Урок 5

Совместное использование функций и методов

Перегрузка операторов. Шаблоны функций и классов. Явная и частичная специализация шаблонов. Примеры простых программ.

Перегрузка операторов инкремента и декремента

Перегрузка операторов инкремента (++) и декремента (−−) довольно проста, но есть нюанс. Есть две версии операторов инкремента и декремента: префикс (++x, --y) и постфикс (x++, y--).

Поскольку операторы инкремента и декремента являются унарными и изменяют свои операнды, то перегрузку следует выполнять через методы класса.

Перегрузка операторов инкремента и декремента версии префикс.

```
#include <iostream>
using namespace std;

class Day
{
private:
    int m_day;
public:
    Day(int day=1) : m_day(day)
    { }
    Day& operator++();
    Day& operator--();
    void getDay() { cout << m_day << endl;}
};

Day& Day::operator++()
{
    // Если значением переменной m_day является 32, то выполняем сброс на 1
    if (m_day == 32)
        m_day = 1;
    // в противном случае просто увеличиваем m_day на единицу
    else
        ++m_day;

    return *this;
}

Day& Day::operator--()
{
      // Если значением переменной m_day является 0, то присваиваем m_day значение 31
    if (m_day == 0)
```

```
        m_day = 8;
    // в противном случае просто уменьшаем m_day на единицу
    else
        --m_day;
    return *this;
}

int main()
{
    Number number(8);
    number.getNumber();
    (++number).getNumber();
    (--number).getNumber();
    return 0;
}
Day& Day::operator--()
{
      // Если значением переменной m_day является 0, то присваиваем m_day
значение 31
    if (m_day == 0)
        m_day = 31;
    // в противном случае просто уменьшаем m_day на единицу
    else
        --m_day;
    return *this;
}

int main()
{
    Day day(31);
    day.getDay();
    (++day).getDay();
    (--day).getDay();
    return 0;
}
```

Результат выполнения программы:

31

1

31

Здесь класс **Day** содержит число от 1 до 31. Мы перегрузили операторы инкремента/декремента таким образом, чтобы они увеличивали/уменьшали **m_day** в соответствии с заданным диапазоном.

Обратите внимание, мы возвращаем скрытый указатель ***this** в функциях перегрузки операторов (то есть текущий объект класса **Day**). Таким образом мы можем связать выполнение нескольких операторов в одну «цепочку».

Перегрузка операторов инкремента и декремента версии постфикс

Операторы инкремента и декремента версии префикс и постфикс имеют одно имя, оба унарные и принимают один параметр одного и того же типа данных. Как же их различать при перегрузке?

C++ использует **фиктивную переменную** или **фиктивный параметр** для операторов версии постфикс. Этот фиктивный целочисленный параметр используется с одной целью: отличить версию постфикс операторов инкремента/декремента от версии префикс. Перегрузим префиксную и постфиксную версию операторов инкремента/декремента в том же классе:

```
#include <iostream>
using namespace std;

class Day
{
private:
    int m_day;
public:
    Day(int day=1) : m_day (day)
    { }

    Day& operator++();   // версия префикс
    Day & operator--();  // версия префикс

    Day operator++(int); // версия постфикс
    Day operator--(int); // версия постфикс

    void getDay() { cout << m_day << endl;}
};

//префиксная версия инкремента
Day & Day::operator++()
{
    if (m_day == 32)
        m_day = 1;
    else
        ++m_day;

    return *this;
}

//префиксная версия декремента
Day & Day::operator--()
{
    if (m_number == 0)
        m_number = 31;
    else
        --m_number;

    return *this;
}
```

```
// постфиксная версия инкремента
Day Day::operator++(int)
{
      // Создаем временный объект класса Day с текущим значением переменной
m_day
    Day temp(m_day);

    // Используем оператор инкремента версии префикс для реализации перегрузки
оператора инкремента версии постфикс
    ++(*this); // реализация перегрузки

    // возвращаем временный объект
    return temp;
}

// постфиксная версия декремента
Day Day::operator--(int)
{
      // Создаем временный объект класса Day с текущим значением переменной
m_day
    Day temp(m_day);

    // Используем оператор декремента версии префикс для реализации перегрузки
оператора декремента версии постфикс
    --(*this); // реализация перегрузки

    // возвращаем временный объект
    return temp;
}

int main()
{
    Day day(31);

    (day++).getDay();
    (day).getDay();
    (day--).getDay();
    (day).getDay();

    return 0;
}
```

