Runtime Support

CMPT 379: Compilers

Instructor: Anoop Sarkar

anoopsarkar.github.io/compilers-class

Runtime Support

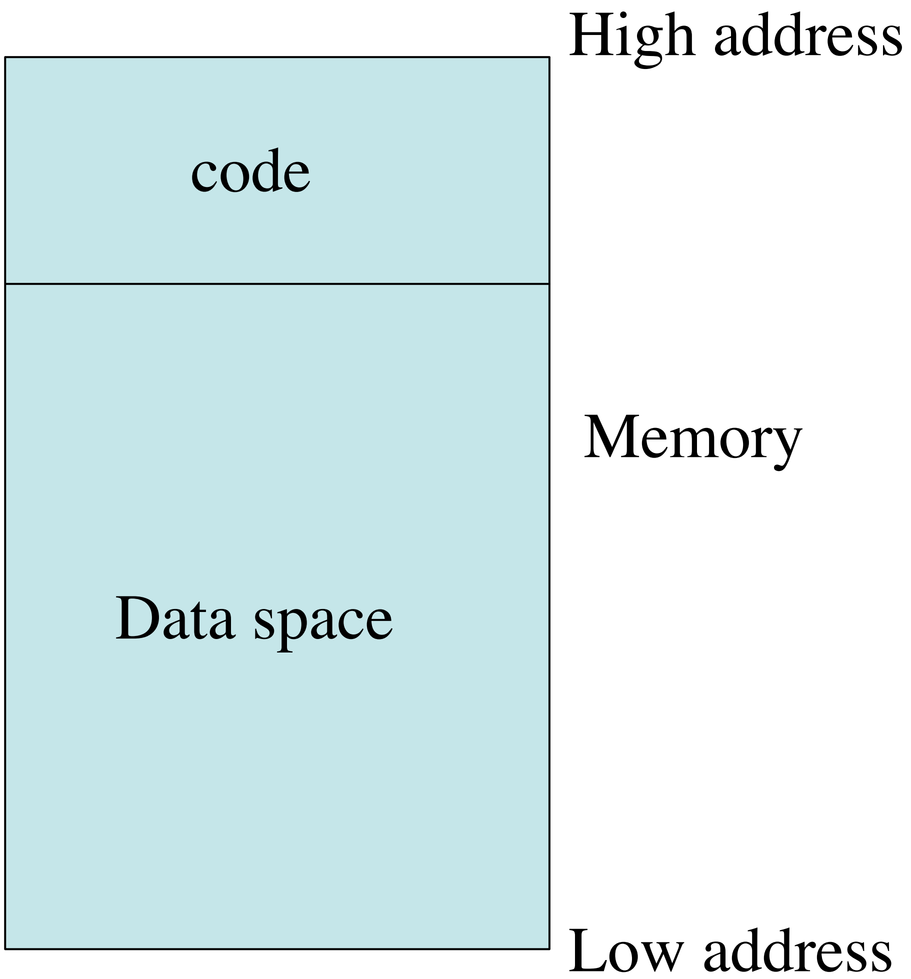
- Management of runtime resources
- Correspondence between:
  - Static (compile-time) structures
  - Dynamic (run-time) structures
- Storage organization
  - Using memory to store data structures of the executing program

Invoke the Program

- Execution of the program is initially under the control of the operating system
- When program is invoked:
  - The OS allocates space for the program
  - The code is loaded into part of the memory
  - The OS jumps to the entry point (i.e., main)

Memory

- Compiler is responsible for:
  - Generating code
  - Orchestrating use of the data area



Procedure Activation

- Two assumptions about programming languages
  - Execution is sequential; control moves from one point in a program to another in a well-defined order
    - Violated by concurrency
  - When a procedure is called, control always returns to the point immediately after the call
    - Violated in: Programming languages with exception

Procedure Activation

- An invocation of procedure P is an *activation of P*
- The lifetime of an activation of P is
  - All the steps to execute P
  - Including all the steps in procedures P calls

Procedure Activation

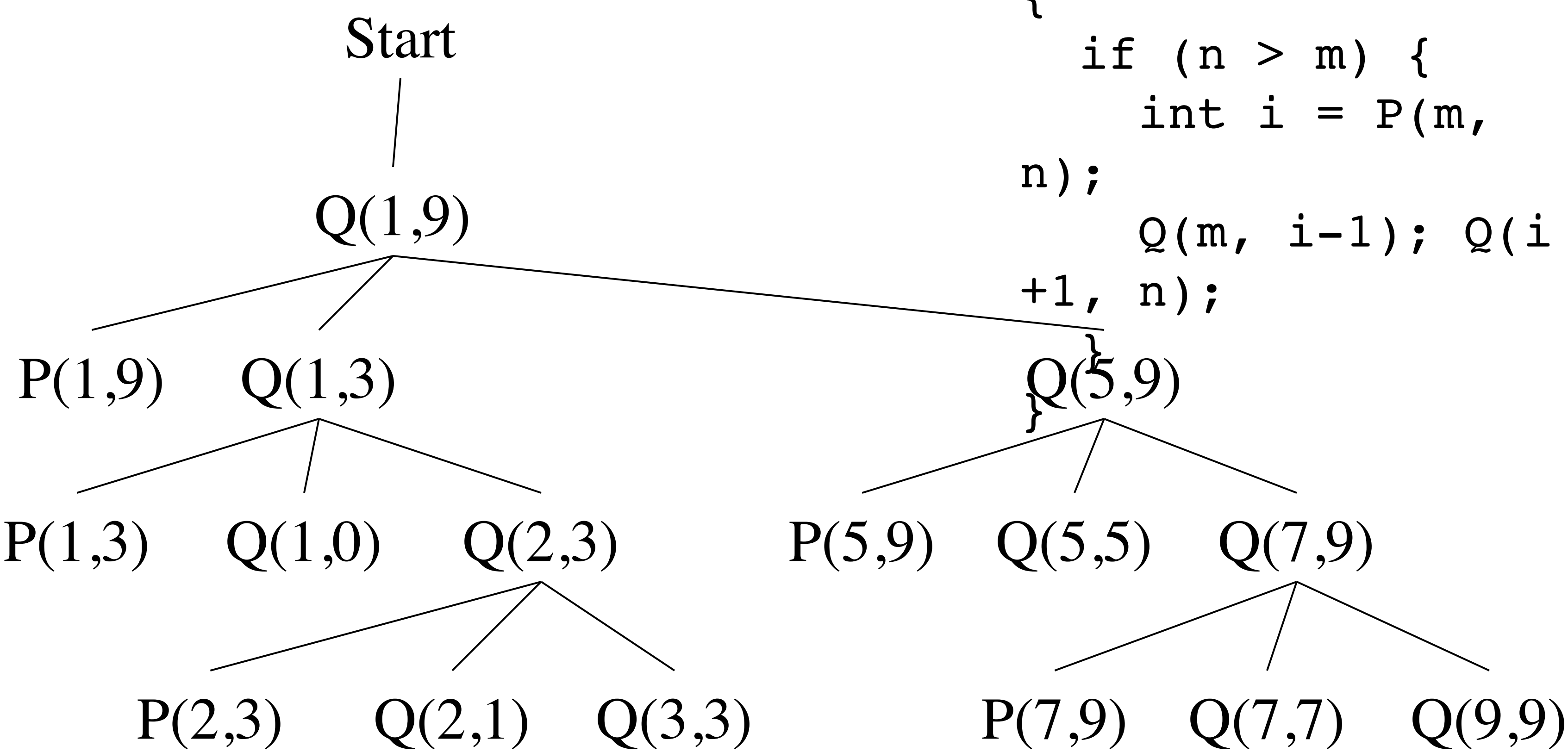
- The lifetime of a variable x is the portion of execution in which x is defined (until x is de-allocated)
- Note that
  - Lifetime is a dynamic (run-time) concept
  - Scope is a static concept

Activation Trees

- Observation
  - When P calls Q, then Q returns before P returns
- Lifetimes of procedure activations are properly nested
- Activation lifetimes (sequence of function calls) can be depicted as a tree: *activation tree*

Activation Tree

```
int Q(int m, int n)
{
  if (n > m) {
    int i = P(m, n);
    Q(m, i-1); Q(i+1, n);
  }
}
```

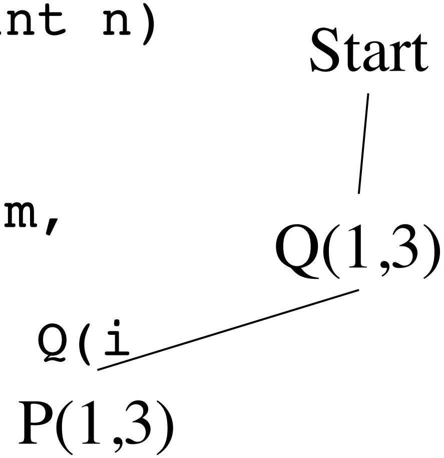


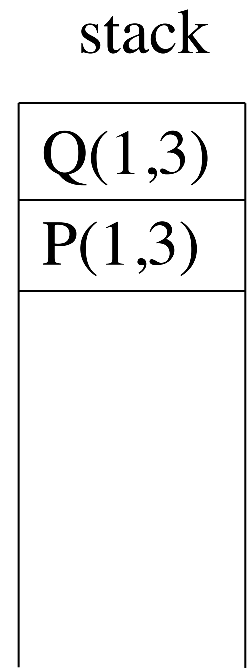
Activation Tree

- The activation tree depends on run-time behavior
- The activation tree may be different for every program input
- Since activations are properly nested, a stack can track currently active procedures

Stack of Active Procedures

```
int Q(int m, int n)
{
  if (n > m) {
    int i = P(m,
n);
    Q(m, i-1); Q(i
+1, n);
  }
}
```

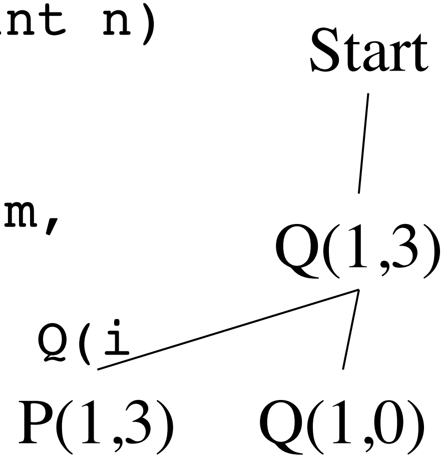


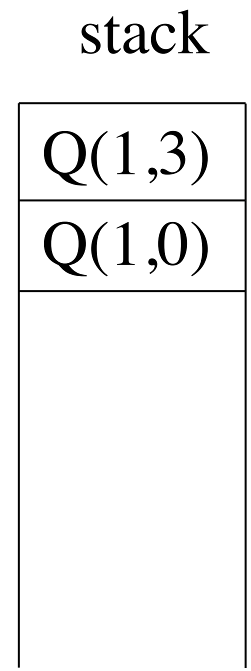


Stack does not keep track of entire activation tree, just active procedures

Stack of Active Procedures

```
int Q(int m, int n)
{
  if (n > m) {
    int i = P(m,
n);
    Q(m, i-1); Q(i
+1, n);
  }
}
```

