CSC 533: Organization of Programming Languages

Spring 2005

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++ and Java as example languages

Primitive types: integer

languages often provide several sizes/ranges

in C++/Java

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

absolute sizes are implementation dependent in C++ TRADEOFFS?

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

```
1 = 00000001        -1 = 11111111
2 = 00000010        -2 = 11111110
3 = 00000011        -3 = 11111101
```

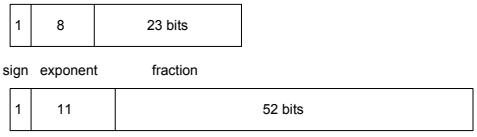
Primitive types: floating-point

again, languages often provide several sizes/ranges

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

C++ also has 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, …

Primitive types: boolean

introduced in ALGOL 60

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

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)

Primitive types: character

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

in 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);
```


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, …

Primitive types: pointer (cont.)

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

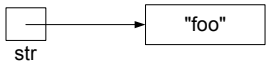
- `*` dereferencing operator
- `&` address-of operator

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

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

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

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

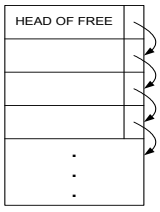


Heap management

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

simplest case: assume heap is divided into equal-size cells, each with a pointer

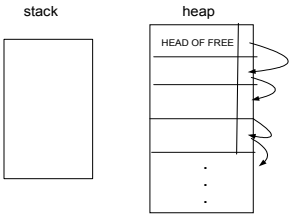
- can treat the heap as a linked list of memory cells

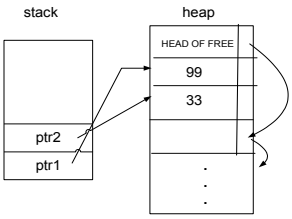


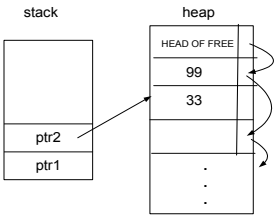
- keep pointer to head of free list
- to allocate space, take from front of free list
- to deallocate, put back at front

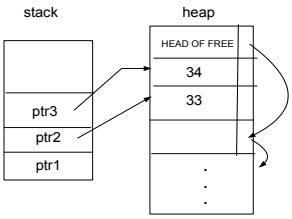
Heap example

```
int * ptr1 = new int;
*ptr1 = 99;

int * ptr2 = new int;
*ptr2 = 33;

delete ptr1;

int * ptr3 = new int;
*ptr3 = *ptr2 + 1;
```









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;
}
```

system allocated memory is deallocated automatically (lifetime ends with scope)

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

```
void Bar()
{
    int * ptr = new int;
    …
}
```

explicit allocation (new) requires explicit deallocation (delete)

would like to automatically and safely reclaim heap memory

2 common techniques: reference counts, garbage collection

Reference counts

along with each heap element, store a reference count

- indicates the number of pointers to the heap element

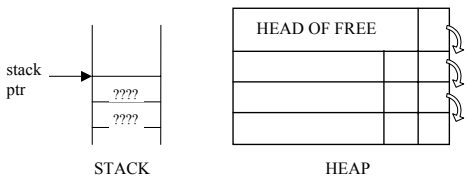
- when space is allocated, its reference count is set to 1
- each time a new pointer is set to it, increment the reference count
- each time a pointer is destroyed, decrement the reference count

provides 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

Reference counts example

```
int * ptr1 = new int;
*ptr1 = 9;
int * ptr2 = new int;
*ptr2 = 75;
/* CHECKPOINT 1 */

ptr1 = ptr2;
/* CHECKPOINT 2 */

if (*ptr1 <= *ptr2) {
  int * temp = new int;
  *temp = 1024;
  ptr1 = temp;
  /* CHECKPOINT 3 */
}

int * ptr3 = new int;
*ptr3 = 21;
/* CHECKPOINT 4 */
```



Reference counts (cont.)

unfortunately, reference counts are very costly

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

```
int *p, *q;
…
p = q;
```

➔

1) access *p, decrement reference count
2) if count = 0, deallocate
3) copy l-value of q to p
4) access *q, increment reference count

reference counts are popular in parallel programming

- work is spread evenly

Garbage collection

philosophy: dangling refs are *much* worse than garbage refs
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)

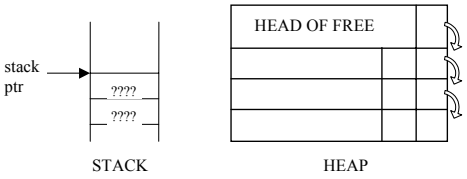
e.g., toothpaste tube analogy

2 stages in garbage collection

1. mark all "active" elements
   recursively traverse all data structures (from the stack),
   mark all used memory cells
2. sweep through the heap sequentially, collecting all unmarked cells

Garbage collection example

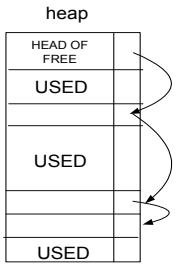
```
int * ptr1 = new int;
*ptr1 = 9;
int * ptr2 = new int;
*ptr2 = 75;
/* CHECKPOINT 1 */

ptr1 = ptr2;
/* CHECKPOINT 2 */

if (*ptr1 <= *ptr2) {
  int * temp = new int;
  *temp = 1024;
  ptr1 = temp;
  /* CHECKPOINT 3 */
}

int * ptr3 = new int;
*ptr3 = 21;
/* CHECKPOINT 4 */
```



