Lecture 35:

Operator Overloading

Ioan Raicu
Department of Electrical Engineering & Computer Science
Northwestern University

EECS 211
Fundamentals of Computer Programming II
May 26th, 2010

11.1 Introduction

- This chapter shows how to enable C++'s operators to work with objects—a process called operator overloading.
- One example of an overloaded operator built into C++ is **<<**, which is used both as the stream insertion opera-tor and as the bitwise left-shift operator..
- C++ overloads the addition operator (**+**) and the subtraction operator (**-**).
- These operators perform differently, depending on their context in integer, floating-point and pointer arithmetic.
- C++ enables you to overload most operators—the compiler generates the appropriate code based on the context.

11.2 Fundamentals of Operator Overloading

- The fundamental types can be used with C++’s rich collection of operators.
- You can use operators with user-defined types as well.
- Although C++ does not allow new operators to be created, it does allow most existing operators to be overloaded so that, when they’re used with objects, they have meaning appro-priate to those objects.

Software Engineering Observation 11.1

Operator overloading contributes to C++'s extensibility—one of the language's most appealing attributes.

Good Programming Practice 11.1

Use operator overloading when it makes a program clearer than accomplishing the same operations with function calls.

Good Programming Practice 11.2

Overloaded operators should mimic the functionality of their built-in counterparts—for example, the + operator should be overloaded to perform addition, not subtraction. Avoid excessive or inconsistent use of operator overloading, as this can make a program cryptic and difficult to read.

11.2 Fundamentals of Operator Overloading (cont.)

- An operator is overloaded by writing a non-`static` member function definition or global function definition as you normally would, except that the function name now becomes the keyword `operator` followed by the symbol for the operator being overloaded.
  – For example, the function name `operator+` would be used to overload the addition operator (`+`).
- When operators are overloaded as member functions, they must be non-`static`, because they must be called on an object of the class and operate on that object.

11.2 Fundamentals of Operator Overloading (cont.)

- To use an operator on class objects, that operator *must be overloaded—with three ex-ceptions*.
- The assignment operator (=) may be used with every class to perform memberwise assignment of the class's data members.
  - Dangerous for classes with pointer members; we'll explicitly overload the assignment operator for such classes.
- The address (&) and comma (,) operators may also be used with objects of any class without overloading.
  - The address operator re-turns a pointer to the object.
  - The comma operator evaluates the expression to its left then the expression to its right, and returns the value of the latter expression.
- Operator overloading is not automatic—you must write op-era-tor-overloading functions to perform the desired operations.

11.3 Restrictions on Operator Overloading

- Most of C++’s operators can be overloaded (Fig. 11.1).
- Figure 11.2 shows the operators that cannot be overloaded.

Operators that can be overloaded							
+	-	*	/	%	^	&	|
~	!	=	<	>	+=	-=	*=
/=	%=	^=	&=	|=	<<	>>	>>=
<<=	==	!=	<=	>=	&&	||	++
--	->*	,	->	[]	()	new	delete
new[]	delete[]						

Fig. 11.1 | Operators that can be overloaded.

Operators that cannot be overloaded			
.	.*	::	?:

Fig. 11.2 | Operators that cannot be overloaded.

11.3 Restrictions on Operator Overloading (cont.)

- The precedence of an operator cannot be changed by overloading.
- The associativity of an operator (i.e., whether the operator is applied right-to-left or left-to-right) cannot be changed by overloading.
- It isn't possible to change the "arity" of an operator (i.e., the number of operands an operator takes): Overloaded unary operators remain unary operators; overloaded binary operators remain binary operators.
- C++'s only ternary operator (`?:`) cannot be overloaded.
- Operators `&`, `*`, `+` and `-` all have both unary and binary versions; these unary and binary versions can each be overloaded.

Common Programming Error 11.1

Attempting to change the "arity" of an operator via operator overloading is a compilation error.

11.3 Restrictions on Operator Overloading (cont.)

- It isn't possible to create new operators; only existing operators can be overloaded.
- You could overload an existing operator to perform exponentiation.

Common Programming Error 11.2

Attempting to create new operators via operator overloading is a syntax error.

11.3 Restrictions on Operator Overloading (cont.)

- The meaning of how an operator works on fundamental types cannot be changed by operator overloading.
  - You cannot, for example, change the mean-ing of how **+** adds two integers.
- Operator overloading works only with objects of user-defined types or with a mixture of an object of a user-defined type and an object of a fundamental type.

Software Engineering Observation 11.2

At least one argument of an operator function must be an object or reference of a user-defined type. This prevents you from changing how operators work on fundamental types.

Common Programming Error 11.3

Attempting to modify how an operator works with objects of fundamental types is a compilation error.

11.3 Restrictions on Operator Overloading (cont.)

- Overloading an assignment operator and an addition operator to allow statements like
  - `object2 = object2 + object1;`
- does not imply that the += operator is also overloaded to allow statements such as
  - `object2 += object1;`
- Such behavior can be achieved only by explicitly overloading operator += for that class.

