

## 6. Names, Scopes, and Bindings

### Overview

- Names
- Binding time
- Object lifetime
- Object storage management
  - Static allocation
  - Stack allocation
  - Heap allocation
- Scope rules
- Static and dynamic scoping
- Reference environments
- Overloading

Note: Study Chapter 3 of the textbook, except Sections 3.3.3, 3.3.4, and 3.6. We encourage you to study Section 3.6 of the textbook, but you are not required to do so.

### Names and Abstractions: What's in a Name?

- *Names* enable programmers to refer to *variables*, *constants*, *operations*, and *types* using identifier names rather than low-level hardware components
- *Names* also form *control* and *data abstractions* for complicated program fragments (control) and data structures
- *Control abstraction*:
  - *Subroutines* (*procedures* and *functions*) allow programmers to focus on manageable subset of program text
  - Subroutine interface hides implementation details, e.g. `sort(MyArray)`
- *Data abstraction*:
  - Object-oriented *classes* hide data representation details behind a set of operations
- *Abstraction* in the context of *high-level programming languages* refers to the degree or level of language features
  - Level of machine-independence
  - "Power" of constructs

### Binding Time

- A *binding* is an association between a name and the thing that is named
- *Binding time* is the time at which an *implementation decision* is made to create a binding
  1. *Language design time*: the design of specific program constructs (syntax), primitive types, and meaning (semantics)
  2. *Language implementation time*: fixation of implementation constants such as numeric precision, run-time memory sizes, max identifier name length, number and types of built-in exceptions, etc.
  3. *Program writing time*: the programmer's choice of algorithms and data structures
  4. *Compile time*: the time of translation of high-level constructs to machine code and choice of memory layout for objects
  5. *Link time*: the time at which multiple object codes (machine code files) and libraries are combined into one executable
  6. *Load time*: the time at which the operating system loads the executable in memory
  7. *Run time*: the time during which a program executes (runs)

### Binding Time Examples

<i>Language feature</i>	<i>Binding time</i>
Syntax, e.g. <code>if (a&gt;0) b:=a;</code> in C or <code>if a&gt;0 then b:=a end if</code> in Ada	Language design
Keywords, e.g. <code>class</code> in C++ and Java	Language design
Reserved words, e.g. <code>main</code> in C and <code>writeln</code> in Pascal	Language design
Meaning of operators, e.g. <code>+</code> (add)	Language design
Primitive types, e.g. <code>float</code> and <code>struct</code> in C	Language design
Internal representation of literals, e.g. <code>3.1</code> and <code>"foo bar"</code>	Language implementation
The specific type of a variable in a C or Pascal declaration	Compile time
Storage allocation method for a variable	Language design, language implementation, and/or compile time
Linking calls to static library routines, e.g. <code>printf</code> in C	Linker
Merging multiple object codes into one executable	Linker
Loading executable in memory and adjusting absolute addresses	Loader (OS)
Nonstatic allocation of space for variable	Run time

## The Effect of Binding Time

- *Early binding times* (before run time) are associated with greater efficiency
  - Compilers try to fix decisions that can be taken at compile time to avoid to generate code that makes a decision at run time
  - Syntax and static semantics checking is performed only once at compile time and does not impose any run-time overheads
- *Late binding times* (at run time) are associated with greater flexibility
  - Interpreters allow programs to be extended at run time
  - Languages such as Smalltalk-80 with *polymorphic* types allow variable names to refer to objects of multiple types at run time
  - *Method binding* in object-oriented languages must be late to support *dynamic binding*

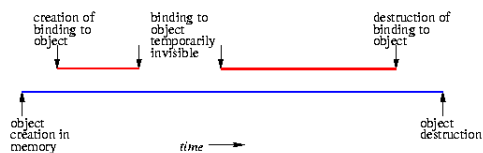
## Object Lifetime

- Key events in object lifetime
  - Object creation
  - Creation of bindings
  - References to variables, subroutines, types are made using bindings
  - Deactivation and reactivation of temporarily unusable bindings
  - Destruction of bindings
  - Destruction of objects
- *Binding lifetime*: time between creation and destruction of binding to object
  - E.g. a Java reference variable is assigned the address of an object
  - E.g. a function's formal argument is bound to an actual argument (object)
- *Object lifetime*: time between creation and destruction of an object

