

top

КОМПЬЮТЕРНАЯ
АКАДЕМИЯ

ОБЪЕКТНО -ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ НА C++

Урок 4

Конструктор переноса.
Специальные перегрузки.
Дружественные функции
и перегрузка

Содержание

1. Конструктор перемещения	4
2. Применение перемещения при перегрузке оператора присваивания	30
3. Заданные по умолчанию и удаленные методы.....	39
4. Специальные перегрузки — перегрузка []	55
5. Специальные перегрузки — перегрузка ()	76
6. Специальные перегрузки — перегрузка оператора преобразования типов.....	113
7. Список операторов, которые невозможно перегрузить	123
8. Статический полиморфизм и перегрузка операторов, как частный случай.....	124
9. Перегрузка операторов глобальными функциями	125

10. Перегрузка операторов дружественными функциями	133
11. Перегрузка ввода-вывода	146
12. Дружественные классы	159
13. Домашнее задание	164

Материалы урока прикреплены к данному PDF-файлу. Для доступа к ним, урок необходимо открыть в программе Adobe Acrobat Reader.

1. Конструктор перемещения

Для изучения нового материала нам понадобится знакомый по теме «Конструктор копирования» пример с динамическим массивом `DynArray`. Вооружившись знаниями прошлого урока о перегрузке различных операторов, в том числе оператора присваивания, добавим в наш класс перегрузку оператора копирующего присваивания на основе имеющегося экземпляра `DynArray`.

Пример 1

```
#include <iostream>
class DynArray
{
    int* arr;
    int size;
public:
    DynArray(int sizeP)
        : arr{ new int[sizeP] {} }, size{ sizeP }
    {
        std::cout << "DynArr constructed for " << size
                    << " elements, for " << this << '\n';
    }
    DynArray() : DynArray(5) {}
    DynArray(const DynArray& object)
        : arr{ new int[object.size] }, size{ object.size }
    {
        /* В списке инициализаторов полей класса выше
           выделяем новый блок динамической памяти того же
           размера, что и в копируемом экземпляре класса
           DynArray. Следующим циклом копируем элементы
           из оригинального блока памяти во вновь
           выделенный. */
    }
}
```

```

    for (int i{ 0 }; i < size; ++i)
    {
        arr[i] = object.arr[i];
    };
    std::cout << "DynArr copy constructed for "
                << size << " elements, for " << this
                << '\n';
}
DynArray& operator=(const DynArray& object)
{
    // проверка на самоприсваивание
    if (!(this == &object))
    {
        /* проверяем на невозможность "переиспользовать"
           блок памяти, выделенный под имеющийся
           массив */
        if (size != object.size)
        {
            /* в случае невозможности "переиспользования"
               необходимо освободить память, УЖЕ
               занимаемую элементами текущего
               динамического массива */
            delete[] arr;
            /* выделяем новый блок памяти согласно
               размеру копируемого массива */
            arr = new int[object.size];
        }
        size = object.size;
        /* Следующим циклом копируем элементы
           из оригинального блока памяти во вновь
           выделенный */
        for (int i{ 0 }; i < size; ++i)
        {
            arr[i] = object.arr[i];
        };
    }
}

```

```

        std::cout << "DynArr copy assigned for "
                    << size << " elements, for " << this
                    << '\n';
        return *this;
    }

    int getElem(int idx)const { return arr[idx]; }
    void setElem(int idx, int val) { arr[idx] = val; }
    void print()const;
    void randomize();
    ~DynArray()
    {
        std::cout << "Try to free memory from DynArray for"
                    << arr << " pointer\n";
        delete[] arr;
        std::cout << "DynArr destructed for " << size
                    << " elements, for " << this << '\n';
    }
};

void DynArray::print()const
{
    for (int i{ 0 }; i < size; ++i)
    {
        std::cout << arr[i] << ' ';
    }
    std::cout << '\n';
}

void DynArray::randomize()
{
    for (int i{ 0 }; i < size; ++i)
    {
        arr[i] = rand() % 10;
    }
}

int main()
{

```

```

DynArray ar1{ 10 };
ar1.randomize();
std::cout << "ar1 elements: ";
ar1.print();
DynArray ar2{ ar1 };
std::cout << "ar2 elements: ";
ar2.print();
std::cout << "Copy-assignment test\n";
DynArray ar3{ 5 };
std::cout << "ar3 elements before copy: ";
ar3.print();
ar3 = ar2;
std::cout << "ar3 elements after copy: ";
ar3.print();

return 0;
}

```

Вывод результата работы кода из примера 1:

```

Microsoft Visual Studio Debug Console
DynArr constructed for 10 elements, for 00C5F9A0
ar1 elements: 1 7 4 0 9 4 8 8 2 4
DynArr copy constructed for 10 elements, for 00C5F990
ar2 elements: 1 7 4 0 9 4 8 8 2 4
Copy-assignment
DynArr constructed for 5 elements, for 00C5F980
ar3 elements before copy: 0 0 0 0 0
DynArr copy assigned for 10 elements, for 00C5F980
ar3 elements after copy: 1 7 4 0 9 4 8 8 2 4
Try to free memory from DynArray for012B5230 pointer
DynArr destructed for 10 elements, for 00C5F980
Try to free memory from DynArray for012B4700 pointer
DynArr destructed for 10 elements, for 00C5F990
Try to free memory from DynArray for012B4898 pointer
DynArr destructed for 10 elements, for 00C5F9A0

```

Рисунок 1

Комментарии в примере 1 и вывод результатов свидетельствуют о том, что у нас получился класс, полностью поддерживающий семантику копирования как при инициализации (конструктор копирования) так и при присваивании (оператор присваивания копированием). Тестами подтверждено, что происходит именно глубокое копирование с выделением нового блока памяти под копии.

Однако возникает вопрос, всегда ли необходимо глубокое копирование и насколько оно оптимально? На самом деле такое глубокое копирование нужно не всегда. Нередко возможно обойтись без глубокого копирования, а заменить его на более эффективный вариант — перемещение. Что это за «перемещение» и в каких случаях оно необходимо, мы и рассмотрим далее.

В примерах по работе с динамическим массивом мы часто создавали массив и тут же заполняли его случайными числами, давайте же избавимся от повторения кода и напишем для этого функцию! Параметром функции выступит количество элементов в массиве, возвращать же функция будет динамический массив заданного размера, заполненный случайными числами. Изменим пример 1 добавив к нему вышеописанную функцию и заменим функцию `main` для тестирования работы.

Пример 2

```
DynArray arrayFactory(int arrSize)
{
    DynArray arr{ arrSize };
    arr.randomize();
}
```



```

    return arr;
}

int main()
{
    DynArray ar1{ arrayFactory(10) };
    std::cout << "ar1 elements: ";
    ar1.print();
    return 0;
}

```

Вывод результата работы кода из примера 2:

```

Microsoft Visual Studio Debug Console
DynArr constructed for 10 elements, for 007CFC00
DynArr copy constructed for 10 elements, for 007CFD0C
Try to free memory from DynArray for00C2BE30 pointer
DynArr destructed for 10 elements, for 007CFC00
ar1 elements: 1 7 4 0 9 4 8 8 2 4
Try to free memory from DynArray for00C24DB0 pointer
DynArr destructed for 10 elements, for 007CFD0C

```

Рисунок 2

Внимательнее изучим нашу функцию — «фабрику массивов». Для выполнения поставленных перед ней задач необходимо создать экземпляр класса `DynArray`, указанного значением параметра `arrSize` размера. Экземпляр создается как локальная переменная — вызовом конструктора `DynArray` принимающего параметром количество элементов массива. Далее на экземпляре класса производится вызов функции-члена `randomize()` и, собственно, заполнение массива случайными числами. Казалось бы, задача решена, но именно тут начинается самая интересная часть.

Как нам вернуть результат работы нашей функции? Можем ли мы вернуть ссылку на экземпляр класса, созданный внутри функции? Нет, ведь его время жизни завершится с завершением работы функции, и мы получим «висячую» ссылку на уже удаленный экземпляр `DynArray`. Возможно, стоит выделять экземпляр класса в динамической памяти и тогда мы сможем вернуть указатель на экземпляр класса? Это сработает, но пользоваться такой функцией будет крайне неудобно. Придется работать не с экземпляром класса, а с указателем на экземпляр, что не так удобно. Однако более важной проблемой станет необходимость ручного освобождения памяти, выделенной под экземпляр класса, ведь простого завершения времени жизни собственно указателя недостаточно для освобождения памяти, занимаемой экземпляром класса. Что же остается? А остается лишь возврат экземпляра класса по значению.

Возврат по значению в общем случае происходит копированием значения, возвращаемого функцией во временный объект, являющийся хранилищем возвращаемого значения функции после ее завершения. Копирование значения для экземпляров классов приводит к запуску конструктора копирования, что может быть не самым оптимальным решением, как будет показано ниже.

Временный объект-хранилище возвращаемого значения доступен в месте вызова функции и его время жизни — выражение, в котором была вызвана функция. Именно значение этого объекта используется как правая часть оператора присваивания в случае сохранения возвращаемого значения функции в некую переменную.

На этапе копирования возвращаемого значения в этот временный объект компилятор может применить оптимизацию и при определенных условиях избавиться от дополнительного копирования. Описание данного процесса оптимизации (RVO, Return Value Optimization) выходит за рамки изучаемой нами в настоящий момент темы, детали и тонкости реализации не требуются для понимания собственно изучаемой темы. Необходимо лишь знание того, что данная оптимизация может иметь место.

Вернемся, к примеру 2. Внутри функции `main` мы создаем экземпляр класса `DynArray` инициализируя его результатом работы функции `arrayFactory`. О создании внутри функции экземпляра класса `DynArray` свидетельствует первая строка в выводе 2. После завершения работы функции и получения результата во временном объекте (в данном случае как раз имеет место выше-описанная RVO-оптимизация) временный объект — экземпляр класса `DynArray` — выступает инициализатором создаваемого нами массива `ar1`. Происходит копирующая инициализация — вторая строка вывода 2. Время жизни временного объекта завершено и для него вызывается деструктор — четвертая строка вывода 2. Затем мы выводим на экран содержимое созданного таким образом массива — пятая строка вывода 2. Функция `main`, а с ней и тестовое приложение, завершает свою работу, завершается время жизни `ar1` и для него вызывается деструктор — последняя строка вывода 2.

Работа примера 2 происходит без ошибок, но оптимально ли возникающее в примере копирование? Внутри функции при создании экземпляра `DynArray`

выделяется блок динамической памяти, в котором и хранятся элементы массива. Далее в процессе копирования возвращаемого значения в `ar1` создается еще один блок динамической памяти, в который копируются значения из блока памяти, принадлежащего временному объекту. И все эти «тяжелые» и ресурсоёмкие операции выделения памяти и копирования порождают копию блока памяти, который тут же будет освобожден! Крайне неоптимальное, хотя и необходимое для согласованной работы класса `DynArray`, поведение.

Намного оптимальнее было бы передать владение блоком динамической памяти с хранимыми там элементами от временного объекта к объекту `ar1` — произвести не глубокое, а поверхностное копирование. Но мы уже прекрасно знаем, чем это грозит — преждевременным освобождением блока динамической памяти деструктором временного объекта и сбоем работы деструктора `ar1` при попытке освободить уже освобожденный блок динамической памяти. Причиной тому — наличие у временного объекта указателя на не требующий удаления по высказанным выше соображения блок динамической памяти.

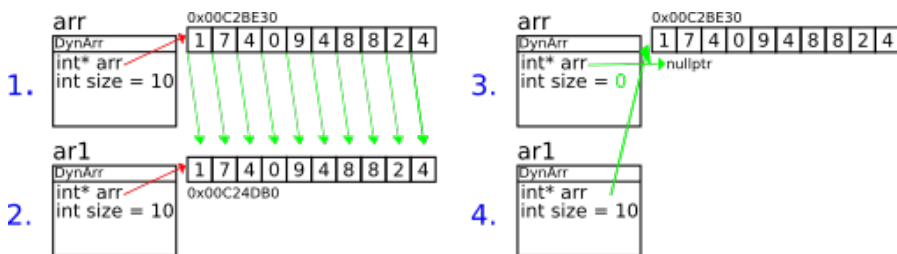


Рисунок 3

Было бы крайне желательно лишить временный объект этого указателя. Проиллюстрируем желаемое поведение рисунком 3.

Фрагменты 1, 2 отражают обычное поведение, использующее копирование. Фрагмент 1 показывает состояние объекта `arr` внутри функции, фрагмент 2 показывает состояние объекта `ar1` после выделения нового блока памяти и копирования в него (зеленые стрелки) всех элементов из блока памяти, принадлежащего `arr`. Фрагменты 3 и 4 отражают состояние объектов `arr` и `ar1` в процессе возврата значения из функции `arrayFactory`. Тут мы видим, что объект `ar1` «отобрал право владения» блоком памяти у `arr` без выделения собственного блока памяти и выполнения копирования элементов.

Но как достичь такого поведения? Ведь все это должно происходить в процессе работы пускай и модифицированного, но все же конструктора копирования, а его единственным параметром является `const DynArray&`, то есть константная ссылка. О необходимости константности данной ссылки было рассказано в уроке, рассматривавшем тему конструктора копирования. Выходит, желаемая оптимизация невозможна и является лишь «фантазией»? Опираясь лишь на имеющиеся в нашем распоряжении инструменты и конструкции языка — да. Но мы не остановимся на пути к нашей цели и изучим новые инструменты и конструкции языка, а потом вновь вернемся к решению нашей задачи с оптимизацией копирования.

Для начала давайте вспомним что такое ссылка. Ссылка — это псевдоним какого-то именованного объекта в памяти.

Пример 3

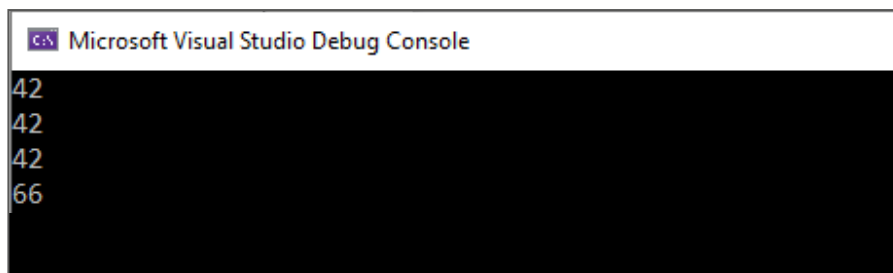
```
#include <iostream>

int main()
{
    int x{ 42 };
    int& refX{ x };
    const int& cRefX{ x };
    const int& cRefXX{ x + 24 };

    std::cout << x << '\n'
              << refX << '\n'
              << cRefX << '\n'
              << cRefXX << '\n';

    return 0;
}
```

Вывод результата работы кода из примера 3:



```
Microsoft Visual Studio Debug Console

42
42
42
66
```

Рисунок 4

Итак, `x` — это некая переменная, имеющая свое расположение в оперативной памяти связанное с собственно именем `x`, `refX` — это другое имя, связанное с той же областью памяти. Все, что мы можем сделать с `x`,

мы можем сделать и с `refX`. `cRefX` — иное имя области памяти, хранящей `x`, но позволяющее лишь чтение, но не модификацию данной области памяти. И, наконец, самое интересное — `cRefXX`. Это константная ссылка, но на что? На выражение? На место в памяти, хранящее результат выражения, его значение. Без модификатора `const` невозможно создать данную ссылку, так как тогда была бы возможность изменять результат выражения, что кажется весьма странным. Во всяком случае пока. Что же делает «странная» ссылка `cRefXX`? Обычно, время жизни временного объекта, являющегося результатом выражения — собственно само выражение. После вычисления всего выражения `x + 24` временный объект уничтожается. Константная ссылка `cRefXX` продлевает время жизни этого временного объекта до времени жизни самой ссылки. `refX` и `cRefX` — обычные, знакомые нам по курсу, `lvalue`-ссылки. `Lvalue`-ссылка — это ссылка на объект в памяти имеющий свое имя. Последняя ссылка, `cRefXX`, немного не соответствует такому определению, так как временный объект не имеет своего имени. В старых стандартах C++ это было исключением из общего правила.

Начиная со стандарта C++11, появилась возможность создавать неконстантные ссылки на временные, «безымянные» объекты, по сути — ссылки на значения. Такие ссылки называются `rvalue`-ссылки. Декларация `rvalue`-ссылки производится добавлением двух амперсандов после типа ссылки:

```
RefType&& refName{ rvalue object };
```

Рассмотрим ряд характерных примеров.

Пример 4

```

#include <iostream>

int max(int a, int b)
{
    return a > b ? a : b;
}

int main()
{
    int&& rvalRef { 2 + 3};
    rvalRef += 3;
    std::cout << rvalRef << '\n';

    int&& res{ max(3, 5) };
    res += max(6, 4);
    std::cout << res << '\n';
    int x{ 42 };

    //    Следующие строки закомментированы специально!
    //int&& rvalBad{ x }; /* Невозможно инициализировать
                           rvalue-ссылку lvalue-объектом.*/
    //int&& rvalBad1{ res }; /* rvalue-ссылка сама является
                              lvalue-объектом */

    int& lvalRef{ res };
    std::cout << lvalRef << '\n';
    return 0;
}

```

Вывод результата работы кода из примера 4:

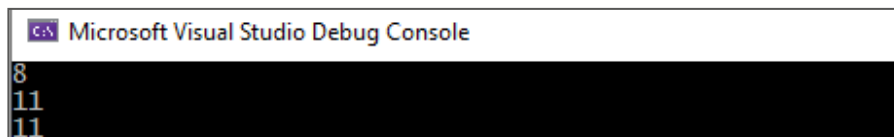


Рисунок 5

Прокомментируем увиденное в примере 4. Итак, `rvalue`-ссылку можно инициализировать только `rvalue`-объектом, именно это происходит с `rvalRef`, она инициализируется результатом выражения `2+3`, продлевая время жизни временного объекта, хранящего результат выражения до времени жизни самой ссылки и давая возможность изменять этот объект.

Другим возможным способом инициализации `rvalue`-ссылки является ее инициализация результатом функции, возвращающей значение — ссылка `res` в нашем примере. Результат функции `max(3,5)` — не что иное, как временный `rvalue`-объект, прекрасно подходящий как инициализатор. Аналогично `rvalRef`, время жизни продлевается и есть возможность изменять объект-результат работы функции.

Следует уделить особое внимание тому, что `rvalue`-ссылку нельзя инициализировать, иначе говоря «привязать», к `lvalue`-объекту! Ссылку `rvalBad` из примера 4 невозможно привязать к переменной `x`, так как `x` — не что иное, как именованный объект в памяти, а значит, `lvalue`! Ссылку `rvalBad1` невозможно создать по причине того, что `res` — это `lvalue`-объект! Как же так? Ведь `res` — `rvalue`-ссылка? Все дело в том, что `res` связан с анонимным, безымянным объектом-результатом функции, но сам объект `res` уже не анонимен, он-то как раз и именует, деанонимизирует объект-результат функции! Посему `rvalue`-ссылка сама по себе является `lvalue`-объектом, что и демонстрирует ссылка `lvalRef` из примера 4.

Подведем некоторые итоги: начиная со стандарта C++11, существуют два вида ссылок: `lvalue`-ссылки,

то есть ссылки на именованные объекты, и *rvalue*-ссылки, ссылающиеся на безымянные значения, временные объекты. Обе разновидности ссылок могут быть константными и неконстантными, требуют «привязки» к соответствующему объекту при инициализации, не поддерживают «перепривязку» на другой объект после инициализации. Невозможно создать неконстантную *lvalue*-ссылку на *rvalue*-значение. Невозможно создать *rvalue*-ссылку непосредственно на *lvalue*-значение. Однако при помощи вызова `std::move` возможно преобразовать *lvalue*-значение к *rvalue*-значению. Использование преобразованного к *rvalue* *lvalue*-значения может иметь определенные последствия, но об этом немногим позднее. Также позднее будет показана практическая польза и прикладное назначение такого преобразования, пока важно лишь знать о наличии такой возможности.

Пример 5

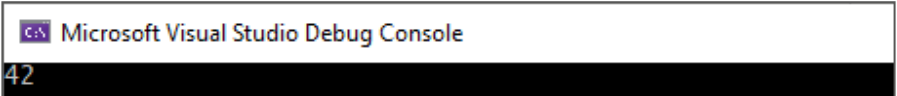
```
#include <iostream>

int main()
{
    int x{ 42 };
    int&& rvalRef{ std::move(x) };

    std::cout << rvalRef << '\n';

    return 0;
}
```

Вывод результата работы кода из примера 5:

The image shows a screenshot of the Microsoft Visual Studio Debug Console. The title bar at the top reads "Microsoft Visual Studio Debug Console". Below the title bar, the number "42" is displayed in a white font on a black background.

42

Рисунок 6

В завершении описания *rvalue*-ссылок необходимо отметить возможность создания отдельных перегрузок функции для разных категорий значений *lvalue* и *rvalue*.

Пример 6

```
#include <iostream>

// funA overloads with rvalue overload
void funA(int& val)
{
    std::cout << "funA() called for int&\n";
}

void funA(const int& val)
{
    std::cout << "funA() called for const int&\n";
}

void funA(int&& val)
{
    std::cout << "funA() called for int&&\n";
}

// funB overloads without rvalue overload
void funB(int& val)
{
    std::cout << "funB() called for int&\n";
}

void funB(const int& val)
{

```

```

    std::cout << "funB() called for const int&\n";
}

int main()
{
    int val{ 42 };
    const int cVal{ 26 };

    std::cout << "funA overloads with rvalue overload:\n";
    std::cout << "lvalue\n";
    funA(val); // lvalue -> int&
    std::cout << "const lvalue\n";
    funA(cVal); // const lvalue -> const int&
    std::cout << "rvalue\n";
    funA(80 + 1); // rvalue -> int&&
    std::cout << "moved lvalue\n";
    funA(std::move(val)); // moved lvalue -> int&&

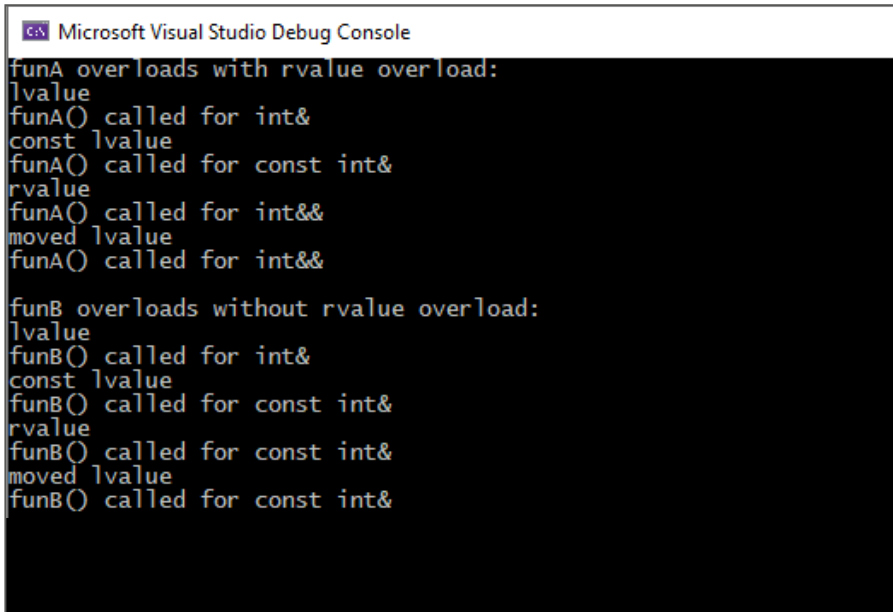
    std::cout << '\n';

    std::cout << "funB overloads without rvalue "
                "overload:\n";
    std::cout << "lvalue\n";
    funB(val); // lvalue -> int&
    std::cout << "const lvalue\n";
    funB(cVal); // const lvalue -> const int&
    std::cout << "rvalue\n";
    funB(80 + 1); // rvalue -> const int&
    std::cout << "moved lvalue\n";
    funB(std::move(val)); // moved lvalue -> const int&

    return 0;
}

```

Вывод результата работы кода из примера 6:



```

Microsoft Visual Studio Debug Console

funA overloads with rvalue overload:
lvalue
funA() called for int&
const lvalue
funA() called for const int&
rvalue
funA() called for int&&
moved lvalue
funA() called for int&&

funB overloads without rvalue overload:
lvalue
funB() called for int&
const lvalue
funB() called for const int&
rvalue
funB() called for const int&
moved lvalue
funB() called for const int&

```

Рисунок 7

В случае наличия отдельной перегрузки функции для **rvalue**, вызов функции с **rvalue**-аргументом вызовет именно ее, в случае же отсутствия такой перегрузки, ее заменит перегрузка для **const lvalue**. Возможность «отката» до перегрузки **const lvalue** в случае отсутствия **rvalue**-перегрузки также окажется весьма полезной и практичной в дальнейшем.