Common Programming Error 11.4

Assuming that overloading an operator such as `+` *overloads related operators such as* `+=` *or that overloading* `==` *overloads a related operator like* `!=` *can lead to errors. Operators can be overloaded only explicitly; there is no implicit overloading.*

11.4 Operator Functions as Class Members vs. Global Functions

- Operator functions can be member functions or global functions.
  - Global functions are often made `friend`s for performance reasons.
- Member functions use the `this` pointer implicitly to obtain one of their class object arguments (the left operand for binary operators).
- Arguments for both operands of a binary operator must be explicitly listed in a global function call.

11.4 Operator Functions as Class Members vs. Global Functions (cont.)

- When overloading `()`, `[]`, `->` or any of the assignment operators, the operator overloading function must be de-clared as a class member.
- For the other operators, the operator overloading functions can be class members or standalone functions.

11.4 Operator Functions as Class Members vs. Global Functions (cont.)

- Whether an operator function is implemented as a member function or as a global function, the operator is still used the same way in expressions.
- When an operator function is implemented as a member function, the leftmost (or only) operand must be an object (or a reference to an object) of the operator's class.
- If the left operand must be an object of a different class or a fundamental type, this op-er-ator function must be im-plemented as a global function (as we'll do with `<<` and `>>`).
- A global operator function can be made a `friend` of a class if that function must access `private` or `protected` members of that class directly.
- Operator member functions of a specific class are called only when the left operand of a binary operator is specifically an object of that class, or when the single operand of a unary operator is an object of that class.

11.4 Operator Functions as Class Members vs. Global Functions (cont.)

- The overloaded stream insertion operator (`<<`) is used in an expression in which the left operand has type `ostream &`, as in `cout << classObject`.
- To use the operator in this manner where the right operand is an object of a user-defined class, it must be overloaded as a global function.
- Similarly, the overloaded stream extraction operator (`>>`) is used in an expression in which the left operand has type `istream &`, as in `cin >> classObject`, and the right operand is an object of a user-defined class, so it, too, must be a global function.
- Each of these overloaded operator functions may require access to the `private` data members of the class object being output or input, so these overloaded operator functions can be made `friend` functions of the class for performance reasons.

Performance Tip 11.1

It's possible to overload an operator as a global, non-`friend` *function, but such a function requiring access to a class's* `private` *or* `protected` *data would need to use* set *or* get *functions provided in that class's* `public` *interface. The overhead of calling these functions could cause poor performance, so these functions can be inlined to improve performance.*

11.4 Operator Functions as Class Members vs. Global Functions (cont.)

- You might choose a global function to overload an operator to enable the operator to be commutative, so an object of the class can appear on the right side of a binary operator.
- The `operator+` function, which deals with an object of the class on the left, can still be a member function.
- The global function simply swaps its arguments and calls the member function.

11.5 Overloading Stream Insertion and Stream Extraction Operators

- You can input and output fundamental-type data using the stream extraction operator `>>` and the stream insertion operator `<<`.
- The C++ class libraries overload these operators to process each fundamental type, including pointers and C-style `char *` strings.
- You can also overload these operators to perform input and output for your own types.
- The program of Figs. 11.3–11.5 overloads these operators to input and output `PhoneNumber` objects in the format "`(000) 000-0000`." The program assumes telephone numbers are input correctly.

```cpp
// Fig. 11.3: PhoneNumber.h
// PhoneNumber class definition
#ifndef PHONENUMBER_H
#define PHONENUMBER_H

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

class PhoneNumber
{
   friend ostream &operator<<( ostream &, const PhoneNumber & );
   friend istream &operator>>( istream &, PhoneNumber & );
private:
   string areaCode; // 3-digit area code
   string exchange; // 3-digit exchange
   string line; // 4-digit line
}; // end class PhoneNumber

#endif
```

Fig. 11.3 | `PhoneNumber` class with overloaded stream insertion and stream extraction operators as `friend` functions.

```cpp
// Fig. 11.4: PhoneNumber.cpp
// Overloaded stream insertion and stream extraction operators
// for class PhoneNumber.
#include <iomanip>
#include "PhoneNumber.h"
using namespace std;

// overloaded stream insertion operator; cannot be
// a member function if we would like to invoke it with
// cout << somePhoneNumber;
ostream &operator<<( ostream &output, const PhoneNumber &number )
{
   output << "(" << number.areaCode << ") "
      << number.exchange << "-" << number.line;
   return output; // enables cout << a << b << c;
} // end function operator<<

