Operator Overloading

- Most programmers implicitly use **overload** operators regularly. For example, the addition operator (+) operates quite differently on integers, floats and doubles.

- Can only add user-defined operand types (i.e., the class that you have created) to existing operators, cannot add new operators to the existing systems.

Operator Overloading (2)

- The number of operands that an operator takes cannot be altered.
- Cannot redefine the way that basic operations work, should act (and have the semantics) as much like the equivalent C operators as possible.
- Cannot change the precedence level or associativity of an operator.

Operator Overloading (3)

- Operator functions may not have default parameters.

- Can overload everything but

  . :: ?: sizeof

Operator Overloaded Definitions

There are two approaches:

- Overloaded by class member functions.
- Using "friend“.

Defined by Member Functions

- When an operator function is implemented as a member function, the leftmost (or only) operand) must be a class object (or a reference to a class object) of the operator's class.

Defined by Member Functions (2)

- Unary operations:

  the left operand, or the only operand in unary operator overloads, is replaced by "this“ pointer, so is **not** listed in the operator overload function's argument list.

- E.g., the following overloads **unary minus**,

  some_class operator-(void);

  some_class obj;

  -obj;

Defined by Member Functions (3)

- Binary operators:
  1. ***some_class operator-(some_class &r);***
     some_class x, y;
     x - y;
  2. ***some_class operator-(int r);***
     some_class x;
     x - 1; // 1 – x; ??
- The compiler tells which is which by checking the arguments in the definition.

Defined by Member Functions (4)

- In a general case, at least one operand of the function must be a user-defined type – needs other approaches.

  For example,

  some_class operator-(some_class left, int right);

  some_class operator-(int left, some_class right);

Defined using “friend”

- If the left operand must be an object of a different class, this operator function must be implemented as a non-member function using "friend".

- A friend function does not have a "this" pointer.

Defined using “friend” (cont)

- Using friend operator function, you can allow objects to be used in operations involving built-in types where the built-in type is on the left side of the operator.

  For example,

friend some_class operator-(int left, some_class right);

some_class obj;

1 - obj;

Operator++()

- There is no way to determine whether an overloaded ++ or - - preceded or followed its operand as in operator++().

some_class operator++();

++obj;

some_class operator++(int x)

obj++;

in this case, *x* will be passed the value 0.

- ***some_class& operator++() {***

  ****this += 1;*** // increment

  return *this; // fetch

  }

- ***const some_class operator++(int) {*** // silently pass 0

  some_class oldValue = *this; // fetch

  ++(*this); // increment

  return oldValue;

  }

A Closer Look at Assignment Operator

- Cannot use “friend” to overload assignment operator.

- It takes a reference parameter to prevent a copy of the object on the right side of the assignment.
- It returns a reference, not an object.

- Assignment(=) operator is a special operator that will be provided by the constructor to the class when programmer has not provided (overloaded) as member of the class (like copy constructor).
- When programmer is overloading = operator using friend function, two = operations will exists:
  1) compiler is providing = operator
  2) programmer is providing (overloading) = operator by friend function.
- Then simply ambiguity will be created and compiler will gives error. Its compilation error.

Type Conversion

- ***class some_class {***
  operator int () {}; // convert some_class to int
  operator char *() {}; // convert some_class to char *
  }

- Takes no argument.
- Specifies no return type.

Case Study: an Array Class

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

private:
    int size;     // size of the array
    int *ptr;     // pointer to the first element of the array

    static int arrayCount;          // # of Arrays instantiated
    // int Array::arrayCount = 0;   // initialize static member
```

Array Class (2)

Array(int = 10); // default constructor

Array(const Array &); // copy constructor

// Array(const Array &init) : size(init.size) {

Array Class (3)

```
const Array & operator=( const Array & right );       // assign array
    // if ( &right == this ) return *this;            // check for self-assignment
    //
    // for array of different sizes, deallocate original
    // left side array, then allocate new left side array.
    //
    // if ( size != right.size ) {
    //       delete [size] ptr;
    //       size = right.size;
    //       ptr = new int [size];
    // }
    // for (int i = 0; i < size; i++)
    //       ptr[i] = right.ptr[i];
    //
    // return *this;                 // enables x = y = z;
```

Array Class (4)

```
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 ); }

// reference return creates an lvalue
int & operator[]( int subscript );          // subscript operator
    //  assert ( 0 <= subscript && subscript < size );
    //  return ptr[subscript];

