

• 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

// function example
<pre>#include <iostream></iostream></pre>
using namespace std;
int addition (int a, int b) {
int result;
result=a+b;
return result;
}
int main () {
int z;
z = addition (5,3);
<pre>cout << "The result is " << z <<endl;< pre=""></endl;<></pre>
}

- What are the parameters? arguments?
- What is the value of: addition (5,3)?
- What is the output?

Void function

• A function that returns no value:

```
void printAddition (int a, int b) {
    int result;
    result=a+b;
    cout << "the answer is: " << result << endl;
}</pre>
```

- * use void as the return type.
- the function call is now a statement (it does not have a value)

int main () {
 printAddition (5,3);
}

Arguments passed by value

5

- <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.

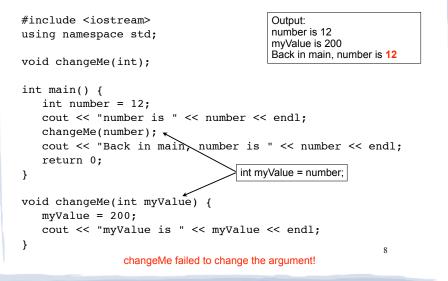
Prototypes

- 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).

Example: Pass by Value



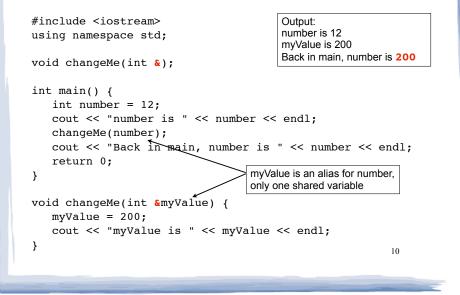
Parameter passing by Reference

- <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

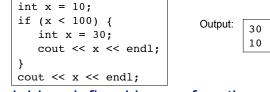


Scope of variables

- 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 its definition.
 - Local variable (defined inside of a function): can be accessed inside the block in which it is defined, after its definition.
 - * **Parameter**: can be accessed anywhere inside of its function body.
- Variables are destroyed at the end of their scope.

More scope rules

- Variables in the same exact scope cannot have the same name
 - Parameters and local function variables cannot have the same name
 - Variable defined in inner block can hide a variable with the same name in an outer block.



• 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 using an index along with the array name.
- 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)¹³

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;

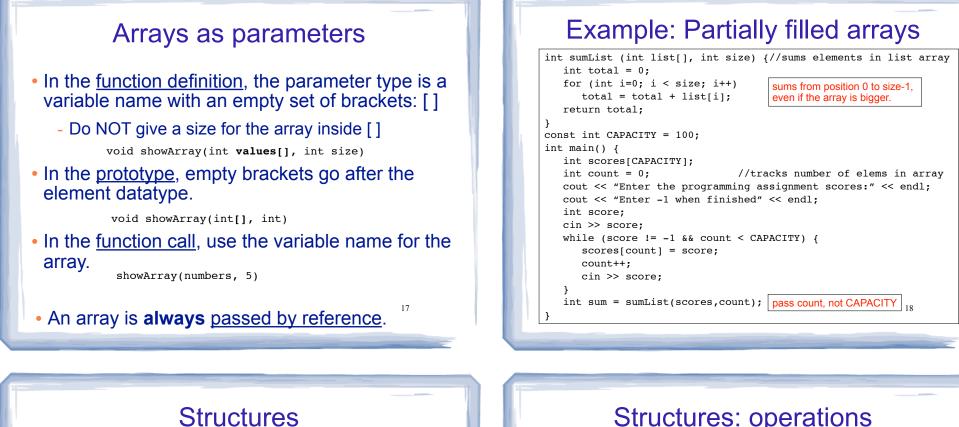
- scores[2] is a variable of type float
- rules about subscripts (aka indexes):
 - they always start at 0, last subscript is size-1
 - the subscript must have type int
 - they can be any expression
- watchout: brackets used both to declare the array and to access elements.

Working with arrays and array elements

- An array element (like scores[2]):
- 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 15

Example: Processing arrays

Computing the average of an array of scores:



- 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;
};
                                     Defines new variables
Student student1, student2;
```

```
student1.name = "John Smith";
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;</pre>
 - input: cin >> student2;
 - Must do these member by member

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

21

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

Example: Overloaded functions

```
double calcWeeklyPay (int hours, double payRate) {
   return hours * payRate;
}
double calcWeeklyPay (double annSalary) {
   return annSalarv / 52:
}
                                 Output:
                                 Enter hours worked and pay rate: 37 19.5
                                 Pav is: 721.5
int main () {
                                 Enter annual salary: 75000
                                Pay is: 1442.31
   int h;
   double r;
   cout << "Enter hours worked and pay rate: ";
   cin >> h >> r;
   cout << "Pay is: " << calcWeeklyPay(h,r) << endl;</pre>
   cout << "Enter annual salary: ";</pre>
   cin >> r:
   cout << "Pay is: " << calcWeeklyPay(r) << endl;</pre>
   return 0;
                                                           23
```

Overloaded Functions

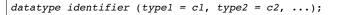
- <u>Overloaded functions</u> have the same name but different parameter lists.
- The parameter lists of each overloaded function must have different types and/or number of parameters.
- Compiler will determine which version of the function to call by matching arguments to parameter lists

22

24

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:



* 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>
```

25

• 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
```