

Programming with Functions

July 25, 2010

We Know A Lot about Functions Already

C++ functions are conceptually similar to the $f(x)$ you know from mathematics.

```
5 int main()
6 {
7     double x;
8     cout << "Enter_a_value_for_the_square_root." << endl;
9     cin >> x;
10
11    double x_root = sqrt( x );
12    cout << "The_sqrt_of_" << x << "_is_" << x_root << endl;
13
14    return 0;
15 }
```

How many functions are shown here?

Describe the input(s) and output(s)

We Know A Lot about Functions Already

C++ functions are conceptually similar to the $f(x)$ you know from mathematics.

```

5 int main()
6 {
7     double x;
8     cout << "Enter_a_value_for_the_square_root." << endl;
9     cin >> x;
10
11    double x_root = sqrt( x );
12    cout << "The_sqrt_of_" << x << "is_" << x_root << endl;
13
14    return 0;
15 }

```

How many functions are shown here?*

Describe the input(s) and output(s)

\sqrt{x} ~ double sqrt(double x)

*There are actually six — but we won't learn about the other four until we study *operator overloading*.

$$f(x,y) \sim \text{double } f(\text{double } x, \text{double } y)$$

- ▶ Put two values in (x and y), get one out.
 1. The inputs are called **arguments**.
 2. In mathematics we might say “the value of $f(x,y)$ ” or “the value of f at (x,y) ”, in C++ we say **f returns the value...**
 3. In mathematics, all functions take some type of numerical inputs, in C++ functions might take `string` or `bool` inputs, they may not take any arguments at all!
- ▶ Functional composition works as expected in C++:

$$z = f(\sin(y), 3) \quad \equiv \quad \text{double } z = f(\sin(y), 3);$$

Why Functions?

Logic Encapsulation & Reuse — D.R.Y.

- ▶ Provides re-usability of common or often used algorithms (such as prompting a user for input, or calculating a complex multi-variable equation).
- ▶ Let's you maintain “algorithm focus” — don't clutter up *any* function (specifically `main()`) with “little bits of logic” that aren't intrinsic to the task at hand.
- ▶ Allows you to provide a single, explicit interface to data or algorithms specific to your application. For instance, an interface for calculating specialized equations, access to a laboratory device, or access to a database of collected data, ...

Types of Functions

User-Defined Function A function you have written.

Library Function Function you use but didn't write.

Library functions come in many different forms: commercial libraries, FOSS libraries, system libraries, your partner's library.

`main()` is a user-defined function provided for every application.

Function Anatomy

Parts of a Function

```
1 /**
2  * Main function comment describes what the function does, special properties
3  * of its arguments, what it returns, and its return interpretation.
4  */
5 return_type function_name( arg1_type arg1_name, arg2_type arg2_name )
6 {
7     variable declarations;
8
9     function statements;
10
11     return value;
12 }
```

- ▶ All `_names` must be valid **identifiers** (A-Za-z_1-90).
- ▶ Each `argX_type` is a variable, array, or class type (int, double, ifstream).
- ▶ **function definition** = The whole thing.
- ▶ **function body** = { variable declarations → return value; }
- ▶ **function header** = **function definition** - **function body**

Parts of a Function

```
5 /**
6  * Returns the probability that you will flip h heads out of
7  * N total coin tosses with an UNFAIR coin with bias q for heads
8  */
9 double coin_toss( int heads, int N, double q )
10 {
11     double combinations = binomial_coefficient( N, N-heads );
12     double one_combo_probability = pow(q, heads)*pow(1-q, N-heads );
13     return combinations*one_combo_probability;
14 }
```

- ▶ The type of value returned matches the `return_type`.
- ▶ The variables `heads`, `N`, and `q` are **formal parameters** of the `cointoss` function.
- ▶ Everything between the `()` of the function header is called the **parameter declarations** or **argument list**.
- ▶ Functions may **call** other functions! **Our `main()`s usually call other functions, don't they...**

Parts of a Function

```

5 /**
6  * Returns the probability that you will flip h heads out of
7  * N total coin tosses with an UNFAIR coin with bias q for heads
8  */
9 double coin_toss( int heads, int N, double q )
10 {
11     double combinations = binomial_coefficient( N, N-heads);
12     double one_combo_probability = pow(q, heads)*pow(1-q,N-heads);
13     return combinations*one_combo_probability;
14 }

