Перегрузка операций

Для чего нужна перегрузка операций

- Для некоторых типов данных естественными может оказаться использование операций над базовыми типами
  - += и + для конкатенации строк
  - -- и ++ для итераторов
  - арифметические операции для векторов и комплексных чисел
  - -> и * для **умных указателей**
  - [] для массивов и ассоциативных контейнеров
  - () для функторов (объектов функций)
  - = для классов с собственным конструктором копирования
  - операции сравнения для строк и других типов

Перегрузка операций

- Для пользовательских типов данных C++ позволяет задать собственные операции
  - Некоторые из них **всегда** определяются внутри класса
    - =, +=, -=, *= и т.п.
  - Некоторые – снаружи
    - Как правило, операции, в которых применяются базовые типы
- Синтаксис:
  - **<тип> operator X(параметры)**

Ограничения

- Приоритет операций над пользовательскими типами тот же, что и для базовых типов
- Нельзя переопределить операцию точка (.) и sizeof
- Бинарные операции остаются бинарными
- Унарные - унарными

Пример 1

Моделирование класса «Двухмерный вектор»

Информация о предметной области

- Для двухмерных векторов определены операции сложения, вычитания и умножения на скаляр, а также операции проверки на равенство (и неравенство)
  - При моделировании класса векторов весьма удобно будет перегрузить соответствующие им арифметические операции
- При перегрузке операций следует руководствоваться:
  - Особенностями данных операций в предметной области
  - Архитектурой класса
  - Требованиями и ограничениями языка C++
  - Здравым смыслом

Каркас класса CVector2D

```
class CVector2D
{
public:
    CVector2D(double x0 = 0, double y0 = 0)
        :x(x0), y(y0)
    {
    }

    // методы и операции над векторами

    // данные объявляем публичными, т.к. в данном конкретном
случае
    // нет необходимости создавать для них методы
    double  x, y;
};
```

Перегрузка оператора сложения векторов

- При перегрузке данного оператора принимаем во внимание следующие особенности
  - Оператор сложения является бинарным оператором
    - Оба аргумента оператора сложения являются **двухмерными векторами**, значения которых **не изменяются** во время его выполнения
      - Имеет смысл передавать по константной ссылке
  - Оператор сложения векторов возвращает **новый константный вектор**, координаты которого – суммы соответствующих координат аргументов
    - **Константность обязательна**, чтобы не допустить конструкции вида:
      (vector1 + vector2) = vector3;
      (для целых и действительных чисел данная операция также запрещена)
- Оператор сложения векторов можно реализовать тремя способами:
  - Как оператор, объявленный внутри класса
    - В этом случае **левым аргументом** оператора будет являться **текущий экземпляр класса**, а **правый аргумент** будет передаваться через **единственный параметр**
  - Как оператор, объявленный вне класса
    - В этом случае оператор будет принимать два аргумента
  - Как **дружественный** оператор, объявленный вне класса
    - Отличается от предыдущего способа **возможностью доступа к приватным и защищенным** методам и данным класса

Реализация оператора сложения внутри класса CVector2D

```
class CVector2D
{
public:
    …
    CVector2D const operator +(CVector2D const& vector2)const
    {
        // левым аргументом является текущий экземпляр класса
        // правым – единственный аргумент vector2
        return CVector2D(x + vector2.x, y + vector2.y);
    }
    …
};
```

Реализация оператора сложения вне класса CVector2D

```
class CVector2D
{
public:
    …
    …
};

CVector2D const operator +(CVector2D const& vector1,
    CVector2D const& vector2)
{
    return CVector2D(vector1.x + vector2.x, vector1.y +
vector2.y);
}
```

Реализация дружественного оператора сложения

```
class CVector2D
{
public:
    …
    CVector2D const friend operator +(CVector2D const& vector1,
                                      CVector2D const& vector2);

    …
};

CVector2D const operator +(CVector2D const& vector1,
    CVector2D const& vector2)
{
    return CVector2D(vector1.x + vector2.x, vector1.y +
vector2.y);
}
```

Выбор предпочтительного способа перегрузки

- В данном случае предпочтительным способом является реализация оператора сложения **внутри класса**
  - Естественность данного оператора для класса векторов
  - Возможность внесения изменений в исходный код класса CVector2D
  - Наиболее краткая форма записи

Пример использования перегруженного оператора +

```
class CVector2D
{
public:
    …
    CVector2D const operator +(CVector2D const& vector2)const;
    …
};

…

int main(int argc, char * argv[])
{
    CVector2D a(3.0, 5.8);
    CVector2D b(7.3, 8.8);

    CVector2D c = a + b + CVector2D(3, 9);

    return 0;
}
```

