Run-time Environments - 3

Y.N. Srikant
Computer Science and Automation
Indian Institute of Science
Bangalore 560 012

NPTEL Course on Principles of Compiler Design

Outline of the Lecture

- What is run-time support? (in part 1)
- Parameter passing methods (in part 1)
- Storage allocation (in part 2)
- Activation records (in part 2)
- Static scope and dynamic scope
- Passing functions as parameters
- Heap memory management
- Garbage Collection

Static Scope and Dynamic Scope

- *Static Scope*
  - A global identifier refers to the identifier with that name that is declared in the closest enclosing scope of the program text
  - Uses the *static* (unchanging) relationship between blocks in the program text
- *Dynamic Scope*
  - A global identifier refers to the identifier with that name associated with the most recent activation record
  - Uses the actual sequence of calls that is executed in the *dynamic* (changing) execution of the program
- Both are identical as far as local variables are concerned

Static Scope and Dynamic Scope : An Example

```
int x = 1, y = 0;
int g(int z)
  { return x+z;}
int f(int y) {
    int x; x = y+1;
    return g(y*x);
}
y = f(3);
```

After the call to g,
Static scope: x = 1
Dynamic scope: x = 4

x	**1**	**outer block**
y	**0**	

y	**3**	**f(3)**
x	**4**	

z	**12**	**g(12)**

Stack of activation records
after the call to *g*

Static Scope and Dynamic Scope: Another Example

```
float r = 0.25;
void show() { printf("%f",r); }
void small() {
   float r = 0.125; show();
}
int main (){
show(); small(); printf("\n");
show(); small(); printf("\n");
}
```

