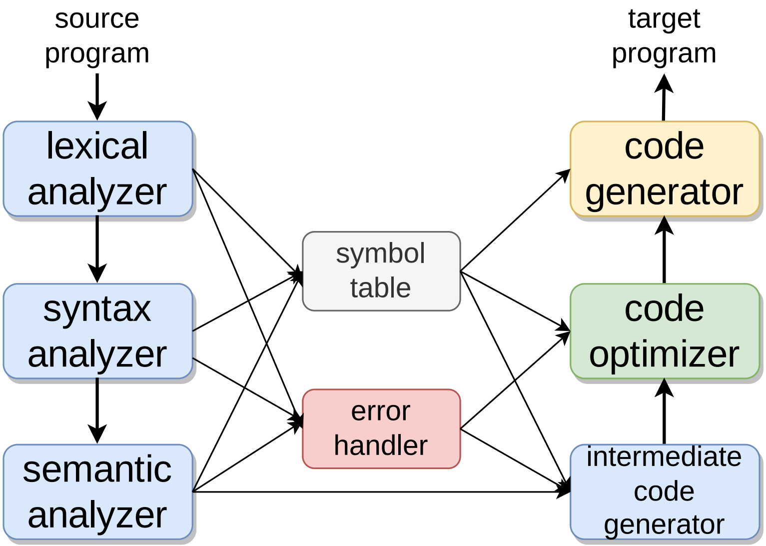
CS 335: Runtime Environments

Swarnendu Biswas

Department of Computer Science and Engineering,
Indian Institute of Technology Kanpur

Sem 2023-24-II

An Overview of Compilation



Abstraction Spectrum

- Translating source code requires dealing with all programming language abstractions
  - For example, names, procedures, objects, control flow, and exceptions
- Physical computer operates in terms of several primitive operations
  - For example, arithmetic, data movement, and control jumps
- It is not enough to just translate intermediate code to machine code, need to manage memory when a program is executing

Runtime Environment

Definition

A runtime environment is a set of **data structures** maintained at run time to **implement high-level program structures**

- Examples of data structures are stack, heap, and virtual function tables
- Program structures depend on the features of the source and the target language, examples are procedures and inheritance

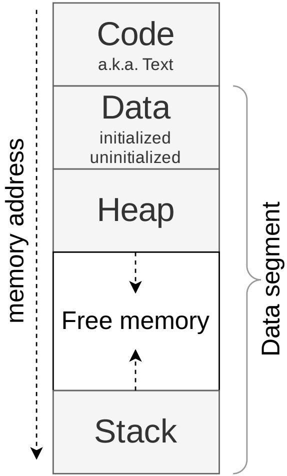
- Compilers create and manage the runtime environment in which the target programs execute
- Runtime deals with the layout, allocation, and deallocation of storage locations, linkages between procedures, and passing parameters among other concerns

Issues Dealt by Runtime Environments

- How to pass parameters when a procedure is called?
- What happens to locals when procedures return from an activation?
- Can a procedure refer to nonlocal names? If yes, then how?
- How to support recursive procedures?
- …

Storage Organization

- Target program runs in its own logical address space
- Size of generated code is usually fixed at compile time unless code is loaded or produced dynamically
- Compiler can place the executable at fixed addresses
- Runtime storage can be subdivided into
  - Target code
  - Static data objects such as global constants
  - Stack to keep track of procedure activations and local data
  - Heap to keep all other information like dynamic data



Virtual Address Space

```cpp
#include <cstdlib>
#include <iostream>
int main() {
  int x = 3;
  std::cout << "Start of code segment: " <<
            << (void*)&main << "\nStart of heap segment: "
            << new int << "\nStart of stack segment: " << &x << "\n";
  return EXIT_SUCCESS;
}
```

```
$ g++ va-space.cpp -o va-space
$ ./va-space
  Start of code segment: 0x55da0d8df1e9
  Start of heap segment: 0x55da0f8722c0
  Start of stack segment: 0x7ffd7d557b44
```

The Abstraction: Address Spaces

Program Segments

```c
int g_i_data = 2; // initialized global variables are in data
// Uninitialized global or global initialized to zero are in .bss
float g_f_bss;
long g_l_bss = 0;
const int MAX = 10000; // .rodata
const int MIN = 100; // .rodata

int main() {
  static double s_d_bss; // uninitialized static in .bss
  // Initialized static in .data
  static int s_i_data = 77;
  static char s_str[] = "CS335!\n";
  const float pi = 3.14; // local constant in .rodata
  // Local non-static variables are on the stack
  int l_value = 42;
  return 0;
}
```

Program Segments

```
$ g++ --save-temps -o segments.out segments.cpp
$ size segments.out-segments.o
  text data bss dec hex filename
  135    16  24 175  af segments.out-segments.o
$ objdump -CS -s -j .data segments.out-segments.o
  ...
  0000000000000000 <g_i_data>:
     0:    02 00 00 00                                             ....
  0000000000000004 <main::s_i_data>:
     4:    4d 00 00 00                                             M...
  0000000000000008 <main::s_str>:
     8:    43 53 33 33 35 21 0a 00                                 CS335!..
$ objdump -CS -s -j .bss segments.out-segments.o
  0000000000000000 <g_f_bss>:
          ...
  0000000000000008 <g_l_bss>:
          ...
  0000000000000010 <main::s_d_bss>:
          ...
```

Strategies for Storage Allocation

Static allocation

- Lay out storage at compile time only by studying the program text
- Memory allocated at compile time will be in the static area

Dynamic allocation

- Storage allocation decisions are made when the program is running
- Stack allocation — Manage run-time allocation with a stack storage
  - Local data are allocated on the stack
- Heap allocation — Memory allocation and deallocation can be done at any time
  - Requires memory reclamation support

