W8.2 Operator Overloading

- **Fundamentals of Operator Overloading**
- **Restrictions on Operator Overloading**
- **Operator Functions as Class Members vs. as friend Functions**
- **Overloading Stream Insertion and Extraction Operators**
- **Overloading Unary Operators**
- **Overloading Binary Operators**
- **Case Study: An Array Class**
- **Converting between Types**
- **Case Study: A `String` Class**
- **Overloading ++ and --**
- **Case Study: A `Date` Class**

Introduction

- Operator overloading
  - Enabling C++'s operators to work with class objects
  - Using traditional operators with user-defined objects
  - Requires great care; when overloading is misused, program difficult to understand
  - Examples of already overloaded operators
    - Operator **`<<`** is both the stream-insertion operator and the bitwise left-shift operator
    - **`+`** and **`-`**, perform arithmetic on multiple types
  - Compiler generates the appropriate code based on the manner in which the operator is used

Fundamentals of Operator Overloading

- Overloading an operator
  - Write function definition as normal
  - Function name is keyword **`operator`** followed by the symbol for the operator being overloaded
  - **`operator+`** used to overload the addition operator (**`+`**)
- Using operators
  - To use an operator on a class object it must be overloaded unless the assignment operator(**`=`**)or the address operator (**`&`**)
    - Assignment operator by default performs memberwise assignment
    - Address operator (&) by default returns the address of an object

Restrictions on Operator Overloading

- C++ operators that can be overloaded

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

- C++ Operators that cannot be overloaded

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

Restrictions on Operator Overloading

- Overloading restrictions
  - Precedence of an operator cannot be changed
  - Associativity of an operator cannot be changed
  - Arity (number of operands) cannot be changed
    - Unary operators remain unary, and binary operators remain binary
    - Operators **`&`**, **`*`**, **`+`** and **`-`** each have unary and binary versions
    - Unary and binary versions can be overloaded separately
- No new operators can be created
  - Use only existing operators
- No overloading operators for built-in types
  - Cannot change how two integers are added
  - Produces a syntax error

Operator Functions as Class Members vs. as friend Functions

- Member vs non-member
  - Operator functions can be member or non-member functions
  - When overloading **`( )`**, **`[ ]`**, **`->`** or any of the assignment operators, must use a member function
- Operator functions as member functions
  - Leftmost operand must be an object (or reference to an object) of the class
    - If left operand of a different type, operator function must be a non-member function
- Operator functions as non-member functions
  - Must be **`friend`**s if needs to access private or protected members
  - Enable the operator to be commutative (a+b, b+a)

Overloading Stream-Insertion and Stream-Extraction Operators

- Overloaded **<<** and **>>** operators
  - Overloaded to perform input/output for user-defined types
  - Left operand of types `ostream &` and `istream &`
  - Must be a non-member function because left operand is not an object of the class
  - Must be a `friend` function to access private data members

```cpp
// Fig. 8.3: fig08_03.cpp
// Overloading the stream-insertion and
// stream-extraction operators.
#include <iostream>

using std::cout;
using std::cin;
using std::endl;
using std::ostream;
using std::istream;

#include <iomanip>

using std::setw;

class PhoneNumber {
   friend ostream &operator<<( ostream&, const PhoneNumber & );
   friend istream &operator>>( istream&, PhoneNumber & );

private:
   char areaCode[ 4 ];  // 3-digit area code and null
   char exchange[ 4 ];  // 3-digit exchange and null
   char line[ 5 ];      // 4-digit line and null
};

// 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 &num )
{
```

Notice function prototypes for overloaded operators **>>** and **<<**

They must be `friend` functions.

Outline

1. Class definition

1.1 Function definitions

```cpp
   output << "(" << num.areaCode << ") "
          << num.exchange << "-" << num.line;
   return output;     // enables cout << a << b << c;
}

istream &operator>>( istream &input, PhoneNumber &num )
{
   input.ignore();                        // skip (
   input >> setw( 4 ) >> num.areaCode; // input area code
   input.ignore( 2 );                     // skip ) and space
   input >> setw( 4 ) >> num.exchange; // input ex
   input.ignore();                        // skip das
   input >> setw( 5 ) >> num.line;     // input lin
   return input;      // enables cin >> a >> b >> c
}

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

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

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

   // cout << phone invokes operator<< function by
   // issuing the call operator<<( cout, phone ).
   cout << "The phone number entered was: " << phone << endl;
   return 0;
}
```

The function call

`cin >> phone;`

interpreted as

`operator>>(cin, phone);`

`input` is an alias for `cin`, and `num` is an alias for `phone`.

Outline

1.1 Function definition

1.2 Initialize variables

2. Get input

ject

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

Outline

Program Output

Overloading Unary Operators

- Overloading unary operators
  - Can be overloaded with no arguments or one argument
  - Should usually be implemented as member functions
    - Avoid `friend` functions and classes because they violate the encapsulation of a class
  - Example declaration as a member function:

```cpp
class String {
public:
   bool operator!() const;
   ...
};
```

Overloading Unary Operators

- Example declaration as a non-member function

```cpp
class String {
   friend bool operator!( const String & )
   ...
}
```

Overloading Binary Operators

- Overloaded Binary operators
  - Non-static member function, one argument
  - Example:

```
class String {
public:
   const String &operator+=(
                  const String & );
   ...
};
```

  - `y += z` is equivalent to `y.operator+=( z )`

Overloading Binary Operators

- Non-member function, two arguments
- Example:

```
class String {
   friend const String &operator+=(
                 String &, const String & );
   ...
};
```

- `y += z` is equivalent to `operator+=( y, z )`

Case Study: An Array class

- Implement an `Array` class with
  - Range checking
  - Array assignment
  - Arrays that know their size
  - Outputting/inputting entire arrays with `<<` and `>>`
  - Array comparisons with `= =` and `!=`

```
// Fig. 8.4: array1.h
// Simple class Array (for integers)
#ifndef ARRAY1_H
#define ARRAY1_H

#include <iostream>

using std::ostream;
using std::istream;

class Array {
   friend ostream &operator<<( ostream &, const Array & );
   friend istream &operator>>( istream &, Array & );
public:
   Array( int = 10 );                   // default const
   Array( const Array & );              // copy construc
   ~Array();                            // destructor
   int getSize() const;                 // return size
   const Array &operator=( const Array & ); // assign arrays
   bool operator==( const Array & ) const;  // compare equal

   // Determine if two arrays are not equal and
   // return true, otherwise return false (uses operator==).
   bool operator!=( const Array &right ) const
      { return ! ( *this == right ); }

   int &operator[]( int );              // subscript operator
   const int &operator[]( int ) const;  // subscript operator
   static int getArrayCount();          // Return count of
                                        // arrays instantiated.
private:
   int size; // size of the array
   int *ptr; // pointer to first element of array
   static int arrayCount;  // # of Arrays instantiated
```

Notice all the overloaded operators used to implement the class.