- Under static scoping, the output is

  0.25 0.25

  0.25 0.25

- Under dynamic scoping, the output is

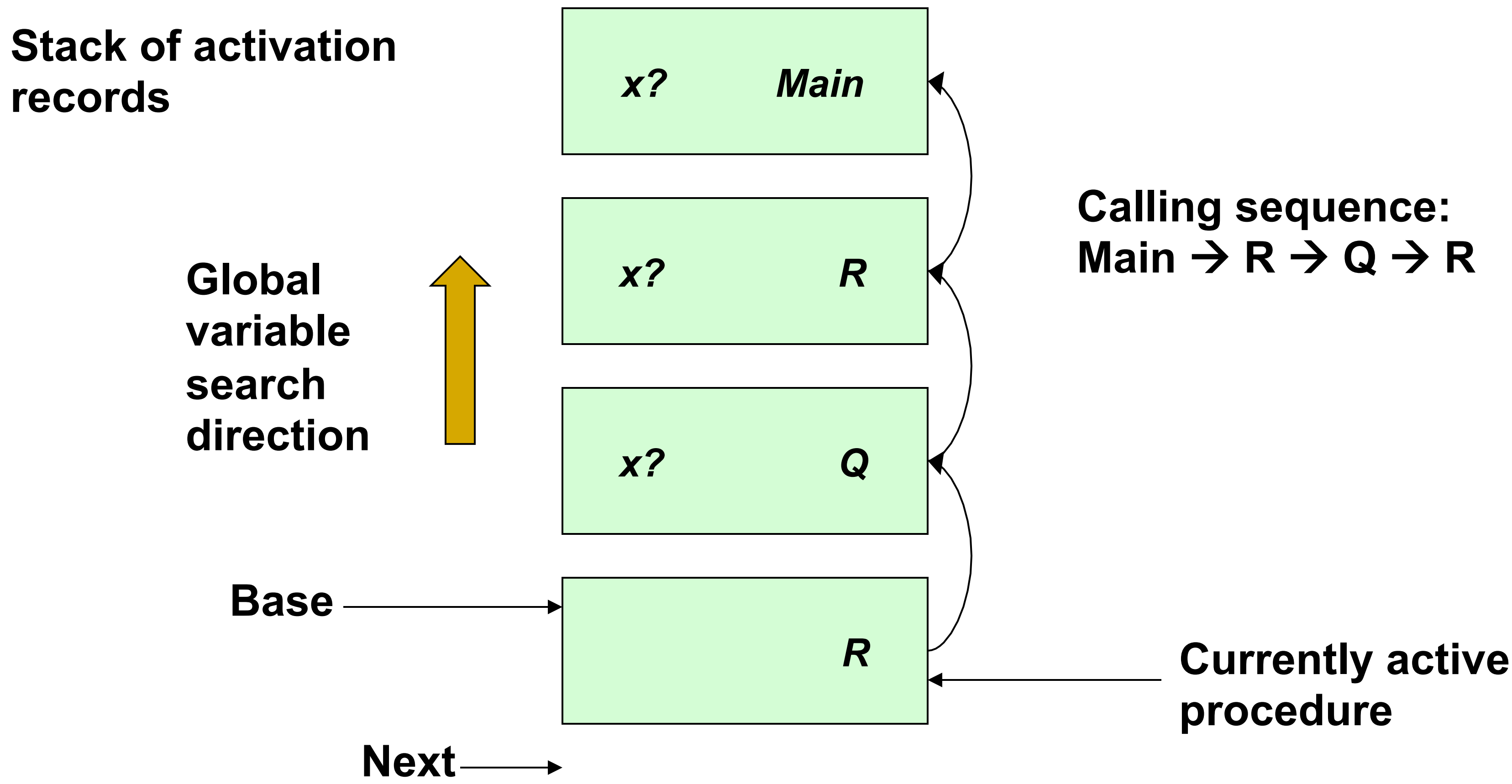
  0.25 0.125

  0.25 0.125

Implementing Dynamic Scope – Deep Access Method

- Use *dynamic link* as *static link*
- Search activation records on the stack to find the first AR containing the non-local name
- The depth of search depends on the input to the program and cannot be determined at compile time
- Needs some information on the identifiers to be maintained at runtime within the ARs
- Takes longer time to access globals, but no overhead when activations begin and end

