

Глава 12

Динамическая идентификация и приведение типов.

В этой главе рассказывается о двух сравнительно новых инструментах C++: динамической идентификации типа (Run-Time Type Identification, RTTI) и новых, более совершенных операторах приведения типов (casting operators). Динамическая идентификация типа даёт возможность определить тип объекта во время выполнения программы. Новые операторы приведения типов предоставляют более безопасные и управляемые способы выполнения операций приведения типов, по сравнению с существовавшими ранее. Как вы увидите в дальнейшем, один из операторов приведения типов, а именно оператор **dynamic_cast**, относится непосредственно к RTTI, поэтому имело смысл объединить эти две темы в одной главе.

12.1. Понятие о динамической идентификации типа.

Поскольку динамическая идентификация типа не характерна для языков программирования, в которых не поддерживается полиморфизм (например, C), это понятие может оказаться для вас неизвестным. В языках, в которых не поддерживается полиморфизм, информация о типе объекта во время выполнения программы просто не нужна, так как тип каждого объекта известен уже на этапе компиляции программы (вернее даже тогда, когда программа ещё пишется). С другой стороны, в языках, поддерживающих полиморфизм (таких, как C++), возможны ситуации, в которых тип объекта на этапе компиляции не известен, поскольку до выполнения программы не определена точная природа объекта. Как вы знаете, в C++ полиморфизм реализуется через иерархии классов, виртуальные функции и указатели базовых классов. При таком подходе указатель базового класса может использоваться либо для указания на объект базового класса, либо для указания на *объект любого класса, производного от этого базового*. Следовательно, не всегда есть возможность заранее узнать тип объекта, на который будет указывать указатель базового класса в каждый данный момент времени. В таких случаях определение типа объекта должно происходить во время выполнения программы, а для этого служит механизм динамической идентификации типа.

Информацию о типе объекта получают с помощью оператора **typeid**. Для использования оператора **typeid** в программу следует включить заголовок **<typeinfo>**. Ниже представлена основная форма оператора **typeid**:

typeid (объект)

Здесь *объект* – это тот объект, информацию о типе которого необходимо получить. Оператор **typeid** возвращает ссылку на объект типа **type_info**, который и описывает тип объекта *объект*. В классе **type_info** определены следующие открытые члены:

```
bool operator==(const type_info &объект);
bool operator!=(const type_info &объект);
bool before(const type_info &объект);
const char *name();
```

Сравнение типов обеспечивают перегруженные операторы `==` и `!=`. Функция `before()` возвращает истину, если вызывающий объект в порядке сортировки расположен раньше объекта заданного в качестве параметра. (Эта функция обычно предназначена только для внутреннего использования. Её возвращаемое значение вряд ли может пригодиться при операциях с наследованием или иерархиями классов.) Функция `name()` возвращает указатель на имя типа.

Хотя оператор `typeid` позволяет получать типы разных объектов, наиболее полезен он будет, если в качестве его аргумента задать указатель полиморфного базового класса. В этом случае оператор автоматически возвращает тип реального объекта, на который указывает указатель. Этим объектом может быть как объект базового класса, так и объект любого класса, производного от этого базового. (Вспомните, указатель базового класса может указывать либо на объект базового класса, либо на объект любого класса, производного от этого базового.) Таким образом, с помощью оператора `typeid` указана ссылка на объект полиморфного класса, оператор возвращает тип реального объекта, на который имеется ссылка. Этим объектом, так же как и в случае с указателем, может быть объект производного класса. Когда оператор `typeid` применяют к не полиморфному классу, получают указатель или ссылку базового типа.

Ниже представлена вторая форма оператора `typeid`, в которой в качестве аргумента указывают имя типа:

`typeid (имя_типа)`

Обычно с помощью данной формы оператора `typeid` получают объект типа `type_info`, который можно использовать в инструкции сравнения типов.

Поскольку оператор `typeid` чаще всего применяют к разыменованному указателю (т.е. к указателю, к которому уже был применён оператор `*`). Для обработки положения, когда разыменованный указатель равен нулю, была придумана специальная исключительная ситуация `bad_typeid`, которую в этом случае возбуждает оператор `typeid`.

Динамическая идентификация типа используется далеко не в каждой программе. Тем не менее, если вы работаете с полиморфными типами данных, она позволяет в самых разнообразных ситуациях определять типы обрабатываемых объектов.