Результат выполнения программы:

31

1

1

31

В данном коде:

1. Отделили версию постфикс от версии префикс, используя целочисленый фиктивный параметр в версии постфикс.
2. Поскольку фиктивный параметр не используется в реализации самой перегрузки, то мы даже не даем ему имя. Так что компилятор будет рассматривать эту переменную как простую

заглушку (заполнитель места) и даже не будет предупреждать нас, что мы объявили переменную, но никогда ее не использовали.

3. Операторы версий префикс и постфикс выполняют одно задание — оба увеличивают/уменьшают значение переменной объекта. Разница между ними только в значении, которое они возвращают. Операторы версии префикс возвращают объект после того, как он был увеличен или уменьшен. В версии постфикс нам нужно возвращать объект до того, как он будет увеличен или уменьшен.

   Поэтому мы использовали временный объект с текущим значением переменной-члена. Тогда можно будет увеличить/уменьшить исходный объект, а возвратить обратно временный.

   Обратите внимание: это означает, что возврат значения по ссылке невозможен, так как мы не можем возвратить ссылку на локальную переменную, которая будет уничтожена после завершения выполнения тела функции. Также это означает, что операторы версии постфикс обычно менее эффективны, чем операторы версии префикс — из-за дополнительных расходов ресурсов на создание временного объекта и выполнения возврата по значению вместо возврата по ссылке

4. Наконец, мы реализовали перегрузку операторов версии постфикс через уже перегруженные операторы версии префикс. Таким образом сократили дублированный код и упростили внесение будущих изменений в класс, то есть поддержку кода.

Шаблоны функций

Шаблоны функций — это инструкции, согласно которым создаются локальные версии функции для определенного набора параметров и типов данных. Это мощный инструмент в C++, который упрощает разработку.

Например, нам нужно запрограммировать функцию, которая выводила бы на экран элементы массива. Задача несложная, но чтобы написать такую функцию, надо знать тип данных массива, который будем выводить на экран. А если тип данных не один? Мы хотим, чтобы функция выводила массивы типа **int**, **double**, **float** и **char**. Можно, конечно, перегрузить функцию, но тогда появится 4 версии одной функции, только с другими типами параметра. Есть способ решить эту проблему проще — использовать шаблоны.

Создадим шаблон функции, определяющий максимальное из двух чисел:

```
template <typename T>
T max(T a, T b)
{
    return (a > b) ? a : b;
}
```

Для объявления шаблона функции сначала пишется ключевое слово ***template***, которое сообщает компилятору, что дальше мы будем объявлять параметры шаблона. Они указываются в угловых скобках (<>). Для создания типов параметров шаблона используются ключевые слова ***typename*** и ***class***. В базовых случаях использования шаблонов функций разницы между **typename** и **class** нет, поэтому можете выбрать любое. Если используете ключевое слово **class**, то фактический тип параметров не обязательно должен быть классом. Затем называем тип параметра шаблона (обычно «**T**», сокращенно от **Type**, что говорит о том, что это может быть любой тип данных). Если требуется несколько типов параметров шаблона, то они разделяются запятыми: **template <typename T1, typename T2>**.