// const reference return creates an rvalue
const int & operator[]( int ) const;
```

- **int & operator[](int subscript);**
  // The non-const version allows for l-value assignment with
  // non-const objects.
- **const int & operator[](int) const;**
  // The const version allows for r-value usage with const objects.

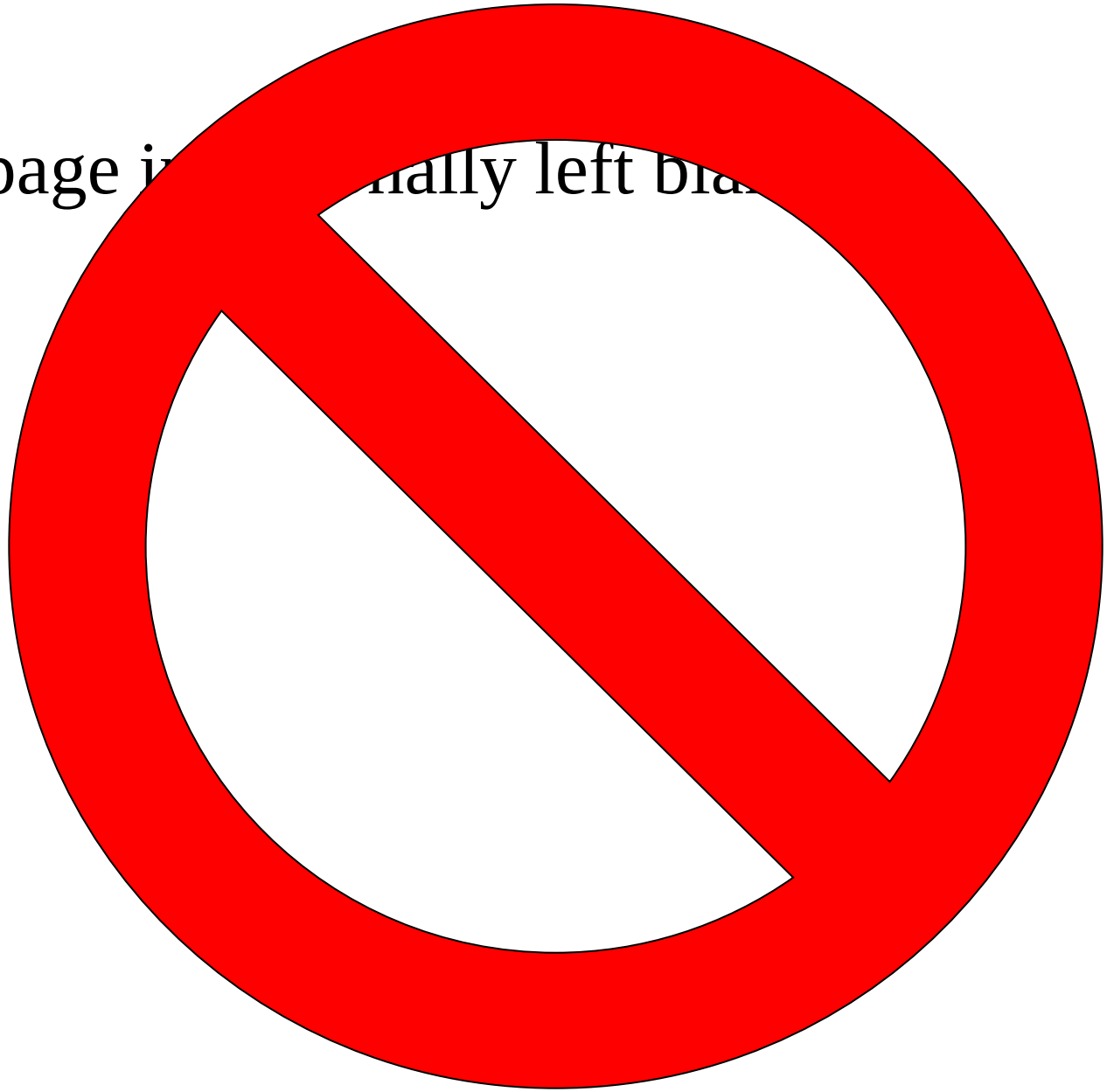
```
const Array ca;          // ca is a const Array
Array nca;               // nca is a non-const Array

ca[0];          // uses the 'const' version
nca[0];         // uses the 'non-const' version

ca[0] = 5;      // ERROR - const version returns a const reference...
                // cannot assign to a const reference
foo = ca[0];    // OK

nca[0] = 5;     // OK - non-const version allows assignment
foo = nca[0];   // also OK
```

String Class