Примеры

1. В следующей программе демонстрируется использование оператора `typeid`. Сначала с помощью этого оператора мы получаем информацию об одном из встроенных типов данных C++ - типе `int`. Затем оператор `typeid` даёт нам возможность вывести на экран типы объектов, на которые указывает указатель `p`, являющийся указателем базового класса `BaseClass`.

```
// Пример использования оператора typeid
#include <iostream>
#include <typeinfo>
using namespace std;

class BaseClass {
    virtual void f() {} // делаем класс BaseClass полиморфным
    // ...
};
```

```

class Derived1 : BaseClass {
    // ...
};

class Derived2 : BaseClass {
    // ...
};

int main()
{
    int i;
    BaseClass *p, baseob;
    Derived1 ob1;
    Derived2 ob2;

    // Вывод на экран встроенного типа данных
    cout << "Тип переменной i - это ";
    cout << typeid(i).name() << endl;

    // Обработка полиморфных типов
    p = &baseob;
    cout << "Указатель p указывает на объект типа ";
    cout << typeid(*p).name() << endl;

    p = &ob1;
    cout << "Указатель p указывает на объект типа ";
    cout << typeid(*p).name() << endl;

    p = &ob2;
    cout << "Указатель p указывает на объект типа ";
    cout << typeid(*p).name() << endl;

    return 0;
}

```

Программа выводит на экран следующую информацию:

```

Тип переменной i – это int
Указатель p указывает на объект типа BaseClass
Указатель p указывает на объект типа Derived1
Указатель p указывает на объект типа Derived2

```

Как уже отмечалось, когда в качестве аргумента оператора **typeid** задан указатель полиморфного базового класса, реальный тип объекта, на который указывает указатель определяется во время выполнения программы, что очевидно по выводимой на экран информации. В качестве эксперимента прокомментируйте виртуальную функцию **f()** в определении базового класса **BaseClass** и посмотрите, что получится.

2. Ранее уже говорилось, что когда в качестве аргумента оператора **typeid** указана ссылка полиморфного базового класса, возвращаемым типом будет тип реального объекта, на который дана ссылка. Чаще всего это свойство используется в ситуациях, когда объекты передаются функциям по ссылке. Например, в следующей программе в объявлении функции **WhatType()** объект типа **BaseClass** задан параметром-ссылкой. Это означает, что функции **WhatType()** можно передавать ссылки на объекты типа **BaseClass** или типов, производных от класса **BaseClass**. Если в операторе **typeid** задать такой параметр, то он возвратит тип реального объекта.

```

// Использование оператора typeid со ссылкой в качестве аргумента
#include <iostream>

```

```

#include <typeinfo>
using namespace std;

class BaseClass {
    virtual void f() {} // делаем класс BaseClass полиморфным
    // ...
};

class Derived1 : BaseClass {
    // ...
};

class Derived2 : BaseClass {
    // ...
};

// Задание ссылки в качестве параметра функции
void WhatType(BaseClass &ob)
{
    cout << "ob - это ссылка на объект типа ";
    cout << typeid(ob).name() << endl;
}

int main()
{
    int i;
    BaseClass baseob;
    Derived1 ob1;
    Derived2 ob2;

    WhatType(baseob)
    WhatType(ob1)
    WhatType(ob2)

    return 0;
}

```

Программа выводит на экран следующую информацию:

```

ob – это ссылка на объект типа BaseClass
ob – это ссылка на объект типа Derived1
ob – это ссылка на объект типа Derived2

```

3. Хотя получение типа объекта в некоторых ситуациях оказывается весьма полезным, часто бывает необходимо узнать, соответствуют ли друг другу типы нескольких объектов. Это легко сделать, зная, что объект типа **type_info**, возвращаемый оператором **typeid**, перегружает операторы **==** и **!=**. В представленной ниже программе показано использование этих операторов.

```

// Использование операторов == и != с оператором typeid
#include <iostream>
#include <typeinfo>
using namespace std;

class X {
    virtual void f() {}
};

class Y {
    virtual void f() {}
};

```

```

int main()
{
    X x1, x2;
    Y y1;

    if(typeid(x1) == (typeid(x2)))
        cout << "Тип объектов x1 и x2 одинаков\n";
    else
        cout << "Тип объектов x1 и x2 не одинаков\n";

    if(typeid(x1) != (typeid(y1)))
        cout << "Тип объектов x1 и y1 не одинаков\n";
    else
        cout << "Тип объектов x1 и y1 одинаков\n";

    return 0;
}