Реализация оператора вычитания векторов

- В данном случае оператор сложения практически полностью идентичен оператору сложения
- Предпочитаемый способ перегрузки – реализация также внутри класса CVector2D

```
class CVector2D
{
public:
    …
    CVector2D const operator -(CVector2D const& vector2)const
    {
        // левым аргументом является текущий экземпляр класса
        // правым – единственный аргумент vector2
        return CVector2D(x - vector2.x, y - vector2.y);
    }
    …
};
```

Перегрузка оператора умножения вектора и скаляра

- Умножение вектора на скаляр – более сложная операция, т.к. использует **разные типы аргументов** и является **коммутативной**
  - Один из аргументов – вектор (CVector2D), второй – скаляр (double)
  - Из-за коммутативности данной операции существуют 2 версии данного оператора:
    - Вектор * Скаляр
    - Скаляр * Вектор

Перегрузка оператора произведения вектора и скаляра

- При перегрузке данного оператора принимаем во внимание следующие особенности
  - Данный оператор является бинарным оператором с параметрами различных типов
  - Оператор возвращает **новый константный вектор**, координаты которого – произведения координат исходного вектора на скаляр
    - **Константность обязательна**, чтобы не допустить конструкции вида:
      (vector1 * 3) = vector2;
      (10 * vector3) = vector4;
- Существуют две версии данного оператора
  - Операция умножения вектора на скаляр
    - В нашем случае перегружается внутри класса аналогично оператору сложения и вычитания
  - Операция умножения скаляра на вектор
    - Данная операция **не может быть перегружена внутри класса**, т.к. левый аргумент (скаляр) имеет тип, отличный от класса CVector2D
    - В данном случае следует перегрузить его обычным образом вне класса
      - Нет необходимости в создании дружественной операции, т.к. операции не требуется доступ к приватным данным и методам класса CVector2D

Реализация оператора произведения вектора и скаляра

```
class CVector2D
{
public:
    …
    CVector2D const operator *(double scalar)const
    {
        // левым аргументом является текущий экземпляр класса
        // правым – единственный аргумент vector2
        return CVector2D(x * scalar, y * scalar);
    }

    …
};

CVector2D const operator *(double scalar,
    CVector2D const& vector)
{
    return CVector2D(scalar * vector.x, scalar *vector.y);
}
```

Пример использования

```
#include “Vector2D.h”

int main(int argc, char * argv[])
{
    CVector2D a(3.0, 2.1);
    CVector2D b(4.0, 5.1);

    CVector2D c = a * 3.4;
    CVector2D d = 8.4 * b;

    CVector2D e = (a + b) * 3 + (c – d) * 4;

    return 0;
}
```

Реализация оператора деления вектора на скаляр

- Для векторов также определена операция деления вектора на скаляр
  - Результатом данной операции является вектор с координатами, равными отношению координат исходного вектора к скаляру
  - Данная операция перегружается внутри класса аналогично операции умножения вектора на скаляр

```
class CVector2D
{
public:
    …
    CVector2D const operator /(double scalar)const
    {
        return CVector2D(x / scalar, y / scalar);
    }
    …
};
```

Перегрузка присваивающих выражений

- Помимо операций +, - и * могут понадобиться данные действия в составе операции присваивания:
  - vector1 += vector2;
  - vector3 *= 3.8;
  - vector4 -= vector1;
- Особенностью данных операций является то, что они **модифицируют операнд в левой части**, но не модифицируют операнд в правой
  - Кроме того, важно, чтобы они возвращали **ссылку на левый операнд**, чтобы можно было использовать выражения, допустимые для встроенных типов данных:
    - (a += b) /= c;

Реализация оператора +=

```
class CVector2D
{
public:
    …
    CVector2D& operator +=(CVector2D const& vector)
    {
        x += vector.x;
        y += vector.y;

        return *this;
    }
    // операторы *=, /=, -= перегружаются аналогичным образом
    …
};
```

Перегрузка операторов сравнения

- Операторы сравнения сравнивают значения операндов, не изменяя их, и возвращают результат типа bool, соответствующий результату сравнения
- Для двухмерных векторов такими операциями являются операторы:
  - ==
  - !=

Реализация операторов == и !=

```
class CVector2D
{
public:
    …
    bool operator ==(CVector2D const& other)const
    {
        return (x == other.x) && (y == other.y);
    }

    bool operator !=(CVector2D const& other)const
    {
        return (x != other.x) || (y != other.y);
    }
    …
};
```

Умные указатели

Что такое умный указатель?