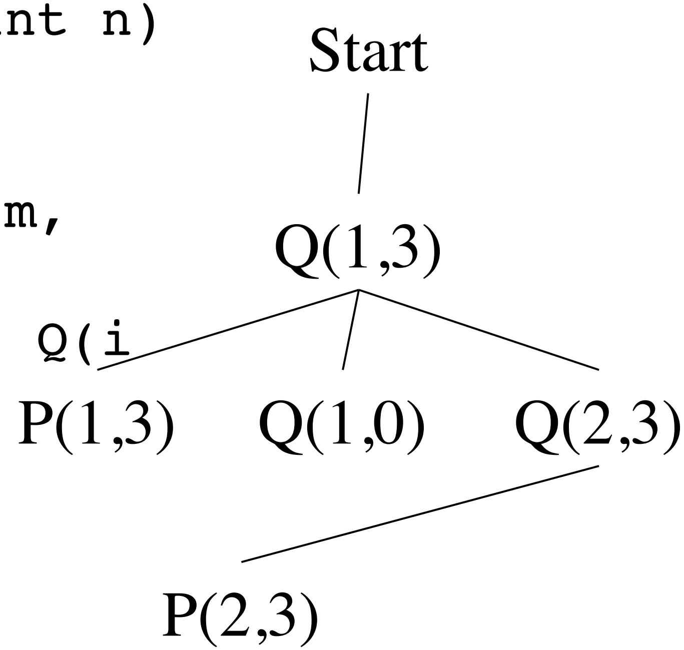


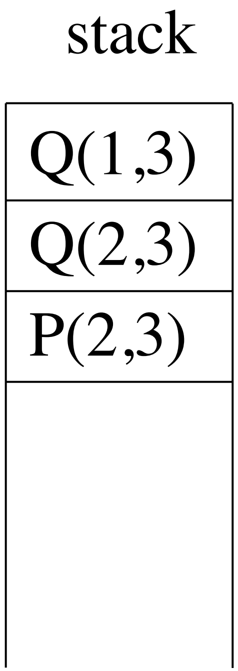


Stack does not keep track of entire activation tree, just active procedures

Stack of Active Procedures

```
int Q(int m, int n)
{
  if (n > m) {
    int i = P(m,
n);
    Q(m, i-1); Q(i
+1, n);
  }
}
```

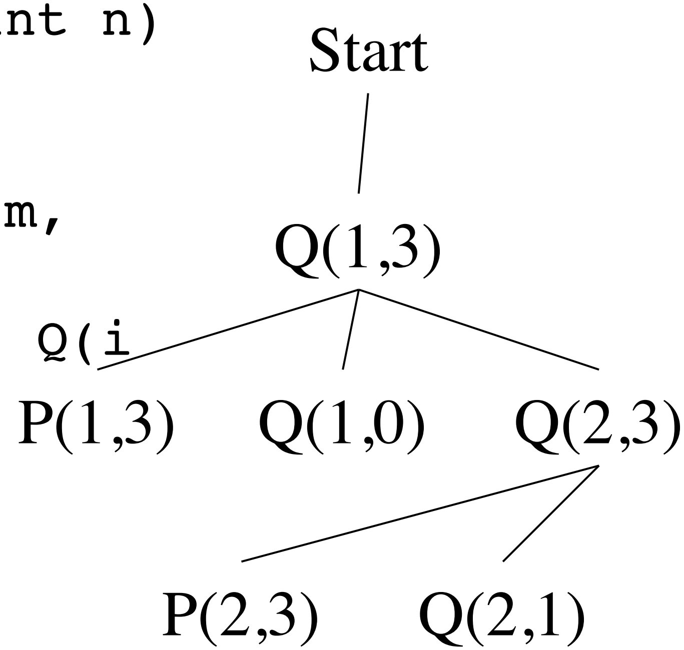


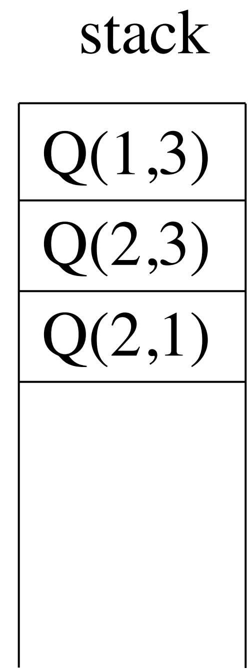


Stack does not keep track of entire activation tree, just active procedures

Stack of Active Procedures

```
int Q(int m, int n)
{
  if (n > m) {
    int i = P(m,
n);
    Q(m, i-1); Q(i
+1, n);
  }
}
```

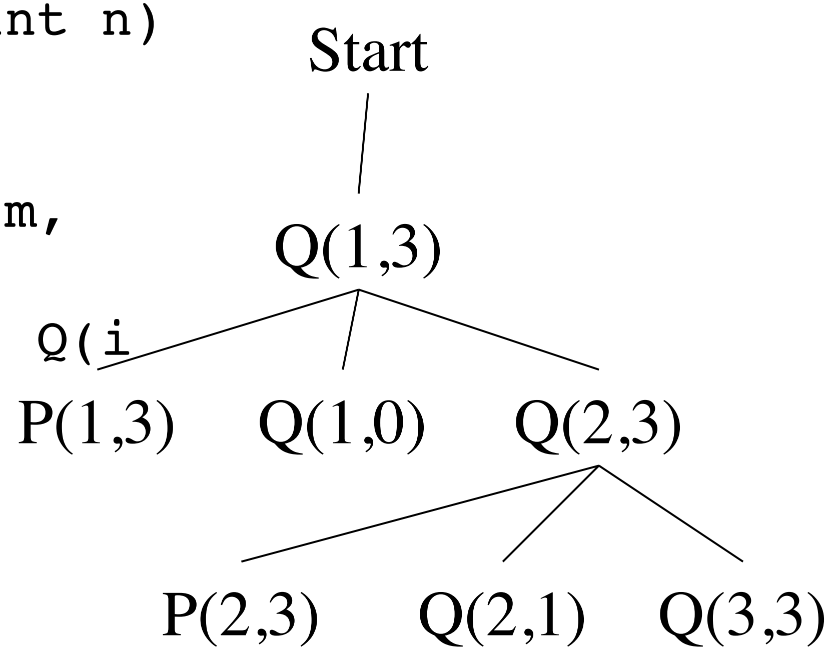


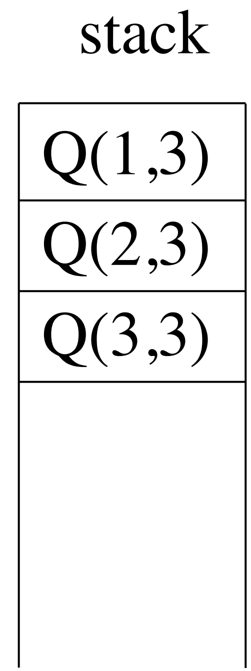


Stack does not keep track of entire activation tree, just active procedures

Stack of Active Procedures

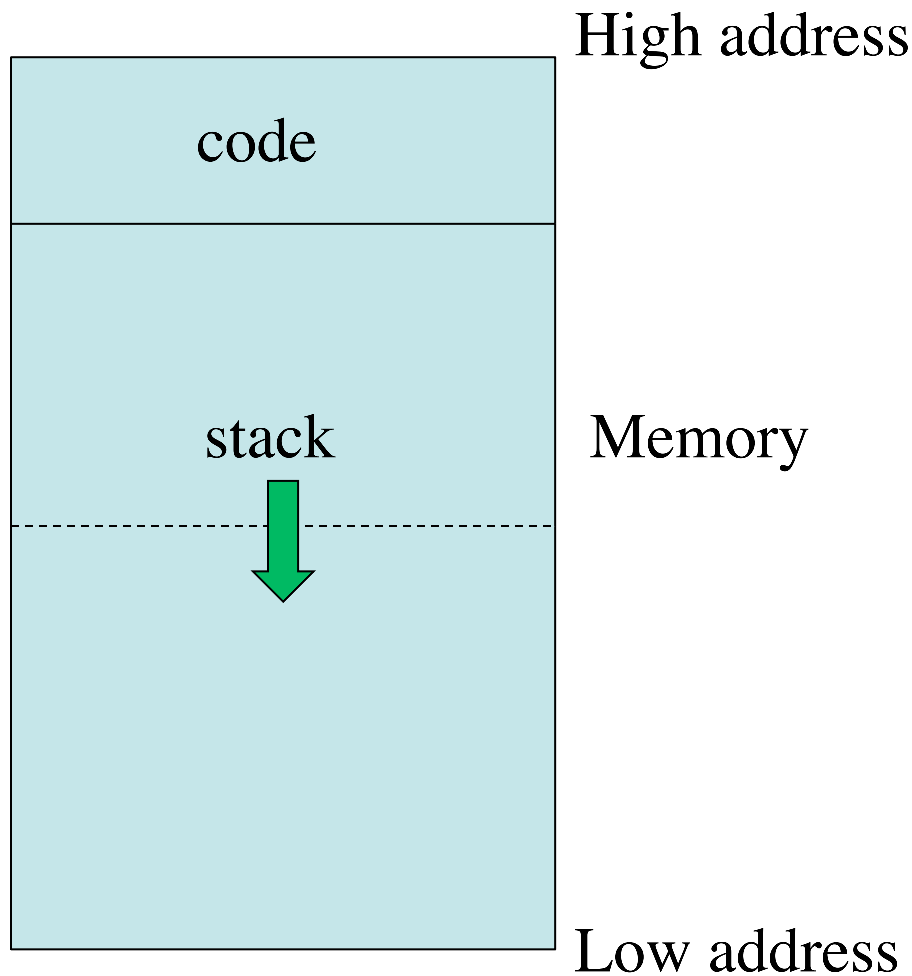
```
int Q(int m, int n)
{
  if (n > m) {
    int i = P(m,
n);
    Q(m, i-1); Q(i
+1, n);
  }
}
```