## Object Lifetime Example

- Example C++ fragment:

```
{ SomeClass& myobject = *new SomeClass;
...
{ OtherClass& myobject = *new OtherClass;
... myobject // is bound to other object
...
}
... myobject // is visible again
...
delete myobject;
}
```

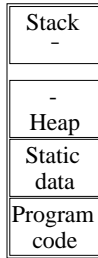


## Object Storage Management

- An object has to be stored in memory during its lifetime
- *Static objects* have an absolute storage address that is retained throughout the execution of the program
  - Global variables
  - Subroutine code
  - Class method code
- *Stack objects* are allocated in last-in first-out order, usually in conjunction with subroutine calls and returns
  - Actual arguments of a subroutine
  - Local variables of a subroutine
- *Heap objects* may be allocated and deallocated at arbitrary times, but require an expensive storage management algorithm
  - E.g. Java class instances are always stored on the heap

## Typical Program and Data Layout in Memory

- Program code is at the bottom of the memory region (code section)
- Static data objects are stored in static region (data section)
- Stack grows downward (data section)
- Heap grows upward (data section)



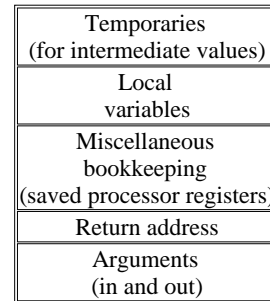
- The code section is protected from run-time modification

## Static Allocation

- Program code is statically allocated in most implementations of imperative languages
- Statically allocated variables are *history sensitive*
  - Global variables
  - `static` local variables in C function retain value even after function returns
- Advantage of statically allocated object is the *fast* access due to absolute addressing of the object
- Static allocation does not work for local variables in potentialy

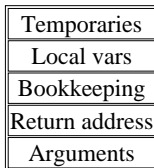
recursive subroutines

- Every (recursive) subroutine call must have separate instantiations of local variables
- Fortran 77 has no recursion
  - Both global and local variables can be statically allocated as decided by compiler
  - Avoids overhead of creation and destruction of local objects for every subroutine call
  - Typical static subroutine data memory layout:



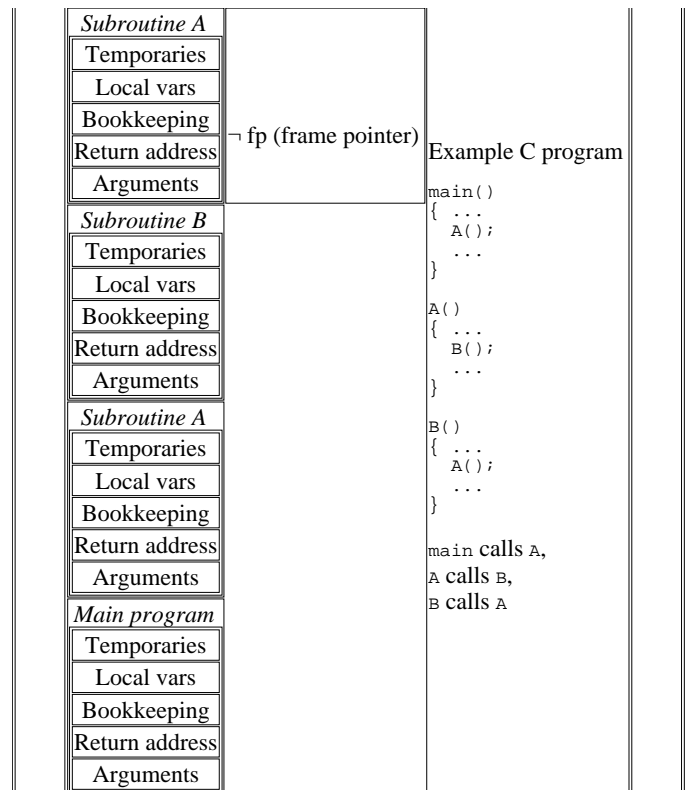
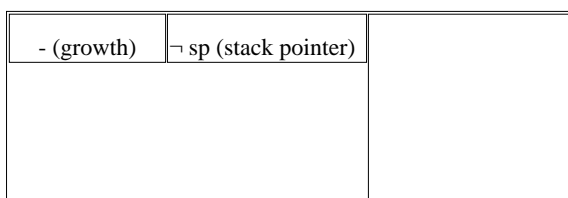
## Stack-Based Allocation

