Chapter 3

Function

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)

A Larger Example:
Display 3.1 A Predefined Function That Returns a Value (1 of 2)

Display 3.1 A Predefined Function That Returns a Value

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

int main( )
{
    const double COST_PER_SQ_FT = 10.50;
    double budget, area, lengthSide;

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

    area = budget/COST_PER_SQ_FT;
    lengthSide = sqrt(area);
```

A Larger Example:
Display 3.1 A Predefined Function That Returns a Value (2 of 2)

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

    return 0;
}
```

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.
```

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

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 prototoype
- An "informational" declaration for compiler
- Tells compiler how to interpret calls
  - Syntax:
    <return_type> FnName(<formal-parameter-list>);
  - Example:
    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:

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

- Notice proper indenting

Function Example:
Display 3.5 A User Defined Function (1 of 2)

Display 3.5 A Function Using a Random Number Generator

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

double totalCost(int numberParameter, double priceParameter);
//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);
```

Function declaration; also called the function prototype

Function call

Function Example:
Display 3.5 A User Defined Function (2 of 2)

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

    return 0;
}

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

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

Function head

Function body

Function definition

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

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 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 \n"
                << cDegrees << " degrees Celsius.\n";
}
```

- Notice: no return statement
  - Optional for void functions

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 it’s scope"
- Local variables preferred
  - Maintain individual control over data
  - Functions should declare whatever local data needed to "do their 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!

Parameters

- Two methods of passing arguments as parameters
- Call-by-value
  - "copy" of value is passed
- Call-by-reference
  - "address of" actual argument is passed

Call-by-Value Parameters

- Copy of actual argument passed
- Considered "local variable" inside function
- If modified, only "local copy" changes
  - Function has no access to "actual argument" from caller
- This is the default method
  - Used in all examples before Chapter 4

Call-by-Value Example: **Display 4.1** Formal Parameter Used as a Local Variable (1 of 3)

Display 4.1 Formal Parameter Used as a Local Variable

```
//Law office billing program.
#include <iostream>
using namespace std;

const double RATE = 150.00; //Dollars per quarter hour.

double fee(int hoursWorked, int minutesWorked);
//Returns the charges for hoursWorked hours and
//minutesWorked minutes of legal services.

int main( )
{
    int hours, minutes;
    double bill;
```

Call-by-Value Example: **Display 4.1** Formal Parameter Used as a Local Variable (2 of 3)

```
    cout << "Welcome to the law office of\n"
         << "Dewey, Cheatham, and Howe.\n"
         << "The law office with a heart.\n"
         << "Enter the hours and minutes"
         << " of your consultation:\n";
    cin >> hours >> minutes;

    bill = fee(hours, minutes);

    cout.setf(ios::fixed);
    cout.setf(ios::showpoint);
    cout.precision(2);
    cout << "For " << hours << " hours and " << minutes
         << " minutes, your bill is $" << bill << endl;

    return 0;
}
```

The value of **minutes** *is not changed by the call to* **fee**.

(continued)

Call-by-Value Example: **Display 4.1** Formal Parameter Used as a Local Variable (3 of 3)

Display 4.1 Formal Parameter Used as a Local Variable

```
double fee(int hoursWorked, int minutesWorked)
{
    int quarterHours;

    minutesWorked = hoursWorked*60 + minutesWorked;
    quarterHours = minutesWorked/15;
    return (quarterHours*RATE);
}
```

`minutesWorked` *is a local variable initialized to the value of* `minutes`.

SAMPLE DIALOGUE

Welcome to the law office of
Dewey, Cheatham, and Howe.
The law office with a heart.
Enter the hours and minutes of your consultation:
5 46
For 5 hours and 46 minutes, your bill is $3450.00

Call-By-Reference Parameters

- Used to provide access to caller's actual argument
- Caller's data can be modified by called function!
- Typically used for input function
  - To retrieve data for caller
  - Data is then "given" to caller
- Specified by ampersand, &, after type in formal parameter list

Call-By-Reference Example:
Display 4.1 Call-by-Reference Parameters (1 of 3)

Display 4.2 Call-by-Reference Parameters