Stack does not keep track of entire activation tree, just active procedures

Memory Organization

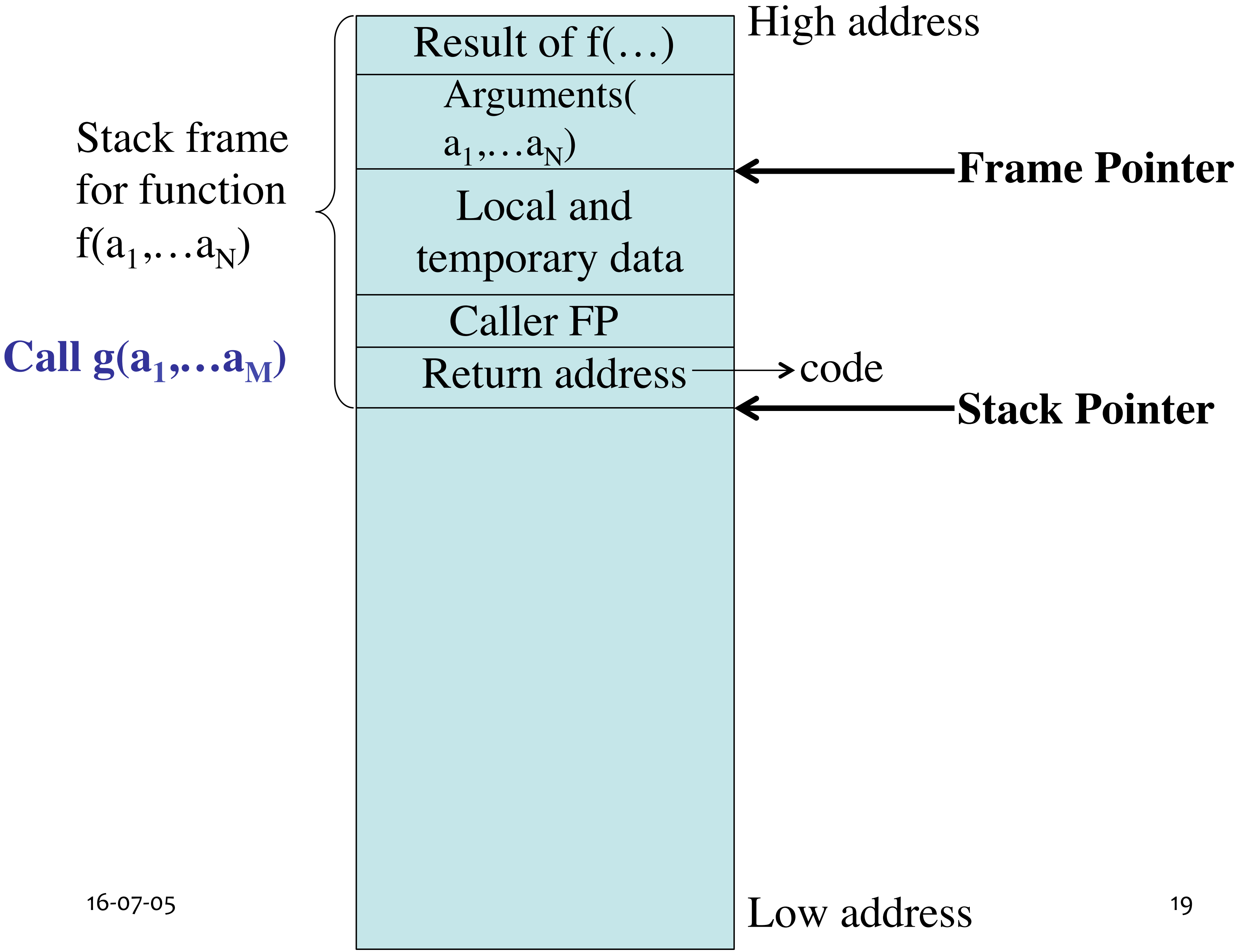


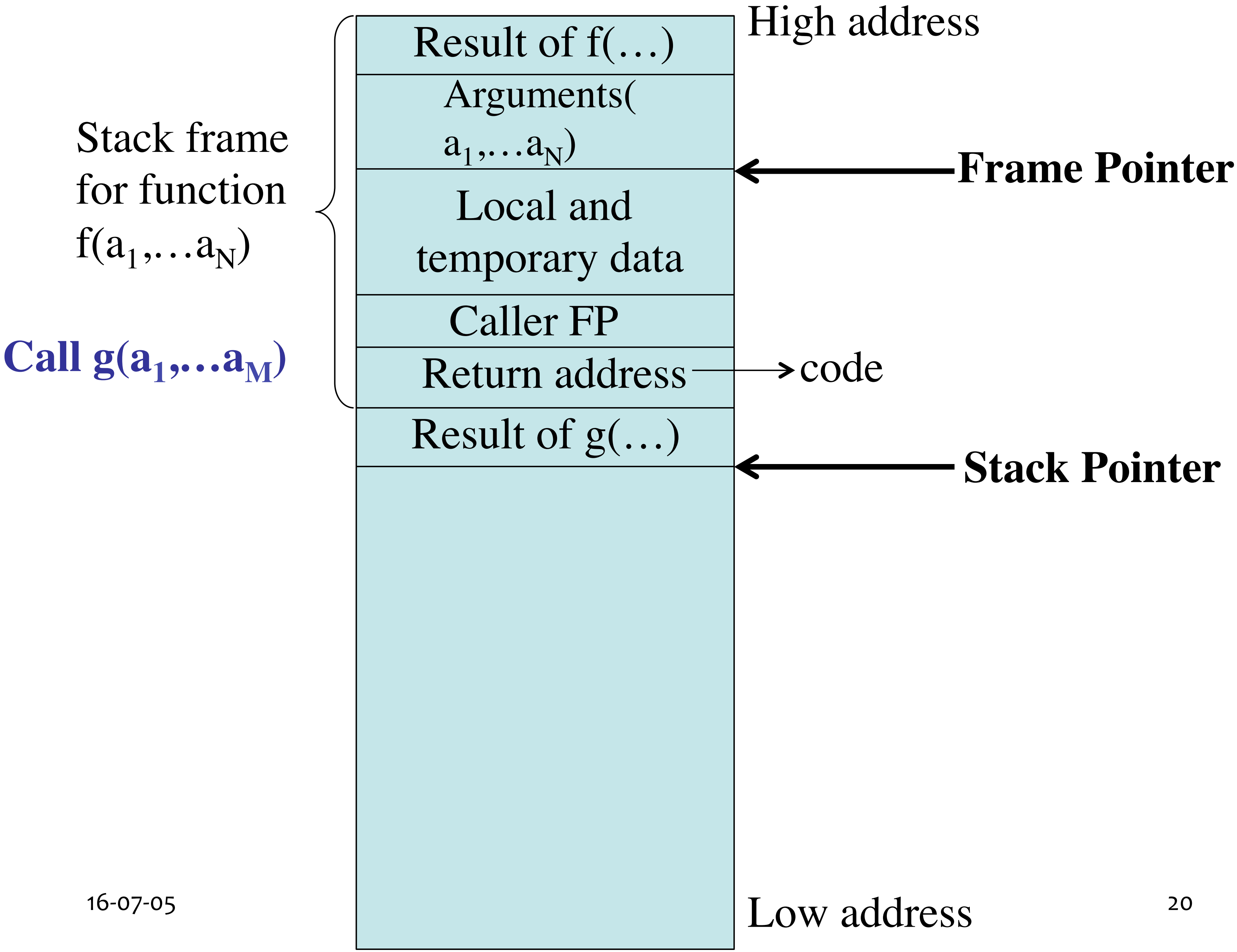
Activation Records

- The information needed to manage one procedure activation is called an *activation record* (AR) or *frame*
- If procedure F calls G, then G's activation record contains mix of info about F and G
- F is suspended until G complete, at which point F resumes
- G's AR contains information needed to
  - Complete execution of G
  - Resumes execution of F

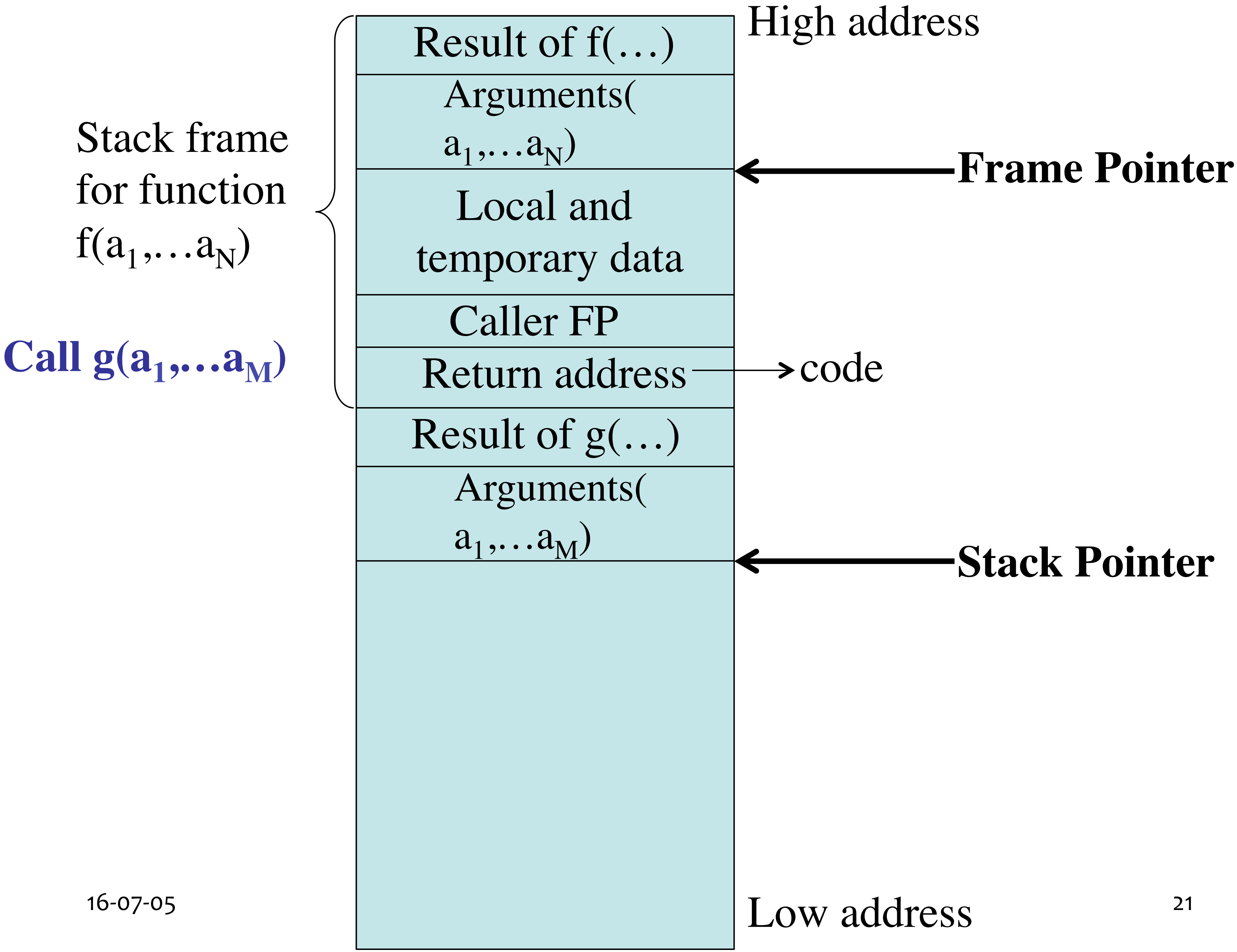
Activation Records

- A frame contains:
  - Control link (pointer to the caller frame)
  - Local data
  - Snapshot of machine state (important registers)
  - Return address
  - Link to global data
  - Parameters passed to function
  - Return value for the caller

