LECTURE NOTES

On

CS 403 PROGRAMMING PARADIGMS

B.Tech CSE VII SEMESTER

(KTU)

Ms. Anju Kuriakose, M.Tech

Assistant Professsor

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

(NBA Accredited)

Syllabus

Module I

Names, Scopes and Bindings:- Names and Scopes, Binding Time, Scope Rules, Storage Management, Binding of Referencing Environments.

Control Flow:- Expression Evaluation, Structured and Unstructured Flow, Sequencing, Selection, Iteration, Recursion, Non-determinacy.

Module II

Data Types:-Type Systems, Type Checking, Records and Variants, Arrays, Strings, Sets, Pointers and Recursive Types, Lists, Files and Input/Output, Equality Testing and Assignment.

Module III

Subroutines and Control Abstraction: - Static and Dynamic Links, Calling Sequences, Parameter Passing, Generic Subroutines and Modules, Exception Handling, Co-routines.

Module IV

Functional and Logic Languages:- Lambda Calculus, Overview of Scheme, Strictness and Lazy Evaluation, Streams and Monads, Higher-Order Functions, Logic Programming in Prolog, Limitations of Logic Programming.

Module V

Data Abstraction and Object Orientation:-Encapsulation, Inheritance, Constructors and Destructors, Aliasing, Overloading, Polymorphism, Dynamic Method Binding, Multiple Inheritance. Innovative features of Scripting Languages:-Scoping rules, String and Pattern Manipulation, Data Types, Object Orientation.

Module VI

Concurrency:- Threads, Synchronization. Run-time program Management:- Virtual Machines, Late Binding of Machine Code, Reflection, Symbolic Debugging, Performance Analysis.

MODULE III

Subroutines and Control Abstraction: - Static and Dynamic Links, Calling Sequences, Parameter Passing, Generic Subroutines and Modules, Exception Handling, Co-routines.

Subroutines and Control abstraction

- **Abstraction** is a process by which the programmer can associate a name with a potentially complicated program fragment, which can then be thought of in terms of its purpose or function, rather than in terms of its implementation.
- Two types of abstraction:
  1. **Data abstraction**: The principal purpose of the abstraction is to represent information. Eg: Class
  2. **Control abstraction**: The principal purpose of the abstraction is to perform a well defined operation. Eg: Subroutines
- Subroutines are *control abstractions*: they allow the programmer to hide arbitrarily complicated code behind a simple interface.
- Subroutines are the basis of *procedural abstraction*: the abstraction of computations.
- Each subprogram has one entry point. The calling program (caller) is suspended during execution of the called subprogram (callee). Control returns to the caller when subprogram is terminated.

  Two types of subroutines:
  1. **Function**: a subroutine that returns a value.
  2. **Procedure**: a subroutine that does not return a value.

Static and dynamic links

Stack

- Storage consumed by parameters and local variables can be allocated on a stack.
- **Stack frame** or **Activation record** contains arguments and/or return values, bookkeeping information, local variables, and/or temporaries. On return, stack frame is popped from stack.

- **Stack Pointer(SP)**: contains address of either the first unused location at the top of the stack or the last used location.
- **Frame Pointer(FP)**: contains an address within frame. Objects in the frame are accessed via displacement addressing with respect to the frame pointer.
- Displacement addressing: Objects in the frame are accessed via offset from the frame pointer.
- Variable size things are towards top of the frame, since can't be preset, with dope in fixed size part.
- If no variable-size objects, then everything has fixed offset from stack pointer.
  - In this case, frame pointer isn't needed at all
  - Frees up a register for use.

