Subroutines & Control Abstractions

Module 3

Static Links

- Each stack frame contains a reference to the frame of lexicallysurrounding subroutine
- This reference is static link

Dynamic Link

- The saved value of frame pointer is called dynamic link

Example

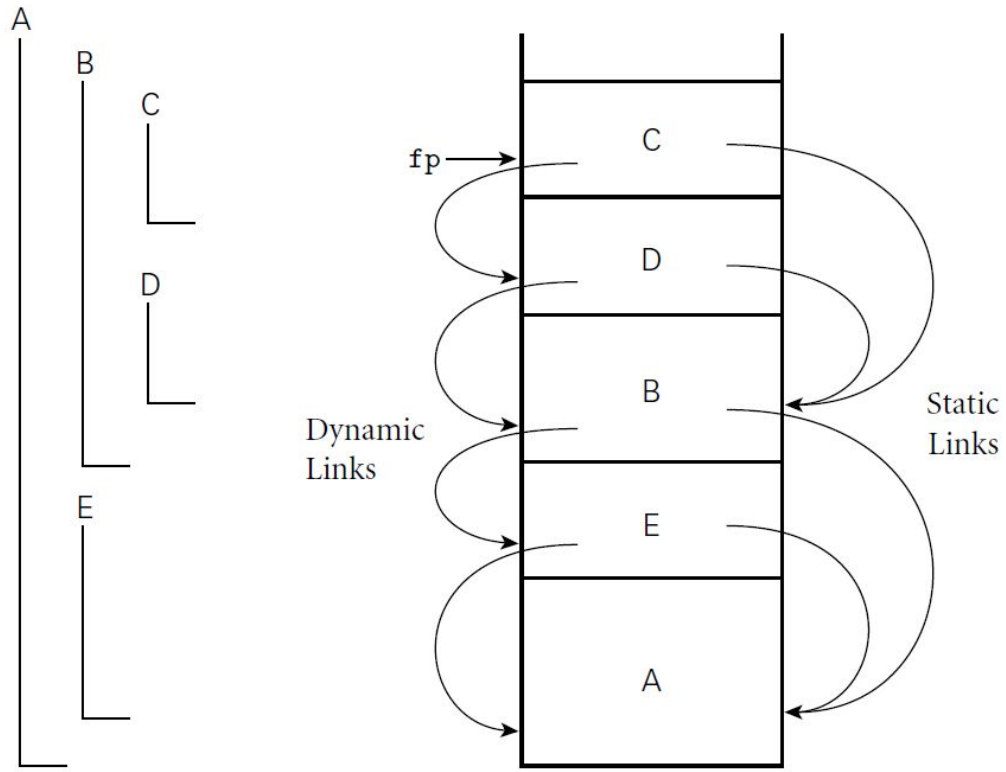


Figure 8.1 Example of subroutine nesting, taken from Figure 3.5. Within B, C, and D, all five routines are visible. Within A and E, routines A, B, and E are visible, but C and D are not. Given the calling sequence A, E, B, D, C, in that order, frames will be allocated on the stack as shown at right, with the indicated static and dynamic links.

Maintaining the Static Chain

- Part of the work to maintain static chain must be done by caller
- The approach is
- If the callee is nested directly inside the caller
  - the callee's static link should refer to the caller's frame.
- If the callee is at the same level of nesting as the caller
  - callees static link is a copy of callers static link
- If callee is k levels higher than caller
  - Callees static link is found by following k static links back from callers static link
  - Copy static link found here

Calling Sequences

- Maintenance of the subroutine call stack done by
- Calling Sequence (Setting up call, making needed data, where to return)
- Prologue (Prepares stack and registers for use)
- Epilogue ((Restores stack and registers)

- Calling sequence is the combined operations of the caller, the prologue, and the epilogue.