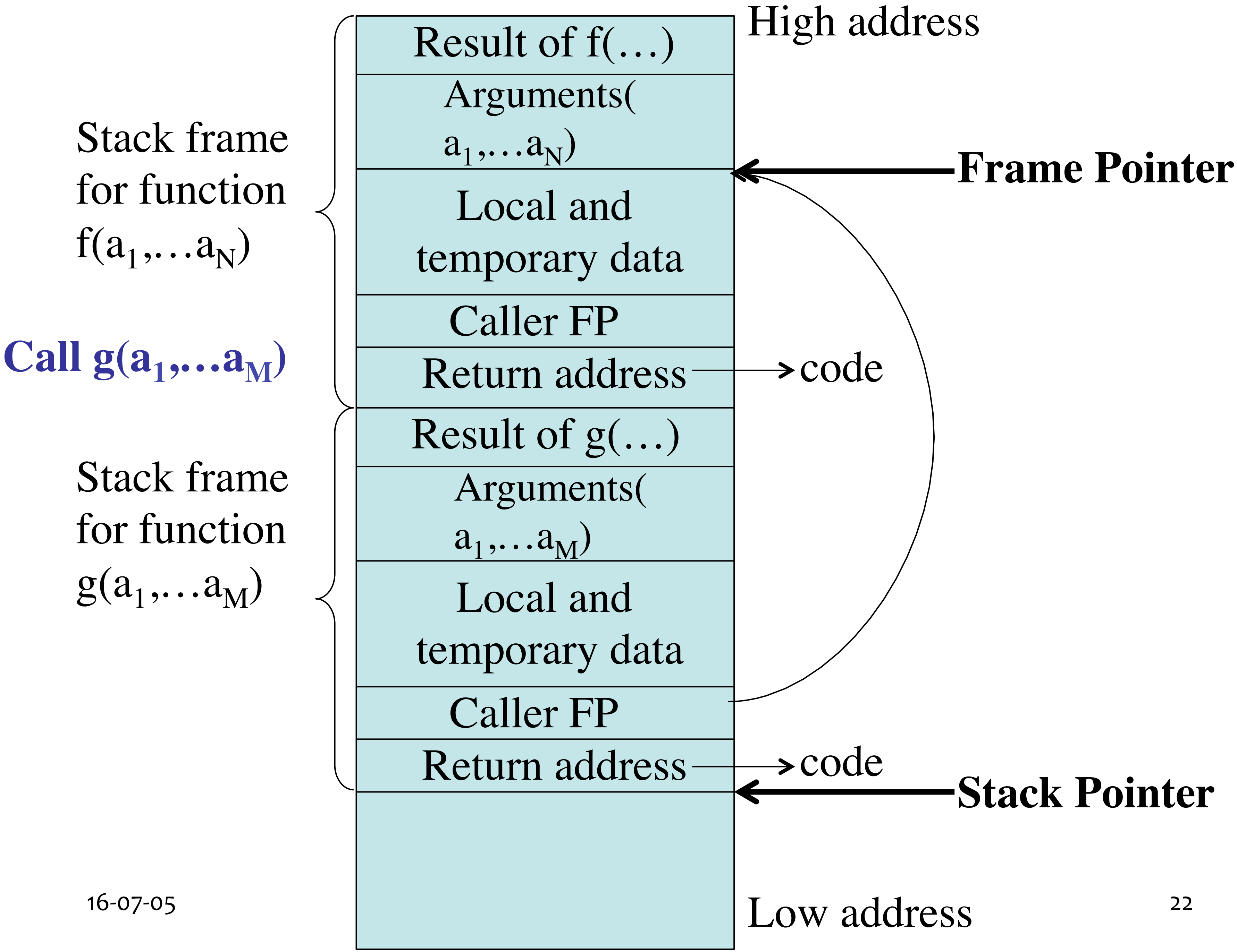

High address
Result of f(…)
Arguments($a_1,\ldots a_N$)
Frame Pointer
Stack frame for function $f(a_1,\ldots a_N)$
Local and temporary data
Caller FP
Call $g(a_1,\ldots a_M)$
Return address
code
Stack Pointer
Low address

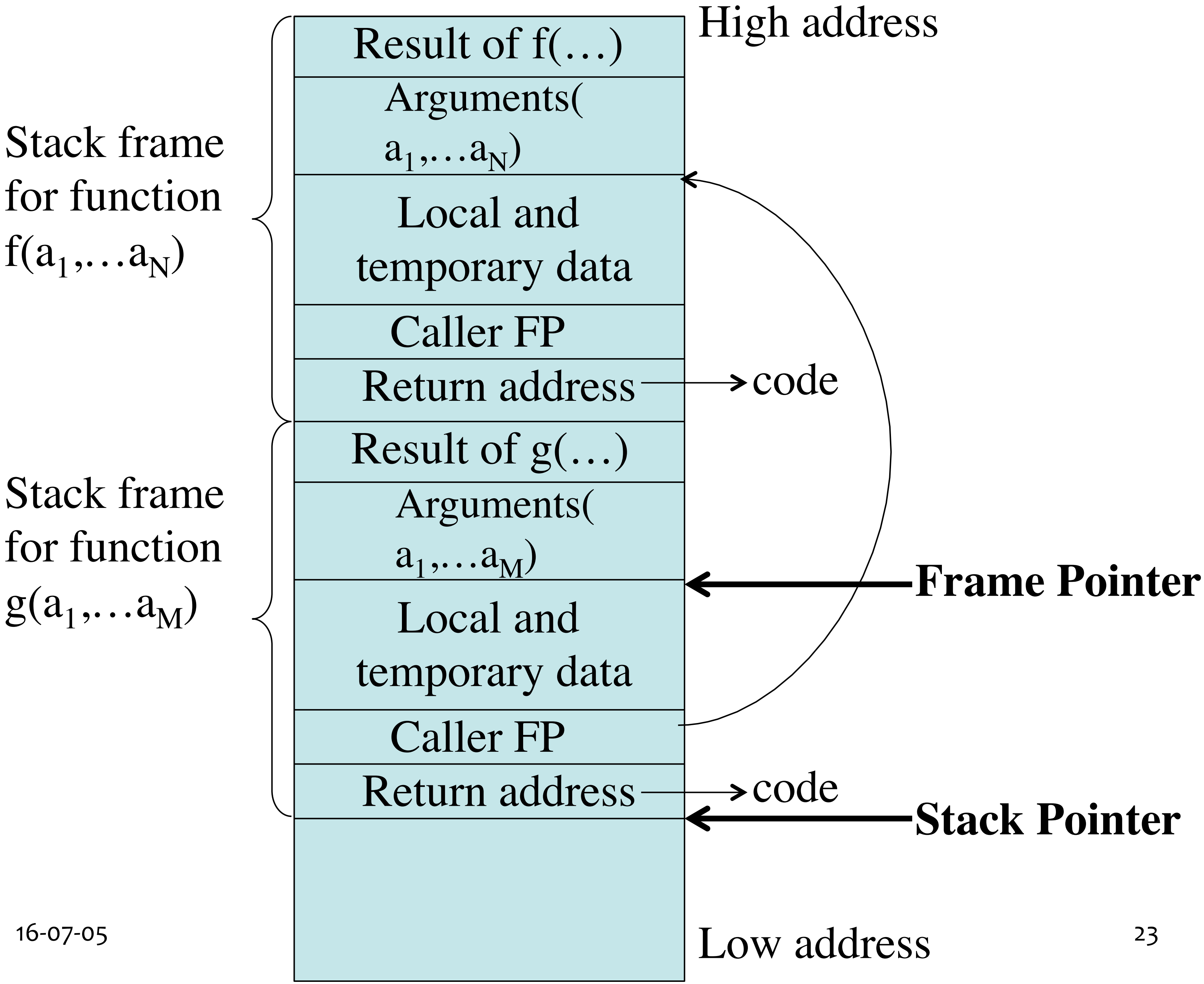


Result of f(…)
Arguments(a_1,…a_N)
Local and temporary data
Caller FP
Return address
code
Result of g(…)
High address
Low address
Frame Pointer
Stack Pointer
Stack frame for function f(a_1,…a_N)
Call g(a_1,…a_M)

