

C++ Basics - 3

Rahul Deodhar

@rahuldeodhar

www.rahuldeodhar.com

rahuldeodhar@gmail.com

Topics for today

- Functions
- Classwork

Topics for today

- Homework
 - Program
 - Others

Procedural Abstraction & Functions

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

Pre-defined 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);`

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.
```

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`

Some Predefined Functions

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

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`

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;
```

Class Work

- 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++

$$\sqrt{(x + y)}$$

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

$$x^{(y+7)}$$

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) No ;
{
 //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);
```

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;
    bill = total_cost(number, price); ← function call

    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 total_cost(int number_par, double price_par) ← function heading
{
    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

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
```

Function Call Details

- The values of the arguments are plugged into the formal parameters (Call-by-value)
- 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

Details of a Function Call (part 1 of 2)

Anatomy of the Function Call in Display 3.3

- 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 3.3).
- 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`:

```
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 number

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);
}
```

Details of a Function Call (part 2 of 2)

Anatomy of the Function Call in Display 3.3 (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 $(\text{subtotal} + \text{subtotal} * \text{TAX_RATE})$, 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);
```

Alternate Declarations

- Two forms for function declarations
 1. List formal parameter names
 2. List types of formal parameters, 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!**

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';
}
```

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.

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

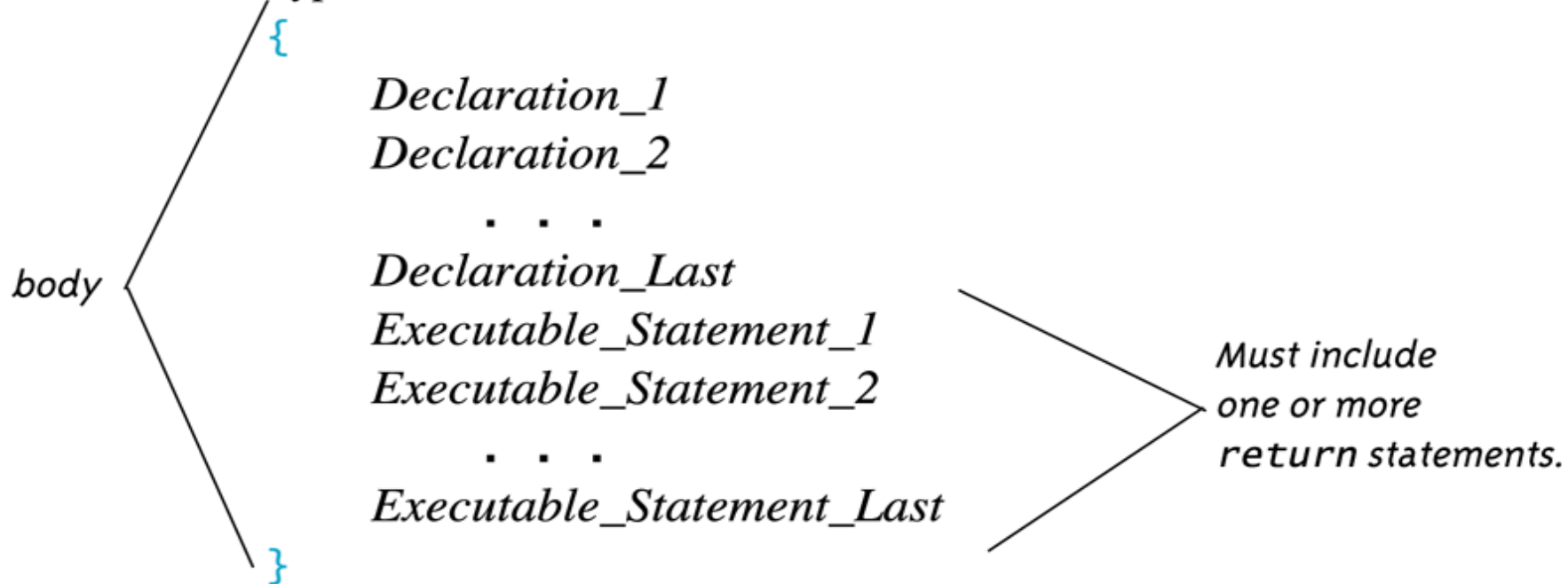
Syntax for a Function That Returns a Value

Function Declaration

Type_Returned *Function_Name* (*Parameter_List*);
Function_Declaration_Comment

Function Definition

Type_Returned *Function_Name* (*Parameter_List*) ← function header



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.

Class Work

- 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?

