C++ is Fun – Part Three
at Turbine/Warner Bros.!

Russell Hanson

Let's go over homework!

- Hope you enjoyed the homework, you did, right? Right, guys?

- Cool.

Homework! :?

- Notes on system(“PAUSE”) and endl vs. “\n”

```
    cout << first <<'\n';
    // cout << first << endl;
    system("PAUSE");
    return 0;
}
```

```
$ ./PartDeux.exe
Enter two words:
Big Apple
Press any key to continue . . .

Big Ripe Apple
```

```
    cout << first <<'\n';
    // cout << first << endl;
    // system("PAUSE");
    return 0;
}
```

```
$ ./PartDeux.exe
Enter two words:
Big Apple
Big Ripe Apple
```

```
    // cout << first <<'\n';
    cout << first << endl;
    system("PAUSE");
    return 0;
}
```

```
$ ./PartDeux.exe
Enter two words:
Big Apple
Big Ripe Apple
Press any key to continue . . .
```

```
template <class Ch, class Tr>
      basic_ostream<Ch, Tr>& endl(basic_ostream<Ch, Tr>&);     // put '\n' and flush
template <class Ch, class Tr>
      basic_ostream<Ch, Tr>& ends(basic_ostream<Ch, Tr>&);     // put '\0' and flush
template <class Ch, class Tr>
      basic_ostream<Ch, Tr>& flush(basic_ostream<Ch, Tr>&);    // flush stream

template <class Ch, class Tr>
      basic_istream<Ch, Tr>& ws(basic_istream<Ch, Tr>&);       // eat whitespace
```

As you may recall, "*Homework for next Monday (pick 2, minimum)*"

1) Write a program that uses the modulus operator to determine if a number is odd or even. If the number is evenly divisible by 2, it is an even number. A remainder indicates it is odd. The number can be input by the user or read from a file.
2) Write an if statement that performs the following logic: if the variable *sales* is greater than 50,000, then assign 0.25 to the *commissionRate* variable, and assign 250 to the *bonus* variable.
3) Accept two strings as input at the prompt/command line, such as "Big" and "Apple." Join or concatenate the two words with a third word, such as "Ripe" and print the three words together with the third word the middle, "Big Ripe Apple".
4) Accept 5 integers on the command line, either all at once or separately. Save these to an array, vector, or list. Print the integers in the range 2 through 4, leaving off the first and the last. Bonus: Ask for the size of the array to be used, so it can be 5, 6, or 7 etc. Double Bonus: Allow a variable number of input numbers, stop input using a stop character or command the letter "s" say, then print all the input integers leaving off the first and the last.

1) Write a program that uses the modulus operator to determine if a number is odd or even. If the number is evenly divisible by 2, it is an even number. A remainder indicates it is odd. The number can be input by the user or read from a file.

```
string oddOrEven(int number)
{
            string answer = "";
            if(number % 2 == 0) {
                        answer = "even";
            }
            else        {
                        answer = "odd";
            }
            return answer;
}
void questionOne()
{
            int number = 0;
            cout << "Please enter your number. I will determine if it is odd or even." << endl;
            cin >> number;
            string answer = oddOrEven(number);
            cout << "The number you entered is: " << answer << endl;
}

// int main

if(questionToAnswer == 1)
                        {
                                    questionOne();
                        }
```

2) Write an if statement that performs the following logic: if the variable *sales* is greater than 50,000, then assign 0.25 to the *commissionRate* variable, and assign 250 to the *bonus* variable.

```
void questionTwo()
{
          int sales = 0;
          double commissionRate = 0;
          int bonus = 0;
          cout << "Please enter the sales amount for the year:" << endl;
          cin >> sales;
          if(sales > 50000)
          {
                    commissionRate = 0.25;
                    bonus = 250;
          }

          cout << "Your sales commission rate is: " << commissionRate << endl;
          cout << "Your bonus is: " << bonus << endl;
}
```

3) Accept two strings as input at the prompt/command line, such as “Big” and “Apple.” Join or concatenate the two words with a third word, such as “Ripe” and print the three words together with the third word the middle, “Big Ripe Apple”.

```
string concatenate(string first, string second, string middle)
{
        stringstream ss;
        ss << first << " " << middle << " " << second;
        string s = ss.str();
        return s;
}

void questionThree()
{
        string firstWord;
        string secondWord;
        string middleWord;
        cout << "Please enter a word:" << endl;
        cin >> firstWord;
        cout << "Please enter a second word:" << endl;
        cin >> secondWord;
        cout << "Please enter a word to put in the middle:" << endl;
        cin >> middleWord;
        string phrase = concatenate(firstWord, secondWord, middleWord);
        cout << "Your final sentence is: " << phrase << endl;
}
```

Functions with variable length arguments

<u>Example 1 :</u> A function accepts variable arguments of known data-type
(A simple average function, that takes variable number of arguments)

Code: C++

```cpp
#include <stdio.h>
#include <stdarg.h>

float avg( int Count, ... )
{
	va_list Numbers;
	va_start(Numbers, Count);
	int Sum = 0;
	for(int i = 0; i < Count; ++i )
		Sum += va_arg(Numbers, int);
	va_end(Numbers);
	return (Sum/Count);
}

int main()
{
	float Average = avg(10, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9);
	printf("Average of first 10 whole numbers : %f\n", Average);
	return 0;
}
```

Output of the above code is :

```
Average of first 10 whole numbers : 4.000000
```

Here is another example of variable argument function, which is a simple printing function:

```
void my_printf( char *format, ... ) {
  va_list argptr;

  va_start( argptr, format );

  while( *format != '\0' ) {
    // string
    if( *format == 's' ) {
      char* s = va_arg( argptr, char * );
      printf( "Printing a string: %s\n", s );
    }
    // character
    else if( *format == 'c' ) {
      char c = (char) va_arg( argptr, int );
      printf( "Printing a character: %c\n", c );
      break;
    }
    // integer
    else if( *format == 'd' ) {
      int d = va_arg( argptr, int );
      printf( "Printing an integer: %d\n", d );
    }

    *format++;
  }

  va_end( argptr );
}


int main( void ) {

  my_printf( "sdc", "This is a string", 29, 'X' );

  return( 0 );
}
```

This code displays the following output when run:

```
Printing a string: This is a string
Printing an integer: 29
Printing a character: X
```

4) Accept 5 integers on the command line, either all at once or separately. Save these to an array, vector, or list. Print the integers in the range 2 through 4, leaving off the first and the last. Bonus: Ask for the size of the array to be used, so it can be 5, 6, or 7 etc. Double Bonus: Allow a variable number of input numbers, stop input using a stop character or command the letter "s" say, then print all the input integers leaving off the first and the last.

```
void questionFour()
{
            int dataStructChosen;
            cout << "Welcome to store and retrieve numbers. Would you like to store numbers in 1) an array, 2) a vecto
            cin >> dataStructChosen;
            if(dataStructChosen == 1)
            {
                        int size = 5;
                        do
                        {
                                    cout << "What size array would you like to use? (select a size of 5 or greater ple
                                    cin >> size;
                        } while(size < 5);
                        int *myArray = new int[size];
                        for(int i = 0; i <= size - 1; i++)
                        {
                                    int number;
                                    cout << "Please enter a number " << ":" << endl;
                                    cin >> number;
                                    myArray [i] = number;
                        }
                        cout << "Here are the values from position 2 to " << size - 1 << ": " << endl;
                        for(int j = 0; j < size - 2; j++)
                        {
                                    cout << "The array value at position " << j+2 << " is: " << myArray[j+1] << endl;
                        }
            }
```