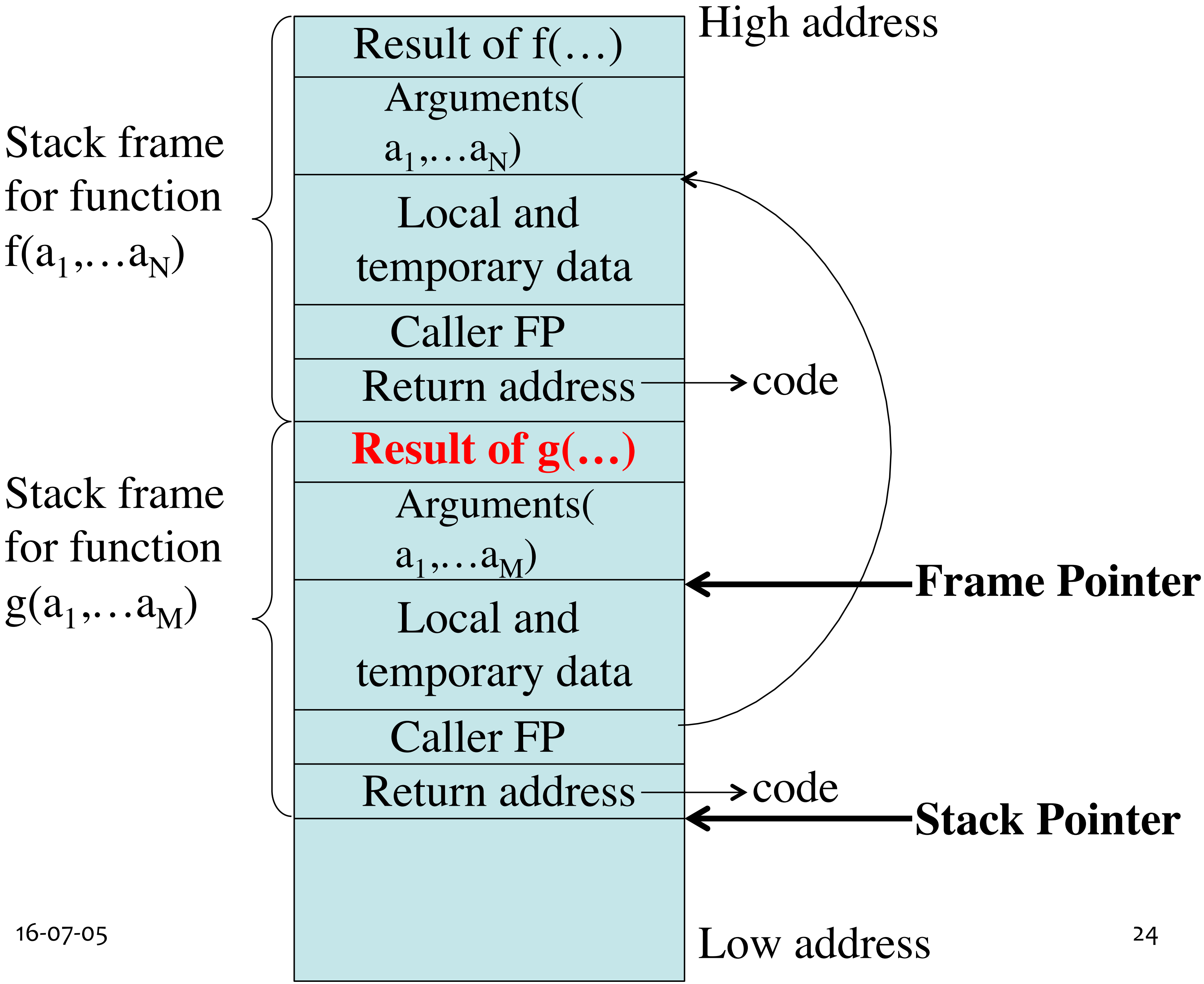


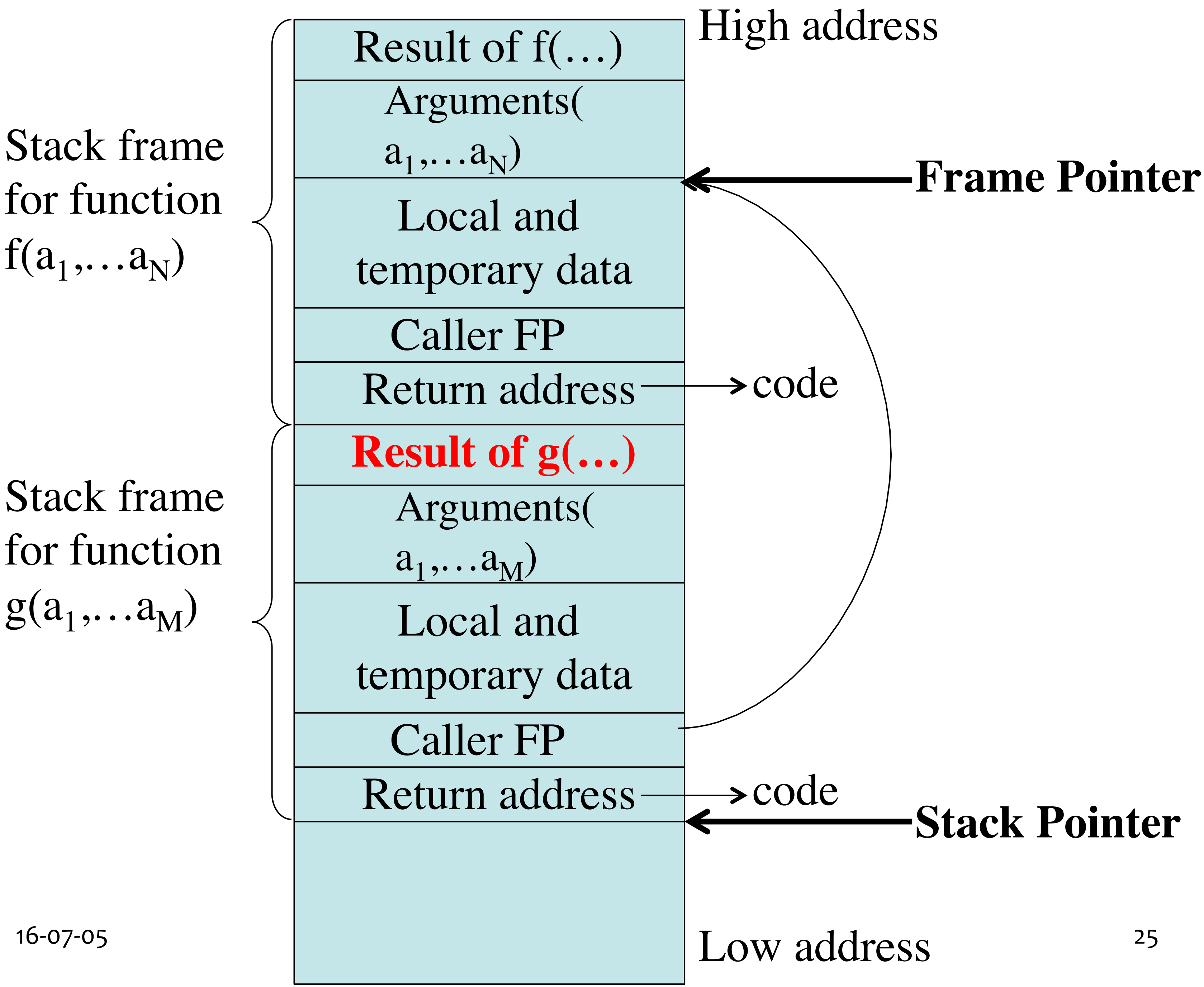
High address
Result of f(…)
Arguments(a1,…aN)
Stack frame for function f(a1,…aN)
Frame Pointer
Local and temporary data
Caller FP
Call g(a1,…aM)
Return address
code
Result of g(…)
Arguments(a1,…aM)
Stack Pointer
Low address

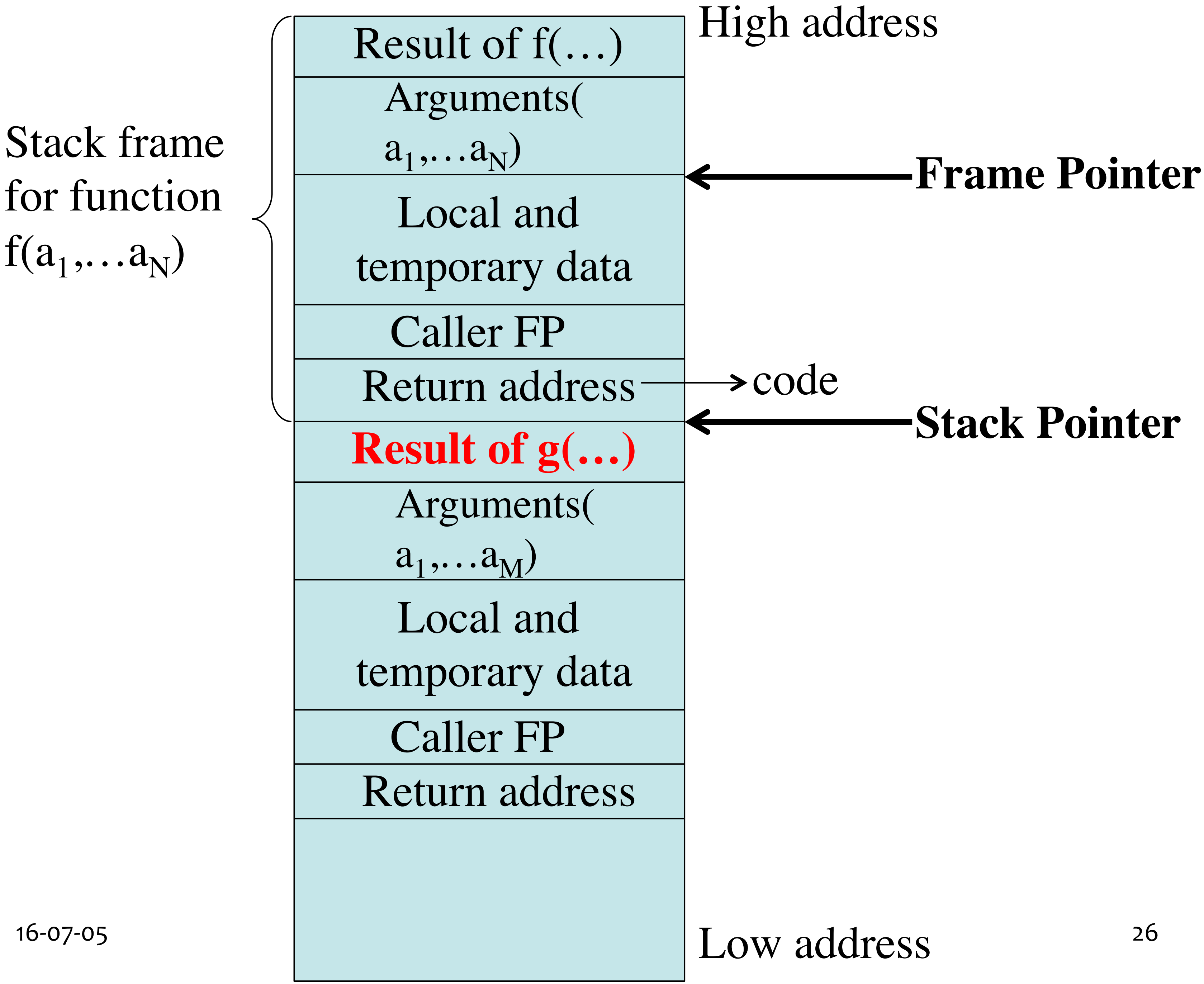

High address
Result of f(…)
Arguments($a_1,\ldots a_N$)
Frame Pointer
Stack frame for function $f(a_1,\ldots a_N)$
Local and temporary data
Caller FP
Call $g(a_1,\ldots a_M)$
Return address
code
Result of g(…)
Arguments($a_1,\ldots a_M$)
Stack frame for function $g(a_1,\ldots a_M)$
Local and temporary data
Caller FP
Return address
code
Stack Pointer
Low address



High address
Result of f(…)
Arguments($a_1,\ldots a_N$)
Stack frame for function $f(a_1,\ldots a_N)$
Local and temporary data
Caller FP
Return address
code
Result of g(…)
Arguments($a_1,\ldots a_M$)
Stack frame for function $g(a_1,\ldots a_M)$
Frame Pointer
Local and temporary data
Caller FP
Return address
code
Stack Pointer
Low address


High address
Result of f(…)
Arguments($a_1,\ldots a_N$)
Stack frame for function $f(a_1,\ldots a_N)$
Local and temporary data
Caller FP
Return address
code
Result of g(…)
Arguments($a_1,\ldots a_M$)
Stack frame for function $g(a_1,\ldots a_M)$
Frame Pointer
Local and temporary data
Caller FP
Return address
code
Stack Pointer
Low address

