Introduction to Functions

- **Building blocks** of C/C++ programs
  - named differently in other languages. Ex: *procedures, subprograms, methods*
  - in C++, they are termed **functions**
- **I-P-O** of C/C++ functions
  - Input—Process—Output
  - use **functions** as basic subparts to any program
- Starting from **predefined** functions
  - What !? We’ve seen them before…

Predefined Functions (1/2)

- Libraries are full of functions for our use!
- Two types of functions:
  - that return a value (int, float, char,…)
  - that do not return a value (void)
- In *algebra*, a function is defined as a *rule* or *correspondence between values*, called the function’s *arguments*, and the unique value of the function associated with the arguments
- Ex: \( f(x) = 2x + 5 \), \( f(1) = 7 \), \( f(2) = 9 \), and \( f(3) = 11 \)
  - 1, 2, and 3 are *arguments*
  - 7, 9, and 11 are the *corresponding values*
Predefined Functions (2/2)

- Predefined functions are organized into separate libraries ⇒ require #include <XXX>
- Examples:
  - `<cmath>`: some Math functions in C
  - `<cstdlib>`: standard general utilities in C
  - `<iostream>`: for cout, cin and other I/O
- Some examples of the predefined Math functions in `<cmath>` are:
  - `sqrt(x)`
  - `pow(x,y)`
  - `floor(x)`

Power Function `pow(x, y)`

- Power function `pow(x, y)`
  - compute \( x^y \) (\( x \) to the power of \( y \))
- Ex:
  ```c
  float x = 3.0, y = 2.0;
  double result = pow(x, y);
  cout << result;
  ```
  ⇒ 9.0 is displayed since 3.0^2.0 = 9.0
- `pow(x, y)` returns a value of type `double`
- \( x \) and \( y \) are called the parameters of the function `pow(x, y)`
- Function `pow(x, y)` has two parameters

Square Root Function `sqrt(x)`

- Square root function `sqrt(x)`
  - compute the non-negative square root of \( x \), for \( x \geq 0.0 \)
  - return data of type `double`
  - require only one parameter
- Ex:
  ```c
  float x = 2.25;
  double result = sqrt(x);
  cout << result;
  ```
  1.5 is displayed since \( \sqrt{2.25} = 1.50 \)
- What happen if giving it a negative value?

Round Functions: `ceil(x)` & `floor(x)`

- Round functions
  - `ceil(x)` returns the smallest integral value not less than \( x \) ⇒ `ceil(x) ≥ x`
  - `floor(x)` returns the largest integral value not greater than \( x \) ⇒ `floor(x) ≤ x`
  - return data of type `double`
  - require only one parameter
- Ex: `float x = 49.50;`
  ```c
  double x_up = ceil(x);
  double x_dn = floor(x);
  cout << x_up << " vs " << x_dn ;
  ```
  display on screen 50 vs 49
More Predefined Functions

- Require `#include <cstdlib>`
- Library contains functions like:
  - `abs(x)` // return absolute value of an int
  - `labs(x)` // return absolute value of a long int
- `fabs(x)` returns absolute value of a `float` number
  - actually in library `<cmath>`
  - can be confusing
- Libraries were added after C++ was born, in incremental phases
  - refer to appendices/manuals for details

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
    - this call terminates the program
- void functions can still have arguments
- All aspects same as functions that "return a value"
  - just different in that they do not return a value!

Random Number Generator

- `rand()` picks a *randomly chosen* number
  - take no arguments
  - return value between 0 & `RAND_MAX`
  - `RAND_MAX` is *library-dependent* but at least 32767
- App#1: scaling `rand() % r`
  - squeeze random number into smaller range r. ex: `rand() % 6`
  - return a random value between 0 & `(r-1)`
- App#2: shifting `rand() % r + b`
  - shift the starting value from the base b. ex: `rand() % 6 + 1`
  - shift range between b & `(r-1+b)`

