LECTURE 13

Names, Scopes, and Bindings:
Memory Management Schemes

REVIEW

Last lecture, we introduced the concept of *names*.

- Names enable programmers to refer to variables, constants, operations, and types using identifier names rather than low-level hardware addresses.
- Names are also control and data abstractions of complicated program fragments and data structures.

Names must be associated with a number of properties. For example,

- The type of a variable.
- The memory location of a compiled function.
- The value stored in a variable.

REVIEW

The act of associating a name with its properties is known as *binding*. The *binding time* is the moment when a name becomes bound to one or more of its properties. There are a number of possible binding times.

- **Language design time**: when the language is first created.
  - Examples: meaning of keywords such as while or for in C, or the size of the integer data type in Java, the number of primitive types in C++.
- **Language implementation time**: when the compiler is implemented.
  - Examples: the size of integers in C.

REVIEW

- **Compilation time.**
  - Examples: the types of variables in C, the location of statically allocated variables, and the layout of the activation records of functions.
- **Link time.**
  - Example: the address of externally-defined functions such as printf().
- **Load time.**
  - Example: the absolute address of the program in memory prior to execution.
- **Run time.**
  - Example: actual values stored in variables; in the case of dynamically-typed languages, types of variables.

REVIEW

What are the names used here?

```
int i, x = 0;
void main() {
  for (i = 1; i <= 50; i++)
    x += do_something(x);
}
```

REVIEW

What are the names used here?

- int
- i
- x
- =
- void
- main
- for
- do_something
- etc…

```
int i, x = 0;
void main() {
  for (i = 1; i <= 50; i++)
    x += do_something(x);
}
```

REVIEW

The design of the C language defines the meaning of `int`, but the implementation of the compiler will decide the range of `int`.

```
int i, x = 0;
void main() {
  for (i = 1; i <= 50; i++)
    x += do_something(x);
}
```

REVIEW

The types of `i` and `x` are bound at compile time, but their values are bound at run time.

```
int i, x = 0;
void main() {
  for (i = 1; i <= 50; i++)
    x += do_something(x);
}
```

REVIEW

If `do_something` is defined in another file, its implementation is bound at link time.

```
int i, x = 0;
void main() {
  for (i = 1; i <= 50; i++)
    x += do_something(x);
}
```

REVIEW

The addresses of `main`, `i`, `x`, and `do_something` are bound at load time.

```
int i, x = 0;
void main() {
  for (i = 1; i <= 50; i++)
    x += do_something(x);
}
```

REVIEW

Last time, we also identified the key events in an *object's lifetime* and a *binding's lifetime*.

- Object creation.
- Creation of bindings.
- The object is manipulated via its binding.
- Deactivation and reactivation of (temporarily invisible) bindings. (in-and-out of scope).
- Destruction of bindings.
- Destruction of object.

STORAGE MANAGEMENT

Obviously, objects need to be stored somewhere during the execution of the program. The lifetime of the object, however, generally decides the storage mechanism used. We can divide them up into three categories.