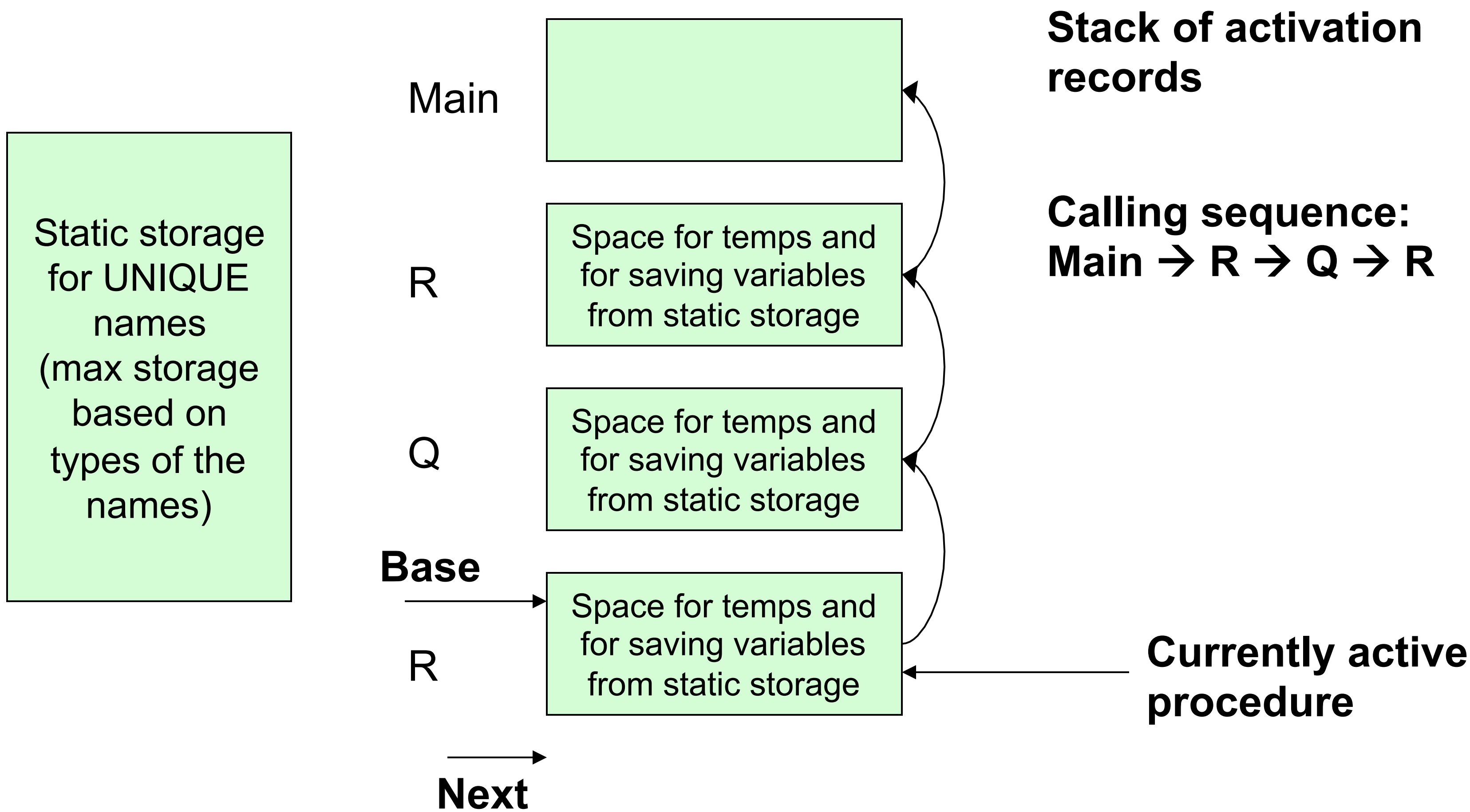
Deep Access Method - Example



Implementing Dynamic Scope – Shallow Access Method

- Allocate maximum static storage needed for *each* name (based on the types)
- When a new AR is created for a procedure *p*, a local name *n* in *p* takes over the static storage allocated to name *n*
  - Global variables are also accessed from the static storage
  - Temporaries are located in the AR
  - Therefore, all variable (not temp) accesses use static addresses
- The previous value of *n* held in static storage is saved in the AR of *p* and is restored when the activation of *p* ends
- Direct and quick access to globals, but some overhead is incurred when activations begin and end

