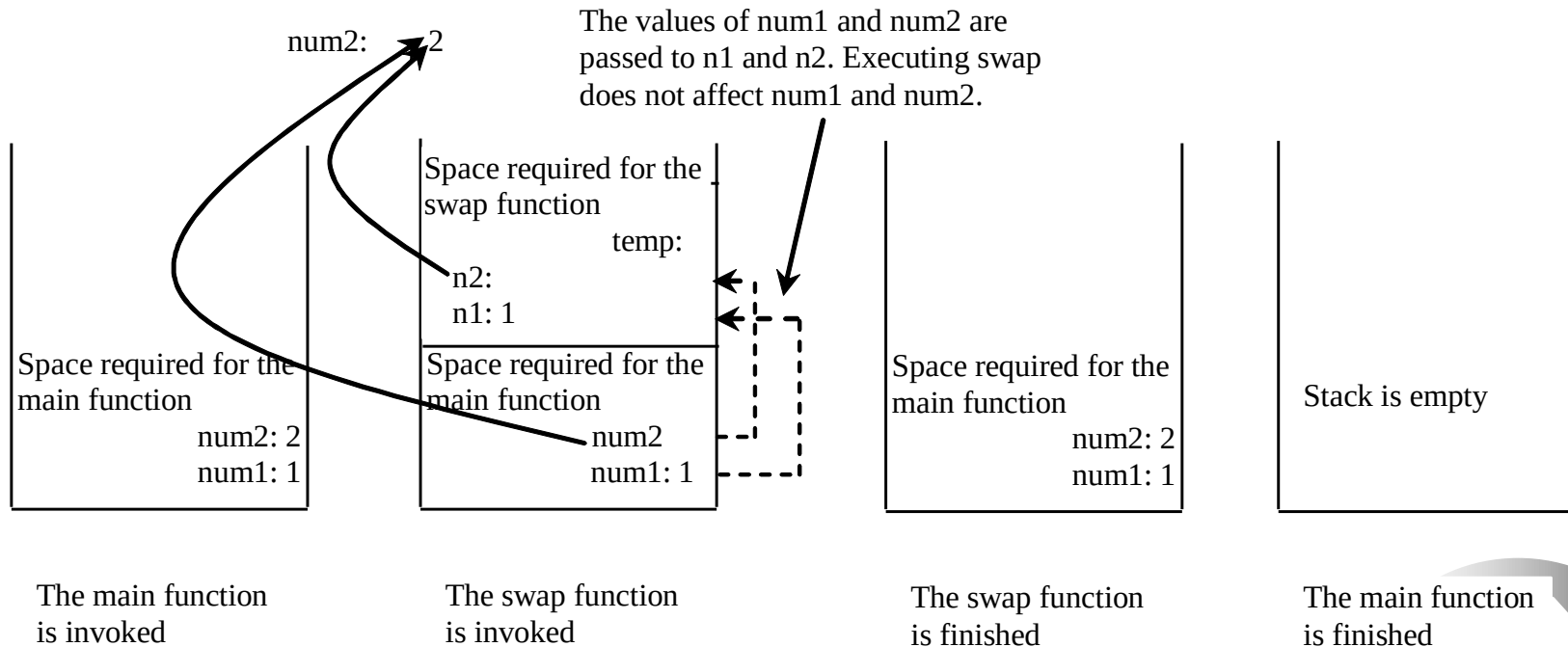


Chapter 6 Functions



Pass by Reference, cont.



Reference Variables

C++ provides a special type of variable, called a *reference variable*, which can be used as a function parameter to reference the original variable. A reference variable is an alias for another variable. Any changes made through the reference variable are actually performed on the original variable. To declare a reference variable, place the ampersand (&) in front of the name. For example, see Listing 6.4.

TestReferenceVariable

TestReferenceVariable



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

int main()
{
    int count = 1;
    int& r = count;
    cout << "count is " << count << endl;
    cout << "r is " << r << endl;

    r++;
    cout << "count is " << count << endl;
    cout << "r is " << r << endl;