```

Программа выводит на экран следующую информацию:

```

Тип объектов x1 и x2 одинаков
Тип объектов x1 и y1 не одинаков

```

4. Хотя в предыдущих примерах и были показаны некоторые приёмы работы с оператором **typeid**, главных его достоинств мы не увидели, поскольку типы объектов были известны уже на этапе компиляции программы. В следующем примере этот пробел восполнен. В программе определена простая иерархия классов, предназначенных для рисования на экране разного рода геометрических фигур. На вершине иерархии находится абстрактный класс **Shape**. Его наследуют четыре класса: **Line**, **Square**, **Rectangle** и **Nullshape**. Функция **generator()** генерирует объект и возвращает указатель на него. (Функцию, предназначенную для создания объектов, иногда называют *фабрикой объектов*.) То, какой именно объект создаётся, определяет генератор случайных чисел **rand()**. В функции **main()** реализован вывод получающихся объектов разных типов на экран, исключая объекты типа **Nullshape**, у которых нет какой бы то ни было формы. Поскольку объекты возникают случайно, заранее неизвестно, какой объект будет создан следующим. Следовательно, для определения типа создаваемых объектов требуется динамическая идентификация типа.

```

#include <iostream>
#include <cstdlib>
#include <typeinfo>
using namespace std;

class Shape {
public:
    virtual void example() = 0;
};

class Rectangle : public Shape {
public:
    void example() {
        cout << "*****\n* *\n* *\n*****\n";
    }
};

class Triangle : public Shape {
public:
    void example() {
        cout << "*\n* *\n* *\n*****\n";
    }
}

```

```

};

class Line : public Shape {
public:
    void example() {
        cout << "*****\n";
    }
};

class NullShape : public Shape {
public:
    void example() {
    }
};

// Фабрика производных от класса Shape объектов
Shape *genegator()
{
    switch(rand() % 4) {
        case 0:
            return new Line;
        case 1:
            return new Rectangle;
        case 2:
            return new Triangle;
        case 3:
            return new NullShape;
    }
    return NULL;
}

int main()
{
    int i;
    Shape *p;

    for(i=0; i<10; i++) {
        p = genegator(); // создание следующего объекта

        cout << typeid(*p).name() << endl;

        // рисует объект, если он не типа NullShape
        if(typeid(*p) != typeid(NullShape))
            p->example();
    }

    return 0;
}

```

Программа выводит на экран следующее:

```

+++++
+++++
+++++

```

5. Оператор **typeid** может работать с классами-шаблонами. Например, рассмотрим следующую программу. В ней для хранения некоторых значений создаётся иерархия классов-шаблонов. Виртуальная функция **get_val()** возвращает определённое в каждом классе значение. Для класса **Num** это значение соответствует самому числу. Для класса **Square** – это квадрат числа. Для класса **Sqr_root** – это квадратный корень числа. Объекты,

производные от класса **Num**, генерирует функция **generator()**. С помощью оператора **typeid** определяется тип генерируемых объектов.

```
// Использование оператора typeid с шаблонами
#include <iostream>
#include <cstdlib>
#include <cmath>
#include <typeinfo>
using namespace std;

template <class T> class Num {
public:
    T x;
    Num(T i) { x = i; }
    virtual T get_val() { return x; }
};

template <class T>
class Squary : public Num<T> {
public:
    Squary(T i) : Num<T>(i) {}
    T get_val() { return x*x; }
};

template <class T>
class Sqr_root : public Num<T> {
public:
    Sqr_root(T i) : Num<T>(i) {}
    T get_val() { return sqrt((double) x); }
};

// Фабрика производных от класса Num объектов
Num<double> *generator()
{
    switch(rand() % 2) {
        case 0: return new Squary<double> (rand() % 100);
        case 1: return new Sqr_root<double> (rand() % 100);
    }
    return NULL;
}

int main()
{
    Num<double> ob1(10), *p1;
    Squary<double> ob2(100.0);
    Sqr_root<double> ob3(999.2);
    int i;

    cout << typeid(ob1).name() << endl;
    cout << typeid(ob2).name() << endl;
    cout << typeid(ob3).name() << endl;

    if(typeid(ob2) == typeid(Squary<double>))
        cout << "is Squary<double>\n";

    p1 = &ob2;

    if(typeid(*p1) != typeid(ob1))
        cout << "Значение равно: " << p1->get_val();
    cout << "\n\n";

    cout << "Теперь генерируем объекты\n";
}
```

```

for(i=0; i<10; i++) {
    p1 = generator(); // получение следующего объекта

    if(typeid(*p1) == typeid(Squary<double>))
        cout << "Квадрат объекта: ";
    if(typeid(*p1) == typeid(Sqr_root<double>))
        cout << "Квадратный корень объекта: ";

    cout << "Значение равно: " << p1->get_val();
    cout << endl;
}

return 0;
}

