Names, Scopes, and Bindings

In this set of notes you will learn about:

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

Note: These slides cover Chapter 3 of the textbook, except Section 3.6. You may skip reading Sections 3.3.3, 3.3.4, and 3.6 of the textbook.

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 addresses
- Names are also control and data abstractions of complicated program fragments 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 simple 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).

---

**Language feature** | **Binding time**
---|---
Syntax, e.g. `if (a>0) b:=a;` in C or `if a>0 then b:=a end if` in Ada | Language design
Keywords, e.g. `class` in C++ and Java | Language design
Reserved words, e.g. `main` in C and `writeln` in Pascal | Language design
Meaning of operators, e.g. `+` (add) | Language design
Primitive types, e.g. `float` and `struct` in C | Language design
Internal representation of literals, e.g. `3.1` and "foo bar" | 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 library routines, e.g. `printf` 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 generating code that makes a decision at run time
  - Syntax and static semantics checking is performed only once
- *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

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:

```cpp
{ 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 somewhere 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)

Stack
  - Heap
  Static data
  Program code

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

<table>
<thead>
<tr>
<th>Temporaries (for intermediate values)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Local variables</td>
</tr>
<tr>
<td>Miscellaneous bookkeeping</td>
</tr>
<tr>
<td>(saved processor registers)</td>
</tr>
<tr>
<td>Return address</td>
</tr>
<tr>
<td>Arguments (in and out)</td>
</tr>
</tbody>
</table>
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 *prologue* code that creates a frame on a subroutine call at run time
  - Compiler generates *epilogue* code that destroys a frame on a subroutine return at run time
- 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

```
Subroutine A
Temporaries
Local vars
Bookkeeping
Return address
Arguments

Subroutine B
Temporaries
Local vars
Bookkeeping
Return address
Arguments

Main program
Temporaries
Local vars
Bookkeeping
Return address
Arguments
```

Example C program
```
main()
{
  ... 
  A();
  ...
}

A()
{
  ...
  B();
  ...
}

B()
{
  ...
  A();
  ...
}

main calls A,
A calls B,
B calls A
```
**Example Frame**

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

<table>
<thead>
<tr>
<th>Temporaries</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Local vars</td>
<td></td>
</tr>
<tr>
<td>var</td>
<td>offset</td>
</tr>
<tr>
<td>foo</td>
<td>-18</td>
</tr>
<tr>
<td>bar</td>
<td>-16</td>
</tr>
<tr>
<td>p</td>
<td>-12</td>
</tr>
</tbody>
</table>

Example Pascal procedure

```pascal
procedure P(a:integer, var b:real) (* a is passed by value
       b is passed by reference, which is a pointer to b’s value *)
begin...
end
```

**Bookkeeping**

(8 words)

<table>
<thead>
<tr>
<th>Return address</th>
<th>(2 words)</th>
</tr>
</thead>
<tbody>
<tr>
<td>¬fp</td>
<td></td>
</tr>
</tbody>
</table>

**Arguments**

<table>
<thead>
<tr>
<th>arg</th>
<th>offset</th>
<th>size</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>0</td>
<td>2 words</td>
</tr>
<tr>
<td>b</td>
<td>2</td>
<td>2 words</td>
</tr>
</tbody>
</table>

- 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

<table>
<thead>
<tr>
<th>Object</th>
<th>free (4 words)</th>
<th>Object</th>
<th>Object</th>
<th>free (12 words)</th>
<th>Object</th>
<th>free (1 word)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td></td>
<td>B</td>
<td>C</td>
<td></td>
<td>D</td>
<td></td>
</tr>
</tbody>
</table>

- 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 $2^k$ and $2^k - 1$, 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

<table>
<thead>
<tr>
<th>Static</th>
<th>Stack</th>
<th>Heap</th>
</tr>
</thead>
</table>

- Static
- Stack
- Heap
<table>
<thead>
<tr>
<th>Language</th>
<th>Scope Type</th>
<th>Allocation Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ada</td>
<td>local variables of fixed size and subroutine arguments</td>
<td>implicit: local variables of variable size, explicit: <code>new</code> (destruction with garbage collection or explicit with unchecked deallocation)</td>
</tr>
<tr>
<td>C</td>
<td>global variables; <code>static</code> local variables, e.g. <code>f()</code> { <code>static</code> int n; ... }</td>
<td>local variables and subroutines arguments</td>
</tr>
<tr>
<td>C++</td>
<td>global variables; <code>static</code> members</td>
<td>local variables and subroutine arguments</td>
</tr>
<tr>
<td>Java</td>
<td>N/A</td>
<td>local variables with primitive types (e.g. <code>int</code> and <code>char</code>)</td>
</tr>
<tr>
<td>Fortran77</td>
<td>global variables (in common blocks), local variables, and subroutine arguments (implementation dependent), <code>SAVE</code> forces static allocation</td>
<td>local variables and subroutine arguments (implementation dependent)</td>
</tr>
<tr>
<td>Pascal</td>
<td>global variables (dependent on compiler)</td>
<td>global variables (dependent on compiler), local variables, and subroutine arguments</td>
</tr>
</tbody>
</table>
**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 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

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

<table>
<thead>
<tr>
<th>Binding with static scoping: program execution (shown below) binds a in first() to global variable a</th>
<th>Binding with dynamic scoping: program execution (shown below) binds a in first() to local variable a of second()</th>
</tr>
</thead>
<tbody>
<tr>
<td>a: integer ←-Ø</td>
<td></td>
</tr>
<tr>
<td>main()</td>
<td></td>
</tr>
<tr>
<td>a:= 2</td>
<td></td>
</tr>
<tr>
<td>second()</td>
<td></td>
</tr>
<tr>
<td>a: integer</td>
<td></td>
</tr>
<tr>
<td>first()</td>
<td></td>
</tr>
<tr>
<td>a:= 1</td>
<td></td>
</tr>
<tr>
<td>main()</td>
<td></td>
</tr>
<tr>
<td>a:= 2</td>
<td></td>
</tr>
<tr>
<td>second()</td>
<td></td>
</tr>
<tr>
<td>a: integer</td>
<td></td>
</tr>
<tr>
<td>first()</td>
<td></td>
</tr>
<tr>
<td>a:= 1</td>
<td></td>
</tr>
<tr>
<td>write_integer(a)</td>
<td></td>
</tr>
</tbody>
</table>

Program output = 1  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 subroutines definitions and adopt the closest nested scope rule with slight variations in implementation

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.
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 in 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:

```plaintext
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:

<table>
<thead>
<tr>
<th>C</th>
<th>Ada</th>
</tr>
</thead>
<tbody>
<tr>
<td>int t = a;</td>
<td>declare t: integer</td>
</tr>
<tr>
<td>a = b;</td>
<td>begin</td>
</tr>
<tr>
<td>b = t;</td>
<td>t := a;</td>
</tr>
<tr>
<td></td>
<td>a := b;</td>
</tr>
<tr>
<td></td>
<td>b := t;</td>
</tr>
<tr>
<td>end;</td>
<td>end;</td>
</tr>
</tbody>
</table>

- In C++ and Java, declarations may appear where statements may appear and the scope extends to the end of the innermost 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 each other.
  - 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.

Note: The textbook provides a discussion on modules (pp.122-129) which is neither brief nor comprehensive enough. You are not required to study modules described in this part of the textbook.
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
```
## 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.

Note: Textbook Sections 3.3.3 and 3.3.4 deal with the binding stack implementation issues. You are not required to study these sections.

### 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:

```plaintext
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)
```

```plaintext
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)
```
### Implementation of Deep Bindings: Subroutine Closures

- The referencing environment is bundled with the subroutine as a *closure* when it is 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.

<table>
<thead>
<tr>
<th>Deep binding: reference environment of ( \text{older} ) is established with the first reference to ( \text{older} ), which is when it is passed as an argument to ( \text{show} )</th>
<th>Shallow binding: reference environment of ( \text{older} ) is established with the call to ( \text{older} ) in ( \text{show} )</th>
</tr>
</thead>
</table>
| \[
\begin{align*}
\text{thres} & := 35 \\
\text{show}(p, \text{older})
\end{align*}
\] | \[
\begin{align*}
\text{thres} & := 35 \\
\text{show}(p, \text{older})
\end{align*}
\] |
| \[
\begin{align*}
\text{main}(p) \\
\text{thres} & := 35 \\
\text{show}(p, \text{older})
\end{align*}
\] | \[
\begin{align*}
\text{main}(p) \\
\text{thres} & := 35 \\
\text{show}(p, \text{older})
\end{align*}
\] |
| **Return** \( p . \text{age} > \text{thres} \) | **Return** \( p . \text{age} > \text{thres} \) |
| \[
\begin{align*}
\text{return} & \text{ if } \langle \text{return value is true} \rangle \\
\text{write}(p)
\end{align*}
\] | \[
\begin{align*}
\text{return} & \text{ if } \langle \text{return value is true} \rangle \\
\text{write}(p)
\end{align*}
\] |

Program prints person \( p \) if older than 35

Program prints person if older than 20
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 \texttt{thres} in \texttt{older} to the global declaration of \texttt{thres}, so shallow binding does not make sense

```plaintext
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 \texttt{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:

```pseudocode
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: procedure
      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 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++:

```cpp
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):

```haskell
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:

```cpp
template<class T> class list
{  T* p;
   int size;
public:
   int length() { return size; }
};
```