Operator Overloading

Dynamic Memory Management

You can control the *allocation* and *deallocation* of memory in a program for objects and for arrays of *any built-in or user-defined type*.

This is known as **dynamic memory management** and is performed with the operators **new** and **delete**.

You can use the **new** operator to dynamically **allocate** (i.e., reserve) the exact amount of memory required to hold an *object* or *built-in array* at execution time.

The object or built-in array is created in the **free store** (also called the **heap**)—*a region of memory assigned to each program for storing dynamically allocated objects*.

Once memory is allocated, you can access it via the *pointer returned by operator new*.

When you no longer need the memory, you can *return* it to the free store by using the delete operator to **deallocate** (i.e., release) the memory, which can then be reused by future new operations.

Dynamic Memory Management

*Obtaining Dynamic Memory with **new:***

Time* timePtr = **new** Time; // or new Time**()**

1. The new operator allocates storage of the proper size for an object of type Time,
2. calls a constructor to initialize the object,
3. returns a pointer to the type specified to the right of the new operator (i.e., a Time*).

In the preceding statement, class Time's default constructor is called, because we did not supply arguments to initialize the Time object.

If new is unable to find sufficient space in memory for the object, it indicates that an error occurred by throwing **bad_alloc** exception.

Dynamic Memory Management

*Releasing Dynamic Memory with **delete**:*

delete timePtr;

1. The delete operator first calls the destructor for the object to which timePtr points,
2. then deallocates the memory associated with the object, returning the memory to the free store or heap.

- Not releasing dynamically allocated memory when it's no longer needed can cause the system to run out of memory prematurely. This is sometimes called a "**memory leak**."

- Do not delete memory that was not allocated by new. Doing so results in undefined behavior.

- After you delete a block of dynamically allocated memory, be sure not to delete the same block **again**. One way to guard against this is to immediately set the pointer to **nullptr**. **Deleting a nullptr has no effect**.

Dynamic Memory Management

Initializing Dynamic Memory:

You can provide an initializer for a newly created fundamental-type variable, as in

```
double* ptr{new double{3.14159}};
// or double* ptr = new double(3.14159);
```

which initializes a newly created double to 3.14159 and assigns the resulting pointer to *ptr*.

The same syntax can be used to specify arguments to an object's constructor, as in

```
Time* timePtr{new Time{12, 45, 0}};
```

which initializes a new Time object to 12:45 PM and assigns its pointer to timePtr.

Dynamic Memory Management

*Dynamically Allocating Built-In Arrays with **new[]***

You can also use the new operator to allocate **built-in arrays** dynamically:

Time* timesArray = **new Time**[**10**] {};

1. The new[] operator allocates storage of the proper size for 10-element array of '*Time*'s,
2. calls a constructor for every object in the array (default constructor in this case),
3. returns a pointer to the **first** element of a dynamically allocated 10-element array.

- The list-initializer braces **{}** following new int[10]—which are allowed as of **C++11**—initialize the array's elements, setting *fundamental-type elements* to 0, *bools* to false and *pointers* to nullptr.

- If the list-initializer braces **{}** are not present, and if the elements are class objects, they're initialized by their **default constructors**.

Dynamic Memory Management

*Dynamically Allocating Built-In Arrays with **new[]***

- The list-initializer braces may also contain a comma-separated list of initializers for the array's elements:

```
int* gradesArray = new int[10] {1, 2, 3};
A* gradesArray = new A[10] {1, 2, 3}; // will call A(1), A(2), A(3) for object 1-3
                                      // and A() for object 4-10
```

- The size of a built-in array created at compile time must be specified **using an integral constant expression**; however, a dynamically allocated array's size can be specified using ***any*** **nonnegative integral expression.**

- *Note, that there are compilers that support built-in arrays with **non-const** size*. However, the C++ standard says that **built-in array size must be constant expression.**

Dynamic Memory Management

*Releasing Dynamically Allocated Built-In Arrays with **delete[]***

To deallocate the memory to which timesArray points, use the statement

delete[] timesArray;

1. the statement first calls the destructor for every object in the array,
2. then deallocates the memory.

If the preceding statement did *not* include the square brackets ([]):

delete timesArray;

the result is ***undefined***—some compilers call the destructor only for the first object in the array in this case, resulting to **memory leak**.

- The result of deleting a single object with operator *delete[]* is also **undefined**:

  delete[] time; // time is a single object

- Using delete or delete[] on a nullptr has no effect, there is **NO need** to write

```
if (nullptr != time) {
    delete time;
}
```

Dynamic Memory Management

void* operator new (std::size_t size);

Allocates *size* bytes of storage and returns a non-null pointer to the first byte of this block.

void* operator new (std::size_t size, const std::nothrow_t& nothrow_value) noexcept;

Same as above, except that on failure it returns a null pointer instead of throwing an exception.

void* operator new (std::size_t size, void* ptr) noexcept;

Simply returns *ptr* (no storage is allocated).
Notice though that, if the function is called by a **new-expression**, the proper initialization will be performed (for class objects, this includes calling its default constructor).