Outline

1. Class definition

1.1 Function prototypes

```
};

#endif
// Fig 8.4: array1.cpp
// Member function definitions for class Array
#include <iostream>

using std::cout;
using std::cin;
using std::endl;

#include <iomanip>

using std::setw;

#include <cstdlib>
#include <cassert>
#include "array1.h"

// Initialize static data member at file scope
int Array::arrayCount = 0;   // no objects yet

// Default constructor for class Array (default size 10)
Array::Array( int arraySize )
{
   size = ( arraySize > 0 ? arraySize : 10 );
   ptr = new int[ size ]; // create space for array
   assert( ptr != 0 );    // terminate if memory not allocated
   ++arrayCount;          // count one more object

   for ( int i = 0; i < size; i++ )
      ptr[ i ] = 0;          // initialize array
```

Outline

1. Load header

1.1 Function definitions

1.2 `Array` constructor

```
}

// Copy constructor for class Array
// must receive a reference to prevent infinite recursion
Array::Array( const Array &init ) : size( init.size )
{
   ptr = new int[ size ]; // create space for array
   assert( ptr != 0 );    // terminate if memory not allocated
   ++arrayCount;          // count one more object

   for ( int i = 0; i < size; i++ )
      ptr[ i ] = init.ptr[ i ];  // copy init into object
}

// Destructor for class Array
Array::~Array()
{
   delete [] ptr;            // reclaim space for array
   --arrayCount;             // one fewer object
}

// Get the size of the array
int Array::getSize() const { return size; }

// Overloaded assignment operator
// const return avoids: ( a1 = a2 ) = a3
const Array &Array::operator=( const Array &right )
{
   if ( &right != this ) {  // check for self-assignment

      // for arrays of different sizes, deallocate original
      // left side array, then allocate new left side array.
      if ( size != right.size ) {
         delete [] ptr;         // reclaim space
```

Outline

1.3 `Array` destructor

1.4 `operator=` (assignment)

```
      size = right.size;     // resize this object
      ptr = new int[ size ]; // create space for array copy
      assert( ptr != 0 );    // terminate if not allocated
   }

   for ( int i = 0; i < size; i++ )
      ptr[ i ] = right.ptr[ i ];  // copy array into object
 }

 return *this;   // enables x = y = z;
}

// Determine if two arrays are equal and
// return true, otherwise return false.
bool Array::operator==( const Array &right ) const
{
   if ( size != right.size )
      return false;    // arrays of different sizes

   for ( int i = 0; i < size; i++ )
      if ( ptr[ i ] != right.ptr[ i ] )
         return false; // arrays are not equal

   return true;         // arrays are equal
}

// Overloaded subscript operator for non-const Arrays
// reference return creates an lvalue
int &Array::operator[]( int subscript )
{
   // check for subscript out of range error
   assert( 0 <= subscript && subscript < size );
```

Outline

1.5 operator==

(equality)

1.6 operator[]

(subscript for non-const arrays)

```

   return ptr[ subscript ]; // reference return
}

// Overloaded subscript operator for const Arrays
// const reference return creates an rvalue
const int &Array::operator[]( int subscript ) const
{
   // check for subscript out of range error
   assert( 0 <= subscript && subscript < size );

   return ptr[ subscript ]; // const reference return
}

// Return the number of Array objects instantiated
// static functions cannot be const
int Array::getArrayCount() { return arrayCount; }

// Overloaded input operator for class Array;
// inputs values for entire array.
istream &operator>>( istream &input, Array &a )
{
   for ( int i = 0; i < a.size; i++ )
      input >> a.ptr[ i ];

   return input;   // enables cin >> x >> y;
}

// Overloaded output operator for class Array
ostream &operator<<( ostream &output, const Array &a )
{
```

Outline

1.6 operator[]

(subscript for const arrays)

1.7 getArrayCount

1.8 operator>>

(input array)

1.9 operator<<

(output array)

```
   int i;

   for ( i = 0; i < a.size; i++ ) {
      output << setw( 12 ) << a.ptr[ i ];

      if ( ( i + 1 ) % 4 == 0 ) // 4 numbers per row of output
         output << endl;
   }

   if ( i % 4 != 0 )
      output << endl;

   return output;   // enables cout << x << y;
}
// Fig. 8.4: fig08_04.cpp
// Driver for simple class Array
#include <iostream>

using std::cout;
using std::cin;
using std::endl;

#include "array1.h"

int main()
{
   // no objects yet
   cout << "# of arrays instantiated = "
        << Array::getArrayCount() << '\n';

```

```
# of arrays instantiated = 0
```

Outline

1. Load header

```
   // create two arrays and print Array count
   Array integers1( 7 ), integers2;
   cout << "# of arrays instantiated = "
        << Array::getArrayCount() << "\n\n";

   // print integers1 size and contents
   cout << "Size of array integers1 is "
        << integers1.getSize()
        << "\nArray after initialization:\n"
        << integers1 << '\n';

   // print integers2 size and contents
   cout << "Size of array integers2 is "
        << integers2.getSize()
        << "\nArray after initialization:\n"
        << integers2 << '\n';

   // input and print integers1 and integers2
   cout << "Input 17 integers:\n";
   cin >> integers1 >> integers2;
   cout << "After input, the arrays contain:\n"
        << "integers1:\n" << integers1
        << "integers2:\n" << integers2 << '\n';

   // use overloaded inequality (!=) operator
   cout << "Evaluating: integers1 != integers2\n";
   if ( integers1 != integers2 )
      cout << "They are not equal\n";

   // create array integers3 using integers1 as an
   // initializer; print size and contents
   Array integers3( integers1 );

