Урок 4

Отношения между объектами в C++

Композиция. Агрегация. Ассоциация. Зависимость. Контейнерные классы. Операторы приведения типа. Примеры программ на C++.

Отношения между объектами в C++

Прежде чем изучать основные паттерны программирования, необходимо разобраться в отношениях между классами и объектами в C++. Это позволит понять связи между сущностями при использовании их в паттернах. В данном уроке рассмотрим основные типы отношений:

- композицию;
- агрегацию;
- ассоциацию;
- зависимость.

Эти типы отношений имеют аналогии в реальной жизни. Например:

- квадрат **является** геометрической фигурой;
- автомобиль **имеет** руль;
- программист **использует** клавиатуру;
- цветок **зависит** от растения;
- ученик является **членом** класса;
- мозг существует как **часть** человека.

Здесь мы приводим в пример типы отношений, которые есть и в программировании. Рассмотрим их подробнее.

Композиция

Композиция — это построение сложного объекта из более простых. Например, организм состоит из органов или автомобиль — из двигателя, шасси и других элементов.

Для реализации композиции объект и часть должны иметь следующие отношения:

- часть (член) является составляющей объекта (класса);
- часть (член) может принадлежать только одному объекту (классу) в каждом конкретном случае;
- часть (член) существует под управлением объекта (класса);
- часть (член) не знает о существовании объекта (класса).

Рассмотрим пример на языке программирования.

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

class Human
{
public:
    void Think()
    {
        brain.Think();
    }
private:
    class Brain
    {
    public:
        void Think()
        {
            cout << "Я думаю!" << endl;
        }
    };
    Brain brain;
};

int main()
{
    Human human;
    human.Think();
}
```

В данном примере метод **Think()** можно вызвать, только если есть объект класса **Human**. Иными словами, объект класса **Brain** существует только тогда, когда существует объект класса **Human**. Класс **Brain** находится в private-области класса **Human**, таким образом достигается инкапсуляция данного класса.

Агрегация

Для реализации агрегации целое и его части должны соответствовать следующим отношениям:

- часть (член) является составляющей целого (класса);
- часть (член) может принадлежать более чем одному целому (классу) в каждом конкретном случае;
- часть (член) существует не под управлением целого (класса);
- часть (член) не знает о существовании целого (класса).

Как и в случае с композицией, отношения в агрегации однонаправленные и представлены в формате «часть — целое». Но в отличие от композиции, части могут принадлежать более чем одному целому, и оно не управляет существованием и временем жизни частей. При создании и уничтожении агрегации целое не несет ответственности за создание и уничтожение своих частей.

Приведем наглядный пример. Комната является частью квартиры, следовательно здесь подходит композиция, потому что комната без квартиры существовать не может. А вот мебель не является неотъемлемой частью квартиры. Но в то же время квартира содержит мебель, поэтому следует использовать агрегацию.

Рассмотрим классы **Human** и **Cap**. В реальной жизни кепка является частью одежды человека. И хотя кепка принадлежит человеку, она может принадлежать и другим объектам — например, манекену в

магазине. Человек не несет ответственность за создание и уничтожение кепки. При этом человек знает, что у него есть кепка, но сама она не в курсе, что является частью человека.

```
class Cap {
public:
    string getColor() {
        return color;
    }
private:
    // Пусть все кепки будут красными.
    string color = "красный";
};

class Human
{
public:
    void InspectTheCap () {
        cout << "Моя кепка имеет " << cap.getColor() << " цвет.";
    }
private:
    Cap cap;
};

int main()
{
    Human human;
    human.InspectTheCap();
}
```

В данном примере реализована агрегация.

Рассмотрим более сложный пример с сотрудником и отделом, где он работает:

```
class Worker
{
private:
    string m_name;

public:
    Worker(string name) : m_name(name)
    { }
    string getName() { return m_name; }
};

class Department
{
private:
    // Для простоты добавим только одного работника
    Worker *m_worker;
public:
    Department(Worker *worker = nullptr) : m_worker(worker)
    {  }
};

int main()
{
    // Создаем нового работника
    Worker *worker = new Worker("Anton");
    {
        // Создаем Отдел и передаем Работника в Отдел через параметр
конструктора
        Department department(worker);
    } // department выходит из области видимости и уничтожается здесь

