Ch 6(+7) Functions
Part 4

CS 1428
Fall 2011

Jill Seaman

Lecture 23

Functions and Arrays

- An **array element** can be passed to any parameter with the same type:

```
double square (double);

int main() {
   double numbers[5] = {2.2, 3.3, 5.11, 7.0, 3.2};

   for (int i=0; i<5; i++)
      cout << square(numbers[i]) << " ";
   cout << endl;
   return 0;
}

double square (double x) {
   return x * x;
}
```

Output:
4.84 10.89 26.1121 49 10.24

Functions and Arrays

- An **array element** can be passed by Reference. What is output by this program?

```
void changeMe(int &);

int main() {
   int numbers[5] = {2, 3, 5, 7, 3};

   for (int i=0; i<5; i++)
      changeMe(numbers[i]);

   for (int i=0; i<5; i++)
      cout << numbers[i] << " ";
   cout << endl;
}
void changeMe(int &myValue) {
   myValue = 200;
}
```


Passing arrays to functions

- An **array** can be passed to a function that has an array parameter

```
void showArray(int[], int);

int main() {
   int numbers[5] = {2, 3, 5, 7, 3};
   showArray(numbers,5);
   return 0;
}

void showArray(int values[], int size) {
   for (int i=0; i<5; i++)
      cout << values[i] << " ";
   cout << endl;
}
```

Output:
2 3 5 7 3

Passing arrays to functions

- In the function definition, the parameter type is a variable name with an empty set of brackets: [ ]
  - Do NOT give a size for the parameter

```
void showArray(int values[], int size)
```

- In the prototype, empty brackets go after the element datatype.

```
void showArray(int[], int)
```

- In the function call, use the variable name for the array.

```
showArray(numbers, 5)
```


Passing arrays to functions

- Usually functions that take an array argument also take the size of the array as an argument, so they know how many elements to process.

- The size parameter is just a regular int parameter and must be listed in the parameter list and included in the function call.

Passing arrays to functions

- An array is **always** passed by reference.
- The parameter name is an alias to the array being passed in, even though it has no &.
- Changes made to the array (elements) inside the function affect the array in the function call.

Passing arrays to functions

- Changing an array inside a function:

```
void incrArray(int[], int);
void showArray(int[], int);

int main() {
   int numbers[5] = {2, 3, 5, 7, 3};
   incrArray(numbers,5);
   showArray(numbers,5);
   return 0;
}

void incrArray(int values[], int size) {
   for (int i=0; i<5; i++)
      (values[i])++;          //values[i]=values[i]+1;
}
```

Output:
3 4 6 8 4

Overloading Functions

- Overloaded functions have the same name but different parameter lists.
- Used to create functions that perform the same task over different sets of arguments.
- 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 based on arguments and parameter lists

Example: Overloaded functions

```
// http://www.cplusplus.com/doc/tutorial/functions2/
#include <iostream>
using namespace std;

int operate (int a, int b) {
  return (a*b);
}

float operate (float a, float b) {
  return (a/b);
}

int main () {
  int x=5,y=2;
  float n=5.0,m=2.0;
  cout << operate (x,y) << endl;
  cout << operate (n,m) << endl;
  return 0;
}
```

Output:
10
2.5

Example: Overloaded functions

```
double calcWeeklyPay (int hours, double payRate) {
   return hours * payRate;
}
double calcWeeklyPay (double annSalary) {
   return annSalary / 52;
}


int main () {
   int h;
   double r;
   cout << "Enter hours worked and pay rate: ";
   cin >> h >> r;
   cout << "Pay is: " << calcWeeklyPay(h,r) << endl;
   cout << "Enter annual salary: ";
   cin >> r;
   cout << "Pay is: " << calcWeeklyPay(r) << endl;
   return 0;
}
```

Output:
Enter hours worked and pay rate: 37 19.5
Pay is: 721.5
Enter annual salary: 75000
Pay is: 1442.31

Example: Overloaded functions

prototypes

- Different number of arguments:

```
int sum (int, int);
int sum (int, int, int);
int sum (int, int, int, int);
```

- More common usage: convert to a type

```
string getStr (int);
string getStr (double);
string getStr (char);
...
```

Default Arguments

- A default argument is a value passed to the parameter when the argument is left out of the function call.
- The default argument is usually listed in the function prototype:

```
int showArea (double = 20.0, double = 10.0);
```

- Default arguments are literals (or constants) with an = in front of them, occurring after the data types listed in a function prototype

Default Arguments

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

- 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
```

Example: Default Arguments

```
void displayStars(int = 10, int = 1);

int main () {
   displayStars();      // uses 10 x 1
   cout << endl;
   displayStars(5);     // uses 5 x 1
   cout << endl;
   displayStars(7, 3); // uses 7 x 3
   return 0;
}

void displayStars(int cols, int rows) {
   for (int down = 0; down < rows; down++) {
      for (int across = 0; across < cols; across++)
         cout << "*";
      cout << endl;
   }
}
```

Output:

```
**********

*****

*******
*******
*******
```


Default Arguments

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

```
displayStars(5);     // uses 5 x 1
displayStars( ,7);   // NO, won't work for 10 x 7
```

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

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

Default Arguments

- Default arguments are like overloaded functions

```
void displayStars(int = 10, int = 1);
```

- is like declaring 3 overloaded functions:

```
void displayStars();          // uses 10 and 1
void displayStars(int);       // uses arg and 1
void displayStars(int, int); // uses arg1 and arg2
```


Stubs and Drivers

- Useful for testing and debugging program and function logic and design
- Stub: A dummy function used in place of an actual function as a temporary placeholder
- Usually displays a message indicating it was called. May also display parameters

```
void processList(int values[], int size) {
   cout << "Inside ProcessList, unfinished.\n";
}
```

Stubs and Drivers

- Driver: A function that tests another function by calling it, usually with constant values
- Various arguments are passed and return values are tested
- Usually no input from user

```
int main() {
   int testList1[] = {3,4,2,6,7,10};
   int testList2[] = {};
   processList(testList1,6);
   showArray(testList1,6);
   processList(testList2,0);
   //don't need to show an empty list
}
```