```

Fig. 11.4 | Overloaded stream insertion and stream extraction operators for class `PhoneNumber`. (Part 1 of 2.)

```cpp
// overloaded stream extraction operator; cannot be
// a member function if we would like to invoke it with
// cin >> somePhoneNumber;
istream &operator>>( istream &input, PhoneNumber &number )
{
   input.ignore(); // skip (
   input >> setw( 3 ) >> number.areaCode; // input area code
   input.ignore( 2 ); // skip ) and space
   input >> setw( 3 ) >> number.exchange; // input exchange
   input.ignore(); // skip dash (-)
   input >> setw( 4 ) >> number.line; // input line
   return input; // enables cin >> a >> b >> c;
} // end function operator>>
```

Fig. 11.4 | Overloaded stream insertion and stream extraction operators for class `PhoneNumber`. (Part 2 of 2.)

```cpp
// Fig. 11.5: fig11_05.cpp
// Demonstrating class PhoneNumber's overloaded stream insertion
// and stream extraction operators.
#include <iostream>
#include "PhoneNumber.h"
using namespace std;

int main()
{
   PhoneNumber phone; // create object phone

   cout << "Enter phone number in the form (123) 456-7890:" << endl;

   // cin >> phone invokes operator>> by implicitly issuing
   // the global function call operator>>( cin, phone )
   cin >> phone;

   cout << "The phone number entered was: ";

   // cout << phone invokes operator<< by implicitly issuing
   // the global function call operator<<( cout, phone )
   cout << phone << endl;
} // end main
```

Fig. 11.5 | Overloaded stream insertion and stream extraction operators. (Part 1 of 2.)

```
Enter phone number in the form (123) 456-7890:
(800) 555-1212
The phone number entered was: (800) 555-1212
```

Fig. 11.5 | Overloaded stream insertion and stream extraction operators. (Part 2 of 2.)

11.5 Overloading Stream Insertion and Stream Extraction Operators (cont.)

- The stream extraction operator function `operator>>` (Fig. 11.4, lines 21–30) takes `istream` refer-ence `input` and `PhoneNumber` reference `number` as argu-ments and returns an `istream` reference.
- Operator function `operator>>` inputs phone numbers of the form
  - `(800) 555-1212`
- When the compiler sees the expression
  - `cin >> phone`
- it generates the global function call
  - `operator>>( cin, phone );`
- When this call executes, reference parameter `input` becomes an alias for `cin` and reference parameter `number` becomes an alias for `phone`.

11.5 Overloading Stream Insertion and Stream Extraction Operators (cont.)

- The operator function reads as `string`s the three parts of the telephone number.
- Stream manipulator `setw` limits the number of characters read into each `string`.
- The parentheses, space and dash characters are skipped by calling `istream` member function `ignore`, which discards the specified number of characters in the input stream (one character by default).

11.5 Overloading Stream Insertion and Stream Extraction Operators (cont.)

- Function `operator>>` returns `istream` reference `input` (i.e., `cin`).
- This enables input operations on `PhoneNumber` objects to be cascaded with input operations on other `PhoneNumber` objects or on objects of other data types.

11.5 Overloading Stream Insertion and Stream Extraction Operators (cont.)

- The stream insertion operator function takes an `ostream` reference (`output`) and a `const PhoneNumber` reference (`number`) as arguments and returns an `os-tream` reference.
- Function `operator<<` displays objects of type `PhoneNumber`.
- When the compiler sees the expression
  - `cout << phone`

  it generates the global function call
  - `operator<<( cout, phone );`
- Function `operator<<` displays the parts of the telephone number as `string`s, because they're stored as `string` objects.

Error-Prevention Tip 11.1

Returning a reference from an overloaded `<<` *or* `>>` *operator function is typically successful because* `cout`, `cin` *and most stream objects are global, or at least long-lived. Returning a reference to an automatic variable or other temporary object is dangerous—this can create "dangling references" to nonexisting objects.*

11.5 Overloading Stream Insertion and Stream Extraction Operators (cont.)

- The functions `operator>>` and `operator<<` are declared in `PhoneNumber` as global, `friend` functions
  - global functions because the object of class `PhoneNumber` is the operator's right operand.

Software Engineering Observation 11.3

New input/output capabilities for user-defined types are added to C++ without modifying standard input/output library classes. This is another example of C++'s extensibility.

11.12 Overloading ++ and --

- The prefix and postfix versions of the increment and decrement operators can all be overloaded.
- To overload the increment operator to allow both prefix and postfix increment usage, each overloaded operator function must have a distinct signature, so that the compiler will be able to determine which version of ++ is intended.
- The prefix versions are overloaded exactly as any other prefix unary operator would be.

