Names, Bindings and Scopes

204315 OPL

Outline

- Binding
- Scopes
- Names

Name, Binding and Scope

- A name is character strings used to represent something in the program
  - 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 in which the binding is active

Binding

Binding Time

- The point at which a binding is created
- Today we talk about the binding of identifiers to the variables they name

Static vs dynamic binding

- The terms STATIC and DYNAMIC are generally used to refer to things bound before run time and at run time, respectively
- 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

Binding lifetime

- Key events
  - creation of objects
  - creation of bindings
  - references to variables (which use bindings)
  - (temporary) deactivation of bindings
  - reactivation of bindings
  - destruction of bindings
  - destruction of objects

Garbage and dangling reference

- 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

Garbage collection

- Garbage Collection (GC):
  - In languages that deallocation of objects is not explicit.
  - Manual deallocation errors are among the most common and costly bugs in real-world programs.
  - Objects are to be deallocated implicitly when it is no longer possible to reach them from any program variable.
- Methodologies:,
  - Reference counting: Garbage is identified by having a reference count of zero.
  - Tracing GC: Garbage is object which is not reachable. an object is reachable if it is referenced by at least one variable in the program, either directly or through references from other reachable objects.

Scope

Scope: definition

- A scope is a program section of maximal size in which no bindings change, or at least in which no re-declarations are permitted
- The scope of a binding is determined statically or dynamically

Static scope

- In static scope rules, bindings are defined by the physical (lexical) structure of the program
- Initially, simple versions (such as early Basic) had only 1 scope - the entire program.
- Later, distinguished between local and global (such as in Fortran).
  - Local variable only “exists” during the single execution of that subroutine.
  - So, the variable is recreated and reinitialized each time it enters that procedure or subroutine. (Think of variables in C created in a loop, and which don’t exist outside the loop.)

```
1 for (int i=0; i<10; i++) {
2     cout << i;
3 }
4 cout << i
```

Static Scope Rules Examples

- one big scope (old Basic)

```
10 INPUT "What is your name: "; U$
20 PRINT "Hello "; U$
30 INPUT "How many stars do you want: "; N
40 S$ = ""
50 FOR I = 1 TO N
60 S$ = S$ + "*"
70 NEXT I
80 PRINT S$
90 INPUT "Do you want more stars? "; A$
100 IF LEN(A$) = 0 THEN GOTO 90
110 A$ = LEFT$(A$, 1)
120 IF A$ = "Y" OR A$ = "y" THEN GOTO 30
130 PRINT "Goodbye "; U$
140 END
```

Static Scope Rules Examples

- scope of a function (variables live through a function execution)
- Nested subroutines (have access to the variables defined in the parent)
- What get printed ?

```
1  def f1(a):
2      c = a + 1
3      def f2(b):
4          b = 5
5          print(b)
6          print(c)
7      f2(c)
8
9  f1(3)
```

```
1  def f1(a):
2      c = a + 1
3      f2(c)
4
5  def f2(b):
6      b = 5
7      print(b)
8      print(c)
9
10 f1(3)
```

Scope Rules [1/2]

- In most languages with subroutines, we OPEN a new scope on subroutine entry:
  - create bindings for new local variables,
  - deactivate bindings for global variables that are re-declared (these variable are said to have a "hole" in their scope)
  - make references to variables
- On subroutine exit:
  - destroy bindings for local variables
  - reactivate bindings for global variables that were deactivated

Scope Rules [2/2]

- Scope can also refer to something other than subroutine call.
- Example:

```
void test() {
    SmartStack<string> r;
    r.push("X");  r.push("Y");  r.push("Z");
    cout << "r ";  printStack(r);
    // enter new scope so that t can be constructed
    if (true) {
            SmartStack<string> t(r);
            t.pop(); t.push(“W”);
            cout << "r ";  printStack(r);
            cout << "t ";  printStack(t);
    }  // t is destroyed
}  // r is destroyed
```

Program Storage management

- Storage Allocation mechanisms
  - Static allocation
  - Stack-based
  - Heap-based

Static allocation

- Normally, allocation for the followings
  - code
  - Global variables
  - static or own variables
  - explicit constants (including strings, sets, etc.)
  - scalars may be stored in the instructions

Stack-based allocation

- Allocation for new function call
  - arguments and returns
  - local variables
  - temporaries
  - bookkeeping (saved registers, line number static link, etc.)
- Why a stack?
  - allocate space for recursive routines
    (not necessary in FORTRAN – no recursion)
  - reuse space
    (in all programming languages)