Typical Calling Sequence

The caller

1. saves any **caller-saves registers** whose values will be needed after the call

2. computes the values of **arguments** and moves them into the stack or registers

3. computes the **static link** (if this is a language with nested subroutines), and passes it as an extra, hidden argument

4. uses a special subroutine call instruction to **jump to the subroutine**, simultaneously passing the return address on the stack or in a register

In its prologue, the callee

1. allocates a frame by subtracting an appropriate constant from the sp

2. saves the old frame pointer into the stack, and assigns it an appropriate new value

3. saves any callee-saves registers that may be overwritten by the current routine (including the static link and return address, if they were passed in registers)

- After the subroutine has completed, the epilogue

  1. moves the return value (if any) into a register or a reserved location in the stack

  2. restores callee-saves registers if needed

  3. restores the fp and the sp

  4. jumps back to the caller

- Finally, the caller

  1. moves the return value to wherever it is needed

  2. restores caller-saves registers if needed return address

Displays

- access to an object in a scope k levels
- out requires that the static chain be dereferenced k times.

- It needs k + 1 memory accesses

- This number can be reduced to a constant by use of a display.
- An object k levels out can be found by using j=i-k
- j- static link to be found out
- i frame number of the subroutine
- k no: of nesting levels

RegisterWindows

- RISC machines use register window as an alternative to saving and restoring registers on subroutine calls
- Complete set of registers are called register set
- More registers means more delay
- Register file is divided into fixed size windows
- Each register window is assigned to a procedure

- Each window is divided into 3:
- Parameter registers: hold passed parameters and results to be passed
- Local registers: hold local variables
- Temporary registers: hold parameters to exchange

In Line Expansion

- Alternative to stack-based calling conventions
- Subroutines to be expanded in-line at the point of call
- A copy of the “called” routine becomes a part of the “caller”

- Avoids overheads like:

1. space allocation
2. Branch delays from the call and return
3. Maintaining the static chain or display
4. Saving and restoring registers

Special-Purpose Parameters

Conformant Arrays

- Size is not known till run time

- Example:

function DotProduct(A,B: array[lower..upper:integer]of real]:real

Default (Optional) Parameters

- one that need not necessarily be provided by the caller
- if the value of a variable is missing, then a preestablished default value will be used instead.

- Here default_width uses width(default value ofbuilt in type)

```
type field is integer range 0..integer'last;
type number_base is integer range 2..16;
default_width : field := integer'width;
default_base : number_base := 10;
procedure put(item : in integer;
width : in field := default_width;
base : in number_base := default_base);
```

Named Parameters

- parameters are positional
- the first actual parameter corresponds to the first formal parameter, the second actual to the second formal, and so on

- Ada, Common Lisp, Fortran 90,Modula-3, and Python allow parameters to be named
- also called keyword parameters

Example in Ada

- put(item => 37, base => 8);
- Because the parameters are named, their order does not matter
- put(base => 8, item => 37);
- We can even mix the two approaches

- put(37, base => 8);
- Positional arguments cannot follow named arguments. It throws compilation error
- Named arguments can follow positional arguments.

Variable Numbers of Arguments

- Lisp, Python, and C and its descendants allow subroutines that take a variable number of arguments

```
int printf(char *format, ...)
{
 ...
}
```

- The ellipsis (...) in the function header is a part of the language syntax. It indicates that there are additional parameters following the format

- Their type and numbers unspecified

Steps to use variable arg list in C/C++

- Define a function with its last parameter as ellipses and the one just before the ellipses is always an int which will represent the number of arguments.
- Create a va_list type variable in the function definition. This type is defined in stdarg.h header file.
- Use int parameter and va_start macro to initialize the va_list variable to an argument list. The macro va_start is defined in stdarg.h header file.
- Use va_arg macro and va_list variable to access each item in argument list.
- Use a macro va_end to clean up the memory assigned to va_list variable.

```
#include <stdarg.h> /* macros and type definitions */
int printf(char *format, ...)
{
    va_list args;
    va_start(args, format);
    ...
    char cp = va_arg(args, char);
    ...
    double dp = va_arg(args, double);
    ...
    va_end(args);
}
```

Function Returns

- Algol,Fortran,Pascal □ return expression
- Return marks the termination of function

rtn:=expression

…

Return rtn

Disadvantage of Local Return Variable

- Because rtn is a local variable, most compilers will allocate it within the stack frame of the function

- The return statement must then perform an unnecessary copy to move that variable's value into the return location allocated by the caller.