11.12 Overloading ++ and -- (cont.)

- Suppose, for example, that we want to add 1 to the day in `Date` object `d1`.
- When the compiler sees the preincrementing expression `++d1`, the compiler generates the member-function call
  - `d1.operator++()`
- The prototype for this operator function would be
  - `Date &operator++();`
- If the prefix increment operator is implemented as a global function, then, when the compiler sees the expression `++d1`, the compiler generates the function call
  - `operator++( d1 )`
- The prototype for this operator function would be declared in the `Date` class as
  - `Date &operator++( Date & );`

11.12 Overloading ++ and -- (cont.)

- Overloading the postfix increment operator presents a challenge, because the compiler must be able to distinguish between the signatures of the overloaded prefix and postfix increment operator functions.
- The convention that has been adopted in C++ is that, when the compiler sees the postincrementing expression `d1++`, it generates the member-function call
  - `d1.operator++( 0 )`
- The prototype for this function is
  - `Date operator++( int )`
- The argument `0` is strictly a "dummy value" that enables the compiler to distinguish between the prefix and postfix increment operator functions.
- The same syntax is used to differentiate between the prefix and postfix decrement operator functions.

11.12 Overloading ++ and -- (cont.)

- If the postfix increment is implemented as a global function, then, when the compiler sees the expression `d1++`, the compiler generates the function call
  - `operator++( d1, 0 )`
- The prototype for this function would be
  - `Date operator++( Date &, int );`
- Once again, the `0` argument is used by the compiler to distinguish between the prefix and postfix increment operators implemented as global functions.
- The postfix increment operator returns `Date` objects by value, whereas the prefix increment operator returns `Date` objects by reference, because the postfix increment operator typically returns a temporary object that contains the original value of the object before the increment occurred.

Performance Tip 11.2

The extra object that is created by the postfix increment (or decrement) operator can result in a significant performance problem—especially when the operator is used in a loop. For this reason, you should use the postfix increment (or decrement) operator only when the logic of the program requires postincrementing (or postdecrementing).

11.13 Case Study: A Date Class

- The program of Figs. 11.9–11.11 demonstrates a `Date` class, which uses overloaded prefix and postfix increment operators to add 1 to the day in a `Date` object, while causing appropriate increments to the month and year if necessary.

```
// Fig. 11.9: Date.h
// Date class definition with overloaded increment operators.
#ifndef DATE_H
#define DATE_H

#include <iostream>
using namespace std;

class Date
{
   friend ostream &operator<<( ostream &, const Date & );
public:
   Date( int m = 1, int d = 1, int y = 1900 ); // default constructor
   void setDate( int, int, int ); // set month, day, year
   Date &operator++(); // prefix increment operator
   Date operator++( int ); // postfix increment operator
   const Date &operator+=( int ); // add days, modify object
   static bool leapYear( int ); // is date in a leap year?
   bool endOfMonth( int ) const; // is date at the end of month?
private:
   int month;
   int day;
   int year;
```

Fig. 11.9 | Date class definition with overloaded increment operators. (Part 1 of 2.)

```

   static const int days[]; // array of days per month
   void helpIncrement(); // utility function for incrementing date
}; // end class Date

#endif
```

Fig. 11.9 | `Date` class definition with overloaded increment operators. (Part 2 of 2.)

```
// Fig. 11.10: Date.cpp
// Date class member- and friend-function definitions.
#include <iostream>
#include <string>
#include "Date.h"
using namespace std;

// initialize static member; one classwide copy
const int Date::days[] =
   { 0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };

// Date constructor
Date::Date( int m, int d, int y )
{
   setDate( m, d, y );
} // end Date constructor

// set month, day and year
void Date::setDate( int mm, int dd, int yy )
{
   month = ( mm >= 1 && mm <= 12 ) ? mm : 1;
   year = ( yy >= 1900 && yy <= 2100 ) ? yy : 1900;

```

Fig. 11.10 | `Date` class member- and `friend`-function definitions. (Part 1 of 5.)

```cpp
   // test for a leap year
   if ( month == 2 && leapYear( year ) )
      day = ( dd >= 1 && dd <= 29 ) ? dd : 1;
   else
      day = ( dd >= 1 && dd <= days[ month ] ) ? dd : 1;
} // end function setDate

// overloaded prefix increment operator
Date &Date::operator++()
{
   helpIncrement(); // increment date
   return *this; // reference return to create an lvalue
} // end function operator++

// overloaded postfix increment operator; note that the
// dummy integer parameter does not have a parameter name
Date Date::operator++( int )
{
   Date temp = *this; // hold current state of object
   helpIncrement();

   // return unincremented, saved, temporary object
   return temp; // value return; not a reference return
} // end function operator++
```

Fig. 11.10 | `Date` class member- and `friend`-function definitions. (Part 2 of 5.)

```
// add specified number of days to date
const Date &Date::operator+=( int additionalDays )
{
   for ( int i = 0; i < additionalDays; i++ )
      helpIncrement();

   return *this; // enables cascading
} // end function operator+=

// if the year is a leap year, return true; otherwise, return false
bool Date::leapYear( int testYear )
{
   if ( testYear % 400 == 0 ||
      ( testYear % 100 != 0 && testYear % 4 == 0 ) )
      return true; // a leap year
   else
      return false; // not a leap year
} // end function leapYear