```
//Program to demonstrate call-by-reference parameters.
#include <iostream>
using namespace std;

void getNumbers(int& input1, int& input2);
//Reads two integers from the keyboard.

void swapValues(int& variable1, int& variable2);
//Interchanges the values of variable1 and variable2.

void showResults(int output1, int output2);
//Shows the values of variable1 and variable2, in that order.

int main( )
{
    int firstNum, secondNum;

    getNumbers(firstNum, secondNum);
    swapValues(firstNum, secondNum);
    showResults(firstNum, secondNum);
    return 0;
}
```

Call-By-Reference Example:
Display 4.1 Call-by-Reference Parameters (2 of 3)

```
void getNumbers(int& input1, int& input2)
{
    cout << "Enter two integers: ";
    cin >> input1
        >> input2;
}

void swapValues(int& variable1, int& variable2)
{
    int temp;

    temp = variable1;
    variable1 = variable2;
    variable2 = temp;
}

void showResults(int output1, int output2)
{
    cout << "In reverse order the numbers are: "
         << output1 << " " << output2 << endl;
}
```

Call-By-Reference Example:
Display 4.1 Call-by-Reference Parameters (3 of 3)

Display 4.2 Call-by-Reference Parameters

SAMPLE DIALOGUE

```
Enter two integers: 5 6
In reverse order the numbers are: 6 5
```

Call-By-Reference Details

- What's really passed in?
- A "reference" back to caller's actual argument!
  - Refers to memory location of actual argument
  - Called "address", which is a unique number referring to distinct place in memory

Display 4.3 Comparing Argument Mechanisms

```
//Illustrates the difference between a call-by-value
//parameter and a call-by-reference parameter.
#include <iostream>
using namespace std;

void doStuff(int par1Value, int& par2Ref);
//par1Value is a call-by-value formal parameter and
//par2Ref is a call-by-reference formal parameter.

int main( )
{
    int n1, n2;

    n1 = 1;
    n2 = 2;
    doStuff(n1, n2);
    cout << "n1 after function call = " << n1 << endl;
    cout << "n2 after function call = " << n2 << endl;
    return 0;
}
```

```
void doStuff(int par1Value, int& par2Ref)
{
    par1Value = 111;
    cout << "par1Value in function call = "
           << par1Value << endl;
    par2Ref = 222;
    cout << "par2Ref in function call = "
           << par2Ref << endl;
}
```

Sample Dialogue

```
par1Value in function call = 111
par2Ref in function call = 222
n1 after function call = 1
n2 after function call = 222
```

Constant Reference Parameters

- Reference arguments inherently "dangerous"
  - Caller's data can be changed
  - Often this is desired, sometimes not
- To "protect" data, & still pass by reference:
  - Use const keyword
    - void sendConstRef(const int &par1,
      const int &par2);
    - Makes arguments "read-only" by function
    - No changes allowed inside function body

Mixed Parameter Lists

- Can combine passing mechanisms
- Parameter lists can include pass-by-value and pass-by-reference parameters
- Order of arguments in list is critical:
  void mixedCall(int& par1, int par2, double& par3);
  - Function call:
    mixedCall(arg1, arg2, arg3);
    - arg1 must be integer type, is passed by reference
    - arg2 must be integer type, is passed by value
    - arg3 must be double type, is passed by reference

Overloading

- Same function name
- Different parameter lists
- Two separate function definitions
- Function "signature"
  - Function name & parameter list
  - Must be "unique" for each function definition
- Allows same task performed on different data

Overloading Example: Average

- Function computes average of 2 numbers:

```
double average(double n1, double n2)
{
        return ((n1 + n2) / 2.0);
}
```

- Now compute average of 3 numbers:

```
double average(double n1, double n2, double n3)
{
        return ((n1 + n2 + n3) / 3.0);
}
```

- Same name, two functions

Overloaded Average() Cont'd

- Which function gets called?
- Depends on function call itself:
  - avg = average(5.2, 6.7);
    - Calls "two-parameter average()"
  - avg = average(6.5, 8.5, 4.2);
    - Calls "three-parameter average()"
- Compiler resolves invocation based on signature of function call
  - "Matches" call with appropriate function
  - Each considered separate function

Overloading Resolution

- 1st: Exact Match
  - Looks for exact signature
    - Where no argument conversion required
- 2nd: Compatible Match
  - Looks for "compatible" signature where automatic type conversion is possible:
    - 1st with promotion (e.g., int→double)
      - No loss of data
    - 2nd with demotion (e.g., double→int)
      - Possible loss of data

Overloading Resolution Example

- Given following functions:
  1. void f(int n);
  2. void f(long int n);
  3. void f(char *n);
  4. void f(double d, int n);

  Input: f(0);

Automatic Type Conversion and Overloading

- Numeric formal parameters typically made "double" type
- Allows for "any" numeric type
  - Any "subordinate" data automatically promoted
    - int → double
    - float → double
    - char → double

Automatic Type Conversion and Overloading Example

- double mpg(double miles, double gallons)
  {
  return (miles/gallons);
  }
- Example function calls:
  - mpgComputed = mpg(5, 20);
    - Converts 5 & 20 to doubles, then passes
  - mpgComputed = mpg(5.8, 20.2);
    - No conversion necessary
  - mpgComputed = mpg(5, 2.4);
    - Converts 5 to 5.0, then passes values to function

Default Arguments

- Allows omitting some arguments
- Specified in function declaration/prototype
  - void showVolume(int length,
    int width = 1,
    int height = 1);
    - Last 2 arguments are defaulted
  - Possible calls:
    - showVolume(2, 4, 6); //All arguments supplied
    - showVolume(3, 5); //height defaulted to 1
    - showVolume(7); //width & height defaulted to 1

Default Arguments Example:
Display 4.1 Default Arguments (1 of 2)

Display 4.8 Default Arguments

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

void showVolume(int length, int width = 1, int height = 1);
//Returns the volume of a box.
//If no height is given, the height is assumed to be 1.
//If neither height nor width is given, both are assumed to be 1.

int main( )
{
    showVolume(4, 6, 2);
    showVolume(4, 6);
    showVolume(4);

    return 0;
}

void showVolume(int length, int width, int height)
```

Default arguments

A default argument should not be given a second time.

Default Arguments Example:
Display 4.1 Default Arguments (2 of 2)

```
{
    cout << "Volume of a box with \n"
         << "Length = " << length << ", Width = " << width << endl
         << "and Height = " << height
         << " is " << length*width*height << endl;
}
```

SAMPLE DIALOGUE

```
Volume of a box with
Length = 4, Width = 6
and Height = 2 is 48
Volume of a box with
Length = 4, Width = 6
and Height = 1 is 24
Volume of a box with
Length = 4, Width = 1
and Height = 1 is 4
```

Testing and Debugging Functions

- Many methods:
  - Lots of cout statements
    - In calls and definitions
    - Used to "trace" execution
  - Compiler Debugger
    - Environment-dependent
  - assert Macro
    - Early termination as needed

The assert Macro

- Assertion: a true or false statement
- Used to document and check correctness
  - Preconditions & Postconditions
    - Typical assert use: confirm their validity
  - Syntax:
    assert(<assert_condition>);
    - No return value
    - Evaluates assert_condition
    - Terminates if false, continues if true
- Predefined in library <cassert>
  - Macros used similarly as functions

An assert Macro Example

- Given Function Declaration:

```
void computeCoin( int coinValue,
                  int& number,
                  int& amountLeft);
//Precondition: 0 < coinValue < 100
                0 <= amountLeft <100
//Postcondition: number set to max. number
                 of coins
```

- Check precondition:
  - assert ((0 < currentCoin) && (currentCoin < 100)
    && (0 <= currentAmountLeft) && (currentAmountLeft < 100));
  - If precondition not satisfied → condition is false → *program execution terminates*!

assert On/Off

- Preprocessor provides means
- #define NDEBUG
  #include <cassert>
- Add "#define" line before #include line
  - Turns OFF all assertions throughout program
- Remove "#define" line (or comment out)
  - Turns assertions back on