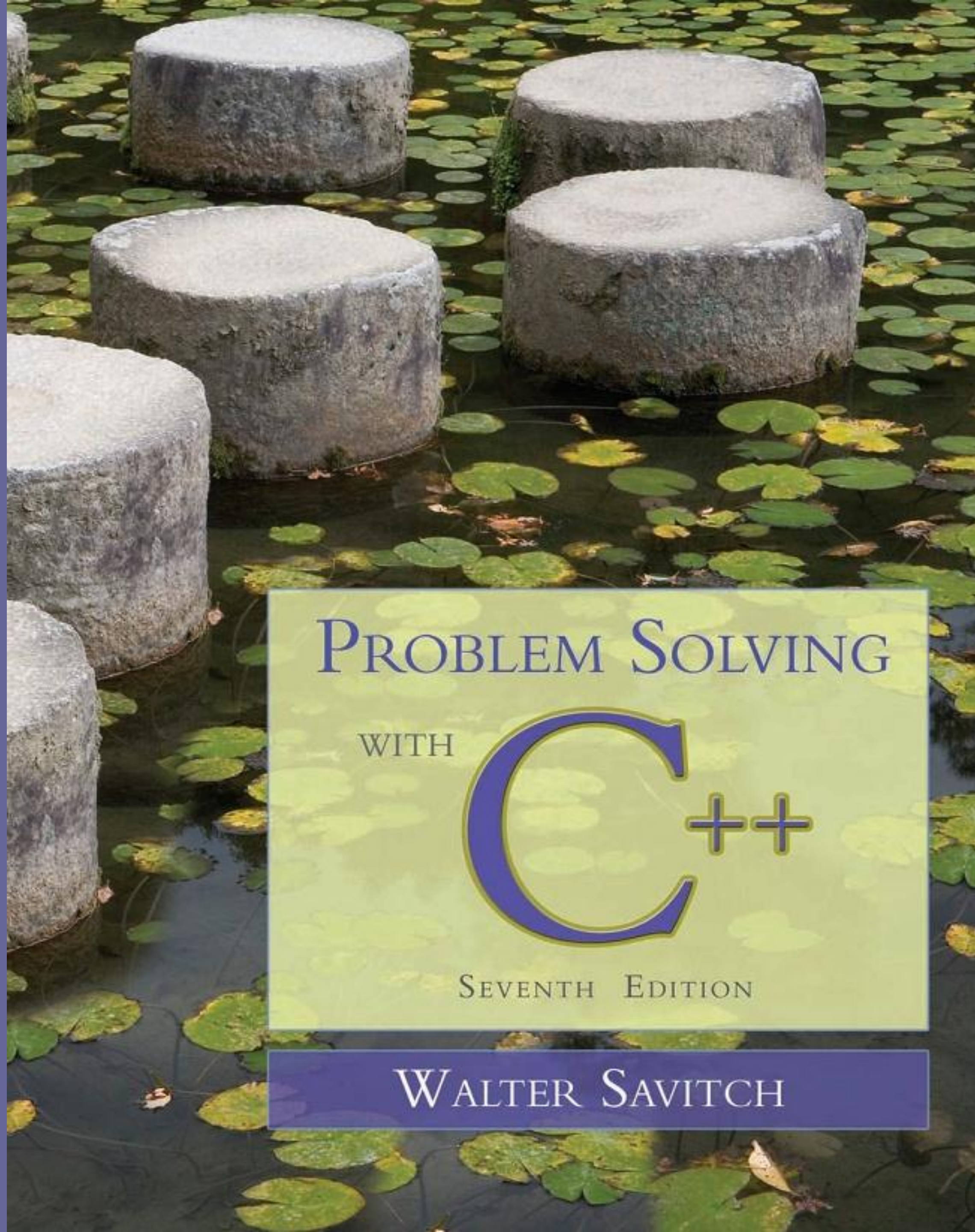


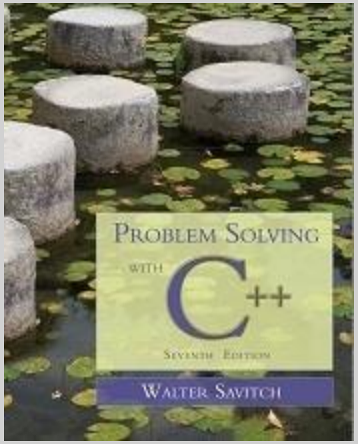
Problem Solving with C++

Seventh Edition

Walter Savitch

Chapter 4

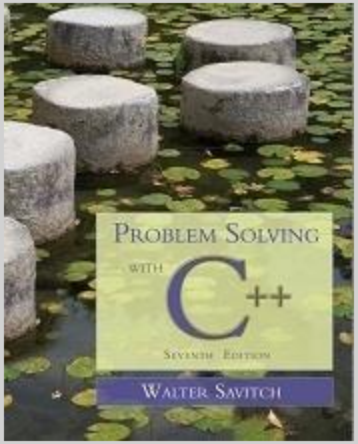
Procedural Abstraction and Functions That Return a Value



Overview

4.1

Top-Down Design



Top Down Design

- To write a program
  - Develop the algorithm that the program will use
  - Translate the algorithm into the programming language

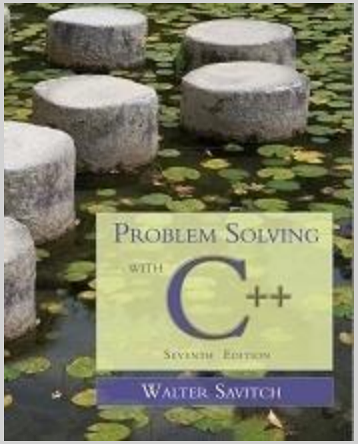
- Top Down Design
  (also called stepwise refinement)
  - Break the algorithm into subtasks
  - Break each subtask into smaller subtasks
  - Eventually the smaller subtasks are trivial to implement in the programming language

