Programming Languages

Names, Scope, and Bindings

Chapter 3

Storage Management

Lifetime and Storage Management

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

Events in the Life of an Object

C++:

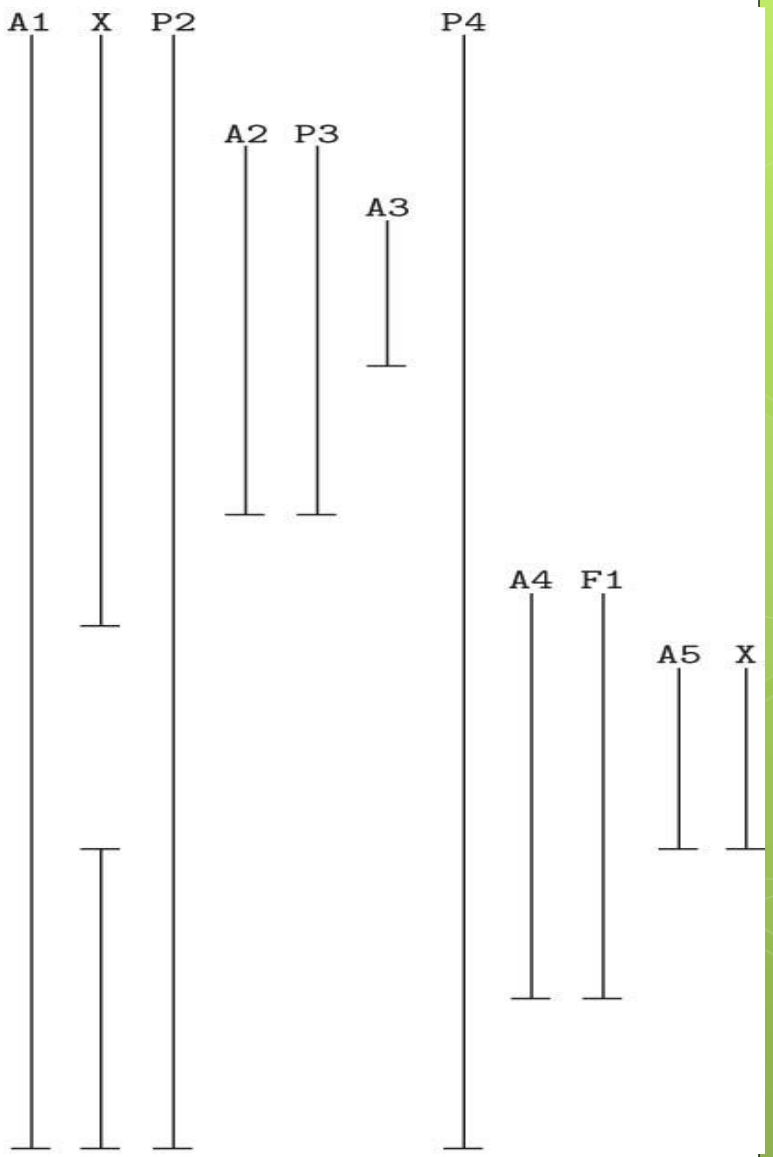
```
void square(int * pNumber) {
        cout << "In square(): " << pNumber
                                << endl;
        *pNumber  *=  *pNumber;
}
```

Call via

```
        int number = 8;
        …
        square(&number);
        …
```