Static Allocation

Names are bound to storage locations at compilation time

- Bindings do not change, so no runtime support is required
- Names are bound to the same location on every invocation
- Values are retained across activations of a procedure

Limitations

- Size of all data objects must be known at compile time
- Data structures cannot be created dynamically
- Recursive procedures are not allowed

Allocating Arrays Statically

```
#include <cstdlib>
#include <iostream>
using std::cout;
#define NUM_ELEMS (1 << 30)
int main() {
  int large_array[NUM_ELEMS];
  cout << "Allocation successful!";
  for (int i = 0; i < NUM_ELEMS; i++) {
    large_array[i] = 0;
    cout << "Array[i]: " << large_array[i] << "\n";
  }
  return EXIT_SUCCESS;
}
```

```
$ g++ static-large-array.cpp -o static-large-array.out
$ ./static-large-array.out
  './static-large-array.out' terminated by signal SIGSEGV (Address boundary error)
```

Static vs Dynamic Allocation

Static Allocation

- Variable access is fast
  - Addresses are known at compile time
- Cannot support recursion

Dynamic Allocation

- Variable access is slow
  - Accesses need redirection through stack/heap pointer
- Supports recursion

Stack vs Heap Allocation

Stack

- Allocation/deallocation is automatic
- Fast allocation, requires only adjusting the stack pointer
- Space for allocation is limited

Heap

- Allocation/deallocation is explicit
- Allocation is more expensive
- Challenge is heap fragmentation

Comparing the Cost of Stack and Heap Allocations

```
#define NUM_ITERS (1e9)
using HR = std::chrono::high_resolution_clock;
using HRTimer = HR::time_point;
using std::chrono::duration_cast;
using std::chrono::microseconds;
void on_stack() { int i; }
void on_heap() { int* i = new int; }
int main() {
  HRTimer start = HR::now();
  for (int i = 0; i < NUM_ITERS; ++i) { on_stack(); }
  HRTimer end = HR::now();
  auto duration = duration_cast<microseconds>(end - start).count();
  cout << "Time for per on_stack alloc: " << (float)duration / NUM_ITERS << "us\n";
  start = HR::now();
  for (int i = 0; i < NUM_ITERS; ++i) { on_heap(); }
  end = HR::now();
  duration = duration_cast<microseconds>(end - start).count();
  cout << "Time for per heap alloc: " << ((float)duration / NUM_ITERS) / 2 << " us\n";
  return EXIT_SUCCESS;
}
```

Comparing the Cost of Stack and Heap Allocations

```
#define NUM_ITERS (1e9)
using HR = std::chrono::high_resolution_clock;
using HRTimer = HR::time_point;
using std::chrono::duration_cast;
using std::chrono::microseconds;
void on_stack() { int i; }
void on_heap() { int* i = new int; }
int main() {
  HRTimer start = HR::now();
  for (int i = 0; i < NUM_ITERS; ++i) { on_stack(); }
  HRTimer end = HR::now();
  auto duration = duration_cast<microseconds>(end - start).count();
  cout << "Time for per on_stack alloc: " << (float)duration / NUM_ITERS << "us\n";
  start = HR::now();
  for (int i = 0; i < NUM_ITERS; ++i) { on_heap(); }
  end = HR::now();
  duration = duration_cas
  cout << "Time for per h                                                        ;
  return EXIT_SUCCESS;
}
```

```
$ g++ stack-heap-cost.cpp -o stack-heap-cost.out
$ ./stack-heap-allocation.out
  Time for per stack alloc: 0.0017 us
  Time for per heap alloc: 0.0069 us
```

Procedure Abstraction

Activations, calling conventions, and accessing local and nonlocal data

Procedure Calls

- Procedure definition is a declaration that associates an identifier with a statement (procedure body)
  - Formal parameters appear in a declaration while actual parameters appear when a procedure is called

+ Important abstraction in programming
  - Provides control abstraction and a name space
  - Defines critical interfaces among large parts of a software
+ Creates a controlled execution environment
  - Each procedure has its own private named storage or name space
  - Executing a call instantiates the callee's name space

Control Abstraction

- Each language has rules to
  - Invoke a procedure (pass control by manipulating the PC)
  - Map a set of arguments from the caller's name space to the callee's name space (pass data)
  - Allocate space for local variables when a procedure executes
  - Return control to the caller, and continue execution after the call
- **Linkage convention** standardizes the actions taken by the compiler and the OS to make a procedure call

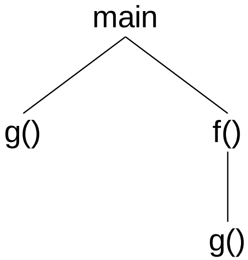
More on Procedure Calls

- Each execution of a procedure P is an **activation** of the procedure P
- A procedure is recursive if an activation can **begin before** an earlier activation of the same procedure has ended
  - If a procedure is recursive, several activations may be alive at the same time
- The **lifetime** of an activation of P is the sum of all the steps to execute P and all the steps in procedures that P calls
- Given activations of two procedures, their lifetimes are either non-overlapping or nested

Activation Tree

- Depicts the way control enters and leaves activations
  - Root represents the activation of `main()`
  - Each node represents the activation of a procedure
  - Node *a* is the parent of *b* if control flows from *a* to *b*
  - Node *a* is to the left of *b* if lifetime of *a* occurs before *b*
- Flow of control in a program corresponds to depth-first traversal of the activation tree

```
int g() { return 42; }
int f() { return g(); }
int main() {
  g();
  f();
}
```