Benefits of Top Down Design

- Subtasks, or functions in C++, make programs
  - Easier to understand
  - Easier to change
  - Easier to write
  - Easier to test
  - Easier to debug
  - Easier for teams to develop

4.2

Predefined Functions



Predefined Functions

- C++ comes with libraries of predefined functions
- Example: sqrt function
  - the_root = sqrt(9.0);
  - returns, or computes, the square root of a number
  - The number, 9, is called the argument
  - the_root will contain 3.0

Function Calls

- sqrt(9.0) is a function call
  - It invokes, or sets in action, the sqrt function
  - The argument (9), can also be a variable or an expression
- A function call can be used like any expression
  - bonus = sqrt(sales) / 10;
  - Cout << “The side of a square with area “ << area
    << “ is “
    << sqrt(area);

Display 4.1

Function Call Syntax

- Function_name (Argument_List)
  - Argument_List is a comma separated list:

    (Argument_1, Argument_2, … , Argument_Last)
- Example:
  - side = sqrt(area);
  - cout << “2.5 to the power 3.0 is “
    << pow(2.5, 3.0);

Function Libraries

- Predefined functions are found in libraries
- The library must be “included” in a program to make the functions available
- An include directive tells the compiler which library header file to include.
- To include the math library containing sqrt():

```
#include <cmath>
```

- Newer standard libraries, such as cmath, also require the directive

```
using namespace std;
```

Other Predefined Functions

- abs(x) --- int value = abs(-8);
  - Returns absolute value of argument x
  - Return value is of type int
  - Argument is of type x
  - Found in the library cstdlib
- fabs(x) --- double value = fabs(-8.0);
  - Returns the absolute value of argument x
  - Return value is of type double
  - Argument is of type double
  - Found in the library cmath

Display 4.2

Type Casting

- Recall the problem with integer division:
  int total_candy = 9, number_of_people = 4;
  double candy_per_person;
  candy_per_person = total_candy / number_of_people;
  - candy_per_person = 2, not 2.25!
- A Type Cast produces a value of one type
  from another type
  - static_cast<double>(total_candy) produces a double representing the integer value of total_candy

Type Cast Example

- int total_candy = 9, number_of_people = 4;
  double candy_per_person;
  candy_per_person = static_cast<double>(total_candy)
  / number_of_people;
  - candy_per_person now is 2.25!
  - This would also work:
    candy_per_person = total_candy /
    static_cast<double>(number_of_people);
  - This would not!
    candy_per_person = static_cast<double>(total_candy /
    number_of_people);

Integer division occurs before type cast

Old Style Type Cast

- C++ is an evolving language
- This older method of type casting may be discontinued in future versions of C++

```
candy_per_person =
double(total_candy)/number_of_people;
```

Section 4.2 Conclusion

- Can you
  - Determine the value of d?

    double d = 11 / 2;

  - Determine the value of

pow(2,3)	fabs(-3.5)	sqrt(pow(3,2))
7 / abs(-2)	ceil(5.8)	floor(5.8)

  - Convert the following to C++

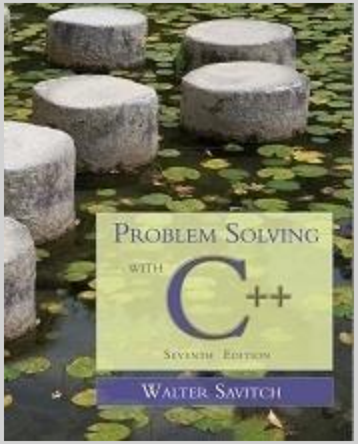
$$\sqrt{x+y}$$

$$\frac{-b+\sqrt{b^2-4ac}}{2a}$$

$$x^{y+7}$$

4.3

Programmer-Defined Functions



Programmer-Defined Functions

- Two components of a function definition
  - Function declaration (or function prototype)
    - Shows how the function is called
    - Must appear in the code before the function can be called
    - Syntax:
      Type_returned Function_Name(Parameter_List);
      //Comment describing what function does

      ;

  - Function definition
    - Describes how the function does its task
    - Can appear before or after the function is called
    - Syntax:

```
Type_returned  Function_Name(Parameter_List)
  {
       //code to make the function work
  }
```

Function Declaration

- Tells the return type
- Tells the name of the function
- Tells how many arguments are needed
- Tells the types of the arguments
- Tells the formal parameter names
  - Formal parameters are like placeholders for the actual arguments used when the function is called
  - Formal parameter names can be any valid identifier
- Example:
  double total_cost(int number_par, double price_par);
  // Compute total cost including 5% sales tax on
  // number_par items at cost of price_par each

Function Definition

- Provides the same information as the declaration
- Describes how the function does its task

function header

- Example:

```
double total_cost(int number_par, double price_par)
{
    const double TAX_RATE = 0.05; //5% tax
    double subtotal;
     subtotal = price_par * number_par;
    return (subtotal + subtotal * TAX_RATE);
}
```

function body

The Return Statement

- Ends the function call
- Returns the value calculated by the function
- Syntax:

  return expression;

  - expression performs the calculation
    or
  - expression is a variable containing the calculated value
- Example:

  return subtotal + subtotal * TAX_RATE;

The Function Call

- Tells the name of the function to use
- Lists the arguments
- Is used in a statement where the returned value makes sense
- Example:

```
double bill = total_cost(number, price);
```

Display 4.3

Function Call Details

- The values of the arguments are plugged into the formal parameters (Call-by-value mechanism with call-by-value parameters)
  - The first argument is used for the first formal parameter, the second argument for the second formal parameter, and so forth.
  - The value plugged into the formal parameter is used in all instances of the formal parameter in the function body

Display 4.4 (1)

Display 4.4 (2)

Alternate Declarations

- Two forms for function declarations
  - List formal parameter names
  - List types of formal parmeters, but not names
  - First aids description of the function in comments
- Examples:
  double total_cost(int number_par, double price_par);

  double total_cost(int, double);
- Function headers must always list formal parameter names!