```

- ▶ The type of value returned matches the `return_type`.
- ▶ The variables `heads`, `N`, and `q` are **formal parameters** of the `cointoss` function.
- ▶ Everything between the `()` of the function header is called the **parameter declarations** or **argument list**.
- ▶ Functions may **call** other functions! **Our `main()`s usually call other functions, don't they...**

$$\binom{N}{heads} q^{heads} (1-q)^{N-heads} = \left(\frac{\prod_{i=1}^N i}{\prod_{i=1}^{heads} i \prod_{i=1}^{N-heads} i} \right) q^{heads} (1-q)^{N-heads}$$

Try questions 1–3.

Using Functions

Calling a Function from main()

```
1 /**
2  * Use a function by defining it before it is used.
3  */
4 #include <iostream>
5 using namespace std;
6 /** Returns true if legs a, b, c form a triangle */
7 bool valid_triangle( double a, double b, double c)
8 {
9     if( a > b && a > c ) {
10         // a greatest
11         return b + c > a;
12     } else if( b > a && b > c ) {
13         // b greatest
14         return a + c > b;
15     }
16
17     // otherwise, c greatest
18     return a + b > c;
19 }
20 int main()
21 {
22     double aleg(3), bleg(4), cleg(5);
23     // Function call, result saved
24     bool okTriangle = valid_triangle( aleg, bleg, cleg );
25     if( okTriangle ) {
26         cout << "OK Triangle!_" << aleg << "_" << bleg <<
27             "_" << cleg << endl;
28     }
29     // Exit the main function with code 0
30     return 0;
31 }
```

Everything Up Front
before main()
OK

Calling a Function from main()

```
6 int main()
7 {
8     double aleg(3), bleg(4), cleg(5);
9     // Function call, result saved
10    bool okTriangle = valid_triangle( aleg, bleg, cleg );
11    if( okTriangle ) {
12        cout << "OK_Triangle!_" << aleg << "_" << bleg <<
13            "_" << cleg << endl;
14    }
15    // Exit the main function with code 0
16    return 0;
17 }
18 /** Returns true if legs a, b, c form a triangle */
19 bool valid_triangle( double a, double b, double c )
20 {
21     if( a > b && a > c ) {
22         // a greatest
23         return b + c > a;
24     } else if( b > a && b > c ) {
25         // b greatest
26         return a + c > b;
27     }
28
29     // otherwise, c greatest
30     return a + b > c;
31 }
```

Everything Down Below
after main()
NOT OK

In function 'int main()':
10: error: 'valid_triangle' was not declared in this scope

Prototypes, Header files, and Implementation Files

Calling a Function from main()

function prototype = function header + ;

Prototype Before main
OK

The function prototype and the function header must agree (outside of variable names).

Function prototypes can even

be in other files
(Consider sqrt()
from #include <cmath>.)

```

6 /** Returns true if legs a, b, c form a triangle */
7 bool valid_triangle( double a, double b, double c);
8
9 int main()
10 {
11     double aleg(3), bleg(4), cleg(5);
12     // Function call, result saved
13     bool okTriangle = valid_triangle( aleg, bleg, cleg );
14     if( okTriangle ) {
15         cout << "OK Triangle!" << aleg << " " << bleg <<
16             " " << cleg << endl;
17     }
18     // Exit the main function with code 0
19     return 0;
20 }
21
22 bool valid_triangle( double a, double b, double c)
23 {
24     if( a > b && a > c ) {
25         // a greatest
26         return b + c > a;
27     } else if( b > a && b > c ) {
28         // b greatest
29         return a + c > b;
30     }
31     // otherwise, c greatest
32     return a + b > c;
33 }

```

Calling a Function from `main()`

```

6
7 /** Returns true if legs a, b, c form a triangle */
8 bool valid_triangle( double a, double b, double c);
9
10 int main()
11 {
12     double aleg(3), bleg(4), cleg(5);
13     // Function call, result saved
14     bool okTriangle = valid_triangle( aleg, bleg, cleg );
15     if( okTriangle ) {
16         cout << "OK Triangle!" << aleg << " " << bleg <<
17             " " << cleg << endl;
18     }
19     // Exit the main function with code 0
20     return 0;
21 }
22
23 // NO DEFINITION FOR valid_triangle!