Call stack

- A storage for function calls

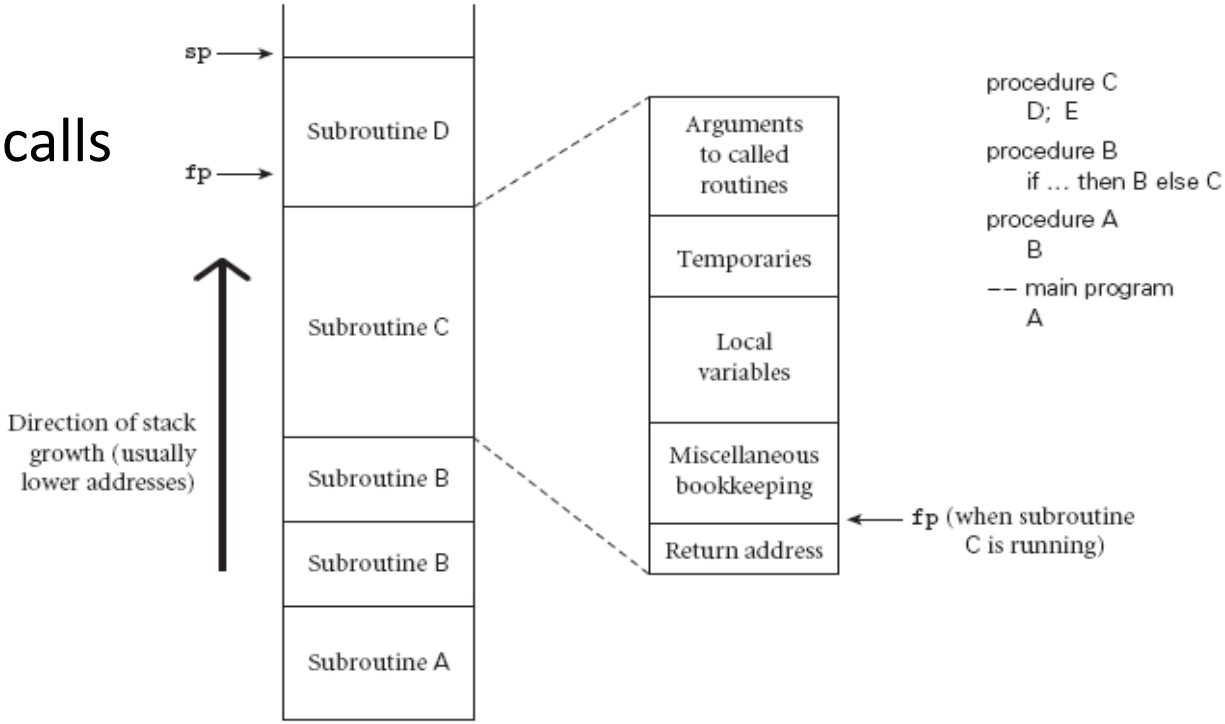
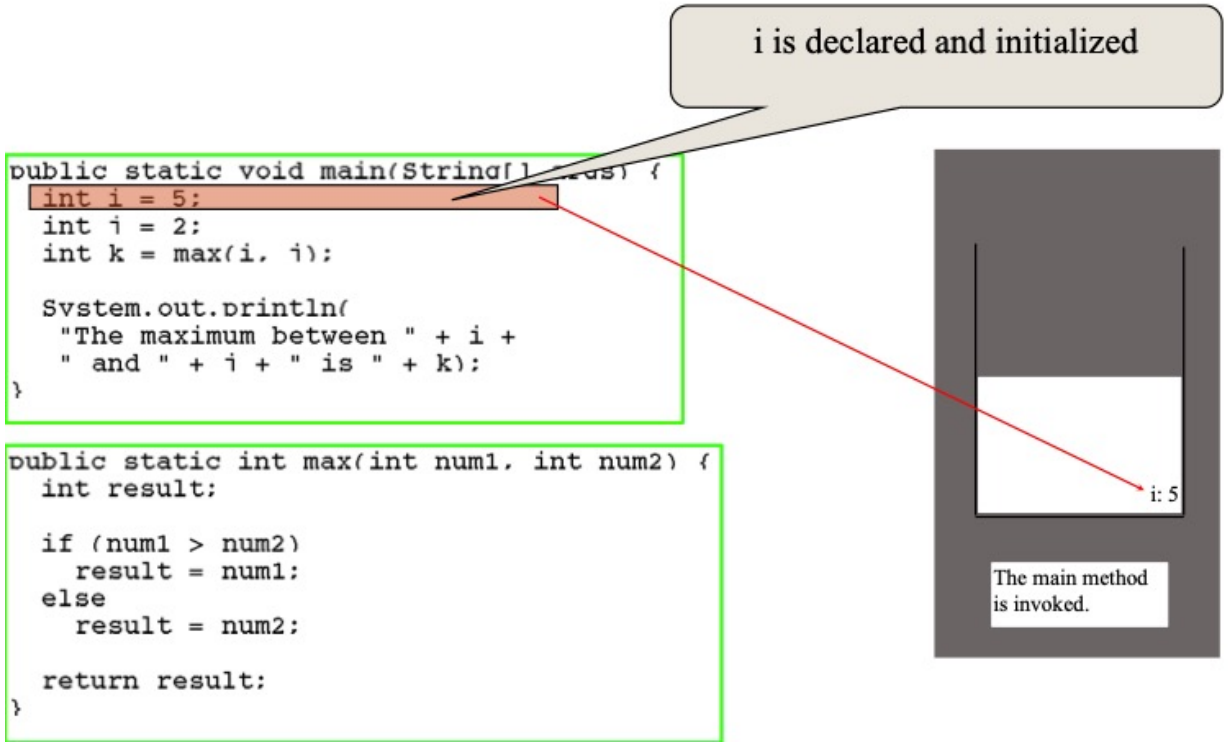
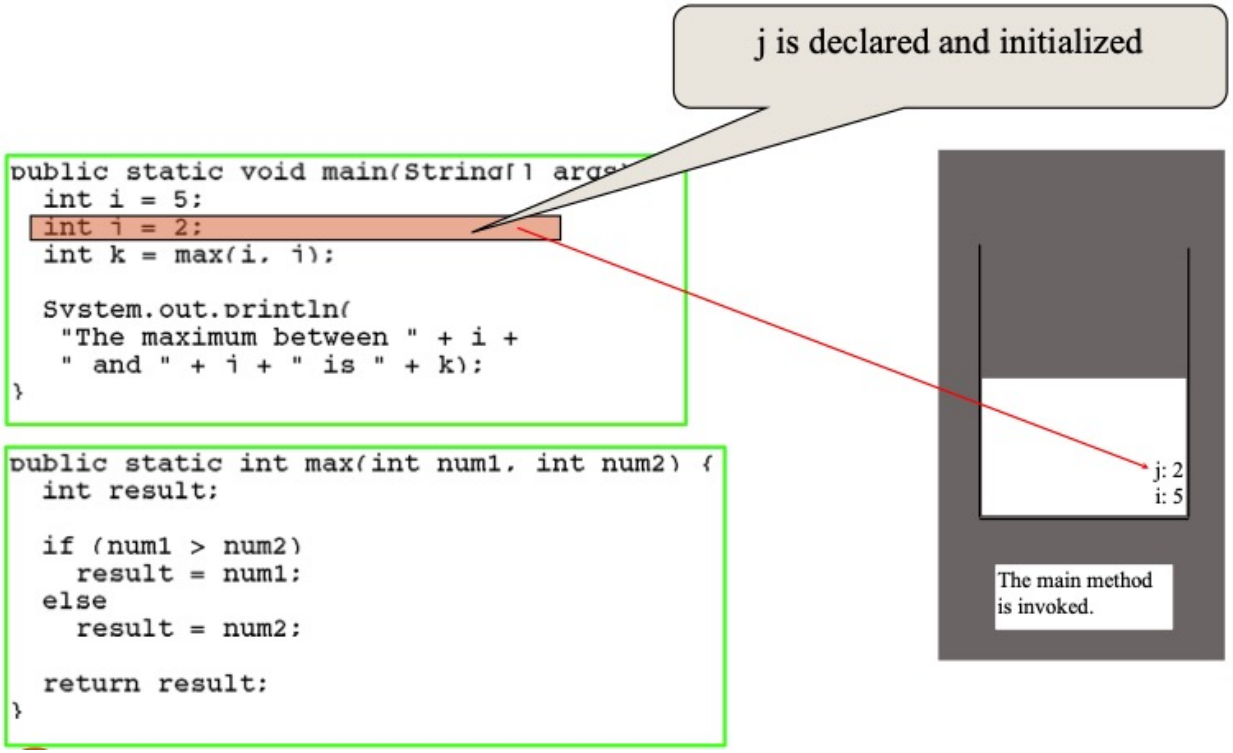


Figure 3.1 Stack-based allocation of space for subroutines. We assume here that subroutines have been called as shown in the upper right. In particular, B has called itself once, recursively, before calling C. If D returns and C calls E, E's frame (activation record) will occupy the same space previously used for D's frame. At any given time, the stack pointer (`sp`) register points to the first unused location on the stack (or the last used location on some machines), and the frame pointer (`fp`) register points to a known location within the frame of the current subroutine. The relative order of fields within a frame may vary from machine to machine and compiler to compiler.