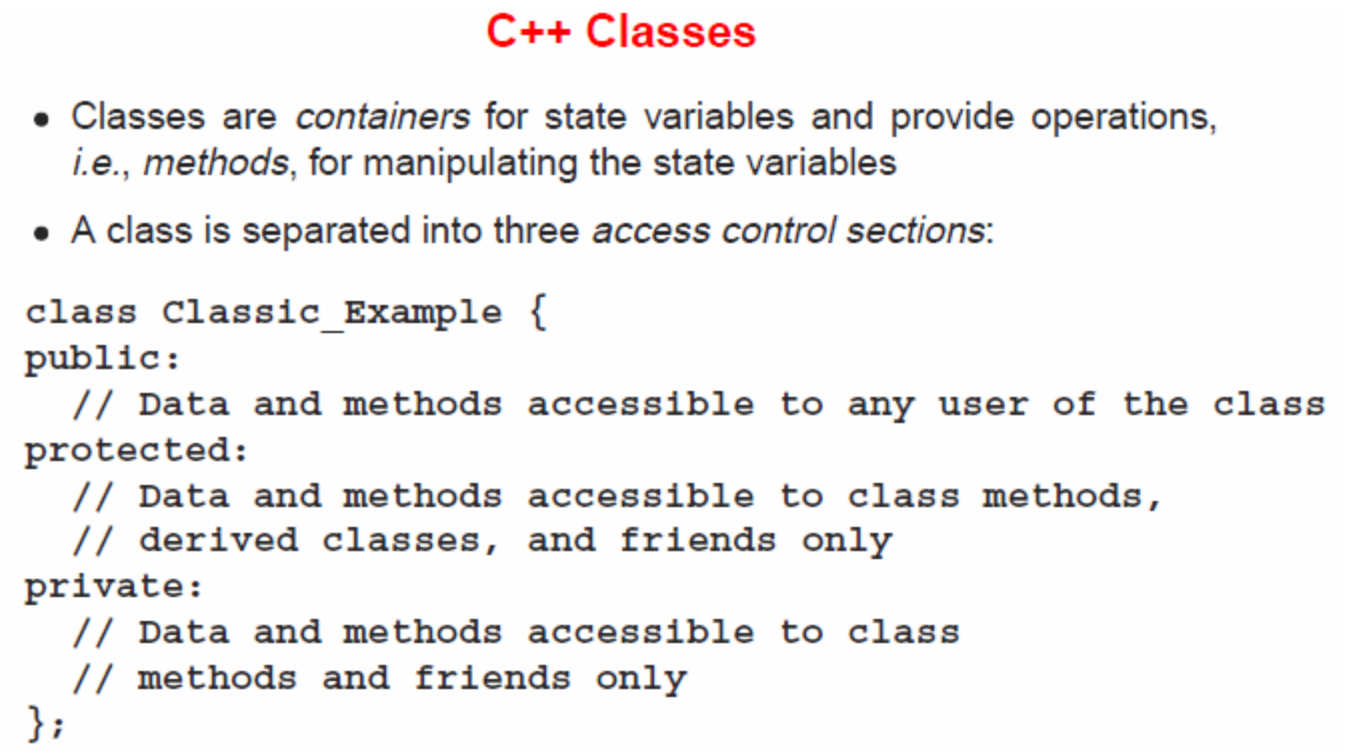
```
else if (dataStructChosen == 2)
{
           int size = 5;
           do
           {
cout << "What size vector would you like to use? (select a size of 5 or greater please)" << endl;
                      cin >> size;
           } while(size < 5);
           vector<int> myVector;
           for(int i = 0; i <= size - 1; i++)
           {
                      int number;
                      cout << "Please enter a number:" << endl;
                      cin >> number;
                      myVector.push_back(number);
           }
           cout << "Here are the values from position 2 to " << size - 1 << ":"  << endl;
           for(int j = 0; j < size - 2; j++)
           {
                      cout << "The vector value at position " << j + 2 << " is: " << myVector[j+1] << endl;
           }
}
```

```
else if (dataStructChosen == 3){
       int size = 5;
       do
       {
                  cout << "What size list would you like to use? (select a size of 5 or greater please)" << e
                  cin >> size;
       } while(size < 5);
       list<int> myList;
       for(int i = 0; i <= size - 1; i++)
       {
                  int number;
                  cout << "Please enter a number:" << endl;
                  cin >> number;
                  myList.push_back(number);
       }
       cout << "Here are the values from position 2 to " << size - 1 << ":"  << endl;
       for(list<int>::iterator it=myList.begin(); it != myList.end(); it++)
       {
                  if(it == myList.begin())
                  {
                             continue;
                  }
                  if(next(it) == myList.end())
                  {
                             continue;
                  }
                  else
                  {
                             cout << "The list value at position is: " << (*it) << endl;
                  }
       }
  }
```

The ‘+’ operator can concatenate strings, not strings and ints

```
int main() {
std::string a = "Hello ";
std::string b = "World";
std::string c = a + b;
// std::string c = a + b + 3 + "4" + "hello"; // Doesn't work
//       23       IntelliSense: no operator "+" matches these operands
//            operand types are: std::basic_string<char, std::char_traits<char>,
//            std::allocator<char>> + int

cout << c << endl;
int myInt = 3;
std::stringstream ss;
ss << a << b << myInt << 4 << "hello";
string newstring = ss.str();
cout << newstring << endl;
}
```

- 2) Objects, encapsulation, abstract data types, data protection and scope



C++ Classes

- Classes are *containers* for state variables and provide operations, *i.e.*, *methods*, for manipulating the state variables
- A class is separated into three *access control sections*:

```
class Classic_Example {
public:
  // Data and methods accessible to any user of the class
protected:
  // Data and methods accessible to class methods,
  // derived classes, and friends only
private:
  // Data and methods accessible to class
  // methods and friends only
};
```

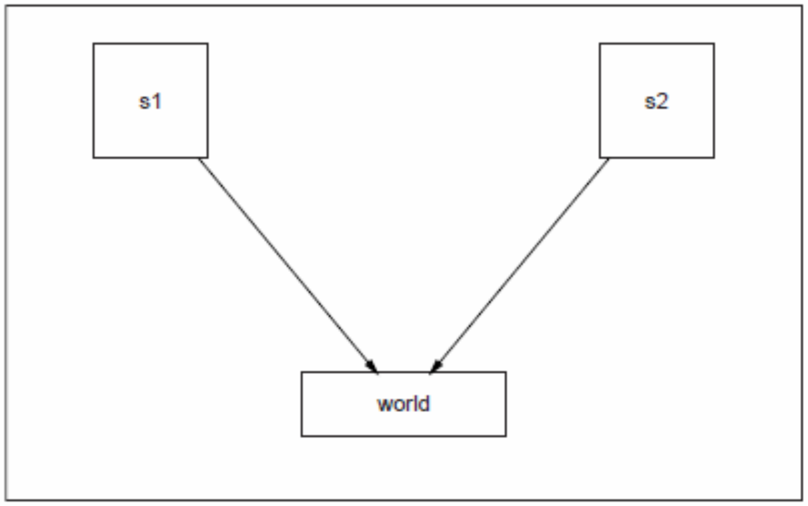
Assignment and Initialization (cont'd)

```
class String {
public:
  String (const char *t)
    : len_ (t == 0 ? 0 : strlen (t)) {
    if (this->len_ == 0)
      throw range_error ();
    this->str_ = strcpy (new char [len_ + 1], t);
  }
  ~String (void) { delete [] this->str_; }
// . . .
private:
  size_t len_;
  char *str_;
};
```

