that the calling procedure can continue its execution after the call.

RUN-TIME ENVIRONMENTS

Storage Allocation Strategies

- Stack Allocation – Calling Sequences
  - ✓The code in calling sequence is often divided between the calling procedure (**caller**) and a procedure it calls (**callee**).

RUN-TIME ENVIRONMENTS

Storage Allocation Strategies

- Stack Allocation – Calling Sequences
  - ✓When designing calling sequences and the layout of activation record, the following principles are helpful,
    1. Values communicated between the caller and the callee are generally placed at the beginning of the callee's activation record, so they are as close as possible to the caller's activation record.

RUN-TIME ENVIRONMENTS

Storage Allocation Strategies

- Stack Allocation – Calling Sequences
  - ✓When designing calling sequences and the layout of activation record, the following principles are helpful,
    2. Fixed length items are generally placed in the middle. Such items typically include the control link, the access link, and the machine status field.
    3. Items whose size may not be known early enough are placed at the end of the activation record.

RUN-TIME ENVIRONMENTS

Storage Allocation Strategies

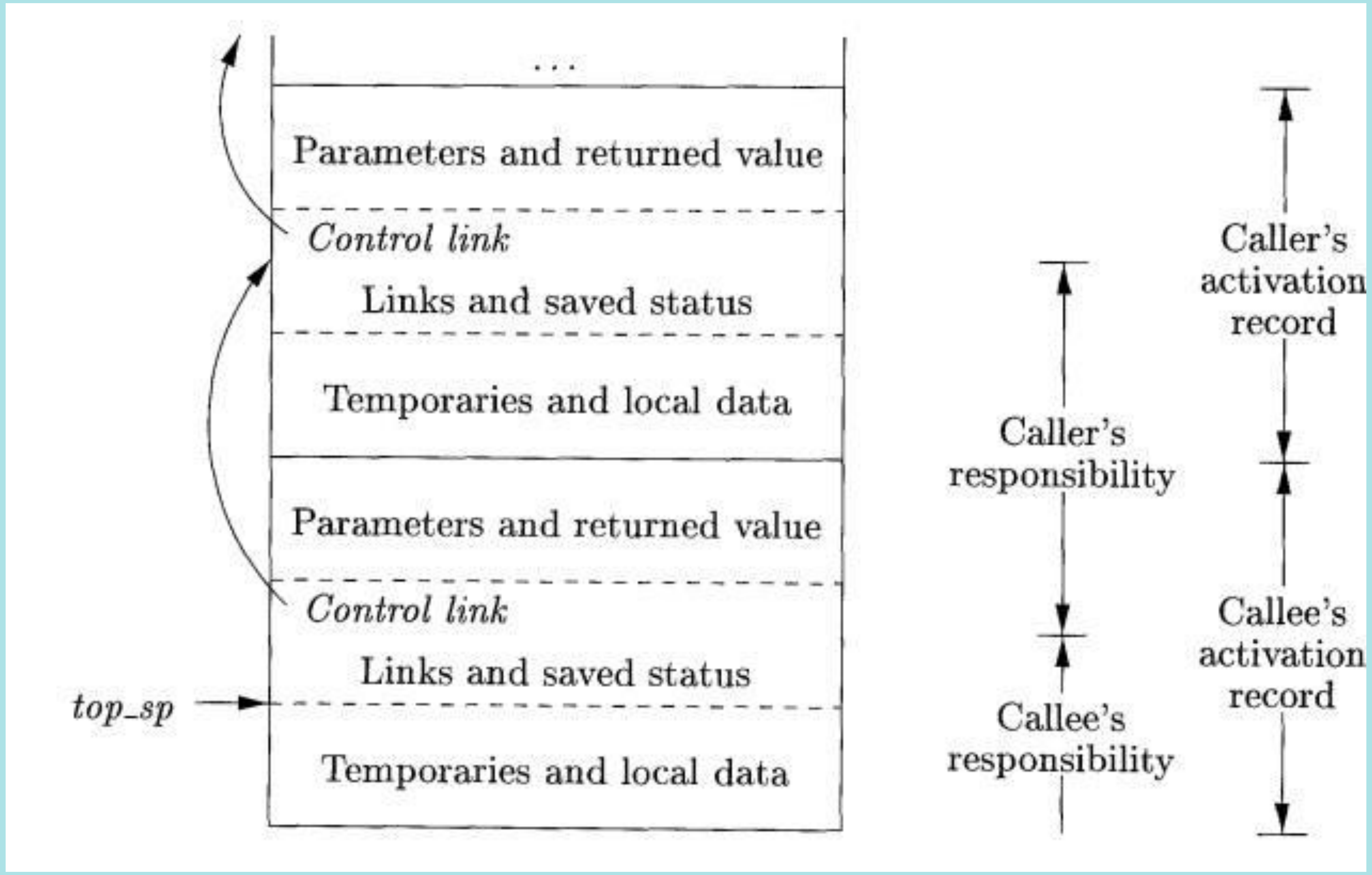
- Stack Allocation – Calling Sequences
  - ✓When designing calling sequences and the layout of activation record, the following principles are helpful,
    4. We must locate the top of the stack pointer judiciously. A common approach is to have it point to the end of fixed length fields in the activation record. Fixed length data can then be accessed by fixed offsets, known to the intermediate code generator, relative to the top of the stack pointer.