Shallow Access Method - Example



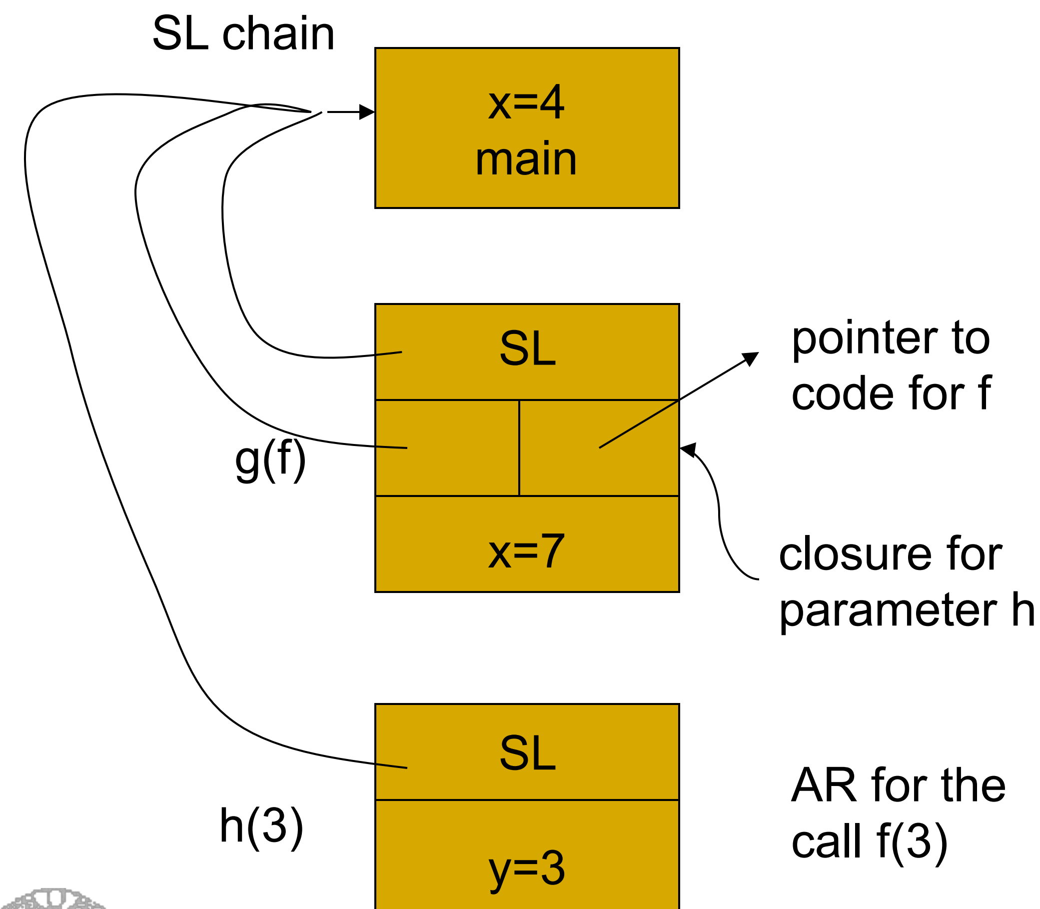
Passing Functions as Parameters

An example:

```
main()
{ int x = 4;
  int f (int y) {
     return x*y;
  }
  int g (int → int h){
     int x = 7;
     return h(3) + x;
   }
   g(f);
}
```

- A language has first-class functions if functions can be
  - declared within any scope
  - passed as arguments to other functions
  - returned as results of functions
- In a language with first-class functions and static scope, a function value is generally represented by a closure
  - a pair consisting of a pointer to function code and
  - a pointer to an activation record
- Passing functions as arguments is very useful in structuring of systems using callbacks

Passing Functions as Parameters – Implementation



An example:

```
main()
{ int x = 4;
  int f (int y) {
     return x*y;
  }
  int g (int → int h){
     int x = 7;
     return h(3) + x;
   }
   g(f);
}
```

Passing Functions as Parameters: Implementation

An example:

```
main()
{ int x = 4;
  int f (int y) {
     return x*y;
  }
  int g (int → int h){
     int x = 7;
     return h(3) + x;
   }
   g(f);
}
```

- In this example, when executing the call h(3), h is really f and 3 is the parameter y of f
- Without passing a closure, the AR of the main program cannot be accessed, and hence, the value of x within f will not be 4
- In the call g(f), f is passed as a closure
- Closure may also contain information needed to set up AR (e.g., size of space for local variables, etc.)
- When processing the call h(3), after setting up an AR for h (i.e., f), the SL for the AR is set up using the AR pointer in the closure for f that has been passed to the call g(f)

Heap Memory Management

- Heap is used for allocating space for objects created at run time
  - For example: nodes of dynamic data structures such as linked lists and trees
- Dynamic memory allocation and deallocation based on the requirements of the program
  - *malloc*() and *free*() in C programs
  - *new*() and *delete*() in C++ programs
  - *new*() and garbage collection in Java programs
- Allocation and deallocation may be *completely manual* (C/C++), *semi-automatic* (Java), or *fully automatic* (Lisp)

Memory Manager

- Manages heap memory by implementing mechanisms for allocation and deallocation, both manual and automatic
- Goals
  - Space efficiency: minimize fragmentation
  - Program efficiency: take advantage of locality of objects in memory and make the program run faster
  - Low overhead: allocation and deallocation must be efficient
- Heap is maintained either as a doubly linked list or as bins of free memory chunks (more on this later)

Allocation and Deallocation

- In the beginning, the heap is one large and contiguous block of memory
- As allocation requests are satisfied, chunks are cut off from this block and given to the program
- As deallocations are made, chunks are returned to the heap and are free to be allocated again (*holes*)
- After a number of allocations and deallocations, memory becomes fragmented and is not contiguous
- Allocation from a fragmented heap may be made either in a *first-fit* or *best-fit* manner
- After a deallocation, we try to *coalesce* contiguous holes and make a bigger hole (free chunk)

First-Fit and Best-Fit Allocation Strategies

- The *first-fit* strategy picks the **first** available chunk that satisfies the allocation request
- The *best-fit* strategy searches and picks the smallest (**best**) possible chunk that satisfies the allocation request
- Both of them chop off a block of the required size from the chosen chunk, and return it to the program
- The rest of the chosen chunk remains in the heap

