Chapter 7
Runtime Environments

2022 Spring&Summer

Outline

- Runtime Environment:
- Three kinds of runtime environments:
  - (1) Fully static environment; FORTRAN77
  - (2) stack-based environment; C C++
  - (3) fully dynamic environment; LISP
- Parameter Passing Mechanisms

7.3.1 Stack-Based Environments Without Local Procedures

Dealing with variable-length data

Data may vary both in the number of data objects and in the size of each object.

(1) The number of arguments in a call may vary from call to call.

(2) The size of an array parameter or a local array variable may vary from call to call

printf("%d%s%c",n,prompt,ch)	Has four arguments
printf("Hello, world\n")	Has only one argument

7.3.1 Stack-Based Environments Without Local Procedures

Dealing with variable-length data

- C compiler typically deal with this by pushing the arguments to a call *in reverse order* onto the runtime stack. The first parameter is always located at a fixed offset from the *fp* in the implementation described above.

- Another option is to use a processor mechanism such as *ap*(argument pointer) in VAX architecture.

7.3.1 Stack-Based Environments Without Local Procedures

Local Temporaries and Nested Declarations

- Local temporaries are partial results of computations that must be saved across procedure calls, for example:

$$x[i] = (i + j) * (i/k + f(j))$$

- The three partial results need to be saved across the call to *f*:

  The *address* of x[i];

  The sum $i+j$;

  The quotient i/k;

7.3.1 Stack-Based Environments Without Local Procedures

Local Temporaries and Nested Declarations

- Nested declarations present a similar problem.

```
void p( int x, double y)
{ char a;
  int i;
...
A: {double x;
        int j;
        ...
        }
...
B:{char *a;
        int k;
        ...
        }
        ...
}
```

7.3.1 Stack-Based Environments Without Local Procedures

- The nested local declarations do not need to be allocated until entered;
- The declarations of A and B do not need to be allocated simultaneously.
- A compiler could treat a block just like a procedure and create a new activation record each time a block is entered and discard it on exit. -- *This would be inefficient.*
- A simpler method is to treat them in a similar way to temporary expression.

7.3.2 Stack-Based Environment with local Procedures

- Consider the non-local and non-global references

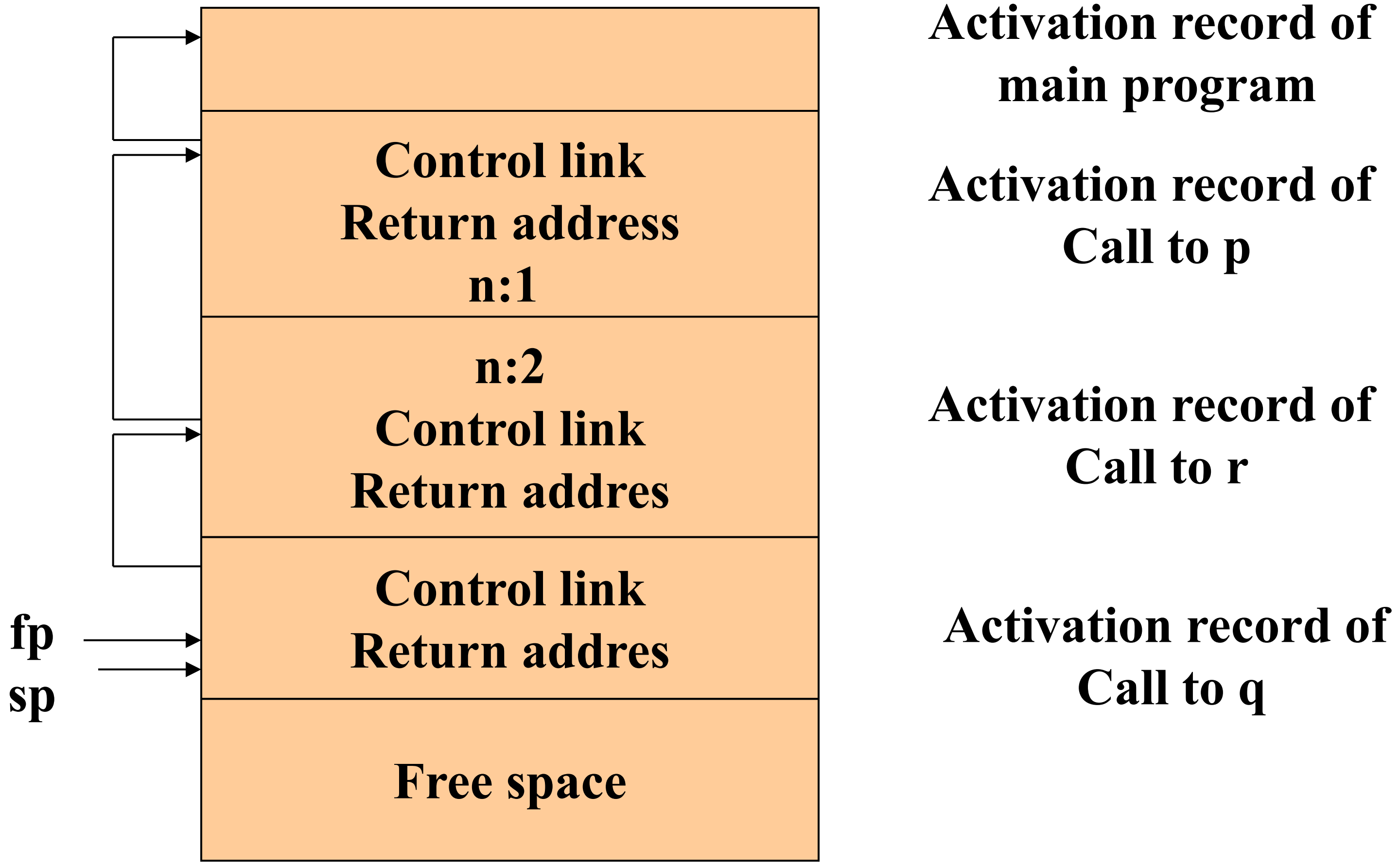
  Example: Pascal program showing *nonlocal,nonglobal reference*

