



CS49K Programming Languages

Chapter 3 Name, Scope and Binding

LECTURE 5: MEMORY MODEL FOR PROGRAMMING LANGUAGES

DR. ERIC CHOU

IEEE SENIOR MEMBER

Objectives

- Variable names, binding and scope.
- Binding: Allocation of Memory
- Memory Models
- Scope Rules
- Reference Environments

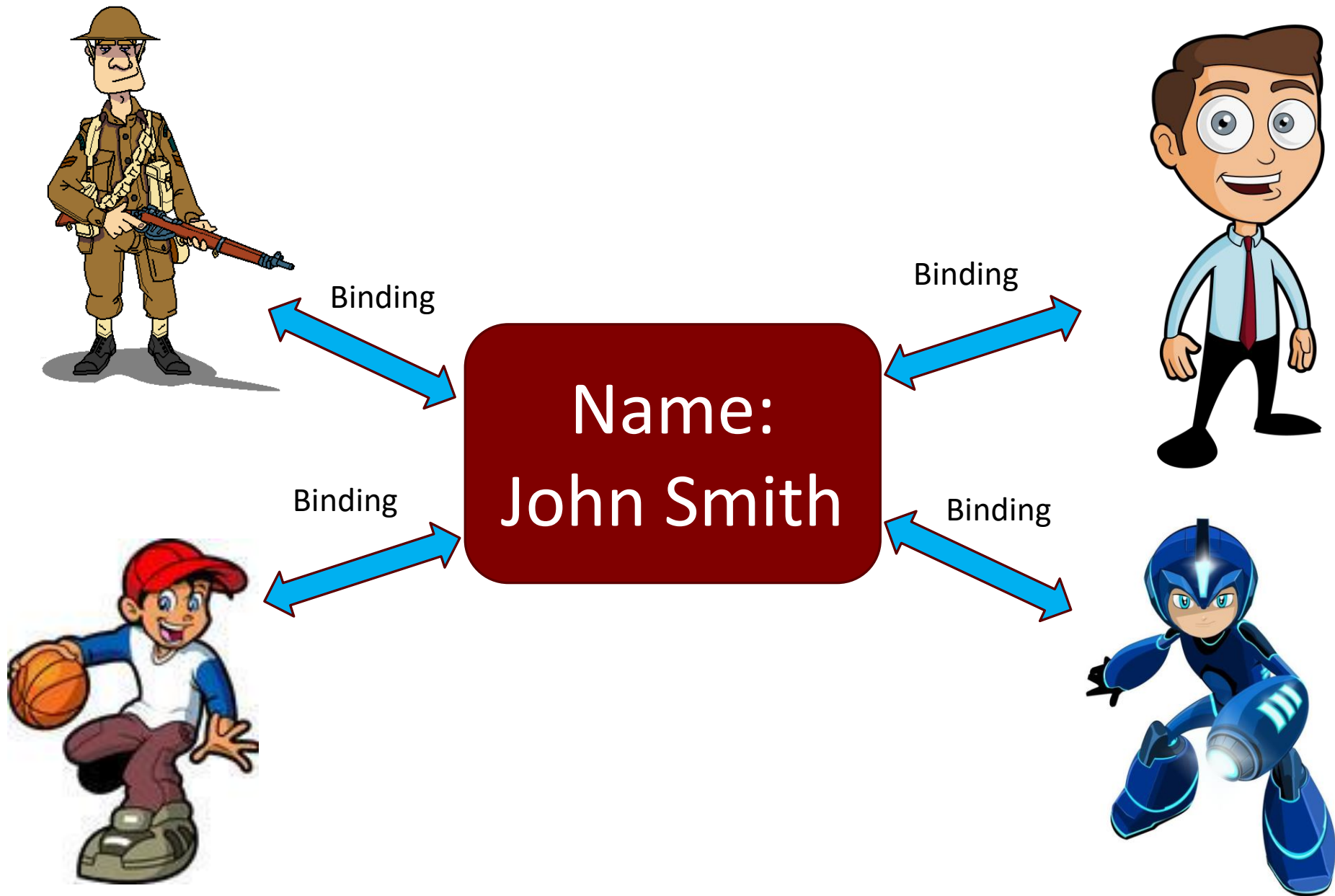
Overview of Names, Scopes, and Bindings

SECTION 1

Name, Scope, and Binding

id, data lifetime, and data memory association

- A name is exactly what you think it is
 - Most names are identifiers
 - symbols (like '+') can also be names
- A binding is an association between two things, such as a name and the thing it names
- The scope of a binding is the part of the program (textually) in which the binding is active



Name Scope Binding

Static and Instance Members:

Static and Instance Variable

Visibility:

public	+
protected	#
default	~
private	-

Global and Local:

Name Space

Package Level

Module Level

Functional Level

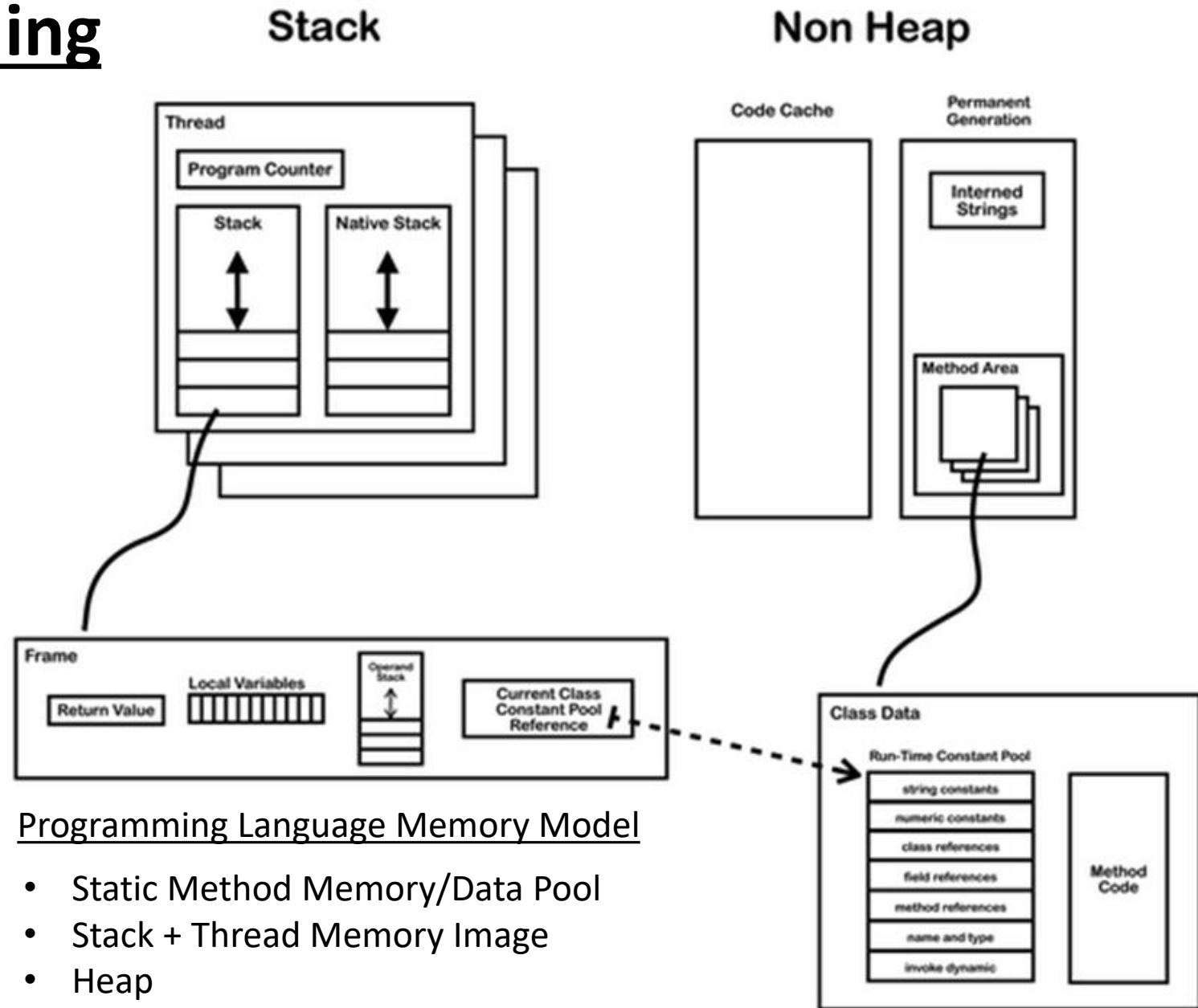
Parameter List

Functional Level

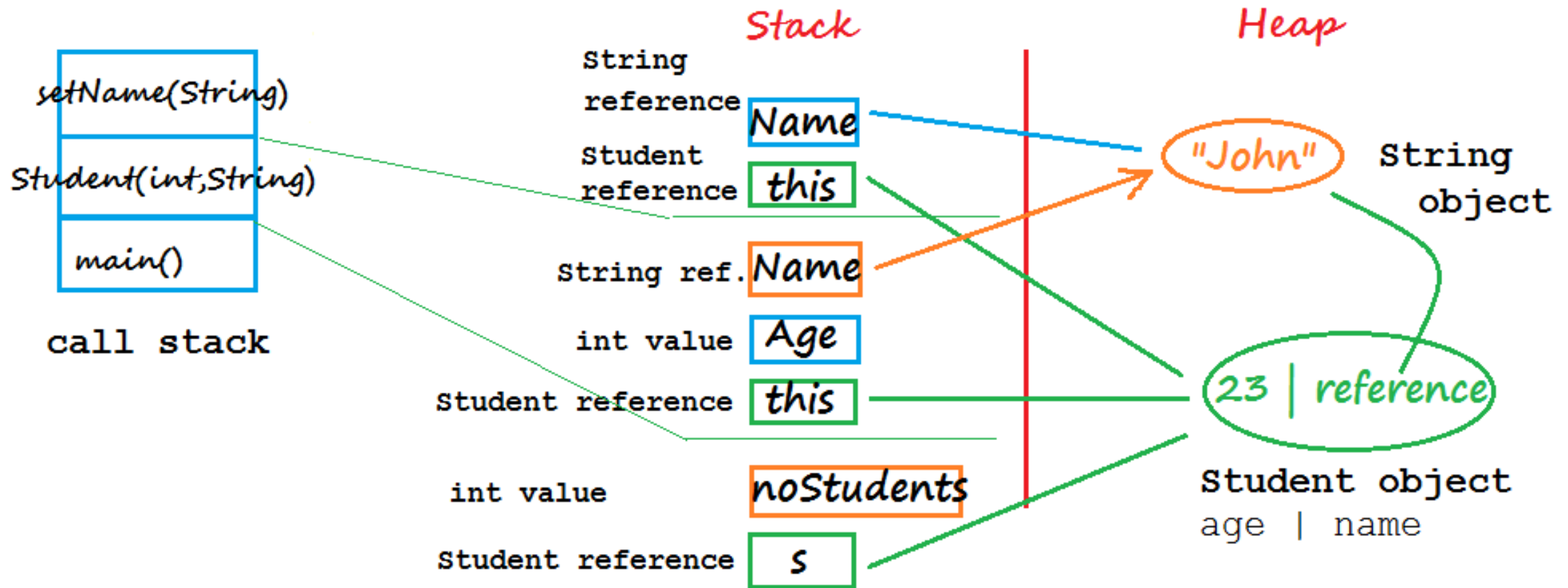
Loop Level

Block Level

Scope Related Language Struct



Name, Scope, and Binding



Name, Scope, and Binding

- Binding Time is the point at which a binding is created or, more generally, the point at which any implementation decision is made
 - language design time
 - program structure, possible type
 - language implementation time
 - I/O, arithmetic overflow, type equality (if unspecified in manual)

Static vs. Dynamic Binding

- **Binding**

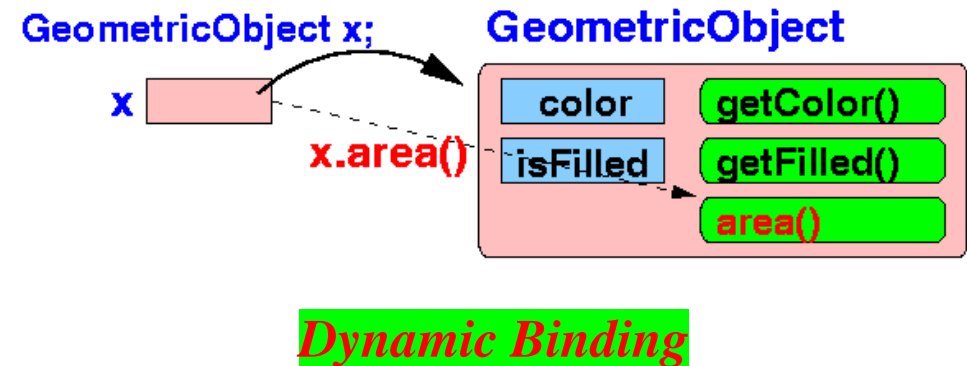
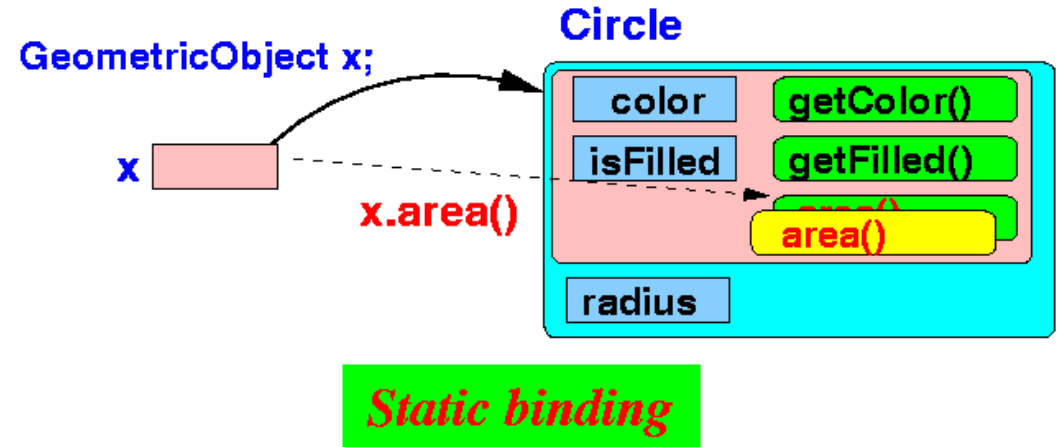
The determination of which method in the class hierarchy is to be used for a particular object.

- **Static (Early) Binding**

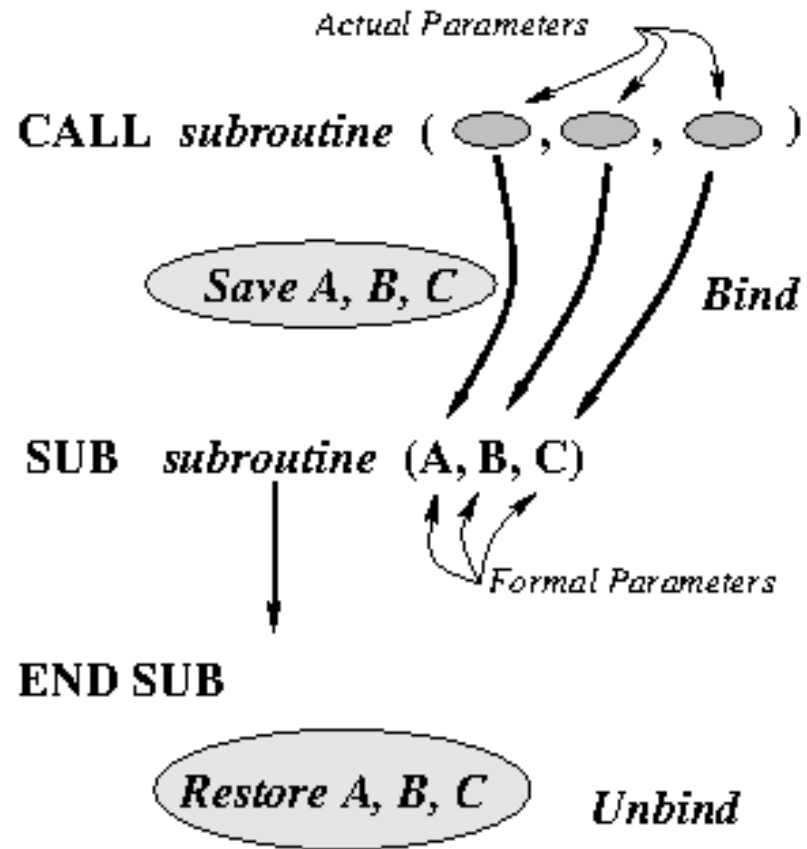
When the compiler can determine which method in the class hierarchy to use for a particular object.

- **Dynamic (Late) Binding**

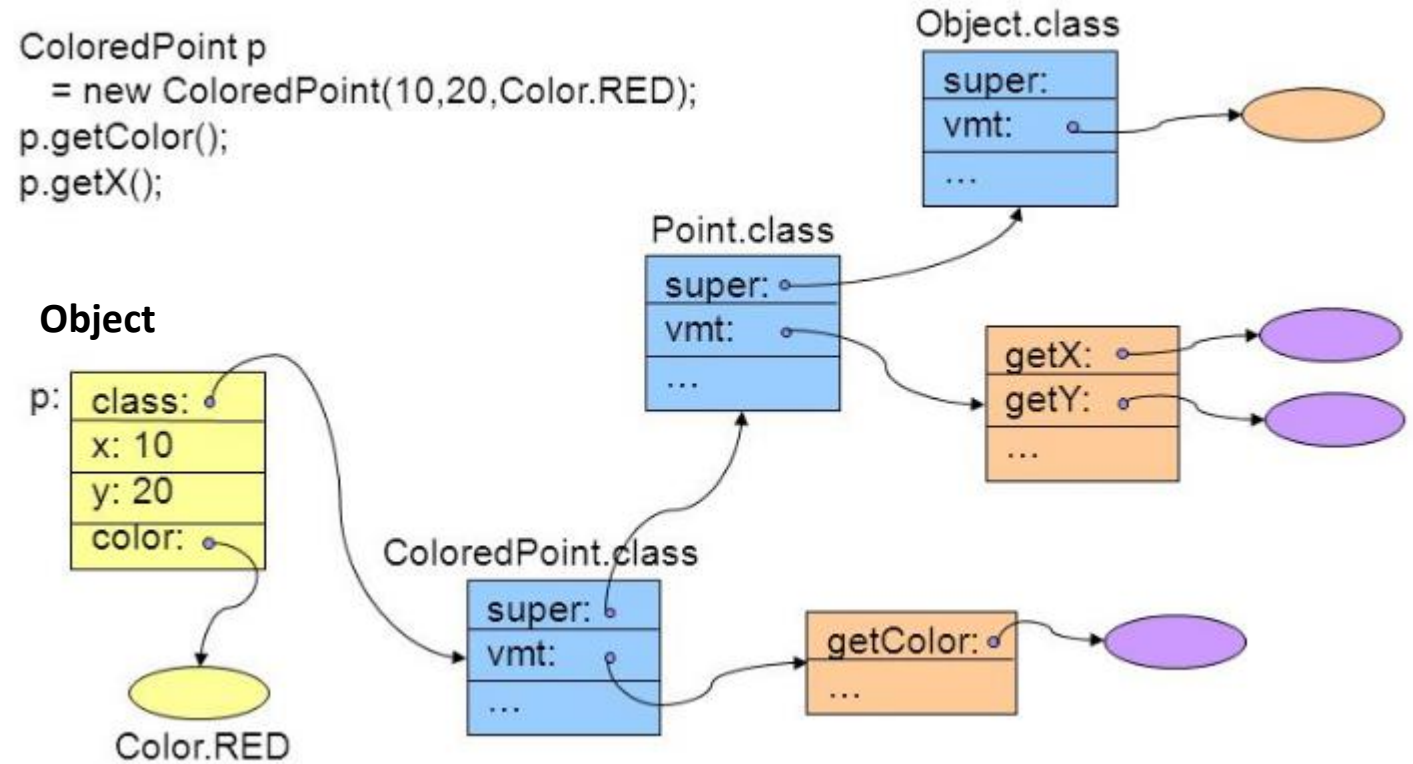
When the determination of which method in the class hierarchy to use for a particular object occurs during program execution.