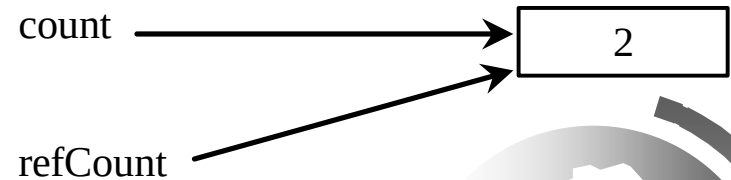
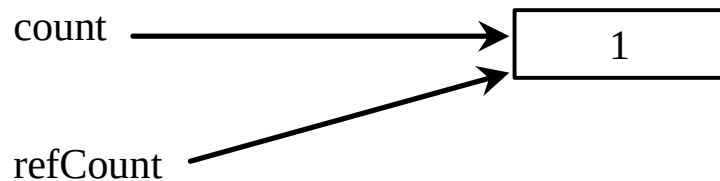
    count = 10;
    cout << "count is " << count << endl;
    cout << "r is " << r << endl;

    return 0;
}
```



Pass By Reference

You can use a reference variable as a parameter in a function and pass a regular variable to invoke the function. The parameter becomes an alias for the original variable. This is known as *pass-by-reference*. When you change the value through the reference variable, the original value is actually changed.



SwapByReference

SwapByReference



```

#include <iostream>
using namespace std;

// Swap two variables
void swap(int& n1, int& n2)
{
    cout << "\tInside the swap function" << endl;
    cout << "\tBefore swapping n1 is " << n1 <<
        " n2 is " << n2 << endl;

    // Swap n1 with n2
    int temp = n1;
    n1 = n2;
    n2 = temp;

    cout << "\tAfter swapping n1 is " << n1 <<
        " n2 is " << n2 << endl;
}

int main()
{
    // Declare and initialize variables
    int num1 = 1;
    int num2 = 2;

    cout << "Before invoking the swap function, num1 is "
        << num1 << " and num2 is " << num2 << endl;

    // Invoke the swap function to attempt to swap two variables
    swap(num1, num2);

    cout << "After invoking the swap function, num1 is " << num1 <<
        " and num2 is " << num2 << endl;

    return 0;
}

```



Pass By Reference

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

void increment(int& n) // Pass by reference
{
    n++;
    cout << "n inside the function is " << n << endl;
}

int main()
{
    int x = 1;
    cout << "Before the call, x is " << x << endl;
    increment(x);
    cout << "After the call, x is " << x << endl;

    return 0;
}
```

IncrementWithPassByReference

IncrementWithPassByReference



Constant Reference Parameters

```
// Return the max between two numbers
int max(const int& num1, const int& num2)
{
    int result;
    if (num1 > num2)
        result = num1;
    else
        result = num2;
    return result;
}
```



Pass-by-Value or Pass-by-Reference

In pass-by-value, the actual parameter and its formal parameter are independent variables. In pass-by-reference, the actual parameter and its formal parameter refer to the same variable. Pass-by-reference is more efficient than pass-by-value. However, the difference is negligible for parameters of primitive types such as int and double. So, *if a primitive data type parameter is not changed in the function, you should declare it as pass-by-value parameter.*



Case Study: Converting Hexadecimals to Decimals

Write a function that converts a hexadecimal number into a decimal number.