Assignment and Initialization (cont'd)

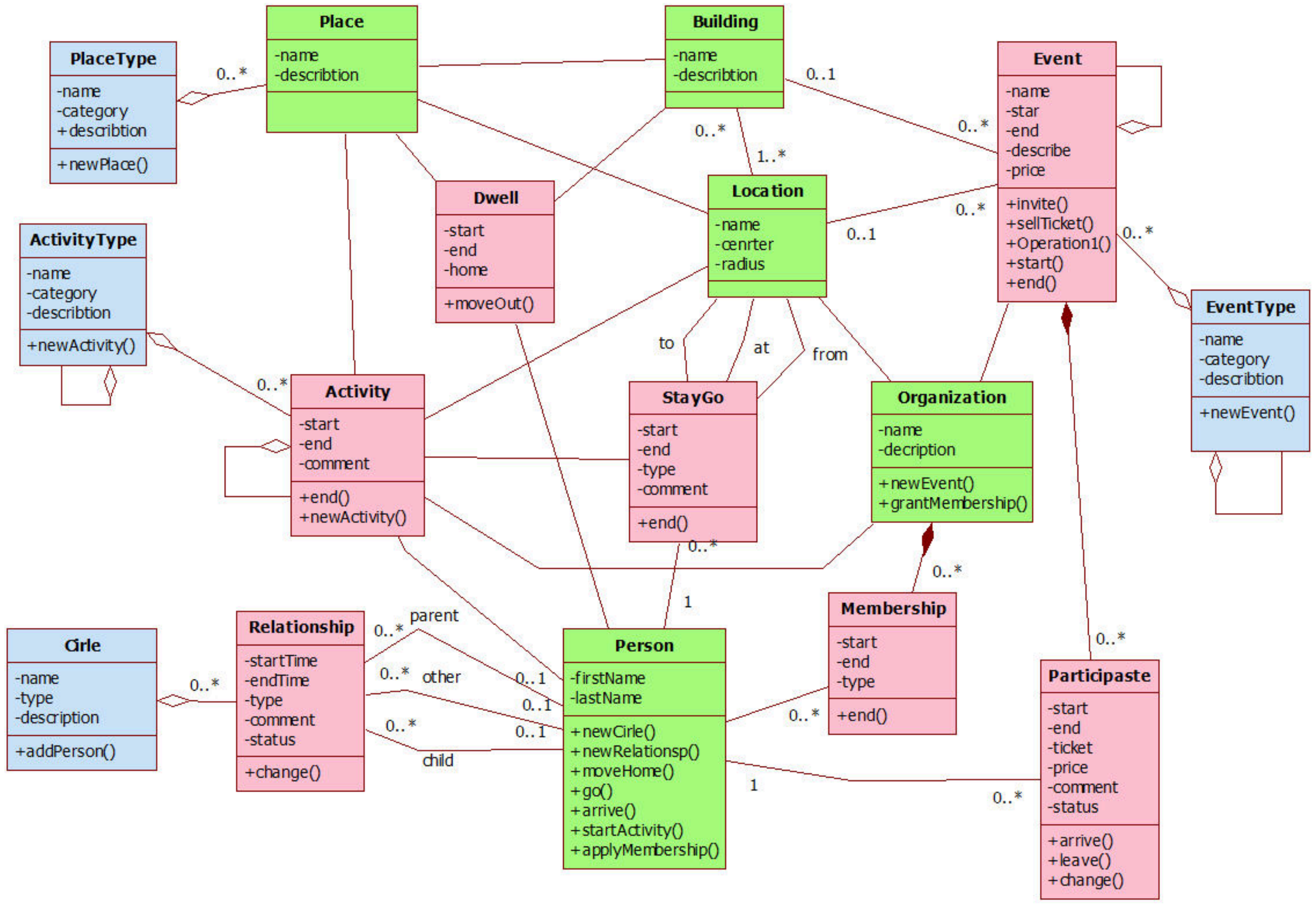
```
void foo (void) {
  String s1 ("hello");
  String s2 ("world");

  s1 = s2; // leads to aliasing
  s1[2] = 'x';
  assert (s2[2] == 'x'); // will be true!
  // . . .
  // double deletion in destructor calls!
}
```

Assignment and Initialization (cont'd)



- Note that both `s1.s` and `s2.s` point to the dynamically allocated buffer storing world (this is known as *aliasing*)

“Objects” and “Classes” ??



Local and Global Variables

CONCEPT: A local variable is defined inside a function and is not accessible outside the function. A global variable is defined outside all functions and is accessible to all functions in its scope.

