Operator Overloading

User-defined Operator Overloading

"operator" Functions

To overload an operator, you use a special function form called an *operator function*, in the form of `operatorΔ()`, where `Δ` denotes the operator to be overloaded:

return-type **operator**Δ(*parameter-list*)

For example, `operator+()` overloads the + operator; `operator<<()` overloads the << operator. Take note that `Δ` must be an existing C++ operator. You cannot create you own operator.

Example: Overloading '+' Operator for the Point Class as Member Function

In this example, we shall overload the `'+'` operator in the `Point` class to support addition of two `Point` objects. In other words, we can write `p3 = p1+p2`, where `p1`, `p2` and `p3` are `Point` objects, similar to the usual arithmetic operation. We shall construct a new `Point` instance `p3` for the sum, without changing the `p1` and `p2` instances.

```
/* The Point class Header file (Point.h) */
#ifndef POINT_H
#define POINT_H

class Point {
private:
   int x, y; // Private data members

public:
   Point(int x = 0, int y = 0); // Constructor
   int getX() const; // Getters
   int getY() const;
   void setX(int x); // Setters
   void setY(int y);
   void print() const;
   const Point operator+(const Point & rhs) const;
         // Overload '+' operator as member function of the class
};

#endif
```

Program Notes:

- We overload the + operator via a member function `operator+()`, which shall add this instance (left operand) with the `rhs` operand, construct a new instance containing the sum and and return it *by value*. We cannot return by reference a local variable created inside the function, as the local variable would be destroyed when the function exits.
- The `rhs` operand is passed by reference for performance.
- The member function is declared `const`, which cannot modify data members.
- The return value is declared `const`, so as to prevent it from being used as *lvalue*. For example, it prevents writing `(p1+p2) = p3`, which is meaningless and could be due to misspelling `(p1+p2) == p3`.

```
/* The Point class Implementation file (Point.cpp) */
#include "Point.h"
#include <iostream>
using namespace std;

// Constructor - The default values are specified in the declaration
Point::Point(int x, int y) : x(x), y(y) { } // Using initializer list

// Getters
int Point::getX() const { return x; }
int Point::getY() const { return y; }

// Setters
void Point::setX(int x) { this->x = x; }
void Point::setY(int y) { this->y = y; }

// Public Functions
void Point::print() const {
   cout << "(" << x << "," << y << ")" << endl;
}

// Member function overloading '+' operator
const Point Point::operator+(const Point & rhs) const {
   return Point(x + rhs.x, y + rhs.y);
}
```

Program Notes:

- The function allocates a new `Point` object with the sums of x's and y's, and returns this object by `const` value.

```
#include "Point.h"
#include <iostream>
using namespace std;

int main() {
```

```
    Point p1(1, 2), p2(4, 5);
    // Use overloaded operator +
    Point p3 = p1 + p2;
    p1.print();  // (1,2)
    p2.print();  // (4,5)
    p3.print();  // (5,7)

    // Invoke via usual dot syntax, same as p1+p2
    Point p4 = p1.operator+(p2);
    p4.print();  // (5,7)

    // Chaining
    Point p5 = p1 + p2 + p3 + p4;
    p5.print();  // (15,21)
}
```

Program Notes:

- You can invoke the overloaded operator via `p1+p2`, which will be translated into the dot operation `p1.operator+(p2)`.
- The + operator supports chaining (cascading) operations, as `p1+p2` returns a `Point` object.

Restrictions on Operator Overloading

- The overloaded operator must be an existing and valid operator. You cannot create your own operator such as ⊕.
- Certain C++ operators cannot be overloaded, such as `sizeof`, dot (`.` and `.*`), scope resolution (`::`) and conditional (`?:`).
- The overloaded operator must have at least one operands of the user-defined types. You cannot overload an operator working on fundamental types. That is, you can't overload the `'+'` operator for two `int`s (fundamental type) to perform subtraction.
- You cannot change the syntax rules (such as associativity, precedence and number of arguments) of the overloaded operator.

Overloading Operator via "`friend`" non-member function

Why can't we always use Member Function for Operator Overloading?

