CSC 533: Programming Languages

Spring 2019

Data types

- primitive types (integer, float, boolean, char, pointer)
- heap management
- reference counts vs. garbage collection
- partition & copy vs. mark & sweep

We will focus on C, C++, and Java as example languages

Primitive types: integer

languages often provide several sizes/ranges

in C/C++/Java

`short`	(2 bytes in Java)
`int`	(4 bytes in Java)
`long`	(8 bytes in Java)

absolute sizes are implementation dependent in C/C++

TRADEOFFS?

- Java has a `byte` type (1 byte)
- in C/C++, `char` is considered an integer type
- most languages use 2's complement notation for negatives

```
1 = 00…000001        -1 = 11…111111
2 = 00…000010        -2 = 11…111110
3 = 00…000011        -3 = 11…111101
```

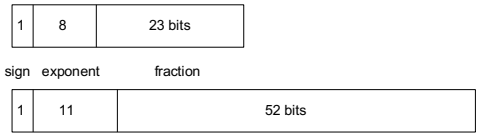
Primitive types: floating-point

again, languages often provide several sizes/ranges

in C/C++/Java `float` (4 bytes in Java)
`double` (8 bytes in Java)

C/C++ also have a `long double` type

- historically, floating-points have been stored in a variety of formats
  same basic components: sign, fraction, exponent
- in 1985, IEEE floating-point formats were standardized



(sign)fraction x $2^{exponent}$

special bit patterns represent:
- `infinity`
- `NaN`

other number types: decimal, fixed-point, rational, …

3

Primitive types: Boolean

introduced in ALGOL 60

C does not have a boolean type, conditionals use zero (false) and nonzero (true)

C++ has `bool` type
- really just *syntactic sugar*, automatic conversion between `int` and `bool`

Java has `boolean` type
- no conversions between `int` and `boolean`

implementing Booleans

- could use a single bit, but not usually accessible
- use smallest easily-addressable unit (e.g., byte)

4

Primitive types: character

stored as numeric codes, e.g., ASCII (C/C++) or UNICODE (Java)

in C/C++, `char` is an integer type

- can apply integer operations, mix with integer values

```
char ch = 'A';
ch = ch + 1;
cout << ch << endl
```

```
char ch = '8';
int d = ch - '0';
cout << d << endl;
```

in Java, `char` to `int` conversion is automatic

- but must explicitly cast `int` to `char`

```
char next = (char)(ch + 1);
```

5

Primitive types: pointer

a pointer is nothing more than an address (i.e., an integer)

useful for:

- dynamic memory management (allocate, dereference, deallocate)
- indirect addressing (point to an address, dereference)

PL/I was the first language to provide pointers

- pointers were not typed, could point to any object

  ➔ no static type checking for pointers

ALGOL 68 refined pointers to a specific type

in many languages, pointers are limited to dynamic memory management
e.g., Pascal, Ada, Java, …

6

Primitive types: pointer (cont.)

C/C++ allows for low-level memory access using pointers

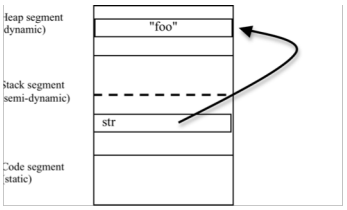
`*` dereferencing operator `&` address-of operator

```
int x = 6;
int * ptr1 = &x;
int * ptr2 = ptr1;
*ptr2 = 3;
```

in C/C++, the 0 (NULL) address is reserved, attempted access → ERROR

Java does not provide explicit pointers, but every object is really a pointer

```
String str = "foo";
```



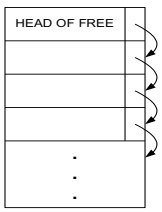
7

Heap management

pointers access memory locations from the heap
(a dynamically allocated storage area)

the heap is divided into equal-size cells, each with a pointer

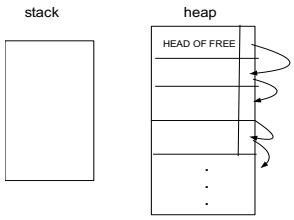
- the pointer fields are used initially to organize the heap as a linked list