Local Variables

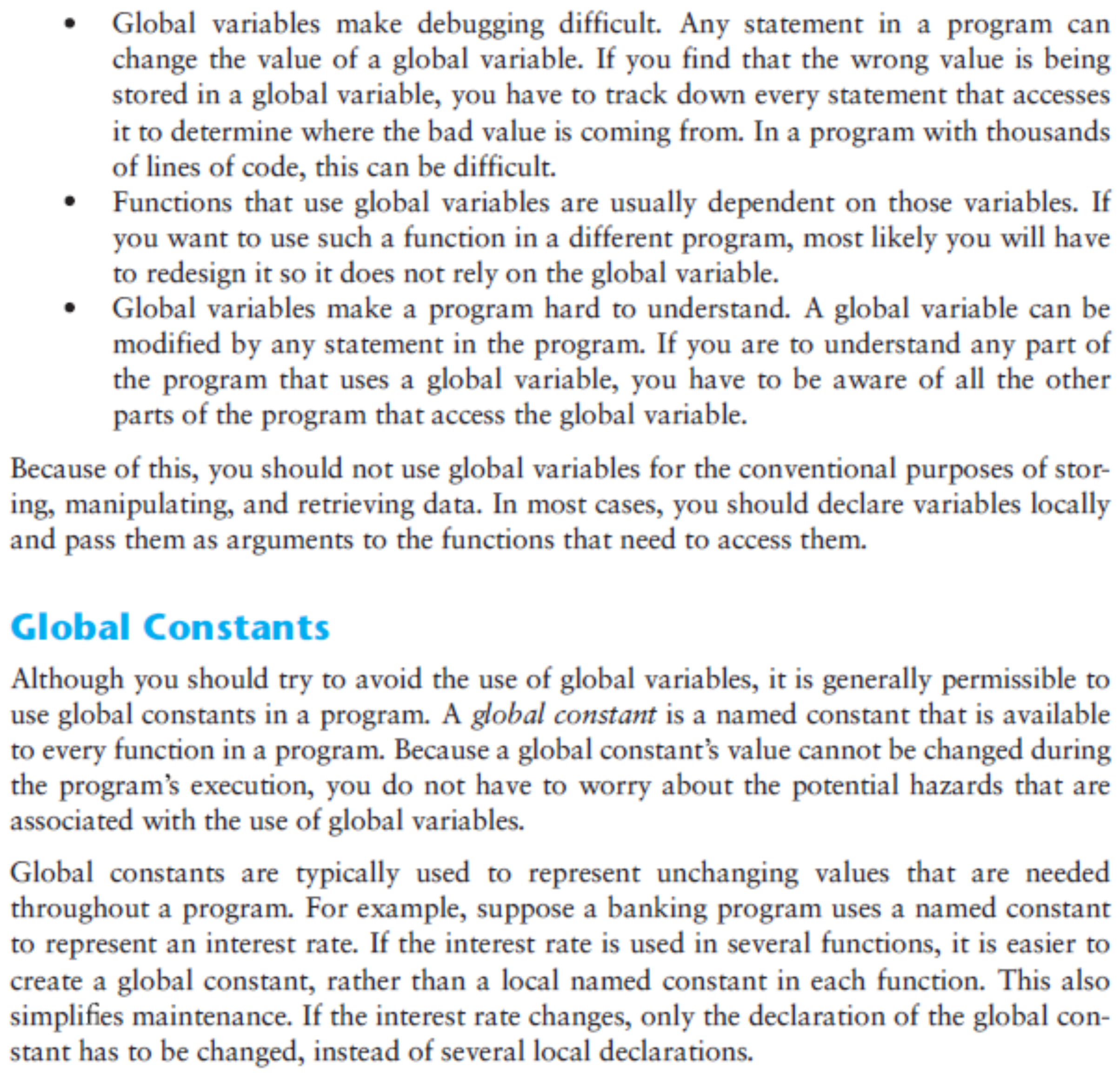
Variables defined inside a function are *local* to that function. They are hidden from the statements in other functions, which normally cannot access them. Program 6-16 shows that because the variables defined in a function are hidden, other functions may have sepa-

```
int main()
{
   int num = 1;   // Local variable

   cout << "In main, num is " << num << endl;
   anotherFunction();
   cout << "Back in main, num is " << num << endl;
   return 0;
}

//*******************************************************
// Definition of anotherFunction                        *
// It has a local variable, num, whose initial value    *
// is displayed.                                        *
//*******************************************************

void anotherFunction()
{
   int num = 20;  // Local variable

   cout << "In anotherFunction, num is " << num << endl;
}
```

Program Output

```
In main, num is 1
In anotherFunction, num is 20
Back in main, num is 1
```

Global Variables vs. Global Constants



- Global variables make debugging difficult. Any statement in a program can change the value of a global variable. If you find that the wrong value is being stored in a global variable, you have to track down every statement that accesses it to determine where the bad value is coming from. In a program with thousands of lines of code, this can be difficult.
- Functions that use global variables are usually dependent on those variables. If you want to use such a function in a different program, most likely you will have to redesign it so it does not rely on the global variable.
- Global variables make a program hard to understand. A global variable can be modified by any statement in the program. If you are to understand any part of the program that uses a global variable, you have to be aware of all the other parts of the program that access the global variable.

Because of this, you should not use global variables for the conventional purposes of storing, manipulating, and retrieving data. In most cases, you should declare variables locally and pass them as arguments to the functions that need to access them.

Global Constants

Although you should try to avoid the use of global variables, it is generally permissible to use global constants in a program. A *global constant* is a named constant that is available to every function in a program. Because a global constant's value cannot be changed during the program's execution, you do not have to worry about the potential hazards that are associated with the use of global variables.

Global constants are typically used to represent unchanging values that are needed throughout a program. For example, suppose a banking program uses a named constant to represent an interest rate. If the interest rate is used in several functions, it is easier to create a global constant, rather than a local named constant in each function. This also simplifies maintenance. If the interest rate changes, only the declaration of the global constant has to be changed, instead of several local declarations.

6.3 Function Prototypes

CONCEPT: A function prototype eliminates the need to place a function definition before all calls to the function.

Before the compiler encounters a call to a particular function, it must already know the function's return type, the number of parameters it uses, and the type of each parameter. (You will learn how to use parameters in the next section.)

One way of ensuring that the compiler has this information is to place the function definition before all calls to that function. This was the approach taken in Programs 6-1, 6-2, 6-3, and 6-4. Another method is to declare the function with a *function prototype*. Here is a prototype for the `displayMessage` function in Program 6-1:

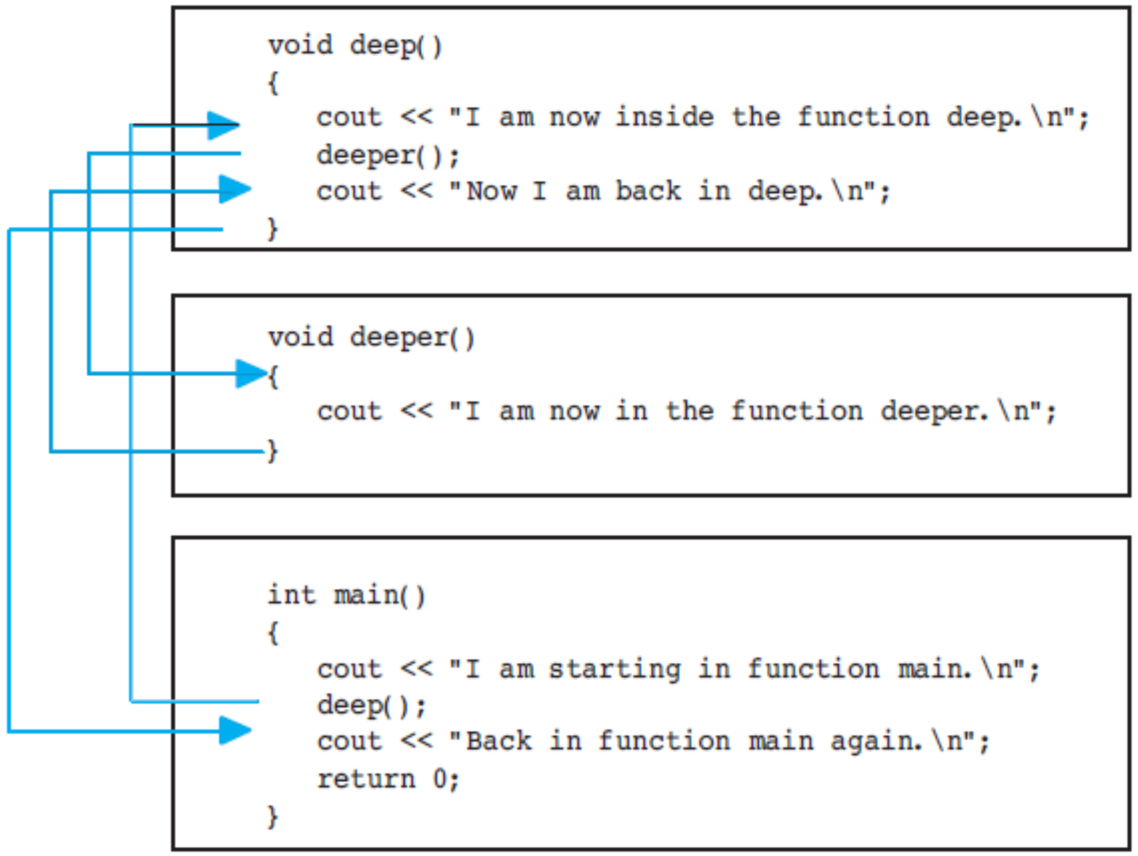
```
void displayMessage();
```

The prototype looks similar to the function header, except there is a semicolon at the end. The statement above tells the compiler that the function `displayMessage` has a `void` return type (it doesn't return a value) and uses no parameters.