The member function `operatorΔ()` can only be invoked from an object via the dot operator, e.g., `p1.operatorΔ(p2)`, which is equivalent to `p1 Δ p2`. Clearly the left operand `p1` should be an object of that particular class. Suppose that we want to overload a binary operator such as `*` to multiply the object `p1` with an `int` literal, `p1*5` can be translated into `p1.operator*(5)`, but `5*p1` cannot be represented using member function. One way to deal with this problem is only allow user to write `p1*5` but not `5*p1`, which is not user friendly and break the rule of commutativity. Another way is to use a non-member function, which does not invoke through an object and dot operator, but through the arguments provided. For example, `5*p1` could be translated to `operator+(5, p1)`.

In brief, you cannot use member function to overload an operator if the left operand is not an object of that particular class.

"*friend*" Functions

A regular non-member function cannot directly access the private data of the objects given in its arguments. A special type of function, called `friends`, are allowed to access the private data.
A "friend" function of a class, marked by the keyword `friend`, is a function defined outside the class, yet its argument of that class has unrestricted access to all the class members (`private`, `protected` and `public` data members and member functions). Friend functions can enhance the performance, as they eliminate the need of calling public member functions to access the private data members.

Example: Overloading `<<` and `>>` Operators of `Point` class using non-member `friend` Functions

In this example, we shall overload `<<` and `>>` operators to support stream insertion and extraction of `Point` objects, i.e., `cout << aPoint`, and `cin >> aPoint`. Since the left operand is not a `Point` object (`cout` is an `ostream` object and `cin` is an `istream` object), we cannot use member function, but need to use non-member function for operator overloading. We shall make these functions `friends` of the `Point` class, to allow them to access the private data members directly for enhanced performance.

```
/* The Point class Header file (Point.h) */
#ifndef POINT_H
#define POINT_H

#include <iostream>

// Class Declaration
class Point {
private:
   int x, y;

public:
   Point(int x = 0, int y = 0);
   int getX() const; // Getters
```

```
   int getY() const;
   void setX(int x); // Setters
   void setY(int y);

   friend std::ostream & operator<<(std::ostream & out, const Point & point);
   friend std::istream & operator>>(std::istream & in, Point & point);
};

#endif
```

Program Notes:

- Friends are neither `public` or `private`, and can be listed anywhere within the class declaration.
- The `cout` and `cin` need to be passed into the function by reference, so that the function accesses the `cout` and `cin` directly (instead of a clone copy by value).
- We return the `cin` and `cout` passed into the function by reference too, so as to support cascading operations. For example, `cout << p1 << endl` will be interpreted as `(cout << p1) << endl`.
- In `<<`, the reference parameter `Point` is declared as `const`. Hence, the function cannot modify the `Point` object. On the other hand, in `>>`, the `Point` reference is non-const, as it will be modified to keep the input.
- We use fully-qualified name `std::istream` instead of placing a "`using namespace std;`" statement in the header. It is because this header could be included in many files, which would include the `using` statement too and may not be desirable.

```
/* The Point class Implementation file (Point.cpp) */
#include <iostream>
#include "Point.h"
using namespace std;

// Constructor - The default values are specified in the declaration
Point::Point(int x, int y) : x(x), y(y) { } // using member initializer list

// Getters
int Point::getX() const { return x; }
int Point::getY() const { return y; }

// Setters
void Point::setX(int x) { this->x = x; }
void Point::setY(int y) { this->y = y; }

ostream & operator<<(ostream & out, const Point & point) {
   out << "(" << point.x << "," << point.y << ")";  // access private data
   return out;
}
```

```
istream & operator>>(istream & in, Point & point) {
   cout << "Enter x and y coord: ";
   in >> point.x >> point.y;  // access private data
   return in;
}
```

Program Notes:

- The function definition does not require the keyword `friend`, and the *`ClassName`*`::` scope resolution qualifier, as it does not belong to the class.
- The `operator<<()` function is declared as a friend of `Point` class. Hence, it can access the private data members x and y of its argument `Point` directly. `operator<<()` function is NOT a friend of `ostream` class, as there is no need to access the private member of `ostream`.
- Instead of accessing private data member x and y directly, you could use public member function `getX()` and `getY()`. In this case, there is no need to declare `operator<<()` as a friend of the `Point` class. You could simply declare a regular function prototype in the header.

```
// Function prototype
ostream & operator<<(ostream & out, const Point & point);

// Function definition
ostream & operator<<(ostream & out, const Point & point) {
   out << "(" << point.getX() << "," << point.getY() << ")";
   return out;
}
```

