# Chapter 9: Completing the Basics

C++ FOR ENGINEERS AND SCIENTISTS

# Objectives

- In this chapter, you will learn about:
  - Exception handling
  - Exceptions and file checking
  - The **string** class
  - Character manipulation functions
  - Input data validation
  - Namespaces and creating a personal library
  - Common programming errors

# Exception Handling

- Traditional C++ approach to error handling uses a function to return a specific value to indicate specific operations
- Latest C++ compilers have added a technique designed for error detection and handling referred to as **exception handling**
- When an error occurs while a function is executing, an exception is created
- An exception is a value, a variable, or an object containing information about the error at the point the error occurs

Refer to page 512 for more explanations and examples

# Exception Handling (continued)

- **Throwing an exception:** Process of generating an exception
- In general two fundamental types of errors can cause C++ exceptions
  - Those resulting from inability to obtain a required resource, over which the programmer has no control
  - Errors than can be checked and handled, over which the programmer has control

# Exception Handling (continued)

| Terminology | Description |
|---|---|
| Exception | A value, a variable, or an object that identifies a specific error that has occurred while a program is running |
| Throw an exception | Send the exception to a section of code that processes the detected error |
| Catch or handle an exception | Receive a thrown exception and process it |
| Catch clause | The section of code that processes the error |
| Exception handler | The code that throws and catches an exception |

**Table 9.1** Exception-Handling Terminology

# Exception Handling (continued)

- General syntax of code required to throw and catch and exception:

```
try
{
   // one or more statements, at least one of which
   // should be capable of throwing an exception
}
catch(exceptionDataType parameterName)
{
   // one or more statements
}
```

# Exception Handling

division-by-zero error

```
try
{
  cout << "Enter the numerator: (whole number only): ";
  cin  >> numerator;
  cout << "Enter the denominator: (whole number only): ";
  cin  >> denominator;
  if (denominator == 0)
    throw denominator;
  else
    result = numerator/denominator;
}
```

# Exception Handling

division-by-zero error

```
catch(int e)
  {
    cout << "A denominator value of " << e << " is invalid." << endl;
    exit (1);
  }
```

Refer to pages 516,517 for more explanations and examples

# Exceptions and File Checking

- Error checking and processing with exception handling is used extensively in C++ programs that use one or more files

Refer to page 519 for more explanations and examples

# Exceptions and File Checking (continued)

- A rigorous check is usually required when opening output file
  - If it exists, file will be found
  - If it does not exist, file will be created
- In some cases, file can be opened for input and, if file is found, a further check can be made to ensure that user explicitly approves overwriting it

Refer to page 521 for more explanations and examples

# Opening Multiple Files

- To open two files at the same time, assume you want to read data from a character-based file one character at a time and write this data to a file