```

Программа выводит на экран следующее:

```

+++++
+++++
+++++

```

12.2. Оператор `dynamic_cast`.

Хотя в C++ продолжают поддерживаться характерные для языка C операторы приведения типов, имеется и несколько новых. Это операторы `dynamic_cast`, `const_cast`, `reinterpret_cast` и `static_cast`. Поскольку оператор `dynamic_cast` имеет непосредственное отношение к динамической идентификации типа, мы рассмотрим его первым. Остальные операторы приведения типов рассматриваются в следующем разделе.

Оператор `dynamic_cast` реализует приведение типов в динамическом режиме, что позволяет контролировать правильность этой операции во время работы программы. Если при выполнении оператора `dynamic_cast` приведения типа не произошло, будет выдана ошибка приведения типов. Ниже представлена основная форма оператора `dynamic_cast`:

`dynamic_cast <целевой_тип> (выражение)`

Здесь *целевой_тип* – это тип, которым должен стать тип параметра *выражение* после выполнения операции приведения типов. Целевой тип должен быть типом указателя или ссылки и результат выполнения параметра *выражение* тоже должен стать указателем или ссылкой. Таким образом, оператор `dynamic_cast` используется для приведения типа одного указателя к типу другого или типа одной ссылки к типу другой.

Основное назначение оператора `dynamic_cast` заключается в реализации операции приведения полиморфных типов. Например, пусть дано два полиморфных класса **V** и **D**, причём класс **D** является производным от класса **V**, тогда оператор `dynamic_cast` всегда может привести тип указателя **D*** к типу указателя **V***. Это возможно потому, что указатель базового класса всегда может указывать на объект производного класса. Оператор `dynamic_cast` может также привести тип указателя **V*** к типу указателя **D***, но только в том случае, если объект, на который указывает указатель, действительно является объектом типа **D**. Как правило, оператор `dynamic_cast` выполняется успешно, когда указатель (или ссылка) после приведения типов становится указателем (или ссылкой) либо на объект целевого типа, либо на объект производного от целевого типа. В противном случае приведения типов не происходит. При неудачной попытке приведения типов

результатом выполнения оператора **dynamic_cast** является нуль, если в операции использовались указатели. Если же в операции использовались ссылки, возбуждается исключительная ситуация **bad_cast**/

Рассмотрим простой пример. Предположим, что **Base** - это базовый класс, а **Derived** – это класс, производный от класса **Base**.

```
Base *bp, b_ob;
Derived *dp, d_ob;

bp = &d_ob;    // указатель базового класса
               // указывает на объект производного класса
dp = dynamic_cast<Derived *>(bp)
if (!dp) cout << "Приведение типов прошло успешно";
```

Здесь приведение типа указателя **bp** базового класса к типу указателя **dp** производного класса прошло успешно, поскольку указатель **bp** на самом деле указывает на объект производного класса **Derived**. Таким образом, после выполнения этого фрагмента программы на экране появится сообщение **Приведение типов прошло успешно**. Однако в следующем фрагменте операция приведения типов заканчивается неудачей, поскольку указатель **bp** указывает на объект базового класса **Base**, а приводить тип объекта базового класса к типу объекта производного класса неправильно.

```
bp = &b_ob;    // указатель базового класса
               // указывает на объект базового класса
dp = dynamic_cast<Derived *>(bp)
if (!dp) cout << "Приведения типов не произошло";
```

Поскольку попытка приведения типов закончилась неудачей, на экран будет выведено сообщение **Приведения типов не произошло**.

