Modular Programming
2/16/01 Lecture #5  16.070

• Outline
  ➢ Need for, and benefits of, modular programs
  ➢ How to design modular programs
  ➢ Using functions to build modular programs
Modular Programming - Need and Approach

- Typical industry programming projects consist of thousands of lines of code or more

- One huge monolithic program would be unmanageable, incomprehensible, difficult to design, write, debug, test, etc.

- Modular Programming: Divide-and-Conquer approach to programming
  - Divide into sub-problems
  - Size of modules reduced to humanly comprehensible and manageable level

- Analogous approaches: manufacturing a spacecraft, or a car
  - Manufacture individual components, test separately
  - Assemble into larger components, integrate and test
Modular Programming - Process

- Late in the conceive phase, program is divided into small, independent modules that are separately named and individually invokeable program elements
  - Partitioning based on
    - Intensity of information exchange among internal steps
    - Likelihood that internal steps will change
    - Hiding interfaces to hardware and humans
- Modules are designed, written, tested, debugged by individuals or small teams
  - Allows for multiple programmers to work in parallel
- Modules are integrated to become a software system that satisfies the problem requirements
  - To integrate successfully, original decision must be good and interfaces between modules must be correct
Modular Programming - Process

- Requirements
- Conceive
  - module 1 spec
  - module 2 spec
  - module 3 spec
- Design
  - module 1 design
  - module 2 design
  - module 3 design
- Build, Test
  - build, test 1
  - build, test 2
  - build, test 3
- Integrate & Test
  - module code 1
  - module code 2
  - module code 3
- Final Code
Modular Program Example

• **Problem/Goal**: Design and implement a program that moves Robbie the Robot based on user input from keyboard

• **Requirements Specification**: Develop a program that does the following:
  ➢ Display a menu of choices for robot motion
  ➢ Read keyboard input
    – Input can be any one of 4 arrows showing movement up, down, left, right
    – Input other than the arrows terminates the program.
  ➢ Move Robbie based on keyboard input
Modular Program Example - cont.

• **Analysis:** Input = A value for keyboard input; Output = Direction of robot movement

• **Design:** Pseudo-code algorithm
  
  Print menu
  Read request
  If request is ↑ move robot forward
  Else if request is ← move robot left
  Else if request is ⇒ move robot right
  Else if request is ↓ move robot backward
  Else print "Terminating program."
  End if
Modular Program Example - cont.

- Algorithm contains two sub-problems
  - Print menu and read request -- based on interfacing with humans
  - Move Robot -- based in computation/interfacing with robot

- Perform top-down stepwise refinement
  - Define algorithms for sub-problems
  - Refine them - may need to be broken down into sub-sub-problems
  - At lowest level, assign name to each module and combine named modules into higher level sub-problem algorithm (e.g., Move Robot => move_robot)
Structure Chart for Top-Level Pseudo-code Algorithm

```
main_program

print_menu
move_robot
```
Pseudo-code for print-menu function

- Print_menu:

  print "This program moves Robbie the Robot based on user input."
  print "Enter ↑ to move Robbie forward"
  print "Enter ← to move Robbie left"
  print "Enter → to move Robbie right"
  print "Enter ↓ to move Robbie backward"
  read request
Advantages of Modular Programming

• Manageable: Reduce problem to smaller, simpler, humanly comprehensible problems

• Divisible: Modules can be assigned to different teams/programmers
  ➢ Enables parallel work, reducing program development time
  ➢ Facilitates programming, debugging, testing, maintenance

• Portable: Individual modules can be modified to run on other platforms

• Re-usable: Modules can be re-used within a program and across programs
Using Functions to Build Modular Programs

• In C, modules are implemented as Functions

• A function is a block of code that performs a specific task

• The role of a Function is to
  ➢ Accept input (optional)
  ➢ Perform a task
  ➢ Return output (optional)

• Functions are identified by unique, programmer-defined names
Function Elements

• Function Definition - used to define/implement a function

• Function Calls - used to invoke/run a function
  ➢ Passing data between functions

• Function Declarations/Prototypes - used to identify a function to the compiler prior to calling the function
Function Definitions consist of