```

Fig. 11.10 | `Date` class member- and `friend`-function definitions. (Part 3 of 5.)

```
// determine whether the day is the last day of the month
bool Date::endOfMonth( int testDay ) const
{
   if ( month == 2 && leapYear( year ) )
      return testDay == 29; // last day of Feb. in leap year
   else
      return testDay == days[ month ];
} // end function endOfMonth

// function to help increment the date
void Date::helpIncrement()
{
   // day is not end of month
   if ( !endOfMonth( day ) )
      day++; // increment day
   else
      if ( month < 12 ) // day is end of month and month < 12
      {
         month++; // increment month
         day = 1; // first day of new month
      } // end if
      else // last day of year
      {
```

Fig. 11.10 | `Date` class member- and `friend`-function definitions. (Part 4 of 5.)

```
            year++; // increment year
            month = 1; // first month of new year
            day = 1; // first day of new month
         } // end else
   } // end function helpIncrement

   // overloaded output operator
   ostream &operator<<( ostream &output, const Date &d )
   {
      static string monthName[ 13 ] = { "", "January", "February",
         "March", "April", "May", "June", "July", "August",
         "September", "October", "November", "December" };
      output << monthName[ d.month ] << ' ' << d.day << ", " << d.year;
      return output; // enables cascading
   } // end function operator<<
```

Fig. 11.10 | `Date` class member- and `friend`-function definitions. (Part 5 of 5.)

```cpp
// Fig. 11.11: fig11_11.cpp
// Date class test program.
#include <iostream>
#include "Date.h" // Date class definition
using namespace std;

int main()
{
   Date d1; // defaults to January 1, 1900
   Date d2( 12, 27, 1992 ); // December 27, 1992
   Date d3( 0, 99, 8045 ); // invalid date

   cout << "d1 is " << d1 << "\nd2 is " << d2 << "\nd3 is " << d3;
   cout << "\n\nd2 += 7 is " << ( d2 += 7 );

   d3.setDate( 2, 28, 1992 );
   cout << "\n\n  d3 is " << d3;
   cout << "\n++d3 is " << ++d3 << " (leap year allows 29th)";

   Date d4( 7, 13, 2002 );

   cout << "\n\nTesting the prefix increment operator:\n"
      << "  d4 is " << d4 << endl;
```

Fig. 11.11 | Date class test program. (Part 1 of 3.)

```
    cout << "++d4 is " << ++d4 << endl;
    cout << "  d4 is " << d4;

    cout << "\n\nTesting the postfix increment operator:\n"
       << "  d4 is " << d4 << endl;
    cout << "d4++ is " << d4++ << endl;
    cout << "  d4 is " << d4 << endl;
} // end main
```

Fig. 11.11 | Date class test program. (Part 2 of 3.)

```
d1 is January 1, 1900
d2 is December 27, 1992
d3 is January 1, 1900

d2 += 7 is January 3, 1993

  d3 is February 28, 1992
++d3 is February 29, 1992 (leap year allows 29th)

Testing the prefix increment operator:
  d4 is July 13, 2002
++d4 is July 14, 2002
  d4 is July 14, 2002

Testing the postfix increment operator:
  d4 is July 14, 2002
d4++ is July 14, 2002
  d4 is July 15, 2002
```

Fig. 11.11 | `Date` class test program. (Part 3 of 3.)

14.1 Templates Introduction

- Function templates and class templates enable you to specify, with a single code segment, an entire range of related (overloaded) functions—called function-template specializations—or an entire range of related classes—called class-template specializations.
- This technique is called generic programming.
- Note the distinction between templates and template specializations:
  - Function templates and class templates are like stencils out of which we trace shapes.
  - Function-template specializations and class-template specializations are like the separate trac-ings that all have the same shape, but could, for example, be drawn in different colors.
- In this chapter, we present a function template and a class tem-plate.

Software Engineering Observation 14.1

Most C++ compilers require the complete definition of a template to appear in the client source-code file that uses the template. For this reason and for reusability, templates are often defined in header files, which are then `#include`*d into the appropriate client source-code files. For class templates, this means that the member functions are also defined in the header file.*

14.2 Function Templates

- Overloaded functions normally perform *similar or identical operations on different types of data*.
- If the operations are *identical for each type*, *they can be expressed more com-pactly and conveniently using function templates*.
- Initially, you write a single function-template definition.
- Based on the argument types provided explicitly or inferred from calls to this function, the compiler generates separate source-code functions (i.e., function-template specializations) to handle each function call appropriately.

Error-Prevention Tip 14.1

Function templates, like macros, enable software reuse. Unlike macros, function templates help eliminate many types of errors through the scrutiny of full C++ type checking.

14.2 Function Templates (cont.)

- All function-template definitions begin with keyword `template` followed by a list of template parameters to the function template enclosed in angle brackets (< and >); each template parameter that represents a type must be preceded by either of the interchangeable keywords `class` or `typename`, as in
    - `template<typename T>`
  - Or
    - `template<class ElementType>`
  - Or
    - `template<typename BorderType, typename FillType>`
- The type template parameters of a function-template definition are used to specify the types of the arguments to the function, to specify the return type of the function and to declare variables within the function.
- Keywords `typename` and `class` used to specify function-template parameters actually mean “any fundamental type or user-defined type.”

