CS 536

Introduction to Programming Languages and Compilers

Charles N. Fischer

Lecture 11

Handling Overloaded Declarations

Two approaches are popular:

1. Create a single symbol table entry using the method's name.
   The entry will contain a list of parameter combinations, one for each declaration.
   Thus for the declarations

   ```
   int max(int v1, int v2)
   ```

   and

   ```
   int max(int v1, int v2, int v3)
   ```

   we would have *one* symbol table entry for `max` and in that entry *two* parameter lists:
   (int,int) and (int,int,int)

2. Name Mangling
   Parameter information is appended to the method's name to form a *unique* name for the method. If a code of "I" is used for an integer parameter, we create two method names: `max(II)` and `max(III)`.

Calling Overloaded Methods

If all overloadings of a method name share a single symbol table entry, we check the actual parameters against each declared parameter list. Exactly one should match.
Thus given a call of `max(1,2)`, we check the list (int,int) against each declared parameter sequence.

If we use name mangling, we use the actual parameters to form the expected method name, and look it up in the symbol table. For `max(1,2)` we lookup `max(II)`.
Note that this approach makes clear error messages a bit more difficult. Thus if we see a call `max(1,2,3,4)` we *shouldn't* say `max` is undeclared (it isn't!)

Reading Assignment

Read Chapters 9 and 12 of Crafting a Compiler.

Virtual Memory & Run-Time Memory Organization

The compiler decides how data and instructions are placed in memory.

It uses an *address space* provided by the hardware and operating system.

This address space is usually *virtual*—the hardware and operating system map instruction- level addresses to "actual" memory addresses.

Virtual memory allows:

- Multiple processes to run in *private*, *protected* address spaces.
- *Paging* can be used to extend address ranges beyond actual memory limits.

Run-Time Data Structures

Static Structures

For static structures, a *fixed* address is used throughout execution.

This is the oldest and simplest memory organization.

In current compilers, it is used for:

- Program code (often read- only & sharable).
- Data literals (often read- only & sharable).
- Global variables.
- Static variables.

Stack Allocation

Modern programming languages allow recursion, which requires *dynamic allocation*.

Each recursive call allocates a *new copy* of a routine's local variables.

The number of local data allocations required during program execution is not known at compile- time.

To implement recursion, all the data space required for a method is treated as a distinct data area that is called a *frame* or *activation record*.

Local data, within a frame, is accessible only while a subprogram is active.

In mainstream languages like C, C++ and Java, subprograms must return in a stack- like manner—the most recently called subprogram will be the first to return.

A frame is pushed onto a *run-time stack* when a method is called (activated).

When it returns, the frame is popped from the stack, freeing the routine's local data.

As an example, consider the following C subprogram:

```
p(int a)
  { double b;
  double c[10];
  b = c[a] * 2.51;
}
```

Procedure **p** requires space for the parameter **a** as well as the local variables **b** and **c**.

It also needs space for control information, such as the return address.

The compiler records the space requirements of a method.

The *offset* of each data item relative to the start of the frame is stored in the symbol table.

The total amount of space needed, and thus the size of the frame, is also recorded.

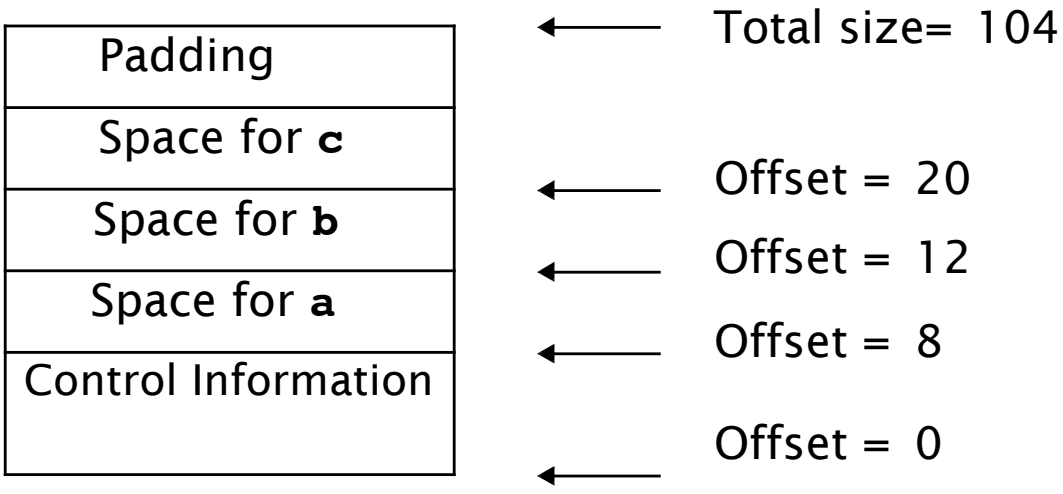
Assume **p**'s control information requires 8 bytes (this size is usually the same for all methods).