# Opening Multiple Files (continued)

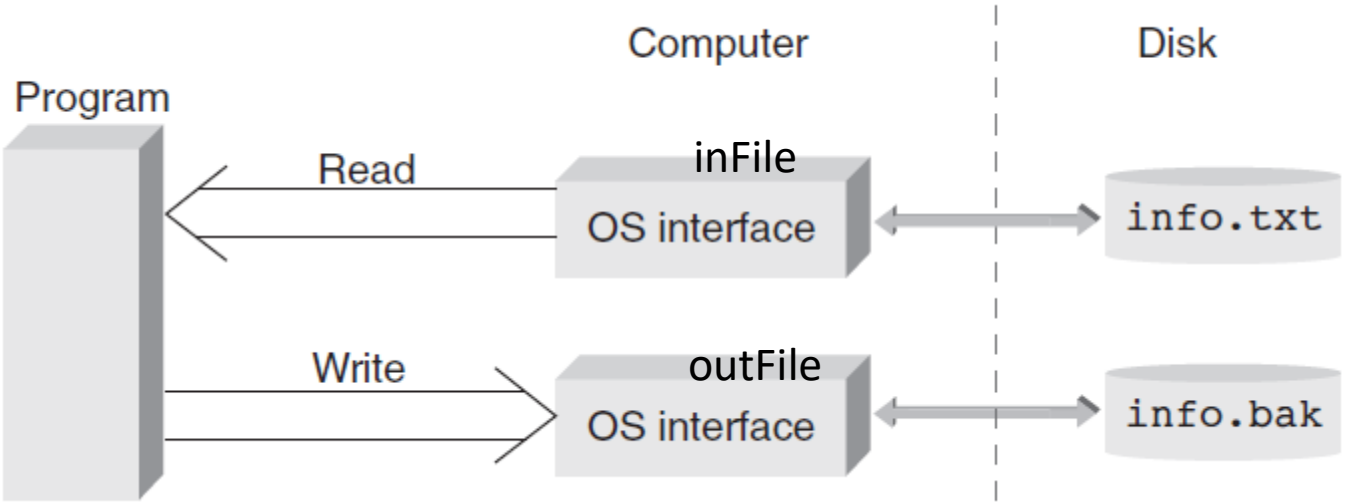



**Figure 9.2** The file copy stream structure

Refer to page 523 for more explanations and examples

# The `string` Class

- The `string` class permits string literal values
- **String literal:** Any sequence of characters enclosed in quotation marks
- By convention, first character in string is always designated as position zero

# The `string` Class (continued)

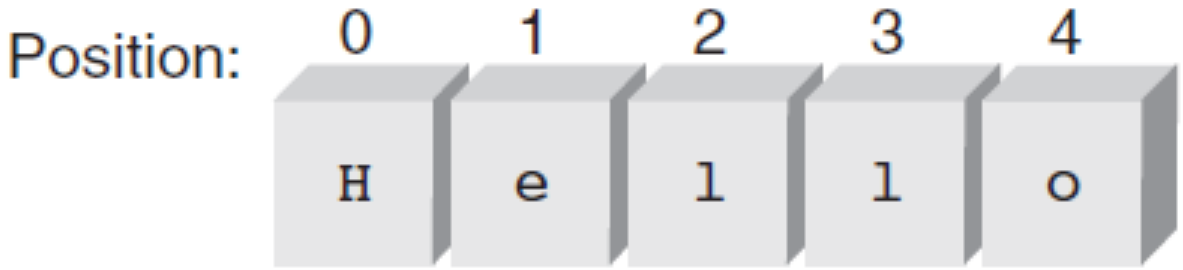



**Figure 9.3** The storage of a string as a sequence of characters

This position value is also referred to as both the character's **index value** and its **offset value**

# `string` Class Functions

- `string` class provides a number of functions for declaring, creating, and initializing a string
- In earlier versions of C++, process of creating a new object is referred to as **instantiating an object**
- The methods that perform the tasks of creating and initializing are called **constructor methods,** or **constructors**, for short

# `string` Class Functions

| Constructor | Description | Examples |
|---|---|---|
| `string objectName = value` | Creates and initializes a string object to a value that can be a string literal, a previously declared string object, or an expression containing string literals and string objects | `string str1 = "Good Morning";`<br>`string str2 = str1;`<br>`string str3 = str1 + str2;` |
| `string objectName(stringValue)` | Produces the same initialization as the preceding item | `string str1("Hot");`<br>`string str1(str1 + " Dog");` |
| `string objectName(str, n)` | Creates and initializes a string object with a substring of string object `str`, starting at index position *n* of `str` | `string str1(str2, 5)`<br>If `str2` contains the string `Good Morning`, then `str1` becomes the string `Morning` |

