Functions

- Programmer-Defined Functions
- Local Variables in Functions
- Overloading Function Names
- void Functions,
- Call-By-Reference Parameters in Functions

Programmer-Defined Functions

```cpp
/* calculate_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 << number << " items at "
         << "$" << price << " each.\n"
         << "Final bill, including tax, is $" << bill <<endl;
    return 0;
}

double total_cost (int number_par, double price_par)
{
    const double TAX_RATE = 0.05;
    double subtotal;

    subtotal = price_par * number_par;

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

function declaration

function call

function header

function body

function definition

Programmer-Defined Functions

- Two components
  - Function **declaration** (or function prototype)
    - Shows how the function is called
    - Must appear in the code before the function can be called
    - Syntax:

```
Type_returned  Function_Name(Parameter_List);
//Comment describing what function does
```

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

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

Function **Declaration**

- Tells the return type
- Tells the name of the function
- Tells how many arguments are needed
- Tells the types of the arguments
- Tells the formal parameter names
  - Formal parameters are like placeholders for the actual arguments used when the function is called
  - Formal parameter names can be any valid identifier
- Example:

```
double total_cost(int number_par, double price_par);
// Compute total cost including 5% sales tax on
// number_par items at cost of price_par each
```

Function **Definition**

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

- 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;
```

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

```cpp
/* calculate_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 << number << " items at "
         << "$" << price << " each.\n"
         << "Final bill, including tax, is $" << bill <<endl;
    return 0;
}

double total_cost (int number_par, double price_par)
{
    const double TAX_RATE = 0.05;
    double subtotal;

    subtotal = price_par * number_par;

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

1. Before the function is called, values of the variable **number** and **price** are set to 2 and 10, by **cin** statements.

As for this function call, number and price are arguments

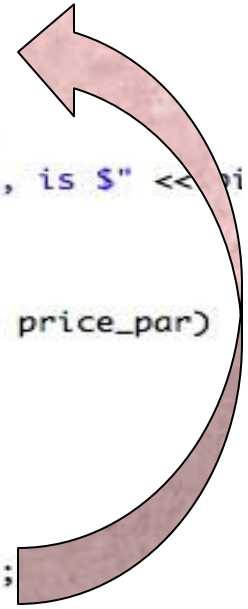
2. The function call executes and the value of **number** (which is 2) plugged in for **number_par** and value of **price** (which is 10.10) plugged in for **price_par**.

```
/* calculate_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 << number << " items at "
         << "$" << price << " each.\n"
         << "Final bill, including tax, is $" << bil
    return 0;
}

double total_cost (int number_par, double price_par)
{
    const double TAX_RATE = 0.05;
    double subtotal;

    subtotal = price_par * number_par;

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

3. The body of the function executes with **number_par** set to 2 and **price_par** set to 10.10, producing the value 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**.

```cpp
/* calculate_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 << number << " items at "
         << "$" << price << " each.\n"
         << "Final bill, including tax, is $" <<
    return 0;
}

double total_cost (int number_par, double price_par)
{
    const double TAX_RATE = 0.05;
    double subtotal;

    subtotal = price_par * number_par;

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



5. The value 21.21 is returned to where the function was **invoked** or **called**. The result is that **total_cost (number, price)** is replaced by the return value of **21.21**. The value of **bill** is set equal to 21.21 when the statement **bill=total_cost(number,price);** ends.

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

- 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);
```

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)

Function Declarations

- Two forms for function declarations
  - List formal parameter names
  - List types of formal parameters, but not names
  - Description of the function in comments
- Examples:

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

double total_cost(int, double);
```

- But in definition, 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 order**!
- 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 …

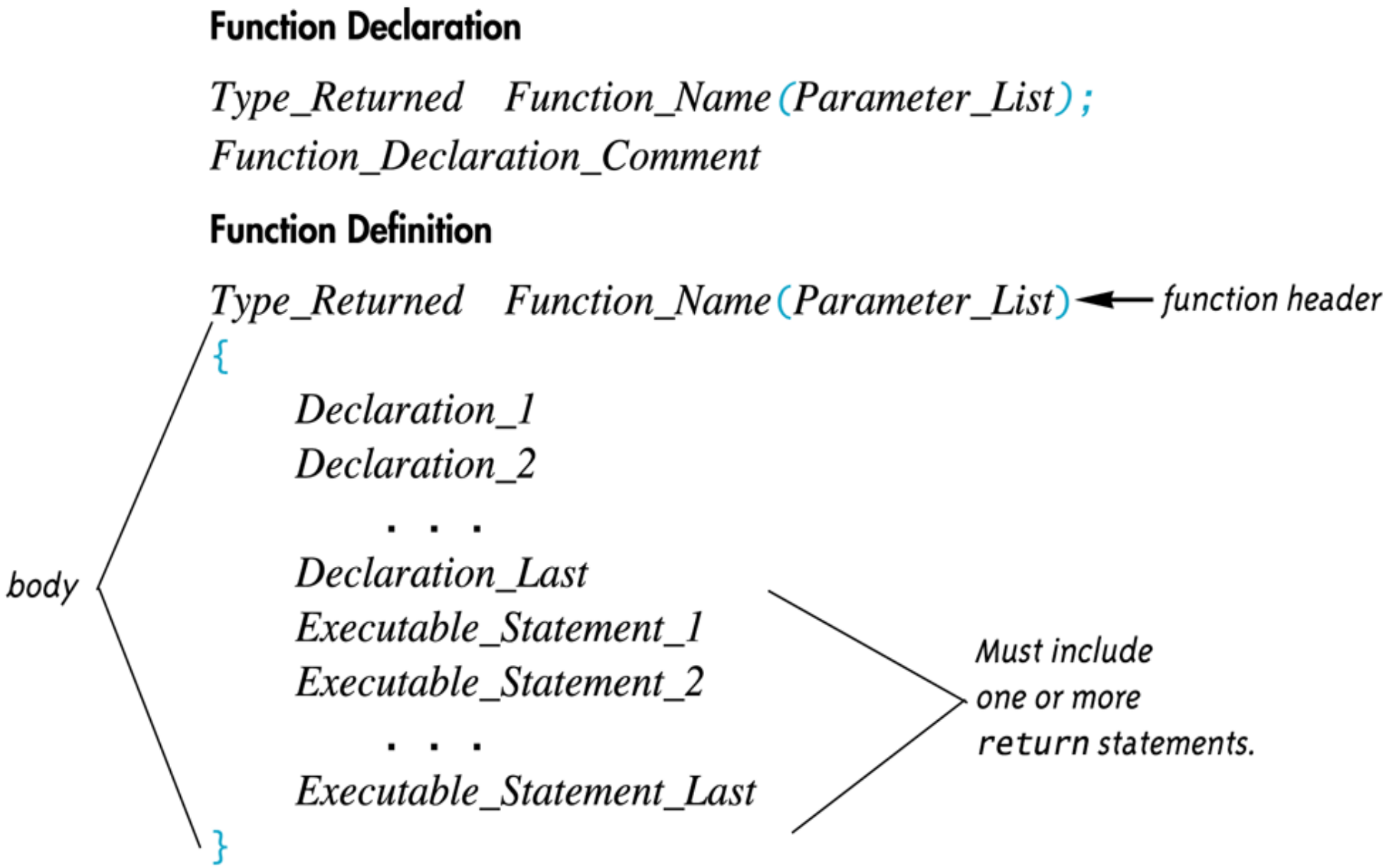
- Variables must be declared before they are used
- Variables are typically declared before the executable statements begin

```
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);
}
```

- At least one `return` statement must end the function
  - Each branch of an `if-else` statement or a `switch` statement **might** have its own return statement

Example: `char grade(int received_par, int min_score_par)`

Syntax for a Function That Returns a Value



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.

Formal Parameter Names

- Functions are designed as self-contained modules
- 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

Recall the memory structure of a program.

Example next

Simpler Formal Parameter Names

Function Declaration

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

Function Definition

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

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

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

Local Variables in Functions

Local variables in a function

- Variables declared in a function:
  - Are **local** to that function, i.e., 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 << "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
  - declared outside any function body
  - declared outside the main function body
  - declared before any function that uses it
  - available to more than one function as well as the main part of the program
- Example:

```
const double PI = 3.14159;
double area(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 keyword **const** is not used
  - Generally make programs more difficult to understand and maintain

Formal Parameters are Local Variables

- Formal Parameters are 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**, as 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 << "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.