Изучив новое понятие **rvalue**-ссылок, мы можем вернуться к исходной задаче оптимизации копирования. Напомним, что оптимизация должна заключаться в упразднении лишнего копирования блока памяти из временного объекта. Решением будет создание специальной перегрузки конструктора «копирования», которая будет принимать параметром **rvalue**-ссылку на экземпляр класса

DynArray. Такой конструктор называется конструктором перемещения или move-конструктором.

Конструктор перемещения — это специальный конструктор, служащий для перемещения ресурсов, которыми владеет существующий временный экземпляр класса в новый экземпляр того же класса без выполнения их глубокого копирования. Примером ресурса может быть блок динамической памяти. Конструктор перемещения можно определить явно, для этого необходимо использовать специальную сигнатуру:

```
ClassName(ClassName&& object);
```

где **ClassName** — имя класса, для которого определяется конструктор. По сути, конструктор перемещения — это конструктор, принимающий экземпляр того же класса по **rvalue**-ссылке. Пример сигнатуры для **DynArray**:

```
DynArray(const DynArray&& object);
```

Давайте определим явно конструктор перемещения для класса **DynArray** и протестируем его работу.

Пример 7

```
#include <iostream>
class DynArray
{
    int* arr;
    int size;
public:
    DynArray(int sizeP)
        : arr{ new int[sizeP] {} }, size{ sizeP }
    {
```

```

        std::cout << "DynArr constructed for " << size
                    << " elements, for " << this << '\n';
    }
    DynArray() : DynArray(5) {}
    DynArray(const DynArray& object)
        : arr{ new int[object.size] }, size{ object.size }
    {
        /* В списке инициализаторов полей класса выше
        выделяем новый блок динамической памяти того же
        размера, что и в копируемом экземпляре класса
        DynArray. Следующим циклом копируем элементы
        из оригинального блока памяти во вновь
        выделенный. */
        for (int i{ 0 }; i < size; ++i)
        {
            arr[i] = object.arr[i];
        }
        std::cout << "DynArr copy constructed for "
                    << size << " elements, for " << this
                    << '\n';
    }

    DynArray(DynArray&& object)
        : arr{ object.arr }, size{ object.size }
        /* копируем указатель на выделенный в исходном
        объекте блок динамической памяти и размер этого
        блока в инициализируемый конструктором объект */
    {
        /* "отбираем" у исходного объекта право владения
        блоком динамической памяти и устанавливаем
        размер блока в 0 */
        object.arr = nullptr;
        object.size = 0;
        std::cout << "DynArr move constructed for "
                    << size << " elements, for " << this
                    << '\n';
    }
}

```

```

DynArray& operator=(const DynArray& object)
{
    // проверка на самоприсваивание
    if (!(this == &object))
    {
        /* проверяем на невозможность "переиспользовать"
           блок памяти, выделенный под имеющийся массив */
        if (size != object.size)
        {
            /* в случае невозможности "переиспользования"
               необходимо освободить память,
               УЖЕ занимаемую элементами текущего
               динамического массива */
            delete[] arr;
            /* выделяем новый блок памяти согласно
               размера копируемого массива */
            arr = new int[object.size];
        }
        size = object.size;

        /* альтернативный способ копирования массива
           более эффективный с точки зрения времени
           выполнения ценой трех дополнительных
           указателей */
        // указатель на начало массива-назначения
        // копирования
        int* dest{ arr };
        // указатель на начало массива-источника
        // копирования
        int* src{ object.arr };
        /* константный указатель на следующий
           за последним элементом в массиве-назначении –
           "конец массива источника" */
        int* const end{ arr + size };

        // пока указатель "назначения" не превышает
        // "конец"...
    }
}

```



```

        while (dest < end)
        {
            /* кладем по адресу указателя dest значение,
               хранимое по адресу src затем инкрементируем
               оба указателя */
            *dest++ = *src++;
        }
        // массив из obj.arr скопирован в arr.
    }

    std::cout << "DynArr copy assigned for "
               << size << " elements, for " << this
               << '\n';
    return *this;
}

int getElem(int idx) const { return arr[idx]; }
void setElem(int idx, int val) { arr[idx] = val; }
void print() const;
void randomize();
~DynArray()
{
    std::cout << "Try to free memory from DynArray for"
               << arr << " pointer\n";
    delete[] arr;
    std::cout << "DynArr destructed for " << size
               << " elements, for " << this << '\n';
}
};

void DynArray::print() const
{
    for (int i{ 0 }; i < size; ++i)
    {
        std::cout << arr[i] << ' ';
    }
    std::cout << '\n';
}

```

```

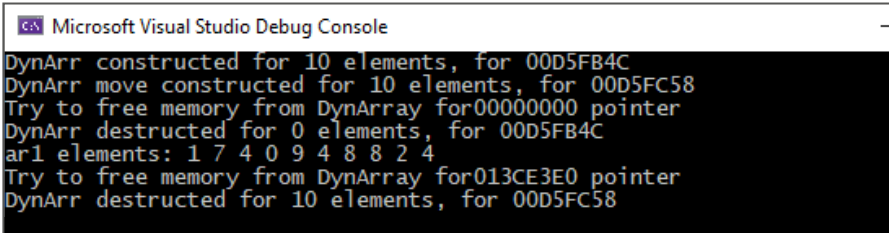
void DynArray::randomize()
{
    for (int i{ 0 }; i < size; ++i)
    {
        arr[i] = rand() % 10;
    }
}

DynArray arrayFactory(int arrSize)
{
    DynArray arr{ arrSize };
    arr.randomize();
    return arr;
}

int main()
{
    DynArray ar1{ arrayFactory(10) };
    std::cout << "ar1 elements: ";
    ar1.print();
    return 0;
}

```

Вывод результата работы кода из примера 7:



```

Microsoft Visual Studio Debug Console
DynArr constructed for 10 elements, for 00D5FB4C
DynArr move constructed for 10 elements, for 00D5FC58
Try to free memory from DynArray for00000000 pointer
DynArr destructed for 0 elements, for 00D5FB4C
ar1 elements: 1 7 4 0 9 4 8 8 2 4
Try to free memory from DynArray for013CE3E0 pointer
DynArr destructed for 10 elements, for 00D5FC58

```

Рисунок 8

Пример 7 идентичен примеру 2 с той лишь разницей, что в примере 7 присутствует явно определенный

конструктор перемещения и в целях расширения кругозора читателей, продемонстрирован альтернативный способ копирования массива в операторе присваивания. Данный вариант копирования компилируется в более эффективный, более быстрый машинный код. Более того, работа с массивом таким способом, через указатели, превосходит то, как мы будем работать с массивами и подобными им контейнерами в последующих частях нашего курса (тема «Итераторы»).

В примере 7, аналогично примеру 2, происходит создание экземпляра класса `DynArray` внутри функции (первая строка вывода) и возврат результата в виде временного объекта — экземпляра `DynArray`. И ввиду того, что временный объект — `rvalue`-значение, а в классе `DynArray` есть перегрузка конструктора именно для `rvalue`, запускается соответствующая перегрузка, а именно — конструктор перемещения. Работу конструктора перемещения необходимо рассмотреть детальнее.

Сперва через список инициализаторов во вновь создаваемый экземпляр класса копируется указатель на блок динамической памяти и количество элементов в исходном объекте, производится аналог поверхностного копирования.

Дальше в теле конструктора производится присваивание значения `nullptr` в указатель на блок динамической памяти исходного объекта и значения `0` в количество его элементов. Это стало возможным благодаря тому, что не в пример конструктору копирования, исходный объект конструктору перемещения передается через неконстантную `rvalue`-ссылку! Вот и обещанная ранее

иллюстрация пользы от модификации объекта посредством `rvalue`-ссылки. Работа конструктора перемещения завершается выводом отладочной информации о том, что сработал именно конструктор перемещения, — вторая строка вывода. После завершения конструктора перемещения инициализированный им экземпляр класса `DynArray` непосредственно использует тот блок динамической памяти, который был выделен при создании экземпляра класса внутри функции `arrayFactory`, без выделения нового блока и копирования в него данных из исходного. Оптимизация в действии!

Может показаться, что все эти сложности не стоят лишней копии 10 `int`-значений, возможно, но в общем случае динамический массив может хранить много больше элементов, и тогда-то накладные расходы на создание копии существенно возрастут и оптимизация уже не будет казаться столь несущественной.

Открытым остался один важный вопрос — что произойдет с временным объектом при вызове деструктора, не возникнут ли неожиданности? Нет, ведь временный объект был модифицирован конструктором перемещения. В деструкторе временного объекта произойдет попытка, бесполезная и безвредная, освободить память по указателю `nullptr`. Именно это отражено в третьей строке вывода. Таким образом временный объект будет без негативных последствий освобожден, не затронув при этом блок динамической памяти, который он изначально выделял и которым он «владел», — четвертая строка вывода.

Далее в нашем примере производится вывод элементов полученного в результате работы конструктора

перемещения, массива — строка 5 вывода. Функция `main`, а с ней и тестовое приложение завершает свою работу, завершается время жизни `ar1` и для него вызывается деструктор — последняя строка вывода.

Таким образом нам удалось реализовать наше, на первый взгляд нереальное, желание избегать лишних накладных расходов, связанных с глубоким копированием из объекта, чье время жизни и так подходит к концу. Осталось лишь прояснить вопрос того, как же работал пример 2, в рамках которого отсутствовала реализация конструктора, принимающего `rvalue`-ссылку? А здесь следует вспомнить возможность «откатиться» до `const lvalue&`-перегрузки в случае отсутствия `rvalue&`-перегрузки, а, как известно, `const lvalue&` — не что иное, как конструктор копирования! Вот и получается, что в случае отсутствия явно определенного конструктора перемещения его роль может на себя взять конструктор копирования, хотя это, как правило, менее эффективно.

2. Применение перемещения при перегрузке оператора присваивания

Аналогично «парности» конструктора копирования и оператора копирующего присваивания, существует «парность» конструктора перемещения и оператора, перемещающего присваивания. Такой оператор присваивания станет полезным, когда потребуется присвоить экземпляру класса значение временного объекта того же класса без излишнего глубокого копирования. Сигнатура такого оператора выглядит нижеследующим образом:

```
ClassName& operator=(ClassName&& object);
```

где `ClassName` — имя класса, для которого определяется оператор. Пример сигнатуры для `DynArray`:

```
DynArray& operator=(DynArray&& object);
```

Давайте определим явно оператор перемещающего присваивания для класса `DynArray` и протестируем его работу.

Пример 8

```
#include <iostream>
class DynArray
{
    int* arr;
    int size;
```

```

public:
    DynArray(int sizeP)
        : arr{ new int[sizeP] {} }, size{ sizeP }
    {
        std::cout << "DynArr constructed for " << size
                    << " elements, for " << this << '\n';
    }

    DynArray() : DynArray(5) {}
    DynArray(const DynArray& object)
        : arr{ new int[object.size] }, size{ object.size }
    {
        for (int i{ 0 }; i < size; ++i)
        {
            arr[i] = object.arr[i];
        };
        std::cout << "DynArr copy constructed for "
                    << size << " elements, for " << this
                    << '\n';
    }

    DynArray(DynArray&& object)
        : arr{ object.arr }, size{ object.size }
    {
        object.arr = nullptr;
        object.size = 0;
        std::cout << "DynArr move constructed for "
                    << size << " elements, for " << this
                    << '\n';
    }

    DynArray& operator=(const DynArray& object)
    {
        if (!(this == &object))
        {
            if (size != object.size)
            {
                delete arr;
            }
        }
    }

```

```

        arr = new int[object.size];
    }

    size = object.size;
    int* dest{ arr };
    int* src{ object.arr };
    int* const end{ arr + size };

    while (dest < end)
    {
        *dest++ = *src++;
    }
}

std::cout << "DynArr copy assigned for "
           << size << " elements, for " << this
           << '\n';

return *this;
}

DynArray& operator=(DynArray&& object)
{
    if (!(this == &object))
    {
        delete arr;

        arr = object.arr;
        size = object.size;
        object.arr = nullptr;
        object.size = 0;
    }

    std::cout << "DynArr move assigned for "
               << size << " elements, for " << this
               << '\n';

    return *this;
}

```



```

int getElem(int idx)const { return arr[idx]; }
void setElem(int idx, int val) { arr[idx] = val; }
void print()const;
void randomize();
~DynArray()
{
    std::cout << "Try to free memory from DynArray for"
                << arr << " pointer\n";
    delete[] arr;
    std::cout << "DynArr destructed for " << size
                << " elements, for " << this << '\n';
}
};

void DynArray::print()const
{
    for (int i{ 0 }; i < size; ++i)
    {
        std::cout << arr[i] << ' ';
    }
    std::cout << '\n';
}

void DynArray::randomize()
{
    for (int i{ 0 }; i < size; ++i)
    {
        arr[i] = rand() % 10;
    }
}

DynArray arrayFactory(int arrSize)
{
    DynArray arr{ arrSize };
    arr.randomize();
    return arr;
}

```

```

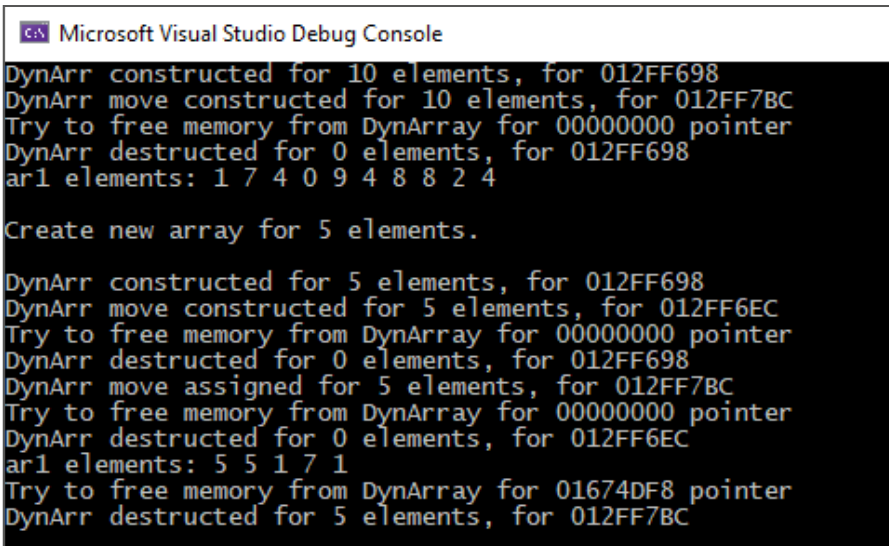
int main()
{
    DynArray ar1{ arrayFactory(10) };
    std::cout << "ar1 elements: ";
    ar1.print();

    std::cout << "\nCreate new array for 5 elements.\n\n";
    ar1 = arrayFactory(5);
    std::cout << "ar1 elements: ";
    ar1.print();

    return 0;
}

```

Вывод результата работы кода из примера 8:



```

Microsoft Visual Studio Debug Console

DynArr constructed for 10 elements, for 012FF698
DynArr move constructed for 10 elements, for 012FF7BC
Try to free memory from DynArray for 00000000 pointer
DynArr destructed for 0 elements, for 012FF698
ar1 elements: 1 7 4 0 9 4 8 8 2 4

Create new array for 5 elements.

DynArr constructed for 5 elements, for 012FF698
DynArr move constructed for 5 elements, for 012FF6EC
Try to free memory from DynArray for 00000000 pointer
DynArr destructed for 0 elements, for 012FF698
DynArr move assigned for 5 elements, for 012FF7BC
Try to free memory from DynArray for 00000000 pointer
DynArr destructed for 0 elements, for 012FF6EC
ar1 elements: 5 5 1 7 1
Try to free memory from DynArray for 01674DF8 pointer
DynArr destructed for 5 elements, for 012FF7BC

```

Рисунок 9

Рассмотрим работу примера 8. Как и в примере 7, сперва мы создаем экземпляр `ar1`, используя функцию

`arrayFactory(10)`, и выводим элементы получившегося массива на экран. Затем мы присваиваем имеющемуся экземпляру `ar1` результат работы функции `arrayFactory(5)`. Остановимся на работе оператора присваивания подробнее. Функция `arrayFactory(5)` создает экземпляр класса `DynArray` — строка 9 вывода, заполняет случайными числами и возвращает во временный объект — результат работы функции. Создание временного объекта происходит с использованием перемещения — строка 10 вывода. Завершается время жизни созданного функцией объекта и для него выполняется деструктор, не затрагивающий блок памяти, которым владеет объект-результат работы функции — строки 11-12 вывода. Затем происходит вызов оператора, перемещающего присваивания. Его реализация многим похожа на реализацию конструктора перемещения, но есть ряд важных отличий. Так как это не конструктор, создающий совершенно новый объект, а оператор присваивания, то необходимо проверить и исключить самоприсваивание, далее крайне важно освободить блок динамической памяти, который использовал экземпляр `ar1` до момента вызова оператора присваивания. И, наконец, выполнить действия, аналогичные конструктору перемещения, «отобрать» владение блоком динамической памяти у временного объекта — строка 13 вывода. Таким образом блок динамической памяти, изначально выделенный внутри функции `arrayFactory(5)`, сменив двух «владельцев», оказался в распоряжении экземпляра `ar1`. Далее завершится время жизни временного объекта-результата функции, и он безболезненно разрушится, никак не повлияв на не принадлежащий ему уже

блок памяти — строки 14-15 Вывода. И в завершении массив выводится на экран.

Пара — конструктор перемещения и оператора перемещающего присваивания, позволила дважды избежать нецелесообразного глубокого копирования, реализовав для класса `DynArray` семантику перемещения. Семантика перемещения позволяет, наряду с семантикой копирования, не только копировать один экземпляр класса в другой, но и перемещать один объект класса в другой, даже если перемещение производится не из временного объекта. Как можно вызвать перегрузку, принимающую `rvalue`-ссылку для невременного именованного объекта-`lvalue`? Конечно же, при помощи `std::move`-преобразования! Изменим тестовый код в `main` из примера 8.

Пример 9

```
int main()
{
    DynArray ar1{ arrayFactory(10) };
    std::cout << "ar1 elements: ";
    ar1.print();
    std::cout << "\nMove content from ar1 to ar2.\n\n";
    DynArray ar2{ std::move(ar1) }; /* ar1 теперь "пуст",
                                    но "жив"! */

    std::cout << "ar1 elements: ";
    ar1.print();
    std::cout << "ar2 elements: ";
    ar2.print();
    std::cout << "\nReuse ar1.\n\n";
    ar1 = arrayFactory(5); /* ar1 "жив" а следовательно
                           может быть повторно
                           использован */
}
```

```

std::cout << "ar1 elements: ";
ar1.print();

return 0;
}

```

Вывод результата работы кода из примера 9:

```

Microsoft Visual Studio Debug Console

DynArr constructed for 10 elements, for 0077FC44
DynArr move constructed for 10 elements, for 0077FD78
Try to free memory from DynArray for 00000000 pointer
DynArr destructed for 0 elements, for 0077FC44
ar1 elements: 1 7 4 0 9 4 8 8 2 4

Move content from ar1 to ar2.

DynArr move constructed for 10 elements, for 0077FD68
ar1 elements:
ar2 elements: 1 7 4 0 9 4 8 8 2 4

Reuse ar1.

DynArr constructed for 5 elements, for 0077FC44
DynArr move constructed for 5 elements, for 0077FC98
Try to free memory from DynArray for 00000000 pointer
DynArr destructed for 0 elements, for 0077FC44
DynArr move assigned for 5 elements, for 0077FD78
Try to free memory from DynArray for 00000000 pointer
DynArr destructed for 0 elements, for 0077FC98
ar1 elements: 5 5 1 7 1
Try to free memory from DynArray for 00995CD0 pointer
DynArr destructed for 10 elements, for 0077FD68
Try to free memory from DynArray for 00994DF8 pointer
DynArr destructed for 5 elements, for 0077FD78

```

Рисунок 10

Прежде чем разобрать работу примера 9, следует понять, что именно делает функция `std::move`. Единственное, что выполняет вызов функции `std::move` это

получая параметром объект произвольной категории — `lvalue` или `rvalue` — производит безусловное преобразование к `rvalue`, лишь это и более ничего. Дальнейшая судьба преобразованного таким образом объекта зависит от производимых с ним действий. Сама функция `std::move` никак не модифицирует внутреннее состояние своего аргумента.

3. Заданные по умолчанию и удаленные методы

Среди всевозможных функций-членов, которые программист может создать в классе, ряд имеет особое, специальное значение — это так называемые *special member function* или специальные функции-члены. В случае, когда программист не предоставил свои реализации для вышеупомянутых специальных функций-членов, компилятор сгенерирует для них версии по умолчанию. К таким функциям-членам относятся:

- конструктор по умолчанию,
- конструктор копирования,
- оператор копирующего присваивания,
- конструктор перемещения,
- оператор перемещающего присваивания,
- деструктор.