Assume parameter a requires 4 bytes, local variable **b** requires 8 bytes, and local array **c** requires 80 bytes.

Many machines require that word and doubleword data be *aligned*, so it is common to pad a frame so that its size is a multiple of 4 or 8 bytes.

This guarantees that at all times the top of the stack is properly aligned.

Here is **`p`**'s frame:



Within **p**, each local data object is addressed by its offset relative to the start of the frame.

This offset is a fixed constant, determined at compile- time.

We normally store the start of the frame in a register, so each piece of data can be addressed as a (Register, Offset) pair, which is a standard addressing mode in almost all computer architectures.

For example, if register R points to the beginning of **p**'s frame, variable **b** can be addressed as (R, 12), with 12 actually being added to the contents of R at run- time, as memory addresses are evaluated.

Normally, the literal **`2.51`** of procedure **`p`** is *not* stored in **`p`**'s frame because the values of local data that are stored in a frame disappear with it at the end of a call.

It is easier and more efficient to allocate literals in a *static area*, often called a *literal pool* or *constant pool*. Java uses a constant pool to store literals, type, method and interface information as well as class and field names.

Accessing Frames at Run- Time

During execution there can be many frames on the stack. When a procedure `A` calls a procedure `B`, a frame for `B`'s local variables is pushed on the stack, covering `A`'s frame. `A`'s frame can't be popped off because `A` will resume execution after `B` returns.

For recursive routines there can be hundreds or even thousands of frames on the stack. All frames but the topmost represent suspended subroutines, waiting for a call to return.

The topmost frame is *active*; it is important to access it directly.

The active frame is at the top of the stack, so the *stack top*

register could be used to access it.

The run- time stack may also be used to hold data other than frames.

It is unwise to require that the currently active frame always be at *exactly* the top of the stack.

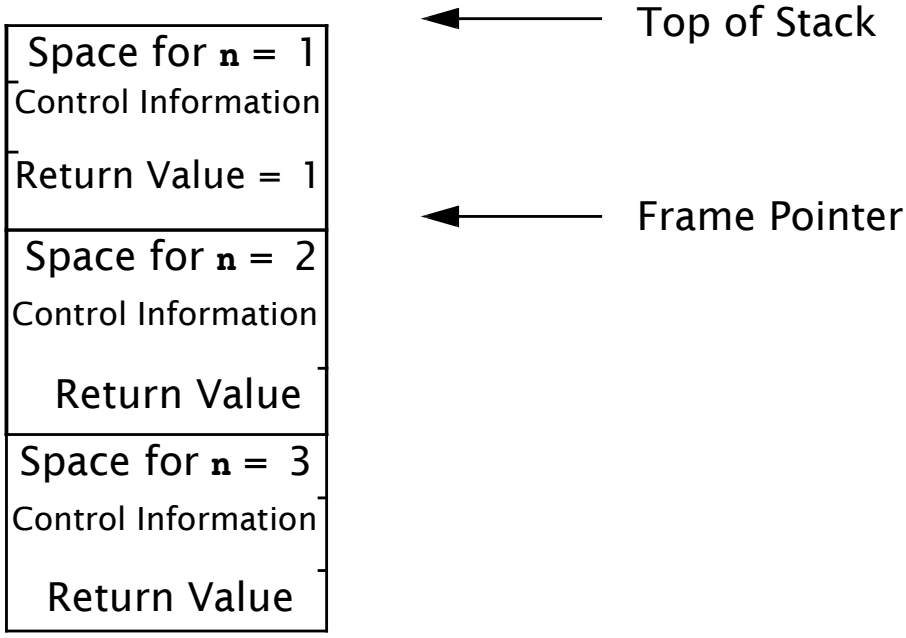
Instead a distinct register, often called the *frame pointer*, is used to access the current frame.

This allows local variables to be accessed directly as offset + frame pointer, using the indexed addressing mode found on all modern machines.

Consider the following recursive function that computes factorials.

```
int fact(int n)
  { if (n > 1)
    return n * fact(n-1);
  else
    return 1;
}
```

The run- time stack corresponding to the call **`fact(3)`** (when the call of **`fact(1)`** is about to return) is:



We place a slot for the function's return value at the very beginning of the frame.

Upon return, the return value is conveniently placed on the stack, just beyond the end of the caller's frame. Often compilers return scalar values in specially

designated registers, eliminating unnecessary loads and stores. For values too large to fit in a register (arrays or objects), the stack is used.