Scope of Variables

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

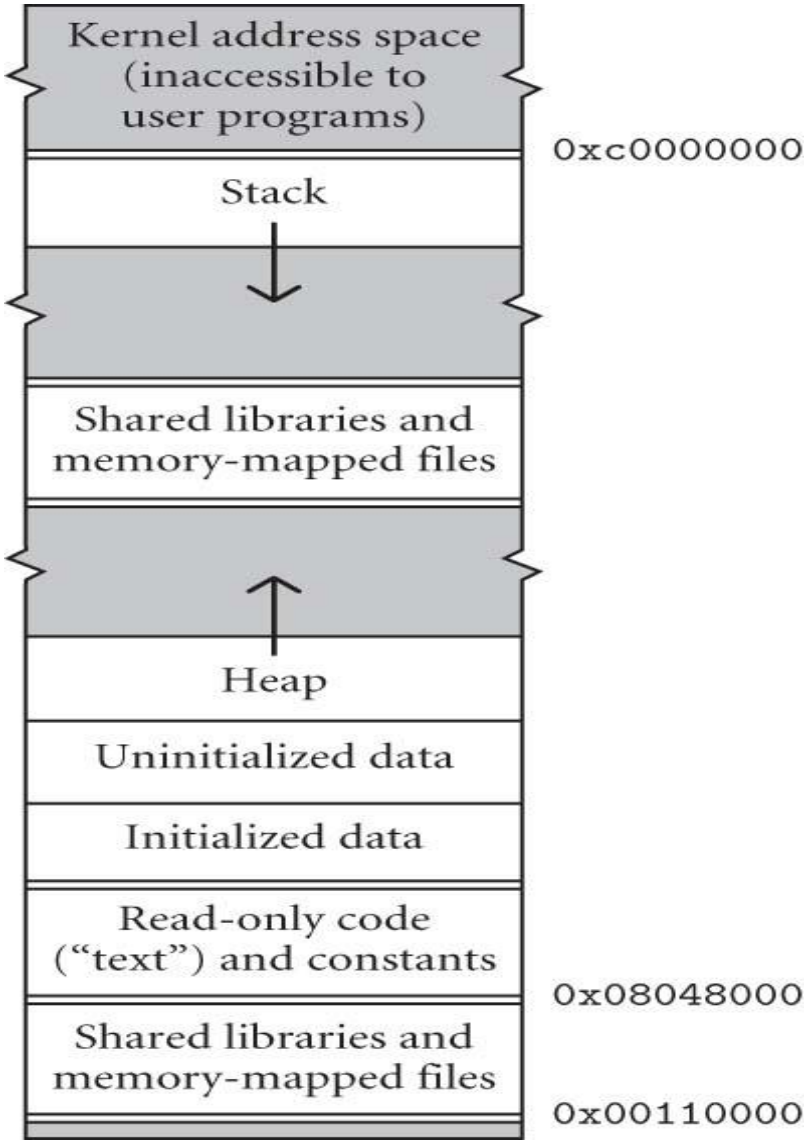


Memory Management

Three primary mechanism for managing the lifetime of an object in memory:

- Static allocation
- Stack allocation
- Heap allocation

Typical Memory Layout



Memory Layout

A typical memory layout shown with 0x00000000 at the bottom and 0xFFFFFFFF at the top

- Read-only (text) segment - contains executable instructions
- Initialized data segment – contains the global variables and static variables that are initialized by the programmer, not read-only
- Uninitialized data segment – also called the “bss” segment, (“block started by symbol”) data in this segment is initialized to 0 before the program starts executing.

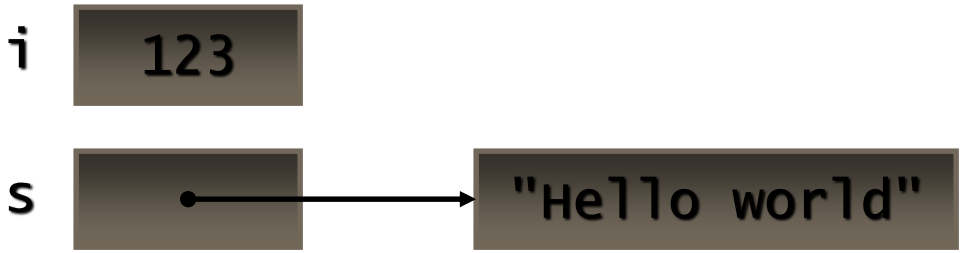
Static Variables

Efficient since variables are given an absolute address that is fixed throughout the program's execution