- To solve the above problem about variable access, we add an extra piece of bookkeeping information called the *access link* to each activation record.

7.3.2 Stack-Based Environment with local Procedures

```
program nonlocalRef;

procedure  p;
var n: integer;

    procedure q;
    begin
        (* a reference to n is
        now non-local and
         non-global *)
    end;  (*q*)

    procedure r(n:integer);
    begin
        q;
    end; (*r*)

begin  (* p*)
     n :=1;
     r(2);
end;  (*p*)

begin (*main*)
   p;
end.
```

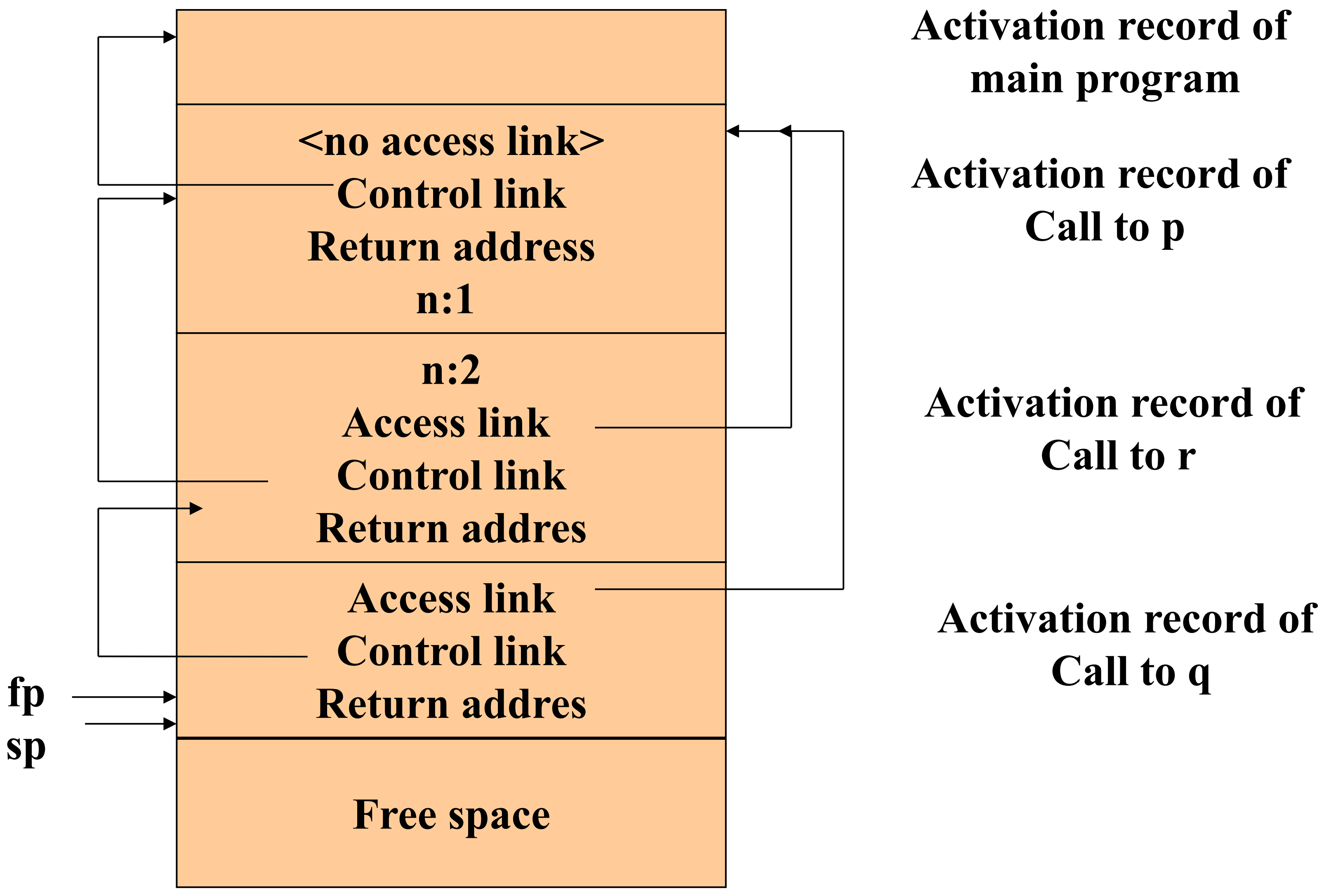
7.3.2 Stack-Based Environment with local Procedures