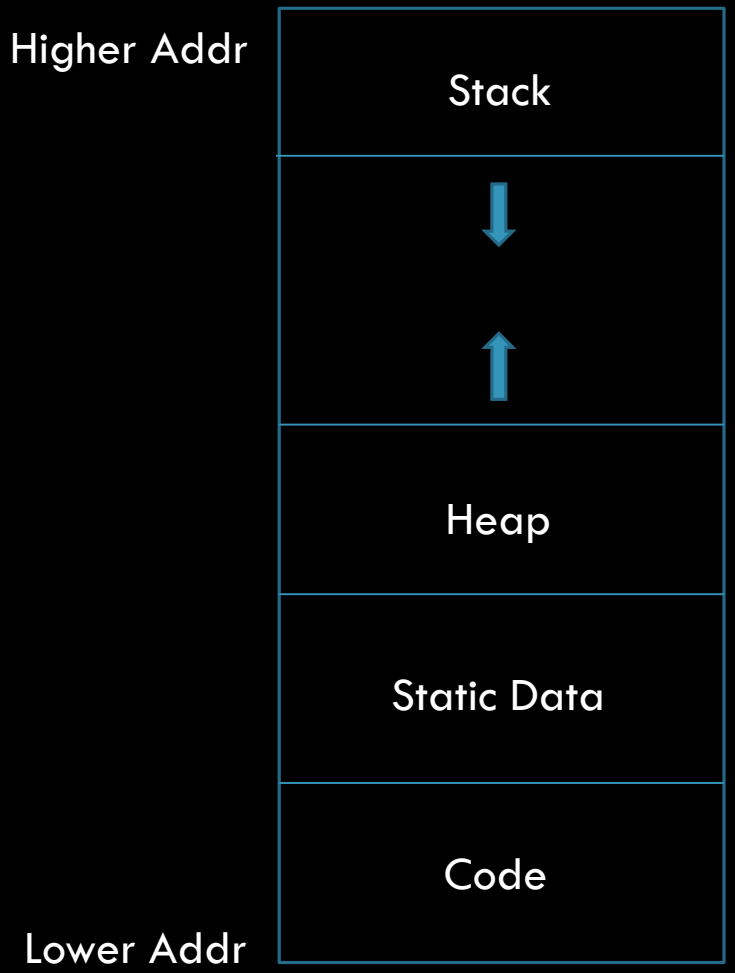
- The objects that are alive throughout the execution of a program (e.g. global variables).
- The objects that are alive within a routine (e.g. local variables).
- The objects whose lifetime can be dynamically changed (the objects that are managed by the ‘new/delete’ constructs).

STORAGE MANAGEMENT

The three types of objects correspond to three principal storage allocation mechanisms.

- *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 an expensive storage management algorithm.
  - Dynamically allocated data in C++.
  - Java class instances are always stored on the heap.

TYPICAL PROGRAM/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/C++ functions keep state across function invocations.
  - *Static data members* are "shared" by objects and keep state during program lifetime.
- Advantage of statically allocated objects is the fast access due to absolute addressing of the object.
  - Can static allocation be used for local variables?

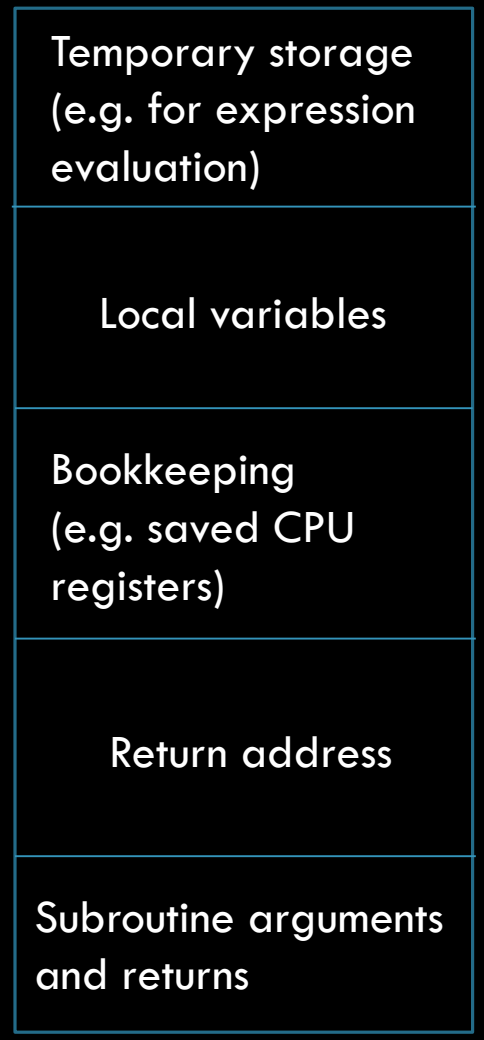
STATIC ALLOCATION

- 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/C++ functions keep state across function invocations.
  - *Static data members* are "shared" by objects and keep state during program lifetime.
- Advantage of statically allocated objects is the fast access due to absolute addressing of the object.
  - Can static allocation be used for local variables?
  - No, statically allocated local variables have only one copy of each variable. Cannot deal with the cases when multiple copies of a local variable are alive!
  - When does this happen?