- Each instance of a subroutine at run time has a *frame* on the run-time stack (also called *activation record*)
  - Compiler generates subroutine *calling sequence* to setup frame, call the routine, and to destroy the frame afterwards
  - Subroutine *prologue* and *epilogue* code operate and maintain the frame
- Frame layouts vary between languages and implementations
- Typical frame layout:



- A *frame pointer* (fp) points to the frame of the currently active subroutine at run time (always topmost frame on stack)
- Subroutine arguments, local variables, and return values are accessed by constant address offsets from fp
- The stack pointer (sp) points to free space on the stack

## Stack-Based Allocation Example



## Example Frame

- The word sizes of the types of local variables and arguments determines the fp offset in a frame

Temporaries			<p>Example Pascal procedure</p> <pre> <b>procedure</b> P(a:<b>integer</b>, var b:<b>real</b>)   (* a is passed by value    b is passed by reference,    which is a pointer to b's value    *)   var     foo:<b>integer</b>; (* integer: 2 words *)     bar:<b>real</b>;   (* float: 4 words *)     p:^<b>integer</b>; (* pointer: 2 words *)   <b>begin</b>     ...   <b>end</b> </pre>
Local vars			
var	offset	size	
foo	-18	2 words	
bar	-16	4 words	
p	-12	2 words	
Bookkeeping (offset = -10, 8 words)			
Return address (offset = -2, 2 words) ¬ fp points here			
Arguments			
arg	offset	size	
a	0	2 words	
b	2	2 words	

- The compiler determines the slots for the local variables and arguments in a frame
- The fp of the previous active frame is saved in the current frame and restored after the call

## Heap-Based Allocation

- Implicit** heap allocation:
  - Java class instances are always placed on the heap
  - Scripting languages and functional languages make extensive use of the heap for storing objects
  - Some procedural languages allow array declarations with run-time dependent array size
  - Resizable character strings
- Explicit** heap allocation:
  - Statements and/or functions for allocation and deallocation
- Heap allocation is performed by searching heap for available free space

Object A	free (4 words)	Object B	Object C	free (12 words)	Object D	free (10 words)
----------	----------------	----------	----------	-----------------	----------	-----------------

- Request allocation for object E of 10 words:

Object E (10 words)

- Deletion of objects leaves free blocks in the heap that can be reused
- Internal heap fragmentation:** If allocated object is smaller than the free block the extra space is wasted
- External heap fragmentation:** Smaller free blocks cannot always be reused resulting in wasted space

## Heap Allocation Algorithms

- Maintain a linked list of free heap blocks
- First-fit:** select the first block that is large enough on the list of free heap blocks
- Best-fit:** search entire list for the smallest free block that is large enough to hold the object
- If an object is smaller than the block, the extra space can be added to the list of free blocks
- When a block is freed, adjacent free blocks are coalesced
- Buddy system:** maintain heap pools of standard sized blocks of size  $2^k$ 
  - If no free block is available for object of size between  $2^{k-1}+1$  and  $2^k$  then find block of size  $2^{k+1}$  and split it in half, adding the halves to the pool of free  $2^k$  blocks, etc.
- Fibonacci heap:** maintain heap pools of standard size blocks according to Fibonacci numbers
  - More complex but leads to slower internal fragmentation