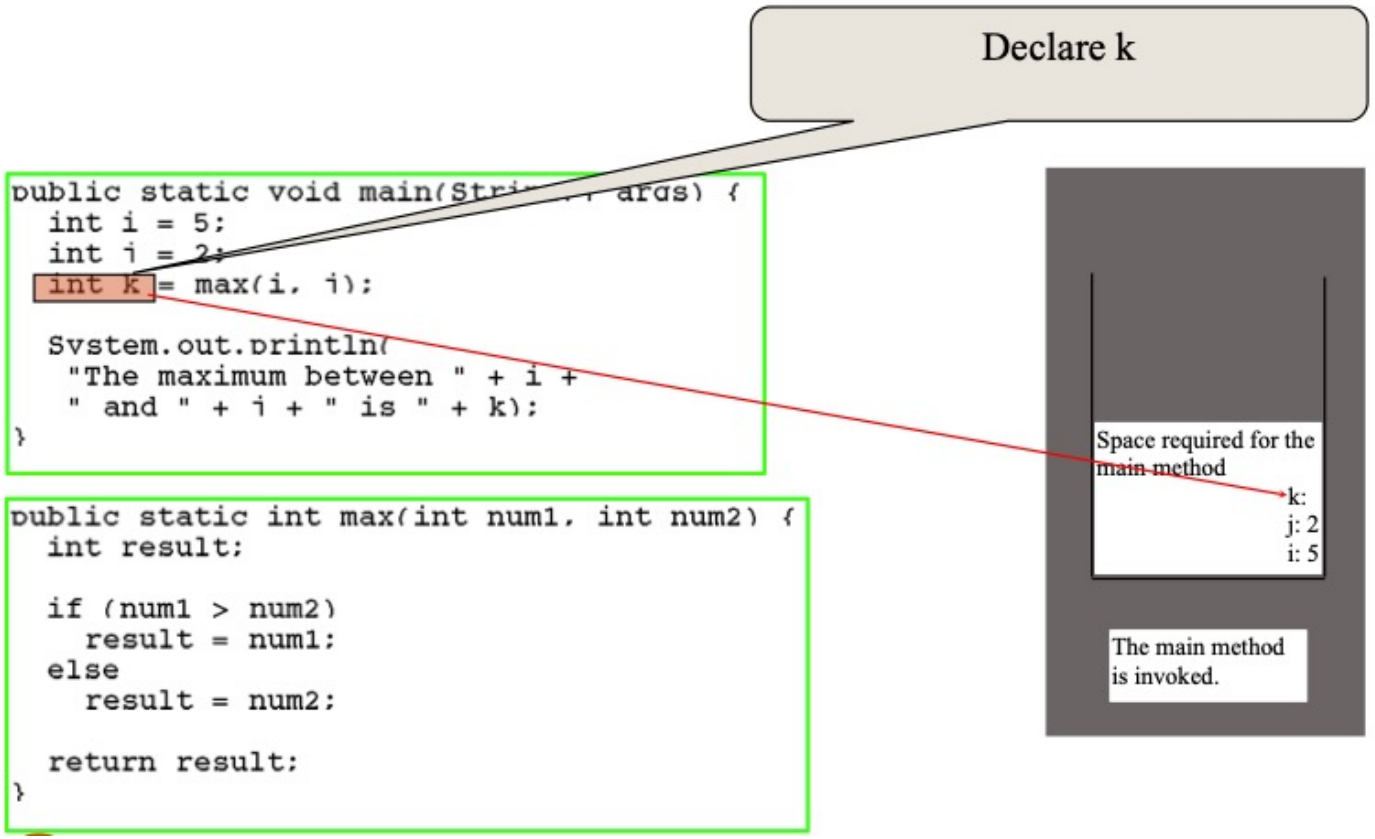
Call stack example in ? [try to answer]