STATIC ALLOCATION IN FORTRAN 77

Fortran 77 has no recursion.

- Global and local variables are statically allocated as decided by the compiler.
- Global and local variables are references to 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.



Typical static subroutine frame layout

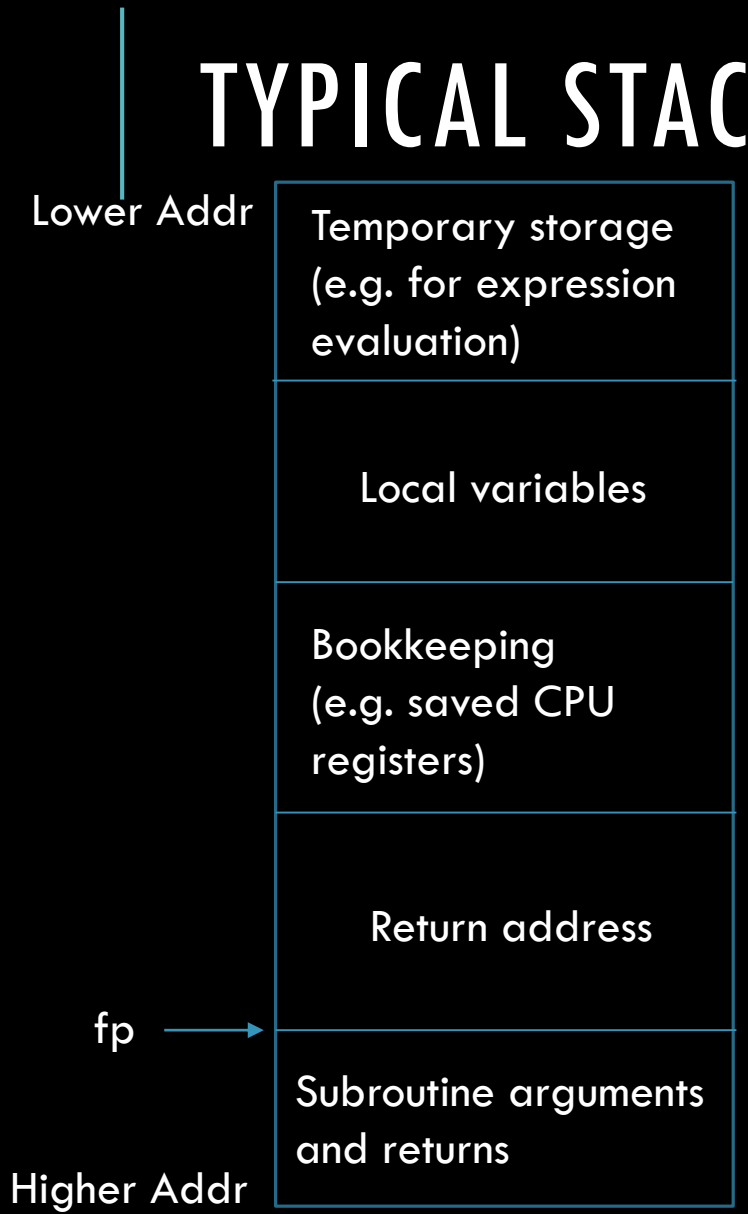
STACK ALLOCATION

Each instance of a subroutine that is active has a subroutine frame (sometimes called activation record) on the run-time stack.

- Compiler generates subroutine calling sequence to setup frame, call the routine, and to destroy the frame afterwards.