Генерация специальных функций-членов позволяет пользовательским классам иметь схожее с базовыми типами C++ и структурами поведение. Появляется возможность создавать, уничтожать, копировать и перемещать классы, которые не предоставили свои реализации данных функций-членов. Это крайне удобно для несложных классов, не реализующих чего-то, что требует нетривиальных версий специальных функций-членов. Однако компилятор не всегда производит автоматическую генерацию всех или некоторых

специальных функций-членов. Вот ряд правил, которым подчиняется автоматическая генерация специальных функций членов:

- явное определение собственной версии произвольного конструктора предотвращает автоматическую генерацию компилятором конструктора по умолчанию;
- явное определение конструктора перемещения или оператора перемещающего присваивания предотвращает автоматическую генерацию компилятором как конструктора копирования, так и оператора копирующего присваивания;
- явное определение конструктора копирования, оператора копирующего присваивания, конструктора перемещения, оператора перемещающего присваивания или деструктора предотвращает автоматическую генерацию компилятором как конструктора перемещения, так и оператора перемещающего присваивания.

Все вышеперечисленное также может отразиться на механизме наследования. Например, в случае отсутствия в классе-родителе явно определенного или сгенерированного компилятором конструктора без параметров, конструктора по умолчанию, для классов наследников будет невозможна автоматическая генерация конструктора по умолчанию.

Рассмотрим пример, иллюстрирующий предотвращение автоматической генерации конструктора по умолчанию. Создадим простой класс, моделирующий точку, изначально без конструкторов, только с соответствующими функциями-сеттерами.

Пример 10

```
#include <iostream>

class Point
{
    int x;
    int y;
public:
    Point& setX(int pX) { x = pX; return *this; }
    Point& setY(int pY) { y = pY; return *this; }

    void showPoint() const
    {
        std::cout << '(' << x << ',' << y << ')';
    }
};

int main()
{
    Point p0;
    p0.setX(10).setY(20);
    p0.showPoint();
    std::cout << '\n';
    return 0;
}
```

Вывод результата работы кода из примера 10:

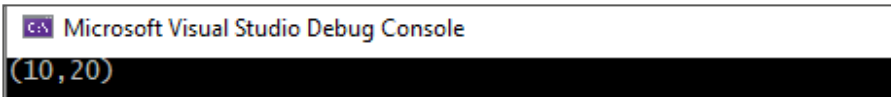


Рисунок 11

Результат работы примера очевиден, так как нет явно определенного конструктора по умолчанию, компилятор сгенерировал свою версию, именно ею мы пользуемся,

создавая экземпляр класса `Point p0`. Но что будет, если мы добавим какой-то конструктор с параметрами? Например, вот такой:

```
Point(int pX, int pY) : x{ pX }, y{ pY } {}
```

Пример выше перестанет компилироваться с ошибкой

```
E0291 no default constructor exists for class "Point"
```

Все верно, согласно вышеизложенным правилам, явное определение конструктора с параметрами предотвратило автоматическую генерацию компилятором собственной версии конструктора по умолчанию. Как решить данную проблему? Очевидным решением будет добавить явное определение конструктора по умолчанию с пустым телом, «конструктор-пустышка».

```
Point() {}
```

Однако более элегантным и оптимальным с точки зрения результирующего кода будет использовать ключевое слово `default`, для информирования компилятора о необходимости сгенерировать версию по умолчанию.

```
Point() = default;
```

Рассмотрим обновленный пример.

Пример 11

```
#include <iostream>

class Point
{
```

```

    int x;
    int y;
public:
    Point() = default;
    Point(int pX, int pY) : x{ pX }, y{ pY } {}
    Point& setX(int pX) { x = pX; return *this; }
    Point& setY(int pY) { y = pY; return *this; }

    void showPoint() const
    {
        std::cout << '(' << x << ',' << y << ')';
    }
};

int main()
{
    Point p0;
    Point p1{ 28,29 };

    p0.setX(10).setY(20);
    std::cout << "p0: ";
    p0.showPoint();
    std::cout << '\n';
    std::cout << "p1: ";
    p1.showPoint();
    std::cout << '\n';

    return 0;
}

```

Вывод результата работы кода из примера 11:

```

Microsoft Visual Studio Debug Console
p0: (10,20)
p1: (28,29)

```

Рисунок 12

Теперь мы можем воспользоваться как явно определенным конструктором с параметрами для инициализации экземпляра `p1`, так и версией конструктора по умолчанию, которую сгенерировал компилятор для инициализации экземпляра `p0`.

Еще одним способом применения ключевого слова `default` является явное объявление специальной функции-члена класса, которая должна быть сгенерирована компилятором с отличным от `public` модификатором доступа.

```
class Point
{
    int x;
    int y;
    Point() = default;

public:
    Point(int pX, int pY) : x{ pX }, y{ pY } {}
    Point& setX(int pX) { x = pX; return *this; }
    Point& setY(int pY) { y = pY; return *this; }

    void showPoint() const
    {
        std::cout << '(' << x << ',' << y << ')';
    }
};
```

Теперь использовать конструктор по умолчанию можно только внутри самого класса `Point`, а попытки его использования извне будут приводить к ошибкам на этапе компиляции.

Использовать ключевое слово `default` возможно только для специальных функций-членов, которые

компилятор может генерировать автоматически. Для других функций-членов класса использование невозможно и бессмысленно. Ключевое слово `default` может показаться не особо полезным или вовсе ненужным, но это не так! С каждым новым стандартом язык C++ старается вносить все более удобные средства для повышения семантичности кода, так, чтобы программа не только работала, но и чтобы ее текст явно и недвусмысленно выражал намерения и мысли автора. Ключевое слово `default` — одно из таких нововведений, пришедших со стандартом C++11. Ведь что такое специальная функция-член с пустым телом? Это оплошность — автор забыл прописать тело? Это заготовка — в следующей версии появится тело соответствующей функции? А использование ключевого слова `default` сразу вносит совершенно недвусмысленную ясность — данная специальная функция-член должна быть сгенерирована компилятором автоматически!

Следующим ключевым словом, повышающим ясность и семантичность кода, является ключевое слово `delete`, применяемое к произвольным функциям-членам класса, а также к обычным функциям не-членам какого-либо класса. Смысл его заключается в явном запрете использовать функцию с данной сигатурой.

В предыдущем разделе урока было рассказано про семантику перемещения и про ее положительные стороны в плане эффективности использования памяти и избегания неоправданного копирования. Освоив семантику перемещения, нам захочется запретить семантику копирования для некоторых классов. Как же фактически выразить в программе на C++ запрет на копирование экземпляров

класса? Самым простым способом будет просто не декларировать конструктор копирования и оператор присваивания копированием. Тогда, при наличии конструктора перемещения и/или оператора присваивания перемещением компилятор неявно запретит использовать копирование, попытки копирования экземпляров класса будут приводить к ошибкам компиляции. В качестве иллюстрации рассмотрим знакомый нам пример динамического массива, из которого мы просто удалим конструктор копирования и оператор присваивания копированием:

Пример 12

```
#include <iostream>
class DynArray
{
    int* arr;
    int size;
public:
    DynArray(int sizeP)
        : arr{ new int[sizeP] {} }, size{ sizeP }
    {
        std::cout << "DynArr constructed for " << size
                    << " elements, for " << this << '\n';
    }
    DynArray() : DynArray(5) {}
    DynArray(DynArray&& object)
        : arr{ object.arr }, size{ object.size }
    {
        object.arr = nullptr;
        object.size = 0;
        std::cout << "DynArr move constructed for "
                    << size << " elements, for " << this
                    << '\n';
    }
}
```

```

DynArray& operator=(DynArray&& object)
{
    if (!(this == &object))
    {
        delete arr;
        arr = object.arr;
        size = object.size;
        object.arr = nullptr;
        object.size = 0;
    }
    std::cout << "DynArr move assigned for "
               << size << " elements, for " << this
               << '\n';
    return *this;
}
int getElem(int idx)const { return arr[idx]; }
void setElem(int idx, int val) { arr[idx] = val; }
void print()const;
void randomize();
~DynArray()
{
    std::cout << "Try to free memory from DynArray for"
               << arr << " pointer\n";
    delete[] arr;
    std::cout << "DynArr destructed for " << size
               << " elements, for " << this << '\n';
}
};

void DynArray::print()const
{
    for (int i{ 0 }; i < size; ++i)
    {
        std::cout << arr[i] << ' ';
    }
    std::cout << '\n';
}

```

```

void DynArray::randomize()
{
    for (int i{ 0 }; i < size; ++i)
    {
        arr[i] = rand() % 10;
    }
}

DynArray arrayFactory(int arrSize)
{
    DynArray arr{ arrSize };
    arr.randomize();
    return arr;
}

int main()
{
    DynArray ar1{ arrayFactory(10) };
    std::cout << "ar1 elements: ";
    ar1.print();

    DynArray ar2{ ar1 };
    std::cout << "ar2 elements: ";
    ar2.print();

    return 0;
}

```

При компиляции данного примера у нас возникнет две ошибки компиляции:

```

E1776 function "DynArray::DynArray(const DynArray &)"
      (declared implicitly) cannot be referenced --
      it is a deleted function
C2280 'DynArray::DynArray(const DynArray &)': attempting
      to reference a deleted function

```


Первая ошибка говорит о том, что компилятор автоматически неявно декларировал некий «удаленный» вариант конструктора, который невозможно использовать, вторая ошибка говорит о попытке использования «удаленной» функции. В принципе, все как и должно быть, но, опять же, неясно, отсутствие функций-членов для копирования — это ошибка или так сделано намеренно?

«Явное лучше, чем неявное» — цитата из так называемого «Дзена Python», свода правил программирования и проектирования, которого придерживаются разработчики языка программирования Python, как нельзя лучше подходит в данном случае. Рассмотрим, как же можно явно выразить запрет на копирование экземпляров класса. До появления стандарта C++11 классическим способом явного запрета копирования было объявление конструктора копирования и оператора присваивания копированием с модификатором доступа `private` без собственно определения этих функций-членов.

```
private:
    DynArray(DynArray& object);
    DynArray& operator=(DynArray& object);
```

Однако данный метод также не лишен недостатков. Все еще есть возможность попытаться использовать «спрятанные» функции-члены изнутри самого класса, что приведет к неочевидным на первый взгляд ошибкам, а также все еще неясно, отсутствие определения — это ошибка или так сделано специально?

Наиболее ясный и непротиворечивый способ — использование ключевого слова `delete` для явного указания того, что данные функции-члены «удалены» и их использование запрещено.

```
public:
    DynArray(DynArray& object) = delete;
    DynArray& operator=(DynArray& object) = delete;
```

Добавив такие декларации в пример 12, мы получим те же сообщения об ошибке, что были при компиляции самого примера 12 с единственной разницей, что на сей раз функции-члены удалены нами явно, а не компилятором не явно.

Рассмотрим еще одну проблему, в решении которой нам может помочь ключевое слово `delete` в применении к функции-члену класса. В этом нам поможет знакомый класс `Point`:

Пример 13

```
#include <iostream>

class Point
{
    int x;
    int y;

public:
    Point() = default;
    Point(int pX, int pY) : x{ pX }, y{ pY } {}
    Point& setX(int pX) { x = pX; return *this; }
    Point& setY(int pY) { y = pY; return *this; }
```

```

void showPoint() const
{
    std::cout << '(' << x << ',' << y << ')';
}
};


int main()
{
    Point p0;

    p0.setX(28.02).setY(29.02);
    std::cout << "p0: ";
    p0.showPoint();
    std::cout << '\n';

    return 0;
}

```

Вывод результата работы кода из примера 13:



Microsoft Visual Studio Debug Console
p0: (28,29)

Рисунок 13

В примере 13 мы воспользовались функциями-членами `setX` и `setY` с `double`-литералами в качестве параметров. Вследствие неявного преобразования типа `double` к `int` стал возможен вызов сеттеров, декларированных с `int`-параметрами. Нередко такая неявная конвертация безвредна или даже желательна. Однако что делать, если автор класса хочет явно запретить неоднозначное использование каких-то функций-членов из-за нежелательного неявного преобразования типов? Разумеется, использовать ключевое слово `delete` для явного запрета

использования функции-члена с нежелательными параметрами. Добавим в определение класса `Point` из примера 13 фрагмент:

```
public:
    Point& setX(double pX) = delete;
    Point& setY(double pY) = delete;
```

Таким образом мы явно запретили использовать в качестве параметров значения типа `double` и неявно, через ближайшее преобразование, типа `float`.

Аналогичным образом возможно использовать ключевое слово `delete` для свободных функций, не являющихся членами какого-либо класса:

Пример 14

```
#include <iostream>

int max(int a, int b) { return a > b ? a : b; }
template <typename T1, typename T2> int max(T1 a, T2 b) =
    delete;

int main()
{
    std::cout << max(20, 30) << '\n';
    std::cout << max(20.5, 30) << '\n';
    std::cout << max('z', false) << '\n';
    std::cout << max('a', 'b') << '\n';

    return 0;
}
```

Пример 14 на первый взгляд кажется немного сложным для понимания, но немного пояснений, и все станет понятно. Итак, мы определили свободную функцию с двумя `int` параметрами для нахождения максимума из двух целых чисел. Как мы помним, все базовые типы C++ имеют возможность взаимного преобразования друг в друга, посему мы потенциально можем вызвать функцию `max` с любой комбинацией двух типов фактических параметров. В некоторых случаях это естественное и желаемое поведение, но если разработчик функции этого не желает? Как явно запретить вызов функции с параметрами отличными от `int`? Очень просто — применим к решению этой проблемы определение шаблона функции и ключевое слово `delete`.

Объявим шаблон функции `max` с двумя разнотипными шаблонными параметрами и пометим такой шаблон как удаленный, используя ключевое слово `delete`. Теперь при попытке вызвать функцию `max` с двумя фактически параметрами `int`, компилятор вызовет наличествующую функцию `max`. Однако в случае вызова функции `max` с любыми другими типами и комбинациями типов фактических параметров компилятор будет вынужден искать наиболее подходящую перегрузку. По правилам поиска перегрузок функций, наиболее подходящим окажется определенный нами шаблон, который, собственно, и запретит использование функции — явно и недвусмысленно. Единственный вопрос, который все еще мог остаться, так это зачем в шаблоне два параметра типа, почему бы не обойтись одним? Все дело в том, что в случае, когда функция `max` будет вызвана с двумя разнотипными

фактическими параметрами, то, по все тем же правилам поиска наиболее подходящей перегрузки, такой шаблон не подойдет, ведь он рассчитан на два однотипных параметра. Присутствующий же в примере шаблон запретит все, кроме первого `max(20,30)`, вызовы функции с любыми комбинациями типов фактических параметров.

4. Специальные перегрузки — перегрузка []

Изучая различные темы урока, мы часто использовали в качестве примера динамический массив `DynArray`. Обратимся же к нему и для иллюстрации данного раздела нашего урока. В примере 12 содержится последняя версия класса `DynArray`, начнем именно с нее. В настоящий момент доступ к элементам массива реализован парой методов

```
int getElem(int idx) const { return arr[idx]; }  
void setElem(int idx, int val) { arr[idx] = val; }
```

В полной ли мере удобен такой подход? Нет. Разделение на функцию-член для чтения элемента и отдельную функцию-член для записи — не самое удобное решение. Более того, при использовании функции-члена `setElem` может возникнуть неоднозначность, например, что означает `setElem(1,2)` — присваивание элементу с индексом `1` значения `2` или присваивание значения `1` элементу с индексом `2`? Не видя сигнатуры функции, это совершенно неясно! Исправить ситуацию поможет использование привычного нам по работе с обычными массивами способа доступа к элементам через квадратные скобки. Для этого нам понадобится перегрузить `operator[]` функцией-членом класса `DynArray`. Перегрузка данного оператора допускается только в виде функции-члена класса и никак иначе. Перегрузка оператора `operator[]` может

принимать лишь один параметр и возвращать значение через оператор `return`.

Прежде чем перейти к, собственно, реализации перегрузки `operator[]`, необходимо рассмотреть два важных вопроса. Вопрос первый — что именно будет возвращать перегрузка `operator[]`? Если элемент массива по значению, то не удастся использовать данную перегрузку для модификации элемента массива, а ведь не в последнюю очередь для удобной модификации нам и нужна данная перегрузка. Если же возвращать элемент массива по неконстантной ссылке, то как обеспечить семантику константности нашего массива? Решением данного вопроса будет реализовать две перегрузки `operator[]`: одну как константную функцию-член, возвращающую элемент массива по значению, и одну — как обычную функцию-член, возвращающую элемент массива по неконстантной ссылке.

Второй вопрос, который нам необходимо решить, — проверять ли допустимость индекса массива, и если да, то что делать в случае выхода индекса за допустимые пределы? Отсутствие проверки допустимости индекса в обычных массивах C++ — дань производительности в ущерб надежности и безопасности. Считается, что программист сам в состоянии отследить допустимость индекса. Наш массив — учебный пример, поэтому не стоит полагаться в полной мере на программистов-студентов, стоит все же помочь им и таки проконтролировать допустимость индексов. Однако с этим решением связана еще одна проблема: а что делать в случае недопустимого индекса? Какое значение возвращать как ошибку? Хорошего

ответа на данном этапе нашего обучения пока нет, немногим далее по курсу будет изучен механизм исключений C++, который отлично подходит для решения данной проблемы. В настоящий же момент мы воспользуемся временным решением «заглушкой» — макросом `assert`. Суть работы `assert` — вычислить некоторое логическое утверждение и, в случае его ложности, вывести на экран само ложное утверждение, номер строки в программе с макросом `assert` и досрочно аварийно завершить работу программы. Чтобы сделать вывод `assert` более информативным, после проверяемого утверждения через `&&` (and) можно добавить произвольную строку, которая также отразится на экране в случае выполнения самого `assert`.

Сигнатура оператора индексации, «оператора квадратных скобки», имеет нижеследующий вид:

```
ReturnType operator[](IndexType index) const // 1
ReturnType& operator[](IndexType index) // 2
```

где `ReturnType` — тип возвращаемого значения, а `IndexType` — тип формального параметра-индекса. Вариант 1 — константная перегрузка функции-члена, вариант 2 — не константная перегрузка.

Пример для `DynArray`:

```
int operator[](int idx) const // 1
int& operator[](int idx) // 2
```

Теперь, обладая всем необходимым, можно приступить к реализации оператора индексации для класса `DynArray`.

Пример 15

```

#include <iostream>
#include <cassert>

class DynArray
{
    int* arr;
    int size;

public:
    DynArray(int sizeP)
        : arr{ new int[sizeP] {} }, size{ sizeP }
    {
        std::cout << "DynArr constructed for " << size
                    << " elements, for " << this << '\n';
    }
    DynArray() : DynArray(5) {}
    DynArray(DynArray& object) = delete;
    DynArray& operator=(DynArray& object) = delete;
    DynArray(DynArray&& object)
        : arr{ object.arr }, size{ object.size }
    {
        object.arr = nullptr;
        object.size = 0;
        std::cout << "DynArr move constructed for "
                    << size << " elements, for " << this
                    << '\n';
    }

    DynArray& operator=(DynArray&& object)
    {
        if (!(this == &object))
        {
            delete arr;

            arr = object.arr;
            size = object.size;
        }
    }
};

```

```

        object.arr = nullptr;
        object.size = 0;
    }
    std::cout << "DynArr move assigned for "
                << size << " elements, for " << this
                << '\n';

    return *this;
}
// константная перегрузка, возвращающая элемент
// по значению
int operator[](int idx) const
{
    assert(idx >= 0 and idx < size and "Index is out "
                                             "of range!");

    return arr[idx];
}
// не константная перегрузка, возвращающая элемент
// по ссылке
int& operator[](int idx)
{
    assert(idx >= 0 and idx < size and "Index is out"
                                             "of range!");

    return arr[idx];
}
void print() const;
void randomize();
~DynArray()
{
    std::cout << "Try to free memory from DynArray for"
                << arr << " pointer\n";
    delete[] arr;
    std::cout << "DynArr destructed for " << size
                << " elements, for " << this << '\n';
}
};

```

```

void DynArray::print()const
{
    for (int i{ 0 }; i < size; ++i)
    {
        std::cout << arr[i] << ' ';
    }
    std::cout << '\n';
}

void DynArray::randomize()
{
    for (int i{ 0 }; i < size; ++i)
    {
        arr[i] = rand() % 10;
    }
}

DynArray arrayFactory(int arrSize)
{
    DynArray arr{ arrSize };
    arr.randomize();
    return arr;
}

int main()
{
    const int arrSize{ 10 };
    DynArray ar1{ arrayFactory(arrSize) };
    std::cout << "ar1 elements : ";
    ar1.print();

    std::cout << "\nChange every ar1 element to its "
               "square:\n";
    for (int i{ 0 }; i < arrSize; ++i)
    {
        ar1[i] *= ar1[i];
        std::cout << "ar1[" << i << "] = " << ar1[i] << '\n';
    }
}

```

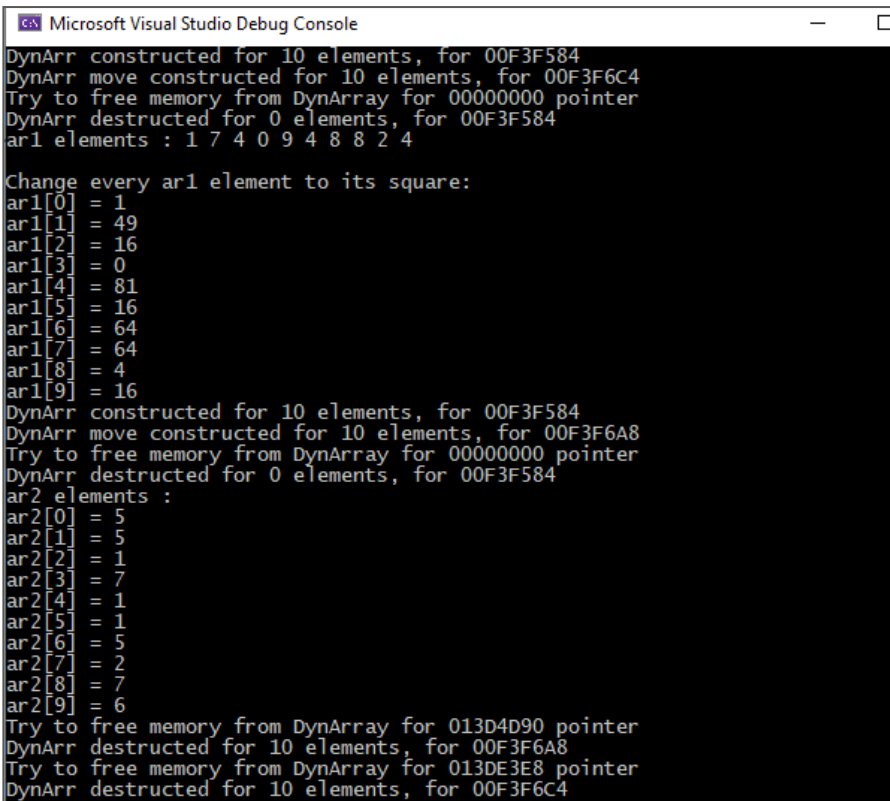
```

const DynArray ar2{ arrayFactory(arrSize) };
std::cout << "ar2 elements :\n";
for (int i{ 0 }; i < arrSize; ++i)
{
    std::cout << "ar2[" << i << "] = " << ar2[i] << '\n';
}

return 0;
}

```

Вывод результата работы из примера 15.



```

Microsoft Visual Studio Debug Console
DynArr constructed for 10 elements, for 00F3F584
DynArr move constructed for 10 elements, for 00F3F6C4
Try to free memory from DynArray for 00000000 pointer
DynArr destructed for 0 elements, for 00F3F584
ar1 elements : 1 7 4 0 9 4 8 8 2 4

Change every ar1 element to its square:
ar1[0] = 1
ar1[1] = 49
ar1[2] = 16
ar1[3] = 0
ar1[4] = 81
ar1[5] = 16
ar1[6] = 64
ar1[7] = 64
ar1[8] = 4
ar1[9] = 16
DynArr constructed for 10 elements, for 00F3F584
DynArr move constructed for 10 elements, for 00F3F6A8
Try to free memory from DynArray for 00000000 pointer
DynArr destructed for 0 elements, for 00F3F584
ar2 elements :
ar2[0] = 5
ar2[1] = 5
ar2[2] = 1
ar2[3] = 7
ar2[4] = 1
ar2[5] = 1
ar2[6] = 5
ar2[7] = 2
ar2[8] = 7
ar2[9] = 6
Try to free memory from DynArray for 013D4D90 pointer
DynArr destructed for 10 elements, for 00F3F6A8
Try to free memory from DynArray for 013DE3E8 pointer
DynArr destructed for 10 elements, for 00F3F6C4

```

Рисунок 14

В примере 15 определяются две перегрузки функции-члена `operator[]`: одна для не константных объектов, вторая для константных. Далее мы проверяем работоспособность обоих вариантов: для `ar1` мы заменяем каждый элемент его квадратом и используя все ту же перегрузку, выводим массив на экран в более «нестандартном», чем предусмотрено функцией-членом `print()` виде. Аналогично проверяем работоспособность константной перегрузки, просто выводим массив на экран.

