Review: Arrays, pointers, structures
(Chapter 1)

CS 3358
Spring 2015

Jill Seaman

Data Types

- Data Type:
  - set of values
  - set of operations over those values
- example: Integer
  - whole numbers, -32768 to 32767
  - +, -, *, /, %, ==, !=, <, >, <=, >=, ...
- Which operation is not valid for float?

Data Types (C/C++)

- Scalar (or Basic) Data Types (atomic values)
  - Arithmetic types
    - Integers
      - short, int, long
      - char, bool
    - Floating points
      - float, double, long double
- Composite (or Aggregate) Types:
  - Arrays: ordered sequence of values of the same type
  - Structures: named components of various types

Review: Arrays

- An array contains multiple values of the *same type*.
- values are stored consecutively in memory.
- An array definition in C++: `int numbers[5];`
- Array indices (subscripts) are zero-based

  `numbers[0]   ...   numbers[4]`
- the subscript can be ANY integer expression:

  `numbers[2]   numbers[i]   numbers[(i+2)/2]`
- What operations can be performed over (entire) arrays?

First-Class vs Second-Class objects

- **first-class objects** can be manipulated in the usual ways without special cases and exceptions
  - copy (=, assignment)
  - comparison (==, <, ...)
  - input/output (<<, >>)
- **second-class objects** can be manipulated only in restricted ways, may have to define operations yourself
  - Usually primitive (built-in) data types

First-Class vs Second-Class objects: Strings

- **second-class object:** C-String (char array)
  - strcpy
  - strlen
  - strcat
  - strcmp

Special functions

- **first-class object:** string class (standard library)
  - =
  - size() member function
  - ==, <, ...
  - +

The "usual" operators

First-Class vs Second-Class objects: arrays

- **second-class object:** primitive array
  - = does not copy elements
  - length undefined
  - ==, <, ... do not perform as expected

usual operations are not defined

- **first-class object:** vector class (standard template library)
  - =
  - size() member function
  - ==, <, ...

The "usual" operators

vector and string

- Included in standard (template) library
- class definitions used for first class objects
- The definitions provide an interface that hides the implementation from the programmer.
- Programmer does not need to understand the implementation to use the types.
- Vector: like an array, can contain elements of any single given type.

Using vector

- Include file

```
#include <vector>
```

- To define a vector give a name, element type, and optional size (default is 0):

```
vector<int> a(3);  // 3 int elements
```

- Can use [ ] to access the elements (0-based):

```
a[2] = 12;
```

- Use the size member function to get the size:

```
cout << a.size() << endl; //outputs 3
```


Using vector

- Use resize() to change the size of the vector:

```
vector<int> a;   // size is 0
a.resize(4);     // now has 4 elements
```

- Use push_back to increase the size by one and add a new element to the **end**, pop_back removes the last element

```
vector<int> a;      // size is 0
a.push_back(25);    // now has 1 element
a.pop_back();       // now has 0 elements
```

- Implementation of resizing is handled internally (presumably it is done efficiently).

Parameter passing
(for large objects)

- Call by value is the default

```
int findMax(vector<int> a);
```

Problem: lots of copying if a is large

- Call by reference can be used

```
int findMax(vector<int> & a);
```

Problem: may still want to prevent changes to a

- Call by constant reference:

```
int findMax(const vector<int> & a);
```

the “const” won’t allow a to be changed

Multidimensional arrays

- <u>multidimensional array</u>: an array that is accessed by more than one index

```
int table[2][5];    // 2 rows, 5 columns
table[0][0] = 10;   // puts 10 in upper left
```

- There are no first-class versions of this in the STL
- The book defines a first-class version called matrix in ch 3 to represent a 2-dimensional array.
- The primitive version can have more than 2 dimensions.

Pointers

- Pointer: a variable that stores the address of another variable, providing indirect access to it.
- The address operator (&) returns the address of a variable.

```
int x;
cout << &x << endl;    // 0xbffffb0c
```

- An asterisk is used to define a pointer variable

```
int *ptr;
```

- "ptr is a pointer to an int". It can contain addresses of int variables.

```
ptr = &x;
```


Pointers

- The unary operator * is the dereferencing operator.
- *ptr is an alias for the variable that ptr points to.

```
int x = 10;
int *ptr;   //declaration, NOT dereferencing
ptr = &x;   //ptr gets the address of x
*ptr = 7;   //the thing ptr pts to gets 7
```

- Initialization:

```
int x = 10;
int *ptr = &x; //declaration, NOT dereferencing
```

- ptr is a pointer to an int, and it is initialized to the address of x.