```
class String {
   operator char *() const;   // cast operator
   const String & operator+=( const String & );
         // concatenation
   String operator()( int index, int subLength );
         // return a substring
```

避免像(s1+=s2)+=s3

This page i nally left bla



Chapter 8 - Operator Overloading

Outline

8.1 Introduction

- Operator overloading
  - 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

8.2 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 (**`+`**)

8.3 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

8.3 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

8.4 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

8.5 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

8.6 Overloading Unary Operators

- Overloading unary operators
  - Can be overloaded with no arguments or one argument
  - Example declaration as a member function:

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


8.6 Overloading Unary Operators

– Example declaration as a non-member function

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


8.7 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 )`

8.7 Overloading Binary Operators

- Non-member function, two arguments
- Example:

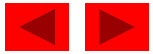
```
class String {
   friend const String &operator+=(
                  String &, const String & );
   ...
};
```

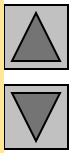
`y += z` is equivalent to `operator+=( y, z )`

8.8 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 **!=**

請詳細閱讀





Outline

1. Class definition

1.1 Function prototypes

```
1  // Fig. 8.4: array1.h
2  // Simple class Array (for integers)
3  #ifndef ARRAY1_H
4  #define ARRAY1_H
5
6  #include <iostream>
7
8  using std::ostream;
9  using std::istream;
10
11 class Array {
12    friend ostream &operator<<( ostream &, const Array & );
13    friend istream &operator>>( istream &, Array & );
14 public:
15    Array( int = 10 );                    // default constructor
16    Array( const Array & );               // copy constru
17    ~Array();                             // destructor
18    int getSize() const;                  // return size
19    const Array &operator=( const Array & ); // assign arrays
20    bool operator==( const Array & ) const; // compare equal
21
22    // Determine if two arrays are not equal and
23    // return true, otherwise return false (uses operator==).
24    bool operator!=( const Array &right ) const
25       { return ! ( *this == right ); }
26
27    int &operator[]( int );               // subscript operator
28    const int &operator[]( int ) const;   // subscript operator
29    static int getArrayCount();           // Return count of
30                                          // arrays instantiated.
31 private:
32    int size; // size of the array
33    int *ptr; // pointer to first element of array
34    static int arrayCount;  // # of Arrays instantiated
```

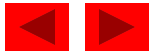
Notice all the overloaded operators used to implement the class.

8.9 Converting between Types

- Cast operator
  - 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



8.9 Converting between Types

```
A::operator int() const;
```

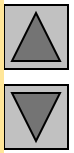
- Declares an overloaded cast operator function for converting an object of **A** into an integer

```
A::operator otherClass() const;
```

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

8.10 Case Study: A `String` Class

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

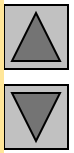


Outline

1. Class definition

1.1 Member functions, some definitions

```
1  // Fig. 8.5: string1.h
2  // Definition of a String class
3  #ifndef STRING1_H
4  #define STRING1_H
5
6  #include <iostream>
7
8  using std::ostream;
9  using std::istream;
10
11 class String {
12    friend ostream &operator<<( ostream &, const String & );
13    friend istream &operator>>( istream &, String & );
14
15 public:
16    String( const char * = "" ); // conversion/default ctor
17    String( const String & );    // copy constructor
18    ~String();                   // destructor
19    const String &operator=( const String & );  // assignment
20    const String &operator+=( const String & ); // concatenation
21    bool operator!() const;                  // is String empty?
22    bool operator==( const String & ) const; // test s1 == s2
23    bool operator<( const String & ) const;  // test s1 < s2
24
25    // test s1 != s2
26    bool operator!=( const String & right ) const
27       { return !( *this == right ); }
28
29    // test s1 > s2
30    bool operator>( const String &right ) const
31       { return right < *this; }
32
33    // test s1 <= s2
```



Outline

1.2 Member variables

```
34    bool operator<=( const String &right ) const
35       { return !( right < *this ); }
36
37    // test s1 >= s2
38    bool operator>=( const String &right ) const
39       { return !( *this < right ); }
40
41    char &operator[]( int );             // subscript operator
42    const char &operator[]( int ) const; // subscript operator
43    String operator()( int, int );       // return a substring
44    int getLength() const;               // return string length
45
46 private:
47    int length;                   // string length
48    char *sPtr;                   // pointer to start of string
49
50    void setString( const char * );  // utility function
51 };
52
53 #endif
54 // Fig. 8.5: string1.cpp
55 // Member function definitions for class String
56 #include <iostream>
57
58 using std::cout;
59 using std::endl;
60
61 #include <iomanip>
62
63 using std::setw;
64
```

8.11 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**