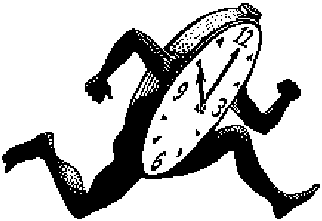
Runtime Environments

Stephen A. Edwards

Columbia University

Fall 2018



Storage Classes

The Stack and Activation Records

In-Memory Layout Issues

The Heap

Automatic Garbage Collection

Shared Libraries and Dynamic Linking

Objects and Inheritance

Exceptions

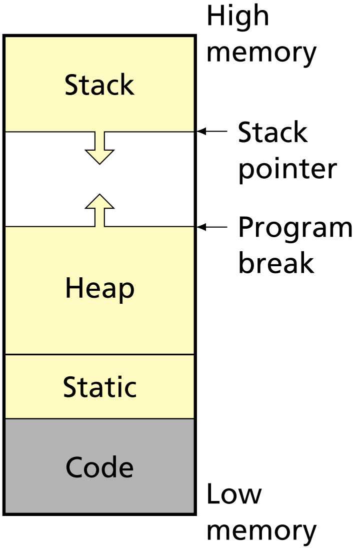
Storage Classes

Storage Classes and Memory Layout

Stack: objects created/destroyed in last-in, first-out order

Heap: objects created/destroyed in any order; automatic garbage collection optional

Static: objects allocated at compile time; persist throughout run



Static Objects

```
class Example {
  public static final int a = 3;

  public void hello() {
    System.out.println("Hello");
  }
}
```

Examples

Static class variable

Code for hello method

String constant “Hello”

Information about the Example class

Advantages

Zero-cost memory management

Often faster access (address a constant)

No out-of-memory danger

Disadvantages

Size and number must be known beforehand

Wasteful if sharing is possible

The Stack and Activation Records

Stack-Allocated Objects

Natural for supporting recursion.

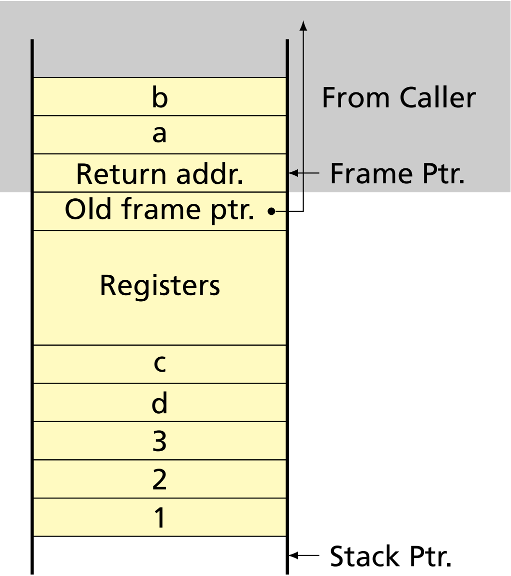
Idea: some objects persist from when a procedure is called to when it returns.

Naturally implemented with a stack: linear array of memory that grows and shrinks at only one boundary.

Each invocation of a procedure gets its own *frame* (*activation record*) where it stores its own local variables and bookkeeping information.

An Activation Record: The State Before Calling *bar*

```
int foo(int a, int b) {
  int c, d;
  bar(1, 2, 3);
}
```



Recursive Fibonacci

(Real C)

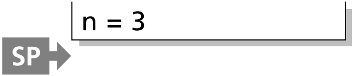
```
int fib(int n) {

  if (n<2)

    return 1;
  else
    return
        fib(n-1)
        +
        fib(n-2);

}
```

(Assembly-like C)

```
int fib(int n) {
     int tmp1, tmp2, tmp3;
     tmp1 = n < 2;
     if (!tmp1) goto L1;
     return 1;
L1:  tmp1 = n - 1;
     tmp2 = fib(tmp1);
L2:  tmp1 = n - 2;
     tmp3 = fib(tmp1);
L3:  tmp1 = tmp2 + tmp3;
     return tmp1;
}
```

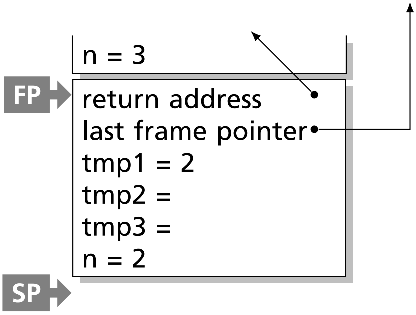
```
           fib(3)
          /      \
     fib(2)      fib(1)
     /    \
fib(1)    fib(0)
```

Executing fib(3)




```
int fib(int n) {
    int tmp1, tmp2, tmp3;
    tmp1 = n < 2;
    if (!tmp1) goto L1;
    return 1;
L1: tmp1 = n - 1;
    tmp2 = fib(tmp1);
L2: tmp1 = n - 2;
    tmp3 = fib(tmp1);
L3: tmp1 = tmp2 + tmp3;
    return tmp1;
}
```

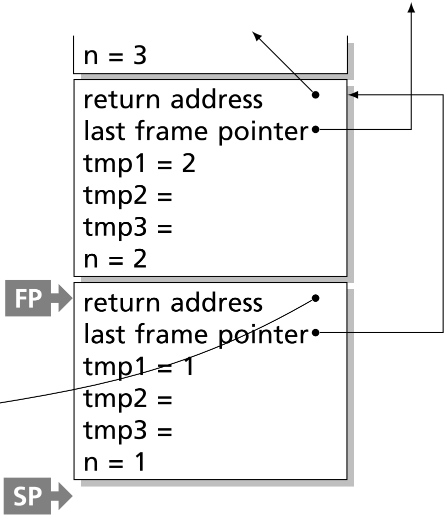
Executing fib(3)




```
int fib(int n) {
    int tmp1, tmp2, tmp3;
    tmp1 = n < 2;
    if (!tmp1) goto L1;
    return 1;
L1: tmp1 = n - 1;
    tmp2 = fib(tmp1);
L2: tmp1 = n - 2;
    tmp3 = fib(tmp1);
L3: tmp1 = tmp2 + tmp3;
    return tmp1;
}
```

