Runtime Environments

CIS*4650 (Winter 2020)

Review

- Front-end analysis: scanning, parsing, and static semantic analysis
  - Language-specific and machine-independent
  - Most of the error-checking captured

- Back-end synthesis: machine-specific and relying on coding solutions
  - Runtime environments: organization of registers and memory for program execution
  - Code generation: convert intermediate code to target machine/assembly code
  - Code optimization: transform code for efficiency in time and space

Three Kinds of Environments

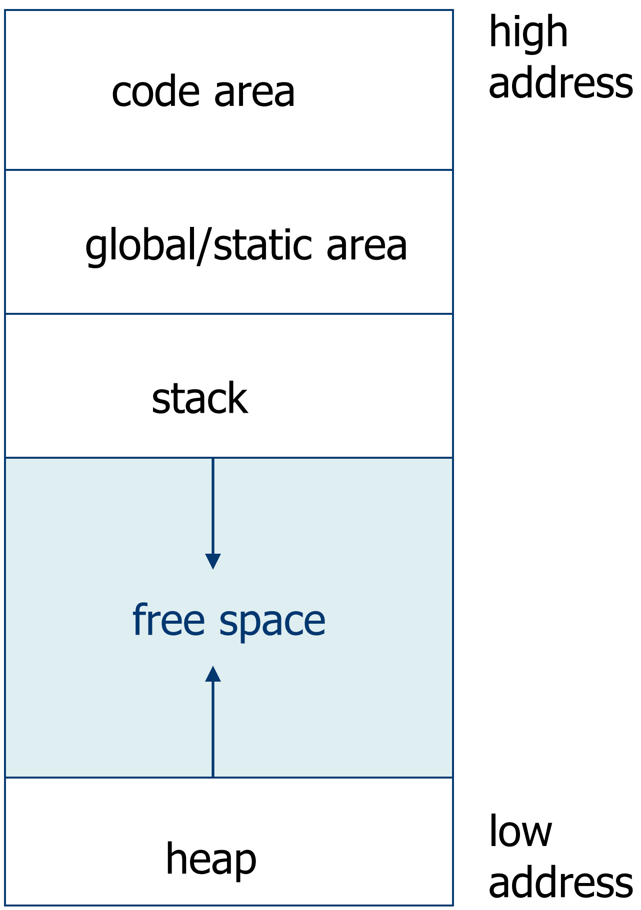
- Fully-static environments:
  - All data are statically allocated in memory before execution
  - e.g., FORTRAN77

- Stack-based environments:
  - Memory for recursive calls can’t be allocated statically, but can be maintained as a stack
  - e.g., C/C++, Pascal, and Ada

- Fully-dynamic environments:
  - Allowing the reference to a local variable in a procedure (usually resulting in a dangling reference in a stack-based environment)
  - e.g., LISP

Memory Organization

- Entries to procedures and addresses to global data can be computed at compile time
- Data are often allocated at the execution time in the form of stack and/or heap
- Stack and heap can compete for the same free space or be given with separate spaces



Global/Static Data Area

- Global data can be allocated statically at compile time
  - In FORTRAN77, all data belong to this class
  - In Pascal, only global variables are in this class
  - In C, external and static variables as well as global variables are in this class
- Small constants such as 0 and 1 are inserted directly into the code
- Large constants, especially strings, are stored in the global area

Activation Records

- Unit of memory allocated to a procedure, which should contain at least the following sections:

Space for arguments (parameters)
Space for bookkeeping info such as return address
Space for local variables
Space for local temporaries

- Activation records are also called stack frames in a stack-based environment

Registers

- Registers are part of the runtime environment and may be used to store temporaries, local variables, and even global data.

- Special-purpose registers:
  - program counter (pc): keep track of the current instruction during execution
  - stack pointer (sp): points to the top (lowest address) of the stack area
  - frame pointer (fp): points to the current activation record
  - argument pointer (ap): points to the argument area of an activation record

Calling Sequence

- Sequence of operations that must occur when a function is called:
  - Call sequence: operations performed during a call such as allocating memory for activation record, computing argument values, and setting the necessary registers.
  - Return sequence: operations performed on return such as placing the return value for the caller, adjusting register values, and possibly releasing memory for the called activation record.
- Caller vs. Callee: how to divide the tasks between the two?