Pragmatic `rand()` & Buckets

- `rand()` returns only pseudo-random numbers
  - not really random
  - Ex: `rand() % 2` alternates between 0 and 1.
- `RAND_MAX` is not *evenly divisible* if n is large
  - some remainders appear more often than others ⇒ why? Hint: `RAND_MAX=32767`
  - Ex: `n=20000`, `Pr(10000) = 2*Pr(15000)`
- Divide `RAND_MAX` into *buckets* of equal sizes
  - some remainders appear more often than others ⇒ why? Hint: `RAND_MAX=32767`
  - Ex: `n=20000`, `Pr(10000) = 2*Pr(15000)`
- Divide `RAND_MAX` into *buckets* of equal sizes
  - `const int bucket = RAND_MAX / n;`
  - `do {`
    - `r = rand() / bucket;`
    - `drop 32767!!`
  - `} while (r >= n);`
Exercises of Using `rand()`

- (Ex 1) randomize an integer in $[0, 100)$
  
  ```
  const int bucket = RAND_MAX / 100;
  do {r = rand()/bucket;} while (r >= 100);
  ```

- (Ex 2) randomize an integer in $[11, 22)$
  
  ```
  const int bucket = RAND_MAX / 12;
  do {r = rand()/bucket;} while (r >= 12);
  r += 11;
  ```

- (Ex 3) randomize a double in $[0.0, 3.0]$:
  
  ```
  r = ((RAND_MAX – rand()) / static_cast<double>(RAND_MAX)) * 3.0;
  ```

Random Number Seed

- Pseudorandom numbers
  
  - a special “sequence“ of random numbers is defined when calling `rand()`
  
  ```
  srand(seed)
  ```

  - `srand(seed)` uses "seed" to alter the sequence of generating random numbers
  
  - a void function
  
  - require only one argument - the "seed"

  - seed can be any value, including system time
  
  ```
  EX: srand(1067); srand(time(NULL));
  ```

- `time()` returns system time as numeric value in library `<ctime>`

Even More Predefined Functions

<table>
<thead>
<tr>
<th>library</th>
<th>function</th>
<th>description</th>
<th>argument type</th>
<th>returned data type</th>
<th>example</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>cstdlib</td>
<td>abs</td>
<td>absolute value</td>
<td>int</td>
<td>int</td>
<td>abs(-7)</td>
<td>7</td>
</tr>
<tr>
<td>cstdlib</td>
<td>rand</td>
<td>random number</td>
<td>void</td>
<td>int</td>
<td>rand()</td>
<td>&lt;32768</td>
</tr>
<tr>
<td>cmath</td>
<td>cos</td>
<td>cosine</td>
<td>float/double</td>
<td>float/double</td>
<td>cos(60.0)</td>
<td>0.5</td>
</tr>
<tr>
<td>cmath</td>
<td>log</td>
<td>natural logarithm</td>
<td>float/double</td>
<td>float/double</td>
<td>log(5.5000)</td>
<td>1.70475</td>
</tr>
<tr>
<td>cmath</td>
<td>exp</td>
<td>exponential</td>
<td>float/double</td>
<td>float/double</td>
<td>exp(5.0000)</td>
<td>148.413</td>
</tr>
<tr>
<td>cctype</td>
<td>tolower</td>
<td>to lowercase letter</td>
<td>int</td>
<td>int</td>
<td>tolower('M')</td>
<td>m</td>
</tr>
<tr>
<td>cctype</td>
<td>toupper</td>
<td>to uppercase letter</td>
<td>int</td>
<td>int</td>
<td>toupper('F')</td>
<td>F</td>
</tr>
</tbody>
</table>

Learning Objectives

- **Predefined** functions
  
  - those that return a value and those that do not

- **Programmer-defined** functions
  
  - defining, declaring and calling
  
  - recursive functions

- **Scope** of variables
  
  - local variables
  
  - global constants and global variables
  
  - Blocks and nested scopes
Programmer-Defined Functions

- **Programmer-defined** functions
  - a.k.a. *user-defined* functions
  - now we can write our own functions!
- Building blocks of programs
  - divide & conquer
  - readability
  - re-use
- Your **definition(s)** can go in either:
  - the same file as main() resides, or
  - separate file(s) so others can use it (them) too

Using Programmer-Defined Functions

- Three pieces to using functions:
  1. **function declaration** (a.k.a. **prototype**)
     - information for compiler
     - to properly interpret calls
  2. **function definition**
     - actual implementation of C/C++ code
     for what function does
  3. **function call**
     - transfer control to the function

Piece 1: Function Declaration

- Function declaration (prototype)
  - an informational declaration for compiler
- Guide compiler how to interpret calls
  - Syntax:
    ```
    <return_type> FnName(<parameter-list>);
    ```
  - Example:
    ```
    double totalCost(int numberParameter, double priceParameter);
    ```
- Placed before any calls
  - in **declaration space** of main()
  - or above main() in **global space**