```

```
# of arrays instantiated = 2
```

```
Size of array integers1 is 7
Array after initialization:
           0           0           0           0
           0           0           0
```

```
Size of array integers2 is 10
Array after initialization:
           0           0           0           0
           0           0           0           0
           0           0
```

```
Input 17 integers:
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17
After input, the arrays contain:
integers1:
           1           2           3           4
           5           6           7
integers2:
           8           9          10          11
          12          13          14          15
          16          17
```

```
Evaluating: integers1 != integers2
They are not equal
```

Outline

1.1 Initialize objects

```
   cout << "\nSize of array integers3 is "
        << integers3.getSize()
        << "\nArray after initialization:\n"
        << integers3 << '\n';

   // use overloaded assignment (
   cout << "Assigning integers2 to
   integers1 = integers2;
   cout << "integers1:\n" << integers1
        << "integers2:\n" << integers

   // use overloaded equality (==) ope
   cout << "Evaluating: integers1 ==
   if ( integers1 == integers2 )
      cout << "They are equal\n\n";

   // use overloaded subscript operat
   cout << "integers1[5] is " << inte

   // use overloaded subscript operat
   cout << "Assigning 1000 to integers1
   integers1[ 5 ] = 1000;
   cout << "integers1:\n" << integer

   // attempt to use out of range su
   cout << "Attempt to assign 1000 t
   integers1[ 15 ] = 1000;  // ERROR

   return 0;
}
```

```
Size of array integers3 is 7
Array after initialization:
           1           2           3           4
           5           6           7
```

```
Assigning integers2 to integers1:
integers1:
           8
          12
          16
integers2:
           8           9          10          11
```

```
Evaluating: integers1 == integers2
They are equal
integers1[5] is 13
```

```
Attempt to assign 1000 to integers1[15]
Assertion failed: 0 <= subscript && subscript < size, file Array1.cpp, line 95 abnormal program termination
```

```
Assigning 1000 to integers1[5]
integers1:
           8           9          10          11
          12        1000          14          15
          16          17
```

Outline

```
# of arrays instantiated = 0
# of arrays instantiated = 2

Size of array integers1 is 7
Array after initialization:
           0           0           0           0
           0           0           0

Size of array integers2 is 10
Array after initialization:
           0           0           0           0
           0           0           0           0
           0           0

Input 17 integers:
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17
After input, the arrays contain:
integers1:
           1           2           3           4
           5           6           7
integers2:
           8           9          10          11
          12          13          14          15
          16          17

Evaluating: integers1 != integers2
They are not equal

Size of array integers3 is 7
Array after initialization:
           1           2           3           4
           5           6           7
```

Outline

Program Output

```
Assigning integers2 to integers1:
integers1:
           8           9          10          11
          12          13          14          15
          16          17
integers2:
           8           9          10          11
          12          13          14          15
          16          17

Evaluating: integers1 == integers2
They are equal

integers1[5] is 13
Assigning 1000 to integers1[5]
integers1:
           8           9          10          11
          12        1000          14          15
          16          17

Attempt to assign 1000 to integers1[15]
Assertion failed: 0 <= subscript && subscript < size, file Array1.cpp,
line 95 abnormal program termination
```

Outline

Program Output

Converting between Types

- Cast operator
  - Forces conversions among built-in types
  - Specifies conversions between user defined and built-in types
  - Conversion operator must be a non-**static** member function
  - Cannot be a **friend** function
  - Do not specify return type
    - Return type is the type to which the object is being converted
  - For user-defined class **A**

    A::operator char *() const;

    - Declares an overloaded cast operator function for creating a **char *** out of an **A** object

Converting between Types

A::operator int() const;

- Declares an overloaded cast operator function for converting an object of **A** into an integer

A::operator otherClass() const;

- Declares an overloaded cast operator function for converting an object of **A** into an object of **otherClass**

- Compiler and casting
  - Casting can prevent the need for overloading
  - If an object s of user-defined class String appears in a program where an ordinary **char *** is expected, such as

    cout << s;

    The compiler calls the overloaded cast operator function **operator char *** to convert the object into a **char *** and uses the resulting **char *** in the expression

Case Study: A String Class

- Build a class to handle strings
  - Class **string** in standard library (more Chapter 19)
- Conversion constructor
  - Single-argument constructors that turn objects of other types into class objects

```
// Fig. 8.5: string1.h
// Definition of a String class
#ifndef  STRING1_H
#define  STRING1_H

#include <iostream>

using std::ostream;
using std::istream;

class String {
   friend ostream &operator<<( ostream &, const String & );
   friend istream &operator>>( istream &, String & );

public:
   String( const char * = "" ); // conversion/default ctor
   String( const String & );    // copy constructor
   ~String();                   // destructor
   const String &operator=( const String & );  // assignment
   const String &operator+=( const String & ); // concatenation
   bool operator!() const;                    // is String empty?
   bool operator==( const String & ) const; // test s1 == s2
   bool operator<( const String & ) const;  // test s1 < s2

   // test s1 != s2
   bool operator!=( const String & right ) const
      { return !( *this == right ); }

   // test s1 > s2
   bool operator>( const String &right ) const
      { return right < *this; }

   // test s1 <= s2
```

