CSC 533: Programming Languages

Spring 2012

Data types
- primitive types (integer, float, boolean, char, pointer)
- heap management, garbage collection
- complex data types (string, enum, subrange, array, record, …)
- expressions and assignments

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

1

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
  ```

2

1

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

1	8	23 bits

sign exponent   fraction

1	11	52 bits

$(sign)fraction \times 2^{exponent}$

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 `bool`

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

4

2

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';              char ch = '8';
ch = ch + 1;                int d = ch - '0';
cout << ch << endl          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.)

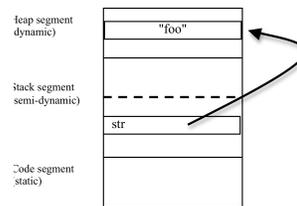C/C++ allows for low-level memory access using pointers

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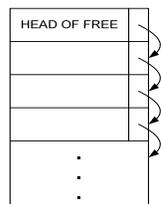
```
String str = "foo";
```

Heap segment (dynamic)	"foo"
Stack segment (semi-dynamic)	
	str
Code segment (static)	

7

Heap management

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

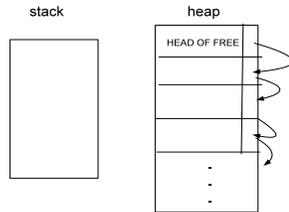the heap is divided into equal-size cells, each with a pointer
- the pointer fields are used initially to organize the heap as a linked list

| HEAD OF FREE |
| |
| |
| |
| ⋮ |

- keep pointer to head of free list

- to allocate space, take from front of free list

- to deallocate, put back at front

8

4

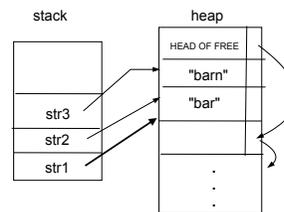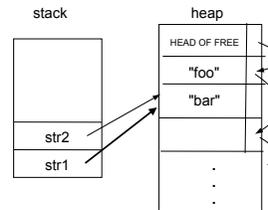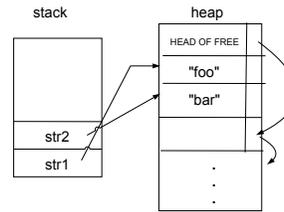# Heap example (Java w/ autoboxing)

stack | heap

HEAD OF FREE

.
.
.

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

/* CHECKPOINT 1 */

str1 = str2;

/* CHECKPOINT 2 */

String str3 = str1+"n";

/* CHECKPOINT 3 */
```

stack | heap

HEAD OF FREE
"foo"
"bar"

str2
str1

.
.
.

stack | heap

HEAD OF FREE
"foo"
"bar"

str2
str1

.
.
.

stack | heap

HEAD OF FREE
"barn"
"bar"

str3
str2
str1

.
.
.