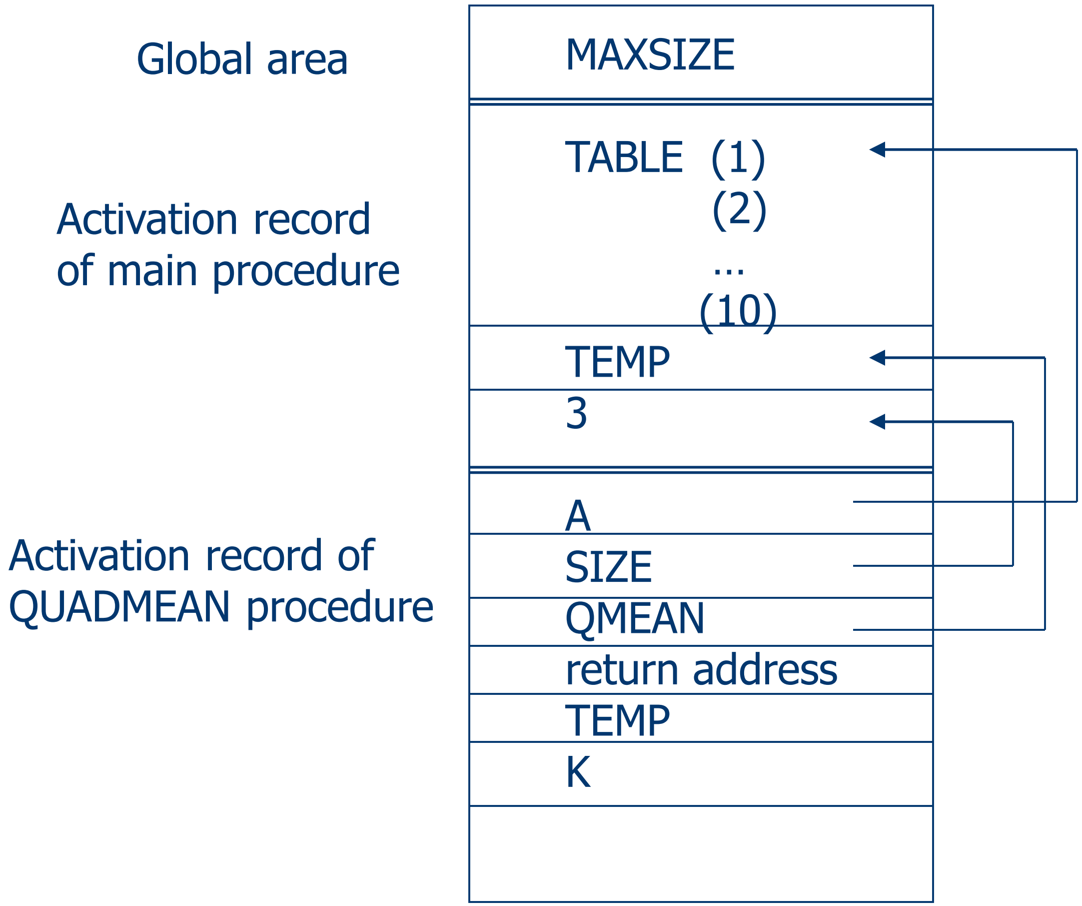
Fully-Static Runtime Environments

- Example FORTRAN77 program:

```
PROGRAM TEST
COMMON MAXSIZE
INTEGER MAXSIZE
REAL TABLE(10), TEMP
MAXSIZE = 10
READ *, TABLE(1), TABLE(2), TABLE(3)
CALL QUADMEAN(TABLE, 3, TEMP)
PRINT *, TEMP
END
```

```
   SUBROUTINE QUADMEAN(A, SIZE, QMEAN)
   COMMON MAXSIZE
   INTEGER MAXSIZE, SIZE
   REAL A(SIZE), QMEAN, TEMP
   INTEGER K
   TEMP = 0.0
   IF ((SIZE.GT.MAXSIZE).OR.(SIZE.LT.1)) GOTO 99
   DO 10 K = 1, SIZE
     TEMP = TEMP + A(K)*A(K)
10 CONTINUE
99 QMEAN = SORT(TEMP/SIZE)
   RETURN
   END
```

Fully-Static Runtime Environments



Fully-Static Runtime Environments

- Each procedure has only a single activation record (no recursive calls allowed)
- All variables, either local or global, can be accessed directly via fixed addresses (no pointers or dynamic allocation)
- Calling sequence:
  - call sequence: compute arguments and store their values to the activation record, save the return address, and jump to new procedure
  - return sequence: jump to the return address in the caller

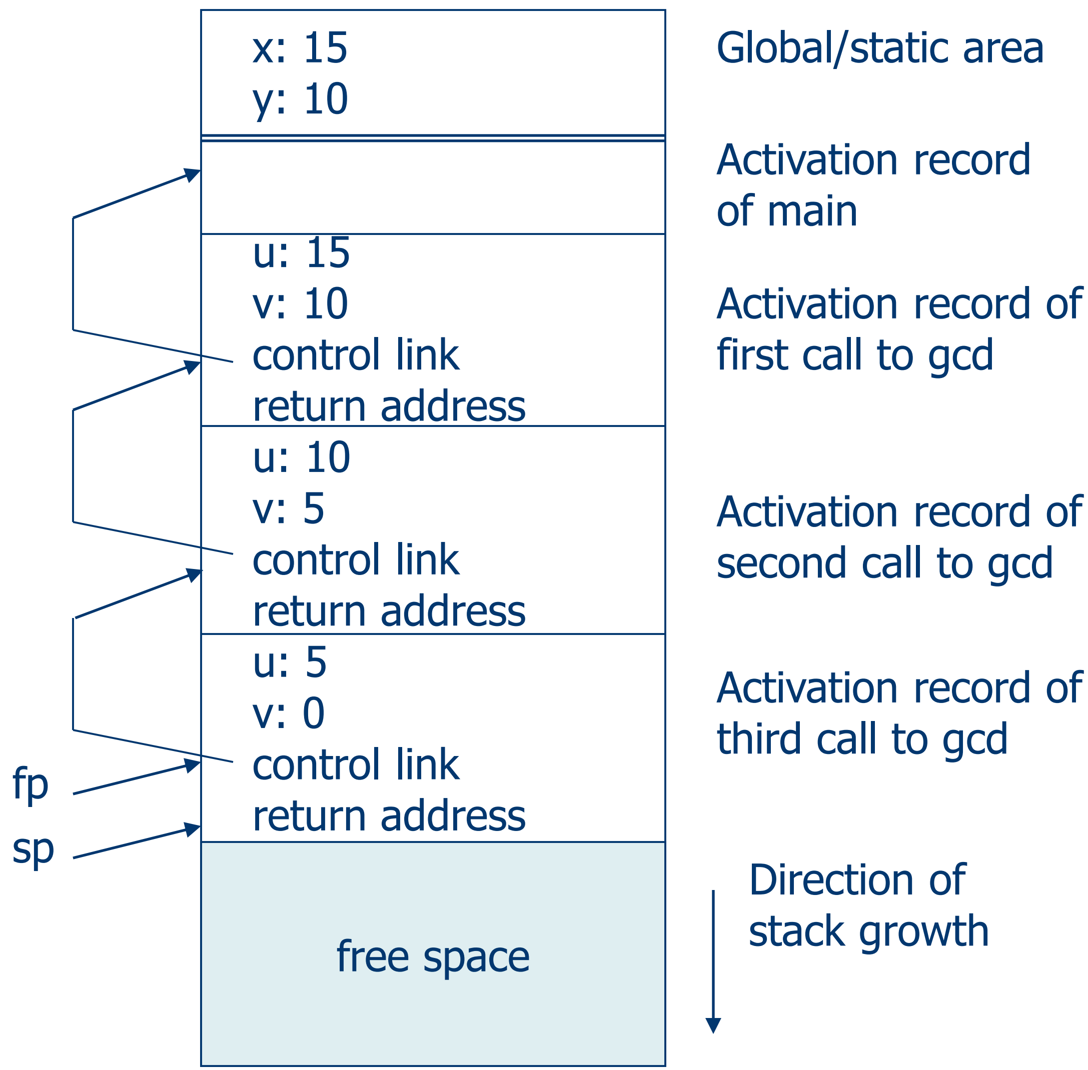
- Parameters are passed by references and arrays don't need to be copied
- Constant arguments such as 3 must be stored so that its location can be passed to the procedure
- Temporaries are placed at the end of the activation record