    // worker продолжает существовать
    cout << worker->getName() << " still exists!";
    delete worker;

    return 0;
}
```

Здесь **Работник** создается независимо от **Отдела**, а затем переходит в параметр конструктора класса **Отдела**. Когда **department** уничтожается, указатель **m_worker** уничтожается также, но сам **Работник** не удаляется — он существует до тех пор, пока не будет уничтожен в **main()**.

При решении задачи определяйте тип отношений в зависимости от того, какой из них будет самым простым и соответствующим потребностям вашей программы.

Ассоциация

В ассоциации два несвязанных объекта должны соответствовать следующим отношениям:

- первый объект (член) не связан со вторым объектом (классом);
- первый объект (член) может принадлежать одновременно сразу нескольким объектам (классам);
- первый объект (член) существует не под управлением второго объекта (класса);
- первый объект (член) может знать или не знать о существовании второго объекта (класса).

В отличие от композиции или агрегации, где объект является частью целого, в ассоциации объекты между собой не связаны. Подобно тому, как происходит при агрегации, первый объект может принадлежать нескольким объектам одновременно и не управляться ими. Но в отличие от агрегации, где отношения однонаправленные, в ассоциации они могут быть и двунаправленными — когда оба объекта знают о существовании друг друга.

Ассоциации реализовываются по-разному. Чаще всего для этого используют указатели, где классы указывают на объекты друг друга. Покажем пример, как класс **Водитель** может иметь однонаправленную связь с классом **Автомобиль** без переменной-члена в виде указателя на объект этого класса:

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

class Car
{
private:
    string m_name;
    int m_id;

public:
    Car(string name, int id) : m_name(name), m_id(id)
    { }

    string getName() { return m_name; }
    int getId() { return m_id;  }
};


// Данный класс содержит автомобили и имеет функцию для "выдачи" автомобиля
class CarLot
{
private:
    static Car s_carLot[4];

public:
    // Удаляем конструктор по умолчанию, чтобы нельзя было создать объект
этого класса
    CarLot() = delete;

    static Car* getCar(int id)
    {
        for (int count = 0; count < 4; ++count)
            if (s_carLot[count].getId() == id)
                return &(s_carLot[count]);

        return nullptr;
    }
};

Car CarLot::s_carLot[4] = { Car("Camry", 5), Car("Focus", 14), Car("Vito", 73),
Car("Levante", 58) };

class Driver
{
private:
    string m_name;
    int m_carId; // для связывания классов используется эта переменная

public:
```

```
    Driver(string name, int carId) : m_name(name), m_carId(carId)
    {  }
    string getName() { return m_name; }
    int getCarId() { return m_carId; }

};

int main()
{
    Driver d("Ivan", 14); // Ivan ведет машину с ID 14

    Car *car = CarLot::getCar(d.getCarId()); // Получаем этот Автомобиль из
CarLot

    if (car)
        cout << d.getName() << " is driving a " << car->getName() << '\n';
    else
        cout << d.getName() << " couldn't find his car\n";

    return 0;
}

Car CarLot::s_carLot[4] = { Car("Camry", 5), Car("Focus", 14), Car("Vito", 73),
Car("Levante", 58) };

class Driver
{
private:
    string m_name;
    int m_carId; // для связывания классов используется эта переменная

public:
    Driver(string name, int carId) : m_name(name), m_carId(carId)
    {  }
    string getName() { return m_name; }
    int getCarId() { return m_carId; }

};



int main()
{
    Driver d("Ivan", 14); // Ivan ведет машину с ID 14

    Car *car = CarLot::getCar(d.getCarId()); // Получаем этот Автомобиль из
CarLot

    if (car)
        cout << d.getName() << " is driving a " << car->getName() << '\n';
    else
        cout << d.getName() << " couldn't find his car\n";

    return 0;
}
```

Результат выполнения программы:

Ivan is driving a Focus

В примере выше у нас есть **CarLot** (Гараж), в котором находятся наши автомобили. У Водителя, которому нужен Автомобиль, нет на него указателя, но есть Идентификатор Автомобиля, который он может использовать, чтобы получить Автомобиль из Гаража, когда ему это нужно.