Outline

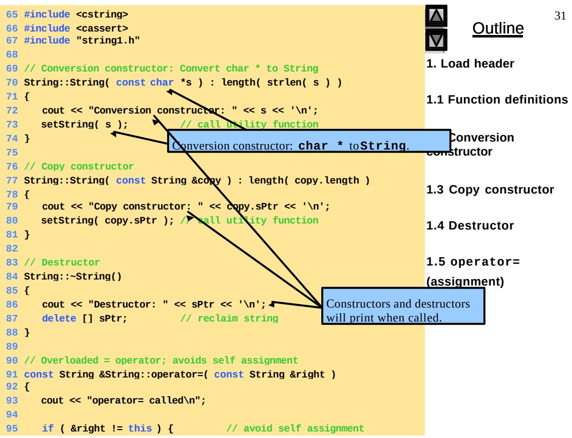
1. Class definition

1.1 Member functions, some definitions

```
   bool operator<=( const String &right ) const
      { return !( right < *this ); }

   // test s1 >= s2
   bool operator>=( const String &right ) const
      { return !( *this < right ); }

   char &operator[]( int );              // subscript operator
   const char &operator[]( int ) const; // subscript operator
   String operator()( int, int );       // return a substring
   int getLength() const;               // return string length

private:
   int length;                   // string length
   char *sPtr;                   // pointer to start of string

   void setString( const char * );  // utility function
};

#endif
// Fig. 8.5: string1.cpp
// Member function definitions for class String
#include <iostream>

using std::cout;
using std::endl;

#include <iomanip>

using std::setw;

```

Outline

1.2 Member variables



```cpp
#include <cstring>
#include <cassert>
#include "string1.h"

// Conversion constructor: Convert char * to String
String::String( const char *s ) : length( strlen( s ) )
{
   cout << "Conversion constructor: " << s << '\n';
   setString( s );         // call utility function
}

// Copy constructor
String::String( const String &copy ) : length( copy.length )
{
   cout << "Copy constructor: " << copy.sPtr << '\n';
   setString( copy.sPtr ); // call utility function
}

// Destructor
String::~String()
{
   cout << "Destructor: " << sPtr << '\n';
   delete [] sPtr;         // reclaim string
}

// Overloaded = operator; avoids self assignment
const String &String::operator=( const String &right )
{
   cout << "operator= called\n";

   if ( &right != this ) {         // avoid self assignment
```

Conversion constructor: **char *** to **String**.

Constructors and destructors will print when called.

31

Outline

1. Load header

1.1 Function definitions

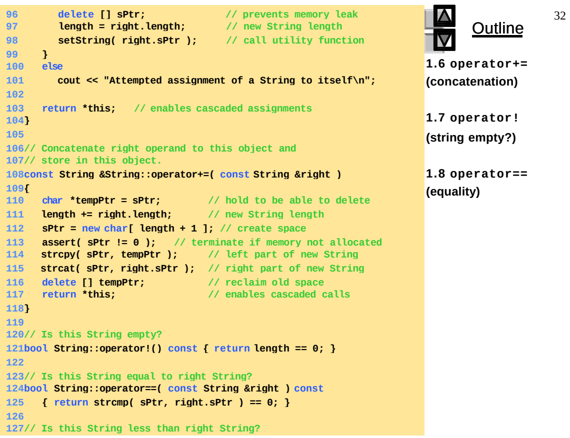
1.2 Conversion constructor

1.3 Copy constructor

1.4 Destructor

1.5 operator=

(assignment)



```cpp
      delete [] sPtr;              // prevents memory leak
      length = right.length;       // new String length
      setString( right.sPtr );     // call utility function
   }
   else
      cout << "Attempted assignment of a String to itself\n";

   return *this;   // enables cascaded assignments
}

// Concatenate right operand to this object and
// store in this object.
const String &String::operator+=( const String &right )
{
   char *tempPtr = sPtr;        // hold to be able to delete
   length += right.length;      // new String length
   sPtr = new char[ length + 1 ]; // create space
   assert( sPtr != 0 );   // terminate if memory not allocated
   strcpy( sPtr, tempPtr );     // left part of new String
   strcat( sPtr, right.sPtr );  // right part of new String
   delete [] tempPtr;           // reclaim old space
   return *this;                // enables cascaded calls
}

// Is this String empty?
bool String::operator!() const { return length == 0; }

// Is this String equal to right String?
bool String::operator==( const String &right ) const
   { return strcmp( sPtr, right.sPtr ) == 0; }

// Is this String less than right String?
```

32

Outline

1.6 operator+=

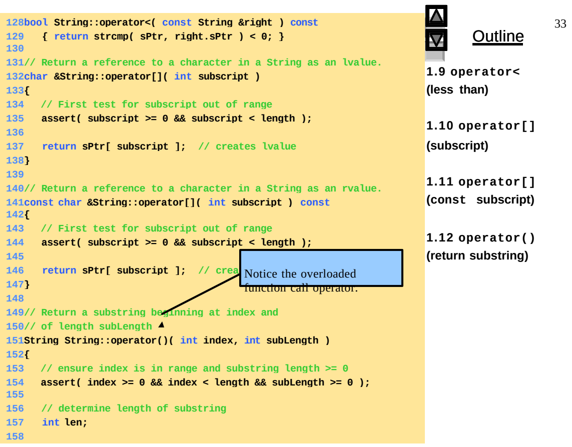
(concatenation)

1.7 operator!

(string empty?)

1.8 operator==

(equality)



```cpp
bool String::operator<( const String &right ) const
   { return strcmp( sPtr, right.sPtr ) < 0; }

// Return a reference to a character in a String as an lvalue.
char &String::operator[]( int subscript )
{
   // First test for subscript out of range
   assert( subscript >= 0 && subscript < length );

   return sPtr[ subscript ];  // creates lvalue
}

// Return a reference to a character in a String as an rvalue.
const char &String::operator[]( int subscript ) const
{
   // First test for subscript out of range
   assert( subscript >= 0 && subscript < length );

   return sPtr[ subscript ];  // crea
}

// Return a substring beginning at index and
// of length subLength
String String::operator()( int index, int subLength )
{
   // ensure index is in range and substring length >= 0
   assert( index >= 0 && index < length && subLength >= 0 );

   // determine length of substring
   int len;

```

Notice the overloaded function call operator.