```

Missing Function
Definition
NOT OK

This is a *linking* error!

```

In function 'main':
14: undefined reference to 'valid_triangle(double, double, double)'
ld returned 1 exit status

```

Calling a Function from main()

```

4 /** Returns true if legs a, b, c form a triangle */
5 bool valid_triangle( int a, int b, int c);
6 int main()
7 {
8     double aleg(3), bleg(4), cleg(5);
9     // Function call, result saved
10    bool okTriangle = valid_triangle( aleg, bleg, cleg );
11    if( okTriangle ) {
12        cout << "OK_Triangle!_" << aleg << "_" << bleg <<
13            "_" << cleg << endl;
14    }
15    // Exit the main function with code 0
16    return 0;
17 }
18 bool valid_triangle( double a, double b, double c)
19 {
20     if( a > b && a > c ) {

```

Wrong Prototype
NOT OK

```

In function 'main':
10: undefined reference to 'valid_triangle(int, int, int)'
ld returned 1 exit status

```

Separating Prototype and Implementation

Header File: triangles.h

```

1 /**
2  * Header file for triangle utilities
3  */
4 #pragma once
5 #include <iostream>
6 using namespace std;
7 /** Returns true if legs a, b, c form a triangle */
8 bool valid_triangle( double a, double b, double c);

```

Implementation File: triangles.cpp

```

1 /**
2  * Utilities for triangles
3  */
4 #include "triangles.h"
5 bool valid_triangle( double a, double b, double c )
6 {
7     if( a > b && a > c ) {
8         // a greatest
9         return b + c > a;
10    } else if( b > a && b > c ) {
11        // b greatest
12        return a + c > b;
13    }
14    // otherwise, c greatest
15    return a + b > c;
16 }

```

Keep in the same folder or directory.

Note the

`#pragma once`
in the header file.

Note the

`#include "triangles.h"`
in the implementation file.

Separating Prototype and Implementation

```
4 #include <iostream>
5 using namespace std;
6
7 #include "triangles.h"
8
9 int main()
10 {
11     double aleg(3), bleg(4), cleg(5);
12     // Function call, result saved
13     bool okTriangle = valid_triangle( aleg, bleg, cleg );
14     if( okTriangle ) {
15         cout << "OK_Triangle!_" << aleg << "_" << bleg <<
16             "_" << cleg << endl;
17     }
18     // Exit the main function with code 0
19     return 0;
20 }
```

Keep in the same folder or directory.

Note the

`#include "triangles.h"`
in the application source file.

Calling Functions from Other Functions

Since `main()` is
just another function in C++,
everything we've learned about calling functions from
`main()`
applies to
calling functions from other functions.

Strange Functions

FUNCTIONS II

void Functions and Empty Argument Lists

```
4 /**
5  * Print a "Hello World" message to cout.
6  */
7 void say_hello_world( void )
8 {
9     cout << "Hello_world, I'm_a_function!" << endl;
10    return;
11 }
```

- ▶ The void as a return type indicates no return value.
These are void functions.
- ▶ The void keyword in place of an argument list, or simply an empty argument list: (), indicates no function arguments.
- ▶ void functions are useful for their **side-effects**.
- ▶ return statement can still be used (**without an argument**), but it is not required.

Try questions 7–8.

Passing Arguments by Value and Reference

Argument Passing

```
1 #include <iostream>
2 using namespace std;
3
4 /** increments arg by one */
5 int increment( int arg )
6 {
7     ++arg;
8     return arg;
9 }
10
11 int main()
12 {
13     int v(0), w;
14     cout << "v=" << v <<
15         "before_increment" << endl;
16
17     w = increment( v );
18
19     cout << "v=" << v << "w=" << w <<
20         "after_increment" << endl;
21
22     return 0;
23 }
```

```
v=0 before increment
v=0 w=1 after increment
```

Argument Passing

Call by Value

```

1 #include <iostream>
2 using namespace std;
3
4 /** increments arg by one */
5 int increment( int arg )
6 {
7     ++arg;
8     return arg;
9 }
10
11 int main()
12 {
13     int v(0), w;
14     cout << "v=" << v <<
15         "\_before\_increment" << endl;
16
17     w = increment( v );
18
19     cout << "v=" << v << "\_w=" << w <<
20         "\_after\_increment" << endl;
21
22     return 0;
23 }

