Names, Scopes, and Binding

CSCI 3136
Principles of Programming Languages

Faculty of Computer Science
Dalhousie University

Winter 2012

Reading: Chapter 3
A *name* is a mnemonic character string representing something else:

- x, sin, f, prog1, null? are names
- 1, 2, 3, "test" are not names
- +, <=, … may be names if they are not built-in operators
Names, Binding, and Scope

A **name** is a mnemonic character string representing something else:

- x, sin, f, prog1, null? are names
- 1, 2, 3, "test" are not names
- +, <=, ... may be names if they are not built-in operators

A **binding** is an association between two entities:

- Name and memory location (for variables)
- Name and function

Typically a binding is between a name and the object it refers to.
Names, Binding, and Scope

A name is a mnemonic character string representing something else:

• x, sin, f, prog1, null? are names
• 1, 2, 3, "test" are not names
• +, <=, ... may be names if they are not built-in operators

A binding is an association between two entities:

• Name and memory location (for variables)
• Name and function

Typically a binding is between a name and the object it refers to.

A referencing environment is a complete set of bindings active at a certain point in a program.
Names, Binding, and Scope

A name is a mnemonic character string representing something else:

- x, sin, f, prog1, null? are names
- 1, 2, 3, "test" are not names
- +, <=, ... may be names if they are not built-in operators

A binding is an association between two entities:

- Name and memory location (for variables)
- Name and function

Typically a binding is between a name and the object it refers to.

A referencing environment is a complete set of bindings active at a certain point in a program.

The scope of a binding is the region of a program or time interval(s) in the program’s execution during which the binding is active.
Names, Binding, and Scope

A *name* is a mnemonic character string representing something else:
- x, sin, f, prog1, null? are names
- 1, 2, 3, "test" are not names
- +, <=, … may be names if they are not built-in operators

A *binding* is an association between two entities:
- Name and memory location (for variables)
- Name and function

Typically a binding is between a name and the object it refers to.

A *referencing environment* is a complete set of bindings active at a certain point in a program.

The *scope of a binding* is the region of a program or time interval(s) in the program’s execution during which the binding is active.

A *scope* is a maximal region of the program where no bindings are destroyed (e.g., body of a procedure).
Binding Times

Compile time
- Map high-level language constructs to machine code
- Layout static data in memory

Link time
- Resolve references between separately compiled modules

Load time
- Assign machine addresses to static data

Run time
- Bind values to variables
- Allocate dynamic data and assign to variables
- Allocate local variables of procedures on the stack
Importance of Binding Time

Early binding

- Faster code
- Typical in compiled languages

Late binding

- Greater flexibility
- Typical in interpreted languages
Object and Binding Lifetime

Object lifetime

- Period between the creation and destruction of the object
- Example: time between creation and destruction of a dynamically allocated variable in C++ using `new` and `delete`

Binding lifetime

- Period between the creation and destruction of the binding (name-to-object association)

Two common mistakes

- Dangling reference: no object for a binding (e.g., a pointer refers to an object that has already been deleted)
- Memory leak: no binding for an object (preventing the object from being deallocated)
Storage Allocation

An object’s lifetime corresponds to the mechanism used to manage the space where the object resides.

**Static object**
- Object stored at a fixed absolute address
- Object’s lifetime spans the whole execution of the program.

**Object on stack**
- Object allocated on stack in connection with a subroutine call
- Object’s lifetime spans period between invocation of the subroutine and return from the subroutine.

**Object on heap**
- Object stored on heap
- Object created/destroyed at arbitrary times
  - Explicitly by programmer or
  - Implicitly by garbage collector
Example: Object Creation and Destruction in C++

- **Local objects** are local to functions and blocks and exist while the execution is inside the function or block.
- **Heap objects** are allocated/deallocated using `new/delete`.
- **Non-static member objects** of a parent object exist while the parent object exists.
- **Array elements** exist while the array exists.
- **Local static objects** of functions/blocks exist after the first invocation of the function until termination.
- **Global, namespace, class static objects** exist while the program runs.
- **Temporary objects** in expressions exist during the evaluation of the expression.
- User-supplied allocation function may change the lifetime of heap objects.
Static Objects

- Global variables
- Static variables local to subroutines that retain their value between invocations
- Constant literals
- Tables for run-time support: debugging, type checking, etc.
- Space for subroutines, including local variables in languages that do not support recursion (e.g., early versions of FORTRAN)
Stack-Based Allocation

The stack is used to allocate space for subroutines in languages that permit recursion.

The *stack frame* (activation record) stores

- Arguments and local variables of the subroutine,
- The return value(s) of the subroutine,
- The return address,
- ...

The subroutine *calling sequence* maintains the stack:

- Before call, the caller pushes arguments and return address onto the stack.
- After being called (prologue), the subroutine (“callee”) initializes local variables, etc.
- Before returning (epilogue), the subroutine cleans up local data.
- After the call returns, the caller retrieves return value(s) and restores the stack to its state before the call.
Stack Frame (Activation Record)

Compiler determines

- **Frame pointer**: a register pointing to a known location within the current stack frame
- Offsets from the frame pointer specifying the location of objects in the stack frame

The absolute size of the stack frame may not be known at compile time (e.g., variable-size arrays allocated on the stack).

Stack pointer

- Register pointing to the first unused location on the stack (used as the starting position for the next stack frame)

Specified at runtime

- The absolute location of the stack frame in memory (on the stack)
- The size of the stack frame
Stack Frame Before, During and After Subroutine Call

Before and after

lower addresses

stack grows this way

Subroutine A

Subroutine B

higher addresses

During

lower addresses

stack grows this way

Temporary data

Local variables

Bookkeeping

Return address

Arguments and return values

Subroutine B

Subroutine A

higher addresses

FP

SP

Names, Scopes, and Binding
CSCI 3136: Principles of Programming Languages
Heap-Based Allocation

The heap is a region of memory where blocks can be allocated and deallocated at arbitrary times and in arbitrary order.

Heap management

- **Free list**: list of blocks of free memory
- The allocation algorithm searches for a block of adequate size to accommodate the allocation request.

![Diagram of heap memory allocation](image)
First-Fit and Best-Fit Allocation

General allocation strategy

- Find a free block that is at least as big as the requested amount of memory.
- Mark requested number of bytes (plus padding) as allocated.
- Return rest of the free block to free list.
First-Fit and Best-Fit Allocation

General allocation strategy

- Find a free block that is at least as big as the requested amount of memory.
- Mark requested number of bytes (plus padding) as allocated.
- Return rest of the free block to free list.