33

Outline

1.9 operator<

(less than)

1.10 operator[]

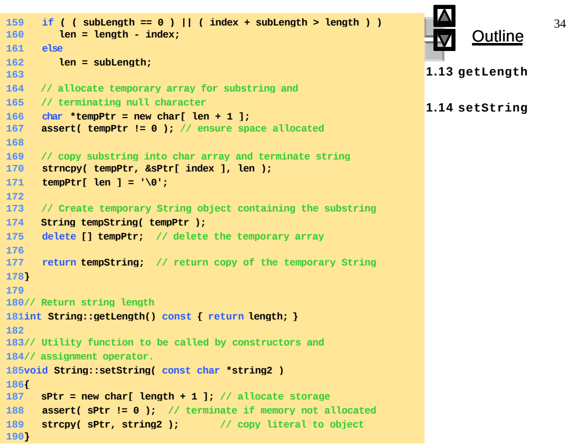
(subscript)

1.11 operator[]

(const subscript)

1.12 operator()

(return substring)



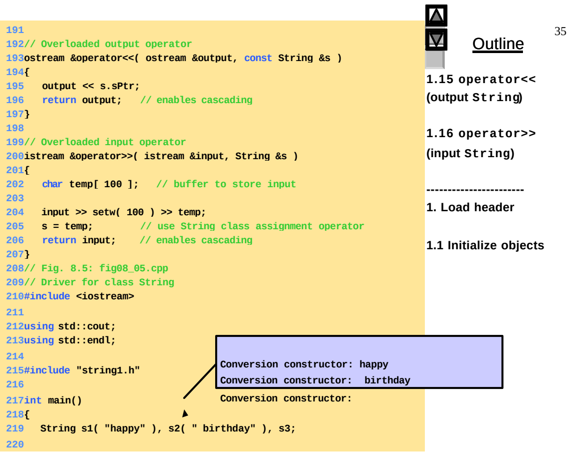
```cpp
   if ( ( subLength == 0 ) || ( index + subLength > length ) )
      len = length - index;
   else
      len = subLength;

   // allocate temporary array for substring and
   // terminating null character
   char *tempPtr = new char[ len + 1 ];
   assert( tempPtr != 0 ); // ensure space allocated

   // copy substring into char array and terminate string
   strncpy( tempPtr, &sPtr[ index ], len );
   tempPtr[ len ] = '\0';

   // Create temporary String object containing the substring
   String tempString( tempPtr );
   delete [] tempPtr;  // delete the temporary array

   return tempString;  // return copy of the temporary String
}

// Return string length
int String::getLength() const { return length; }

// Utility function to be called by constructors and
// assignment operator.
void String::setString( const char *string2 )
{
   sPtr = new char[ length + 1 ]; // allocate storage
   assert( sPtr != 0 );  // terminate if memory not allocated
   strcpy( sPtr, string2 );       // copy literal to object
}
```

34

Outline

1.13 getLength

1.14 setString



```cpp

// Overloaded output operator
ostream &operator<<( ostream &output, const String &s )
{
   output << s.sPtr;
   return output;   // enables cascading
}

// Overloaded input operator
istream &operator>>( istream &input, String &s )
{
   char temp[ 100 ];   // buffer to store input

   input >> setw( 100 ) >> temp;
   s = temp;         // use String class assignment operator
   return input;     // enables cascading
}
// Fig. 8.5: fig08_05.cpp
// Driver for class String
#include <iostream>

using std::cout;
using std::endl;

#include "string1.h"

int main()
{
   String s1( "happy" ), s2( " birthday" ), s3;

```

```
Conversion constructor: happy
Conversion constructor:  birthday
Conversion constructor:
```

35

Outline

1.15 operator<<

(output String)

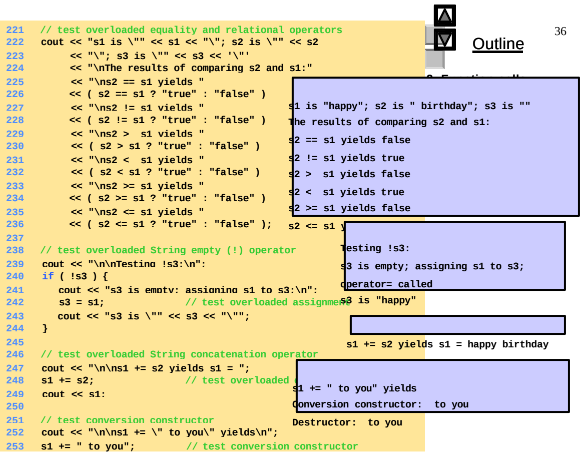
1.16 operator>>

(input String)

----

1. Load header

1.1 Initialize objects



```cpp
   // test overloaded equality and relational operators
   cout << "s1 is \"" << s1 << "\"; s2 is \"" << s2
        << "\"; s3 is \"" << s3 << '\"'
        << "\nThe results of comparing s2 and s1:"
        << "\ns2 == s1 yields "
        << ( s2 == s1 ? "true" : "false" )
        << "\ns2 != s1 yields "
        << ( s2 != s1 ? "true" : "false" )
        << "\ns2 >  s1 yields "
        << ( s2 > s1 ? "true" : "false" )
        << "\ns2 <  s1 yields "
        << ( s2 < s1 ? "true" : "false" )
        << "\ns2 >= s1 yields "
        << ( s2 >= s1 ? "true" : "false" )
        << "\ns2 <= s1 yields "
        << ( s2 <= s1 ? "true" : "false" );

   // test overloaded String empty (!) operator
   cout << "\n\nTesting !s3:\n";
   if ( !s3 ) {
      cout << "s3 is empty; assigning s1 to s3:\n";
      s3 = s1;              // test overloaded assignment
      cout << "s3 is \"" << s3 << "\"";
   }

   // test overloaded String concatenation operator
   cout << "\n\ns1 += s2 yields s1 = ";
   s1 += s2;              // test overloaded
   cout << s1;

   // test conversion constructor
   cout << "\n\ns1 += \" to you\" yields\n";
   s1 += " to you";        // test conversion constructor
```

```
s1 is "happy"; s2 is " birthday"; s3 is ""
The results of comparing s2 and s1:
s2 == s1 yields false
s2 != s1 yields true
s2 >  s1 yields false
s2 <  s1 yields true
s2 >= s1 yields false
s2 <= s1 y
```