Stack-Based Runtime Environments

- Activation records for recursive calls can't be allocated statically, but can be maintained as a stack if the caller doesn't reference local variables in the callee

```
#include <stdio.h>

int x, y;

int gcd( int u, int v ) {
  if( v == 0 ) return u;
  else return gcd( v, u%v );
}

int main( ) {
  scanf( "%d%d", &x, &y );
  printf( "%d\n", gcd(x, y) );
  return 0;
}
```

Stack-Based Runtime Environments



Stack-Based Runtime Environments

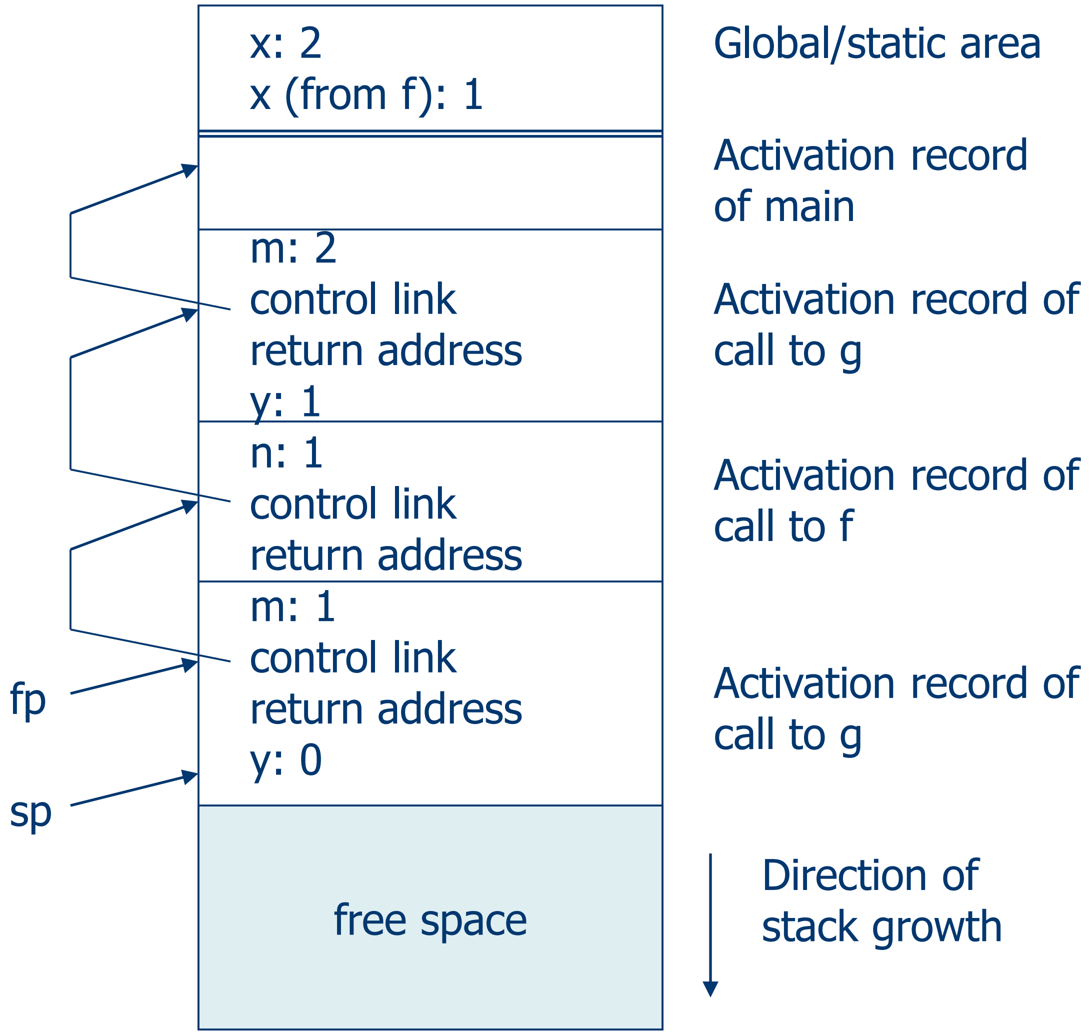
➢ Example with indirect recursion and a static variable:

```
int x = 2;

void g( int );   /* prototype */

void f( int n ) {
   static int x = 1;
   g( n );
   x--;
}
```

```
void g( int m ) {
   int y = m - 1;
   if( y > 0 ) {
      f( y );
      x--;
      g( y );
   }
}

int main( ) {
   g( x );
   return 0;
}
```