Оператор **dynamic_cast** в некоторых случаях можно использовать вместо оператора **typeid**. Например, опять предположим, что **Base** – это базовый класс, а **Derived** – это класс, производный от класса **Base**. В следующем фрагменте указателю **dp** присваивается адрес объекта, на который указывает указатель **bp**, но только в случае, если это действительно объект класса **Derived**.

```
Base *bp;
Derived *dp;
// ...
if (typeid(bp) == typeid(Derived)) dp = (Derived *) bp;
```

В данном примере для фактического выполнения операции приведения типов используется стиль языка C. Это безопасней, поскольку инструкция **if** с помощью оператора **typeid** проверяет правильность выполнения операции ещё до того, как она действительно происходит. Тем не менее для этого есть более короткий путь. Оператор **typeid** с инструкцией **if** можно заменить оператором **dynamic_cast**:

```
dp = dynamic_cast<Derived *>(bp)
```

Поскольку оператор **dynamic_cast** заканчивается успешно только в том случае, если объект, к которому применяется операция приведения типов, является либо объектом целевого типа, либо объектом производного от целевого типа, то после выполнения приведенной выше инструкции указатель **dp** будет либо нулевым, либо указателем на объект типа **Derived**. Таким образом, оператор **dynamic_cast** успешно завершается только

при правильном приведении типов, а это значит, что в определённых ситуациях он может упростить логику программы.

Примеры

1. В следующей программе продемонстрировано использование оператора **dynamic_cast**:

```
// Использование оператора dynamic_cast
#include <iostream>
using namespace std;

class Base {
public:
    virtual void f() { cout << "Внутри класса Base\n"; }
};

class Derived : public Base {
public:
    void f() { cout << "Внутри класса Derived\n"; }
};

int main()
{
    Base *bp, b_ob;
    Derived *dp, d_ob;

    dp = dynamic_cast<Derived *> (&d_ob);
    if(dp) {
        cout << "Тип Derived * к типу Derived * приведен успешно\n";
        dp->f();
    } else
        cout << "Ошибка\n";

    cout << endl;

    bp = dynamic_cast<Base *> (&d_ob);
    if(bp) {
        cout << "Тип Derived * к типу Base * приведен успешно\n";
        bp->f();
    } else
        cout << "Ошибка\n";

    cout << endl;

    bp = dynamic_cast<Base *> (&b_ob);
    if(bp) {
        cout << "Тип Base * к типу Base * приведен успешно\n";
        bp->f();
    } else
        cout << "Ошибка\n";

    cout << endl;

    dp = dynamic_cast<Derived *> (&b_ob);
    if(dp) {
        cout << "Ошибка\n";
    } else
        cout << "Тип Base * к типу Derived * не приведен\n";

    cout << endl;
}
```

```

bp = &d_ob; // bp указывает на объект типа Derived
dp = dynamic_cast<Derived*>(bp);
if(dp) {
    cout << "Указатель bp к типу Derived * приведен успешно\n" <<
        "поскольку bp в действительности указывает\n" <<
        "на объект типа Derived\n";
    dp->f();
} else
    cout << "Ошибка\n";

cout << endl;

bp = &b_ob; // bp указывает на объект типа Base
dp = dynamic_cast<Derived*>(bp);
if(dp)
    cout << "Ошибка\n";
else {
    cout << "Указатель bp к типу Derived * не приведен\n" <<
        "поскольку bp в действительности указывает\n" <<
        "на объект типа Base\n";
}

cout << endl;

dp = &d_ob; // dp указывает на объект типа Derived
bp = dynamic_cast<Base*>(dp);
if(bp) {
    cout << "Указатель dp к типу Base * приведен успешно\n" <<
        bp->f();
} else
    cout << "Ошибка\n";

return 0;
}

```

Программа выводит на экран следующее:

```

+++++
+++++
+++++

```

2. В следующем примере показано, как оператор **typeid** можно заменить оператором **dynamic_cast**.

```

// Использование оператора dynamic_cast для замены оператора typeid
#include <iostream>
#include <typeinfo>
using namespace std;

class Base {
public:
    virtual void f() {}
};

class Derived : public Base {
public:
    void derivedOnly() {
        cout << "Это объект класса Derived\n";
    }
};