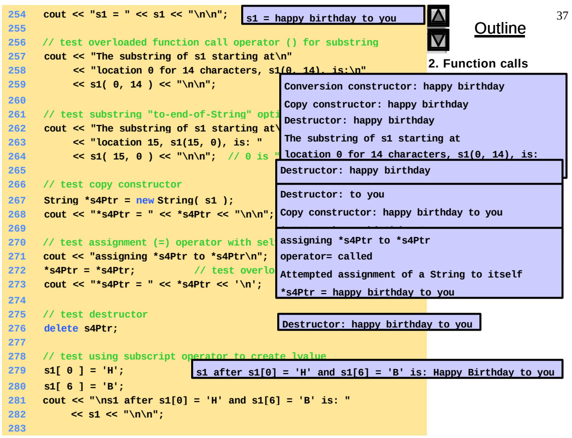
```
Testing !s3:
s3 is empty; assigning s1 to s3;
operator= called
s3 is "happy"
```

```
s1 += s2 yields s1 = happy birthday
```

```
s1 += " to you" yields
Conversion constructor:  to you
Destructor:  to you
```

36

Outline



```
   cout << "s1 = " << s1 << "\n\n";

   // test overloaded function call operator () for substring
   cout << "The substring of s1 starting at\n"
        << "location 0 for 14 characters, s1(0, 14), is:\n"
        << s1( 0, 14 ) << "\n\n";

   // test substring "to-end-of-String" opti
   cout << "The substring of s1 starting at\
        << "location 15, s1(15, 0), is: "
        << s1( 15, 0 ) << "\n\n";  // 0 is "

   // test copy constructor
   String *s4Ptr = new String( s1 );
   cout << "*s4Ptr = " << *s4Ptr << "\n\n";

   // test assignment (=) operator with sel
   cout << "assigning *s4Ptr to *s4Ptr\n";
   *s4Ptr = *s4Ptr;          // test overlo
   cout << "*s4Ptr = " << *s4Ptr << '\n';

   // test destructor
   delete s4Ptr;

   // test using subscript operator to create lvalue
   s1[ 0 ] = 'H';
   s1[ 6 ] = 'B';
   cout << "\ns1 after s1[0] = 'H' and s1[6] = 'B' is: "
       << s1 << "\n\n";

```

```
s1 = happy birthday to you
```

```
Conversion constructor: happy birthday
Copy constructor: happy birthday
Destructor: happy birthday
The substring of s1 starting at
location 0 for 14 characters, s1(0, 14), is:
```

```
Destructor: happy birthday
```

```
Destructor: to you
Copy constructor: happy birthday to you
```

```
assigning *s4Ptr to *s4Ptr
operator= called
Attempted assignment of a String to itself
*s4Ptr = happy birthday to you
```

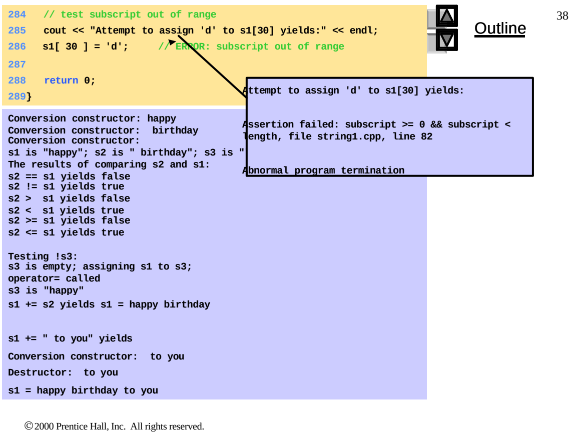
```
Destructor: happy birthday to you
```

```
s1 after s1[0] = 'H' and s1[6] = 'B' is: Happy Birthday to you
```

Outline

2. Function calls

37



```
   // test subscript out of range
   cout << "Attempt to assign 'd' to s1[30] yields:" << endl;
  s1[ 30 ] = 'd';      // ERROR: subscript out of range

   return 0;
}
```

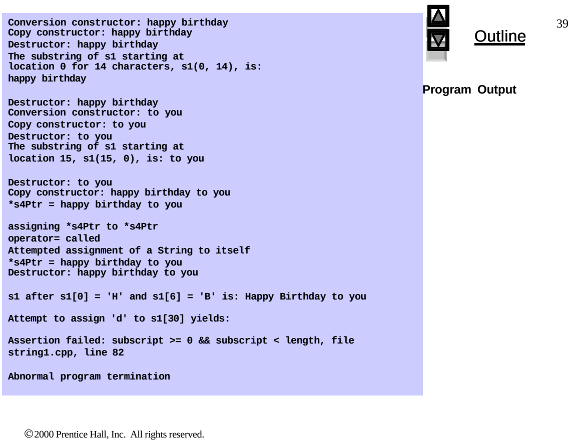
```
Conversion constructor: happy
Conversion constructor:  birthday
Conversion constructor:
s1 is "happy"; s2 is " birthday"; s3 is "
The results of comparing s2 and s1:
s2 == s1 yields false
s2 != s1 yields true
s2 >  s1 yields false
s2 <  s1 yields true
s2 >= s1 yields false
s2 <= s1 yields true

Testing !s3:
s3 is empty; assigning s1 to s3;
operator= called
s3 is "happy"
s1 += s2 yields s1 = happy birthday


s1 += " to you" yields
Conversion constructor:  to you
Destructor:  to you
s1 = happy birthday to you
```

```
Attempt to assign 'd' to s1[30] yields:


Assertion failed: subscript >= 0 && subscript <
length, file string1.cpp, line 82


Abnormal program termination
```

Outline

38

© 2000 Prentice Hall, Inc. All rights reserved.