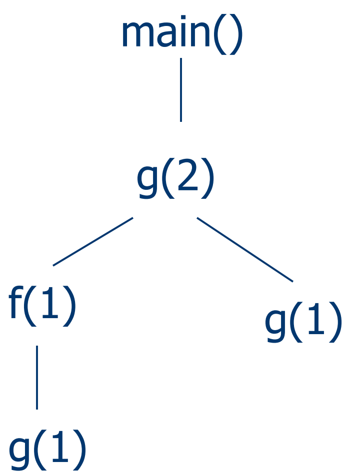
Stack-Based Runtime Environments

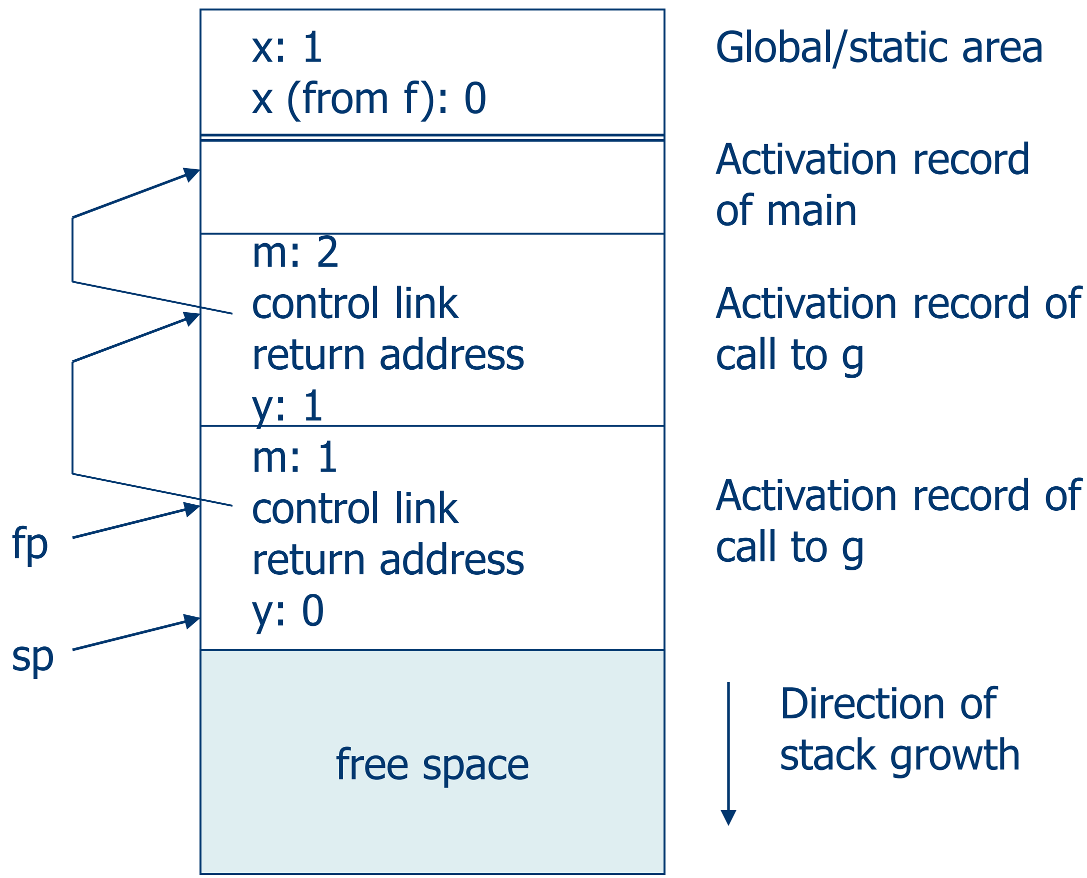
- Stack environment during the second call to g



Stack-Based Runtime Environments

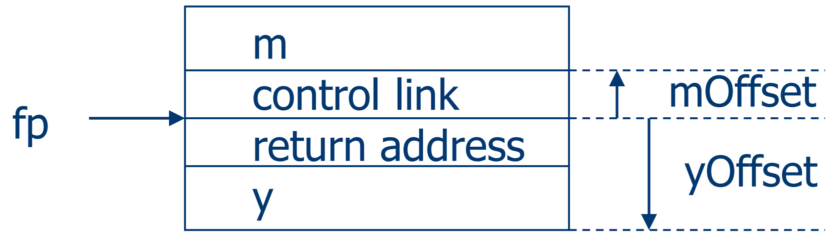
- Stack environment during the third call to g





Access to Names

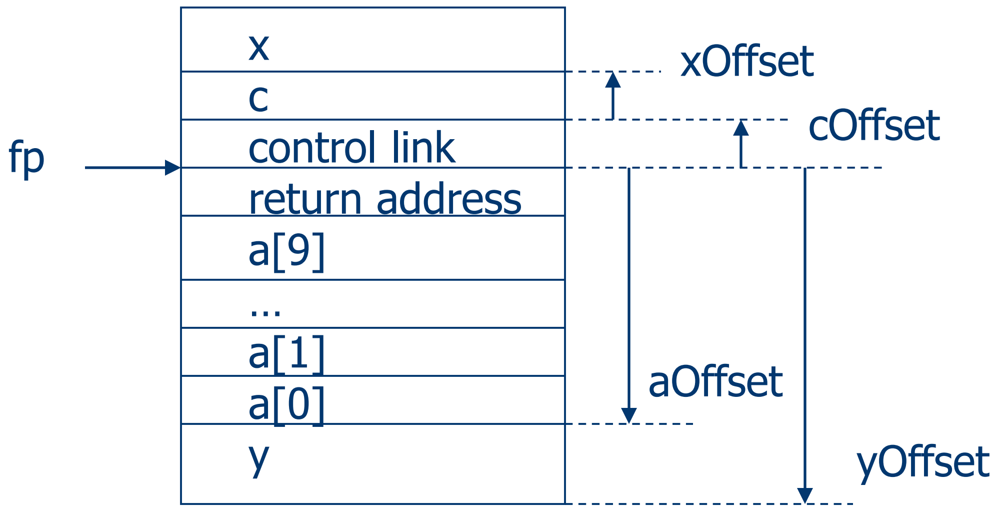
- Dynamically allocated parameters and local variables can't be accessed by fixed addresses
  - Solution: use offsets from the current frame pointer (fp).



  - e.g., assuming that integer requires 2 bytes and address requires 4 bytes, then we have: mOffset = +4(fp) and yOffset = -6(fp).

Access to Names

```
void f( int x, char c ) {
   int a[10];
   double y;
   …
}
```



Name	Offset
x	+5
c	+4
a	-24
y	-32

a[i] offset: (-24+2*i)(fp)

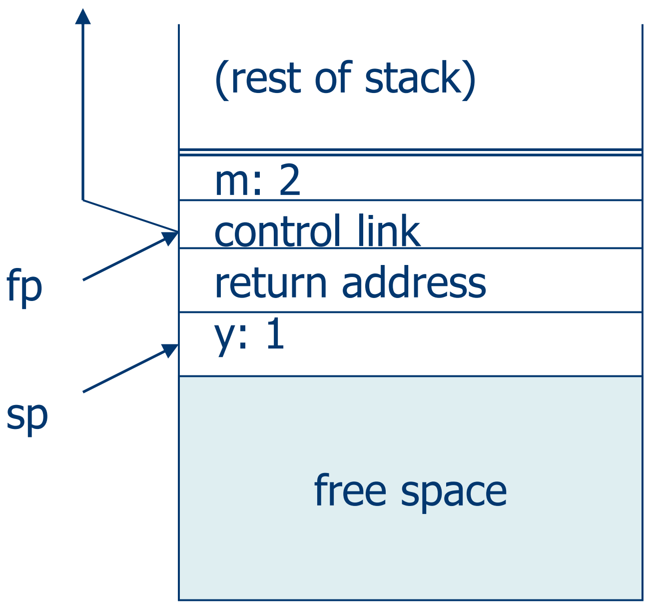
Calling Sequences

- Call Sequence:
  1. Compute arguments and push values to the activation record of callee
  2. Push the current fp as the control link
  3. Copy current sp to fp so that it points to the current activation record
  4. Push the return address to the new activation record
  5. Perform a jump to the code of the callee
  6. Move down sp for all local variables in the callee

