Principles of Programming Languages

http://www.di.unipi.it/~andrea/Didattica/PLP-14/

Prof. Andrea Corradini
Department of Computer Science, Pisa

Lesson 12

- Concepts of Programming Languages
  - Programming languages and Abstraction
  - Names, bindings and scope

Up to now…

- We have seen:
  - Abstract Machines, compilation and interpretation
  - Structure of compiler
  - Lexical analysis
  - Parsing
  - Syntax-Directed Translation: foundations

➔ Topics based on syntax only

- Following phases of compilation:
  - Intermediate code generation, e.g. control structures
  - Semantic analysis, e.g. type checking
  - Target code generation and optimization

➔ Topics requiring knowledge of programming language concepts / semantics

- We will see them in future lectures…

Summary

- PL's as defining Abstract Machines
- Abstraction mechanisms
- Names and abstractions
- Binding time
- Object lifetime
- Object storage management
  - Static allocation
  - Stack allocation
  - Heap allocation

Definition of Programming Languages

- A PL is defined via **syntax**, **semantics** and **pragmatics**
- The **syntax** is concerned with the form of programs: how expressions, commands, declarations, and other constructs must be arranged to make a well-formed program.
- The **semantics** is concerned with the meaning of (well-formed) programs: how a program may be expected to behave when executed on a computer.
- The **pragmatics** is concerned with the way in which the PL is intended to be used in practice. Pragmatics include the *paradigm(s)* supported by the PL.

Paradigms

A **paradigm** is a style of programming, characterized by a particular selection of key concepts

- **Imperative programming**: variables, commands, procedures.
- **Object-oriented (OO) programming**: objects, methods, classes.
- **Concurrent programming**: processes, communication.
- **Functional programming**: values, expressions, functions.
- **Logic programming**: assertions, relations.

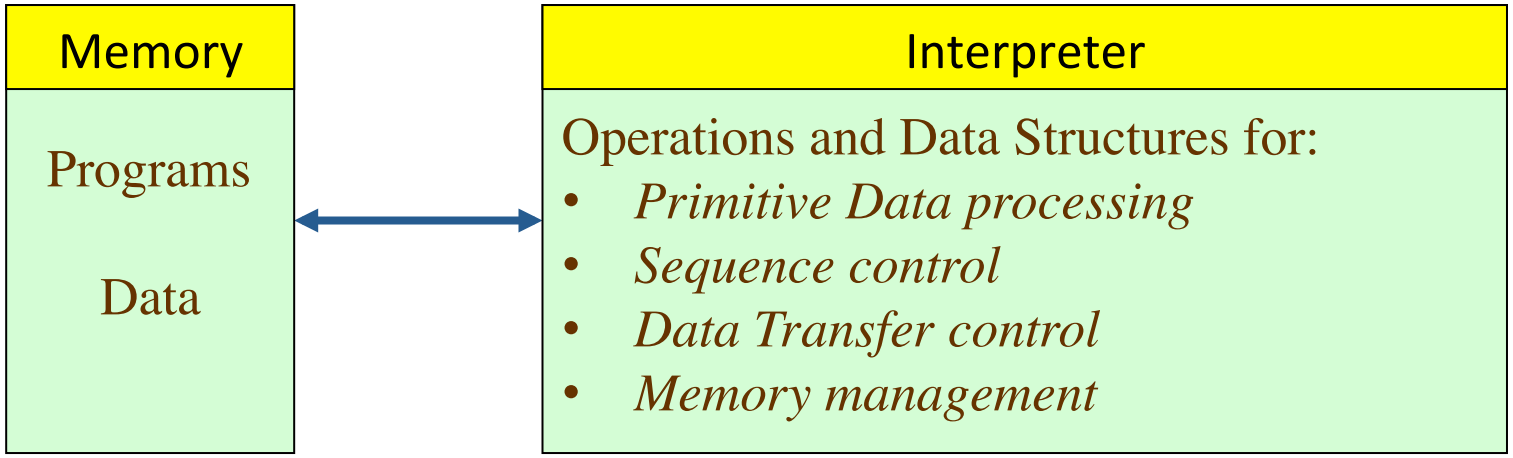
Classification of languages according to paradigms is misleading

