

Procedures and Run-Time Storage

Compiler Design

CSE 504

- 1 Parameterless Procedures
- 2 Parameter Passing
- 3 Storage Organization

Last modified: Mon Mar 21 2016 at 12:11:19 EDT
Version: 1.3 20:58:09 2012/04/09
Compiled at 12:32 on 2016/03/21

Procedures

- Procedures are abstractions of statements.

Procedures

- Procedures are abstractions of statements.
 - The effect of a procedure call is “*as though the statements in its body were executed*”

Procedures

- Procedures are abstractions of statements.
 - The effect of a procedure call is “*as though the statements in its body were executed*”

```
int x;  
int y;  
void p() {  
    y = x+1;  
}  
x = 10;  
p();
```

Procedures

- Procedures are abstractions of statements.
 - The effect of a procedure call is “*as though the statements in its body were executed*”

```
int x;  
int y;  
void p() {  
    y = x+1;  
}  
x = 10;  
p();
```

$x=10, y=11$

Procedures

- Procedures are abstractions of statements.
 - The effect of a procedure call is “*as though the statements in its body were executed*”

```
int x;
int y;
void p() {
    y = x+1;
}
x = 10;
p();
```

$x=10, y=11$

```
int x;
int y;
void p() {
    y = x+1;
}
x = 10;
y = x+1;
```

Procedures

- Procedures are abstractions of statements.
 - The effect of a procedure call is “*as though the statements in its body were executed*”

```
int x;
int y;
void p() {
    y = x+1;
}
x = 10;
p();
```

$x=10, y=11$

```
int x;
int y;
void p() {
    y = x+1;
}
x = 10;
y = x+1;
```

$x=10, y=11$

Local Variables in Procedures

```
int x;  
int y;  
void p() {  
    int x;  
    x = 20;  
    y = x+1;  
}  
x = 10;  
p();
```

x=10, y=21

Local Variables in Procedures

```
int x;  
int y;  
void p() {  
    int x;  
    x = 20;  
    y = x+1;  
}  
x = 10;  
p();
```

x=10, y=21

```
int x;  
int y;  
void p() {  
    int x;  
    x = 20;  
    y = x+1;  
}  
x = 10;
```

Local Variables in Procedures

```

int x;
int y;
void p() {
    int x;
    x = 20;
    y = x+1;
}
x = 10;
p();

```

x=10, y=21

```

int x;
int y;
void p() {
    int x;
    x = 20;
    y = x+1;
}
x = 10;
{
    int x;
    x=20;
    y=x+1;
}

```

Local Variables in Procedures

```

int x;
int y;
void p() {
    int x;
    x = 20;
    y = x+1;
}
x = 10;
p();

```

x=10, y=21

```

int x;
int y;
void p() {
    int x;
    x = 20;
    y = x+1;
}
x = 10;
{
    int x;
    x=20;
    y=x+1;
}

```

x=10, y=21

The Meaning of Nonlocal Variables

```
int x;  
int y;  
void p() {  
    y = x+1;  
}  
void q() {  
    int y;  
    y = 0;  
    call p();  
    x = y+1;  
}  
x = 10;  
y = 20;  
q();
```

x=1, y=11

The Meaning of Nonlocal Variables

```
int x;  
int y;  
void p() {  
    y = x+1;  
}  
void q() {  
    int y;  
    y = 0;  
    call p();  
    x = y+1;  
}  
x = 10;  
y = 20;  
q();
```

x=1, y=11

```
int x;  
int y;  
void p() ...  
void q() ...  
x = 10;  
y = 20;
```

The Meaning of Nonlocal Variables

```

int x;
int y;
void p() {
    y = x+1;
}
void q() {
    int y;
    y = 0;
    call p();
    x = y+1;
}
x = 10;
y = 20;
q();

```

$x=1, y=11$

```

int x;
int y;
void p() ...
void q() ...
x = 10;
y = 20;
{
    int y;
    y = 0;
    y = x+1;
    x = y+1;
}

```

The Meaning of Nonlocal Variables

```

int x;
int y;
void p() {
    y = x+1;
}
void q() {
    int y;
    y = 0;
    call p();
    x = y+1;
}
x = 10;
y = 20;
q();