RUN-TIME ENVIRONMENTS

Storage Allocation Strategies

- Stack Allocation – Calling Sequences
  - ✓The calling sequence and its division between caller and callee are as follows
    1. The caller evaluates the actual parameters.
    2. The caller stores a return address and the old value of top_sp into the callee's activation record. The caller then increments the top_sp to the respective positions.
    3. The callee saves the register values and other status information.
    4. The callee initializes its local data and begins execution.

Division of tasks between caller and callee


. . .
Parameters and returned value
Control link
Links and saved status
Temporaries and local data
Parameters and returned value
Control link
Links and saved status
top_sp
Temporaries and local data
Caller's responsibility
Callee's responsibility
Caller's activation record
Callee's activation record

RUN-TIME ENVIRONMENTS

Storage Allocation Strategies

- Stack Allocation – Calling Sequences
  - ✓ Return sequence is as follows,
    1. The callee places the return value next to the parameters.
    2. Using the information in the machine status field, the callee restores top_sp and other registers, and then branches to the return address that the caller placed in the status field.

RUN-TIME ENVIRONMENTS

Storage Allocation Strategies

- Stack Allocation – Return Sequence
  3. Although top_sp has been decremented, the caller knows where the return value is, relative to the current value of top_sp; the caller, therefore, may use that value.