Subroutine frame layouts vary between languages, implementations, and machine platforms.

TYPICAL STACK-ALLOCATED SUBROUTINE FRAME



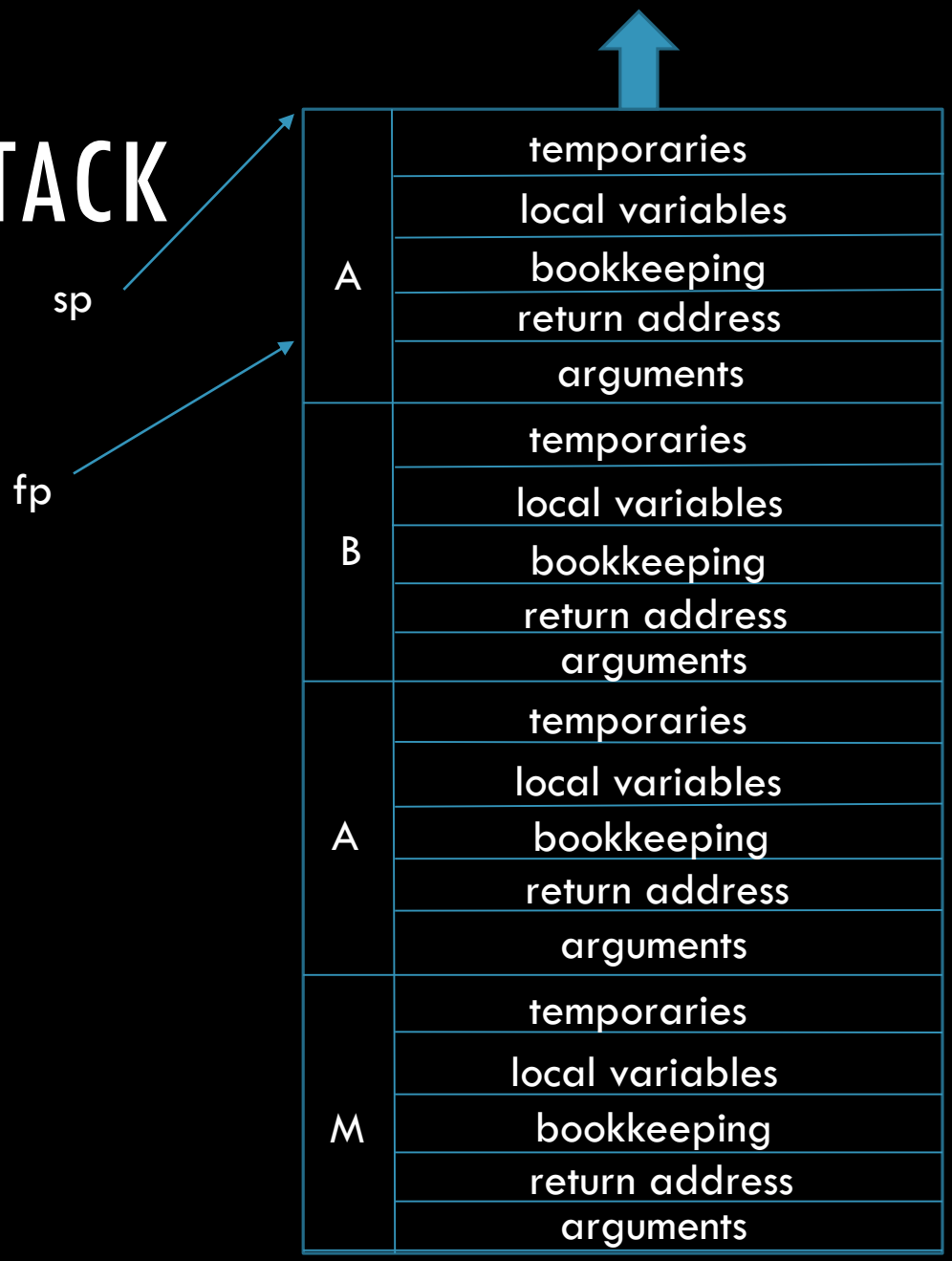
Typical subroutine frame layout

- Most modern processors have two registers: `fp` (frame pointer) and `sp` (stack pointer) to support efficient execution of subroutines in high level languages.
- 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.