Implementation of a Programming Language *L*

- Programs written in ***L*** must be executable
- Language ***L*** implicitly defines an Abstract Machine M_L having ***L*** as machine language
- Implementing M_L on an existing host machine M_O (via compilation or interpretation or both) makes programs written in ***L*** executable

(Recap) Abstract Machine for a Language **L**

- Given a programming language **L**, an **Abstract Machine M_L for L** is *a collection of data structures and algorithms which can perform the storage and execution of programs written in **L***
- Structure of an abstract machine:



Programming Languages and Abstraction

- PL's are born to abstract from machine languages
- Each PL generation abstracts more
- Modern languages include abstraction mechanisms to be used by programmers
- The study of PL's focuses on the study of abstractions, abstraction mechanisms, and how they can be implemented (efficiently…)

Abstraction mechanisms

Abstraction Mechanisms are independent of the concept to be abstracted

- Naming
  - Associate a name with a possibly complex entity
- Abstraction by Parametrization
  - Abstracts from identity of data by using parameters
    - Procedures with parameter
    - Generic types, …
- Abstraction by Specification
  - Abstracts from implementation details
    - Interfaces
    - Function prototypes, …

PL's defining Abstract Machines

The definition of a PL mainly consists of defining

- the Abstract Machine components, and
- the abstraction mechanism **[abs]** to extend them:
  - **Primitive Data processing [DP]**
    - Data types and operations
    - Procedures, …
  - **Sequence control [SC]**
    - Control structures, …
  - **Data Transfer control [DTC]**
    - Parameter passing mechanisms
    - Scoping rules, …
  - **Memory management [MM]**
    - Static / stack / heap allocation mechanisms, …

Programming Language Concepts

- The proposed view allows us to relate the typical concepts of Programming Languages in a unifying framework
  - Names, bindings and scope **[abs]**
  - Values and data types **[DP]**
  - Variables and storage management **[MM]**
  - Control abstraction **[abs, SC]**
  - Data abstraction **[abs, DP]**
  - Generic abstraction **[abs, DP]**
  - Concurrency **[SC]**
- We will discuss them in generality, making reference to concrete examples when useful

Names and abstraction

- Used by programmers (but also by language designers) to refer to variables, constants, operations, types, …
- Names are fundamental for abstraction mechanisms
- Their use applies to all components of abstract machines
  - Control abstraction:
    - Subroutines (procedures and functions) allow programmers to focus on manageable subset of program text, hiding implementation details
    - Control flow constructs (if-then, while, for, return) hide low-level machine ops
  - Data abstraction:
    - Object-oriented classes hide data representation details behind a set of operations

Binding Time

- A **binding** is an association between a **name** and an **entity**
- An entity that can have an associated name is called **denotable**
- **Binding time** is the time at which a ***decision is made*** to create a name ↔ entity binding (the actual binding can be created later):
  - **Language design time**: the design of specific program constructs (syntax), primitive types, and meaning (semantics)
  - **Language implementation time**: fixation of implementation constants such as numeric precision, run-time memory sizes, max identifier name length, number and types of built-in exceptions, etc. (if not fixed by the language specification)

Binding Time (2)

- **Program writing time**: the programmer's choice of algorithms and data structures
- **Compile time**: the time of translation of high-level constructs to machine code and choice of memory layout for data objects
- **Link time**: the time at which multiple object codes (machine code files) and libraries are combined into one executable (e.g. external names are bound)
- **Load time**: when the operating system loads the executable in memory (e.g. physical addresses of static data)
- **Run time**: when a program executes

Binding Time Examples

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

Binding Time Examples (cont'd)

- Language implementation
  - Internal representation of types and literals (type ↔ byte encoding, if not specified by language)
    - 3.1 (IEEE 754) and "foo bar" (\0 terminated or embedded string length)
  - Storage allocation method for variables (static/stack/heap)
- Compile time
  - The specific type of a variable in a declaration (name↔type)
  - Storage allocation mechanism for a global or local variable (name↔allocation mechanism)

Binding Time Examples (cont'd)