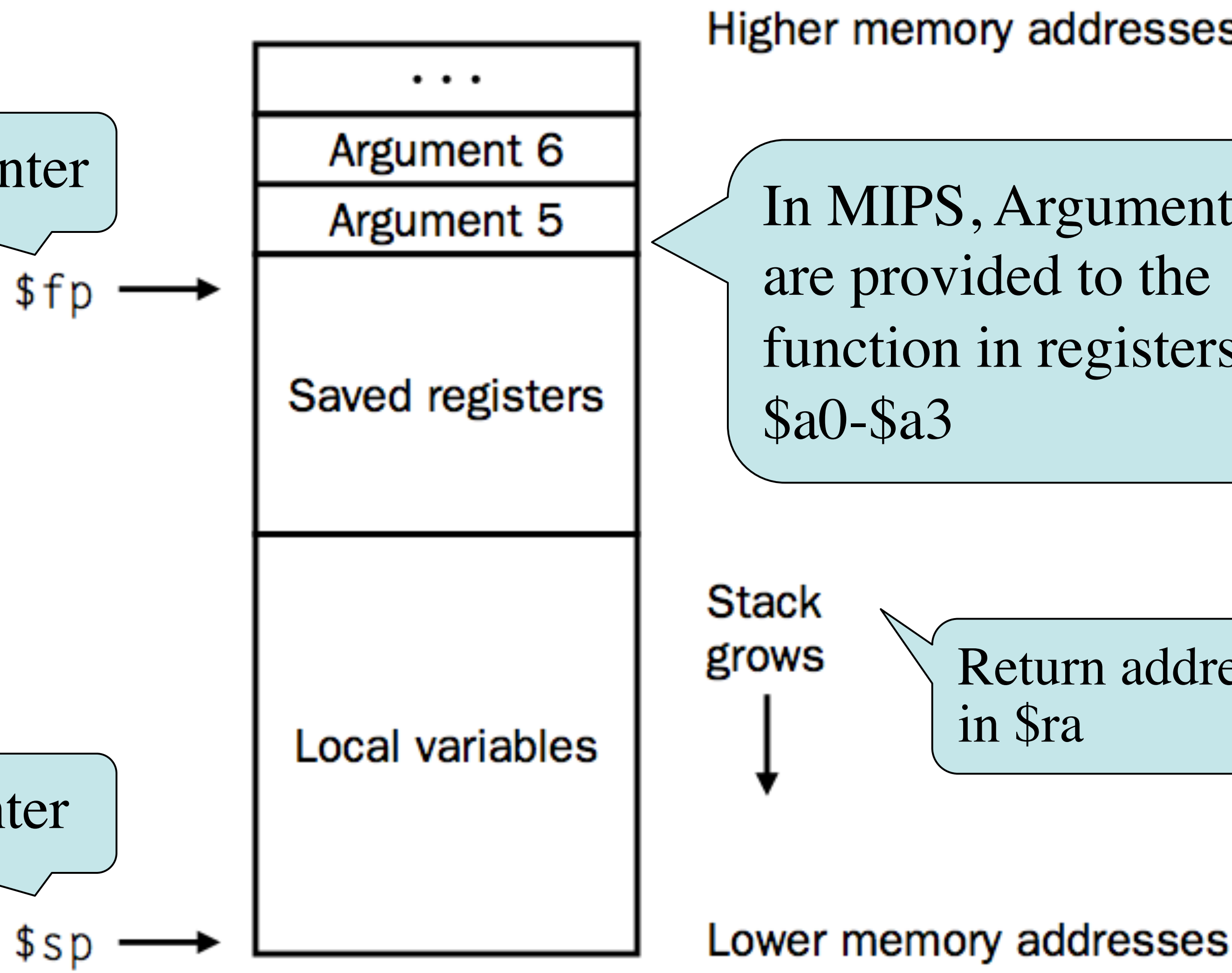

High address
Result of f(…)
Arguments($a_1,\ldots a_N$)
Local and temporary data
Caller FP
Return address
code
Frame Pointer
Stack frame for function $f(a_1,\ldots a_N)$
Result of g(…)
Arguments($a_1,\ldots a_M$)
Local and temporary data
Caller FP
Return address
code
Stack Pointer
Stack frame for function $g(a_1,\ldots a_M)$
Low address


High address
Result of f(…)
Arguments(
a1,…aN)
Frame Pointer
Stack frame
for function
f(a1,…aN)
Local and
temporary data
Caller FP
Return address
code
Stack Pointer
Result of g(…)
Arguments(
a1,…aM)
Local and
temporary data
Caller FP
Return address
Low address

Activation Record Organization

- There is nothing magic about this organization
  - Can rearrange order of frame elements
  - Can divide caller/callee responsibilities differently
  - An organization is better if it improves execution speed or simplifies code generation
- Real compilers hold as much of the frame as possible in registers
  - Especially the method result and arguments

Stack frame



Frame pointer

In MIPS, Argument 1-4 are provided to the function in registers $a0-$a3

Return address in $ra

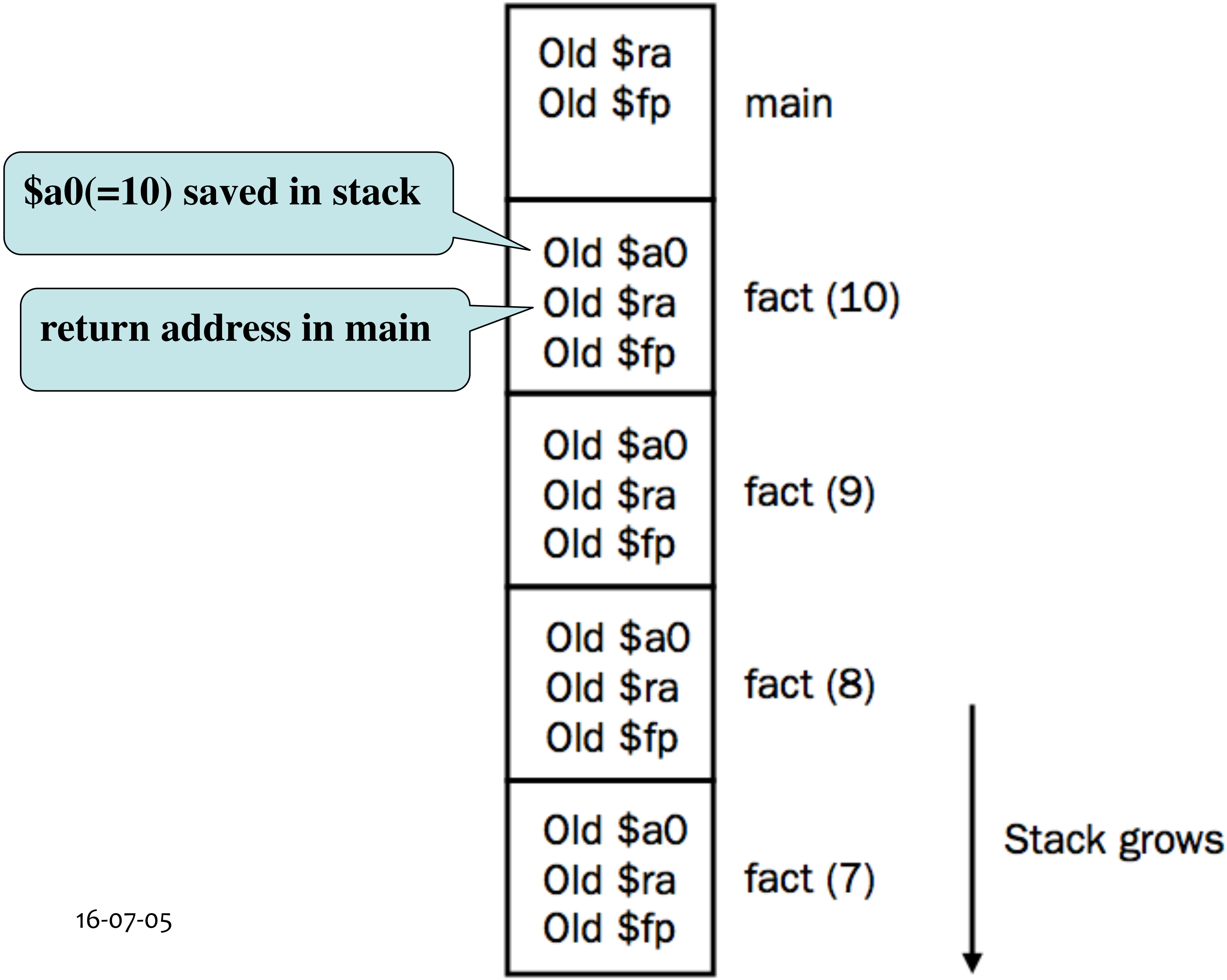
Stack pointer

```
#include <stdio.h>

main ()
{
    int n = 10;
    printf("The factorial of 10 is %d\n", fact(n));
}

int fact (int n)
{
    if (n < 1)
        return(1);
    else
        return(n * fact(n - 1));
}
```

Stack

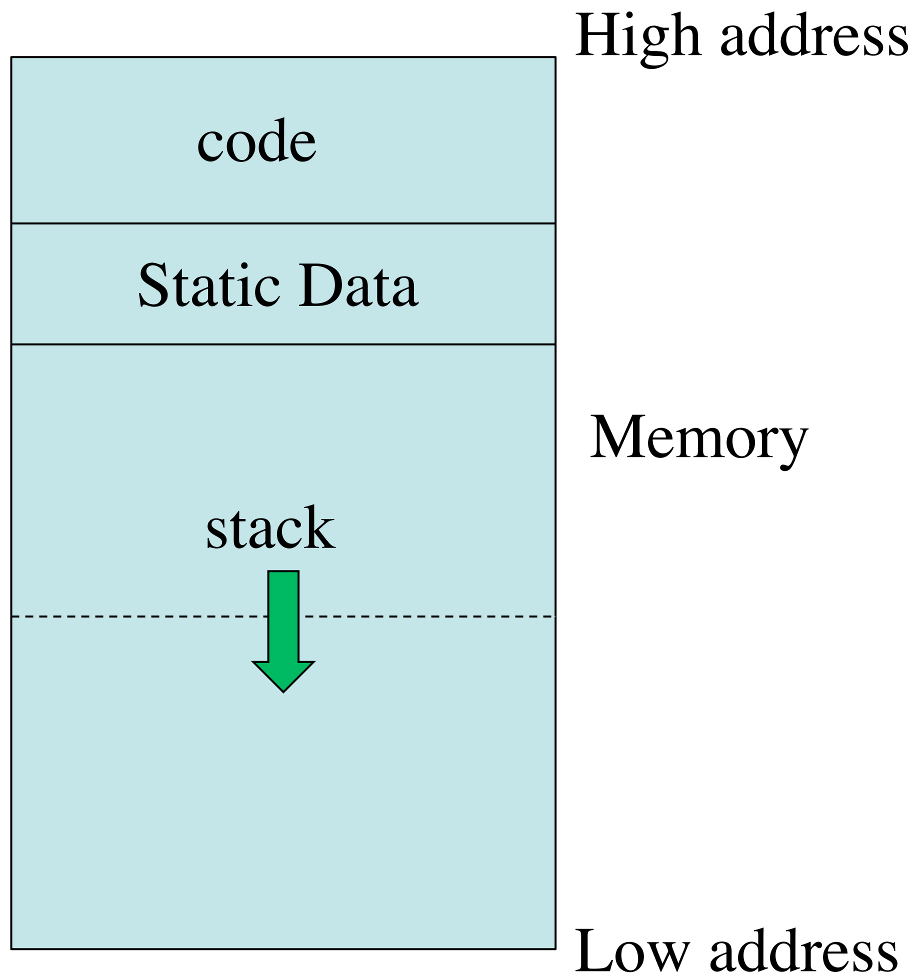


Old $ra
Old $fp
main
$a0(=10) saved in stack
return address in main
Old $a0
Old $ra
Old $fp
fact (10)
Old $a0
Old $ra
Old $fp
fact (9)
Old $a0
Old $ra
Old $fp
fact (8)
Old $a0
Old $ra
Old $fp
fact (7)
Stack grows

