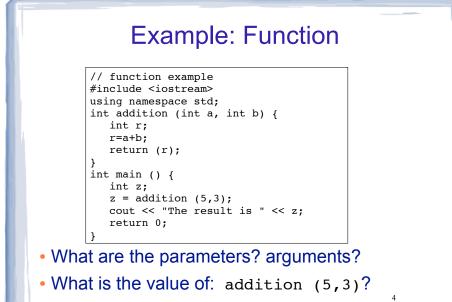


## Function Call, Return Statement

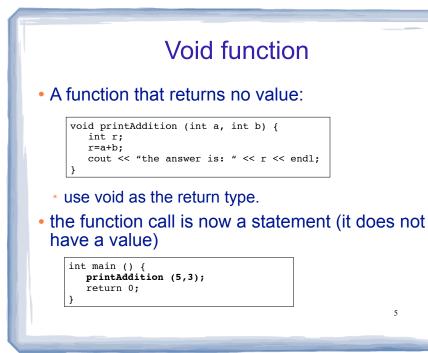
• Function call expression

identifier ( expression1, . . )

- \* Causes control flow to enter body of function named identifier.
- parameter1 is initialized to the value of expression1, and so on for each parameter
- \* expression1 is called an argument.
- Return statement: return expression;
  - inside a function, causes function to stop, return control to caller.
- The value of the return *expression* becomes the value of the function call



• What is the output?



# Prototypes

- In a program, function definitions must occur before any calls to that function
- To override this requirement, place a prototype of the function before the call.
- The pattern for a prototype:

datatype identifier (type1, type2, ...);

 the function header without the body (parameter names are optional).

6

# Arguments passed by value

- <u>Pass by value</u>: when an argument is passed to a function, its value is *copied* into the parameter.
- It is implemented using variable initialization (behind the scenes):

#### int param = argument;

- Changes to the parameter in the function body do **not** affect the value of the argument in the call
- The parameter and the argument are stored in separate variables; separate locations in memory.

### Example: Pass by Value #include <iostream> Output: number is 12 using namespace std; myValue is 200 Back in main, number is 12 void changeMe(int); int main() { int number = 12;cout << "number is " << number << endl;</pre> changeMe(number); 🔨 cout << "Back in main, number is " << number << endl;</pre> return 0; int mvValue = number: } void changeMe(int myValue) { myValue = 200; cout << "myValue is " << myValue << endl;</pre> ļ 8 changeMe failed to change the argument!

# Parameter passing by Reference

- <u>Pass by reference</u>: when an argument is passed to a function, the function has direct access to the original argument (no copying).
- Pass by reference in C++ is implemented using a reference parameter, which has an ampersand (&) in front of it:

void changeMe (int &myValue);

- A reference parameter acts as an **alias** to its argument, it is NOT a separate storage location.
- Changes to the parameter in the function **DO** affect the value of the argument

### Example: Pass by Reference #include <iostream> Output: number is 12 using namespace std; myValue is 200 Back in main. number is 200 void changeMe(int &); int main() { int number = 12;cout << "number is " << number << endl;</pre> changeMe(number); cout << "Back in main, number is " << number << endl; return 0; myValue is an alias for number, } only one shared variable void changeMe(int &myValue) { myValue = 200;cout << "myValue is " << myValue << endl;</pre> 10

# Scope of variables

- For a given variable definition, in which part of the program can it be accessed?
- \* **Global variable** (defined outside of all functions): can be accessed anywhere, after the definition.
- \* Local variable (defined inside of a function): can be accessed inside the block in which it is defined, after the definition.
- \* **Parameter**: can be accessed anywhere inside of its function body.
- Variables are destroyed at the end of their scope.

### 

Variables defined in one function cannot be seen from another.

# Arrays

- An array is:
  - A series of elements of the same type
  - placed in contiguous memory locations
  - that can be individually referenced by adding an index to a unique identifier.
- To declare an array:

datatype identifier [size];

int numbers[5];

- datatype is the type of the elements
- identifier is the name of the array
- size is the number of elements (constant)<sup>13</sup>

# Array initialization

• To specify contents of the array in the definition:

float scores[3] = {86.5, 92.1, 77.5};

- creates an array of size 3 containing the specified values.

float scores[10] = {86.5, 92.1, 77.5};

- creates an array containing the specified values followed by 7 zeros (partial initialization).

float scores[] = {86.5, 92.1, 77.5};

- creates an array of size 3 containing the specified values (size is determined from list).

# Array access

• to access the value of any of the elements of the array individually as if it was a normal variable:

scores[2] = 89.5;

- scores[2] is a variable of type float
- rules about subscripts:
  - they always start at 0, last subscript is size-1
  - the subscript must have type int
  - they can be any expression
- watchout: brackets used both to declare the array and to access elements.