```

x=1, y=11

```

int x;
int y;
void p() ...
void q() ...
x = 10;
y = 20;
{
    int y;
    y = 0;
    y = x+1;
    x = y+1;
}

```

x=12, y=20

The Meaning of Nonlocal Variables

```

int x;
int y;
void p() {
    y = x+1;
}
void q() {
    int y;
    y = 0;
    call p();
    x = y+1;
}
x = 10;
y = 20;
q();

```

x=1, y=11

Static Scope

```

int x;
int y;
void p() ...
void q() ...
x = 10;
y = 20;
{
    int y;
    y = 0;
    y = x+1;
    x = y+1;
}

```

x=12, y=20

Dynamic Scope

Procedures and Scopes

- **Static Scope:** Non-local variables in a procedure refer to the environment of the procedure's *definition*.
Most languages such as C/C++/Java, Pascal/Modula/Ada, Scheme, SML/OCAML, ...
- **Dynamic Scope:** Non-local variables refer to the environment of the procedure's *use*.
Languages such as Lisp, Perl, etc.
Mostly used in *interpreted* languages.

Parameter Passing

- **By value:** Evaluate the actual parameter; set the formal parameter to its value.
- **By reference:** Evaluate the *location* of the actual parameter; set the formal parameter to refer to the same location.
- **By name:** Evaluate the actual parameter only when the formal parameter is *used*.
- **By value-result:** Evaluate the actual parameter; set the formal parameter to its value.
When the procedure is done, set the actual parameter to the formal parameter's value.

Managing the store

- Each procedure invocation creates new locations in store corresponding to

Managing the store

- Each procedure invocation creates new locations in store corresponding to
 - The formal parameters

Managing the store

- Each procedure invocation creates new locations in store corresponding to
 - The formal parameters
 - The local variables

Managing the store

- Each procedure invocation creates new locations in store corresponding to
 - The formal parameters
 - The local variables
- When a procedure returns, these locations are no longer accessible.

Managing the store

- Each procedure invocation creates new locations in store corresponding to
 - The formal parameters
 - The local variables
- When a procedure returns, these locations are no longer accessible.
- The locations of formal parameters and local variables are usually grouped into an *activation record*.

Managing the store

- Each procedure invocation creates new locations in store corresponding to
 - The formal parameters
 - The local variables
- When a procedure returns, these locations are no longer accessible.
- The locations of formal parameters and local variables are usually grouped into an *activation record*.
- In common cases, when a procedure can be invoked from only within the scope of its definition, the activation records can be organized as a *stack*.

Managing the store

- Each procedure invocation creates new locations in store corresponding to
 - The formal parameters
 - The local variables
- When a procedure returns, these locations are no longer accessible.
- The locations of formal parameters and local variables are usually grouped into an *activation record*.
- In common cases, when a procedure can be invoked from only within the scope of its definition, the activation records can be organized as a *stack*.
Global variables may be placed in a separate area (e.g. heap).

Managing the store

- Each procedure invocation creates new locations in store corresponding to
 - The formal parameters
 - The local variables
- When a procedure returns, these locations are no longer accessible.
- The locations of formal parameters and local variables are usually grouped into an *activation record*.
- In common cases, when a procedure can be invoked from only within the scope of its definition, the activation records can be organized as a *stack*.
Global variables may be placed in a separate area (e.g. heap).
- References to nonlocal variables will be locations in activation records of other procedures (e.g. deeper in the stack) or to the global area.

More on storage organization later.

Parameter Passing: By Name

Parameter Passing: By Name

```
int x;  
int y;  
void p(int i) {  
    x = i+2;  
    y = i+1;  
}  
x = 10;  
call p(x);
```

Parameter Passing: By Name

```
int x;  
int y;  
void p(int i) {  
    x = i+2;  
    y = i+1;  
}  
x = 10;  
call p(x);
```

By Value:

x=12, y=11

Parameter Passing: By Name

```
int x;  
int y;  
void p(int i) {  
    x = i+2;  
    y = i+1;  
}  
x = 10;  
call p(x);
```

By Value:

x=12, y=11

By Reference:

x=12, y=13

Parameter Passing: By Name

```
int x;
int y;
void p(int i) {
    x = i+2;
    y = i+1;
}
x = 10;
call p(x);
```

By Value:

$x=12, y=11$

By Reference:

$x=12, y=13$

```
int x;
int y;
void p(int i) {
    x = i+2;
    y = i+1;
}
x = 10;
```

$x = \underline{x}+2;$

$y = \underline{x}+1;$

Parameter Passing: By Name

```
int x;
int y;
void p(int i) {
    x = i+2;
    y = i+1;
}
x = 10;
call p(x);
```

By Value:

$x=12, y=11$

By Reference:

$x=12, y=13$

```
int x;
int y;
void p(int i) {
    x = i+2;
    y = i+1;
}
x = 10;
```

$x = \underline{x}+2;$

$y = \underline{x}+1;$

$x=12, y=13$

Parameter Passing By Name: Local Variables

Parameter Passing By Name: Local Variables

```
int x; int y;  
void p(int i) {  
    int x; x = 20;  
    x = i+2;  
    y = i+x;  
}  
x = 10;
```

Parameter Passing By Name: Local Variables

```
int x; int y;  
void p(int i) {  
    int x; x = 20;  
    x = i+2;  
    y = i+x;  
}  
x = 10;  
call p(x);
```

Parameter Passing By Name: Local Variables

```
int x; int y;  
void p(int i) {  
    int x; x = 20;  
    x = i+2;  
    y = i+x;  
}  
x = 10;  
call p(x);
```

```
{ // replace i in p by x  
  int x; x = 20;  
  x = x + 2;  
  y = x + x;  
}
```

Parameter Passing By Name: Local Variables

```
int x; int y;  
void p(int i) {  
    int x; x = 20;  
    x = i+2;  
    y = i+x;  
}  
x = 10;  
call p(x);
```

```
{ // replace i in p by x  
  int x; x = 20;  
  x = x + 2;  
  y = x + x;  
}
```

x=10, y=44

Parameter Passing By Name: Local Variables

```
int x; int y;  
void p(int i) {  
    int x; x = 20;  
    x = i+2;  
    y = i+x;  
}  
x = 10;  
call p(x);
```

```
{ // replace i in p by x  
  int x; x = 20;  
  x = x + 2;  
  y = x + x;  
}
```

x=10, y=44

```
int x; int y;  
void p(int i) {  
    int z; z = 20;  
    z = i+2;  
    y = i+z;  
}  
x = 10;
```

Parameter Passing By Name: Local Variables

```
int x; int y;  
void p(int i) {  
    int x; x = 20;  
    x = i+2;  
    y = i+x;  
}  
x = 10;  
call p(x);
```

```
{ // replace i in p by x  
  int x; x = 20;  
  x = x + 2;  
  y = x + x;  
}
```

x=10, y=44

```
int x; int y;  
void p(int i) {  
    int z; z = 20;  
    z = i+2;  
    y = i+z;  
}  
x = 10;  
call p(x);
```

Parameter Passing By Name: Local Variables

```
int x; int y;  
void p(int i) {  
    int x; x = 20;  
    x = i+2;  
    y = i+x;  
}  
x = 10;  
call p(x);
```

```
{ // replace i in p by x  
  int x; x = 20;  
  x = x + 2;  
  y = x + x;  
}
```

x=10, y=44

```
int x; int y;  
void p(int i) {  
    int z; z = 20;  
    z = i+2;  
    y = i+z;  
}  
x = 10;  
call p(x);
```

```
{ // replace i in p by x  
  int z; z = 20;  
  z = x + 2;  
  y = x + z;  
}
```

Parameter Passing By Name: Local Variables

```
int x; int y;  
void p(int i) {  
    int x; x = 20;  
    x = i+2;  
    y = i+x;  
}  
x = 10;  
call p(x);
```

```
{ // replace i in p by x  
  int x; x = 20;  
  x = x + 2;  
  y = x + x;  
}
```

x=10, y=44

```
int x; int y;  
void p(int i) {  
    int z; z = 20;  
    z = i+2;  
    y = i+z;  
}  
x = 10;  
call p(x);
```

```
{ // replace i in p by x  
  int z; z = 20;  
  z = x + 2;  
  y = x + z;  
}
```

x=10, y=22

Parameter Passing By Name: The Details

On every call to a procedure:

- 1 Rename all *local* variables of the procedure to fresh names so that there is no conflict between local variables and variables in the actual parameters.
- 2 In the procedure body, replace every occurrence of formal parameters by the *expressions* representing the actual parameters.
- 3 Evaluate the procedure body.