Example in SR

- Some languages eliminate the need for a local variable
- by allowing the result of a function to have a name in its own right.

```
procedure A_max(ref A[1:*]: int) returns rtn : int
rtn := low(int)
fa i := 1 to ub(A) ->
if A[i] > rtn -> rtn := A[i] fi
af
End
```

No local copy of rtn will kept in stack.

Python – Multivalue Returns

```
def foo():
return 2, 3
...
i, j = foo()
```

Parameter Passing Modes

- Most subroutines are parameterized:

formal parameters – Parameters in subroutine declaration

Actual parameters: Parameters in subroutine call

Parameter Modes

- C, Fortran, ML, and Lisp follow a single rule to all parameters

- Pascal, Modula, and Ada, provide 2 or more rules based on modes
- Let x be a global variable to be passed to procedure P
- We have 2 alternatives:
- Provide P with a copy of x's value
- Provide P with x's address

Parameter Passing Modes

- Call By Value
- Call By Reference

```
x : integer – – global
procedure foo(y : integer)
y := 3
print x
. . .
end foo
x := 2
foo(x)
print x
```

Call by value : 2 printed twice

Call by reference: 3 printed twice

Call by Value /Result

- copies the actual parameter into the formal parameter at the beginning of subroutine execution

- copies the formal parameter back into the actual parameter when the subroutine returns

```
x : integer – – global
procedure foo(y : integer)
y := 3
print x
. . .
end foo

x := 2
foo(x)
print x
```

Call by name/result: 2 printed first then 3 printed

Call-by-Sharing

- Most people know only
  - Call-by-value
  - call-by-reference
  - But most common evaluation strategy is not both of them.
  - Languages like javascript,java,Python,Ruby etc use call by sharing evaluation strategy
  - we pass the reference and let the actual and formal parameters to share the same object

Call-by-Sharing

- Here caller of the function shares objects with the function being called
- The called function can mutate those objects so long as the objects themselves are mutable
- Because the objects are shared ,any changes made will be reflected in the calling function
- This convention is applied to all objects being passed
- For immutable objects call by sharing is indistinguishable from call by value

Read-Only Parameters

- Available in Modula3
- Formal parameter declared Readonly cannot be changed by called routine
- Similar to const in C

- Pascal
- Call by value - default
- Call by Reference is used if preceeded by var in formal parameter

- C
- Normally call by value is used

- Fortran
- Use Call by Reference
- no call by value support

Parameter Modes in Ada

- 3 parameter-passing modes
- in, out, and in out

- In
- Parameters pass information from the caller to the callee They can be read by the callee but not written.

- Out
  - Parameters pass information from the callee to the caller.
  - Ada 83 they can be written by the callee but not read
  - In Ada 95 they can be both read and written
- In out
  - They can be both read and written.
  - Changes to out or in out parameters always change the actual parameter

Implementation

- in --> call-by-value
- in out --> call-by-value/result
- out --> simply call-by-result

References in C++

- Specified by preceeding parameter name with &
- C lacks reference variables
- Modify object by passing its address
  - Formal parameter is a pointer
  - Need dereferencing

- C++ introduce reference

```
int i;
int &j = i;
...
i = 2;
j = 3;
cout << i; // prints 3
```

- No dereferencing requiured

Closures as Parameters

- CLosure=(Proc_address,envt)
- Proc_addr --> reference to procedure code
- envt -- > pointer to environment of definition of procedure

```
function foo(x)
    function price(y)
        return x+y
      return price;
variable closure1= foo(1)
variable closure2= foo(5)
closure1(3)  --- prints 4
closure2(3) --- prints 8
```

Call-by-Name (Algol 60 and Simula)

- Re-evaluates actual parameters on every use
- Arguments that are simple variables-- same as call by reference
- Arguments that are expression - re-evaluated on each access