Важно отметить, что для перегрузки функции-члена `operator[]` определено лишь количество формальных параметров, а именно строго один, но не его тип! Типом параметра-индекса может быть все, что разработчик класса сочтет необходимым и целесообразным.

Рассмотрим пример с более нестандартным параметром для `operator[]`, а именно строкой. Наш пример будет моделировать итоговую таблицу спортивных медалей по странам. Вместо полного наименования страны будут использоваться трехбуквенные сокращения. Данный пример будет немного схематичен, но вполне достаточен для иллюстрирования возможности использовать строки в качестве индексов в контейнере. Для простоты примера в нем не будет использоваться динамическая память, как следствие отпадет необходимость в явной реализации семантики копирования и перемещения — автоматически сгенерированные компилятором специальные функции-члены в полной мере справятся с необходимыми задачами.

Для начала нам понадобится класс, моделирующий строку нашей таблицы с медалями. Данный класс будет

хранить строковую аббревиатуру страны, а также общее количество для золотых, серебряных и бронзовых медалей в виде целочисленного массива.

Пример 16

```
#include <iostream>
#include <cassert>

class MedalRow
{
    char country[4];
    int medals[3];

public:
    /* определяем константы для удобного и однозначного
       доступа к элементам массива */
    static const int GOLD{ 0 };
    static const int SILVER{ 1 };
    static const int BRONZE{ 2 };

    MedalRow(const char* countryP, const int* medalsP)
    {
        strcpy_s(country, 4, countryP ? countryP : "NON");
        for (int i{ 0 }; i < 3; ++i)
        {
            medals[i] = medalsP ? medalsP[i] : 0;
        }
    }

    MedalRow() : MedalRow(nullptr, nullptr) {}
    MedalRow& setCountry(const char* countryP)
    {
        if (countryP)
        {
            strcpy_s(country, 4, countryP);
        }
    }
}
```

```

        return *this;
    }
    const char* getCountry()const { return country; }
    int& operator[](int idx)
    {
        assert((idx >= 0 and idx < 3) and "Index out "
               "of range!");
        return medals[idx];
    }

    int operator[](int idx)const
    {
        assert((idx >= 0 and idx < 3) and "Index out "
               "of range!");
        return medals[idx];
    }

    void print()const
    {
        std::cout << '[' << country << "]-(" << " ";
        for (int i{ 0 }; i < 3; ++i)
        {
            std::cout << medals[i];
            if (i < 2) { std::cout << '\t'; }
        }
        std::cout << " )\n";
    }
};

int main()
{
    MedalRow mr;
    mr.setCountry("UKR");
    std::cout << "Country is: " << mr.getCountry() << '\n';
    mr[MedalRow::GOLD] = 3;
    mr[MedalRow::BRONZE] = 2;
    mr[MedalRow::SILVER] = 4;
}

```



```

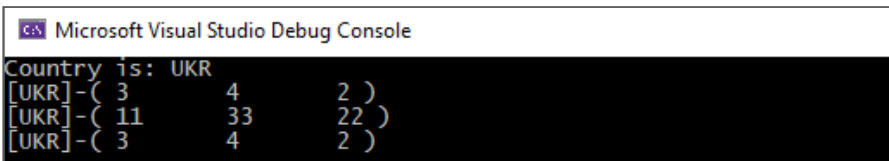
/* Создаем строку-копию, на основе mr */
MedalRow mr1{ mr };
/* Убеждаемся, что создалась копия */
mr1.print();
/* Модифицируем значения строки-копии, выводим ее
   на экран*/
mr1[MedalRow::GOLD] = 11;
mr1[MedalRow::BRONZE] = 22;
mr1[MedalRow::SILVER] = 33;
mr1.print();

/* Убеждаемся, что оригинал остался не модифицированным */
mr.print();

return 0;
}

```

Вывод результата работы из примера 16.



```

Microsoft Visual Studio Debug Console
Country is: UKR
[UKR]-( 3      4      2 )
[UKR]-( 11     33     22 )
[UKR]-( 3      4      2 )

```

Рисунок 15

В примере 16 также используются две перегрузки функции-члена для `operator[]` константная и неконстантная и знакомый нам механизм `assert`. Обратите внимание, что для удобства и большей читаемости кода в классе `MedalRow` присутствуют три статические константы;

```

static const int GOLD{ 0 };
static const int SILVER{ 1 };
static const int BRONZE{ 2 };

```

Протестируем работу `assert` в перегрузке функционального оператора `operator[]`, намеренно указав недопустимый индекс для нашей строки с медалями:

Пример 17

```
#include <iostream>
#include <cassert>

class MedalRow
{
    char country[4];
    int medals[3];

public:
    /* определяем константы для удобного и однозначного
       доступа к элементам массива */
    static const int GOLD{ 0 };
    static const int SILVER{ 1 };
    static const int BRONZE{ 2 };

    MedalRow(const char* countryP, const int* medalsP)
    {
        strcpy_s(country, 4, countryP ? countryP : "NON");

        for (int i{ 0 }; i < 3; ++i)
        {
            medals[i] = medalsP ? medalsP[i] : 0;
        }
    }

    MedalRow() : MedalRow(nullptr, nullptr) {}
    MedalRow& setCountry(const char* countryP)
    {
        if (countryP)
        {
            strcpy_s(country, 4, countryP);
        }
    }
}
```

```

        return *this;
    }
    const char* getCountry()const { return country; }

    int& operator[](int idx)
    {
        assert((idx >= 0 and idx < 3) and "Index out "
              "of range!");
        return medals[idx];
    }

    int operator[](int idx)const
    {
        assert((idx >= 0 and idx < 3) and "Index out "
              "of range!");
        return medals[idx];
    }

    void print()const
    {
        std::cout << '[' << country << "]-(" << " ";
        for (int i{ 0 }; i < 3; ++i)
        {
            std::cout << medals[i];
            if (i < 2) { std::cout << '\t'; }
        }
        std::cout << ")\n";
    }
};

int main()
{
    MedalRow mr;
    mr.setCountry("UKR");
    std::cout << "Country is: " << mr.getCountry() << '\n';
    mr[MedalRow::GOLD] = 3;
    mr[MedalRow::BRONZE] = 2;
}

```

```

/* намеренная ошибка! в массиве нет элемента с таким
   индексом!
   * В процессе работы программы возникнет ошибка!
   * В данном примере это нормально.
   */
mr[8] = 4;

return 0;
}

```

Вывод результата работы из примера 17

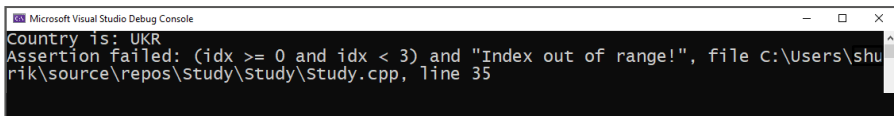


Рисунок 16

Как видно из вывода 17, при попытке обратиться к недопустимому индексу, сработал наш механизм проверки посредством макроса `assert`, что привело к аварийному завершению работы программы и выводу имени файла и строки кода, в которой произошла ошибка. Напомним, что макрос `assert` — лишь временное решение до тех пор, пока не будут изучены исключения C++ (C++ exceptions).

Продолжим создавать модель таблицы спортивных медалей. Реализуем собственно класс-таблицу `MedalsTable`. Данный класс будет хранить статический массив из 10 элементов типа `MedalRow`, а также переменную-член `size`, содержащую количество «заполненных» строк таблицы. Изначально конструктор по умолчанию `MedalRow` создает объект, в котором количество медалей всех видов — 0, и аббревиатура для страны — «NON». Такая

строка в таблице медалей будет считаться «строкой-пустышкой». По умолчанию в классе `MedalsTable` создается массив именно таких «строк-пустышек», и конструктор по умолчанию устанавливает переменную-член `size` в 0, то есть таблица считается пустой.

Далее для доступа к нашей таблице определяются две перегрузки функции-члена для `operator[]`. Тут и начинается наиболее примечательная часть, ради которой и создавался данный пример. В качестве параметра для `operator[]` будет использоваться вовсе не целочисленный индекс, как мы привыкли, а строка! Указывая строку-аббревиатуру страны в качестве фактического параметра, будет обеспечиваться доступ к соответствующей строке таблицы медалей. В случае если строки с такой аббревиатурой нет в таблице и не превышен размер самой таблицы, в нее «прозрачно» добавляется новая строка, соответствующая трехбуквенной аббревиатуре страны. Неконстантная перегрузка возвращает ссылку на найденную или добавленную строку типа `MedalRow`. В константной же версии функции-члена `operator[]` отсутствует механизм автоматического добавления строк для сохранения семантики константности объекта. В случае отсутствия строки, соответствующей аббревиатуре, срабатывает макрос `assert` и работа аварийно завершается. Также константная перегрузка возвращает объект `MedalRow` по константной ссылке.

Для вывода на экран заполненной части таблицы медалей присутствует функция-член `print`. Для поиска строки по трехбуквенной аббревиатуре присутствует приватная функция-член `findCountry`, возвращающая

индекс соответствующей строки `MedalRow` в массиве `MedalsTable` или `-1` при ее отсутствии.

Пример 18

```
#include <iostream>
#include <cassert>

class MedalRow
{
    char country[4];
    int medals[3];

public:
    static const int GOLD{ 0 };
    static const int SILVER{ 1 };
    static const int BRONZE{ 2 };

    MedalRow(const char* countryP, const int* medalsP)
    {
        strcpy_s(country, 4, countryP ? countryP : "NON");

        for (int i{ 0 }; i < 3; ++i)
        {
            medals[i] = medalsP ? medalsP[i] : 0;
        }
    }

    MedalRow() : MedalRow(nullptr, nullptr) {}
    MedalRow& setCountry(const char* countryP)
    {
        if (countryP)
        {
            strcpy_s(country, 4, countryP);
        }
        return *this;
    }

    const char* getCountry()const { return country; }
```

```

int& operator[](int idx)
{
    assert((idx >= 0 and idx < 3) and "Index out "
           "of range!");
    return medals[idx];
}

int operator[](int idx)const
{
    assert((idx >= 0 and idx < 3) and "Index out "
           "of range!");
    return medals[idx];
}

void print()const
{
    std::cout << '[' << country << "]-(" << " ";
    for (int i{ 0 }; i < 3; ++i)
    {
        std::cout << medals[i];
        if (i < 2) { std::cout << '\t'; }
    }
    std::cout << " )\n";
}
};

class MedalsTable
{
public:
    static const int maxSize{ 10 };
private:
    MedalRow medalRows[MedalsTable::maxSize];
    int size;
    int findCountry(const char* country)const
    {
        for (int i{ 0 }; i < size; ++i)
        {

```

```

        if (strcmp(medalRows[i].getCountry(),
                    country) == 0)
        {
            return i;
        }
    }
    return -1;
}

public:
    MedalsTable() : size{ 0 } {};

    MedalRow& operator[](const char* country)
    {
        int idx{ findCountry(country) };

        if (idx == -1)
        {
            assert(size < MedalsTable::maxSize and
                   "Table is FULL!");
            idx = size++;
            medalRows[idx].setCountry(country);
        }

        return medalRows[idx];
    }

    const MedalRow& operator[](const char* country)const
    {
        int idx{ findCountry(country) };
        assert(idx != -1 and "Country not found on const "
                   "table");
        return medalRows[idx];
    }

    void print()const
    {

```



```

        for (int i{ 0 }; i < size; ++i)
        {
            medalRows[i].print();
        }
    }
};

int main()
{
    MedalsTable mt1;

    std::cout << "Medals table #1:\n";

    mt1["UKR"][MedalRow::GOLD] = 14;
    mt1["UKR"][MedalRow::SILVER] = 5;
    mt1["HUN"][MedalRow::BRONZE] = 9;
    mt1["HUN"][MedalRow::GOLD] = 7;
    mt1["POL"][MedalRow::GOLD] = 4;
    mt1["POL"][MedalRow::SILVER] = 2;

    mt1.print();

    // создаем константную копию таблицы №1
    std::cout << "\nMedals table #2:\n";

    const MedalsTable mt2{ mt1 };
    mt2.print();

    // раскомментировав следующую строку можно протестировать
    // проверку отсутствия страны в константной таблице
    // медалей
    // программа аварийно завершиться, что нормально!
    // mt2["SLO"].print();

    return 0;
}

```

Вывод результата работы из примера 18

```

Microsoft Visual Studio Debug Console
Medals table #1:
[UKR]-( 14      5      0 )
[HUN]-(  7      0      9 )
[POL]-(  4      2      0 )

Medals table #2:
[UKR]-( 14      5      0 )
[HUN]-(  7      0      9 )
[POL]-(  4      2      0 )

```

Рисунок 17

Итак, в начале примера 18 создается таблица спортивных медалей `mt1`. Затем, используя перегрузку `operator[]` с параметром «UKR» создается новая строка `MedalRow`, которая возвращается по неконстантной ссылке. Результатом является объект `MedalRow`, для которого также используется перегрузка `operator[]` для указания количества золотых медалей. Следующая строка, хотя и выглядит аналогично предыдущей, но работает немного иначе: в таблице уже есть строка `MedalRow` соответствующая аббревиатуре «UKR», именно она и возвращается по неконстантной ссылке как результат `mt1["UKR"]`. Затем все аналогично предыдущей строке: используя перегрузку `operator[]` для `MedalRow`, устанавливается количество серебряных медалей. Далее все происходит аналогично для «HUN» и «POL». Вызов функции-члена `mt1.print()` выводит на экран заполненную часть таблицы спортивных медалей. Для проверки корректной работы константного экземпляра класса `MedalsTable` создается таблица-копия `mt2` на основе `mt1` и ее содержимое также выводится на экран. Раскомментировав строку `mt2["SLO"].print();`,

можно протестировать ситуацию обращения к несуществующему элементу константного объекта — сработает макрос `assert` и программа аварийно завершится. Пример 18 продемонстрировал возможность использования различных типов фактических параметров для реализации перегрузки `operator[]`.

Подытожим для реализации доступа к элементам некоего контейнера удобно и естественно использовать оператор индексации, то есть квадратные скобки. Перегрузка оператора индексации реализуется через определение функции-члена класса `operator[]`. Данный оператор может перегружаться исключительно функцией-членом класса и никак иначе — это требование стандарта C++. Для обеспечения семантики константности возможно создать две версии перегрузки `operator[]`: константную и не константную. У данной перегрузки может быть лишь один формальный параметр произвольного типа, нет ограничения на целочисленность параметра, хотя именно целочисленный параметр используется чаще всего.

5. Специальные перегрузки — перегрузка ()

Настало время познакомиться с еще одним специальным оператором, доступным для перегрузки в C++ — оператором вызова функции, иначе — оператором «круглые скобки» или `operator()`. Данный оператор можно перегрузить исключительно функцией-членом класса. Также `operator()` имеет ряд уникальных особенностей, которые открывают интересные возможности его применения.

Итак, начнем с особенностей оператора вызова функции. Практически все перегружаемые операторы имеют строго определенное количество аргументов, в некоторых случаях регламентируя даже тип аргументов, оставляя лишь тип возвращаемого значения на усмотрение разработчика класса. Однако оператор вызова функции не такой. Сколько в общем случае аргументов может быть у произвольной функции? Произвольное количество. Вот и в случае перегрузки оператора вызова функции количество аргументов, равно как и их тип и порядок следования, не регламентируется. Данная особенность позволяет реализовывать операторы с нетипичным, особенным назначением.

Для каких целей следует использовать перегрузку оператора вызова функции? Непростой вопрос! Большая гибкость, с одной стороны, дает большие возможности, однако, с другой стороны, требует ответственного подхода

при их использовании. Семантика обычных операторов хоть сколько-нибудь формализована, к примеру, оператор сложения выполняет сложение, слияние, объединение, конкатенацию. Оператор проверки эквивалентности, `operator==`, по тем или иным критериям проверяет эквивалентность двух экземпляров классов. А какова семантика оператора вызова функции? В чистом виде для произвольного класса ее нет, она совершенно открыта.

Такая открытость, с одной стороны позволяет произвольную реализацию, но, с другой стороны, суть, смысл действий, выполняемых перегрузкой оператора вызова функции, интуитивно неясен. Хуже того, всегда есть шанс спутать вызов перегруженного оператора вызова функции для экземпляра класса с, собственно, вызовом некоторой функции. Посему мы рекомендуем не злоупотреблять перегрузкой данного оператора дабы не вносить неоднозначность в код.

На этом можно было бы завершить описание, так и не начав его, но мы все же рассмотрим пример полезной и семантически оправданной перегрузки оператора вызова функции на основе класса двумерного динамического массива. Завершим же раздел совершенно новым и неожиданным способом применения оператора вызова функции — созданием функторов.

Начнем с сигнатуры и общего вида данной перегрузки:

```
ReturnType operator>() {} // 1
ReturnType operator()(ParamTypeA paramA) {} // 2
ReturnType operator()(ParamTypeA paramA,
                      ParamTypeB paramB) {} // 3
```

где `ReturnType` — тип возвращаемого значения, а `ParamTypeA`, `ParamTypeB` — типы формальных параметров. Вариант 1 — перегрузка функции-члена без формальных параметров, вариант 2 — с одним формальным параметром, вариант 3 — с двумя. Еще раз отметим, что формальных параметров при перегрузке оператора вызова функции может быть произвольное количество. Также следует отметить, что при определении перегрузки круглые скобки следуют дважды: сперва как часть имени оператора, а затем как скобки для формальных параметров.

Пример для перегрузки, возвращающей целое число и принимающей два целочисленных формальных параметра.

```
int operator()(int y, int x){}
```

Перейдем к рассмотрению практического примера, иллюстрирующего семантически оправданное использование перегрузки оператора вызова функции. Примером послужит класс двумерного динамического массива. Классический пример для такого рода иллюстраций! Классический, да не в полной мере. Обычно двумерный динамический массив создается в два этапа: сперва динамически выделяется память под контейнер строк нашего массива, а затем поочередно выделяется память под отдельные строки двумерного динамического массива. В итоге получается конструкция, которая на этапе использования ничем не отличается от двумерного статического массива — тот же доступ к элементам через

оператор квадратные скобки, который необходимо использовать дважды: для доступа из контейнера к массиву элементов и затем второй раз: для непосредственного доступа к элементу массива-строки.

Приведем упрощенный пример класса, реализующего вышеописанную методику создания двумерного динамического массива.

Пример 19

```
#include <iostream>

class Dyn2DArr
{
    int sizeY;
    int sizeX;

public:
    int** data;
    Dyn2DArr(int sizeYP, int sizeXP)
        : sizeY{ sizeYP }, sizeX{ sizeXP },
          data{ new int*[sizeYP] }
    {
        for (int y{ 0 }; y < sizeY; ++y)
        {
            data[y] = new int[sizeX];
        }
    }

    void print()const
    {
        for (int y{ 0 }; y < sizeY; ++y)
        {
            for (int x{ 0 }; x < sizeX; ++x)
            {
                std::cout << data[y][x] << '\t';
            }
        }
    }
};
```

```

        }
        std::cout << '\n';
    }
    std::cout << '\n';
}

~Dyn2DArr()
{
    for (int y{ 0 }; y < sizeY; ++y)
    {
        delete[] data[y];
    }
    delete[] data;
}

};

int main()
{
    int rows{ 3 };
    int columns{ 3 };
    int counter{ 1 };

    Dyn2DArr arr2d{ rows, columns };

    for (int y{ 0 }; y < rows; ++y)
    {
        for (int x{ 0 }; x < columns; ++x)
        {
            arr2d.data[y][x] = counter++;
        }
    }

    arr2d.print();

    return 0;
}

```


Вывод результата работы из примера 19.



```

Microsoft Visual Studio Debug Console
1      2      3
4      5      6
7      8      9
  
```

Рисунок 18

В примере 19 мы сделали переменную-член `data` публичной, что не очень хорошо с точки зрения инкапсуляции состояния класса, однако сделано для сохранения методологии доступа к элементам массива через двойные квадратные скобки. Привычный доступ к элементам через квадратные скобки используется как в функции-члене `print`, так и в функции `main` при заполнении массива тестовыми значениями. Большую проблему, чем публичная переменная-член `data`, представляет сама методология создания двумерного динамического массива через множественное выделение/освобождение динамической памяти по `new/delete`. Как известно, вызов `new/delete` влечет за собой большие накладные расходы, которые, желательно, минимизировать.

Чтобы избавиться от множественного выделения/освобождения памяти, в корне изменим саму модель данных для двумерного динамического массива. Вместо выделения памяти под собственно элементы массива по частям, выполним выделение памяти одним большим блоком, так чтоб размер блока вместил все элементы двумерного массива сразу. Однако при таком подходе мы лишаемся привычного способа доступа к элементам через пару квадратных скобок, ведь блок памяти-то один!

Эту проблему можно было бы решить, создав контейнер как в Примере 19 и заполнив его адресами соответствующих элементов из большого блока памяти, своего рода «нарезав» его построчно.