## Garbage Collection

- Explicit manual deallocation errors are among the most expensive and hard to detect problems in real-world applications
  - If an object is deallocated too soon, a reference to the object becomes a *dangling reference*
  - If an object is never deallocated, the program *leaks memory*
- Automatic garbage collection** removes all objects from the heap that are not accessible, i.e. are not referenced
  - Used in Lisp, Scheme, Prolog, Ada, Java, Haskell
  - Disadvantage is GC overhead, but GC algorithm efficiency has been improved
  - Not always suitable for real-time programming

## Storage Allocation Mechanisms Compared

Static	Stack	Heap
--------	-------	------

Ada	N/A	local variables of fixed size and subroutine arguments	implicit: local variables of variable size; explicit: <b>new</b> (destruction with garbage collection or explicit with <b>unchecked deallocation</b> )
C	global variables; <b>static</b> local variables, e.g. f() { <b>static</b> int n; ...	local variables and subroutines arguments	explicit with <b>malloc</b> and <b>free</b> functions
C++	global variables; <b>static</b> members	local variables and subroutine arguments	explicit: <b>new</b> and <b>delete</b>
Java	N/A	local variables with primitive types (e.g. <b>int</b> and <b>char</b> )	implicit: all class instances (destruction with garbage collection)

Fortran77	global variables (in common blocks), local variables, and subroutine arguments (implementation dependent); <b>SAVE</b> forces static allocation	local variables and subroutine arguments (implementation dependent)	N/A
Pascal	global variables (dependent on compiler)	global variables (dependent on compiler), local variables, and subroutine arguments	explicit: <b>new</b> and <b>dispose</b>

## Scope

- *Scope*: the textual region of a program in which a name-to-object binding is active
- *Statically scoped language*: the scope of bindings is determined at compile time
  - Used by almost all but a few programming languages
  - More intuitive to user compared to dynamic scoping
- *Dynamically scoped language*: the scope of bindings is determined at run time
  - Used in Lisp (early versions), APL, Snobol, and Perl

## Static Versus Dynamic Scoping

- The following pseudo-code program demonstrates the effect of scoping on variable bindings

```

a:integer
procedure first
  a:=1
procedure second
  a:integer
  first()
procedure main
  a:=2
  second()
  write_integer(a)

```

*Binding with static scoping:  
program execution (shown below) binds a in first() to global variable a*

```

a:integer <-----+
main()             |
a:=2               |
second()          |
  a:integer       |
  first()         |
    a:=1 -----+
    write_integer(a)

```

*Program output = 1*

*Binding with dynamic scoping:  
program execution (shown below) binds a in first() to local variable a of second()*

```

a:integer
main()
a:=2
second()
  a:integer <---+
  first()       |
    a:=1 -----+
    write_integer(a)

```

*Program output = 2*

## Static Scoping

- The bindings between names and objects can be determined at compile time by examination of the program text
- *Scope rules* of a program language define the *scope* of variables and subroutines, which is the region of program text in which a name-to-object binding is usable
  - Early Basic: all variables are global and visible everywhere
  - Fortran 77: the scope of a local variable is limited to a subroutine; the scope of a global variable is the whole program text unless it is hidden by a local variable declaration with the same variable name
  - Algol 60, Pascal, and Ada: these languages allow nested subroutine definitions and adopt the *closest nested scope rule* with slight variations in implementation

```

end;
...
begin (* body of P2: P3, P2, A2,
      X of P1, P1, A1 are visible *)
end;
procedure P4(A4:T4);
...
function F1(A5:T5):T6;
var X:integer;
...
begin (* body of F1: X of F1, F1, A5,
      P4, A4, P2, P1, A1 are visible *)
end;
...
begin (* body of P4: F1, P4, A4, P2,
      X of P1, P1, A1 are visible *)
end;
...
begin (* body of P1: X of P1,
      P1, A1, P2, and P4 are visible *)
end
    
```