ABCD =>

$$A*16^3 + B*16^2 + C*16^1 + D*16^0$$

$$= ((A*16 + B)*16 + C)*16 + D$$

Hex2Dec

Hex2Dec



```

#include <iostream>
#include <string>
#include <cctype>
using namespace std;

// Converts a hex number as a string to decimal
int hex2Dec(const string& hex);

// Converts a hex character to a decimal value
int hexCharToDecimal(char ch);

int main()
{
    // Prompt the user to enter a hex number as a string
    cout << "Enter a hex number: ";
    string hex;
    cin >> hex;

    cout << "The decimal value for hex number " << hex
         << " is " << hex2Dec(hex) << endl;

    return 0;
}

int hex2Dec(const string& hex)
{
    int decimalValue = 0;
    for (unsigned i = 0; i < hex.size(); i++)
        decimalValue = decimalValue * 16 + hexCharToDecimal(hex[i]);

    return decimalValue;
}

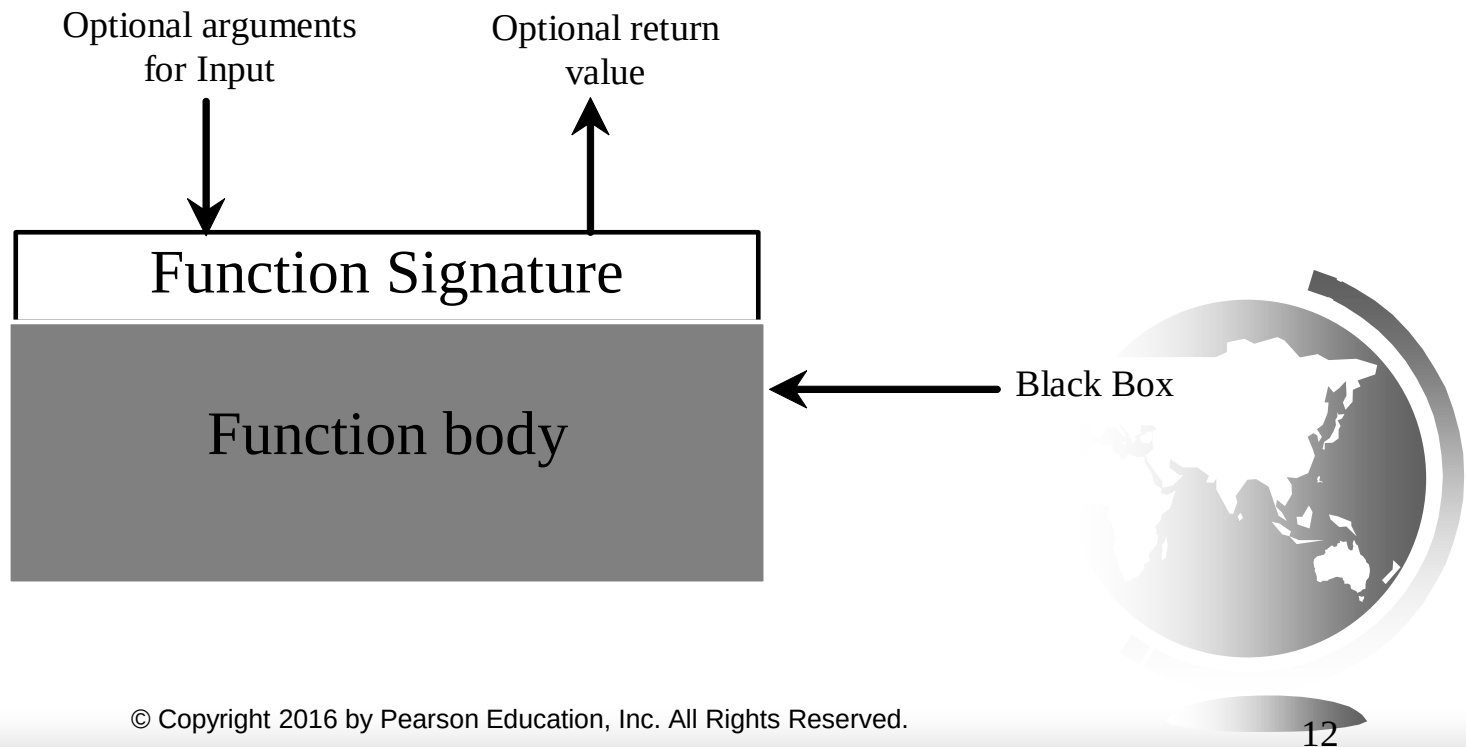
int hexCharToDecimal(char ch)
{
    ch = toupper(ch); // Change it to uppercase
    if ('A' <= ch && ch <= 'F')
        return 10 + ch - 'A';
    else // ch is '0', '1', ..., or '9'
        return ch - '0';
}

```



Function Abstraction

You can think of the function body as a black box that contains the detailed implementation for the function.



Benefits of Functions

- Write a function once and reuse it anywhere.
- Information hiding. Hide the implementation from the user.
- Reduce complexity.



Stepwise Refinement

The concept of Function abstraction can be applied to the process of developing programs. When writing a large program, you can use the “divide and conquer” strategy, also known as *stepwise refinement*, to decompose it into subproblems. The subproblems can be further decomposed into smaller, more manageable problems.

