CMPSC 16  
Problem Solving with Computers I  
Spring 2016

Lecture 6: Procedural Abstraction and  
Functions That Return a Value

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

```cpp
bonus = sqrt(sales) / 10;
cout << "The side of a square with area " << area << " is " << sqrt(area);
```

---

**A Function Call**

```cpp
//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 with" << endl
    << "a side length of " << length_side << endl
    << "feet on each side."

    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)**
  ```c
  int value = abs(-8);
  ```
  - Returns absolute value of argument x
  - Return value is of type int
  - Argument is of type int
  - Found in the library cstdlib

- **fabs(x)**
  ```c
  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

---

<table>
<thead>
<tr>
<th>Name</th>
<th>Description Type of Arguments</th>
<th>Type of Value Returned</th>
<th>Example</th>
<th>Value</th>
<th>Library Header</th>
</tr>
</thead>
<tbody>
<tr>
<td>sqrt</td>
<td>square root</td>
<td>double</td>
<td>double</td>
<td>sqrt(4.0)</td>
<td>2.0</td>
</tr>
<tr>
<td>pow</td>
<td>powers</td>
<td>double</td>
<td>double</td>
<td>pow(2.0,3.0)</td>
<td>8.0</td>
</tr>
<tr>
<td>abs</td>
<td>absolute value for int</td>
<td>int</td>
<td>int</td>
<td>abs(-7)</td>
<td>7</td>
</tr>
<tr>
<td>labs</td>
<td>absolute value for long</td>
<td>long</td>
<td>long</td>
<td>labs(-70000)</td>
<td>70000</td>
</tr>
<tr>
<td>fabs</td>
<td>absolute value for double</td>
<td>double</td>
<td>double</td>
<td>fabs(-7.5)</td>
<td>7.5</td>
</tr>
<tr>
<td>ceil</td>
<td>ceiling (round up)</td>
<td>double</td>
<td>double</td>
<td>ceil(3.2)</td>
<td>4.0</td>
</tr>
<tr>
<td>floor</td>
<td>floor (round down)</td>
<td>double</td>
<td>double</td>
<td>floor(3.2)</td>
<td>3.0</td>
</tr>
</tbody>
</table>
Random Number Generation

Really pseudo-random numbers
1. Seed the random number generator only once
   ```
   #include <cstdlib>
   #include <ctime>

   srand(time(0));
   ```

2. The `rand()` function returns a random integer that is greater than or equal to 0 and less than `RAND_MAX`
   ```
   rand();
   ```

Random Numbers

- Use `%` and `+` to scale to the number range you want
- For example to get a random number from 1-6 to simulate rolling a six-sided die:
  ```
  int die = (rand() % 6) + 1;
  ```
- Can you simulate rolling two dice?
- Generating a random number `x` where `10 < x < 21`?
Type Casting

Recall the problem with integer division:

```cpp
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

```cpp
static_cast<double>(total_candy)
```

produces a double representing the integer value of `total_candy`

Type Cast Example

```cpp
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 work:
     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++

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

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

---

### 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 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

Example:

```c
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);
}
```

The Return Statement

- Return statement ends the function call
- Returns the value calculated by the function
- Syntax:
  ```c
  return expression;
  ```
  - expression performs the calculation
  or
  - expression is a variable containing the calculated value

Example:

```c
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:

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

---

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

double total_cost(int number_par, double price_par);
// 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 $":
    cin >> price;
    bill = total_cost(number, price);
    cout.setf(ios::fixed);
    cout.setf(ios::showpoint);
    cout.precision(2);
    cout << number << 
    " items at 
    "$ << price << 
    " each.\n"
    << "$ " << bill << " Final bill, including tax, is $ 
    <$ " << bill
    << endl;
    return 0;
}

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);
}
```