The Power of Parameter Passing By Name

```
int x;  
void p(int i, int j) {  
    if (i==0)  
        x = 0;  
    else  
        x = j;  
}
```

The Power of Parameter Passing By Name

```
int x;  
void p(int i, int j) {  
    if (i==0)  
        x = 0;  
    else  
        x = j;  
}  
call p(0, 10/0);
```

The Power of Parameter Passing By Name

```
int x;  
void p(int i, int j) {  
    if (i==0)  
        x = 0;  
    else  
        x = j;  
}
```

```
call p(0, 10/0);
```

```
if (0 == 0)  
    x = 0;  
else  
    x = 10/0;
```

The Power of Parameter Passing By Name

```
int x;  
void p(int i, int j) {  
    if (i==0)  
        x = 0;  
    else  
        x = j;  
}
```

```
call p(0, 10/0);
```

```
if (0 == 0)  
    x = 0;  
else  
    x = 10/0;
```

$x=0$

Parameter Passing: Summary

- **By-Value, By-Reference:** Each actual parameter is evaluated exactly once.

Parameter Passing: Summary

- **By-Value, By-Reference:** Each actual parameter is evaluated exactly once.
Strict Evaluation: Actual parameters are evaluated whether or not they are needed in the procedure.

Parameter Passing: Summary

- **By-Value, By-Reference:** Each actual parameter is evaluated exactly once.
Strict Evaluation: Actual parameters are evaluated whether or not they are needed in the procedure.
- **By-Name:** Actual parameters may be evaluated zero, one, or more times!

Parameter Passing: Summary

- **By-Value, By-Reference:** Each actual parameter is evaluated exactly once.
Strict Evaluation: Actual parameters are evaluated whether or not they are needed in the procedure.
- **By-Name:** Actual parameters may be evaluated zero, one, or more times!
 - Can be used to define procedures and functions that do short-circuit evaluation.

Parameter Passing: Summary

- **By-Value, By-Reference:** Each actual parameter is evaluated exactly once.
Strict Evaluation: Actual parameters are evaluated whether or not they are needed in the procedure.
- **By-Name:** Actual parameters may be evaluated zero, one, or more times!
 - Can be used to define procedures and functions that do short-circuit evaluation.
 - **Lazy Evaluation:** Actual parameters are evaluated at most once, and only when they are needed in the procedure

Parameter Passing: Summary

- **By-Value, By-Reference:** Each actual parameter is evaluated exactly once.
Strict Evaluation: Actual parameters are evaluated whether or not they are needed in the procedure.
- **By-Name:** Actual parameters may be evaluated zero, one, or more times!
 - Can be used to define procedures and functions that do short-circuit evaluation.
 - **Lazy Evaluation:** Actual parameters are evaluated at most once, and only when they are needed in the procedure
and only to the extent needed by the procedure!

Storage Areas

Abstract machines usually offer the following storage areas:

- **Code:** Instructions for the program. (Usually static: does not change after the program is loaded into memory).
- **Static Area:** Space for variables with global scope (lifetime is same as that of the program).
- **Stack:** Space for variables/data structures that are created at run-time (whose lifetime is **same as** the procedure where they were created).
- **Heap:** Space for data structures that can be dynamically allocated or deallocated (whose lifetime is different from the procedure where they were created).

Nesting of Procedure Calls: Quicksort

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

```
int partition(int m, int n) { //Choose a pivot value "v", and
    // reorder sub-array a[m..n] such that a[m..p-1] are all <
    // a[p] = v, and a[p+1...n] are all >= v;      return p
...}
```

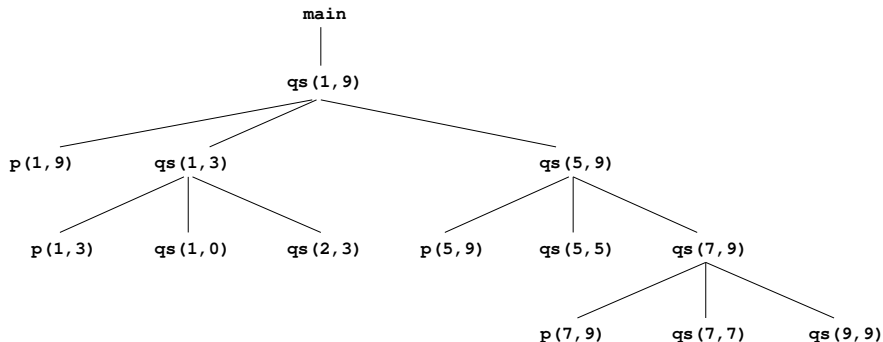
```
void quicksort(int m, int n) {
    int i;
    if (n > m) {
        i = partition(m, n);
        quicksort(m, i-1);
        quicksort(i+1, n);
    }
}
```