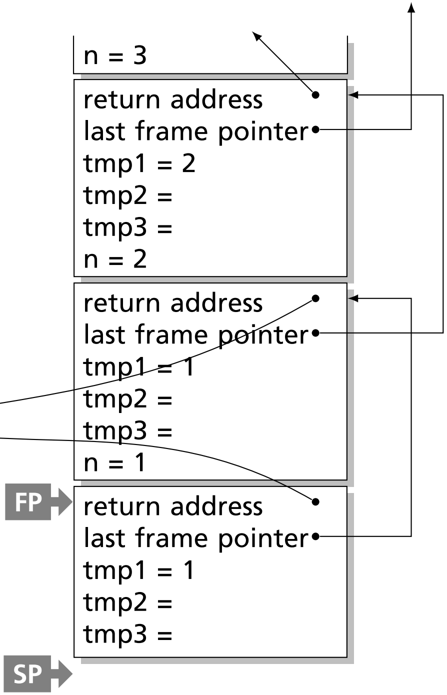
Executing fib(3)

```
int fib(int n) {
    int tmp1, tmp2, tmp3;
    tmp1 = n < 2;
    if (!tmp1) goto L1;
    return 1;
L1: tmp1 = n - 1;
    tmp2 = fib(tmp1);
L2: tmp1 = n - 2;
    tmp3 = fib(tmp1);
L3: tmp1 = tmp2 + tmp3;
    return tmp1;
}
```



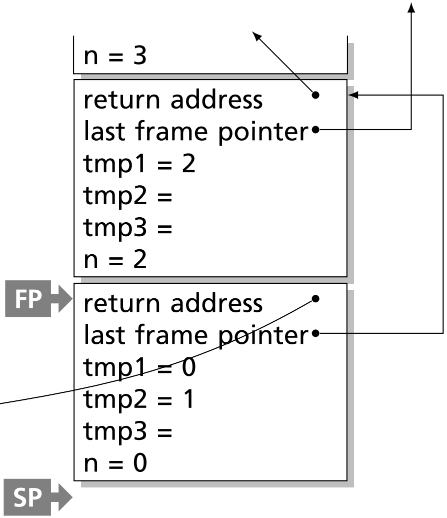
Executing fib(3)

```
int fib(int n) {
    int tmp1, tmp2, tmp3;
    tmp1 = n < 2;
    if (!tmp1) goto L1;
    return 1;
L1: tmp1 = n - 1;
    tmp2 = fib(tmp1);
L2: tmp1 = n - 2;
    tmp3 = fib(tmp1);
L3: tmp1 = tmp2 + tmp3;
    return tmp1;
}
```



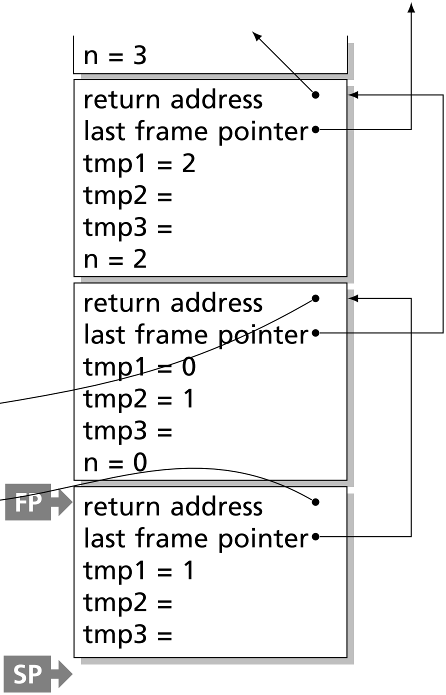
Executing fib(3)

```
int fib(int n) {
    int tmp1, tmp2, tmp3;
    tmp1 = n < 2;
    if (!tmp1) goto L1;
    return 1;
L1: tmp1 = n - 1;
    tmp2 = fib(tmp1);
L2: tmp1 = n - 2;
    tmp3 = fib(tmp1);
L3: tmp1 = tmp2 + tmp3;
    return tmp1;
}
```



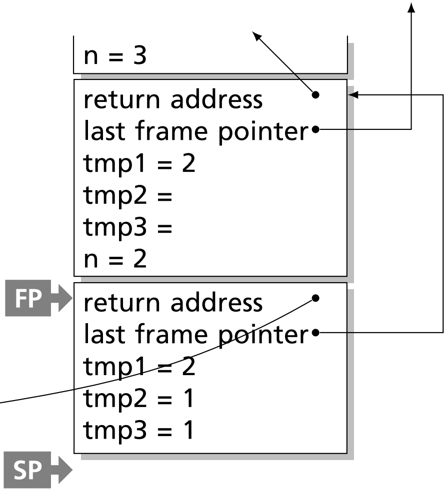
Executing fib(3)

```
int fib(int n) {
    int tmp1, tmp2, tmp3;
    tmp1 = n < 2;
    if (!tmp1) goto L1;
    return 1;
L1: tmp1 = n - 1;
    tmp2 = fib(tmp1);
L2: tmp1 = n - 2;
    tmp3 = fib(tmp1);
L3: tmp1 = tmp2 + tmp3;
    return tmp1;
}
```

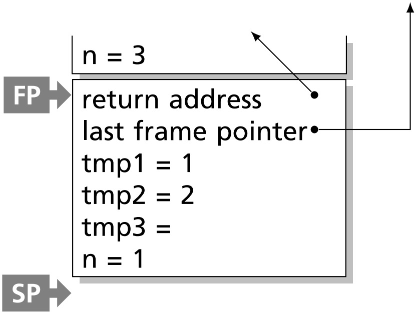


Executing fib(3)

```
int fib(int n) {
    int tmp1, tmp2, tmp3;
    tmp1 = n < 2;
    if (!tmp1) goto L1;
    return 1;
L1: tmp1 = n - 1;
    tmp2 = fib(tmp1);
L2: tmp1 = n - 2;
    tmp3 = fib(tmp1);
L3: tmp1 = tmp2 + tmp3;
    return tmp1;
}
```



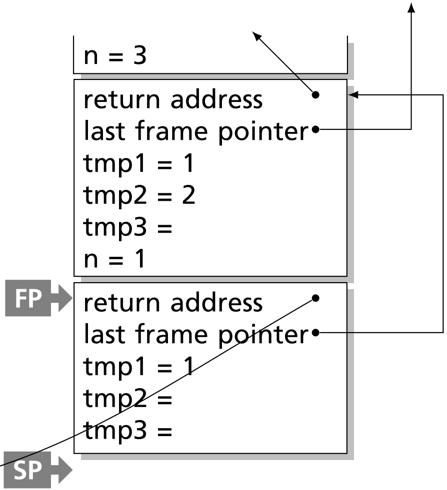
Executing fib(3)




```
int fib(int n) {
    int tmp1, tmp2, tmp3;
    tmp1 = n < 2;
    if (!tmp1) goto L1;
    return 1;
L1: tmp1 = n - 1;
    tmp2 = fib(tmp1);
L2: tmp1 = n - 2;
    tmp3 = fib(tmp1);
L3: tmp1 = tmp2 + tmp3;
    return tmp1;
}
```