RUN-TIME ENVIRONMENTS

Storage Allocation Strategies

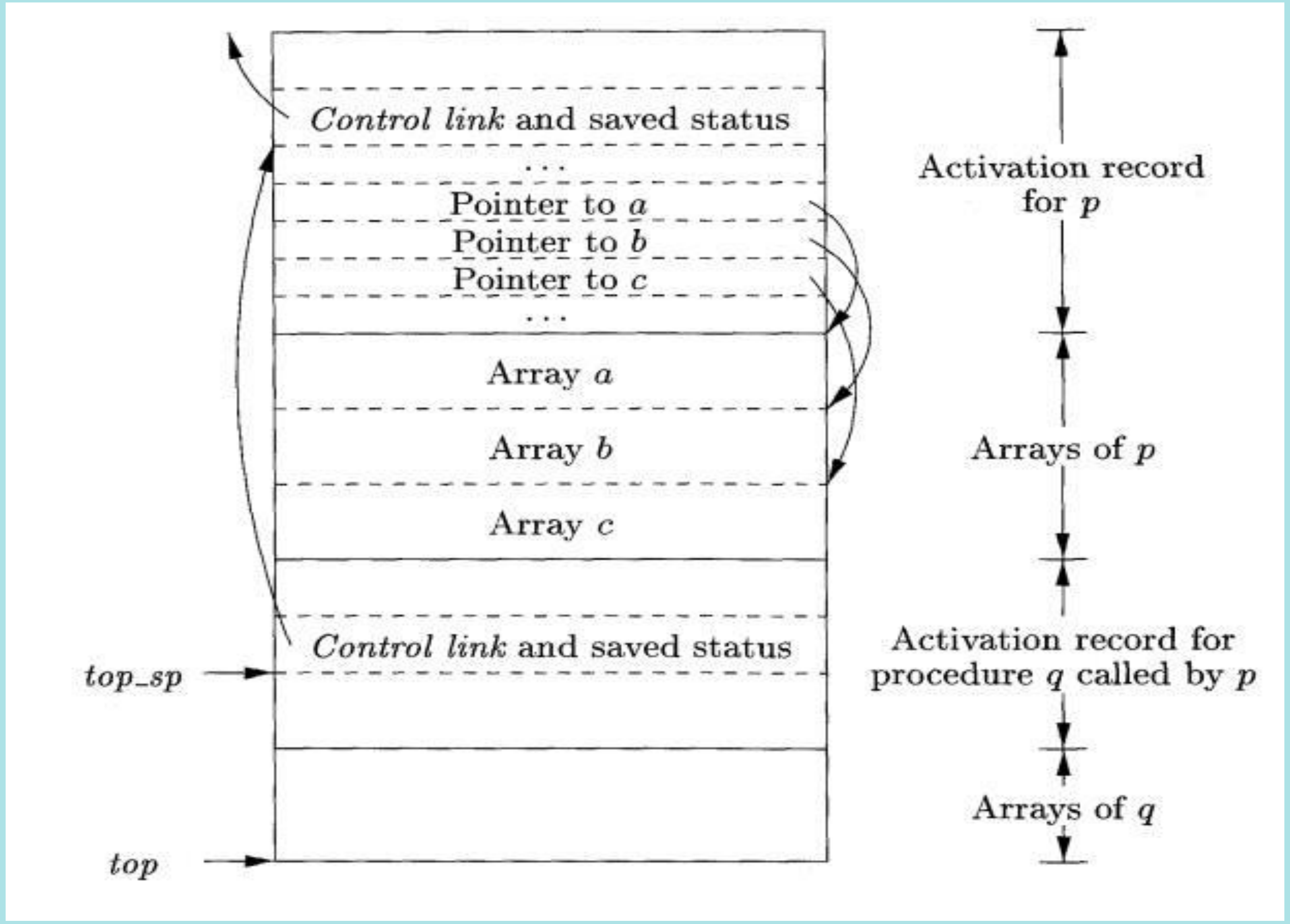
- Stack Allocation – Variable length data on the stack
  1. The run-time memory-management system must deal frequently with the allocation of space for objects the sizes of which are not known at compile time, but which are local to a procedure and thus may be allocated on the stack.
  2. In modern languages, objects whose size cannot be determined at compile time are allocated space in the heap.

RUN-TIME ENVIRONMENTS

Storage Allocation Strategies

- Stack Allocation – Variable length data on the stack