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”

Using 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

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;  
}
```

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 & 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

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);  
}
```

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
 - Computer the area of the pizza using $\pi * r^2$
 - 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 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);
}
- Now radius will include fractional parts
 - radius = diameter / **2.0** ; // This would also work

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;
```

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!
```

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

Class Work

- 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?

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

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.*

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
```

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

## 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;
}
```

---

## 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
```

---

# 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 : 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



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

---

```
//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.

## 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
```

---

# 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

## 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;
}
```

---

## Using Namespaces (part 2 of 2)

---

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

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

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

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

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

---

# Example: Factorial

- $n!$  Represents the factorial function
  - $n! = 1 \times 2 \times 3 \times \dots \times n$
  - 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
- $$n * n-1 * n-2 * \dots * 1$$

## 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



# 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

## 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;
}
```

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

two arguments

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

three arguments

## 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
```

---

# 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);
}
```

## 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);
}
```

---

## 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!
```

---



# 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**

# Class Work

- 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?

# Void Functions

# void-Functions

- In top-down design, a subtask might produce
  - No value (just input or output for example)
  - One value
  - More than one value
- We have seen how to implement functions that return one value
- A void-function implements a subtask that returns no value or more than one value

# void-Function Definition

- Two main differences between void-function definitions and the definitions of functions that return one value
  - Keyword **void** replaces the type of the value returned
    - **void means that no value is returned by the function**
  - The return statement does not include an expression

- Example:

```
void show_results(double f_degrees, double c_degrees)
{
 using namespace std;
 cout << f_degrees
 << " degrees Fahrenheit is equivalent to " << endl
 << c_degrees << " degrees Celsius." << endl;
 return;
}
```

## Syntax for a *void* Function Definition

---

### *void* Function Declaration

*void* *Function\_Name*(*Parameter\_List*);

*Function\_Declaration\_Comment*

### *void* Function Definition

*void* *Function\_Name*(*Parameter\_List*) ← *function header*

{

*Declaration\_1*

*Declaration\_2* ← *You may intermix the declarations with the executable statements.*

. . .

*Declaration\_Last*

*Executable\_Statement\_1*

*Executable\_Statement\_2*

. . .

*Executable\_Statement\_Last*

}

*body*

May (or may not) include one or more return statements.

# Using a void-Function

- void-function calls are executable statements
  - They do not need to be part of another statement
  - They end with a semi-colon
- Example:

```
show_results(32.5, 0.3);
```

**NOT:** `cout << show_results(32.5, 0.3);`



# void-Function Calls

- Mechanism is nearly the same as the function calls we have seen
  - Argument values are substituted for the formal parameters
    - It is fairly common to have no parameters in void-functions
      - In this case there will be no arguments in the function call
  - Statements in function body are executed
  - Optional return statement ends the function
    - Return statement does not include a value to return
    - Return statement is implicit if it is not included

# Example: Converting Temperatures

- The functions just developed can be used in a program to convert Fahrenheit temperatures to Celsius using the formula

$$C = (5/9) (F - 32)$$

- Do you see the integer division problem?

## void Functions (part 1 of 2)

---

```
//Program to convert a Fahrenheit temperature to a Celsius temperature.
#include <iostream>

void initialize_screen();
//Separates current output from
//the output of the previously run program.

double celsius(double fahrenheit);
//Converts a Fahrenheit temperature
//to a Celsius temperature.

void show_results(double f_degrees, double c_degrees);
//Displays output. Assumes that c_degrees
//Celsius is equivalent to f_degrees Fahrenheit.

int main()
{
 using namespace std;
 double f_temperature, c_temperature;

 initialize_screen();
 cout << "I will convert a Fahrenheit temperature"
 << " to Celsius.\n"
 << "Enter a temperature in Fahrenheit: ";
 cin >> f_temperature;

 c_temperature = celsius(f_temperature);

 show_results(f_temperature, c_temperature);
 return 0;
}

//Definition uses iostream:
void initialize_screen()
{
 using namespace std;
 cout << endl;
 return; ← This return is optional.
}

```

---

## **void Functions (part 2 of 2)**

---

```
double celsius(double fahrenheit)
{
 return ((5.0/9.0)*(fahrenheit - 32));
}

//Definition uses iostream:
void show_results(double f_degrees, double c_degrees)
{
 using namespace std;
 cout.setf(ios::fixed);
 cout.setf(ios::showpoint);
 cout.precision(1);
 cout << f_degrees
 << " degrees Fahrenheit is equivalent to\n"
 << c_degrees << " degrees Celsius.\n";
 return; ← This return is optional.
}
```

### **Sample Dialogue**