**Table 9.2** string Class Constructors
(Require the Header File string)

# `string` Class Functions

| | | |
|---|---|---|
| `string objectName(str, n, p)` | Creates and initializes a string object with a substring of string object `str`, starting at index position *n* of `str` and containing *p* characters | `string str1(str2, 5, 2)` If `str2` contains the string `Good Morning`, then `str1` becomes the string `Mo` |
| `string objectName(n, char)` | Creates and initializes a string object with *n* copies of `char` | `string str1(5, '*')` This makes `str1 = "*****"` |
| `string objectName;` | Creates and initializes a string object to represent an empty character sequence (same as `string objectName = "";`, so the string's length is 0) | `string message;` |

**Table 9.2** continue

Refer to page 529 for more explanations and examples

# String Input and Output (continued)

- In addition to string being initialized with constructors listed in Table 9.2, strings can be input from the keyboard and displayed on the screen

| C++ Object or Function | Description |
|---|---|
| `cout` | General-purpose screen output object |
| `cin` | General-purpose keyboard input object that stops reading string input when white space is encountered |
| `getline(cin, strObj)` | General-purpose keyboard input function that inputs all characters entered, stores them in the string `strObj`, and stops accepting characters when it receives a newline character (`\n`) |

**Table 9.3** `string` Class Input and Output

# String Input and Output (continued)

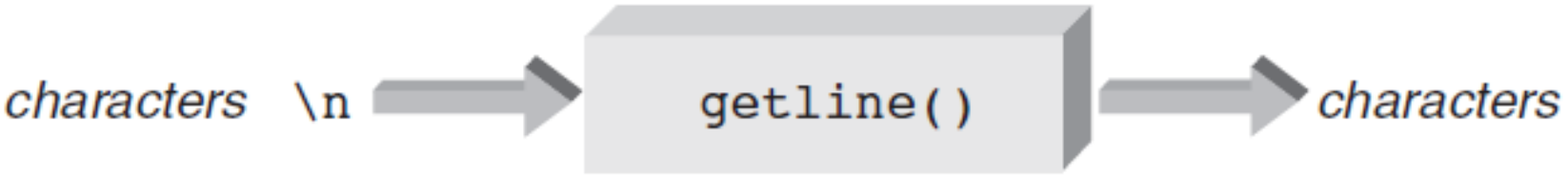



**Figure 9.5** Inputting a string with `getline()`

# String Input and Output (continued)

## Program 9.7

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

int main()
{
  string message;      // declare a string object

  cout << "Enter a string:\n";
  getline(cin, message);
  cout << "The string just entered is:\n"
       << message << endl;