- Call By Need - (Miranda,Haskel,R)
- Memorized version of call by name

```
begin
  integer n;
  Procedure P(K:integer)
    begin
          print(n);
           n:=n+1;`
           print(K);
end
n:=0;
P(n+10);
end;
```

Call by name 10 11

Generic Subroutines and Modules

- Subroutines are used to perform an operation for a variety of different object types
- An operating system tends to make heavy use of queues, to hold processes, memory descriptors, file buffers, device control blocks, and a host of other objects.
- The characteristics of the queue data structure are independent of the characteristics of the items placed in the queue

Problem

- Mechanisms for declaring enqueue and dequeue in most languages require that the type of the items be declared, statically

- Pascal or Fortran
- -static declaration means
- Programmer must create separate copies of enqueue and dequeue for every type item

- C

- Define a queue of pointers to arbitrary objects
- But use of such a queue requires type casts that abandon compile time checking

Implicit parametric polymorphism

- Provides a solution
- - Allows to declare subroutines whose parameter types are incompletely defined
- **Disadvantage**
- delays type checking until run time
- makes the compiler slower and complicated

- Another alternative is to provide explicitly polymorphic generic facility
- Supported in Ada, C++,Clu,Eiffel,Modula3,Java,C#

- Generic modules or classes are useful for creating containers
- Generic subroutines(methods) are needed in generic modules(classes)

Generic Parameters

- What can be passed as parameters depends on language
- Java and c# allows to pass generic types
- Ada and C++ allows ordinary types

```
generic
type item is private;
-- can be assigned; other characteristics are hidden
max_items : in integer := 100; -- 100 items max by default
package queue is
procedure enqueue(it : in item);
function dequeue return item;
private
subtype index is integer range 1..max_items;
items : array(index) of item;
next_free, next_full : index := 1;
end queue;
```

```
package body queue is
procedure enqueue(it : in item) is
begin
items(next_free) := it;
next_free := next_free mod max_items + 1;
end enqueue;
function dequeue return item is
rtn : item := items(next_full);
begin
next_full := next_full mod max_items + 1;
return rtn;
end dequeue;
end queue;
```

```
package ready_list is new queue(process);
-- assume type process has previously been declared


package int_queue is new queue(integer, 50);
-- only 50 items long, instead of the default 100
```

C++

```
template<class item, int max_items = 100>
class queue {
item items[max_items];
int next_free;
int next_full;
public:
queue() {
next_free = next_full = 0; // initialization
}
```

```
void enqueue(item it) {
items[next_free] = it;
next_free = (next_free + 1) % max_items;
}
item dequeue() {
item rtn = items[next_full];
next_full = (next_full + 1) % max_items;
return rtn;
}
};
```

- queue<process> ready_list;
- queue<int, 50> int_queue;

Implementation Options

- Generics implemented in many ways
- Ada,C++ -> Static mechanism
- All the work required to create and use multiple instances of the generic code takes place at compile time.
- Compiler creates a seperate copy of the code for every instance

- Java5--> all instances of a given generic will share the same code at run time

Generic Parameter Constraints

- Because a generic is an abstraction,
- its interface must provide all the information needed by a user
- Several languages, including Clu, Ada, Java, and C#, attempt to enforce this rule by constraining generic parameters

- Ada: type item is private;

Generic Parameter Constraints

- Private type
- permissible operations : assignment,testing for equality and inequality, and accessing a few standard attributes(e.g., size).

- To prohibit testing for equality and inequality,
- the programmer can declare the parameter to be limited private.

- type item is (>);

- supports comparison for ordering (<, >, etc.) and the attributes first and last

Generic Sorting in Ada

- Ada - specify the operations of a generic type using with clause

```
generic
type T is private;
type T_array is array (integer range <>) of T;
with function "<"(a1, a2 : T) return boolean;
procedure sort(A : in out T_array);
```

- Without the with clause, procedure sort would be unable to compare elements of A for ordering, because type T is private.