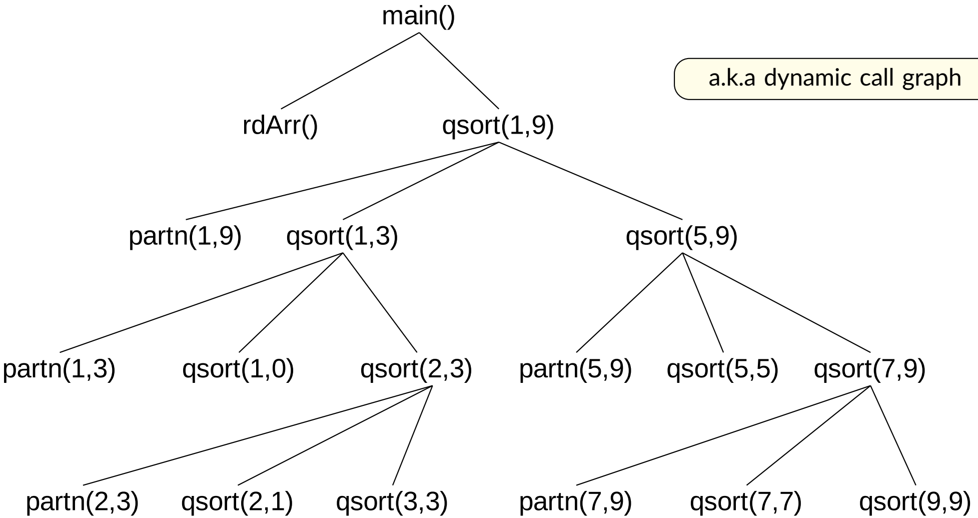


Quicksort Code

```c
int a[11];
void readArray() {
  int i;
  ...
}
int main() {
  readArray();
  a[0] = -99999;
  a[10] = 99999;
  quicksort(1, 9);
}
```

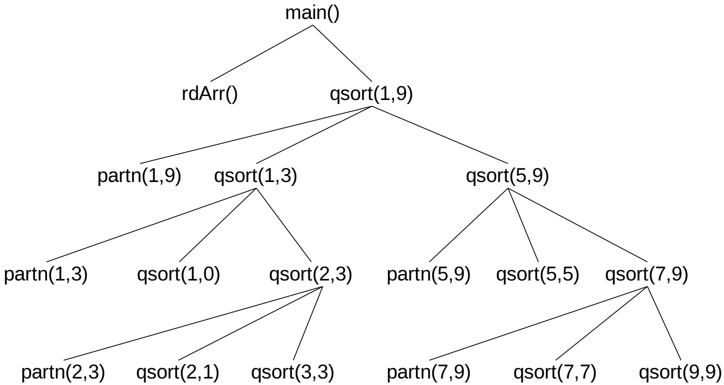
```c
void quicksort(int m, int n) {
  int i;
  if (n > m) {
    i = partition(m, n);
    quicksort(m, i-1);
    quicksort(i+1, n);
  }
}
int partition(int m, int n) {
  ...
}
```

One Possible Activation Tree



a.k.a dynamic call graph

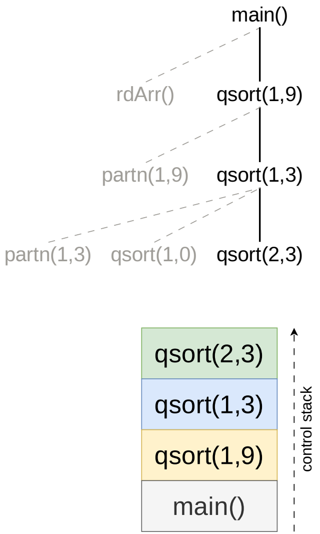
Example of Procedure Activations




```
enter main()
  enter readArray()
  leave readArray()
  enter quicksort(1,9)
    enter partition(1,9)
    leave partition(1,9)
    enter quicksort(1,3)
    ...
    leave quicksort(1,3)
    enter quicksort(5,9)
    ...
    leave quicksort(5,9)
  leave quicksort(1,9)
leave main()
```

Control Stack

- Procedure calls and returns are usually managed by a run-time stack called the **control stack**
- Each live activation has an **activation record** (also called a **frame**) on the control stack
  - Stores control information and data needed to manage the activation
- A frame is pushed when activation begins and popped when activation ends
- Suppose node *n* is at the top of the stack, then the stack contains the nodes along the path from *n* to the root



Is a Stack Sufficient?

When will a control stack work?

- Once a function returns, its activation record cannot be referenced again
- Every activation record has either finished executing or is an ancestor of the current activation record
- We do not need to store old nodes in the activation tree

When will a control stack not work?

- A function's activation record can be referenced after the function returns
- Function closures – procedure and run-time context to define free variables
  - A variable that a procedure refers to and that is declared outside the procedure's own scope is called a **free variable**

Function Closure

Definition

Function closure stores a function together with its execution environment

- The environment maps each free variable to the value or reference that the name was bound to when the closure was created

- Popularly used in languages where functions are first-class objects
  - Functions can be returned as results from higher-order functions or passed as arguments to other function calls

```python
# Python example
def f(x): # returns a closure
  def g(y):
    return x+y
  return g
def h(x): # returns a closure
  return lambda y: x+y
# Assign closures to variables
a = f(1)
b = h(1)
assert a(5) == 6
assert b(5) == 6
# Use closures without binding to
# variables (anonymous)
assert f(1)(5) == 6
assert h(1)(5) == 6
```

Closure (computer programming)

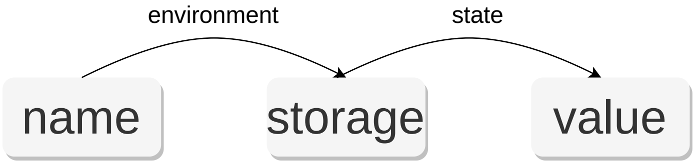
Environment and State

Environment

- Refers to a function that maps a name to a storage location
- Maps a name to an l-value

State

- Refers to a function that maps a storage location to the stored value
- Maps the l-value to an r-value



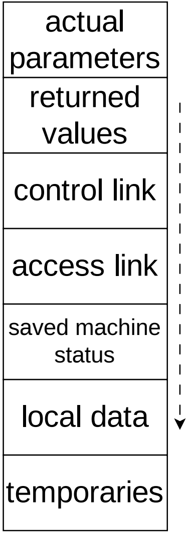
An assignment changes state, not the environment

An expression evaluated to a location is a l-value.