  return 0;
}
```

Refer to pages 531-533 for more explanations and examples

# String Processing

- Strings can be manipulated using the `string` class functions or the character-at-a-time functions described in Section 9.4

- Most commonly used function in Table 9.4 is `length()`, which returns number of characters in the string

| Function/Operation | Description | Example |
|---|---|---|
| `int length()` | Returns the length of the string | `string1.length()` |

# String Processing

- Two string expressions can be compared for equality using the standard relational operators

| | | |
|---|---|---|
| `==` `!=`<br>`<` `<=`<br>`>` `>=` | Relational operators return `true` if the relation is satisfied; otherwise, return `false` | `string1 == string2`<br>`string1 <= string2`<br>`string1 > string2` |

"Hello" < "hello"

"123" > "1227"

Refer to pages 534-537 for more explanations and examples

# String Processing (continued)

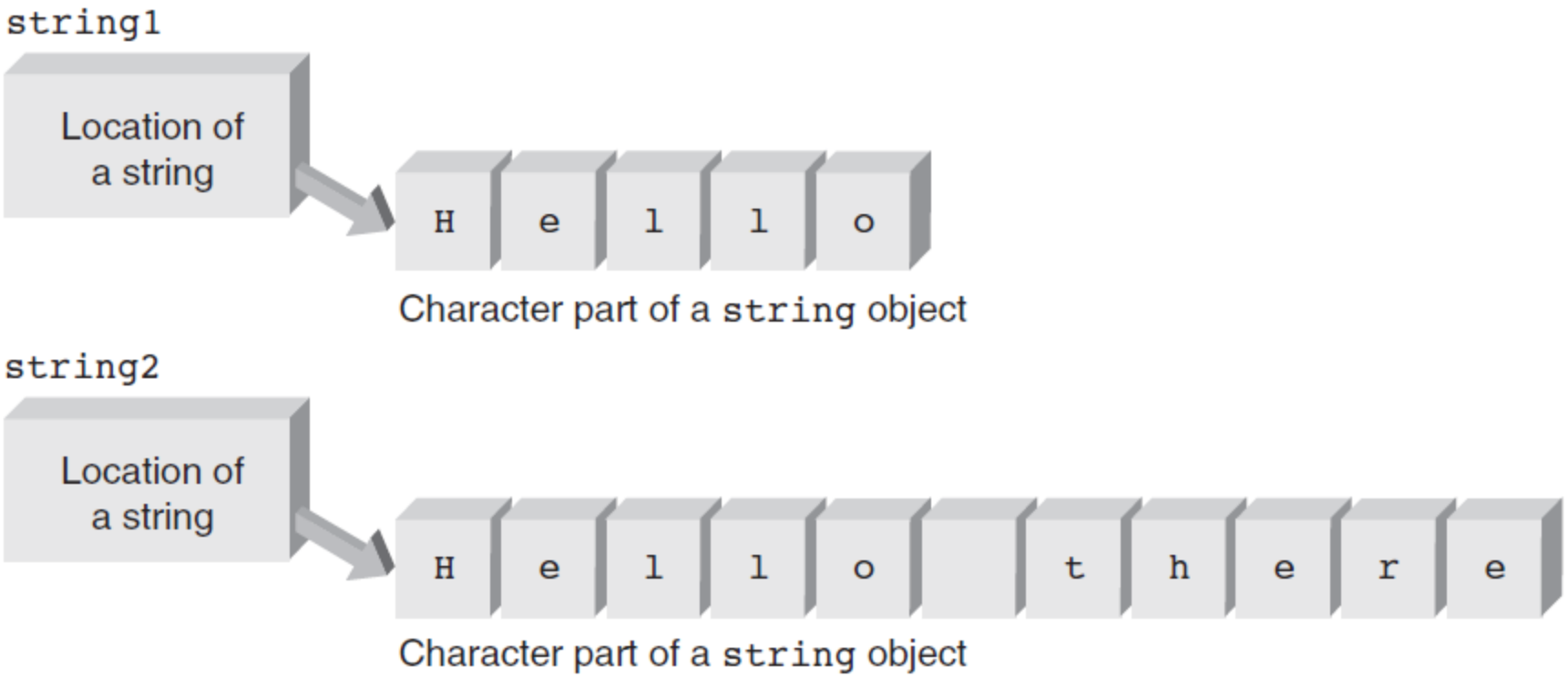



**Figure 9.7** The initial strings used in Program 9.9

Refer to page 539 for more explanations and examples

# String Processing (continued)

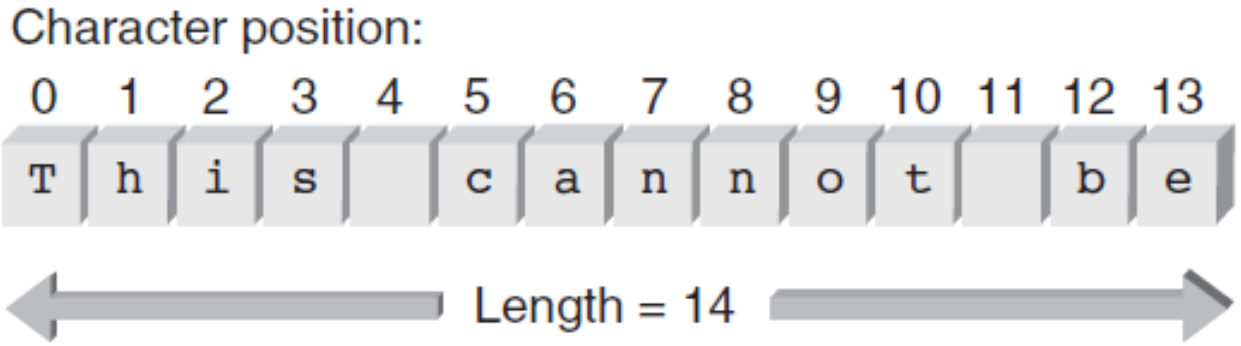



**Figure 9.8** Initial storage of a string object

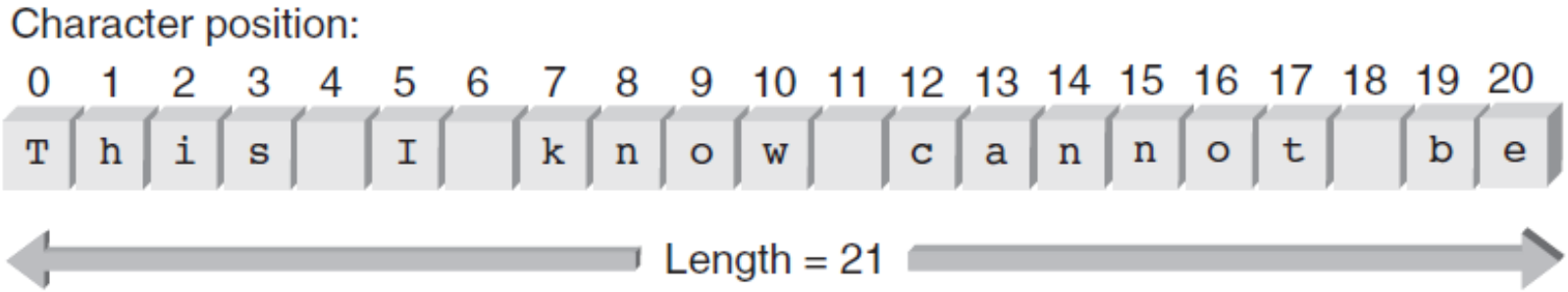