Types : Value versus References

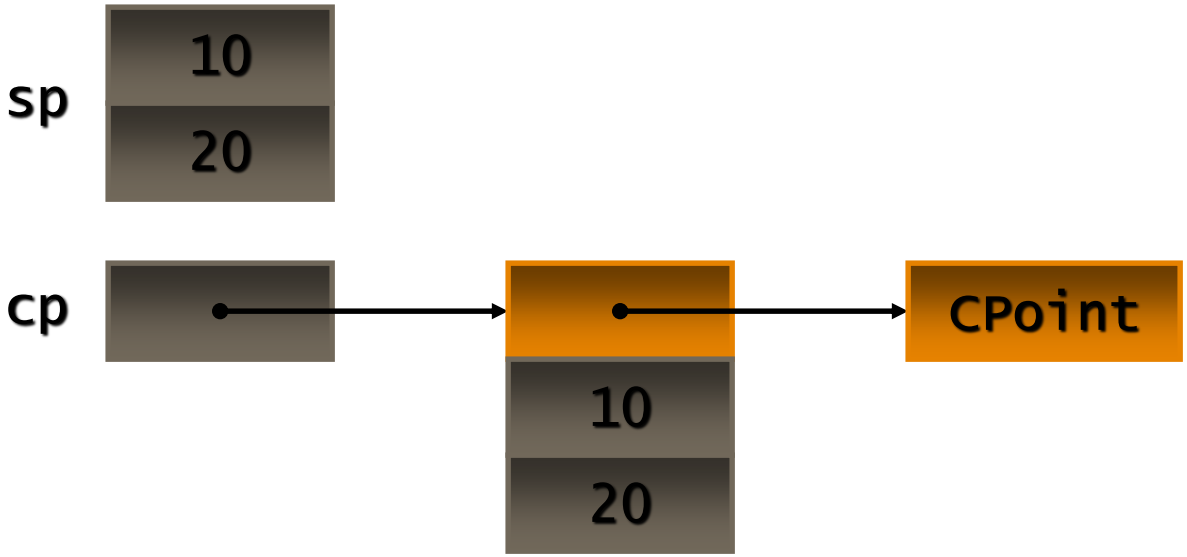
- Value types
  - Directly contain data
  - Cannot be null
- Reference types
  - Contain references to objects
  - May be null

```
int i = 123;
string s = "Hello world";
```



Classes And Structs in C#

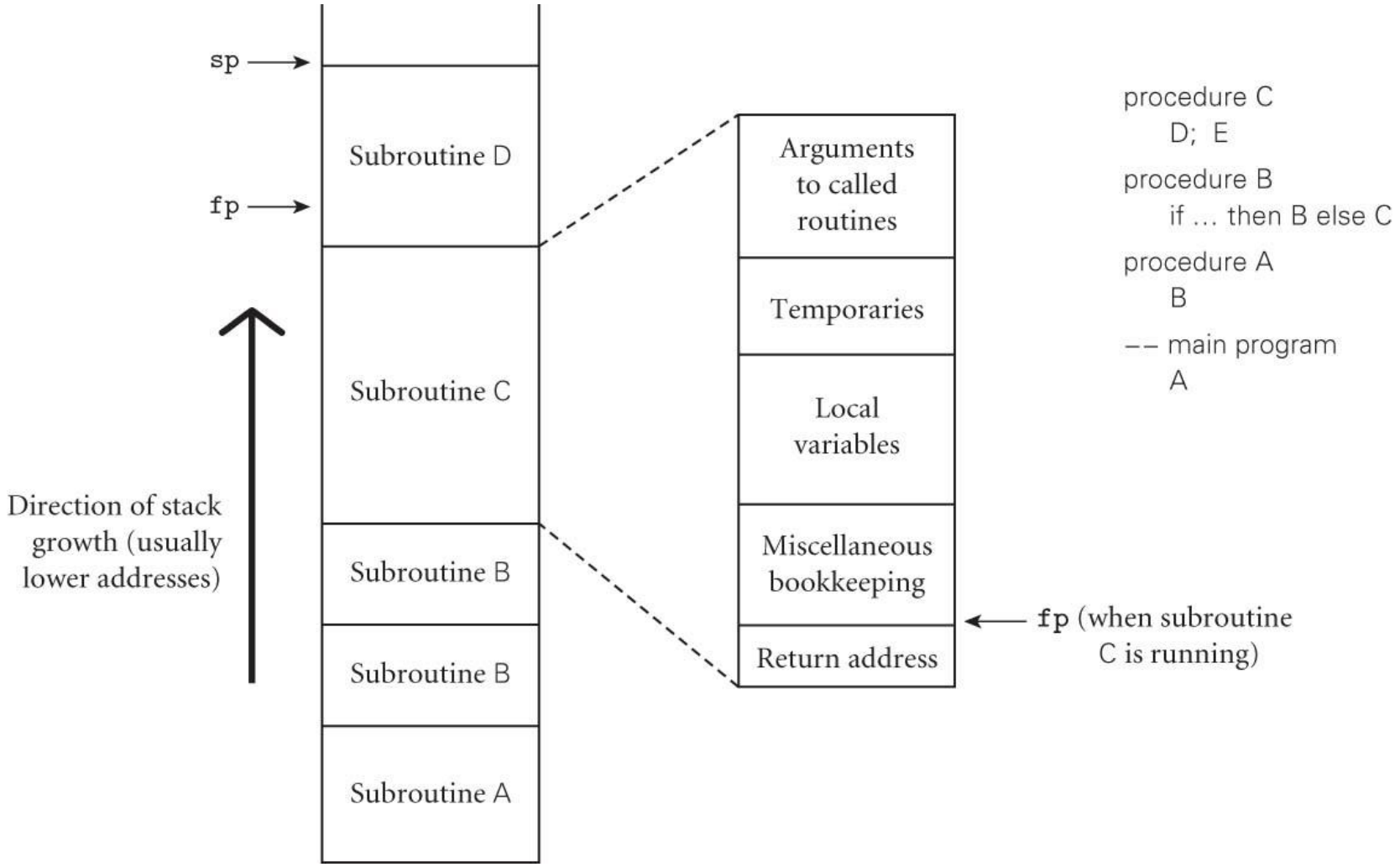
```
class CPoint  { int x, y; ... }
struct SPoint { int x, y; ... }

CPoint cp = new CPoint(10, 20);
SPoint sp = new SPoint(10, 20);
```