## Closest Nested Scope Rule

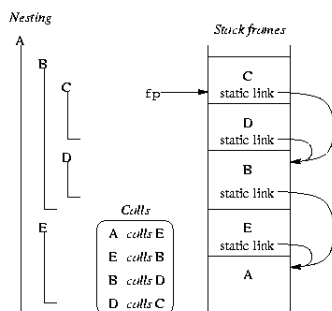
- To find the object referenced by a given name, we look for a declaration in the current innermost scope. If there is none, we look for a declaration in the immediately surrounding scope, etc.

```

procedure P1(A1:T1)
var X:real;
...
procedure P2(A2:T2);
...
procedure P3(A3:T3);
...
begin (* body of P3: P3, A3, P2, A2,
      X of P1, P1, A1 are visible *)
    
```

## Implementation of Static Scope: Static Links

- Scope rules are designed so that we can only refer to variables that are alive: the variable *must* have been stored in the frame of a subroutine
- If a variable is not in the local scope, we are sure there is a frame for the surrounding scope somewhere below on the stack:
  - The current subroutine can *only* be called when it was visible
  - The current subroutine is visible *only* when the surrounding scope is active
- Each frame on the stack contains a *static link* pointing to the frame of the *static parent*
- Example: subroutines C and D are nested in B (B is static parent of C and D), B in A, and E in A



## Bindings to Non-Local Objects: Static Chains

- The static links form a *linked list* of static parent frames
- When a subroutine at nesting level  $j$  has a reference to a local object of a surrounding scope nested at level  $k$ ,  $k-j$  static links forms a *static chain* that is traversed to get to the frame containing the object
  - Example: subroutine A is at nesting level 1 and C at nesting level 3. When C accesses a local object of A, 2 static links are traversed to get to A's frame
- Non-local objects can be *hidden* by local name-to-object bindings and the scope is said to have a *hole* in which the non-local binding is temporarily inactive but not destroyed:

```

procedure P1;
var X:real;
procedure P2;
var X:integer;
begin ... (* X of P1 is hidden *)
end;
begin ...
end
    
```

- When P2 is called, no extra code needs to be executed to inactivate the binding of  $x$  to  $P1$
- Some languages (e.g. Ada and C++) have *qualifiers* or *scope resolution operators* to access non-local objects that are hidden, e.g.  $P1.X$  in Ada to access variable  $x$  of  $P1$  and  $::x$  to access global variable  $x$  in C++

## Blocks and Local Variable Scope

- In Algol, C, and Ada local variables can be declared in a *block* or *compound statement*:

C	Ada
<pre>{ int t = a;   a = b;   b = t; }</pre>	<pre>declare t:integer begin   t := a;   a := b;   b := t; end;</pre>

- In C++ and Java, declarations may appear where statements may appear and the scope extends to the end of the block

```
C++ and Java  
{ int a,b;  
  ...  
  int t;  
  t=a;  
  a=b;  
  b=t;  
  ...  
}
```

- The local objects are stored in the part of a subroutine frame reserved for temporaries

## Modules and Object Scope

- *Modules* are the most important feature of a programming language that supports the construction of large applications
  - Teams of programmers can work on separate module files in a large project
- Modules encapsulate variables, data types, and subroutines such that
  - Objects inside are visible to eachother
  - Objects inside are not visible outside unless exported
  - Objects outside are not visible inside unless imported
- A *module interface* specifies exported variables, data types, and subroutines
- The *module implementation* is compiled separately and implementation details are hidden from the user of the module
- No language support for modules in C and Pascal
- Modula-2 modules, Ada packages, C++ namespaces
- Java class source files and libraries can be viewed as modules

## Dynamic Scope

- Scope rule: the "current" binding for a given name is the one encountered most recently *during execution*
- Typically adopted in (early) functional languages that are interpreted
- Perl v5 allows you to choose scope method for each variable separately
- With dynamic scope:
  - Name-to-object bindings *cannot* be determined by a compiler in general
  - Easy for interpreter to look up name-to-object binding in a stack of declarations
- Generally considered to be "a bad programming language feature"
  - Hard to keep track of active bindings when reading a program text
  - Most languages are now compiled, or a compiler/interpreter mix
- Sometimes useful:
  - Unix environment variables have dynamic scope