Поскольку тип аргумента функции, передаваемый в тип **T**, может быть классом, а классы, как правило, не рекомендуется передавать по значению, то лучше сделать параметры и возвращаемое значение нашего шаблона функции константными ссылками. Тогда предыдущая функция перепишется следующим образом:

```
#include <iostream>
using namespace std;

template <typename T>
const T& t_max(const T& a, const T& b)
{
    return (a > b) ? a : b;
}
```

Использовать шаблоны функций можно так же, как и обычные функции:

```
int main()
{
    int i = t_max(4, 8);
    cout << i << '\n';

    double d = t_max(7.56, 21.434);
    cout << d << '\n';

    char ch = t_max('b', '9');
    cout << ch << '\n';

    return 0;
}
```

Результат выполнения программы:

8

21.434

B

Поскольку все три вызова функции **t_max()** имеют параметры разных типов, то компилятор создаст 3 **экземпляра шаблона функции** для каждого типа данных. Шаблоны функций работают как с фундаментальными типами данных (**char**, **int**, **double** и другими), так и с классами. Экземпляр шаблона компилируется как обычная функция. В обычной функции любые операторы или вызовы других функций, которые используются в ней, должны быть определены/перегружены — иначе вы получите ошибку компиляции. Аналогично, любые операторы или вызовы других функций, которые присутствуют в шаблоне функции, должны быть определены/перегружены для работы с передаваемыми типами данных.

Рассмотрим пример:

```
#include <iostream>
using namespace std;

template <typename T>
const T& t_max(const T& a, const T& b)
{
    return (a > b) ? a : b;
}

class Day
{
private:
    int m_day;
public:
    Day(int day) : m_day(day)
    { }
    // перегружаем оператор >
    friend bool operator>(const Day &d1, const Day &d2)
    {
        return (d1.m_day > d2.m_day);
    }
};
int main()
{
    Day seven(7);
    Day twelve(12);

    Day bigger = t_max(seven, twelve);

    return 0;
}
```

Чтобы использовать шаблон функции, необходимо перегрузить оператор сравнения для класса **Dollars**. После перегрузки можем спокойно использовать шаблон функции.

Обратите внимание: стандартная библиотека C++ имеет в своем арсенале шаблон функции **max()**, который находится в заголовочном файле **algorithm**. Поэтому вы можете не реализовывать эту функцию вручную в будущем. Данный шаблон функции находится в стандартном пространстве имен (**namespace std;**) — именно поэтому мы создали функцию **t_max()**, а не просто **max()**.

Шаблоны классов

На предыдущем уроке мы создавали динамический массив-класс целых чисел. Чтобы реализовать такой класс для вещественных чисел типа **float** или **double**, идеальным решением будет добавить шаблон класса.

```
template <class T> // шаблон класса
class Array
{
private:
    int m_length;
    T *m_data;

public:
    Array()
    {
        m_length = 0;
        m_data = nullptr;
    }

    Array(int length)
    {
        m_data = new T[length];
        m_length = length;
    }

    ~Array()
    {
        delete[] m_data;
    }

    void Erase()
    {
        delete[] m_data;
        // Указываем m_data значение nullptr, чтобы на выходе не получить
висячий указатель!
        m_data = nullptr;
        m_length = 0;
    }


    T& operator[](int index)
    {
        assert(index >= 0 && index < m_length);
        return m_data[index];
    }

    int getLength(); // определяем метод и шаблон метода getLength() ниже
};

// метод, определенный вне тела класса, нуждается в собственном определении
шаблона метода
template <typename T>
int Array<T>::getLength() { return m_length; }
// обратите внимание, имя класса - Array<T>, а не просто Array
```

Эта версия почти идентична версии **ArrayInt**, за исключением того, что мы добавили объявление параметра шаблона класса и изменили тип данных на **T**.

Обратите внимание: мы определили функцию **getLength()** вне тела класса. Это необязательно, просто привели для примера — так как каждый метод шаблона класса, объявленный вне тела класса, нуждается в собственном объявлении шаблона. Имя шаблона класса — **Array<T>**.