- Note that these are just regular functions!
- **new** is an **operator** in C++ that uses these functions!

Dynamic Memory Management

operator new function VS new operator

Operator new is a function that just allocates raw memory.

The **new operator** does the following job:

- The memory for the object is allocated from heap using **operator new function**.
- The constructor of the class is **invoked by complier** to properly initialize this memory.

1. **Operator vs function**: **new** is an operator as well as a keyword whereas "**operator new"** is only a function.
2. **New** calls "**Operator new function**": "new operator" calls "operator new()" function, like the way **"+ operator" calls "operator +()" function**.
3. "**Operator new function**" can be overloaded: Operator new can be overloaded just like functions allowing us to do customized tasks.

The same assertion is true for "**delete operator"** and **"operator delete function".**

Dynamic Memory Management

operator new function can be called explicitly as a regular function, but in C++, **new** is an **operator** with a very specific behavior.

For the following example:

MyClass * p1 = new MyClass(1);

1. first '**operator new'** function is called with the size of its type specifier as first argument:

   MyClass * p1 = (MyClass*) ::operator new (sizeof(MyClass));

2. and if this is successful, the p1 automatically is initialized:
   MyClass::MyClass(int) constructor is called by compiler on p1.

```
class Car
{
public:
   Car(string name, int year)
      : _name(name),
        _year(year)
   {
      cout << "Constructor called" << endl;
   }
   void display()
   {
      cout << "Name: " << _name << endl;
      cout << "Year: " << _year << endl;
   }
   static void* operator new(size_t size)
   {
      cout << "overloaded 'operator new' function is called" << endl;
      void* p = malloc(size);
      return p;
   }
   static void operator delete(void* ptr)
   {
      cout << "overloaded 'operator delete' function is called" << endl;
      free(ptr);
   }
```

- **NOTE**: Both overloaded new and delete operator functions must be **static members** since there are class specific. They are implicitly defined as static if static is not specified.

```
 ~Car()
   {
      cout << "Destructor called" << endl;
   }
public:
   string _name;
   int _year;
};

int main()
{
   Car *c = new Car("BMW", 2020);
   c->display();
   delete c;
}
```

Result:

```
overloaded 'operator new' function is called
Constructor called
Name: BMW
Year: 2020
Destructor called
overloaded 'operator delete' function is called
```

- This proves that **new operator** first calls ‘operator new’ function, and then the corresponding constructor is called. Whereas **delete operator** first calls the destructor and then “operator delete function” to deallocate the memory.

Example

```
1  // operator new example
2  #include <iostream>     // std::cout
3  #include <new>          // ::operator new
4
5  struct MyClass {
6    int _i;
7    MyClass(int i = 45) : _i(i) {std::cout << "constructed [" << this << "]" << " _i=" << _i << std::endl;}
8    ~MyClass() {std::cout << "destructed [" << this << "]" << " _i=" << _i << std::endl; }
9  };
10
11 int main () {
12
13   std::cout << "1: ";
14   MyClass * p1 = new MyClass();
15
16       // allocates memory by calling: operator new (sizeof(MyClass))
17       // and then constructs an object at the newly allocated space
18
19   std::cout << "2: ";
20   MyClass * p2 = new (std::nothrow) MyClass(2);
21       // allocates memory by calling: operator new (sizeof(MyClass),std::nothrow)
22       // and then constructs an object at the newly allocated space
23
```

Example

```
24    std::cout << "3: ";
25    new (p2) MyClass(3);
26  //    ::operator new (sizeof(MyClass),p2);
27        // does not allocate memory -- calls: operator new (sizeof(MyClass),p2)
28        // but constructs an object at p2
29
30    // Notice though that calling this function directly does not construct an object:
31    std::cout << "4: ";
32    MyClass * p3 = (MyClass*) operator new (sizeof(MyClass));
33        // allocates memory by calling: operator new (sizeof(MyClass))
34        // but does not call MyClass's constructor
35    std::cout << std::endl;
36
37    delete p1;
38    delete p2;
39    delete p3;
40
41    return 0;
42  }
```

Result:

1: constructed [0x505368] _i=45
2: constructed [0x505378] _i=2
3: constructed [0x505378] _i=3
4:
destructed [0x505368] _i=45
destructed [0x505378] _i=3
destructed [0x505388] _i=0

For more details see https://cplusplus.com/reference/new/operator%20new/

Array Class Example

Pointer-based arrays have many problems:

- A program can easily “walk off” either end of a built-in array, because C++ does not check whether subscripts fall outside the range of the array.
- An entire built-in array cannot be input or output at once; each element must be read or written individually.
- Two built-in arrays cannot be meaningfully compared with equality or relational operators.
- When a built-in array is passed to a general-purpose function designed to handle arrays of any size, the array’s *size* must be passed as an additional argument.
- One built-in array cannot be assigned to another with the assignment operator(s).