**Figure 9.9** The string after the insertion

# String Processing (continued)

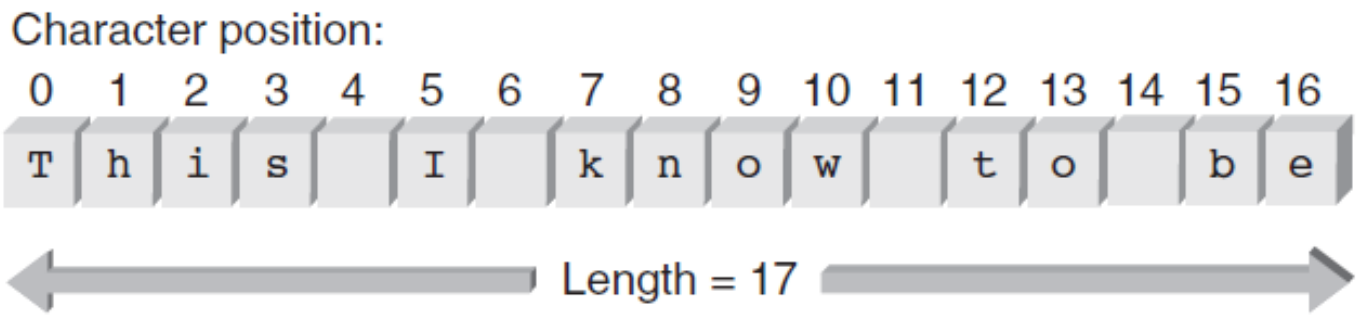



**Figure 9.10** The string after the replacement

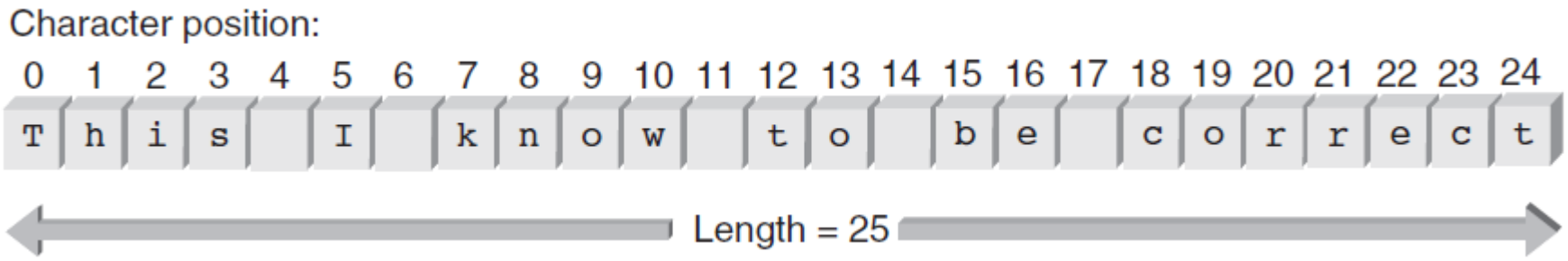



**Figure 9.11** The string after the append

Refer to pages 541,542 for more explanations and examples

# Character Manipulation Functions

- In addition to string functions provided by `string` class, the C++ language provides several useful character class functions
- Function declaration (prototype) for each function is contained in the header file `string` or `cctype`, which must be included in any program using these functions

| Function Prototype | Description | Example |
|---|---|---|
| `int isalpha(charExp)` | Returns a `true` (non-zero integer) if `charExp` evaluates to a letter; otherwise, it returns a `false` (zero integer) | `isalpha('a')` |

**Table 9.5** Character Library Functions

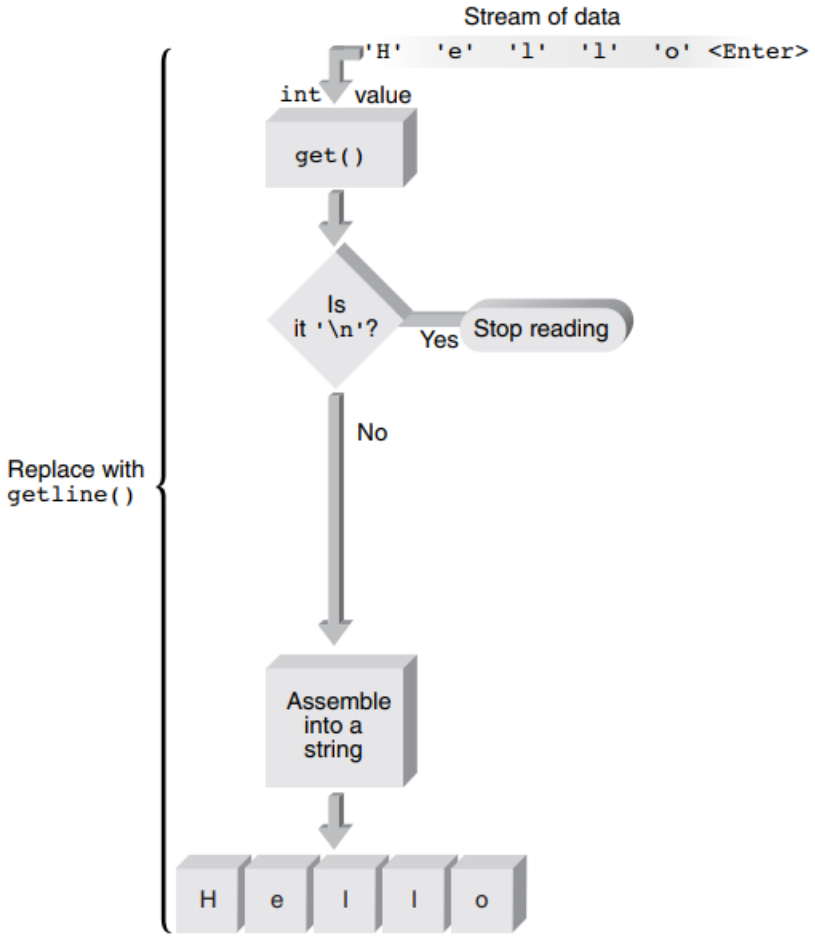
# Character Manipulation Functions

| | | |
|---|---|---|
| `int isdigit(charExp)` | Returns a `true` (non-zero integer) if `charExp` evaluates to a digit (0 through 9); otherwise, it returns a `false` (zero integer) | `isdigit('a')` |
| `int ispunct(charExp)` | Returns a `true` (non-zero integer) if `charExp` evaluates to a punctuation character; otherwise, returns a `false` (zero integer) | `ispunct('!')` |