First fit

- Find the first block large enough to accommodate the allocation request.
First-Fit and Best-Fit Allocation

General allocation strategy

• Find a free block that is at least as big as the requested amount of memory.
• Mark requested number of bytes (plus padding) as allocated.
• Return rest of the free block to free list.

First fit

• Find the first block large enough to accommodate the allocation request.

Best fit

• Find the smallest block large enough to accommodate the allocation request.
Heap Fragmentation Problem

Internal fragmentation

- Often only blocks of certain sizes (e.g., $2^k$) are allocated.
- This may lead to part of an allocated block being unused.

External fragmentation

- Unused space may consist of many small blocks.
- Thus, although the total free space may exceed the allocation request, no block may be large enough to accommodate it.

Neither best-fit nor first-fit is guaranteed to minimize external fragmentation. Which strategy is better depends on the size distribution of the allocation requests.
Cost of Allocation on a Heap

Single free list

- Linear cost to find a block to accommodate each allocation request
Cost of Allocation on a Heap

Single free list
- Linear cost to find a block to accommodate each allocation request

Buddy system
- Blocks of size $2^{n_0}, 2^{n_0+1}, 2^{n_0+2}, \ldots$
- Separate free list for blocks of size $2^k$, for each $k$
- If block of size $2^k$ is unavailable, split block of size $2^{k+1}$
- If block of size $2^k$ is deallocated and its buddy is free, merge them into a block of size $2^{k+1}$
- Worst-case cost: $\log(\text{memory size})$
Cost of Allocation on a Heap

Single free list

• Linear cost to find a block to accommodate each allocation request

Buddy system

• Blocks of size $2^{n_0}, 2^{n_0+1}, 2^{n_0+2}, \ldots$
• Separate free list for blocks of size $2^k$, for each $k$
• If block of size $2^k$ is unavailable, split block of size $2^{k+1}$
• If block of size $2^k$ is deallocated and its buddy is free, merge them into a block of size $2^{k+1}$
• Worst-case cost: $\log(\text{memory size})$
Cost of Allocation on a Heap

Single free list
- Linear cost to find a block to accommodate each allocation request

Buddy system
- Blocks of size $2^{n_0}$, $2^{n_0+1}$, $2^{n_0+2}$, …
- Separate free list for blocks of size $2^k$, for each $k$
- If block of size $2^k$ is unavailable, split block of size $2^{k+1}$
- If block of size $2^k$ is deallocated and its buddy is free, merge them into a block of size $2^{k+1}$
- Worst-case cost: $\log(\text{memory size})$
**Cost of Allocation on a Heap**

**Single free list**
- Linear cost to find a block to accommodate each allocation request

**Buddy system**
- Blocks of size $2^{n_0}, 2^{n_0+1}, 2^{n_0+2}, \ldots$
- Separate free list for blocks of size $2^k$, for each $k$
- If block of size $2^k$ is unavailable, split block of size $2^{k+1}$
- If block of size $2^k$ is deallocated and its buddy is free, merge them into a block of size $2^{k+1}$
- Worst-case cost: $\log(\text{memory size})$
Cost of Allocation on a Heap

Single free list
- Linear cost to find a block to accommodate each allocation request

Buddy system
- Blocks of size $2^{n_0}, 2^{n_0+1}, 2^{n_0+2}, \ldots$
- Separate free list for blocks of size $2^k$, for each $k$
- If block of size $2^k$ is unavailable, split block of size $2^{k+1}$
- If block of size $2^k$ is deallocated and its buddy is free, merge them into a block of size $2^{k+1}$
- Worst-case cost: $\log(\text{memory size})$
Cost of Allocation on a Heap

Single free list

- Linear cost to find a block to accommodate each allocation request

Buddy system

- Blocks of size $2^{n_0}, 2^{n_0+1}, 2^{n_0+2}, \ldots$
- Separate free list for blocks of size $2^k$, for each $k$
- If block of size $2^k$ is unavailable, split block of size $2^{k+1}$
- If block of size $2^k$ is deallocated and its buddy is free, merge them into a block of size $2^{k+1}$
- Worst-case cost: $\log(\text{memory size})$
Cost of Allocation on a Heap

Single free list
- Linear cost to find a block to accommodate each allocation request

Buddy system
- Blocks of size $2^{n_0}, 2^{n_0+1}, 2^{n_0+2}, \ldots$
- Separate free list for blocks of size $2^k$, for each $k$
- If block of size $2^k$ is unavailable, split block of size $2^{k+1}$
- If block of size $2^k$ is deallocated and its buddy is free, merge them into a block of size $2^{k+1}$
- Worst-case cost: $\log(\text{memory size})$
**Cost of Allocation on a Heap**

**Single free list**
- Linear cost to find a block to accommodate each allocation request

**Buddy system**
- Blocks of size $2^{n_0}, 2^{n_0+1}, 2^{n_0+2}, \ldots$
- Separate free list for blocks of size $2^k$, for each $k$
- If block of size $2^k$ is unavailable, split block of size $2^{k+1}$
- If block of size $2^k$ is deallocated and its buddy is free, merge them into a block of size $2^{k+1}$
- Worst-case cost: $\log(\text{memory size})$
Cost of Allocation on a Heap

Single free list
- Linear cost to find a block to accommodate each allocation request

Buddy system
- Blocks of size $2^{n_0}, 2^{n_0+1}, 2^{n_0+2}, \ldots$
- Separate free list for blocks of size $2^k$, for each $k$
- If block of size $2^k$ is unavailable, split block of size $2^{k+1}$
- If block of size $2^k$ is deallocated and its buddy is free, merge them into a block of size $2^{k+1}$
- Worst-case cost: $\log(\text{memory size})$
Cost of Allocation on a Heap

Single free list
- Linear cost to find a block to accommodate each allocation request

Buddy system
- Blocks of size $2^{n_0}, 2^{n_0+1}, 2^{n_0+2}, \ldots$
- Separate free list for blocks of size $2^k$, for each $k$
- If block of size $2^k$ is unavailable, split block of size $2^{k+1}$
- If block of size $2^k$ is deallocated and its buddy is free, merge them into a block of size $2^{k+1}$
- Worst-case cost: $\log(\text{memory size})$
Cost of Allocation on a Heap