We will create a powerful **Array** class that addresses these problems.

Array Class Example

```
2  // Array class test program.
3  #include <iostream>
4  #include <stdexcept>
5  #include "Array.h"
6  using namespace std;
7
8  int main() {
9     Array integers1{7}; // seven-element Array
10    Array integers2; // 10-element Array by default
11
12    // print integers1 size and contents
13    cout << "Size of Array integers1 is " << integers1.getSize()
14       << "\nArray after initialization: " << integers1;
15
16    // print integers2 size and contents
17    cout << "\nSize of Array integers2 is " << integers2.getSize()
18       << "\nArray after initialization: " << integers2;
19
20    // input and print integers1 and integers2
21    cout << "\nEnter 17 integers:" << endl;
22    cin >> integers1 >> integers2;
23
24    cout << "\nAfter input, the Arrays contain:\n"
25       << "integers1: " << integers1
26       << "integers2: " << integers2;
```

```
28      // use overloaded inequality (!=) operator
29      cout << "\nEvaluating: integers1 != integers2" << endl;
30
31      if (integers1 != integers2) {
32         cout << "integers1 and integers2 are not equal" << endl;
33      }
34
35      // create Array integers3 using integers1 as an
36      // initializer; print size and contents
37      Array integers3{integers1}; // invokes copy constructor
38
39      cout << "\nSize of Array integers3 is " << integers3.getSize()
40         << "\nArray after initialization: " << integers3;
41
42      // use overloaded assignment (=) operator
43      cout << "\nAssigning integers2 to integers1:" << endl;
44      integers1 = integers2; // note target Array is smaller
45
46      cout << "integers1: " << integers1 << "integers2: " << integers2;
47
48      // use overloaded equality (==) operator
49      cout << "\nEvaluating: integers1 == integers2" << endl;
50
51      if (integers1 == integers2) {
52         cout << "integers1 and integers2 are equal" << endl;
53      }
54
55      // use overloaded subscript operator to create rvalue
56      cout << "\nintegers1[5] is " << integers1[5];
57
58      // use overloaded subscript operator to create lvalue
59      cout << "\n\nAssigning 1000 to integers1[5]" << endl;
60      integers1[5] = 1000;
61      cout << "integers1: " << integers1;
```

```
63      // attempt to use out-of-range subscript
64      try {
65         cout << "\nAttempt to assign 1000 to integers1[15]" << endl;
66         integers1[15] = 1000; // ERROR: subscript out of range
67      }
68      catch (out_of_range& ex) {
69         cout << "An exception occurred: " << ex.what() << endl;
70      }
71   }
```

```
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

Enter 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
integers1 and integers2 are not equal

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

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
integers1 and integers2 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]
An exception occurred: Subscript out of range
```

```
2  // Array class definition with overloaded operators.
3  #ifndef ARRAY_H
4  #define ARRAY_H
5
6  #include <iostream>
7
8  class Array {
9     friend std::ostream& operator<<(std::ostream&, const Array&);
10    friend std::istream& operator>>(std::istream&, Array&);
11
12 public:
13    explicit Array(int = 10); // default constructor
14    Array(const Array&); // copy constructor
15    ~Array(); // destructor
16    size_t getSize() const; // return size
17
18    const Array& operator=(const Array&); // assignment operator
19    bool operator==(const Array&) const; // equality operator
20
21    // inequality operator; returns opposite of == operator
22    bool operator!=(const Array& right) const {
23       return ! (*this == right); // invokes Array::operator==
24    }
25
26    // subscript operator for non-const objects returns modifiable lvalue
27    int& operator[](int);
28
29    // subscript operator for const objects returns rvalue
30    int operator[](int) const;
31 private:
32    size_t size; // pointer-based array size
33    int* ptr; // pointer to first element of pointer-based array
34 };
35
36 #endif
```

```
1  // Fig. 10.11: Array.cpp
2  // Array class member- and friend-function definitions.
3  #include <iostream>
4  #include <iomanip>
5  #include <stdexcept>
6
7  #include "Array.h" // Array class definition
8  using namespace std;
9
10 // default constructor for class Array (default size 10)
11 Array::Array(int arraySize)
12    : size{(arraySize > 0 ? static_cast<size_t>(arraySize) :
13          throw invalid_argument{"Array size must be greater than 0"})},
14      ptr{new int[size]{}} { /* empty body */ }
```

The initializer for ptr

new int[size]{}

uses an empty initializer list to set all the elements of the dynamically allocated built-in array to 0.

```
16  // copy constructor for class Array;
17  // must receive a reference to an Array
18  Array::Array(const Array& arrayToCopy)
19     : size{arrayToCopy.size}, ptr{new int[size]} {
20     for (size_t i{0}; i < size; ++i) {
21        ptr[i] = arrayToCopy.ptr[i]; // copy into object
22     }
23  }
```

- The compiler generated copy constructor will just assign the ptr of the arrayToCopy object to the new created object ptr, and both Array objects will point to the same dynamically allocated memory.
- In that case, the first destructor to execute would delete the dynamically allocated memory, and the other object's ptr would point to memory that's no longer allocated, a situation called a **dangling pointer.**
- The argument to a copy constructor should be a const reference to allow a const object to be copied.
- The arraytoCopy parameter must be &, otherwise it will result to recursion (compiler gives error message in this case).