Order of Arguments

- Compiler checks that the types of the arguments are correct and in the correct sequence.
- Compiler cannot check that arguments are in the correct logical order
- Example: Given the function declaration:
  char grade(int received_par, int min_score_par);

  int received = 95, min_score = 60;

  cout << grade(min_score, received);

  - Produces a faulty result because the arguments are not in the correct logical order. The compiler will not catch this!

Display 4.5 (1)

Display 4.5 (2)

Function Definition Syntax

- Within a function definition
  - Variables must be declared before they are used
  - Variables are typically declared before the executable statements begin
  - At least one return statement must end the function
    - Each branch of an if-else statement might have its own return statement

Display 4.6

Placing Definitions

- A function call must be preceded by either
  - The function's declaration

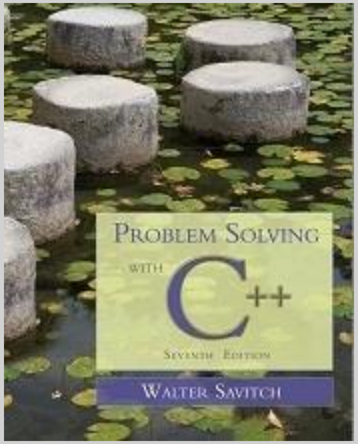
    or
  - The function's definition
    - If the function's definition precedes the call, a declaration is not needed
- Placing the function declaration prior to the main function and the function definition after the main function leads naturally to building your own libraries in the future.

Section 4.3 Conclusion

- Can you
  - Write a function declaration and a function definition for a function that takes three arguments, all of type int, and that returns the sum of its three arguments?
  - Describe the call-by-value parameter mechanism?
  - Write a function declaration and a function definition for a function that takes one argument of type int and one argument of type double, and that returns a value of type double that is the average of the two arguments?

4.4

Procedural Abstraction



Procedural Abstraction

- The Black Box Analogy
  - A black box refers to something that we know how to use, but the method of operation is unknown
  - A person using a program does not need to know how it is coded
  - A person using a program needs to know what the program does, not how it does it
- Functions and the Black Box Analogy
  - A programmer who uses a function needs to know what the function does, not how it does it

  - A programmer needs to know what will be produced if the proper arguments are put into the box

Information Hiding

- Designing functions as black boxes is an example of information hiding
  - The function can be used without knowing how it is coded
  - The function body can be "hidden from view"

Function Implementations and The Black Box

- Designing with the black box in mind allows us
  - To change or improve a function definition without forcing programmers using the function to change what they have done
  - To know how to use a function simply by reading the function declaration and its comment

Display 4.7

Procedural Abstraction and C++

- Procedural Abstraction is writing and using functions as if they were black boxes
  - Procedure is a general term meaning a “function like” set of instructions
  - Abstraction implies that when you use a function as a black box, you abstract away the details of the code in the function body

Procedural Abstraction and Functions

- Write functions so the declaration and comment is all a programmer needs to use the function
  - Function comment should tell all conditions required of arguments to the function
  - Function comment should describe the returned value
  - Variables used in the function, other than the formal parameters, should be declared in the function body

Formal Parameter Names

- Functions are designed as self-contained modules
- Different programmers may write each function
- Programmers choose meaningful names for formal parameters
  - Formal parameter names may or may not match variable names used in the main part of the program
  - It does not matter if formal parameter names match other variable names in the program
  - Remember that only the value of the argument is plugged into the formal parameter

Display 4.8

Case Study Buying Pizza

- What size pizza is the best buy?
  - Which size gives the lowest cost per square inch?
  - Pizza sizes given in diameter
  - Quantity of pizza is based on the area which is proportional to the square of the radius

Buying Pizza Problem Definition

- Input:
  - Diameter of two sizes of pizza
  - Cost of the same two sizes of pizza
- Output:
  - Cost per square inch for each size of pizza
  - Which size is the best buy
    - Based on lowest price per square inch
    - If cost per square inch is the same, the smaller size will be the better buy

Buying Pizza Problem Analysis

- Subtask 1
  - Get the input data for each size of pizza
- Subtask 2
  - Compute price per inch for smaller pizza
- Subtask 3
  - Compute price per inch for larger pizza
- Subtask 4
  - Determine which size is the better buy
- Subtask 5
  - Output the results

Buying Pizza Function Analysis

- Subtask 2 and subtask 3 should be implemented as a single function because
  - Subtask 2 and subtask 3 are identical tasks
    - The calculation for subtask 3 is the same as the calculation for subtask 2 with different arguments
  - Subtask 2 and subtask 3 each return a single value
- Choose an appropriate name for the function
  - We'll use unitprice

Buying Pizza unitprice Declaration

- double unitprice(int diameter, int double price);
  //Returns the price per square inch of a pizza
  //The formal parameter named diameter is the
  //diameter of the pizza in inches. The formal
  // parameter named price is the price of the
  // pizza.

Buying Pizza Algorithm Design

- Subtask 1
  - Ask for the input values and store them in variables
    - diameter_small diameter_large
      price_small price_large
- Subtask 4
  - Compare cost per square inch of the two pizzas using the less than operator
- Subtask 5
  - Standard output of the results

Buying Pizza unitprice Algorithm

- Subtasks 2 and 3 are implemented as calls to function unitprice
- unitprice algorithm
  - Compute the radius of the pizza πr^2
  - Computer the area of the pizza using
  - Return the value of (price / area)

Buying Pizza unitprice Pseudocode

- Pseudocode
  - Mixture of C++ and english
  - Allows us to make the algorithm more precise without worrying about the details of C++ syntax

- unitprice pseudocode
  - radius = one half of diameter;
    area = π * radius * radius
    return (price / area)

Buying Pizza The Calls of unitprice

- Main part of the program implements calls of unitprice as
  - ```
    double unit_price_small, unit_price_large;
    unit_price_small = unitprice(diameter_small,
    price_small);
    unit_price_large = unitprice(diameter_large,
    price_large);
    ```
    ```

Buying Pizza First try at unitprice

- double unitprice (int diameter, double price)
```
{
    const double PI = 3.14159;
    double radius, area;

    radius = diameter / 2;
    area = PI * radius * radius;
    return (price / area);
}
```
  - Oops! Radius should include the fractional part

Buying Pizza Second try at unitprice

- double unitprice (int diameter, double price)