```

v=0 before increment
v=0 w=1 after increment

Call by Reference&

```

1 #include <iostream>
2 using namespace std;
3
4 /** increments arg by one */
5 int increment( int& arg )
6 {
7     ++arg;
8     return arg;
9 }
10
11 int main()
12 {
13     int v(0), w;
14     cout << "v=" << v <<
15         "\_before\_increment" << endl;
16
17     w = increment( v );
18
19     cout << "v=" << v << "\_w=" << w <<
20         "\_after\_increment" << endl;
21
22     return 0;
23 }

```

v=0 before increment
v=1 w=1 after increment

Call by Value

```

1 #include <iostream>
2 using namespace std;
3
4 /** increments arg by one */
5 int increment( int arg )
6 {
7     ++arg;
8     return arg;
9 }
10
11 int main()
12 {
13     int v(0), w;
14     cout << "v=" << v <<
15         " before increment" << endl;
16
17     w = increment( v );
18
19     cout << "v=" << v << " w=" << w <<
20         " after increment" << endl;
21
22     return 0;
23 }

```

```

v=0 before increment
v=0 w=1 after increment

```

- ▶ main provides a *copy* of v's *value* to increment().
- ▶ **Two different memory locations are used for the same value!**
- ▶ when increment() changes the value of arg, it changes the value in the memory location increment() “owns.”
- ▶ the contents of main()'s memory location for v is unchanged.

Call by Reference&

- ▶ main provides its *memory location* of v to increment().
- ▶ C++ Reference == Memory Location
- ▶ **One memory location is referred to by two different names.** increment() calls it arg, and main() calls it v.
- ▶ when increment() changes the value of its arg, it is “seen” as a side-effect in main.
- ▶ `increment(3);` is not allowed, the argument must be a variable.

```

1 #include <iostream>
2 using namespace std;
3
4 /** increments arg by one */
5 int increment( int& arg )
6 {
7     ++arg;
8     return arg;
9 }
10
11 int main()
12 {
13     int v(0), w;
14     cout << "v=" << v <<
15         " before increment " << endl;
16
17     w = increment( v );
18
19     cout << "v=" << v << " w=" << w <<
20         " after increment " << endl;
21
22     return 0;
23 }

```

```

v=0 before increment
v=1 w=1 after increment

```

The & Address Operator

Appending & to any type creates a variable that stores a *memory address* for an object of that particular type.

<code>int& a</code>	Stores the memory location for an integer variable
<code>double& b</code>	Stores the memory location for a double variable
<code>ofstream& c</code>	Stores the memory location for an output file stream object

Try questions 9–12.

const Parameters

FUNCTIONS III

const Qualifier with Formal Parameters

```
1 int brokenFunction( const int a, const double& b )
2 {
3     // this function can access the values of
4     // formal parameters a and b
5     double c = a + b;
6
7     // it can copy their value to local variables
8     // and modify *local* copies that "belong" to
9     // brokenFunction
10    int aye = a;
11    double bee = b;
12    aye++;
13    bee += c;
14
15    // but it CANNOT modify the
16    // formal parameters themselves!
17    b = a = 0;
18
19    return 0;
20 }
```

The const qualifier may be used on either call-by-value or pass-by-reference parameters. . .

the result is that the compiler **produces error messages** when the programmer breaks a promise.

```
In function 'int brokenFunction(int, const double&)':
17: error: assignment of read-only parameter 'a'
17: error: assignment of read-only reference 'b'
```

Arrays as Function Arguments

Passing Arrays to Functions

Arrays may be of arbitrary length,
so we don't want to copy each element value.
(CPUs are faster, memory busses not so much : ()

Arrays are always passed by reference —
this is a C++'ism, the programmer cannot change this.

Array Argument “Recipes”

There are other ways to weasel arrays into functions as parameters, but this is the only universally agreed upon **best practice**.

So learn these rules, and use only these rules.

Array Argument “Recipes”

```
11 /**
12  * Elements of array may be changed by arrayFunction
13  * count is the number of valid elements in array
14  */
15 void arrayFunction( int array[], const int count );
16
17 /**
18  * Elements of array may NOT be changed by arrayFunction
19  * count is the number of valid elements in array
20  */
21 void arrayFunction( const int array[], const int count );
```

Note the
const int count
parameter.