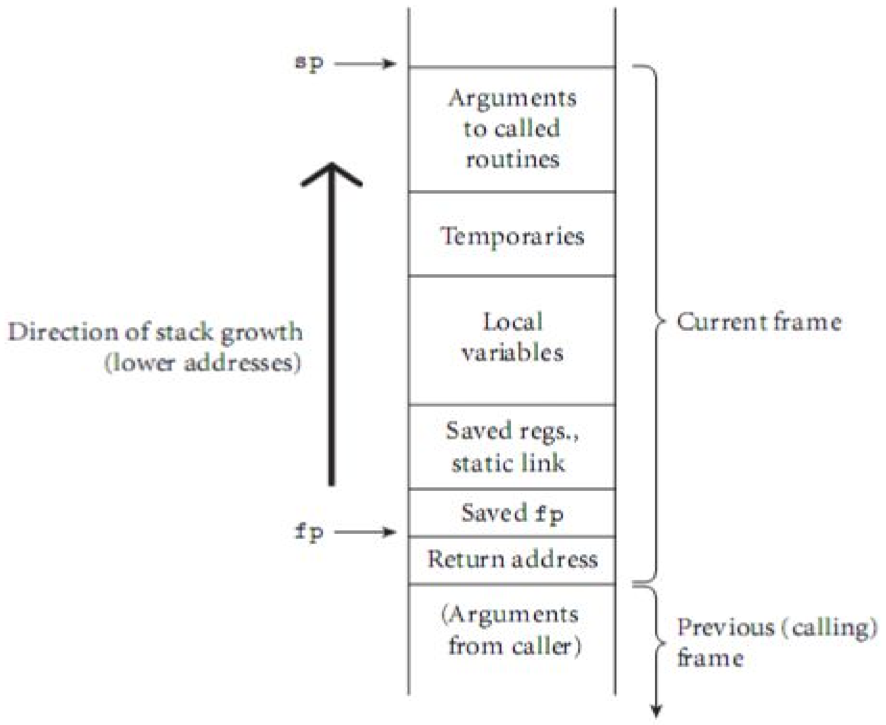


Figure 8.2 A typical stack frame. Though we draw it growing upward on the page, the stack actually grows downward toward lower addresses on most machines. Arguments are accessed at positive offsets from the fp. Local variables and temporaries are accessed at negative offsets from the fp. Arguments to be passed to called routines are assembled at the top of the frame, using positive offsets from the sp.

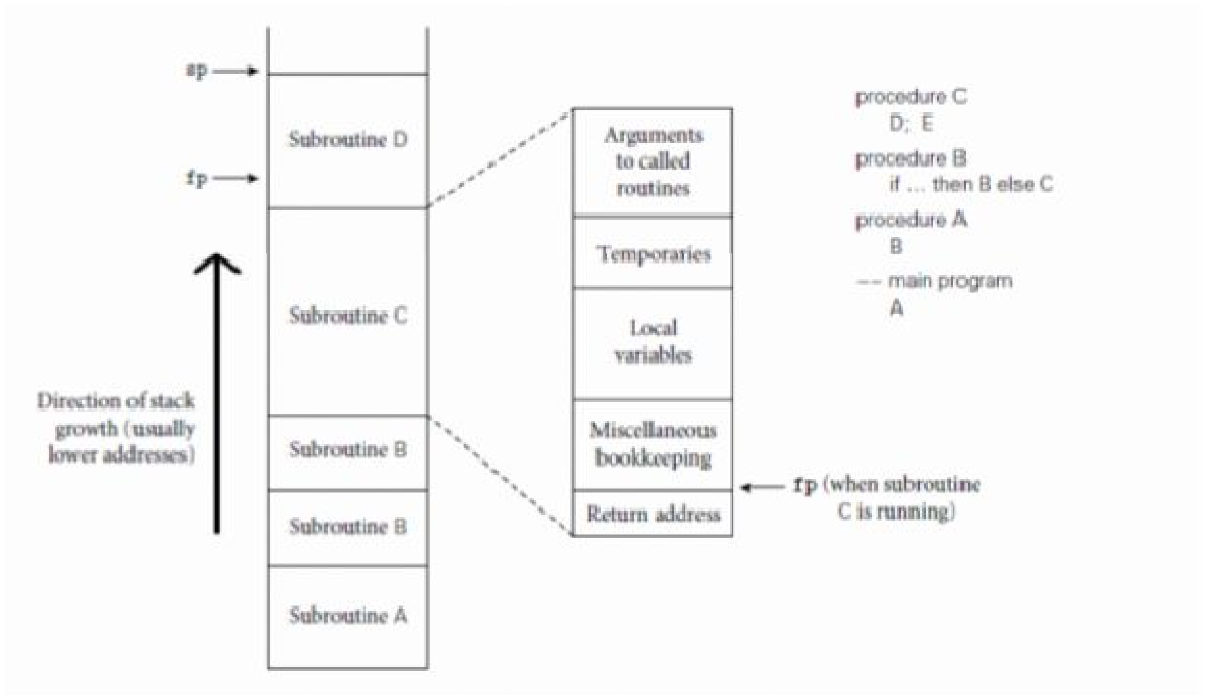


Figure 3.1 **Stack-based allocation of space for subroutines.** We assume here that subroutines have been called as shown in the upper right. In particular, B has called itself once, recursively, before calling C. If D returns and C calls E, E's frame (activation record) will occupy the same space previously used for D's frame. At any given time, the stack pointer (sp) register points to the first unused location on the stack (or the last used location on some machines), and the frame pointer (fp) register points to a known location within the frame of the current subroutine. The relative order of fields within a frame may vary from machine to machine and compiler to compiler.