Pointers: watchout

- What is wrong with each of the following?

```
int *ptr = &x;
int x = 10;
```

```
int x = 10;
int *ptr = x;
```

```
int x = 10;
int y = 99;
int *ptr = &y;
*ptr = x;
ptr = &x;
```


Pointers: watchout

- What is wrong with each of the following?

```
int *ptr = &x;
int x = 10;
```

x is not declared yet

```
int x = 10;
int *ptr = x;
```

x is not an address

```
int x = 10;
int y = 99;
int *ptr = &y;
*ptr = x;
ptr = &x;
```

y gets 10 (changes y)
ptr points to x (changes ptr)

Pointers: More examples

- What is happening in each of the following?

```
int *ptr = NULL;
```

```
int x = 10;
int *ptr = &x;
*ptr += 5;
*ptr++;
```

```
int x = 10, y = 99;
int *ptr1 = &x, *ptr2 = &y;

ptr1 = ptr2;
*ptr1 = *ptr2;

if (ptr1==ptr2) ...
if (*ptr1==*ptr2) ...
```


Pointers: More examples

- What is happening in each of the following?

```
int *ptr = NULL;
```

sets ptr to 0 (null ptr)

```
int x = 10;
int *ptr = &x;
*ptr += 5;
*ptr++;
```

changes x to 15

changes ptr to point to location after x (returns its value)

```
int x = 10, y = 99;
int *ptr1 = &x, *ptr2 = &y;

ptr1 = ptr2;
*ptr1 = *ptr2;

if (ptr1==ptr2) ...
if (*ptr1==*ptr2) ...
```

makes ptr1 pt to what ptr2 pts to

copies what ptr2 points to into the location ptr1 points to

do the ptrs point to the same location?

do the ptrs point to the same values?

Dynamic Memory Allocation

- Automatic variables: variables that are created when declared, and destroyed at the end of their scope.
- Dynamic memory allocation allows you to create and destroy anonymous variables on demand, during run-time.
- "new" operator requests dynamically allocated memory and returns address of newly created anonymous variable.

```
string *ptr;
ptr = new string("hello");
cout << *ptr << endl;
cout << "Length: " << (*ptr).size() << endl;
```


Dynamic Memory Allocation: delete

- When you are finished using a variable created with new, use the delete operator to destroy it.

```
int *ptr;
ptr = new int;
*ptr = 100;
...
delete ptr;
```

- Do not "delete" pointers whose values were NOT dynamically allocated using new.
- Do not forget to delete dynamically allocated variables (memory leaks: allocated but inaccessible memory).

Reference Variables

- Reference Type: an alias to another variable.
- It's like a constant pointer variable that is always implicitly dereferenced.

```
int x = 25;
int &y = x;    // int & is the reference type
y +=3;         // this changes x to 3
```

- Reference variables MUST be initialized when they are declared.
  - No way to change their address value later (assignment dereferences them)
- C++ call-by-reference parameters are really reference variables used as parameters.

Structures

- A structure stores a collection of objects of various types
- Each object in the structure is a member, and is accessed using the dot member operator.

```
struct Student {
   int idNumber;
   string name;
   int age;
   string major;
};
```

Defines a new data type

```
Student student1, student2;
student1.name = "John Smith";
```

Defines new variables

Structures: operations

- Valid operations over entire structs:
  - assignment: `student1 = student2;`
  - function call: `myFunc(gradStudent,x);`
- Invalid operations over structs:
  - comparison: `student1 == student2`
  - output: `cout << student1;`
  - input: `cin >> student2;`
  - Must do these member by member

Pointers to structures

- We can define pointers to structures

```
Student s1 = {12345,"Jane Doe", 18, "Math"};
Student *ptr = &s1;
```

- To access the members via the pointer:

```
cout << *ptr.name << end;    // ERROR: *(ptr.name)
```

- dot operator has higher precedence, so use ():

```
cout << (*ptr).name << end;
```

- or equivalently, use ->:

```
cout << ptr->name << end;
```

Indigenous vs exogenous data

- Consider two structure definitions:

```
struct Student {
   int idNumber;
   string name;
   int age;
   string major;
};
```

```
struct Teacher {
   int idNumber;
   string *name;
};
```

- indigenous data: completely contained within the structure

```
all Students members
```

- exogenous data: reside outside the structure, and are pointed to from the structure.

```
Teacher's name
```


Shallow copy vs deep copy

- Consider structure assignment:

```
Student s1, s2;
...
s1 = s2;
```

```
Teacher t1, t2;
...
t1 = t2;
```

- By default, it is member by member copy.
- This is fine for Student, but not the Teachers
- t1.name and t2.name share the same memory, point to the same place.
- changing t1->name will also change t2->name
- delete t1.name; will make t2.name invalid.

Shallow copy vs deep copy

- Shallow copy: copies top level data only. For pointers, the address is copied, not the values pointed to. This is the default for =.
- Deep copy: copies the pointed at values instead of their addresses. May require allocating new memory for the new value.

Assert

- requires #include <cassert>
- void assert (int expression); //prototype
- If the expression is equal to zero (false), a message is written to the screen and the program is terminated.

```
Assertion failed: expression, file filename, line line number
```

```
int findMax (vector<int> a) {
   assert (a.size() > 0);
   int max = a[0];
   //code to find maximum goes here
   return max;
};
```