  3. It is also possible to allocate objects, arrays, or other structures of unknown size on the stack.
  4. We avoid the expense of garbage collecting their space. Note that the stack can be used only for an object if it is local to a procedure and becomes inaccessible when the procedure returns.

Access to dynamically allocated arrays

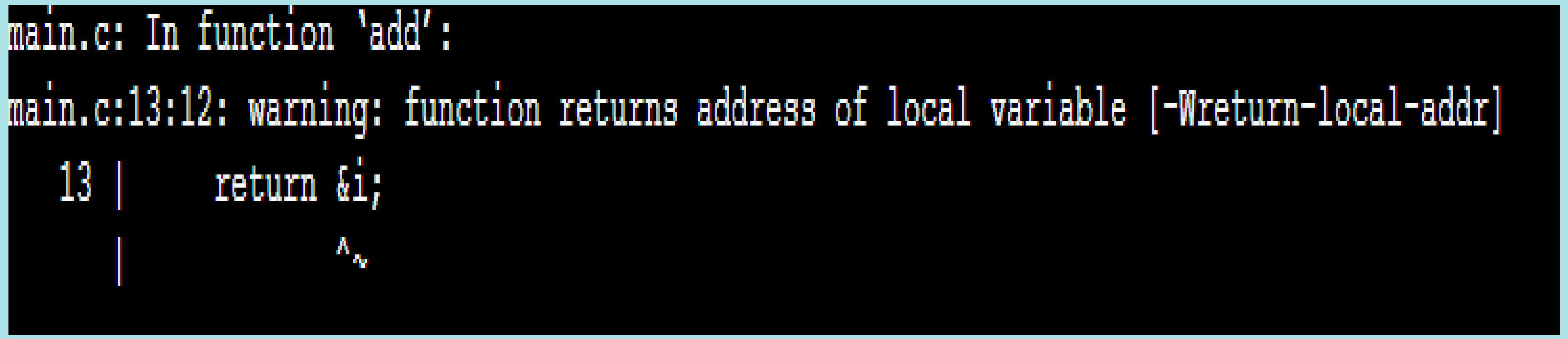

Control link and saved status
...
Pointer to *a*
Pointer to *b*
Pointer to *c*
...
Array *a*
Array *b*
Array *c*
Control link and saved status
top_sp
top
Activation record for *p*
Arrays of *p*
Activation record for procedure *q* called by *p*
Arrays of *q*