Static and dynamic link

- In a language with nested subroutines and static scoping (Pascal, Ada, or Scheme), objects that lie in the surrounding subroutines and that are thus neither local nor global, can be found by maintaining a static chain. Each stack frame contains a reference to the frame of the lexically surrounded subroutine. This reference is called **static link.** The saved value of the frame pointer which will be restored on subroutine return is called **dynamic link**.
- Static chain: Each stack frame contains a reference to the lexically surrounding subroutine.
- Static and dynamic links maintained on stack:
  - Static link store context for scope purposes .
  - Dynamic link store saved value of the frame pointer, so that it returns to correct calling method.

- ➢ Dynamic links allow one to walk back the frame pointer linearly down the call stack.

➢ Static links allow one to walk back the frame pointer from a lexical viewpoint.

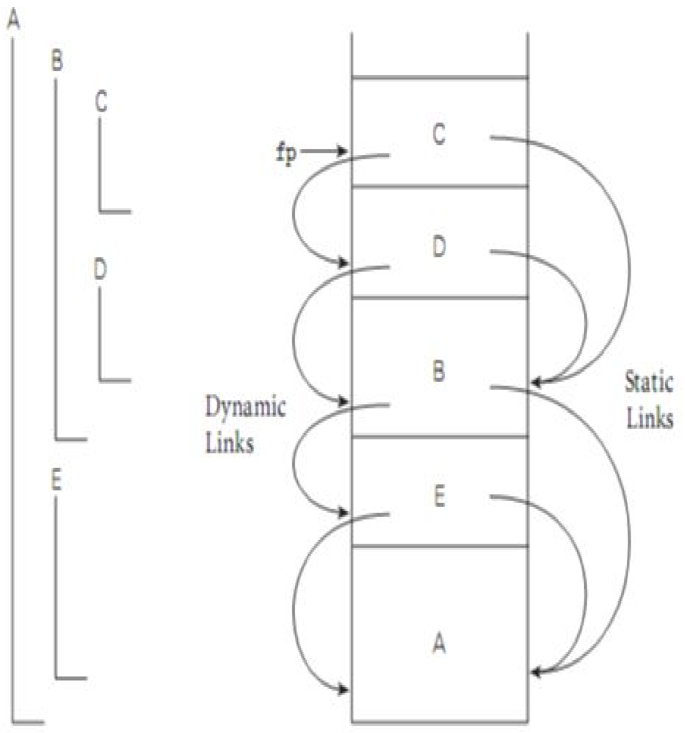


Figure 8.1 **Example of subroutine nesting, taken from Figure 3.5.** Within B, C, and D, all five routines are visible. Within A and E, routines A, B, and E are visible, but C and D are not. Given the calling sequence A, E, B, D, C, in that order, frames will be allocated on the stack as shown at right,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

Calling sequences

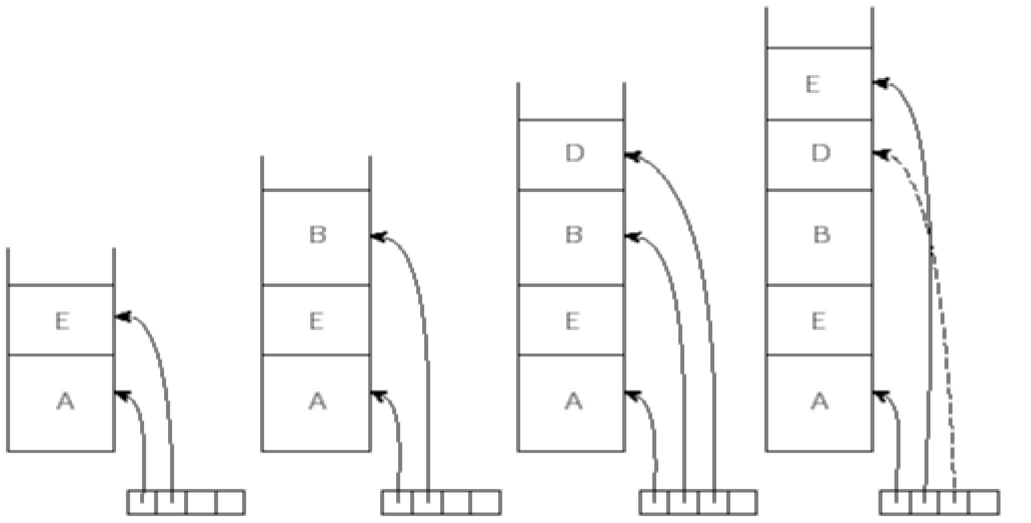
- ➢ The calling sequence is the code a caller executes to set up a new subroutine.
- ➢ Code executed by the caller immediately before and after a subroutine call.
- ➢ Prologue(code executed at the beginning) and Epilogue(code executed at the end)
  - o Tasks on the way in (**Prologue**): Pass parameters, save return address, change program counter, change stack pointer, save registers that might be overwritten, changing frame pointer to new frame, and initializing code for new objects in the new frame.

