Chapter 3

Function Basics
Learning Objectives

- Predefined Functions
  - Those that return a value and those that don’t
- Programmer-defined Functions
  - Defining, Declaring, Calling
  - Recursive Functions
- Scope Rules
  - Local variables
  - Global constants and global variables
  - Blocks, nested scopes
Introduction to Functions

- Building Blocks of Programs
- Other terminology in other languages:
  - Procedures, subprograms, methods
  - In C++: functions
- I-P-O
  - Input – Process – Output
  - Basic subparts to any program
  - Use functions for these ‘pieces’
Predefined Functions

- Libraries full of functions for our use!
- Two types:
  - Those that return a value
  - Those that do not (void)
- Must ‘#include’ appropriate library
  - e.g.:
    - `<cmath>`, `<cstdlib>` (Original ‘C’ libraries)
    - `<iostream>` (for cout, cin)
Using Predefined Functions

- Math functions very plentiful
  - Found in library `<cmath.h>`
  - Most return a value (the ‘answer’)

- Example: theRoot = sqrt(9.0);

- Components:
  - `sqrt` = name of library function
  - `theRoot` = variable used to assign ‘answer’ to
  - `9.0` = argument or ‘starting input’ for function

- In I-P-O:
  - I = 9.0
  - P = ‘compute the square root’
  - O = 3, which is returned & assigned to `theRoot`
The Function Call

- Back to this assignment:
  
  \[ \text{theRoot} = \sqrt{9.0}; \]

  - The expression \( \sqrt{9.0} \) is known as a function call, or function invocation
  - The argument in a function call \( (9.0) \) can be a literal, a variable, or an expression
  - The call itself can be part of an expression:
    - \( \text{bonus} = \sqrt{\text{sales}}/10; \)
    - A function call is allowed wherever it’s legal to use an expression of the function’s return type
A Larger Example

Display 3.1, page 94

```
// A Predefined Function That Returns a Value

1  // Computes the size of a doghouse that can be purchased
2  // given the user's budget.
3  #include <iostream>
4  #include <cmath>
5  using namespace std;

6  int main( )
7  {
8      const double COST_PER_SQ_FT = 10.59;
9      double budget, area, lengthSide;

10     cout << "Enter the amount budgeted for your doghouse $";
11     cin >> budget;

12     area = budget/COST_PER_SQ_FT;
13     lengthSide = sqrt(area);

14     cout.setf(ios::fixed);
15     cout.setf(ios::showpoint);
16     cout.precision(2);
17     cout << "For a price of $" << budget << endl
18       << "I can build you a luxurious square doghouse\n"
19       << "that is " << lengthSide
20       << " feet on each side.\n";

21     return 0;
22  }

SAMPLE DIALOGUE

Enter the amount budgeted for your doghouse $25.00
For a price of $25.00
I can build you a luxurious square doghouse
that is 1.54 feet on each side.
```
More Predefined Functions

- `#include <cstdlib>`
  - Library contains functions like:
    - `abs()` // Returns absolute value of an int
    - `labs()` // Returns absolute value of a long int
    - `*fabs()` // Returns absolute value of a float
  - `*fabs()` is actually in library `<cmath>`!
    - Can be confusing
    - Remember: libraries were added after C++ was ‘born’, in incremental phases
    - Refer to appendices/manuals for details
More Math Functions

- **pow(x, y)**
  - Returns x to the power y
  ```
  double result, x = 3.0, y = 2.0;
  result = pow(x, y);
  cout << result;
  ```
  - Here 9.0 is displayed since $3.0^{\, 2.0} = 9.0$

- Notice this function receives two arguments
  - A function can have any number of arguments, of varying data types
### Even More Math Functions