Using `friend` is recommended, as it enhances performance. Furthermore, the overloaded operator becomes part of the extended public interface of the class, which helps in ease-of-use and ease-of-maintenance.

```
#include <iostream>
#include "Point.h"
using namespace std;

int main() {
   Point p1(1, 2), p2;

   // Using overloaded operator <<
   cout << p1 << endl;    // support cascading
   operator<<(cout, p1);  // same as cout << p1
   cout << endl;

   // Using overloaded operator >>
   cin >> p1;
```

```
    cout << p1 << endl;
    operator>>(cin, p1);  // same as cin >> p1
    cout << p1 << endl;
    cin >> p1 >> p2;      // support cascading
    cout << p1 << endl;
    cout << p2 << endl;
}
```

The overloaded >> and << can also be used for file input/output, as the file IO stream `ifstream/ofstream` (in `fstream` header) is a subclass of `istream/ostream`. For example,

```
#include <fstream>
#include "Point.h"
using namespace std;
int main() {
   Point p1(1, 2);
   ofstream fout("out.txt");
   fout << p1 << endl;

   ifstream fin("in.txt"); // contains "3 4"
   fin >> p1;
   cout << p1 << endl;
}
```

Overloading Binary Operators

All C++ operators are either *binary* (e.g., x + y) or *unary* (e.g. !x, -x), with the exception of *tenary* conditional operator (? :) which cannot be overloaded.
Suppose that we wish to overload the binary operator == to compare two `Point` objects. We could do it as a *member function* or *non-member function*.

Suppose that we wish to overload the binary operator == to compare two `Point` objects. We could do it as a *member function* or *non-member function*.

1. To overload as a *member function*, the declaration is as follows:

```
class Point {
public:
   bool operator==(const Point & rhs) const;  // p1.operator==(p2)
   ......
};
```

The compiler translates "`p1 == p2`" to "`p1.operator==(p2)`", as a member function call of object `p1`, with argument `p2`.

Member function can only be used if the left operand is an object of that particular class.

2. To overload as a *non-member function*, which is often declared as a `friend` to access the private data for enhanced performance, the declaration is as follows:

```
class Point {
   friend bool operator==(const Point & lhs, const Point & rhs); // operator==(p1,p2)
   ......
};
```

The compiler translates the expression "`p1 == p2`" to "`operator==(p1, p2)`".

Overloading Unary Operators

Most of the unary operators are prefix operators, e.g., `!x`, `-x`. Hence, prefix is the norm for unary operators. However, unary increment and decrement come in two forms: prefix (`++x`, `--x`) and postfix (`x++`, `x--`). We to a mechanism to differentiate the two forms.

Unary Prefix Operator

Example of unary prefix operators are `!x`, `-x`, `++x` and `--x`. You could do it as a non-member function as well as member function. For example, to overload the prefix increment operator `++`:

1. To overload as a non-member `friend` function:

```
class Point {
   friend Point & operator++(Point & point);
   ......
};
```

The compiler translates "`++p`" to "`operator++(p)`".

2. To overload as a member function:

```
class Point {
public:
   Point & operator++();  // this Point
   ......
};
```

The compiler translates "`++p`" to "`p.operator++()`".

You can use either member function or non-member friend function to overload unary operators, as their only operand shall be an object of that class.

Unary Postfix Operator

The unary increment and decrement operators come in two forms: prefix (++x, --x) and postfix (x++, x--). Overloading postfix operators (such as x++, x--) present a challenge. It ought to be differentiated from the prefix operator (++x, --x). A "dummy" argument is therefore introduced to indicate postfix operation as shown below. Take note that postfix ++ shall save the old value, perform the increment, and then return the saved value by value.

1. To overload as non-member `friend` function:

```
class Point {
   friend const Point operator++(Point & point, int dummy);
};
```

   The compiler translates "`pt++`" to "`operator++(pt, 0)`". The `int` argument is strictly a *dummy value* to differentiate prefix from postfix operation.

2. To overload as a member function:

```
class Point {
public:
   const Point operator++(int dummy);  // this Point
   ......
};
```

   The compiler translates "`pt++`" to "`pt.operator++(0)`".