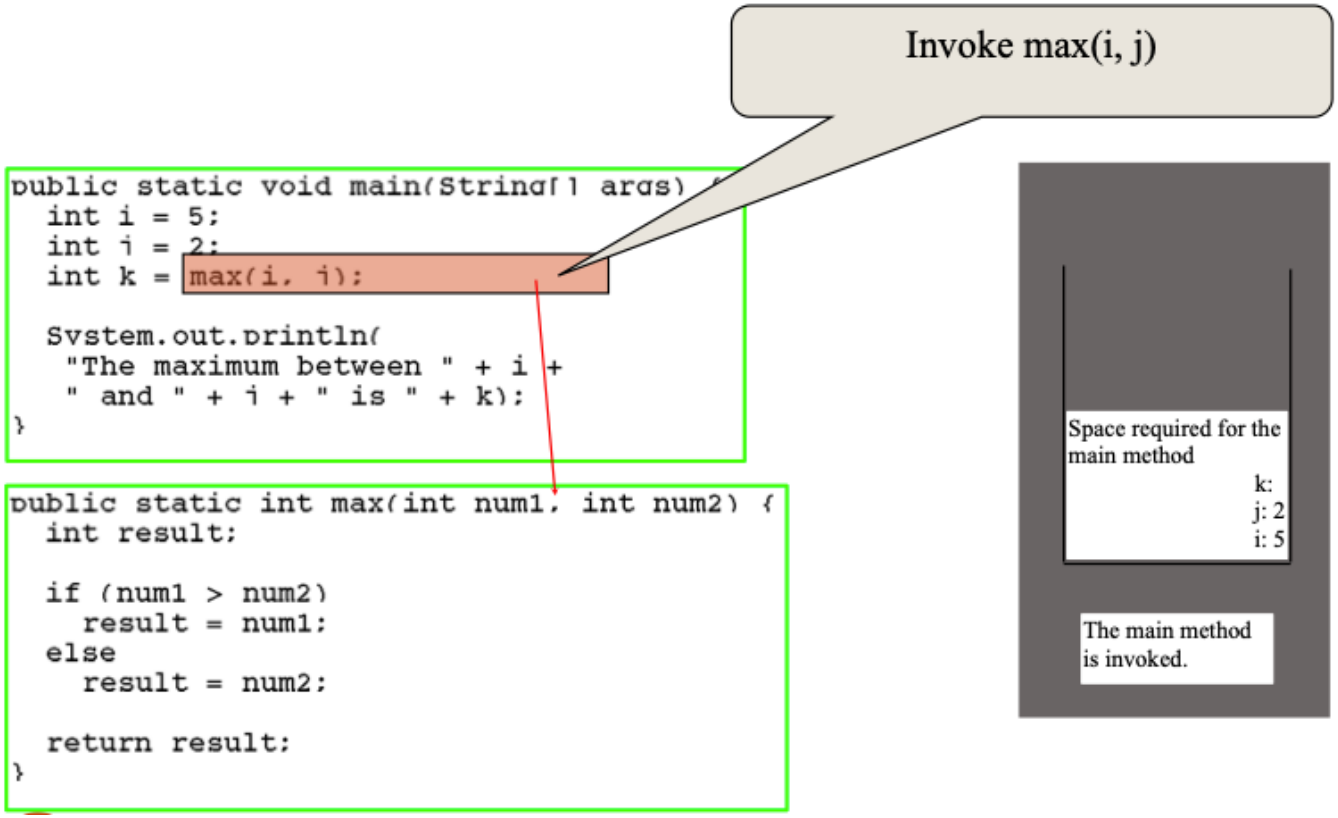
Call stack example



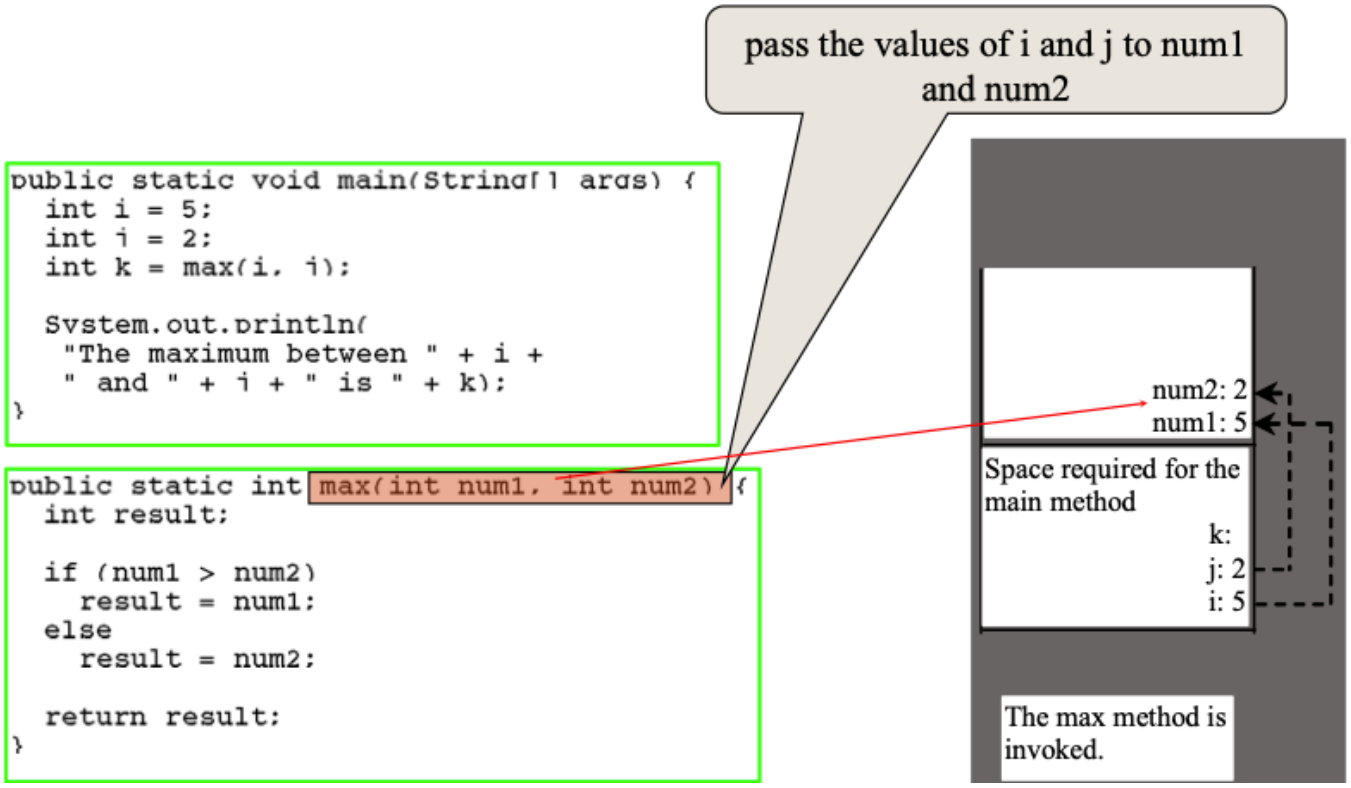
Call stack example



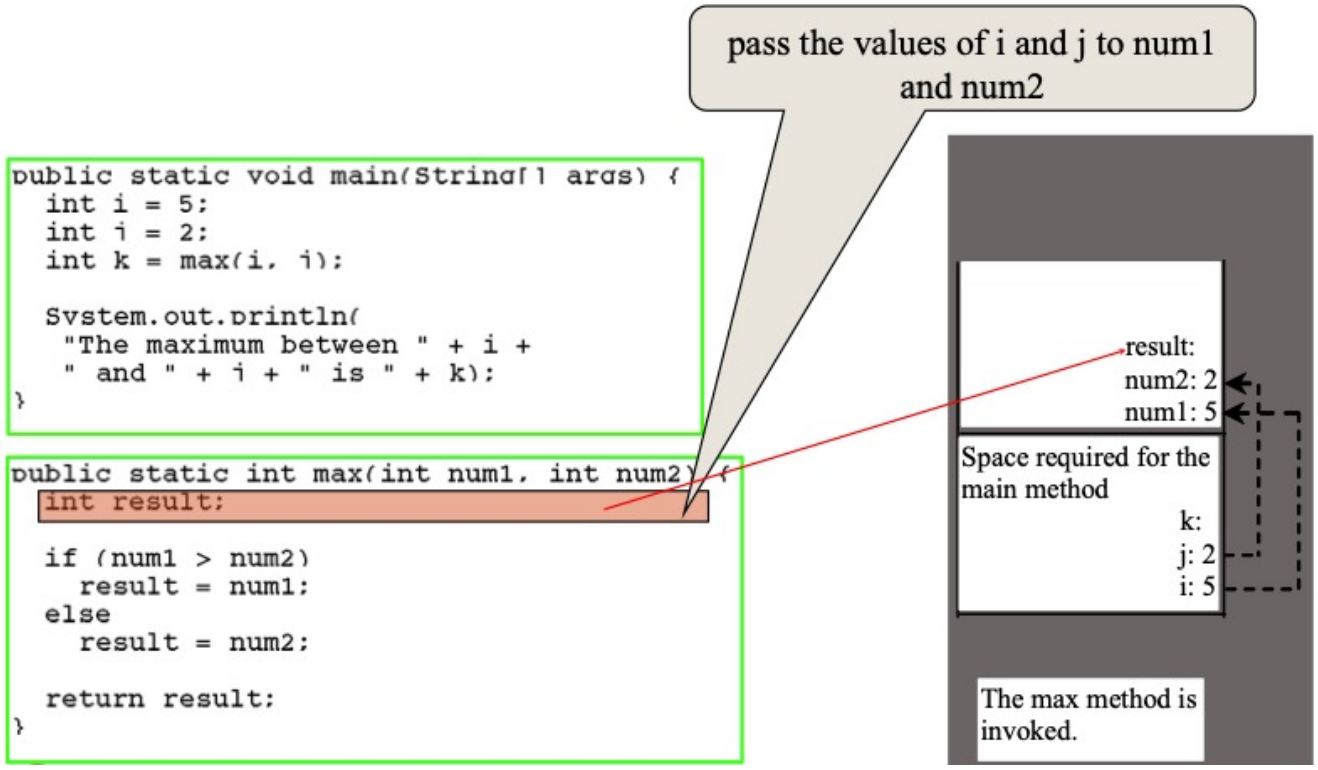
Call stack example



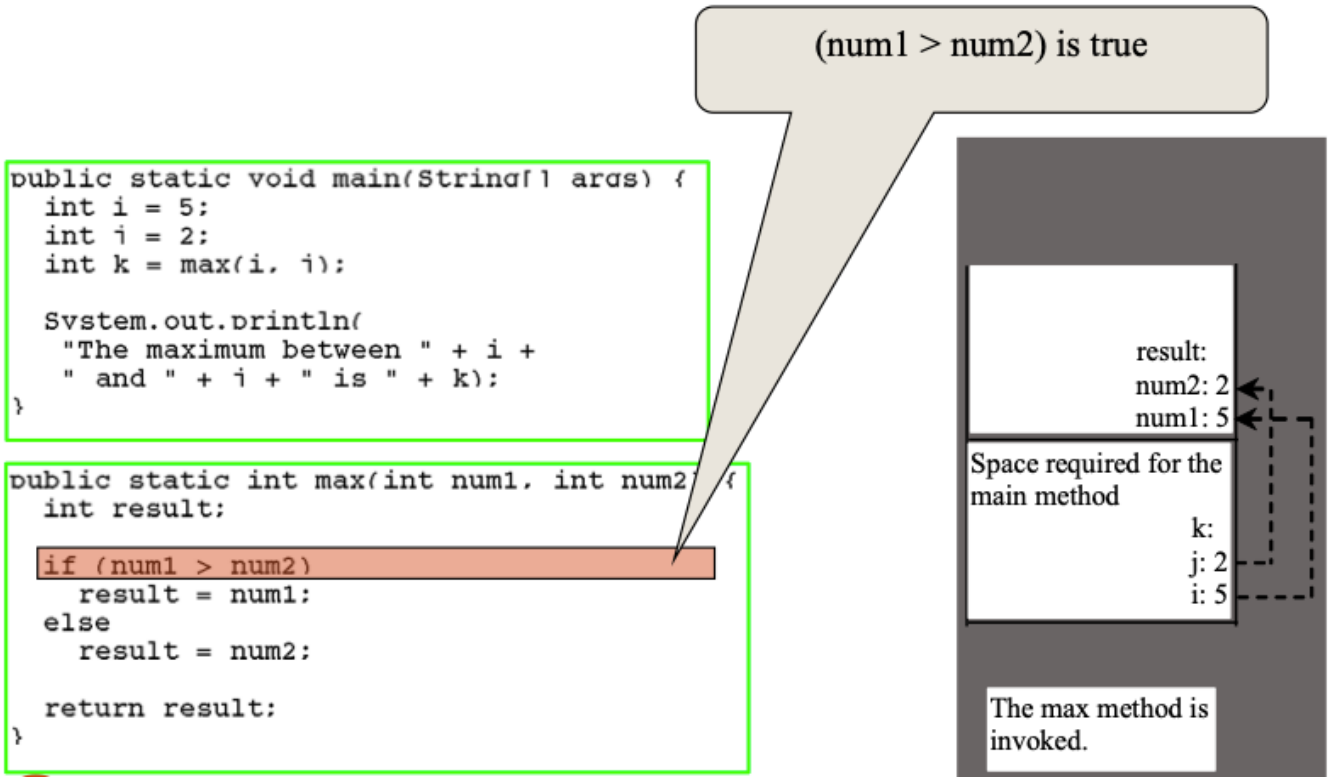
Call stack example



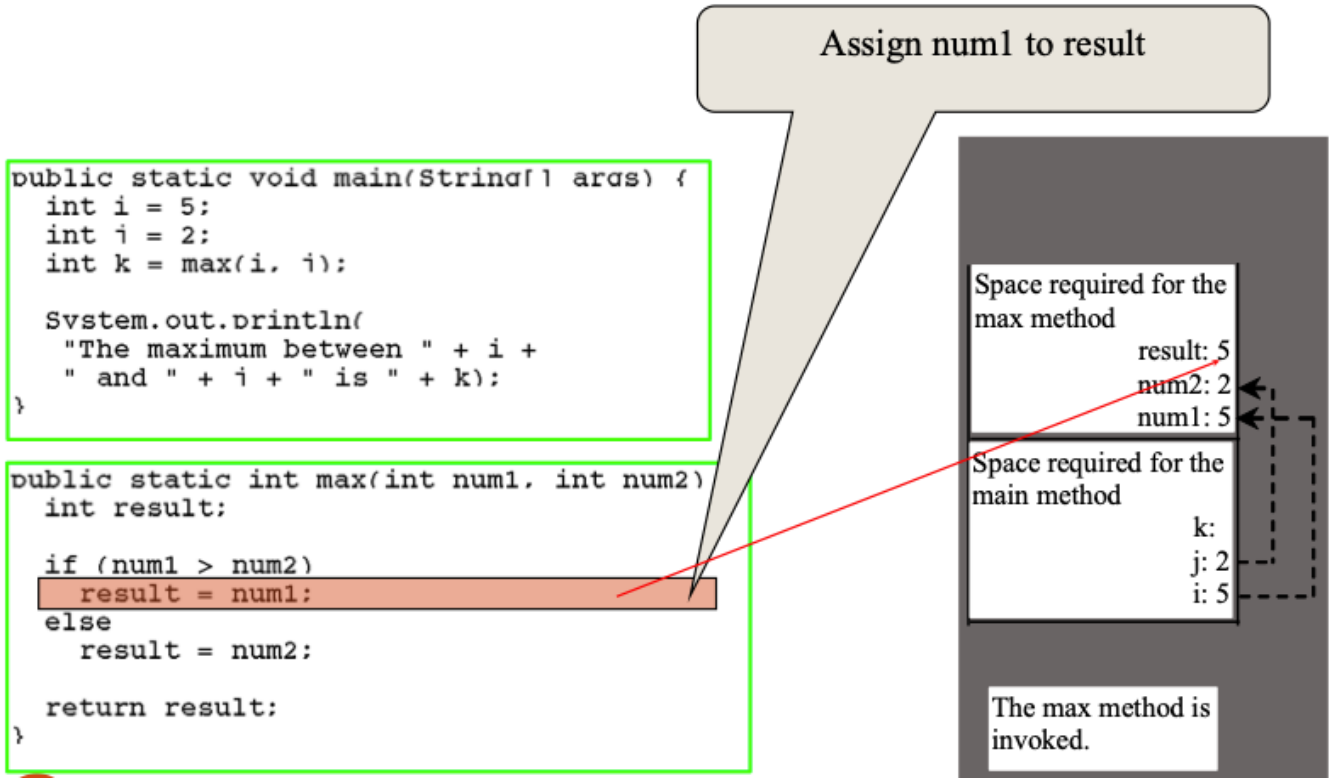
Call stack example



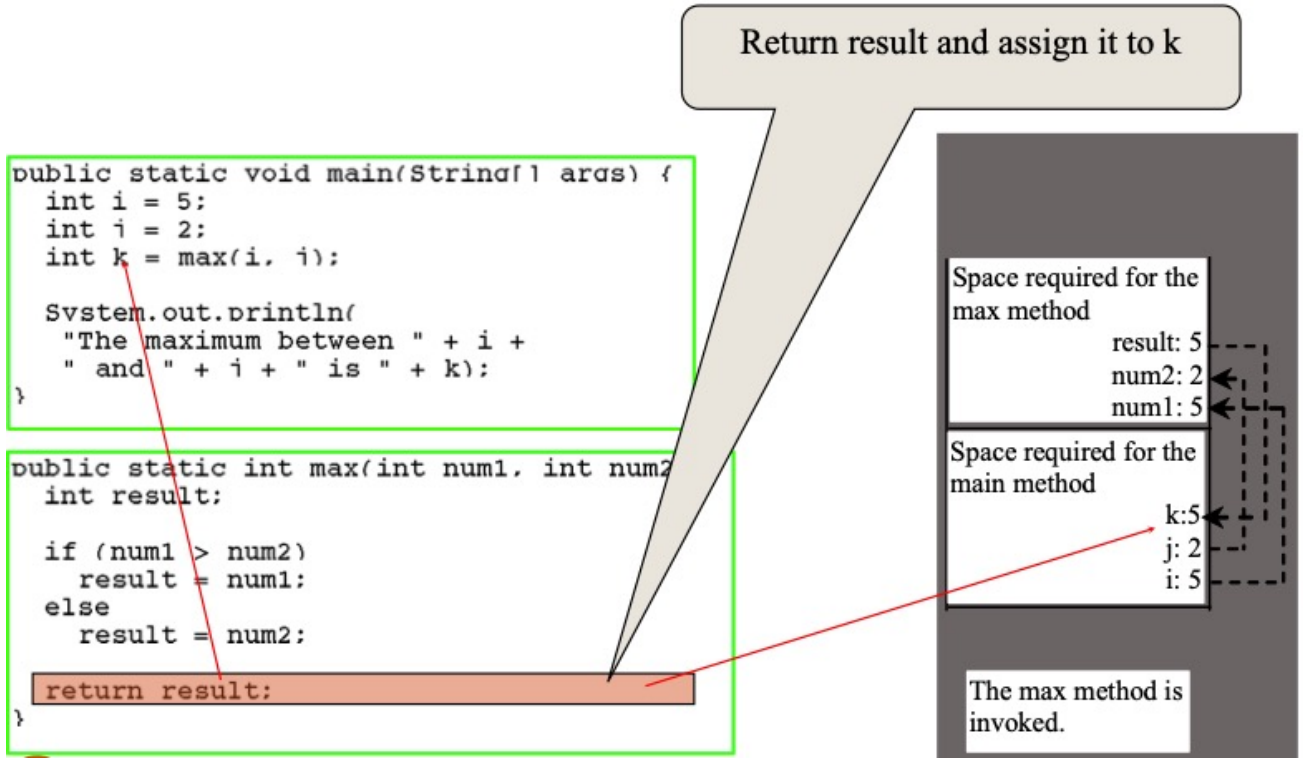