Copy constructors are invoked when

1. passing an object by value to a function,
2. returning an object by value from a function or
3. initializing an object with a copy of another object of the same class:

Array integers3 = integers1; // **invokes copy constructor**

```
25  // destructor for class Array
26  Array::~Array() {
27     delete[] ptr; // release pointer-based array space
28  }
29
30  // return number of elements of Array
31  size_t Array::getSize() const {
32     return size; // number of elements in Array
33  }
```

- If after deleting dynamically allocated memory, the pointer *will continue to exist in memory*, set the pointer's value to nullptr to indicate that the pointer no longer points to memory in the free store.

- When the pointer is set to nullptr, the program loses access to that free-store space, which could be reallocated for a different purpose.

- If you *do not set the pointer* to nullptr, your code could inadvertently access the reallocated memory, causing subtle, nonrepeatable logic errors.

- We did not set ptr to nullptr in the destructor, because after the destructor executes, the Array object no longer exists in memory.

```
35  // overloaded assignment operator;
36  // const return avoids: (a1 = a2) = a3
37  const Array& Array::operator=(const Array& right) {
38     if (&right != this) { // avoid self-assignment
39        // for Arrays of different sizes, deallocate original
40        // left-side Array, then allocate new left-side Array
41        if (size != right.size) {
42           delete[] ptr; // release space
43           size = right.size; // resize this object
44           ptr = new int[size]; // create space for Array copy
45        }
46
47        for (size_t i{0}; i < size; ++i) {
48           ptr[i] = right.ptr[i]; // copy array into object
49        }
50     }
51
52     return *this; // enables x = y = z, for example
53  }
```

- operator= tests for self-assignment since **integers1=integers1** call would unnecessarily copy the elements of the Array into itself.
- the member function returns the current object (i.e., *this in line 52) as a constant reference; this enables cascaded Array assignments such as x = y = z, but prevents ones like (x = y) = z.

❖ Not providing a copy constructor, overloaded assignment operator and destructor for a class when objects of that class contain pointers to dynamically allocated memory is a potential **logic error** (memory leak in case of not providing a destructor).

```
55  // determine if two Arrays are equal and
56  // return true, otherwise return false
57  bool Array::operator==(const Array& right) const {
58     if (size != right.size) {
59        return false; // arrays of different number of elements
60     }
61
62     for (size_t i{0}; i < size; ++i) {
63        if (ptr[i] != right.ptr[i]) {
64           return false; // Array contents are not equal
65        }
66     }
67
68     return true; // Arrays are equal
69  }
```

```
// inequality operator; returns opposite of == operator
bool operator!=(const Array& right) const {
   return ! (*this == right); // invokes Array::operator==
}
```

- Writing operator!= in this manner enables you to reuse operator==, which reduces the amount of code that must be written in the class.
- Also, the full function definition for operator!= is in the Array header. This allows the compiler to **inline** the definition of operator!=.

❑ **Note**, that the object of a class can look at the **private data** of any other object of that class (for example, see line 63 where private right.ptr is accessed).

```
71  // overloaded subscript operator for non-const Arrays;
72  // reference return creates a modifiable lvalue
73  int& Array::operator[](int subscript) {
74     // check for subscript out-of-range error
75     if (subscript < 0 || subscript >= size) {
76        throw out_of_range{"Subscript out of range"};
77     }
78
79     return ptr[subscript]; // reference return
80  }
81
82  // overloaded subscript operator for const Arrays
83  // const reference return creates an rvalue
84  int Array::operator[](int subscript) const {
85     // check for subscript out-of-range error
86     if (subscript < 0 || subscript >= size) {
87        throw out_of_range{"Subscript out of range"};
88     }
89
90     return ptr[subscript]; // returns copy of this element
91  }
```

- Each definition of operator[] determines whether the subscript it receives as an argument is in range—and if not, each throws an **out_of_range** exception.
- If the subscript is in range, the non-const version of operator[] returns the appropriate Array element as **a reference so that it may be used as a modifiable lvalue** (e.g., on the left side of an assignment statement).
- If the subscript is in range, the const version of operator[] returns a **copy** of the appropriate element of the Array.

```
93  // overloaded input operator for class Array;
94  // inputs values for entire Array
95  istream& operator>>(istream& input, Array& a) {
96     for (size_t i{0}; i < a.size; ++i) {
97        input >> a.ptr[i];
98     }
99
100    return input; // enables cin >> x >> y;
101 }
102
103 // overloaded output operator for class Array
104 ostream& operator<<(ostream& output, const Array& a) {
105    // output private ptr-based array
106    for (size_t i{0}; i < a.size; ++i) {
107       output << a.ptr[i] << "  ";
108    }
109
110    output << endl;
111    return output; // enables cout << x << y;
112 }
```

- Again, these stream insertion and stream extraction operator functions **cannot be members of class Array**, because the Array object is always mentioned on the right side of the stream insertion or stream extraction operator:

  cout << arrayObject or cin >> arrayObject

Array Class Example

- Prior to C++11, you could prevent class objects from being copied or assigned by declaring as **private** the class's copy constructor and overloaded assignment operator.