```
if (isdigit(ch))
  cout << "The character just entered is a digit" << endl;
else if (ispunct(ch))
  cout << "The character just entered is a punctuation mark" << endl;
```

Refer to pages 544-547 for more explanations and examples

# Character I/O



# Character I/O

| Function | Description | Example |
|---|---|---|
| `cout.put(charExp)` | Places the character value of `charExp` on the output stream. | `cout.put('A');` |
| `cin.get(charVar)` | Extracts the next character from the input stream and assigns it to the variable `charVar`. | `cin.get(key);` |
| `cin.peek(charVar)` | Assigns the next character from the input stream to the variable `charVar` *without* extracting the character from the stream. | `cin.peek(nextKey);` |
| `cin.putback(charExp)` | Pushes a character value of `charExp` back onto the input stream. | `cin.putback(cKey);` |

**Table 9.6** Basic Character I/O Functions (Require the header file `cctype`)

# Character I/O (continued)

| Function | Description | Example |
|---|---|---|
| `cin.ignore(n, char)` | Ignores a maximum of the next *n* input characters, up to and including the detection of `char`. If no arguments are specified, ignores the next single character on the input stream. | `cin.ignore(80,'\n');`<br>`cin.ignore();` |

**Table 9.6** Basic Character I/O Functions (Require the header file `cctype`) (continued)

# Phantom Newline Character Revisited

## Program 9.15

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

int main()
{
  char fkey;

  cout << "Type in a character: ";
  cin.get(fkey);
  cout << "The key just accepted is " << int(fkey) << endl;

  return 0;
}
```

```
Type in a character: m
The key just accepted is 109
```