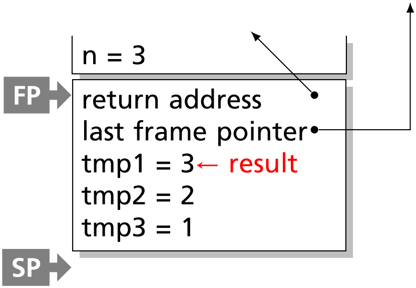
Executing fib(3)

```
int fib(int n) {
    int tmp1, tmp2, tmp3;
    tmp1 = n < 2;
    if (!tmp1) goto L1;
    return 1;
L1: tmp1 = n - 1;
    tmp2 = fib(tmp1);
L2: tmp1 = n - 2;
    tmp3 = fib(tmp1);
L3: tmp1 = tmp2 + tmp3;
    return tmp1;
}
```



Executing fib(3)

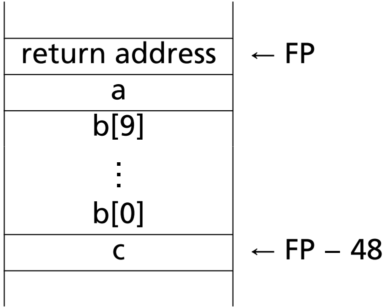
```
int fib(int n) {
    int tmp1, tmp2, tmp3;
    tmp1 = n < 2;
    if (!tmp1) goto L1;
    return 1;
L1: tmp1 = n - 1;
    tmp2 = fib(tmp1);
L2: tmp1 = n - 2;
    tmp3 = fib(tmp1);
L3: tmp1 = tmp2 + tmp3;
    return tmp1;
}
```



Allocating Fixed-Size Arrays

Local arrays with fixed size are easy to stack.

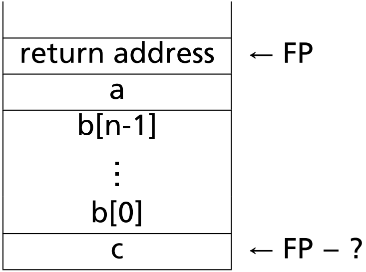
```
void foo()
{
  int a;
  int b[10];
  int c;
}
```



Allocating Variable-Sized Arrays

Variable-sized local arrays aren't as easy.

```
void foo(int n)
{
  int a;
  int b[n];
  int c;
}
```

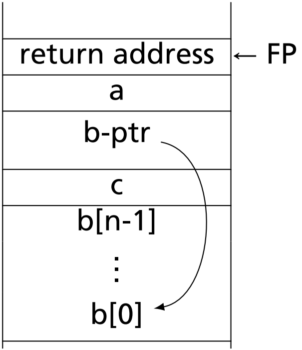


Doesn't work: generated code expects a fixed offset for c.
Even worse for multi-dimensional arrays.

Allocating Variable-Sized Arrays

As always:
add a level of indirection

```
void foo(int n)
{
  int a;
  int b[n];
  int c;
}
```



Variables remain constant offset from frame pointer.

Nesting Function Definitions

```
let articles words =

  let report w =

    let count = List.length
      (List.filter ((=) w) words)
    in w ^ ": " ^
       string_of_int count

  in String.concat ", "
    (List.map report ["a"; "the"])

in articles
    ["the"; "plt"; "class"; "is";
     "a"; "pain"; "in";
     "the"; "butt"]
```

```
let count words w = List.length
  (List.filter ((=) w) words) in

let report words w = w ^ ": " ^
  string_of_int (count words w) in

let articles words =
  String.concat ", "
    (List.map (report words)
     ["a"; "the"]) in

articles
    ["the"; "plt"; "class"; "is";
     "a"; "pain"; "in";
     "the"; "butt"]
```

Produces “a: 1, the: 2”

Implementing Nested Functions with Access Links

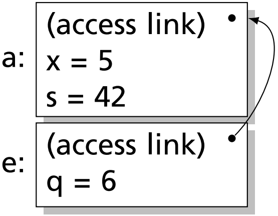
a:

(access link) •
x = 5
s = 42

```
let a x s =
  let b y =
    let c z = z + s in
    let d w = c (w+1) in
    d (y+1) in (* b *)
  let e q = b (q+1) in
e (x+1) (* a *)
```

What does “a 5 42” give?

Implementing Nested Functions with Access Links



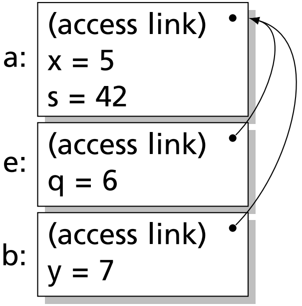

```
let a x s =
  let b y =
    let c z = z + s in
    let d w = c (w+1) in
    d (y+1) in (* b *)
  let e q = b (q+1) in
e (x+1) (* a *)
```

What does “a 5 42” give?

Implementing Nested Functions with Access Links

```
let a x s =
  let b y =
    let c z = z + s in
    let d w = c (w+1) in
    d (y+1) in (* b *)
  let e q = b (q+1) in
e (x+1) (* a *)
```

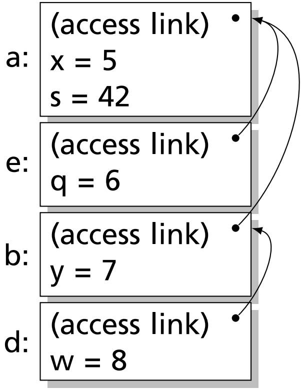
What does “a 5 42” give?



Implementing Nested Functions with Access Links

```
let a x s =
  let b y =
    let c z = z + s in
    let d w = c (w+1) in
    d (y+1) in (* b *)
  let e q = b (q+1) in
e (x+1) (* a *)
```

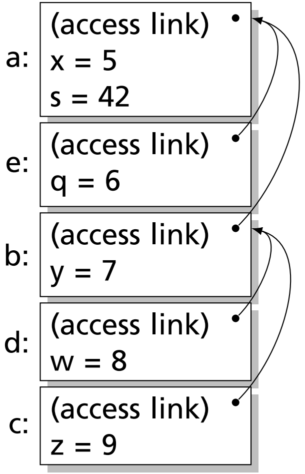
What does “a 5 42” give?



Implementing Nested Functions with Access Links

```
let a x s =
  let b y =
    let c z = z + s in
    let d w = c (w+1) in
    d (y+1) in (* b *)
  let e q = b (q+1) in
e (x+1) (* a *)
```

What does “a 5 42” give?

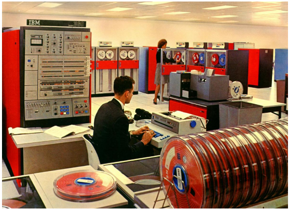


In-Memory Layout Issues

Layout of Records and Unions

Modern processors have byte-addressable memory.