- As of C++11, you can simply delete these functions from your class. For example:

  Array(const Array&) = **delete**;
  const Array& operator=(const Array&) = **delete**;

- Though you can delete any member function, it's most commonly used with member functions that the compiler can auto-generate—*the default constructor, copy constructor, assignment operator*.

Operators as members VS non-member

Whether an operator function is implemented as a member function or as a non-member function, *the operator is still used the same way in expressions*. So which is best?

- 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 operator function must be implemented as a non-member function.
- A non-member operator function can be made a **friend** of a class if that function must access private or protected members of that class directly.
- The same operator function cannot be implemented **both** as a member and non-member function:

```
Integer Integer::operator+(const Integer& rhs);
Integer operator+(const Integer& lhs, const Integer& rhs);
```

  Compiler error: Use of overloaded operator '+' is ambiguous.

- Operator member functions of a specific class are called (implicitly by the compiler) **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.

Operators as members VS non-member

Another reason why you might choose a non-member function to overload an operator is to enable the operator to be **commutative**.

Suppose we want to support the following expressions for **int** and **Integer** class:

- Integer + Integer
- Integer + int
- int + Integer

Thus, we require the addition operator to be ***commutative.***

- The problem is that the class object must appear on the **left** of the addition operator if that operator is to be overloaded as a **member function.**
- So, we also need to overload the operator as a **non-member function** to allow the int+Integer syntax.
- The "Integer Integer::operator+(const Integer& rhs)" can be member or non-member function.
- The "Integer Integer::operator+(int rhs)" can be member or non-member function.
- The "Integer operator+(int lhs, const Integer& rhs)" should be non-member function and can **simply** call 1) operator**+(Integer lhs, int rhs)** or
  2) **"rhs+lhs"** by calling member function "Integer Integer::operator+(int rhs)".

Operators as members VS non-member

```
class Integer {
public:
   Integer(int i=0)
      : _i(i)
   {
      std::cout << "Integer::Integer(" << _i << ") called" << endl;
   }

   Integer operator+(const Integer& rhs) const {
      std::cout << "Integer::operator+(const Integer& rhs) called" << endl;
      return Integer(_i + rhs._i);
   }

private:
   int _i;
};
```

- Note!: **const** function allows “const_Integer + Integer” expression.

```
int main()
{
    Integer i1(1);
    Integer i2(2);
    Integer i3(0);

    i3 = i1 + i2; // OK
    i2 = i1 + 10; // OK
//    i2 = 10 + i1; // Compile Error
}
```

Result:

```
Integer::Integer(1) called
Integer::Integer(2) called
Integer::Integer(0) called

Integer::operator+(const Integer& rhs) called
Integer::Integer(3) called

Integer::Integer(10) called
Integer::operator+(const Integer& rhs) called
Integer::Integer(11) called
```

Operators as members VS non-member

```
class Integer {
   friend Integer operator+(int lhs, const Integer& rhs);
public:
…
   Integer operator+(const Integer& rhs) const {
      std::cout << "Integer::operator+(const Integer& rhs) called" << endl;
      return Integer(_i + rhs._i);
   }
…
};

Integer operator+(int lhs, const Integer& rhs)
{
   std::cout << "operator+(int, const Integer&) called" << endl;
   return rhs + lhs;
}
```

```
int main()
{
   Integer i1(1);
   Integer i2(2);
   Integer i3(0);

   i3 = i1 + i2; // OK
   i2 = i1 + 10; // OK
   i2 = 10 + i1; // OK
}
```

Result:

```
Integer::Integer(1) called
Integer::Integer(2) called
Integer::Integer(0) called

Integer::operator+(const Integer& rhs) called
Integer::Integer(3) called

Integer::Integer(10) called
Integer::operator+(const Integer& rhs) called
Integer::Integer(11) called

operator+(int, const Integer&) called
Integer::Integer(10) called
Integer::operator+(const Integer& rhs) called
Integer::Integer(11) called
```

Or we could write only the non-member operator+(Integer, Integer):

```
class Integer {
   friend Integer operator+(const Integer& lhs, const Integer& rhs);
public:
 Integer(int i=0)
       : _i(i)
   {
      std::cout << "Integer::Integer(" << _i << ") called" << endl;
   }
private:
   int _i;
};

Integer operator+(const Integer& lhs, const Integer& rhs)
{
   std::cout << "operator+(Integer, Integer) called" << endl;
   return Integer(lhs._i + rhs._i);
}
```