- Linker
  - Linking calls to static library routines (function↔address)
    - **`printf`** (in libc)
  - Merging and linking multiple object codes into one executable
- Loader
  - Loading executable in memory and adjusting absolute addresses
    - Mostly in older systems that do not have virtual memory
- Run time
  - Dynamic linking of libraries (library function↔library code)
    - DLL, dylib
  - Nonstatic allocation of space for variable (variable↔address)
    - Stack and heap

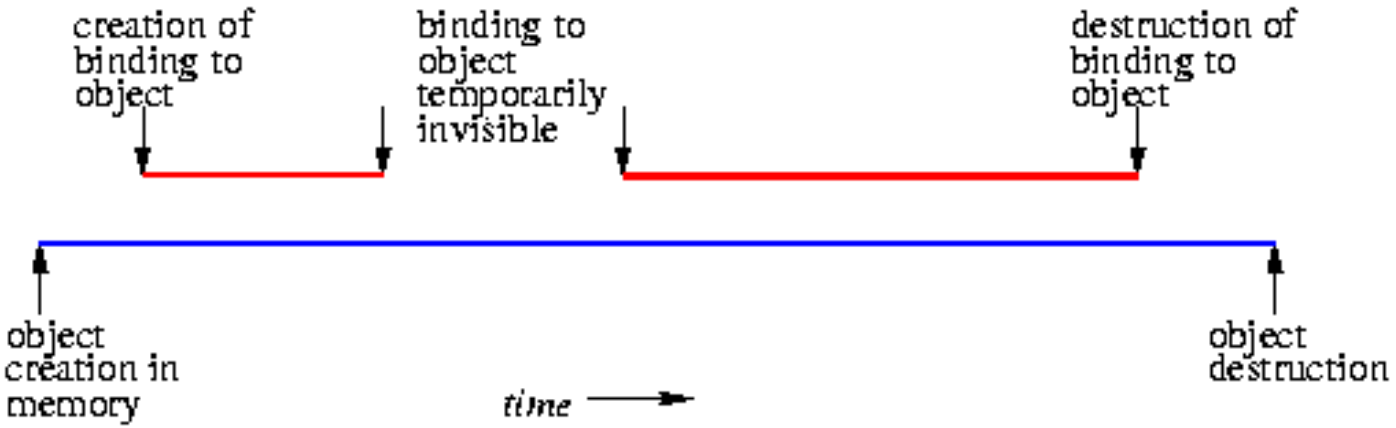
The Effect of Binding Time

- ***Early binding times*** (before run time) are associated with greater efficiency and clarity of program code
  - Compilers make implementation decisions at compile time (avoiding to generate code that makes the decision at run time)
  - Syntax and static semantics checking is performed only once at compile time and does not impose any run-time overheads
- ***Late binding times*** (at run time) are associated with greater flexibility (but may leave programmers sometimes guessing what's going on)
  - Interpreters allow programs to be extended at run time
  - Languages such as Smalltalk-80 with polymorphic types allow variable names to refer to objects of multiple types at run time
  - Method binding in object-oriented languages must be late to support **dynamic binding**
- Usually "**static**" means "before runtime", **dynamic** "at runtime"

Binding Lifetime versus Object Lifetime

- Key events in object lifetime:
  - Object creation
  - Creation of bindings
  - The object is manipulated via its binding
  - Deactivation and reactivation of (temporarily invisible) bindings
  - Destruction of bindings
  - Destruction of objects
- **Binding lifetime**: time between creation and destruction of binding to object
  - Example: a pointer variable is set to the address of an object
  - Example: a formal argument is bound to an actual argument
- **Object lifetime:** time between creation and destruction of an object

Binding Lifetime versus Object Lifetime (cont'd)