Call stack example



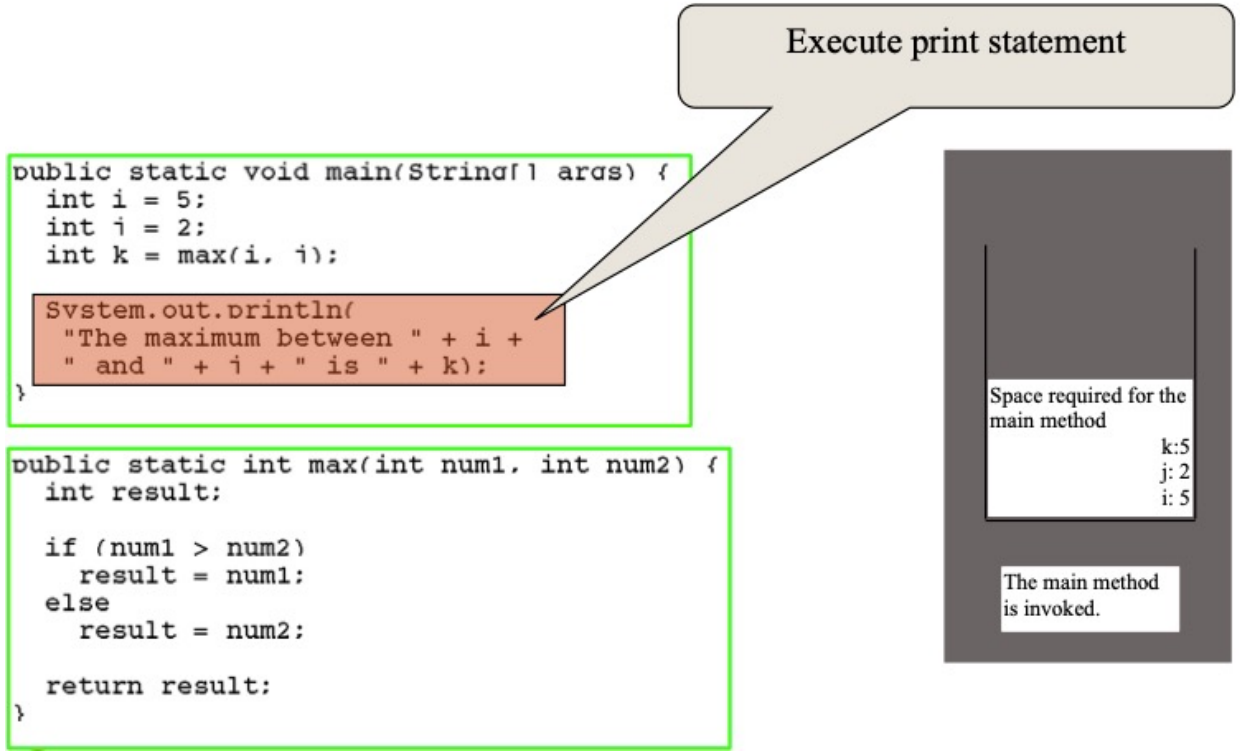
Call stack example



Call stack example



Call stack example



Heap-Based Allocation

- Heap-Based Allocation
  - Heap is for dynamic allocation
  - A heap is a region of storage in which sub-blocks
- Can be allocated and deallocated at arbitrary times
- Dynamically allocated pieces of data structures:
  - objects, Strings, lists, and sets, whose size may
  - change as a result of an assignment statement or
  - other update operation

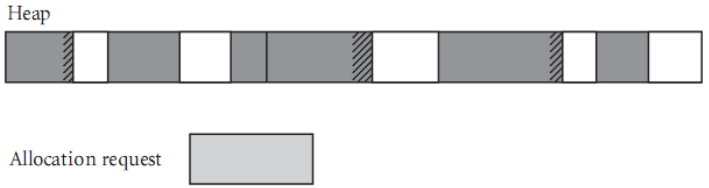
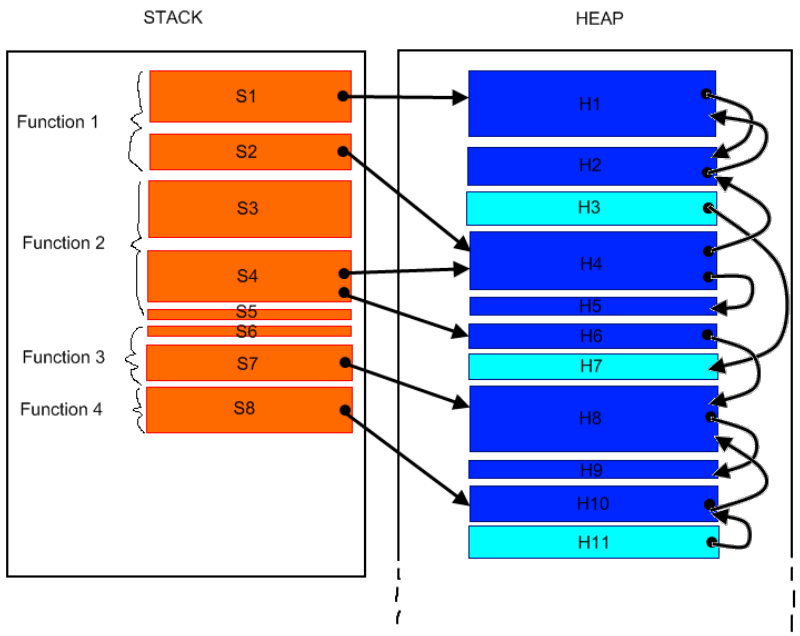
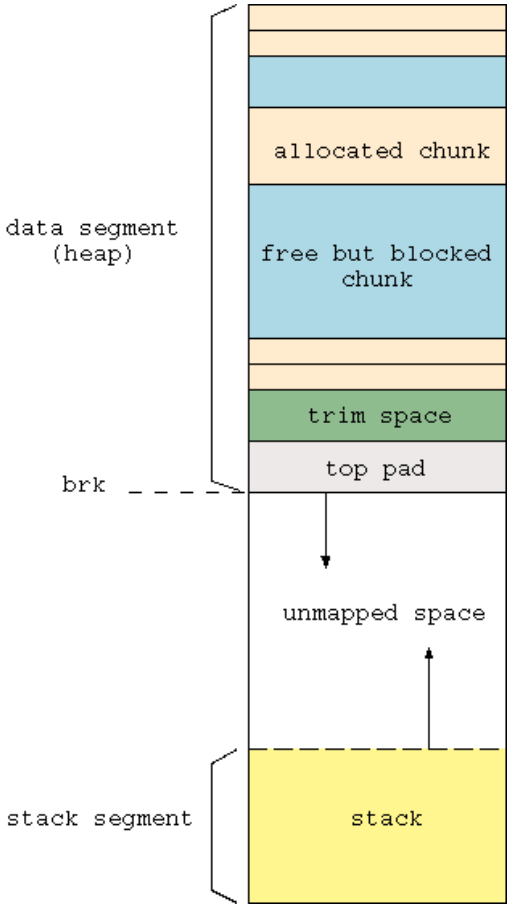


Figure 3.2 **Fragmentation.** The shaded blocks are in use; the clear blocks are free. Cross-hatched space at the ends of in-use blocks represent internal fragmentation. The discontiguous free blocks indicate external fragmentation. While there is more than enough total free space remaining to satisfy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

Heap and stack

- The heap and the stack are closely linked.