#### Display 3.2 Some Predefined Functions

<table>
<thead>
<tr>
<th>NAME</th>
<th>DESCRIPTION</th>
<th>TYPE OF ARGUMENTS</th>
<th>TYPE OF VALUE RETURNED</th>
<th>EXAMPLE</th>
<th>VALUE</th>
<th>LIBRARY HEADER</th>
</tr>
</thead>
<tbody>
<tr>
<td>sqrt</td>
<td>Square root</td>
<td>double</td>
<td>double</td>
<td>sqrt(4.0)</td>
<td>2.0</td>
<td>cmath</td>
</tr>
<tr>
<td>pow</td>
<td>Powers</td>
<td>double</td>
<td>double</td>
<td>pow(2.0, 3.0)</td>
<td>8.0</td>
<td>cmath</td>
</tr>
<tr>
<td>abs</td>
<td>Absolute value for int</td>
<td>int</td>
<td>int</td>
<td>abs(-7)</td>
<td>7</td>
<td>cstdlib</td>
</tr>
<tr>
<td>labs</td>
<td>Absolute value for long</td>
<td>long</td>
<td>long</td>
<td>labs(-70000) labs(70000)</td>
<td>70000</td>
<td>70000</td>
</tr>
<tr>
<td>fabs</td>
<td>Absolute value for double</td>
<td>double</td>
<td>double</td>
<td>fabs(-7.5) fabs(7.5)</td>
<td>7.5</td>
<td>cmath</td>
</tr>
<tr>
<td>ceil</td>
<td>Ceiling (round up)</td>
<td>double</td>
<td>double</td>
<td>ceil(3.2) ceil(3.9)</td>
<td>4.0</td>
<td>4.0</td>
</tr>
<tr>
<td>floor</td>
<td>Floor (round down)</td>
<td>double</td>
<td>double</td>
<td>floor(3.2) floor(3.9)</td>
<td>3.0</td>
<td>3.0</td>
</tr>
<tr>
<td>exit</td>
<td>End program</td>
<td>int</td>
<td>void</td>
<td>exit(1);</td>
<td>None</td>
<td></td>
</tr>
<tr>
<td>rand</td>
<td>Random number</td>
<td>None</td>
<td>int</td>
<td>rand( )</td>
<td>Varies</td>
<td></td>
</tr>
<tr>
<td>srand</td>
<td>Set seed for rand</td>
<td>unsigned int</td>
<td>void</td>
<td>srand(42);</td>
<td>None</td>
<td></td>
</tr>
</tbody>
</table>
Predefined Void Functions

- No returned value
- Performs an action, but sends no ‘answer’
- When called, it’s a statement itself
  - `exit(1);` // No return value, so not assigned
    - This call terminates program
    - void functions can still have arguments
- All aspects same as functions that ‘return a value’
  - They just don’t return a value!
Random Number Generator

- Return ‘randomly chosen’ number
- Used for simulations, games
  - rand()
    - Takes no arguments
    - Returns value between 0 & RAND_MAX
  - Scaling
    - Squeezes random number into smaller range
      rand() % 6
    - Returns random value between 0 & 5
  - Shifting
    rand() % 6 + 1
    - Shifts range between 1 & 6 (e.g.: die roll)
Random Number Seed

- Pseudorandom numbers
  - Calls to `rand()` produce given ‘sequence’ of random numbers
- Use ‘seed’ to alter sequence

```c
srand(seed_value);
```
- void function
- Receives one argument, the ‘seed’
- Can use any seed value, including system time:
  ```c
  srand(time(0));
  ```
- `time()` returns system time as numeric value
- Library `<time>` contains `time()` functions
Random Examples

- Random double between 0.0 & 1.0:
  \[(\text{RAND\_MAX} - \text{rand()})/\text{static\_cast}\langle\text{double}\rangle(\text{RAND\_MAX})\]
  - Type cast used to force double-precision division

- Random int between 1 & 6:
  \[\text{rand()} \% 6 + 1\]
  - ‘\%’ is modulus operator (remainder)

- Random int between 10 & 20:
  \[\text{rand()} \% 10 + 10\]
Programmer-Defined Functions

- Write your own functions!
- Building blocks of programs
  - Divide & Conquer
  - Readability
  - Re-use
- Your ‘definition’ can go in either:
  - Same file as main()
  - Separate file so others can use it, too
Components of Function Use