Создавать экземпляры шаблона класса нужно таким образом:

Array<int> intArray(10);

Array<double> doubleArray(10);

Шаблоны классов работают так же, как и шаблоны функций: компилятор копирует шаблон класса, заменяя типы параметров шаблона класса на фактические (передаваемые) типы данных, а затем компилирует эту копию. Если у вас есть шаблон класса, но вы его не используете, то компилятор не будет его даже компилировать.

Шаблоны классов идеально подходят для реализации контейнерных классов, так как очень часто таким классам приходится работать с разными типами данных, а шаблоны позволяют это организовать в минимальном количестве кода. Рассмотренный на прошлом уроке **std::vector** — это шаблон класса. В стандартной библиотеке C++ много предопределенных шаблонов классов, доступных для использования.

Параметр non-type шаблона

Параметр non-type шаблона — это специальный параметр шаблона, который заменяется не типом данных, а конкретным значением. Этим значением может быть:

- целочисленное значение или перечисление;
- указатель или ссылка на объект класса;
- указатель или ссылка на функцию;
- указатель или ссылка на метод класса.

Например, для создания статического массива-класса необходим еще один параметр — длина массива. Этот параметр передается в шаблон класса:

```
template <class T, int size> // size является параметром non-type шаблона класса
class StaticArray
{
private:
    // параметр non-type шаблона класса отвечает за размер выделяемого массива
    T m_array[size];
    ...
}
```

Экземпляр класса будет создаваться следующим выражением:

StaticArray<int, 10>.

Данный класс известен из стандартной библиотеки C++ - **std::array.**

Явная специализация шаблона функции

При создании экземпляра шаблона функции для определенного типа данных компилятор копирует шаблон функции и заменяет параметр типа шаблона функции на передаваемый тип данных. Это

означает, что все экземпляры функции имеют одну реализацию, но разные типы данных. Хотя в большинстве случаев это именно то, что требуется, иногда может понадобиться, чтобы реализация шаблона функции для одного типа данных отличалась от реализации для другого.

Специализация шаблонов именно для этого и предназначена.

Рассмотрим следующий шаблон класса:

```
#include <iostream>
using namespace std;

template <class T>
class Day
{
private:
    T m_day;
public:
    Day(T day)
    {
        m_day = day;
    }

    ~Day()
    { }

    void print()
    {
        cout << m_day << '\n';
    }
};
```

Теоретически данный класс может работать со многими типами данных (хотя этого может и не требоваться). Но что, если мы попытаемся использовать шаблон класса с типом данных **char***? При создании экземпляра шаблона для типа **char***, конструктор **Day<char*>** присвоит переменной **m_day** указатель **day**. Тогда функция **print()** выведет на экран значение указателя, а не сам текст. Чтобы решить эту проблему, используем явную специализацию шаблона класса:

```
template <>
Day<char*>::Day(char* day)
{
    // Определяем длину day
    int length=0;
    while (day[length] != '\0')
        ++length;
    ++length; // +1, учитывая нуль-терминатор

    // Выделяем память для хранения значения day
    m_day = new char[length];

    // Копируем фактическое значение day в m_day
    for (int count=0; count < length; ++count)
        m_day[count] = day[count];
}

template <>
Repository<char*>::~Repository()
{
    delete[] m_day;
}
```

Теперь при выделении переменной типа **Day<char*>** именно этот конструктор будет использоваться вместо стандартного. Чтобы избежать утечки памяти для типа **char*** используем деструктор, поскольку **m_day** не будет удален, когда переменная day выйдет из области видимости. Теперь, когда переменные типа **Day<char*>** выйдут из области видимости, память, выделенная в специальном конструкторе, будет удалена в специальном деструкторе.

Хотя во всех примерах выше мы работаем с методами класса, вы также можете аналогично выполнять явную специализацию шаблонов обычных функций.

Явная специализация шаблона класса

Специализация шаблона класса позволяет специализировать шаблон класса для работы с определенным типом данных (или сразу несколькими, если описано несколько параметров шаблона).

Специализация шаблона класса рассматривается компилятором как полностью отдельный и независимый класс, хотя и выделяется как обычный шаблон класса. Это означает, что мы можем изменить в классе что угодно, включая его реализацию, методы, спецификаторы доступа.