Зависимость

Зависимость возникает, когда один объект обращается к функциональности другого, чтобы выполнить задание. Эти отношения слабее ассоциации, но все же любое изменение объекта, который предоставляет свою функциональность зависимому объекту, может стать причиной сбоя в его работе. Зависимость всегда однонаправленная.

Хорошим примером зависимости, которую вы уже видели много раз, является **std::cout**. Классы используют **std::cout** для вывода в консоль, но не наоборот.

Часто путают, чем зависимость отличается от ассоциации.

В C++ ассоциации — это отношения между двумя классами на уровне классов. То есть первый класс сохраняет прямую или косвенную связь со вторым через переменную-член. Например, в классе Врач есть массив указателей на объекты класса Пациент в виде переменной-члена. Вы всегда можете спросить у Врача, кто его Пациенты. Класс Водитель содержит идентификатор Автомобиля в виде целочисленной переменной-члена. Водитель знает, к чему привязан Автомобиль и как получить к нему доступ.

Зависимости обычно не представлены на уровне классов, то есть зависимый объект не связан со вторым объектом через переменную-член. Зависимый объект создается при необходимости (например, при открытии файла для записи данных) или передается в функцию в качестве параметра.

Контейнерные классы

Контейнерный класс в C++ — это класс, предназначенный для хранения и организации нескольких объектов определенного типа данных (пользовательских или фундаментальных). Есть много разных контейнерных классов, у каждого из которых свои преимущества, недостатки или ограничения. Наиболее часто используемым контейнером в программировании является массив. Хотя в C++ есть стандартные массивы, большинство программистов используют контейнерные классы-массивы — например, **vector**.

Обычно **функциональность классов-контейнеров** в C++ следующая:

- создание пустого контейнера (через конструктор);
- добавление нового объекта в контейнер;
- удаление объекта из контейнера;
- просмотр количества объектов, находящихся на данный момент в контейнере;
- очистка контейнера от всех объектов;
- доступ к сохраненным объектам;
- сортировка объектов/элементов (не всегда).

Функциональность контейнерных классов может быть меньше, чем указанная в этом перечне.

В C++ есть стандартные контейнерные классы. Эта библиотека называется Стандартной библиотекой шаблонов (STL).

Стандартная библиотека шаблонов

STL — это часть Стандартной библиотеки C++, которая содержит набор шаблонов контейнерных классов, алгоритмов и итераторов. Преимущество STL в том, что эти классы можно использовать без необходимости писать и отлаживать их самостоятельно. Плюс вы получаете их эффективные версии.

Контейнеры STL делятся на три основные категории:

- последовательные;
- ассоциативные;
- адаптеры.

Рассмотрим их более подробно.

Последовательные контейнеры (или контейнеры последовательности) — это контейнерные классы, элементы которых находятся в последовательности. Их определяющей характеристикой является то, что вы можете вставить свой элемент куда угодно в контейнере. Наиболее распространенным примером последовательного контейнера является массив: при вставке четырех элементов в массив они будут находиться в таком же порядке, как вы их вставляли.

STL содержит 6 контейнеров последовательности:

std::vector;

std::deque;

std::array;

std::list;

std::forward_list;

std::basic_string.

Класс **vector** рассмотрим более подробно.

Ассоциативные контейнеры — это контейнерные классы, которые автоматически сортируют все свои элементы. По умолчанию ассоциативные контейнеры выполняют сортировку элементов, используя оператор сравнения.

- **set** — это контейнер, в котором хранятся только уникальные элементы, повторения запрещены. Элементы сортируются в соответствии с их значениями.
- **multiset** — это set, но в котором допускаются повторяющиеся элементы.
- **map** — это set, в котором каждый элемент является парой «ключ — значение». Ключ используется для сортировки и индексации данных и должен быть уникальным. А значение — это фактические данные.
- **multimap** — это map, который допускает дублирование ключей. Все ключи отсортированы в порядке возрастания, и вы можете посмотреть значение по ключу.