Common Programming Error 14.1

Not placing keyword `class` *or keyword* `typename` *before each type template parameter of a function template is a syntax error.*

14.2 Function Templates (cont.)

- Let's examine function template `printArray` in Fig. 14.1, lines 7–14.
- Function template `printArray` declares (line 7) a single template parameter `T` (`T` can be any valid identifier) for the type of the array to be printed by function `print-Array`; `T` is referred to as a type template parameter, or type parameter.

```cpp
// Fig. 14.1: fig14_01.cpp
// Using template functions.
#include <iostream>
using namespace std;

// function template printArray definition
template< typename T >
void printArray( const T * const array, int count )
{
   for ( int i = 0; i < count; i++ )
      cout << array[ i ] << " ";

   cout << endl;
} // end function template printArray

int main()
{
   const int aCount = 5; // size of array a
   const int bCount = 7; // size of array b
   const int cCount = 6; // size of array c

```

Fig. 14.1 | Function-template specializations of function template `printArray`. (Part 1 of 3.)

```
    int a[ aCount ] = { 1, 2, 3, 4, 5 };
    double b[ bCount ] = { 1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7 };
    char c[ cCount ] = "HELLO"; // 6th position for null

    cout << "Array a contains:" << endl;

    // call integer function-template specialization
    printArray( a, aCount );

    cout << "Array b contains:" << endl;

    // call double function-template specialization
    printArray( b, bCount );

    cout << "Array c contains:" << endl;

    // call character function-template specialization
    printArray( c, cCount );
} // end main
```

Fig. 14.1 | Function-template specializations of function template `printArray`. (Part 2 of 3.)

```
Array a contains:
1 2 3 4 5
Array b contains:
1.1 2.2 3.3 4.4 5.5 6.6 7.7
Array c contains:
H E L L O
```

Fig. 14.1 | Function-template specializations of function template `printArray`. (Part 3 of 3.)

14.2 Function Templates (cont.)

- When the compiler detects a `printArray` function invocation in the client program (e.g., lines 29, 34 and 39), the compiler uses its overload resolution capabilities to find a definition of function `printArray` that best matches the function call.
- In this case, the only `printArray` function with the appropriate number of parameters is the `printArray` function template (lines 7–14).
- Consider the function call at line 29.
- The compiler compares the type of `printArray`’s first argument (`int *` at line 29) to the `printArray` function template’s first parameter (`const T * const` at line 8) and deduces that replacing the type parameter `T` with `int` would make the argument consistent with the parameter.
- Then, the compiler substitutes `int` for `T` throughout the template definition and compiles a `printArray` specialization that can display an array of `int` values.

14.2 Function Templates (cont.)

- The function-template specialization for type `int` is
  ```
  void printArray( const int * const array, int count )
  {
     for ( int i = 0; i < count; i++ )
        cout << array[ i ] << " ";

     cout << endl;
  } // end function printArray
  ```
- As with function parameters, the names of template parameters must be unique inside a template definition.
- Template parameter names need not be unique across different function templates.
- Figure 14.1 demonstrates function template `printArray`.
- It's important to note that if `T` (line 7) represents a user-defined type (which it does not in Fig. 14.1), there must be an overloaded stream insertion operator for that type; otherwise, the first stream insertion operator in line 11 will not compile.

Common Programming Error 14.2

If a template is invoked with a user-defined type, and if that template uses functions or operators (e.g., `==`, `+`, `<=`) with objects of that class type, then those functions and operators must be overloaded for the user-defined type. Forgetting to overload such operators causes compilation errors.

Performance Tip 14.1

Although templates offer software-reusability benefits, remember that multiple function-template specializations and class-template specializations are instantiated in a program (at compile time), despite the fact that the templates are written only once. These copies can consume considerable memory. This is not normally an issue, though, because the code generated by the template is the same size as the code you'd have written to produce the separate overloaded functions.

14.3 Overloading Function Templates

- Function templates and overloading are intimately related.
- The function-template specializations gener-ated from a function template all have the same name, so the compiler uses overloading resolution to invoke the proper function.
- A function template may be overloaded in several ways.
  - We can provide other function templates that specify the same function name but different function parameters.
  - We can provide nontemplate func-tions with the same function name but different function arguments.

Common Programming Error 14.3

A compilation error occurs if no matching function definition can be found for a particular function call or if there are multiple matches that the compiler considers ambiguous.

14.4 Class Templates

- It's possible to understand the concept of a "stack" (a data structure into which we insert items at the top and retrieve those items in last-in, first-out order) independent of the type of the items be-ing placed in the stack.
- However, to instantiate a stack, a data type must be specified.
- Wonderful opportunity for software reusability.
- We need the means for describing the notion of a stack generically and instanti-ating classes that are type-specific versions of this generic stack class.
- C++ provides this capability through class templates.