NOTE: Function prototypes are also known as *function declarations.*

WARNING! You must place either the function definition or either/the function prototype ahead of all calls to the function. Otherwise the program will not compile.

Function prototypes are usually placed near the top of a program so the compiler will encounter them before any function calls. Program 6-5 is a modification of Program 6-3. The definitions of the functions `first` and `second` have been placed after `main`, and a function prototype has been placed after the `using namespace std` statement.



```
void deep()
{
   cout << "I am now inside the function deep.\n";
   deeper();
   cout << "Now I am back in deep.\n";
}
```

```
void deeper()
{
   cout << "I am now in the function deeper.\n";
}
```

```
int main()
{
   cout << "I am starting in function main.\n";
   deep();
   cout << "Back in function main again.\n";
   return 0;
}
```

```cpp
// This program has three functions: main, first, and second.
#include <iostream>
using namespace std;

// Function Prototypes
void first();
void second();

int main()
{
   cout << "I am starting in function main.\n";
   first();    // Call function first
   second();   // Call function second
   cout << "Back in function main again.\n";
   return 0;
}

//*************************************
// Definition of function first.      *
// This function displays a message.  *
//*************************************

void first()
{
   cout << "I am now inside the function first.\n";
}

//*************************************
// Definition of function second.     *
// This function displays a message.  *
//*************************************

void second()
{
   cout << "I am now inside the function second.\n";
}
```

WARNING! When passing a variable as an argument, simply write the variable name inside the parentheses of the function call. Do not write the data type of the argument variable in the function call. For example, the following function call will cause an error:

```
displayValue( int x); // Error!
```

The function call should appear as

```
displayValue( x);     // Correct
```

Default Arguments

CONCEPT: Default arguments are passed to parameters automatically if no argument is provided in the function call.

It's possible to assign *default arguments* to function parameters. A default argument is passed to the parameter when the actual argument is left out of the function call. The default arguments are usually listed in the function prototype. Here is an example:

```
void showArea(double = 20.0, double = 10.0);
```

Default arguments are literal values or constants with an = operator in front of them, appearing after the data types listed in a function prototype. Since parameter names are optional in function prototypes, the example prototype could also be declared as

```
void showArea(double length = 20.0, double width = 10.0);
```

In both example prototypes, the function `showArea` has two `double` parameters. The first is assigned the default argument 20.0 and the second is assigned the default argument 10.0. Here is the definition of the function:

```
void showArea(double length, double width)
{
   double area = length * width;
   cout << "The area is " << area << endl;
}
```

The default argument for `length` is 20.0 and the default argument for `width` is 10.0. Because both parameters have default arguments, they may optionally be omitted in the function call, as shown here:

```
showArea();
```

When a function uses a mixture of parameters with and without default arguments, the parameters with default arguments must be defined last. In the `calcPay` function, `hours` could not have been defined before either of the other parameters. The following prototypes are illegal:

```
// Illegal prototype
void calcPay(int empNum, double hours = 40.0, double payRate);

// Illegal prototype
void calcPay(double hours = 40.0, int empNum, double payRate);
```

Here is a summary of the important points about default arguments:

- The value of a default argument must be a literal value or a named constant.
- When an argument is left out of a function call (because it has a default value), all the arguments that come after it must be left out too.
- When a function has a mixture of parameters both with and without default arguments, the parameters with default arguments must be declared last.

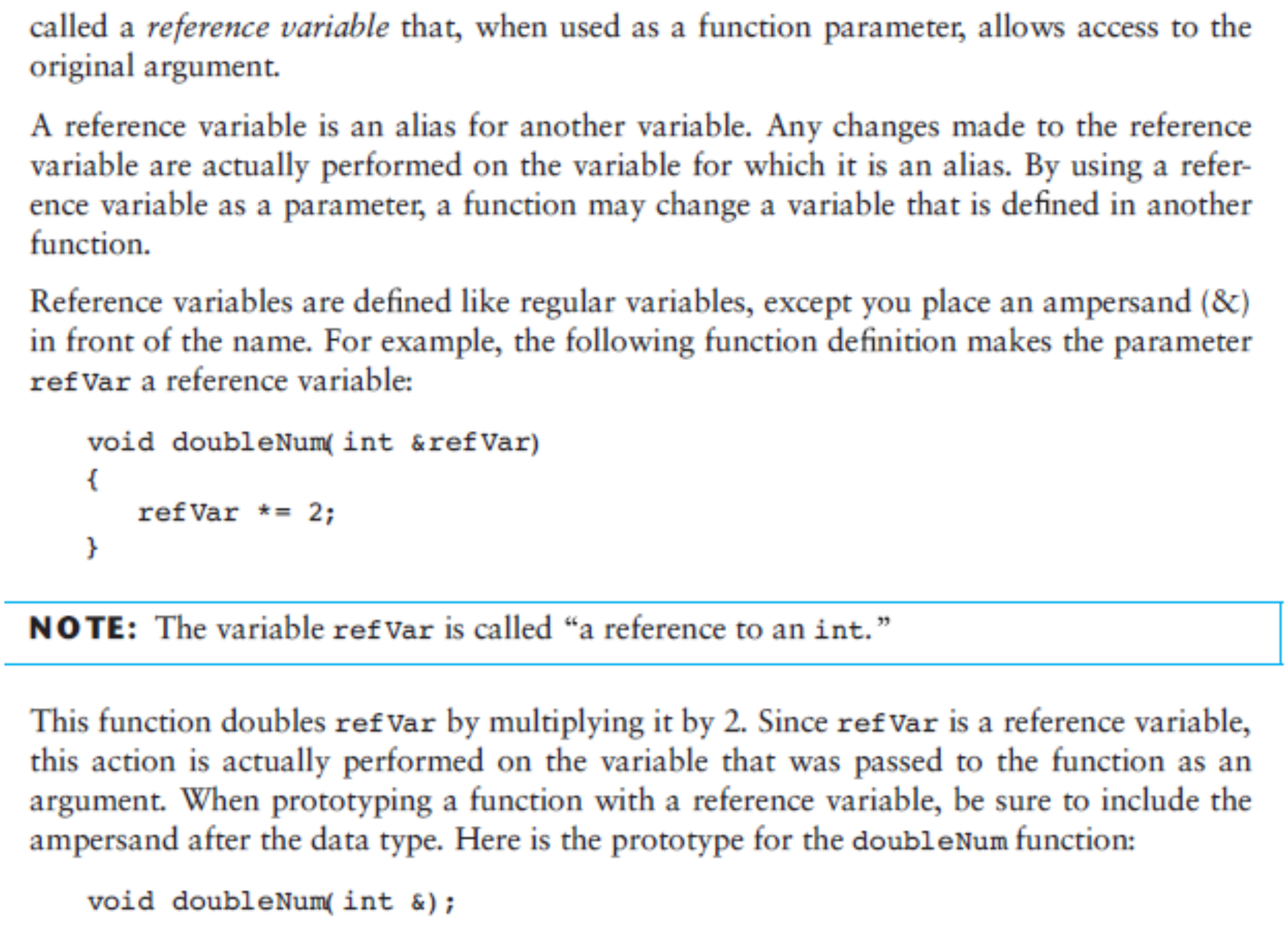
Program 6-9

```cpp
// This program demonstrates that changes to a function parameter
// have no effect on the original argument.
#include <iostream>
using namespace std;

// Function Prototype
void changeMe(int);

int main()
{
   int number = 12;

   // Display the value in number.
   cout << "number is " << number << endl;

   // Call changeMe, passing the value in number
   // as an argument.
   changeMe(number);

   // Display the value in number again.
   cout << "Now back in main again, the value of ";
   cout << "number is " << number << endl;
   return 0;
}

//**************************************************************
// Definition of function changeMe.                            *
// This function changes the value of the parameter myValue.   *
//**************************************************************

void changeMe(int myValue)
{
   // Change the value of myValue to 0.
   myValue = 0;

   // Display the value in myValue.
   cout << "Now the value is " << myValue << endl;
}
```