Single free list
- Linear cost to find a block to accommodate each allocation request

Buddy system
- Blocks of size $2^{n_0}, 2^{n_0+1}, 2^{n_0+2}, \ldots$
- Separate free list for blocks of size $2^k$, for each $k$
- If block of size $2^k$ is unavailable, split block of size $2^{k+1}$
- If block of size $2^k$ is deallocated and its buddy is free, merge them into a block of size $2^{k+1}$
- Worst-case cost: $\log(\text{memory size})$
Cost of Allocation on a Heap

Single free list
- Linear cost to find a block to accommodate each allocation request

Buddy system
- Blocks of size $2^{n_0}$, $2^{n_0+1}$, $2^{n_0+2}$, …
- Separate free list for blocks of size $2^k$, for each $k$
- If block of size $2^k$ is unavailable, split block of size $2^{k+1}$
- If block of size $2^k$ is deallocated and its buddy is free, merge them into a block of size $2^{k+1}$
- Worst-case cost: $\log(\text{memory size})$
Cost of Allocation on a Heap

Single free list

- Linear cost to find a block to accommodate each allocation request

Buddy system

- Blocks of size $2^{n_0}, 2^{n_0+1}, 2^{n_0+2}, \ldots$
- Separate free list for blocks of size $2^k$, for each $k$
- If block of size $2^k$ is unavailable, split block of size $2^{k+1}$
- If block of size $2^k$ is deallocated and its buddy is free, merge them into a block of size $2^{k+1}$
- Worst-case cost: $\log(\text{memory size})$
Cost of Allocation on a Heap

Single free list

- Linear cost to find a block to accommodate each allocation request

Buddy system

- Blocks of size $2^{n_0}, 2^{n_0+1}, 2^{n_0+2}, \ldots$
- Separate free list for blocks of size $2^k$, for each $k$
- If block of size $2^k$ is unavailable, split block of size $2^{k+1}$
- If block of size $2^k$ is deallocated and its buddy is free, merge them into a block of size $2^{k+1}$
- Worst-case cost: $\log(\text{memory size})$
Cost of Allocation on a Heap

Single free list
- Linear cost to find a block to accommodate each allocation request

Buddy system
- Blocks of size $2^{n_0}, 2^{n_0+1}, 2^{n_0+2}, \ldots$
- Separate free list for blocks of size $2^k$, for each $k$
- If block of size $2^k$ is unavailable, split block of size $2^{k+1}$
- If block of size $2^k$ is deallocated and its buddy is free, merge them into a block of size $2^{k+1}$
- Worst-case cost: $\log(\text{memory size})$
Cost of Allocation on a Heap

Single free list
- Linear cost to find a block to accommodate each allocation request

Buddy system
- Blocks of size $2^{n_0}, 2^{n_0+1}, 2^{n_0+2}, \ldots$
- Separate free list for blocks of size $2^k$, for each $k$
- If block of size $2^k$ is unavailable, split block of size $2^{k+1}$
- If block of size $2^k$ is deallocated and its buddy is free, merge them into a block of size $2^{k+1}$
- Worst-case cost: $\log(\text{memory size})$
Cost of Allocation on a Heap

Single free list
- Linear cost to find a block to accommodate each allocation request

Buddy system
- Blocks of size $2^{n_0}, 2^{n_0+1}, 2^{n_0+2}, \ldots$
- Separate free list for blocks of size $2^k$, for each $k$
- If block of size $2^k$ is unavailable, split block of size $2^{k+1}$
- If block of size $2^k$ is deallocated and its buddy is free, merge them into a block of size $2^{k+1}$
- Worst-case cost: $\log(\text{memory size})$
Cost of Allocation on a Heap

**Single free list**
- Linear cost to find a block to accommodate each allocation request

**Buddy system**
- Blocks of size $2^{n_0}, 2^{n_0+1}, 2^{n_0+2}, \ldots$
- Separate free list for blocks of size $2^k$, for each $k$
- If block of size $2^k$ is unavailable, split block of size $2^{k+1}$
- If block of size $2^k$ is deallocated and its buddy is free, merge them into a block of size $2^{k+1}$
- Worst-case cost: $\log(\text{memory size})$
Cost of Allocation on a Heap

Single free list

- Linear cost to find a block to accommodate each allocation request

Buddy system

- Blocks of size $2^{n_0}, 2^{n_0+1}, 2^{n_0+2}, \ldots$
- Separate free list for blocks of size $2^k$, for each $k$
- If block of size $2^k$ is unavailable, split block of size $2^{k+1}$
- If block of size $2^k$ is deallocated and its buddy is free, merge them into a block of size $2^{k+1}$
- Worst-case cost: $\log(\text{memory size})$
Cost of Allocation on a Heap

Single free list
- Linear cost to find a block to accommodate each allocation request

Buddy system
- Blocks of size $2^{n_0}, 2^{n_0+1}, 2^{n_0+2}, \ldots$
- Separate free list for blocks of size $2^k$, for each $k$
- If block of size $2^k$ is unavailable, split block of size $2^{k+1}$
- If block of size $2^k$ is deallocated and its buddy is free, merge them into a block of size $2^{k+1}$
- Worst-case cost: $\log(\text{memory size})$
Cost of Allocation on a Heap

Single free list

• Linear cost to find a block to accommodate each allocation request

Buddy system

• Blocks of size $2^{n_0}, 2^{n_0+1}, 2^{n_0+2}, \ldots$
• Separate free list for blocks of size $2^k$, for each $k$
• If block of size $2^k$ is unavailable, split block of size $2^{k+1}$
• If block of size $2^k$ is deallocated and its buddy is free, merge them into a block of size $2^{k+1}$
• Worst-case cost: $\log(\text{memory size})$
Cost of Allocation on a Heap

Single free list
- Linear cost to find a block to accommodate each allocation request

Buddy system
- Blocks of size $2^{n_0}, 2^{n_0+1}, 2^{n_0+2}, \ldots$
- Separate free list for blocks of size $2^k$, for each $k$
- If block of size $2^k$ is unavailable, split block of size $2^{k+1}$
- If block of size $2^k$ is deallocated and its buddy is free, merge them into a block of size $2^{k+1}$
- Worst-case cost: $\log(\text{memory size})$

Fibonacci heap
- Block sizes that are Fibonacci numbers: 1, 1, 2, 3, 5, 8, 13, 21, 34, \ldots
- Less fragmentation
Deallocation on a Heap

Explicit deallocation by programmer

• Used in Pascal, C, C++, ...
• Efficient
• May lead to bugs that are difficult to find:
  – Dangling pointers/references from deallocating too soon
  – Memory leaks from not deallocating

Automatic deallocation by garbage collector

• Used in Java, functional and logic programming languages, ...
• Can add significant runtime overhead
• Safer
Scopes

The **scope of a binding** is the region of a program or time interval(s) in the program’s execution during which the binding is active.