• Function type
  ➢ Specified in the header, which is the first line of the Function
  ➢ Identifies the type of value to be returned; e.g., integer
  ➢ If no value is to be returned, type = void

• Function name - main or unique user-defined

• (Optional) List of parameters enclosed in parentheses (void if none)

• Function body - variable declarations/statements to express algorithm, in brackets
Example of a Function Definition

• Simple function that prints menu to screen
  
  ```c
  void print_menu(void)
  {
      printf("This program ...;
      printf("Enter ↑ to move ...;
      ...
  }
  }
  ```

• Add in the read request
  
  ```c
  char print_menu(void)
  {
      char request;
      printf("This program ...;
      printf("Enter ↑ to move ...;
      ...
      scanf("%c", &request);
      return request;
  }
  ```
Function Calls

• To run a function, it must be called by another function

• To call a function
  - List name of function e.g., `print_menu()`, `move_robot(user_input)`
  - List of actual parameters/arguments, if any, enclosed in parentheses

• When a function is called
  - Program control passes to called function
  - Code corresponding to called function is executed
  - Function code is kept in separate area of main memory during execution
  - After function body completes execution, program control returns to calling function
Parameter Passing

- **Arguments** can be used to supply input to a function

- Values of actual parameters are copied to memory locations of corresponding formal parameters in function's data area

  - Call function `move_robot(user_input)`
  - Value of `user_input` copied into `direction`

- After function completes, program control returned to calling function

  - If function was to return value, value is returned
  - Called function cannot access memory location of actual parameters
int main(void)
{
    char user_input;
    ...
    user_input = print_menu();
    move_robot(user_input);
    ...
}

void move_robot(char direction)
{
    ...
    /* move robot in direction */
    /* specified by passed */
    /* argument */
    ...
}
Passing Data Between Functions

- Data can be passed between functions in three ways
  
  ➢ Input to a function: Parameters
    - Used to send values to function
    - Parameters are variables declared in formal parameter list of function header:
      ```c
      void move_robot(char direction);
      ```
    - Calling function can send data (actual parameters/arguments) to called function:
      ```c
      move_robot(user_input);
      ```
    - One-to-one correspondence between actual and formal parameters
  
  ➢ Output from a function: return a value from a function

  ➢ Global variables, declared in source file outside/before function defs
Function Declarations  (also called Function Prototypes)

- A function must be declared before it can be called
- Notifies the compiler that you intend to define and use this function
  - Called function will have a definition consistent with prototype
  - List of parameters/arguments, if any, enclosed in parentheses
- Consists of
  - Function type  e.g.,  `char print_menu(void);`
  - Function name
  - List of function parameter types enclosed in parentheses
  - Terminating semicolon  e.g.,  `void move_robot(char direction);`
Scope of Functions

- Function Declaration/Prototype defines region in program in which function may be used by other functions
  
  ➢ Global prototypes: Function prototype placed outside function definitions
    - Scope begins where prototype is placed and extends to end of source file
    - Any function in program may use it
    - Usually appear before definition of function `main`
  
  ➢ Local prototypes: Function prototype placed in function definitions
    - Scope begins where prototype is placed and extends to end of function in which it appears
    - Must have calls to the declared function in the parent function
Style in Modular Programming

• Step-wise refinement until expressible in 1-2 short English sentences
• Optimize module size. Rule of thumb: 2-50 lines
• Restrict number of functions called by a function. Rule of thumb: ~7
• Assign descriptive names to functions that reflect purpose of function
• In comments before function definition header, clearly identify all I/O
• To pass data, use parameters instead of global variables
• Use function prototypes uniformly
• Design functions that can be used in other programs
• Use functions that have already been defined and tested
Summary

• Today we learned
  ➢ What modular programming is
  ➢ Why we want to design modular programs
  ➢ How to implement modular programs using functions

• Readings:
  ➢ Review chapter C6 with this lecture
  ➢ For Tuesday (Monday classes), read C5.1-C5.8 on Variables and Operators.
  ➢ For Wednesday, read C4
  ➢ For Friday, read C9

• Note: Material for Weeks 3 and 4 have been swapped to provide you with more programming tools

• Errata: 16.070 TA office is 33-112