Program Output

```
number is 12
Now the value is 0
Now back in main again, the value of number is 12
```

Using reference variable as parameters



called a *reference variable* that, when used as a function parameter, allows access to the original argument.

A reference variable is an alias for another variable. Any changes made to the reference variable are actually performed on the variable for which it is an alias. By using a reference variable as a parameter, a function may change a variable that is defined in another function.

Reference variables are defined like regular variables, except you place an ampersand (&) in front of the name. For example, the following function definition makes the parameter `refVar` a reference variable:

```
void doubleNum(int &refVar)
{
   refVar *= 2;
}
```

NOTE: The variable `refVar` is called "a reference to an `int`."

This function doubles `refVar` by multiplying it by 2. Since `refVar` is a reference variable, this action is actually performed on the variable that was passed to the function as an argument. When prototyping a function with a reference variable, be sure to include the ampersand after the data type. Here is the prototype for the `doubleNum` function:

```
void doubleNum(int &);
```

Program 6-25

```
// This program uses a reference variable as a function
// parameter.
#include <iostream>
using namespace std;

// Function prototype. The parameter is a reference variable.
void doubleNum(int &);

int main()
{
   int value = 4;

   cout << "In main, value is " << value << endl;
   cout << "Now calling doubleNum..." << endl;
   doubleNum(value);
   cout << "Now back in main. value is " << value << endl;
   return 0;
}

//**********************************************************
// Definition of doubleNum.                                *
// The parameter refVar is a reference variable. The value *
// in refVar is doubled.                                   *
//**********************************************************

void doubleNum (int &refVar)
{
   refVar *= 2;
}
```

Program Output

```
In main, value is 4
Now calling doubleNum...
Now back in main. value is 8
```

The parameter `refVar` in Program 6-25 "points" to the `value` variable in function `main`. When a program works with a reference variable, it is actually working with the variable it references, or points to. This is illustrated in Figure 6-15.

Static Member Functions

You declare a static member function by placing the `static` keyword in the function's prototype. Here is the general form:

```
static ReturnType FunctionName (ParameterTypeList);
```

A function that is a static member of a class cannot access any nonstatic member data in its class. With this limitation in mind, you might wonder what purpose static member functions serve. The following two points are important for understanding their usefulness:

- Even though static member variables are declared in a class, they are actually defined outside the class declaration. The lifetime of a class's static member variable is the lifetime of the program. This means that a class's static member variables come into existence before any instances of the class are created.
- A class's static member functions can be called before any instances of the class are created. This means that a class's static member functions can access the class's static member variables *before* any instances of the class are defined in memory. This gives you the ability to create very specialized setup routines for class objects.

Contents of Budget. h (Version 2)

```
#ifndef BUDGET_H
#define BUDGET_H

// Budget class declaration
class Budget
{
private:
   static double corpBudget;  // Static member variable
   double divisionBudget;     // Instance member variable

public:
   Budget()
      { divisionBudget = 0; }

   void addBudget(double b)
      { divisionBudget += b;
        corpBudget += b; }

   double getDivisionBudget() const
      { return divisionBudget; }

   double getCorpBudget() const
      { return corpBudget; }

   static void mainOffice(double);  // Static member function
};

#endif
```

Contents of Budget.cpp

```
#include "Budget.h"

// Definition of corpBudget static member variable
double Budget::corpBudget = 0;

//**********************************************************
// Definition of static member function mainOffice.        *
// This function adds the main office's budget request to  *
// the corpBudget variable.                                *
//**********************************************************

void Budget::mainOffice(double moffice)
{
   corpBudget += moffice;
}
```

```
// This program demonstrates a static member function.
#include <iostream>
#include <iomanip>
#include "Budget.h"
using namespace std;

int main()
{
   int count;                     // Loop counter
   double mainOfficeRequest;      // Main office budget request
   const int NUM_DIVISIONS = 4;   // Number of divisions

   // Get the main office's budget request.
   // Note that no instances of the Budget class have been defined.
   cout << "Enter the main office's budget request: ";
   cin >> mainOfficeRequest;
   Budget::mainOffice(mainOfficeRequest);

   Budget divisions[NUM_DIVISIONS]; // An array of Budget objects.

   // Get the budget requests for each division.
   for (count = 0; count < NUM_DIVISIONS; count++)
   {
      double budgetAmount;
      cout << "Enter the budget request for division ";
      cout << (count + 1) << ": ";
      cin >> budgetAmount;
      divisions[count].addBudget(budgetAmount);
   }

   // Display the budget requests and the corporate budget.
   cout << fixed << showpoint << setprecision(2);
   cout << "\nHere are the division budget requests:\n";
   for (count = 0; count < NUM_DIVISIONS; count++)
   {
      cout << "\tDivision " << (count + 1) << "\t$ ";
      cout << divisions[count].getDivisionBudget() << endl;
   }
   cout << "\tTotal Budget Requests:\t$ ";
   cout << divisions[0].getCorpBudget() << endl;

   return 0;
```

Program Output with Example Input Shown in Bold
```
Enter the main office's budget request: 100000 [Enter]
Enter the budget request for division 1: 100000 [Enter]
Enter the budget request for division 2: 200000 [Enter]
Enter the budget request for division 3: 300000 [Enter]
Enter the budget request for division 4: 400000 [Enter]

Here are the division budget requests:
      Division 1    $ 100000.00
      Division 2    $ 200000.00
      Division 3    $ 300000.00
      Division 4    $ 400000.00
      Total Requests (including main office): $ 1100000.00
```

Notice in line 17 the statement that calls the static function `mainOffice`:

```
Budget::mainOffice(amount);
```

Calls to static member functions do not use the regular notation of connecting the function name to an object name with the dot operator. Instead, static member functions are called by connecting the function name to the class name with the scope resolution operator.