Generic Sorting in Java

```
public static <T extends Comparable<T>> void sort(T A[]) {
...
if (A[i].compareTo(A[j]) >= 0) ...
...
}
...
Integer[] myArray = new Integer[50];
...
sort(myArray);
```

- Comparable is an interface
- This method returns −1, 0, or 1, respectively, depending on whether the current object is less than, equal to, or greater than the object passed as a parameter

Generic Sorting in C#

```
static void sort<T>(T[] A) where T : IComparable {
...
if (A[i].CompareTo(A[j]) >= 0) ...
...
}
...
int[] myArray = new int[50];
sort(myArray);
```

Generic Sorting in C++

- C++ can be extremely simple

```
template<class T>
void sort(T A[], int A_size) { ...
```

Implicit Instantiation

- Because a class is a type, we must create an instance of a generic class to use it

- queue<int, 50> *my_queue = new queue<int, 50>();

- Some languages require generic subroutines to be instantiated explicitly.

- Eg: In Ada

```
procedure int_sort is new sort(integer, int_array, "<");
...
int_sort(my_array);
```

- C++, Java, and C# donot require this

- Given the c++ sorting routine and the following objects:
- int ints[10];
- double reals[50];
- string strings[30];

- We can perform following calls without instantiating anything explicitly
- sort(ints, 10);
- sort(reals, 50);
- sort(strings, 30);

Coroutines

Defenition

- Coroutines are general control structures where
- flow control is cooperatively passed between two different routines without returning

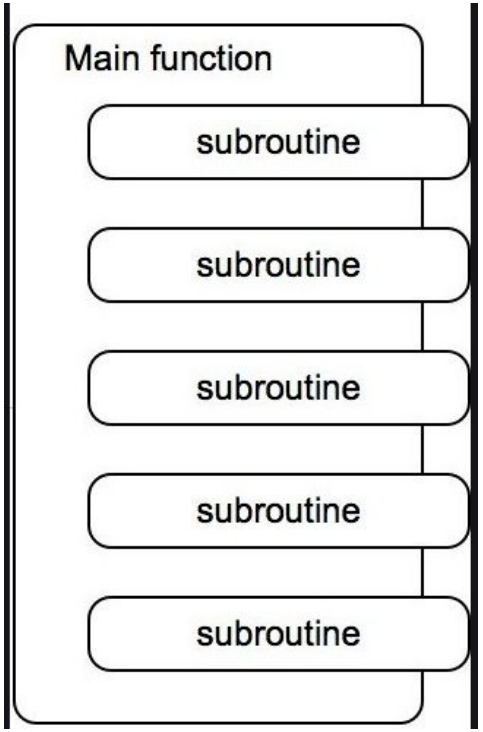
- Coroutine is represented by a closure
- A code address
- A reference environment

- Coroutines are execution contexts that executes concurrently

- They execute one at a time and transfer control to each others explicitly

- Useful for implementing iteerartors and threads

Subroutines are called by a main function that is responsible for coordinating the use of several subroutines



Working

Example

- A “screen-saver” program that
- paints a black picture on the screen of an inactive workstation
- keeps the picture moving
- Performs “sanity checks” on the file system in the background, looking for corrupted files.

- Program
- Loop
- – – update picture on screen
- – – perform next sanity check

- successive sanity checks depend on each other.
- The file-system checking codecontain nested loops.
- Inorder to break it into pieces that can be interleaved with the screen updates, the programmer must save state of nested computation

Alternative

```
us, cfs : coroutine

coroutine check_file_system
        -- initialize
        detach
        for all files
                ...
                transfer(us)
                ...
                        transfer(us)
                ...
                transfer(us)
                ...
```

```
coroutine update_screen
        -- initialize
        detach
        loop
                ...
                transfer(cfs)
                ...

begin           -- main
        us := new update_screen
        cfs := new check_file_system
        transfer(us)
```

- When first created
  - Coroutine performs initialisation operations
- Detach from main which
  - Creates a coroutine object to transfer control
- Returns a reference of this coroutine to caller
- Transfer Operation
  - Saves current pc
  - Resumes coroutine specified as parameter
- Main- default coroutine