- **3 Pieces to using functions:**
  - Function Declaration/prototype
    - Information for compiler
    - To properly interpret calls
  - Function Definition
    - Actual implementation/code for what function does
  - Function Call
    - Transfer control to function
Function Declaration

- Also called function prototype
- An ‘informational’ declaration for compiler
- Tells compiler how to interpret calls
  - Syntax:
    
    ```c
    <return_type> FnName(<formal-parameter-list>);
    ```
  - Example:
    ```c
    double totalCost( int numberParameter,
                     double priceParameter);
    ```
- Placed before any calls
  - In declaration space of main()
  - Or above main() in global space
Function Definition

- Implementation of function
- Just like implementing function main()
- Example:

```c
double totalCost(int numberParameter, double priceParameter)
{
    const double TAXRATE = 0.05;
    double subTotal;
    subtotal = priceParameter * numberParameter;
    return (subtotal + subtotal * TAXRATE);
}
```

- Notice proper indenting
Function Definition Placement

- Placed after function main()
  - NOT ‘inside’ function main()!
- Functions are ‘equals’; no function is ever ‘part’ of another
- Formal parameters in definition
  - ‘Placeholders’ for data sent in
    - ‘Variable name’ used to refer to data in definition
- return statement
  - Sends data back to caller
Function Call

- Just like calling predefined function
  bill = totalCost(number, price);
- Recall: totalCost returns double value
  - Assigned to variable named ‘bill’
- Arguments here: number, price
  - Recall arguments can be literals, variables, expressions, or combination
  - In function call, arguments often called ‘actual arguments’
  - Because they contain the ‘actual data’ being sent
Function Example

Display 3.5, page 105

```
#include <iostream>
using namespace std;

// Computes the total cost, including 5% sales tax, on numberParameter items at a cost of priceParameter each.

int main()
{
    double price, bill;
    int number;

cout << "Enter the number of items purchased: ";
cin >> number;
cout << "Enter the price per item $":
cin >> price;

    bill = totalCost(number, price);

cout.setf(ios::fixed);
cout.setf(ios::showpoint);
cout.precision(2);
cout << number << " items at 
" << "$ " << price << " each.
" << "Final bill, including tax, is $ " << bill
   << endl;

    return 0;
}

// Function definition

double totalCost(int numberParameter, double priceParameter)
{
    const double TAXRATE = 0.05; // 5% sales tax
    double subtotal;

    subtotal = priceParameter * numberParameter;
    return (subtotal + subtotal*TAXRATE);
}
```

Sample Dialogue

Enter the number of items purchased: 2
Enter the price per item: $10.10
2 items at $10.10 each.
Final bill, including tax, is $21.21
Alternative Function Declaration

- Recall: Function declaration is ‘information’ for compiler

- Compiler only needs to know:
  - Return type
  - Function name
  - Parameter list

- Formal parameter names not needed:
  double totalCost(int, double);
  Still ‘should’ put in formal parameter names
  Improves readability
Parameter vs. Argument

- Terms often used interchangeably
- Formal parameters/arguments
  - In function declaration
  - In function definition’s header
- Actual parameters/arguments
  - In function call
- Technically parameter is ‘formal’ piece while argument is ‘actual’ piece*
  - *Terms not always used this way
Functions Calling Functions

- We’re already doing this!
  - main() IS a function!
- Only requirement:
  - Function’s declaration must appear first
  - Function’s definition typically elsewhere
    - After main()’s definition
    - Or in separate file
- Common for functions to call many other functions
- Function can even call itself → ‘Recursion’
Boolean Return-Type Functions

- Return-type can be any valid type
  - Given function declaration/prototype:
    
    ```c
    bool appropriate(int rate);
    ```
  - And function’s definition:
    
    ```c
    bool appropriate (int rate)
    {
        return (((rate>=10)&&(rate<20))||(rate==0);
    }
    ```
  - Returns ‘true’ or ‘false’
  - Function call, from some other function:
    