```
{
    const double PI = 3.14159;
    double radius, area;

     radius = diameter / static_cast<double>(2) ;
     area = PI * radius * radius;
     return (price / area);
}
```

Display 4.10 (1)

Display 4.10 (2)

  - Now radius will include fractional parts
    - radius = diameter / 2.0 ;   // This would also work

Program Testing

- Programs that compile and run can still produce errors
- Testing increases confidence that the program works correctly
  - Run the program with data that has known output
    - You may have determined this output with pencil and paper or a calculator
  - Run the program on several different sets of data
    - Your first set of data may produce correct results in spite of a logical error in the code
      - Remember the integer division problem? If there is no fractional remainder, integer division will give apparently correct results

Use Pseudocode

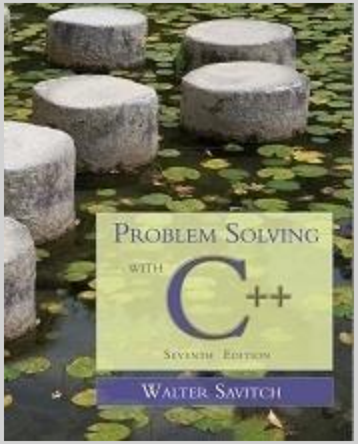
- Pseudocode is a mixture of English and the programming language in use
- Pseudocode simplifies algorithm design by allowing you to ignore the specific syntax of the programming language as you work out the details of the algorithm
  - If the step is obvious, use C++
  - If the step is difficult to express in C++, use English

Section 4.4 Conclusion

- Can you
  - Describe the purpose of the comment that accompanies a function declaration?
  - Describe what it means to say a programmer should be able to treat a function as a black box?
  - Describe what it means for two functions to be black box equivalent?

4.5

Local Variables



Local Variables

- Variables declared in a function:
  - Are local to that function, they cannot be used from outside the function
  - Have the function as their scope
- Variables declared in the main part of a program:
  - Are local to the main part of the program, they cannot be used from outside the main part
  - Have the main part as their scope

Display 4.11 (1)

Display 4.11 (2)

Global Constants

- Global Named Constant
  - Available to more than one function as well as the main part of the program
  - Declared outside any function body
  - Declared outside the main function body
  - Declared before any function that uses it
- Example:

```
const double PI = 3.14159;
double volume(double);
 int main()
{…}
```

  - PI is available to the main function and to function volume

Display 4.12 (1)

Display 4.12 (2)

Global Variables

- Global Variable -- rarely used when more than one function must use a common variable
  - Declared just like a global constant except const is not used
  - Generally make programs more difficult to understand and maintain

Formal Parameters are Local Variables

- Formal Parameters are actually variables that are local to the function definition
  - They are used just as if they were declared in the function body
  - Do NOT re-declare the formal parameters in the function body, they are declared in the function declaration
- The call-by-value mechanism
  - When a function is called the formal parameters are initialized to the values of the arguments in the function call

Display 4.13 (1)

Display 4.13 (2)

Namespaces Revisited

- The start of a file is not always the best place for

  using namespace std;

- Different functions may use different namespaces
  - Placing using namespace std; inside the starting brace of a function
    - Allows the use of different namespaces in different functions
    - Makes the "using" directive local to the function

Display 4.14 (1)

Display 4.14 (2)

Example: Factorial

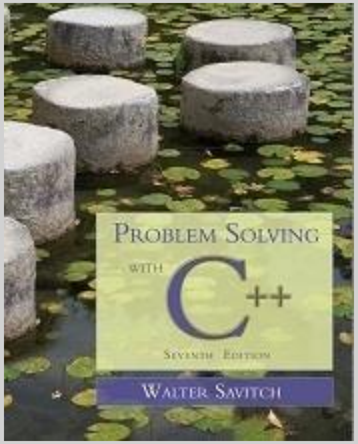
- n! Represents the factorial function
- n! = 1 x 2 x 3 x … x n
- The C++ version of the factorial function found in Display 3.14
  - Requires one argument of type int, n
  - Returns a value of type int
  - Uses a local variable to store the current product
  - Decrements n each time it does another multiplication

    n * n-1 * n-2 * … * 1

Display 4.15

4.6

Overloading Function Names



Overloading Function Names

- C++ allows more than one definition for the same function name
  - Very convenient for situations in which the “same” function is needed for different numbers or types of arguments
- Overloading a function name means providing more than one declaration and definition using the same function name

Overloading Examples

- double ave(double n1, double n2)
  {
          return ((n1 + n2) / 2);
  }
- double ave(double n1, double n2, double n3)
  {
      return ((n1 + n2 + n3) / 3);
  }
  - Compiler checks the number and types of arguments in the function call to decide which function to use

    cout << ave(10, 20, 30);

    uses the second definition

Overloading Details

- Overloaded functions
  - Must have different numbers of formal parameters
    AND / OR
  - Must have at least one different type of parameter

  - Must return a value of the same type

Display 4.16

Overloading Example

- Revising the Pizza Buying program
  - Rectangular pizzas are now offered!
  - Change the input and add a function to compute the unit price of a rectangular pizza
  - The new function could be named unitprice_rectangular
  - Or, the new function could be a new (overloaded) version of the unitprice function that is already used
    - Example:

```
double unitprice(int length, int width, double price)
  {
     double area = length * width;
      return (price / area);
   }
```

Display 4.17 (1 – 3)

Automatic Type Conversion

- Given the definition

```
double mpg(double miles, double gallons)
    {
        return (miles / gallons);
    }
```

what will happen if mpg is called in this way?