Control link
Return address
n:1
n:2
Control link
Return addres
Control link
Return addres
Free space
fp
sp
Activation record of main program
Activation record of Call to p
Activation record of Call to r
Activation record of Call to q

7.3.2 Stack-Based Environment with local Procedures

- *access link* represents the *defining environment* of the procedure; access link is sometimes also called the static link.
- *control link* represents the calling environment of the procedure.

7.3.2 Stack-Based Environment with local Procedures



7.3.2 Stack-Based Environment with local Procedures

- The calling sequence:

  (1) The access link must be pushed onto the runtime stack just before the *fp* during a call

  (2) The *sp* must be adjusted by an extra amount to remove the access link after an exit.

- How to find the *access link* of a procedure during a call.

  (1) Using the (compile-time) nesting level information attached to the declaration of the procedure;

  (2) Generate an access chain as if to access a variable at the same nesting level.

7.3.2 Stack-Based Environment with local Procedures

DISPLAY

A *display* is a data structure that may be used as an alternative to static links for maintaining access to nonlocal variables. It is an array of frame pointers, indexed by static nesting depth. Element Di of the display always points to the most recently called function whose static nesting depth is i .

The bookkeeping performed by a function *f* , whose static nesting depth is *i*, looks like:

Copy Di to save location in stack frame

Copy frame pointer to Di

··· body of f ···

Copy save location back to Di

7.4 DYNAMIC MEMORY

- A stack-based environment will result in a dangling reference .

  The procedure is exited if a local variable in a procedure can be returned to the caller.

- The simplest example:

  *int * dangle(void)*

  { int x;

  return &x;}

- An assignment *addr = dangle( )* now cause *addr* to point to an unsafe location in the activation stack.

7.4.1 Fully Dynamic Runtime Environments

- More complex instance of a dangling reference occurs if a local function can be returned by a call.

```
typedef int(* proc)(void);
proc g(int x)
{int f(void) /*illegal local function*/
        {return x;}
        return f;}
main()
{
    proc c;
    c= g(2);
    printf("%d\n",c());/* should print 2*/
    return 0;
}
```

7.4.1 Fully Dynamic Runtime Environments

- C language avoids the above problem.
  prohibiting local procedure;
- Other languages, like Mudula-2, have local procedures as well as procedure variables, parameters, and returned values.
  - must state a special rule that makes such program erroneous.
  - in Modula-2 the rule is that only global procedures can be arguments or returned values.