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 */
```

stack
ptr

????
????

STACK

HEAD OF FREE

HEAP

12

6

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

STACK	CURRENT PARTITION	when current partition is full	STACK	BACKUP PARTITION
	BACKUP PARTITION	copy to backup & make it current		CURRENT PARTITION
	HEAP			HEAP
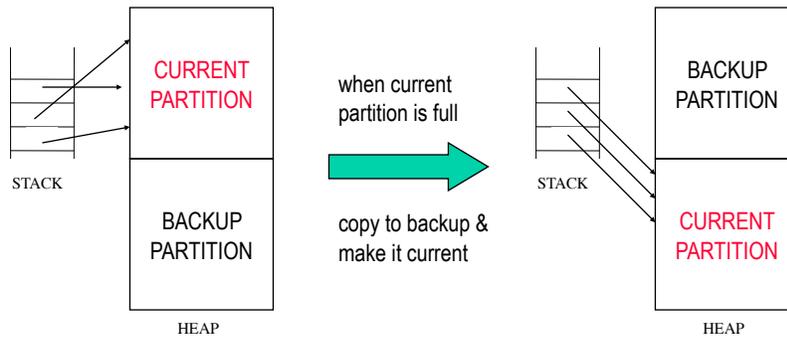
15

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 */
```
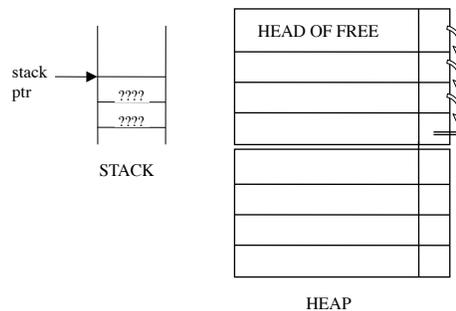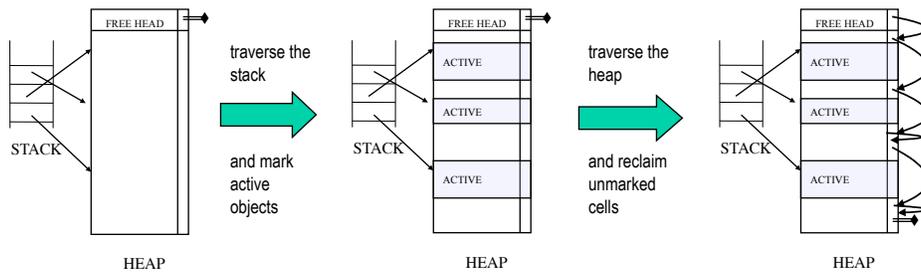
stack ptr

????
????

STACK

HEAD OF FREE

HEAP

16

8

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

FREE HEAD — traverse the stack — and mark active objects — STACK — HEAP

FREE HEAD — ACTIVE — ACTIVE — ACTIVE — traverse the heap — and reclaim unmarked cells — STACK — HEAP

FREE HEAD — ACTIVE — ACTIVE — ACTIVE — STACK — HEAP

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 */
```
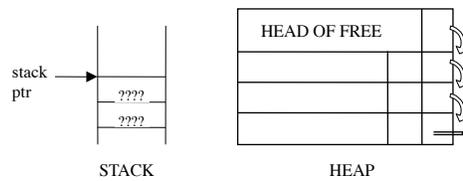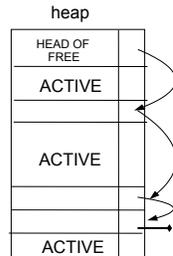

stack ptr — ???? — ???? — STACK

HEAD OF FREE — HEAP

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, …

heap

HEAD OF FREE
ACTIVE
ACTIVE
ACTIVE

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

10

Complex data types

early languages had limited data types
- FORTRAN elementary types + arrays
- COBOL   introduced structured data type for record
- PL/I   included many data types, with the intent of supporting
            a wide range of applications

better approach: ALGOL 68 provided a few basic types & a few flexible combination
methods that allow the programmer to structure data

common types/structures:

string	enumeration	subrange
array	record	union
set	list	. . .

Strings

- can be a primitive type (e.g., Scheme, SNOBOL)
- can be a special kind of character array (e.g., Pascal, Ada, C)

In C++ & Java, OOP can make the string type appear primitive
- C++ string type is part of the Standard Template Library   `#include <string>`
- Java String type is part of the `java.lang` package (automatically loaded)
- both are classes built on top of '\0'-terminated, C-style strings

```
String str = "Dave";        str  [ ] ——→  'D' 'a' 'v' 'e' '\0'
```

- Java strings are immutable – can't change individual characters, but can reassign an entire new value