0
1
2
3



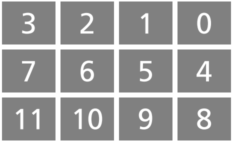
The IBM 360 (c. 1964) helped to popularize byte-addressable memory.

Many data types (integers, addresses, floating-point numbers) are wider than a byte.

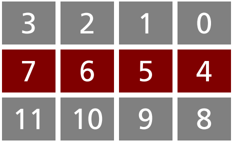
16-bit integer:			1	0
32-bit integer:	3	2	1	0

Layout of Records and Unions

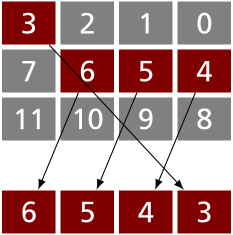
Modern memory systems read data in 32-, 64-, or 128-bit chunks:



Reading an aligned 32-bit value is fast: a single operation.



It is harder to read an unaligned value: two reads plus shifting



SPARC and ARM prohibit unaligned accesses

MIPS has special unaligned load/store instructions

x86, 68k run more slowly with unaligned accesses

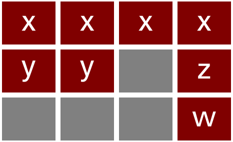
Padding

To avoid unaligned accesses, the C compiler pads the layout of unions and records.

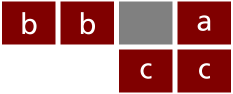
Rules:

- Each n-byte object must start on a multiple of n bytes (no unaligned accesses).
- Any object containing an n-byte object must be of size mn for some integer m (aligned even when arrayed).

```c
struct padded {
  int x;    /* 4 bytes */
  char z;   /* 1 byte  */
  short y;  /* 2 bytes */
  char w;   /* 1 byte  */
};
```




```c
struct padded {
  char a;   /* 1 byte  */
  short b;  /* 2 bytes */
  short c;  /* 2 bytes */
};
```

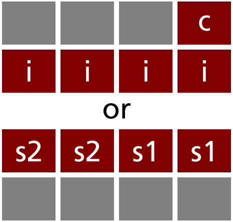


Unions

A C *struct* has a separate space for each field; a C *union* shares one space among all fields

```
union intchar {
  int i;    /* 4 bytes */
  char c;   /* 1 byte  */
};
```

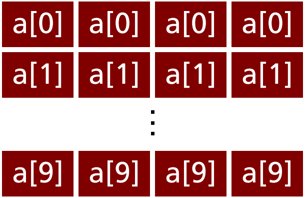

```
union twostructs {
  struct {
    char c;    /* 1 byte */
    int i;     /* 4 bytes */
  } a;
  struct {
    short s1; /* 2 bytes */
    short s2; /* 2 bytes */
  } b;
};
```



Arrays

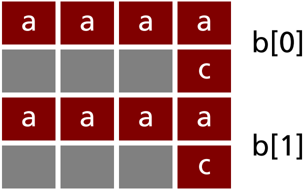
Basic policy in C: an array is just one object after another in memory.

```
int a[10];
```



This is why you need padding at the end of *structs*.

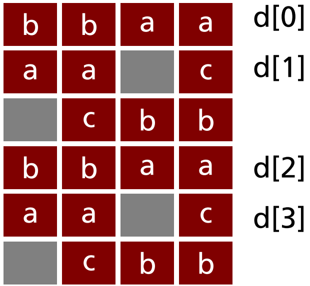
```
struct {
  int a;
  char c;
} b[2];
```



Arrays and Aggregate types

The largest primitive type dictates the alignment

```
struct {
  short a;
  short b;
  char c;
} d[4];
```



Arrays of Arrays

```
char a[4];
```

a[3]	a[2]	a[1]	a[0]

```
char a[3][4];
```

a[0][3]	a[0][2]	a[0][1]	a[0][0]	a[0]
a[1][3]	a[1][2]	a[1][1]	a[1][0]	a[1]
a[2][3]	a[2][2]	a[2][1]	a[2][0]	a[2]

The Heap

Heap-Allocated Storage

Static works when you know everything beforehand and always need it.

Stack enables, but also requires, recursive behavior.

A *heap* is a region of memory where blocks can be allocated and deallocated in any order.

(These heaps are different than those in, e.g., heapsort)

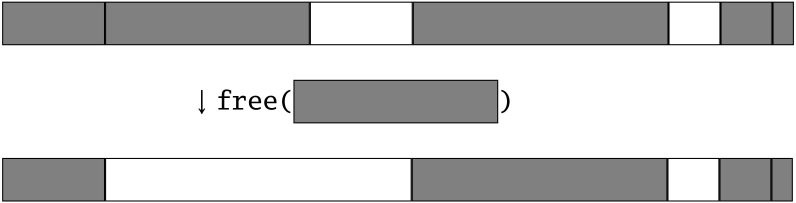
Dynamic Storage Allocation in C

```c
struct point {
   int x, y;
};

int play_with_points(int n)
{
  int i;
  struct point *points;

  points = malloc(n * sizeof(struct point));

  for ( i = 0 ; i < n ; i++ ) {
    points[i].x = random();
    points[i].y = random();
  }

  /* do something with the array */

  free(points);
}
```

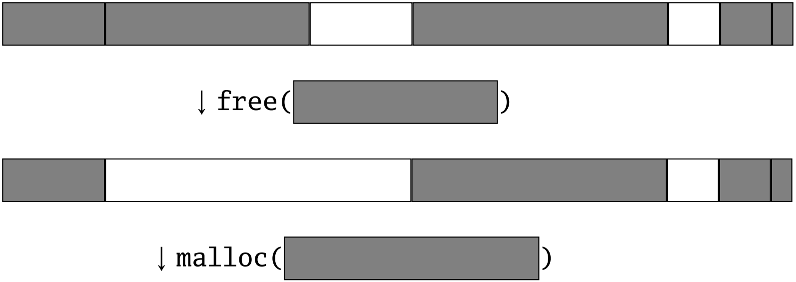
Dynamic Storage Allocation

Dynamic Storage Allocation

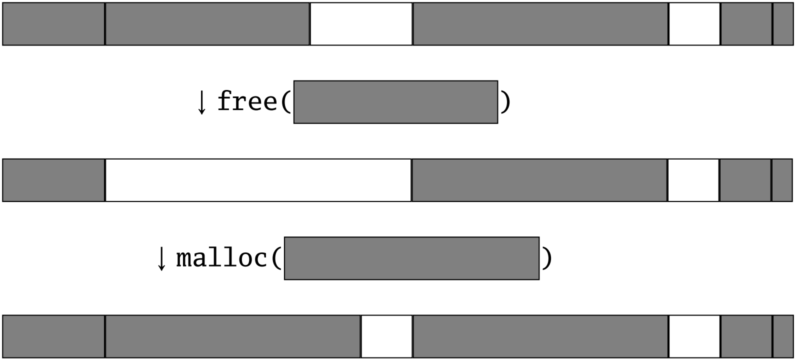
Dynamic Storage Allocation



Dynamic Storage Allocation



Dynamic Storage Allocation



Dynamic Storage Allocation

Rules:

- Each allocated block contiguous (no holes)
- Blocks stay fixed once allocated

`malloc()`

- Find an area large enough for requested block
- Mark memory as allocated

`free()`

- Mark the block as unallocated

Simple Dynamic Storage Allocation

Maintaining information about free memory

Simplest: Linked list

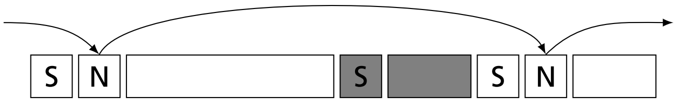
The algorithm for locating a suitable block

Simplest: First-fit

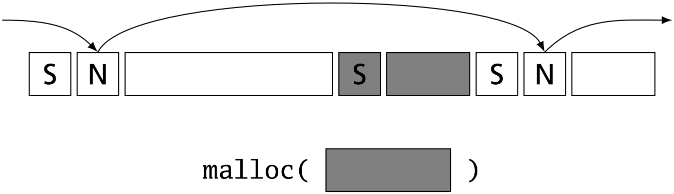
The algorithm for freeing an allocated block

Simplest: Coalesce adjacent free blocks

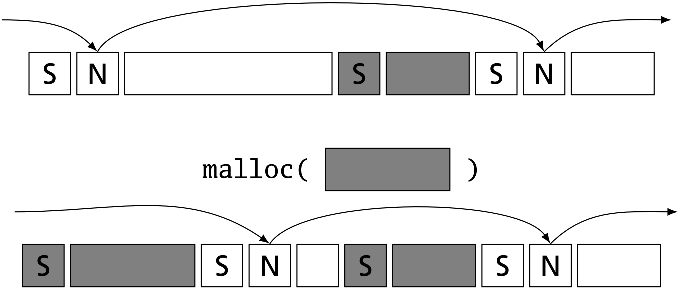
Simple Dynamic Storage Allocation



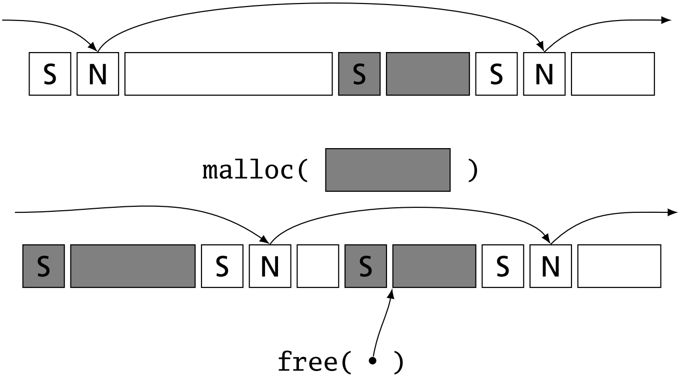
Simple Dynamic Storage Allocation



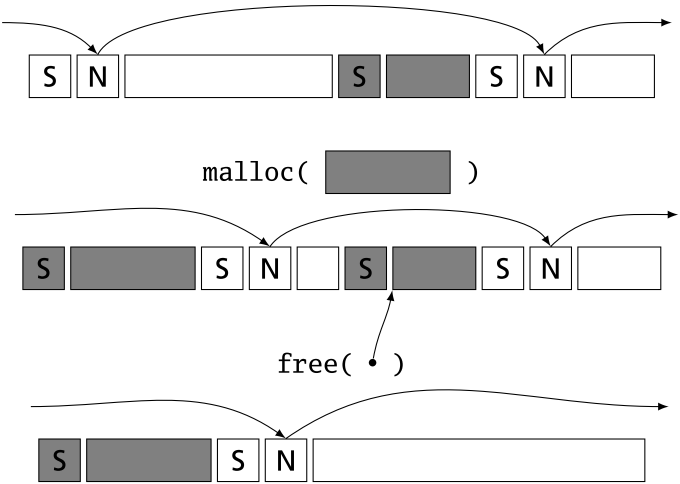
Simple Dynamic Storage Allocation



Simple Dynamic Storage Allocation



Simple Dynamic Storage Allocation



Dynamic Storage Allocation

Many, many other approaches.

Other “fit” algorithms

Segregation of objects by size

More clever data structures

Heap Variants

Memory pools: Differently-managed heap areas

Stack-based pool: only free whole pool at once

- Nice for build-once data structures

Single-size-object pool:

- Fit, allocation, etc. much faster
- Good for object-oriented programs

Fragmentation

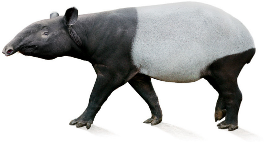
`malloc(`  `)` seven times give

`free()` four times gives

`malloc(`  `)` ?

Need more memory; can’t use fragmented memory.

Zebra

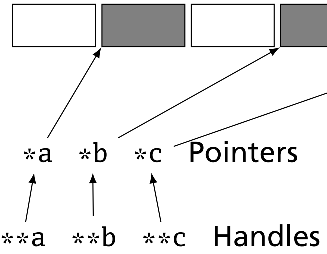


Tapir

Fragmentation and Handles

Standard CS solution: Add another layer of indirection.

Always reference memory through “handles.”

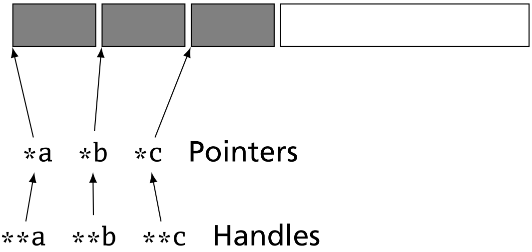


The original Macintosh did this to save memory.

Fragmentation and Handles

Standard CS solution: Add another layer of indirection.

Always reference memory through “handles.”



The original Macintosh did this to save memory.