Software Engineering Observation 14.2

Class templates encourage software reusability by enabling type-specific versions of generic classes to be instantiated.

14.4 Class Templates (cont.)

- Class templates are called parameterized types, because they require one or more type parameters to specify how to customize a "generic class" template to form a class-template specialization.
  - Each time an additional class-template specialization is needed, you use a concise, simple notation, and the compiler writes the source code for the specialization you require.

14.4 Class Templates (cont.)

- Note the `Stack` class-template definition in Fig. 14.2.
- It looks like a conventional class definition, except that it's preceded by the header (line 6)
    - `template< typename T >`
- to specify a class-template definition with type parameter `T` which acts as a placeholder for the type of the `Stack` class to be created.
- The type of element to be stored on this `Stack` is men-tioned generically as `T` throughout the `Stack` class header and member-function definitions.
- Due to the way this class template is designed, there are two constraints for nonfundamental data types used with this `Stack`
    – they must have a default constructor
    – their assignment operators must properly copy objects into the `Stack`

```
// Fig. 14.2: Stack.h
// Stack class template.
#ifndef STACK_H
#define STACK_H

template< typename T >
class Stack
{
public:
   Stack( int = 10 ); // default constructor (Stack size 10)

   // destructor
   ~Stack()
   {
      delete [] stackPtr; // deallocate internal space for Stack
   } // end ~Stack destructor

   bool push( const T & ); // push an element onto the Stack
   bool pop( T & ); // pop an element off the Stack

```

Fig. 14.2 | Class template `Stack`. (Part 1 of 4.)

```
   // determine whether Stack is empty
   bool isEmpty() const
   {
      return top == -1;
   } // end function isEmpty

   // determine whether Stack is full
   bool isFull() const
   {
      return top == size - 1;
   } // end function isFull

private:
   int size; // # of elements in the Stack
   int top; // location of the top element (-1 means empty)
   T *stackPtr; // pointer to internal representation of the Stack
}; // end class template Stack

```

Fig. 14.2 | Class template `Stack`. (Part 2 of 4.)

```
// constructor template
template< typename T >
Stack< T >::Stack( int s )
   : size( s > 0 ? s : 10 ), // validate size
     top( -1 ), // Stack initially empty
     stackPtr( new T[ size ] ) // allocate memory for elements
{
   // empty body
} // end Stack constructor template

// push element onto Stack;
// if successful, return true; otherwise, return false
template< typename T >
bool Stack< T >::push( const T &pushValue )
{
   if ( !isFull() )
   {
      stackPtr[ ++top ] = pushValue; // place item on Stack
      return true; // push successful
   } // end if

   return false; // push unsuccessful
} // end function template push

```

Fig. 14.2 | Class template `Stack`. (Part 3 of 4.)

```
// pop element off Stack;
// if successful, return true; otherwise, return false
template< typename T >
bool Stack< T >::pop( T &popValue )
{
   if ( !isEmpty() )
   {
      popValue = stackPtr[ top-- ]; // remove item from Stack
      return true; // pop successful
   } // end if

   return false; // pop unsuccessful
} // end function template pop

#endif
```

Fig. 14.2 | Class template `Stack`. (Part 4 of 4.)

14.4 Class Templates (cont.)

- The member-function definitions of a class template are function templates.
- The member-function definitions that appear outside the class template definition each begin with the header
    - `template< typename T >`
- Thus, each definition resembles a conventional function definition, except that the `Stack` element type always is listed generically as type parameter `T`.
- The binary scope resolution operator is used with the class-template name to tie each member-function definition to the class template's scope.
- When `doubleStack` is instantiated as type `Stack<double>`, the `Stack` constructor function-template specialization uses `new` to create an array of elements of type `double` to represent the stack.

14.4 Class Templates (cont.)

- Now, let's consider the driver (Fig. 14.3) that exercises the `Stack` class template.
- The driver begins by instantiating object `doubleStack` of size `5`.
- This object is declared to be of class `Stack< double >` (pronounced "`Stack` of `double`").
- The compiler associates type `double` with type parameter `T` in the class template to produce the source code for a `Stack` class of type `double`.
- Although templates offer software-reusability benefits, remember that mul-tiple class-template specializations are instantiated in a program (at compile time), even though the template is written only once.

```cpp
// Fig. 14.3: fig14_03.cpp
// Stack class template test program.
#include <iostream>
#include "Stack.h" // Stack class template definition
using namespace std;

int main()
{
   Stack< double > doubleStack( 5 ); // size 5
   double doubleValue = 1.1;

   cout << "Pushing elements onto doubleStack\n";

   // push 5 doubles onto doubleStack
   while ( doubleStack.push( doubleValue ) )
   {
      cout << doubleValue << ' ';
      doubleValue += 1.1;
   } // end while

   cout << "\nStack is full. Cannot push " << doubleValue
      << "\n\nPopping elements from doubleStack\n";