Адаптеры — это специальные предопределенные контейнерные классы, которые адаптированы для выполнения конкретных заданий. Самое интересное заключается в том, что вы сами можете выбрать, какой последовательный контейнер должен использовать адаптер.

- **stack (стек)** — это контейнерный класс, элементы которого работают по принципу **LIFO** («**L**ast **I**n, **F**irst **O**ut» — «последним пришел, первым ушел»), то есть элементы вставляются (вталкиваются) в конец контейнера и удаляются (выталкиваются) оттуда же (из конца контейнера).
- **queue** (**очередь**) — это контейнерный класс, элементы которого работают по принципу **FIFO** («**F**irst **I**n, **F**irst **O**ut» — «первым пришел, первым ушел»), то есть элементы вставляются (вталкиваются) в конец контейнера, но удаляются (выталкиваются) из начала контейнера.

- **priority_queue** (**очередь с приоритетом**) — это тип очереди, в которой все элементы отсортированы. При вставке элемента он автоматически сортируется. Элемент с наивысшим приоритетом (самый большой) находится в самом начале очереди с приоритетом. Удаление элементов из такой очереди выполняется с самого начала очереди с приоритетом.

Введение в std::vector

Для хранения однотипных данных в C++ используются массивы. Существуют массивы с фиксированной длиной и динамические, длина которых может изменяться по ходу исполнения программы. Динамические массивы очень популярны, так как зачастую мы не знаем заранее, сколько элементов будет в массиве. Для реализации динамических массив в C++ можно использовать операторы **new** и **delete**.

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

int main()
{
    int num;                        // размер массива
    cin >> num;                     // получение от пользователя размера массива
    int *p_array = new int[num]; // Выделение памяти для массива
    for (int i = 0; i < num; i++) {
                                    // Заполнение массива
        cin >> p_array[i];
    }
    // ....
    delete [] p_array;              // очистка памяти
    return 0;
}
```

Но существует более удобная версия динамических массивов — **std::vector**. С его помощью можно создавать массивы, длина которых задается во время выполнения, не используя операторы **new** и **delete**, то есть явно не указывая выделение и освобождение памяти. **std::vector** находится в заголовочном файле **<vector>**.

Создание динамического массива целых чисел следующее:

```
#include <vector>
using namespace std;
// нет необходимости указывать длину при инициализации
vector<int> array;
vector<int> array2 = { 10, 8, 6, 4, 2, 1 };
vector<int> array3 { 10, 8, 6, 4, 2, 1 };
```

Доступ к элементам динамического массива выполняется так же, как и в обычном массиве, — через оператор []:

array2[0] = 3;

Векторы отслеживают свою длину с помощью функции **size()**:

array2.size();

Можно изменить длину вектора с помощью функции **resize()**:

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

void print(vector<int> &a) {
    cout << "The length is: " << a.size() << '\n';

    for (int i=0; i<a.size(); i++)
        cout << a[i] << ' ';

    cout << endl;
}

int main()
{
    vector<int> array { 0, 1, 2, 3 };

    array.resize(7); // изменяем длину array на 7
    print(array);

    array.resize(2);
    print(array);

    return 0;
}
```

Результат выполнения программы:

The length is: 7

0 1 2 3 0 0 0

The length is: 2

0 1

Как видно из результата, можно увеличивать длину вектора, при этом пустые элементы будут инициализироваться начальными значениями (для типа **int** — 0). Или уменьшать — при этом часть элементов будет теряться.

Для добавления нового элемента в вектор существует функция **push_back()**. Для удаления последнего элемента вектора используйте функцию **pop_back()**, а для удаления всех элементов массива — **clear()**. Функция **empty()** проверяет вектор на пустоту.