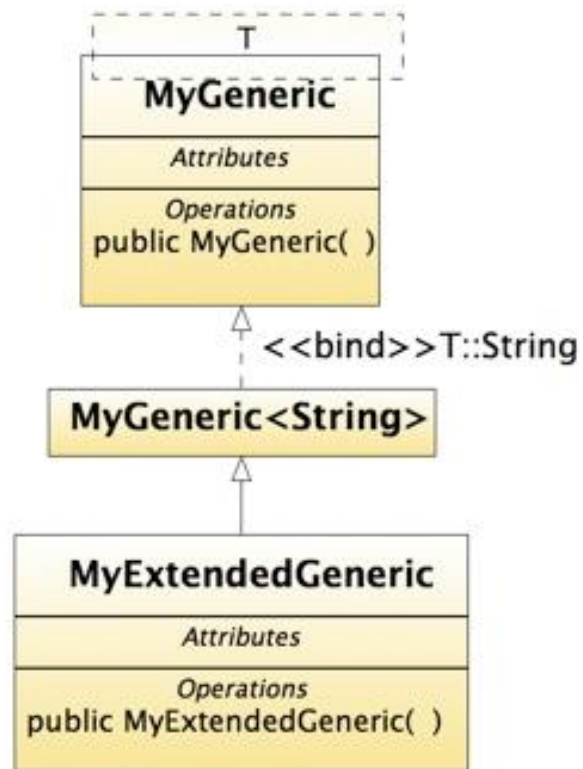
Parametric Binding



Polymorphic Method Binding



Parametric Type Binding



Type variable

General Definitions

■ Parametric polymorphism

```
public class NonGenericBox<T> {  
    private T object;  
  
    public void set(final T object) {  
        this.object = object;  
    }  
  
    public T get() {  
        return this.object;  
    }  
}  
  
public void useOfNonGenericBox() {  
    final NonGenericBox<String> aNonGenericBox = new NonGenericBox<String>();  
    aNonGenericBox.set(new String());  
    final String myString = (String) aNonGenericBox.get();  
    System.out.println(myString);  
}
```

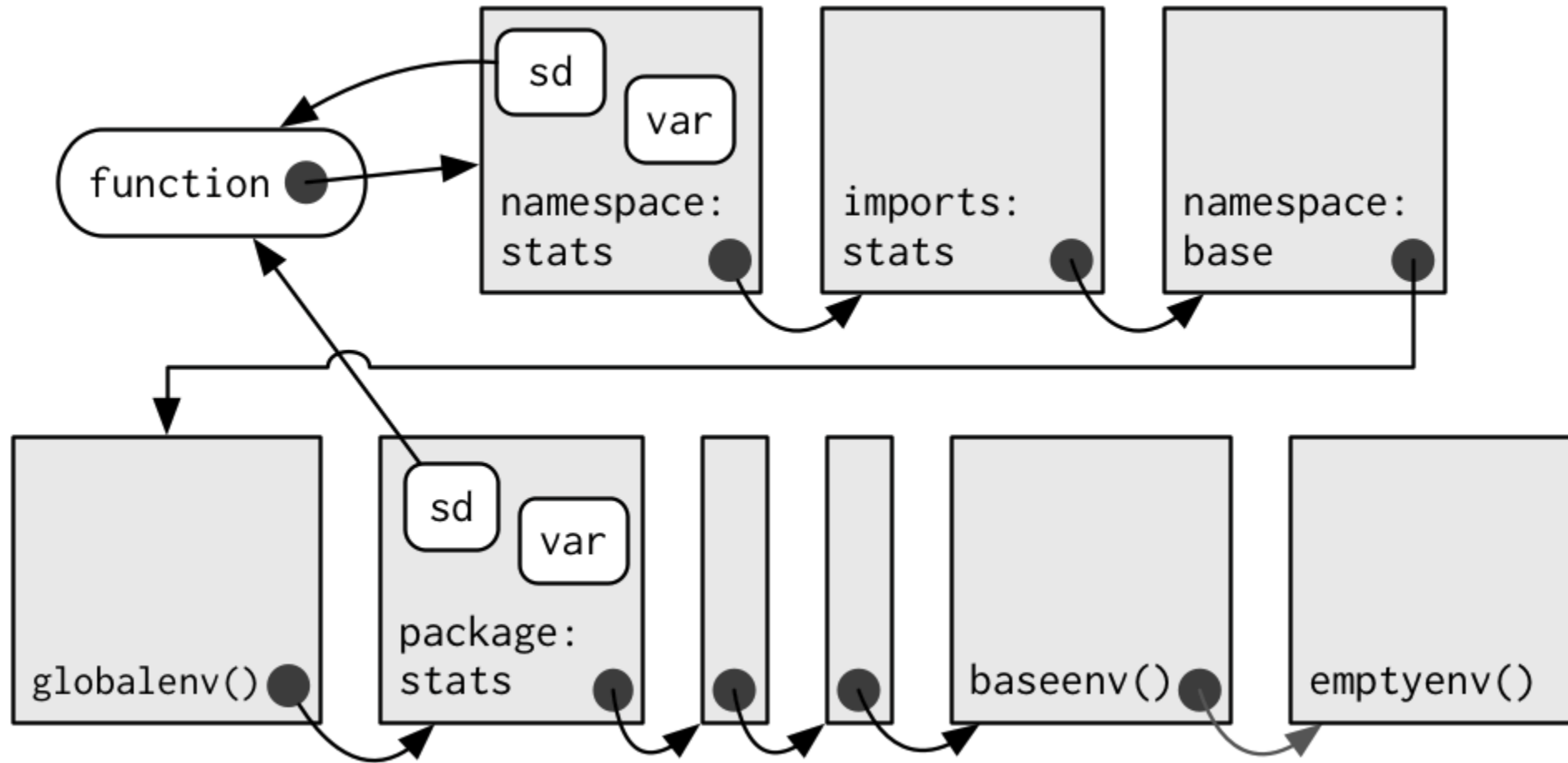
Type parameter

Generic type declaration

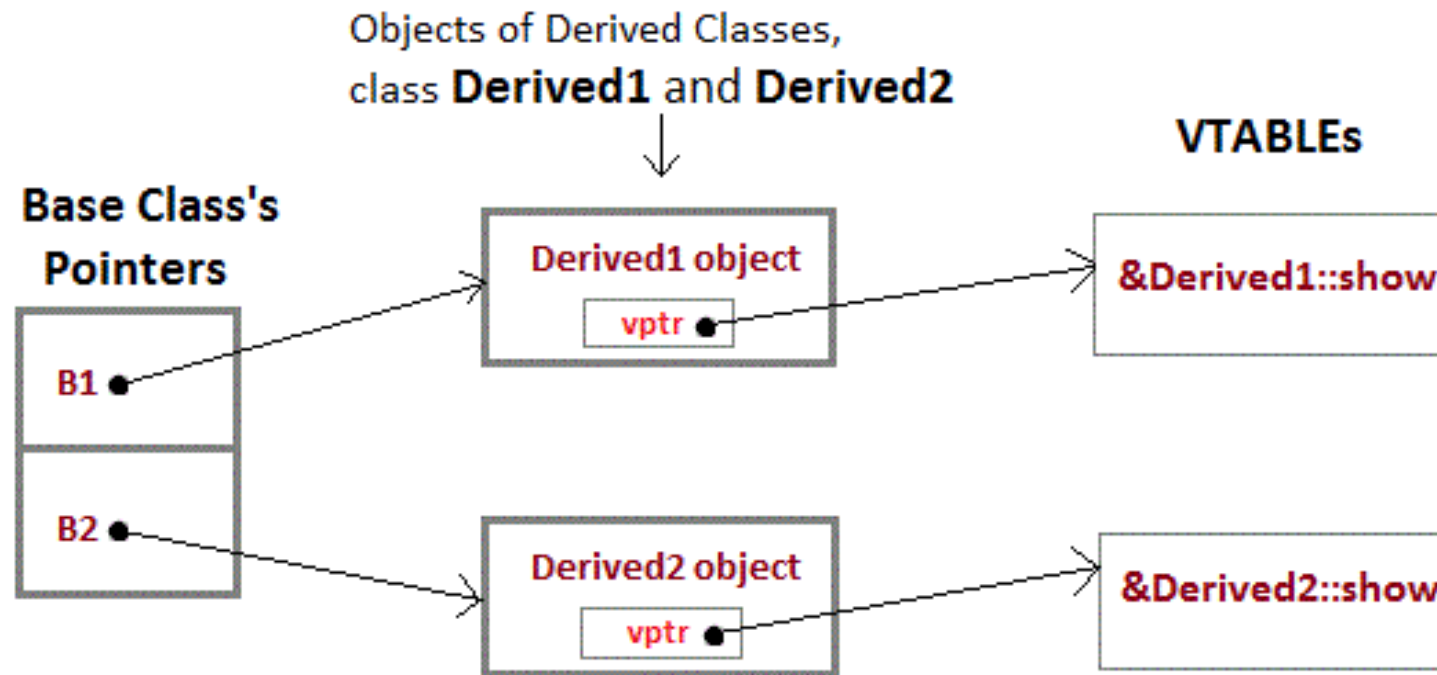
Parameterised methods

Type argument

C++ namespace for Binding



C++ Virtual Function for Dynamic Binding



vptr, is the vpointer, which points to the Virtual Function for that object.

VTABLE, is the table containing address of Virtual Functions of each class.

Binding

SECTION 2

The Notion of Binding Time

Static Binding Time

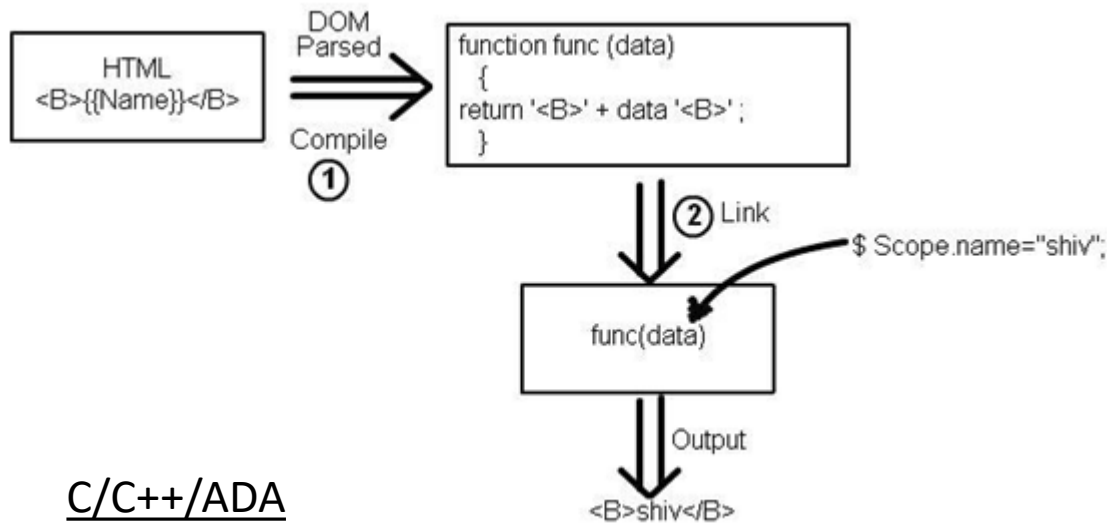
- **Language design time:** In most languages, the control flow constructs, the set of fundamental (primitive) types, the available constructors for creating complex types, and many other aspects of language semantics are chosen when the language is designed.
- **Language implementation time:** Most language manuals leave a variety of issues to the discretion of the language implementer. Typical examples include the precision (number of bits) of the fundamental types, the coupling of I/O to the operating system's notion of files, the organization and maximum sizes of stack and heap, and the handling of run-time exceptions such as arithmetic overflow.
- **Program writing time:** Programmers, of course, choose algorithms, data structures, and names.
- **Compile time:** Compilers choose the mapping of high-level constructs to machine code, including the layout of statically defined data in memory.

The Notion of Binding Time

Dynamic Binding Time [Link-Load-Go(static) and Load-Go-Link-Go(dynamic)]

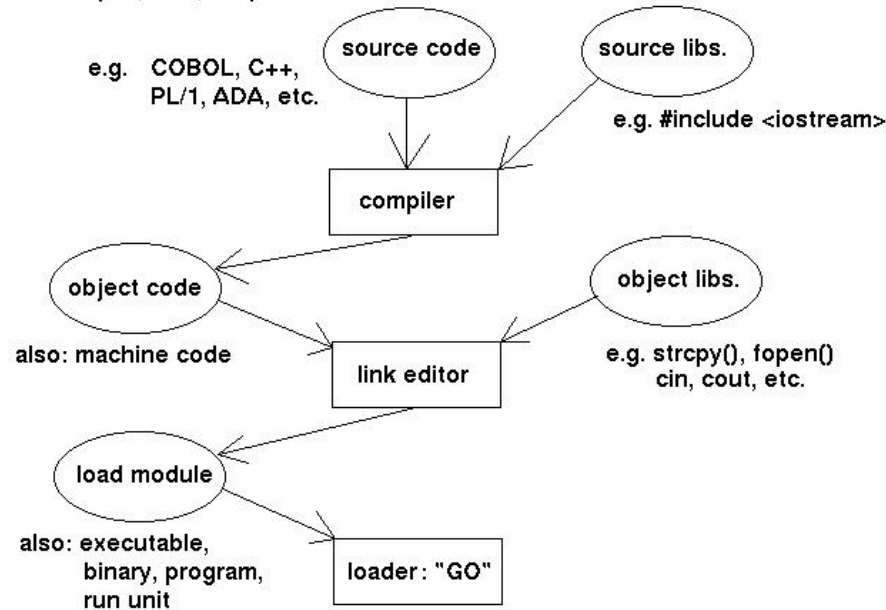
- **Link time(maybe static):** Since most compilers support separate compilation—compiling different modules of a program at different times—and depend on the availability of a library of standard subroutines, a program is usually not complete until the various modules are joined together by a linker. The linker chooses the overall layout of the modules with respect to one another, and resolves inter-module references. When a name in one module refers to an object in another module, the binding between the two is not finalized until link time.
- **Load time:** Load time refers to the point at which the operating system loads the program into memory so that it can run. In primitive operating systems, the choice of machine addresses for objects within the program was not finalized until load time. Most modern operating systems distinguish between virtual and physical addresses. Virtual addresses are chosen at link time; physical addresses can actually change at run time. The processor's memory management hardware translates virtual addresses into physical addresses during each individual instruction at run time.
- **Run time:** Run time is actually a very broad term that covers the entire span from the beginning to the end of execution. Bindings of values to variables occur at run time, as do a host of other decisions that vary from language to language.

Compile Link Go for JavaScript

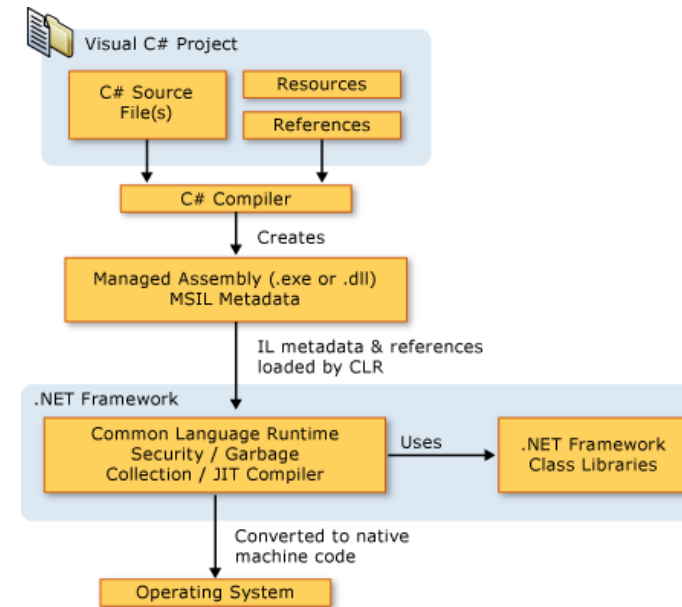


C/C++/ADA

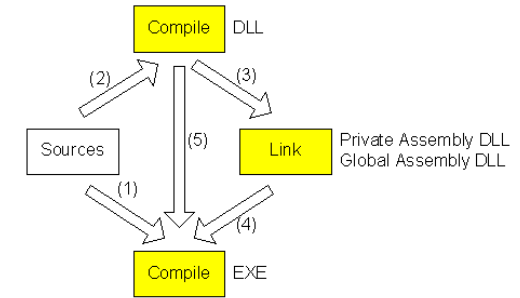
The Compile, Link, Go process:



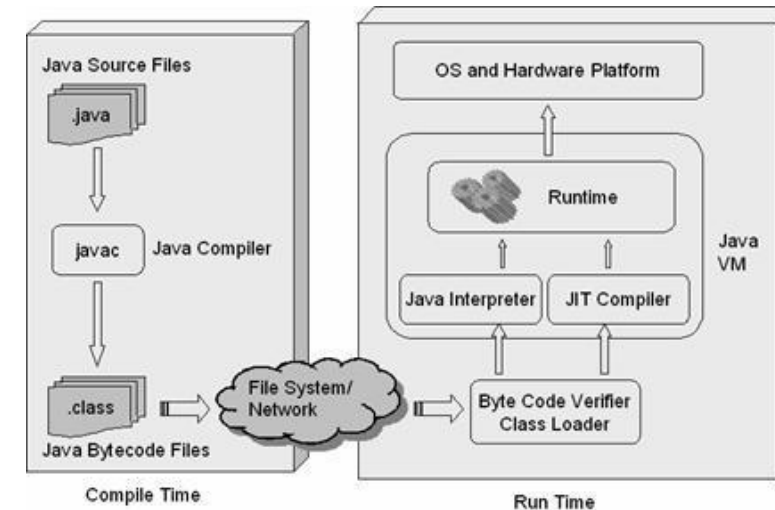
C# .NET Dual-Path Compile – Link – JIT Compile - Go



Visual C++



Java Compile – Load – Interpret – Link – JIT Compile - Go



Implementation Decisions

- Run time
 - value/variable bindings, sizes of strings/array, static/heap/stack
- Subsumes
 - Program Start-up Time (Load)
 - Module Entry Time (Used)
 - Elaboration Time (point at which a declaration is first "seen")
 - Procedure Entry Time (Called)
 - Block entry time (Run-Over)
 - Statement execution time (Execution Time)

Name, Scope, and Binding

- The terms STATIC and DYNAMIC are generally used to refer to things bound before run time and at run time, respectively
 - “static” is a coarse term; so is "dynamic"
- It is difficult to overstate the importance of Binding Times in programming languages

Name, Scope, and Binding

- In general, early binding times are associated with greater efficiency
- Later binding times are associated with greater flexibility
- Compiled languages tend to have early binding times
- Interpreted languages tend to have later binding times
- Today we talk about the binding of identifiers to the variables they name

Scope Rules - control bindings

- Fundamental to all programming languages is the ability to name data, i.e., to refer to data using symbolic identifiers rather than addresses
- Not all data is named! For example, dynamic storage in C or Pascal is referenced by pointers, not names

Lifetime and Storage Management

SECTION 3

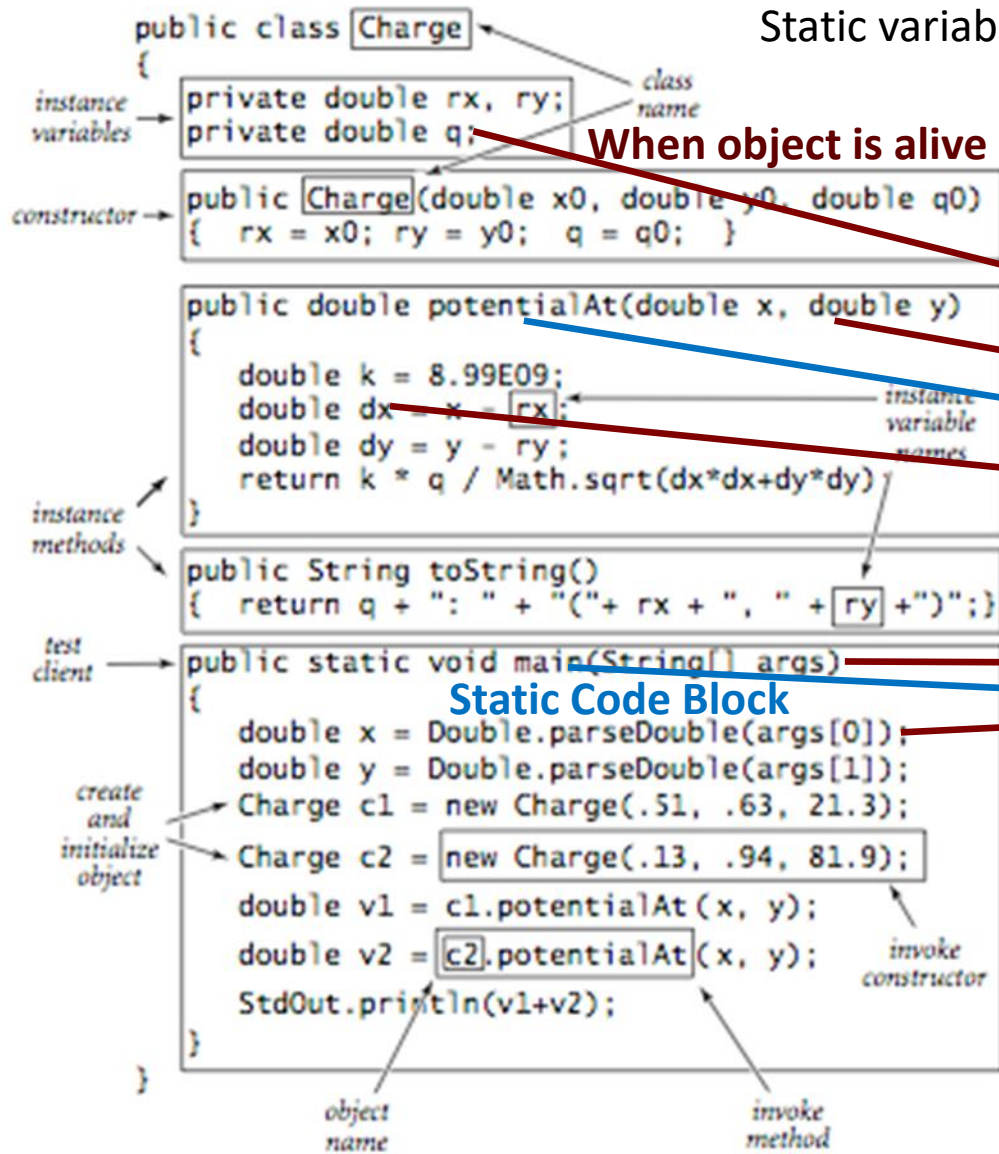
Lifetime and Storage Management

Key events in memory management:

- Creation and destruction of objects
- Creation and destruction of bindings
- Deactivation and reactivation of bindings that may temporarily be unavailable
- References to variables, Subroutines, Types and so on, all of which use bindings.