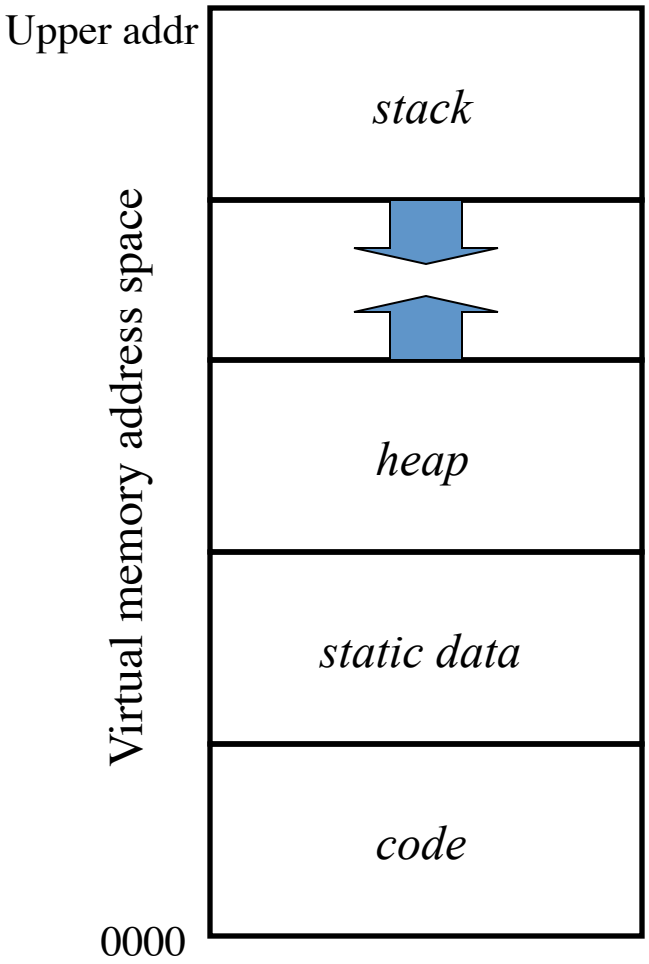


- Bindings are temporarily invisible when code is executed where the binding (name ↔ object) is out of scope

Object Storage

- Objects (program data and code) have to be stored in memory during their lifetime
- **Static objects** have an absolute storage address that is retained throughout the execution of the program
  - Global variables and data
  - Subroutine code and class method code
- **Stack objects** are allocated in last-in first-out order, usually in conjunction with subroutine calls and returns
  - Actual arguments passed by value to a subroutine
  - Local variables of a subroutine
- **Heap objects** may be allocated and deallocated at arbitrary times, but require a storage management algorithm
  - Example: Lisp lists
  - Example: Java class instances are always stored on the heap

Typical Program and Data Layout in Memory

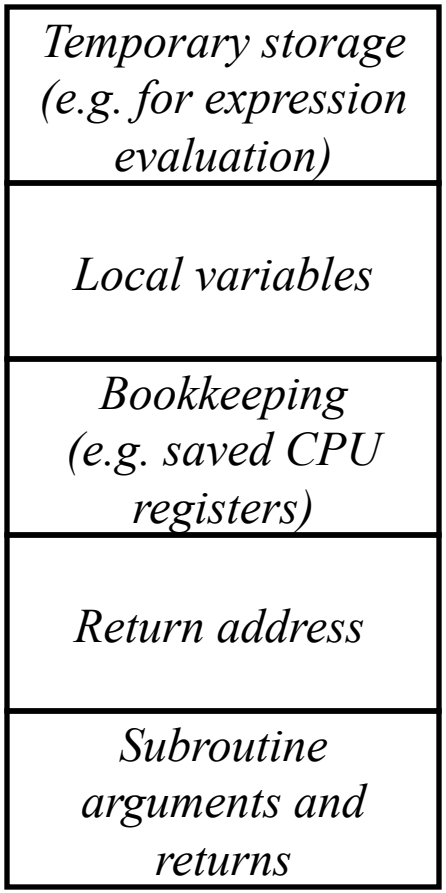


- Program code is at the bottom of the memory region (code section)
  - The code section is protected from run-time modification by the OS
- Static data objects are stored in the static region
- Stack grows downward
- Heap grows upward