- Умный указатель (**smart pointer**) – класс (обычно шаблонный), имитирующий интерфейс обычного указателя и добавляющий новую функциональность
  - Перегрузка операций * и ->, специфичных для простых указателей
  - Инкапсуляция семантики владения ресурсом для борьбы с утечками памяти и «висячими» ссылками

Исходный код класса CMyClassPtr

```
class CMyClassPtr
{
public:
     CMyClassPtr(CMyClass * pMyClass):m_pMyClass(pMyClass){}
     // деструктор автоматически удаляет управляемый объект
     ~CMyClassPtr(){delete m_pMyClass;}

     CMyClass* operator->()
     {
          assert(m_pMyClass != NULL);
          return m_pMyClass;
     }

     CMyClass& operator*()
     {
          assert(m_pMyClass != NULL);
          return * m_pMyClass;
     }
private:
     // запрещаем копирование и присваивание указателей CMyClassPtr
     CMyClassPtr(CMyClassPtr const&);
     CMyClassPtr& operator=(CMyClassPtr const&);

     CMyClass *m_pMyClass;
};
```

Пример использования класса CMyClassPtr

```
class CMyClass
{
public:
     void DoSomethingImportant()
     {
          …
     }
};

class CMyClassPtr
{
     …
};

int main(int argc, char * argv[])
{
     CMyClassPtr pMyClass(new CMyClass());
     if (…)
     {
          pMyClass->DoSomethingImportant();
     }
     else
     {
          return 1; // нет нужды в вызове delete (это сделает умный указатель CMyClassPtr)
     }
     return 0;
}
```

Стандартные умные указатели

- Библиотека STL содержит шаблонынй класс auto_ptr, обеспечивающий политику владения объектом в динамической памяти
  - Недостаток: нельзя использовать в составе контейнеров STL (а также во многих других контейнерах)
- Библиотека boost предлагает шаблонные классы shared_ptr, scoped_ptr и intrusive_ptr, предоставляющие различные способы владения и управления объектом
  - shared_ptr, например, основывается на подсчете количества ссылок на объект и может использоваться в составе контейнеров STL

Перегрузка унарного плюса и минуса

Унарный плюс и унарный минус

- Помимо инфиксных операций бинарного плюса и бинарного минуса есть их унарные префиксные версии
- Их также можно при желании перегрузить (всеми тремя способами)
  - Наиболее предпочтительный – перегрузка внутри класса
    - В этом случае текущий экземпляр класса считается аргументом данного оператора

Пример перегрузки унарного минуса

```
class CVector2D
{
public:
    …
    CVector2D const operator –()const;
    {
        return CVector2D(-x, -y);
    }

    CVector2D const operator +()const;
    {
        // возвращаем копию
        return *this;
    }

    …
};
```

Перегрузка оператора присваивания

Автоматически сгенерированный оператор присваивания

- Оператор присваивания, как и конструктор копирования может быть автоматически сгенерирован компилятором в случае необходимости
  - Автоматически сгенерированный оператор присваивания выполняет вызов операторов присваивания для всех своих полей, а также родительского класса (в случае его наличия родителя)
- В ряде случаев компилятор не может сгенерировать оператор присваивания
  - Класс содержит ссылки или константы
  - В родительском классе оператор присваивания объявлен приватным

Когда нужен собственный оператор присваивания?

- Как правило, во всех случаях, когда классу нужен собственный конструктор копирования
  - Создание копии не сводится к обычному копированию полей класса
- Оператор присваивания должен возвращать ссылку на левый операнд, чтобы были возможны следующие выражения, допустимые для встроенных типов:
  - if ((a = b) == c) {…}
- Оператор присваивания должен корректно обрабатывать некоторые особенные ситуации
  - Например, присваивание самому себе не должно приводить к порче данных
  - Наиболее надежный способ – использовать конструктор копирования для создания копии

Пример некорректной реализации присваивания строк

```
class CMyString
{
public:
    …
    CMyString& operator =(CMyString const& other)
    {
        delete [] m_pChars;
        m_pChars = new char[other.m_length + 1];
        memcpy(m_pChars, other.m_pChars, m_length + 1);
        m_length = other.m_length;
        return *this;
    }
    …
private:
    char * m_pChars;
    size_t m_length;
};
```

Некорректная работа оператора в случае самоприсваивания:

CMyString s(“some string”);
s = s;

Пример корректной реализации присваивания строк

```
class CMyString
{
public:
    …
    CMyString& operator =(CMyString const& other)
    {
        if (&other != this)
        {
            CMyString tmpCopy(other);
            std::swap(m_pChars, tmpCopy.m_pChars);
            std::swap(m_length, tmpCopy.m_length);
        }
        return *this;
    }
    // сходным образом перегружаем операторы
    CMyString& operator +=(CMyString const& other);
    CMyString& operator =(const char* pChars);
    CMyString& operator +=(const char* pChars);
    …
private:
    char * m_pChars;
    size_t m_length;
};
```

Запрет операции присваивания

- В ряде случае операция присваивания объектов может быть нежелательной
  - С экземпляром объекта связываются какие-то внешние объекты, например, файловый дескриптор или сетевое соединение
- Операцию присваивания для объектов можно запретить, объявив оператор присваивания в приватной области класса
  - Реализацию можно при этом не писать

Перегрузка оператора индексации []

Оператор индексации

- Является унарным оператором, обычно использующимся для доступа к элементам контейнера
  - В качестве типа индекса может использоваться произвольный тип
- Поскольку доступ к элементам может быть как на чтение, так и на запись, существуют две формы данного оператора
  - Оператор доступа для чтения является константным и возвращает константу или константную ссылку на элемент контейнера
  - Оператор доступа для записи является неконстантным и возвращает ссылку на элемент контейнера
- Программист может перегрузить данный оператор иными способами, однако это может ввести в заблуждение других программистов

Пример: доступ к символам строки

```
class CMyString
{
public:
    …
    // оператор индексированного доступа для чтения
    const char operator[](unsigned index)const
    {
        assert(index < m_length);
        return m_pChars[index];
    }

    // оператор индексированного доступа для записи
    char & operator[](unsigned index)
    {
        assert(index < m_length);
        return m_pChars[index];
    }
    …
private:
    char * m_pChars;
    size_t m_length;
};
```

Перегрузка операций инкремента и декремента

Особенности перегрузки операторов инкремента и декремента

- Для некоторых типов данных могут быть определены операции инкремента и декремента
  - Итераторы, счетчики
- Операторы инкремента и декремента являются унарными операциями
- Префиксные и постфиксные версии данных операторов имеют различную семантику и перегружаются по-разному

Перегрузка префиксной формы инкремента и декремента

- **Префиксная** операция выполняет модификацию объекта и возвращает **ссылку** на измененное значение объекта
  - Возвращается ссылка, т.к. измененный результат может в дальнейшем быть модифицирован, как в случае с оператором ++ для встроенных типов данных:
    - ++counter += n;
- Синтаксис префиксной формы операторов:
  - Type& operator++()
  - Type& operator--()

Перегрузка постфиксной формы инкремента и декремента

- **Постфиксная** операция выполняет модификацию объекта и возвращает **временную константную копию** объекта до модификации
  - Копия должна быть константной, чтобы **не допустить** операций вроде:
    - counter++ -= 3;
- Синтаксис постфиксной формы операторов:
  - Type const operator++(int)
  - Type const operator--(int)
  - Целочисленный параметр фактически не используется и служит лишь для различия от префиксной формы
- С точки зрения здравого смысла постфиксную форму операторов инкремента и декремента следует основывать на их префиксной форме

Пример - счетчик

```
class CCounter
{
public:
     explicit CCounter(unsigned maxValue, counter = 0)
          :m_maxValue(maxValue), m_counter(counter){}
     unsigned GetValue()const{return m_counter;}
     unsigned GetMaxValue()const{return m_maxValue;}

     CCounter& operator++()
     {
          ++m_counter;
          if (m_counter >= m_maxValue)
          {
               m_counter = 0;
          }
          return *this;
     }
     CCounter const operator++(int)  // постфиксная форма инкремента
     {
          // создаем копию, выполняем предынкремент и возвращаем копию
          CCounter tmpCopy(*this);
          ++*this;
          return tmpCopy;
     }
private:
     unsigned m_maxValue, m_counter;
};
```

Конструктор может быть помечен как явный при помощи ключевого слова explicit , чтобы запретить возможность его **неявного** вызова в ситуациях, вроде следующих:

CCounter counter(20, 5);
counter = 10;
эквивалентно:
CCounter counter(20, 5);
counter = CCounter(10, 0);