SUBROUTINE FRAMES ON THE STACK

Subroutine frames 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 frame onto when a subroutine is called.
- The frame pointer (`fp`) points to the frame 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

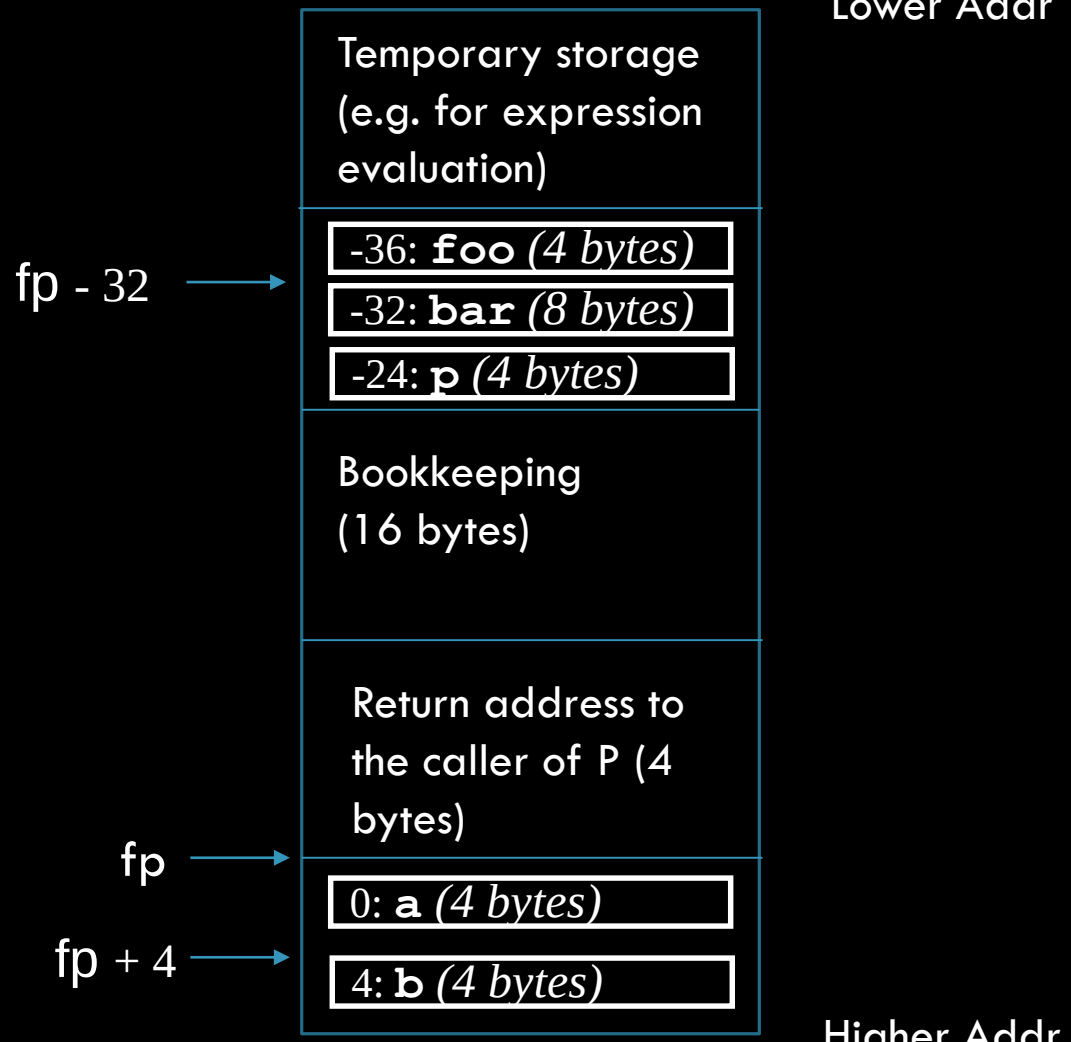


EXAMPLE SUBROUTINE FRAME

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

The heap is used to store objects who lifetime is dynamic.