A **scope** is a maximal region of the program where no bindings are destroyed (e.g., body of a procedure).

**Lexical (static) scoping**
- Binding based on nesting of blocks
- Can be determined at compile time

**Dynamic scoping**
- Binding depends on flow of execution at run time
- Can be determined only at run time
Lexical Scope

The current binding for a name is the one encountered in the smallest enclosing lexical unit.

Lexical units

- Packages, modules, source files
- Classes
- Procedures and nested subroutines
- Blocks
- Records and structures
**Lexical Scope**

The current binding for a name is the one encountered in the smallest enclosing lexical unit.

**Lexical units**

- Packages, modules, source files
- Classes
- Procedures and nested subroutines
- Blocks
- Records and structures

**Variant:** The current binding for a name is the one encountered in the smallest enclosing lexical unit and preceding the current point in the program text.

**Examples:**

- C, Java, Prolog, Scheme, ...
Lexical Scoping in Scheme

The scope of the red variable is shaded in each example.

(... (define x ...) fun1 fun2 ... funk)

(lambda (x ...) fun1 fun2 ... funk)

(let ((x exp1) (y exp2) (z exp3)) fun1 fun2 ... funk)

(let* ((x exp1) (y exp2) (z exp3)) fun1 fun2 ... funk)

(letrec ((x exp1) (y exp2) (z exp3)) fun1 fun2 ... funk)
(define (fun a b)
  (let ((x '())
         (y '())
         (z '()))
    (set! x (+ a b))
    (let ((a 3)
           (b 4))
      (let ((a 5)
             (b 6))
        (set! y (+ a b)))
      (set! z (+ a b)))
  (list x y z)))

What does (fun 1 2) return?
Lexical Scoping in Scheme: Example

(define (fun a b)
  (let ((x '())
        (y '())
        (z '()))
    (set! x (+ a b))
    (let ((a 3)
           (b 4))
      (let ((a 5)
            (b 6))
        (set! y (+ a b)))
    (set! z (+ a b)))
  (list x y z)))

What does (fun 1 2) return?
'(3 11 7)
procedure P1( A1 : T1 );
  var X : real;
procedure P2( A2 : T2 );
  procedure P3( A3 : T3 );
    begin
      ...
    end;
  begin
    ...
  end;
procedure P4( A4 : T4 );
  function F1( A5: T5 ) : T6;
    var X : integer;
    begin
      ...
    end;
  begin
    ...
  end;
begin
  ...
end;
Procedure P1 (A1 : T1);
  var X : real;
  procedure P2 (A2 : T2);
    procedure P3 (A3 : T3);
    begin
      ...
      end;
      begin
      ...
      end;
  procedure P4 (A4 : T4);
    function F1 (A5 : T5) : T6;
    var X : integer;
    begin
      ...
      end;
    begin
      ...
    end;
begin
  ...
end;
Lexical Scoping in Pascal

procedure P1( A1 : T1 );
  var X : real;
procedure P2( A2 : T2 );
  procedure P3( A3 : T3 );
  begin
    ...
    end;
  begin
    ...
  end;
procedure P4( A4 : T4 );
  function F1( A5 : T5 ) : T6;
  var X : integer;
  begin
    ...
    end;
  begin
    ...
  end;
begin
  ...
end;

What’s visible inside P1?
  P1, A1, X₁, P2, P4

What’s visible inside P2?
  P1, A1, X₁, P2, P3, A2

What’s visible inside P3?

What’s visible inside P4?

What’s visible inside F1?
procedure P1( A1 : T1 );
var X : real;
procedure P2( A2 : T2 );
  procedure P3( A3 : T3 );
  begin
    ...
  end;
begin
  ...
end;
procedure P4( A4 : T4 );
function F1( A5: T5 ) : T6;
var X : integer;
begin
  ...
end;
begin
  ...
end;

What’s visible inside P1?
P1, A1, X1, P2, P4

What’s visible inside P2?
P1, A1, X1, P2, P3, A2

What’s visible inside P3?
P1, A1, X1, P2, P3, A2, A3

What’s visible inside P4?

What’s visible inside F1?
Lexical Scoping in Pascal

```pascal
procedure P1( A1 : T1 );
  var X : real;
procedure P2( A2 : T2 );
  procedure P3( A3 : T3 );
    begin
      ...
    end;
begin
  ...
end;
procedure P4( A4 : T4 );
  function F1( A5: T5 ) : T6;
    var X : integer;
    begin
      ...
    end;
begin
  ...
end;
```

What’s visible inside P1?
- P1, A1, X, P2, P4

What’s visible inside P2?
- P1, A1, X, P2, P3, A2

What’s visible inside P3?
- P1, A1, X, P2, P3, A2, A3

What’s visible inside P4?
- P1, A1, X, P2, P4, A4, F1

What’s visible inside F1?
procedure P1( A1 : T1 );
    var X : real;
procedure P2( A2 : T2 );
    procedure P3( A3 : T3 );
    begin
        ...
        end;
    begin
        ...
    end;
procedure P4( A4 : T4 );
    function F1( A5: T5 ) : T6;
    var X : integer;
    begin
        ...
        end;
    begin
        ...
    end;
begin
    ...
end;
Implementation of Lexical Scoping: Static Chains

The stack frame of each invocation has a static link to the stack frame of the most recent invocation of the lexically surrounding subroutine.

To reference a variable in some outer scope, the chain of static links is traversed, followed by adding the offset of the variable relative to the stack frame it is in.