Lifetime and Storage Management

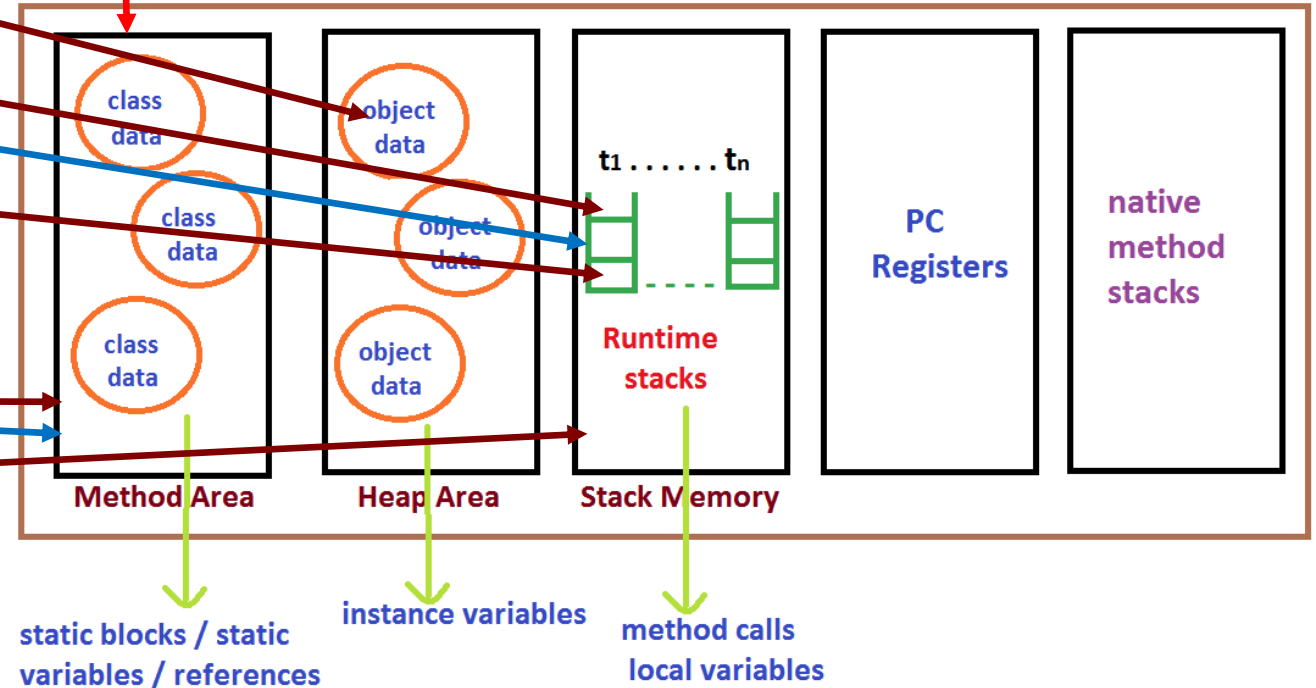
- The period of time from creation to destruction is called the **Lifetime** of a binding
 - If object outlives binding it's garbage
 - If binding outlives object it's a dangling reference
- The textual region of the program in which the binding is **active** is its scope
- In addition to talking about the **scope of a binding**, we sometimes use the word **scope** as a noun all by itself, without an indirect object



When object is alive

Static variables: static int x;
final static int I;

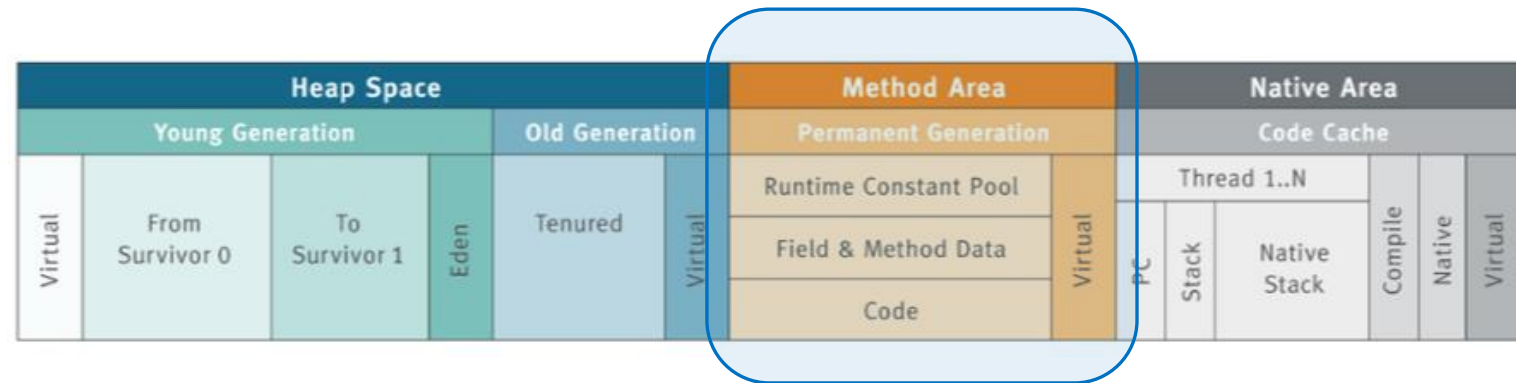
Static Code Block



Anatomy of a class

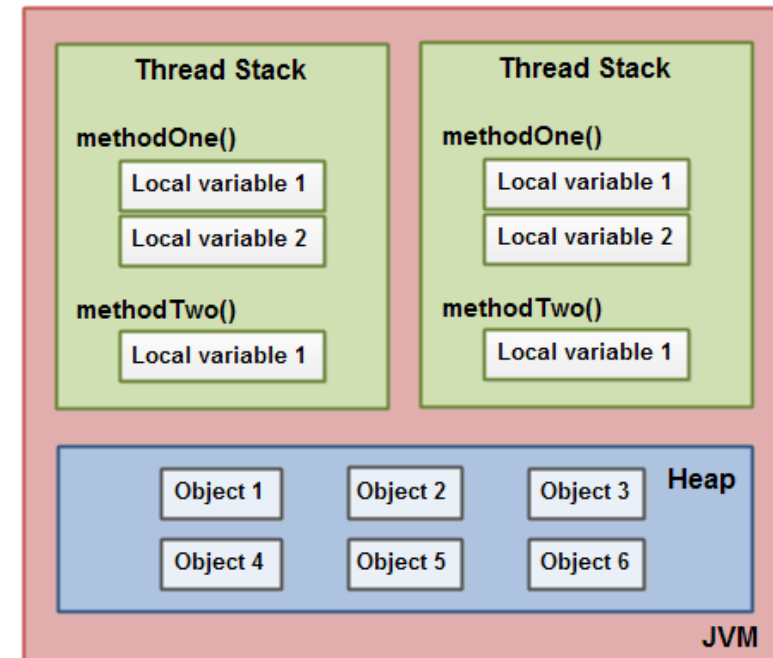
Lifetime and Storage Management

- Storage Allocation mechanisms
 - Static
 - Stack
 - Heap
- Static allocation for
 - code
 - globals
 - static or own variables
 - explicit constants (including strings, sets, etc)
 - scalars may be stored in the instructions



Lifetime and Storage Management

- Central stack for
 - parameters
 - local variables
 - temporaries
- Why a stack?
 - allocate space for recursive routines
(not necessary in FORTRAN – no recursion)
 - reuse space
(in all programming languages)



Lifetime and Storage Management

- Contents of a stack frame (cf., Figure 3.1)
 - arguments and returns
 - local variables
 - temporaries
 - bookkeeping (saved registers, line number static link, etc.)
- Local variables and arguments are assigned fixed OFFSETS from the stack pointer or frame pointer at compile time

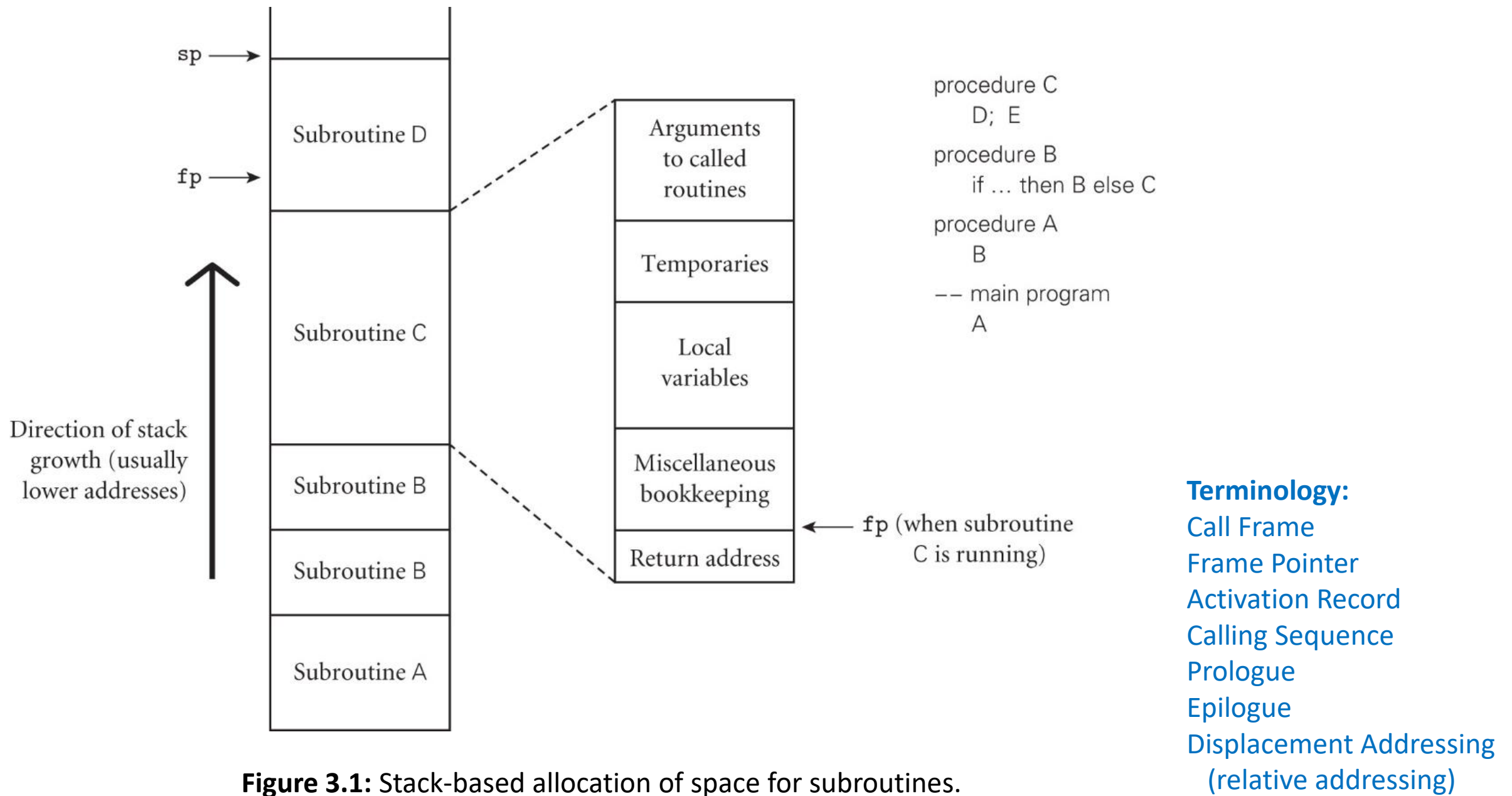


Figure 3.1: Stack-based allocation of space for subroutines.

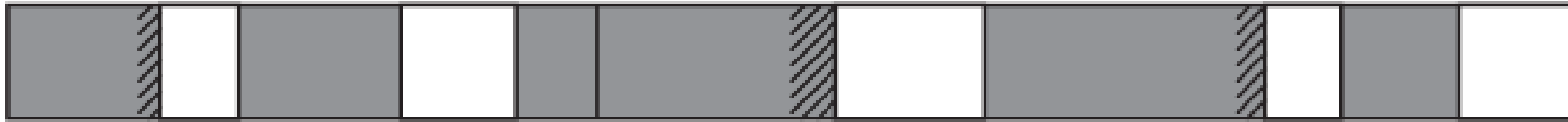
Stack and Call Frame

Maintenance of stack is responsibility of calling sequence and subroutine **prologue** and **epilogue**

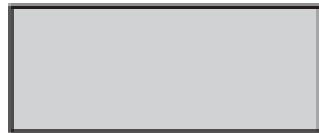
- space is saved by putting as much in the prologue and epilogue as possible
- time may be saved by putting more in the body of the subroutine
 - putting stuff in the caller instead
or
 - combining what's known in both places
(inter-procedural optimization)

Heap for dynamic allocation

Heap



Allocation request



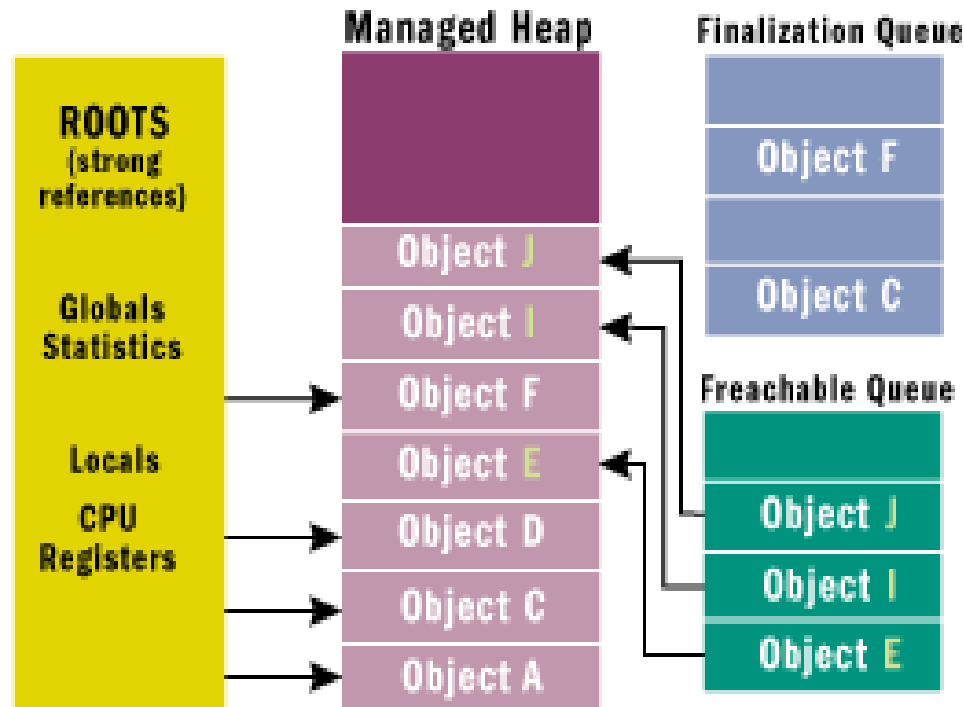
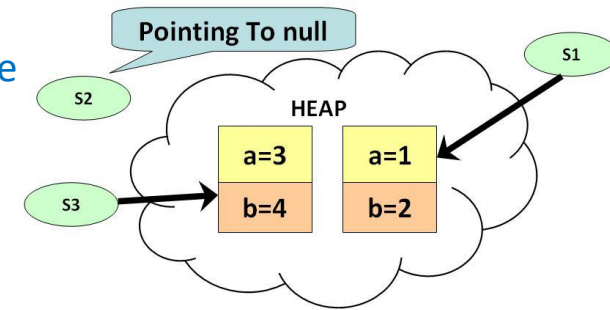
Terminology:

- Constructor
- Destructor
- malloc()
- free()
- Heap
- Fragmentation
- First-fit algorithm
- Best-fit algorithm

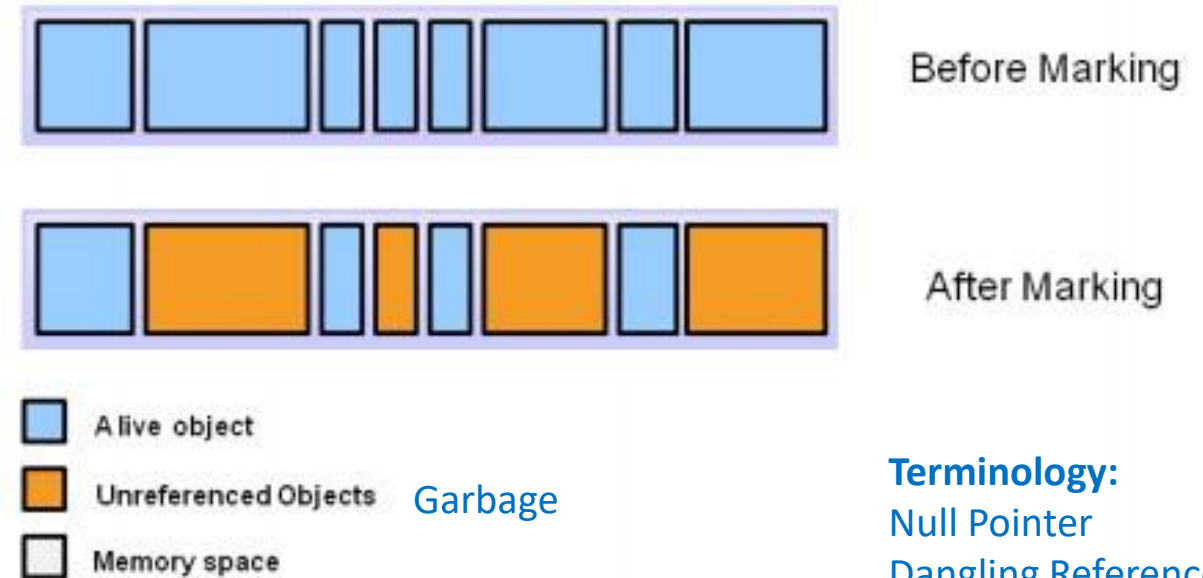
Figure 3.2 Fragmentation. The shaded blocks are in use; the clear blocks are free. Cross-hatched space at the ends of in-use blocks represent internal fragmentation. The discontinuous free blocks indicate external fragmentation. While there is more than enough total free space remaining to satisfy an allocation request of the illustrated size, no single remaining block is large enough.