Stack Allocation

- Since coroutines are concurrent
- cannot share a single stack
- Solution is to give each coroutine a statically allocated stack space
- Followed in Modula2
- Requires programmer to give size and location of stack when initialisezing a coroutine
- More stack space needed - error
- Less stack space - wastage
- Another solution, stack frames are allocated from heap
- Followed in Lisp and Scheme
- Problems of overflow and internal fragmenttaion

Stack Allocation

- Since coroutines are concurrent
  - Cannot share a single stack
- Solution is to give each coroutine a statically allocated stack space.
- Followed in Modula-2
- Requires the programmer to give the size and location of the stack when initializing a coroutine.

- More stack space needed - error
- Less stack space - wastage

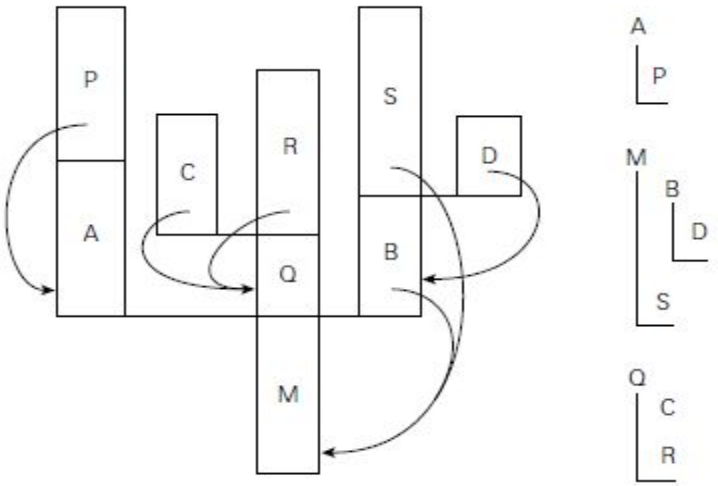
- Another solution, stack frames are allocated from heap

- Followed in LISP and Scheme
- Problems of overflow and internal fragmentation are avoided
- Overhead of each subroutine call is increased

- It is a run-time error for the coroutine to need additional space.
- Some Modula-2 implementations catch the overflow and halt with an error message;
- others display abnormal behavior.
- If the coroutine uses less space than it is given, the excess is
- simply wasted.

- An intermediate option
  - Allocate the stack in large, fixed-size “chunks.”
  - Each subroutine call checks to see whether
  - there is sufficient space in the current chunk to hold the frame
  - If not, another chunk is allocated
  - At each return, epilog clears the chunk if it had the last frame
- It can use the stack if the compiler ensures that the subroutine will not perform a transfer before returning
- If coroutines can be created at arbitrary levels of nesting
- 2 or more coroutines can share access to objects in that scope
- For supporting sharing option a cactus tree is used

- Each branch off the stack contains the frames of a separate coroutine



Each branch to side represents creation of a coroutine(A,B,C,and D)
Static links are shown as arrows
Dynamic links are shown by vertical arrangement

Transfer

- To transfer from one coroutine to another
  - Run-time system must change the program counter (PC),stack and register contents
- These changes are encapsulated in transfer operation
- One coroutine calls transfer
- Different one returns
- Changing PC from one coroutine to another needs to remember return address
  - Old coroutine calls transfer one location
  - New coroutine returns to different location
- How to change stacks?
  - Approach is to change stack pointer register
  - Avoid using frame pointer inside transfer

- Transfer
- Push return address and callee-saves registers onto current stack
- change sp
- Pop new return address and registers of new stack
- return

```
transfer:
push all registers other than sp (including ra)
*current_coroutine := sp
Current_coroutine := r1 – – argument passed to transfer
sp := *r1
pop all registers other than sp (including return address)
return
```

- Data structure representing coroutine is a context block
- Context block contains a single value
- Coroutine's sp of recent transfer
- Transfer expects a pointer to context block as argument
- By dereferencing the pointer it can get the sp of next coroutine to run