Example: Overloading Prefix and Postfix ++ for the `Counter` Class

```
/* The Counter class Header file (Counter.h) */
#ifndef COUNTER_H
#define COUNTER_H
#include <iostream>

class Counter {
private:
   int count;
public:
   Counter(int count = 0);   // Constructor
   int getCount() const;     // Getters
   void setCount(int count); // Setters
   Counter & operator++();               // ++prefix
   const Counter operator++(int dummy); // postfix++

   friend std::ostream & operator<<(std::ostream & out, const Counter & counter);
};
```

```
#endif
```

Program Notes:

- The prefix function returns a reference to this instance, to support chaining (or cascading), e.g., ++++c as ++(++c). However, the return reference can be used as lvalue with unexpected operations (e.g., ++c = 8).
- The postfix function returns a const object by value. A const value cannot be used as lvalue. This prevents chaining such as c++++. Although it would be interpreted as (c++)++. However, (c++) does not return this object, but an temporary object. The subsequent ++ works on the temporary object.
- Both prefix and postfix functions are non-const, as they modify the data member count.

```
/* The Counter class Implementation file (Counter.cpp) */
#include "Counter.h"
#include <iostream>
using namespace std;

// Constructor - The default values are specified in the declaration
Counter::Counter(int c) : count(c) { } // using member initializer list

// Getters
int Counter::getCount() const { return count; }

// Setters
void Counter::setCount(int c) { count = c; }

// ++prefix, return reference of this
Counter & Counter::operator++() {
   ++count;
   return *this;
}

// postfix++, return old value by value
const Counter Counter::operator++(int dummy) {
   Counter old(*this);
   ++count;
   return old;
}
// Overload stream insertion << operator
ostream & operator<<(ostream & out, const Counter & counter) {
   out << counter.count;
   return out;
}
```

Program Notes:

- The prefix function increments the `count`, and returns this object by reference.
- The postfix function saves the old value (by constructing a new instance with this object via the copy constructor), increments the `count`, and return the saved object by value.
- Clearly, postfix operation on object is less efficient than the prefix operation, as it create a temporary object. If there is no subsequent operation that relies on the output of prefix/postfix operation, use prefix operation.

```
#include "Counter.h"
#include <iostream>
using namespace std;

int main() {
   Counter c1;
   cout << c1 << endl;      // 0
   cout << ++c1 << endl;    // 1
   cout << c1 << endl;      // 1
   cout << c1++ << endl;    // 1
   cout << c1 << endl;      // 2
 }
```

Program Notes:

- Take note of the difference in `cout << c1++` and `cout << ++c1`. Both prefix and postfix operators work as expected.

Example: Putting them together in `Point` Class

This example overload binary operator << and >> as non-member functions for stream insertion and stream extraction. It also overload unary ++ (postfix and prefix) and binary += as member function; and +, += operators.

`Point.h`

```
/* The Point class Header file (Point.h) */
#ifndef POINT_H
#define POINT_H
#include <iostream>

class Point {
private:
   int x, y;

public:
   explicit Point(int x = 0, int y = 0);
   int getX() const;
```

```
   int getY() const;
   void setX(int x);
   void setY(int y);
   Point & operator++();              // ++prefix
   const Point operator++(int dummy); // postfix++
   const Point operator+(const Point & rhs) const; // Point + Point
   const Point operator+(int value) const;           // Point + int
   Point & operator+=(int value);           // Point += int
   Point & operator+=(const Point & rhs); // Point += Point

   friend std::ostream & operator<<(std::ostream & out, const Point & point); // out << po
   friend std::istream & operator>>(std::istream & in, Point & point);        // in >> poi
   friend const Point operator+(int value, const Point & rhs); // int + Point
};

#endif
```

Point.cpp

```
/* The Point class Implementation file (Point.cpp) */
#include "Point.h"
#include <iostream>
using namespace std;

// Constructor - The default values are specified in the declaration
Point::Point(int x, int y) : x(x), y(y) { }

// Getters
int Point::getX() const { return x; }
int Point::getY() const { return y; }

// Setters
void Point::setX(int x) { this->x = x; }
void Point::setY(int y) { this->y = y; }

// Overload ++Prefix, increase x, y by 1
Point & Point::operator++() {
   ++x;
   ++y;
   return *this;
}

// Overload Postfix++, increase x, y by 1
const Point Point::operator++(int dummy) {
   Point old(*this);
   ++x;
```

```
  ++y;
  return old;
}

// Overload Point + int. Return a new Point by value
const Point Point::operator+(int value) const {
  return Point(x + value, y + value);
}

// Overload Point + Point. Return a new Point by value
const Point Point::operator+(const Point & rhs) const {
  return Point(x + rhs.x, y + rhs.y);
}

// Overload Point += int. Increase x, y by value
Point & Point::operator+=(int value) {
  x += value;
  y += value;
  return *this;
}

// Overload Point += Point. Increase x, y by rhs
Point & Point::operator+=(const Point & rhs) {
  x += rhs.x;
  y += rhs.y;
  return *this;
}

// Overload << stream insertion operator
ostream & operator<<(ostream & out, const Point & point) {
  out << "(" << point.x << "," << point.y << ")";
  return out;
}

// Overload >> stream extraction operator
istream & operator>>(istream & in, Point & point) {
  cout << "Enter x and y coord: ";
  in >> point.x >> point.y;
  return in;
}

// Overload int + Point. Return a new point
const Point operator+(int value, const Point & rhs) {
  return rhs + value;  // use member function defined above
}
```