When a method returns, its frame is popped from the stack and the frame pointer is reset to point to the caller's frame.

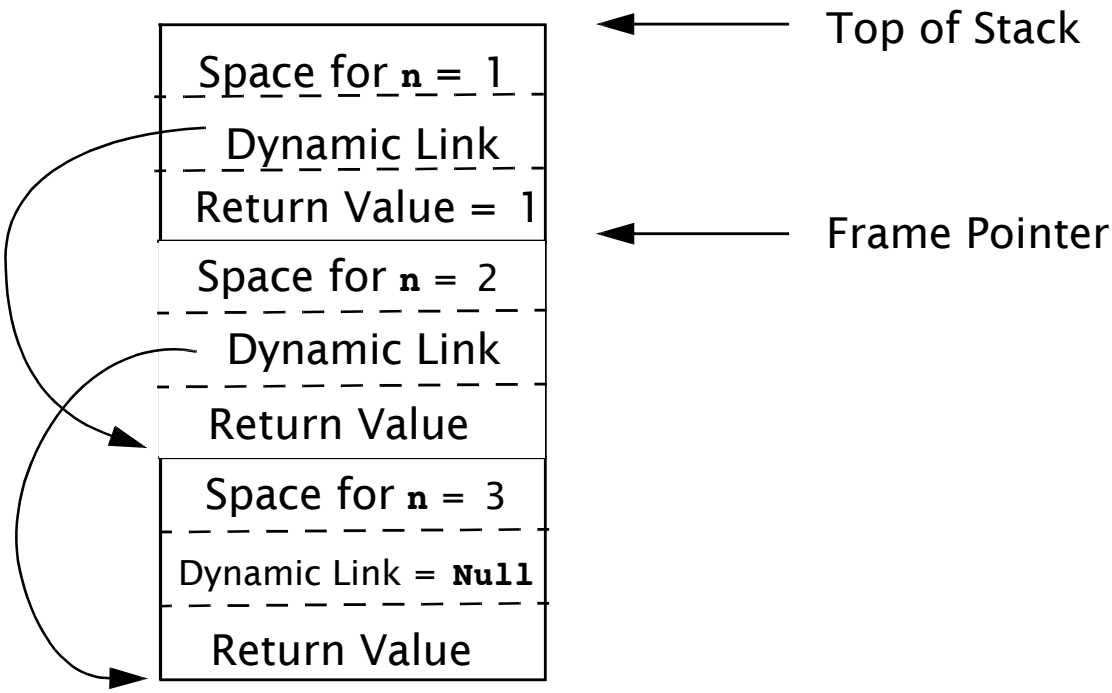
In simple cases this is done by adjusting the frame pointer by the size of the current frame.

Dynamic Links

Because the stack may contain more than just frames (e.g., function return values or registers saved across calls), it is common to save the caller's frame pointer as part of the callee's control information.

Each frame points to its caller's frame on the stack. This pointer is called a *dynamic link* because it links a frame to its dynamic (run-time) predecessor.

The run- time stack corresponding to a call of **`fact(3)`**, with dynamic links included, is:



Classes and Objects

C, C++ and Java do not allow procedures or methods to nest.

A procedure may not be declared within another procedure.

This simplifies run- time data access—all variables are either global or local.

Global variables are statically allocated. Local variables are part of a single frame, accessed through the frame pointer.

Java and C++ allow classes to have *member functions* that have direct access to instance variables.

Consider:

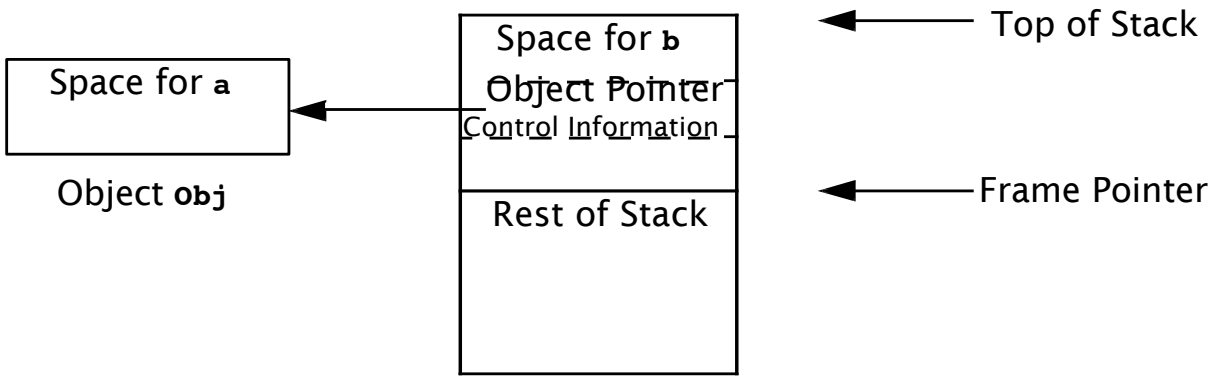
```
class K
{ int a;
  int sum()
  { int b;
  return a+b;
} }
```

Each object that is an instance of class **K** contains a member function **sum**. Only one translation of **sum** is created; it is shared by all instances of **K**.