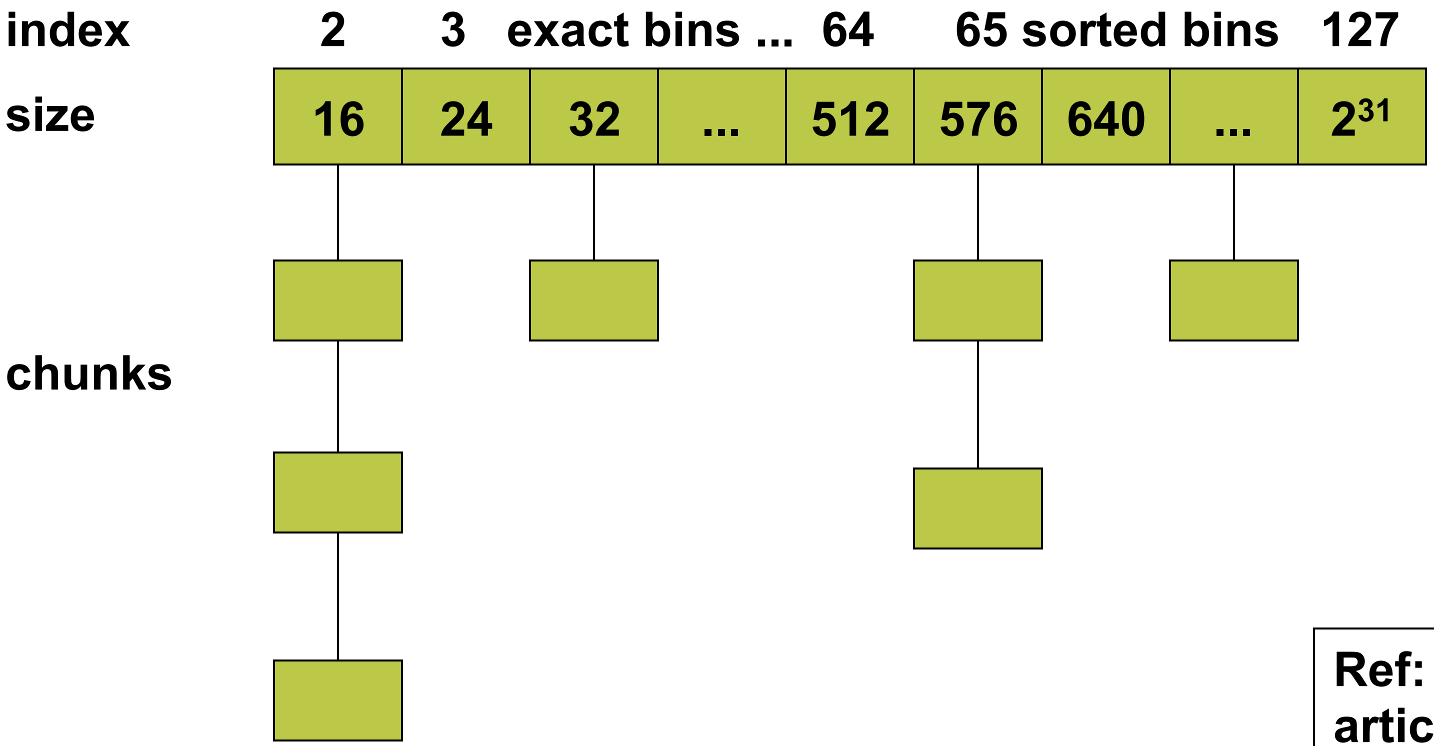
First-Fit and Best-Fit Allocation Strategies

- Best-fit strategy has been shown to reduce fragmentation in practice, better than first-fit strategy
- *Next-fit* strategy tries to allocate the object in the chunk that has been split recently
  - Tends to improve speed of allocation
  - Tends to improve spatial locality since objects allocated at about the same time tend to have similar reference patterns and life times (cache behaviour may be better)