```
I will convert a Fahrenheit temperature to Celsius.
Enter a temperature in Fahrenheit: 32.5
32.5 degrees Fahrenheit is equivalent to
0.3 degrees Celsius.
```

---

# void-Functions Why Use a Return?

- Is a return-statement ever needed in a void-function since no value is returned?
  - Yes!
    - What if a branch of an if-else statement requires that the function ends to avoid producing more output, or creating a mathematical error?
    - void-function in Display 4.3, avoids division by zero with a return statement

## Use of *return* in a *void* Function


---

### Function Declaration

```
void ice_cream_division(int number, double total_weight);
//Outputs instructions for dividing total_weight ounces of
//ice cream among number customers.
//If number is 0, nothing is done.
```

### Function Definition

```
//Definition uses iostream:
void ice_cream_division(int number, double total_weight)
{
 using namespace std;
 double portion;

 if (number == 0)  return;
 portion = total_weight/number;
 cout.setf(ios::fixed);
 cout.setf(ios::showpoint);
 cout.precision(2);
 cout << "Each one receives "
 << portion << " ounces of ice cream." << endl;
}
```

*If number is 0, then the  
function execution ends here.*

---

# The Main Function

- The main function in a program is used like a void function...do you have to end the program with a return-statement?
  - Because the main function is defined to return a value of type int, the return is needed
  - C++ standard says the return 0 can be omitted, but many compilers still require it

# Class Work

- Can you
  - Describe the differences between void-functions and functions that return one value?
  - Tell what happens if you forget the return-statement in a void-function?
  - Distinguish between functions that are used as expressions and those used as statements?



# Call by Reference

# Call-by-Reference Parameters

- Call-by-value is not adequate when we need a sub-task to obtain input values
  - Call-by-value means that the formal parameters receive the values of the arguments
  - To obtain input values, we need to change the variables that are arguments to the function
    - Recall that we have changed the values of formal parameters in a function body, but we have not changed the arguments found in the function call
- Call-by-reference parameters allow us to change the variable used in the function call
  - Arguments for call-by-reference parameters must be variables, not numbers

# Call-by-Reference Example

- ```
void get_input(double& f_variable)
{
    using namespace std;
    cout << " Convert a Fahrenheit temperature"
         << " to Celsius.\n"
         << " Enter a temperature in Fahrenheit: ";
    cin >> f_variable;
}
```
- ‘&’ symbol (ampersand) identifies f_variable as a call-by-reference parameter
 - Used in both declaration and definition!

Call-by-Reference Parameters (part 1 of 2)

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

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

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

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

int main()
{
    int first_num, second_num;

    get_numbers(first_num, second_num);
    swap_values(first_num, second_num);
    show_results(first_num, second_num);
    return 0;
}

//Uses iostream:
void get_numbers(int& input1, int& input2)
{
    using namespace std;
    cout << "Enter two integers: ";
    cin >> input1
        >> input2;
}

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

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

Call-by-Reference Parameters (part 2 of 2)

```
//Uses iostream:
void show_results(int output1, int output2)
{
    using namespace std;
    cout << "In reverse order the numbers are: "
         << output1 << " " << output2 << endl;
}
```

Sample Dialogue

Enter two integers: 5 10

In reverse order the numbers are: 10 5

Call-By-Reference Details

- Call-by-reference works almost as if the argument variable is substituted for the formal parameter, not the argument's value
- In reality, the memory location of the argument variable is given to the formal parameter
 - Whatever is done to a formal parameter in the function body, is actually done to the value at the memory location of the argument variable

Behavior of Call-by-Reference Arguments (part 1 of 2)

Anatomy of a Function Call from Display 4.4 Using Call-by-Reference Arguments

- 0 Assume the variables `first_num` and `second_num` have been assigned the following memory address by the compiler:

```
first_num  —————> 1010
second_num —————> 1012
```

(We do not know what addresses are assigned and the results will not depend on the actual addresses, but this will make the process very concrete and thus perhaps easier to follow.)

- 1 In the program in Display 4.4, the following function call begins executing:

```
get_numbers(first_num, second_num);
```

- 2 The function is told to use the memory location of the variable `first_num` in place of the formal parameter `input1` and the memory location of the `second_num` in place of the formal parameter `input2`. The effect is the same as if the function definition were rewritten to the following (which is not legal C++ code, but does have a clear meaning to us):