Activation Tree

- Nodes of the tree are all procedure calls done in one execution of a program
- Root of the tree is the call to `main`
- q is a descendant of p if a call to p results in a call to q .
- The sequence of procedure calls $\equiv$ pre-order traversal of activation tree.
- The sequence of returns $\equiv$ post-order traversal of activation tree.

Activation Tree: QuickSort

Activation tree for one execution of quicksort:



Activation Stack

Stack of current activations

- If a procedure p has been called, but has not yet returned, then that activation of p is “live”.
- If control is in one activation, say N of a procedure, then all activations on the path from root to N of the activation tree are “live”.
- The information regarding the live activations are kept on a stack, with the most recent call on top of the stack.

Activation Record

Information about a single activation of a procedure:

- Local data (space for local variables and temporaries)
-
- Book-keeping information (next slide)
-
- Return address
 - Return value
 - Saved machine status (volatile registers)
-
- Actual parameters of the procedure

Book-keeping in Activation Records

- **Control Link:** a link to the previous activation record on stack.
Example: in x86, the following code is commonly found at the beginning of every procedure:

```
    pushl %ebp  
    movl  %esp, %ebp
```

The *Base Pointer* `ebp` points to the current activation record. When a call is made, the pointer to old activation record is first saved on stack (forming the control link). Then `ebp` is made to point to the current activation record (which is on top of stack)

- **Access Link:** a link used to access non-local variables.

Space for variables

- Space for formal parameters and local variables of a procedure is allocated in the activation record.
- These variables are accessed using relative addressing (relative to the “base” of the activation record)
- Space for global variables is allocated in *Static Area*.

Nested Procedure Definitions

- In languages like C, procedures are all defined at the “global” level.
- In languages like Pascal and SML, procedure definitions can themselves be nested.

For example:

```
void p(int i) {  
    int x;  
    void q(int j) {  
        int y;  
        ... some expression with x ...  
    }  
    ....  
    call q  
}
```

Access/Static Links

```
void p(int i) {  
    int x;  
    void q(int j) {  
        int y;  
        void r() {  
            int z;  
            ... use of x,y,z ...  
        }  
        ... use of x,y ...  
        call r  
    }  
    ... use of x ...  
    call q  
}
```

- Note that q can access x, which is in p's activation record.
- q's activation record contains a link to p's activation record, called its *access link* or *static link*.
In general, access link is set to the activation record of a procedure's lexical parent.
- This link is used as the base address for accessing non-local variables.
- For r to access x, it will dereference *two* access links.
... since x is in r's grandparent.

Nested *Higher-Order* Procedures

- In languages like C, procedures are all defined at the “global” level, but procedure values can be passed/returned from other procedures via “function pointers”.

There is no need for access links: all access are either local (current activation record) or global (heap).

- In languages like Pascal and Ada, procedure definitions can themselves be nested, but all procedures are first-order: there is no analogue of “function pointers”.

Access links are used for non-local variables.

“Static displays” can be used to ensure constant-time access to non-local variables (instead of following chains of access links).

- In many other languages (e.g. Scheme, ML, Python and Javascript), procedure definitions can themselves be nested, and procedures are higher-order as well.

Stack-based allocation of activation records is not possible any more!

Nested *Higher-Order* Procedures: An Example

```
def p(i):  
    x = 2  
    def q(j):  
        return j+x*i # note non-local access to i and x  
    return q # p is higher-order!
```

```
h = p(3) # h will be a procedure!!
```

```
# Note, "p" is no longer active: the call has returned
```

```
# But any subsequent call to "h" will need "x". E.g.
```

```
print h(5) # this will need x's value!!!
```

- Each activation record will be a **closure** with sufficient information to access all necessary variables.
- Activation records/closures will be kept in heap.