Another example

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
```

Block Scope

Local and global variables conform to the rules of **Block Scope**

- The code block, generally specified by the `{ }`, where an identifier like a variable is declared. It determines the scope of the identifier.

- Blocks can be nested

Block Scope Revisited

```
 1    #include <iostream>
 2    using namespace std;
 3
 4    const double GLOBAL_CONST = 1.0;
 5
 6    int function1 (int param);
 7
 8    int main()
 9    {
10        int x;
11        double d = GLOBAL_CONST;
12
13        for (int i = 0; i < 10; i++)
14        {
15            x = function1(i);
16        }
17        return 0;
18    }
19
20    int function1 (int param)
21    {
22        double y = GLOBAL_CONST;
23        ...
24        return 0;
25    }
```

Local and Global scope are examples of Block scope. A variable can be directly accessed only within its scope.

Block scope: Variable ***i*** *has scope from lines 13-16*

Local scope to ***main:*** *Variable* ***x*** *has scope from lines 10-18 and variable* ***d*** *has scope from lines 11-18*

Global scope: The constant ***GLOBAL_CONST*** *has scope from lines 4-25 and the function* ***function1*** *has scope from lines 6-25*

Local scope to ***function1:*** *Variable* ***param*** *has scope from lines 20-25 and variable* ***y*** *has scope from lines 22-25*

A variable can be directly accessed only within its scope.
Local and Global scopes are examples of Block Scope.

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));
}

double volume(double radius)
{
    using namespace std;

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

Overloading Function Names

Overloading Function Names

- Overloading a function name means
  providing more than one declaration and definition using the same function name
- 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

```
//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);
}

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

two arguments

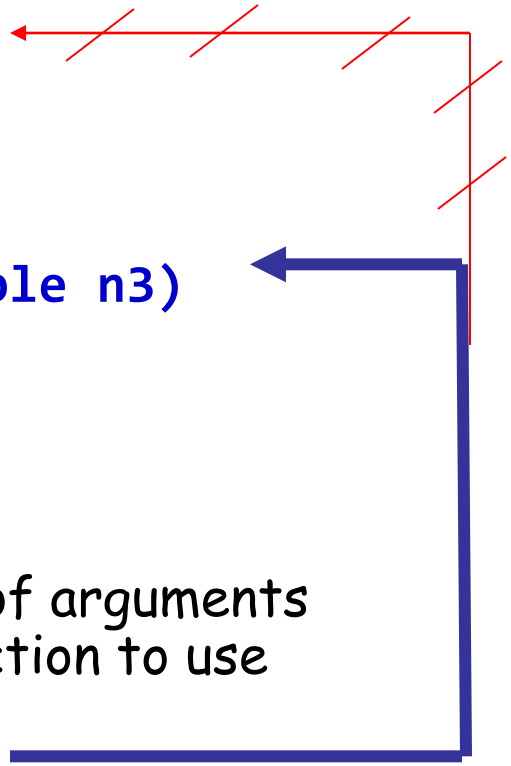
three arguments

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 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 return a value of the same type

  in addition, they ...
  - must have different numbers of formal parameters
    AND / OR
  - must have at least one different type of parameter