Automatic Garbage Collection

Automatic Garbage Collection

Entrust the runtime system with freeing heap objects

Now common: Java, C#, Javascript, Python, Ruby, OCaml and most functional languages

Advantages

Much easier for the programmer

Greatly improves reliability: no memory leaks, double-freeing, or other memory management errors

Disadvantages

Slower, sometimes unpredictably so

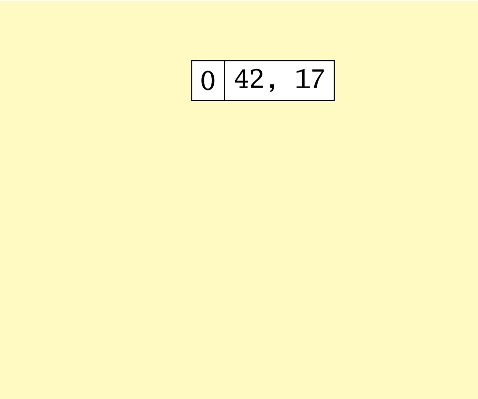
May consume more memory

Reference Counting

What and when to free?

- Maintain count of references to each object
- Free when count reaches zero

```
let a = (42, 17) in
let b = [a;a] in
let c = (1,2)::b in
b
```

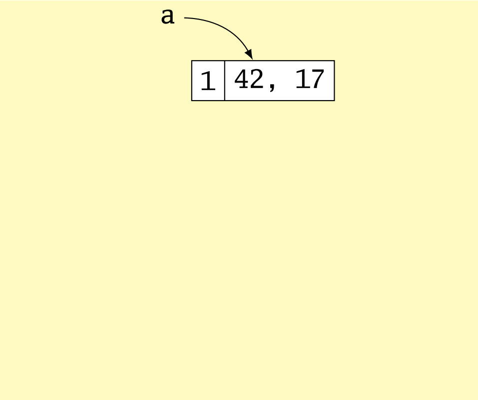


Reference Counting

What and when to free?

- Maintain count of references to each object
- Free when count reaches zero

```
let a = (42, 17) in
let b = [a;a] in
let c = (1,2)::b in
b
```

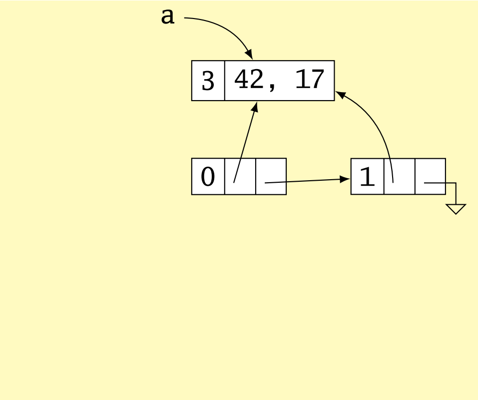


Reference Counting

What and when to free?

- Maintain count of references to each object
- Free when count reaches zero

```
let a = (42, 17) in
let b = [a;a] in
let c = (1,2)::b in
b
```

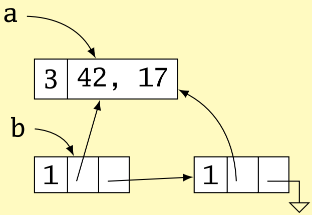


Reference Counting

What and when to free?

- Maintain count of references to each object
- Free when count reaches zero

```
let a = (42, 17) in
let b = [a;a] in
let c = (1,2)::b in
b
```

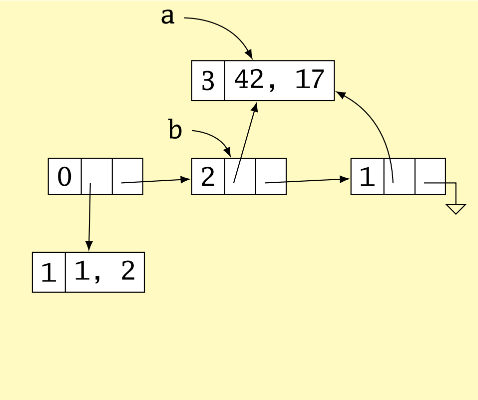


Reference Counting

What and when to free?

- Maintain count of references to each object
- Free when count reaches zero

```
let a = (42, 17) in
let b = [a;a] in
let c = (1,2)::b in
b
```

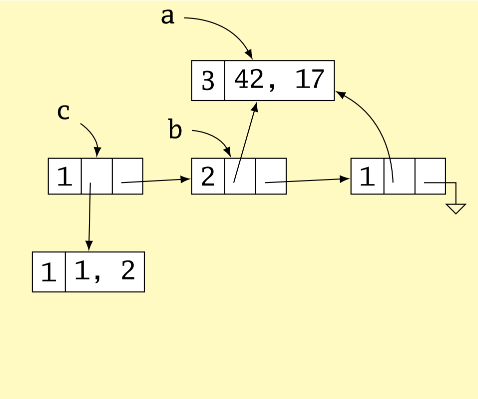


Reference Counting

What and when to free?

- Maintain count of references to each object
- Free when count reaches zero

```
let a = (42, 17) in
let b = [a;a] in
let c = (1,2)::b in
b
```

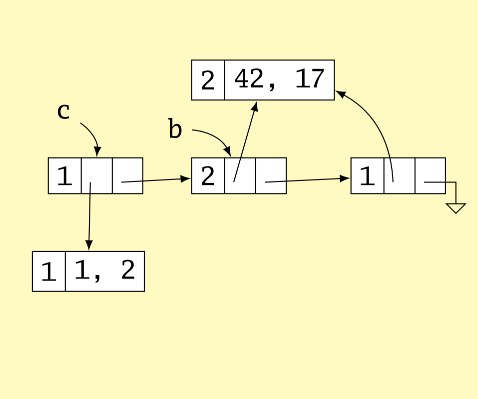


Reference Counting

What and when to free?

- Maintain count of references to each object
- Free when count reaches zero

```
let a = (42, 17) in
let b = [a;a] in
let c = (1,2)::b in
b
```

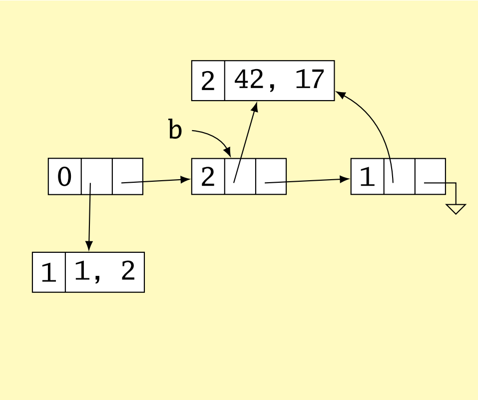


Reference Counting

What and when to free?

- Maintain count of references to each object
- Free when count reaches zero

```
let a = (42, 17) in
let b = [a;a] in
let c = (1,2)::b in
b
```

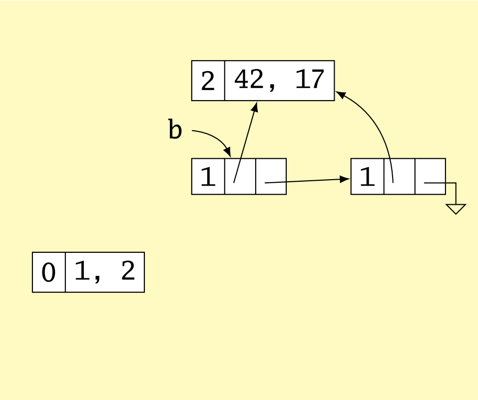


Reference Counting

What and when to free?