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

void print(vector<int> &a) {
    cout << "The length is: " << a.size() << '\n';

    for (int i=0; i<a.size(); i++)
        cout << a[i] << ' ';

    cout << endl;
}

int main()
{
    vector<int> array { 0, 1, 2, 3 };

    array.push_back(4);
    print(array);

    array.pop_back();
    print(array);

    array.clear();
    print(array);

    if (array.empty())
        cout << "Vector is empty.\n";
    else
        cout << "Vector is not empty.\n";

    return 0;
}
```

Результат выполнения программы:

The length is: 5

0 1 2 3 4

The length is: 4

0 1 2 3

The length is: 0

Vector is empty.

Итераторы STL

Итератор — это объект, который способен перебирать элементы контейнерного класса, а пользователю при этом не обязательно знать, как реализован этот контейнерный класс. Во многих контейнерах (особенно в списке и ассоциативных контейнерах **set**, **map** и т.д.) итераторы являются основным способом доступа к элементам.

Об итераторе можно думать как об указателе на определенный элемент контейнерного класса с дополнительным набором перегруженных операторов для выполнения четко определенных функций:

- **Оператор *** возвращает элемент, на который в данный момент указывает итератор.

- **Оператор ++** перемещает итератор к следующему элементу контейнера. Большинство итераторов также предоставляют **оператор −−** для перехода к предыдущему элементу.
- **Операторы ==** и **!=** используются для определения того, указывают ли два итератора на один и тот же элемент. Для сравнения значений, на которые указывают два итератора, нужно сначала разыменовать их, а затем использовать оператор == или !=.
- **Оператор =** присваивает итератору новую позицию (обычно начало или конец элементов контейнера). Чтобы присвоить другому объекту значение элемента, на который указывает итератор, нужно сначала разыменовать итератор, а затем использовать оператор =.

Каждый контейнерный класс имеет **4 основных метода для работы с оператором =**:

- **begin()** возвращает итератор, представляющий начало элементов контейнера.
- **end()** возвращает итератор, представляющий элемент, который находится после последнего элемента в контейнере.
- **cbegin()** возвращает константный (только для чтения) итератор, представляющий начало элементов контейнера.
- **cend()** возвращает константный (только для чтения) итератор, представляющий элемент, который находится после последнего элемента в контейнере.

Наконец, все контейнеры предоставляют (как минимум) **два типа итераторов**:

- **container::iterator** — итератор для чтения/записи;
- **container::const_iterator** — итератор только для чтения.

Рассмотрим пример использования итераторов для вектора.

Заполним вектор пятью числами и, с помощью итераторов, выведем значения вектора:

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

int main()
{
    vector<int> myVector;
    for (int count=0; count < 5; ++count)
        myVector.push_back(count);

    vector<int>::const_iterator it; // объявляем итератор только для чтения
    it = myVector.begin(); // присваиваем ему начало вектора
    while (it != myVector.end()) // пока итератор не достигнет конца
    {
        cout << *it << " "; // выводим значение элемента, на который указывает
итератор
        ++it; // и переходим к следующему элементу
    }
    cout << '\n';
}
```

Результат выполнения программы выше:

0 1 2 3 4

Создание контейнерного класса

Создадим контейнерный класс-массив, который будет реализовывать функциональность векторов. Но наш класс-массив будет только целочисленным.

Создадим файл **ArrayInt.h** и в нем объявим класс **ArrayInt**. В этом классе будет две переменные — данные массива (реализовано через указатель) и длина массива. Создадим два конструктора: один для пустого массива, другой для массива с заданной длиной и значениями. Также потребуется деструктор, который выполняет очистку памяти.

```
#ifndef ARRAYINT_H
#define ARRAYINT_H

#include <cassert> // для assert()

class ArrayInt
{
private:
    int m_length;
    int *m_data;

public:
    ArrayInt(): m_length(0), m_data(nullptr)
    { }

    ArrayInt(int length):
    m_length(length)
    {
        assert(length >= 0);

        if (length > 0)
            m_data = new int[length];
        else
            m_data = nullptr;
    }

    ~ArrayInt()
    {
        delete[] m_data;
    }
};

#endif
```

Теперь напишем функцию **erase()**, которая будет выполнять очитку массива и сбрасывать его длину на 0 (аналог функции **clear()** в векторах) и функцию для возврата длины массива (аналог **size()**):

```
    void erase()
    {
        delete[] m_data;

        // Здесь нам нужно указать m_data значение nullptr, чтобы на выходе не
было висячего указателя
        m_data = nullptr;
        m_length = 0;
    }
    int getLength() { return m_length; }
```