```
str = str.substring(0, 1) + "el" + str.substring(2, 5);
```

reason: structure sharing is used to save memory

Enumerations & subranges

an *enumeration* is a user-defined ordinal type
- all possible values (symbolic constants) are enumerated

in C++ & Java: `enum Day {Mon, Tue, Wed, Thu, Fri, Sat, Sun};`
- C++: enum values are mapped to ints by the preprocessor (*kludgy*)
  ```
  Day today = Wed;              // same as  today = 2;
  cout << today << endl;        // prints 2
  today = 12;                   // illegal
  ```
- Java: enum values are treated as new, unique values
  ```
  Day today = Day.Wed;
  System.out.println(today);    // prints Wed
  ```

some languages allow new types that are *subranges* of other types
- subranges inherit operations from the parent type
- can lead to clearer code (since more specific), safer code (since range checked)

in Ada: `subtype Digits is INTEGER range 0..9;`
no subranges in C, C++ or Java

23

Arrays

an *array* is a homogeneous aggregate of data elements that supports random access via indexing

design issues:
- index type (C/C++ & Java only allow int, others allow any ordinal type)
- index range (C/C++ & Java fix low bound to 0, others allow any range)
- bindings
  - static (index range fixed at compile time, memory static)
    - FORTRAN, C/C++ (for globals)
  - fixed stack-dynamic (range fixed at compile time, memory stack-dynamic)
    - Pascal, C/C++ (for locals)
  - stack-dynamic (range fixed when bound, memory stack-dynamic)
    - Ada
  - heap-dynamic (range can change, memory heap-dynamic)
    - C/C++ & Java (using new), JavaScript
- dimensionality (C/C++ & Java only allow 1-D, but can have array of arrays)
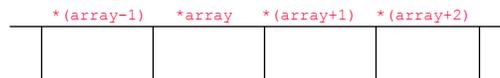
24

C/C++ arrays

C/C++ think of an array as a pointer to the first element

- when referred to, array name is converted to its starting address

```
int counts[NUM_LETTERS];              // counts ≡ &counts[0]
```

- array indexing is implemented via pointer arithmetic: `array[k] ≡ *(array+k)`

```
       *(array-1)   *array   *(array+1)  *(array+2)
```

the pointer type determines the distance added to the pointer

since an array is a pointer, can dynamically allocate memory from heap

```
int * nums = new int[numNums];        // allocates array of ints
```

- can resize by allocating new space, copying values, and reassigning the pointer

the C++ `vector` class encapsulates a dynamic array, with useful methods

25

Java arrays

in Java, arrays are reference types (dynamic objects)

must:	1) declare an array	`int nums[];`
	2) allocate space	`nums = new int[20];`
	can combine:	`int nums[] = new int[20];`

- as in C/C++, array indices start at 0
- unlike C/C++, bounds checking performed, can access length field

```
for (int i = 0; i < nums.length; i++) {
    System.out.println(nums[i]);
}
```

- like C++, Java also provides a more flexible `ArrayList` class
  but can only store objects (no primitives)

26

13

Records

a *record* is a (possibly) heterogeneous aggregate of data elements, each identified by a field name

heterogeneous → flexible   access by field name → restrictive

in C, a `struct` can group data values into a new type of object

```
struct Person {
    string lastName, firstName;
    char middleInit;
    int age;
};
```

C++: has both `struct` and `class`
- only difference: default protection (`public` in struct, `private` in class)
- structs can have methods, but generally used for C-style structures

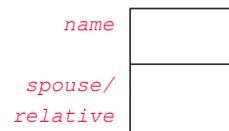Java: simplifies so that only `class`

27

Unions (variant records)

a *union* is allowed to store different values at different times

```
struct Person {
    string name;                              name
    union {
        string spouse;                      spouse/
        string relative;                    relative
    }
};
```

C/C++ do no type checking wrt unions

```
Person p;
p.relative = "Mom";
cout << p.spouse << endl;
```

in Ada, a tag value forces type checking (can only access one way)

no unions in Java

28

14

Assignments and expressions

when an assignment is evaluated,
- expression on rhs is evaluated first, then assigned to variable on lhs

within an expression, the order of evaluation can make a difference

```
x = 2;                        foo(x++, x);
y = x + x++;
```

in C/C++, if not covered by precedence/associativity rules, order is undefined
(i.e., implementation dependent) – similarly, in Pascal, Ada, …   WHY?

one exception: boolean expressions with and/or are evaluated left-to-right

```
for (int i = 0; i < size && nums[i] != 0; i++) {
    . . .
}
```

in Java, expressions are always evaluated left-to-right

29

15