```
void get_numbers(int& <the variable at memory location 1010>,
                 int& <the variable at memory location 1012>)
{
    using namespace std;
    cout << "Enter two integers: ";
    cin >> <the variable at memory location 1010>
        >> <the variable at memory location 1012>;
}
```

Since the variables in locations 1010 and 1012 are `first_num` and `second_num`, the effect is thus the same as if the function definition were rewritten to the following:

```
void get_numbers(int& first_num, int& second_num)
{
    using namespace std;
    cout << "Enter two integers: ";
    cin >> first_num
        >> second_num;
}
```

Behavior of Call-by-Reference Arguments (*part 2 of 2*)

Anatomy of the Function Call in Display 4.4 (*concluded*)

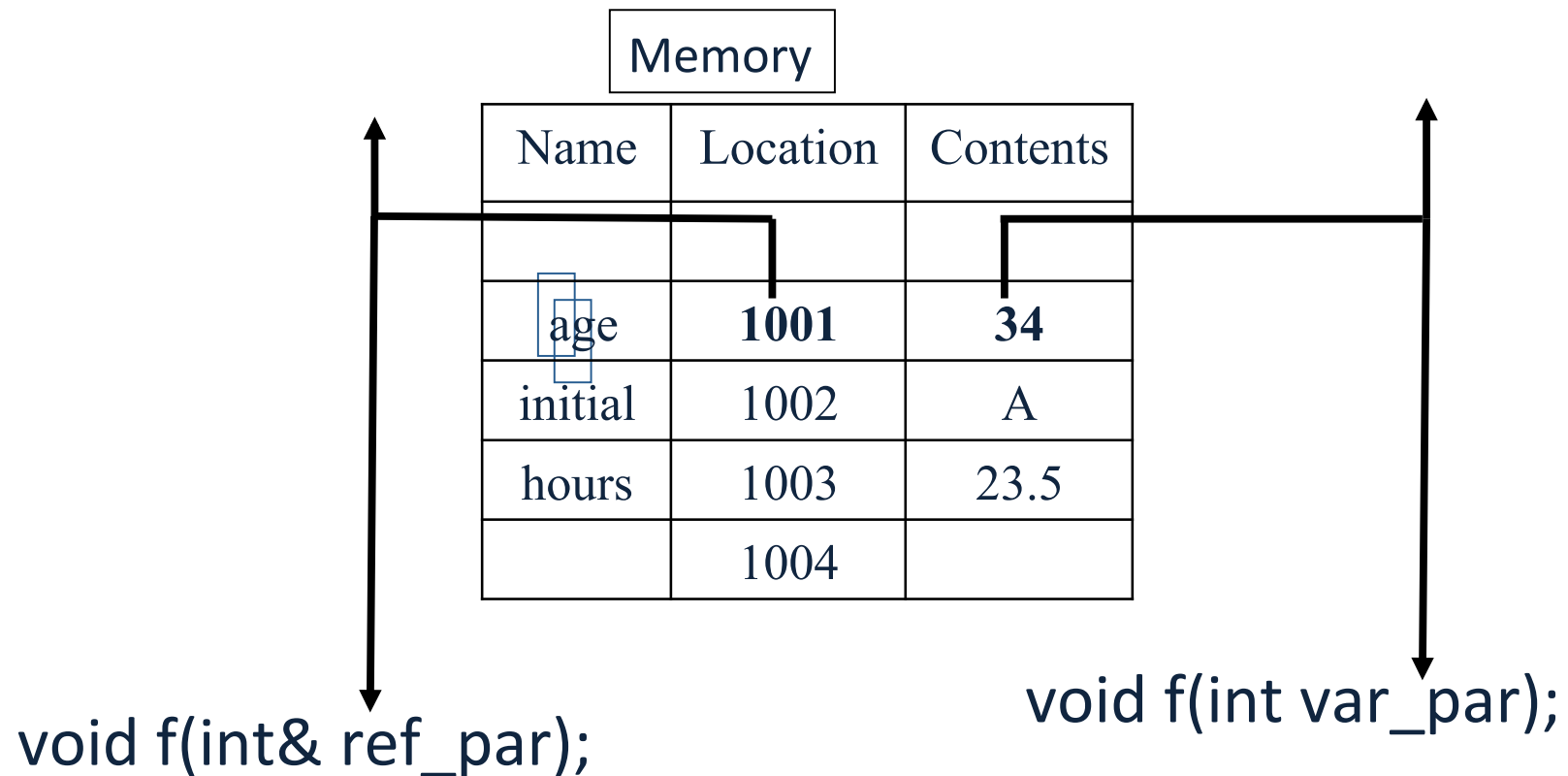
- 3** The body of the function is executed. The effect is the same as if the following were executed:

```
{  
    using namespace std;  
    cout << "Enter two integers: ";  
    cin >> first_num  
        >> second_num;  
}
```

- 4** When the `cin` statement is executed, the values of the variables `first_num` and `second_num` are set to the values typed in at the keyboard. (If the dialogue is as shown in Display 4.4, then the value of `first_num` is set to 5 and the value of `second_num` is set to 10.)
- 5** When the function call ends, the variables `first_num` and `second_num` retain the values that they were given by the `cin` statement in the function body. (If the dialogue is as shown in Display 4.4, then the value of `first_num` is 5 and the value of `second_num` is 10 at the end of the function call.)
-

Call By Reference vs Call by Value

- Call-by-reference
 - The function call:
f(age);
- Call-by-value
 - The function call:
f(age);



Example: swap_values

- ```
void swap(int& variable1, int& variable2)
{
 int temp = variable1;
 variable1 = variable2;
 variable2 = temp;
}
```
- If called with `swap(first_num, second_num);`
  - `first_num` is substituted for `variable1` in the parameter list
  - `second_num` is substituted for `variable2` in the parameter list
  - `temp` is assigned the value of `variable1` (`first_num`) since the next line will lose the value in `first_num`
  - `variable1` (`first_num`) is assigned the value in `variable2` (`second_num`)
  - `variable2` (`second_num`) is assigned the original value of `variable1` (`first_num`) which was stored in `temp`

# Mixed Parameter Lists

- Call-by-value and call-by-reference parameters can be mixed in the same function

- Example:

```
void good_stuff(int& par1, int par2, double& par3);
```

- par1 and par3 are call-by-reference formal parameters

- Changes in par1 and par3 change the argument variable

- par2 is a call-by-value formal parameter

- Changes in par2 do not change the argument variable

# Choosing Parameter Types

- How do you decide whether a call-by-reference or call-by-value formal parameter is needed?
  - Does the function need to change the value of the variable used as an argument?
  - Yes? Use a call-by-reference formal parameter
  - No? Use a call-by-value formal parameter

## Comparing Argument Mechanisms

---

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

void do_stuff(int par1_value, int& par2_ref);
//par1_value is a call-by-value formal parameter and
//par2_ref is a call-by-reference formal parameter.

int main()
{
 using namespace std;
 int n1, n2;

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

void do_stuff(int par1_value, int& par2_ref)
{
 using namespace std;
 par1_value = 111;
 cout << "par1_value in function call = "
 << par1_value << endl;
 par2_ref = 222;
 cout << "par2_ref in function call = "
 << par2_ref << endl;
}
```

### Sample Dialogue

```
par1_value in function call = 111
par2_ref in function call = 222
n1 after function call = 1
n2 after function call = 222
```

---

# Inadvertent Local Variables

- If a function is to change the value of a variable the corresponding formal parameter must be a call-by-reference parameter with an ampersand (&) attached
- Forgetting the ampersand (&) creates a call-by-value parameter
  - The value of the variable will not be changed
  - The formal parameter is a local variable that has no effect outside the function
  - Hard error to find...it looks right!

## Inadvertent Local Variable

---

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

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

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

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

int main()
{
 using namespace std;
 int first_num, second_num;

 get_numbers(first_num, second_num);
 swap_values(first_num, second_num);
 show_results(first_num, second_num);
 return 0;
}

void swap_values(int variable1, int variable2)
{
 int temp;

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

*forgot the & here*

*forgot the & here*

*inadvertent local variables*

<The definitions of get\_numbers and show\_results are the same as in Display 4.4.>

## Sample Dialogue

```
Enter two integers: 5 10
In reverse order the numbers are: 5 10
```

---

# Class Work

- Can you
  - Write a *void*-function definition for a function called `zero_both` that has two reference parameters, both of which are variables of type *int*, and sets the values of both variables to 0.
  - Write a function that returns a value and has a call-by-reference parameter?
  - Write a function with both call-by-value and call-by-reference parameters



# Using Procedural Abstraction

# Using Procedural Abstraction

- Functions should be designed so they can be used as black boxes
- To use a function, the declaration and comment should be sufficient
- Programmer should not need to know the details of the function to use it

# Functions Calling Functions

- A function body may contain a call to another function
  - The called function declaration must still appear before it is called
    - Functions cannot be defined in the body of another function
  - Example: **void order(int& n1, int& n2)**

```
 {
 if (n1 > n2)
 swap_values(n1, n2);
 }
```

    - swap\_values called if n1 and n2 are not in ascending order
    - After the call to order, n1 and n2 are in ascending order