Перегрузим оператор индексации [], чтобы иметь доступ к элементам массива:

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

Перегрузка осуществляется через метод класса. Функция перегрузки оператора [] всегда будет принимать один параметр: значение индекса. В случае с **IntArray** нужно, чтобы пользователь просто указал в квадратных скобках индекс для возврата значения элемента по нему. Обратите внимание, что оператор индексации использует возврат по ссылке. Если мы захотим изменить значение элемента массива, то выражение **array[k]** (где **k** — номер элемента массива) будет стоять слева от оператора присваивания =. Это означает, что **array[k]** должно быть **l-value** (переменной с адресом памяти). Поэтому перегруженная функция должна возвращать ссылку на элемент, а не просто его значение.

Напишем функцию, которая реализует возможность изменять размер массива (аналог **resize()**):

```
    // Функция resize изменяет размер массива. Все существующие элементы
сохраняются. Процесс медленный
    void resize(int newLength)
    {
        // Если массив уже нужной длины — return
        if (newLength == m_length)
            return;

        // Если нужно сделать массив пустым — делаем это и затем return
        if (newLength <= 0)
        {
            erase();
            return;
        }

        // Теперь знаем, что newLength >0
        // Выделяем новый массив
        int *data = new int[newLength];

        // Затем нужно разобраться с количеством копируемых элементов в новый
массив
        // Нужно скопировать столько элементов, сколько их есть в меньшем из
массивов
        if (m_length > 0)
        {
            int elementsToCopy = (newLength > m_length) ? m_length : newLength;

            // Поочередно копируем элементы
            for (int index=0; index < elementsToCopy ; ++index)
                data[index] = m_data[index];
        }

        // Удаляем старый массив, так как он нам уже не нужен
        delete[] m_data;

        // И используем вместо старого массива новый! Обратите внимание,
m_data указывает
        // на тот же адрес, на который указывает наш новый динамически
выделенный массив. Поскольку
        // данные были динамически выделенные — они не будут уничтожены, когда
выйдут из области видимости
        m_data = data;
        m_length = newLength;
    }
```

Напишем функцию **insertBefore()** для добавления нового элемента в массив (в реализации эта функция очень похожа на **resize()**):

```
    void insertBefore(int value, int index)
    {
        // Проверка корректности передаваемого индекса
        assert(index >= 0 && index <= m_length);

        // Создаем новый массив на один элемент больше старого массива
        int *data = new int[m_length+1];

        // Копируем все элементы до index-а
        for (int before=0; before < index; ++before)
            data[before] = m_data[before];

        // Вставляем новый элемент в новый массив
        data [index] = value;

        // Копируем все значения после вставляемого элемента
        for (int after=index; after < m_length; ++after)
            data[after+1] = m_data[after];

        // Удаляем старый массив и используем вместо него новый
        delete[] m_data;
        m_data = data;
        ++m_length;
    }
```

Теперь легко реализовать аналог функции **push_back()** в векторах:

```
void push_back(int value) { insertBefore(value, m_length); }
```

Динамическое приведение типов

В C++ есть 5 типов **cast**:

- **C-style_cast**,
- **static_cast**,
- **const_cast**,
- **dynamic_cast**
- **reinterpret_cast**.

Сейчас рассмотрим **dynamic_cast.**

Применяя полиморфизм на практике, вы часто будете сталкиваться с ситуациями, когда у вас есть указатель на родительский класс, но нужно получить доступ к данным, которые есть только в дочернем.

```
#include <iostream>
#include <string>
using namespace std;
class Parent
{
protected:
    string m_name;

public:
    Parent(string name) : m_name(name)
    { }
    virtual ~Parent() {}
};

class Child: public Parent
{
protected:
     string m_ patronymic;

public:
    Child(string name, string patronymic) :
      Parent(name), m_ patronymic (patronymic)
    { }
    const string& getName() { return m_name; }
};

Parent* Create()
{
        return new Child("Alex", "Mike");
}

int main()
{
    Parent *p = Create();

    // как мы выведем имя объекта класса Child здесь, имея лишь один указатель
класса Parent?

    delete p;
    return 0;
}
```

В этой программе функция **Create()** всегда возвращает указатель класса **Parent**, но он может указывать либо на объект класса **Parent**, либо на объект класса **Child** — в этом случае будем вызывать **Child::getName()**?