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!
```

void Functions

Function regarded as code to do some subtask

- A subtask might produce
  - No value (e.g., input or output) to be used by a calling function.
  - One value to be used by the calling function.
  - Multiple values to be used by the calling function.
- We have seen how to implement functions that return **one** value, through a `return` statement
- A void-function implements a subtask that ...
  - either does not give back any value to the calling function
    - no return statement

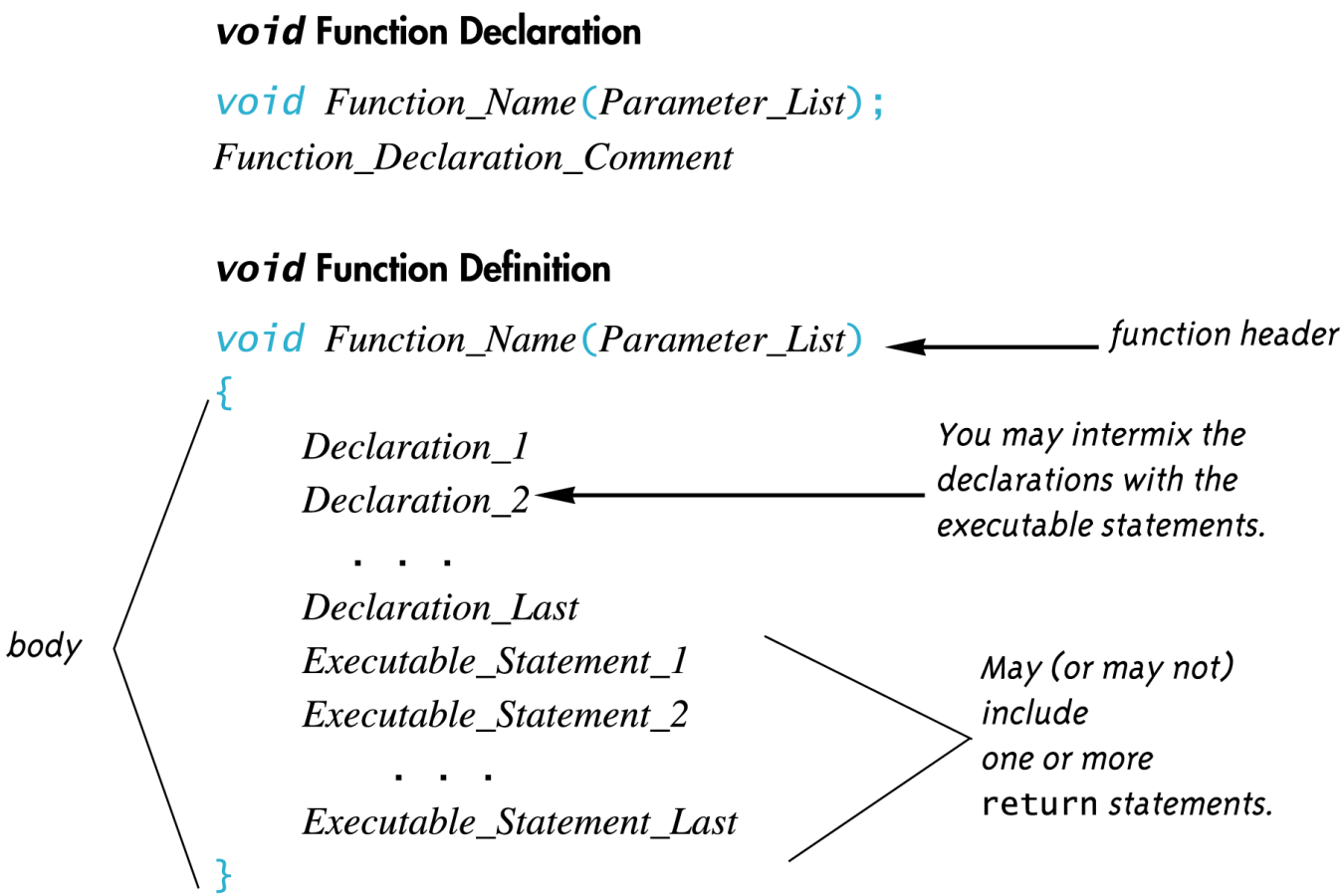
      or use **`return;`**
  - or gives back multiple values to the calling function, via the call-by-reference parameters

void-Function Definition

- Differences between **void-function** definitions and the definitions of functions that return **one** value thru return statement.
  - Keyword void replaces the type of the value returned
    - void means that no value is returned by the function thru return statement
  - The return statement does not include and expression, or can be removed in some situations.
- Example:

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

Syntax for a *void* Function Definition



Calling a void-Function

- A void-function call
  - does not need to be part of another statement
  - it ends 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 (or plugged in) 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

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 << 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 for some scenarios, e.g.
    - 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 the example on next page (Display 5.3), 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

Call-By-Reference Parameters in Functions

Call-by-Reference Parameters

- **Call-by-value**
  - A call-by-value parameter of a function receives the values of the corresponding argument during the execution of the function call
  - Any change made to the value of the parameter in the function body dose not affect the value of the argument
- **Call-by-reference**
  - A call-by-reference parameter of a function is just another name of the corresponding argument during the execution of the function call
    - The call-by-reference parameter and the argument refers to the same memory bock.
  - Any change made on the value of the parameter in the function body is essentially the change made on the value of the argument
  - Arguments for call-by-reference parameters must be variables, not numbers

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
```

Example: swap_values

```
void swap(int& variable1, int& variable2)
{
      int temp = variable1;
      variable1 = variable2;
      variable2 = temp;
}
```

- & symbol (ampersand) identifies `variable1` and `variable2` as call-by-reference parameters
  - used in both declaration and definition!
- If called with statement …

  `swap(first_num, second_num);`
  - `first_num` is substituted for `variable1` in the parameter list
    `first_num` and `variable1` are two names for the same variable
  - `second_num` is substituted for `variable2` in the parameter list
    `second_num` and `variable2` are two names for the same variable
  - `temp` is assigned the value of `variable1` (or `first_num`)
  - `variable1` (or `first_num`) is assigned the value in `variable2` (or `second_num`)
  - `variable2` (or `second_num`) is assigned the original value of `variable1` (or `first_num`) which was stored in `temp`

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

Mixed Parameter Lists

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

- Example, consider the following function declaration
  ```
  void good_stuff(int& par1, int par2, double& par3);
  ```
  - par1 and par3 are call-by-reference formal parameters
    - Changes in par1 and par3 are the changes made on the corresponding argument variables during function call.
  - par2 is a call-by-value formal parameter
    - Changes in par2 do not change the argument variable during function call

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
```