Variable-sized elements

memory management is trickier if allocated in unequal sizes

- allocating/deallocating memory can leave gaps in the heap (*fragmentation*)



suppose need to store 3 element array

- must be contiguous

here, 3 spaces are free, but
allocations/deallocations have not left 3 contiguous

must be able to defragment (a.k.a compactify) the heap

- copy/shift all used memory in order to coalesce free space

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 STL (Standard Template Library)

```
#include <string>
```

operators/functions include

```
<<    >>    +    +=    ==    !=    <    >    <=    >=
[]    length    contains    substr    find
```

Java String type is part of the `java.lang` package (automatically loaded)

```
+    +=    equals    length    charAt    substring    indexOf
```

C++ strings

C++ string class is built on top of C-style strings

in C, the convention is to store a string as a char array, terminated with '\0'

```
char str[] = "Dave";
```

str → | 'D' | 'a' | 'v' | 'e' | '\0' |

the C library file `string.h` contains useful routines (e.g, `strlen`, `strcmp`)

C-style strings are very low-level

- must perform own allocation/deallocation
- not safe – can access the '\0' character
- messy!

C++ class encapsulates C-style strings nicely

- handles memory management automatically
- safe, hides the '\0' character
- defines standard operators

Java strings

in Java, strings are reference types (dynamic objects)

```
String str1 = new String("Hello");
String str2 = "World";
```

Strings are immutable

- can't change individual characters, but can reassign an entire new value

```
str1 = str1 + "!";
str2 = str2.substring(0, 2) + "u" + str2.substring(3, 5);
```

reason: structure sharing is used to save memory

Java also provides `StringBuffer` class

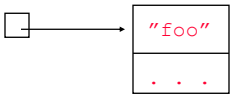
- allows changes to individual characters, can convert to and from `String`

Java strings (cont.)

in Java, all non-primitive types are objects

- implemented as pointers to dynamically allocated data
- non-primitive types are known as *reference types*

```
String str1 = "foo";
```



as a result:

- reference types appear to be passed by-reference (more later)
- cannot use == to compare reference types

  `String` provides `equals` method for comparisons

```
if (str1.equals(str2)) { . . . }
```


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++ or Java

Arrays

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

design issues:

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

C++ arrays

C++ thinks 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]
```

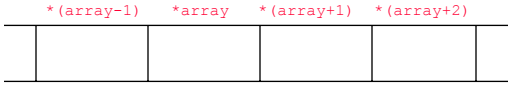
which explains why…

1. can't assign arrays as a whole
2. don't specify array size in a parameter

```
void Init(int array[], int size);
void Init(int * array, int size);
```

3. arrays appear to be passed by-reference (more later)
4. array indexing is not bounds checked

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



the pointer type determines the distance added to the pointer

dynamic arrays & vectors

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

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

for (int i = 0; i < numNums; i++) {     // can access like any other array
    nums[i] = i;
}

. . .

delete nums;                            // responsible for deallocating
```

vector class encapsulates a dynamic array, provides useful methods

- similar to how string class encapsulates a C-style string, adds functionality

```
vector<int> nums(numNums);          // creates a vector of ints
```

- ✓ constructor allocates the array from the heap, destructor deallocates
- ✓ `[]` operator performs bounds checking on index
- ✓ `resize` method truncates or expands the vector:
  allocates new array, copies entries, deallocates old array, resets array pointer

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++, array indices start at 0
- unlike C++, performs bounds checking, 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

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++: have both `struct` and `class` (only `class` in Java)

- only difference: default protection (public in struct, private in class)
- structs can have member functions, but generally used for C-style structures

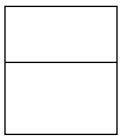
```
struct Person {
    string lastName, firstName;
    char middleInit;
    int age;
};
```

- assignment, copy constructor are provided by default, must define other ops

Unions (variant records)

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

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



C++ does 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

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;
y = x + x++;
```

```
foo(x++, x);
```

in 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

Type coercion in expressions

most languages (including C++) do some type coercion automatically

```
double sum = 3.2 + 5;          // sum = 8.2
```

```
class Rational
{
  public:
    Rational(int num = 0, int denom = 1);
    int GetNumerator();
    int GetDenominator();
    Rational & operator+(const Rational & rhs);
  private:
    int numerator, denominator;
};

-----------------------------------------------

Rational r1(2, 3);            // r1 = 2/3
Rational r2(3);               // r2 = 3/1
Rational r3;                  // r3 = 0/1

r3 = r1 + r2;                 // r3 = 11/3

r3 = r1 + 5;                  // r3 = 17/3
```

when define new C++ types (classes), constructors define implicit coercions

- if have constructor that takes an int as argument, then the compiler will be able to coerce ints to the new type
- this can be handy, but extremely dangerous!!!!
- programmer must ensure that coercion paths are unique

coercion is limited in Java, explicit