## Function Calling Another Function (part 1 of 2)

---

```
//Program to demonstrate a function calling another function.
#include <iostream>

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

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

void order(int& n1, int& n2);
//Orders the numbers in the variables n1 and n2
//so that after the function call n1 <= n2.

void give_results(int output1, int output2);
//Outputs the values in output1 and output2.
//Assumes that output1 <= output2

int main()
{
 int first_num, second_num;

 get_input(first_num, second_num);
 order(first_num, second_num);
 give_results(first_num, second_num);
 return 0;
}

//Uses iostream:
void get_input(int& input1, int& input2)
{
 using namespace std;
 cout << "Enter two integers: ";
 cin >> input1 >> input2;
}
```

---

## Function Calling Another Function (part 2 of 2)

---

```
void swap_values(int& variable1, int& variable2)
{
 int temp;

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

```
void order(int& n1, int& n2)
{
 if (n1 > n2)
 swap_values(n1, n2);
}
```

*These function definitions can be in any order.*

```
//Uses iostream:
void give_results(int output1, int output2)
{
 using namespace std;
 cout << "In increasing order the numbers are: "
 << output1 << " " << output2 << endl;
}
```

### Sample Dialogue

Enter two integers: 10 5

In increasing order the numbers are: 5 10

---

# Pre and Postconditions

- Precondition
  - States what is assumed to be true when the function is called
    - Function should not be used unless the precondition holds
- Postcondition
  - Describes the effect of the function call
  - Tells what will be true after the function is executed (when the precondition holds)
  - If the function returns a value, that value is described
  - Changes to call-by-reference parameters are described

# swap\_values revisited

- Using preconditions and postconditions the declaration of `swap_values` becomes:

```
void swap_values(int& n1, int& n2);
 //Precondition: variable1 and variable 2 have
 // been given values
 // Postcondition: The values of variable1 and
 // variable2 have been
 // interchanged
```

# Function celsius

- Preconditions and postconditions make the declaration for celsius:

```
double celsius(double fahrenheit);
//Precondition: fahrenheit is a temperature
// expressed in degrees Fahrenheit
//Postcondition: Returns the equivalent temperature
// expressed in degrees Celsius
```



# Preconditions and postconditions?

- Preconditions and postconditions
  - should be the first step in designing a function
  - specify what a function should do
    - Always specify what a function should do before designing how the function will do it
  - Minimize design errors
  - Minimize time wasted writing code that doesn't match the task at hand

# Case Study - Supermarket Pricing

- Problem definition
  - Determine retail price of an item given suitable input
  - 5% markup if item should sell in a week
  - 10% markup if item expected to take more than a week
    - 5% for up to 7 days, changes to 10% at 8 days
  - Input
    - The wholesale price and the estimate of days until item sells
  - Output
    - The retail price of the item

# Supermarket Pricing: Analysis

- Three main subtasks
  - Input the data
  - Compute the retail price of the item
  - Output the results
- Each task can be implemented with a function
  - Notice the use of call-by-value and call-by-reference parameters in the following function declarations

# Supermarket Pricing: get\_input

- `void get_input(double& cost, int& turnover);`  
//Precondition: User is ready to enter values  
//  
//                   correctly.  
//Postcondition: The value of cost has been set to  
//  
//                   the wholesale cost of one item.  
//  
//                   The value of turnover has been  
//  
//                   set to the expected number of  
//  
//                   days until the item is sold.

# Supermarket Pricing:Function price

- `double price(double cost, int turnover);`  
//Precondition: cost is the wholesale cost of one  
// item. turnover is the expected  
// number of days until the item is  
// sold.  
//Postcondition: returns the retail price of the item

# Supermarket Pricing: give\_output

- void give\_output(double cost, int turnover, double price);  
//Precondition: cost is the wholesale cost of one item;  
// turnover is the expected time until sale  
// of the item; price is the retail price of  
// the item.  
//Postcondition: The values of cost, turnover, and price  
// been written to the screen.

# Supermarket Pricing: main function

- With the functions declared, we can write the main function:

```
int main()
{
 double wholesale_cost, retail_price;
 int shelf_time;

 get_input(wholesale_cost, shelf_time);
 retail_price = price(wholesale_cost, shelf_time);
 give_output(wholesale_cost, shelf_time, retail_price);
 return 0;
}
```

# Supermarket Pricing: Algorithm Design

- Implementations of `get_input` and `give_output` are straightforward, so we concentrate on the price function
- pseudocode for the price function
  - If `turnover <= 7` days then
    - `return (cost + 5% of cost);`
  - else
    - `return (cost + 10% of cost);`



## Supermarket: Constants for The price Function

- The numeric values in the pseudocode will be represented by constants
  - Const double LOW\_MARKUP = 0.05; // 5%
  - Const double HIGH\_MARKUP = 0.10; // 10%
  - Const int THRESHOLD = 7; // At 8 days use  
//HIGH\_MARKUP

# Coding The price Function

- The body of the price function