Structs in C#

- Like classes, except
  - Stored in-line, not heap allocated
  - Assignment copies data, not reference
  - No inheritance
- Ideal for light weight objects
  - Complex values, points, rectangle, color
  - int, float, double, etc., are all structs
- Benefits
  - No heap allocation, so not garbage collected
  - More efficient use of memory

Stack Frame


sp
fp
Subroutine D
Subroutine C
Subroutine B
Subroutine B
Subroutine A
Direction of stack growth (usually lower addresses)
Arguments to called routines
Temporaries
Local variables
Miscellaneous bookkeeping
Return address
fp (when subroutine C is running)
procedure C
D; E
procedure B
if … then B else C
procedure A
B
-- main program
A

Heap-Dynamic Variables

	C	C++	Java	C#

Declare pointer/ reference variable	struct Employee {
	  char name[30],
	  char address[50];
	};
	
	int *myInt;
		Struct and classes very similar and both typically stored in the heap (struct variables are public, while class variables are private)
		
		1. Employee employee;
		  // Declares reference and
		  // allocates storage on the stack
		
		2. Employee *employee;
		  // storage for pointer is on the stack
		
		3. int *myInt;
		
		4. int *myArray;
			No structs.
			No pointers.
			
			Reference variables.
			
			1. Employee employee;
			
			2. Integer myInt;
			  // Wrapper class
			
			3. int[] myArray;
				Has struct (value type, stored on stack and when copy get a new copy) and class (reference type, stored on heap and when assign to another variable, uses the same copy)
				
				Similar to C++
				
				Can declare pointers in unsafe mode.

Allocate storage	struct Employee
	  employee;
	
	myInt =
	  (int) malloc(
	  sizeof(int)
	  );
		1. Storage already allocated. Reference public methods via:
		  employee.doSomething();
		
		2. employee = new Employee();
		  // Storage for object is on the heap
		Reference public methods via:
		employee->doSomething();
		which is equivalent to:
		(*employee).doSomething();
		
		3. myInt = new int;
		
		4. myArray = new int[10];
		
		Can also use malloc and free
			1. employee = new Employee();
			
			2. myInt = new Integer(5);
			
			3. myArray = new int[10];
				Similar to C++ but no pointers unless in unsafe mode.

De-allocate storage	free(myInt);	1. When object goes out of scope its destructor is called automatically. 2. delete employee; 3. delete myInt; 4. delete [] myIntArray;	Garbage collection (can have a finialize method which closes files, terminates network connections, etc. finalize is called automatically by the garbage collector before it reclaims the space for the object).	Garbage collection or, you could try to speed this up with: employee = null; but this is not recommended. System.GC.Collect();