COP4020
Programming Languages

Names, Scopes, and Bindings

Prof. Robert van Engelen
Overview

- Abstractions and names
- Binding time
- Object lifetime
- Object storage management
  - Static allocation
  - Stack allocation
  - Heap allocation
- Scope rules
- Static versus dynamic scoping
- Reference environments
- Overloading and polymorphism
Name = Abstraction

- Names enable programmers to refer to variables, constants, operations, and types
- Names are control abstractions and data abstractions for 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
    - Control flow constructs (if-then, while, for, return) hide low-level machine ops
  - 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 working with code and data
- Enhances the level of machine-independence
- "Power" of constructs
Binding Time

- A binding is an association between a name and an entity
- Binding time is the time at which an implementation decision is made to create a name ↔ entity binding:
  - Language design time: the design of specific program constructs (syntax), primitive types, and meaning (semantics)
  - 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.
  - Program writing time: the programmer's choice of algorithms and data structures
  - Compile time: the time of translation of high-level constructs to machine code and choice of memory layout for data objects
  - Link time: the time at which multiple object codes (machine code files) and libraries are combined into one executable
  - Load time: when the operating system loads the executable in memory
  - Run time: when a program executes
Binding Time Examples

Language design:

- Syntax (names ↔ grammar)
  - `if (a>0) b=a;` (C syntax style)
  - `if a>0 then b:=a end if` (Ada syntax style)
- Keywords (names ↔ builtins)
  - `class` (C++ and Java), `endif` or `end if` (Fortran, space insignificant)
- Reserved words (names ↔ special constructs)
  - `main` (C), `writeln` (Pascal)
- Meaning of operators (operator ↔ operation)
  - `+` (add), `%` (mod), `**` (power)
- Built-in primitive types (type name ↔ type)
  - `float, short, int, long, string`

Language implementation

- Internal representation of types and literals (type ↔ byte encoding)
  - 3.1 (IEEE 754) and "foo bar" (\0 terminated or embedded string length)
- Storage allocation method for variables (static/stack/heap)
Binding Time Examples (cont’d)

- Compile time
  - The specific type of a variable in a declaration (name↔type)
  - Storage allocation method for a global or local variable (name↔allocation mechanism)

- Linker
  - Linking calls to static library routines (function↔address)
    - `printf` (in libc)
  - Merging and linking multiple object codes into one executable

- Loader
  - Loading executable in memory and adjusting absolute addresses
    - Mostly in older systems that do not have virtual memory

- Run time
  - Dynamic linking of libraries (library function↔library code)
    - DLL, dylib
  - Nonstatic allocation of space for variable (variable↔address)
    - Stack and heap
The Effect of Binding Time

- **Early binding times** (before run time) are associated with greater efficiency and clarity of program code
  - Compilers make implementation decisions at compile time (avoiding to generate code that makes the 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 (but may leave programmers sometimes guessing what’s going on)
  - 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*
Binding Lifetime versus Object Lifetime

- Key events in object lifetime:
  - Object creation
  - Creation of bindings
  - The object is manipulated via its binding
  - Deactivation and reactivation of (temporarily invisible) bindings
  - Destruction of bindings
  - Destruction of objects

- **Binding lifetime**: time between creation and destruction of binding to object
  - Example: a pointer variable is set to the address of an object
  - Example: a formal argument is bound to an actual argument

- **Object lifetime**: time between creation and destruction of an object
Binding Lifetime versus Object Lifetime (cont’d)

- Bindings are temporarily invisible when code is executed where the binding (name ↔ object) is out of scope
- *Memory leak*: object never destroyed (binding to object may have been destroyed, rendering access impossible)
- *Dangling reference*: object destroyed before binding is destroyed
- *Garbage collection* prevents these allocation/deallocation problems
C++ Example

```cpp
{  
    SomeClass* myobject = new SomeClass;
    ...
    {
        OtherClass myobject;
        ... // the myobject name is bound to other object
        ...
    }
    ...
    // myobject binding is visible again
    ...
    myobject->action() // myobject in action():
        // the name is not in scope
        // but object is bound to ‘this’
    delete myobject;
    ...
    ...
    // myobject is a dangling reference
}
```
Object Storage

- Objects (program data and code) have to be stored in memory during their lifetime

- Static objects have an absolute storage address that is retained throughout the execution of the program
  - Global variables and data
  - Subroutine code and class method code