## Dynamic Scope Problem

- In the following example program, function `scaled_score` probably does not do what the programmer intended: with dynamic scoping, `max_score` in `scaled_score` is bound to `foo`'s local variable `max_score` after `foo` calls `scaled_score`, which was the most recent binding during execution

```
max_score:integer  
function scaled_score(raw_score:integer):real  
  return raw_score/max_score*100  
  ...  
procedure foo  
  max_score:real:=0  
  ...  
  foreach student in class  
    student.percent:=scaled_score(student.points)  
    if student.percent>max_score  
      max_score:=student.percent
```

- Also: a compiler cannot always determine the type of a non-local variable in a subroutine and run-time type checking is required

## Implementation of Dynamic Scope: Binding Stack

- Each time a subroutine is called, its local variables are pushed on a stack with their name-to-object binding
- When a reference to a variable is made, the stack is searched top-down for the variable's name-to-object binding
- After the subroutine returns, the bindings of the local variables are popped
- Different implementations of a binding stack are used in programming languages with dynamic scope, each with advantages and disadvantages

## Referencing Environments

- If a subroutine is passed as an argument to another subroutine, when are the static/dynamic scoping rules applied?
  - When the reference to the subroutine is first created (i.e. when it is passed as an argument)
  - Or when the argument subroutine is finally called
- That is, what is the *referencing environment* of a subroutine passed as an argument?
  - Eventually the subroutine passed as an argument is called and may access non-local variables which by definition are in the referencing environment of usable bindings
- The choice is fundamental in languages with dynamic scope
- The choice is limited in languages with static scope

## Deep and Shallow Binding in Dynamically Scoped Languages

- The following program demonstrates the difference between deep and shallow binding:

```

thres:integer
function older(p:person):boolean
    return p.age>thres
procedure show(p:person, c:function)
    thres:integer
    thres:=20
    if c(p)
        write(p)
procedure main(p)
    
```

<pre> thres:=35 show(p, older)                 </pre>	
<p><i>Deep binding: reference environment of older is established with the first reference to older, which is when it is passed as an argument to show</i></p>	<p><i>Shallow binding: reference environment of older is established with the call to older in show</i></p>
<pre> main(p)   thres:=35 &lt;-----+   show(p, older)   thres:<b>integer</b>   thres:=20   older(p)   <b>return</b> p.age&gt;thres --+   <b>if</b> &lt;return value is true&gt;     write(p)                 </pre>	<pre> main(p)   thres:=35   show(p, older)   thres:<b>integer</b>   thres:=20 &lt;-----+   older(p)   <b>return</b> p.age&gt;thres --+   <b>if</b> &lt;return value is true&gt;     write(p)                 </pre>
<p><i>Program prints person p if older than 35</i></p>	<p><i>Program prints person p if older than 20</i></p>

## Implementation of Deep Bindings: Subroutine Closures

- The referencing environment is bundled with the subroutine as a *closure* and passed as an argument
- A subroutine closure contains
  - A pointer to the subroutine code
  - The current set of name-to-object bindings
- Depending on the implementation, the whole current set of bindings may have to be copied or the head of a list is copied if linked lists are used to implement a stack of bindings

## Deep and Shallow Binding in Statically Scoped Languages

- Shallow binding has never been implemented in any statically scoped language
  - Deep binding in a statically scoped language is obvious choice
  - Shallow bindings require more work by a compiler and at run time
- For example, in the program below, static scoping binds `thres` in `older` to the globally declared `thres`, so shallow binding does not make sense

```
thres:integer
function older(p:person):boolean
  return p.age>thres
procedure show(p:person, c:function)
  thres:integer
  thres:=20
  if c(p)
    write(p)
  procedure main(p)
    thres:=35
    show(p, older)
```