Example

- Access variable \( x \) in procedure \( A \) from within procedure \( C \).
Dynamic Scope

The current binding for a given name is the one

- encountered most recently during execution,
- not hidden by another binding for the same name, and
- not yet destroyed by exiting its scope.

```plaintext
# Static scoping
sub f {  # Dynamic scoping
    my $a = 1;
    print "f:$a\n";
    &printa();
}
sub printa {print "p:$a\n";}
$a = 2;
&f();

# Dynamic scoping
sub g {
    local $a = 1;
    print "g:$a\n";
    &printa();
}
sub printa {print "p:$a\n";}
$a = 2;
&g();
```
Lexical vs Dynamic Scoping: Example

a : integer -- global declaration

procedure first
    a := 1

procedure second
    a : integer -- local declaration
    first()

a := 2

if read_integer() > 0
    second()
else
    first()

write_integer(a)

What does this program print

• Under lexical scoping?
• Under dynamic scoping?
Lexical vs Dynamic Scoping: Example

a : integer -- global declaration

procedure first
  a := 1

procedure second
  a : integer -- local declaration
  first()

a := 2

if read_integer() > 0
  second()
else
  first()

write_integer(a)

What does this program print
  • Under lexical scoping?
  • Under dynamic scoping?

Dynamic scoping is usually a bad idea!

Dynamic scoping is usually a bad idea!
Shallow and Deep Binding

If a subroutine is passed as a parameter, when are the free variables bound?

**Shallow binding**

- When the routine is called.

**Deep binding**

- When the routine is first passed as a parameter.

This is important using both static and dynamic scoping and is known as the **funarg problem**.
int x = 10;

function f( int a ) {
    x = x + a;
}

function g( function h ) {
    int x = 30;
    h( 100 );
    print( x );
}

function main() {
    g( f );
    print( x );
}
int x = 10;

function f( int a ) {
    x = x + a;
}

function g( function h ) {
    int x = 30;
    h( 100 );
    print( x );
}

function main() {
    g( f );
    print( x );
}

What is the output of this program using deep binding?
30
110

What is the output of this program using shallow binding?
int $x = 10$;

function $f(\ int \ a )$ {
    $x = x + a$;
}

function $g(\ function \ h )$ {
    int $x = 30$;
    $h( 100 )$;
    print( $x$ );
}

function $main()$ {
    $g( f )$;
    print( $x$ );
}

What is the output of this program using deep binding?

30
110

What is the output of this program using shallow binding?

130
10
Example 1 of the funarg Problem

(define x 1)

(define increase_x
  (lambda ()
    (set! x (+ x 1)))))

(define execute
  (lambda (f)
    (let ((x 20))
      (display (list "inner x before:" x))
      (f)
      (display (list "inner x after:" x)))))

(display (list "outer x before:" x))
(execute increase_x)
(display (list "outer x after:" x))
Example 1 of the funarg Problem

\[(\text{define } x \ 1)\]
\[(\text{define } \text{increase}_x\ (\lambda ()\ (\text{set! } x \ (+ x \ 1))))\]
\[(\text{define } \text{execute}\ (\lambda (f)\ (\text{let } ((x \ 20))\ (\text{display} \ (\text{list} \ "\text{inner }x \ \text{before:}\ x))\ (f)\ (\text{display} \ (\text{list} \ "\text{inner }x \ \text{after:} \ x)))\)))\]
\[(\text{display} \ (\text{list} \ "\text{outer }x \ \text{before:}\ x))\]
\[(\text{execute } \text{increase}_x)\]
\[(\text{display} \ (\text{list} \ "\text{outer }x \ \text{after:} \ x))\]

This prints
\[(\text{outer }x \ \text{before:} \ 1)\]
\[(\text{inner }x \ \text{before:} \ 20)\]
\[(\text{inner }x \ \text{after:} \ 20)\]
\[(\text{outer }x \ \text{after:} \ 2)\]

which is what one would expect, given the lexical scoping of Scheme (the \(x\) inside \text{execute} should not be visible to \text{increase}_x).
Example 2 of the funarg Problem

type person = record
    age : integer;
end;

(* age threshold *)
threshold : integer;
people : database;

function older_than( p : person )
    : boolean
begin
    return p.age >= threshold
end;

procedure print_person( p : person )
begin
    (* use line_length to format data *)
end;

procedure print_selected_records(
    db : database;
    predicate, print_routine : procedure )
var line_length : integer;
begin
    if device_type( stdout ) = terminal
        then line_length := 80
        else line_length := 132;
    for each record r in db
        if predicate( r )
            then print_routine( r );
    threshold := 35;
    print_selected_records( people, older_than, print_person );

Function older_than expects deep binding (but not necessarily lexical scoping).

Procedure print_person assumes shallow binding (and dynamic scoping).
Example 2 of the funarg Problem

type person = record
  age : integer;
end;

(* age threshold *)
threshold : integer;
people : database;

function older_than( p : person ) : boolean
begin
  return p.age >= threshold
end;

procedure print_person( p : person )
begin
  (* use line_length to format data *)
end;

procedure print_selected_records(
  db : database;
  predicate, print_routine : procedure )
var line_length : integer;
begin
  if device_type( stdout ) = terminal
    then line_length := 80
    else line_length := 132;
  for each record r in db
    if predicate( r )
      then print_routine( r );
  end;

  threshold := 35;
  print_selected_records(
    people, older_than, print_person );

This is an example of terrible programming style: the behaviour of both older_than and print_person depends on variable values not explicitly associated with them.

Function older_than should take the age threshold as an argument.

Procedure print_person and its associated settings should be wrapped in a class.
Subroutine Closures

When using deep binding, a closure is a bundle of

- A referencing environment and
- A reference to the subroutine.

Deep binding is

- The default in statically scoped languages and
- An option in dynamically scoped languages.

Closures become really interesting when returning functions as the results of function calls:

- When the function is invoked, the scope it refers to may no longer exist and thus needs to be preserved explicitly in the closure.
- This can be used to implement poor man’s objects.
Subroutine Closures: Example

(define (new-stack)
  (let ((stack '()))
    (lambda (op . args)
      (cond ((eq? op 'push)
              (set! stack (append (reverse args) stack)))
            ((eq? op 'pop)
             (let ((top (car stack)))
                (set! stack (cdr stack))
                top))
            ((eq? op 'empty)
             (null? stack)))))))

> (define st1 (new-stack))  > (st1 'pop)
> (define st2 (new-stack))   3
> (st1 'push 1 2 3)          > (st2 'pop)
> (st1 'empty)              4
#f
> (st2 'empty)              > (st1 'pop)
#t
> (st2 'push 4)             1
A **frame** is a collection of variable-object bindings.