- Maintain count of references to each object
- Free when count reaches zero

```
let a = (42, 17) in
let b = [a;a] in
let c = (1,2)::b in
b
```

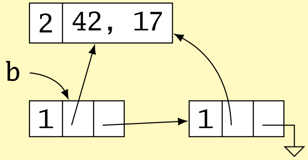


Reference Counting

What and when to free?

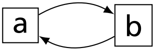
- Maintain count of references to each object
- Free when count reaches zero

```
let a = (42, 17) in
let b = [a;a] in
let c = (1,2)::b in
b
```



Issues with Reference Counting

Circular structures defy reference counting:



Neither is reachable, yet both have non-zero reference counts.

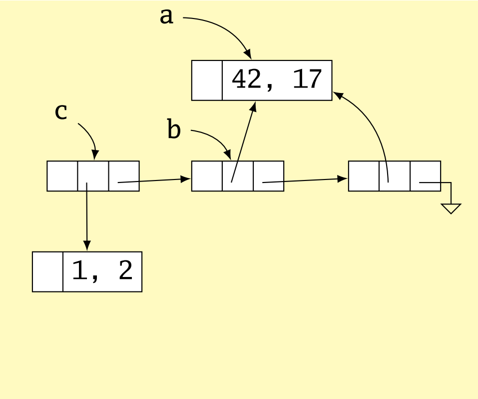
High overhead (must update counts constantly), although incremental

Mark-and-Sweep

What and when to free?

- Stop-the-world algorithm invoked when memory full
- Breadth-first-search marks all reachable memory
- All unmarked items freed

```
let a = (42, 17) in
let b = [a;a] in
let c = (1,2)::b in
b
```

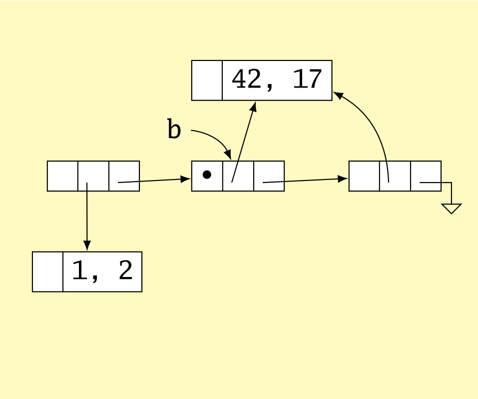


Mark-and-Sweep

What and when to free?

- Stop-the-world algorithm invoked when memory full
- Breadth-first-search marks all reachable memory
- All unmarked items freed

```
let a = (42, 17) in
let b = [a;a] in
let c = (1,2)::b in
b
```

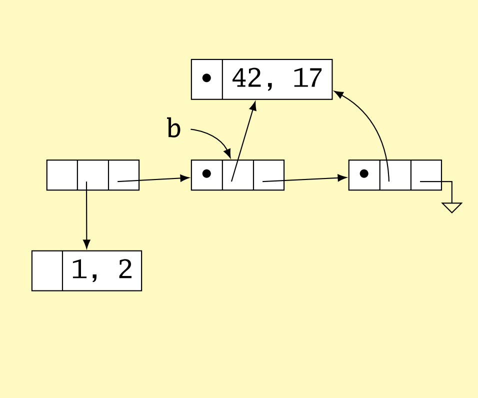


Mark-and-Sweep

What and when to free?

- Stop-the-world algorithm invoked when memory full
- Breadth-first-search marks all reachable memory
- All unmarked items freed

```
let a = (42, 17) in
let b = [a;a] in
let c = (1,2)::b in
b
```

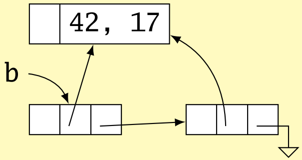


Mark-and-Sweep

What and when to free?

- Stop-the-world algorithm invoked when memory full
- Breadth-first-search marks all reachable memory
- All unmarked items freed

```
let a = (42, 17) in
let b = [a;a] in
let c = (1,2)::b in
b
```



Mark-and-Sweep

Mark-and-sweep is faster overall; may induce big pauses

Mark-and-compact variant also moves or copies reachable objects to eliminate fragmentation

Incremental garbage collectors try to avoid doing everything at once

Most objects die young; generational garbage collectors segregate heap objects by age

Parallel garbage collection tricky

Real-time garbage collection tricky

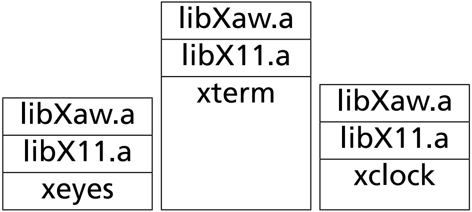
Shared Libraries and Dynamic Linking

Shared Libraries and Dynamic Linking

The 1980s GUI/WIMP revolution required many large libraries (the Athena widgets, Motif, etc.)

Under a *static linking* model, each executable using a library gets a copy of that library's code.

Address 0:

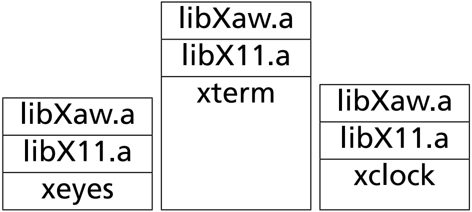


Shared Libraries and Dynamic Linking

The 1980s GUI/WIMP revolution required many large libraries (the Athena widgets, Motif, etc.)

Under a *static linking* model, each executable using a library gets a copy of that library's code.

Address 0:



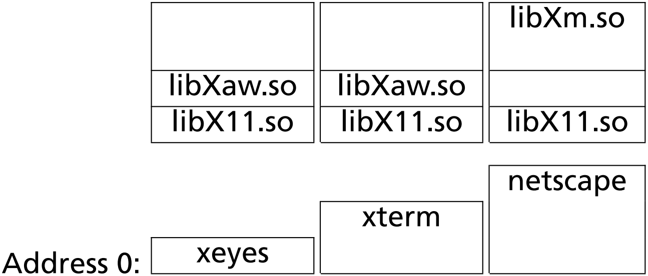
Wasteful: running many GUI programs at once fills memory with nearly identical copies of each library.

Something had to be done: another level of indirection.

Shared Libraries: First Attempt

Most code makes assumptions about its location.

