

Compiler Principle and Technology

Prof. Dongming LU
Apr. 25th, 2015

7. Runtime Environments



Contents

Part One

7.1 Memory Organization During Program Execution

7.2 Fully Static Runtime Environments

7.3 Stack-Based Runtime Environments

Part Two

7.4 Dynamic Memory

7.5 Parameter Passing Mechanisms



7.4 Dynamic Memory

A series of horizontal lines in shades of red and white, extending across the bottom of the slide.

7.4.1 Fully Dynamic Runtime Environments

- A stack-based environment will result in a **dangling reference** ----when 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;
}
```

Where, the address of a local variable is returned.

- `addr = dangle()` will cause `addr` to **point to an unsafe location in the activation stack**



- ◆ A 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;
}
```



- **C language** avoids the above problem by prohibiting local procedure
- **Mudula-2** has local procedures as well as procedure variables, parameters, and returned values, **must state a special rule that makes such program erroneous**
- In the **functional programming languages**, such as **LISP** and **ML**, the functions must be able to be locally defined, passed as parameters, and returned as results.



- A stack-based runtime environment is **inadequate**, and a more general form of environment is required, called **Fully Dynamic Environment**
- In fully dynamic environment, the basic structure of **activation record remains the same**:
 - The space must be allocated for parameters and local variables, and there is still a need for control and access links
- **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

A series of horizontal lines in white and light blue colors, of varying lengths, extending from the left edge of the slide towards the right, positioned below the title.

- OO language requires **special mechanisms** in the runtime environment to implement their added features:
 - Objects, methods, inheritance, and dynamic binding
- OO languages **vary greatly** in their requirements for the runtime environment.
- Two good representatives of the extreme:
 - Smalltalk requires fully dynamic environment;
 - C++ retains the stack-based environment of C in much of its design efforts.



- Object in memory can be viewed as a *cross between a traditional record structure and an activation record*
- *One mechanism* would be for initialization code to copy all the currently inherited features (and methods) directly into the record structure (with methods as code pointer);
- *An alternative* is to keep a complete description of the class structure in memory at each point during execution, with inheritance maintained by superclass pointers, and all method pointers kept as field in the class structure (Inheritance graph).
- *Another alternative* is to compute the list of code pointers for available methods of each class, and store this in (static) memory as a virtual function table. It is the method of choice in C++.



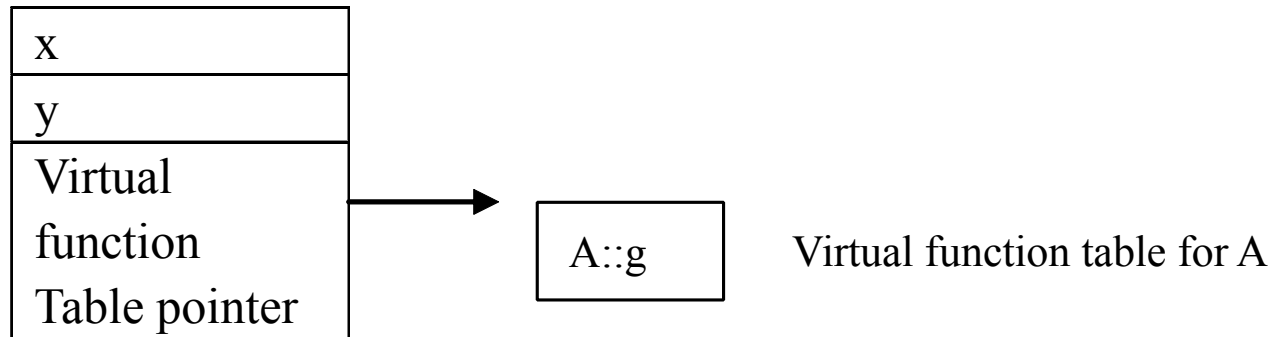
Example: Consider the following C++ class declarations:

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

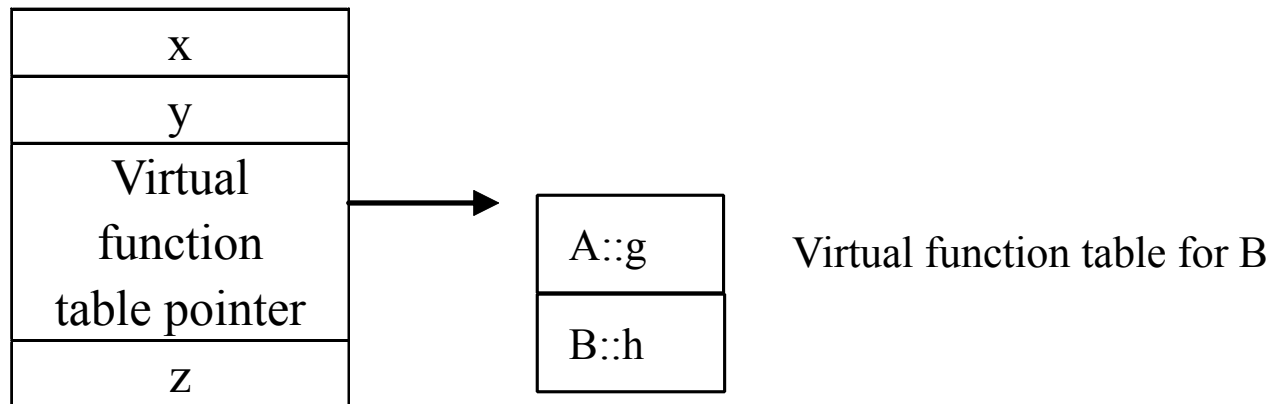
```
class B : public A
{
    public:
    double z;
    void f();
    virtual void h();
}
```



An object of class A would appear in memory as follows:



While the object of class B would appear as follows:



Note: The virtual function pointer, once added to the object structure, remains in a fixed location. The **f does not obey dynamic binding in C++, since it is not declared “virtual”**.

7.4.3 Heap Management

A series of horizontal lines in shades of red and white, located at the bottom of the slide. The lines vary in length and thickness, creating a decorative border.

- In most language, even a stack-based environment needs some dynamic capabilities in order to handle **pointer allocation and de-allocation**
- HEAP, the data structure to handle such allocation.
- Allocated as a linear block of memory, which provides two operations:
 - (1) **Allocate**, takes a size parameter and returns a pointer to a block of memory of the correct size, or a null pointer if none exists.
 - (2) **Free**, takes a pointer to an allocated block of memory and marks it as being free again



- Two operations exist under different names in many languages:
 New and **dispose** in Pascal;
 New and **delete** in C++;
 Malloc and **free** in the C language.
- 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**.
- The 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.



- A different implementation of malloc and free, using a circular linked list data structure that keep track of both allocated and free block.
- The code is given in the following figures:

```
#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 men[MEMSIZE];
static header *memptr=NULL;
```



```
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 = 1;
      memptr->s.freesize = MEMSIZE-1;
    }

    for(p=memptr;
        (p->s.next!=memptr) && (p->s.freesize < nunits);
        p=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);
}
```