It is **NOT** the “size” of the
array.

It **IS** how many **valid**
elements are in the array!

Array Argument “Recipes”

```
11 /**
12  * Elements of array may be changed by arrayFunction
13  * count is the number of valid elements in array
14  */
15 void arrayFunction( int array[], const int count );
16
17 /**
18  * Elements of array may NOT be changed by arrayFunction
19  * count is the number of valid elements in array
20  */
21 void arrayFunction( const int array[], const int count );
```

Don't supply an array size inside the [] of the parameter, not needed for 1d arrays passed to functions.

Note the use of `const` to dictate whether the array elements may be changed or not.

Array Argument “Recipes”

Note the static const int **global** variable (line 31) specifying the number of **columns** these functions expect in the array parameter.

This is the **ONLY** acceptable use of a **global** variable.

```

27 /**
28  * the only acceptable global variable in a header file is
29  * a const such as this ...
30  */
31 static const int COLUMNS(32);
32
33 /**
34  * Elements of array2d may be changed by arrayFunction
35  * rows is the number of valid rows in array2d
36  * cols is the number of valid columns in array2d
37  */
38 // This proto is for partially filled rows, complete columns
39 void arrayFunction( double array2d[][COLUMNS], const int rows );
40 // This proto is for partially filled rows AND columns
41 void arrayFunction( double array2d[][COLUMNS], const int rows,
42                    const int cols );

```

Array Argument “Recipes”

Note the connection to *2d* array declarations.

Only the size of the **first dimension** may be omitted.

```
27 /**
28  * the only acceptable global variable in a header file is
29  * a const such as this...
30  */
31 static const int COLUMNS(32);
32
33 /**
34  * Elements of array2d may be changed by arrayFunction
35  * rows is the number of valid rows in array2d
36  * cols is the number of valid columns in array2d
37  */
38 // This proto is for partially filled rows, complete columns
39 void arrayFunction( double array2d[][COLUMNS], const int rows );
40 // This proto is for partially filled rows AND columns
41 void arrayFunction( double array2d[][COLUMNS], const int rows,
42                    const int cols );
```

Array Argument “Recipes”

Again, note that rows and columns represent the **number of valid elements** in the array.

They **do not** represent the “size” of the array.

```
27 /**
28  * the only acceptable global variable in a header file is
29  * a const such as this ...
30  */
31 static const int COLUMNS(32);
32
33 /**
34  * Elements of array2d may be changed by arrayFunction
35  * rows is the number of valid rows in array2d
36  * cols is the number of valid columns in array2d
37  */
38 // This proto is for partially filled rows, complete columns
39 void arrayFunction( double array2d[][COLUMNS], const int rows );
40 // This proto is for partially filled rows AND columns
41 void arrayFunction( double array2d[][COLUMNS], const int rows,
42                    const int cols );
```

Array Argument “Recipes”

```
33 /**
34  * Elements of array2d may be changed by arrayFunction
35  * rows is the number of valid rows in array2d
36  * cols is the number of valid columns in array2d
37  */
38 // This proto is for partially filled rows, complete columns
39 void arrayFunction( double array2d[][COLUMNS], const int rows );
40 // This proto is for partially filled rows AND columns
41 void arrayFunction( double array2d[][COLUMNS], const int rows,
42                    const int cols );
43
44 // Elements of array2d may NOT be changed by arrayFunction
45 void arrayFunction( const double array2d[][COLUMNS], const int rows );
46
47 void arrayFunction( const double array2d[][COLUMNS], const int rows,
48                    const int cols );
```

What do the first const terms on lines 45 & 447 mean?

Passing Arrays as Function Parameters

[Edit Program](#)
[Run Program](#)

```

9  /**
10 * Returns the average value of data.
11 */
12 double average( const double data[], const int size );
13 /**
14 * Returns the number of elements in data that are
15 * greater than threshold.
16 */
17 int gt_threshold( const double data[], const int size ,
18                 double threshold );
19
20 int main()
21 {
22     const int SIZE(32);
23     double userData[SIZE];
24     int count(0);
25     // read data from user
26     cout << "Enter double values , 'end' to stop." << endl;
27     while( count<SIZE && cin >> userData[count] ) {
28         count++;
29     }
30     // find average
31     double mean = average( userData , count );
32     // find number of elements > 0.72*mean
33     const double FACTOR(0.72);
34     int gtFactor =
35         gt_threshold( userData , count , FACTOR*mean );

```

Notice that the `[]` are **NOT** part of an array's name...