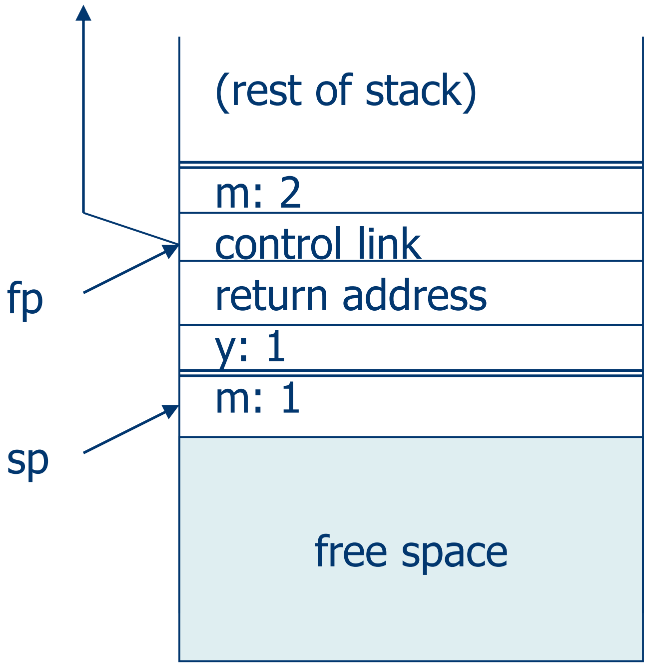
- Return Sequence:
  1. Copy the current fp to sp
  2. Load the control link to fp
  3. Perform a jump to the return address
  4. Change sp to pop the arguments

Calling Sequences

- before the last call to g:

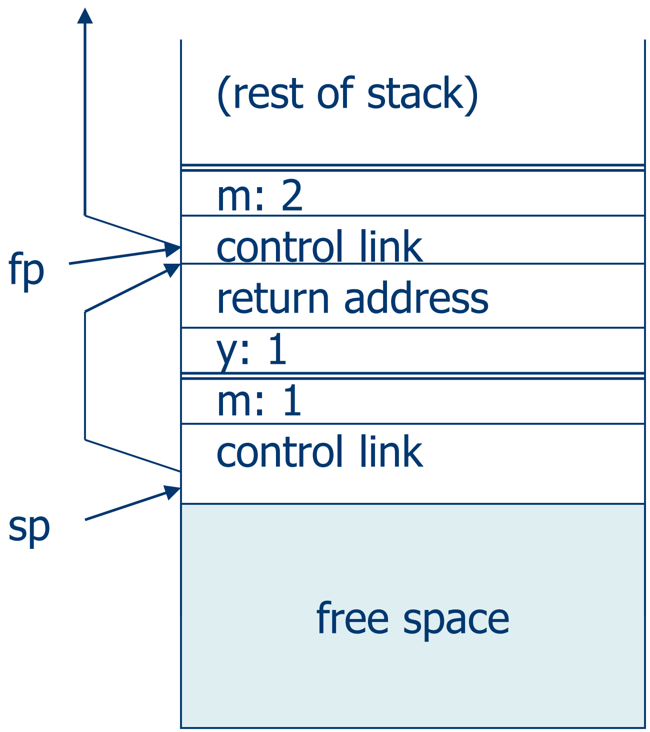


- as new call to g is made:

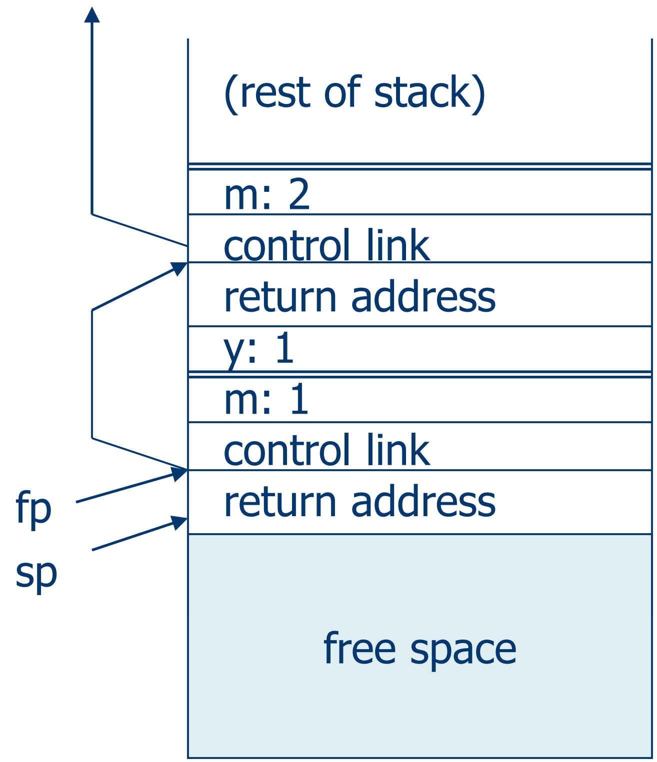


Calling Sequences

- fp is pushed onto stack:

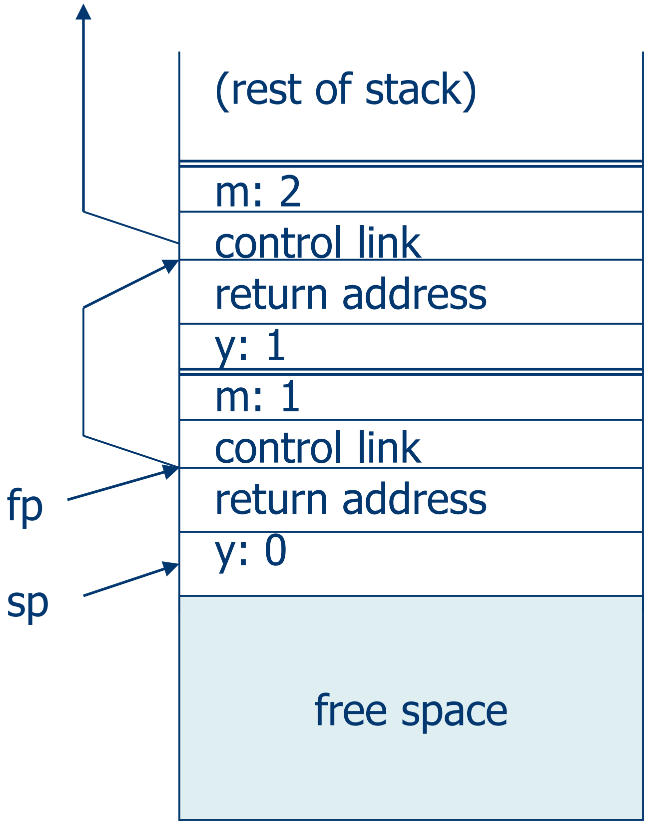


- sp is copied to fp, return address is pushed onto stack, jump to new call to g is made:

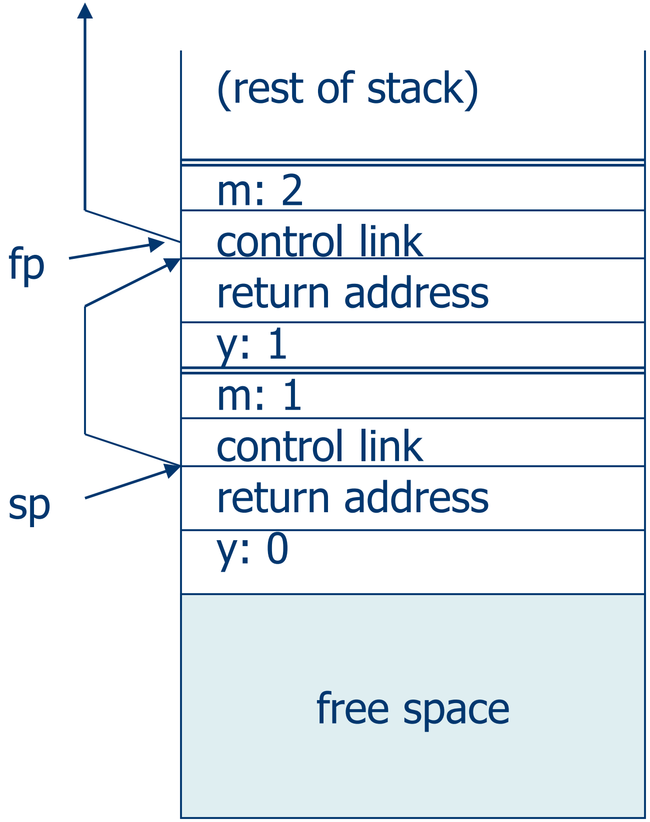


Calling Sequences

- new y is allocated and initialized



- On exit, after copying fp to sp and loading control link to fp

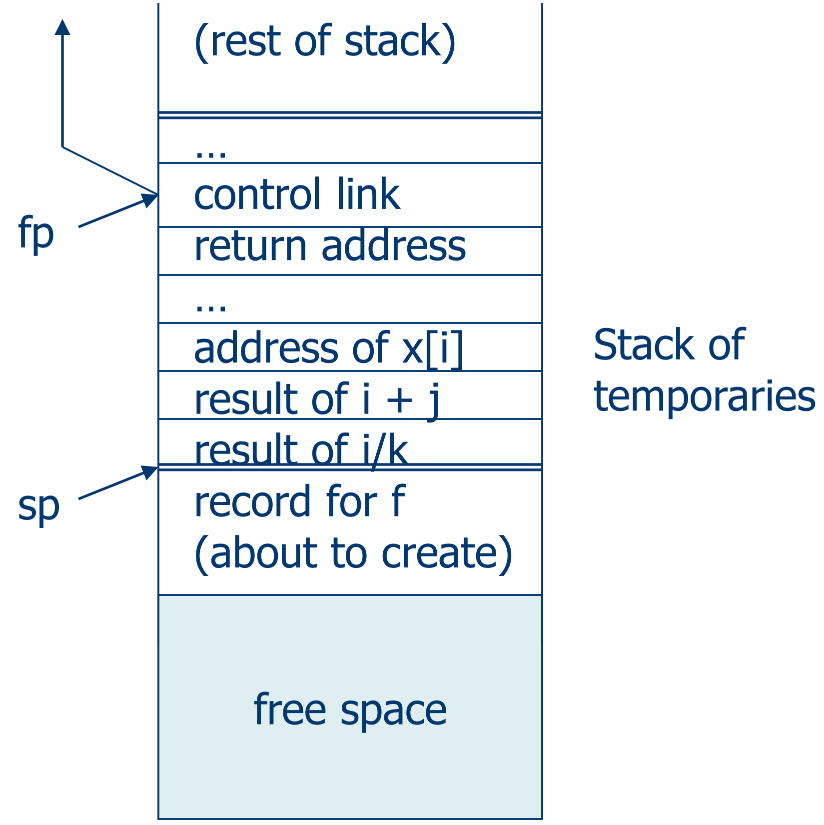


Local Temporaries

- Computing partial results:

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

-The calling sequence using sp works without change.