Frames can point to “parent frames”, which allows the construction of a chain of frames.

A **referencing environment** is a chain of frames represented by pointing to the first frame in the chain.

A variable $x$ in an environment $E$ is unbound if none of $E$’s frames binds $x$. Otherwise its value is the value bound to $x$ in the first frame that provides such a binding.

What are the values of $x$ and $y$ in environments $A$ and $B$, respectively?

$\begin{align*}
A & : x: 3 \\
  & : y: 5 \\
\end{align*}$

$\begin{align*}
B & : z: 6 \\
  & : x: 7 \\
& : m: 1 \\
  & : y: 2 \\
\end{align*}$
A **frame** is a collection of variable-object bindings.

Frames can point to “parent frames”, which allows the construction of a chain of frames.

A **referencing environment** is a chain of frames represented by pointing to the first frame in the chain.

A variable $x$ in an environment $E$ is unbound if none of $E$’s frames binds $x$. Otherwise its value is the value bound to $x$ in the first frame that provides such a binding.

A **closure** is the code of a function paired with a pointer to a referencing environment.

---

What are the values of $x$ and $y$ in environments $A$ and $B$, respectively?

**Example of a closure**

```
(define (square x) (* x x))
```

Parameters: $x$

Body: $(* x x)$
(define (square x)
  (* x x))

(define (sum-of-squares x y)
  (+ (square x) (square y)))

(define (f a)
  (sum-of-squares (+ a 1) (* a 2)))
Closures: Extended Example (2)

```latex
\text{square:} \quad \text{sum-of-squares:} \quad \text{f:}
```

- Parameters: \( a \)
  - Body: \((\text{sum-of-squares} (+ a 1) (* a 2))\)

- Parameters: \( x, y \)
  - Body: \(+ (\text{square} x) (\text{square} y)\)

- Parameters: \( x \)
  - Body: \((* x x)\)

Invocation \((f 5)\)

- \(E_1\) \(a: 5\)
  - \((\text{sum-of-squares} (+ a 1) (* a 2))\)

- \(E_2\) \(x: 6\), \(y: 10\)
  - \(+ (\text{square} x) (\text{square} y)\)

- \(E_3\) \(x: 6\)
  - \((* x x)\)

- \(E_4\) \(x: 10\)
  - \((* x x)\)
Closures Simulate Objects

(define new-counter
  (lambda ()
    (define c 0)
    (define counter (lambda ()
                       (set! c (+ c 1))
                       c))
    counter))

new-counter:

Parameters: —
Body: (define c 0)
  (define counter
    (lambda ()
      (set! c (+ c 1))
      c))
counter
Closures Simulate Objects

```
(define new-counter
  (lambda ()
    (define c 0)
    (define counter (lambda ()
                        (set! c (+ c 1))
                        c))
    counter))

> (define cnta (new-counter))
```

Parameters: —

Body: (define c 0)
  (define counter (lambda ()
                        (set! c (+ c 1))
                        c))
  counter
Closures Simulate Objects

```
(define new-counter
  (lambda ()
    (define c 0)
    (define counter (lambda ()
                      (set! c (+ c 1))
                      c))
    counter))
```

```
(define cnta (new-counter))
```

Diagram:

```
new-counter: 0
Parameters: —
Body: (define c 0)
  (define counter
    (lambda ()
      (set! c (+ c 1))
      c))
    counter
```

Names, Scopes, and Binding
Closures Simulate Objects

\[ (\text{define new-counter} \quad \text{->} \quad (\text{define cnta (new-counter)}) \]

\[ (\text{define new-counter} \quad \text{(lambda} () \quad \text{(define c 0)}) \quad \text{(define counter (lambda} () \quad \text{(set! c (+ c 1)) \text{)} \quad \text{c)} \quad \text{counter})) \]

Diagram:

- **Parameters:** —
- **Body:** (define c 0)
  - (define counter (lambda ()
    - (set! c (+ c 1))
    - c)
  - counter

- **Parameters:** —
- **Body:** (set! c (+ c 1))
  - c

Names, Scopes, and Binding
CSCI 3136: Principles of Programming Languages
Closures Simulate Objects

Parameters: —
Body: (define new-counter
   (lambda ()
      (define c 0)
      (define counter (lambda ()
         (set! c (+ c 1))
         c))
      counter))

> (define cnta (new-counter))
> (cnta)
1

new-counter:  •
cnta:  •
Parameters: —
Body: (define c 0)
   (define counter (lambda ()
      (set! c (+ c 1))
      c))
counter

Parameters: —
Body: (set! c (+ c 1))
c

Names, Scopes, and Binding
CSCI 3136: Principles of Programming Languages
Closures Simulate Objects

(define new-counter
  (lambda ()
    (define c 0)
    (define counter (lambda ()
                       (set! c (+ c 1))
                       c))
    counter))

(new-counter)
> (define cnta (new-counter))
> (cnta)
1

Parameters: —
Body: (define c 0)
  (define counter (lambda ()
                    (set! c (+ c 1))
                    c))
  counter

Parameters: —
Body: (set! c (+ c 1))
c

Names, Scopes, and Binding
CSCI 3136: Principles of Programming Languages
Closures Simulate Objects

```
(define new-counter
  (lambda ()
    (define c 0)
    (define counter (lambda ()
      (set! c (+ c 1))
      c))
    counter))

> (define cnta (new-counter))
> (cnta)
1
```

```
Name: new-counter
Parameters: —
Body: (define c 0)
  (define counter (lambda ()
    (set! c (+ c 1))
    c))
  counter

Name: cnta
Parameters: —
Body: (set! c (+ c 1))
  c
```
Closures Simulate Objects

(define new-counter
  (lambda ()
    (define c 0)
    (define counter (lambda ()
                      (set! c (+ c 1))
                      c))
    counter))

> (define cnta (new-counter))
> (cnta)
  1

new-counter:  cnta:  
Parameters: — Body: (define c 0)
(dangle)
(dangle)
(dangle)
(dangle)
(dangle)
(dangle)

<table>
<thead>
<tr>
<th>Parameters:  —</th>
<th>Body: (define c 0)</th>
<th>Parameters:  —</th>
<th>Body: (set! c (+ c 1))</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(define c 0)</td>
<td></td>
<td>(set! c (+ c 1))</td>
</tr>
<tr>
<td></td>
<td>(define counter</td>
<td></td>
<td>c</td>
</tr>
<tr>
<td></td>
<td>(lambda ()</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>(set! c (+ c 1))</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>c)</td>
<td></td>
<td>c</td>
</tr>
</tbody>
</table>