```
cout << mpg(45, 2) << " miles per gallon";
```

- The values of the arguments will automatically be converted to type double (45.0 and 2.0)

Type Conversion Problem

- Given the previous mpg definition and the following definition in the same program

```
int mpg(int goals, int misses)
 // returns the Measure of Perfect Goals
  {
       return (goals – misses);
  }
```

what happens if mpg is called this way now?

```
cout << mpg(45, 2) << " miles per gallon";
```

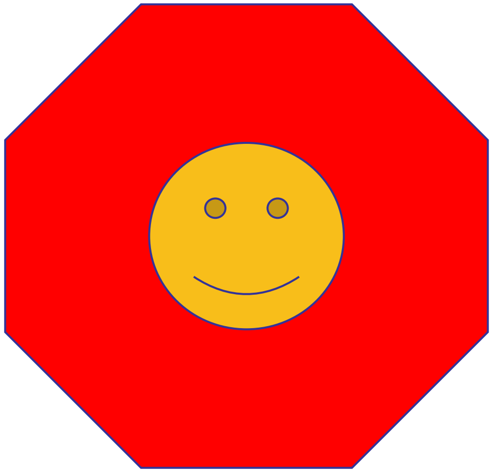
  - The compiler chooses the function that matches parameter types so the Measure of Perfect Goals will be calculated

Do not use the same function name for unrelated functions

Section 4.6 Conclusion

- Can you
  - Describe Top-Down Design?
  - Describe the types of tasks we have seen so far that could be implemented as C++ functions?
  - Describe the principles of
    - The black box
    - Procedural abstraction
    - Information hiding
  - Define "local variable"?
  - Overload a function name?

Chapter 4 -- End



A Function Call

```
//Computes the size of a dog house 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, length_side;

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

    area = budget/COST_PER_SQ_FT;
    length_side = sqrt(area);

    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 dog house\n"
         << "that is " << length_side
         << " feet on each side.\n";

    return 0;
}
```

Sample Dialogue

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

Display 4.1

Back Next

Display 4.2

Back Next

Some Predefined Functions

Name	Description	Type of Arguments	Type of Value Returned	Example	Value	Library Header
`sqrt`	square root	*double*	*double*	`sqrt(4.0)`	2.0	cmath
`pow`	powers	*double*	*double*	`pow(2.0,3.0)`	8.0	cmath
`abs`	absolute value for *int*	*int*	*int*	`abs(-7)` `abs(7)`	7 7	cstdlib
`labs`	absolute value for *long*	*long*	*long*	`labs(-70000)` `labs(70000)`	70000 70000	cstdlib
`fabs`	absolute value for *double*	*double*	*double*	`fabs(-7.5)` `fabs(7.5)`	7.5 7.5	cmath
`ceil`	ceiling (round up)	*double*	*double*	`ceil(3.2)` `ceil(3.9)`	4.0 4.0	cmath
`floor`	floor (round down)	*double*	*double*	`floor(3.2)` `floor(3.9)`	3.0 3.0	cmath

Display 4.3 (1/2)

A Function Definition (*part 1 of 2*)

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

double total_cost(int number_par, double price_par);
```

function declaration

```
//Computes the total cost, including 5% sales tax,
//on number_par items at a cost of price_par 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;
```

function call

```
    bill = total_cost(number, price);

    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;
}
```

function heading

```
double total_cost(int number_par, double price_par)
{
    const double TAX_RATE = 0.05; //5% sales tax
    double subtotal;

    subtotal = price_par * number_par;
    return (subtotal + subtotal*TAX_RATE);
}
```

function body

function definition

Display 4.3 (2/2)

Back Next

A Function Definition (*part 2 of 2*)

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
```

Display 4.4 (1/2)

Back Next

DISPLAY 4.4 Details of a Function Call *(part 1 of 2)*

Anatomy of the Function Call in Display 4.3

0 Before the function is called, the values of the variables `number` and `price` are set to `2` and `10.10`, by `cin` statements (as you can see in the Sample Dialogue in Display 4.3).

1 The following statement, which includes a function call, begins executing:

```
bill = total_cost(number, price);
```

2 The value of `number` (which is `2`) is plugged in for `number_par` and the value of `price` (which is `10.10`) is plugged in for `price_par`:

plug in value of `number`

```
double total_cost(int number_par, double price_par)
{
    const double TAX_RATE = 0.05; //5% sales tax
    double subtotal;

    subtotal = price_par * number_par;
    return (subtotal + subtotal*TAX_RATE);
}
```

plug in value of `price`

producing the following:

```
double total_cost(int 2, double 10.10)
{
    const double TAX_RATE = 0.05; //5% sales tax
    double subtotal;

    subtotal = 10.10 * 2;
    return (subtotal + subtotal*TAX_RATE);
}
```

Display 4.4 (2/2)

Back Next

DISPLAY 4.4 Details of a Function Call *(part 2 of 2)*

Anatomy of the Function Call in Display 4.3 (concluded)

3 The body of the function is executed, that is, the following is executed:

```
{
    const double TAX_RATE = 0.05; //5% sales tax
    double subtotal;

    subtotal = 10.10 * 2;
    return (subtotal + subtotal*TAX_RATE);
}
```

4 When the *return* statement is executed, the value of the expression after *return* is the value returned by the function. In this case, when

```
return (subtotal + subtotal*TAX_RATE);
```

is executed, the value of `(subtotal + subtotal*TAX_RATE)`, which is `21.21`, is returned by the function call

```
total_cost(number, price)
```

and so the value of `bill` (on the left-hand side of the equal sign) is set equal to `21.21` when the following statement finally ends:

```
bill = total_cost(number, price);
```

Incorrectly Ordered Arguments (*part 1 of 2*)

```
//Determines user's grade. Grades are Pass or Fail.
#include <iostream>
using namespace std;

char grade(int received_par, int min_score_par);
//Returns 'P' for passing, if received_par is
//min_score_par or higher. Otherwise returns 'F' for failing.