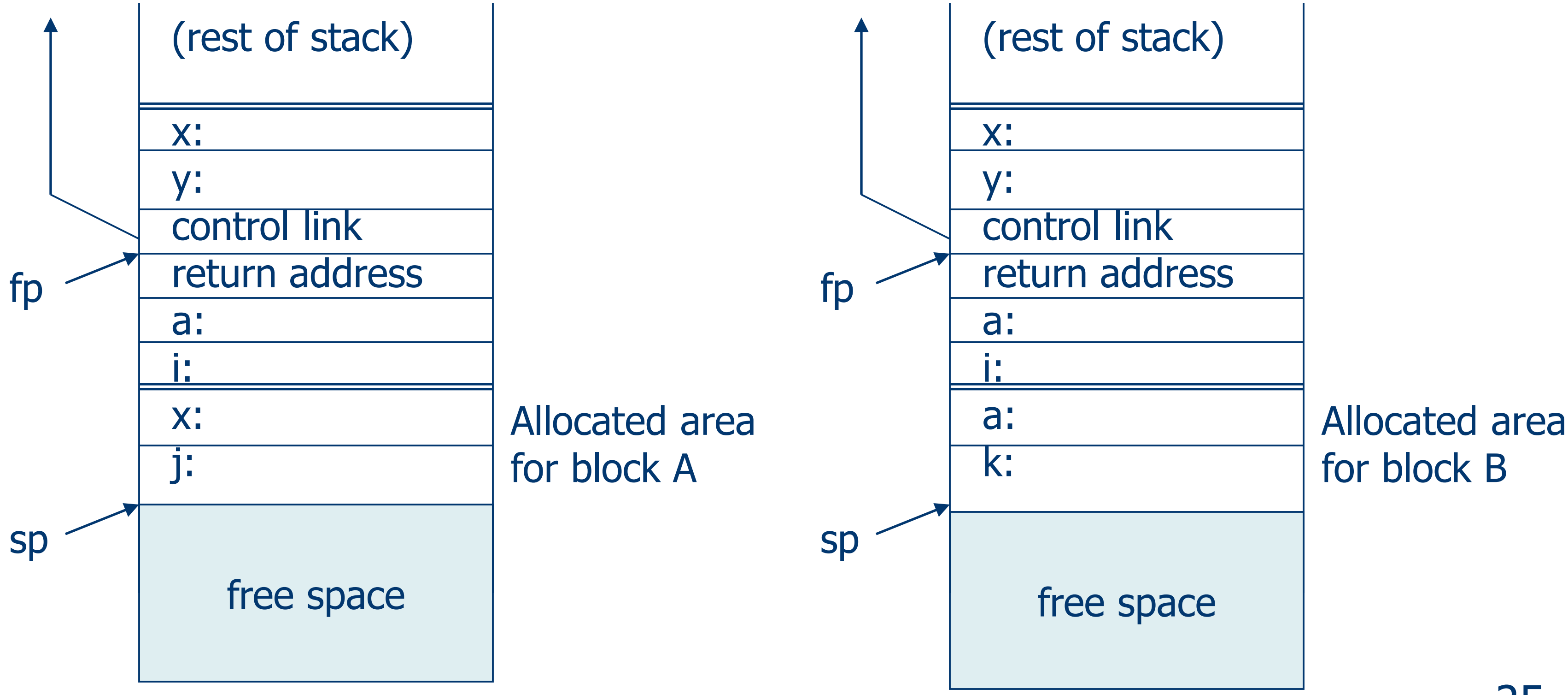
Nested Blocks

- New activation records could be created for each block and discarded on exit
- Why not efficient?
  - No parameters
  - No return address
  - Always executed immediately

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

Nested Blocks

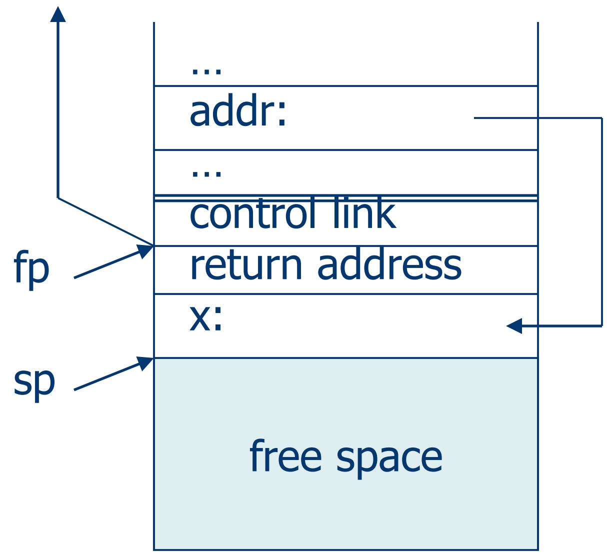
- A simpler solution is to allocate temporaries on entry to a block and de-allocate them on exit



Fully-Dynamic Environments

- Dangling references in a stack-based environment:

```
int * dangle( void ) {
   int x;
   return &x;
}
...
int * addr = dangle();
```



- Fully-dynamic environment will de-allocate activation records only when all references to them have disappeared
  - Finding and de-allocating inaccessible areas of memory during execution is called ***garbage collection***.

Heap Management

- Even a stack-based environment may need dynamic allocations/de-allocations through pointers.

- A heap can grow linearly while interfering as little as possible with the stack.

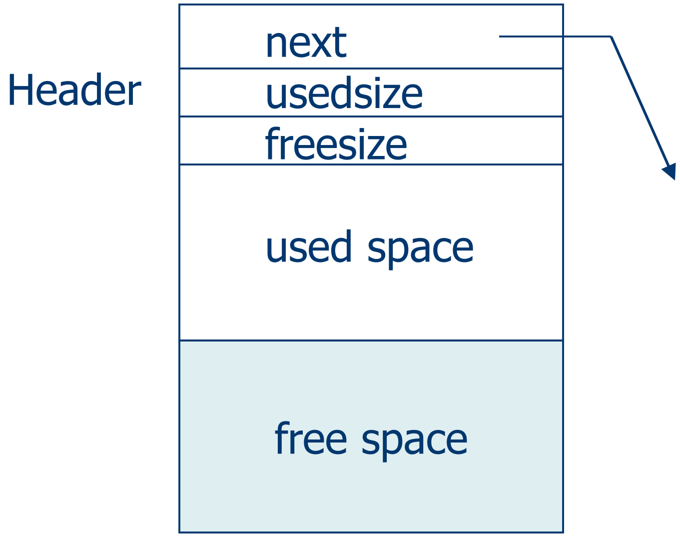
- Users are responsible for allocating and freeing heap spaces.
  - A heap supports two operations (e.g., in C language):

```
void * malloc( unsigned int nbytes );
void free( void * ptr );
```

