

Dereferencing and initializing

- 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 as Function Parameters

• Use pointers to implement pass by reference.

```
//prototype: void changeVal(int *);
void changeVal (int *val) {
    *val = *val * 11;
}
int main() {
    int x;
    cout << "Enter an int " << endl;
    cin >> x;
    changeVal(&x);
    cout << x << endl;</pre>
```

 How is it different from using reference parameters?

Pointers and Arrays

• You can treat an array variable as if it were a pointer to its first element.

int numbers[] = {10, 20, 30, 40, 50};	Output:
<pre>cout << "first: " << numbers[0] << endl; cout << "first: " << *numbers << endl;</pre>	first: 10 first: 10 0xbffffb00
<pre>cout << &(numbers[0]) << endl; cout << numbers << endl;</pre>	0xbffffb00

Pointer Arithmetic

 When you add a value n to a pointer, you are actually adding n times the size of the data type being referenced by the pointer.

int numbers[] = {10, 20, 30, 40, 50};

// sizeof(int) is 4. // Let us assume numbers is stored at 0xbfffb00 // Then numbers+1 is really 0xbffffb00 + 1*4, or 0xbffffb04 // And numbers+2 is really 0xbffffb00 + 2*4, or 0xbffffb08 // And numbers+3 is really 0xbffffb00 + 3*4, or 0xbffffb0c cout << "second: " << numbers[1] << endl;</pre> Output: cout << "second: " << *(numbers+1) << endl;</pre> second: 20 cout << "size: " << sizeof(int) << endl;</pre> second: 20 cout << numbers << endl;</pre> size: 4 0xbffffb00 cout << numbers+1 << endl;</pre> 0xbffffb04

Note: array[index] is equivalent to *(array + index)

 Pointers and Arrays
 pointer operations * + can be used with array variables.

 int list[10]; cin >> *(list+3);
 subscript operations: [] can be used with pointers.

5

7

int list[] = {1,2,3}; int *ptr = list; cout << ptr[2];</pre>

Comparing Pointers

 pointers (addresses) maybe compared using the relational operators:

< <= > >= == !=

Examples: [int]

int arr[25];

cout << (&arr[1] > &arr[0]) << endl; cout << (arr == &arr[0]) << endl; cout << (arr <= &arr[20]) << endl; cout << (arr > arr+5) << endl;</pre>

- What is the difference?
 - ptr1 < ptr2
 - *ptr1 < *ptr2

Dynamic Memory Allocation

- When a function is called, memory for local variables is automatically allocated.
- When a function exits, memory for local variables automatically disappears.
- Must know ahead of time the maximum number of variables you may need.
- Dynamic Memory allocation allows your program to create variables on demand, during run-time.

The new operator

• "new" operator requests dynamically allocated memory for a certain data type:

```
int *iptr;
iptr = new int;
```

- new operator returns <u>address</u> of newly created <u>anonymous</u> variable.
- use dereferencing operator to access it:

```
*iptr = 11;
cin >> *iptr;
int value = *iptr / 3;
```

Dynamically allocated arrays

dynamically allocate arrays with new:

```
int *iptr; //for dynamically allocated array
int size;
cout << "Enter number of ints: ";
cin >> size;
iptr = new int[size];
for (int i=1; i<size; i++) {
    iptr[i] = i;
```

 Program will throw an exception and terminate if not enough memory available to allocate

delete!

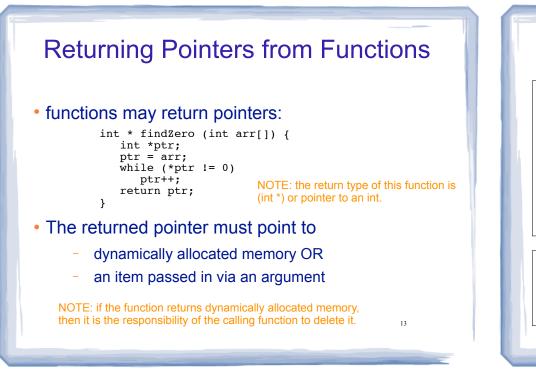
• When you are finished using a variable created with new, use the delete operator to destroy it:

```
int *ptr;
double *array;
ptr = new int;
array = new double[25];
...
delete ptr;
delete [] array; // note [] required for dynamic arrays!
```

 Do not "delete" pointers whose values were NOT dynamically allocated using new!