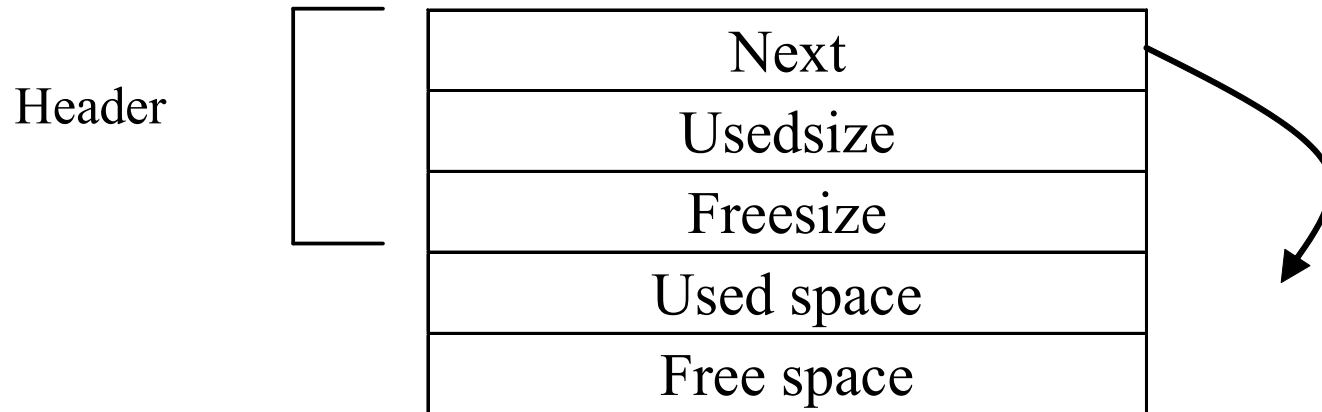


```
void free(void *ap)
{
    header *bp, *p, *prev;
    bp = (header *)ap - 1;
    for (prev=memptr, p=memptr->s.next;
        (p!=bp) && (p!=memptr); prev=p, p=p->s.next);
        if (p!=bp) return;
        /* corrupted list, do nothing */
        prev->s.freesize +=p->s.usedsize + p->s.freesize;
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



Note:

- (1) Uses a statically allocated array of size MEMSIZE as the heap
- (2) Define a data type HEADER to hold each memory block's bookkeeping information



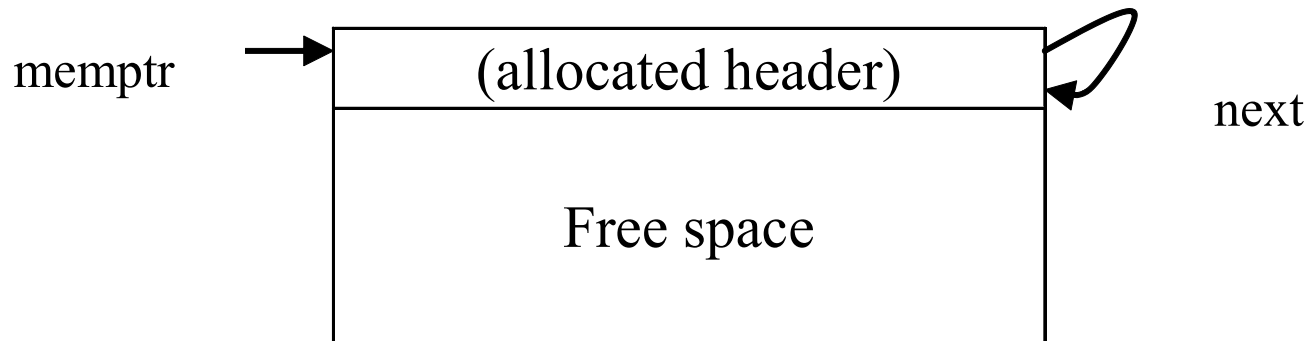
Next: a pointer to the next block in the list;

Usedsize: the size of the currently allocated space;

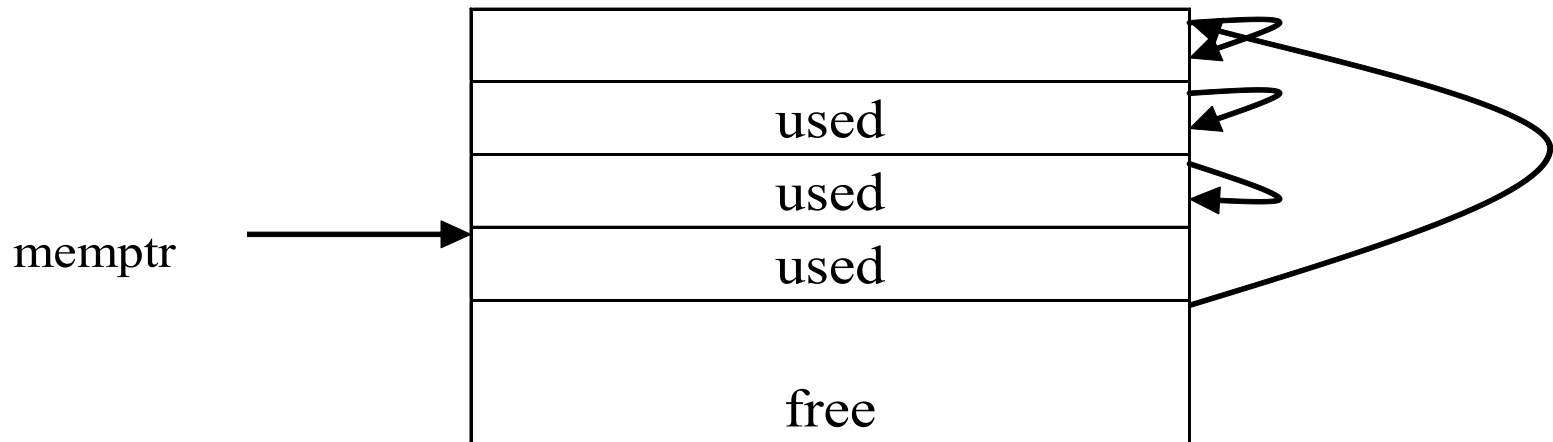
Freesize: the size of any following free space.

Use a union declaration and an align data type to align the memory elements on a reasonable byte boundary.

(3) Memptr : a pointer to a block that has some free space, which is initialized to NULL. On the first call to malloc, memptr is set to the beginning of the heap array.

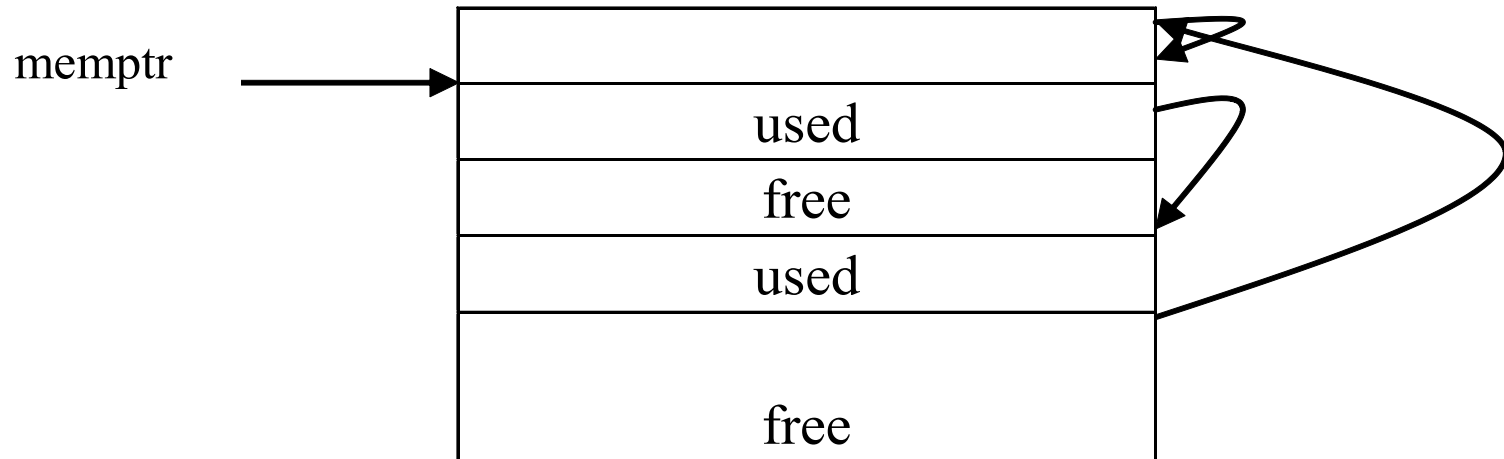


- (4) The initial header allocated on the first call to malloc will never be freed.
- (5) “first fit” algorithm is adopted to allocate the new block.
After three calls to malloc, the list will look as follows:



(6) Consider the free procedure; the memptr will be set to point to the block containing the memory just freed.

After the middle block of the three used blocks was freed, the list would look as follows:



7.4.4 Automatic Management of the heap

- The use of malloc and free to perform dynamic allocation and de-allocation of pointer is called **manual method**.
- By contrast, the runtime stack is managed automatically by the calling sequence.
- **Garbage collection**: the process of reclamation of allocated but no longer used storage without an explicit call to free.
- The standard technique is to **perform mark and sweep garbage collection**.
- In this method no memory is freed until a call to malloc fails, which does this in two passes.
 - Follows all pointers recursively, **starting with all currently accessible pointer values and marks each block of storage reached**; an extra bit for the marking.
 - Sweeps linearly through memory, **returning unmarked blocks to free memory**.
 - Also perform memory compaction to leave only one large block of **contiguous free space** at the other end.



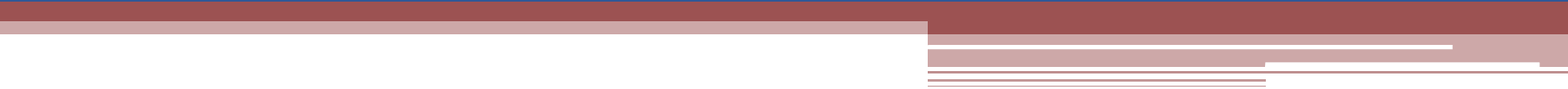
- The **Mark and Sweep garbage collection** has several **drawbacks**:
 - Requires extra storage;
 - The double pass through memory cause a significant delay in processing
- A bookkeeping improvement can be made to this process by splitting available memory into two halves and allocating storage only from one half at a time.