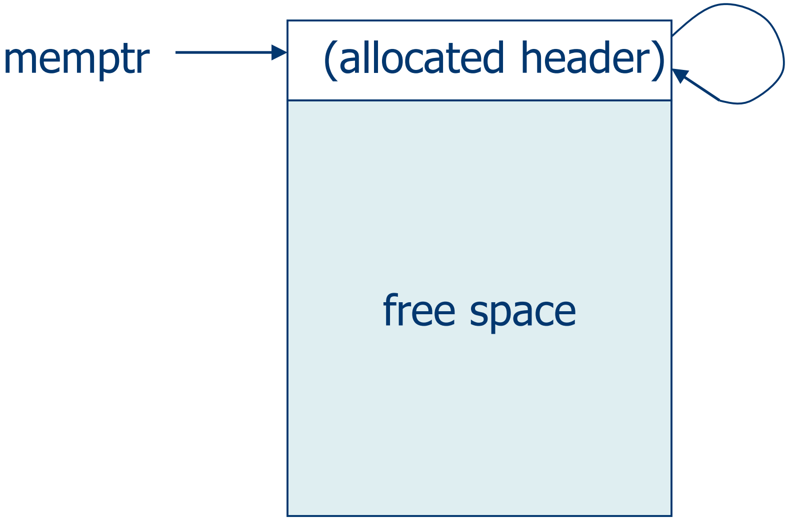
Heap Management

- A circular linked list for available spaces:

- Structure of a memory block

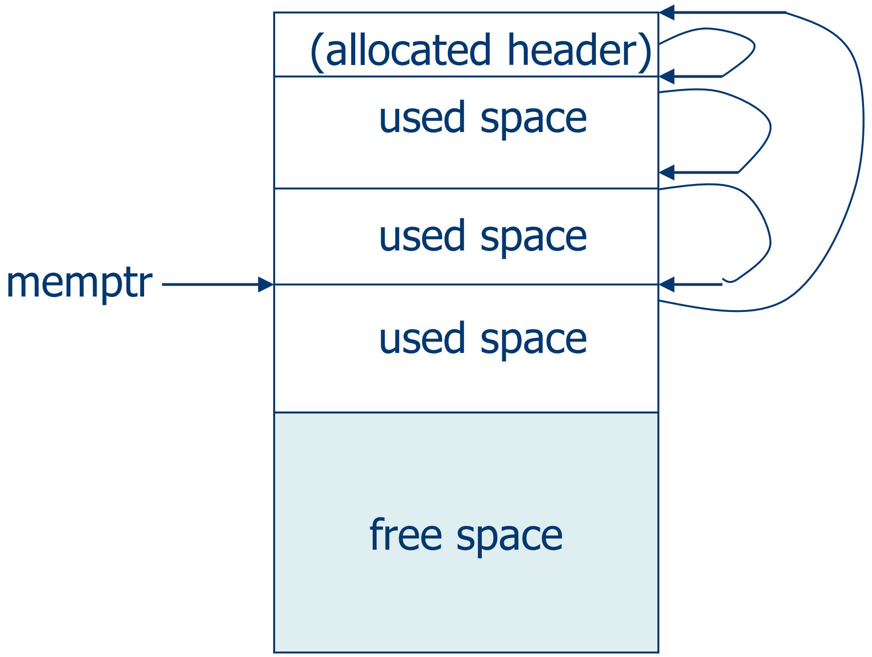


- Initial header (never deleted)

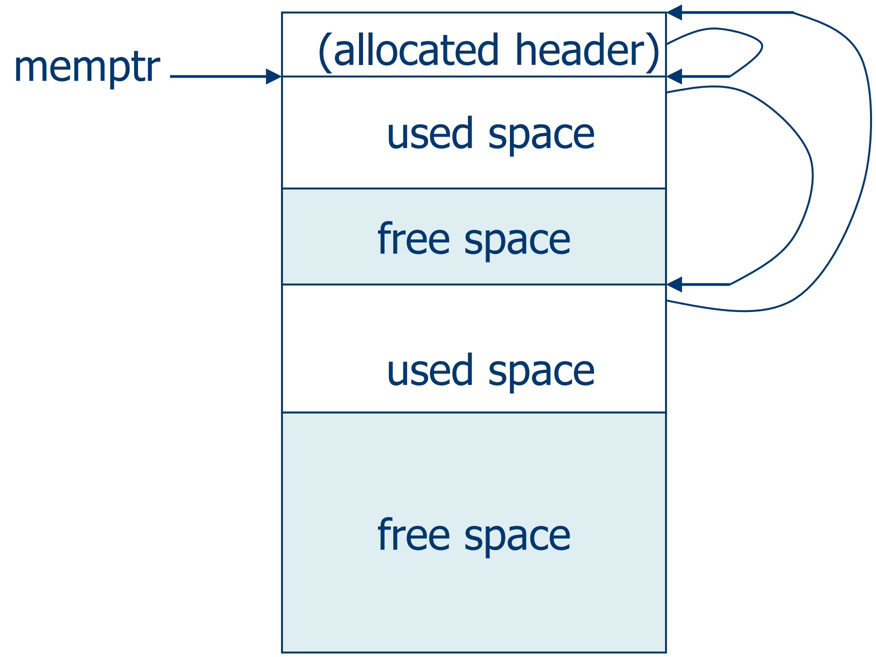


Heap Management

- After three calls to malloc:



- After middle block is freed:



Parameter Passing

- Parameters correspond to locations in the activation record of a procedure call

- Parameter passing: bind parameters to their arguments (different forms such as addresses and values)

- Parameter passing methods:
  - Pass by value (or call by value)
  - Pass by reference (or call by reference)
  - Pass by value-result

Pass By Value

- Arguments are expressions whose values are computed and copied to parameters
  - The only parameter passing mechanism in C

```
/* incorrect. Why? */
void inc2( int x ) {
   ++x;
   ++x;
}


/* correct solution */
void inc2( int * x ) {
   ++(*x);
   ++(*x);
}
```

```
/* will x[] be initialized correctly? */
void init( int x[], int size ) {
   int i = 0;
   for( i = 0; i < size; i++ )
      x[i] = 0;
}
```

Pass By Reference

- Arguments must be variables (at least in principle) so that their addresses can be copied to parameters
  - The only parameter passing mechanism in FORTRAN77
  - Parameters are essentially aliases for their arguments

```
/* C++ solution */
void inc2( int & x ) {
   ++x;
   ++x;
}
```

Pass By Value-Result

- Similar to pass by reference except that no actual alias is established
  - The argument value is copied and used in the procedure and the final value of the parameter is copied back to the argument.

```
/* what is the value of a after p(a, a) if different parameter passing
   methods are used? */
void p( int x, int y ) {
   ++x;
   ++y;
}

int main( ) {
   int a = 1;
   p(a, a);
   return 0;
}
```