```
– {
 if (turnover <= THRESHOLD)
 return (cost + (LOW_MARKUP * cost));
 else
 return (cost + (HIGH_MARKUP * cost));
}
```

- See the complete program in

## Supermarket Pricing (part 1 of 3)

---

```
//Determines the retail price of an item according to
//the pricing policies of the Quick-Shop supermarket chain.
#include <iostream>

const double LOW_MARKUP = 0.05; //5%
const double HIGH_MARKUP = 0.10; //10%
const int THRESHOLD = 7; //Use HIGH_MARKUP if do not
 //expect to sell in 7 days or less.

void introduction();
//Postcondition: Description of program is written on the screen.

void get_input(double& cost, int& turnover);
//Precondition: User is ready to enter values correctly.
//Postcondition: The value of cost has been set to the
//wholesale cost of one item. The value of turnover has been
//set to the expected number of days until the item is sold.

double price(double cost, int turnover);
//Precondition: cost is the wholesale cost of one item.
//turnover is the expected number of days until sale of the item.
//Returns the retail price of the item.

void give_output(double cost, int turnover, double price);
//Precondition: cost is the wholesale cost of one item; turnover is the
//expected time until sale of the item; price is the retail price of the item.
//Postcondition: The values of cost, turnover, and price have been
//written to the screen.

int main()
{
 double wholesale_cost, retail_price;
 int shelf_time;

 introduction();
 get_input(wholesale_cost, shelf_time);
 retail_price = price(wholesale_cost, shelf_time);
 give_output(wholesale_cost, shelf_time, retail_price);
 return 0;
}
```

---

## Supermarket Pricing (part 2 of 3)

---

```
//Uses iostream:
void introduction()
{
 using namespace std;
 cout << "This program determines the retail price for\n"
 << "an item at a Quick-Shop supermarket store.\n";
}

//Uses iostream:
void get_input(double& cost, int& turnover)
{
 using namespace std;
 cout << "Enter the wholesale cost of item: $";
 cin >> cost;
 cout << "Enter the expected number of days until sold: ";
 cin >> turnover;
}

//Uses iostream:
void give_output(double cost, int turnover, double price)
{
 using namespace std;
 cout.setf(ios::fixed);
 cout.setf(ios::showpoint);
 cout.precision(2);
 cout << "Wholesale cost = $" << cost << endl
 << "Expected time until sold = "
 << turnover << " days" << endl
 << "Retail price = $" << price << endl;
}

//Uses defined constants LOW_MARKUP, HIGH_MARKUP, and THRESHOLD:
double price(double cost, int turnover)
{
 if (turnover <= THRESHOLD)
 return (cost + (LOW_MARKUP * cost));
 else
 return (cost + (HIGH_MARKUP * cost));
}
```

---

## Supermarket Pricing (*part 3 of 3*)

---

### Sample Dialogue

```
This program determines the retail price for
an item at a Quick-Shop supermarket store.
Enter the wholesale cost of item: $1.21
Enter the expected number of days until sold: 5
Wholesale cost = $1.21
Expected time until sold = 5 days
Retail price = $1.27
```

---

# Supermarket Program Testing

- Testing strategies
  - Use data that tests both the high and low markup cases
  - Test **boundary conditions**, where the program is expected to change behavior or make a choice
    - In function price, 7 days is a boundary condition
    - Test for exactly 7 days as well as one day more and one day less

# Class Work

- Can you
  - Define a function in the body of another function?
  - Call one function from the body of another function?
  - Give preconditions and postconditions for the predefined function `sqrt`?

# Testing and Debugging



# Testing and Debugging Functions

- Each function should be tested as a separate unit
- Testing individual functions facilitates finding mistakes
- Driver programs allow testing of individual functions
- Once a function is tested, it can be used in the driver program to test other functions

## Driver Program (part 1 of 2)

---

```
//Driver program for the function get_input.
#include <iostream>

void get_input(double& cost, int& turnover);
//Precondition: User is ready to enter values correctly.
//Postcondition: The value of cost has been set to the
//wholesale cost of one item. The value of turnover has been
//set to the expected number of days until the item is sold.