Приведем измененный вариант класса:

Пример 20

```
#include <iostream>

class Dyn2DArrLinear
{
    int sizeY;
    int sizeX;

public:
    int** data;
    Dyn2DArrLinear(int sizeYP, int sizeXP)
        : sizeY{ sizeYP }, sizeX{ sizeXP },
          data{ new int*[sizeYP] }
    {
        /*
         * выделяем блок памяти для хранения всех элементов
         * двумерного динамического массива.
         */
        int* dataElements{ new int[sizeY * sizeX] };

        for (int y{ 0 }; y < sizeY; ++y)
        {
            // "нарезаем" блок построчно
            data[y] = dataElements + y * sizeX;
        }
    }

    void print()const
    {
```

```

    for (int y{ 0 }; y < sizeY; ++y)
    {
        for (int x{ 0 }; x < sizeX; ++x)
        {
            std::cout << data[y][x] << '\t';
        }
        std::cout << '\n';
    }
    std::cout << '\n';
}

~Dyn2DArrLinear()
{
    /*
     * адрес начала большого блока dataElements
     * в конструкторе совпадает с адресом первой
     * строки нашего двумерного динамического массива.
     * Сперва освобождаем память из-под элементов
     * массива, затем — память контейнера строк.
     */
    delete[] data[0];
    delete[] data;
}

};

int main()
{
    int rows{ 3 };
    int columns{ 3 };
    int counter{ 1 };

    Dyn2DArrLinear arr2d{ rows, columns };

    for (int y{ 0 }; y < rows; ++y)
    {
        for (int x{ 0 }; x < columns; ++x)
        {

```

```

        arr2d.data[y][x] = counter++;
    }
}

arr2d.print();

return 0;
}

```

Вывод результата работы из примера 20.



Рисунок 19

Как видим, Пример 20 функционально ничем не отличается от Примера 19, мы лишь избавились от множественных `new/delete`. Все тот же привычный и понятный способ доступа к элементам через пару квадратных скобок. Однако этот привычный способ стоит нам дополнительно блока памяти под контейнер адресов строк и «лишним» накладным расходам на двойное разыменование: сперва адреса строки, а затем непосредственного элемента.

Посмотрим, как можно избавиться от недостатков из примеров 19 и 20. Для этого откажемся от контейнера строк, оставив выделение блока памяти под элементы единым непрерывным фрагментом. Как результат, мы потеряем возможность доступа к элементам через двойные квадратные скобки. Сама модель двумерного массива требует «двумерности» доступа к элементам по индексу строки и индексу столбца. Перегрузить оператор

квадратные скобки для указания двух индексов не удастся, ведь, как известно из предыдущего раздела, `operator[]` принимает лишь один-единственный формальный параметр-индекс. И именно тут и приходит на помощь гибкость оператора вызова функции — возможность произвольного количества формальных параметров.

Для доступа к элементам линейно выделенного блока памяти как к двумерному массиву создадим класс `Matrix`. Функционально он повторяет предыдущие классы с той разницей, что его переменную-член `data` нет необходимости делать публичной, что исправляет проблему инкапсуляции. Также в примере класса `Matrix` присутствуют методы добавления/удаления строк/столбцов для демонстрации простоты и эффективности данных утилитарных подзадач с учетом линейности блока данных. Для доступа к собственно элементам нашего двумерного массива с максимально удобной семантикой в классе реализовано две перегрузки оператора вызова функции: константная и неконстантная.

Пример 21

```
#include <iostream>

class Matrix
{
    int sizeY;
    int sizeX;
    int* data;
    int index2D(int y, int x)
        const { return y * sizeX + x; }
    int index2D(int y, int x, int sizeXP)
        const { return y * sizeXP + x; }
```

```

public:
    Matrix(int sizeYP, int sizeXP)
        : sizeY{ sizeYP }, sizeX{ sizeXP },
          data{ new int[sizeYP * sizeXP] }{}
    int operator()(int y, int x)
        const { return *(data + index2D(y, x)); }
    int& operator()(int y, int x)
        { return *(data + index2D(y, x)); }

    void deleteColumn(int columnPos)
    {
        --sizeX;
        int* newData{ new int[sizeY * sizeX] };

        for (int y{ 0 }; y < sizeY; ++y)
        {
            for (int x{ 0 }; x < sizeX; ++x)
            {
                *(newData + index2D(y, x)) =
                    *(data + index2D(y, x + (x >=
                        columnPos)));
            }
        }
        delete[] data;
        data = newData;
    }

    void addColumn(int columnPos, int* newCol = nullptr)
    {
        int* newData{ new int[sizeY * (sizeX + 1)] };

        for (int y{ 0 }; y < sizeY; ++y)
        {
            for (int x{ 0 }; x < sizeX; ++x)
            {
                *(newData + index2D(y, x + (x >= columnPos),
                    sizeX + 1)) = *(data + index2D(y, x));
            }
        }
    }

```

```

    }
    *(newData + index2D(y, columnPos, sizeX + 1)) =
        newCol ? *(newCol + y) : 0;
}
delete[] data;
data = newData;
++sizeX;
}

void deleteRow(int rowPos)
{
    --sizeY;
    int* newData{ new int[sizeY * sizeX] };

    for (int y{ 0 }; y < sizeY; ++y)
    {
        for (int x{ 0 }; x < sizeX; ++x)
        {
            *(newData + index2D(y, x)) =
                *(data + index2D(y + (y >= rowPos),
                    x));
        }
    }
    delete[] data;
    data = newData;
}

void addRow(int rowPos, int* newRow = nullptr)
{
    int* newData{ new int[(sizeY + 1) * sizeX] };

    for (int y{ 0 }; y < sizeY; ++y)
    {
        for (int x{ 0 }; x < sizeX; ++x)
        {
            *(newData + index2D(y + (y >= rowPos), x)) =
                *(data + index2D(y, x));
        }
    }
}

```

```

    }

    for (int x{ 0 }; x < sizeX; ++x)
    {
        *(newData + index2D(rowPos, x)) =
            newRow ? *(newRow + x) : 0;
    }
    delete[] data;
    data = newData;
    ++sizeY;
}

void print()const
{
    for (int y{ 0 }; y < sizeY; ++y)
    {
        for (int x{ 0 }; x < sizeX; ++x)
        {
            std::cout << (*this)(y, x) << '\t';
        }
        std::cout << '\n';
    }
    std::cout << '\n';
}

~Matrix() { delete[] data; }
};

int main()
{
    /* Установите значение USER_INPUT в 1 чтоб разрешить
       пользовательский ввод размеров матрицы */
#define USER_INPUT 0;
    int rows{ 3 };
    int columns{ 3 };
    int counter{ 1 };

```



```

#if USER_INPUT == 1
    std::cout << "Enter matrix rows count\n";
    std::cin >> rows;
    std::cout << "Enter matrix columns count\n";
    std::cin >> columns;
#endif

    Matrix matrix{ rows, columns };
    for (int y{ 0 }; y < rows; ++y)
    {
        for (int x{ 0 }; x < columns; ++x)
        {
            matrix(y, x) = counter++;
        }
    }

    matrix.print();

    matrix.deleteColumn(2);
    matrix.print();

    int* newColumn{ new int[columns] {11,22,33} };
    matrix.addColumn(0, newColumn);
    matrix.print();

    matrix.deleteRow(2);
    matrix.print();

    int* newRow{ new int[rows] {111,222,333} };
    matrix.addRow(2, newRow);
    matrix.print();

    delete[] newRow;
    delete[] newColumn;

    return 0;
}

```

Вывод результата работы из примера 21.

```

Microsoft Visual Studio Debug Console
1      2      3
4      5      6
7      8      9

1      2
3      4
5      6

11     1      2
22     3      4
33     5      6

11     1      2
22     3      4

11     1      2
22     3      4
111    222    333

```

Рисунок 20

Работа примера 21 требует некоторых пояснений. Ключевой моделью данных в классе является обычный линейно выделенный блок памяти, своего рода одномерный массив. Для получения линейного индекса по паре индексов, как в двумерном массиве — в классе `Matrix` присутствует две перегрузки функции-члена `index2D`. Первая для вычислений индекса использует размер строки из переменной-члена `sizeX`, вторая позволяет указать коэффициент поправки размера строки, данная перегрузка используется в функциях-членах добавления/удаления столбцов.

Вышеописанные функции-члены используют в своей работе перегрузки оператора вызова функции для доступа к элементам, хранящимся в линейно выделенном блоке памяти по двум индексам: строки и столбца. Необходимость двух перегрузок вызвана необходимостью

реализации семантики константности, как в примере из предыдущего раздела о перегрузке оператора доступа к элементу по индексу, `operator[]`. Перегрузка `operator()` позволяет получать доступ к элементу матрицы двумерного массива, указывая в круглых скобках после имени экземпляра класса индекс строки и столбца.

```
Matrix matrix{ 3,3 };
matrix(1, 2) = 42;
```

Перегрузка оператора вызова функции для класса `Matrix` семантически оправдана, хотя, безусловно, требует пояснений перед ее использованием. Данный способ доступа к элементам используется как внутри самого класса `Matrix` в функции-члене `print` так и в функции `main` при заполнении массива тестовыми значениями.

Для тестирования функциональности класса `Matrix` в примере 21 создается экземпляр класса, затем он заполняется тестовыми значениями. Далее производится удаление/добавление столбца, а затем удаление/добавление строки. Вывод 21 свидетельствует о корректности функционирования реализованных в классе возможностей.

Вопрос сравнения методик создания двумерного динамического массива в плане эффективности в зависимости от различных сценариев последующего использования двумерного динамического массива, хотя и выходит за рамки темы урока, также крайне интересен. Рекомендуются рассмотреть его отдельно.

Использование перегрузки оператора вызова функции для доступа к элементам в классах-контейнерах — один

из примеров использования `operator()`. Однако не меньший и даже многим больший интерес и семантический смысл представляют собой классы функторы.

Класс функтор представляет из себя класс, у которого реализована перегрузка `operator()`. Экземпляры такого класса используются как обычные функции, но с рядом уникальных особенностей и возможностей. Рассмотрим для начала простой пример класса функтора, реализующего счетчик. Каждый вызов «функции» возвращает следующее по порядку целое число. Конструктор класса может принимать в качестве формального параметра стартовое значение счетчика, конструктор по умолчанию создает счетчик стартующий с нуля. Функция-член `resetTo` позволит изменить позицию отсчета на указанное формальным параметром значение.

Пример 22

```
#include <iostream>

class Counter
{
    int cnt;

public:
    Counter(int start) : cnt{ start } {};
    Counter() : Counter(0) {};

    int operator()() { return cnt++; }
    void resetTo(int start) { cnt = start; }
};

int main()
```

```

{
    const int maxCnt{ 5 };
    Counter cnt1{};

    for (int i{ 0 }; i < maxCnt; ++i)
    {
        std::cout << cnt1() << ' ';
    }
    std::cout << '\n';

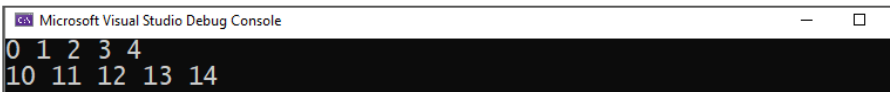
    cnt1.resetTo(10);

    for (int i{ 0 }; i < maxCnt; ++i)
    {
        std::cout << cnt1() << ' ';
    }
    std::cout << '\n';

    return 0;
}

```

Вывод результата работы из примера 22.



```

Microsoft Visual Studio Debug Console
0 1 2 3 4
10 11 12 13 14

```

Рисунок 21

Работа примера не вызывает сложности в понимании. В классе `Counter` присутствует переменная-член `cnt` значение которой и возвращает, каждый раз инкрементируя, перегрузка оператора вызова функции. Создав экземпляр класса, далее мы просто вызываем «функцию» `cnt1()` для получения очередного значения счетчика. Ничего сложного и полезного? Ведь мы можем просто создать

функцию со статической переменной, которая будет иметь тот же эффект. Или не совсем тот же?

Пример 23

```
#include <iostream>

int cnt()
{
    static int counter{ 0 };
    return counter++;
}

int main()
{
    const int maxCnt{ 5 };

    for (int i{ 0 }; i < maxCnt; ++i)
    {
        std::cout << cnt() << ' ';
    }
    std::cout << '\n';

    return 0;
}
```

Вывод результата работы из примера 23.



Рисунок 22

Функция со статической переменной способна лишь отчасти реализовать возможности класса функтора. Ведь функция одна, статическая переменная в ней тоже одна, удобного способа извне функции задать значение

статической переменной нет. Используя функцию со статической переменной, невозможно иметь несколько разных счетчиков, статическая переменная-то одна! Всех этих недостатков лишен функтор. Следует рассматривать функтор как функцию, которая может иметь состояние. Наличие у функции состояния позволяет при одном и том же вызове функции, с одними и теми же фактическими параметрами или при их отсутствии получать разный результат, основываясь в том числе и на состоянии, а не только на фактических параметрах. Также у функторов присутствует удобный и гибкий способ получения/модификации хранимого состояния при необходимости.

Пример 24

```
#include <iostream>

class Counter
{
    int cnt;
public:
    Counter(int start) : cnt{ start } {};
    Counter() : Counter(0) {};

    int operator()() { return cnt++; }
    void resetTo(int start) { cnt = start; }
};

int main()
{
    const int maxCnt{ 5 };
    Counter cnt1{};
    Counter cnt2{100};
```

```

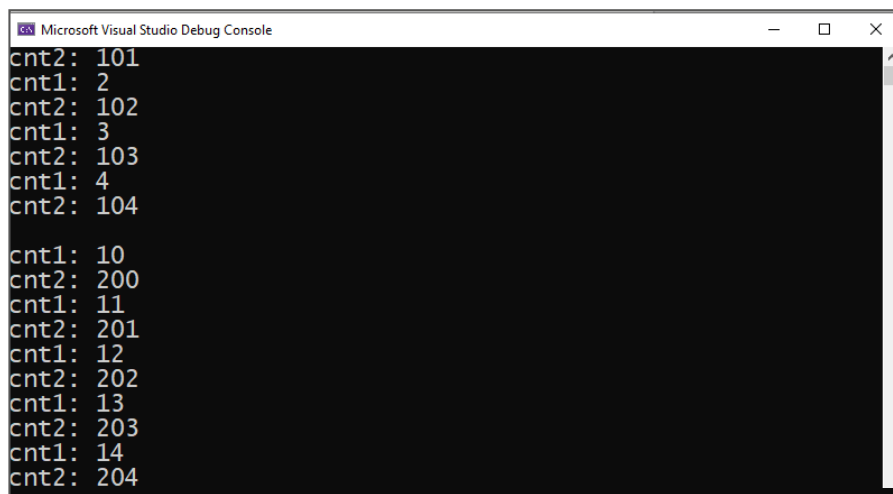
for (int i{ 0 }; i < maxCnt; ++i)
{
    std::cout << "cnt1: " << cnt1() << '\n';
    std::cout << "cnt2: " << cnt2() << '\n';
}
std::cout << '\n';
cnt1.resetTo(10);
cnt2.resetTo(200);

for (int i{ 0 }; i < maxCnt; ++i)
{
    std::cout << "cnt1: " << cnt1() << '\n';
    std::cout << "cnt2: " << cnt2() << '\n';
}
std::cout << '\n';

return 0;
}

```

Вывод результата работы из примера 24.



```

Microsoft Visual Studio Debug Console
cnt2: 101
cnt1: 2
cnt2: 102
cnt1: 3
cnt2: 103
cnt1: 4
cnt2: 104

cnt1: 10
cnt2: 200
cnt1: 11
cnt2: 201
cnt1: 12
cnt2: 202
cnt1: 13
cnt2: 203
cnt1: 14
cnt2: 204

```

Рисунок 23

В примере 24 создано два независимых счетчика, каждый со своим начальным отсчетом. Так же произведен «перезаряд» счетчиков на новые начала отсчета. При этом получение очередного значения счетчика — это просто «вызов функции». Ничего подобного не достичь используя функцию из примера 23!

Рассмотрим еще один пример. Для этого вооружившись знаниями о шаблонах функций, указателях и указателях на функции реализуем ряд шаблонов функций. Первым будет шаблон функции вывода одномерного массива на экран. Шаблон `print` принимает указатель на начало массива и указатель на следующий за последним элементом массива, а так же опционально — разделитель, который будет выводиться между элементами массива. Шаблон `copy_if` немного сложнее: он принимает пару указателей на начало/конец (конец также в виде следующего за последним элементом массива) массива источника и массива назначения, а также функцию-предикат, которая будет вызываться для каждого элемента массива источника, и лишь в случае возврата `true` из этой функции элемент будет копироваться в массив назначения. Также шаблон `copy_if` подсчитывает количество скопированных элементов, которые прошли фильтрацию функцией-предикатом.

Практический смысл и полезность шаблона `copy_if` заключается в обобщенном решении задачи копирования элементов массива источника в массив назначения согласно какому-то условию. Вместо реализации для каждого условия своего варианта копирования, само условие становится параметром обобщенного, независящего

от условия, алгоритма копирования. В качестве условий копирования будут выступать обычные функции, принимающие в качестве формального параметра целое число и возвращающие `true/false`. Функция `even` — для копирования только четных значений, `odd` — не четных, `greater3` — значений больше 3 и, наконец, `all` — для копирования всех значений.

Пример 25

```
#include <iostream>

template <typename T>
void print(T* begin, T* end, char delimiter = ' ')
{
    while (begin != end)
    {
        std::cout << *begin++ << delimiter;
    }
    std::cout << '\n';
}

template <typename T, typename Predicate>
int copy_if(T* srcB, T* srcE, T* destB, T* destE,
            Predicate pred)
{
    int copyCount{ 0 };
    while (destB != destE and srcB != srcE)
    {
        if (pred(*srcB))
        {
            *destB++ = *srcB;
            ++copyCount;
        }
        ++srcB;
    }
}
```

```

    return copyCount;
}

bool odd(const int e1)
{
    return e1 % 2 != 0;
}

bool even(const int e1)
{
    return e1 % 2 == 0;
}

bool greater3(const int e1)
{
    return e1 > 3;
}

bool all(const int e1)
{
    return true;
}

int main()
{
    const int size{ 10 };
    int arr1[size]{ 1,2,3,4,5,6,7,8,9,10 };
    int arr2[size]{};

    /*
     * указатель на первый элемент, начало массива
     * arr1 — arr1Begin указатель на следующий за
     * последним элементом, конец массива arr1 — arr1End
     */
    int* const arr1Begin{ arr1 };
    int* const arr1End{ arr1 + size };

```

```

/*
 * указатель на первый элемент, начало массива arr2 –
 * arr2Begin
 * указатель на следующий за последним элементом, конец
 * массива arr2 – arr2End
 */
int* const arr2Begin{ arr2 };
int* const arr2End{ arr2 + size };

/*
 * указатель на следующий за последним скопированным
 * в массив arr2 элемент – arr2NewEnd
 */
int* arr2NewEnd{};

std::cout << "Original arr1:\n";
print(arr1, arr1 + size);
std::cout << "Original arr2:\n";
print(arr2, arr2 + size);
std::cout << '\n';

std::cout << "arr2 copy of arr1 even elements
              only:\n";
arr2NewEnd = arr2Begin + copy_if(arr1Begin, arr1End,
                                arr2Begin, arr2End, even);
print(arr2, arr2NewEnd);
std::cout << '\n';

std::cout << "arr2 copy of arr1 odd elements only:\n";
arr2NewEnd = arr2Begin + copy_if(arr1Begin, arr1End,
                                arr2Begin, arr2End, odd);
print(arr2, arr2NewEnd);
std::cout << '\n';

std::cout << "arr2 copy of arr1 elements greater 3
              only:\n";

```

```

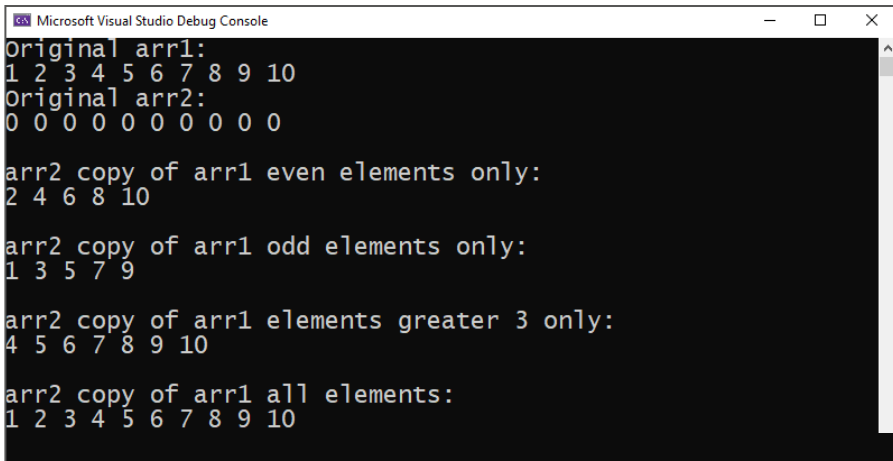
arr2NewEnd = arr2Begin + copy_if(arr1Begin, arr1End,
                                arr2Begin, arr2End, greater3);
print(arr2, arr2NewEnd);
std::cout << '\n';

std::cout << "arr2 copy of arr1 all elements:\n";
arr2NewEnd = arr2Begin + copy_if(arr1Begin, arr1End,
                                arr2Begin, arr2End, all);
print(arr2, arr2NewEnd);
std::cout << '\n';

return 0;
}

```

Вывод результата работы из примера 25.



```

Microsoft Visual Studio Debug Console
Original arr1:
1 2 3 4 5 6 7 8 9 10
Original arr2:
0 0 0 0 0 0 0 0 0 0
arr2 copy of arr1 even elements only:
2 4 6 8 10
arr2 copy of arr1 odd elements only:
1 3 5 7 9
arr2 copy of arr1 elements greater 3 only:
4 5 6 7 8 9 10
arr2 copy of arr1 all elements:
1 2 3 4 5 6 7 8 9 10

```

Рисунок 24

В примере 25 создается пара тестовых массивов `arr1`, заполненных значениями от 1 до 10, и массив `arr2`, в который будет производиться копирование элементов по некоторому условию. Далее, используя описанные

шаблоны функций и функции, проводится тестирование функционала условного копирования с задачей условия в виде функции-предиката. Вывод 25 свидетельствует о корректности работы программы.

Обратите внимание, в шаблоне `copy_if` присутствует два параметра-типа шаблона: первый — параметр для обобщения типа элементов массива, второй — параметр для обобщения типа функции предиката. Мы не конкретизировали тип функции предиката в виде указателя на функцию, возвращающую `bool` и принимающую аргумент типа `T`, первый параметр типа в шаблоне, в качестве параметра. Причина данного решения станет ясна далее.

