

605.404 Object-Oriented Programming with C++
Summer 2007
Lecture 06

6	06/26	Functions	S: Chaps 7
---	-------	-----------	------------

Additional Resources Related to This Lecture:

LECTURE:

Stroustrup Chapter 7 Functions

- I. 7.1 Function Declarations
 - A. signature {return type, function name, {formal parameters}*}
 - B. note that programming languages cannot distinguish polymorphic functions by return type only
 - 1. int dog()
 - 2. char dog()
 - C. compiler ignores names in function prototype definition
 - D. 7.1.1 Function Definitions
 - 1. unused arguments
 - 2. inline functions
 - E. 7.1.2 Static Variables
 - 1. allocation of memory that is available and referenced for all function calls
 - 2. initialized only the first time the thread of execution reaches its definition
- II. 7.2 Argument Passing
 - A. identical to semantics of initialization
 - B. type of actual argument checked against type of corresponding formal argument; all standard and user-defined type conversions are performed
 - C. pass-by-value and pass-by-reference actions
 - D. 7.2.1 Array Arguments
 - 1. an argument for an entire array is neither a call-by-value nor a call-by-reference argument. It is an array argument
 - 2. three parts:
 - a) the address (location in memory) of the first indexed variable
 - b) the base type of the array (which determines how much memory each indexed variable uses)
 - c) the size of the array (the number of indexed variables)
 - 3. first two parts are in the argument definition as in int array[] but the size usually needs an additional formal parameter/actual argument pair as in int array_size

4. two dimensional example

```
//prototype
void print_assignments (const CELL array[][MAX_SIZE], const int size);
//call
print_assignments (problem, problem_size); //function
void print_assignments (const CELL array[][MAX_SIZE], const int size) {
    int i,
        j;
    assert(size != 0);
    for (i = 1; i <= size; i++) {
        cout << "\n Worker " << setw(2) << i << " assigned job ";
        for(j = 1; j <= size; j++)
            if ((array[i][j].value == 0) &&
                (array[i][j].allocated))
                cout << setw(2) << j;
    }
    return;
}
```

III. 7.3 Value Return

- A. recursive functions
- B. return statement is considered to initialize an unnamed variable of the returned type
- C. can use a call of a void function as a return parameter (see Templates later)

IV. 7.4 Overloaded Function Names

- A. overloading a method means providing more than one method with the same name in the same class with different signatures to differentiate them
- B. overriding a method means replacing the superclass's implementation of a method with a method in the current class; signatures must be identical
- C. rules for mapping call to function page 149
- D. 7.4.1 Overloading and Return Type
 - 1. not considered in overload resolution
- E. 7.4.2 Overloading and Scopes
 - 1. use of namespaces or using-declarations or using-directives
- F. 7.4.3 Manual Ambiguity Resolution
- G. 7.4.4 Resolution for Multiple Arguments

V. 7.5 Default Arguments

- A. a very important part of signature planning
- B. a default argument cannot be repeated or changed in a subsequent declaration in the same scope

VI. 7.6 Unspecified Number of Arguments

- A. only when the number of arguments AND the type of arguments vary is the ellipsis necessary

VII. 7.7 Pointer to Function

- A. pointers to functions have argument types declared just like the functions themselves
- B. there is no implicit conversion of argument or return types when pointers to functions are assigned or initialized

VIII. 7.8 Macros

- A. cannot be overloaded

- B. cannot be recursive
- C. enclose arguments in ()
- D. use /* */ comment form
- E. 7.8.1 Conditional Compilation