... so they are **NOT** used when specifying an array parameter to a function.

(Lines 31 & 35)

An Unlikely Limitation

Functions **cannot** return arrays!

If you need an array "returned" by a function, then one must be passed in (non-const) for the function to value and the calling routine to access **after** the function completes executing.

Bubble Sort

Objects as Function Parameters

FUNCTIONS IV

Objects as Function Parameters

As a programmer writing a function, you do not know all the ways the function might be used: by yourself, your programming team, or the next generation of programmers that inherit your code.

Since the amount of data in, for instance, a string variable can be arbitrarily large, **never accept a string as a pass by value parameter**. Always use pass by reference.

Objects as Function Parameters

As a programmer writing a function, you do not know all the ways the function might be used: by yourself, your programming team, or the next generation of programmers that inherit your code.

Since the amount of data in, for instance, a string variable can be arbitrarily large, **never accept a string as a pass by value parameter**. Always use pass by reference.

This rule extends to **all** objects, whether they are authored by you or someone else. The **the best practice** in C++ is to pass objects by reference only, **using the const keyword** to dictate whether it may be changed by the function or not.

You will be graded on this simple rule.

I/O Functions in C++

When writing a function that reads or writes to the data files, `cin`, or `cout`, pass the **stream parameter** as an `istream` or `ostream` **reference**.

Always **return** the stream reference from the function.

Then the function will work, magically, for both data files and `cin` or `cout`.

I/O Functions in C++

When writing a function that reads or writes to the data files, `cin`, or `cout`, pass the **stream parameter** as an `istream` or `ostream` **reference**.

Always **return** the stream reference from the function.

Then the function will work, magically, for both data files and `cin` or `cout`.

[Edit Program](#)

[Run Program](#)

Function Parameter Guidelines

	const Value	Value	Reference	const Reference
Fundamental Types	OK	YES	YES	OK
Arrays	N/A	N/A	YES	YES
Objects (instances)	NO	NO	YES	YES
Array Counts or Sizes	YES	NO	NO	YES

Passing array counts or sizes as const parameters is not enforced by C++, but it is a pragmatic programmer's approach.

Certainly these values should **not** be changed by the function, making them const allows the compiler to enforce this conceptual rule-of-thumb.

Overloaded Functions

Function Signatures

```
1 /**
2  * Main function comment describes what the function does, special properties
3  * of its arguments, what it returns, and its return interpretation.
4  */
5 return_type function_name( arg1_type arg1_name, arg2_type arg2_name )
6 {
7     variable_declarations;
8
9     function_statements;
10
11     return value;
12 }
```

function signature = function_name + (sequence of) argument const'ness + _types

- ▶ The argument names (`arg1_name`, `arg2_name`) are **NOT** part of the function signature.
- ▶ The `return_type` and its const'ness are **NOT** part of the function signature.
- ▶ The const'ness of call-by-value parameters is **NOT** part of the function signature.

Function Signatures

```
1 /**
2  * Main function comment describes what the function does, special properties
3  * of its arguments, what it returns, and its return interpretation.
4  */
5 return_type function_name( arg1_type arg1_name, arg2_type arg2_name )
6 {
7     variable_declarations;
8
9     function_statements;
10
11     return value;
12 }
```

function signature = function_name + (sequence of) argument const'ness + _types

- ▶ Internally, the C++ compiler and linker distinguishes between like-named functions that have different signatures.

A Simple Overloading Example

```
15 // prompt for a double , ignore cin failure
16 void prompt_for_input( const string& prompt ,
17     double& response )
18 {
19     cout << prompt << endl;
20     cin >> response ;
21     return ;
22 }
23
24 // prompt for a string , ignore cin failure
25 void prompt_for_input( const string& prompt ,
26     string& response )
27 {
28     cout << prompt << endl;
29     cin >> response ;
30     return ;
31 }
32
33 // prompt for a int , ignore cin failure
34 void prompt_for_input( const string& prompt ,
35     int& response )
36 {
37     cout << prompt << endl;
38     cin >> response ;
39     return ;
40 }
```

Overloaded Input Functions

- ▶ Define three overloaded functions for user input.
- ▶ Each takes a prompt argument, and a *reference* to a variable location where user input will be written.