Один из способов — добавить виртуальную функцию **getName()** в класс **Parent**. Но, используя этот вариант, будем загрязнять класс **Parent** тем, что должно быть заботой только класса **Child**.

C++ позволяет неявно преобразовать указатель класса **Child** в указатель класса **Parent** — фактически, это и делает **Create()**. Эта конвертация называется **приведением к базовому типу,** или **повышающим приведением типа**. Но как конвертировать указатель класса **Parent** обратно в указатель класса **Child**?

В C++ оператор **dynamic_cast** используется именно для этого. Хотя динамическое приведение позволяет выполнять не только конвертацию указателей родительского класса в указатели дочернего класса, это наиболее распространенное применение **dynamic_cast**. Этот процесс называется **приведением к дочернему типу,** или **понижающим приведением типа**. Вот пример его использования:

```
int main()
{
    Parent *p = Create();
    // используем dynamic_cast для конвертации указателя класса Parent в
указатель класса Child
    Child *ch = dynamic_cast<Child*>(p);

    cout << "The name of the Child is: " << ch->getName() << '\n';
    delete p;
    return 0;
}
```

Пример выше работает только из-за того, что указатель **p** на самом деле указывает на объект класса **Child**, поэтому конвертация успешна. Если бы он изначально указывал на объект класса **Parent**, то конвертация была бы невозможна и **dynamic_cast** вернул бы нулевой указатель. Именно поэтому после использования **dynamic_cast** необходимо выполнять проверку на нулевой указатель.

Также обратите внимание на **случаи, в которых понижающее приведение с использованием dynamic_cast не работает**:

1. Наследование типа **private** или **protected**.

2. Классы, которые не объявляют или не наследуют классы с какими-либо виртуальными функциями. Если бы в примере выше мы удалили виртуальный деструктор класса **Parent**, то преобразование через **dynamic_cast** не выполнилось бы.

Хотя в последнем примере мы использовали динамическое приведение с указателем (наиболее распространенная практика), **dynamic_cast** также может использоваться и со ссылками. Поскольку в C++ не существует «нулевой ссылки», то **dynamic_cast** не может вернуть «нулевую ссылку» при сбое. Вместо этого **dynamic_cast** генерирует исключение типа **std::bad_cast** (об исключениях — в следующем уроке).

В общем, лучше использовать виртуальные функции, чем понижающее приведение. Кроме ряда случаев, когда понижающее приведение предпочтительнее:

1. Если вы не можете изменить родительский класс, чтобы добавить в него свою виртуальную функцию (например, если он является частью стандартной библиотеки C++). При этом, чтобы использовать понижающее приведение, в родительском классе должны уже присутствовать виртуальные функции.

2. Если вам нужен доступ к чему-либо, что есть только в дочернем классе — например, к функции доступа.

3. Если добавление виртуальной функции в родительский класс не имеет смысла. Если при этом вам не нужно создавать объект родительского класса, можете использовать чистую виртуальную функцию.

Написание игры Blackjack

В прошлый раз мы создали иерархию классов игры Blackjack. В данном уроке рассмотрим функциональность каждого класса, а также напишем функцию **main()**.

Класс Card

Член класса	Описание
rank m_Rank	Значение карты (туз, двойка, тройка и так далее). **rank** — это перечисление, куда входят все 13 значений
suit m_Suit	Масть карты (трефы, бубны, червы и пики). **suit** — это перечисление, содержащее четыре возможные масти
bool m_IsFaceUp	Указывает, как расположена карта — вверх лицом или рубашкой. Влияет на то, отображается она или нет
int GetValue()	Возвращает значение карты
void Flip()	Переворачивает карту. Может использоваться для того, чтобы перевернуть карту лицом вверх или вниз