- Tasks on the way out (**Epilogue**): passing return parameters/values, executing finalization code for local objects, deallocating the stack frame, restoring registers, and restoring the PC.

The **calling sequence** might operate as follows:

- The caller
  - Saves any caller saves registers whose values will be needed after the call.
  - Computes the values of arguments and moves them into the stack or registers.
  - Computes the static link (if this is a language with nested s ubroutines) and passes it as an extra hidden arguments.
  - Uses a special subroutine call instruction to jump to the subroutine, simultaneously passing the return address on the stack or in a register.
- In prolog, Callee
  - Allocates a frame by subtracting an appropriate constant from sp.
  - saves the old frame pointer into the stack and assigns it an appropriate new value.
  - Saves any callee save registers that may be overwritten by the current routine.
- In epilog, Callee
  - puts return value into registers or a reserved location in the stack.
  - Restores calle save registers if needed.
  - Restores the fp and sp
  - Jumps back to the return address.
- Finally the caller,
  - Moves the return value to wherever it is needed.
  - Restores caller saves registers if needed.

Display: used to reduce memory accesses to an object k levels out.



Inline expansion

- ➢ In some languages, programmers can actually flag routines that should be expanded inline –stack overhead is avoided.
- ➢ Allows certain subroutines to be expanded in-line at the point of call.
- ➢ Avoids several overheads (space allocation, branch delays, maintaining static chain or display, saving and restoring registers)
- ➢ Example (in C++ or C99):

```
inline int max(int a,int b)
{ return a > b ? a : b}
```

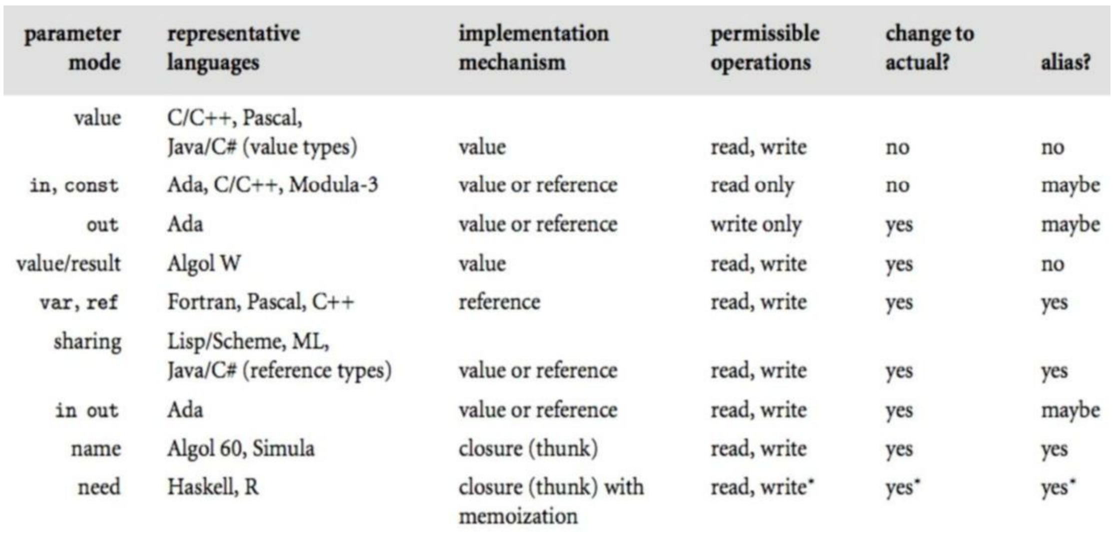
- ➢ Ada does something similar, but keyword is:

```
pragma inline(max);
```