And all 3 versions will work!

```
int main()
{
    Integer i1(1);
    Integer i2(2);
    Integer i3(0);

    i3 = i1 + i2; // OK
    i2 = i1 + 10; // OK
    i2 = 10 + i1; // OK
}
```

Result:

```
Integer::Integer(1) called
Integer::Integer(2) called
Integer::Integer(0) called

operator+(Integer, Integer) called
Integer::Integer(3) called

Integer::Integer(10) called
operator+(Integer, Integer) called
Integer::Integer(11) called

Integer::Integer(10) called
operator+(Integer, Integer) called
Integer::Integer(11) called
```

Operators as members VS non-member

If we know that overriding all the possible combinations of the operator+ will bring **performance benefits**, then we should provide that versions.

For example, see std::string class operator+ functions defined in <string> header file:

1. string operator+ (const string& lhs, const string& rhs);
2. string operator+ (const string& lhs, const char* rhs);
3. string operator+ (const char* lhs, const string& rhs);
4. string operator+ (const string& lhs, char rhs);
5. string operator+ (char lhs, const string& rhs);

For example, if the function-2 of std::string will not be provided, then the string(const char* rhs) constructor will be called and the function-1 will do the + operation.

However, directly working with “const char* rhs” in **function-2 will save one string(const char* rhs) constructor call**.

Converting between types

The compiler knows how to perform certain conversions among fundamental types:

```
int i=456;
double d = i;
```

But what about user-defined types?

For example, how to convert std::string to Message object?

Such conversions can be performed with **conversion constructors**—constructors that can be called with a single argument (we'll refer to these as **single-argument constructors**).

Such constructors can turn objects of other types (including fundamental types) into objects of a particular class.

Converting between types

```
class Message {
   public:
   Message(std::string s) : _msg(s) {
      std::cout << "Message::Message(std::string) called" << std::endl;
   }
   std::string _msg;
};

void f(Message msg)
{
   std::cout << "f(Message) called: Message._msg = " << msg._msg << std::endl;
}

int main()
{
   std::string str("Hello World");
   f(str);
}
```

Result:

```
Message::Message(std::string) called
f(Message) called: Message._msg = Hello World
```

Conversion constructor Message(std::string) was used to *convert* std::string object to Message object.

Converting between types

But how to convert Message object to std::string object?

Conversion operator (also called a cast operator) also can be used to convert an object of one class to another type.

Such a conversion operator must be a **non-static member function**.

Message::operator string() const;

declares an overloaded cast operator function for converting an object of class **Message** into a temporary **string** object.

The operator function is declared **const** because it does *not* modify the original object.

The return type of an overloaded cast operator function **is implicitly the type to which the object is being converted**.

Converting between types

When the compiler sees the expression

static_cast<std::**string**>(**message**)

it generates the call

message.operator **string**()

One of the nice features of cast operators and conversion constructors is that, when necessary, **the compiler can call these functions *implicitly* to create *temporary* objects.**

```
class Message {
   public:
   Message(std::string s) : _msg(s) {
      std::cout << "Message::Message(std::string) called" << std::endl;
   }
   operator string()  const {
      std::cout << "Message::operator string() called" << std::endl;
      return _msg; // Note that the return type is implicitly defined as std::string
   }
   std::string _msg;
};

void f(std::string s)
{
   std::cout << "f(std::string) called: s=" << s << std::endl;
}

int main()
{
   Message msg("Hello World");
   f(msg);
}
```

Result:

```
Message::Message(std::string) called
Message::operator string() called
f(std::string) called: s=Hello World
```

Converting between types

- When a conversion constructor or conversion operator is used to perform an implicit conversion, C++ can apply **only one** implicit constructor or operator function call (i.e., a single user-defined conversion) to try to match the needs of another overloaded operator.

- **The compiler will not satisfy an overloaded operator's needs by performing a series of implicit, user-defined conversions**.

```
class A {
   public:
…
   operator int () const {
      return _i;
   }
   private:
   int _i;
};
```

```
class Message {
   public:
…
   operator A() const {
      return A(0);
   }
…
};
```

```
void g(int i)
{
   std::cout << "g(int) called: i=" << i << std::endl;
}

int main()
{
   Message msg("Hello World");
   g(msg);
}
```

Compile Error: cannot convert ‘Message’ to ‘int’

- Compiler will **NOT** be able to convert **Message->A->int.**

Explicit Constructors and Conversion Operators

Any constructor that can be called with a single argument and is not declared **explicit** can be used by the compiler to perform an implicit conversion.

The constructor's argument is converted to an object of the class in which the constructor is defined.

The **conversion is automatic -** a cast is not required.