int main()
{
 using namespace std;
 double wholesale_cost;
 int shelf_time;
 char ans;

 cout.setf(ios::fixed);
 cout.setf(ios::showpoint);
 cout.precision(2);
 do
 {
 get_input(wholesale_cost, shelf_time);

 cout << "Wholesale cost is now $"
 << wholesale_cost << endl;
 cout << "Days until sold is now "
 << shelf_time << endl;

 cout << "Test again?"
 << " (Type y for yes or n for no): ";
 cin >> ans;
 cout << endl;
 } while (ans == 'y' || ans == 'Y');

 return 0;
}
```

---

## Driver Program (part 2 of 2)

---

```
//Uses iostream:
void get_input(double& cost, int& turnover)
{
 using namespace std;
 cout << "Enter the wholesale cost of item: $";
 cin >> cost;
 cout << "Enter the expected number of days until sold: ";
 cin >> turnover;
}
```

### Sample Dialogue

```
Enter the wholesale cost of item: $123.45
Enter the expected number of days until sold: 67
Wholesale cost is now $123.45
Days until sold is now 67
Test again? (Type y for yes or n for no): y
```

```
Enter the wholesale cost of item: $9.05
Enter the expected number of days until sold: 3
Wholesale cost is now $9.05
Days until sold is now 3
Test again? (Type y for yes or n for no): n
```

---

# Stubs

- When a function being tested calls other functions that are not yet tested, use a **stub**
- A stub is a simplified version of a function
  - Stubs are usually provide values for testing rather than perform the intended calculation
  - Stubs should be so simple that you have confidence they will perform correctly
  - Function price is used as a stub to test the rest of the supermarket pricing program below.

## Program with a Stub (part 1 of 2)

---

```
//Determines the retail price of an item according to
//the pricing policies of the Quick-Shop supermarket chain.
#include <iostream>

void introduction();
//Postcondition: Description of program is written on the screen.

void get_input(double& cost, int& turnover);
//Precondition: User is ready to enter values correctly.
//Postcondition: The value of cost has been set to the
//wholesale cost of one item. The value of turnover has been
//set to the expected number of days until the item is sold.

double price(double cost, int turnover);
//Precondition: cost is the wholesale cost of one item.
//turnover is the expected number of days until sale of the item.
//Returns the retail price of the item.


void give_output(double cost, int turnover, double price);
//Precondition: cost is the wholesale cost of one item; turnover is the
//expected time until sale of the item; price is the retail price of the item.
//Postcondition: The values of cost, turnover, and price have been
//written to the screen.

int main()
{
 double wholesale_cost, retail_price;
 int shelf_time;

 introduction();
 get_input(wholesale_cost, shelf_time);
 retail_price = price(wholesale_cost, shelf_time);
 give_output(wholesale_cost, shelf_time, retail_price);
 return 0;
}

//Uses iostream:
void introduction()
{
 using namespace std;
 cout << "This program determines the retail price for\n"
 << "an item at a Quick-Shop supermarket store.\n";
}


```



## Program with a Stub (part 2 of 2)

---

```
//Uses iostream:
void get_input(double& cost, int& turnover)
{
 using namespace std;
 cout << "Enter the wholesale cost of item: $";
 cin >> cost;
 cout << "Enter the expected number of days until sold: ";
 cin >> turnover;
}

//Uses iostream:
void give_output(double cost, int turnover, double price)
{
 using namespace std;
 cout.setf(ios::fixed);
 cout.setf(ios::showpoint);
 cout.precision(2);
 cout << "Wholesale cost = $" << cost << endl
 << "Expected time until sold = "
 << turnover << " days" << endl
 << "Retail price= $" << price << endl;
}

//This is only a stub:
double price(double cost, int turnover)
{
 return 9.99; //Not correct, but good enough for some testing.
}
```

*fully tested function*

*function being tested*

*stub*

## Sample Dialogue

This program determines the retail price for an item at a Quick-Shop supermarket store.

```
Enter the wholesale cost of item: $1.21
Enter the expected number of days until sold: 5
Wholesale cost = $1.21
Expected time until sold = 5 days
Retail price = $9.99
```

---

# Rule for Testing Functions

- Fundamental Rule for Testing Functions
  - Test every function in a program in which every other function in that program has already been fully tested and debugged.

# Class Work

- Can you
  - Describe the fundamental rule for testing functions?
  - Describe a driver program?
  - Write a driver program to test a function?
  - Describe and use a stub?
  - Write a stub?



# Home Work

# Homework

- Pick any two class work examples and write a program for them.