Использование функций-предикатов и обобщенного алгоритма копирования — элегантное решение, однако по силе ли обобщенному копированию более сложные задачи? Например, скопировать в массив назначения элементы исходного массива. Без идущих подряд одинаковых элементов копироваться должен только первый элемент, но не идущие подряд одинаковые с ним элементы-дубли. Или еще задача: копировать элементы в массив назначения, но так, чтоб их сумма не превысила некоторый порог. Казалось бы, написать соответствующие функции-предикаты — и задачи решены! Однако написать обычную функцию, которая бы смогла решить данную задачу непросто, ведь теперь функция-предикат не может определить необходимость копирования лишь по значению очередного элемента, этого мало, требуется ретроспектива, требуется «знание» о предыдущих элементах, требуется состояние. Конечно же, применив

определенную шноровку, возможно написать функцию, использующую статические переменные для решения данной задачи, но это совершенно громоздкое, негибкое, да и просто неэлегантное решение. А так как бытует мнение о том, что то, что выглядит красиво и работает не менее хорошо, то мы воспользуемся именно таким красивым решением.

Элегантно решить выше поставленную задачу возможно, используя функтор — функциональный объект, обладающий состоянием. В данном случае наличие состояния нам как нельзя кстати.

Для решения первой задачи реализуем функтор `NoSequence`:

```
class NoSequence
{
    bool init;
    int prevEl;

public:
    NoSequence() : init{ false }, prevEl{ 0 } {}

    bool operator()(int el)
    {
        if (init)
        {
            bool result{ prevEl != el };
            if (result)
            {
                prevEl = el;
            }
            return result;
        }
        init = true;
    }
}
```

```

        prevEl = el;
        return true;
    }
};

```

При вызове перегруженного оператора «круглые скобки», `operator()`, с очередным элементом массива в качестве фактического параметра происходит следующее: сперва необходимо выяснить, впервые ли вызван данный функтор, если да, то каков бы ни был элемент массива, его необходимо безусловно копировать в массив назначения, так как даже если весь массив-источник состоит из совершенно одинаковых, идущих последовательно, элементов, то первый — уникальный. Чтоб отличить повторные запуски функтора от первого, переменная-член `init` присваивается в `true`. Также необходимо сохранить значение скопированного элемента в переменную-член `prevEl`, предыдущий элемент, для последующего анализа. В случае если функтор запущен повторно, `init == true`, для принятия решения о копировании необходимо сравнить очередной элемент массива, фактический параметр с которым был запущен функтор, с предыдущим скопированным элементом `prevEl` и только в случае их неравенства копировать очередной элемент в массив назначения, обновив значение переменной-члена `prevEl`. Если же очередной элемент массива-источника тот же, что и ранее сохраненный в переменной-члене `prevEl`, копирование не произойдет. Разумеется, решение о копировании или не копировании функтор выражает в виде возвращаемого перегрузкой оператора вызова

функции, `operator()`, значения `true` — копировать, и `false` в противном случае.

Пример 26

```
#include <iostream>

template <typename T>
void print(T* begin, T* end, char delimiter = ' ')
{
    while (begin != end)
    {
        std::cout << *begin++ << delimiter;
    }
    std::cout << '\n';
}

template <typename T, typename Predicate>
int copy_if(T* srcB, T* srcE, T* destB, T* destE,
            Predicate pred)
{
    int copyCount{ 0 };
    while (destB != destE and srcB != srcE)
    {
        if (pred(*srcB))
        {
            *destB++ = *srcB;
            ++copyCount;
        }
        ++srcB;
    }
    return copyCount;
}

class NoSequence
{
    bool init;
```

```

    int prevEl;

public:
    NoSequence() : init{ false }, prevEl{ 0 } {}

    bool operator()(int el)
    {
        if (init)
        {
            bool result{ prevEl != el };
            if (result)
            {
                prevEl = el;
            }
            return result;
        }
        init = true;
        prevEl = el;
        return true;
    }
};

int main()
{
    const int size{ 10 };
    int arr1[size]{ 1,1,1,4,5,5,7,8,9,9 };
    int arr2[size]{};

    int* const arr1Begin{ arr1 };
    int* const arr1End{ arr1 + size };

    int* const arr2Begin{ arr2 };
    int* const arr2End{ arr2 + size };

    int* arr2NewEnd{};

    std::cout << "Original arr1:\n";

```

```

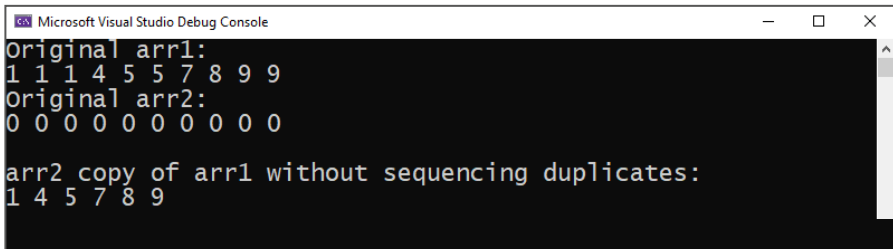
print(arr1, arr1 + size);
std::cout << "Original arr2:\n";
print(arr2, arr2 + size);
std::cout << '\n';

std::cout << "arr2 copy of arr1 without sequencing "
            "duplicates:\n";
arr2NewEnd = arr2Begin + copy_if(arr1Begin, arr1End,
                                arr2Begin, arr2End, NoSequence{});
print(arr2, arr2NewEnd);
std::cout << '\n';

return 0;
}

```

Вывод результата работы из примера 26.



```

Microsoft Visual Studio Debug Console
Original arr1:
1 1 1 4 5 5 7 8 9 9
Original arr2:
0 0 0 0 0 0 0 0 0 0
arr2 copy of arr1 without sequencing duplicates:
1 4 5 7 8 9

```

Рисунок 25

Вывод результата из примера 26 подтверждает корректность работы ранее реализованного функтора. В массив назначения скопировались лишь не идущие подряд элементы. Особый интерес представляет способ, которым функтор был передан в экземпляр шаблона `copy_if`

```

copy_if(arr1Begin, arr1End, arr2Begin, arr2End,
        NoSequence{});

```

Экземпляр функтора был анонимно создан и инициализирован непосредственно при вызове экземпляра шаблона в виде конструкции `NoSequence{}`.

В решении второй задачи поможет функтор `SumLimit`.

```
class SumLimit
{
    int sumLimit;
    int sum;

public:
    SumLimit(int sumLimitP, int startSumP) :
        sumLimit{ sumLimitP }, sum{ startSumP } {}
    SumLimit(int sumLimitP) : SumLimit{ sumLimitP, 0 } {}

    bool operator()(int el)
    {
        if (sum + el < sumLimit)
        {
            sum += el;
            return true;
        }

        return false;
    }
};
```

В функторе реализовано два конструктора: один устанавливает лимит суммы элементов исходного массива, после которого прекращается копирование, а второй дополнительно позволяет установить начальную сумму элементов, если стартовать надо не с нуля. Перегруженный оператор вызова функции достаточно прост: он лишь накапливает сумму элементов массива, которые получает как фактический параметр, и проверяет, чтоб сумма не

превысила лимит. В случае превышения лимита функтор начинает возвращать `false`, что равносильно запрету копирования.

Пример 27

```
#include <iostream>

template <typename T>
void print(T* begin, T* end, char delimiter = ' ')
{
    while (begin != end)
    {
        std::cout << *begin++ << delimiter;
    }
    std::cout << '\n';
}

template <typename T, typename Predicate>
int copy_if(T* srcB, T* srcE, T* destB, T* destE,
            Predicate pred)
{
    int copyCount{ 0 };
    while (destB != destE and srcB != srcE)
    {
        if (pred(*srcB))
        {
            *destB++ = *srcB;
            ++copyCount;
        }
        ++srcB;
    }
    return copyCount;
}

class SumLimit
{
```

```

    int sumLimit;
    int sum;

public:
    SumLimit(int sumLimitP, int startSumP) :
        sumLimit{ sumLimitP }, sum{ startSumP } {}
    SumLimit(int sumLimitP) : SumLimit{ sumLimitP, 0 } {}

    bool operator()(int el)
    {
        if (sum + el < sumLimit)
        {
            sum += el;
            return true;
        }
        return false;
    }
};

int main()
{
    const int maxSum{ 16 };
    const int size{ 10 };
    int arr1[size]{ 1,2,3,4,5,6,7,8,9,10 };
    int arr2[size]{};

    int* const arr1Begin{ arr1 };
    int* const arr1End{ arr1 + size };

    int* const arr2Begin{ arr2 };
    int* const arr2End{ arr2 + size };

    int* arr2NewEnd{};

    std::cout << "Original arr1:\n";
    print(arr1, arr1 + size);

```

```

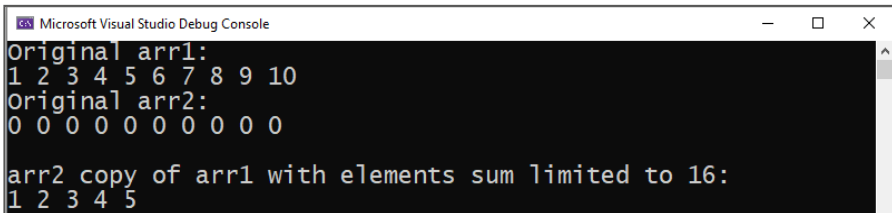
std::cout << "Original arr2:\n";
print(arr2, arr2 + size);
std::cout << '\n';

std::cout << "arr2 copy of arr1 with elements sum "
            "limited to "
            << maxSum << ":\n";
arr2NewEnd = arr2Begin + copy_if(arr1Begin, arr1End,
                                arr2Begin, arr2End, SumLimit{ maxSum });
print(arr2, arr2NewEnd);
std::cout << '\n';

return 0;
}

```

Вывод результата работы из примера 27.



```

Microsoft Visual Studio Debug Console
Original arr1:
1 2 3 4 5 6 7 8 9 10
Original arr2:
0 0 0 0 0 0 0 0 0 0
arr2 copy of arr1 with elements sum limited to 16:
1 2 3 4 5

```

Рисунок 26

Примеры 26 и 27 иллюстрируют практическое применение нетривиальных функторов для элегантного решения задач. Продемонстрированный подход также интересен тем, что вместо монолитного решения различных задач копирования в виде специфичной конкретной задачи функции, решение разбивается на ряд более простых подзадач, каждую из которых можно решать, тестировать и отлаживать независимо от других, что, несомненно удобно и эффективно. Также данный пример

активно демонстрирует возможность переиспользования ранее написанного кода для решения разных, внешне непохожих задач.

Методы и способы, продемонстрированные в данном разделе еще встретятся нам далее по курсу, когда будет изучаться стандартная библиотека шаблонов C++ STL (Standard Template Library). STL — мощнейший инструмент современного C++, позволяющий писать чистый и эффективный код. Но всему свое время, пока мы лишь немного приоткрыли завесу над магией STL.

6. Специальные перегрузки — перегрузка оператора преобразования типов

Одной из задач, которую в C++ решают классы в связке с перегрузкой операторов, является возможность программистам создавать новые сложные типы данных с семантическим поведением, аналогичные базовым типам языка. Как известно, все базовые типы имеют возможность взаимного преобразования друг в друга. Однако вопрос преобразования классов, типов, создаваемых программистами, пока детально не рассматривался. Настало время сделать это.

Существует возможность определить оператор преобразования для класса в произвольный базовый тип либо же в произвольный иной класс.

С учетом внутренней структуры и назначения класса не всякое преобразование имеет смысл, да и вообще возможно. Целесообразность и возможность определяется разработчиком класса. Перегрузка оператора преобразования типа допускается исключительно функцией-членом класса.

Рассмотрим общий вид оператора преобразования типа, его сигнатуру:

```
operator typename() {} // 1
operator typename() const {} // 2
explicit operator typename() const {} // 3
```

Вариант 1 — неконстантная функция-член, реализующая преобразование класса к типу `typename`. Так как маловероятно, а в подавляющем большинстве случаев бессмысленно модифицировать состояние класса в процессе преобразования типа, то практический интерес Вариант 1 не представляет. Вариант 2 — константная функция-член, реализующая преобразование класса к типу `typename`. Вариант 3 — константный вариант функции-члена, реализующий явное преобразование класса к типу `typename` и запрещающий использование данного оператора в неявных преобразованиях.

Особое внимание следует обратить на отсутствие типа возвращаемого значения для оператора преобразования типа. Очевидно и естественно возвращать результат того типа, к которому происходит преобразование. Также следует отметить запрет использования формальных параметров в операторе преобразования типа.

Проиллюстрируем перегрузку оператора преобразования типа на примере класса `Point`.

Пример 28

```
#include <iostream>

class Point
{
    int x;
    int y;

public:
    Point() = default;
    Point(int pX, int pY) : x{ pX }, y{ pY } {}
```

```

Point& setX(int pX) { x = pX; return *this; }
Point& setY(int pY) { y = pY; return *this; }

void showPoint() const
{
    std::cout << '(' << x << ', ' << y << ')';
}

operator bool() const { return x and y; }
};

int main()
{
    const int pointsCount{ 3 };
    Point points[pointsCount]{ {0,0}, {28,29}, {0,26} };

    bool isZero{false};

    for (auto point{ points }, pointsEnd{ points +
        pointsCount }; point != pointsEnd; ++point)
    {
        isZero = *point;

        if (isZero)
        {
            std::cout << "Zero Point detected!\n";
        }
        else
        {
            point->showPoint();
            std::cout << '\n';
        }
    }

    return 0;
}

```

Вывод результата работы из примера 28.

```
Microsoft Visual Studio Debug Console
(0,0)
Zero Point detected!
(0,26)
```

Рисунок 27

Поясним работу примера 28. Для всех базовых типов C++ существует преобразование к типу `bool` по принципу «нулевое значение» исходного типа соответствует `false`, все остальные значения исходного типа — `true`. Реализуем аналогичную логику преобразования типа `Point` к типу `bool` — в случае, если обе координаты имеют значение ноль, результат преобразования — `false`, в остальных случаях — `true`. Логика преобразования реализована функцией-членом

```
operator bool() const { return x and y; }
```

простым использованием операции «логического И» для координат точки.

В примере 28 создается массив из 3 точек, у одной из которых обе координаты равны нулю. Далее в цикле `for` происходит перебор элементов массива, а при помощи `bool` переменной `isZero` и перегрузки оператора преобразования для типа `bool` происходит проверка на то, что точка «нулевая». Важно заметить, что присваивание переменной `isZero` происходит, используя неявное преобразование типа `Point` в `bool`.

В ряде случаев, желательным является запрет на использование перегрузки оператора преобразования

типа в неявных преобразованиях. Для этого, аналогично конструкторам, используется ключевое слово **explicit**. В случае добавления к оператору преобразования типа в пример 28 ключевого слова **explicit**

```
explicit operator bool() const { return x and y; }
```

пример перестанет компилироваться с ошибками

```
E0413 no suitable conversion function from "Point"
      to "bool" exists
C2440 '=': cannot convert from 'Point' to 'bool'
```

Чтоб исправить данные ошибки, необходимо выполнить явное преобразование типа в виде:

```
isZero = (bool)*point;
```

Следует помнить о выполнении контекстного преобразования типа в **bool** в ряде случаев, а именно:

- в управляющем (условном) выражении условного оператора **if**, а также циклов **while**, **do-while**, **for**.
- у аргументов встроенных операторов **||** логическое ИЛИ, **&&** логическое И, **!** логическое ОТРИЦАНИЕ.
- в первом выражении тернарного оператора **?**.

В контексте вышеперечисленных случаев происходит явное преобразование типов к **bool**, ведь именно тип **bool** требуется контекстом соответствующих операторов или конструкций языка. Иными словами, даже в случае объявления перегрузки оператора преобразования типа к **bool**, как **explicit**, данный оператор будет использоваться для выполнения контекстного преобразования типа **bool**.

Пример 29

```

#include <iostream>

class Point
{
    int x;
    int y;

public:
    Point() = default;
    Point(int pX, int pY) : x{ pX }, y{ pY } {}
    Point& setX(int pX) { x = pX; return *this; }
    Point& setY(int pY) { y = pY; return *this; }

    void showPoint() const
    {
        std::cout << '(' << x << ',' << y << ')';
    }

    explicit operator bool() const { return x and y; }
};

int main()
{
    const int pointsCount{ 3 };
    Point points[pointsCount]{ {0,0}, {28,29}, {0,26} };

    bool isZero{ false };

    for (auto point{ points },
         pointsEnd{ points + pointsCount };
         point != pointsEnd; ++point)
    {
        isZero = (bool)*point; /* использовано для
                                наглядной демонстрации
                                необходимости явного
                                преобразования типа */
    }
}

```

```

    if (*point)
    {
        std::cout << "Zero Point detected!\n";
    }
    else
    {
        point->showPoint();
        std::cout << '\n';
    }
}

return 0;
}

```

Вывод результата работы из примера 29.

```

Microsoft Visual Studio Debug Console
(0,0)
Zero Point detected!
(0,26)

```

Рисунок 28

В примере 29 для выражения в условном операторе `if` происходит контекстное преобразование типа к `bool` без необходимости выполнять явное преобразование типа.

Перегрузка оператора преобразования типа может использоваться так же с отличными от базовых типов типами. Рассмотрим это на примере взаимного преобразования типа `Point` в `Point3D` (класс полностью аналогичный обычной точке, только для трех измерений вместо двух).

Пример 30

```

#include <iostream>

class Point3D;

class Point
{
    int x;
    int y;

public:
    Point() = default;
    Point(int pX, int pY) : x{ pX }, y{ pY } {}
    Point& setX(int pX) { x = pX; return *this; }
    Point& setY(int pY) { y = pY; return *this; }

    void showPoint() const
    {
        std::cout << '(' << x << ',' << y << ')';
    }

    explicit operator bool() const { return x and y; }
    explicit operator Point3D() const;
};

class Point3D
{
    int x;
    int y;
    int z;

public:
    Point3D() = default;
    Point3D(int pX, int pY, int pZ) : x{ pX }, y{ pY },
                                     z{ pZ } {}
    Point3D& setX(int pX) { x = pX; return *this; }

```



```

Point3D& setY(int pY) { y = pY; return *this; }
Point3D& setZ(int pZ) { z = pZ; return *this; }

void showPoint() const
{
    std::cout << '(' << x << ', ' << y << ', ' << z << ')';
}

explicit operator bool() const { return x and y and z; }
explicit operator Point() const;
};

Point::operator Point3D() const { return { x,y,0 }; }
Point3D::operator Point() const { return { x, y }; }

int main()
{
    Point point{ 26,7 };
    Point pointConv{};

    Point3D point3D{ 24, 7, 76 };
    Point3D point3DConv{};

    pointConv = (Point)point3D;
    point3DConv = (Point3D)point;

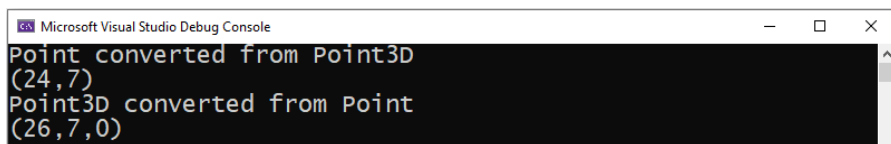
    std::cout << "Point converted from Point3D\n";
    pointConv.showPoint();

    std::cout << "\nPoint3D converted from Point\n";
    point3DConv.showPoint();
    std::cout << '\n';

    return 0;
}

```

Вывод результата работы из примера 30.



```
Microsoft Visual Studio Debug Console
Point converted from Point3D
(24,7)
Point3D converted from Point
(26,7,0)
```

Рисунок 29

В примере 30 используются два класса: знакомый нам `Point` и его трехмерный аналог, получившийся из `Point` добавлением `Z`-координаты. Здесь специально не используется наследование, дабы не усложнять пример. В обоих классах предусмотрена возможность взаимного преобразования. Так как класс `Point3D` еще не объявлен на момент объявления класса `Point`, то для описания оператора преобразования типа к `Point3D` внутри класса `Point`, используется предварительная декларация класса `Point3D`. Для корректного определения операторов преобразования типов в `Point` и `Point3D`, компилятору необходимо иметь полное описание данных классов в контексте занимаемой ими памяти, посему внутри классов помещены лишь декларации операторов преобразования типов, а, собственно, их определение вынесено за соответствующие классы. Вывод работы примера 30 подтверждает корректность написания кода.

Перегрузка операторов преобразования типов позволяет реализовывать семантически полные и удобные в работе классы.

7. Список операторов, которые невозможно перегрузить

В языке C++ существует ряд операторов, которые невозможно перегрузить каким бы то ни было способом. Список таких операторов:

- `::` оператор разрешения области,
- `.*` оператор-указатель на член структуры/класса,
- `.` оператор доступа к члену структуры/класса,
- `?:` тернарный оператор.

Невозможность перегружать данные операторы связана с тем, что они выполняют глубоко специфичные и, что более важно, независящие от типов операндов действия. Тот же тернарный оператор просто вычисляет либо одно, либо второе выражение на основе приведенного к типу `bool` выражения до знака вопроса. Как таковые типы операндов тернарного оператора в процессе его работы активной роли не играют. Аналогично обстоит дело с остальными операторами. Даже если предположить возможность их перегрузки, то сложно представить практический и семантически целостный смысл, который можно было бы возложить на их перегрузки. Ведь одной из основных, хотя и не единственной, задачей перегрузки операторов в C++ является наделение новых типов данных семантически целостным и предсказуемым поведением, способом обработки, схожим с базовыми типами C++. А перегрузка «неперегружаемых» операторов, вероятнее всего, нарушила бы эту семантическую целостность и ввела бы совершенно неожиданное поведение, что неприемлемо, а посему недоступно.

8. Статический полиморфизм и перегрузка операторов как частный случай

Перегрузка операторов реализует возможность семантически схожей либо даже одинаковой обработки различных типов данных в C++. При этом практически нет ограничений и разницы для работы с какими типами данных реализована перегрузка, для базовых типов C++ либо же для разрабатываемых классов. Единообразие в работе с различными типами есть не что иное, как воплощение полиморфизма, его ad-hock разновидности (специальный полиморфизм или специализированный полиморфизм). Данная разновидность полиморфизма является статическим видом полиморфизма, так как вся работа по выбору конкретной реализации кода для соответствующего или соответствующих типов происходит на этапе компиляции, а не в процессе исполнения. С точки зрения лучшей производительности, возможности анализа программистом либо же автоматическими программными средствами, статический полиморфизм предпочтительнее динамического, который будет изучаться позднее в рамках курса.

9. Перегрузка операторов глобальными функциями

Для операторов, которые не модифицируют свои операнды оптимальным способом перегрузки являются глобальные или дружественные функции. Начнем рассмотрение данного варианта перегрузки с глобальных функций, а дружественным функциям будет посвящен следующий раздел урока.

Итак, рассмотрим перегрузку арифметических операторов `+`, `-`, `*`, `/` на примере известного нам класса `Fraction`. Данные операторы являются бинарными операторами, которые не модифицируют свои аргументы и возвращают результат в виде нового временного объекта. Сигнатура функции для такого рода операторов выглядит следующим образом:

```
ClassName operator+(const ClassName& left,  
                    const ClassName& right); //1  
ClassName operator+(const ClassName& left, int right); //2  
ClassName operator+(int left, const ClassName& right); //3
```

Вариант 1 — перегрузка оператора в случае, когда оба аргумента — классы `ClassName`.