Global Variables

- All references to a global variable point to the same object
  - Cannot store a global in an activation record
- Globals are assigned a fixed address once
  - Variables with fixed address are "statically allocated"
- Depending on the language, there may be other statically allocated values

Memory Organization



Heap Allocation

- Any value that outlives the procedure that creates it cannot be kept in AR

  int* foo() {int * bar = new int[size]; return bar;}

  The bat value must survive de-allocation of foo’s AR

- Languages with dynamically allocated data use a heap to store dynamic data

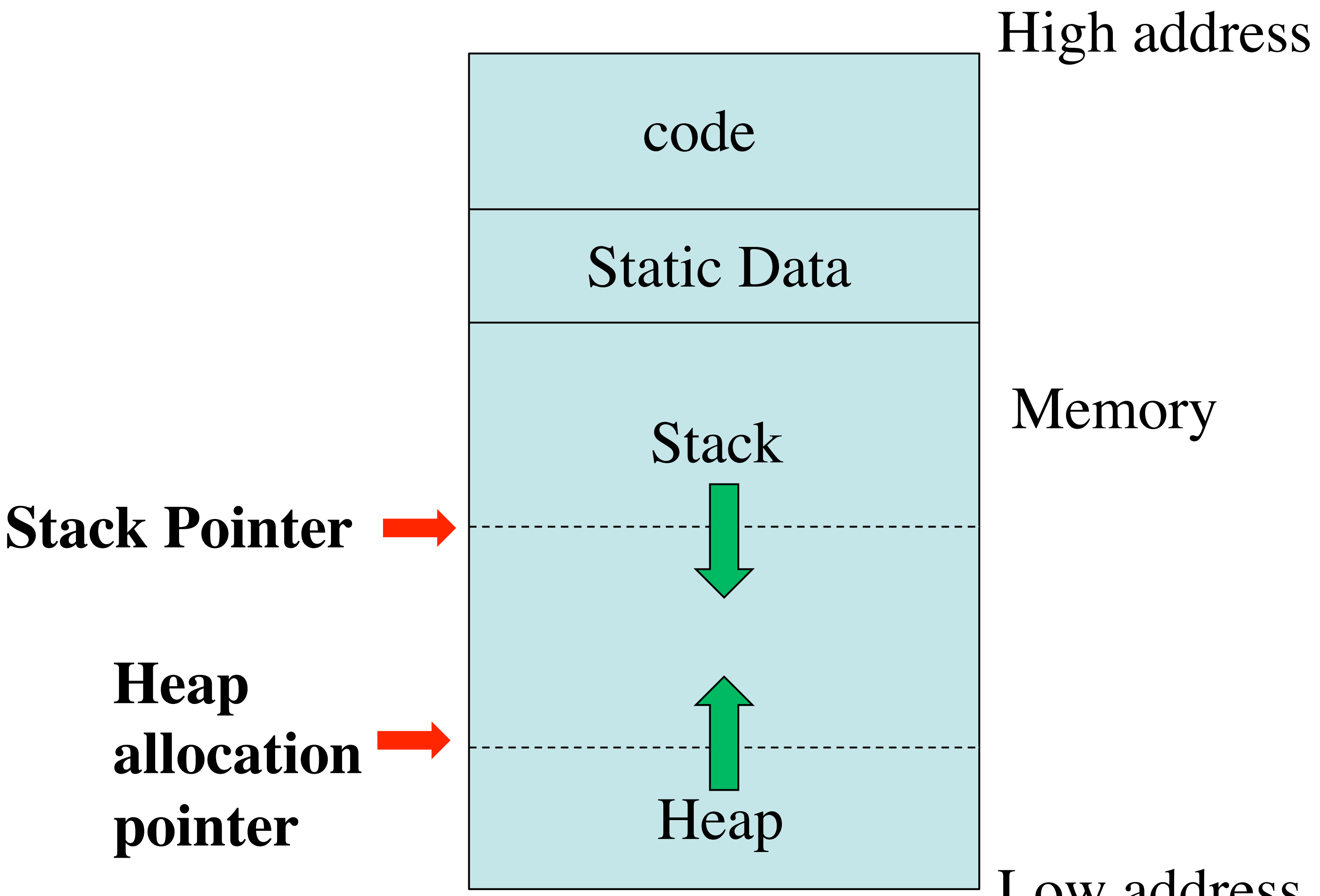
Memory organization

- The code area contains object code
  - For many languages, fixed size and read only
- The static area contain data (not code) with fixed addresses (e.g., global data)
  - Fixed size, may be readable or writable
- The stack contains and AR for each currently active procedure
  - Each AR usually fixed size, contains locals
- Heap contains all other data
  - In C, heap is managed by *malloc* and *free*

Heap and Stack Management

- Both the heap and stack grow
- Must take care that they do not grow into each other
- Solution: start heap and stack at opposite ends of memory and let them grow towards each other

Memory Organization



Alignment

- Most modern machines are 32 or 64 bit
  - 8 bits in a byte
  - 4 or 8 bytes in a word
  - Machines are either byte or word addressable
- Data is word aligned if it begins at a word boundary
- Most machines have some alignment restrictions
  - Or performance penalties for poor alignment

Padding

- Example: A string

“Hello”

Takes 6 characters (including a terminating \0)

H	e	l	l	o	\0		

- To word align next word, add 2 “padding” characters
- The padding is not part of the string, it’s juts unused memory

Padding

- Compilers may insert unused bytes called "padding bytes" after structure members to ensure that each member is appropriately aligned.

```
struct widget {
    char m1;
    int m2;
    char m3;
};
```

On a word aligned machine: add 3 bytes of padding after m1 and m3

Summary

- Run-time support for functions
- Dealing with (potentially infinite) recursion
- Activation records for each function invocation
- Storage allocation for activation records in recursive function calls
- Stack allocation is easiest to implement while retaining recursion
- Functional PLs use heap allocation

Extra Slides

Storage Allocation for Functions

- Stack Allocation √
  - Storage for recursive functions is organized as a stack: last-in first-out (LIFO) order
  - Activation records are associated with each function activation
  - Activation records are pushed onto the stack when a call is made to the function
  - Size of activation records can be fixed or variable

Storage Allocation for Functions

- Stack Allocation √
  - Sometimes a minimum size is required
  - Variable length data is handled using pointers
  - Locals are deleted after activation ends
  - Caller locals are reinstated and execution continues
  - C, Pascal and most modern programming languages

Storage Allocation for Functions

- Heap Allocation
  - In some special cases stack allocation is not possible
  - If local variables must be retained after the activation ends
  - If called activation outlives the caller
  - Anything that violates the last-in first-out nature of stack allocation e.g. closures in Lisp and other functional PLs

Storage Allocation for Functions

- Function Composition: (f•g)(x) = f(g(x))

```
class Compose {
        fun sq (int x) { return (x * x); }
        fun f (fun m) { return (m•h); }
        fun h () { return sq; }
        fun g (fun z) { return (sq•z); }
        int main() {
                fun v = g•h;
                print_int((v())(3));
        }
}
```

Storage Allocation for Functions

- Function Composition: (f•g)(x) = f(g(x))

```
class Compose {
    fun sq (int x) { return (x * x); }
    fun f (fun m) { return (m•h); }
    fun h () { return sq; }
    fun g (fun z) { return (sq•z); }
    int main() {
        fun v = g•h;
        callout("print_int", (v())(3));
    }
}
```

v = g•h

v() = (g•h)()

v() = g(h())

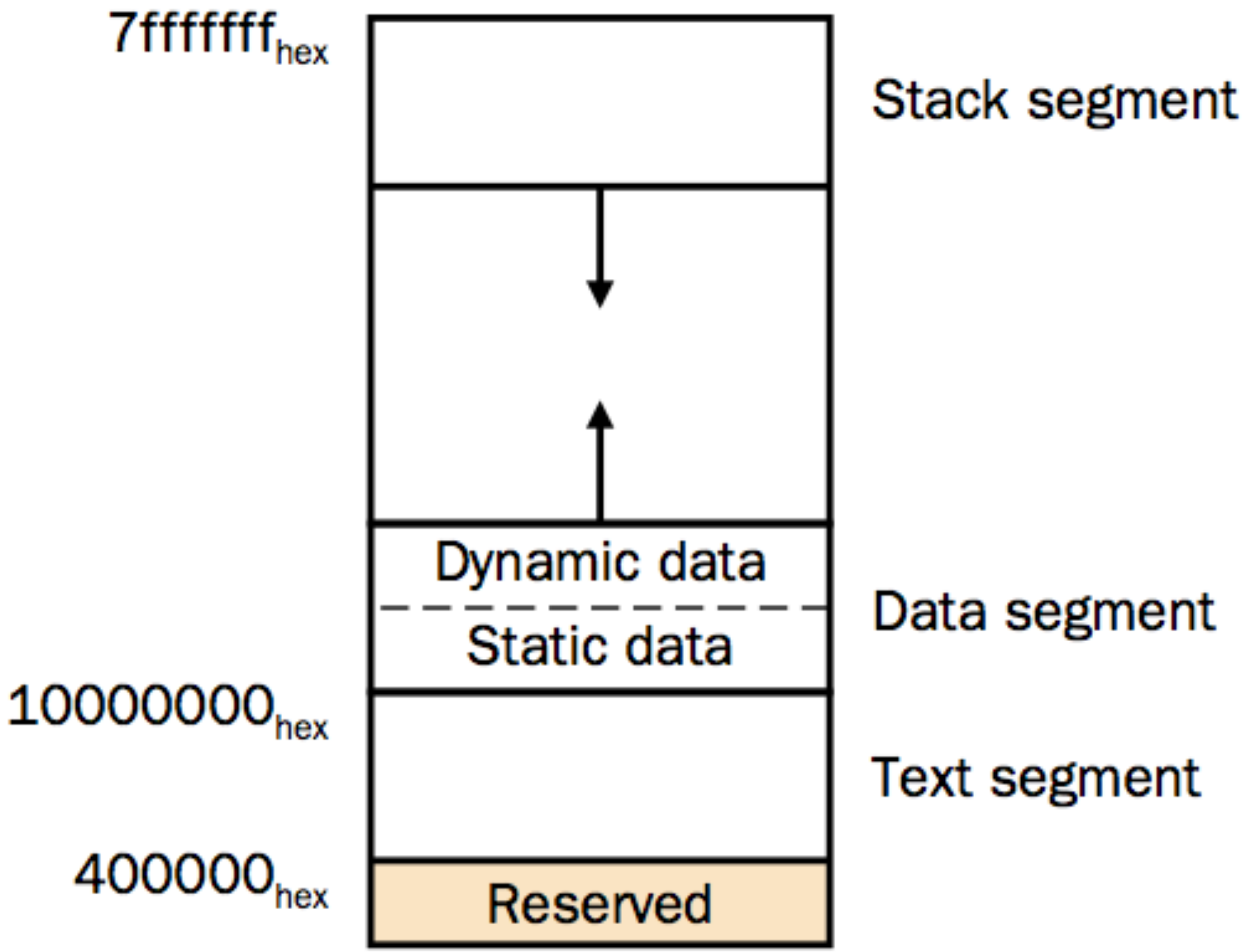
v() = g(sq)

v() = (sq•sq)