Garbage Collection

Dangling Reference



Marking Mark and Sweep (Chapter 8)

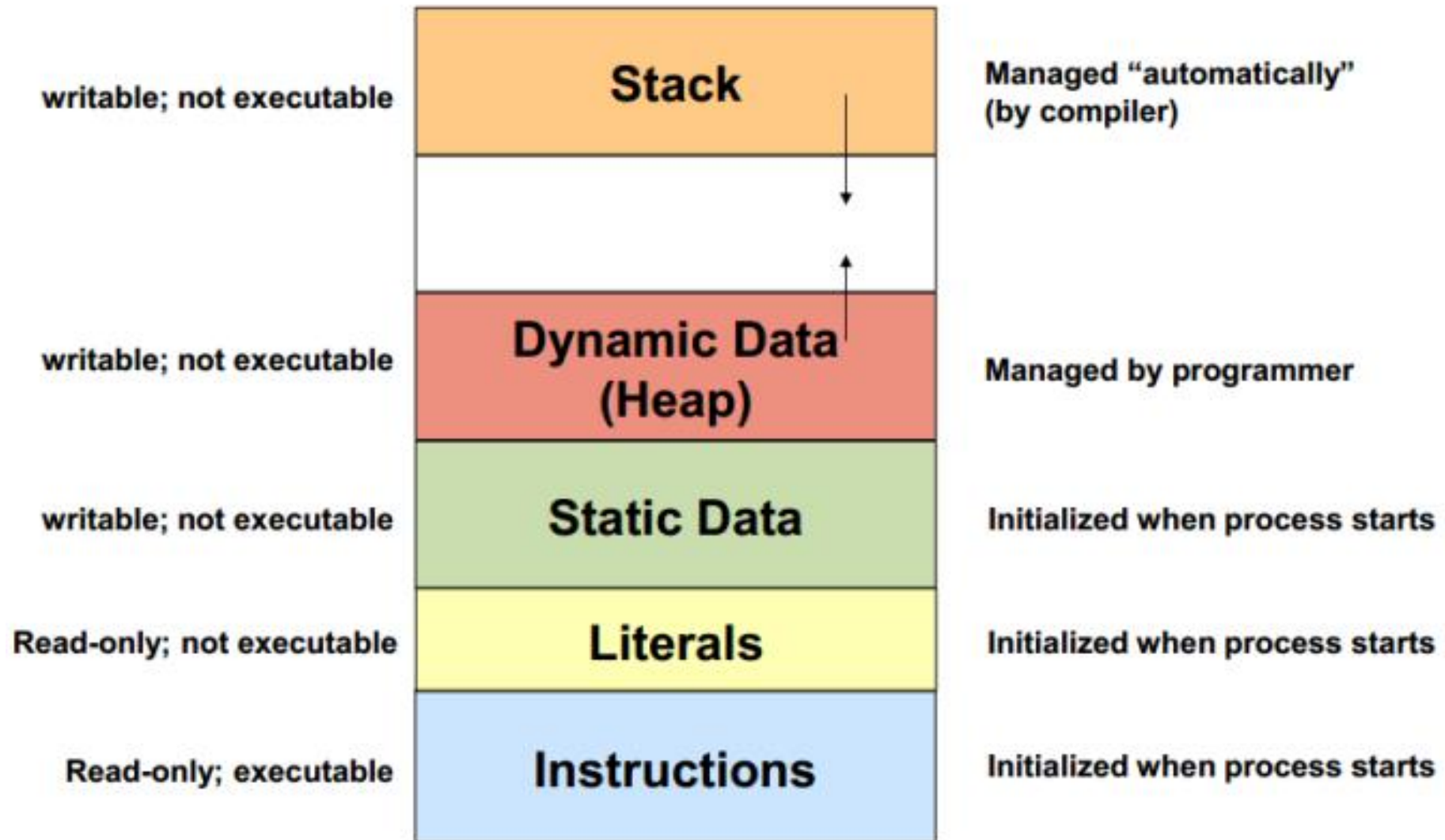


Terminology:
Null Pointer
Dangling Reference
Garbage

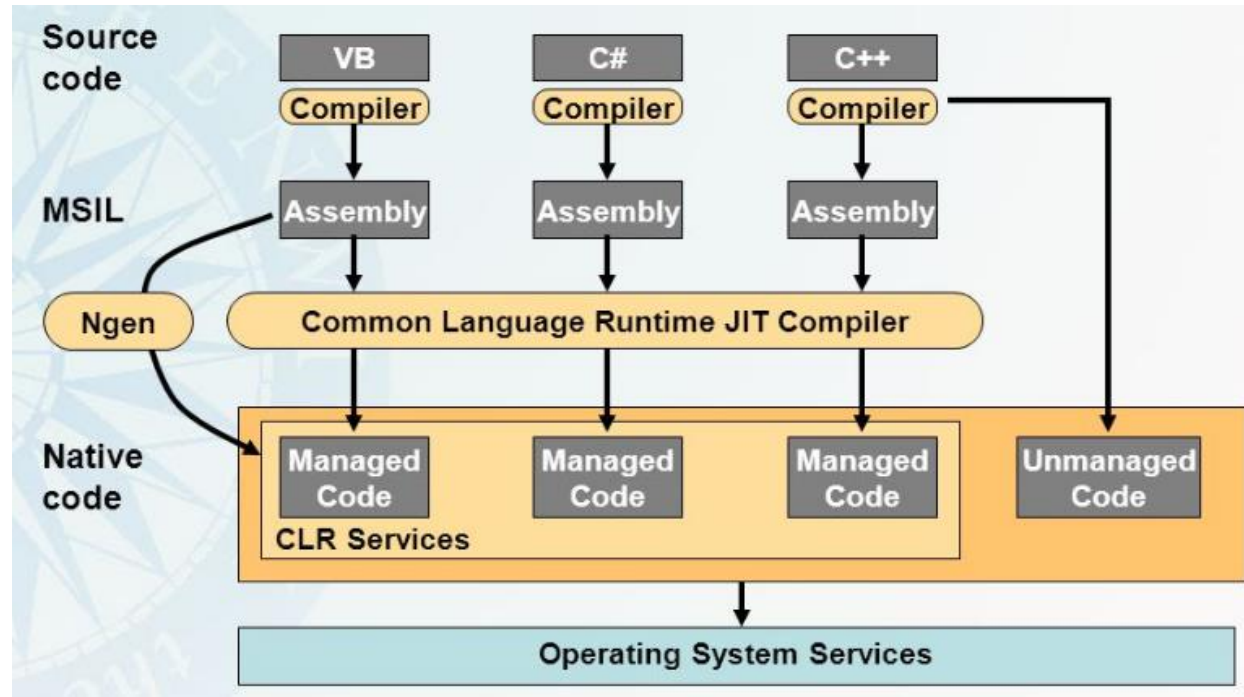
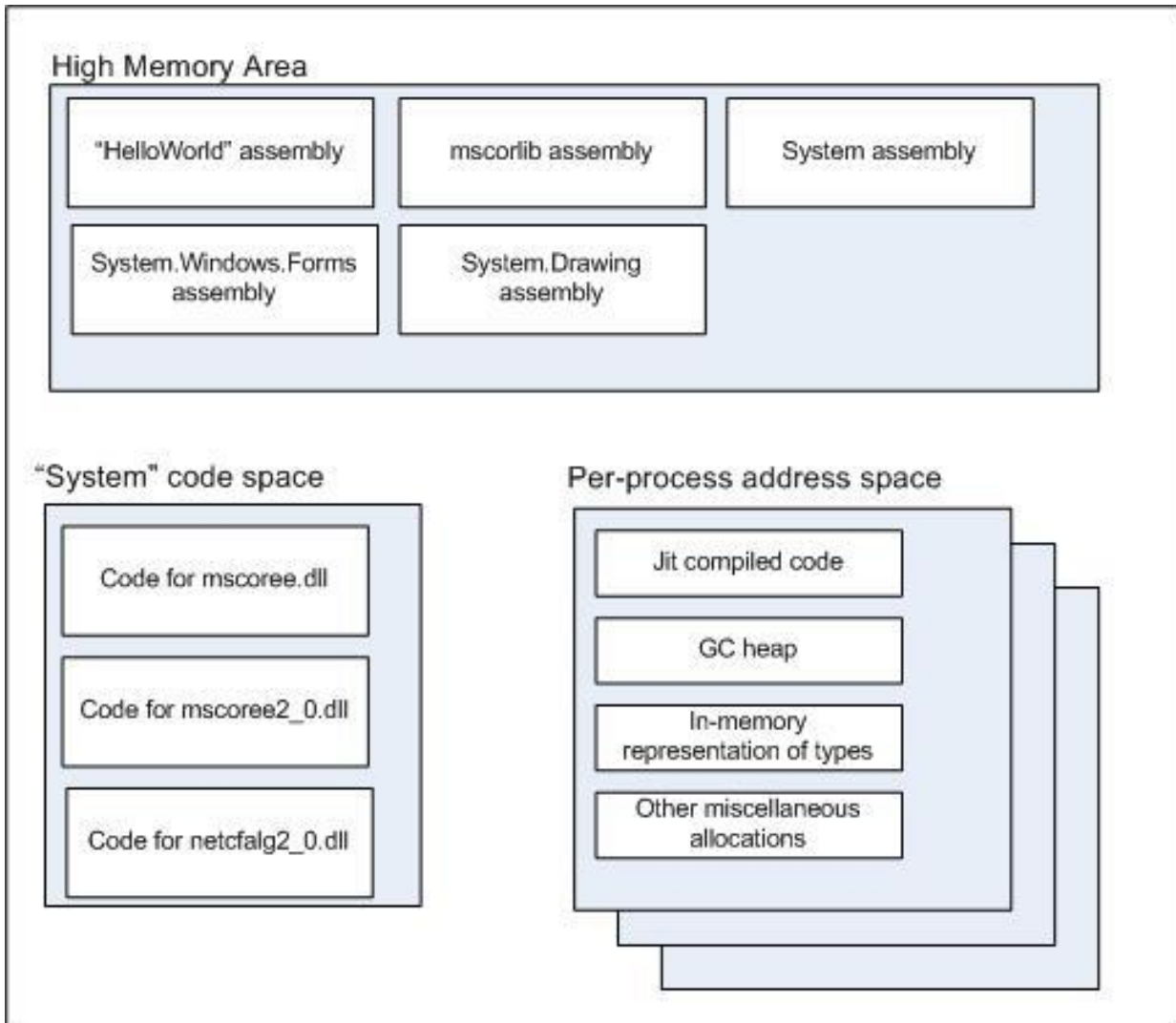
Memory Model

SECTION 4

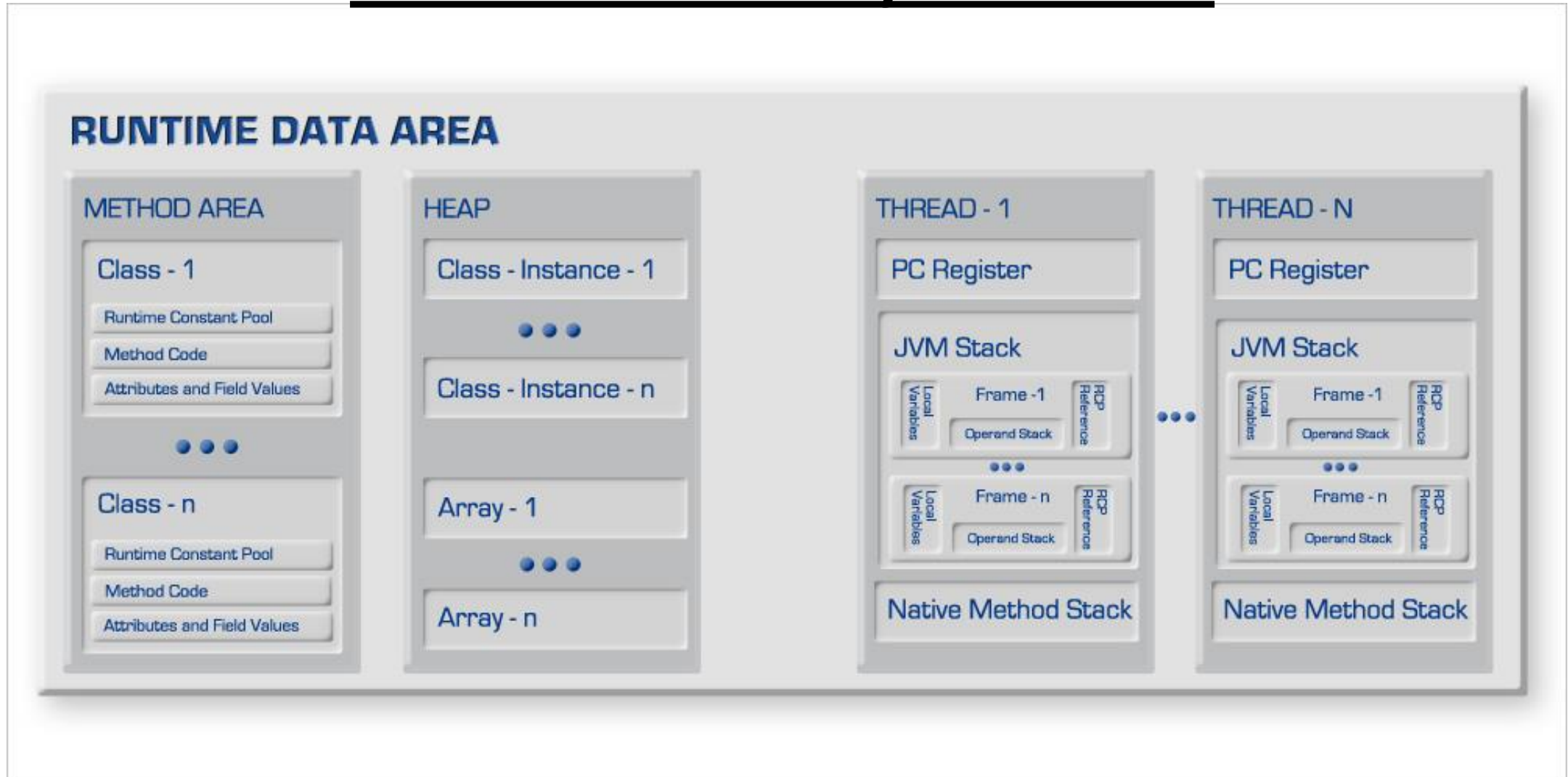
C++ Memory Model



.NET Memory Model

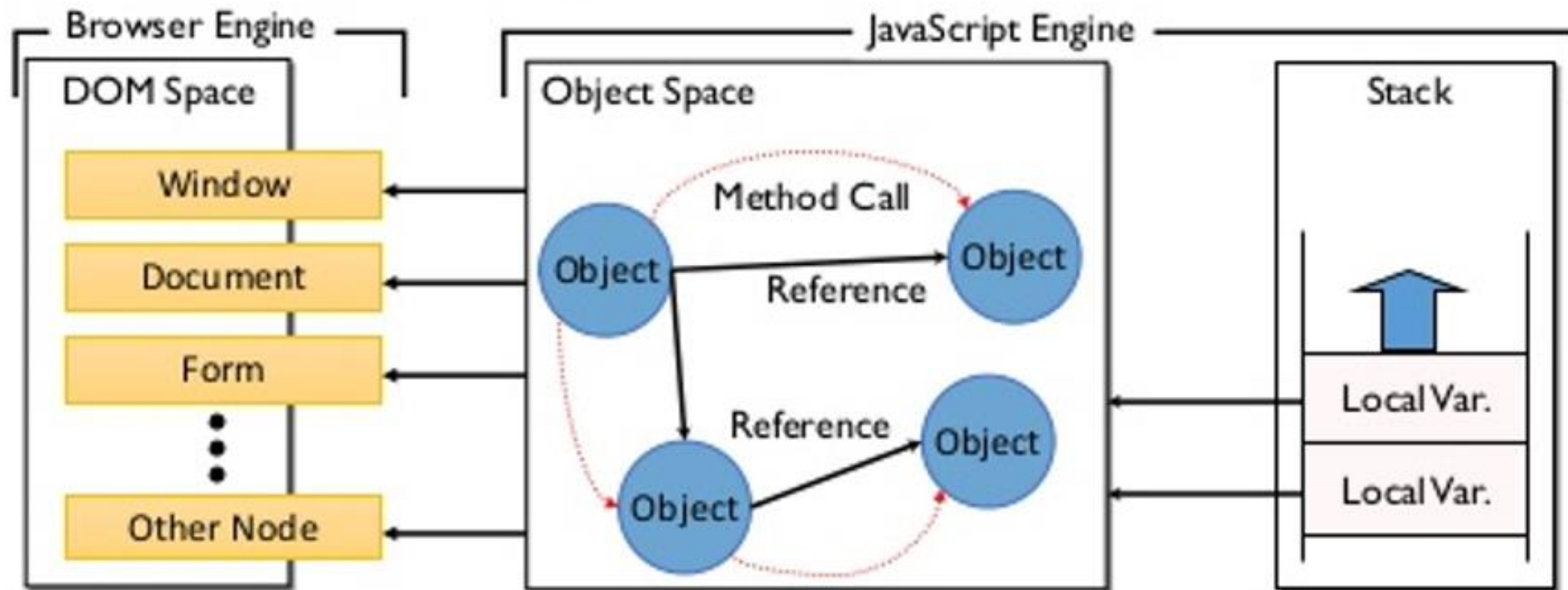


Java Memory Model

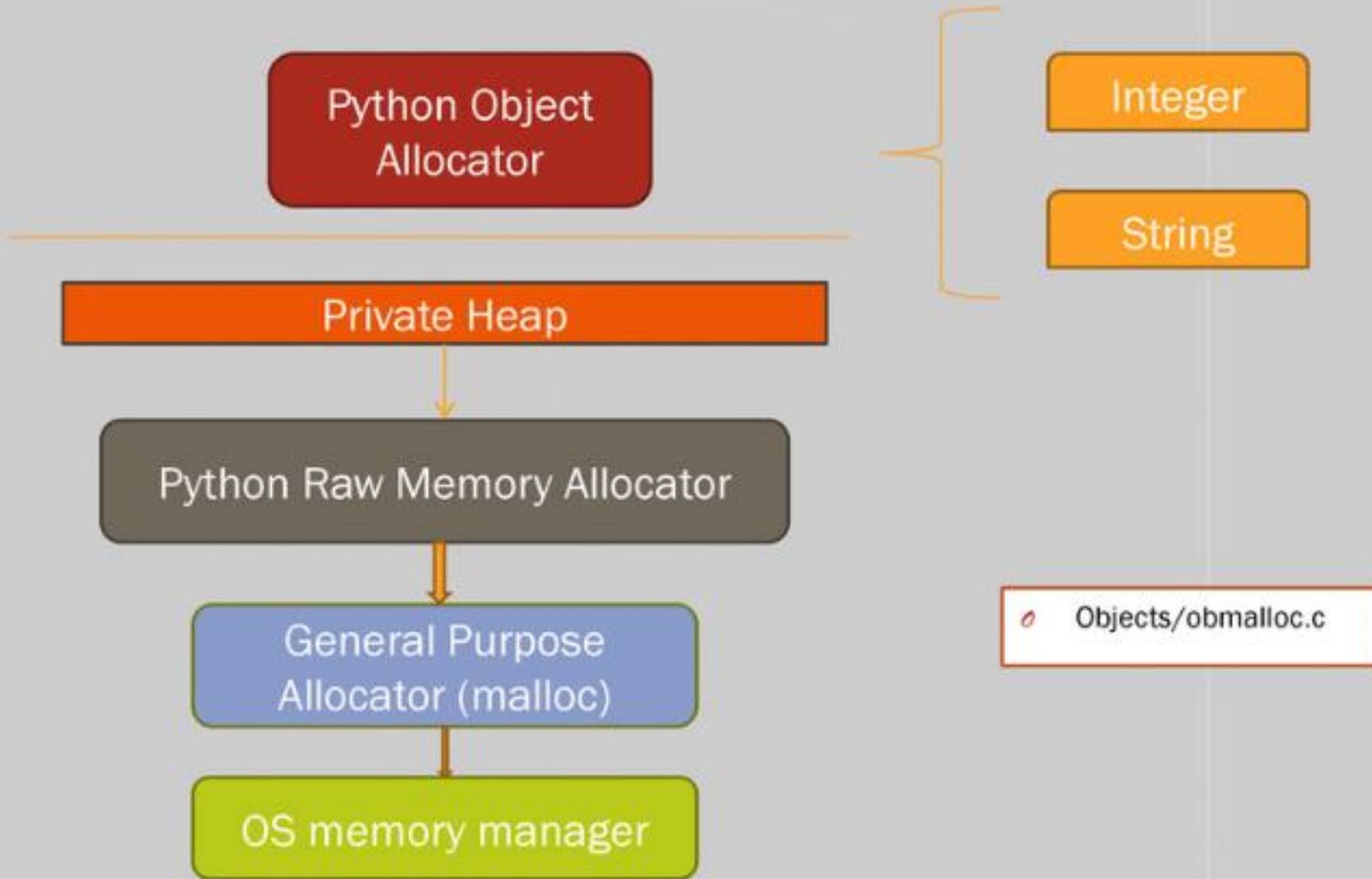


JavaScript Memory Model

- DOM Space: the space where the Document Object Model representing the HTML's layered structure is represented.
- Object Space: the space where all JavaScript objects are located.
- Stack: short-term memory



Python Memory Model



Scope Rules I

Scope Rules

SECTION 5

Scope Rules

- The textual region of the program in which a binding is active is its **scope** (of binding).
- In most languages with subroutines, we open a new scope on subroutine entry:
 - create bindings for new local variables,
 - deactivate bindings for global objects that are hidden by local object of the same name. **[local comes first in binding]**
 - re-declared make references to variables

Static vs. Dynamic Scope

Static scoping

- Local understanding of function behavior
- Know at compile-time what each name refers to
- A bit trickier to implement

Dynamic scoping

- Can be hard to understand behavior of functions
- Requires finding name bindings at runtime
- Easier to implement (just keep a global table of stacks of variable/value bindings)

Static Scoping

Because active bindings are statically linked to a block's lexical ancestor's bindings this scoping method is probably the most intuitive binding mechanism and is featured in most modern and class oriented languages such as Java, C++ and C#

Dynamic Scoping

In languages with dynamic scope, bindings are dependent on the order of which subroutines are called at execution. Because the sequence and flow of control cannot always be predicted in advance, bindings made in dynamically scoped programs cannot be type-checked by the compiler. [\[APL, Snobol, and Tcl\]](#)

Scope Rules

- On subroutine exit:
 - destroy bindings for local variables
 - reactivate bindings for global variables that were deactivated
- Algol 68:
 - Elaboration = process of creating bindings when entering a scope
- Ada (re-popularized the term elaboration):
 - storage may be allocated, tasks started, even exceptions propagated as a result of the elaboration of declarations

Scope Rules

- **Reference Environment:** The set of active **bindings** is called the current referencing environment.
 - The set is principally determined by static or dynamic scope rules.
 - A referencing environment generally corresponds to a sequence of scopes that can be examined to find the current binding for a given name.
- Referencing environments also depend on what are called **Binding Rules**.
 - Specifically, when a reference to a subroutine S is stored in a variable, passed as a parameter to another subroutine, or returned as a function value, one needs to determine when the referencing environment for S is chosen—that is, when the binding between the reference to S and the referencing environment of S is made.
 - The two principal options are:
 - **deep binding**, in which the choice is made when the reference is first created.
 - **shallow binding**, in which the choice is made when the reference is finally used.