Перегрузка операторов потокового ввода-вывода

Потоки ввода-вывода и операторы ввода-вывода в поток

- В STL операции ввода-вывода выполняются при помощи **потоков** данных
  - Поток – специальный объект, свойства которого определяются классом
  - Вывод – запись данных в поток
  - Ввод – чтение данных из потока
- cin и cout – глобальные объекты потоков ввода и вывода
- Для потоков ввода и вывода определены операторы << и >> для каскадной форматированных операций записи данных в поток и чтения данных из потока
  - cout << 3;
  - cin >> var;

Перегрузка операторов потокового ввода-вывода

- Перегрузка операторов форматированного ввода-вывода в потоки STL не может быть выполнена внутри самих классов потоков
  - Внесение модификаций в STL запрещено Стандартом
    - По этой же причине операторы ввода-вывода пользовательских типов не могут быть объявлены друзьями классов потоков, хотя могут быть друзьями самих пользовательских классов
  - Объекты потоков никоим образом не связаны с пользовательскими типами данных
- Для перегрузки операторов ввода-вывода следует объявлять их вне класса
- Операторы форматированного ввода-вывода должны возвращать ссылку на поток

Перегрузка оператора вывода в поток для класса «Счетчик»

```
// выводим информацию о счетчике в виде [counter/maxValue]
// в произвольный поток вывода
template <class T>
std::basic_ostream<T>& operator<<(
    std::basic_ostream<T>& stream, CCounter const& counter)
{
    stream  << "[" << counter.GetValue() << "/«
            << counter.GetMaxValue() << "]";
    return stream;
}
```

Перегрузка оператора чтения из потока для класса «Счетчик»

```
template <class T>
std::basic_istream<T>& operator>>(
    std::basic_istream<T>& stream, CCounter & counter)
{
    std::streamoff pos = stream.tellg();

    unsigned maxValue = 0;
    unsigned currentValue = 0;
    if (
        (stream.get() == '[') && (stream >> currentValue) &&
        (stream.get() == '/') && (stream >> maxValue) &&
        (stream.get() == ']')
        )
    {
        counter = CCounter(maxValue, currentValue);
        return stream;
    }

    stream.seekg(pos);
    stream.setstate(std::ios_base::failbit | stream.rdstate());

    return stream;
}
```

Пример использования перегруженных операций ввода-вывода

```
#include <iostream>
#include "Counter.h"

int main(int argc, char* argv[])
{
    CCounter c(10);

    // считывает данные о счетчике из стандартного ввода в формате:
    // [counter/maxValue]
    cin >> c;

    // выводит данные о счетчике в стандартный вывод в формате:
    // [counter/maxValue]
    cout << c;

    return 0;
}
```

Перегрузка операторов приведения типов

Перегрузка оператора приведения типа

- Иногда возникает необходимость выполнить приведение одного пользовательского типа к другому пользовательскому или встроенному типу данных. Например:
  - Приведение CMyString к const char*
  - Приведение CCounter к unsigned int
  - Приведение CDateTime к CTime
- Язык C++ позволяет в таких случаях обойтись без введения дополнительных методов, вроде GetStringData(), GetTimer(), GetTime() при помощи операторов приведения типа
- Синтаксис оператора приведения к типу Type:
  - operator Type()[const]

Пример: приведение счетчика к unsigned int

```
class CCounter
{
public:
…
     operator unsigned int()const
     {
          return m_counter;
     }
…
};

void f(unsigned value);

int main(int argc, char* argv[])
{
     CCounter c(10);

     f(c);    // будет вызван оператор приведения к типу unsigned int

     unsigned v = c;  // аналогично

     return 0;
}
```

Пример: приведение строкового объекта к const char*

```
class CMyString
{
public:
…
     operator const char*()const
     {
          return m_pChars;
     }
…
};

void f(const char* s);

int main(int argc, char* argv[])
{
     CMyString message("Hello, world");

     f(message);  // будет вызван оператор приведения к const char*

     return 0;
}
```

Не переусердствуйте!