Рассмотрим класс-массив, который может хранить 8 дней:

```
template <class T>
class Day
{
private:
    T m_day[8];

public:
    void set(int index, const T &day)
    {
        m_day[index] = day;
    }

    const T& get(int index)
    {
        return m_day[index];
    }
};
```

Существуют классы, в которых переменная-член имеет тип данных **bool**. Для него эффективнее реализовать хранение переменных не в массиве (на что будет потрачено 8 байт), а в переменной типа **unsigned char**. Тогда на каждое значение bool-переменной будет уходить только по одному биту. В связи с вышесказанным потребуется отдельная специализация для типа данных **bool**.

Частичная специализация шаблона

Частичная специализация шаблона позволяет выполнить специализацию шаблона класса (но не функции), где некоторые параметры шаблона явно определены. Это называется частичной специализацией, потому что специализация шаблона происходит не по всем аргументам. Рассмотрим шаблон класса и его частичную специализацию:

```
// шаблон класса
template< typename T, typename S > class B {};
// его частичная специализация
template< typename U > class B< int, U > {};
```

В данном примере шаблонный класс с двумя параметрами специализируется только по одному из них. Специализация будет работать, когда первый аргумент, **T**, будет указан как **int**. При этом второй аргумент может быть любым — для этого в частичной специализации введен новый параметр **U** и указан в списке аргументов для специализации.

Чтобы выполнить частичную специализацию метода класса, можно сделать частичную специализация шаблона всего класса:

```
template <class T, int size>
class StaticArray
{
// реализация класса StaticArray
};

// частичная специализация шаблона класса
template <int size>
class StaticArray<double, size>
{
// реализация частично специализированного шаблона класса
};
```

Но в данном случае частично специализированный шаблон класса будет во многом дублировать код основного шаблона класса. Это проблема, которую можно решить путем наследования. Для это необходимо создать общий родительский класс:

```
// шаблон базового класса
template <class T, int size>
class StaticArray_Base
{
// реализация класса StaticArray_Base
};

// шаблон производного класса
template <class T, int size>
class StaticArray: public StaticArray_Base<T, size>
{
// полностью наследует функционал StaticArray_Base,
// собственные методы не нужны
};

// частично специализированный шаблон класса для типа double
template <int size>
class StaticArray<double, size>: public StaticArray_Base<double, size>
{
// реализация частично специализированного шаблона класса
};
```

В данном примере дублирование кода не происходит, а функционал остается прежним.

Отметим, что частичная специализация шаблонов нашла свое применение в основном при создании библиотек: **stl**, **boost**, **loki** и других. В библиотеках частичная специализация позволяет относительно просто реализовывать делегаты, события, сложные контейнеры и другие нужные и полезные вещи.

Частичная специализация шаблонов и указатели

В предыдущей части урока мы рассматривали пример с шаблоном класса **Day** и создавали полную специализацию этого шаблона для указателя типа **char***. Но что, если потребуется использовать указатели и на другие типы — например, **int***? Если не написать полную специализацию шаблона и для других типов указателей, программа не будет работать правильно, но получится много дублирующего кода. Выход из этой ситуации — объявить частичную специализацию шаблона класса **Day**, которая работала бы со всеми типами указателей.

```
#include <iostream>
using namespace std;

// Общий шаблон класса Day
template <class T>
class Day
{
private:
    T m_day;
public:
    Day(T day)
    {
        m_day = day;
    }

    ~Day()
    { }

    void print()
    {
        cout << m_day << '\n';
    }
};

// частичная специализация шаблона класса Day для работы с типами указателей
template <typename T>
class Day<T*>
{
private:
    T* m_day;
public:
    Day(T* day) // T - тип указателя
    {
        // Выполняем глубокое копирование
        m_day = new T(*day); // здесь копируется только одно отдельное значение
    }
    ~Day()
    {
        delete m_day; // а здесь выполняется удаление этого значения
    }

    void print()
    {
        cout << *m_day << '\n';
    }
};
```