Note: Binding and Linking are different. Binding is to associate a variable to a memory location. Linking is to associate a set of program implementation to a certain caller.

Scope Rules

- With **Static (Lexical) Scope Rules**, a scope is defined in terms of the physical (lexical) structure of the program
 - The determination of scopes can be made by the compiler
 - All bindings for identifiers can be resolved by examining the program
 - Typically, we choose the most recent, **active binding** made at compile time
 - Most compiled languages, C and Pascal included, employ static scope rules

Scope Rules

- Dynamic scope rules are usually encountered in interpreted languages
 - early Lisp dialects assumed dynamic scope rules.
- Such languages do not normally have type checking at compile time because type determination isn't always possible when dynamic scope rules are in effect

Static Variables in C

- As an example of the use of static variables, consider the code in Figure 3.3.
- The subroutine **label_name** can be used to generate a series of distinct character string names: **L1, L2,**
- A compiler might use these names in its assembly language output.

```
/*  
    Place into *s a new name beginning with the letter 'L' and  
    continuing with the ASCII representation of a unique integer.  
    Parameter s is assumed to point to space large enough to hold any  
    such name; for the short ints used here, 7 characters suffice.  
*/  
void label_name (char *s) {  
    static short int n;          /* C guarantees that static locals  
                                are initialized to zero */  
    sprintf (s, "L%d\0", ++n); /* "print" formatted output to s */  
}
```

Figure 3.3 C code to illustrate the use of static variables.

Note: **n** is increased in every function call.

Scope Rules II

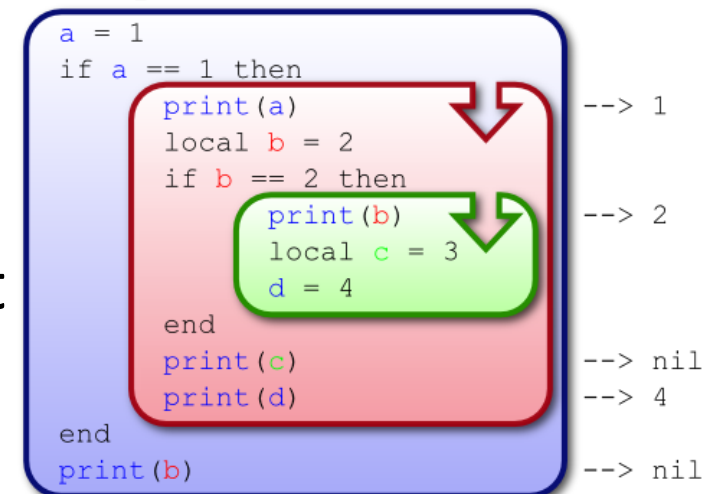
Nested Scope Rules

SECTION 6

Nested Subroutines

- The classical example of static scope rules is the most closely nested rule used in block structured languages such as Algol 60 and Pascal
 - An **identifier** is known in the scope in which it is **declared** and in each enclosed scope, unless it is re-declared in an enclosed scope
 - To resolve a reference to an identifier, we examine the local scope and statically enclosing scopes until a binding is found

Script:



Nested Scope Rules

- A name-to-object binding that is hidden by a nested declaration of the same name is said to have a **hole** in its scope.
- In most languages the object whose name is hidden is inaccessible in the nested scope (unless it has more than one name).
- Some languages allow the programmer to access the outer meaning of a name by applying a **qualifier** or scope resolution operator.
- In Ada, for example, a name may be prefixed by the name of the scope in which it is declared, using syntax that resembles the specification of fields in a record.
- **My_proc.X**, for example, refers to the declaration of **X** in subroutine **My_proc**, regardless of whether some other **X** has been declared in a lexically closer scope.
- In **C++**, which does not allow subroutines to nest, **::X** refers to a global declaration of **X**, regardless of whether the current subroutine also has an **X**.

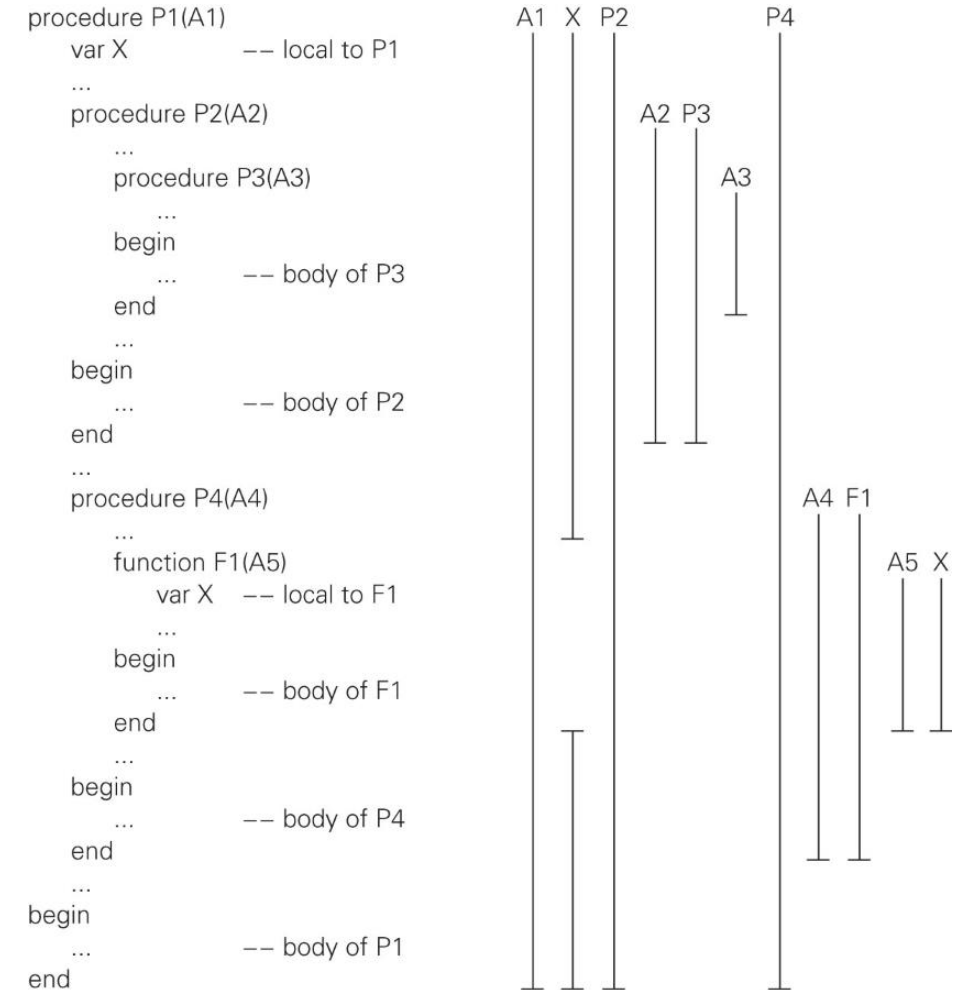


Figure 3.4

Scope Rules

- We will see classes - a relative of modules - later on, when discussing abstraction and object-oriented languages
 - These have even more sophisticated (static) scope rules
- Euclid is an example of a language with lexically-nested scopes in which all scopes are closed
 - rules were designed to avoid Aliases, which complicate optimization and correctness arguments

Note: that the bindings created in a subroutine are destroyed at subroutine exit

- The modules of Modula, Ada, etc., give you closed scopes without the limited lifetime
- Bindings to variables declared in a module are inactive outside the module, not destroyed
- The same sort of effect can be achieved in many languages with *own* (Algol term) or *static* (C term) variables (see Figure 3.5)

Access to non-local variables Static Links

- Each frame points to the frame of the (correct instance of) the routine inside which it was declared
- In the absence of formal subroutines, correct means closest to the top of the stack
- You access a variable in a scope *k* levels out by following *k* static links and then using the known offset within the frame thus found

Access to Non-Local Variables

- The **frame pointer register** points to current **call frame** in the stack. The call frame is current reference environment.
- Using the register as a base for displacement (register plus offset) addressing to find objects in current subroutine.
- But what about non-local variables?

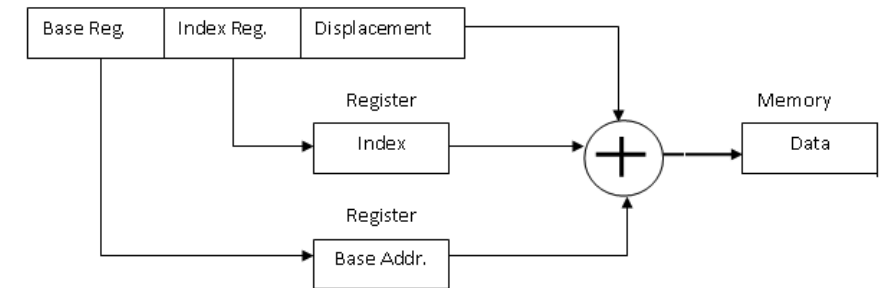


Figure (a) Relative Based Indexed Addressing Mode

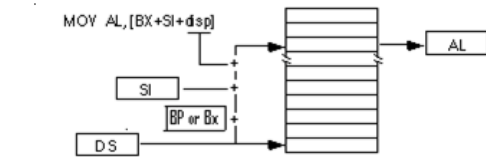


Figure (b) Relative Based Indexed Addressing Mode

Access to Non-Local Variables

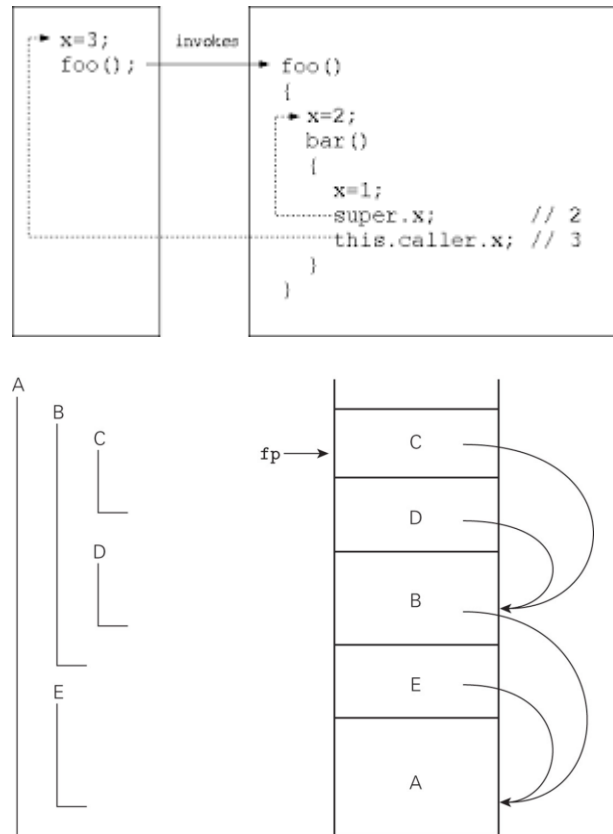


Figure 3.5

The simplest way in which to find the frames of surrounding scopes is to maintain a **static link** in each frame that points to the “parent” frame: the frame of the most recent invocation of the lexically surrounding subroutine.

If subroutine is declared at the outermost nesting level of the program, then its frame will have null static link at run time.

If a subroutine is nested k levels deep, then its frame’s static link, and those of parent, grandparent, and so on, will form a static chain.

[compared to dynamic chain in dynamic binding in polymorphism]

Scope Rules III

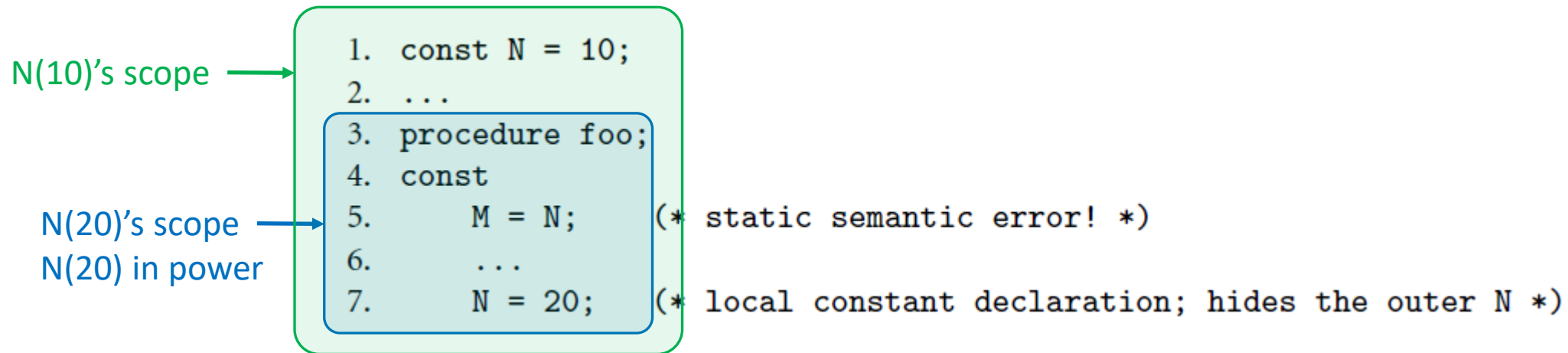
Scope Rule in a Block: Declaration Order

SECTION 7

Declaration Order

In several early languages, including Algol 60, Lisp, required all declarations appear at the beginning of their scope.

Pascal modified the requirement that names must be declared before use. Pascal retained the notion that the scope of a declaration is the surrounding block.



Declaration Order

- Pascal says that the second declaration of N covers all of foo, so the semantic analyzer should complain on line 5 that N is being used before its declaration.
- The error has the potential to be highly confusing, particularly if the programmer meant to use the outer N:

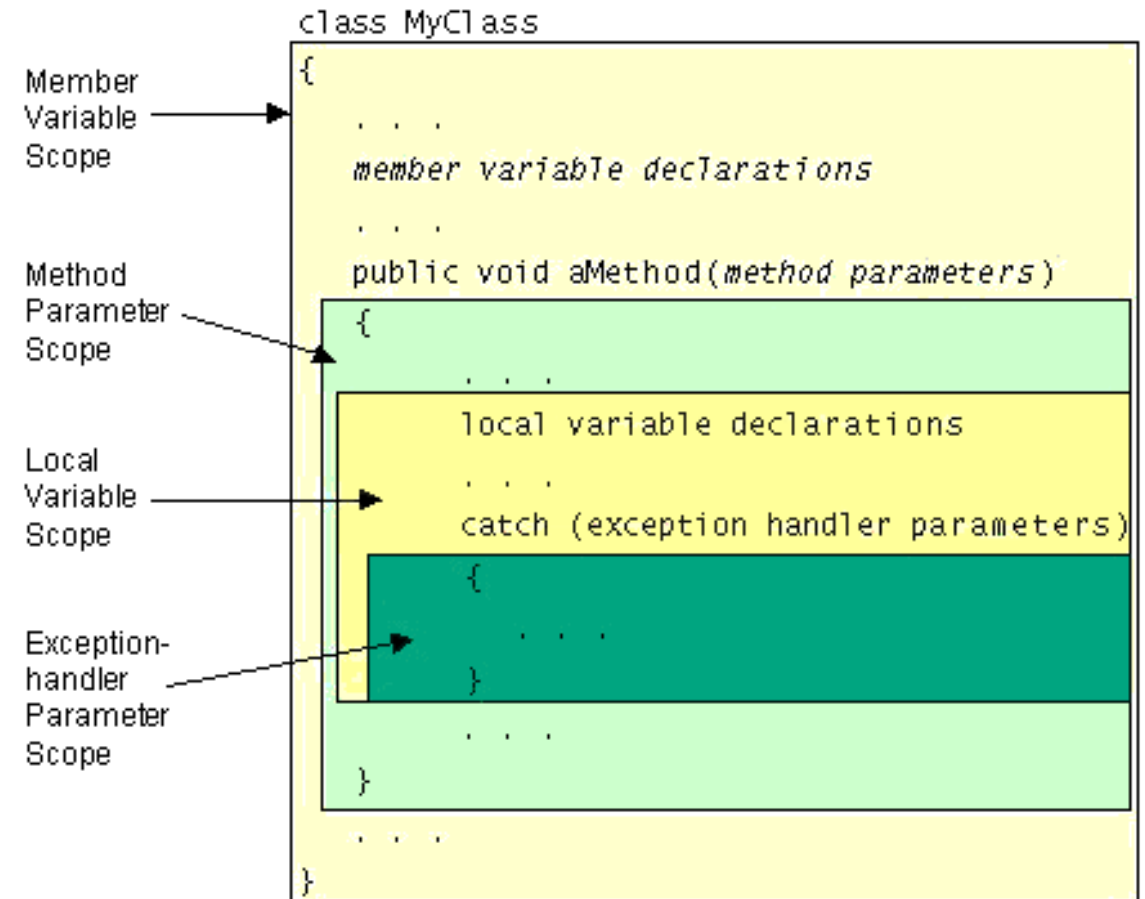
```
const N = 10;
...
procedure foo;
const
    M = N;          (* static semantic error! *)
var
    A : array [1..M] of integer;
    N : real;        (* hiding declaration *)
```

- In order to determine the validity of any declaration that appears to use a name from a surrounding scope, a Pascal compiler must scan the remainder of the scope's declarations to see if the name is hidden.