- Unfortunately, the compiler might use implicit conversions in cases that you do not expect, resulting in ambiguous expressions that generate compilation errors or result in execution-time logic errors.

Explicit Constructors and Conversion Operators

~~explicit~~ Array(int = 10); // default constructor

```
3  #include <iostream>
4  #include "Array.h"
5  using namespace std;
6
7  void outputArray(const Array&); // prototype
8
9  int main() {
10    Array integers1{7}; // 7-element Array
11    outputArray(integers1); // output Array integers1
12    outputArray(3); // convert 3 to an Array and output Array's contents
13 }
14
15 // print Array contents
16 void outputArray(const Array& arrayToOutput) {
17    cout << "The Array received has " << arrayToOutput.getSize()
18       << " elements. The contents are:\n" << arrayToOutput << endl;
19 }
```

```
The Array received has 7 elements.
The contents are: 0  0  0  0  0  0  0

The Array received has 3 elements.
The contents are: 0  0  0
```

The compiler assumes the constructor is a *conversion constructor* and uses it to convert the argument 3 into a temporary Array object containing three elements (**but we would not want this to be happening**!).

Explicit Constructors and Conversion Operators

explicit Array(int = 10); // default constructor

```
3  #include <iostream>
4  #include "Array.h"
5  using namespace std;
6
7  void outputArray(const Array&); // prototype
8
9  int main() {
10    Array integers1{7}; // 7-element Array
11    outputArray(integers1); // output Array integers1
12    outputArray(3); // convert 3 to an Array and output Array's contents
13    outputArray(Array{3}); // explicit single-argument constructor call
14 }
15
16 // print Array contents
17 void outputArray(const Array& arrayToOutput) {
18    cout << "The Array received has " << arrayToOutput.getSize()
19       << " elements. The contents are:\n" << arrayToOutput << endl;
20 }
```

- Line 12: Complie Error: 'void outputArray(const Array &)': cannot convert argument 1 from 'int' to 'const Array &'.
- In line 13 we explicitly create an Array object.

❖ **Always** use the **explicit** keyword on single-argument constructors **unless** they're **intended** to be used as **conversion constructors**.

Explicit Constructors and Conversion Operators

[C++11]: **explicit Conversion Operators**

Just as you can declare single-argument constructors explicit, **you can declare conversion operators explicit** to prevent the compiler from using them to perform implicit conversions.

For example, the prototype

explicit Integer::operator string() const;

declares **Integer**'s **string** cast operator **explicit**, thus requiring you to invoke it explicitly with **static_cast**.

```
void f(std::string s)
{ … }
…
Integer i1(1);
f(i1); // Compile Error, since Integer::operator string() is explicit
f(static_cast<std::string>(i1)); // OK
```

Overloading the Function Call Operator ()

Overloading the **function call operator ()** is powerful, because functions can take an arbitrary number of comma-separated parameters.

In a customized String class, for example, you could overload this operator to select a **substring** from a String.

The overloaded function call operator **must be a non-static member function** and could be defined as

```
String String::operator()(size_t index, size_t length) const
```

In this case, it should be a **const** member function because obtaining a **substring** should not modify the original String object.

For example:
String s("Hello World");
cout << s(6, 5); // prints "World"

The compiler translates s(6, 5) as follows:

s(6,5) -> s.operator()(6, 5)

- Please see [Appendix] for String operator() implementation.

[Appendix] String class example

```
//  String.h
// String class definition.
#ifndef STRING_H
#define STRING_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 constructor
   String( const String & ); // copy constructor
   ~String(); // destructor

   const String &operator=( const String & ); // assignment operator
   const String &operator+=( const String & ); // concatenation operator
```

```
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 );
} // end function operator!=

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

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

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

   char &operator[]( int ); // subscript operator (modifiable lvalue)
   char operator[]( int ) const; // subscript operator (rvalue)
   String operator()( int, int = 0 ) const; // return a substring
   int getLength() const; // return string length
private:
   int length; // string length (not counting null terminator)
   char *sPtr; // pointer to start of pointer-based string

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

#endif
```