Piece 2: Function Definition

- Implementation of function
  - just like implementing function main()
- Ex:
  ```
  double totalCost(int iNumber, double dPrice) {
    const double cTAXRATE = 0.05;
    double dSubtotal;
    dSubtotal = dPrice * iNumber;
    return (dSubtotal * (1+cTAXRATE));
  }
  ```
- Notice **proper indentation**
Function Definition Placement

- Placed after function main() if its prototype is placed at front
  - 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
  - send data back to caller

Piece 3: Function Call

- Just like calling predefined function
  - Ex: dBill = totalCost(number, price);
- Recall: totalCost returns double value
  - assigned to variable named "dBill"
- Two arguments here: number and price
  - arguments can be literals, variables, expressions, or combination
  - in function call, arguments often as known as "actual arguments"
  \[ \Rightarrow \text{ they contain the "actual data" being sent} \]

Alternative Function Declaration

- Recall: function declaration only provides information or compiler
- Compiler only needs to know:
  - return type
  - function name
  - parameter list
- Formal parameter names not needed:
  - double totalCost(int, double);
  - "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 the 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:
    ```
    bool appropriate(int rate);
    ```
  - And function’s definition:
    ```
    bool appropriate(int rate)
    {
        return (((rate>=5)&&(rate<10))||(rate==0));
    }
    ```
  - Function return "true" or "false"
- Function call, from some other function:
  ```
  if (appropriate(input_rate))
      cout << "Rate is valid\n";
  ```

Declaring Void Functions

- Similar to functions returning a value
  - return type specified as "void"
- Example:
  ```
  Function declaration/prototype:
  void showResults(double fDegrees, double cDegrees);
  ```
  - return-type is "void"
  - nothing is really returned
- Function definition:
  ```
  void showResults(double fDegrees, double cDegrees){
      cout.setf(ios::fixed);  // Set fixed point output
      cout.setf(ios::showpoint); // Show point
      cout.precision(1); // Set precision
      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 its 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 is optional for void functions
  - Closing `}` would implicitly return control from void function

**Preconditions & Postconditions**

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

**Special Function `main()`**

- 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 (OS)
    - Tradition holds it should have return statement
    - `return value to "caller" which is the operating system`
    - `should return "int" or "void"`
Learning Objectives

- **Predefined** functions
  - those that return a value and those that do not
- **Programmer-defined** functions
  - defining, declaring and calling
  - recursive functions
- **Scope** of variables
  - local variables
  - global constants and global variables
  - Blocks and nested scopes

Scope of Variables

- **Scope**: section of program where identifier is valid (known or visible)
- **Local variables** (local scope): variables created inside a function or program component (structure)
  - meaningful only when used in expressions inside the function in which it was declared
- **Global variables** (global scope): variables created outside any function
  - can be used by all functions physically placed after global variable declaration

Rules of Scope

- 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"
- Local variable can only be members of **auto**, **static**, or **register** class
  - Refer to the textbook or online for details

Procedural Abstraction

- Need to know **what** function does, not **how** it does it!
  - think like a **black box**
  - the device you only know its usage, but not its details of operations
- Implement functions like black boxes
  - users only need function declarations
  - **NOT** need function definitions
- **Information hiding** ⇒ hide details about **how** a function does its job
Global Constants & 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 cTAXRATE = 0.05;`
  - declare globally so that all functions have the same scope
- Global variables?
  - possible, but *seldom used*
  - *dangerous* because no control over usage!

Blocks and Scopes

- Declare data **inside** compound statement
  - called a block `{ ... }`
  - has a *block scope*
- Note: all function definitions are blocks!
  - provides local *function* scope
- Loop blocks:
  ```c
  for (int ctr=0; ctr<10; ctr++) {
    sum+=ctr;
  }
  ```
  - variable `ctr` has scope in for loop body only

Nested Scope

- Variables with the same name can be declared in multiple blocks in C/C++
  - totally *legal*: due to block scope
  - no ambiguity and distinct/unique within its own scope
- Ex:
  ```c
  {   
    int var = 100; 
    ... 
    {   
      int var = x*y; 
    } 
  }
  ```

Summary (1/2)

- 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 (2/2)

- 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