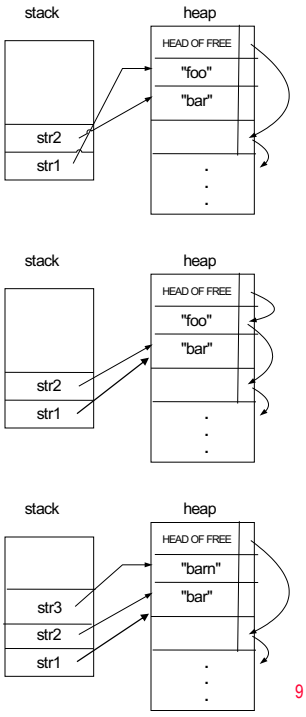
- keep pointer to head of free list
- to allocate space, take from front of free list
- to deallocate, put back at front

8

Heap example




```
String str1 = "foo";
String str2 = "bar";

/* CHECKPOINT 1 */

str1 = str2;

/* CHECKPOINT 2 */

String str3 = str1+"n";

/* CHECKPOINT 3 */
```



9

Pointer problems

returning memory to the free list is easy, but when do you do it?

dangling reference: memory is deallocated, but still have a pointer to it

```
int * Foo() {
   int x = 5;
   return &x;
}
```

a problem in C/C++, since the & operator allows access to stack memory that has already been reclaimed

not a problem in Java since no equivalent to the & operator

garbage reference: pointer is destroyed, but memory has not been deallocated

```
void Bar() {
   Date today = new Date();
    …
}
```

a problem in both C/C++ and Java

when `today`'s lifetime ends, its dynamic memory is inaccessible (in C/C++, must explicitly deallocate dynamic memory w/ `delete`)

would like to automatically and safely reclaim heap memory

2 common techniques: reference counts, garbage collection

10

Reference counts

along with each heap element, store a reference count

- indicates the number of pointers to the heap element

- when space is allocated, its reference count is set to 1
- each time a new pointer is set to it, increment the reference count
- each time a pointer is lost, decrement the reference count

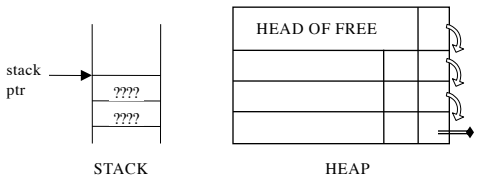
provides a simple method for avoiding garbage & dangling references

- if result of an operation leaves reference count at 0, reclaim memory
- can even double check explicit deallocations

11

Reference counts example

```
String str1 = "foo";
String str2 = "bar";

/* CHECKPOINT 1 */

str1 = str2;

/* CHECKPOINT 2 */

if (str1.equals(str2)) {
  String temp = "biz";
  str2= temp;

  /* CHECKPOINT 3 */
}

String str3 = "baz";

/* CHECKPOINT 4 */
```



12

Reference counts (cont.)

unfortunately, reference counts are very costly

- must update & check reference counts for each assignment, end of lifetime

```
String str1;
String str2;
…
str1 = str2;
```

➔

1) dereference `str1`, decrement count
2) if count = 0, deallocate
3) copy `str1` reference to `str2`
4) dereference `str1`, increment count

reference counts are popular in parallel programming

- work is spread evenly

13

Garbage collection

approach: allow garbage to accumulate, only collect if out of space

as program executes, no reclamation of memory (thus, no cost)
when out of memory, take the time to collect garbage (costly but rare)

e.g., toothpaste tube analogy

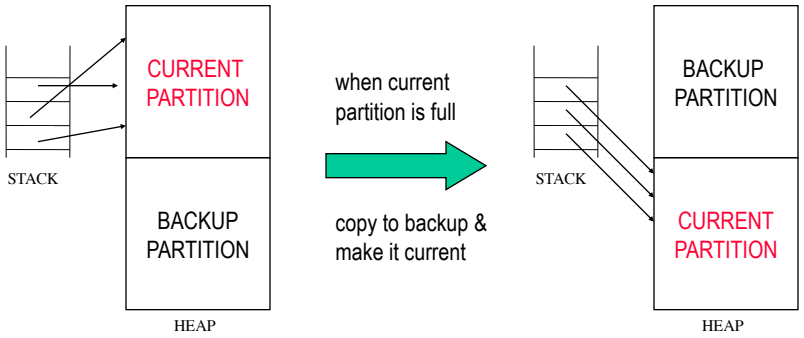
2 common approaches to garbage collection