Overriding the static scope

- C and other languages allow the coder to override scope, so that your value is not re-initialized each time
- Example:

```
/* Place into s a new name beginning with L and continuing
   with a new ASCII number of each integer.
   Each time this is called, the value is incremented
*/
void label_name (char *s) {
   static short int n;  // C will initialize this to 0
   sprintf(s, "L%d\n", ++n); // “print” output to s
}
```

Accessing variables from another scope

- Sometimes can scope to a different binding.
- Examples
  - In python, "*self*" variable
  - In C++, ::X is a global variable X (regardless of current subroutine)

```
#include<iostream>
using namespace std;

int x; // Global x

int main()
{
int x = 10; // Local x
cout << "Value of global x is " << ::x;
cout << "\nValue of local x is " << x;
return 0;
}
```

Access to non-local variables: STATIC LINKS

- Each frame points to the frame of the 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: STATIC LINKS

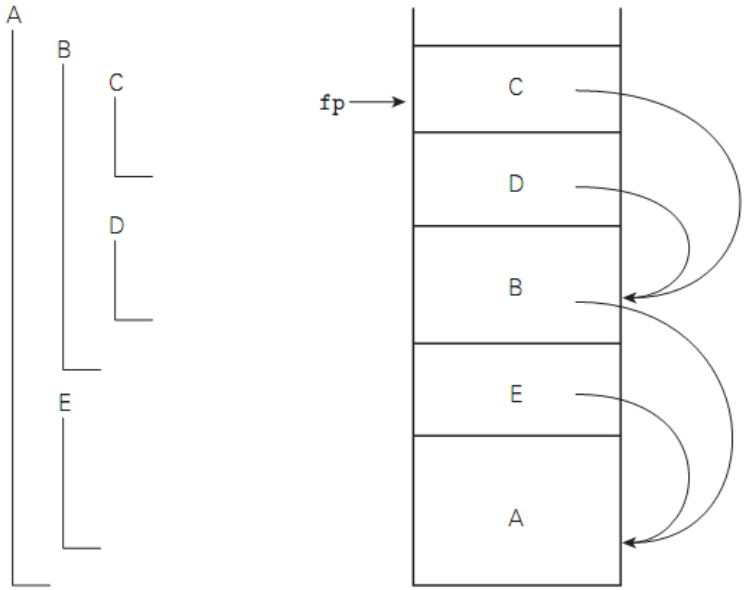


Figure 3.5 **Static chains.** Subroutines A, B, C, D, and E are nested as shown on the left. If the sequence of nested calls at run time is A, E, B, D, and C, then the static links in the stack will look as shown on the right. The code for subroutine C can find local objects at known offsets from the frame pointer. It can find local objects of the surrounding scope, B, by dereferencing its static chain once and then applying an offset. It can find local objects in B's surrounding scope, A, by dereferencing its static chain twice and then applying an offset.

Object Orientation

- Object orientation takes the idea of static scope and extends it even further
- Classes have their own scope with local variables
- Rules are essentially unchanged from the rules for that language, with perhaps a few extensions:
  - Example: In C++, one object can access private variables of another object if they are of the same type.
  - Example: Python goes the other direction: no variables are formally declared, and no variables are private to the class

Classes and Scopes

- In classes, scope is extended even to methods
  - Example: We say mystack.push(5), not push(mystack, 5)
  - The function exists for every stack, and the instance of the stack we are working on is passed as a separate, hidden parameter
  - The "self" variable in python as a perfect example of this, but it is more subtle in other languages

Dynamic scope

- 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

Dynamic Scoping

- Dynamic scope rules: bindings depend on the current state of program execution:
  - They cannot always be resolved by examining the program because they are dependent on calling sequences
- The binding might depend on how a function is called
- To resolve a reference, we use the most recent, active binding made at run time

Example: Dynamic scope

- What get printed?

```
var total = 0
def add():
    total += 1
def myfunc():
    var total = 0
    add()
add()
myfunc()
print total
```

Binding of reference environment

- A reference environment = all the bindings active at a given time.
- When we take a reference to a function, we need to decide which reference environment we want to use
- Deep binding binds the environment at the time the procedure is passed as an argument
- Shallow binding binds the environment at the time the procedure is called

Example

```
var total = 0
def a():
    total += 1
def b(F):
    var total = 0
    F()
    print "B", total
def c():
    var total = 0
    b(a)
    print "C", total
c()
print "T", total
```

	static	dynamic shallow	dynamic deep
B	0	1	0
C	0	0	1
T	1	0	0

Binding of reference environment

```
x: integer := 1
y: integer := 2

procedure add
  x := x + y

procedure second(P:procedure)
  x:integer := 2
  P()

procedure first
  y:integer := 3
  second(add)

first()
write_integer(x)
```

- ***static scoping*** would print 3.

- ***shallow binding*** just traverses up until it finds the nearest variable that corresponds to the name (increments the local x), so the answer would be 1.

- ***dynamic scoping with deep binding***: when add is passed into second the environment of add is x = 1, y = 3 and the x is the global x so it writes 4 into the global x, which is the one picked up by the write_integer.

Closure

- Deep binding is implemented by creating an explicit representation of a referencing environment and bundling it together with a reference to the subroutine (this bundle is called closure).

```
def foo():
    a = 100
    def bar():
        return a
    a += 1
    return bar
f = foo()
print f()
```

Outputs: 101

Implementing closures means we may need to keep around **foo**'s frame even after foo quits.

- heap allocation to keep around objects pointed to by **foo**.
- In languages without closures, we can fake them with objects.

Overloading

- same name, more than one meaning
- some overloading happens in almost all languages
- integer + vs. real + vs. String concatenation
- read and write in Pascal are overloaded based on the number and types of parameters

Polymorphism

- It's worth distinguishing between some closely related concepts:
- overloaded functions - two different things with the same name

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

- polymorphic functions: one thing that works in more than one way
  - Overriding in OO programming, and
  - Generic programming:

```
function min (A : array of Comparable)
```

- generic functions - a syntactic template that can be instantiated in more than one way at compile or even run time via macro processors in C++

Aliasing

- Aliasing: two names point to the same object
  - Makes program hard to understand
  - Makes program slow to compile
  - What are aliases good for? linked data structures
- Example: shallow copy in Python

```
import copy
li1 = [1, 2, [3,5], 4]
li2 = copy.copy(li1)
print ("The original elements before shallow copying")
for i in range(0,len(li1)):
    print (li1[i],end=" ")

print("\r")
li2[2][0] = 7
print ("The original elements after shallow copying")
for i in range(0,len( li1)):
    print (li1[i],end=" ")
```

Macros

- Macros are a way of assigning a name to some syntax.
  - C: Textual substitution.

```
#define MAX(x, y) (x > y ? x : y)
```

- benefit: shorter code, no stack, can choose not to execute some of the code
- Problems with macros:
  - multiple side effects: MAX(a++, b++)
  - scope capture: temporary var used inside macro has same name as a real var
  - for example: t exists outside

```
#define SWAP(a,b) {t = (a); (a) = (b); (b) = t;}
```

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
- 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

References

- Michael L. Scott: Programming Language Pragmatics (Chapter 3)
- Paul Fodor : CSE 307 – Principles of Programming Languages, Stony Brook University