An expression evaluated to a value is a r-value.

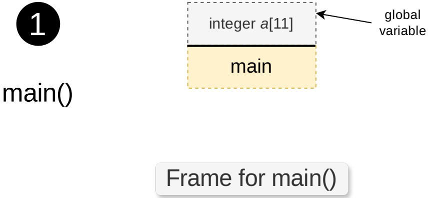
Activation Record

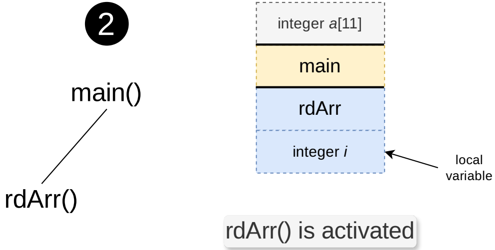
- A pointer to the current activation record is maintained in a register
- Fields in an activation record
  (i) Actual parameters
  (ii) Returned values
  (iii) Control link – Points to the activation record of the caller
  (iv) Access link – access non-local data
  (v) Saved machine status – information about the machine state before the procedure call
    - Return address (value of program counter)
    - Register contents
  (vi) Local data
  (vii) Temporaries

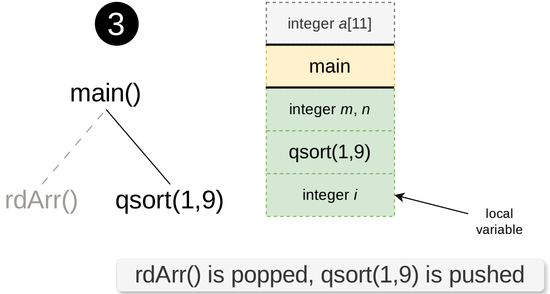


> Contents and position of fields may vary with language and implementations

Sequence of Activation Record Manipulation





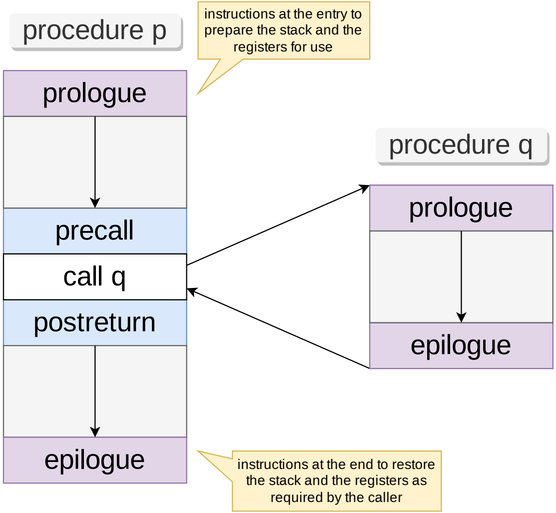


What is in `G()`'s Activation Record when `F()` calls `G()`?

- If a procedure `F` calls `G`, then `G`'s activation record contains information about both `F` and `G`
  - `F` is suspended until `G` completes, at which point `F` resumes
  - `G`'s activation record contains information needed to resume execution of `F`
- `G`'s activation record contains
  - Actual parameters to `G` (supplied by `F`)
  - `G`'s return value (needed by `F`)
  - Space for `G`'s local variables

Procedure Linkage

- Procedure linkage is a contract between the compiler, the OS, and the target machine
- Divides responsibility for naming, allocation of resources, addressability, and protection



Calling and Return Sequence

Calling sequence allocates an activation record on the stack and enters information into its fields

- Responsibility is shared between the caller and the callee

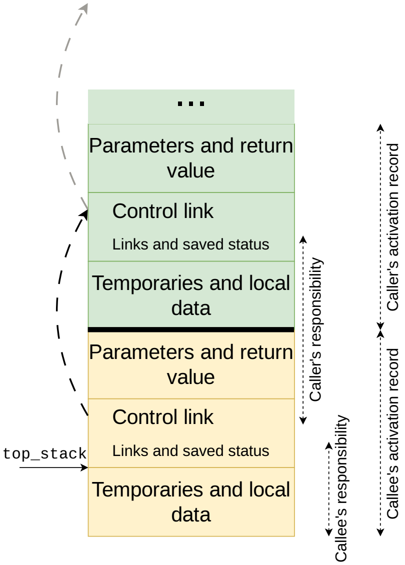
Return sequence restores the state of the machine, so the calling procedure can continue its execution after the call

Calling Sequence

- Place values communicated between caller and callee at the beginning of the callee's activation record, close to the caller's activation record
- Fixed-length items are placed in the middle
- Data items whose size are not known during intermediate code generation are placed at the end of the activation record
- Top-of-stack points to the end of the fixed-length fields
  - Fixed-length data items are accessed by fixed offsets from top-of-stack pointer
  - Variable-length fields records are actually "above" the top-of-stack

Policies and implementation strategies can differ

Division of Tasks Between Caller and Callee



Division of Tasks Between Caller and Callee

Call Sequence

(i) Caller evaluates the actual parameters

(ii) Caller stores a return address and the old value of `top_stack` into the callee's activation record

(iii) Caller then increments `top_stack` past the caller's local data and temporaries and the callee's parameters and status fields

(a) Callee saves the register values and other status information

(b) Callee initializes its local data and begins execution

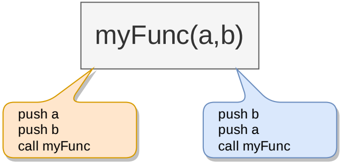
Division of Tasks Between Caller and Callee

Return Sequence

- Callee places the return value next to the parameters
- Callee restores `top_stack` and other registers
- Callee branches to the return address that the caller placed in the status field
- Caller copies return value into its activation record

Calling Conventions

- Specifies how function calls are set up and executed
  - Where are parameters placed? What is the order for passing parameters?
  - How are variadic functions handled?
  - How is the return value passed from the callee to the caller?
  - Which registers should be preserved across calls? Is the caller or the callee responsible for preserving registers?