RUN-TIME ENVIRONMENTS

Storage Allocation Strategies

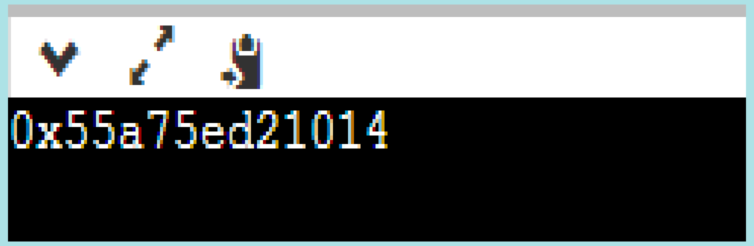
- Dangling Reference
  - ✓ A dangling reference is a reference to an object that no longer exists. Garbage is an object that cannot be reached through a reference.
  - ✓ Dangling references do not exist in garbage collected languages because objects are only reclaimed when they are no longer accessible (only garbage is collected).

```
#include <stdio.h>
int* add( )
{    int i=23;
     return &i;
}
void main()
{
     int *p;
     p=add();
     printf(“%p”,p);
}
```



```
main.c: In function 'add':
main.c:13:12: warning: function returns address of local variable [-Wreturn-local-addr]
   13 |     return &i;
      |            ^~
```

```
#include <stdio.h>
int i;
int* add( )
{    int i=23;
     return &i;
}
void main()
{
     int *p;
     p=add();
     printf("%p",p);
}
```



```
0x55a75ed21014
```

RUN-TIME ENVIRONMENTS

Storage Allocation Strategies

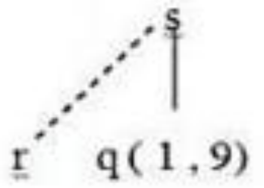
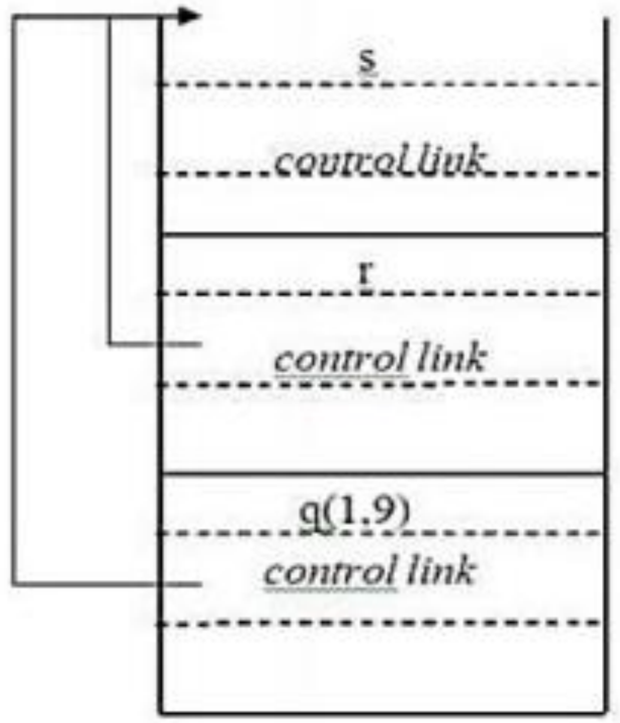
- Heap Allocation
  - ✓ Stack allocation strategy cannot be used if either of the following is possible
    1. The values of local names must be retained when an activation ends.
    2. A called activation outlives the caller.

RUN-TIME ENVIRONMENTS

Storage Allocation Strategies

- Heap Allocation
  - ✓ Heap allocation parcels out pieces of contiguous storage, as needed for activation records or other objects.
  - ✓ Pieces may be deallocated in any order, so over the time the heap will consist of alternate areas that are free and in use.

Storage Allocation Strategies

Position in the activation tree	Activation records in the heap	Remarks
		Retained activation record for r

Divys-Compiler Design PPT

RUN-TIME ENVIRONMENTS

Storage Allocation Strategies

- Heap Allocation
  - ✓ The record for an activation of procedure r is retained when the activation ends.
  - ✓ The record for the new activation q(1, 9) cannot follow s physically.
  - ✓ If the retained activation record for r is deallocated, there will be free space in the heap between the activation records for s and q.