- ➢ Note that both of these are just suggestions to the compiler – can't be enforced, and may be done to other (not suggested) functions as well.
- ➢ Note also that this can't always be done – recursion is an obvious case with issues.
- ➢ Inline expansion :Semantically preferable
- ➢ Macros: Semantic and syntactic problems

Parameter passing

- Parameter passing mechanisms have three basic implementations
  - o *value*
  - o *reference*(aliasing)
  - o *closure/name*
- Many languages (e.g., Pascal, Ada, Modula) provide different modes
- Others have a single set of rules (e.g. C, Fortran, ML, Lisp)
- Parameters - arguments that control certain aspects of a subroutine's behavior or specify the data on which they are to operate.
- Parameters increase the level of abstraction.



parameter mode	representative languages	implementation mechanism	permissible operations	change to actual?	alias?
value	C/C++, Pascal, Java/C# (value types)	value	read, write	no	no
in, const	Ada, C/C++, Modula-3	value or reference	read only	no	maybe
out	Ada	value or reference	write only	yes	maybe
value/result	Algol W	value	read, write	yes	no
var, ref	Fortran, Pascal, C++	reference	read, write	yes	yes
sharing	Lisp/Scheme, ML, Java/C# (reference types)	value or reference	read, write	yes	yes
in out	Ada	value or reference	read, write	yes	maybe
name	Algol 60, Simula	closure (thunk)	read, write	yes	yes
need	Haskell, R	closure (thunk) with memoization	read, write*	yes*	yes*

Figure 8.3 Parameter passing modes.

<u>Parameter passing methods</u>

- Parameter passing methods are the ways in which parameters are transmitted to and/or from called subprograms. Parameter names that appear in the declaration of a subroutine are known as **formal parameters.** Variables and expressions that are passed to a subroutine in a particular call are known as **actual parameters**.

```
type small = 1..100;
var  a : 1..10;
     b : 1..1000;
procedure foo (var n : small);
begin
    n := 100;
    writeln (a);
end;
...
a := 2;
foo (b);     (* ok *)
foo (a);     (* static semantic error *)
```

➢ Formal parameters are characterized by one of three distinct semantic models:

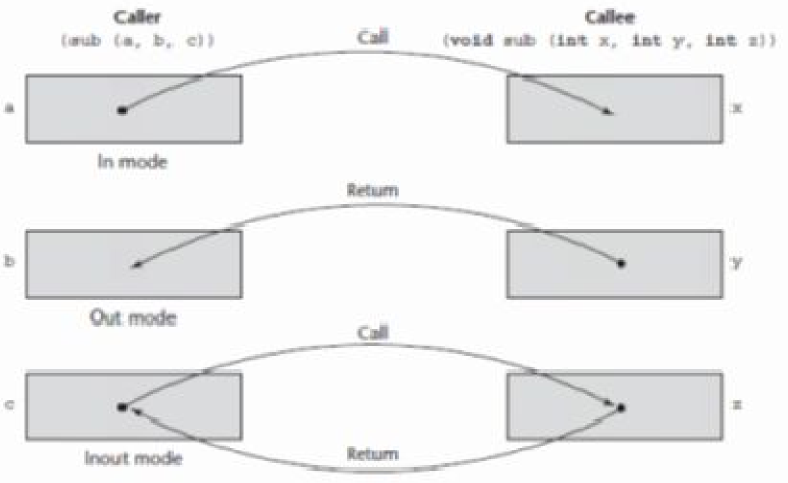
1) They can receive data from the corresponding actual parameter(in mode).

2) They can transmit data to the actual parameter(out mode).

3) They can do both(inout mode).