(CSCI 3136: Principles of Programming Languages)
Closures Simulate Objects

(define new-counter
  (lambda ()
    (define c 0)
    (define counter (lambda ()
                      (set! c (+ c 1))
                      c))
    counter))

> (define cnta (new-counter))
> (cnta)
  1
> (cnta)
  2

new-counter:  cnta:  
Parameters: —  Body: (define c 0)  
Body: (define counter
      (lambda ()
        (set! c (+ c 1))
        c))
      counter

Parameters: —  Body: (set! c (+ c 1))
               c
Closures Simulate Objects

\[(\text{define } \text{new-counter} \ (\lambda () \ (\text{define } c \ 0) \ (\text{define } \text{counter} \ (\lambda () \ (\text{set!} \ c \ (+ \ c \ 1)) \ c)) \ \text{counter}))\]
Closures Simulate Objects

```
(define new-counter
  (lambda ()
    (define c 0)
    (define counter (lambda ()
      (set! c (+ c 1))
      c))
    counter))

(define cnta (new-counter))
> (define cnta (new-counter))
> (cnta)
1
> (cnta)
2
```

Diagram:

```
new-counter:
  cnta:

Parameters: —
Body: (define c 0)
  (define counter (lambda ()
    (set! c (+ c 1))
    c))
  counter

Parameters: —
Body: (set! c (+ c 1))
  c
```
Closures Simulate Objects

(define new-counter
  (lambda ()
    (define c 0)
    (define counter (lambda ()
                      (set! c (+ c 1))
                      c))
    counter))

> (define cnta (new-counter))
> (cnta)
1
> (cnta)
2

new-counter: •
cnta: •

counter
Parameters: —
Body: (define c 0)
  (define counter
    (lambda ()
      (set! c (+ c 1))
      c))
c
Parameters: —
Body: (set! c (+ c 1))
c
Closures Simulate Objects

(define new-counter
  (lambda ()
    (define c 0)
    (define counter (lambda ()
                       (set! c (+ c 1))
                       c))
    counter))

> (define cnta (new-counter))
> (cnta) 1
> (cnta) 2
> (define cntb (new-counter))

new-counter:
  cnta:
    Parameters: —
    Body: (define c 0)
       (define counter (lambda ()
                         (set! c (+ c 1))
                         c))
       counter

Parameters: —
Body: (set! c (+ c 1))
       c
(define new-counter
  (lambda ()
    (define c 0)
    (define counter (lambda ()
                      (set! c (+ c 1))
                      c))
    counter))

> (define cnta (new-counter))
> (cnta)
1
> (cnta)
2
> (define cntb (new-counter))

new-counter:
  cnta:

Parameters: —
Body: (define c 0)
  (define counter
    (lambda ()
      (set! c (+ c 1))
      c))
    counter

c : 2

Parameters: —
Body: (set! c (+ c 1))
  c

c : 0
(define new-counter
  (lambda ()
    (define c 0)
    (define counter (lambda ()
                     (set! c (+ c 1))
                     c))
    counter))
Closures Simulate Objects

\[(\text{define new-counter} \ (\lambda ()
\quad (\text{define } c \ 0)
\quad (\text{define counter} \ (\lambda ()
\quad \quad (\text{set!} \ c \ (+ \ c \ 1))
\quad \quad c))
\quad \text{counter}))\]

\[
> (\text{define cnta} \ (\text{new-counter}))
> (\text{cnta})
\]

1

\[
> (\text{define cntb} \ (\text{new-counter}))
> (\text{cntb})
\]

1

\[
\text{new-counter:} \quad \text{cnta:} \quad \text{cntb:}
\]

Parameters: —

Body: \( (\text{define } c \ 0) \)
\( (\text{define counter} \ (\lambda ()
\quad (\text{set!} \ c \ (+ \ c \ 1))
\quad c)) \)
\( \text{counter} \)
Closures Simulate Objects

(define new-counter
  (lambda ()
    (define c 0)
    (define counter (lambda ()
          (set! c (+ c 1))
          c))
    counter))

> (define cnta (new-counter))
> (cnta)
  1
> (cnta)
  2
> (define cntb (new-counter))
> (cntb)
  1

new-counter:   cnta:  cntb:

Parameters: —
Body: (define c 0)
  (define counter
    (lambda ()
      (set! c (+ c 1))
      c))
  counter

Parameters: —
Body: (set! c (+ c 1))
  c

Names, Scopes, and Binding
CSCI 3136: Principles of Programming Languages
Closures Simulate Objects

(define new-counter
  (lambda ()
    (define c 0)
    (define counter (lambda ()
      (set! c (+ c 1))
      c))
    counter))

> (define cnta (new-counter))
> (cnta)
  1
> (cnta)
  2
> (define cntb (new-counter))
> (cntb)
  1

new-counter:
  cnta:
  cntb:

Parameters: —
Body: (define c 0)
  (define counter
    (lambda ()
      (set! c (+ c 1))
      c))
counter

Parameters: —
Body: (set! c (+ c 1))
c
Closures Simulate Objects

(define new-counter
  (lambda ()
    (define c 0)
    (define counter (lambda ()
                       (set! c (+ c 1))
                       c))
    counter))
Modules

Motivation

• Enable programmers in a team to work independently on different parts of a project.

Requirements

• Modules need to interact with each other through well defined interfaces.
• Internals of modules should be hidden from other modules to avoid unwanted coupling.

From static variables to classes

• Static variables (in C) provide “private objects” to a single subroutine.
• Modules provide the same set of “private objects” to a group of subroutines. (Essentially a single instance of a class.)
• Module types can be instantiated, effectively acting like classes but without inheritance.
• Classes add inheritance to module types.
Each invocation produces a different “variable name” starting with letter l:

```c
void new_name( char *s, char l )
{
    /* This array is automatically filled with zeros
       when initialized */
    static short int name_nums[52];

    int index = l >= 'a' && l <= 'z'
        ? l - 'a'
        : l - 'A' + 26;
    sprintf( s, "%c%d\0", l, name_nums[ index ]++ );
}
```
Save time by compiling a regular expression only the first time it is used (using the regex library by Henry Spencer):

```c
int match_some_dates( char *s )
{
    static regexp *date = NULL;

    if( date == NULL )
    {
        date = regcomp( "[0-9][0-9]?? (Jan|Feb) 200[4-9]" );
    }

    return ( regexec( date, s ) == 0 );
}
```
MODULE stack;
IMPORT element, stack_size;
EXPORT push, pop;

TYPE stack_index = [1..stack_size];
VAR s : ARRAY stack_index OF element;
    top : stack_index;

PROCEDURE push( elem : element );
BEGIN ...
END;

PROCEDURE pop() element;
BEGIN ...
END;

BEGIN 
    top := 1;
END stack.