Варианты 2 и 3 — перегрузка оператора для случаев, когда правый (2) или левый (3) — операнды некоего базового типа, например, `int`. Особое внимание следует обратить на то, что, несмотря на свою симметричность, мало реализовать лишь один из вариантов 2 или 3, для семантически полного поведения следует обязательно реализовывать оба варианта.

Пример 31

```

#include <iostream>

class Fraction
{
    int numerator;
    int denominator;
    int gcd(int a, int b);
    void reduce();

public:
    Fraction(int num, int denom)
        : numerator{ num }, denominator{ denom ? denom : 1 }
    {
        reduce();
    };

    Fraction() : Fraction(1, 1) {};
    void setNumerator(int num) { numerator = num; reduce(); };
    int getNumerator() const { return numerator; };
    void setDenominator(int denom) { denominator =
        denom ? denom : 1; reduce(); };
    int getDenominator() const { return denominator; }
    void print() const;
};

int Fraction::gcd(int a, int b)
{
    int copy;
    while (b)
    {
        a %= b;
        copy = a;
        a = b;
        b = copy;
    }
    return a;
}

```

```

}

void Fraction::reduce()
{
    int gcdVal{ gcd(numerator, denominator) };
    numerator /= gcdVal;
    denominator /= gcdVal;
    if (denominator < 0 and numerator < 0)
    {
        denominator *= -1;
        numerator *= -1;
    }
}

void Fraction::print() const
{
    std::cout << '(' << numerator << " / "
                << denominator << ")";
}

Fraction operator+(const Fraction& left,
                   const Fraction& right)
{
    return Fraction{ left.getNumerator() *
                     right.getDenominator() +
                     right.getNumerator() *
                     left.getDenominator(),
                     left.getDenominator() *
                     right.getDenominator() };
}

Fraction operator+(const Fraction& left, int right)
{
    return Fraction{ left.getNumerator() + right *
                     left.getDenominator(),
                     left.getDenominator() };
}

```

```

Fraction operator+(int left, const Fraction& right)
{
    return right + left;
}

Fraction operator-(const Fraction& left,
                  const Fraction& right)
{
    return Fraction{ left.getNumerator() *
                    right.getDenominator() -
                    right.getNumerator() *
                    left.getDenominator(), left.getDenominator() *
                    right.getDenominator() };
}

Fraction operator-(const Fraction& left, int right)
{
    return Fraction{ left.getNumerator() - right *
                    left.getDenominator(),
                    left.getDenominator() };
}

Fraction operator-(int left, const Fraction& right)
{
    return Fraction{ left * right.getDenominator() -
                    right.getNumerator(),
                    right.getDenominator() };
}

Fraction operator*(const Fraction& left,
                  const Fraction& right)
{
    return Fraction{ left.getNumerator() *
                    right.getNumerator(),
                    left.getDenominator() *
                    right.getDenominator() };
}

```



```

Fraction operator*(const Fraction& left, int right)
{
    return Fraction{ left.getNumerator() * right,
                     left.getDenominator() };
}

Fraction operator*(int left, const Fraction& right)
{
    return right * left;
}

Fraction operator/(const Fraction& left,
                  const Fraction& right)
{
    return Fraction{ left.getNumerator() *
                     right.getDenominator(),
                     left.getDenominator() *
                     right.getNumerator() };
}

Fraction operator/(const Fraction& left, int right)
{
    return Fraction{ left.getNumerator() ,
                     left.getDenominator() * right };
}

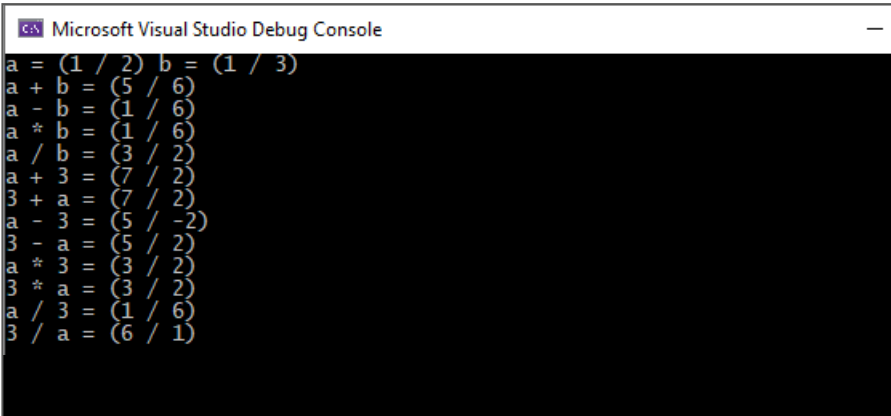
Fraction operator/(int left, const Fraction& right)
{
    return Fraction{ left* right.getDenominator(),
                     right.getNumerator() };
}

int main()
{
    Fraction a{1,2};
    Fraction b{1,3};

```

```
std::cout << "a = "; a.print(); std::cout  
    << " b = "; b.print(); std::cout << '\n';  
  
std::cout << "a + b = "; (a + b).print(); std::cout  
    << '\n';  
  
std::cout << "a - b = "; (a - b).print(); std::cout  
    << '\n';  
  
std::cout << "a * b = "; (a * b).print(); std::cout  
    << '\n';  
  
std::cout << "a / b = "; (a / b).print(); std::cout  
    << '\n';  
  
std::cout << "a + 3 = "; (a + 3).print(); std::cout  
    << '\n';  
  
std::cout << "3 + a = "; (3 + a).print(); std::cout  
    << '\n';  
  
std::cout << "a - 3 = "; (a - 3).print(); std::cout  
    << '\n';  
  
std::cout << "3 - a = "; (3 - a).print(); std::cout  
    << '\n';  
  
std::cout << "a * 3 = "; (a * 3).print(); std::cout  
    << '\n';  
  
std::cout << "3 * a = "; (3 * a).print(); std::cout  
    << '\n';  
  
std::cout << "a / 3 = "; (a / 3).print(); std::cout  
    << '\n';  
  
std::cout << "3 / a = "; (3 / a).print(); std::cout  
    << '\n';  
  
return 0;  
  
}
```

Вывод результата работы из примера 31.



```

Microsoft Visual Studio Debug Console
a = (1 / 2) b = (1 / 3)
a + b = (5 / 6)
a - b = (1 / 6)
a * b = (1 / 6)
a / b = (3 / 2)
a + 3 = (7 / 2)
3 + a = (7 / 2)
a - 3 = (5 / -2)
3 - a = (5 / 2)
a * 3 = (3 / 2)
3 * a = (3 / 2)
a / 3 = (1 / 6)
3 / a = (6 / 1)

```

Рисунок 30

Для реализации арифметических операций в новую версию класса `Fraction` добавлены две `privat`-функции-члены: `gcd` — поиск наибольшего общего делителя и `reduce` — сокращение дроби. Чтобы хранимая в классе дробь всегда была сокращенной, вызов функции `reduce` обеспечен в конструкторе и в соответствующих сеттерах. Собственно, арифметические операции реализуются перегрузкой соответствующих операторов глобальными функциями. Данные функции возвращают в качестве результата новый экземпляр класса `Fraction`. Также следует отметить, что в связи с коммутативностью сложения и умножения достаточно реализовать лишь одну перегрузку `Fraction / int`, а соответствующую вторую перегрузку реализовать через предыдущую, поменяв порядок следования аргументов. Вывод работы программы из примера 31 свидетельствует о корректности проделанной работы.

Все ли хорошо с примером 31? Формально — да, ведь задача решена, арифметические операции дробь/дробь и дробь/целое число реализованы и протестированы. Однако из-за повсеместного вызова соответствующих геттеров, эффективность операций не самая оптимальная. Вызов геттеров необходим в связи с закрытостью переменных-членов класса `Fraction`, из глобальных функций к ним иначе не «добраться». Как быть? Сделать переменные-члены класса `Fraction` открытыми, избавившись от необходимости использования геттеров в глобальных функциях? Ни в коем случае нет! Тем самым мы нарушим принцип инкапсуляции. Выход в другом — сделаем наши глобальные функции дружественными классу `Fraction`.

10. Перегрузка операторов дружественными функциями

Дружественные функции класса `ClassName` — это такие функции не-члены класса `ClassName`, которым предоставляется доступ к `private` и `protected` членам класса `ClassName`. Дружественными или `friend`-функциями могут быть как глобальные функции, так и функции-члены классов. Дружественные функции не становятся функциями-членами класса, друзьями которого они являются.

Для того, чтобы функция стала дружественной классу, необходимо внутри определения класса указать сигнатуру функции, предварив ее ключевым словом `friend`.

```
class ClassB
{
    void bFunction() { /* тело функции */ };
};

class ClassName
{
    // определение класса
    friend void nonMemberFunction(); // 1
    friend void ClassB::bFunction(); // 2
};

void nonMemberFunction() { /* тело функции */ }
```

Вариант 1 — указание глобальной функции, как друга класса `ClassName`, вариант 2 — указание функции-члена класса `ClassB`, как друга класса `ClassName`.

Может показаться, что дружественные функции сводят на нет принцип инкапсуляции, однако это не так, ведь только те функции, которые объявлены дружественными классу получают доступ к `private` и `protected` членам класса, «желания» функции мало, определяющим является «разрешение» и «согласие» на дружбу самого класса. Автор класса в полной мере контролирует кому предоставлять расширенный доступ, а кому — нет. Без ведома и согласия автора класса «подружиться» с классом невозможно. Концепция дружественных функций расширяет источник потенциальных проблем с неверным состоянием переменных-членов класса на строго определенный набор функций-друзей. Соответственно, при отладке работы класса необходимо обратить пристальное внимание на функции-члены класса и функции-друзья класса. Принцип инкапсуляции практически не страдает.

Рассмотрим простой пример применения дружественных функций на основе класса `Point` из предыдущих разделов урока. Реализуем ряд полезных функций по работе с точками, а именно: функцию вычисления расстояния между двумя точками и функцию определения к какому квадранту или оси координат принадлежит заданная точка. Также реализуем сервисную функцию, выводящую на экран строку, соответствующую константе квадранта или оси координат.

Пример 32

```
#include <iostream>

enum Quadrants { ZeroPoint, First, Second, Third, Fourth,
                 XPositive, XNegative, YPositive, YNegative};
```

```

class Point
{
    int x;
    int y;

public:
    Point() = default;
    Point(int pX, int pY) : x{ pX }, y{ pY } {}
    Point& setX(int pX) { x = pX; return *this; }
    Point& setY(int pY) { y = pY; return *this; }
    int getX()const { return x; }
    int getY()const { return y; }

    void show() const
    {
        std::cout << '(' << x << ',' << y << ')';
    }
    friend double distance(const Point& p1, const Point& p2);
    friend int quadrant(const Point& p)
    {
        if (!p.y and !p.x) { return Quadrants::ZeroPoint;}
        if (!p.y) { return p.x > 0 ? Quadrants::XPositive :
            Quadrants::XNegative;}
        if (!p.x) { return p.y > 0 ? Quadrants::YPositive :
            Quadrants::YNegative;}
        if (p.y > 0)
        {
            return p.x > 0 ? Quadrants::First :
                Quadrants::Second;
        }
        return p.x > 0 ? Quadrants::Fourth : Quadrants::Third;
    }
};

double distance(const Point& p1, const Point& p2)
{
    auto xLength{ p2.x - p1.x };

```

```

    auto yLength{ p2.y - p1.y };
    return sqrt(xLength * xLength + yLength * yLength);
}

void quadrantDecode(int quadrant)
{
    if (quadrant == Quadrants::ZeroPoint)
        { std::cout << "Zero point"; }
    else if (quadrant == Quadrants::First)
        { std::cout << "First quadrant"; }
    else if (quadrant == Quadrants::Second)
        { std::cout << "Second quadrant"; }
    else if (quadrant == Quadrants::Third)
        { std::cout << "Third quadrant"; }
    else if (quadrant == Quadrants::Fourth)
        { std::cout << "Fourth quadrant"; }
    else if (quadrant == Quadrants::XPositive)
        { std::cout << "X axis positive"; }
    else if (quadrant == Quadrants::XNegative)
        { std::cout << "X axis negative"; }
    else if (quadrant == Quadrants::YPositive)
        { std::cout << "Y axis positive"; }
    else if (quadrant == Quadrants::YNegative)
        { std::cout << "Y axis negative"; }
}

int main()
{
    Point p1{ 5,5 };
    Point p2{ 10,10 };

    std::cout << "Distance between p1 and p2 is: "
               << distance(p1, p2) << '\n';

    const int testCases{ 9 };
    Point points[testCases]{
        {0,0},

```



```

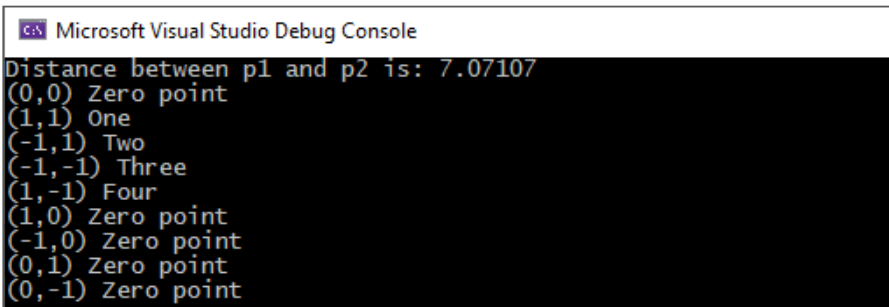
        {1,1},
        {-1,1},
        {-1,-1},
        {1,-1},
        {1,0},
        {-1,0},
        {0,1},
        {0,-1}
    };

    for (int i{ 0 }; i < testCases; ++i)
    {
        points[i].show();
        std::cout << ' ';
        quadrantDecode(quadrant(points[i]));
        std::cout << '\n';
    }

    return 0;
}

```

Вывод результата работы из примера 32.



```

Microsoft Visual Studio Debug Console
Distance between p1 and p2 is: 7.07107
(0,0) Zero point
(1,1) One
(-1,1) Two
(-1,-1) Three
(1,-1) Four
(1,0) Zero point
(-1,0) Zero point
(0,1) Zero point
(0,-1) Zero point

```

Рисунок 31

В примере 32 мы определили две дружественные функции `distance` и `quadrant`. Обе функции получают

доступ к переменным-членам класса `Point` непосредственно, без необходимости использования геттеров. Функцию `quadrant` мы полностью определили внутри класса `Point`. Такое определение дружественной функции не противоречит стандарту и в некоторых случаях является удобным и уместным.

Следует отметить, что вне зависимости от способа определения дружественной функции, определения дружественности внутри класса и, собственно, определения функции вне класса либо же определения дружественности функции и самой функции внутри класса, дружественные функции не становятся функциями-членами класса. У дружественных функций нет неявного указателя `this`, их нельзя вызывать в контексте экземпляра класса.

Настало время переработать пример из предыдущего раздела урока на использование дружественных классов глобальных функций, как перегрузок соответствующих операторов.

Пример 33

```
#include <iostream>

class Fraction
{
    int numerator;
    int denominator;
    int gcd(int a, int b);
    void reduce();

public:
```

```

Fraction(int num, int denom)
    : numerator{ num }, denominator{ denom ? denom : 1
}
{
    reduce();
};

Fraction() : Fraction(1, 1) {};
void setNumerator(int num)
    { numerator = num; reduce(); };
int getNumerator() const { return numerator; };
void setDenominator(int denom)
    { denominator = denom ? denom : 1; reduce(); };
int getDenominator() const { return denominator; }
void print() const;
friend Fraction operator+(const Fraction& left,
                          const Fraction& right);
friend Fraction operator+(const Fraction& left,
                          int right);
friend Fraction operator+(int left,
                          const Fraction& right);
friend Fraction operator-(const Fraction& left,
                          const Fraction& right);
friend Fraction operator-(const Fraction& left,
                          int right);
friend Fraction operator-(int left,
                          const Fraction& right);
friend Fraction operator*(const Fraction& left,
                          const Fraction& right);
friend Fraction operator*(const Fraction& left,
                          int right);
friend Fraction operator*(int left,
                          const Fraction& right);
friend Fraction operator/(const Fraction& left,
                          const Fraction& right);
friend Fraction operator/(const Fraction& left,
                          int right);

```

```

        friend Fraction operator/(int left,
                                   const Fraction& right);
};

int Fraction::gcd(int a, int b)
{
    int copy;

    while (b)
    {
        a %= b;
        copy = a;
        a = b;
        b = copy;
    }

    return a;
}

void Fraction::reduce()
{
    int gcdVal{ gcd(numerator, denominator) };
    numerator /= gcdVal;
    denominator /= gcdVal;

    if (denominator < 0 and numerator < 0)
    {
        denominator *= -1;
        numerator *= -1;
    }
}

void Fraction::print() const
{
    std::cout << '(' << numerator << " / "
               << denominator << ")";
}

```

```
Fraction operator+(const Fraction& left,
                  const Fraction& right)
{
    return Fraction{ left.numerator *
                     right.denominator +
                     right.numerator *
                     left.denominator,
                     left.denominator *
                     right.denominator };
}

Fraction operator+(const Fraction& left, int right)
{
    return Fraction{ left.numerator + right *
                     left.denominator, left.denominator };
}

Fraction operator+(int left, const Fraction& right)
{
    return right + left;
}

Fraction operator-(const Fraction& left,
                  const Fraction& right)
{
    return Fraction{ left.numerator * right.denominator -
                     right.numerator * left.denominator,
                     left.denominator * right.denominator };
}

Fraction operator-(const Fraction& left, int right)
{
    return Fraction{ left.numerator - right *
                     left.denominator,
                     left.denominator };
}
```

```
Fraction operator-(int left, const Fraction& right)
{
    return Fraction{ left * right.denominator -
                     right.numerator, right.denominator };
}

Fraction operator*(const Fraction& left,
                  const Fraction& right)
{
    return Fraction{ left.numerator *
                     right.numerator, left.denominator *
                     right.denominator };
}

Fraction operator*(const Fraction& left, int right)
{
    return Fraction{ left.numerator * right,
                     left.denominator };
}

Fraction operator*(int left, const Fraction& right)
{
    return right * left;
}

Fraction operator/(const Fraction& left,
                  const Fraction& right)
{
    return Fraction{ left.numerator * right.denominator,
                     left.denominator * right.numerator };
}

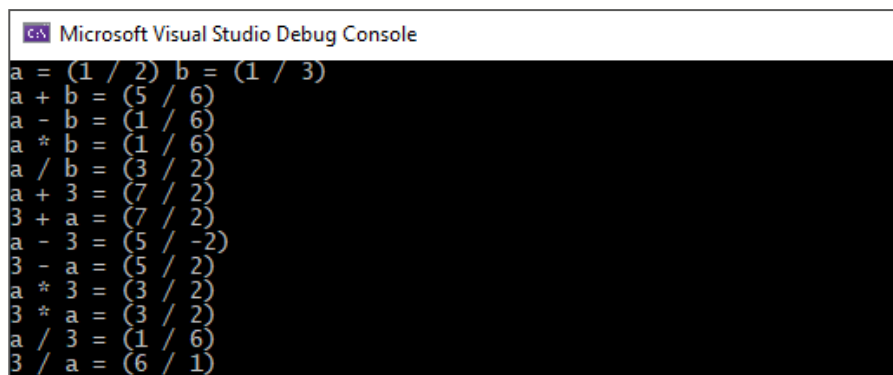
Fraction operator/(const Fraction& left, int right)
{
    return Fraction{ left.numerator ,
                     left.denominator * right };
}
```

```

Fraction operator/(int left, const Fraction& right)
{
    return Fraction{ left * right.denominator,
                     right.numerator };
}
int main()
{
    Fraction a{ 1,2 };
    Fraction b{ 1,3 };
    std::cout << "a = "; a.print(); std::cout
               << " b = "; b.print(); std::cout << '\n';
    std::cout << "a + b = "; (a + b).print(); std::cout
               << '\n';
    std::cout << "a - b = "; (a - b).print(); std::cout
               << '\n';
    std::cout << "a * b = "; (a * b).print(); std::cout
               << '\n';
    std::cout << "a / b = "; (a / b).print(); std::cout
               << '\n';
    std::cout << "a + 3 = "; (a + 3).print(); std::cout
               << '\n';
    std::cout << "3 + a = "; (3 + a).print(); std::cout
               << '\n';
    std::cout << "a - 3 = "; (a - 3).print(); std::cout
               << '\n';
    std::cout << "3 - a = "; (3 - a).print(); std::cout
               << '\n';
    std::cout << "a * 3 = "; (a * 3).print(); std::cout
               << '\n';
    std::cout << "3 * a = "; (3 * a).print(); std::cout
               << '\n';
    std::cout << "a / 3 = "; (a / 3).print(); std::cout
               << '\n';
    std::cout << "3 / a = "; (3 / a).print(); std::cout
               << '\n';
    return 0;
}

```

Вывод результата работы из примера 33.



```

Microsoft Visual Studio Debug Console
a = (1 / 2) b = (1 / 3)
a + b = (5 / 6)
a - b = (1 / 6)
a * b = (1 / 6)
a / b = (3 / 2)
a + 3 = (7 / 2)
3 + a = (7 / 2)
a - 3 = (5 / -2)
3 - a = (5 / 2)
a * 3 = (3 / 2)
3 * a = (3 / 2)
a / 3 = (1 / 6)
3 / a = (6 / 1)

```

Рисунок 32

Как видно из вывода 33, результат работы программы идентичен примеру 31. Вся разница в том, что теперь глобальные функции-перегрузки операторов являются, с «разрешения» и «согласия» класса [Fraction](#), его друзьями. Как следствие, они имеют доступ к [private](#) переменным-членам непосредственно, без необходимости использовать геттеры. Работа арифметических операторов в таком виде станет эффективнее, без накладных расходов, связанных с вызовом геттеров.

Перегрузка операторов дружественными функциями — весьма эффективный инструмент, зачем же тогда перегружать операторы обычными «недругами»? Все дело в том, что класс, для которого необходимо создать перегрузку того или иного оператора, может быть вне нашего административного ведения, его может разрабатывать другой программист. Автор класса может не согласиться включить сторонние функции в перечень дружественных классу функций, а, как мы помним, окончательное

решение «дружить» или «не дружить» принимает автор класса, желания функции совершенно недостаточно. Иной сценарий, при котором «дружба» невозможна, — отсутствие исходного кода некоего класса, ведь мы можем пользоваться готовым классом, реализация которого нам недоступна. В таком случае единственный выход — использовать перегрузку операторов глобальными функциями, которые, в свою очередь, будут обращаться к переменным-членам класса через соответствующие геттеры/сеттеры.

Ряд операторов возможно перегрузить лишь функциями-членами класса:

- оператор `=` — присваивание,
- оператор `()` — вызов функции,
- оператор `[]` — доступ по индексу,
- оператор `->` — доступ к члену через указатель.