- **Called stop-and-copy or two-space garbage collection.**
 - All reached blocks are immediately copied to the second half of storage not in use;
 - No extra mark bit is required and only one pass is required;
 - It also performs compaction automatically;
 - It does little to improve processing delays during storage reclamation.



- **Generational garbage collection**, which adds a permanent storage area to the reclamation scheme of the previous paragraph
- Allocated objects that **survive long enough** are simply copied into permanent space and are never deallocated during subsequent storage reclamations.
- Search only a **very small section of memory for newer storage allocations**, and the time for such a search is reduced to a fraction of a second.



7.5 Parameter Passing Mechanisms

A decorative graphic consisting of several horizontal lines of varying lengths and colors (dark red, light red, and white) extending from the left edge of the slide towards the right, positioned below the title.

- In a procedure call
 - parameters correspond to location in the activation record which will be **filled by the caller with the arguments** before jumping to the code of the called procedure
- The process of building these values
 - The binding of parameters to the arguments
- How the argument values are interpreted by the procedure code **depends on the particular parameter passing mechanisms adopted by the source language**
 - Fortran77 binds parameter to locations rather than values;
 - C views all arguments as values;
 - C++, Pascal and Ada, offer a choice of parameter passing mechanisms.



- The common parameter passing mechanisms:
 - Pass by value
 - Pass by reference
 - Pass by value-result
 - Pass by name (also called delayed evaluation)
- Other issues not addressed by the parameter passing mechanism itself:
 - **The order** in which arguments are evaluated
 - Many languages permit arguments to calls that cause **side effects**
- Example: `f(++x, x);`
 - C compilers evaluate their arguments from right to left



Summary of memory allocation

	fully static	stack-based	dynamic
storing space	Static data area	Stack area	Heap area
suitable for	external variable、static variable、	local variable、parameters	dynamic variable
allocating time	before running the program	before calling the procedure	depends on user
release time	after the program	after the procedure	depends on user
address calculation time	during compiling	during execution	during execution



7.5.1 Pass by Value

- The arguments are expressions that are **evaluated at the time of the call**, and their values become 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

- The following inc2 function written in C does not achieve its desired effect:

```
Void inc2( int x)
/* incorrect ! */
{ ++x; ++x;}
```



- In order to achieve non-local changes, In C, this takes the form of **passing the address** instead of the value:

```
Void inc2(int *x)
/* now ok */
{++(*x);++(*x);}
```

- Pass by value require no special effort on the part of the compiler
- Easily implemented by taking the most straightforward view of argument computation and activation record construction



7.5.2 Pass by Reference

A decorative graphic consisting of several horizontal lines of varying lengths and colors (dark red, light red, and white) extending from the left edge of the slide towards the right, positioned below the title.

- Pass-by-reference passes the location of the variable, so that 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:

```
Void inc2( int & x)
/* C++ reference parameter */
{++x;++x}
```

- Pass by reference **requires the compilers to compute the address of the argument** to store in the local activation record.
- The compile must also **turn local accesses** to the reference parameter **into indirect access modes**.



- For the call like $p(2+3)$ in the FORTRAN77, a compile must **invent an address for the expression $2+3$** , compute the value into this address, and then pass the address to the call
- The features of pass by reference:
 - Not require a copy to be made of the passed value, which is significant for a large structure.
 - Pass an argument by reference and prohibit changes to be made to the argument's value. Such an option is provide by C++
- Void f (const muchdata & X)
 - Where, muchdata is a data type with a large structure, the compile will perform a static check the X never appears on the left of an assignment.



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
 - This is the mechanism of Ada in out parameter
- Pass by value-result is only distinguishable from pass by reference in the presence of aliasing.
- For instance, in the following code(in C syntax),

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

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



- 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.
- Pass by value-result **needs to modify the basic structure of the runtime stack and the calling sequence:**
 - The activation record cannot be freed by the callee
 - The caller must either push the address of arguments onto the stack before constructing the new activation record, or recomputed these address on return from the called procedure.



7.5.4 Pass by Name

- This is **the most complex of the parameter passing** mechanisms, which is also called delayed evaluation.
- **Idea: the argument is not evaluated until its actual use in the called program**
- As an example, in the code:
 Void p (int x)
 { ++x; }
- The effect of p(a[i]) is of evaluating
 ++(a[i])
- Thus if i were to change before the use of x inside P, the result will be different from either pass by reference or pass by value-result.



- For instance, in the code (in C syntax)

```
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.



- The interpretation of pass by name is as follows:
 - The text of an argument at point of call is viewed as a function in its own right.
 - The arguments are evaluated every time the parameter name is reached in the procedure.
 - **The arguments are evaluated in the caller's environment**, while the procedure will be executed in its defined environment
- Pass by name mechanism was used in the language Algol60, which became unpopular for several reasons:
 - Gives surprising and counterintuitive results in the presence of side effects;
 - Difficult to implement;
 - Inefficient.
- A variation on this mechanism called **lazy evaluation** has recently become popular in purely functional languages



End of Part Two

THANKS