Static Allocation

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

Static Allocation in Fortran 77



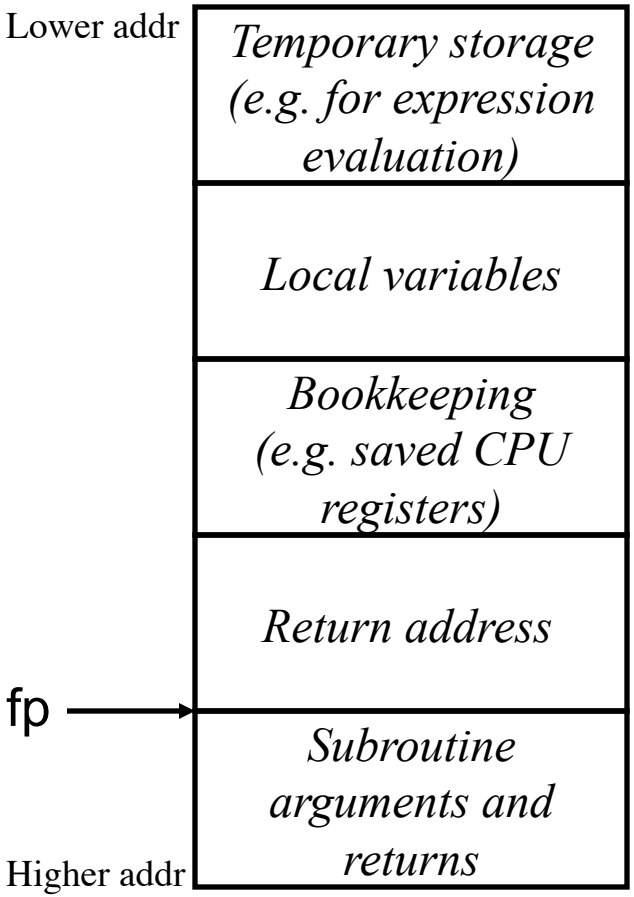
Typical static subroutine frame layout

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

Stack Allocation

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

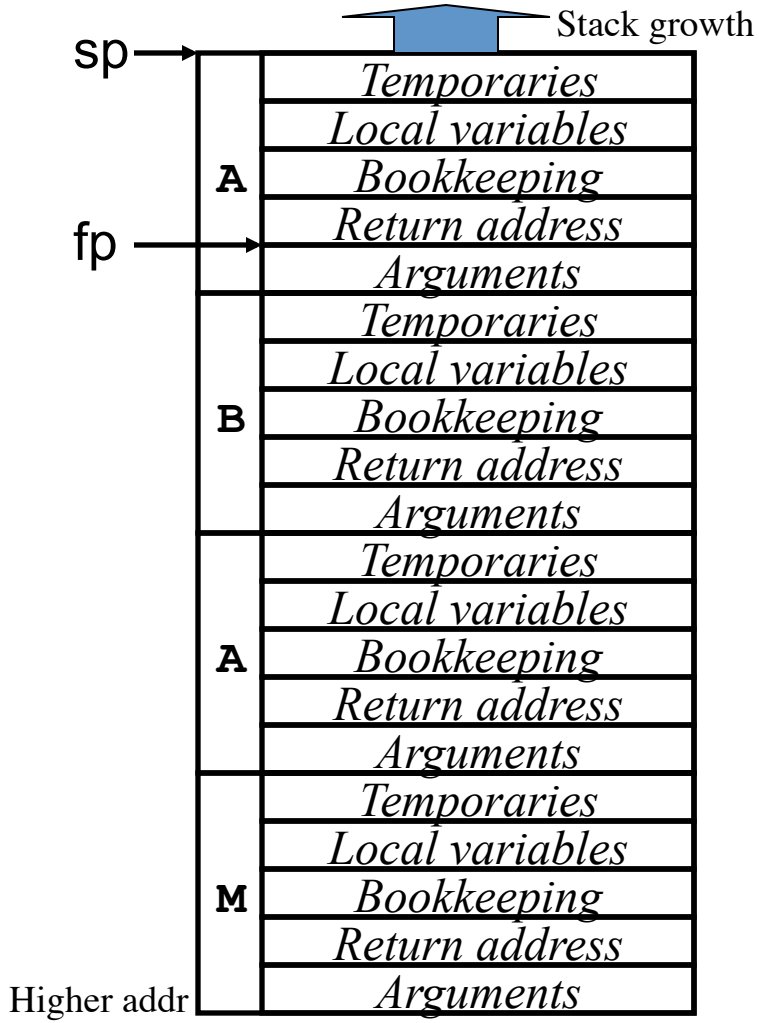
Typical Stack-Allocated Activation Record