Поскольку в нашей частичной специализации копируется только одно значение, то при работе со строками **C-style** копироваться будет только первый символ (так как строка — это массив, а указатель на массив указывает только на первый его элемент). Если же нужно скопировать целую строку, то специализация конструктора (и деструктора) для типа **char*** должна быть полной. В таком случае полная специализация будет иметь приоритет выше, чем частичная. Рассмотрим программу, в которой используется как частичная специализация для работы с типами указателей, так и полная специализация для работы с типом **char***:

```
#include <iostream>
#include <cstring>
using namespace std;
// Общий шаблон класса Day для работы не с указателями
template <class T>
class Day
{
private:
    T m_day;
public:
    Repository(T day)
    {
        m_day = day;
    }

    ~Day()
    { }

    void print()
    {
        cout << m_day << '\n';
    }
};

// Частичная специализация шаблона класса Day для работы с указателями
template <class T>
class Day<T*>
{
private:
    T* m_day;
public:
    Day(T* day)
    {
        m_day = new T(*day);
    }

    ~Day ()
    {
        delete m_day;
    }

    void print()
    {
        cout << *m_day << '\n';
    }
};
}
// Полная специализация шаблона конструктора класса Day для работы с типом
char*
template <>
Day<char*>::Day(char* day)
{
```

```
    // Определяем длину day
    int length = 0;
    while (day[length] != '\0')
        ++length;
    ++length; // +1, учитывая нуль-терминатор

    // Выделяем память для хранения значения day
    m_day = new char[length];



    // Копируем фактическое значение day в m_day
    for (int count = 0; count < length; ++count)
        m_day[count] = day[count];
}

// Полная специализация шаблона деструктора класса Day для работы с типом
char*
template<>
/day<char*>::~Day()
{
    delete[] m_day;
}

// Полная специализация шаблона метода print для работы с типом char*
// Без этого вывод Day<char*> привел бы к вызову Day<T*>::print(), которое
выводит только одно значение (в случае со строкой C-style - только первый
символ)
template<>
void Day<char*>::print()
{
    cout << m_day;
}

int main()
{
    // Объявляем целочисленный объект для проверки работы общего шаблона
класса
    Day<int> myDay(6);
    myDay.print();

    // Объявляем объект с типом указателя для проверки работы частичной
специализации шаблона
    int x = 8;
    Dy<int*> myintptr(&x);

    // Если бы в myintptr выполнилось поверхностное копирование (присваивание
указателя),
    // то изменение значения x изменило бы и значение myintptr
    x = 10;
    myintptr.print();

    // Динамически выделяем временную строку
    char *day = new char[40]{ "First" };

    // Сохраняем число
    Day<char*> myDay(day);

    // Удаляем временную строку
    delete[] name;
```

```
    // Выводим имя
    myname.print();
}
    // Удаляем временную строку
    delete[] day;

    // Выводим имя
    myDay.print();
}
```

Результат выполнения программы:

6
8
First

Таким образом, использование частичной специализации шаблона класса для работы с типами указателей особенно полезно, так как позволяет предусмотреть все возможные варианты использования программы на практике.

Написание игры Blackjack

На этом уроке создадим карту и колоду карт. Предлагаем масти карт и их достоинства оформить в виде перечисления, чтобы легче с ними работать:

```
enum rank {
        ACE = 1, TWO, THREE, FOUR, FIVE, SIX, SEVEN, EIGHT, NINE, TEN,
        JACK, QUEEN, KING
    };
enum suit { CLUBS, DIAMONDS, HEARTS, SPADES };
```

У класса **Card** есть метод, который переворачивает карту. В программе это будет реализовано с помощью переменной типа **bool**, где значению **false** соответствует карта, перевернутая рубашкой вверх, а значению true — рубашкой вниз.

```
void Card::Flip()
{
    m_IsFaceUp = !(m_IsFaceUp);
}
```