RUN-TIME ENVIRONMENTS

Source Language Issues

- A program is made up of procedures.
- Almost all compilers for languages that use procedure, functions, or methods manage their run-time memory as a stack.
- Whenever a procedure is called, the local variable's space is pushed into a stack and popped off from the stack when the procedure terminates.

RUN-TIME ENVIRONMENTS

Source Language Issues

- This arrangement allows space to be shared by procedure calls whose durations do not overlap in time.
- It allows us to compile code for a procedure in such a way that the relative addresses of its non local variables are always the same, regardless of the sequence of procedure calls.

RUN-TIME ENVIRONMENTS

Source Language Issues

- Activation Trees
- Activation Records

Source Language Issues -
Activation Trees

- A program is a set of instructions with some operation associated with it.
- The sequence in which the procedure executes is known as an activation.
- A procedure executes in a sequential manner, and this sequence is easily represented by a tree known as the **activation tree**.

Source Language Issues -
Activation Trees

- If an activation of a procedure p calls procedure q, then that activation of q must end before the activation of p can end.
- There are 3 common cases,
  1. The activation of q terminates normally. Then in any language, control resumes just after the point of p at which the call to q was made.

Source Language Issues - Activation Trees

- There are 3 common cases,

  2. The activation of q or some procedure q called, either directly or indirectly, aborts; i.e., it becomes impossible to continue. In this case, p ends simultaneously with q.

Source Language Issues -
Activation Trees

- There are 3 common cases,
  3. The activation of q terminates because of an exception that q cannot handle. Procedure p can handle the exception. In this case the activation of q has terminated while the activation of p continues, although not necessarily from the point at which the call to q was made. If p cannot handle the exception, then this activation of p terminates at the same time as the activation of q, and presumably the exception will be handled by some other open activation of a procedure.

Source Language Issues -
Activation Trees

- The activation of procedures during the running of an entire program can be represented by a tree called **activation tree**.
- Each node corresponds to one activation.
- Root is the activation of the "main" procedure that initiates the execution of the program.
- At a node for an activation of procedure p, the children correspond to activations of the procedures called by this activation of p.

Source Language Issues - *Activation Trees*

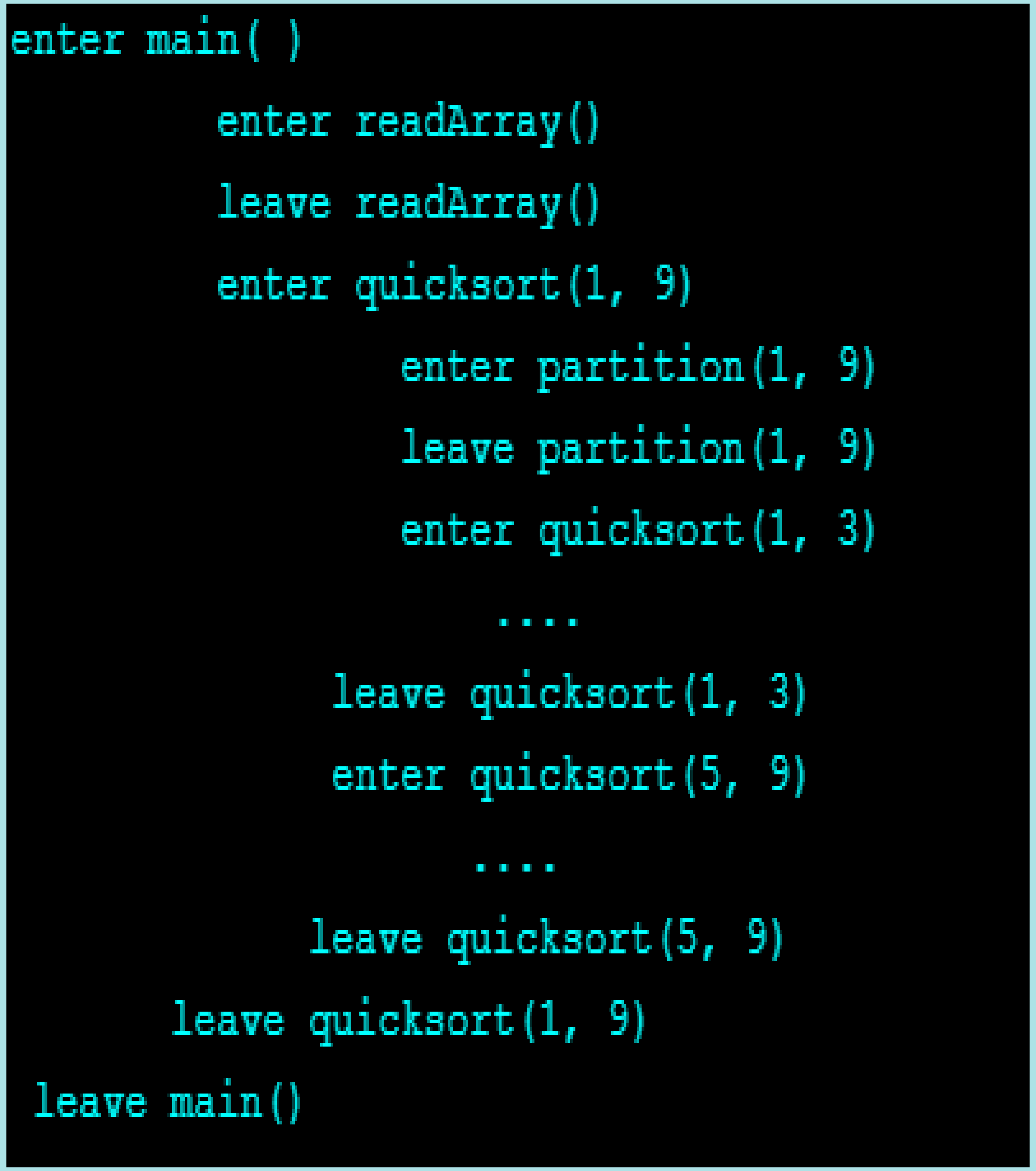
- Activations are shown in the order that they are called, i.e., from left to right.
- Left child must finish before the activation on the right can begin.