12

 Do not forget to delete dynamically allocated variables (Memory Leaks!!).



Returning Pointers from Functions: duplicateArray

int *duplicateArray (int *arr, int size) { int *newArray; if (size ≤ 0) //size must be positive //NULL is 0, an invalid address return NULL; newArray = new int [size]; //allocate new array for (int index = 0; index < size; index++)</pre> newArray[index] = arr[index]; //copy to new array return newArray; Output int a $[5] = \{11, 22, 33, 44, 55\};$ int *b = duplicateArray(a, 5); 0 ok for (int i=0; i<5; i++) 1 ok

if (a[i] == b[i]) cout << i << " ok" << endl; delete [] b; //caller deletes mem

```
0 ok
1 ok
2 ok
3 ok
4 ok
```

14

Structures

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

```
struct Student {
    int idNumber; Defines a new data type
    string name;
    int age;
    string major;
};
Student student1, student2; Defines new variables
```

```
Student student1, student2; Defines new Variables
student1.name = "John Smith"; 15
Student student3 = {123456,"Ann Page",22,"Math"};
```

Structures: operations

- Valid operations over entire structs:
 - assignment: student1 = student2;
 - function call: myFunc(gradStudent,x);
 void myFunc(Student, int); //prototype

Invalid operations over structs:

- **COMPARISON:** student1 == student2
- Output: cout << student1;</pre>
- input: cin >> student2;
- Must do these member by member ¹⁶

Arrays of Structures You can store values of structure types in arrays. Student roster[40]; //holds 40 Student structs

• Each student is accessible via the subscript notation.

roster[0] = student1; //copy student1 into 1st position

17

19

Members of structure accessible via dot notation

cout << roster[0].name << endl;</pre>

Arrays of Structures

• Arrays of structures processed in loops:

cout << roster[i].name << endl;</pre>

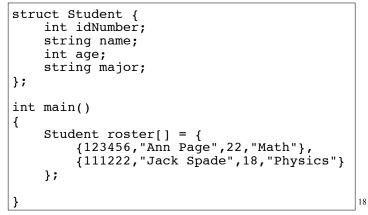
```
Student roster[40];
```

//input

}

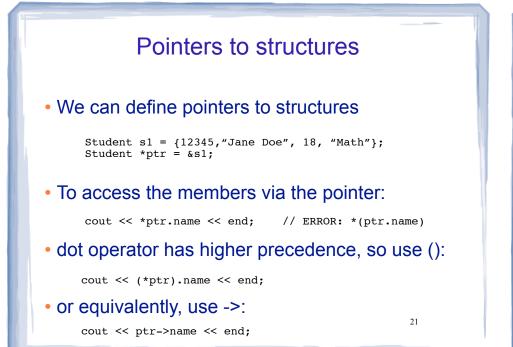
Arrays of Structures: initialization

• To initialize an array of structs:



Passing structures to functions

• Structure variables may be passed as arguments to functions:



Dynamically Allocating Structures

• Structures can be dynamically allocated with new:

Student *sptr; sptr = new Student; sptr->name = "Jane Doe"; sptr->idNum = 12345; ... delete sptr;

• Arrays of structures can also be dynamically allocated:

<pre>Student *sptr; sptr = new Student[100]; sptr[0].name = "John Deer";</pre>	1
<pre> delete [] sptr;</pre>	

No arrows (->) necessary. It's just an array of Student

22

Sample Problem 1

- Array Expander: Write a function expander that accepts an int array and the array's size as arguments. The function should create a new array that is twice the size of the argument array. The function should copy the contents of the argument array to the new array and initialize the unused elements of the second array with 0. The function should return a pointer to the new array.
- Write a driver that calls the expander function, and outputs the resulting array. Be sure to deallocate any dynamically allocated memory.

Sample Problem 2

Soccer Scores: Write a program that stores the following data about a soccer player in a structure:

Player's Name Player's Number

Points Scored by Player

The program should keep an array of 12 of these structures. Each element is for a different player on a team. When the program runs it should ask the user to enter the data for each player. The program should calculate and display the total points earned by the team. The number and name of the player who has earned the most points should also be displayed.