# Phantom Newline Character Revisited (continued)

## Program 9.16

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

int main()
{
  char fkey, skey;

  cout << "Type in a character: ";
  cin.get(fkey);
  cout << "The key just accepted is " << int(fkey) << endl;

  cout << "Type in another character: ";
  cin.get(skey);
  cout << "The key just accepted is " << int(skey) << endl;

  return 0;
}
```

```
Type in a character: m
The key just accepted is 109
Type in another character:
The key just accepted is 10
```

Refer to page 552 for more explanations and examples

# Phantom Newline Character Revisited (continued)

Program 9.17

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

int main()
{
  char fkey, skey;

  cout << "Type in a character: ";
  cin.get(fkey);
  cout << "The key just accepted is " << int(fkey) << endl;
  cin.ignore();

  cout << "Type in another character: ";
  cin.get(skey);
  cout << "The key just accepted is " << int(skey) << endl;

  cin.ignore();
  return 0;
}
```

# A Second Look at User-Input Validation

- Sign of well-constructed, robust program:
  - Code that validates user input and ensures that program doesn't produce unintended results caused by unexpected input
- **User-input validation:** Basic technique for handling invalid data input and preventing seemingly innocuous code from producing unintended results
  - Validates entered data during or after data entry and gives the user a way of reentering data if it is invalid