1. Partition & Copy
2. Mark & Sweep

14

Partition & Copy approach

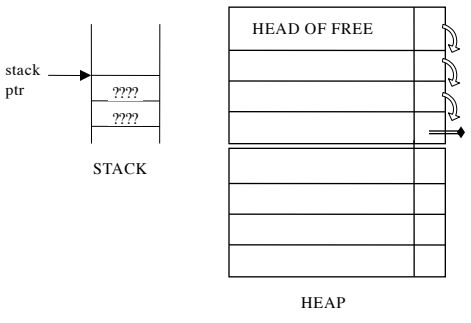
1. divide the memory space into 2 partitions: current + backup
2. when the current partition is full,
   a. sweep through all active objects (from the stack)
   b. copy each active object to the backup partition (contiguously)
   c. when done, make that the current partition



15

Partition & Copy example

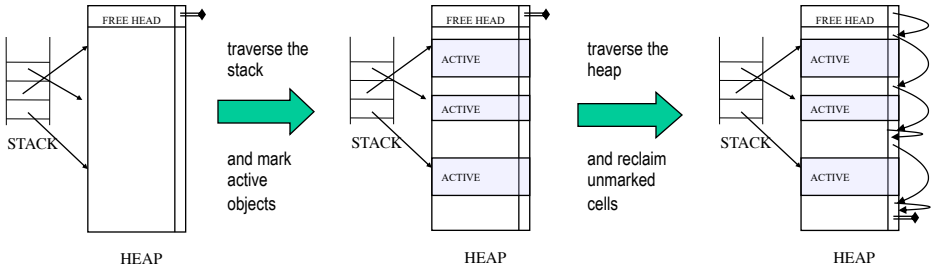
```
String str1= "foo";
String str2= "bar";

/* CHECKPOINT 1 */

str1 = str2;

/* CHECKPOINT 2 */

if (str1.equals(str2)) {
  String temp = "biz";
  str2 = temp;

  /* CHECKPOINT 3 */
}

String str3 = "baz";

/* CHECKPOINT 4 */
```



16

Mark & Sweep approach

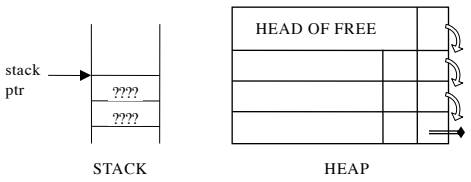
1. mark all active objects
   a. sweep through all active objects (from the stack)
   b. mark each memory cell associated with an active object
2. sweep through the heap and reclaim unmarked cells
   a. traverse the heap sequentially
   b. add each unmarked cell to the FREE list



17

Mark & Sweep example

```
String str1= "foo";
String str2= "bar";

/* CHECKPOINT 1 */

str1 = str2;

/* CHECKPOINT 2 */

if (str1.equals(str2)) {
  String temp = "biz";
  str2 = temp;

  /* CHECKPOINT 3 */
}

String str3 = "baz";

/* CHECKPOINT 4 */
```

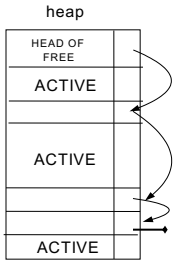


18

Mark & Sweep & Compactify

note: not all memory allocations are the same size

- C/C++/Java: double bigger than float, array elements must be contiguous, …



as memory is allocated & deallocated, fragmentation occurs

e.g., suppose wish to allocate a 3 element array
previous allocations/deallocations have left 3 free cells, but not contiguously
→ must garbage collect (even though free space exists)

using Partition & Copy, not a big problem

- simply copy active objects to other partition – this automatically coalesces gaps

using Mark & Sweep, must add another pass to defragment the space

- once active objects have been identified, must shift them in memory to remove gaps
- COSTLY!

19

Partition & Copy vs. Mark & Sweep & Compactify

Partition & Copy

- wastes memory by maintaining the backup partition
- but quick (especially if few active objects) and avoids fragmentation

Mark & Sweep & Compactify

- able to use the entire memory space for active objects
- but slow (2 complete passes through heap to reclaim and compactify)

Java takes a hybrid approach to provide automatic garbage collection

- memory is divided into two types: new objects and old objects
- the new objects partition is optimized for objects with short lifetimes
  garbage collection happens relatively frequently
  uses Partition & Copy, since it is expected that few active objects will remain
- eventually, persistent new objects are moved to the old objects partition
  garbage collections happens relatively infrequently
  uses Mark & Sweep & Compactify, since many active objects will persist

20