When **sum** executes it needs *two* pointers to access local and object- level data.

Local data, as usual, resides in a frame on the run- time stack.

Data values for a particular instance of **`K`** are accessed through an object pointer (called the **`this`** pointer in Java and C++). When **`obj.sum()`** is called, it is given an extra *implicit parameter* that a pointer to **`obj`**.



When **`a+b`** is computed, **`b`**, a local variable, is accessed directly through the frame pointer. **`a`**, a member of object **`obj`**, is accessed indirectly through the object pointer that is stored in the frame (as all parameters to a method are).

C++ and Java also allow inheritance via subclassing. A new class can extend an existing class, adding new fields and adding or redefining methods.

A subclass **`D`**, of class **`C`**, maybe be used in contexts expecting an object of class **`C`** (e.g., in method calls).

This is supported rather easily—objects of class **`D`** always contain a class **`C`** object within them.

If **`C`** has a field **`F`** within it, so does **`D`**. The fields **`D`** declares are merely *appended* at the end of the allocations for **`C`**.

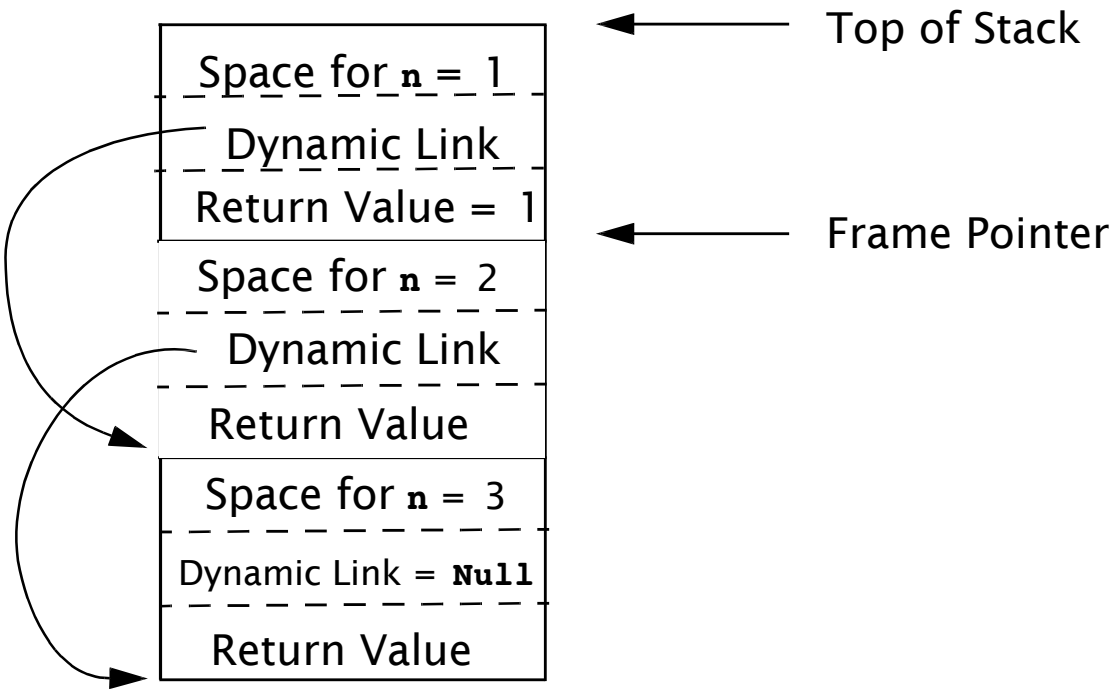
As a result, access to fields of **`C`** within a class **`D`** object works perfectly.

Dynamic Links

Because the stack may contain more than just frames (e.g., function return values or registers saved across calls), it is common to save the caller's frame pointer as part of the callee's control information.

Each frame points to its caller's frame on the stack. This pointer is called a *dynamic link* because it links a frame to its dynamic (run-time) predecessor.

The run- time stack corresponding to a call of **`fact(3)`**, with dynamic links included, is:



Handling Multiple Scopes

Many languages allow procedure declarations to nest. Java now allows classes to nest.

Procedure nesting can be very useful, allowing a subroutine to directly access another routine's locals and parameters.