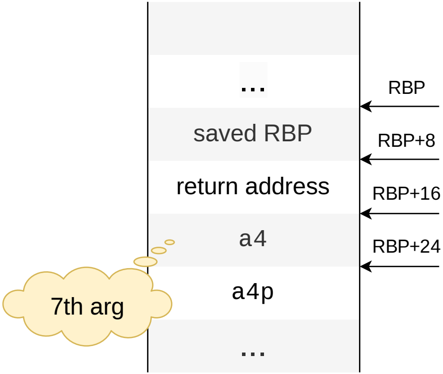
x86-64 calling convention

- First six integral (including pointers) function arguments are passed in registers %rdi, %rsi, %rdx, %rcx, %r8, and %r9
- Subsequent arguments are passed on the stack in the reverse order (arg 7 is at the top)
- The return value is passed in register %rax
- Floating point parameters are passed in %xmm0-%xmm7
- If the function takes a variable number of arguments (like `printf`), then %rax must be set to the number of floating point arguments
- The stack pointer register %rsp must be aligned to 16-byte boundary before the call
- Complete set of rules (System V ABI) are complex

Passing Parameters in x86-64

```
void proc(long a1, long *a1p,
          int a2, int *a2p,
          short a3, short *a3p,
          char a4, char *a4p) {
  *a1p += a1;
  *a2p += a2;
  *a3p += a3;
  *a4p += a4;
}
```

```
$ gcc -S -fno-asynchronous-unwind-tables
-fno-exceptions proc-call.c
  ...
```



x86-64 Assembly for Example Procedure Call

```c
void proc(long a1, long *a1p,
          int a2, int *a2p,
          short a3, short *a3p,
          char a4, char *a4p) {
  *a1p += a1;
  *a2p += a2;
  *a3p += a3;
  *a4p += a4;
}
```

```
$ gcc -Og -S
-fno-asynchronous-unwind-tables
-fno-exceptions proc-call.c
```

```
...
; q is quadword (8B), l is long word
; (4B), and w is word (2B)
; Fetch a4p, move 8 bytes
movq 16(%rsp), %rax
addq %rdi, (%rsi) ; *a1p += a1
addl %edx, (%rcx) ; *a2p += a2
addw %r8w, (%r9) ; *a3p += a3
; Fetch a4 to %dl (low-order 8 bits)
movl 8(%rsp), %edx
addb %dl, (%rax) ; *a4p += a4
ret
...
```

Register Saving Conventions

```
proc1:
  ...
  movq $0x100, %rdx
  call proc2
  addq %rdx, %rax
  ...
  ret
```

```
proc2:
  ...
  subq $0x200, %rdx
  ...
  ret
```

- %rbx, %rbp, and %r12–%r15 are callee-saved registers
- All other registers, excepting %rsp, are caller-saved
- %rax holds the return value, so implicitly caller saved
- %rsp is the stack pointer, so implicitly callee saved

- **Caller** saved
  - Caller saves temporary values in its frame (on the stack) before the call
  - Callee is then free to modify their values
- **Callee** saved
  - Callee saves temporary values in its frame before using them
  - Callee restores them before returning to caller

Use of Callee-Saved Registers

```c
long proc2(long);

long proc1(long x, long y) {
  long u = proc2(y);
  long v = proc2(x);
  return u+v;
}
```

```
$ gcc -O0 -S
-fno-asynchronous-unwind-tables
-fno-exceptions callee-saved-regs.c
```

```
proc1:
  ; x is in %rdi, y is in %rsi
  pushq %rbp ; callee-saved
  movq %rsp, %rbp
  subq $32, %rsp ; allocate memory
  movq %rdi, -24(%rbp)
  movq %rsi, -32(%rbp)
  movq -32(%rbp), %rax
  movq %rax, %rdi
  call proc2@PLT
  movq %rax, -16(%rbp)
  movq -24(%rbp), %rax
  movq %rax, %rdi
  call proc2@PLT
  movq %rax, -8(%rbp)
  movq -16(%rbp), %rdx
  movq -8(%rbp), %rax
  addq %rdx, %rax
  leave
  ret
```

Data Communication between Procedures

Definition

Parameter binding maps the actual parameters at a call site to the callee's formal parameters

- Types of mapping conventions: call by value, call by reference, and call by name

Call by Value

- Convention where the caller evaluates the actual parameters and passes their r-values to the callee
- A formal parameter in the callee is treated like a local name
- Any modification of a value parameter in the callee is not visible in the caller

Call by Reference

- Convention where the compiler passes an address for the formal parameter to the callee
  - Any redefinition of a reference formal parameter is reflected in the corresponding actual
- A formal parameter requires an extra indirection

Call by Name

- Reference to a formal parameter behaves as if the actual parameter had been textually substituted in its place
- Actual parameters are evaluated inside the called function when they are used, not when the function is called
  - Can update the given parameters
  - Renaming is used in case of clashes
- Example: Algol-60

```
procedure double(x);
  real x;
  begin
    x := x*2
  end;
double(c[j])
```

```
int f(int j) {
  int k = j; // k = 9
  i = 2; // modify global i
  k = j; // a[i] is reevaluated, k = 7
}
char array[3] = { 9, 8, 7 };
int i = 0;
f(a[i]);
```

Pass-By-Name Parameter Passing

What is "Call By Name"?