A Simple Overloading Example

```
15 // prompt for a double, ignore cin failure
16 void prompt_for_input( const string& prompt ,
17                       double& response )
18 {
19     cout << prompt << endl;
20     cin >> response;
21     return;
22 }
23
24 // prompt for a string, ignore cin failure
25 void prompt_for_input( const string& prompt ,
26                       string& response )
27 {
28     cout << prompt << endl;
29     cin >> response;
30     return;
31 }
32
33 // prompt for a int, ignore cin failure
34 void prompt_for_input( const string& prompt ,
35                       int& response )
36 {
37     cout << prompt << endl;
38     cin >> response;
39     return;
40 }
```

Overloaded Input Functions

- ▶ The compiler distinguishes between these functions based the referenced variable type (which is the only distinction in their signatures).

A Simple Overloading Example

Using the Overloaded Functions

```
45  /**
46   * A railroad efficiency simulator
47   */
48
49  double engine_horse_power ;
50  double engine_cost_efficiency ;
51
52  int number_of_cars ;
53  double metric_tons_per_car ;
54
55  string startCity ;
56  string endCity ;
57
58  // input from user
59  prompt_for_input( "Enter_the_engine_horse_power:_", engine_horse_power );
60  prompt_for_input( "Enter_the_engine_efficiency:_", engine_cost_efficiency );
61  prompt_for_input( "Enter_the_number_of_cars:_", number_of_cars );
62  prompt_for_input( "Enter_the_mass_of_each_car:_", metric_tons_per_car );
63  prompt_for_input( "Enter_the_city_where_the_train_starts:_", startCity );
64  prompt_for_input( "Enter_the_destination_city:_", endCity );
```

Simply call the overloaded function with the correct prompt and variable.

Why Can't the Compiler Do This?

```
18 /**
19  * functions for user input
20  */
21 // prompt for a double, ignore cin failure
22 double prompt_for_input( const string& prompt )
23 {
24     double response;
25     cout << prompt << endl;
26     cin >> response;
27     return response;
28 }
29
30 // prompt for a string, ignore cin failure
31 string prompt_for_input( const string& prompt )
32 {
33     string response;
34     cout << prompt << endl;
35     cin >> response;
36     return response;
37 }
38
39 // prompt for a integer, ignore cin failure
40 int prompt_for_input( const string& prompt )
41 {
42     int response;
43     cout << prompt << endl;
44     cin >> response;
45     return response;
46 }
```

Why Can't the Compiler Do This?

```
22 double prompt_for_input( const string& prompt )
```

```
31 string prompt_for_input( const string& prompt )
```

```
40 int prompt_for_input( const string& prompt )
```

The signatures of these functions are the same!

```
In function 'string prompt_for_input(const string&)':  
31: error: new declaration 'string prompt_for_input(const string&'  
22: error: ambiguates old declaration 'double prompt_for_input(const string&'  
In function 'int prompt_for_input(const string&)':  
40: error: new declaration 'int prompt_for_input(const string&'  
22: error: ambiguates old declaration 'double prompt_for_input(const string&'
```

Try question 17.

“Random” Numbers

Pseudo-random Number Generators

```
void srand( unsigned int seed );  
           and  
           unsigned int rand();
```

Generates a sequence of pseudo-random integers from a seed.

`srand(seed)` should be used **only once**, as one of the first statements in the `main()` routine. (**Library and utility functions should never call `srand()`.**)

Requires `#include <cstdlib>`.

Seed from the wall clock by including `#include <time.h>` and
`srand(unsigned(time(0)));`

Pseudo-random Number Generators

```
void srand( unsigned int seed );  
           and  
           unsigned int rand();
```

Generates a sequence of pseudo-random integers from a seed.

`srand(seed)` should be used **only once**, as one of the first statements in the `main()` routine. (**Library and utility functions should never call `srand()`.**)

Requires `#include <cstdlib>`.

Seed from the wall clock by including `#include <time.h>` and
`srand(unsigned(time(0)));`

[Run Program](#)

[Edit Program](#)

finis