У класса **Card** также есть метод, который возвращает значение карты. Но оно доступно только в том случае, когда карта перевернута лицом вверх:

```
int Card::GetValue() const
{
    //если карта перевернута лицом вниз, ее значение равно 0
    int value = 0;
    if (m_IsFaceUp)
    {
        // значение - это число, указанное на карте
        value = m_Rank;
        // значение равно 10 для JACK, QUEEN и KING
        if (value > 10)
        {
            value = 10;
        }
    }
    return value;
}
```

Класс **Card**:

```
class Card
{
public:
    enum rank {
        ACE = 1, TWO, THREE, FOUR, FIVE, SIX, SEVEN, EIGHT, NINE, TEN,
        JACK, QUEEN, KING
    };
    enum suit { CLUBS, DIAMONDS, HEARTS, SPADES };

    Card(rank r = ACE, suit s = SPADES, bool ifu = true);
    int GetValue() const;
    void Flip();

private:
    rank m_Rank;
    suit m_Suit;
    bool m_IsFaceUp;
};

Card::Card(rank r, suit s, bool ifu) : m_Rank(r), m_Suit(s), m_IsFaceUp(ifu)
{}

int Card::GetValue() const
{
    int value = 0;
    if (m_IsFaceUp)
    {
        value = m_Rank;
        if (value > 10)
        {
```

```
            value = 10;
        }
    }
    return value;
}

void Card::Flip()
{
    m_IsFaceUp = !(m_IsFaceUp);
}
```

Теперь займемся классом **Hand**. Этот класс представляет собой коллекцию карт. Его конструктор резервирует в векторе **m_Cards** место в памяти под 7 элементов. Метод **Add()** добавляет новую карту в вектор. А метод **Clear()** очищает всю память, занятую вектором.

```
class Hand
{
public:
    Hand();
    // виртуальный деструктор
    virtual ~Hand();

    // добавляет карту в руку
    void Add(Card* pCard);

    // очищает руку от карт
    void Clear();

    //получает сумму очков карт в руке, присваивая тузу
    // значение 1 или 11 в зависимости от ситуации
    int GetTotal() const;

protected:
    vector<Card*> m_Cards;
};

Hand::Hand()
{
    m_Cards.reserve(7);
}
// деструктор по-прежнему виртуальный
// это уже можно не обозначать
Hand::~Hand()
{
    Clear();
}

void Hand::Add(Card* pCard)
{
    m_Cards.push_back(pCard);
}
```

```
void Hand::Clear()
{
    // проходит по вектору, освобождая всю память в куче
    vector<Card*>::iterator iter = m_Cards.begin();
    for (iter = m_Cards.begin(); iter != m_Cards.end(); ++iter)
    {
        delete *iter;
        *iter = 0;
    }
    // очищает вектор указателей
    m_Cards.clear();
}
```

Метод **Clear()** удаляет не только все указатели из вектора **m_Cards**, но и связанные объекты типа **Card** и освобождает занятую ими память. Это работает так же, как в реальном мире, когда в конце кона карты сбрасываются. Виртуальный деструктор вызывает этот метод.

Обратите внимание: хотя деструктор виртуальный, ключевое слово **virtual** не используется за пределами класса, а только внутри его определения.

Теперь представим реализацию метода **GetTotal()**:

```
int Hand::GetTotal() const
{
    // если карт в руке нет, возвращает значение 0
    if (m_Cards.empty())
    {
        return 0;
    }

    //если первая карта имеет значение 0, то она лежит рубашкой вверх:
    // вернуть значение 0
    if (m_Cards[0]->GetValue() == 0)
    {
        return 0;
    }

    // находит сумму очков всех карт, каждый туз дает 1 очко
    int total = 0;
    vector<Card*>::const_iterator iter;
    for (iter = m_Cards.begin(); iter != m_Cards.end(); ++iter)
    {
        total += (*iter)->GetValue();
    }

    // определяет, держит ли рука туз
    bool containsAce = false;
    for (iter = m_Cards.begin(); iter != m_Cards.end(); ++iter)
    {
        if ((*iter)->GetValue() == Card::ACE)
        {
            containsAce = true;
        }
    }

    // если рука держит туз и сумма довольно маленькая, туз дает 11 очков
    if (containsAce && total <= 11)
    {
        // добавляем только 10 очков, поскольку мы уже добавили
        // за каждый туз по одному очку
        total += 10;
    }

    return total;
}
```