---

**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 $21.23
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.

```c
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);
    cout.setf (ios::fixed);
    cout.setf (ios::showpoint);
    cout.precision(2);
    cout << "Number of items at $" << price << each hr."
21.21 << "Final bill, including tax, is $" << bill << end;
    return 0;
}

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 4.4  Details of a Function Call

1. Before the function is called, values of the variable 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).
2. The function call executes, and the value of number (which is 2) plugged in for number_par and value of price (which is 10.10) plugged in for price_par.
3. The body of the function executes with number_par set to 2 and price_par set to 10.10, producing the result 20.20 in subtotal.
4. When the return statement is executed, the value of the expression after return is evaluated and returned by the function, in this case (subtotal + subtotal * TAX_RATE) is (20.20 + (20.20 * 0.05)) or 21.21.
5. The value 21.21 is returned to where the function was invoked. The result is that total_cost (number, price) is replaced by the return value of 21.21. The value of bill (got on the left-hand side of the equation) is set equal to 21.21 when the statement `bill = total_cost (number, price);` finally ends.
Alternate Declarations

- Two forms for function declarations
  - List formal parameter names
    - helpful for description of the function in comments
  - List types of formal parameters, but not names

- 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!
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
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.
**bool Return Values**

- A function can return a bool value
  - Such a function can be used where a boolean expression is expected
    - Makes programs easier to read

- Instead of
  
  ```
  if (((rate >=10) && ( rate < 20)) || (rate == 0))
  ```
  
  it is easier to read
  
  ```
  if (appropriate(rate))
  ```
  
  - If function appropriate returns a bool value based on the the expression above

**Function appropriate**

- To use function appropriate in the if-statement

  ```
  if (appropriate (rate))
  {
      ...
  }
  ```

  appropriate could be defined as

  ```
  bool appropriate(int rate)
  {
      return (((rate >= 10) && (rate < 20)) || (rate == 0));
  }
  ```
Procedural Abstraction

4.4

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
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
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
    \[
    \text{diameter\_small} \quad \text{diameter\_large} \\
    \text{price\_small} \quad \text{price\_large}
    \]

• 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
  – Compute the area of the pizza
  – Return the value of \( \frac{\text{price}}{\text{area}} \)
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 = \pi \times radius \times radius
  return (price / area)

Buying Pizza The Calls of unitprice

- Main part of the program implements calls of unitprice as

```cpp
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
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

---

Buying Pizza (part 1 of 2)

```cpp
// 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);

    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;
}
```

---

Buying Pizza (part 2 of 2)

```cpp
//\ndouble unitprice(int diameter, double price)
{
    const double PI = 3.14159;
    double radius, area;
    radius = diameter*static_cast<double>(3); 
    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: $7.00
Enter diameter of a large pizza (in inches): 13
Enter the price of a large pizza: $14.75
Small pizza:
Diameter = 10 inches
Price = $7.00 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!
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

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

```cpp
// 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 << " end!
    " << "Max number of peas per pod = " << max_count << " end!
    " << "Pod count = " << pod_count << " end!
    " << "Average pea weight = " << average_pea << " ounces" << endl;
    " << "Estimated average yield = " << yield << " ounces" << endl;
    return 0;
}
```

Sample Dialogue

Enter minimum and maximum number of peas in a pod: 30
Enter the number of pods: 10
Enter the weight of an average pea (in ounces): 0.5
Min number of peas per pod = 30
Max number of peas per pod = 10
Pod count = 10
Average pea weight = 0.5 ounces
Estimated average yield = 25000 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

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

cost 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;
}
```

```cpp
//Computes the area of a circle and the volume of a sphere.
//Uses the same radius for both calculations.

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 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
Block Scope

- Local and global variables conform to the rules of Block Scope
  - The code block (generally defined by the { }) where an identifier like a variable is declared determines the scope of the identifier
  - Blocks can be nested
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

```cpp
#include <iostream>

const double GLOBAL_CONST = 1.0;

int function1 (int param);

int main()
{
    int x;
    double d = GLOBAL_CONST;
    for (int i = 0; i < 10; i++)
    {
        x = function1(i);
    }
    return 0;
}

int function1 (int param)
{
    double y = GLOBAL_CONST;
    ...
    return 0;
}
```
Example: Factorial

- n! Represents the factorial function
- n! = 1 x 2 x 3 x ... x n

Factorial function:
- 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
Factorial Function

Function Declaration

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

Function Definition

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

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

```c++
#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;
}

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
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:

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

- Given the definition
  ```cpp
  double mpg(double miles, double gallons)
  {
    return (miles / gallons);
  }
  ```
  what will happen if mpg is called in this way?
  ```cpp
  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