Source Language Issues - Activation Trees

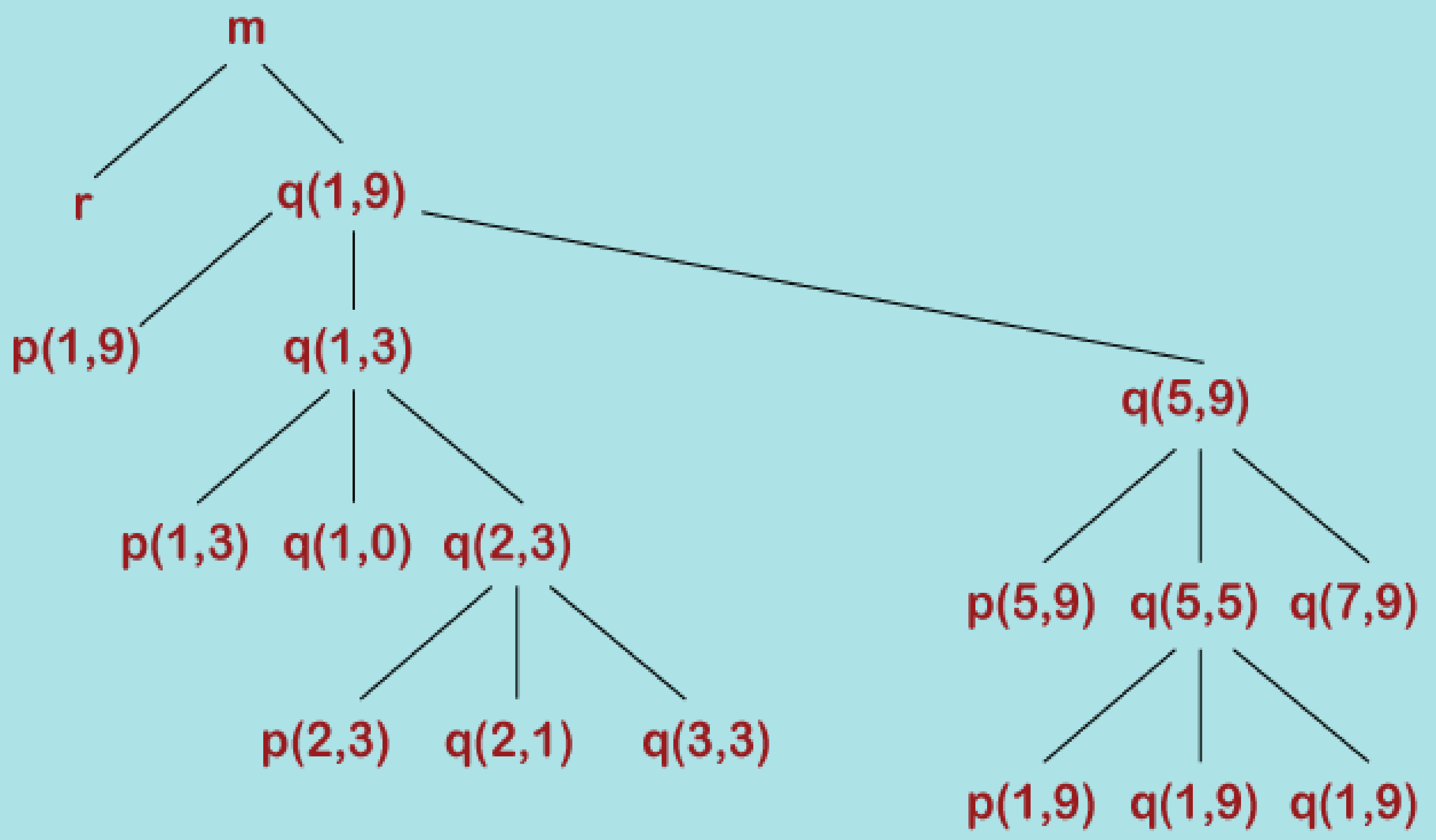
- The sequence of procedure calls correspond to a preorder traversal of the activation tree.
- The sequence of returns correspond to a postorder traversal of the activation tree.

```
int a[11];
 void readArray() {
       int i;
       ...
 }
 int partition(int m, int n)  {
   ....
 }
 void quicksort(int m, int n) {
 int i;
 if (n > m) {
 i = partition(m, n);
 quicksort(m, i-1);
 quicksort(i+1, n);
            }
 }
 main() {
      readArray();
      a[0] = -9999;
      a[10] = 9999;
      quicksort(1,9);
 }
```

```
enter main( )
          enter readArray()
          leave readArray()
          enter quicksort(1, 9)
                   enter partition(1, 9)
                   leave partition(1, 9)
                   enter quicksort(1, 3)
                       ....
                leave quicksort(1, 3)
                enter quicksort(5, 9)
                      ....
               leave quicksort(5, 9)
        leave quicksort(1, 9)
 leave main()
```



```
enter main( )
          enter readArray()
          leave readArray()
          enter quicksort(1, 9)
                   enter partition(1, 9)
                   leave partition(1, 9)
                   enter quicksort(1, 3)
                        ....
                leave quicksort(1, 3)
                enter quicksort(5, 9)
                       ....
               leave quicksort(5, 9)
        leave quicksort(1, 9)
 leave main()
```



Source Language Issues - Activation Records

- Procedure calls and returns are usually managed by a runtime stack called **control stack**.
- Each live activation has an activation record (frame) on the control stack.
- Root of the activation tree will be at the bottom of the stack.
- The latter activation has its record at the top of the stack.

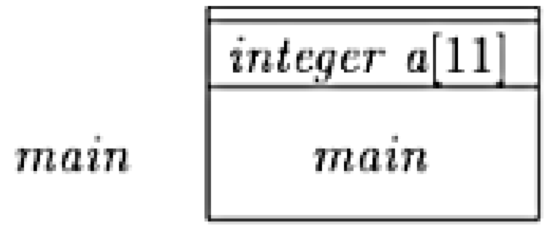
Source Language Issues - Activation Records

Actual parameters
Returned values
Control link
Access Link
Saved machine status
Local data
Temporaries