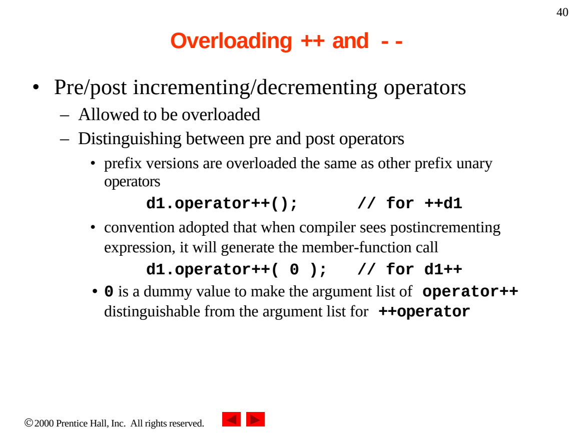
```
Conversion constructor: happy birthday
Copy constructor: happy birthday
Destructor: happy birthday
The substring of s1 starting at
location 0 for 14 characters, s1(0, 14), is:
happy birthday

Destructor: happy birthday
Conversion constructor: to you
Copy constructor: to you
Destructor: to you
The substring of s1 starting at
location 15, s1(15, 0), is: to you

Destructor: to you
Copy constructor: happy birthday to you
*s4Ptr = happy birthday to you

assigning *s4Ptr to *s4Ptr
operator= called
Attempted assignment of a String to itself
*s4Ptr = happy birthday to you
Destructor: happy birthday to you

s1 after s1[0] = 'H' and s1[6] = 'B' is: Happy Birthday to you

Attempt to assign 'd' to s1[30] yields:

Assertion failed: subscript >= 0 && subscript < length, file
string1.cpp, line 82

Abnormal program termination
```

Outline

Program Output

39

© 2000 Prentice Hall, Inc. All rights reserved.



40

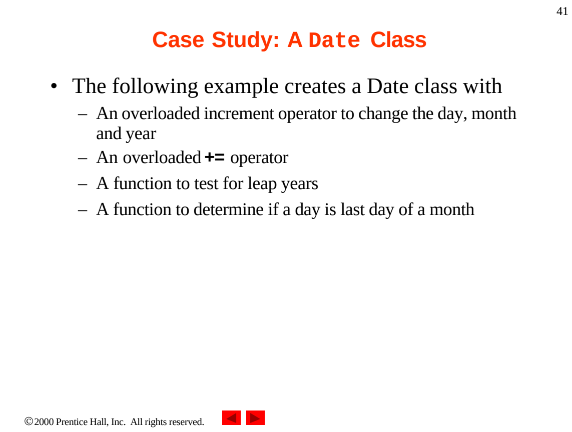
Overloading ++ and - -

- Pre/post incrementing/decrementing operators
  - Allowed to be overloaded
  - Distinguishing between pre and post operators
    - prefix versions are overloaded the same as other prefix unary operators

      `d1.operator++();      // for ++d1`
    - convention adopted that when compiler sees postincrementing expression, it will generate the member-function call

      `d1.operator++( 0 );   // for d1++`
    - `0` is a dummy value to make the argument list of `operator++` distinguishable from the argument list for `++operator`

© 2000 Prentice Hall, Inc. All rights reserved.

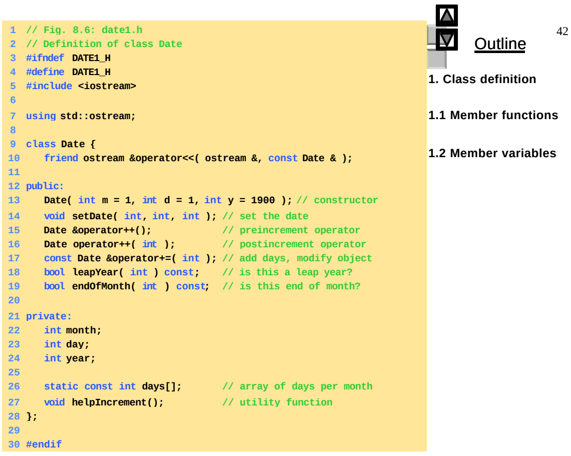


41

Case Study: A `Date` Class

- The following example creates a Date class with
  - An overloaded increment operator to change the day, month and year
  - An overloaded `+=` operator
  - A function to test for leap years
  - A function to determine if a day is last day of a month

© 2000 Prentice Hall, Inc. All rights reserved.



```
// Fig. 8.6: date1.h
// Definition of class Date
#ifndef DATE1_H
#define DATE1_H
#include <iostream>

using std::ostream;

class Date {
   friend ostream &operator<<( ostream &, const Date & );

public:
   Date( int m = 1, int d = 1, int y = 1900 ); // constructor
   void setDate( int, int, int ); // set the date
   Date &operator++();            // preincrement operator
   Date operator++( int );        // postincrement operator
   const Date &operator+=( int ); // add days, modify object
   bool leapYear( int ) const;    // is this a leap year?
   bool endOfMonth( int ) const;  // is this end of month?

private:
   int month;
   int day;
   int year;

   static const int days[];       // array of days per month
   void helpIncrement();          // utility function
};

#endif
```

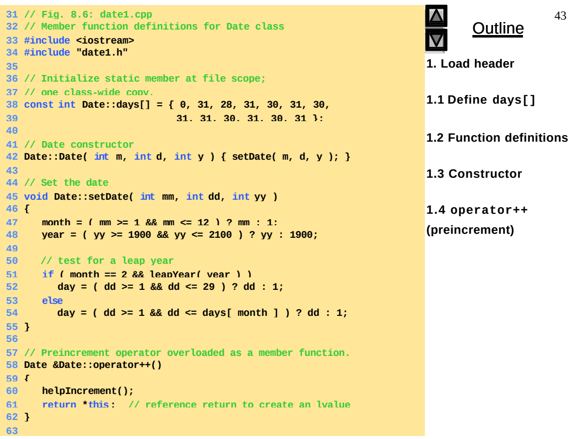
Outline

1. Class definition

1.1 Member functions

1.2 Member variables

42



```
// Fig. 8.6: date1.cpp
// Member function definitions for Date class
#include <iostream>
#include "date1.h"

// Initialize static member at file scope;
// one class-wide 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 ); }

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

   // 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;
}

// Preincrement operator overloaded as a member function.
Date &Date::operator++()
{
   helpIncrement();
   return *this;  // reference return to create an lvalue
}