Данный метод возвращает сумму очков для карт в руке. Если рука держит туз, он считается за 1 или 11 очков в зависимости от остальных карт. Количество очков, которое дает туз, определяется так: если в руке есть туз, он дает 11 очков; затем выполняется проверка, превышает ли сумма очков карт в руке число 21. Если нет — количество очков, которое дает туз, не изменяется. В противном случае туз даст 1 очко.

Практическое задание

1. Реализовать шаблон класса **Pair1**, который позволяет пользователю передавать данные одного типа парами.

   Следующий код:

```
int main()
{
    Pair1<int> p1(6, 9);
    cout << "Pair: " << p1.first() << ' ' << p1.second() << '\n';

    const Pair1<double> p2(3.4, 7.8);
    cout << "Pair: " << p2.first() << ' ' << p2.second() << '\n';

    return 0;
}
```

   … должен производить результат:

   Pair: 6 9
   Pair: 3.4 7.8

2. Реализовать класс **Pair**, который позволяет использовать разные типы данных в передаваемых парах.

   Следующий код:

```
int main()
{
    Pair<int, double> p1(6, 7.8);
    cout << "Pair: " << p1.first() << ' ' << p1.second() << '\n';

    const Pair<double, int> p2(3.4, 5);
    cout << "Pair: " << p2.first() << ' ' << p2.second() << '\n';

    return 0;
}
```

   … должен производить следующий результат:

   Pair: 6 7.8
   Pair: 3.4 5

   Подсказка: чтобы определить шаблон с использованием двух разных типов, просто разделите параметры типа шаблона запятой.

3. Написать шаблон класса **StringValuePair**, в котором первое значение всегда типа **string**, а второе — любого типа. Этот шаблон класса должен наследовать частично специализированный класс **Pair**, в котором первый параметр — **string**, а второй — любого типа данных.

Следующий код:

```
int main()
{
    StringValuePair<int> svp("Amazing", 7);
    std::cout << "Pair: " << svp.first() << ' ' << svp.second() << '\n';
    return 0;
}
```

… должен производить следующий результат:

Pair: Amazing 7

*Подсказка: при вызове конструктора класса **Pair** из конструктора класса **StringValuePair** не забудьте указать, что параметры относятся к классу **Pair**.*

4. Согласно иерархии классов, которая представлена в методичке к уроку 3, от класса **Hand** наследует класс **GenericPlayer**, который обобщенно представляет игрока, ведь у нас будет два типа игроков - человек и компьютер. Создать класс **GenericPlayer**, в который добавить поле **name** - имя игрока. Также добавить 3 метода:
    - **IsHitting()** - чисто виртуальная функция, возвращает информацию, нужна ли игроку еще одна карта.
    - **IsBoosted()** - возвращает bool значение, есть ли у игрока перебор
    - **Bust()** - выводит на экран имя игрока и объявляет, что у него перебор.

Дополнительные материалы

1. Бьерн Страуструп. Программирование. Принципы и практика использования C++.
2. Стивен Прата. Язык программирования C++. Лекции и упражнения.
3. Роберт Лафоре. Объектно-ориентированное программирование в C++.

Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. Онлайн-справочник программиста на C и C++. Перегрузка.
2. Бьерн Страуструп. Программирование. Принципы и практика использования C++.
3. Ральф Джонсон, Ричард Хелм, Эрих Гамма. Приемы объектно-ориентированного программирования. Паттерны проектирования.