Run- time data structures are complicated because multiple frames, corresponding to nested procedure declarations, may need to be accessed.

To see the difficulties, assume that routines *can* nest in Java or C:

```
int p(int a)
  {int q(int b)
   {
    if (b < 0)
      return q(-b);
    else
     return a+b;
  }
  return q(-10);
}
```

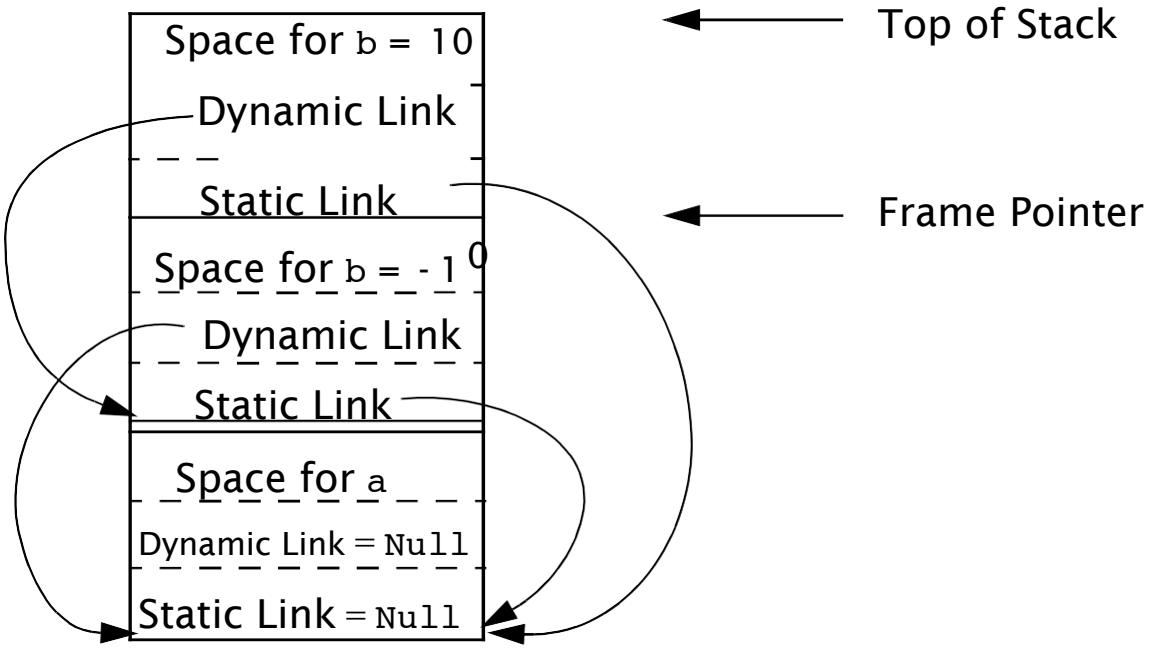
When **q** executes, it may access not only its own frame, but also that of **p**, in which it is nested.

If the depth of nesting is unlimited, so is the number of frames that must be accessible. In practice, the level of nesting actually seen is modest—usually no greater than two or three.

Static Links

Two approaches are commonly used to support access to multiple frames. One approach generalizes the idea of dynamic links introduced earlier. Along with a dynamic link, we'll also include a *static link* in the frame's control information area. The static link points to the frame of the procedure that statically encloses the current procedure. If a procedure is not nested within any other procedure, its static link is **`null`**.

The following illustrates static links:



As usual, dynamic links always point to the next frame down in the stack. Static links always point down, but they may skip past many frames. They always point to the most recent frame of the routine that statically encloses the current routine.

In our example, the static links of both of **q**'s frames point to **p**, since it is **p** that encloses **q**'s definition.

In evaluating the expression **a+b** that **q** returns, **b**, being local to **q**, is accessed directly through the frame pointer. Variable **a** is local to **p**, but also visible to **q** because **q** nests within **p**. **a** is accessed by extracting **q**'s static link, then using that address (plus the appropriate offset) to access **a**.

Displays

An alternative to using static links to access frames of enclosing routines is the use of a *display*.

A display generalizes our use of a frame pointer. Rather than maintaining a single register, we maintain a *set of registers* which comprise the display.