int main()
{
    int score, need_to_pass;
    char letter_grade;

    cout << "Enter your score"
         << " and the minimum needed to pass:\n";
    cin >> score >> need_to_pass;

    letter_grade = grade(need_to_pass, score);

    cout << "You received a score of " << score << endl
         << "Minimum to pass is " << need_to_pass << endl;

    if (letter_grade == 'P')
        cout << "You Passed. Congratulations!\n";
    else
        cout << "Sorry. You failed.\n";

    cout << letter_grade
         << " will be entered in your record.\n";

    return 0;
}

char grade(int received_par, int min_score_par)
{
    if (received_par >= min_score_par)
        return 'P';
    else
        return 'F';
}
```

Display 4.5 (1/2)

Back Next

Display 4.5 (2/2)

Back Next

Incorrectly Ordered Arguments (*part 2 of 2*)

Sample Dialogue

```
Enter your score and the minimum needed to pass:
98 60
You received a score of 98
Minimum to pass is 60
Sorry. You failed.
F will be entered in your record.
```

Display 4.6

Back Next

Syntax for a Function That Returns a Value

Function Declaration

Type_Returned Function_Name(Parameter_List);
Function_Declaration_Comment

Function Definition

```
Type_Returned Function_Name(Parameter_List)   <-- function header
{
    Declaration_1
    Declaration_2
        . . .
    Declaration_Last
    Executable_Statement_1
    Executable_Statement_2
        . . .
    Executable_Statement_Last
}
```

body

Must include one or more `return` statements.

Display 4.7

Back

Next

Definitions That Are Black-Box Equivalent

Function Declaration

```
double new_balance(double balance_par, double rate_par);
//Returns the balance in a bank account after
//posting simple interest. The formal parameter balance_par is
//the old balance. The formal parameter rate_par is the interest rate.
//For example, if rate_par is 5.0, then the interest rate is 5%
//and so new_balance(100, 5.0) returns 105.00.
```

Definition 1

```
double new_balance(double balance_par, double rate_par)
{
    double interest_fraction, interest;

    interest_fraction = rate_par/100;
    interest = interest_fraction*balance_par;
    return (balance_par + interest);
}
```

Definition 2

```
double new_balance(double balance_par, double rate_par)
{
    double interest_fraction, updated_balance;

    interest_fraction = rate_par/100;
    updated_balance = balance_par*(1 + interest_fraction);
    return updated_balance;
}
```

Display 4.8

Back Next

Simpler Formal Parameter Names

Function Declaration

```
double total_cost(int number, double price);
//Computes the total cost, including 5% sales tax, on
//number items at a cost of price each.
```

Function Definition

```
double total_cost(int number, double price)
{
    const double TAX_RATE = 0.05; //5% sales tax
    double subtotal;

    subtotal = price * number;
    return (subtotal + subtotal*TAX_RATE);
}
```

Display 4.9 (1/3)

Back Next

Nicely Nested Loops (*part 1 of 3*)

```
//Determines the total number of green-necked vulture eggs
//counted by all conservationists in the conservation district.
#include <iostream>
using namespace std;

void instructions( );

void get_one_total(int& total);
//Precondition: User will enter a list of egg counts
//followed by a negative number.
//Postcondition: total is equal to the sum of all the egg counts.

int main( )
{
    instructions( );

    int number_of_reports;
    cout << "How many conservationist reports are there? ";
    cin >> number_of_reports;

    int grand_total = 0, subtotal, count;
    for (count = 1; count <= number_of_reports; count++)
    {
        cout << endl << "Enter the report of "
             << "conservationist number " << count << endl;
        get_one_total(subtotal);
        cout << "Total egg count for conservationist "
             << " number " << count << " is "
             << subtotal << endl;
        grand_total = grand_total + subtotal;
    }

    cout << endl << "Total egg count for all reports = "
         << grand_total << endl;

    return 0;
}
```

Display 4.9 (2/3)

Back Next

Nicely Nested Loops ***(part 2 of 3)***

```
//Uses iostream:
void instructions( )
{
    cout << "This program tallies conservationist reports\n"
         << "on the green-necked vulture.\n"
         << "Each conservationist's report consists of\n"
         << "a list of numbers. Each number is the count of\n"
         << "the eggs observed in one"
         << " green-necked vulture nest.\n"
         << "This program then tallies"
         << " the total number of eggs.\n";
}

//Uses iostream:
void get_one_total(int& total)
{
    cout << "Enter the number of eggs in each nest.\n"
         << "Place a negative integer"
         << " at the end of your list.\n";

    total = 0;
    int next;
    cin >> next;
    while (next >= 0)
    {
        total = total + next;
        cin >> next;
    }
}
```

Display 4.9 (3/3)

Back Next

Nicely Nested Loops (*part 3 of 3*)

Sample Dialogue

```
This program tallies conservationist reports
on the green-necked vulture.
Each conservationist's report consists of
a list of numbers. Each number is the count of
the eggs observed in one green-necked vulture nest.
This program then tallies the total number of eggs.
How many conservationist reports are there? 3

Enter the report of conservationist number 1
Enter the number of eggs in each nest.
Place a negative integer at the end of your list.
1 0 0 2 -1
Total egg count for conservationist number 1 is 3

Enter the report of conservationist number 2
Enter the number of eggs in each nest.
Place a negative integer at the end of your list.
0 3 1 -1
Total egg count for conservationist number 2 is 4

Enter the report of conservationist number 3
Enter the number of eggs in each nest.
Place a negative integer at the end of your list.
-1
Total egg count for conservationist number 3 is 0

Total egg count for all reports = 7
```

Display 4.10 (1/2)

Back

Next

Buying Pizza (*part 1 of 2*)

```
//Determines which of two pizza sizes is the best buy.
#include <iostream>
using namespace std;

double unitprice(int diameter, double price);
//Returns the price per square inch of a pizza. The formal
//parameter named diameter is the diameter of the pizza in inches.
//The formal parameter named price is the price of the pizza.