PrintCalendar

PrintCalendar



PrintCalendar Case Study

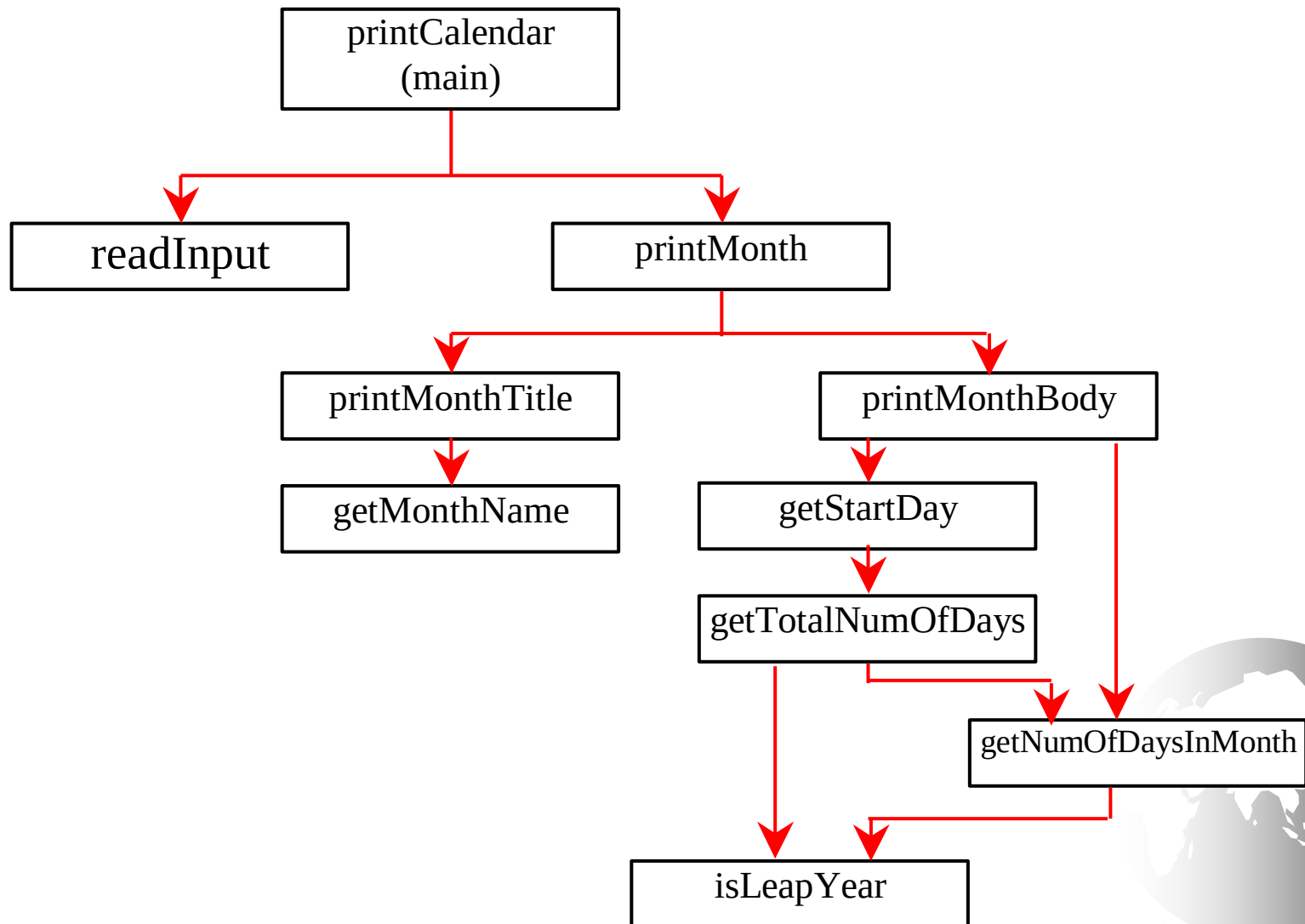
Let us use the PrintCalendar example to demonstrate the stepwise refinement approach.

December 2005

Sun	Mon	Tue	Wed	Thu	Fri	Sat
				1	2	3
4	5	6	7	8	9	10
11	12	13	14	15	16	17
18	19	20	21	22	23	24
25	26	27	28	29	30	31



Design Diagram



Implementation: Top-Down

Top-down approach is to implement one Function in the structure chart at a time from the top to the bottom. Stubs can be used for the Functions waiting to be implemented. A stub is a simple but incomplete version of a Function. The use of stubs enables you to test invoking the Function from a caller. Implement the main Function first and then use a stub for the printMonth Function. For example, let printMonth display the year and the month in the stub. Thus, your program may begin like this:

PrintCalendarSkeleton

PrintCalendarSkeleton



```

#include <iostream>
#include <iomanip>
using namespace std;

void printMonth(int year, int month);
void printMonthTitle(int year, int month);
void printMonthName(int month);
void printMonthBody(int year, int month);
int getStartDay(int year, int month);
int getTotalNumberOfDays(int year, int month);
int getNumberOfDaysInMonth(int year, int month);
bool isLeapYear(int year);

int main()
{
    // Prompt the user to enter year
    cout << "Enter full year (e.g., 2001): ";
    int year;
    cin >> year;

    // Prompt the user to enter month
    cout << "Enter month in number between 1 and 12: ";
    int month;
    cin >> month;

    // Print calendar for the month of the year
    printMonth(year, month);

    return 0;
}

void printMonth(int year, int month)
{
    cout << month << " " << year << endl;
}

```



Implementation: Bottom-Up

Bottom-up approach is to implement one Function in the structure chart at a time from the bottom to the top. For each Function implemented, write a test program to test it. Both top-down and bottom-up Functions are fine. Both approaches implement the Functions incrementally and help to isolate programming errors and makes debugging easy. Sometimes, they can be used together.



Benefits of Stepwise Refinement

Simpler Program

Reusing Functions

Easier Developing, Debugging, and Testing

Better Facilitating Teamwork