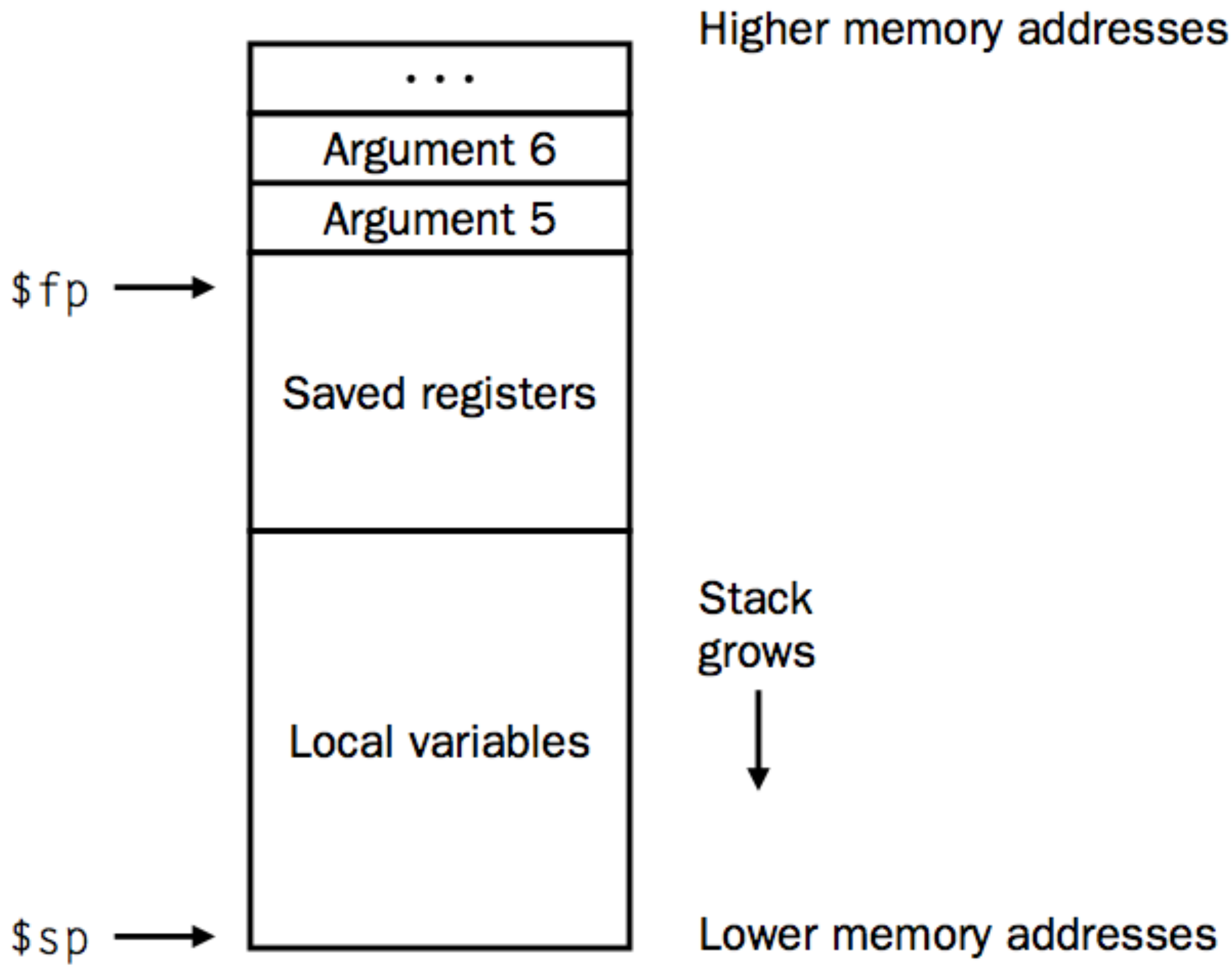
v()(3) = (sq•sq)(3)

v()(3) = (sq(sq(3))

Run-time Memory

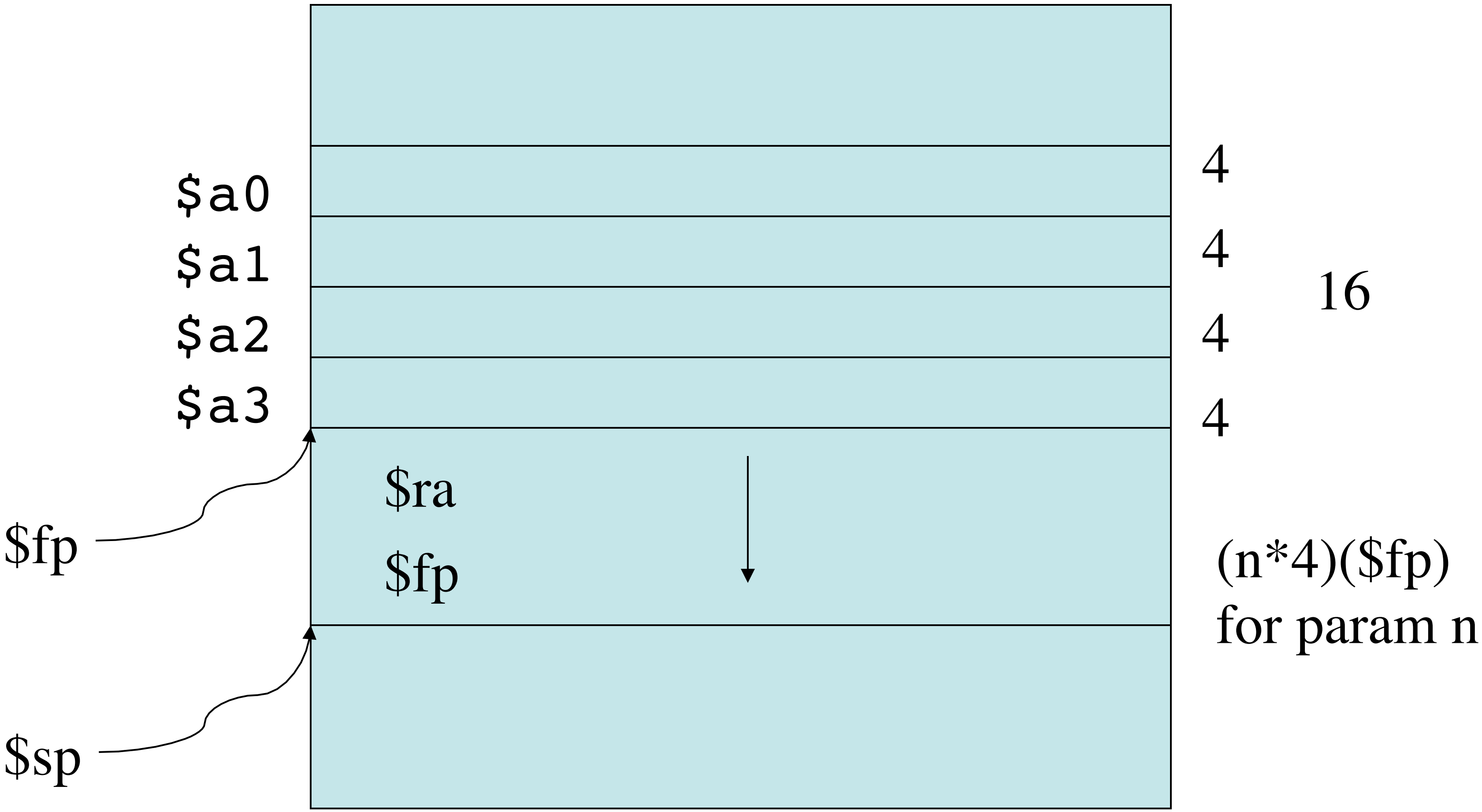


Stack frame



Higher memory addresses
. . .
Argument 6
Argument 5
$fp
Saved registers
Local variables
Stack grows
$sp
Lower memory addresses

Example: MIPS stack frame



Parameter Passing Conventions

- Differences based on:
  - The parameter represents an r-value (the rhs of an expr)
  - An l-value
  - Or the text of the parameter itself
- Call by Value
  - Each parameter is evaluated
  - Pass the r-value to the function
  - No side-effect on the parameter

Parameter Passing Conventions

- Call by Reference
  - Also called call by address/location
  - If the parameter is a name or expr that is an l-value then pass the l-value
  - Else create a new temporary l-value and pass that
  - Typical example: passing array elements a[i]

Parameter Passing Conventions

- Copy Restore Linkage
  - Pass only r-values to the called function (but keep the l-value around for those parameters that have it)
  - When control returns back, take the r-values and copy it into the l-values for the parameters that have it
  - Fortran
- Call by Name
  - Function is treated like a macro (a #define) or in-line expansion
  - The parameters are literally re-written as passed arguments (keep caller variables distinct by renaming)

Parameter Passing Conventions

- Lazy evaluation
  - In some languages, call-by-name is accomplished by sending a function (also called a thunk) instead of an r-value
  - When the r-value is needed the function is called with zero arguments to produce the r-value
  - This avoids the time-consuming evaluation of r-values which may or may not be used by the called function (especially when you consider short-circuit evaluation)
  - Used in lazy functional languages

Parameter Passing Conventions

- Call-by-need
  - Similar to lazy evaluation, but more efficient
  - To avoid executing similar r-values multiple times, some languages used a memo slot to avoid repeated function evaluations
  - A function parameter is only evaluated when used inside the called function
  - When used multiple times there is no overhead due to the memo table
  - Haskell