

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: arrays

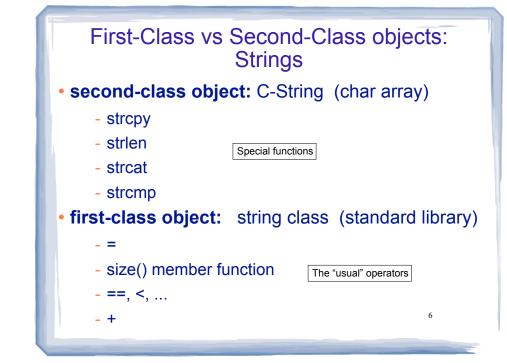
- second-class object: primitive array
 - = does not copy elements
 - length undefined
- usual operations are not defined
- ==, <, ... do not perform as expected
- first-class object: vector class (standard template library)
 - size() member function

The "usual" operators

7

- ==, <, ...

- =



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[3] = 12;• Use the size member function to get the size: cout << a.size() << endl; //outputs 3</pre>

11

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

<pre>vector<int> a;</int></pre>	// 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

 multidimensional array: 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 one (ch 3) called a matrix.
- The primitive version can have more than 2 dimensions.

Pointers	Pointers
 Pointer: an 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 	 The unary operator * is the dereferencing operator. *ptr is an alias for the variable that ptr points to. 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.
• What is wrong with each of the following?	• What is wrong with each of the following?

<pre>int *ptr = &x int x = 10;</pre>	
<pre>int x = 10; int *ptr = x;</pre>	
<pre>int x = 10; int y = 99; int *ptr = &y</pre>	
*ptr = x; ptr = $\&x$	

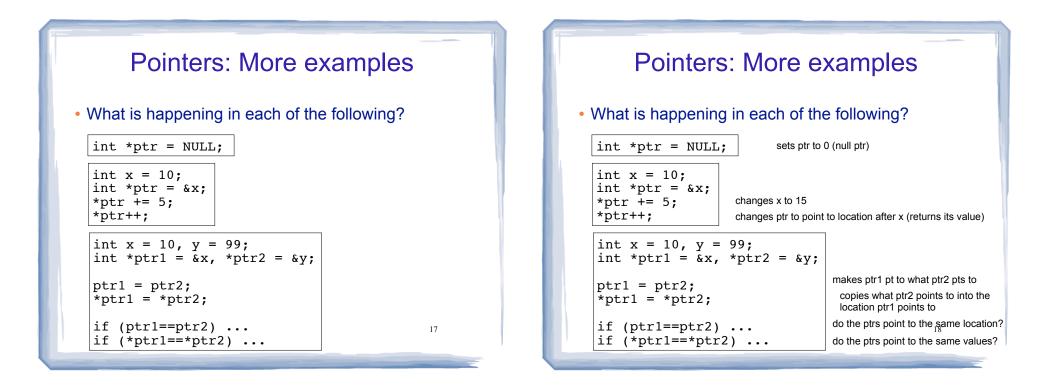
What is wrong with each of the following int *ptr = &x; int x = 10; x is not declared yet

int x = 10; int *ptr = x;

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

x is not an address

15



Dynamic Memory Allocation

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

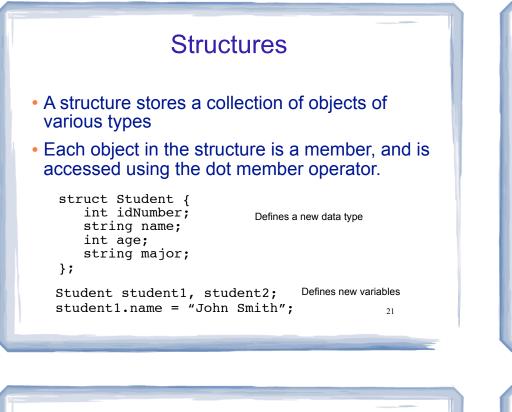
```
string *ptr;
ptr = new string("hello");
cout << *ptr << endl;
cout << "Length: " << (*ptr).size() << endl;</pre>
```

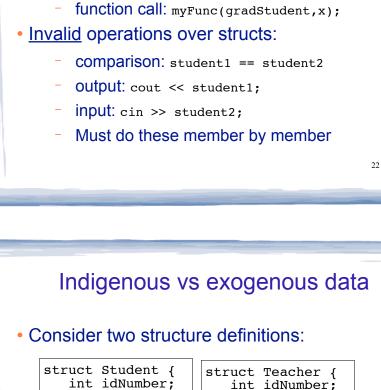
Dynamic Memory Allocation: delete

• When you are finished using a variable created with new, use the <u>delete</u> 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).





Structures: operations

assignment: student1 = student2;

Valid operations over entire structs:

Pointers to structures • We can define pointers to structures Student s1 = {12345, "Jane Doe", 18, "Math"}; Student *ptr = &s1; string name: string *name; int age; }; • To access the members via the pointer: string major; }; cout << *ptr.name << end;</pre> // ERROR: *(ptr.name) indigenous data: completely contained within the dot operator has higher precedence, so use (): structure all Students members cout << (*ptr).name << end;</pre> exogenous data: reside outside the structure. • or equivalently, use ->: and are pointed to from the structure. 23 cout << ptr->name << end;</pre> Teacher's name

24

Shallow copy vs deep copy

• Consider structure assignment:

Student s1, s2;	Teacher t1, t2;
s1 = s2;	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.

25

• delete t1.name; will make t2.name invalid.

Shallow copy vs deep copy

- <u>Shallow copy</u>: copies top level data only. For pointers, the address is copied, not the values pointed to. This is the default
- <u>Deep copy</u>: copies the pointed at values instead of their addresses. Requires allocating new memory.
- Same concepts apply to comparisons.

26