Declaration Order

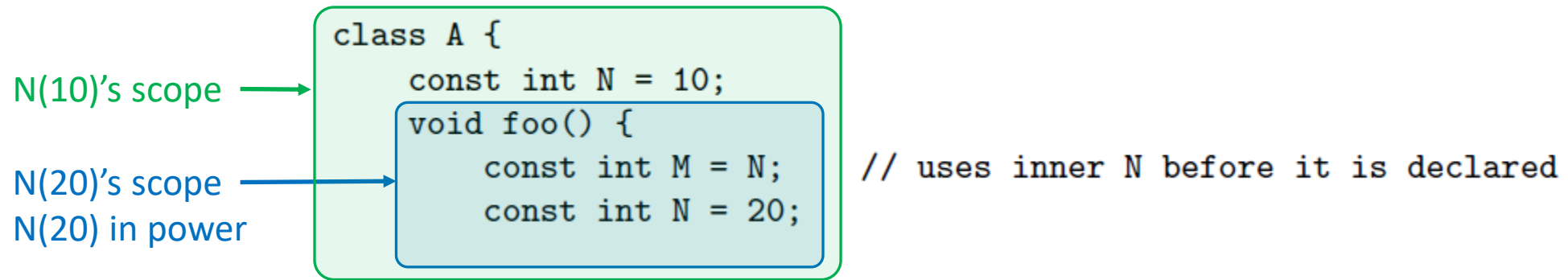
Most Other Languages after Pascal

- To avoid this complication, most Pascal successors specify that the scope of an identifier is not the entire block in which it is declared, but rather the portion of that block from the declaration to the end.
- If our program fragment had been written in Ada, for example, or in C, C++, or Java, no semantic errors would be reported. The declaration of M would refer to the first (outer) declaration of N.
- C++ and Java further relax the rules by dispensing with the define-before-use requirement in many cases. In both languages, members of a class are visible inside all of the class's methods. In Java, classes themselves can be declared in any order.



Whole-block scope in C#

- C# returns to the Pascal notion of whole-block scope. Thus the following is invalid in C#.



Declaration order in Scheme

In the interest of flexibility, modern Lisp dialects tend to provide several options for declaration order. In Scheme, for example, the **letrec** and **let*** constructs define scopes with, respectively, whole-block and declaration-to-end-of-block semantics. The most frequently used construct, **let**, provides yet another option:

```
(let ((A 1))           ; outer scope, with A defined to be 1
  (let ((A 2)          ; inner scope, with A defined to be 2
        (B A))        ;           and B defined to be A
    B))               ; return the value of B
```

Here the nested declarations of A and B don't take effect until after the end of the declaration list. Thus when B is defined, the redefinition of A has not yet taken effect. B is defined to be the outer A, and the code as a whole returns 1.

Declarations and Definitions

- Recursive types and subroutines introduce a problem for languages that require names to be declared before they can be used: how can two declarations each appear before the other?
- C and C++ handle the problem by distinguishing between the declaration of an object and its definition.
- A declaration introduces a name and indicates its scope, but may omit certain implementation details.
- A definition describes the object in sufficient detail for the compiler to determine its implementation.
- If a declaration is not complete enough to be a definition, then a separate definition must appear somewhere else in the scope.

Declarations and Definitions in C

```
struct manager;                /* declaration only */
struct employee {
    struct manager *boss;
    struct employee *next_employee;
    ...
};
struct manager {                /* definition */
    struct employee *first_employee;
    ...
};
```

The initial declaration of `manager` needed only to introduce a name: since pointers are all the same size, the compiler could determine the implementation of `employee` without knowing any `manager` details.

```
void list_tail(follow_set fs);  /* declaration only */
void list(follow_set fs)
{
    switch (input_token) {
        case id : match(id); list_tail(fs);
        ...
    }
void list_tail(follow_set fs)    /* definition */
{
    switch (input_token) {
        case comma : match(comma); list(fs);
        ...
    }
}
```

The initial declaration of **`list_tail`**, however, must include the return type and parameter `list`, so the compiler can tell that the call in **`list`** is correct.

Nested Blocks

Variables declared in nested blocks can be very useful, as for example in the following C code:

```
{  
    int temp = a;  
    a = b;  
    b = temp;  
}
```

Keeping the declaration of temp lexically adjacent to the code that uses it makes the program easier to read, and eliminates any possibility that this code will interfere with another variable named temp.

Note: No run-timework is needed to allocate or deallocate space for variables declared in nested blocks; their space can be included in the total space for local variables allocated in the subroutine prologue and deallocated in the epilogue.

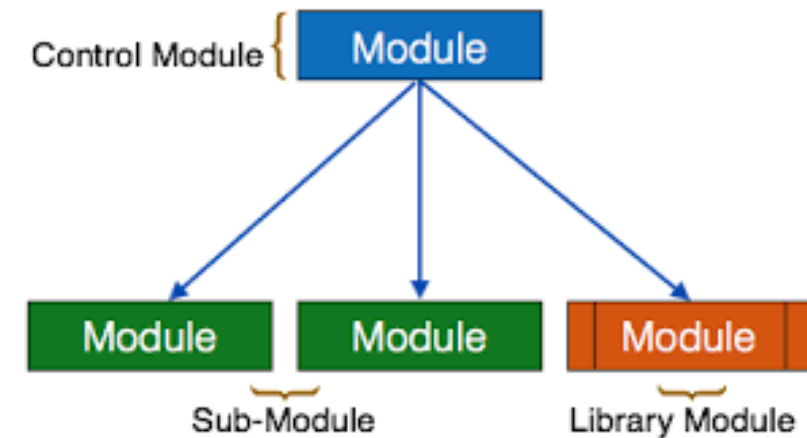
Scope Rules IV

Module, Class, and Dynamic Scoping

SECTION 8

Modules

- Help software development in parallel by a team of engineers.
- The modularization of effort depends critically on the notion of **information hiding**, which makes objects and algorithms invisible, whenever possible, to portions of the system that do not need them.
- Encapsulating Data and Subroutines:
 - Information hiding by subroutines is limited
 - Static variables in subroutines allow programmer to achieve information hiding for a single subroutine.
 - The module construct can help achieve this goal.



Module as Abstraction

A module allows a collection of objects—subroutines, variables, types, and so on—to be encapsulated in such a way that

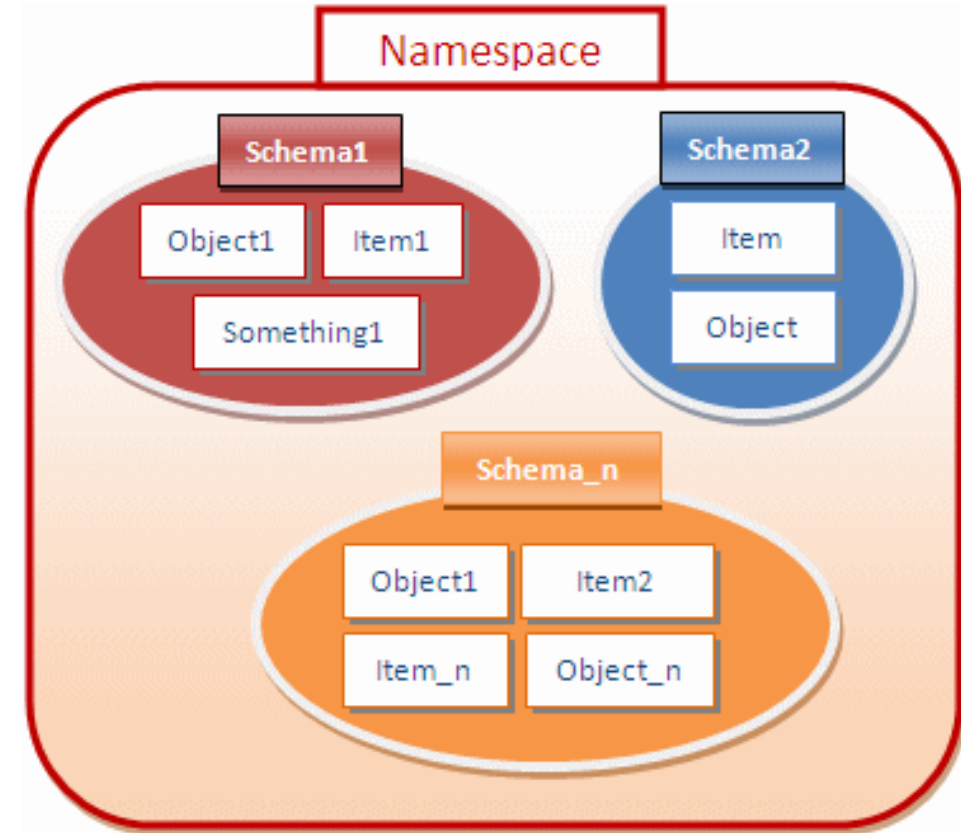
1. objects inside are visible to each other, but
2. objects on the inside are not visible on the outside unless explicitly exported, and
3. (in many languages) objects outside are not visible on the inside unless explicitly imported.

Import and export conventions vary significantly from one language to another, but in all cases, only the visibility of objects is affected; modules do not affect the lifetime of the objects they contain.

Namespace

Using namespace we can have class, global, variables, functions under one name. We can change global scope to Sub-scope.

- Namespace is a logical compartment used to avoid naming collisions.
- The naming conflict or Collision always Occurs may occur –
- When same program is using more than one library with some variables or functions having same name.
- The name collision may occur for global variables, global functions, classes etc.
- Default namespace is global namespace and can access global data and functions by proceeding (::) operator.
- We can create our own namespace. And anything declared within namespace has scope limited to namespace.



Module as Abstraction

Pseudo Random Number Generator in C++

This module would typically be placed in its own file, and then imported wherever it is needed in C++ program.

Bindings of names made inside the namespace may be partially or totally hidden on the outside – but not destroyed. In C++, where namespaces can appear only at the **outermost** level of lexical nesting, integer seed would retain its value throughout the execution of the program, even though it is visible only to **set_seed** and **rand_int**.

```
#include <time.h>
namespace rand_mod {
    unsigned int seed = time(0); // initialize current time of day
    const unsigned int a = 48271;
    const unsigned int m = 0x7fffffff;

    void set_seed(unsigned int s){
        seed = s;
    }
    unsigned int rand_int(){
        return seed = (a * seed) % m;
    }
}
```

Figure 3.6 Pseudorandom number generator module in C++.

Uses the linear congruential method, with a default seed token from the current time of day. While there exist much better (more random) generators, this one is simple, and acceptable for many purposes.

Module as Abstraction

Pseudo Random Number Generator in C++

Outside the **rand_mod** namespace, C++ allows **set_seed** and **rand_int** to be accessed as **rand_mod::set_seed** and **rand_mod::rand_int**. The seed variable could also be accessed as **rand_mod::seed**, but this is probably not a good idea, and the need for the **rand_mod** prefix means it's unlikely to happen by accident.

The need for the prefix can be eliminated, on a name-by-name basis, with a using directive:

```
using rand_mod:: rand_int;  
...  
int = rand_int();
```

Alternatively, the full set of names declared in a namespace can be made available at once:

```
using namespace rand_mod;  
...  
set_seed(12345);  
int r = rand_int();
```

using rand_mod:: rand_int; Less memory used.

using namespace rand_mod; More memory used.

Imports and Exports

- Some languages allow the programmer to specify that names exported from modules be usable only in restricted ways.
- Variables may be exported read-only. Variables of that type may be declared, passed as arguments and possibly compared or assigned to one another, but not manipulated in other way.
- A module that names must be explicitly imported into it is called closed scopes.

Languages with closed module: **Modula 3, Haskell.**

- A module that do not require imports are called to be open scopes.

Languages with selectively open module: **C++, Ada, Java, C#, Python**

Using **import**, **using** directives. Common Option Now.

Modules as Managers

Modules facilitate the construction of abstractions by allowing data to be made private to the subroutines that use them. When used as in Figure 3.6, however, each module defines a single abstraction. Continuing our previous example, there are times when it may be desirable to have more than one pseudorandom number generator. If we want to have several generators, we can make our namespace a “manager” for instances of a generator type, which is then exported from the module. The manager idiom requires additional subroutines to create/initialize and possibly destroy generator instances. [\[Manager like Class\]](#)

```
using rand_mgr::generator;
generator *g1 = rand_mgr::create();
generator *g2 = rand_mgr::create();
...
using rand_mgr::read_int;
int r1 = rand_int(g1);
int r2 = rand_int(g2);
```

Figure 3.7 Manager module for pseudorandom numbers in C++.

```
#include <time.h>
namespace rand_mgr {
    const unsigned int a = 48271;
    const unsigned int m = 0x7fffffff;

    typedef struct {
        unsigned int seed;
    } generator; // struct Like Object

    generator* create() {
        generator* g = new generator;
        g->seed = time(0);
        return g;
    } // function Like Constructor

    void set_seed(generator* g, unsigned int s) {
        g->seed = s;
    } // function Like Member Method

    unsigned int rand_int(generator* g) {
        return g->seed = (a * g->seed) % m;
    } // function Like Member Method
}
```

Module Types and Classes

```
#include <time.h>
namespace rand_mgr {
    const unsigned int a = 48271;
    const unsigned int m = 0x7fffffff;

    typedef struct {
        unsigned int seed;
    } generator;

    generator* create() {
        generator* g = new generator;
        g->seed = time(0);
        return g;
    }

    void set_seed(generator* g, unsigned int s) {
        g->seed = s;
    }

    unsigned int rand_int(generator* g) {
        return g->seed = (a * g->seed) % m;
    }
}
```

```
class rand_gen {
    unsigned int seed = time(0);
    const unsigned int a = 48271;
    const unsigned int m = 0x7fffffff;

public:
    void set_seed(unsigned int s) {
        seed = s;
    }
    unsigned int rand_int() {
        return seed = (a * seed) % m;
    }
}; // default constructor omitted.
```

The way to use class:

```
rand_gen *g = new rand_gen();
...
int r = g->rand_int();
```


Dynamic Scoping

- In a language with dynamic scoping, the bindings between names and objects depend on the flow of control at run time, and in particular on the order in which subroutines are called.
- In comparison to the static scope rules, dynamic scope rules are generally quite simple: the **“current”** binding for a given name is the one encountered most recently during execution, and not yet destroyed by returning from its scope.

Dynamic Scoping

- Consider the program in **Figure 3.9**. If static scoping is in effect, this program Static vs dynamic scoping prints a 1. If dynamic scoping is in effect, the output depends on the value read at line 8 at run time: if the input is positive, the program prints a 2; otherwise it prints a 1.
- Why the difference? At issue is whether the assignment to the variable **n** at line 3 refers to the global variable declared at line 1 or to the local variable declared at line 5.
- Static scope rules require that the reference resolve to the closest lexically enclosing declaration, namely the global **n**. Procedure **first** changes **n** to 1, and line 12 prints this value.
- Dynamic scope rules, on the other hand, require that we choose the most recent, active binding for **n** at run time.

```
1. n : integer                -- global declaration
2. procedure first()
3.     n := 1
4. procedure second()
5.     n : integer            -- local declaration
6.     first()
7.     n := 2
8.     if read_integer() > 0
9.         second()
10.    else
11.        first()
12.    write_integer(n)
```

Figure 3.9: Static vs dynamic scoping

Run-time Errors with Dynamic Scoping

With dynamic scoping, errors associated with the referencing environment may not be detected until run time. In **Figure 3.10**, for example, the declaration of local variable **max_score** in procedure **foo** accidentally redefines a global variable used by function **scaled score**, which is then called from **foo**.

```
max_score : integer      -- maximum possible score

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

Figure 3.10

Since the global **max_score** is an integer, while the local **max_score** is a floating-point number, dynamic semantic checks in at least some languages will result in a type clash message at run time. If the local **max_score** had been an integer, no error would have been detected, but the program would almost certainly have produced incorrect results. This sort of error can be very hard to find. [\[unintended local causes the problem\]](#)

The Meaning of Name Within a Scope

SECTION 9

Name, Object, 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

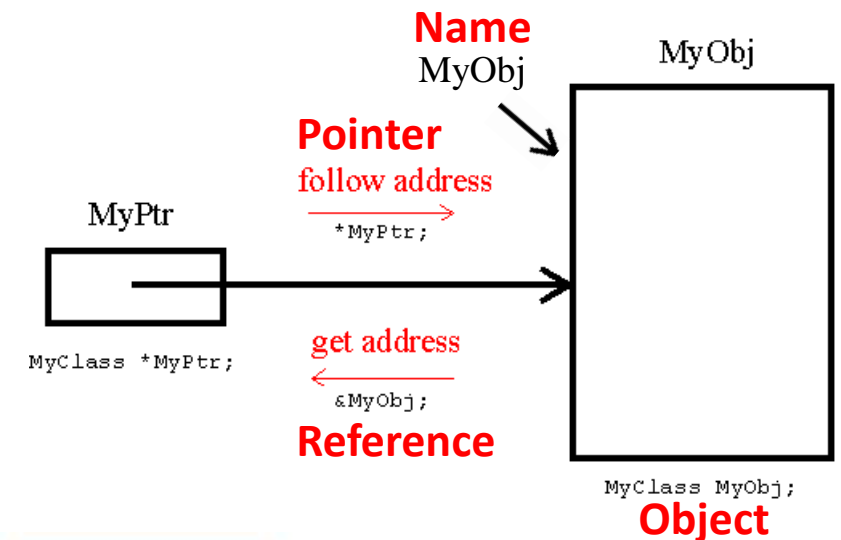
Typically a binding is between a name and the object it refers to.

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).

A Pointer holds the address of an object



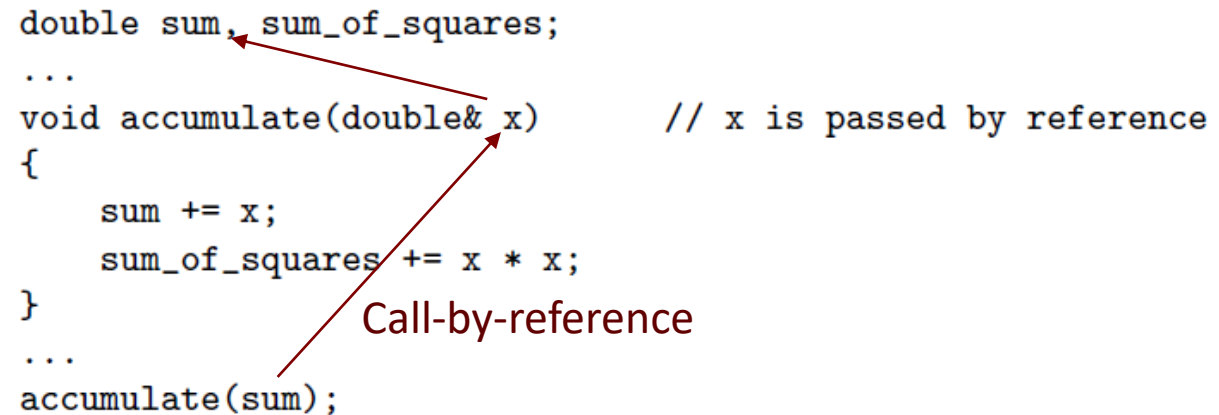
The Meaning of Names within a Scope

- Two or more names that refer to the same object at the same point in the program are said to be **aliases**.
- A name that can refer to more than one object at a given point in the program is said to be **overloaded**.
- Redefinition of an name is **overriding**.
- **Overloading** is related to the more general subject of **polymorphism**, which allows a subroutine or other program fragment to behave in different ways depending on the types of its arguments.

Aliasing

- What are aliases good for?
 - space saving - modern data allocation methods are better
 - multiple representations – unions/variants are better
 - Pointer-based data structure - linked data structures – different way of traversal.
 - Call by reference in C++
(Textbook 4e missing a statement)

```
double sum, sum_of_squares;
...
void accumulate(double& x)      // x is passed by reference
{
    sum += x;
    sum_of_squares += x * x;
}
...
accumulate(sum);
```



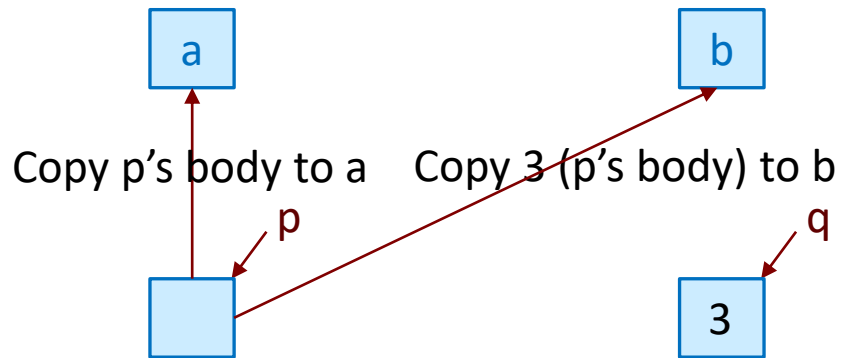
Call-by-reference

accumulate(double &x) is a call-by-reference function. **sum** and **x** are aliases.

So, $\text{sum} = \text{sum} + \text{sum} \rightarrow \text{sum} = 2 * \text{sum}(\text{old})$
 $\text{sum_of_squares} += (2 * \text{sum}(\text{old}))^2$
 $\text{sum_of_squares} += (\text{sum}(\text{new}))^2$

Aliases and Code Improvement

```
int a, b, *p, *q;  
...  
a = *p;    /* read from the variable referred to by p */  
*q = 3;    /* assign to the variable referred to by q */  
b = *p;    /* read from the variable referred to by p */
```



a and *p, b and *q are not aliases.
These operations just copy value
from the body of two pointers.

Overloading

- some overloading happens in almost all languages
 - integer + vs. real +
 - read and write in Pascal
 - function return in Pascal
- some languages get into overloading in a big way
 - Ada
 - C++

Overloading and Overriding functions

- overloaded functions - two different things with the same name; in C++
- overload norm

```
int      norm (int a)      {return a>0 ? a : -a;}
```

```
complex norm (complex c ) { // ...
```

- polymorphic functions -- one thing that works in more than one way
 - in Modula-2: function min (A : array of integer); ...
 - in Smalltalk

Generic Functions

- generic functions (modules, etc.) - a syntactic template that can be instantiated in more than one way **at compile time**
 - via macro processors in C++ **(macro)**
 - built-in in C++ **(template)**
 - in Clu
 - in Ada

Overloading of enumeration constants in Ada

- A slightly more sophisticated form of overloading appears in the enumeration constants of Ada.
- The constants **oct** and **dec** refer either to months or to numeric bases, depending on the context in which they appear.
- Within the symbol table of a compiler, overloading must be handled by arranging for the lookup routine to return a list of meanings for the requested name. The semantic analyzer must then choose from among the elements of the list based on context. When the context is not sufficient to decide, then the semantic analyzer must announce an error. Most languages that allow overloaded enumeration constants allow the programmer to provide appropriate context explicitly.