```

Fig. 14.3 | Class template Stack `test` program. (Part 1 of 3.)

```
   // pop elements from doubleStack
   while ( doubleStack.pop( doubleValue ) )
      cout << doubleValue << ' ';

   cout << "\nStack is empty. Cannot pop\n";

   Stack< int > intStack; // default size 10
   int intValue = 1;
   cout << "\nPushing elements onto intStack\n";

   // push 10 integers onto intStack
   while ( intStack.push( intValue ) )
   {
      cout << intValue++ << ' ';
   } // end while

   cout << "\nStack is full. Cannot push " << intValue
      << "\n\nPopping elements from intStack\n";

   // pop elements from intStack
   while ( intStack.pop( intValue ) )
      cout << intValue << ' ';

   cout << "\nStack is empty. Cannot pop" << endl;
} // end main
```

Fig. 14.3 | Class template Stack `test` program. (Part 2 of 3.)

```
Pushing elements onto doubleStack
1.1 2.2 3.3 4.4 5.5
Stack is full. Cannot push 6.6

Popping elements from doubleStack
5.5 4.4 3.3 2.2 1.1
Stack is empty. Cannot pop

Pushing elements onto intStack
1 2 3 4 5 6 7 8 9 10
Stack is full. Cannot push 11

Popping elements from intStack
10 9 8 7 6 5 4 3 2 1
Stack is empty. Cannot pop
```

Fig. 14.3 | Class template Stack `test` program. (Part 3 of 3.)

14.4 Class Templates (cont.)

- Line 30 instantiates integer stack `intStack` with the declaration
    - `Stack< int > intStack;`
- Because no size is specified, the size defaults to `10` as specified in the default constructor (Fig. 14.2, line 10).

14.4 Class Templates (cont.)

- Notice that the code in function `main` of Fig. 14.3 is almost identical for both the `double-Stack` manipulations in lines 9–28 and the `intStack` manipulations in lines 30–47.
- This presents another opportunity to use a function template.
- Figure 14.4 defines function template `testStack` (lines 10–34) to perform the same tasks as `main` in Fig. 14.3—`push` a series of values onto a `Stack< T >` and `pop` the values off a `Stack< T >`.
- Function template `testStack` uses template parameter `T` (specified at line 10) to represent the data type stored in the `Stack< T >`.
- The function template takes four arguments (lines 12–15)—a reference to an object of type `Stack< T >`, a value of type `T` that will be the first value `push`ed onto the `Stack< T >`, a value of type `T` used to increment the values `push`ed onto the `Stack< T >` and a `string` that represents the name of the `Stack< T >` object for output purposes.
- The output of Fig. 14.4 precisely matches the output of Fig. 14.3.

```
// Fig. 14.4: fig14_04.cpp
// Stack class template test program. Function main uses a
// function template to manipulate objects of type Stack< T >.
#include <iostream>
#include <string>
#include "Stack.h" // Stack class template definition
using namespace std;

// function template to manipulate Stack< T >
template< typename T >
void testStack(
   Stack< T > &theStack, // reference to Stack< T >
   T value, // initial value to push
   T increment, // increment for subsequent values
   const string stackName ) // name of the Stack< T > object
{
   cout << "\nPushing elements onto " << stackName << '\n';

   // push element onto Stack
   while ( theStack.push( value ) )
   {
      cout << value << ' ';
      value += increment;
   } // end while
```

Fig. 14.4 | Passing a `Stack` template object to a function template. (Part 1 of 3.)

```
   cout << "\nStack is full. Cannot push " << value
      << "\n\nPopping elements from " << stackName << '\n';

   // pop elements from Stack
   while ( theStack.pop( value ) )
      cout << value << ' ';

   cout << "\nStack is empty. Cannot pop" << endl;
} // end function template testStack

int main()
{
   Stack< double > doubleStack( 5 ); // size 5
   Stack< int > intStack; // default size 10

   testStack( doubleStack, 1.1, 1.1, "doubleStack" );
   testStack( intStack, 1, 1, "intStack" );
} // end main
```

Fig. 14.4 | Passing a `Stack` template object to a function template. (Part 2 of 3.)

```
Pushing elements onto doubleStack
1.1 2.2 3.3 4.4 5.5
Stack is full. Cannot push 6.6

Popping elements from doubleStack
5.5 4.4 3.3 2.2 1.1
Stack is empty. Cannot pop

Pushing elements onto intStack
1 2 3 4 5 6 7 8 9 10
Stack is full. Cannot push 11

Popping elements from intStack
10 9 8 7 6 5 4 3 2 1
Stack is empty. Cannot pop
```

Fig. 14.4 | Passing a `Stack` template object to a function template. (Part 3 of 3.)

Questions

?