Challenges with Call by Name

```
procedure swap(a, b)
  integer a, b, temp;
  begin
    temp := a
    a := b
    b := temp
 end;
```

What will happen when you call swap(i, x[i])?

```
temp := i
i := x[i]
x[i] := temp
```

Before call	i = 2	x[2] = 5	
After call	i = 5	x[2] = 5	x[5] = 2

Data Access Rules

Name Spaces, and Lexical and Dynamic Scoping

- **Scope** is the part of a program to which a name declaration applies
  - Scope rules provide control over access to data and names
- In **lexical scoping**, a name refers to the definition that is **lexically closest** to the use
  - With lexical (a.k.a., static) scoping, a free variable is bound to the declaration for its name that is lexically closest to the use
- With **dynamic scoping**, a free variable is bound to the variable **most recently created** at run time (e.g., Common Lisp)

> Lexical scoping is more popular, dynamic scoping is relatively challenging to implement
> - Both are identical as far as local variables are concerned

Nested Lexical Scopes in Pascal

```
program Main_0(inp, op);
  var x_1, y_1, z_1: integer;
  procedure Fee_1;
    var x_2: integer;
    begin { Fee_1 }
      x_2 := 1;
      y_1 := x_2*2+1
    end;
  procedure Fie_1;
    var y_2: real;
    procedure Foe_2;
      var z_3: real;
      procedure Fum_3;
        var y_4: real;
        ...
```

- Compilers can use a static coordinate for a name for lexically-scoped languages
- Consider a name x declared in a scope s
- Static coordinate is a pair $\langle l, o \rangle$ where l is the lexical nesting level of s and o is the offset where x is stored in the scope's data area

Scope	x	y	z
Main	$\langle 1,0 \rangle$	$\langle 1,4 \rangle$	$\langle 1,8 \rangle$
Fee	$\langle 2,0 \rangle$	$\langle 1,4 \rangle$	$\langle 1,8 \rangle$
Fie	$\langle 1,0 \rangle$	$\langle 2,0 \rangle$	$\langle 2,8 \rangle$
Foe	$\langle 1,0 \rangle$	$\langle 2,0 \rangle$	$\langle 3,0 \rangle$
Fum	$\langle 1,0 \rangle$	$\langle 4,0 \rangle$	$\langle 3,0 \rangle$

Lexical and Dynamic Scope

```
int x = 1, y = 0;
int g(int z) {
  return x + z;
}          free variable

int f(int y) {
  int x;
  x = y + 1;
  return g(x * y);
}

int main() {
  print(f(3));
}
```

What is printed (i) with lexical scoping and (ii) with dynamic scoping?

Static (Lexical) Scoping vs Dynamic Scoping (Pseudocode)

Lexical and Dynamic Scope in Perl

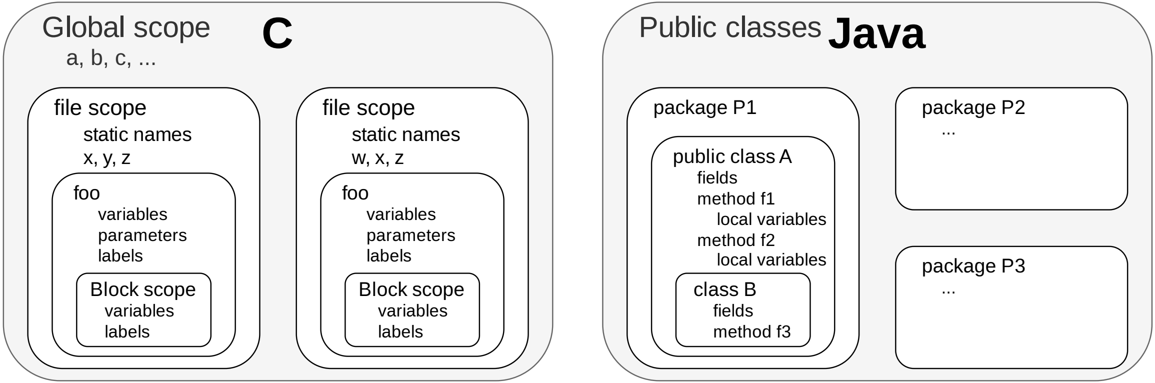
```perl
$x = 10;
sub f
{
  return $x;
}
sub g
{
  # If local is used, x uses
  # dynamic scoping. If my is
  # used, x uses lexical scoping
  local $x = 20;
  return f();
}
print g()."\n";
```

What is printed (i) with lexical scoping and (ii) with dynamic scoping?

```
$ perl scope.pl
```

Static (Lexical) Scoping vs Dynamic Scoping (Pseudocode)

Scoping Rules for C and Java Languages

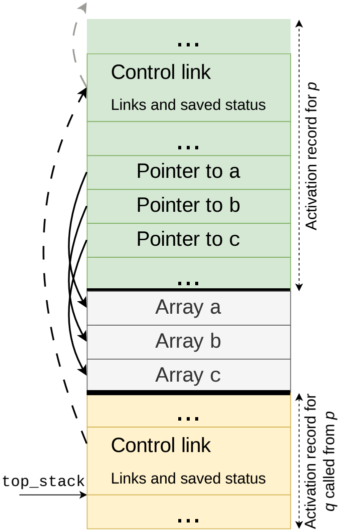


Allocating Activation Records

- Stack allocation
  - Activation records follow LIFO ordering (e.g., Pascal, C, and Java)
- Heap allocation
  - Needed when a procedure can outlive its caller (e.g., implementations of Scheme and ML)
  - Garbage collection support eases complexity
- Static allocation
  - Procedure P cannot have multiple active invocations if it does not call other procedures
  - A **leaf procedure** makes no calls to other procedures
  - Reduces memory requirement and improves performance