    ```c
    if (appropriate(entered_rate))
        cout << “Rate is valid\n”;```
Declaring Void Functions

- Similar to functions returning a value
- Return type specified as ‘void’
- Example:
  - Function declaration/prototype:
    ```c
    void showResults(  double fDegrees,
                        double cDegrees);
    ```
    - Return-type is ‘void’
    - Nothing is returned
Declaring Void Functions

- Function definition:
  void showResults(double fDegrees, double cDegrees)
  {
    cout.setf(ios::fixed);
    cout.setf(ios::showpoint);
    cout.precision(1);
    cout << fDegrees << " degrees fahrenheit equals 
    " << cDegrees << " degrees celsius.\n";
  }

- Notice: no return statement
  Optional for void functions
Calling Void Functions

- Same as calling predefined void functions
- From some other function, like main():
  - showResults(degreesF, degreesC);
  - showResults(32.5, 0.3);
- Notice no assignment, since no value returned
- Actual arguments (degreesF, degreesC)
  - Passed to function
  - Function is called to ‘do it’s job’ with the data passed in
More on Return Statements

- Transfers control back to ‘calling’ function
- For return type other than void, MUST have return statement
- Typically the LAST statement in function definition
- return statement optional for void functions
- Closing } would implicitly return control from void function
Preconditions and Postconditions

- Similar to ‘I-P-O’ discussion
- Comment function declaration:
  ```
  void showInterest(double balance, double rate);
  //Precondition: balance is nonnegative account balance
  //    rate is interest rate as percentage
  //Postcondition: amount of interest on given balance,
  //    at given rate …
  ```
- Often called Inputs & Outputs
main(): ‘Special’

- Recall: main() IS a function
- ‘Special’ in that:
  - One and only one function called main() will exist in a program
- Who calls main()?
  - Operating system
  - Tradition holds it should have return statement
    - Value returned to ‘caller’ → Here: operating system
  - Should return ‘int’ or ‘void’
Scope Rules

- Local variables
  - Declared inside body of given function
  - Available only within that function
- Can have variables with same names declared in different functions
  - Scope is local: ‘that function is its scope’
- Local variables preferred
  - Maintain individual control over data
  - Need to know basis
  - Functions should declare whatever local data needed to ‘do their job’
Procedural Abstraction

- Need to know ‘what’ function does, not ‘how’ it does it!
- Think ‘black box’
  - Device you know how to use, but not it’s method of operation
- Implement functions like black box
  - User of function only needs: declaration
  - Does NOT need function definition
    - Called Information Hiding
    - Hide details of ‘how’ function does it’s job
Global Constants and Global Variables

- Declared ‘outside’ function body
  - Global to all functions in that file
- Declared ‘inside’ function body
  - Local to that function
- Global declarations typical for constants:
  - const double TAXRATE = 0.05;
  - Declare globally so all functions have scope
- Global variables?
  - Possible, but SELDOM-USED
  - Dangerous: no control over usage!
Blocks

- Declare data inside compound statement
  - Called a ‘block’
  - Has ‘block-scope’
- Note: all function definitions are blocks!
  - This provides local ‘function-scope’
- Loop blocks:
  for (int ctr=0; ctr<10; ctr++)
  {
    sum+=ctr;
  }
- Variable ctr has scope in loop body block only
Nested Scope

- Same name variables declared in multiple blocks
- Very legal; scope is ‘block-scope’
  - No ambiguity
  - Each name is distinct within its scope
Summary 1

- Two kinds of functions:
  - ‘Return-a-value’ and void functions
- Functions should be ‘black boxes’
  - Hide ‘how’ details
  - Declare own local data
- Function declarations should self-document
  - Provide pre- & post-conditions in comments
  - Provide all ‘caller’ needs for use
Summary 2

- **Local data**
  - Declared in function definition

- **Global data**
  - Declared above function definitions
  - OK for constants, not for variables

- **Parameters/Arguments**
  - **Formal**: In function declaration and definition
    - Placeholder for incoming data
  - **Actual**: In function call
    - Actual data passed to function