- Stack objects are allocated in last-in first-out order, usually in conjunction with subroutine calls and returns
  - Actual arguments passed by value to a subroutine
  - Local variables of a subroutine

- Heap objects may be allocated and deallocated at arbitrary times, but require an expensive storage management algorithm
  - Example: Lisp lists
  - Example: 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)
  - The code section is protected from run-time modification by the OS
- Static data objects are stored in the static region
- Stack grows downward
- Heap grows upward
Static Allocation

- Program code is statically allocated in most implementations of imperative languages
- Statically allocated variables are history sensitive
  - Global variables keep state during entire program lifetime
  - Static local variables in C functions keep state across function invocations
  - Static data members are “shared” by objects and keep state during program lifetime
- Advantage of statically allocated object is the fast access due to absolute addressing of the object
  - So why not allocate local variables statically?
  - Problem: static allocation of local variables cannot be used for recursive subroutines: each new function instantiation needs fresh locals
Static Allocation in Fortran 77

- Fortran 77 has no recursion
- Global and local variables are statically allocated as decided by the compiler
- Global and local variables are referenced at absolute addresses
- Avoids overhead of creation and destruction of local objects for every subroutine call
- Each subroutine in the program has a subroutine frame that is statically allocated
- This subroutine frame stores all subroutine-relevant data that is needed to execute

**Typical static subroutine frame layout**

<table>
<thead>
<tr>
<th>Temporary storage (e.g. for expression evaluation)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Local variables</td>
</tr>
<tr>
<td>Bookkeeping (e.g. saved CPU registers)</td>
</tr>
<tr>
<td>Return address</td>
</tr>
<tr>
<td>Subroutine arguments and returns</td>
</tr>
</tbody>
</table>
Stack Allocation

- Each instance of a subroutine that is active has a *subroutine frame* (sometimes called *activation record*) on the run-time stack
  - Compiler generates subroutine calling sequence to setup frame, call the routine, and to destroy the frame afterwards
  - Method invocation works the same way, but in addition methods are typically dynamically bound

- Subroutine frame layouts vary between languages, implementations, and machine platforms
## Typical Stack-Allocated Subroutine Frame

<table>
<thead>
<tr>
<th>Lower addr</th>
<th>Temporary storage (e.g. for expression evaluation)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Local variables</td>
</tr>
<tr>
<td></td>
<td>Bookkeeping (e.g. saved CPU registers)</td>
</tr>
<tr>
<td></td>
<td>Return address</td>
</tr>
<tr>
<td></td>
<td>Subroutine arguments and returns</td>
</tr>
</tbody>
</table>

- A **frame pointer** (fp) points to the frame of the currently active subroutine at run time.

- Subroutine arguments, local variables, and return values are accessed by constant address offsets from the fp.
Subroutine Frames on the Stack

- Subroutine frames are pushed and popped onto/from the runtime stack.
- The *stack pointer* (sp) points to the next available free space on the stack to push a new frame onto when a subroutine is called.
- The *frame pointer* (fp) points to the frame of the currently active subroutine, which always the topmost frame on the stack.
- The fp of the previous active frame is saved in the current frame and restored after the call.
- In this example:
  - M called A
  - A called B
  - B called A
Example Subroutine Frame

The size of the types of local variables and arguments determines the fp offset in a frame.

Example Pascal procedure:

```pascal
procedure P(a:integer, var b:real);
(* a is passed by value
   b is passed by reference,
   = pointer to b's value *)
var
  foo:integer;(* 4 bytes *)
  bar:real;    (* 8 bytes *)
  p:^integer; (* 4 bytes *)
begin
  ...
end
```

```
<table>
<thead>
<tr>
<th>Lower addr</th>
<th>Temporaries</th>
</tr>
</thead>
<tbody>
<tr>
<td>-36: foo (4 bytes)</td>
<td></td>
</tr>
<tr>
<td>-32: bar (8 bytes)</td>
<td></td>
</tr>
<tr>
<td>-24: p (4 bytes)</td>
<td></td>
</tr>
</tbody>
</table>

Bookkeeping (16 bytes)

The stack frame for procedure P:

```
<table>
<thead>
<tr>
<th>Lower addr</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0: a (4 bytes)</td>
<td></td>
</tr>
<tr>
<td>4: b (4 bytes)</td>
<td></td>
</tr>
</tbody>
</table>
```

fp-32 |
fp |
fp+4 | Higher addr
Heap Allocation

- Implicit heap allocation:
  - Done automatically
  - Java class instances are 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 runtime dependent array size
  - Resizable character strings

- Explicit heap allocation:
  - Statements and/or functions for allocation and deallocation
  - Malloc/free, new/delete
Heap Allocation Algorithms

- Heap allocation is performed by searching the heap for available free space.
- For example, suppose we want to allocate a new object E of 20 bytes, where would it fit?

<table>
<thead>
<tr>
<th>Object A</th>
<th>Free</th>
<th>Object B</th>
<th>Object C</th>
<th>Free</th>
<th>Object D</th>
<th>Free</th>
</tr>
</thead>
<tbody>
<tr>
<td>30 bytes</td>
<td>8 bytes</td>
<td>10 bytes</td>
<td>24 bytes</td>
<td>24 bytes</td>
<td>8 bytes</td>
<td>20 bytes</td>
</tr>
</tbody>
</table>

- 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 (cont’d)

- Maintain a linked list of free heap blocks
- **First-fit**: select the first block in the list that is large enough
- **Best-fit**: search the 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**: use 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**: use heap pools of standard size blocks according to Fibonacci numbers
  - More complex but leads to slower internal fragmentation
Unlimited Extent

- An object declared in a local scope has *unlimited extent* if its lifetime potentially continues indefinitely, as long as bindings to the object exist that may be out of scope.

- A local stack-allocated variable has a lifetime limited to the lifetime of the subroutine.
  - In C/C++ functions should never return pointers to local variables.

- Unlimited extent requires static or heap allocation.
  - Issues with static: limited, no mechanism to allocate more variables.
  - Issues with heap: should deallocate when no longer referenced by bindings.
  - Solution using heap: garbage collection removes object when no longer bound (by any references).
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 processing
## Storage Allocation Compared

<table>
<thead>
<tr>
<th></th>
<th>Static</th>
<th>Stack</th>
<th>Heap</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ada</td>
<td>N/A</td>
<td>local variables and subroutine arguments of fixed size</td>
<td><em>implicit</em>: local variables of variable size; <em>explicit</em>: new (destruction with garbage collection or explicit with unchecked deallocation)</td>
</tr>
<tr>
<td>C</td>
<td>global variables; static local variables</td>
<td>local variables and subroutine arguments</td>
<td><em>explicit</em> with <code>malloc</code> and <code>free</code></td>
</tr>
<tr>
<td>C++</td>
<td>Same as C, and static class members</td>
<td>Same as C</td>
<td><em>explicit</em> with <code>new</code> and <code>delete</code></td>
</tr>
<tr>
<td>Java</td>
<td>N/A</td>
<td>only local variables of primitive types</td>
<td><em>implicit</em>: all class instances (destruction with garbage collection)</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>
<td>N/A</td>
</tr>
<tr>
<td>Pascal</td>
<td>global variables (compiler dependent)</td>
<td>global variables (compiler dependent), local variables, and subroutine arguments</td>
<td><em>Explicit</em>: <code>new</code> and <code>dispose</code></td>
</tr>
</tbody>
</table>
Scope

Scope is 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 (selectively)
Effect of Static Scoping

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

- `a:integer`

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

Program prints “1”
Effect of Dynamic Scoping

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

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

Program prints “2”
```

Binding depends on execution
Static Scoping

- The bindings between names and objects can be determined 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:

- Look for a declaration in the current innermost scope
- If there is none, look for a declaration in the immediately surrounding scope, etc.
Static Scope Implementation with 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 Static Links

- Subroutines C and D are declared nested in B
  - B is static parent of C and D
- B and E are nested in A
  - A is static parent of B and E
- The fp points to the frame at the top of the stack to access locals
- The static link in the frame points to the frame of the static parent
Static Chains

- How do we access non-local objects?
- The static links form a static chain, which is a linked list of static parent frames.
- When a subroutine at nesting level \( j \) has a reference to an object declared in a static parent at the surrounding scope nested at level \( k \), then \( j-k \) static links forms a static chain that is traversed to get to the frame containing the object.
- The compiler generates code to make these traversals over frames to reach non-local objects.
Example Static Chains

- Subroutine A is at nesting level 1 and C at nesting level 3
- When C accesses an object of A, 2 static links are traversed to get to A's frame that contains that object
Out of Scope

- 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.