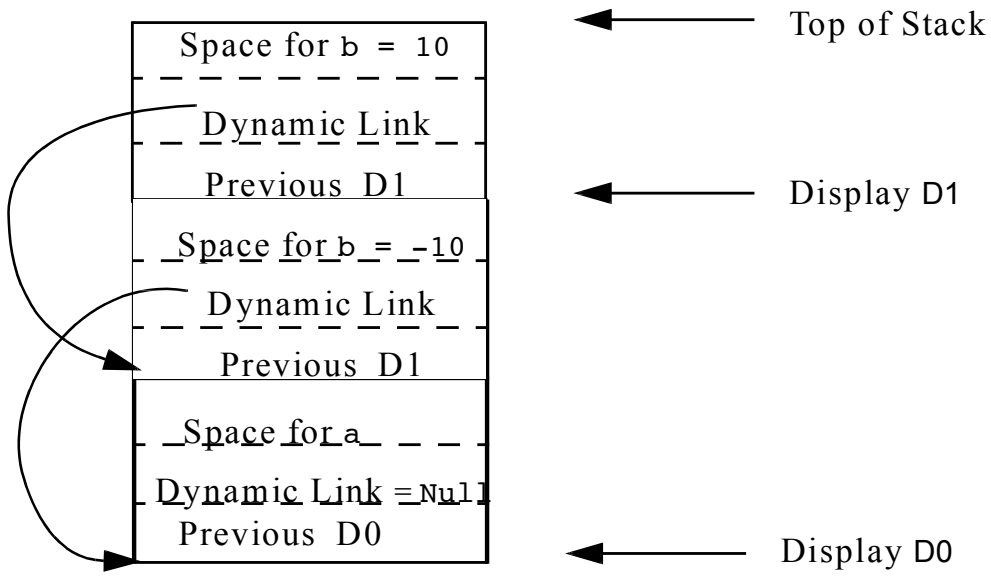
If procedure definitions nest *n* deep (this can be easily determined by examining a program's AST), we need *n+1* display registers.

Each procedure definition is tagged with a nesting level. Procedures not nested within any other routine are at level 0.
Procedures nested within only one routine are at level 1, etc.

Frames for routines at level 0 are always accessed using display register D0. Those at level 1 are always accessed using register D1, etc.

Whenever a procedure **`r`** is executing, we have direct access to **`r`**'s frame plus the frames of all routines that enclose **`r`**. Each of these routines must be at a different nesting level, and hence will use a different display register.

The following illustrates the use of display registers:



Since `q` is at nesting level 1, its frame is pointed to by D1. All of **`q`**'s local variables, including **`b`**, are at a fixed offset relative to D1.

Since **`p`** is at nesting level 0, its frame and local variables are accessed via D0. Each frame's control information area contains a slot for the previous value of the frame's display register. A display register is saved when a call

begins and restored when the call ends. A dynamic link is still needed, because the previous display values doesn't always point to the caller's frame.

Not all compiler writers agree on whether static links or displays are better to use. Displays allow direct access to all frames, and thus make access to all visible variables very efficient. However, if nesting is deep, several valuable registers may need to be reserved. Static links are very flexible, allowing unlimited nesting of procedures. However, access to non- local procedure variables can be slowed by the need to extract and follow static links.

Heap Management

A very flexible storage allocation mechanism is *heap allocation.*

Any number of data objects can be allocated and freed in a memory pool, called a *heap*.

Heap allocation is enormously popular. Almost all non- trivial Java and C programs use **`new`** or **`malloc`**.

Heap Allocation

A request for heap space may be *explicit* or *implicit*.

An explicit request involves a call to a routine like **`new`** or **`malloc`**. An explicit pointer to the newly allocated space is returned.

Some languages allow the creation of data objects of unknown size. In Java, the + operator is overloaded to represent string catenation.

The expression **`Str1 + Str2`** creates a new string representing the catenation of strings **`Str1`** and **`Str2`**. There is no compile- time bound on the sizes of **`Str1`** and **`Str2`**, so heap space must be implicitly allocated to hold the newly created string.

Whether allocation is explicit or implicit, a *heap allocator* is needed. This routine takes a size parameter and examines unused heap space to find space that satisfies the request.

A *heap block* is returned. This block must be big enough to satisfy the space request, but it may well be bigger.

Heaps blocks contain a *header* field that contains the size of the block as well as bookkeeping information.

The complexity of heap allocation depends in large measure on how *deallocation* is done.

Initially, the heap is one large block of unallocated memory. Memory requests can be satisfied by simply modifying an “end of

heap" pointer, very much as a stack is pushed by modifying a stack pointer.

Things get more involved when previously allocated heap objects are deallocated and reused.

Deallocated objects are stored for future reuse on a *free space list*.

When a request for *n* bytes of heap space is received, the heap allocator must search the free space list for a block of sufficient size. There are many search strategies that might be used:

- Best Fit

  The free space list is searched for the free block that matches most closely the requested size. This minimizes wasted heap space, the search may be quite slow.