7.4.1 Fully Dynamic Runtime Environments

- A stack-based runtime environment is inadequate.
- A more general form of environment is required —— *Fully Dynamic Environment*
- It can deallocate activation records only when all references to them have disappeared
- Garbage collection:
  - The tracking of references during execution .
  - The ability to find and deallocate in accessible areas of memory at arbitrary times during execution.

7.4.1 Fully Dynamic Runtime Environments

- In fully dynamic environment, the basic structure of activation record remains the same.
- When control is returned to the caller, the exited activation record remains in memory, to be de-allocated at some later time.

7.4.2 Dynamic Memory in Object-Oriented Languages

- OO languages require special mechanisms in the runtime environment to implement their added features.
- Features :Objects, methods, inheritance, and dynamic binding.
- An object in memory can be viewed as a cross between a traditional record structure and an activation record.
- the instance variable (data members) as the fields of the record .

7.4.2 Dynamic Memory in Object-Oriented Languages

1. One straightforward mechanism :
   - For initialization code to copy all the currently inherited features (and methods) directly into the record structure. This is extremely wasteful of space.
2. Keep a complete description of the class structure in memory at each point during execution.
   - inheritance maintained by superclass pointers
   - all method pointers kept as fields in the class structure.

7.4.2 Dynamic Memory in Object-Oriented Languages

3.Compute the list of code pointers for available methods of each class , and store this in (static) memory as a virtual function table (in c++).

- It can be arranged so that each method has a predictable offset
- A traversal of the class hierarchy with a series of table lookups is no longer necessary.
- Each object contains a pointer to the appropriate virtual function table.

7.4.2 Dynamic Memory in Object-Oriented Languages

```
class A
{public：
double x,y;
void f();
virtual void g();
}

class B : public A
{public：
double z;
void f();
virtual void h();
}
```

7.4.3 Heap Management

- In most language, needs some dynamic capabilities in order to handle pointer allocation and de-allocation.
- *Heap* the data structure

(1) allocate

take a size parameter

return a pointer to a block of memory of the correct size，return a null pointer if none exists.

(2) free

takes a pointer to an allocated block of memory and marks it as being free again.

7.4.3 Heap Management

- A standard method for maintaining the heap and implementing these functions:
  (1) A circular linked list of free blocks.
  (2) Memory is taken by malloc.
  (3) Memory is return by free.
- Drawbacks:
  (1) The free operation can not tell if the pointer is legal or not.
  (2) Care must be taken to coalesce blocks, otherwise, the heap can quickly become fragmented.

7.4.3 Heap Management

- A different implementation of malloc and free.

  use a circular linked list data structure that keep track of both allocated and free block.

7.4.3 Heap Management

```
#define NULL 0
#define MEMSIZE 8096 /* change for different sizes */

typedef double align;
typedef union header
   {struct {union header *next;
           unsigned usedsize;
           unsigned freesize;
           } s;
      align a;
} header;

static header mem[MEMSIZE];
static header *memptr=NULL;
```

7.4.3 Heap Management

```
void *malloc(unsigned nbytes)
{ header *p, *newp;
unsigned nunits;
nunits = (nbytes+sizeof(header)-1)/sizeof(header)+1;
if (memptr == NULL)
      { memptr->s.next = memptr = mem;
       memptr->s.usedsize =
      memptr->s.freesize = M
      }
for(p=memptr;  (p->s.next!
```

```
if (p->s.freesize < nunits) return NULL;
                    /* no block big enough */
newp =p +p->s.usedsize;
newp->s.usedsize = nunits;
newp->s.freesize =p->s.freesize –nunits;
newp->s.next = p->s.next;
p->s.freesize=0;
p->s.next = newp;
memptr = newp;
return (void *) (newp+1);
}
```

7.4.3 Heap Management

```
void free(void *ap)
{ header *bp, *p, *prev;
          bp = (header *)ap –1;
          for (prev=memptr, p=memptr->s.next;
          (p!=bp) && (p!=memptr); prev=p, p=p->s.next);
           if (p!=bp) return;
                    /* corrupted list, do nothing */
           prev->s.freesize +=p->s.usedsize + p->s.freesize;
          prev->s.next = p->s.next;
               memptr = prev;
}
```

7.4.4 Automatic Management of the heap

- Manual method

  the use of *malloc* and *free* to perform dynamic allocation and de-allocation of pointer .