Bin-based Heap

- Free space is organized into *bins* according to their sizes (Lea Memory Manager in GCC)
  - Many more bins for smaller sizes, because there are many more small objects
  - A bin for every multiple of 8-byte chunks from 16 bytes to 512 bytes
  - Then approximately logarithmically (double previous size)
  - Within each "small size bin", chunks are all of the same size
  - In others, they are ordered by size
  - The last chunk in the last bin is the *wilderness chunk*, which gets us a chunk by going to the operating system

Bin-based Heap – An Example



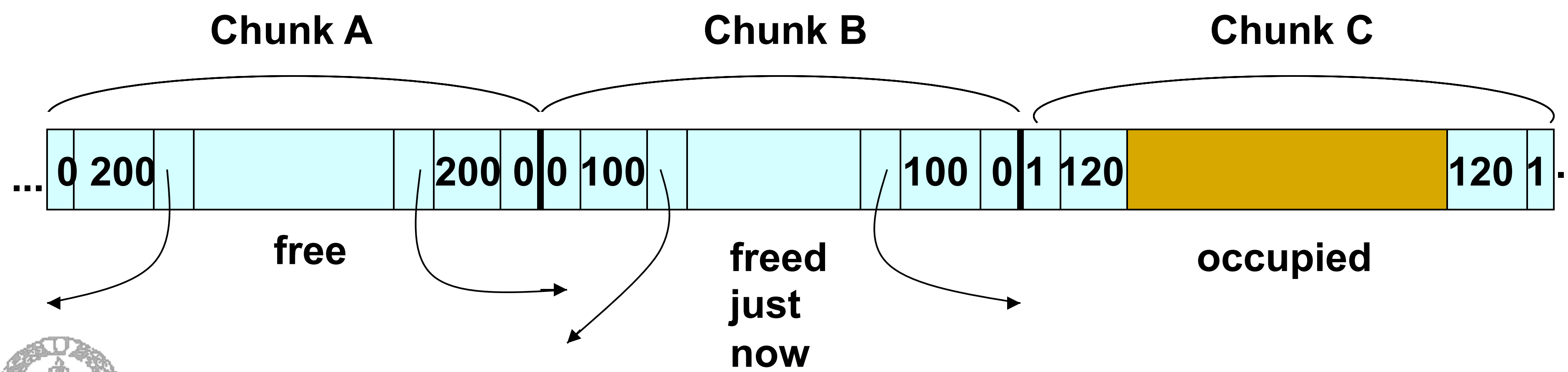
Ref: From Lea's article on memory manager in GCC

Managing and Coalescing Free Space

- Should coalesce adjacent chunks and reduce fragmentation
  - Many small chunks together cannot hold one large object
  - In the Lea memory manager, no coalescing in the exact size bins, only in the sorted bins
  - *Boundary tags* (free/used bit and chunk size) at each end of a chunk (for both used and free chunks)
  - A *doubly linked list* of free chunks

Boundary Tags and Doubly Linked List

3 adjacent chunks. Chunk B has been freed just now and returned to the free list. Chunks A and B can be merged, and this is done just before inserting it into the linked list. The merged chunk AB may have to be placed in a different bin.



Problems with Manual Deallocation

- Memory leaks
  - Failing to delete data that cannot be referenced
  - Important in long running or nonstop programs
- Dangling pointer dereferencing
  - Referencing deleted data
- Both are serious and hard to debug
- Solution: automatic garbage collection

Garbage Collection

- Reclamation of chunks of storage holding objects that can no longer be accessed by a program
- GC should be able to determine types of objects
  - Then, size and pointer fields of objects can be determined by the GC
  - Languages in which types of objects can be determined at compile time or run-time are type safe
    - Java is type safe
    - C and C++ are not type safe because they permit type casting, which creates new pointers
    - Thus, any memory location can be (theoretically) accessed at any time and hence cannot be considered inaccessible

Reachability of Objects

- The *root set* is all the data that can be accessed (reached) directly by a program without having to dereference any pointer
- Recursively, any object whose reference is stored in a field of a member of the root set is also reachable
- New objects are introduced through object allocations and add to the set of reachable objects
- Parameter passing and assignments can propagate reachability
- Assignments and ends of procedures can terminate reachability