Refer to page 554 for more explanations and examples

# Input Data Validation

- Validating user input is essential
- Successful programs anticipate invalid data and prevent it from being accepted and processed
- A common method for validating numerical input data is accepting all numbers as strings
- After string is validated it can be converted to the correct type

# Input Data Validation (continued)

| Function | Description | Example |
|---|---|---|
| `int atoi(stringExp)` | Converts `stringExp` to an integer. Conversion stops at the first noninteger character. | `atoi("1234")` |
| `double atof(stringExp)` | Converts `stringExp` to a double-precision number. Conversion stops at the first character that can't be interpreted as a `double`. | `atof("12.34")` |
| `char[] itoa(integerExp)` | Converts `integerExp` to a character array. The space allocated for the returned characters must be large enough for the converted value. | `itoa(1234)` |

**Table 9.7** C-String Conversion Functions

Refer to pages 558,560 for more explanations and examples

# A Closer Look: Namespaces and Creating a Personal Library

- First step in creating a library is to encapsulate all specialized functions and classes into one or more namespaces and then store the complete code (with or without using a namespace)
- The syntax for creating a namespace:

```
namespace name
{
    // functions and/or classes in here
}   // end of namespace
```

# A Closer Look: Namespaces and Creating a Personal Library (cont'd)

```
namespace dataChecks
{
  bool isvalidInt(string str)
  {
    int start = 0;
    int i;
    bool valid = true;  // assume a valid integer
    bool sign = false;  // assume no sign

// Check for an empty string
    if (int(str.length()) == 0)  valid = false;

// Check for a leading sign
    if (str.at(0) == '-' || str.at(0) == '+')
    {
      sign = true;
      start = 1;  // start checking for digits after the sign
    }

// Check that there's at least one character after the sign
    if (sign && int(str.length()) == 1) valid = false;

// Now check the string, which you know
// has at least one non-sign character
    i = start;
    while (valid && i < int(str.length()))
    {
      if (!isdigit(str.at(i))) valid = false;  // found a
                                               // non-digit character
```

# A Closer Look: Namespaces and Creating a Personal Library (cont'd)

```
    i++;  // move to next character
  }

  return valid;
}

int getanInt()
{
  bool isvalidInt(string);  // function declaration (prototype)
  bool notanint = true;
  string svalue;

  while (notanint)
  {
    try
    {
      cin >> svalue;  // accept a string input
      if (!isvalidInt(svalue)) throw svalue;
    }
    catch(string e)
    {
      cout << "Invalid integer - Please reenter: ";
        continue;  // send control to the while statement
    }
    notanint = false;
  }

  return atoi(svalue.c_str());  // convert to an integer
}
} // end of dataChecks namespace
```

# A Closer Look: Namespaces and Creating a Personal Library (cont'd)

Program 9.20

```
#include <iostream>
#include <string>
using namespace std;
#include <c:\\mylibrary\\dataChecks.cpp>
using namespace dataChecks;

int main()
{
  int value;

  cout << "Enter an integer value: ";
  value = getanInt();
  cout << "The integer entered is: " << value << endl;

  return 0;
}
```

# Common Programming Errors

- Forgetting to include `string` header file when using `string` class object
- Forgetting that newline character, '\n', is a valid input character
- Forgetting to convert `string` class object by using `c_str()` function when converting `string` class objects to numerical data types

# Summary

- String literal is any sequence of characters enclosed in quotation marks
  - Referred to as a string value, a string constant, and, more conventionally, a string
- String can be can be constructed as an object of the `string` class
- `string` class is commonly used for constructing strings for input and output purposes
- Strings can be manipulated by using functions of the class they're objects of or by using the general purpose string and character functions

# Summary (continued)

- `cin` object tends to be of limited usefulness for string input because it terminates input when a blank is encountered
- For `string` class data input, use the `getline()` function
- `cout` object can be used to display `string` class strings

# Homework

- Page 570, exercises 6, 7 and 8