- First Fit

  The first free heap block of sufficient size is used. Unused space within the block is split off and linked as a smaller free space block. This approach is fast, but may "clutter" the beginning of the free space list with a number of blocks too small to satisfy most requests.

- Next Fit

  This is a variant of first fit in which succeeding searches of the free space list begin at the position where the last search ended. The idea is to "cycle through" the entire free space list rather than always revisiting free blocks at the head of the list.

- **Segregated Free Space Lists**
  There is no reason why we must have only *one* free space list. An alternative is to have several, indexed by the size of the free blocks they contain.

Deallocation Mechanisms

Allocating heap space is fairly easy. But how do we deallocate heap memory no longer in use?

Sometimes we may never need to deallocate! If heaps objects are allocated infrequently or are very long- lived, deallocation is unnecessary. We simply fill heap space with “in use” objects.

Virtual memory & paging may allow us to allocate a very large heap area.

On a 64- bit machine, if we allocate heap space at 1 MB/sec, it will take 500,000 *years* to span the entire address space! Fragmentation of a very large heap space commonly forces us to include some form of reuse of heap space.

User-controlled Deallocation

Deallocation can be manual or automatic. Manual deallocation involves explicit programmer-initiated calls to routines like **`free(p)`** or **`delete(p)`**.

The object is then added to a free- space list for subsequent reallocation.

It is the programmer's responsibility to free unneeded heap space by executing deallocation commands. The heap manager merely keeps track of freed space and makes it available for later reuse.

The really hard decision—when space should be freed—is shifted to the programmer, possibly leading to catastrophic *dangling pointer* errors.

Consider the following C program fragment

```
q = p = malloc(1000);
free(p);
/* code containing more malloc's */
q[100] = 1234;
```

After **p** is freed, **q** is a *dangling pointer*. **q** points to heap space that is no longer considered allocated.

Calls to **malloc** may reassign the space pointed to by **q**. Assignment through **q** is illegal, but this error is almost never detected.

Such an assignment may change data that is now part of another heap object, leading to very subtle errors. It may even change a header field or a free- space link, causing the heap allocator itself to fail!

Automatic Garbage Collection

The alternative to manual deallocation of heap space is *garbage collection*.

Compiler- generated code tracks pointer usage. When a heap object is no longer pointed to, it is *garbage*, and is *automatically* collected for subsequent reuse.

Many garbage collection techniques exist. Here are some of the most important approaches:

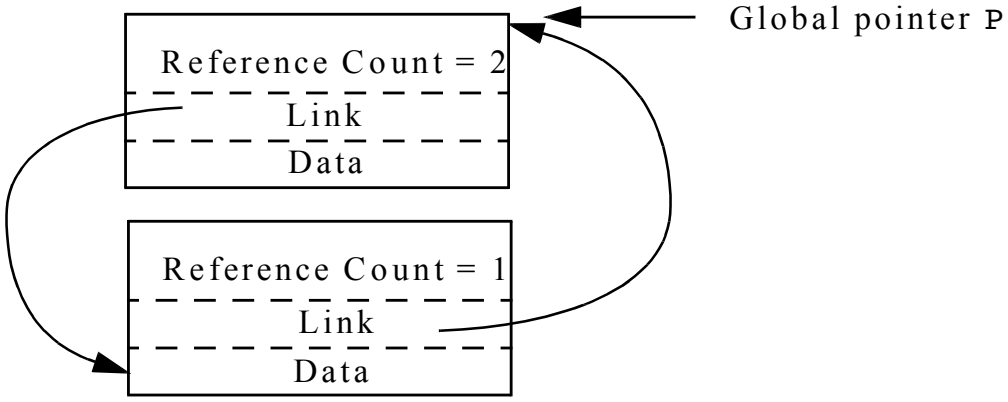
Reference Counting

This is one of the oldest and simplest garbage collection techniques.

A *reference count* field is added to each heap object. It counts how many references to the heap object exist. When an object's reference count reaches zero, it is garbage and may collected.

The reference count field is updated whenever a reference is created, copied, or destroyed. When a reference count reaches zero and an object is collected, all pointers in the collected object are also be followed and corresponding reference counts decremented.

As shown below, reference counting has difficulty with *circular structures.* If pointer **P** is



set to null, the object's reference count is reduced to 1. Both objects have a non- zero count, but neither is accessible through any external pointer. The two objects are garbage, but won't be recognized as such.

If circular structuresare common, then an auxiliary technique, like mark- sweep collection, is needed to collect garbage that reference counting misses.