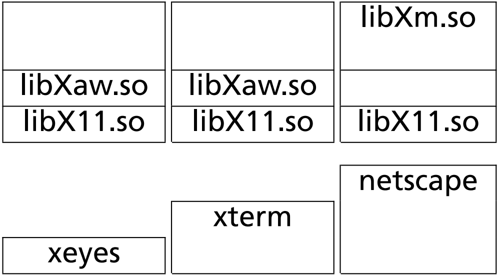
First solution (early Unix System V R3) required each shared library to be located at a unique address:



Shared Libraries: First Attempt

Most code makes assumptions about its location.

First solution (early Unix System V R3) required each shared library to be located at a unique address:

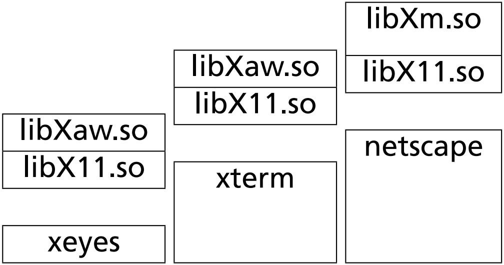
Address 0: 

Obvious disadvantage: must ensure each new shared library located at a new address.

Works fine if there are only a few libraries; tended to discourage their use.

Shared Libraries

Problem fundamentally is that each program may need to see different libraries each at a different address.



Position-Independent Code

Solution: Require the code for libraries to be position-independent. Make it so they can run anywhere in memory.

As always, add another level of indirection:

- All branching is PC-relative
- All data must be addressed relative to a base register.
- All branching to and from this code must go through a jump table.

Position-Independent Code for bar()

Normal unlinked code

```
save  %sp, -112, %sp
sethi  %hi(0), %o0
  R_SPARC_HI22 .bss
mov  %o0, %o0
  R_SPARC_LO10 .bss
sethi  %hi(0), %o1
  R_SPARC_HI22 a
mov  %o1, %o1
  R_SPARC_LO10 a
call  14
  R_SPARC_WDISP30 strcpy
nop
sethi  %hi(0), %o0
  R_SPARC_HI22 .bss
mov  %o0, %o0
  R_SPARC_LO10 .bss
call  24
  R_SPARC_WDISP30 baz
nop
ret
restore
```

gcc -fpic -shared

```
save  %sp, -112, %sp
sethi  %hi(0x10000), %l7
call  8e0   ! add PC to %l7
add  %l7, 0x198, %l7
ld  [ %l7 + 0x20 ], %o0
ld  [ %l7 + 0x24 ], %o1
```

Actually just a stub

```
call  10a24 ! strcpy

nop
ld  [ %l7 + 0x20 ], %o0
```

call is PC-relative

```
call  10a3c ! baz

nop
ret
restore
```

Objects and Inheritance

Single Inheritance

Simple: Add new fields to end of the object

Fields in base class always at same offset in derived class (compiler never reorders)

Consequence: Derived classes can never remove fields

C++

```
class Shape {
  double x, y;
};

class Box : Shape {
  double h, w;
};


class Circle : Shape {
  double r;
};
```

Equivalent C

```
struct Shape {
  double x, y;
};

struct Box {
  double x, y;
  double h, w;
};

struct Circle {
  double x, y;
  double r;
};
```

Virtual Functions

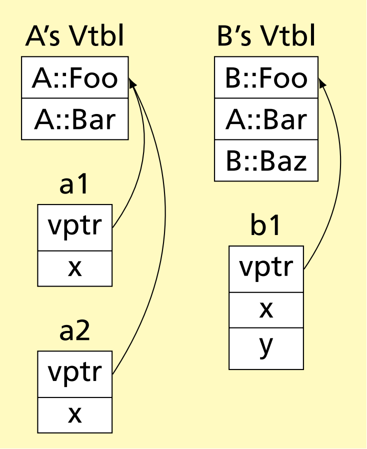
```
class Shape {
  virtual void draw(); // Invoked by object's run-time class
};                     // not its compile-time type.

class Line : public Shape {
  void draw();
}

class Arc : public Shape {
  void draw();
};

Shape *s[10];
s[0] = new Line;
s[1] = new Arc;
s[0]->draw();   // Invoke Line::draw()
s[1]->draw();   // Invoke Arc::draw()
```

Virtual Functions

Trick: add to each object a pointer to the virtual table for its type, filled with pointers to the virtual functions.

Like the objects themselves, the virtual table for each derived type begins identically.

```
struct A {
  int x;
  virtual void Foo();
  virtual void Bar();
};

struct B : A {
  int y;
  virtual void Foo();
  virtual void Baz();
};

A a1;
A a2;
B b1;
```



Exceptions

C++'s Exceptions

```
struct Except {} ex;  // This struct functions as an exception

void top(void) {
  try {
    child();
  } catch (Except e) { // throw sends control here
    printf("oops\n");
  }
}

void child() {
  child2();
}

void child2() {
  throw ex; // Pass control up to the catch block
}
```

C's setjmp/longjmp: Idiosyncratic Exceptions

```
#include <setjmp.h>

jmp_buf closure;          /* return address, stack & frame ptrs. */

void top(void) {
  switch ( setjmp(closure) ) { /* normal: store closure, return 0 */
                               /* longjmp jumps here, returns 1 */


  case 0: child();             /* unexceptional case */
          break;

  case 1: break;               /* longjmp( ,1) called */
  }
}

void child() {
  child2();
}

void child2() {
  longjmp(closure, 1);
}
```

Implementing Exceptions

One way: maintain a stack of exception handlers

```
try {

  child();


} catch (Ex e) {
  foo();
}

void child() {
  child2();
}

void child2() {
  throw ex;
}
```

```
  push(Ex, Handler); // Push handler on stack

  child();
  pop();             // Normal termination
  goto Exit;         // Jump over "catch"
Handler:
  foo();             // Body of "catch"
Exit:

void child() {
  child2();
}

void child2() {
  throw(ex);       // Unroll stack; find handler
}
```

Incurs overhead, even when no exceptions thrown

Implementing Exceptions with Tables

Q: When an exception is *thrown*, where was the last *try*?

A: Consult a table: relevant handler or “pop” for every PC

```
1  void foo() {
2
3    try {
4      bar();
5    } catch (Ex1 e) {
6      a();
7    }
8  }
9
10 void bar() {
11   baz();
12 }
13
14 void baz() {
15
16   try {
17     throw ex1;
18   } catch (Ex2 e) {
19     b();
20   }
21 }
```

Lines	Action
1–2	Pop stack
3–5	Handler @ 5 for Ex1
6–15	Pop stack
16–18	Handler @ 18 for Ex2
19–21	Pop stack