Figure 9.1

The three semantics models of parameter passing when physical moves are used



Implementation models of parameter passing

1.pass-by-value

2.pass-by-result

3.pass-by-value-result

4.pass-by-reference

5.pass-by-name

1. **Pass-by-Value**
    - ➢ The value of the actual parameter is used to initialize the corresponding formal parameter.
    - ➢ In-mode semantics.
    - ➢ Normally implemented by copy.
2. **Pass-by-Result**
    - ➢ Implementation model for out-mode parameters.
    - ➢ No value is transmitted to the subprogram.
3. **Pass-by-Value-Result**
    - ➢ Implementation model for inout-mode parameters in which actual values are copied.
    - ➢ A combination of pass-by-value and pass-by-result.
    - ➢ The value of the actual parameter is used to initialize the corresponding formal parameter.
    - ➢ At subprogram termination, the value of the formal parameter is transmitted back to the actual parameter.
    - ➢ Sometimes called pass-by-copy.
    - ➢ Disadvantage: multiple storage for parameters, the time for copying values, order in which actual parameters are assigned.
4. **Pass-by-Reference**
    - ➢ Second implementation model for inout-mode parameters.
    - ➢ Transmits an access path, usually an address to the called subprogram.
5. **Pass-by-Name**
    - ➢ Inout-mode parameter transmission method that does not correspond to a single implementation model.
    - ➢ Actual parameter is textually substituted for the corresponding formal parameter in all its occurrences in the subprogram.
    - ➢ It is used at compile time by the macros in assembly languages.

Call by value and call by reference

Suppose x is a global variable in a language with a value model of variables, and we wish to pass x as a parameter to subroutine p:

p(x);

From an implementation point of view, we have two principal alternatives: we may provide p with a copy of x's value (call by value) or we may provide it with x's address(call by reference).

In **call by value**, actual parameter is assigned to the corresponding formal parameter when subroutine is called, and the two are independent from then on. It is like creating a local or temporary variable.

In **call by reference**, the formal parameter refers to the same object as the actual parameter, so that changes made by one can be seen by the other

Language specific variations

- Pascal: Call by value is the default, the keyword VAR denotes call by reference.
- Fortran: all parameters passed by reference.
- Smalltalk, Lisp: Actual Parameter is already a reference to the object.
- C: always passed by value.

Call by value vs Call by reference

- Pass by Value
  - Called routine cannot modify the Actual Parameter.
- Pass by Reference
  - Called routine can modify Actual Parameter.

Call by name

- Pretty much only in Algol.
- Re-evaluates the actual parameter on every use
  - For actual parameters that are **simple variables**, it's the same as call by reference.
  - For actual parameters that are **expressions**, the expression is re - evaluated on each access.
- No other language ever used call by name…

Safety and efficiency

- Without language support, working with large objectsthat are not to be modified can be tricky.
  - Call by Value
  - Call by Reference
- Examples of Language Support
  - Modula: READONLY parameter mode
  - C/C++: const keyword

Ada parameter modes

Three parameter passing modes:

- In
  - Passes information from the caller to the callee, can read but not write .
  - Call by Value
- Out
  - Passes information from the callee to the caller, can write but not read .
  - Call by Result (formal parameter is copied to actual parameter when subroutine exits)
- Inout - passes information in both directions.

C++ parameter modes

➢ C passes pointers as addresses, must be explicitly dereferenced when used.

➢ C++ has notion of references:

Parameter passing:

void swap (int &a, int &b)

Variable References: int &j = i;

➢ Function Returns: for objects that don't support copy operations, i.e. file buffers

Closures

➢ Implementation of Deep Binding for a Subroutine.

- Create an explicit representation of the current referencing environment and its bindings.
- Bundle this representation with a reference to the subroutine
- This bundle is called a Closure.
- Closure is also known as thunk.

Closures as parameters

- Closures: a reference to a subroutine and its referencing environment
- Closures can be passed as a parameter
- Eg: Pascal, C, C++, Modula, Scheme

Special-Purpose Parameters

- Named Parameters - parameters that are not positional, also called keyword parameters. An Ada example:
  - funcB (argA => 21, argB => 35);
  - funcB (argB => 35, argA => 21);
- Some languages allow subroutines with a variable number of arguments: C, Lisp, etc.
  - int printf (char *format, ...) {...
- Standard Macros in function body to access extra variables.

Function Returns

- Some languages restrict Return types
  - Algol 60, Fortran: scalars only
  - Pascal, Modula: scalars or pointers only
  - Most imperative languages are flexible
- Return statements specify a value and also cause the immediate termination of the subroutine

Generic subroutines and modules

Generic subroutines

- Supports explicit parametric polymorphism.
- Subroutines provide a natural way to perform an operation for a variety of different object (parameter) values.
- In large programs, the need also often arises to perform an operation for a variety of different object types. An operating system, for example, 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. Unfortunately, the standard