Typical subroutine frame layout

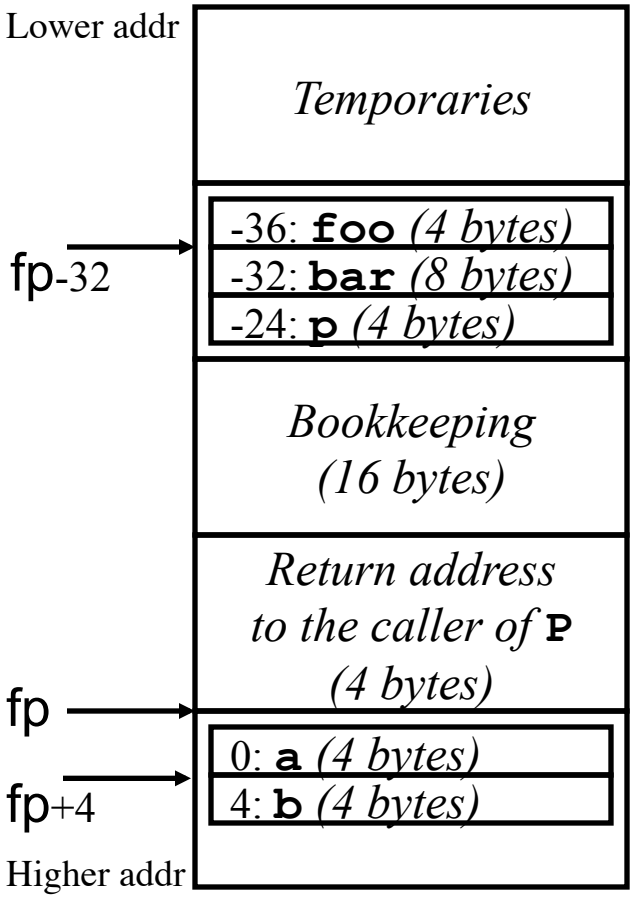
- A *frame pointer* (**fp**) points to the frame of the currently active subroutine at run time
- Subroutine arguments, local variables, and return values are accessed by constant address offsets from the **fp**

Activation Records on the Stack



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

Example Activation Record



- The size of the types of local variables and arguments determines the **fp** offset in a frame
- Example Pascal procedure:

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

Heap Allocation

- **Implicit heap allocation**:
  - Done automatically
  - Java class instances are placed on the heap
  - Scripting languages and functional languages make extensive use of the heap for storing objects
  - Some procedural languages allow array declarations with run-time dependent array size
  - Resizable character strings
- **Explicit heap allocation**:
  - Statements and/or functions for allocation and deallocation
  - Malloc/free, new/delete

Heap Allocation Algorithms

- Heap allocation is performed by searching the heap for available free space
- Deletion of objects leaves free blocks in the heap that can be reused
- *Internal heap fragmentation*: if allocated object is smaller than the free block the extra space is wasted
- *External heap fragmentation*: smaller free blocks cannot always be reused resulting in wasted space

Heap Allocation Algorithms (cont'd)

- Maintain a linked list of free heap blocks
- **First-fit**: select the first block in the list that is large enough
- **Best-fit**: search the entire list for the smallest free block that is large enough to hold the object
- If an object is smaller than the block, the extra space can be added to the list of free blocks
- When a block is freed, adjacent free blocks are merged
- **Buddy system**: use heap pools of standard sized blocks of size 2^k
  - If no free block is available for object of size between $2^{k-1}+1$ and 2^k then find block of size 2^{k+1} and split it in half, adding the halves to the pool of free 2^k blocks, etc.
- **Fibonacci heap**: use heap pools of standard size blocks according to Fibonacci numbers
  - More complex but leads to slower internal fragmentation