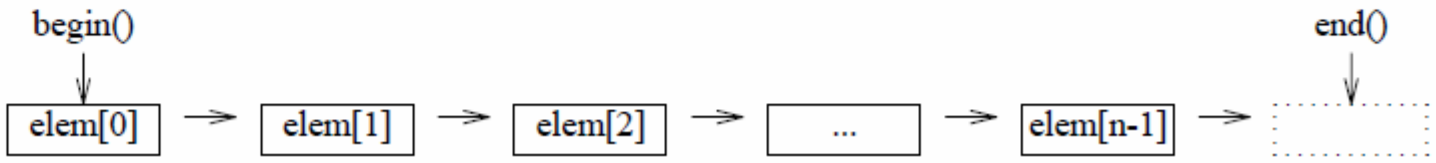
19.2 Iterators and Sequences [iter.iter]

An iterator is a pure abstraction. That is, anything that behaves like an iterator is an iterator (§3.8.2). An iterator is an abstraction of the notion of a pointer to an element of a sequence. Its key concepts are

- ''the element currently pointed to'' (dereferencing, represented by operators `*` and `->`),
- ''point to next element'' (increment, represented by operator `++`), and
- equality (represented by operator `==`).

For example, the built-in type ***int****** is an iterator for an ***int***`[]` and the class ***list<int>::iterator*** is an iterator for a ***list*** class.

A sequence is an abstraction of the notion ''something where we can get from the beginning to the end by using a next-element operation:''



Examples of such sequences are arrays (§5.2), vectors (§16.3), singly-linked lists (§17.8[17]), doubly-linked lists (§17.2.2), trees (§17.4.1), input (§21.3.1), and output (§21.2.1). Each has its own appropriate kind of iterator.

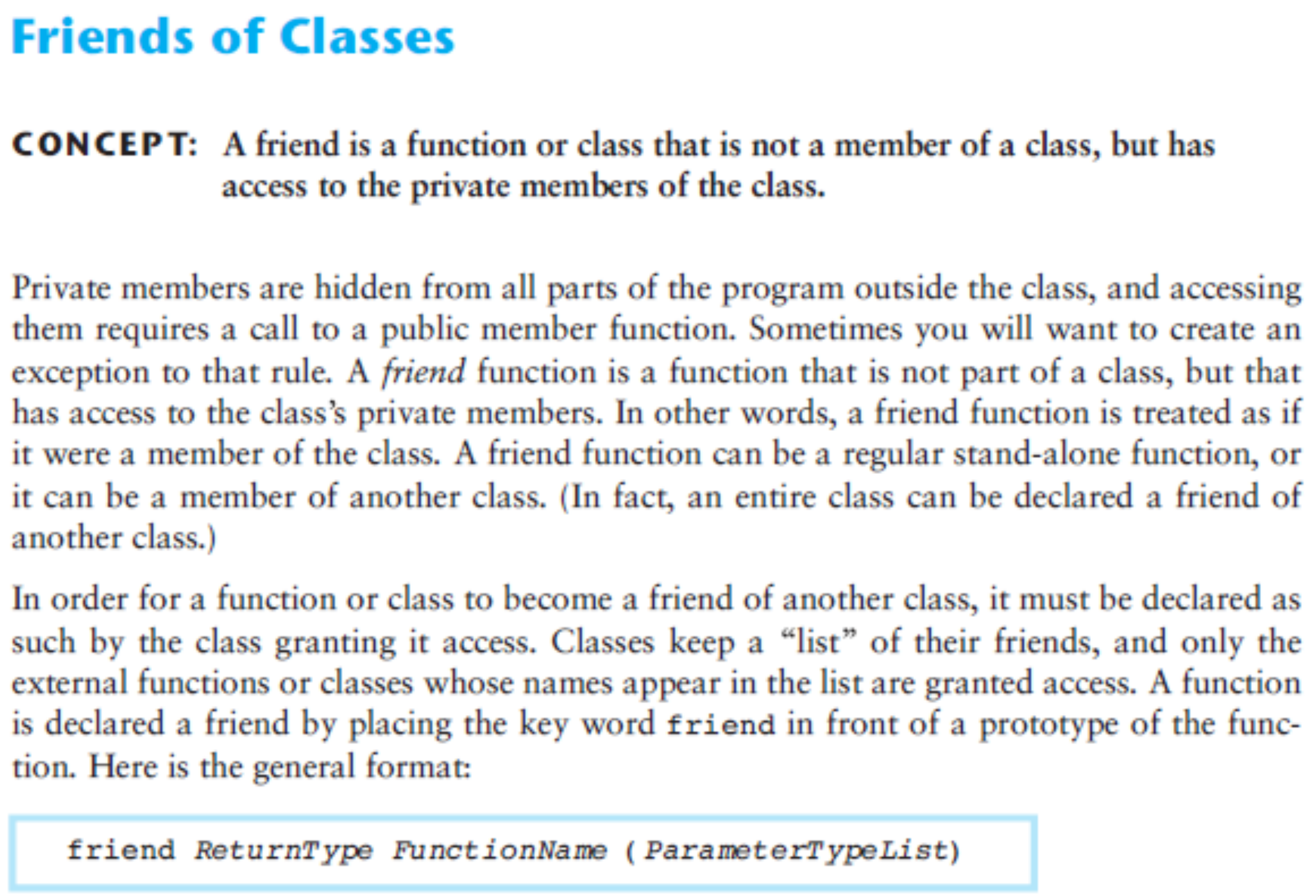
The iterator classes and functions are declared in namespace ***std*** and found in <***iterator***>.

An iterator is *not* a general pointer. Rather, it is an abstraction of the notion of a pointer into an array. There is no concept of a ''null iterator.'' The test to determine whether an iterator points to an element or not is conventionally done by comparing it against the *end* of its sequence (rather than comparing it against a ***null*** element). This notion simplifies many algorithms by removing the need for a special end case and generalizes nicely to sequences of arbitrary types.

An iterator that points to an element is said to be *valid* and can be dereferenced (using `*`, `[]`, or `->` appropriately). An iterator can be invalid either because it hasn't been initialized, because it pointed into a container that was explicitly or implicitly resized (§16.3.6, §16.3.8), because the container into which it pointed was destroyed, or because it denotes the end of a sequence (§18.2). The end of a sequence can be thought of as an iterator pointing to a hypothetical element position one-past-the-last element of a sequence.

In Java, we refer to the *fields* and *methods* of a class. In C++, we use the terms *data members* and *member functions*. Furthermore, in Java, every method must be inside some class. In contrast, a C++ program can also include **free functions** - functions that are not inside any class.

main () is a free function in C++



Friends of Classes

CONCEPT: A friend is a function or class that is not a member of a class, but has access to the private members of the class.

Private members are hidden from all parts of the program outside the class, and accessing them requires a call to a public member function. Sometimes you will want to create an exception to that rule. A *friend* function is a function that is not part of a class, but that has access to the class's private members. In other words, a friend function is treated as if it were a member of the class. A friend function can be a regular stand-alone function, or it can be a member of another class. (In fact, an entire class can be declared a friend of another class.)