- Garbage collection

  the process of reclamation of allocated but no longer used storage without an explicit call to free.

7.4.4 Automatic Management of the heap

- Mark and sweep garbage collection

No memory is freed until a call to malloc fails, which does this in two passes.

(1) Follows all pointers recursively, starting with all currently accessible pointer values and marks each block of storage reached.

(2) Sweeps linearly through memory.

- ➢ returning unmarked blocks to free memory.
- ➢ perform memory compaction to leave only one large block of contiguous free space at the other end.

7.4.4 Automatic Management of the heap

- The Mark and Sweep garbage collection has several drawbacks:

  (1) Requires extra storage;

  (2) The double pass through memory cause a significant delay in processing

7.4.4 Automatic Management of the heap

- Stop-and-copy or two-space garbage collection

  (1) During the marking pass, all reached blocks are immediately copied to the second half of storage not in use;

  (2) No extra mark bit is required and only one pass is required;

  (3) It also performs compaction automatically.

  (4) It does little to improve processing delays during storage reclamation.

7.4.4 Automatic Management of the heap

- Generational garbage collection

  Allocated objects that survive long enough are simply copied into permanent space and are never deallocated during subsequent storage reclamations.

7.5 Parameter Passing Mechanisms

- The common parameter passing mechanisms
  - pass by value
  - pass by reference
  - pass by value-result
  - pass by name

7.5.1 Pass by Value

- The arguments are expressions that are evaluated at the time of the call.
- Their values becomes the values of the parameter during the execution of the procedure.
  - The only parameter passing mechanism available in C;
  - The default in Pascal and Ada.

7.5.2 Pass by Reference

- Pass by reference passes the location of the variable.
- The parameter becomes an alias for the argument.
  - The only parameter passing mechanism in Fortran77
  - In Pascal, pass by reference achieved with the use of var keyword
  - In C++, by the use of special symbol & in the parameter declaration

7.5.3 Pass by Value-Result

- The mechanism achieves a similar result to pass by reference, except that no actual alias is established.
- Known as copy-in, copy-out, or copy- restore.
- The value of the argument is copied and used in the procedure.
- The final value of the parameter is copied back out to the location of the argument.

This is the mechanism of Ada in out parameter.

7.5.3 Pass by Value-Result

- If pass by reference is used, *a* has value 3 after *p* is called.
- If pass by value-result, *a* has value 2 after *p* is called.

```
void p(int x, int y)
{ ++x;
  ++y;
}
main( )
{ int a=1;
  p(a, a);
return 0;
}
```

7.5.3 Pass by Value-Result

- The unspecified issues in this mechanism:
  - The order in which results are copied back to the arguments.
  - The locations of the arguments are calculated only on entry or recalculated on exit.

7.5.4 Pass by Name

- This is the most complex of the parameter passing mechanisms. *(delayed evaluation*)
- Idea:

The argument is not evaluated until its actual use in the called program.

7.5.4 Pass by Name

```
int i;
int a[10];


void p(int x)
{++i;
++x;
}
```

```
main()
{
 i=1;
a[1]=1;
a[2]=2;
p(a[i]);
return 0;
}
```

The result of the call *p(a[i])* is that *a[2]* is set to 3 and *a[1]* is left unchanged.

7.5.4 Pass by Name

- The interpretation of pass by name is as follows:
  (1) The text of an argument at point of call is viewed as a function in its own right.
  (2) The arguments are evaluated every time the parameter name is reached in the procedure.
  (3) The argument will always be evaluated in the caller's environment.

Homework of Chapter 7

7.4 Draw the stack of activation records for the following Pascal program , showing the control and access links , after the second

call to procedure **c**.Describe how the variable **x** is accessed from within **c**.

7.15 Give the output for the following program(written in C synt-ax)using the four parameter passing methods discussed in Section 7.5.

Homework of Chapter 7

7.4

```
program env;

procedure a;
var x: integer;

  procedure b;
        procedure c;
        begin
            x := 2;
            b;
        end;
  begin (* b *)
            c;
        end;

begin (* a *)
     b;
end;

begin (* main *)
       a;
  end.
```

7.15

```
#include <stdio.h>
int i=0;

void p(int x, int y)
{
  x += 1;
  i += 1;
  y += 1;
}

main()
{
  int a[2]={1,1};
  p(a[i],a[i]);
  printf("%d %d\n",a[0],a[1]);
  return 0;
}
```