int main()
{
    int diameter_small, diameter_large;
    double price_small, unitprice_small,
           price_large, unitprice_large;

    cout << "Welcome to the Pizza Consumers Union.\n";
    cout << "Enter diameter of a small pizza (in inches): ";
    cin >> diameter_small;
    cout << "Enter the price of a small pizza: $";
    cin >> price_small;
    cout << "Enter diameter of a large pizza (in inches): ";
    cin >> diameter_large;
    cout << "Enter the price of a large pizza: $";
    cin >> price_large;

    unitprice_small = unitprice(diameter_small, price_small);
    unitprice_large = unitprice(diameter_large, price_large);

    cout.setf(ios::fixed);
    cout.setf(ios::showpoint);
    cout.precision(2);
    cout << "Small pizza:\n"
         << "Diameter = " << diameter_small << " inches\n"
         << "Price = $" << price_small
         << " Per square inch = $" << unitprice_small << endl
         << "Large pizza:\n"
         << "Diameter = " << diameter_large << " inches\n"
         << "Price = $" << price_large
         << " Per square inch = $" << unitprice_large << endl;
```

Display 4.10 (2/2)

Back

Next

Buying Pizza ***(part 2 of 2)***

```
    if (unitprice_large < unitprice_small)
        cout << "The large one is the better buy.\n";
    else
        cout << "The small one is the better buy.\n";
    cout << "Buon Appetito!\n";

    return 0;
}

double unitprice(int diameter, double price)
{
    const double PI = 3.14159;
    double radius, area;

    radius = diameter/static_cast<double>(2);
    area = PI * radius * radius;
    return (price/area);
}
```

Sample Dialogue

```
Welcome to the Pizza Consumers Union.
Enter diameter of a small pizza (in inches): 10
Enter the price of a small pizza: $7.50
Enter diameter of a large pizza (in inches): 13
Enter the price of a large pizza: $14.75
Small pizza:
Diameter = 10 inches
Price = $7.50 Per square inch = $0.10
Large pizza:
Diameter = 13 inches
Price = $14.75 Per square inch = $0.11
The small one is the better buy.
Buon Appetito!
```

Display 4.11 (1/2)

Back Next

Local Variables (*part 1 of 2*)

```
//Computes the average yield on an experimental pea growing patch.
#include <iostream>
using namespace std;

double est_total(int min_peas, int max_peas, int pod_count);
//Returns an estimate of the total number of peas harvested.
//The formal parameter pod_count is the number of pods.
//The formal parameters min_peas and max_peas are the minimum
//and maximum number of peas in a pod.

int main()
{
    int max_count, min_count, pod_count;
    double average_pea, yield;

    cout << "Enter minimum and maximum number of peas in a pod: ";
    cin >> min_count >> max_count;
    cout << "Enter the number of pods: ";
    cin >> pod_count;
    cout << "Enter the weight of an average pea (in ounces): ";
    cin >> average_pea;

    yield =
          est_total(min_count, max_count, pod_count) * average_pea;

    cout.setf(ios::fixed);
    cout.setf(ios::showpoint);
    cout.precision(3);
    cout << "Min number of peas per pod = " << min_count << endl
         << "Max number of peas per pod = " << max_count << endl
         << "Pod count = " << pod_count << endl
         << "Average pea weight = "
         << average_pea << " ounces" << endl
         << "Estimated average yield = " << yield << " ounces"
         << endl;

    return 0;
}
```

This variable named `average_pea` *is local to the* `main` *part of the program.*

Display 4.11 (2/2)

Back Next

Local Variables (*part 2 of 2*)

```
 double est_total(int min_peas, int max_peas, int pod_count)
{
     double average_pea;

     average_pea = (max_peas + min_peas)/2.0;
     return (pod_count * average_pea);
}
```

This variable named average_pea is local to the function est_total.

Sample Dialogue

```
Enter minimum and maximum number of peas in a pod: 4 6
Enter the number of pods: 10
Enter the weight of an average pea (in ounces): 0.5
Min number of peas per pod = 4
Max number of peas per pod = 6
Pod count = 10
Average pea weight = 0.500 ounces
Estimated average yield = 25.000 ounces
```

Display 4.12 (1/2)

Back Next

A Global Named Constant (*part 1 of 2*)

```
//Computes the area of a circle and the volume of a sphere.
//Uses the same radius for both calculations.
#include <iostream>
#include <cmath>
using namespace std;

const double PI = 3.14159;

double area(double radius);
//Returns the area of a circle with the specified radius.

double volume(double radius);
//Returns the volume of a sphere with the specified radius.

int main()
{
    double radius_of_both, area_of_circle, volume_of_sphere;

    cout << "Enter a radius to use for both a circle\n"
         << "and a sphere (in inches): ";
    cin >> radius_of_both;

    area_of_circle = area(radius_of_both);
    volume_of_sphere = volume(radius_of_both);

    cout << "Radius = " << radius_of_both << " inches\n"
         << "Area of circle = " << area_of_circle
         << " square inches\n"
         << "Volume of sphere = " << volume_of_sphere
         << " cubic inches\n";

    return 0;
}
```

Display 4.12 (2/2)

Back Next

A Global Named Constant (***part 2 of 2***)

```
double area(double radius)
{
    return (PI * pow(radius, 2));
}

double volume(double radius)
{
    return ((4.0/3.0) * PI * pow(radius, 3));
}
```

Sample Dialogue

```
Enter a radius to use for both a circle
and a sphere (in inches): 2
Radius = 2 inches
Area of circle = 12.5664 square inches
Volume of sphere = 33.5103 cubic inches
```

Formal Parameter Used as a Local Variable (*part 1 of 2*)

Display 4.13 (1/2)

Back Next

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

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

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

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

    cout << "Welcome to the offices 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;
}

double fee(int hours_worked, int minutes_worked)
{
    int quarter_hours;

    minutes_worked = hours_worked*60 + minutes_worked;
    quarter_hours = minutes_worked/15;
    return (quarter_hours*RATE);
}
```

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

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

Display 4.13 (2/2)

Back Next

Formal Parameter Used as a Local Variable (*part 2 of 2*)

