CSC 533: Programming Languages

Spring 2018

Complex data types & control

- string, enum, subrange, array, record, …
- expressions and assignments
- conditional control & branching

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

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:

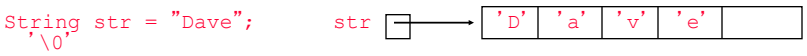
string	enumeration	subrange
array	record	union
set	list	. . .

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



- 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

3

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

4

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)

5

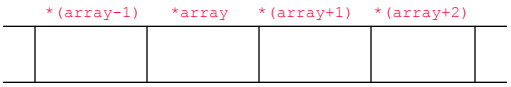
C/C++ arrays

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)`



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

6

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)

7

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

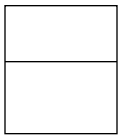
Java: simplifies so that only `class`

8

Unions (variant records)

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

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

9

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/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

10

Conditionals & loops

early control structures were tied closely to machine architecture

e.g., FORTRAN arithmetic if: based on IBM 704 instruction

```
     IF (expression) 10, 20, 30
10   code to execute if expression < 0
     GO TO 40
20   code to execute if expression = 0
     GO TO 40
30   code to execute if expression > 0
40   . . .
```

later languages focused more on abstraction and machine independence

some languages provide counter-controlled loops

e.g., in Pascal:

```
for i := 1 to 100 do
begin
    . . .
end;
```

- counter-controlled loops tend to be more efficient than logic-controlled
- C/C++ and Java don't have counter-controlled loops (for is syntactic sugar for while)

11

Branching

unconditional branching (i.e., GOTO statement) is very dangerous

- leads to *spaghetti code*, raises tricky questions w.r.t. scope and lifetime
  what happens when you jump out of a function/block?
  what happens when you jump into a function/block?
  what happens when you jump into the middle of a control structure?

most languages that allow GOTO's restrict their use

- in C/C++, can't jump into another function
  can jump into a block, but not past declarations

```
void foo() {
    . . .
    goto label2;      // illegal: skips declaration of str
    . . .
  label1:
    string str;
    . . .
  label2:
    goto label1;      // legal: str's lifetime ends before branch
}
```

12

Branching (cont.)

why provide GOTO's at all? (Java doesn't)

- backward compatibility
- some argue for its use in specific cases (e.g., jump out of deeply nested loops)

C/C++ and Java provide statements for more controlled loop branching

- *break:* causes termination of a loop

```
while (true) {
    num = input.nextInt();
    if (num < 0) break;
    sum += num;
}
```

- *continue:* causes control to pass to the loop test

```
while (inputKey != 'Q') {
    if (keyPressed()) {
        inputKey = GetInput();
        continue;
    }
    . . .
}
```

13