Mark-Sweep Collection

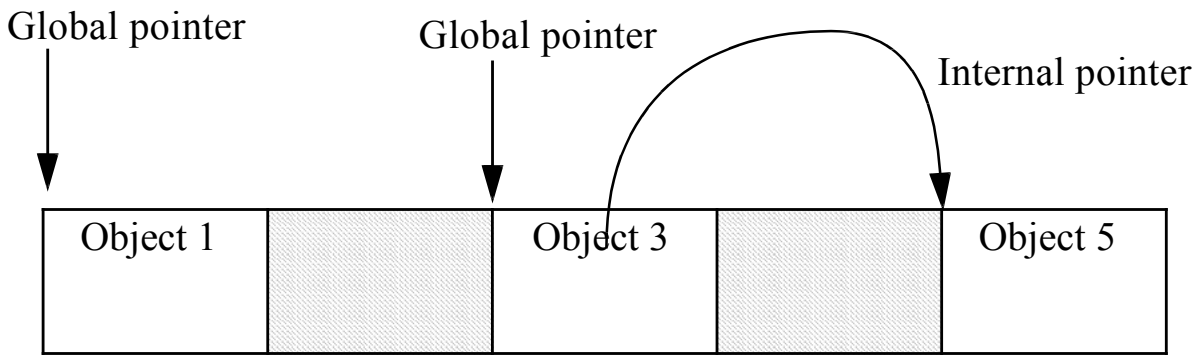
Many collectors, including mark & sweep, do *nothing* until heap space is nearly exhausted.

Then it executes a *marking phase* that identifies all live heap objects.

Starting with global pointers and pointers in stack frames, it marks reachable heap objects. Pointers in marked heap objects are also followed, until all live heap objects are marked.

After the marking phase, any object not marked is garbage that may be freed. We then *sweep* through the heap, collecting all unmarked objects. During the sweep phase we also clear all marks from heap objects found to be still in use.

Mark- sweep garbage collection is illustrated below.



Objects 1 and 3 are marked because they are pointed to by global pointers. Object 5 is marked because it is pointed to by object 3, which is marked. Shaded objects are not marked and will be added to the free- space list.

In any mark- sweep collector, it is vital that we mark *all* accessible heap objects. If we miss a pointer, we may fail to mark a live heap object and later incorrectly free it. Finding all pointers is a bit tricky

in languages like Java, C and C ++, that have pointers mixed with other types within data structures, implicit pointers to temporaries, and so forth. Considerable information about data structures and frames must be available at run- time for this purpose. In cases where we can't be sure if a value is a pointer or not, we may need to do *conservative garbage collection*.

In mark- sweep garbage collection *all* heap objects must be swept. This is costly if most objects are dead. We'd prefer to examine *only* live objects.

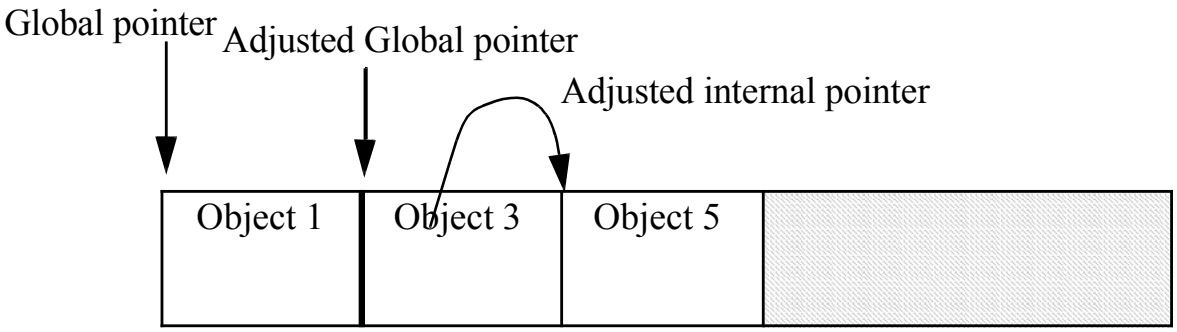
Compaction

After the sweep phase, live heap objects are distributed throughout the heap space. This can lead to poor locality. If live objects span many memory pages, paging overhead may be increased. Cache locality may be degraded too.

We can add a *compaction phase* to mark- sweep garbage collection.

After live objects are identified, they are placed together at one end of the heap. This involves another tracing phase in which global, local and internal heap pointers are found and adjusted to reflect the object's new location.

Pointers are adjusted by the total size of all garbage objects between the start of the heap and the current object. This is illustrated below:



Compaction merges together freed objects into one large block of free heap space. Fragments are no longer a problem.

Moreover, heap allocation is greatly simplified. Using an "end of heap" pointer, whenever a heap request is received, the end of heap pointer is adjusted, making heap allocation no more complex than stack allocation.

Because pointers are adjusted, compaction may not be suitable for languages like C and C++, in which it is difficult to unambiguously identify pointers.