- Перегружать операторы приведения типов следует осторожно, т.к. из-за неявного приведения типов иногда возможны нежелательные последствия
  - Не случайно в классе std::string вместо оператора приведения к const char*, реализовали специальный метод c_str()

Пример нежелательного приведения типов

```
#include <iostream>

class CMyString
{
public:
…
     CMyString const operator+ (const char*)const;
     operator const char*()const
     {
          return m_pChars;
     }
…
};
int main(int argc, char* argv[])
{
     CMyString msg(“5432”);

     // допустим, что мы забыли заключить 1 в кавычки для склейки строк
     std::cout << (msg + 1);
     // фактически вызвав std::cout << (static_cast<const char*>(msg) + 1);
     // поэтому будет выведено «432» вместо «54321»
     return 0;
}
```

Перегрузка оператора (). Функторы

Функторы

- **Функтор** (или объект функции, function object) – объект, для которого определен оператор ()
- Преимущества функторов перед обычными функциями
  - Наличие состояния у функтора
  - Объект функции обладает некоторым типом и может выступать в качестве специализации шаблонов
  - Обычно функтор работает быстрее указателя на функцию

Пример функтора

```
#include <iostream>

class CAddValue
{
public:
	CAddValue(int value):m_value(value)
	{
	}
	void operator()(int & arg)const
	{
		arg += m_value;
	}
private:
	int m_value;
};

int main(int argc, char* argv[])
{
	int value = 10;

	CAddValue f(5);
	std::cout << "Value before applying the functor: " << value << std::endl;
	f(value);
	std::cout << "Value after applying the functor: " << value << std::endl;

	return 0;
}
```

```
Value before applying the functor: 10
Value after applying the functor: 15
```

Пример: использование функтора совместно с алгоритмами STL

```
#include <iostream>
#include <vector>
#include <algorithm>

Int main(int argc, char* argv[])
{
    std::vector<int> arr;
    arr.push_back(10);
    arr.push_back(20);
    arr.push_back(30);

    std::cout << "Original array: " << std::endl;
    std::copy(arr.begin(), arr.end(), std::ostream_iterator<int>(std::cout, ","));

    std::for_each(arr.begin(), arr.end(), CAddValue(5));

    std::cout << std::endl << "Processed array: " << std::endl;
    std::copy(arr.begin(), arr.end(), std::ostream_iterator<int>(std::cout, ","));

    std::cout << std::endl;

    return 0;
}
```

```
Original array:
10,20,30,
Processed array:
15,25,35,
```

Вывод массива в поток вывода

Применение функтора к элементам массива

Использование состояния функтора

- Функтор, в отличие от функции, обладает состоянием
  - Глобальные и статические переменные функций в расчет не берем
  - Состояние функтора, как и обычного объекта, определяется значением полей-данных
- Вызов функтора в разных состояниях может приводить к разным результатам

Пример изменения состояния функтора при каждом его вызове

```
#include <iostream>

class CAddValue
{
public:
    CAddValue(int value, int delta = 0)
        :m_value(value)
        ,m_delta(delta)
    {
    }
    // отметим, что оператор объявлен как неконстантный
    void operator()(int & arg)
    {
        arg += m_value;
        m_value += m_delta;
    }
private:
    int m_value;
    int m_delta;
};
```

Пример использования функтора с изменяющимся состоянием

```
#include <iostream>
#include <vector>
#include <algorithm>

int main(int argc, char* argv[])
{
    std::vector<int> arr;
    arr.push_back(10);
    arr.push_back(20);
    arr.push_back(30);

    std::cout << "Original array: " << std::endl;
    std::copy(arr.begin(), arr.end(), std::ostream_iterator<int>(std::cout, ","));

    std::for_each(arr.begin(), arr.end(), CAddValue(5, 2));

    std::cout << std::endl << "Processed array: " << std::endl;
    std::copy(arr.begin(), arr.end(), std::ostream_iterator<int>(std::cout, ","));

    std::cout << std::endl;

    return 0;
}
```

```
Original array:
10,20,30,
Processed array:
15,27,39,
```

Пример: генератор псевдослучайных чисел

```
class CRandomGenerator
{
public:
	CRandomGenerator(unsigned modulus, unsigned seed = 0,
			unsigned multiplier = 733,
			unsigned summand = 1559)
		:m_modulus(modulus)
		,m_seed(seed)
		,m_multiplier(multiplier)
		,m_summand(summand)
	{
	}
	void Reset(unsigned newSeed)
	{
		m_seed = newSeed;
	}
	unsigned operator()()
	{
		m_seed = m_seed * m_multiplier + m_summand;
		return m_seed % m_modulus;
	}
private:
	unsigned m_modulus;
	unsigned m_seed;
	unsigned m_multiplier;
	unsigned m_summand;
};
```

```
#include <iostream>

int main(int argc, char* argv[])
{
	CRandomGenerator rnd(10);
	for (int i = 0; i < 10; ++i)
	{
		std::cout << rnd() << ", ";
	}
	return 0;
}
```

Output:

9, 6, 7, 2, 1, 6, 7, 8, 3, 8,