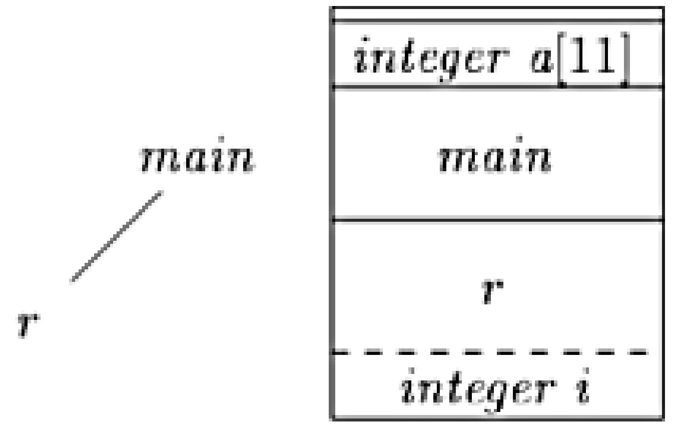
- ✓ **Temporaries:** The values which arise from the evaluation of expression will be held by temporaries.
- ✓ **Local data:** The data belonging to the execution of the procedure is stored in local data.
- ✓ **Saved machine status:** The status of the machine that might contain **register**, **program counter** before the call to the procedure is stored in saved machine status.
- ✓ **Access link:** The data's information outside the local scope is stored in the access link**.**
- ✓ **Control link:** The activation record of the caller is pointed by the control link.
- ✓ **Returned values:** It represents the space for the return value of the called function (if any).
- ✓ **Actual parameters:** It represents the actual parameters used by the calling procedure.

Source Language Issues -
Activation Records

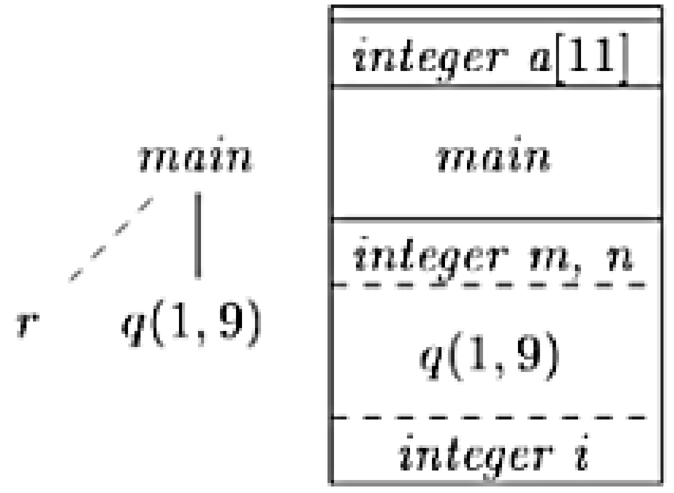
- Procedure calls and returns are usually managed by a runtime stack called **control stack**.
- Each live activation has an activation record (frame) on the control stack.
- Root of the activation tree will be at the bottom of the stack.
- The latter activation has its record at the top of the stack.



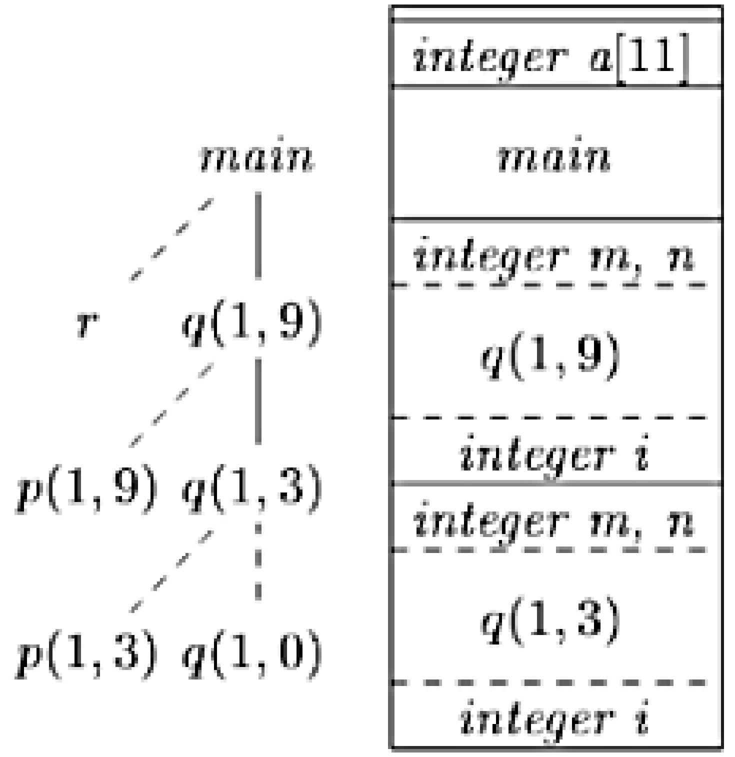
(a) Frame for *main*



(b) *r* is activated



(c) *r* has been popped and *q*(1, 9) pushed



(d) Control returns to *q*(1, 3)