Sample Dialogue

```
Welcome to the offices of
Dewey, Cheatham, and Howe.
The law office with a heart.
Enter the hours and minutes of your consultation:
2 45
For 2 hours and 45 minutes, your bill is $1650.00
```

Display 4.14 (1/2)

Back Next

Using Namespaces ***(part 1 of 2)***

```
//Computes the area of a circle and the volume of a sphere.
//Uses the same radius for both calculations.
#include <iostream>
#include <cmath>

const double PI = 3.14159;

double area(double radius);
//Returns the area of a circle with the specified radius.

double volume(double radius);
//Returns the volume of a sphere with the specified radius.

int main( )
{
    using namespace std;

    double radius_of_both, area_of_circle, volume_of_sphere;

    cout << "Enter a radius to use for both a circle\n"
         << "and a sphere (in inches): ";
    cin >> radius_of_both;

    area_of_circle = area(radius_of_both);
    volume_of_sphere = volume(radius_of_both);

    cout << "Radius = " << radius_of_both << " inches\n"
         << "Area of circle = " << area_of_circle
         << " square inches\n"
         << "Volume of sphere = " << volume_of_sphere
         << " cubic inches\n";

    return 0;
}
```

Display 4.14 (2/2)

Back Next

Using Namespaces ***(part 2 of 2)***

```
double area(double radius)
{
    using namespace std;

    return (PI * pow(radius, 2));
}

double volume(double radius)
{
    using namespace std;

    return ((4.0/3.0) * PI * pow(radius, 3));
}
```

The sample dialogue for this program would be the same as the one for the program in Display 3.11.

Display 4.15

Back Next

Factorial Function

Function Declaration

```
int factorial(int n);
//Returns factorial of n.
//The argument n should be nonnegative.
```

Function Definition

```
int factorial(int n)
{
    int product = 1;
    while (n > 0)
    {
        product = n * product;
        n--;     formal parameter n
    }

    return product;
}
```

Overloading a Function Name

```
//Illustrates overloading the function name ave.
#include <iostream>

double ave(double n1, double n2);
//Returns the average of the two numbers n1 and n2.

double ave(double n1, double n2, double n3);
//Returns the average of the three numbers n1, n2, and n3.

int main()
{
    using namespace std;
    cout << "The average of 2.0, 2.5, and 3.0 is "
         << ave(2.0, 2.5, 3.0) << endl;

    cout << "The average of 4.5 and 5.5 is "
         << ave(4.5, 5.5) << endl;

    return 0;
}
                                        two arguments
double ave(double n1, double n2)
{
    return ((n1 + n2)/2.0);
}
                                                  three arguments
double ave(double n1, double n2, double n3)
{
    return ((n1 + n2 + n3)/3.0);
}
```

Output

```
The average of 2.0, 2.5, and 3.0 is 2.50000
The average of 4.5 and 5.5 is 5.00000
```

Display 4.16

Back

Next

Display 4.17 (1/3)

Back Next

Overloading a Function Name (*part 1 of 3*)

```
//Determines whether a round pizza or a rectangular pizza is the best buy.
#include <iostream>

double unitprice(int diameter, double price);
//Returns the price per square inch of a round pizza.
//The formal parameter named diameter is the diameter of the pizza
//in inches. The formal parameter named price is the price of the pizza.

double unitprice(int length, int width, double price);
//Returns the price per square inch of a rectangular pizza
//with dimensions length by width inches.
//The formal parameter price is the price of the pizza.

int main()
{
    using namespace std;
    int diameter, length, width;
    double price_round, unit_price_round,
           price_rectangular, unitprice_rectangular;

    cout << "Welcome to the Pizza Consumers Union.\n";
    cout << "Enter the diameter in inches"
         << " of a round pizza: ";
    cin >> diameter;
    cout << "Enter the price of a round pizza: $";
    cin >> price_round;
    cout << "Enter length and width in inches\n"
         << "of a rectangular pizza: ";
    cin >> length >> width;
    cout << "Enter the price of a rectangular pizza: $";
    cin >> price_rectangular;

    unitprice_rectangular =
                unitprice(length, width, price_rectangular);
    unit_price_round = unitprice(diameter, price_round);

    cout.setf(ios::fixed);
    cout.setf(ios::showpoint);
    cout.precision(2);
```

Overloading a Function Name *(part 2 of 3)*

```
    cout << endl
         << "Round pizza: Diameter = "
         << diameter << " inches\n"
         << "Price = $" << price_round
         << " Per square inch = $" << unit_price_round
         << endl
         << "Rectangular pizza: Length = "
         << length << " inches\n"
         << "Rectangular pizza: Width = "
         << width << " inches\n"
         << "Price = $" << price_rectangular
         << " Per square inch = $" << unitprice_rectangular
         << endl;

    if (unit_price_round < unitprice_rectangular)
        cout << "The round one is the better buy.\n";
    else
        cout << "The rectangular one is the better buy.\n";
    cout << "Buon Appetito!\n";

    return 0;
}

double unitprice(int diameter, double price)
{
    const double PI = 3.14159;
    double radius, area;

    radius = diameter/static_cast<double>(2);
    area = PI * radius * radius;
    return (price/area);
}

double unitprice(int length, int width, double price)
{
    double area = length * width;
    return (price/area);
}
```

Display 4.17 (2/3)

Back Next

Display 4.17 (3/3)

Back Next

Overloading a Function Name (*part 3 of 3*)

Sample Dialogue

```
Welcome to the Pizza Consumers Union.
Enter the diameter in inches of a round pizza: 10
Enter the price of a round pizza: $8.50
Enter length and width in inches
of a rectangular pizza: 6 4
Enter the price of a rectangular pizza: $7.55

Round pizza: Diameter = 10 inches
Price = $8.50 Per square inch = $0.11
Rectangular pizza: Length = 6 inches
Rectangular pizza: Width = 4 inches
Price = $7.55 Per square inch = $0.31
The round one is the better buy.
Buon Appetito!
```