Variable Length Data on the Stack

- Data may be local to a procedure but the size may not be known at compile time
  - For example, a local array whose size depends upon a parameter
- Data may be allocated in the heap but may require garbage collection
- Possible to allocate variable-sized local data on the stack



Data Access without Nested Procedures

- Consider the C-family of languages
- Any name local to a procedure is non-local to other procedures
- Access rules
  (i) Global variables are in static storage
    - Addresses are fixed and **known at compile time**, use the addresses in the code

  (ii) Any other name must be **local to the activation** at the top of the stack

Access to Non-local Data in Nested Procedures

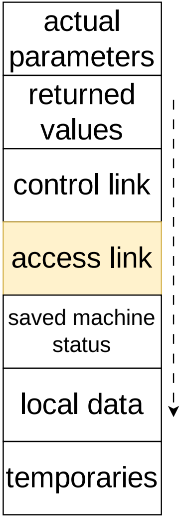
Suppose procedure p at lexical level m is nested in procedure q at level n, and x is declared in q

We aim to resolve a non-local name x in p

- **Finding** the **declaration** for non-local x in p is a **static** decision
- Compiler models the reference by a static distance coordinate $\langle m - n, o \rangle$ where o is offset in the activation record for q
- Compiler needs to translate $\langle m - n, o \rangle$ into a run-time address
- **Finding** the **relevant activation** of q from an activation of p is a **dynamic** decision
- We **cannot** use **compile-time** decisions since there could be many activation records of p and q on the stack
- Two common strategies: access links and displays

Access Links

- Suppose procedure p is nested immediately within procedure q
  - p's nesting depth is $i + 1$ if q's nesting depth is i
  - Procedures not nested within other procedures have nesting depth 1 (e.g., functions in C)
- Access link in any activation of p points to the most recent activation of q
  - Access links form a chain up the nesting hierarchy of activations whose data and procedures are accessible to the currently executing procedure

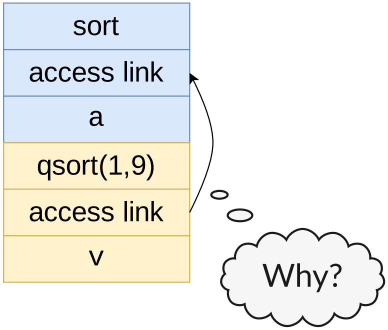


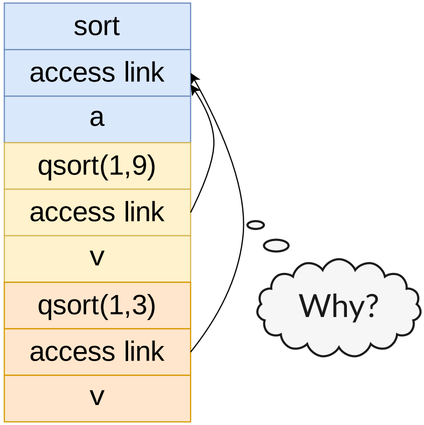
Quicksort in ML using Nested Procedures

```
fun sort (inputFile, outputFile) =
  let
    val a = array(11,0);
    fun readArray(inputFi1e) = .. ;
      ..a.. ; // use of a
    fun exchange(i, j) =
      ..a.. ; // use of a
    fun quicksort(m,n) =
      let
        val v = .. ; // pivot
        fun partition(y,z) =
          ..a..v..exchange.. // use
      in
        ..a..v..partition..quicksort
      end
  in
    ..a..readArray..quicksort..
  end;
```

Procedure	Nesting Depth
sort	1
readArray	2
exchange	2
quicksort	2
partition	3

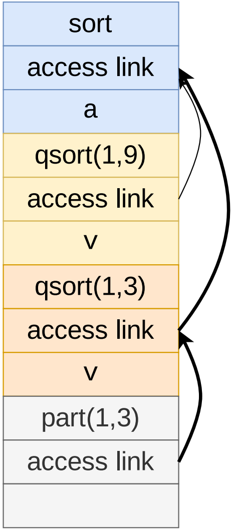
Example of Access Links





How to Find Non-local x?

- Suppose procedure p is at the top of the stack and has depth n_p, and q is a procedure that surrounds p and has depth n_q
  - Usually $n_q < n_p$; $n_q == n_p$ only if p and q are the same
- Follow the access link $(n_p - n_q)$ times to reach an activation record for q
  - That activation record for q will contain a definition for x that is non-local to p



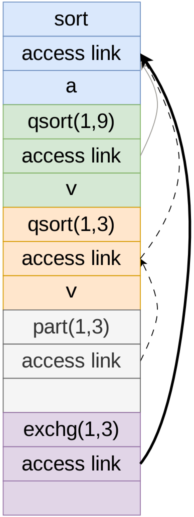
Traversing Access Links

- The code to set up access links is part of the calling sequence and depends upon whether the called procedure is nested within the caller
- Suppose procedure q at depth n_q calls procedure p at depth n_p