# Working with arrays and array elements

- An array element:
- can be used exactly like any variable of the element type.
- you can assign values to it, use it in arithmetic expressions, pass it as an argument to a function.
- Generally there are NO C++ operations you can perform over entire arrays.
- you cannot assign one array to another
- you cannot input into an array
- you cannot compare one array to another 16

# Example: Processing arrays

### Computing the average of an array of scores:

# Arrays as parameters In the <u>function definition</u>, the parameter type is a variable name with an empty set of brackets: []

- Do NOT give a size for the array inside []

void showArray(int values[], int size)

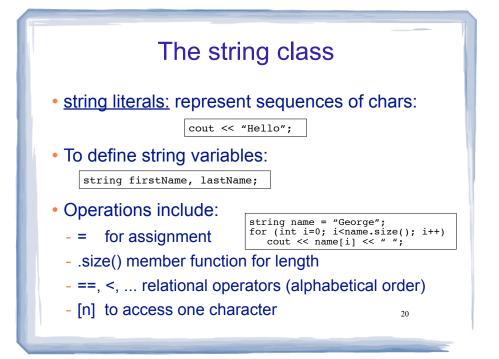
• In the <u>prototype</u>, empty brackets go after the element datatype.

void showArray(int[], int)

• In the <u>function call</u>, use the variable name for the array.

• An array is **always** passed by reference.

### **Example: Partially filled arrays** int sumList (int list[], int size) {//sums elements in list array int total = 0;for (int i=0; i < size; i++) {</pre> sums from position 0 to size-1. total = total + list[i]; even if the array is bigger. return total; const int CAPACITY = 100; int main() { int scores[CAPACITY]; int count = 0;//tracks number of elems in array cout << "Enter the programming assignment scores:" << endl;</pre> cout << "Enter -1 when finished" << endl; int score: cin >> score; while (score != -1 && count < CAPACITY) { scores[count] = score; count++; cin >> score; int sum = sumList(scores,count); pass count, not CAPACITY



### **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; Defines a new data type string name; int age; string major: }; Defines new variables Student student1, student2; student1.name = "John Smith"; 21 Student student3 = {123456,"Ann Page",22,"Math"};

# 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;</p>
  - input: cin >> student2;
  - Must do these member by member

22

# 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;
```

Members of structure accessible via dot notation

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

23

# **Overloaded Functions**

- Overloaded functions have the same name but different parameter lists.
- The parameter lists of each overloaded function must have different types and/or number of parameters.
- Compiler will determine which version of the function to call by matching arguments to parameter lists

Example: Overloaded functions	
<pre>double calcWeeklyPay (int hours, double payRate) {     return hours * payRate; }</pre>	
<pre>double calcWeeklyPay (double annSalary) {     return annSalary / 52;</pre>	
<pre>} int main () {     int h;     double r;</pre>	Output: Enter hours worked and pay rate: 37 19.5 Pay is: 721.5 Enter annual salary: 75000 Pay is: 1442.31
<pre>cout &lt;&lt; "Enter hours worked and pay rate: "; cin &gt;&gt; h &gt;&gt; r; cout &lt;&lt; "Pay is: " &lt;&lt; calcWeeklyPay(h,r) &lt;&lt; endl; cout &lt;&lt; "Enter annual salary: "; cin &gt;&gt; r; cout &lt;&lt; "Pay is: " &lt;&lt; calcWeeklyPay(r) &lt;&lt; endl;</pre>	
return 0; }	25

# **Default Arguments**

- A <u>default argument</u> for a parameter is a value assigned to the parameter when an argument is not provided for it in the function call.
- The default argument patterns:
  - \* in the prototype:

datatype identifier (type1 = c1, type2 = c2, ...);

\* OR in the function header:

```
datatype identifier (type1 p1 = c1, type2 p2 = c2, ...) {
...
}
```

• c1, c2 are constants (named or literals)

```
Example: Default Arguments
```

```
void showArea (double length = 20.0, double width = 10.0)
{
    double area = length * width;
    cout << "The area is " << area << endl;
}</pre>
```

27

• This function can be called as follows:

```
showArea(); ==> uses 20.0 and 10.0
The area is 200
showArea(5.5,2.0); ==> uses 5.5 and 2.0
The area is 11
showArea(12.0); ==> uses 12.0 and 10.0
The area is 120
```

### **Default Arguments: rules**

• When an argument is left out of a **function call**, all arguments that come after it must be left out as well.

showArea(5.5); // uses 5.5 and 10.0
showArea(,7.1); // NO, won't work, invalid syntax

 If not all parameters to a function have default values, the parameters with defaults must come last:

```
int showArea (double = 20.0, double); //NO
int showArea (double, double = 20.0); //OK 28
```