- Some languages, notably Ada and C++ use qualifiers or scope resolution operators to access non-local objects that are hidden:
  - P1.X in Ada to access variable X of P1 and ::X to access global variable X in C++.
Out of Scope Example

- P2 is nested in P1
- P1 has a local variable X
- P2 has a local variable X that hides X in P1
- When P2 is called, no extra code is executed to inactivate the binding of X to P1

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

In this example, 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:

```pseudocode
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
```
Dynamic Scope Implementation with Bindings Stacks

- 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
Effect of Deep Binding in Dynamically-Scoped Languages

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

```plaintext
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:integer
    thres:=35
    show(p,older)

main(p)
    thres:integer
    thres:=35
    show(p,older)
    thres:integer
    thres:=20
    older(p)
    return p.age>thres
    if return value is true
    write(p)

Program prints persons older than 35
```

Program execution:

Program prints persons older than 35
Effect of Shallow Binding in Dynamically-Scoped Languages

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

```plaintext
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:integer
  thres := 35
  show(p,older)

main(p)
```

Program execution:

```
main(p)
thres:integer
thres := 35
show(p,older)
  thres:integer
  thres := 20
  older(p)
    return p.age>thres
  if return value is true
    write(p)

Program prints persons older than 20
```

The program demonstrates the difference between deep and shallow binding. The shallow binding is demonstrated by the `show` procedure, which uses the `older` function directly. The deep binding is demonstrated by the `main` procedure, which uses the `show` procedure that in turn uses the `older` function.
Implementing Deep Bindings with 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.
### Statement Blocks

- In Algol, C, and Ada local variables are declared in a block or compound statement.
- In C++, Java, and C# declarations may appear anywhere statements can be used and the scope extends to the end of the block.
- Local variables declared in nested blocks in a single function are all stored in the subroutine frame for that function (most programming languages, e.g. C/C++, Ada, Java).

```c
{ int t = a;
  a = b;
  b = t;
}
```

```ada
declare t:integer
begin
  t := a;
  a := b;
  b := t;
end;
```

```cpp
{ int a,b;
  ...
  int t;
  t=a;
  a=b;
  b=t;
  ...
}
```

```java
{ int a,b;
  ...
  int t;
  t=a;
  a=b;
  b=t;
  ...
}
```

```c#``
Modules and Module 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 modules in a project
  - No language support for modules in C and Pascal
  - Modula-2 modules, Ada packages, C++ namespaces
  - Java packages

- Scoping: modules encapsulate variables, data types, and subroutines in a package
  - 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
First, Second, and Third-Class Subroutines

- **First-class object**: an object entity that can be passed as a parameter, returned from a subroutine, and assigned to a variable
  - 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
  - Fixed-size 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
  - Labels of goto-statements and subroutines in Ada 83

- Functions in Lisp, ML, and Haskell are unrestricted first-class objects
- With certain restrictions, subroutines are first-class objects in Modula-2 and 3, Ada 95, (C and C++ use function pointers)
First-Class Subroutine Implementation Requirements

function new_int_printer(port:integer):procedure
  procedure print_int(val:int)
  begin
    write(port, val)
  end
begin
  return print_int
end

procedure main
begin
  myprint:procedure
  myprint := new_int_printer(80)
  myprint(7)
end

- Problem: subroutine returned as object may lose part of its reference environment in its closure!
- 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 Subroutine Implementations

- 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 objects

- In imperative languages, local objects have *limited extent* with 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
Overloaded Bindings

- A name that can refer to more than one object is said to be overloaded
  - Example: + (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 the intended binding
- Semantic analyzer of compiler uses type checking to resolve bindings
- Ada and C++ function overloading enables programmer to define alternative implementations depending on argument types
- Ada, C++, and Fortran 90 allow built-in operators to be overloaded with user-defined functions, which enhances expressiveness but may mislead programmers that are unfamiliar with the code
Overloaded Bindings Example

Example in C++:

```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) { ... }
```
Dynamic Bindings

- Polymorphic functions and operators based on overloading are statically bound by the compiler based on type information.
- Polymorphism with *dynamic bindings* is supported by class inheritance (C++ virtual methods).
- Each class has a *virtual method table* (VMT) with pointers to methods, where each method is indexed into the table.
- Method invocation proceeds by getting the class VMT from the object and indexing it to select the method to invoke.