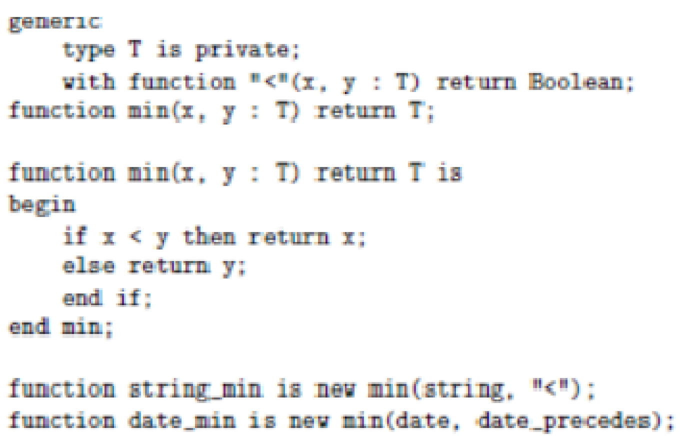
mechanisms for declaring enqueue and dequeue subroutines in most languages require that the type of the items be declared, statically.

- ➢ Polymorphic subroutines
  - Argument types are incompletely specified
  - Can cause slower compilation
  - Sacrifice compile-time type checking
- ➢ In a language like Pascal or Fortran, this static declaration of item type means that the programmer must create separate copies of enqueue and dequeue for every type of item, even though the entire text of these copies.

Generic modules or classes

- ➢ Similar subroutines are created from a single piece of source code
- ➢ Ada, Clu, Modula-3
- ➢ C++ templates
- ➢ Similar to macros, but are actually integrated into the rest of the language
- ➢ Follow scope, naming, and type rules
- ➢ Generic modules or classes are particularly valuable for creating containers: data abstractions that hold a collection of objects, but whose operations are generally oblivious to the type of those objects.
- ➢ Examples of containers include stack, queue, heap, set, and dictionary (mapping) abstractions, implemented as lists, arrays, trees, or hash tables.
- ➢ Generic subroutines (methods) are needed in generic modules (classes), and may also be useful in their own right.

  For example, a min function that works on anything.



```
generic
    type T is private;
    with function "<"(x, y : T) return Boolean;
function min(x, y : T) return T;

function min(x, y : T) return T is
begin
    if x < y then return x;
    else return y;
    end if;
end min;

function string_min is new min(string, "<");
function date_min is new min(date, date_precedes);
```

Figure 3.13 Use of a generic subroutine in Ada.

Various implementations

- C++ and Ada make them static, so takes place at compile time, and makes separate copies for all possibilities.
- Java 5, in contrast, guarantees that all instances will share the same code at run time – objects of class T are treated as instances of the standard base Object that Java supports.

Exception handling

- Exceptions are an unexpected or unusual condition that arises during program execution.
- Exception is a hardware-detected run-time error or unusual condition detected by software.
- It may be detected automatically by the language implementation, or the program may raise it explicitly.
- Most common are various sorts of run time errors.
- Examples
  1. **arithmetic overflow**
  2. **end-of-file on input** (Encountering the end of a file before reading a requested value)
  3. **wrong type for input data**
  4. **user-defined conditions, not necessarily errors**
- Generally, we need the language to "unwind the stack" when exceptions happen.
- Basically, 3 options:
  1. Invent a value that can be used by caller when the real one can't be returned. (not good in general)
  2. Return an explicit status value to caller, which must be inspected after every call. (clunky and annoying)
  3. Rely on the caller to pass a closure for error handling. (clutters and only works in more functional languages)

Handling exceptions

- Exception handling was pioneered by PL/I.
- Utilized an executable statement of the following form:

- ON *condition statement*

- ➢ Handler is nested inside and is not executed on the ON statement but is remembered for future reference.
- ➢ Executes exception when exception condition is encountered.
- ➢ Recent languages provide exception-handling facilities where handlers are lexically bound to blocks of code.
- ➢ General rule is if an exception isn't handled in the current subroutine, then the subroutine returns and exception is raised at the point of call.
- ➢ Keeps propagating up dynamic chain until exception is handled.
- ➢ If not handled, a predefined outermost handler is invoked which will terminate the program.

Exception handler

- ➢ code executed when exception occurs
- ➢ may need a different handler for each type of exception

Why design in exception handling facilities?

- ➢ allow user to explicitly handle errors in a uniform manner.
- ➢ allow user to handle errors without having to check these conditions
- ➢ explicitly in the program everywhere they might occur

Uses of handler

1. Perform some operation that allows the program to recover from the exception and continue executing.
2. If recovery isn't possible, handler can print helpful message before termination.
3. When exception occurs in block of code but can't be handled locally, it is important to declare local handler to clean up resources and then re-raise the exception to propagate back up.