TestPoint.cpp

```
#include <iostream>
#include "Point.h"
using namespace std;

int main() {
   Point p1(1, 2);
   cout << p1 << endl;   // (1,2)

   Point p2(3,4);
   cout << p1 + p2 << endl; // (4,6)
   cout << p1 + 10 << endl; // (11,12)
   cout << 20 + p1 << endl; // (21,22)
   cout << 10 + p1 + 20 + p1 << endl; // (32,34)

   p1 += p2;
   cout << p1 << endl; // (4,6)
   p1 += 3;
   cout << p1 << endl; // (7,9)

   Point p3;  // (0,0)
   cout << p3++ << endl; // (0,0)
   cout << p3 << endl;   // (1,1)
   cout << ++p3 << endl; // (2,2)
}
```

Dynamic Memory Allocation in Object

If you dynamically allocate memory in the constructor, you need to provide your own destructor, copy constructor and assignment operator to manage the dynamically allocated memory. The defaults provided by the C++ compiler do not work for dynamic memory.

Example: `MyDynamicArray`

```
/*
 * The MyDynamicArray class header (MyDynamicArray.h)
 * A dynamic array of double elements
 */
#ifndef MY_DYNAMIC_ARRAY_H
#define MY_DYNAMIC_ARRAY_H

#include <iostream>
```

```
class MyDynamicArray {
private:
   int size_;   // size of array
   double * ptr;  // pointer to the elements

public:
   explicit MyDynamicArray (int n = 8);         // Default constructor
   explicit MyDynamicArray (const MyDynamicArray & a); // Copy constructor
   MyDynamicArray (const double a[], int n);    // Construct from double[]
   ~MyDynamicArray();                           // Destructor

   const MyDynamicArray & operator= (const MyDynamicArray & rhs); // Assignment a1 = a2
   bool operator== (const MyDynamicArray & rhs) const;     // a1 == a2
   bool operator!= (const MyDynamicArray & rhs) const;     // a1 != a2

   double operator[] (int index) const;  // a[i]
   double & operator[] (int index);      // a[i] = x

   int size() const { return size_; }    // return size of array

   // friends
   friend std::ostream & operator<< (std::ostream & out, const MyDynamicArray & a); // out <<
   friend std::istream & operator>> (std::istream & in, MyDynamicArray & a);        // in >> a
};

#endif
```

Program Notes:

- In C++, the you cannot use the same name for a data member and a member function. As I would like to have a public function called `size()`, which is consistent with the C++ STL, I named the data member `size_` with a trailing underscore, following C++'s best practices. Take note that leading underscore(s) are used by C++ compiler for its internal variables (e.g., _xxx for data members and __xxx for local variables).
- As we will be dynamically allocating memory in the constructor, we provide our own version of destructor, copy constructor and assignment operator to manage the dynamically allocated memory. The defaults provided by the C++ compiler do not work on dynamic memory.
- We provide 3 constructors: a default constructor with an optional size, a copy constructor to construct an instance by copying another instance, and a construct to construct an instance by copying from a regular array.
- We provide 2 version of indexing operators: one for read operation (e.g., `a[i]`) and another capable of write operation (e.g., `a[i] = x`). The read version is declared as a `const` member function; whereas the write version return a reference to the element, which can be used as *lvalue* for assignment.

```cpp
/* The MyDynamicArray class implementation (MyDynamicArray.cpp) */
#include <stdexcept>
#include "MyDynamicArray.h"

// Default constructor
MyDynamicArray::MyDynamicArray (int n) {
   if (n <= 0) {
      throw std::invalid_argument("error: size must be greater then zero");
   }

   // Dynamic allocate memory for n elements
   size_ = n;
   ptr = new double[size_];
   for (int i = 0; i < size_; ++i) {
      ptr[i] = 0.0;  // init all elements to zero
   }
}

// Override the copy constructor to handle dynamic memory
MyDynamicArray::MyDynamicArray (const MyDynamicArray & a) {
   // Dynamic allocate memory for a.size_ elements and copy
   size_ = a.size_;
   ptr = new double[size_];
   for (int i = 0; i < size_; ++i) {
      ptr[i] = a.ptr[i];  // copy each element
   }
}

// Construct via a built-in double[]
MyDynamicArray::MyDynamicArray (const double a[], int n) {
   // Dynamic allocate memory for a.size_ elements and copy
   size_ = n;
   ptr = new double[size_];
   for (int i = 0; i < size_; ++i) {
      ptr[i] = a[i];  // copy each element
   }
}

// Override the default destructor to handle dynamic memory
MyDynamicArray::~MyDynamicArray() {
   delete[] ptr;  // free dynamically allocated memory
}

// Override the default assignment operator to handle dynamic memory
```

```
const MyDynamicArray & MyDynamicArray::operator= (const MyDynamicArray & rhs)
{
   if (this != &rhs) {  // no self assignment
      if (size_ != rhs.size_) {
         // reallocate memory for the array
         delete [] ptr;
         size_ = rhs.size_;
         ptr = new double[size_];
      }
      // Copy elements
      for (int i = 0; i < size_; ++i) {
         ptr[i] = rhs.ptr[i];
      }
   }
   return *this;
}

// Overload comparison operator a1 == a2
bool MyDynamicArray::operator== (const MyDynamicArray & rhs) const {
   if (size_ != rhs.size_) return false;

   for (int i = 0; i < size_; ++i) {
      if (ptr[i] != rhs.ptr[i]) return false;
   }
   return true;
}

// Overload comparison operator a1 != a2
bool MyDynamicArray::operator!= (const MyDynamicArray & rhs) const {
   return !(*this == rhs);
}

// Indexing operator - Read
double MyDynamicArray::operator[] (int index) const {
   if (index < 0 || index >= size_) {
      throw std::out_of_range("error: index out of range");
   }
   return ptr[index];
}

// Indexing operator - Writable a[i] = x
double & MyDynamicArray::operator[] (int index) {
   if (index < 0 || index >= size_) {
      throw std::out_of_range("error: index out of range");
   }
```

```
   return ptr[index];
}

// Overload stream insertion operator out << a (as friend)
std::ostream & operator<< (std::ostream & out, const MyDynamicArray & a) {
   for (int i = 0; i < a.size_; ++i) {
      out << a.ptr[i] << ' ';
   }
   return out;
}

// Overload stream extraction operator in >> a (as friend)
std::istream & operator>> (std::istream & in, MyDynamicArray & a) {
   for (int i = 0; i < a.size_; ++i) {
      in >> a.ptr[i];
   }
   return in;
}
```

Program Notes:

- Constructor: [TODO]
- Copy Constructor:
- Assignment Operator:
- Indexing Operator:

```
/* Test Driver for MyDynamicArray class (TestMyDynamicArray.cpp) */
#include <iostream>
#include <iomanip>
#include "MyDynamicArray.h"

int main() {
   std::cout << std::fixed << std::setprecision(1) << std::boolalpha;

   MyDynamicArray a1(5);
   std::cout << a1 << std::endl;  // 0.0 0.0 0.0 0.0 0.0
   std::cout << a1.size() << std::endl;  // 5

   double d[3] = {1.1, 2.2, 3.3};
   MyDynamicArray a2(d, 3);
   std::cout << a2 << std::endl; // 1.1 2.2 3.3

   MyDynamicArray a3(a2);   // Copy constructor
   std::cout << a3 << std::endl; // 1.1 2.2 3.3
```

```
    a1[2] = 8.8;
    std::cout << a1[2] << std::endl;  // 8.8
//  std::cout << a1[22] << std::endl; // error: out_of_range

    a3 = a1;
    std::cout << a3 << std::endl; // 0.0 0.0 8.8 0.0 0.0

    std::cout << (a1 == a3) << std::endl;  // true
    std::cout << (a1 == a2) << std::endl;  // false

    const int SIZE = 3;
    MyDynamicArray a4(SIZE);
    std::cout << "Enter " << SIZE << " elements: ";
    std::cin >> a4;
    if (std::cin.good()) {
       std::cout << a4 << std::endl;
    } else {
       std::cerr << "Invalid input" << std::endl;
    }
    return 0;
}
```