MODULE main;
TYPE element = ...;
CONST stack_size = ...;
FROM stack IMPORT push, pop;

VAR x, y : element;

BEGIN ...
    push( x );
    ...
    y := pop;
    ...
END main.
Visibility is specified using explicit IMPORT and EXPORT statements.

This is an example of closed scopes (as opposed to open scopes where bindings from “outside” are freely passed into the scope).

Closed scopes force programmer to clearly document the interface.

C has no support for modules.

Java, C#, Perl, Python, Ada, and Haskell provide selectively open scopes.
Constructs Similar to Modules

C: Separate compilation units

- Judicious export of variables and functions in include file can simulate EXPORT lists.
- No protection against name clashes between “modules”.

C++: Namespaces + separate compilation units

Java, Perl, Ada: Packages

Clu: Clusters
A Module Deficiency

The stack module cannot be used to provide multiple stacks to an application that requires them.

Solutions

• Not really a solution: duplicate the code over multiple modules with the same name.
• A module that provides explicit means to create, manage, and destroy multiple stacks.
  – Requires the stack as an argument to each stack function.
• Module types (e.g., Simula, Euclid), which allow instantiation of modules.
• Go all the way to classes.
Classes

Every instance of a module type or class has a separate copy of the module type’s or class’s variables.

Classes are module types + inheritance.

```java
public class stack {
    private int stack_size;
    private element[] s;
    private int top = 0;

    public void push( element x )
    {
        ...
    }

    public element pop()
    {
        ...
    }
}

stack A, B;
element x, y;
...
A.push( x );
...
y = B.pop();
...
```
Aliasing and Overloading

Aliasing

- More than one name bound to one object (references, pointers, ...).
- Makes compiler optimization more difficult.

Overloading

- One name bound to more than one object.
Aliasing and Overloading

Aliasing

• More than one name bound to one object (references, pointers, …).
• Makes compiler optimization more difficult.

Examples

```c
int a, b, *p, *q;
a = *p; *q = 3; b = *p;
```

Overloading

• One name bound to more than one object.
Aliasing and Overloading

Aliasing

• More than one name bound to one object (references, pointers, ...).
• Makes compiler optimization more difficult.

Examples

```c
int a, b, *p, *q;
int a = *p; *q = 3; b = *p;
```

```c
double sum, sum_of_squares;
void accumulate( double &x ) {
    sum += x; sum_of_squares += x * x;
}
accumulate( sum );
```

Overloading

• One name bound to more than one object.
Aliasing and Overloading

Aliasing

- More than one name bound to one object (references, pointers, ...).
- Makes compiler optimization more difficult.

Examples

```c
int a, b, *p, *q;
a = *p; *q = 3; b = *p;
```

```c
double sum, sum_of_squares;
void accumulate( double &x ) {
    sum += x; sum_of_squares += x * x;
}
accumulate( sum );
```

Overloading

- One name bound to more than one object.

Example

```c
int operator +( const int &a, const int &b );
string operator +( const string &a, const string &b );
```
Problems with Aliasing

Aliases make code more confusing and make resulting bugs hard to find.

Optimization of code becomes difficult if not impossible.
Problems with Aliasing

Aliases make code more confusing and make resulting bugs hard to find.

Optimization of code becomes difficult if not impossible.

**restrict** keyword in C99

- Used by programmer to tell the compiler that a given pointer is the only means used to update the memory location it references.
- Allows compiler to perform optimizations that would be impossible in the presence of aliasing.
- Resulting optimization may lead to even more obscure bugs if the programmer doesn’t keep their promise and introduces aliases for this pointer.
Overloading

Most languages have some form of overloading (e.g., arithmetic operators).

We normally do not think about this type of overloading, as we simply think about “doing math with numbers” and the right thing happens.
Overloading

Most languages have some form of overloading (e.g., arithmetic operators).

We normally do not think about this type of overloading, as we simply think about “doing math with numbers” and the right thing happens.

Overloading by programmer: C++, Java, C#, Ada

- C++, C#: A.operator+( B )
- Ada: "+( A, B )"
- FORTRAN90: interface construct to associate e.g. “+” with some binary function
Overloading

Most languages have some form of overloading (e.g., arithmetic operators).

We normally do not think about this type of overloading, as we simply think about “doing math with numbers” and the right thing happens.

Overloading by programmer: C++, Java, C#, Ada

- C++, C#: A.operator+( B )
- Ada: "+( A, B )
- FORTRAN90: interface construct to associate e.g. “+” with some binary function

This type of user-specified overloading makes a language more powerful and expressive with a potential to increase clarity (e.g., arithmetic operators for complex numbers).

It does, however, also have the potential for tremendous confusion if the behaviour associated with an overloaded operator does not match what one would intuitively expect this operator to do.
Mechanisms Related to Overloading

Coercion

- Compiler automatically converts an object into an object of another type when required.

   In Java, "" + o forces o.toString()
Mechanisms Related to Overloading

Coercion

- Compiler automatically converts an object into an object of another type when required.

  In Java, "" + o forces o.toString()

Polymorphism

- Single body of code
- Behaviour is customized

  sum :: Num a => [a] -> a
  sum [] = 0
  sum (x:xs) = x + sum xs
Coercion

• Compiler automatically converts an object into an object of another type when required.

In Java, "" + o forces o.toString()

Polymorphism

• Single body of code
• Behaviour is customized

\[
\text{sum} :: \text{Num} \ a \Rightarrow [a] \rightarrow a
\]
\[
\text{sum} \ [ ] \ = \ 0
\]
\[
\text{sum} \ (x:xs) \ = \ x + \text{sum} \ xs
\]

Generics (templates)

• Separate copies of the code generated by compiler for each type

\[
\text{std::vector< int > int_vec;}
\]
\[
\text{std::vector< char > char_vec;}
\]