- 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 PROBLEMS

Heap is a large block of memory (say N bytes).

- Requests for memory of various sizes may arrive randomly.
  - For example, a program executes ‘new’.
- Each request may ask for 1 to N bytes.
- If a request of X bytes is granted, a continuous X bytes in the heap is allocated for the request. The memory will be used for a while and then returned to the system (when the program executes ‘delete’).

The problem: how can we make sure memory is allocated such that as many requests as possible are satisfied?

HEAP ALLOCATION EXAMPLE

Example: 10KB memory to be managed.

```
r1 = req(1K);
r2 = req (2K);
r3 = req(4k);
free(r2);
free(r1);
r4 = req(4k);
```

How we assign memory makes a difference!

- Internal fragment: unused memory within a block.
  - Example: asking for 100 bytes and get a 512 bytes block.
- External fragment: unused memory between blocks.
  - Even when the total available memory is more than a request, the request cannot be satisfied as in the example.

HEAP ALLOCATION ALGORITHMS

Heap allocation is performed by searching the heap for available free space.

For example, suppose we want to allocate a new object E of 20 bytes, where would it fit?

Object A	**Free**	**Object B**	**Object C**	**Free**	**Object D**	**Free**
30 bytes	8 bytes	10 bytes	24 bytes	24 bytes	8 bytes	20 bytes

HEAP ALLOCATION ALGORITHMS

Heap allocation is performed by searching the heap for available free space.

For example, suppose we want to allocate a new object E of 20 bytes, where would it fit?

Object A	**Free**	**Object B**	**Object C**	**Free**	**Object D**	**Free**
30 bytes	8 bytes	10 bytes	24 bytes	24 bytes	8 bytes	20 bytes

Deletion of objects leaves free blocks in the heap that can be reused

- How to keep track of free blocks?
- How to find the right free block for each request?

HEAP ALLOCATION ALGORITHMS

How do we keep track of free blocks?

- Maintain a linked list of free heap blocks.
- To satisfy a request (new), search the list to find one block whose size is equal to or larger than the requested size.
- If the size is equal, remove the block from the free list.
- If the size is larger, modify the size to (size – requested size).
- When an object is deleted (freed), the block of memory is returned to the heap.
- Insert a new block to the free list. If new block can be merged with its neighboring blocks to be a larger block, merge the blocks.

HEAP ALLOCATION ALGORITHMS

How do we pick which block should be used for each request?

There can be many choices.

- **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.

OTHER HEAP ALLOCATION ALGORITHMS

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.

GARBAGE COLLECTION

Explicit manual deallocation errors are among the most expensive and hard to detect problems in real-world applications.

- If an object is deallocated too soon, a reference to the object becomes a dangling reference.
- If an object is never deallocated, the program leaks memory.

Automatic garbage collection removes all objects from the heap that are not accessible, i.e. are not referenced.

- Used in Lisp, Scheme, Prolog, Ada, Java, Haskell.
- Disadvantage is GC overhead, but GC algorithm efficiency has been improved.
- Not always suitable for real-time processing.

GARBAGE COLLECTION

How does it work roughly?

- The language defines the lifetime of objects.
- The runtime keeps track of the number of references (bindings) to each object.
  - Increment when a new reference is made, decrement when the reference is destroyed.
  - Can delete when the reference count is 0.
- Need to determine when a variable is alive or dead based on language specification.

NEXT LECTURE

Names, Scopes, and Bindings: Scoping Issues