Defining exceptions

- ➢ Ada: exception is a built in type, an exception is simply an object of this type.
  - o declare empty_queue : exception;
- ➢ Modula-3
  - o EXCEPTION empty_queue;

- C++ and Java: an exception is an instance of some predefined or user defined class type.
    - class empty_queue{ };

Exception Propagation

```
try{
. . .
//protected block of code
. . .
}catch(end_of_file) {
. . .
}catch(io_error e) {
//handler for any io_error other than end_of_file
. . .
}catch(...) {
//handler for any exception not previously named
//... is a valid token in the case in C++, doesn't
//mean code has been left out.
}
```

- When an exception arises, the handlers are examined in order.
- Control is transferred to the first one that matches the exception.

Implementing exceptions

- Can be made as a linked list stack of handlers.
- When control enters a protected block, handler for that block is added to head of list.
- Propagation down the dynamic chain is done by a handler in the subroutine that performs the work of the subroutine's epilogue code and then reraises the exception.

Problems with this implementation

- Incurs run-time overhead in the common case.
- Every protected block and every subroutine begins with code to push a handler onto the handler list, and ends with code to pop it off the list.

A better implementation

Since blocks of code in machines can translate to continuous blocks of machine instructions, a table can be generated at compile time that captures the correspondence between blocks and handlers.

Starting Address of Code Block	Address of Corresponding Handler

Implementing exceptions

- ➢ Table is sorted by the first field of the table.
- ➢ When an exception occurs, the system performs a binary search of the table to find the handler for the current block.
- ➢ If handler re-raises the exception, the process repeats.

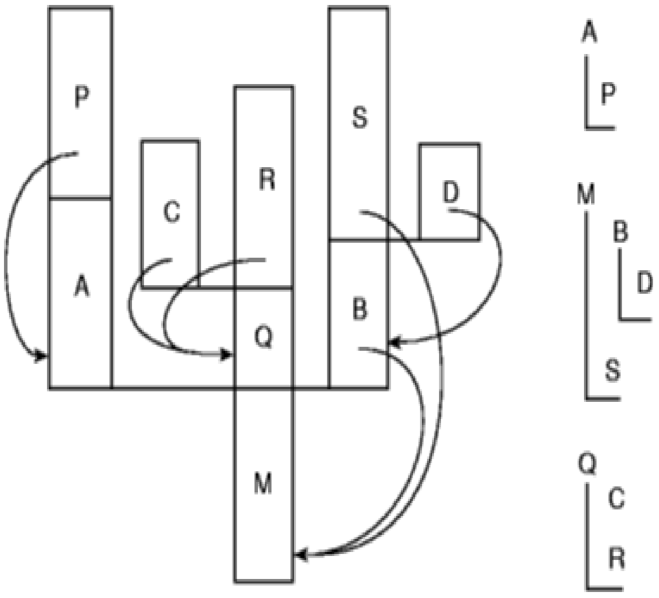
Coroutines

- ➢ Coroutine: a general control abstraction.
- ➢ Special kind of subprogram.
- ➢ A coroutine is represented by a closure(a code address and a referencing environment).

Conventional subprograms	Coroutines
Master-slave relationship between a caller and a called subprogram.	More equal basis, symmetric unit control model
Single entry point	Multiple entry points
Invocation : call	Invocation : resume

- ➢ Coroutines are execution contexts that exist concurrently, but that execute one at a time, and that transfer control to each other explicitly, by name.
- ➢ Coroutines are trivial in assembly language.

- Coroutines can be used to implement
    - iterators
    - threads
- Since they are concurrent (i.e., simultaneously started but not completed) they can't share a single stack because subroutine calls and returns aren't LIFO.
- Instead the run-time system uses a *cactus stack* to allow sharing.



- To go from one co-routine to another, the run-time system must change the PC, stack, and register contents. This is handled in the transfer operation.
- When first created, a coroutine performs any necessary initialization operations, and then detaches itself from the main program.
- The *detach* operation creates a coroutine object to which control can later be transferred and returns a reference to this coroutine to the caller.
- The *transfer* operation saves the current program counter in the current coroutine object and resumes the coroutine specified as a parameter. The main body of the program plays the role of an initial default coroutine.

Transfer

At the beginning of transfer, push the return address and all the other callee save registers onto the current stack. Then change the stack pointer(sp), pop the new return address(ra), and other registers off the new stack and 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 ra)
    return
```

The data structure that represents a coroutine or thread is called a *context block*.

END