Их невозможно реализовать дружественными или глобальными функциями.

11. Перегрузка ввода-вывода

Перегрузка операторов в C++ позволяет реализовывать для новых типов данных, классов C++, поведение и семантику использования, схожую или повторяющую таковое в базовых типах, таких как `int`, `double`, `bool` и так далее. Все базовые типы позволяют выводить свои значения на экран и вводить значения посредством клавиатуры. Используя ту же семантику, можно вводить/выводить данные в разные устройства ввода/вывода. Например, сохранять данные в файл или загружать данные из файла. Для базовых типов такое поведение обеспечивают перегрузки операторов побитового сдвига влево, именуемых в данном контексте «помещение в поток» и оператор побитового сдвига вправо, именуемый «извлечение из потока». Настало время рассмотреть специфику перегрузки ввода/вывода для реализуемых нами классов.

Для начала рассмотрим, как работает вывод на примере базового типа `int`:

```
std::cout << 42;
```

здесь `<<` — бинарный оператор, а `42` и `std::cout` — его операнды. Глобальный объект `std::cout` является экземпляром класса, опосредовано унаследованным от класса `std::ostream`, реализующим базовый интерфейс потокового вывода. Например, класс `ofstream` так же является наследником `std::ostream`. Как следствие, если реализовать перегрузку оператора `<<` для класса `std::ostream`, то будет возможен как вывод на экран, так и в файл. Немного

забегая вперед, отметим, что совершенно аналогично обстоит дело и с потоками ввода, то есть, реализовав перегрузку `>>` для класса `std::istream`, станет возможно проводить ввод с клавиатуры, а также из файла.

Вернемся к оператору помещения в поток. Как осуществить его перегрузку? Подойдет ли нам функция-член класса? Нет, ведь левым операндом является класс `std::ostream`! Функции-члены пригодны для перегрузки операторов, левым аргументом которых является собственно класс-владелец функции. Соответственно остается лишь вариант с глобальной функцией или дружественной функцией.

Еще одной особенностью данного оператора является его возможность работать «по цепочке»:

```
std::cout << "Answer is " << 42;
```

Для реализации такой возможности необходимо обеспечить возврат из перегрузки оператора объекта, пригодного для повторного вызова оператора помещения в поток. Например, выражение выше можно представить в виде:

```
((std::cout << "Answer is ")/*1*/ << 42)/*2*/;
```

То есть выражение 1 в результате должно вернуть `std::cout`, чтоб корректно отработало выражение 2.

Рассуждения выше приводят нас к следующей сигнатуре перегрузки оператора помещения в поток:

```
std::ostream& operator<<(std::ostream& out,  
const ClassName& object);
```

Следует обратить внимание на то, что как передача в функцию, так и возврат из нее экземпляра `std::ostream` необходимо производить по неконстантной ссылке. Передача/ возврат производится по ссылке, чтобы избежать копирования, которое, к слову, запрещает проводить класс `std::ostream` (материалы предыдущих разделов урока могут помочь понять как именно это реализовано). Также обращаем внимание, что ссылка на `std::ostream` неконстантная, ведь объект будет изменяться помещаемыми в него данными. Собственно, помещаемый в поток объект передается в функцию по константной ссылке из соображений экономии на копировании и запрета модификации внутри оператора исходного, помещаемого в поток, экземпляра `ClassName`.

Модифицируем пример 32 и реализуем для класса `Point` перегрузку оператора помещения в поток дружественной функцией, заменив тем самым функциональность, ранее предоставляемую функцией-членом `show`.

Пример 34

```
#include <iostream>

enum Quadrants { ZeroPoint, First, Second, Third, Fourth,
                 XPositive, XNegative, YPositive, YNegative};

class Point
{
    int x;
    int y;

public:
    Point() = default;
```

```

Point(int pX, int pY) : x{ pX }, y{ pY } {}
Point& setX(int pX) { x = pX; return *this; }
Point& setY(int pY) { y = pY; return *this; }
int getX()const { return x; }
int getY()const { return y; }

void show() const
{
    std::cout << '(' << x << ', ' << y << ')';
}

friend double distance(const Point& p1, const Point& p2);
friend int quadrant(const Point& p)
{
    if (!p.y and !p.x) { return
        Quadrants::ZeroPoint; }
    if (!p.y) { return p.x > 0 ? Quadrants::
        XPositive : Quadrants::XNegative; }
    if (!p.x) { return p.y > 0 ? Quadrants::
        YPositive : Quadrants::YNegative; }
    if (p.y > 0)
    {
        return p.x > 0 ? Quadrants::First :
            Quadrants::Second;
    }
    return p.x > 0 ? Quadrants::Fourth : Quadrants::Third;
}
friend std::ostream& operator<<(std::ostream& out,
    const Point& point);
};

double distance(const Point& p1, const Point& p2)
{
    auto xLength{ p2.x - p1.x };
    auto yLength{ p2.y - p1.y };
    return sqrt(xLength * xLength + yLength * yLength);
}

```

```

void quadrantDecode(int quadrant)
{
    if (quadrant == Quadrants::ZeroPoint)
        { std::cout << "Zero point"; }
    else if (quadrant == Quadrants::First)
        { std::cout << "First quadrant"; }
    else if (quadrant == Quadrants::Second)
        { std::cout << "Second quadrant"; }
    else if (quadrant == Quadrants::Third)
        { std::cout << "Third quadrant"; }
    else if (quadrant == Quadrants::Fourth)
        { std::cout << "Fourth quadrant"; }
    else if (quadrant == Quadrants::XPositive)
        { std::cout << "X axis positive"; }
    else if (quadrant == Quadrants::XNegative)
        { std::cout << "X axis negative"; }
    else if (quadrant == Quadrants::YPositive)
        { std::cout << "Y axis positive"; }
    else if (quadrant == Quadrants::YNegative)
        { std::cout << "Y axis negative"; }
}

std::ostream& operator<<(std::ostream& out,
                        const Point& point)
{
    out << '(' << point.x << ',' << point.y << ')';
    return out;
}

int main()
{
    Point p1{ 5,5 };
    Point p2{ 10,10 };

    std::cout << "Distance between p1 " << p1
               << " and p2 " << p2 << " is: "
               << distance(p1, p2) << '\n';
}

```

```

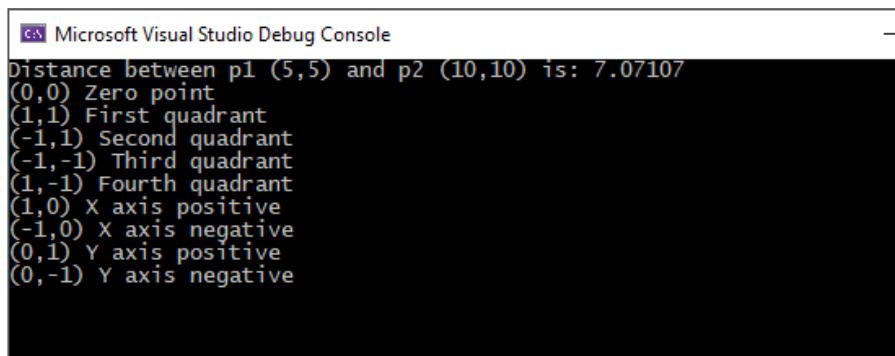
const int testCases{ 9 };
Point points[testCases]{
    {0,0},
    {1,1},
    {-1,1},
    {-1,-1},
    {1,-1},
    {1,0},
    {-1,0},
    {0,1},
    {0,-1}
};

for (int i{ 0 }; i < testCases; ++i)
{
    std::cout << points[i] << ' ';
    quadrantDecode(quadrant(points[i]));
    std::cout << '\n';
}

return 0;
}

```

Вывод результата работы из примера 34



The screenshot shows the Microsoft Visual Studio Debug Console with the following output:

```

Distance between p1 (5,5) and p2 (10,10) is: 7.07107
(0,0) Zero point
(1,1) First quadrant
(-1,1) Second quadrant
(-1,-1) Third quadrant
(1,-1) Fourth quadrant
(1,0) X axis positive
(-1,0) X axis negative
(0,1) Y axis positive
(0,-1) Y axis negative

```

Рисунок 33

Вывод результата работы из примера 34 крайне похож на вывод 32, однако сам код стал более семантическим, появилась возможность помещать объекты класса `Point` в поток, выводя их на экран, как это делается в C++ для всех базовых типов. При реализации перегрузки оператора помещения в поток важно не забывать возвращать ссылку на поток из перегрузки оператора. Такой возврат совершенно «законен», ведь поток вывода существовал до вызова оператора и будет существовать после его завершения, соответственно, мы не получим так называемую «висячую ссылку», наоборот, мы получим возможность сцепленного помещения в поток.

Перегрузка оператора извлечения из потока происходит полностью аналогично оператору помещения в поток, только вместо класса `std::ostream` используется класс `std::istream`. Все соображения касательно способа перегрузки, передачи параметров, возврата значения, совместимость с вводом из файла — все это справедливо и для перегрузки извлечения из потока, реализующего ввод для экземпляров классов в стиле C++. И все же есть одно важное отличие: экземпляр класса, для которого реализуется перегрузка, передается по не константной ссылке, ведь объект в процессе извлечения из потока будет модифицирован!

Сигнатура функции извлечения из потока:

```
std::istream& operator>>(std::istream& in, ClassName& object);
```

Аналогично помещению в поток, для поддержки сцепленного ввода значений необходимо не забывать возвращать экземпляр `std::istream` из перегрузки оператора извлечения из потока.

При реализации извлечения из потока возникает соблазн непосредственно в функции выводить на экран приглашение ввода, поясняющее пользователю что именно он вводит и в каком порядке, данное желание вполне естественно. Однако не следует забывать, что ввод может производиться не только интерактивно, с клавиатуры, но также и совершенно автоматически — из файла, в таком случае вывод сообщений на экран будет нежелательным. Рекомендуются избегать встраивать сообщения с приглашением ввода непосредственно в саму перегрузку оператора извлечения из потока. В качестве альтернативы стоит рассмотреть вспомогательную функцию-член, выводящую приглашение ввода, тогда такую отдельную функцию возможно будет использовать в связке с интерактивным извлечением из потока.

Для демонстрации дополним наш класс `Point` перегрузкой оператора извлечения из потока, а также вспомогательной функцией, выводящей на экран приглашение ввода. Немного модифицируем предыдущий пример 34: позволим пользователю самому вводить точки для определения их принадлежности к соответствующему квадранту или оси координат.

Пример 35

```
#include <iostream>

enum Quadrants { ZeroPoint, First, Second, Third,
                 Fourth, XPositive, XNegative, YPositive,
                 YNegative };

class Point
{
```

```

int x;
int y;

public:
    Point() = default;
    Point(int pX, int pY) : x{ pX }, y{ pY } {}
    Point& setX(int pX) { x = pX; return *this; }
    Point& setY(int pY) { y = pY; return *this; }
    int getX()const { return x; }
    int getY()const { return y; }

    void show() const
    {
        std::cout << '(' << x << ',' << y << ')';
    }

    friend double distance(const Point& p1, const Point& p2);
    friend int quadrant(const Point& p)
    {
        if (!p.y and !p.x) { return Quadrants::
                               ZeroPoint; }
        if (!p.y) { return p.x > 0 ? Quadrants::
                               XPositive : Quadrants::XNegative; }
        if (!p.x) { return p.y > 0 ? Quadrants::
                               YPositive : Quadrants::YNegative; }
        if (p.y > 0)
        {
            return p.x > 0 ? Quadrants::First : Quadrants::
                               Second;
        }
        return p.x > 0 ? Quadrants::Fourth : Quadrants::
                               Third;
    }

    Point& inputPrompt() { std::cout <<
        "Enter point coordinates x and y\n";
        return *this; }

```

```

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

double distance(const Point& p1, const Point& p2)
{
    auto xLength{ p2.x - p1.x };
    auto yLength{ p2.y - p1.y };
    return sqrt(xLength * xLength + yLength * yLength);
}

void quadrantDecode(int quadrant)
{
    if (quadrant == Quadrants::ZeroPoint)
        { std::cout << "Zero point"; }
    else if (quadrant == Quadrants::First)
        { std::cout << "First quadrant"; }
    else if (quadrant == Quadrants::Second)
        { std::cout << "Second quadrant"; }
    else if (quadrant == Quadrants::Third)
        { std::cout << "Third quadrant"; }
    else if (quadrant == Quadrants::Fourth)
        { std::cout << "Fourth quadrant"; }
    else if (quadrant == Quadrants::XPositive)
        { std::cout << "X axis positive"; }
    else if (quadrant == Quadrants::XNegative)
        { std::cout << "X axis negative"; }
    else if (quadrant == Quadrants::YPositive)
        { std::cout << "Y axis positive"; }
    else if (quadrant == Quadrants::YNegative)
        { std::cout << "Y axis negative"; }
}

std::ostream& operator<<(std::ostream& out, const Point& point)
{

```

```

    out << '(' << point.x << ',' << point.y << ')';
    return out;
}

std::istream& operator>>(std::istream& in, Point& point)
{
    in >> point.x >> point.y;
    return in;
}

int main()
{
    Point p1{};
    char menu{ 'n' };
    bool correct{ false };

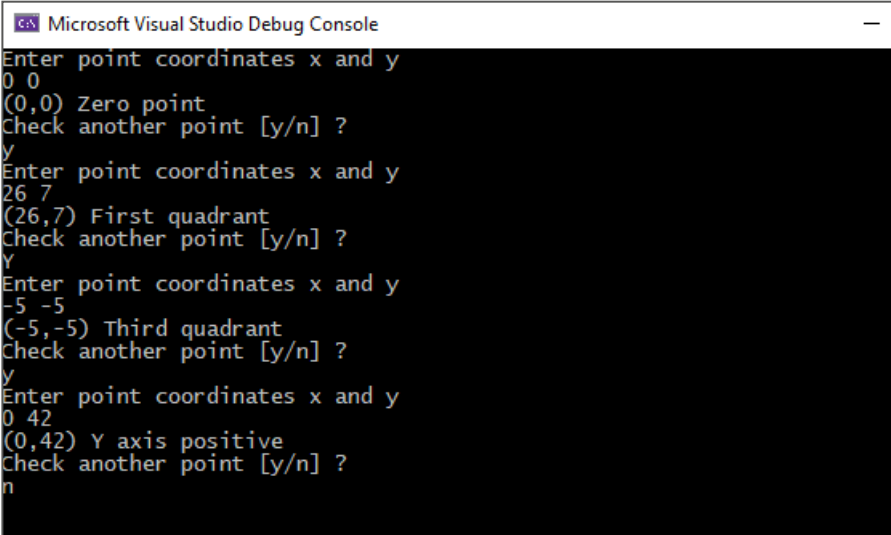
    do
    {
        std::cin >> p1.inputPrompt();
        std::cout << p1 << ' ';
        quadrantDecode(quadrant(p1));
        std::cout << '\n';

        do
        {
            std::cout << "Check another point [y/n] ?\n";
            std::cin >> menu;
            correct = menu == 'y' or menu == 'Y'
                    or menu == 'n' or menu == 'N';
            if (!correct) { std::cout << "Incorrect
                           choice! Try again!\n"; }
        } while (!correct);
    } while (menu == 'y' or menu == 'Y');

    return 0;
}

```

Вывод результата работы из примера 35.



```

Microsoft Visual Studio Debug Console
Enter point coordinates x and y
0 0
(0,0) Zero point
Check another point [y/n] ?
y
Enter point coordinates x and y
26 7
(26,7) First quadrant
Check another point [y/n] ?
Y
Enter point coordinates x and y
-5 -5
(-5,-5) Third quadrant
Check another point [y/n] ?
y
Enter point coordinates x and y
0 42
(0,42) Y axis positive
Check another point [y/n] ?
n
  
```

Рисунок 34

Вывод 35 демонстрирует интерактивную сессию, где пользователь последовательно вводит координаты точек `0,0`; `26,7`; `-5,-5`; `0,42`, а программа определяет принадлежность введенных точек к соответствующим квадрантам или осям координат. В данном примере реализован небольшой «трюк»: функция-член `inputPrompt()` возвращает ссылку на экземпляр класса `Point`, таким образом, для интерактивного ввода точки посредством клавиатуры с одновременным выводом пользователю приглашения ввода возможно использовать конструкцию вида `std::cin >> p1.inputPrompt();`. Для вычисления приведенного выражения сперва необходимо вызвать функцию-член `inputPrompt()`, которая выведет сообщение на экран и вернет ссылку на экземпляр `p1`, который,

в свою очередь, будет использован как правый аргумент в перегрузке оператора извлечения из потока.

Итак, мы научились перегружать операторы помещения в поток и извлечения из потока, теперь мы сможем реализовывать ввод и вывод создаваемых нами классов так же просто и удобно, как и в базовых типах C++.

12. Дружественные классы

Помимо дружественных функций, у класса могут быть также дружественные классы. Для всех функций-членов дружественного класса будет открыт доступ к `private` и `protected` членам того класса, другом которого он является. Синтаксис определения дружбы между классами аналогичен таковому при определении дружественных функций:

```
class FriendClass
{
    // определение класса FriendClass
};

class SomeClass
{
    // определение класса SomeClass
    friend FriendClass;
};
```

В данном примере класс `FriendClass` становится дружественным классу `SomeClass` и, соответственно, все функции-члены класса `FriendClass` получают доступ ко всем членам класса `SomeClass`, даже `private` и `protected`. Однако, следует отметить, что дружественность не взаимна! Функции-члены класса `SomeClass` не получают доступа к `private` и `protected` членам класса `FriendClass`.

Когда следует использовать дружественные классы? В случае, когда классы тесно взаимодействуют друг с другом и взаимодействие критично к производительности,

в таких случаях вполне уместно использовать дружественность классов. Однако следует с осторожностью «выбирать друзей» и применять дружбу классов, если это действительно оправданно. Как более безопасную альтернативу всегда следует рассматривать дружественные функции-члены класса.

Проиллюстрируем дружественность классов на примере базовой реализации класса, моделирующего прямоугольник в двумерном пространстве. Данная реализация не претендует на полноту и совершенную корректность. Для упрощения примера и чтобы сосредоточиться на, собственно, дружественности классов, в нем не выполняется ряд проверок и корректировок.

Пример 36

```
#include <iostream>

class Rectangle;
class Point
{
    int x;
    int y;

public:
    Point() = default;
    Point(int pX, int pY) : x{ pX }, y{ pY } {}
    Point& setX(int pX) { x = pX; return *this; }
    Point& setY(int pY) { y = pY; return *this; }
    int getX()const { return x; }
    int getY()const { return y; }

    void show() const
    {
```



```

        std::cout << '(' << x << ',' << y << ')';
    }

    Point& inputPrompt() { std::cout << "Enter point "
                          << "coordinates x and y\n";
                          return *this; }
    friend std::ostream& operator<<(std::ostream& out,
                                    const Point& point);
    friend std::istream& operator>>(std::istream& in,
                                    Point& point);
    friend Rectangle;
};

class Rectangle
{
    Point leftUpCorner;
    Point rightDownCorner;

public:
    Rectangle() = default;
    Rectangle(const Point& leftUpCornerP,
              int sideAP, int sideBP)
        : leftUpCorner{ leftUpCornerP },
          rightDownCorner{leftUpCornerP.x + sideAP,
                        leftUpCorner.y + sideBP}
    {}
    Rectangle(const Point& leftUpCornerP,
              const Point& rightDownCornerP)
        : leftUpCorner{ leftUpCornerP },
          rightDownCorner{rightDownCornerP}
    {}

    int getSideA() const { return
                        rightDownCorner.x - leftUpCorner.x; }

    int getSideB() const { return
                        rightDownCorner.y - leftUpCorner.y; }

```

```

        friend std::ostream& operator<<(std::ostream& out,
                                         const Rectangle& rectangle);
};

std::ostream& operator<<(std::ostream& out,
                        const Point& point)
{
    out << '(' << point.x << ',' << point.y << ')';
    return out;
}

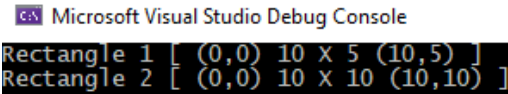
std::istream& operator>>(std::istream& in, Point& point)
{
    in >> point.x >> point.y;
    return in;
}

std::ostream& operator<<(std::ostream& out,
                        const Rectangle& rectangle)
{
    out << "[ " << rectangle.leftUpCorner << ' '
        << rectangle.getSideA()
        << " X "
        << rectangle.getSideB()
        << ' ' << rectangle.rightDownCorner
        << " ]";
    return out;
}

int main()
{
    Rectangle rect1{ {0,0}, 10 ,5 };
    Rectangle rect2{ {0,0}, {10,10} };
    std::cout << "Rectangle 1 " << rect1 << '\n'
        << "Rectangle 2 " << rect2 << '\n';
    return 0;
}

```

Вывод результата работы из примера 36.



```
Microsoft Visual Studio Debug Console
Rectangle 1 [ (0,0) 10 x 5 (10,5) ]
Rectangle 2 [ (0,0) 10 x 10 (10,10) ]
```

Рисунок 35

В примере 36 реализован класс `Rectangle` с двумя переменными-членами типа `Point`: `leftUpCorner` — координата левого верхнего угла и `rightDownCorner` — координата нижнего правого угла. Класс `Rectangle` указан в классе `Point` как дружественный класс. Именно это позволяет эффективно реализовать ряд функций-членов класса `Rectangle` — конструктор, принимающий левый верхний угол и две стороны прямоугольника, а также функции-члены, возвращающие длины двух сторон прямоугольника `getSideA()`, `getSideB()`. Для класса `Rectangle` реализована перегрузка оператора помещения в поток для удобного вывода информации о прямоугольнике на экран. Вывод результата работы из примера 36 свидетельствует о корректности работы программы.

13. Домашнее задание

Задание 1

Реализуйте в классе `MedalsTable` из примера 18 возможность динамически задавать размер таблицы медалей. Текущая реализация для упрощения использует статический массив на 10 элементов. Замените его на динамически выделяемый массив. Для класса `MedalsTable` реализуйте семантику копирования и семантику перемещения (две пары конструктор / оператор присваивания).

Задание 2

Дополните решение из задания 1 перегрузкой оператора помещения в поток (`operator<<`) для классов `MedalRow` и `MedalsTable`, заменив тем самым соответствующие функции-члены `print()` в них.

Задание 3

Дополните решение из задания 2 реализацией оператора вызова функции для класса `MedalsTable`. Перегрузка должна принимать в качестве аргумента идентификатор страны и возвращать одну из констант `MedalRow::GOLD`, `MedalRow::SILVER`, `MedalRow::BRONZE` как константу соответствующую максимальному количеству медалей для заданной страны. То есть если, к примеру, у Польши 2 золотые, 4 серебрянных и одна бронзовая медаль, то перегрузка оператора вызов функции с параметром `POL` вернет `MedalRow::SILVER`.

Задание 4

Модифицируйте функтор `NoSequence` из задания 26, чтоб он игнорировал не менее `N` подряд значений, где `N` — параметр конструктора данного функтора.