ITM 312, Fall 2013

Chapter 6 Lecture Notes

6.1. Modular Programming

a. Modular programming - breaking a program up into smaller, manageable functions or modules
b. Function - a collection of statements to perform a task
c. Why use modular programming?
    i. Improves maintainability of programs
    ii. Simplifies the process of writing programs

6.2. Defining and Calling Functions

a. Function call - statement causes a function to execute
    i. `main()` can call any number of functions
    ii. Functions can call other functions
    iii. Compiler must know the following about a function before it is called:
        1. name
        2. return type
        3. number of parameters
        4. data type of each parameter
b. Function definition - statements that make up a function; includes:
    i. Return type - data type of the value that function returns to the part of the program that called it
    ii. Name - name of the function. Function names follow same rules as variables
    iii. Parameter list - variables containing values passed to the function
    iv. Body - statements that perform the function's task, enclosed in {}
c. Function return type
    i. If a function returns a value, the type of the value must be indicated: `int main()`
    ii. If a function does not return a value, its return type is void:

```
void printHeading()
{
```

```
        cout << "Monthly Sales\n";
    }
```

d. Calling a function
    i. To call a function, use the function name followed by `()` and `;`
    ```
    printHeading();
    ```
    ii. When called, program executes the body of the called function
    iii. After the function terminates, execution resumes in the calling function at point of call.

6.3. Function Prototypes

a. Ways to notify the compiler about a function before a call to the function:
    i. Place function definition before calling function's definition
    ii. Use a function prototype (function declaration) – like the function definition without the body

b. Syntax:
    i. Header: `void printHeading(datatype varname) {}`
    ii. Prototype: `void printHeading(datatype);`

c. Place prototypes near top of program

d. Program must include either prototype or full function definition before any call to the function – compiler error otherwise

e. When using prototypes, can place function definitions in any order in source file

6.4. Sending Data into a Function

a. Can pass values into a function at time of call: `c = pow(a, b);`

b. Values passed to function are arguments

c. Variables in a function that hold the values passed as arguments are parameters
    i. Example:
    ```
    void displayValue(int num)
    {
       cout << "The value is " << num << endl;
    }
    ```

d. Function call notes:
    i. Value of argument is copied into parameter when the function is called

ii. A parameter's scope is the function which uses it
iii. Function can have multiple parameters
iv. There must be a data type listed in the prototype () and an argument declaration in the function header () for each parameter
v. Arguments will be promoted/demoted as necessary to match parameters
vi. When calling a function and passing multiple arguments:
    1. the number of arguments in the call must match the prototype and definition
    2. the first argument will be used to initialize the first parameter, the second argument to initialize the second parameter, etc.

6.5. Passing Data by Value

a. Pass by value: when an argument is passed to a function, its value is copied into the parameter.
b. Changes to the parameter in the function do not affect the value of the argument

6.6. Using Functions in Menu-Driven Programs

a. Use functions to implement user choices from menu
b. Use functions to implement general-purpose tasks:
c. Higher-level functions can call general-purpose functions, minimizing the total number of functions and speeding program development time

6.7. The `return` Statement

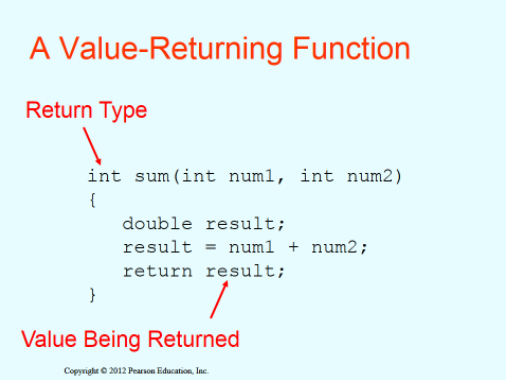
a. Used to end execution of a function
b. Can be placed anywhere in a function
    i. Statements that follow the return statement will not be executed
c. Can be used to prevent abnormal termination of program
d. In a void function without a return statement, the function ends at its last `}`
e. Expressions can be included in the return statement

6.8. Returning a Value from a Function

a. A function can return a value back to the statement that called the function

b. In a value-returning function, the return statement can be used to return a value from function to the point of call.

c. Syntax:



6.9. Returning a Boolean Value

a. Function can return true or false

b. Declare return type in function prototype and heading as `bool`

c. Function body must contain return statement(s) that return true or false

d. Calling function can use return value in a relational expression

e. Used most commonly in `if()` statements to evaluate a condition

6.10. Local and Global Variables

a. Local Variables

i. *Variables defined inside a function are local to that function.* They are hidden from the statements in other functions, which normally cannot access them.

ii. Because the variables defined in a function are hidden, *other functions may have* separate, distinct *variables with the same name*.

iii. Lifetime of a local variable: A function's local variables exist only while the function is executing

iv. Any value stored in a local variable is lost between calls to the function in which the variable is declared.

b. Global Variables

i. A global variable is any variable defined outside all the functions in a program.

ii. The scope of a global variable is the portion of the program from the variable definition to the end.