## First-, Second-, and Third-Class Subroutines

- *First-class object*: an object that can be passed as a parameter, returned from a subroutine, and assigned to a variable
  - E.g. primitive types such as integers in most programming languages
- *Second-class object*: an object that can be passed as a parameter, but not returned from a subroutine or assigned to a variable
  - E.g. arrays in C/C++
- *Third-class object*: an object that cannot be passed as a parameter, cannot be returned from a subroutine, and cannot be assigned to a variable
  - E.g. labels of `goto`-statements and subroutines in Ada 83
- With certain restrictions, subroutines are first-class objects in Modula-2 and 3, Ada 95, C, and C++
- Functions in Lisp, ML, and Haskell are unrestricted first-class objects

## First-Class Subroutines

- Problem: subroutine returned as object may lose part of its reference environment in its closure
- Consider for example the pseudo-code program below:

```
function new_int_printer(port:integer):procedure
  procedure print_int(val:int)
    begin /*print_int*/
      write(port, val)
    end /*print_int*/
  begin /*new_int_printer*/
    return print_int
  end /*new_int_printer*/
  procedure main
  begin /*main*/
    myprint:=new_int_printer(80)
    myprint(7)
  end /*main*/
```

- Procedure `print_int` uses argument `port` of `new_int_printer`, which is in the referencing environment of `print_int`
- After the call to `new_int_printer`, argument `port` should be kept alive somehow (it is normally removed from the run-time stack and it will become a dangling reference)

## First-Class Subroutines (cont'd)

- In functional languages, local objects have *unlimited extent*: their lifetime continue indefinitely
  - Local objects are allocated on the heap
  - Garbage collection will eventually remove unused local objects
- In imperative languages, local objects have *limited extent*: stack allocation
- To avoid the problem of dangling references, alternative mechanisms are used:
  - C, C++, and Java: no nested subroutine scopes
  - Modula-2: only outermost routines are first-class
  - Ada 95 "containment rule": can return an inner subroutine under certain conditions

## Overloading and Bindings

- A name that can refer to more than one object is said to be *overloaded*
  - E.g. `+` (addition) is used for integer and floating-point addition in most programming languages
- Semantic rules of a programming language require that the context of an overloaded name should contain sufficient clues to deduce intended binding
- Semantic analyzer of compiler uses type checking to resolve bindings
- Ada, C++, and Java subroutine overloading enables programmer

to define alternative implementations depending on argument types, for example in C++:

```
struct complex {...};
enum base {dec, bin, oct, hex};
void print_num(int n) ...
void print_num(int n, base b) ...
void print_num(struct complex c) ...
```

- Ada, C++, and Fortran 90 allow built-in operators to be overloaded with user-defined functions

## Polymorphic Subroutines and Templates

- Lisp, ML, Haskell, and Smalltalk allow programmers to write subroutines with *polymorphic* parameters: objects of more than one type
- For example, the Haskell `length` function that returns the length of a list has a polymorphic parameter which is a list of elements of any type (`x:xs` is a list with head `x` and tail `xs`):

```
length (x:xs) = 1 + length xs
length []    = 0
```

- C++ templates accomplish a similar feature for classes, but a concrete implementation is created by compiling the template for each type

```
template<class T> class list
{ private:
  T *next; // pointer to next node in list
  int size; // size of list = next->size + 1
 public:
  int length() { return size; };
};
```

Exercise 1: Consider the following class instances in a C++ program:

```
myClass A;
int main()
{ myClass B;
  myClass *C = new myClass;
  ...
  delete C;
  ...
}
```

- What is the storage allocation for the `A`, `B`, and `C` objects? (static/stack/heap)

- Draw the time line of the binding lifetimes and object lifetimes for `A`, `B`, and `C`.
- Explain what a dangling reference is and illustrate this concept by modifying the code to create a dangling reference for `C`.
- Explain what a memory leak is and illustrate this concept by modifying the code to create a memory leak.

Exercise 2: Explain the difference between internal and external heap fragmentation.

Exercise 3: Compile and run the following C++ program and explain in detail what happens:

```
int main()
{ int a = 1;
  int b = 2;
  int c = 3;
  int *p = &b;
  p[1] = 4;
  cout << "a=" << a << "b=" << b << "c=" << c << endl
}
```