```

Outline

1. Load header

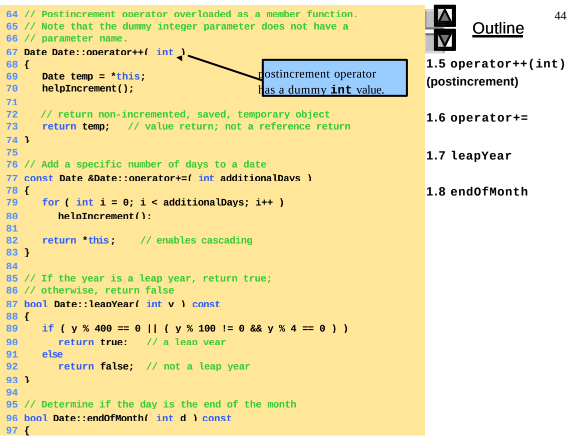
1.1 Define days[]

1.2 Function definitions

1.3 Constructor

1.4 operator++

(preincrement)



```
// Postincrement operator overloaded as a member function.
// Note that the dummy integer parameter does not have a
// parameter name.
Date Date::operator++( int )
{
   Date temp = *this;
   helpIncrement();

   // return non-incremented, saved, temporary object
   return temp;   // value return; not a reference return
}

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

   return *this;    // enables cascading
}

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

// Determine if the day is the end of the month
bool Date::endOfMonth( int d ) const
{
```

postincrement operator has a dummy **int** value.

Outline

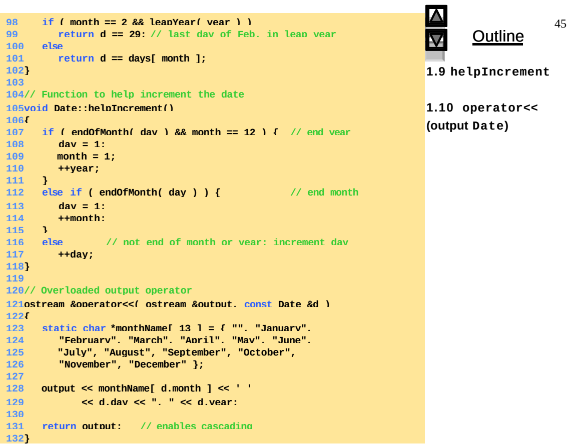
1.5 operator++(int)

(postincrement)

1.6 operator+=

1.7 leapYear

1.8 endOfMonth



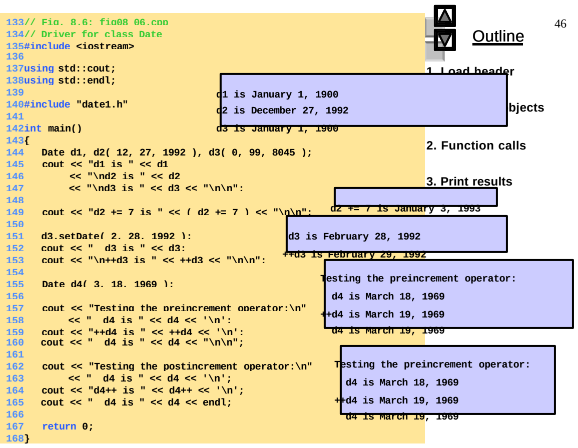
```
   if ( month == 2 && leapYear( year ) )
      return d == 29; // last day of Feb. in leap year
   else
      return d == days[ month ];
}

// Function to help increment the date
void Date::helpIncrement()
{
   if ( endOfMonth( day ) && month == 12 ) {  // end year
      day = 1;
     month = 1;
      ++year;
   }
   else if ( endOfMonth( day ) ) {            // end month
      day = 1;
      ++month;
   }
   else        // not end of month or year; increment day
      ++day;
}

// Overloaded output operator
ostream &operator<<( ostream &output, const Date &d )
{
   static char *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
}
```

Outline

1.9 helpIncrement

1.10 operator<<

(output Date)



```
// Fig. 8.6: fig08_06.cpp
// Driver for class Date
#include <iostream>

using std::cout;
using std::endl;

#include "date1.h"

int main()
{
   Date d1, d2( 12, 27, 1992 ), d3( 0, 99, 8045 );
   cout << "d1 is " << d1
        << "\nd2 is " << d2
        << "\nd3 is " << d3 << "\n\n";

   cout << "d2 += 7 is " << ( d2 += 7 ) << "\n\n";

   d3.setDate( 2, 28, 1992 );
   cout << "  d3 is " << d3;
   cout << "\n++d3 is " << ++d3 << "\n\n";

   Date d4( 3, 18, 1969 );

   cout << "Testing the preincrement operator:\n"
        << "  d4 is " << d4 << '\n';
   cout << "++d4 is " << ++d4 << '\n';
  cout << "  d4 is " << d4 << "\n\n";

   cout << "Testing the postincrement operator:\n"
        << "  d4 is " << d4 << '\n';
   cout << "d4++ is " << d4++ << '\n';
  cout << "  d4 is " << d4 << endl;

   return 0;
}
```

```
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
```

```
Testing the preincrement operator:
 d4 is March 18, 1969
++d4 is March 19, 1969
 d4 is March 19, 1969
```

```
Testing the preincrement operator:
 d4 is March 18, 1969
++d4 is March 19, 1969
 d4 is March 19, 1969
```

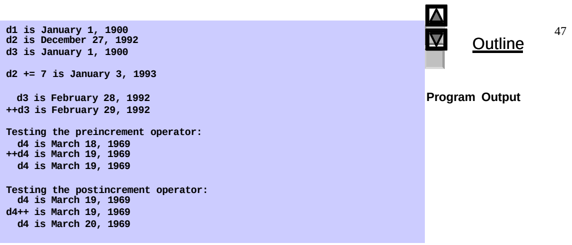
Outline

1. Load header

bjects

2. Function calls

3. Print results



```
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

Testing the preincrement operator:
  d4 is March 18, 1969
++d4 is March 19, 1969
  d4 is March 19, 1969

Testing the postincrement operator:
  d4 is March 19, 1969
d4++ is March 19, 1969
  d4 is March 20, 1969
```

Outline

Program Output