In order for a function or class to become a friend of another class, it must be declared as such by the class granting it access. Classes keep a "list" of their friends, and only the external functions or classes whose names appear in the list are granted access. A function is declared a friend by placing the key word `friend` in front of a prototype of the function. Here is the general format:

```
friend ReturnType FunctionName (ParameterTypeList)
```

Friends

- A class may grant access to its private data and methods by including *friend* declarations in the class definition, *e.g.*,

```
class Vector {
  friend Vector &product (const Vector &,
                          const Matrix &);
private:
  int size_;
  // . . .
};
```

- Function `product` can access `Vector`'s private parts:

```
  Vector &product (const Vector &v, const Matrix &m) {
      int vector_size = v.size_;
      // . . .
```

Static Methods

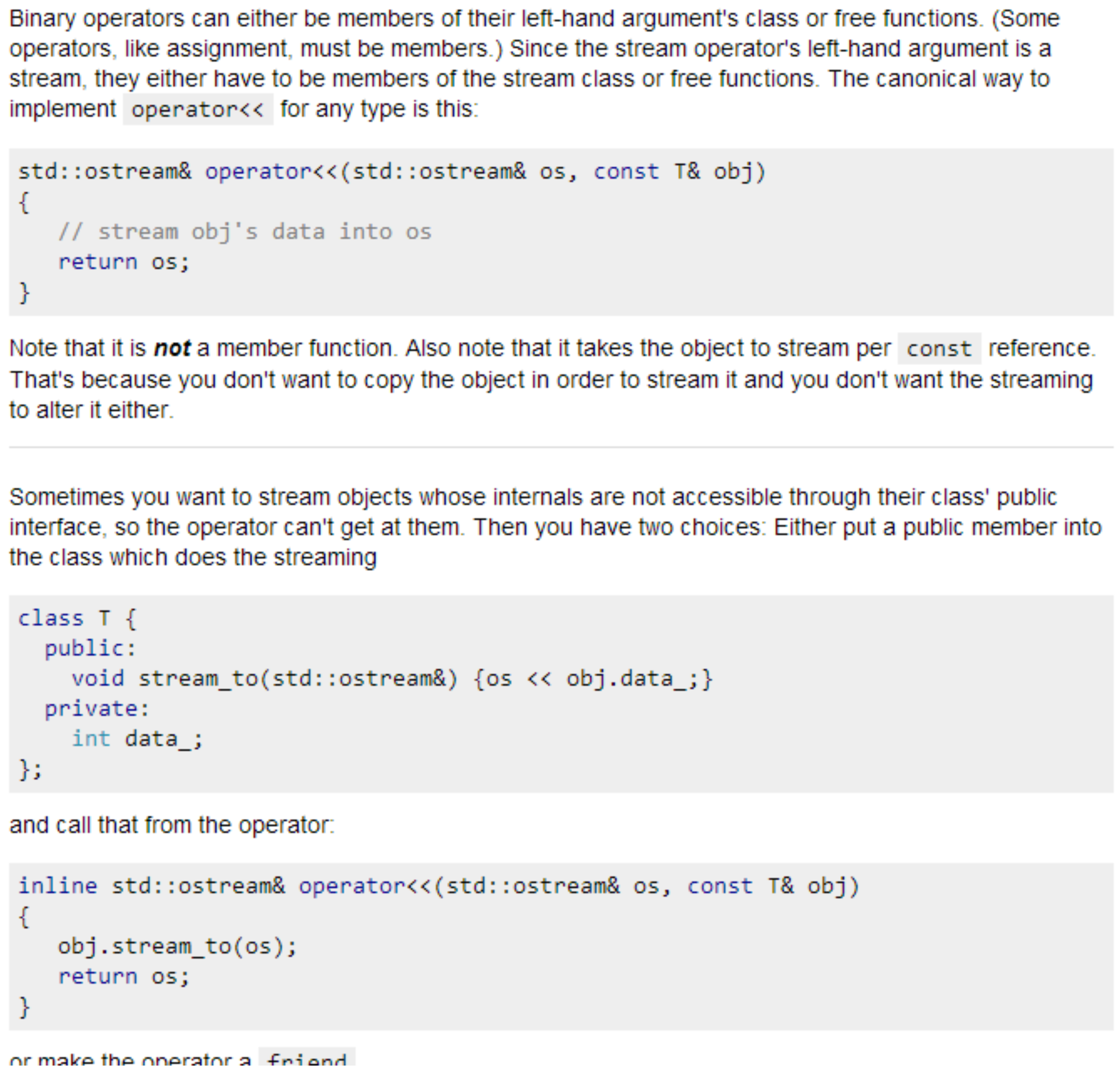
- A static method may be called on an object of a class, or on the class itself *without supplying an object* (unlike non-static methods . . .)
- Note, there is no `this` pointer in a static method

Static Methods (cont'd)

- *i.e.*, a static method cannot access non-static class data and functions

```
class Foo {
public:
  static int get_s1 (void) {
    this->a_ = 10; /* ERROR! */; return Foo::s_;
  }
  int get_s2 (void) {
    this->a_ = 10; /* OK */; return Foo::s_;
  }
private:
  int a_;
  static int s_;
};
```

Operator Overloading



Binary operators can either be members of their left-hand argument's class or free functions. (Some operators, like assignment, must be members.) Since the stream operator's left-hand argument is a stream, they either have to be members of the stream class or free functions. The canonical way to implement `operator<<` for any type is this:

```
std::ostream& operator<<(std::ostream& os, const T& obj)
{
   // stream obj's data into os
   return os;
}
```

Note that it is ***not*** a member function. Also note that it takes the object to stream per `const` reference. That's because you don't want to copy the object in order to stream it and you don't want the streaming to alter it either.

Sometimes you want to stream objects whose internals are not accessible through their class' public interface, so the operator can't get at them. Then you have two choices: Either put a public member into the class which does the streaming

```
class T {
  public:
    void stream_to(std::ostream&) {os << obj.data_;}
  private:
    int data_;
};
```

and call that from the operator:

```
inline std::ostream& operator<<(std::ostream& os, const T& obj)
{
   obj.stream_to(os);
   return os;
}
```

or make the operator a `friend`

Sometimes you want to stream objects whose internals are not accessible through their class' public interface, so the operator can't get at them. Then you have two choices: Either put a public member into the class which does the streaming

```
class T {
  public:
    void stream_to(std::ostream&) {os << obj.data_;}
  private:
    int data_;
};
```

and call that from the operator:

```
inline std::ostream& operator<<(std::ostream& os, const T& obj)
{
   obj.stream_to(os);
   return os;
}
```

or make the operator a `friend`

```
class T {
  public:
    friend std::ostream& operator<<(std::ostream&, const T&);
  private:
    int data_;
};
```

so that it can access the class' private parts:

```
inline std::ostream& operator<<(std::ostream& os, const T& obj)
{
   os << obj.data_;
   return os;
}
```

```cpp
using namespace std;
class Paragraph
{
    public:
        Paragraph(std::string const& init)
            :m_para(init)
        {}
        std::string const&  to_str() const
        {
            return m_para;
        }
        bool operator==(Paragraph const& rhs) const
        {
            return m_para == rhs.m_para;
        }
        bool operator!=(Paragraph const& rhs) const
        {
            // Define != operator in terms of the == operator
            return !(this->operator==(rhs));
        }
        bool operator<(Paragraph const& rhs) const
        {
            return  m_para < rhs.m_para;
        }
    private:
        friend std::ostream & operator<<(std::ostream &os, const Paragraph& p);
        std::string     m_para;
};
std::ostream & operator<<(std::ostream &os, const Paragraph& p)
{
    return os << p.to_str();
}
int main()
{
    Paragraph   p(“My Paragraph");
    Paragraph   q(p);
    std::cout << p << std::endl << (p == q) << std::endl;
}
```

```
$ ./Paragraph.exe
My Paragraph
1
```