```
declare
    type month is (jan, feb, mar, apr, may, jun,
                  jul, aug, sep, oct, nov, dec);
    type print_base is (dec, bin, oct, hex);
    mo : month;
    pb : print_base;
begin
    mo := dec;           -- the month dec (since mo has type month)
    pb := oct;           -- the print_base oct (since pb has type print_base)
    print(oct);          -- error! insufficient context
                        -- to decide which oct is intended
```

Figure 3.11 Overloading of enumeration constants in Ada

- In Ada, for example, one can say
`print(month'(oct));`
- In Modula 3/C#, for example, one can say
`mo := month.dec;`
`pb := print_base.oct;`

Overloading of Built-in Operators

- It is one of the many exciting features of C/C++
- Important technique that has enhanced the power of extensibility of C/C++
- C/C++ tries to make the user-defined data types behave in much the same way as the built-in types
- C++ permits us to add two variables of user-defined types with the same syntax that is applied to the basic types.

- **Operator Overloading in C++:**

```
class complex {  
    double real, imaginary;  
    ...  
public:  
    complex operator+(complex other) {  
        return complex(real + other.real, imaginary + other.imaginary);  
    }  
    ...  
};  
...  
complex A, B, C;  
...  
C = A + B;           // uses user-defined operator+
```

- In **C++** and **C#**, which are object-oriented, **A + B** may be short for either **operator+(A, B)** or **A.operator+(B)**.
- In the latter case, **A** is an instance of a class (module type) that defines an **operator+** function.
- **C#** has similar syntax.

Binding of Referencing Environments I

Binding

SECTION 10

Binding of Referencing Environments

- The **referencing environment** of a statement is the collection of all names that are visible in the statement.
- **Static scope rules** specify that the referencing environment depends on the lexical nesting of program blocks in which names are declared.
- **Dynamic scope rules** specify that the referencing environment depends on the order in which declarations are encountered at run time.

Deep and Shallow Binding

- Procedure **print_selected_records** is a general-purpose routine that knows how to traverse the records in a database. It takes as parameters a database, a predicate to make print/don't print decisions, and a subroutine that knows how to format the data in the records of this database.
- Here we have hypothesized that **print_person** uses the value of nonlocal variable line length to calculate the number and width of columns in its output.
- In a language with dynamic scoping, it is natural for procedure **print_selected_records** to declare and initialize this variable locally, knowing that code inside **print_routine** will pick it up if needed. For this coding technique to work, the referencing environment of **print_routine** must not be created until the routine is actually called by **print_selected_records**.
- This late binding of the referencing environment of a subroutine that has been passed as a parameter is known as **shallow binding**. It is usually the default in languages with **dynamic scoping**.

Note: dynamic scoping using reference environment at run-time. Deep binding fixes RE when function is really called.

```
type person = record
```

```
...
age : integer
...
```

```
threshold : integer
people : database
```

```
function older_than_threshold(p : person) : boolean
return p.age ≥ threshold
```

Adjust for the printer hardware

```
procedure print_person(p : person)
-- Call appropriate I/O routines to print record on standard output.
-- Make use of nonlocal variable line_length to format data in columns.
```

```
...
procedure print_selected_records(db : database;
    predicate, print_routine : procedure)
line_length : integer

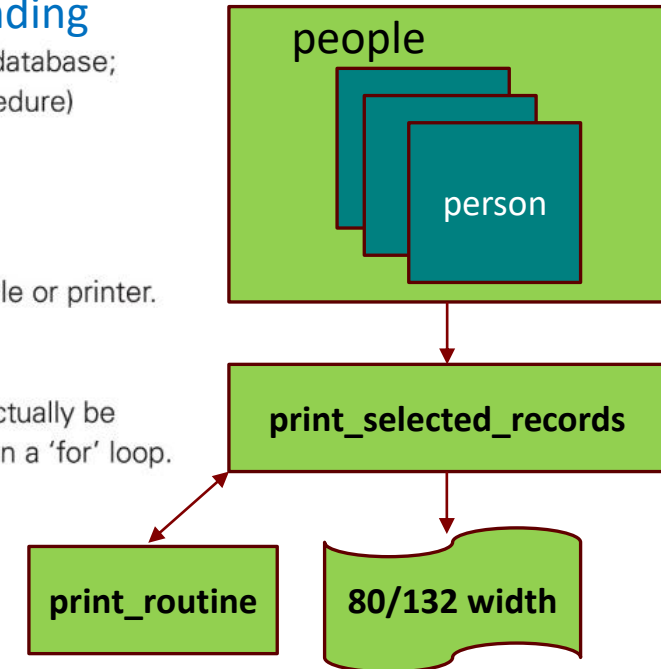
if device_type(stdout) = terminal
    line_length := 80
else -- Standard output is a file or printer.
    line_length := 132
foreach record r in db
    -- Iterating over these may actually be
    -- a lot more complicated than a 'for' loop.
    if predicate(r)
        print_routine(r)
```

```
-- main program
```

```
...
threshold := 35
print_selected_records(people, older_than_threshold, print_person)
```

Figure 3.13

shallow binding



Deep and Shallow Binding

- For function older than threshold, by contrast, shallow binding may not work well. If, for example, procedure **print_selected_records** happens to have a local variable named **threshold**, then the variable set by the main program to influence the behavior of older than threshold will not be visible when the function is finally called, and the predicate will be unlikely to work correctly.
- In such a situation, the code that originally passes the function as a parameter has a particular referencing environment in mind; it does not want the routine to be called in any other environment.
- It therefore makes sense to bind the environment at the time the routine is first passed as a parameter, and then restore that environment when the routine is finally called. This early binding of the referencing environment is known as **deep binding**.

Note: dynamic scoping using reference environment at run-time. Deep binding fixes RE when function parameter is passed.

```
type person = record
```

```
...
age : integer
...
```

```
threshold : integer
people : database
```

Don't like global effects on the threshold.

```
function older_than_threshold(p : person) : boolean
return p.age ≥ threshold
```

```
procedure print_person(p : person)
-- Call appropriate I/O routines to print record on standard output.
-- Make use of nonlocal variable line_length to format data in columns.
...
```

```
procedure print_selected_records(db : database;
    predicate, print_routine : procedure)
line_length : integer

if device_type(stdout) = terminal
    line_length := 80
else -- Standard output is a file or printer.
    line_length := 132
foreach record r in db
    -- Iterating over these may actually be
    -- a lot more complicated than a 'for' loop.
    if predicate(r)
        print_routine(r)
```

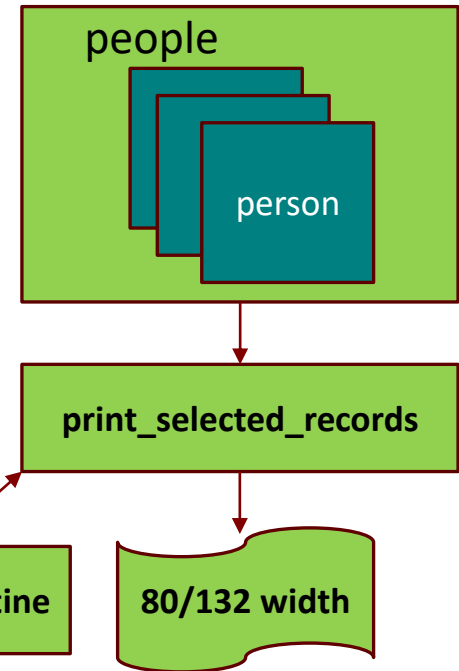
```
-- main program
```

```
... binding fixed here.
```

```
threshold := 35
```

```
print_selected_records(people, older_than_threshold, print_person)
```

Figure 3.13



Subroutine Closures

- If your language lets you
 - (1) store subroutines as values, and
 - (2) nest subroutines, (recursive)then you can make closures.
- A **closure** is a **subroutine** that refers to variables defined in an **enclosing scope**.
- Deep binding is implemented by creating an explicit representation of a referencing environment and bundling it together with a reference to the subroutine.
- The bundle as a whole is referred to as a closure.

Subroutine Closures

- A closure in a language with static scoping capture the current instance of every object, at the time the closure is created. When the closure's subroutine is called, it will find these captured instances, even if newer instances have subsequently been created by recursive calls.
- Static scoping (by declaration of functions) with shallow binding is imaginable.
- Python program using static scoping but deep binding because of the interpreter-based nature.

```
def A(I, P):  
    def B():  
        print(I)  
    # body of A:  
    if I > 1:  
        P()  
    else:  
        A(2, B)  
  
def C():  
    pass      # do nothing  
  
A(1, C)      # main program
```

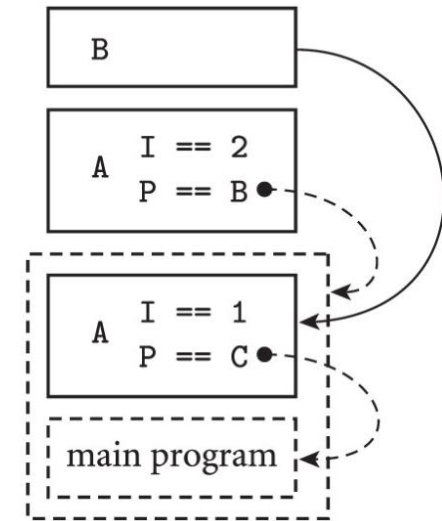
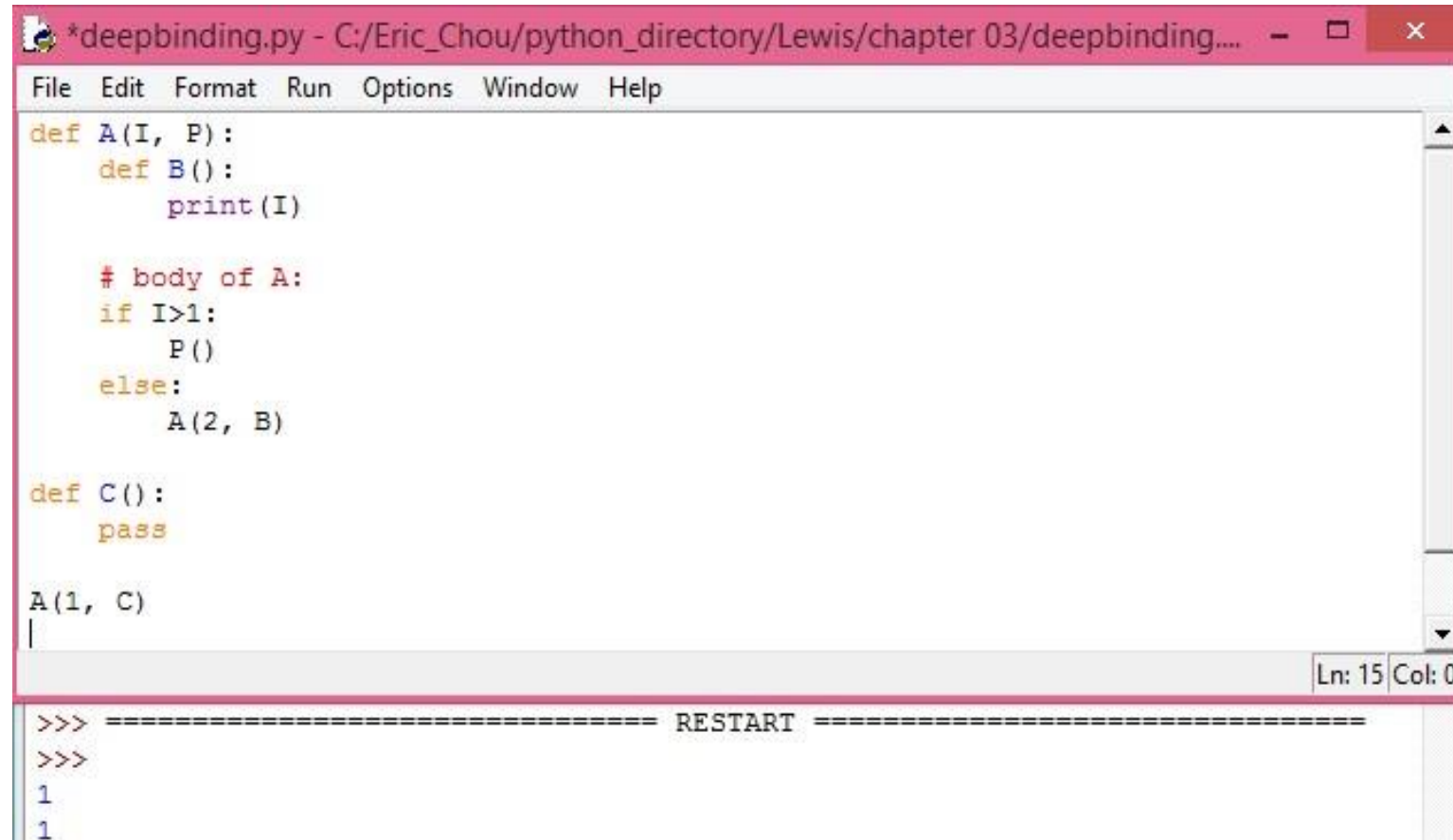


Figure 3.14 Deep binding in Python.

At right is a conceptual view of the run-time stack. Referencing environments captured in closures are shown as dashed boxes and arrows. When B is called via formal parameter P, two instances of I exist. Because the closure for P was created in the initial invocation of A, B's static link (solid arrow) points to the frame of that earlier invocation. B uses that invocation's instance of I in its print statement, and the output is a 1.

Note: With shallow binding, it should print 2.

Demo Program: deepbinding.py



```
*deepbinding.py - C:/Eric_Chou/python_directory/Lewis/chapter 03/deepbinding....
File Edit Format Run Options Window Help

def A(I, P):
    def B():
        print(I)

    # body of A:
    if I>1:
        P()
    else:
        A(2, B)

def C():
    pass

A(1, C)
|

Ln: 15 Col: 0

>>> ===== RESTART =====
>>>
1
1
```

Binding of Referencing Environments II

Closure and Lambda

SECTION 11

First-Class Values and Unlimited Event

- A value in a programming language is said to have first-class status if it can be passed as a parameter, returned from a subroutine, or assigned into a variable.
- By contrast, a “second-class” value can be passed as a parameter, but not returned from a subroutine or assigned into a variable. **[Algol]**
- A “third-class” value cannot even be passed as a parameter. **[Label]**

Note: A language construct is said to be a First-Class value in that language when there are no restrictions on how it can be created and used: when the construct can be treated as a value without restrictions.

Returning a First-Class Sub-Routine in Scheme

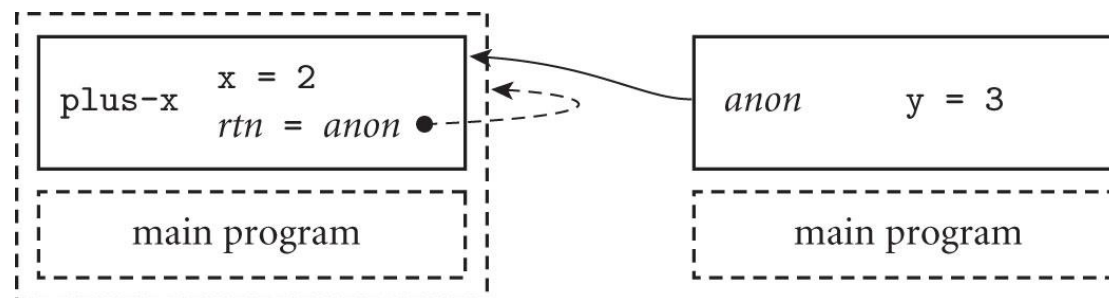
First-class subroutines in a language with nested scopes introduce an additional level of complexity: they raise the possibility that a reference to a subroutine may outlive the execution of the scope in which that routine was declared. Consider the example in Scheme on the right:

$f \Rightarrow \text{lambda } (2)(\text{lambda } (y)(+ 2 y))$
 $(f \ 3) \Rightarrow \text{lambda}(3)(+ 2 y)$

```
1. (define plus-x (lambda (x)
2.   (lambda (y) (+ x y))))
3. ...
4. (let ((f (plus-x 2)))
5.   (f 3))
```

One Reference Environment; returns 5

Here the `let` construct on line 4 declares a new function, **f**, which is the result of calling **plus-x** with argument **2**. Function **plus-x** is defined at line 1. It returns the (unnamed) function declared at line 2. But that function refers to parameter **x** of **plus-x**. When **f** is called at line 5, its referencing environment will include the **x** in **plus-x**, despite the fact that **plus-x** has already returned (see Figure 3.15). Somehow we **must** ensure that **x** remains available.



```
(define plus-x (lambda (x) (lambda (y) (+ x y)
                                )
                  )
)
A = lambda (y) (x + y)
B = lambda (x) (A)
define plus-x (B)
```

Object Closures

In object-oriented languages, there is an alternative way to achieve a similar effect: we can encapsulate our subroutine as a method of a simple object, and let the object's fields hold context for the method.

An Object Closure in Java: (Object works as closure.)