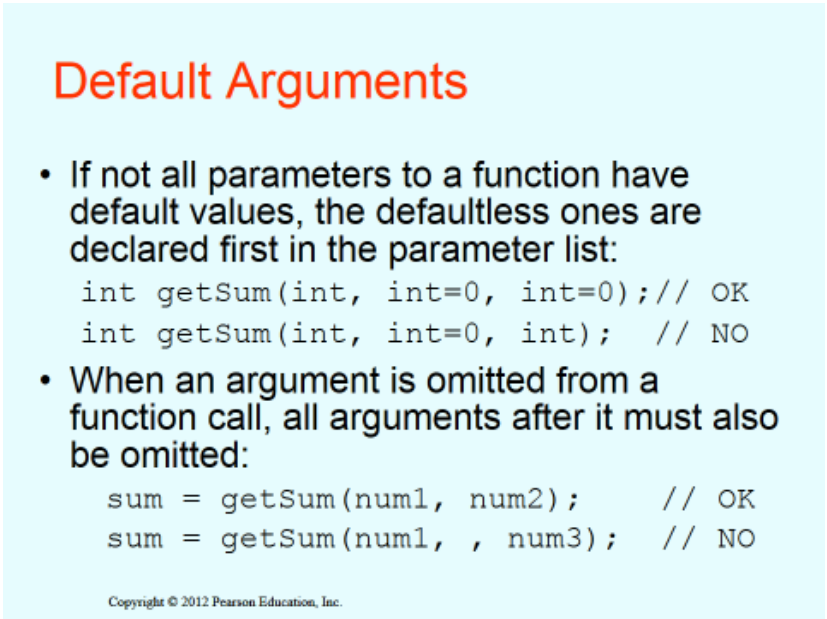
iii. This means that a global variable can be accessed by *all* functions that are defined after the global variable is defined.
iv. Avoid using global variables except for constants
v. Global variables (not constants) are automatically initialized to 0 (numeric) or NULL (character) when the variable is defined. (local vars are not automatically initialized)

6.11. Static Local Variables

a. Local variables only exist while the function is executing. When the function terminates, the contents of local variables are lost.
b. `static` local variables retain their contents between function calls.
c. `static` local variables are defined and initialized only the first time the function is executed. 0 is the default initialization value.

6.12. Default Arguments

a. A Default argument is an argument that is passed automatically to a parameter if the argument is missing on the function call.
b. Must be a constant declared in prototype: `void evenOrOdd(int = 0);`
c. Can be declared in header if no prototype
d. Multi-parameter functions may have default arguments for some or all of them: `int getSum(int, int=0, int=0);`
e.



6.13. Using Reference Variables as Parameters

a. A reference variable is an alias for another variable

b. Defined with an ampersand (&): `void getDimensions(int&, int&);`
c. Changes to a reference variable are made to the variable it refers to
d. Use reference variables to implement passing parameters by reference
e. Each reference parameter must contain &
f. Space between type and & is unimportant
g. Must use & in both prototype and header
h. Argument passed to reference parameter must be a variable – cannot be an expression or constant
i. Use when appropriate – don't use when argument should not be changed by function, or if function needs to return only 1 value

6.14. Overloading Functions

a. Overloaded functions have the same name but different parameter lists
b. Can be used to create functions that perform the same task but take different parameter types or different number of parameters
c. Compiler will determine which version of function to call by argument and parameter lists
d. Example:

Function Overloading Examples

Using these overloaded functions,

```
void getDimensions(int);              // 1
void getDimensions(int, int);         // 2
void getDimensions(int, double);      // 3
void getDimensions(double, double);// 4
```

the compiler will use them as follows:

```
int length, width;
double base, height;
getDimensions(length);              // 1
getDimensions(length, width);       // 2
getDimensions(length, height);      // 3
getDimensions(height, base);        // 4
```


6.15. The `exit()` Function

a. Terminates the execution of a program

b. Can be called from any function
c. Can pass an int value to operating system to indicate status of program termination
d. Usually used for abnormal termination of program
e. Requires `cstdlib` header file
    i. The `cstdlib` header defines two constants that are commonly passed, to indicate success or failure: `exit(EXIT_SUCCESS);` / `exit(EXIT_FAILURE);`
f. Example: `exit(0);`

6.16. Stubs and Drivers

a. Useful for testing and debugging program and function logic and design
b. Stub: A dummy function used in place of an actual function
    i. Usually displays a message indicating it was called. May also display parameters
    ii. Example: `int multiply(int x, int y) { return 0; }`
c. Driver: A function that tests another function by calling it
    i. Various arguments are passed and return values are tested
    ii. Example:

```
int big = 20;
int small = 10;
int answer = max(big, small);
if (answer == big) return true;
else return false;
```