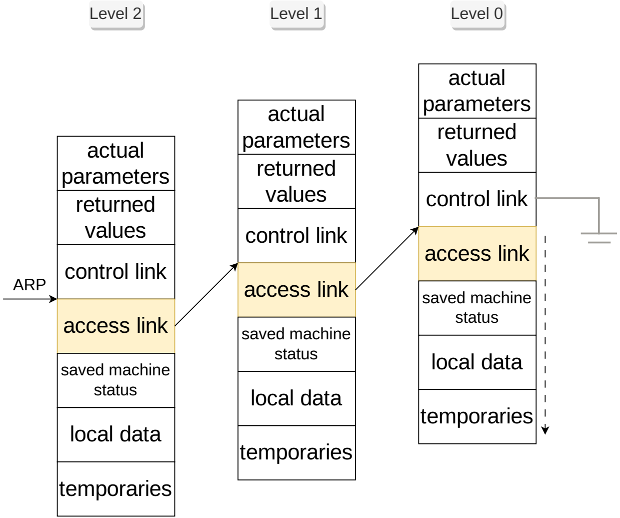
- Case 1: $n_q < n_p$
  - Called procedure p is nested more deeply than q
  - Therefore, p must be declared in q, or the call by q will not be within the scope of p
  - Access link in p should point to the access link of the activation record of the caller q
  - For example, `sort()` calls `quicksort()`, `quicksort()` calls `partition()`
- Case 2: $n_q == n_p$
  - Procedures are at the same nesting level (i.e., recursive call)
  - Access link of called procedure p is the same as q
  - For example, `quicksort(1,9)` calls `quicksort(1,3)`

Traversing Access Links

- Case 3: $n_q > n_p$
  - For example, `partition()` calls `exchange()`
    - Nesting depth of calling function `partition()` is 3
    - Nesting depth of called function `exchange()` is 2
  - For the call within q to be in the scope of p, q must be nested within some procedure r, while p is defined immediately within r
  - Top activation record for r can be found by following chain of access links for $n_q - (n_p - 1)$ hops, starting in the activation record for q
  - Access link for q will go to the activation for r



Traversing Access Links



Coordinate	Code
$\langle 2, 24 \rangle$	`loadAI %r_arp, 24, %r_2`
$\langle 1, 12 \rangle$	`loadAI %r_arp, -4, %r_1` `loadAI %r_1, 12, %r_2`

$\langle 0, 16 \rangle$	`loadAI %r_arp, -4, %r_1`
	`loadAI %r_1, -4, %r_1`
	`loadAI %r_1, 16, %r_2`

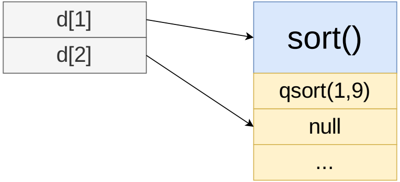
ARP stands for activation record pointer

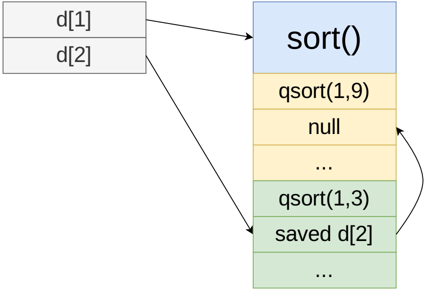
- Assume that the access link is stored at an offset of -4 from the ARP

Displays

Definition

Display is a global array to hold the activation record pointers for the most recent activations of procedures at each lexical level

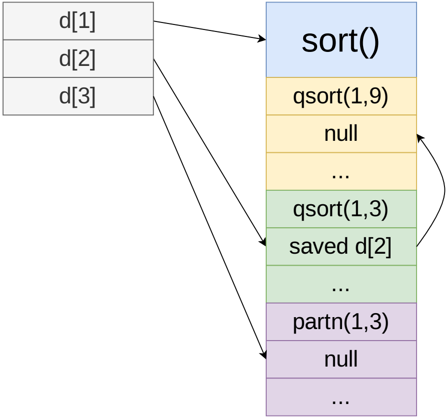


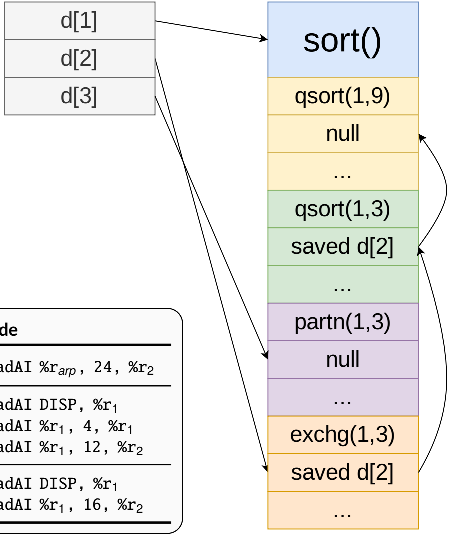


Insight in Using Displays

- Suppose a procedure p is executing and needs to access element x belonging to procedure q
- The runtime **only** needs to search in activations from $d[i]$, where i is the nesting depth of q
- Follow the pointer $d[i]$ to the activation record for q, wherein x should be defined at a known offset

Displays





Coordinate	Code
$\langle 2, 24 \rangle$	`loadAI` $\%r_{arp}$, 24, $\%r_2$
$\langle 1, 12 \rangle$	`loadAI DISP,` $\%r_1$ `loadAI` $\%r_1$, 4, $\%r_1$ `loadAI` $\%r_1$, 12, $\%r_2$
$\langle 0, 16 \rangle$	`loadAI DISP,` $\%r_1$ `loadAI` $\%r_1$, 16, $\%r_2$

Access Links vs Displays

Access Links

- Cost of lookup varies
  - Common case is cheap, but long chains can be costly
- Cost of maintenance is variable

Displays

- Cost of lookup is constant
- Cost of maintenance is constant

References

- A. Aho et al. Compilers: Principles, Techniques, and Tools. Sections 7.1–7.3, 2nd edition, Pearson Education.
- K. Cooper and L. Torczon. Engineering a Compiler. Sections 6.1–6.5, 7.1–7.2, 7.9, 2nd edition, Morgan Kaufmann.