// **String.cpp**

```
#include "String.h" // String class definition

// conversion (and default) constructor converts char * to String
String::String( const char *s )
   : length( ( s != 0 ) ? strlen( s ) : 0 )
{
   cout << "Conversion (and default) constructor: " << s << endl;
   setString( s ); // call utility function
} // end String conversion constructor

// utility function called by constructors and operator=
void String::setString( const char *string2 )
{
   sPtr = new char[ length + 1 ]; // allocate memory

   if ( string2 != 0 ) // if string2 is not null pointer, copy contents
      strcpy( sPtr, string2 ); // copy literal to object
   else // if string2 is a null pointer, make this an empty string
      sPtr[ 0 ] = '\0'; // empty string
} // end function setString
```

```
// copy constructor
String::String( const String& copy )
   : length( copy.length )
{
   cout << "Copy constructor: " << copy.sPtr << endl;
   setString( copy.sPtr ); // call utility function
} // end String copy constructor

// Destructor
String::~String()
{
   cout << "Destructor: " << sPtr << endl;
   delete [] sPtr; // release pointer-based string memory
} // end ~String destructor
```

```
// overloaded = operator; avoids self assignment
const String &String::operator=( const String &right )
{
   cout << "operator= called" << endl;
   if ( &right != this ) // avoid self assignment
   {
      delete [] sPtr; // prevents memory leak
      length = right.length; // new String length
      setString( right.sPtr ); // call utility function
   } // end if
   else {
      cout << "Attempted assignment of a String to itself" << endl;
   }

   return *this; // enables cascaded assignments
} // end function operator=
```

```
// concatenate right operand to this object and store in this object
const String &String::operator+=( const String &right )
{
   size_t newLength = length + right.length; // new length
   char *tempPtr = new char[ newLength + 1 ]; // create memory
   strcpy( tempPtr, sPtr ); // copy sPtr
   strcpy( tempPtr + length, right.sPtr ); // copy right.sPtr
   delete [] sPtr; // reclaim old space
   sPtr = tempPtr; // assign new array to sPtr
   length = newLength; // assign new length to length
   return *this; // enables cascaded calls
} // end function operator+=
```

- char * **strcpy** (char * **destination**, const char * **source**)

Copies the C string pointed by **source** into the array pointed by **destination**, including the terminating null character

```
// is this String empty?
bool String::operator!() const
{
   return length == 0;
} // end function operator!

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

// Is this String less than right String?
bool String::operator<( const String &right ) const
{
   return strcmp( sPtr, right.sPtr ) < 0;
} // end function operator<
```

- int **strcmp** (const char * **str1**, const char * **str2**) – compares str1 with str2;

return value	indicates
<0	the first character that does not match has a lower value in ***ptr1*** than in ***ptr2***
0	the contents of both strings are equal
>0	the first character that does not match has a greater value in ***ptr1*** than in ***ptr2***

```
// return reference to character in String as a modifiable lvalue
char & String::operator[]( int subscript )
{
   // test for subscript out of range
   if ( subscript < 0 || subscript >= length )
   {
      cerr << "Error: Subscript " << subscript
           << " out of range" << endl;
      exit( 1 ); // terminate program
   } // end if

   return sPtr[ subscript ]; // non-const return; modifiable lvalue
} // end function operator[]
```

Returns char& to support indexation and assignment

```
String str("Char");
str[0] = 'c'; // OK
```

```
// return reference to character in String as rvalue
char String::operator[]( int subscript ) const
{
   // test for subscript out of range
   if ( subscript < 0 || subscript >= length )
   {
      cerr << "Error: Subscript " << subscript
           << " out of range" << endl;
      exit( 1 ); // terminate program
} // end if

   return sPtr[ subscript ]; // returns copy of this element
} // end function operator[]
```

Returns char to disable assignment

```
const String str("Char");
str[0] = 'c'; // Error
```

```
// return a substring beginning at index and of length subLength
String String::operator()( int index, int subLength ) const
{
   // if index is out of range or substring length < 0,
   // return an empty String object
   if ( index < 0 || index >= length || subLength < 0 )
      return ""; // converted to a String object automatically

   // determine length of substring
   int len;

   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 ];

   // 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 temporary array
   return tempString; // return copy of the temporary String
} // end function operator()
```

```
// return string length
int String::getLength() const
{
   return length;
} // end function getLength

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

// 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
} // end function operator>>
```

// main.cpp

```
#include "String.h"
int main()
{
   String s1( "happy" );
   String s2( " birthday" );
   String s3;

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

```
// test overloaded String empty (!) operator
   cout << "\n\nTesting !s3:" << endl;

   if ( !s3 )
   {
      cout << "s3 is empty; assigning s1 to s3;" << endl;
      s3 = s1; // test overloaded assignment
      cout << "s3 is \"" << s3 << "\"";
   } // end if

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

   // test conversion constructor
   cout << "\n\ns1 += \" to you\" yields" << endl;
   s1 += " to you"; // test conversion constructor
   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" option
   cout << "The substring of s1 starting at\n"
        << "location 15, s1(15), is: "
        << s1( 15 ) << "\n\n";

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

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

   // test assignment (=) operator with self-assignment
   cout << "assigning *s4Ptr to *s4Ptr" << endl;
   *s4Ptr = *s4Ptr; // test overloaded assignment
   cout << "*s4Ptr = " << *s4Ptr << endl;

   // test destructor
   delete s4Ptr;

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

   // 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;
} // end main
```

```
Conversion (and default) constructor: happy
Conversion (and default) constructor:  birthday
Conversion (and default) 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 (and default) constructor: to you
Destructor: to you
s1 = happy birthday to you

Conversion (and default) 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 (and default) constructor: to you
Copy constructor: to you
Destructor: to you
The substring of s1 starting at
location 15, s1(15), 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:
Error: Subscript 30 out of range
```