Класс Hand

Член класса	Описание
vector<Card*> m_Cards	Коллекция карт. Хранит указатели на объекты типа **Card**
void Add(Card* pCard)	Добавляет карту в руку. Добавляет указатель на объект типа **Card** в вектор **m_Cards**
void Clear()	Очищает руку от карт. Удаляет все указатели из вектора **m_Cards**, устраняя все связанные с ними объекты в куче
int GetTotal()	Возвращает сумму очков карт руки

Класс GenericPlayer

Член класса	Описание
string m_Name	Имя игрока
virtual bool IsHitting() const = 0	Указывает, нужна ли игроку еще одна карта. Чистая виртуальная функция
bool IsBoosted() const	Указывает, что у игрока перебор
void Bust() const	Объявляет, что у игрока перебор

Класс Player:

Член класса	Описание
virtual bool IsHitting() const	Указывает, нужна ли игроку еще одна карта
void Win() const	Объявляет, что игрок выиграл
void Lose() const	Объявляет, что игрок проиграл
void Push() const	Объявляет, что игрок сыграл вничью

Класс House

Член класса	Описание
virtual bool IsHitting() const	Указывает, нужна ли игроку еще одна карта
void FlipFirstCard()	Переворачивает первую карту

Класс Deck

Член класса	Описание
vold Populate()	Создает стандартную колоду из 52 карт
void Shuffle()	Тасует карты
vold Deal (Hand& aHand)	Раздает в руку одну карту
void AddItionalCards (GenericPlayer& aGenerlcPlayer)	Раздает игроку дополнительные карты до тех пор, пока он может и хочет их получать

Класс Game:

Член класса	Описание
Deck m_Deck	Колода карт
House m_House	Рука дилера
vector<Player> m_Players	Группа игроков-людей. Вектор, содержащий объекты типа **Player**
void Play()	Проводит кон игры **Blackjack**

В функции **main()** пользователь должен будет ввести количество игроков и их имена. После этого начнется игровой цикл: создание объекта класса **Game** и вызов метода **play()**. Вот реализация функции **main()**:

```
#include <vector>
using namespace std;
int main()
{
    cout << "\t\tWelcome to Blackjack!\n\n";

    int numPlayers = 0;
    while (numPlayers < 1 || numPlayers > 7)
    {
        cout << "How many players? (1 - 7): ";
        cin >> numPlayers;
    }

    vector<string> names;
    string name;
    for (int i = 0; i < numPlayers; ++i)
    {
        cout << "Enter player name: ";
        cin >> name;
        names.push_back(name);
    }
    cout << endl;

    // игровой цикл
    Game aGame(names);
    char again = 'y';
    while (again != 'n' && again != 'N')
    {
        aGame.Play();
        cout << "\nDo you want to play again? (Y/N): ";
        cin >> again;
    }

    return 0;
}
```

В функции **main()** имена игроков помещаются в вектор, поскольку мы не знаем заранее, сколько человек будет играть. Этот вектор передается в качестве параметра конструктору класса **Game**.

Практические задания

1. Добавить в контейнерный класс, который был написан в этом уроке, методы:
   - для удаления последнего элемента массива (аналог функции pop_back() в векторах)
   - для удаления первого элемента массива (аналог pop_front() в векторах)
   - для сортировки массива
   - для вывода на экран элементов.
2. Дан вектор чисел, требуется выяснить, сколько среди них различных. Постараться использовать максимально быстрый алгоритм.
3. Реализовать класс Hand, который представляет собой коллекцию карт. В классе будет одно поле: вектор указателей карт (удобно использовать вектор, т.к. это по сути динамический

массив, а тип его элементов должен быть - указатель на объекты класса Card). Также в классе Hand должно быть 3 метода:

- метод Add, который добавляет в коллекцию карт новую карту, соответственно он принимает в качестве параметра указатель на новую карту
- метод Clear, который очищает руку от карт
- метод GetValue, который возвращает сумму очков карт руки (здесь предусмотреть возможность того, что туз может быть равен 11).

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. Объектно-ориентированный анализ и проектирование с примерами приложений на C++. Отношения между классами.
2. Бьерн Страуструп. Программирование. Принципы и практика использования C++.
3. Ральф Джонсон, Ричард Хелм, Эрих Гамма. Приемы объектно-ориентированного программирования. Паттерны проектирования.