```
interface IntFunc {
    public int call(int i);
}

class PlusX implements IntFunc {
    final int x;
    PlusX(int n) { x = n; }
    public int call(int i) { return i + x; }
}

...
IntFunc f = new PlusX(2);
System.out.println(f.call(3));           // prints 5
```

Function Objects in C++:

In C++, an object of a class that overrides operator() can be called as if it were a function:

```
class int_func {
public:
    virtual int operator()(int i) = 0;
};                                     // abstract function (no implementation)

class plus_x : public int_func {
    const int x; // x = n defaulted at construction
public:
    plus_x(int n) : x(n) { }
    virtual int operator()(int i) { return i + x; }
};

...
plus_x f(2);    // x = 2
cout << f(3) << "\n";           // prints 5
// i = 3, x = 2 => return 5
```

Object f could also be passed to any function that expected a parameter of class **int_func**.

Object Closures

Delegates in C#:

In C#, a first-class subroutine is an instance of a delegate type:

```
delegate int IntFunc(int i);
```

This type can be instantiated for any subroutine that matches the specified argument and return types. That subroutine may be static, or it may be a method of some object:

```
static int Plus2(int i) { return i + 2; }
...
IntFunc f = new IntFunc(Plus2);
Console.WriteLine(f(3));           // prints 5

class PlusX {
    int x;
    public PlusX(int n) { x = n; }
    public int call(int i) { return i + x; }
}
...
IntFunc g = new IntFunc(new PlusX(2).call);
Console.WriteLine(g(3));           // prints 5
```

Delegates and Unlimited Extent:

Remarkably, though C# does not permit subroutines to nest in the general case, version 2 of the language allows delegates to be instantiated in-line from anonymous methods. These allow us to mimic the code of Example for Scheme:

```
static IntFunc PlusY(int y) {
    return delegate(int i) { return i + y; };
}
...
IntFunc h = PlusY(2);
```

Here **y** has unlimited extent! The compiler arranges to allocate it in the heap, and to refer to it indirectly through a hidden pointer, included in the closure. This implementation incurs the **cost** of dynamic storage allocation only when it is needed; local variables remain in the stack in the common case.

Lambda Expressions

A lambda expression is an anonymous function that you can use to create delegates or expression tree types. [Anonymous Inner Function or Anonymous Inner Class]

Lambda Expression in C#:

```
static IntFunc PlusY(int y){
    return i => i + y;
}
```

Here the keyword delegate of Example 3.35 has been replaced by an => sign that separate the anonymous function's parameter from the body. **[body is the return value]**

In a function with more than one parameter, the parameter list would be parenthesized; in a longer, more complicated function, the body could be a code block, with one or more explicit return statement.

Note: callback is a common application of Lambda.

Variety of Lambda Syntax:

```
(lambda (i j) (> i j) i j)      ; scheme
(int i, int j) => i > j ? i : j  // C#
fun i j -> if i > j then i else j (* OCaml *)
->(i, j){ i > j ? i : j }        # Ruby
```

- Each of these expressions evaluates to the larger of two parameters.
- In Scheme and OCaml, lambda expression is a function.

lambda function

```
; Scheme
(( lambda (i j) (> i j) i j) 5 8) ; evaluate to 8
(* OCaml *)
(fun i j -> if i > j then i else j) 5 8 (* to 8 *)
# Ruby
print ->(i, j){ i > j ? i : j }.call(5, 8) # Ruby like objects
/* C# */
Func<int, int, int> m = (i, j) => i > j ? i : j;
Console.WriteLine(m.Invoke(5, 8));
```

Lambda Expression in C++11

Lambda Introducer
& Capture Clause Parameter List Mutable
Specifications Exception
Specifications Return Type

[**=**] (**int** x) **mutable** **throw()** -> **int**

Lambda Body

```
{  
    int n = x + y;  
    return n;  
}
```

Lambda Expression in C++11

If **V** is a vector of integers, the following will print all elements less than 50:

```
for_each(
    V.begin(), V.end(), [](int e){if (e < 50) cout << e << " "; }
);
```

for_each: a standard library routine that applies its third parameter – a function to every element of a collection in the range specified by its first two parameters.

Variable capture in C++ lambda expressions: (k is a variable outside lambda expression)

```
[=](int e){if (e < k) cout << e << " "; }
[&](int e){if (e < k) cout << e << " "; }
```

These two can be used for the third parameter above. (with k=50).

Capture List:

- [=] //capture all of the variables from the enclosing scope by value [default]
- [&] //capture all of the variables from the enclosing scope by reference
- [] //a lambda with an empty capture clause is one with no external references. It accesses only variables that are local to the lambda
- [j, &k] // capture j's value and k's reference

Lambda Expression in Java

A lambda expression is like a method: it provides a list of formal parameters and a body

The formal parameters of a lambda expression may have either inferred or declared types

→ `s -> s.length()`

→ `(int x, int y) -> x + y`

`() -> 42`

```
(x, y, z) -> {  
    if (x) {  
        return y;  
    } else {  
        return z;  
    }  
}
```

}

Argument List	Arrow Token	Body
<code>(int x, int y)</code>	<code>-></code>	<code>x + y</code>

Return is implicit and can be omitted

A lambda body is either a single expression or a block

Lambda Expression in Java

Lambda expressions appear in Java 8 as well, but in a restricted form. In situations where they might be useful. Java has traditionally relied on an idiom known as a **functional interface**. The **Arrays.sort** routine, for example, expects a parameter of type Comparator.

```
class AgeComparator implements Comparator<Person> {  
    public int compare(Person p1, Person p2){  
        return Integer.compare(p1.age, p2.age);  
    }  
}  
  
Person[] people = ...  
...  
Arrays.sort(People, new AgeComparator());
```

Using Lambda Expression:

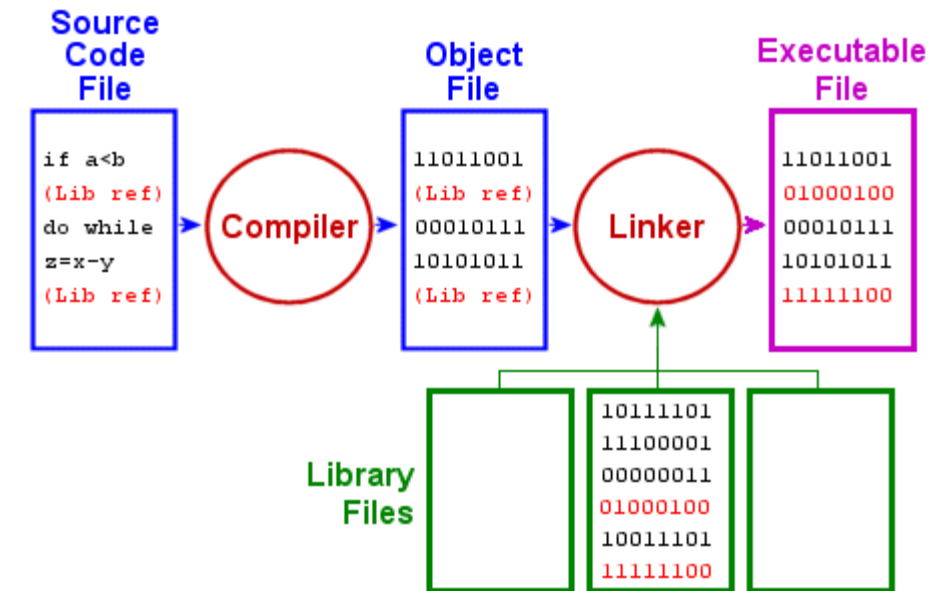
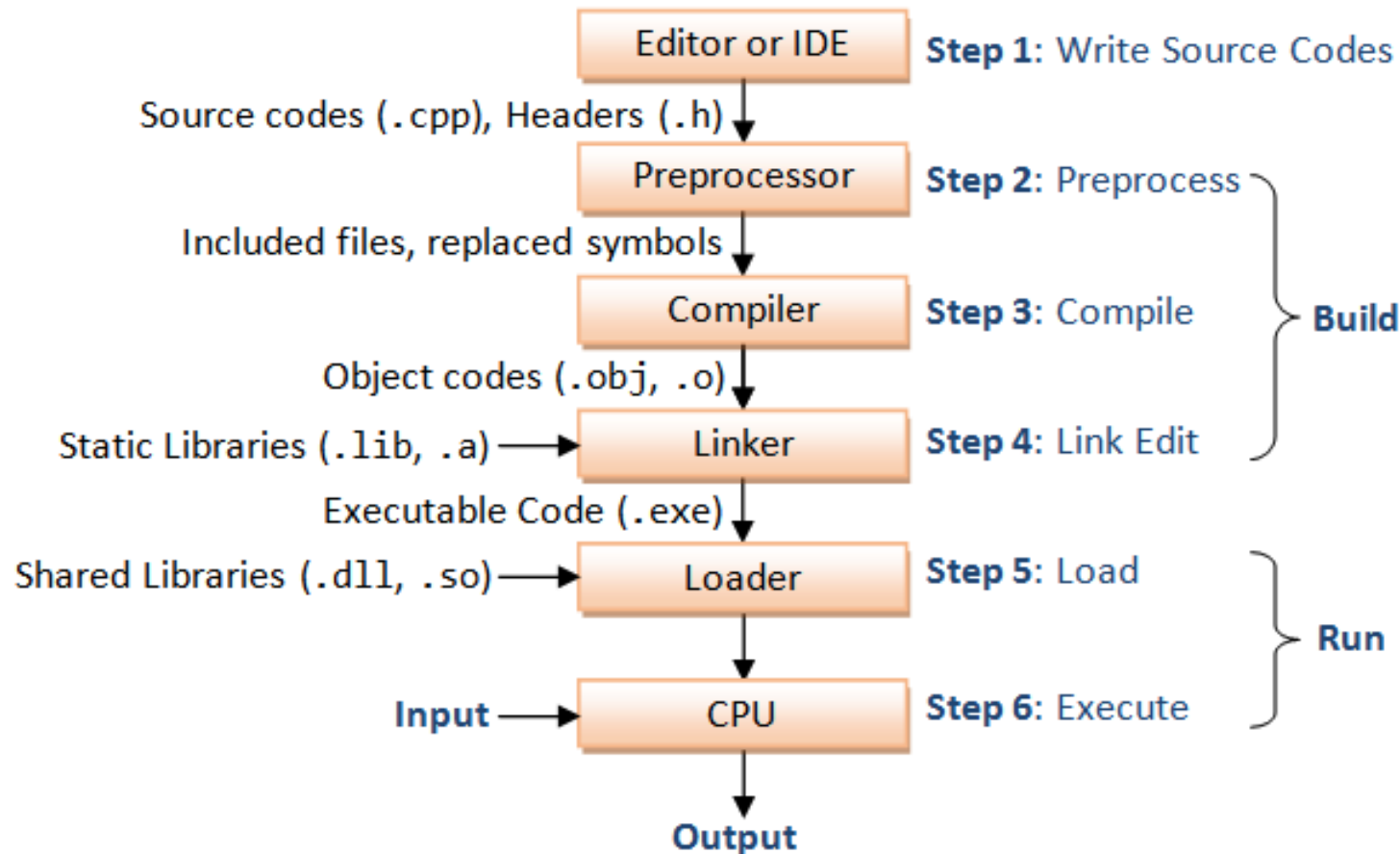
```
Arrays.sort(People, (p1, p2)->  
    Integer.compare(p1.age, p2.age)  
    /*  
     { May have the definition of compare here.  
       if it is custom-designed function.  
     }  
    */  
);
```

```
static <T> void sort(T[] a, Comparator<? super T> c)  
Sorts the specified array of objects according to the order induced by the specified comparator.
```

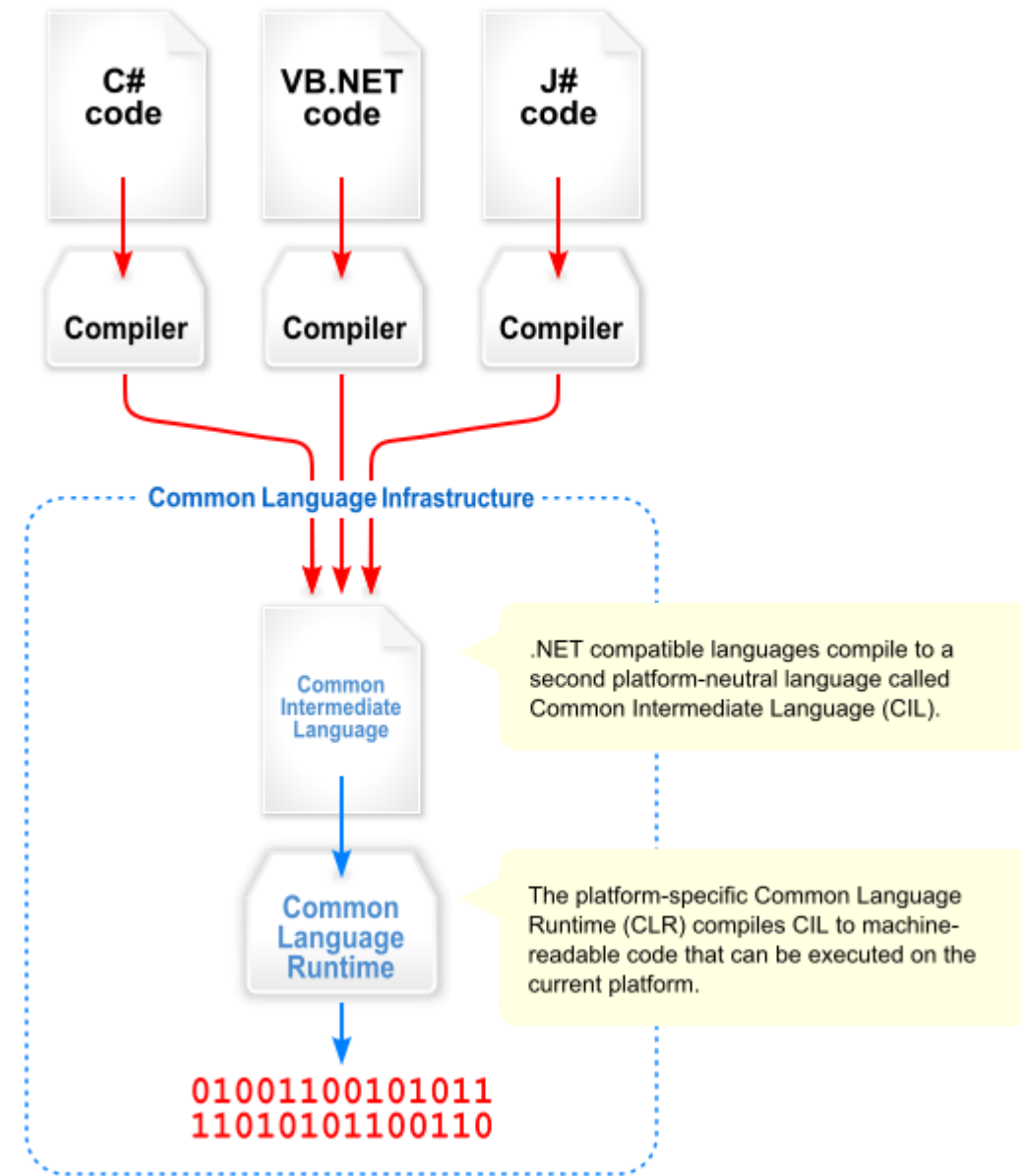
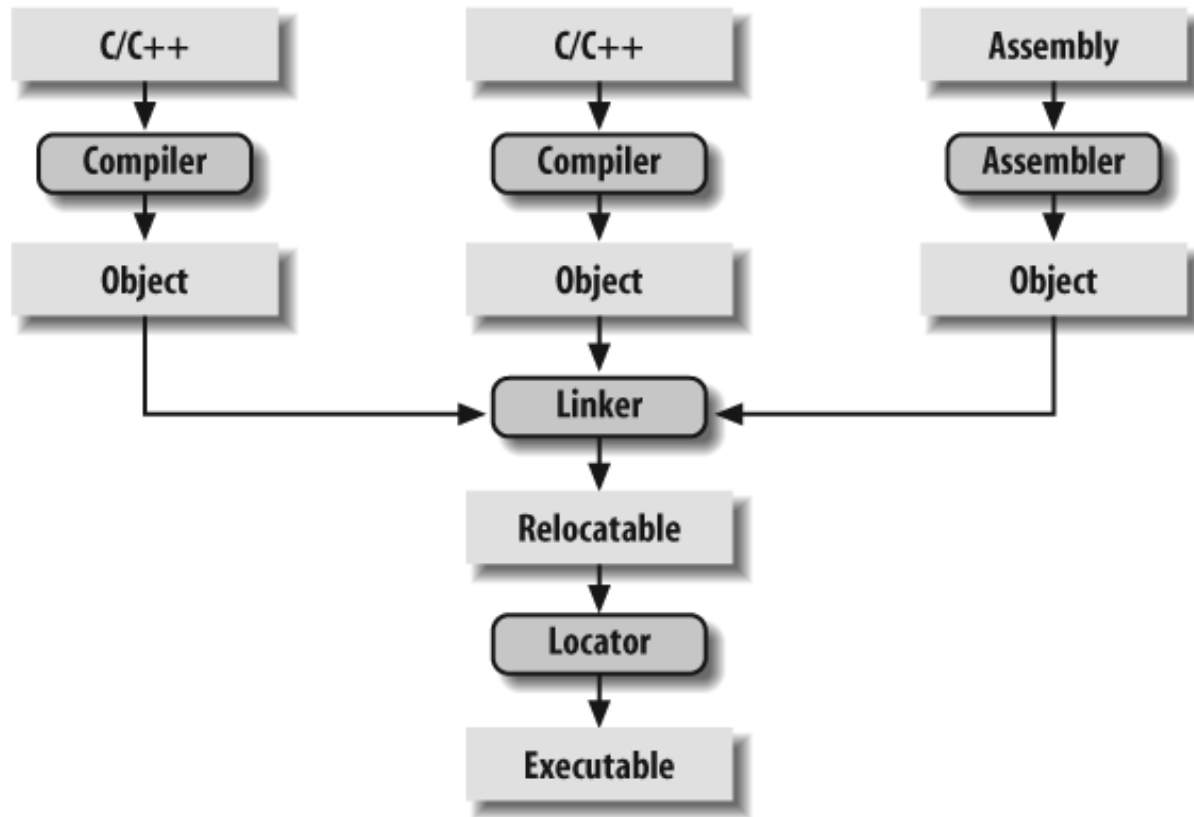
Name in Compilation Process

SECTION 12

C++ Compilation Process



Separate Compilation



Macro Expansion (Preprocessor)

Within the code of Zlib, `infback9.c` has few macros. Let's see an example of `NEEDBITS`.

```
#define NEEDBITS(n) \
do { \
    while (bits < (unsigned)(n)) \
        PULLBYTE(); \
} while (0)
```

But, the macro `NEEDBITS` depends on `PULLBYTE`

```
#define PULLBYTE() \
do { \
    PULL(); \
    have--; \
    hold += (unsigned long)(*next++) << bits; \
    bits += 8; \
} while (0)
```

Macro Expansion

The dependencies continue.

```
#define PULL() \
do { \
    if (have == 0) { \
        have = in(in_desc, &next); \
        if (have == 0) { \
            next = Z_NULL; \
            ret = Z_BUF_ERROR; \
            goto inf_leave; \
        } \
    } \
} while (0)

#define Z_NULL 0

#define Z_BUF_ERROR (-5)
```

Now. What does `NEEDBITS(3)` expand to?

Macro Expansion

Explore Macro Expansion - 5 step(s)

```
#define NEEDBITS(n) do { \
    while (bits < (unsigned)(n)) \
        PULLBYTE(); \
} while (0)
```

Original	Fully Expanded
1 NEEDBITS(3)	1 do { \
	2 while (bits < (unsigned)(3)) \
	3 do { \
	4 do { \
	5 if (have == 0) { \
	6 have = in(in_desc, &next); \
	7 if (have == 0) { \
	8 next = 0; \

Macro Expansion

Explore Macro Expansion - 5 step(s)

```
#define NEEDBITS(n) do { \
    while (bits < (unsigned)(n)) \
        PULLBYTE(); \
}
```

Original	Expansion #1 of 5
1 NEEDBITS(3)	1 do { \
	2 while (bits < (unsigned)(3))
	3 PULLBYTE(); \
	4 } while (0)

Explore Macro Expansion - 5 step(s)

```
#define PULLBYTE() do { \
    PULL(); \
    have--; \
    hold += (unsigned long)(*next++) << bits; \
    bits += 8; \
} while (0)
```

Expansion #1 of 5

```
1 do { \
2     while (bits <
3         PULLBYTE()
4     } while (0)
```

Expansion #2 of 5

```
1 do { \
2     while (bits
3         do { \
4             PULL(); \
5             have--; \
6             hold += (u
7             bits += 8;
8         } while (0); \
9     } while (0)
```

Macro Expansion

Explore Macro Expansion - 5 step(s)

```
#define PULL() do { \
    if (have == 0) { \
        have = in(in_desc, &next); \
        if (have == 0) { \
```

Expansion #2 of 5	Expansion #3 of 5
<pre>1 do { \ 2 while (bits < (unsigned) 8) \ 3 do { \ 4 PULL(); \ 5 have--; \ 6 hold += (have < 8); \ 7 bits += 8; \ 8 } while (0); \ 9 } while (0); \</pre>	<pre>1 do { \ 2 while (bits < (unsigned) 8) \ 3 do { \ 4 do { \ 5 if (have == 0) { \ 6 have = in(in_desc, &next); \ 7 if (have == 0) { \ 8 next = Z_NULL; \ 9 ret = Z_BUF_ERROR; \ 10 goto inf_leave; \ 11 } \ 12 } \ 13 } while (0); \ 14 have--; \</pre>

And so on..

Separate Compilation

Separately-compiled files in C provide a sort of poor man's modules

- Rules for how variables work with separate compilation are messy
- Language has been jury-rigged to match the behavior of the linker
 - **static** on a function or variable outside a function means it is usable only in the current source file (global) (otherwise, it requires a header file and **extern** call)
 - This **static** is a different notion from the **static** variables inside a function

Separate Compilation

- Separately-compiled files in C (continued)
 - **extern** on a variable or function means that it is declared in another source file
 - Functions headers without bodies are **extern** by default
 - **extern** declarations are interpreted as forward declarations if a later declaration overrides them
 - Variables or functions (with bodies) that don't say **static** or **extern** are either *global* or *common* (a Fortran term)

Conclusions

- The morals of the story:
 - language features can be surprisingly subtle
 - designing languages to make life easier for the compiler writer can be a **Good Thing**.
 - most of the languages that are easy to understand are easy to compile, and vice versa

Conclusions

- A language that is easy to compile often leads to
 - a language that is easy to understand
 - more good compilers on more machines (compare Pascal and Ada!)
 - better (faster) code
 - fewer compiler bugs
 - smaller, cheaper, faster compilers
 - better diagnostics