```

```

int main()
{
    Base *bp, b_ob;
    Derived *dp, d_ob;

    // *****
    // использование оператора typeid
    // *****
    bp = &b_ob;
    if(typeid(*bp) == typeid(Derived)) {
        dp = (Derived *) bp;
        dp->derivedOnly();
    } else
        cout << "Тип Base к типу Derived не приведен\n";

    bp = &d_ob;
    if(typeid(*bp) == typeid(Derived)) {
        dp = (Derived *) bp;
        dp->derivedOnly();
    } else
        cout << "Ошибка, приведение типов должно работать!\n";

    // *****
    // использование оператора dynamic_cast
    // *****
    bp = &b_ob;
    dp = dynamic_cast<Derived *>(bp);
    if(dp) dp->derivedOnly();
    else
        cout << "Тип Base к типу Derived не приведен\n";

    bp = &d_ob;
    dp = dynamic_cast<Derived *>(bp);
    if(dp) dp->derivedOnly();
    else
        cout << "Ошибка, приведение типов должно работать!\n";

    return 0;
}

```

Как видите, использование **dynamic_cast** делает проще логику приведения типа указателя базового класса к типу указателя производного класса. После выполнения программы на экран будет выведена следующая информация:

```

+++++
+++++
+++++

```

3. Оператор **dynamic_cast**, как и оператор **typeid**, можно использовать с классами-шаблонами. Например, в следующем примере представлен переработанный класс-шаблон из примера 5 раздела 12.1. Здесь тип объекта, возвращаемый функцией **generator()**, определяется с помощью оператора **dynamic_cast**.

```

// Использование оператора dynamic_cast с шаблонами
#include <iostream>
#include <cstdlib>
#include <cmath>

```

```

#include <typeinfo>
using namespace std;

template <class T> class Num {
public:
    T x;
    Num(T i) { x = i; }
    virtual T get_val() { return x; }
};

template <class T>
class Squary : public Num<T> {
public:
    Squary(T i) : Num<T>(i) {}
    T get_val() { return x*x; }
};

template <class T>
class Sqr_root : public Num<T> {
public:
    Sqr_root(T i) : Num<T>(i) {}
    T get_val() { return sqrt((double) x); }
};

// Фабрика производных от класса Num объектов
Num<double> *generator()
{
    switch(rand() % 2) {
        case 0: return new Squary<double> (rand() % 100);
        case 1: return new Sqr_root<double> (rand() % 100);
    }
    return NULL;
}

int main()
{
    Num<double> ob1(10), *p1;
    Squary<double> ob2(100.0), *p2;
    Sqr_root<double> ob3(999.2), *p3;
    int i;

    cout << "Генерируем несколько объектов\n";
    for(i=0; i<10; i++) {
        p1 = generator();

        p2 = dynamic_cast<Squary<double> *> (p1);
        if(p2) cout << "Квадрат объекта: ";

        p3 = dynamic_cast<Sqr_root<double> *> (p1);
        if(p3) cout << "Квадратный корень объекта: ";

        cout << "Значение равно: " << p1->get_val();
        cout << endl;
    }

    return 0;
}

```

12.2. Операторы `const_cast`, `reinterpret_cast` и `static_cast`.

Хотя оператор **dynamic_cast** самый полезный из новых операторов приведения типов, кроме него программистам доступны ещё три. Ниже представлены их основные формы:

```
const_cast <целевой_тип> (выражение)
reinterpret_cast <целевой_тип> (выражение)
static_cast <целевой_тип> (выражение)
```

Здесь *целевой_тип* – это тип, которым должен стать тип параметра *выражение* после выполнения операции приведения типов. Как правило, указанные операторы обеспечивают более безопасный и интуитивно понятный способ выполнения некоторых видов операций преобразования, чем оператор приведения типов, более характерный для языка С.

Оператор **const_cast** при выполнении операции приведения типов используется для явной подмены атрибутов **const** (постоянный) и/или **volatile** (переменный). Целевой тип должен совпадать с исходным типом, за исключением изменения его атрибутов **const** или **volatile**. Обычно с помощью оператора **const_cast** значение лишают атрибута **const**.

Оператор **static_cast** предназначен для выполнения операций приведения типов над объектами не полиморфных классов. Например, его можно использовать для приведения типа указателя базового класса к типу указателя производного класса. Кроме этого, он подойдёт и для выполнения любой стандартной операции преобразования, но только не в динамическом режиме (т.е. не во время выполнения программы).

Оператор **reinterpret_cast** даёт возможность преобразовать указатель одного типа в указатель совершенно другого типа. Он также позволяет приводить указатель к типу целого и целое к типу указателя. Оператор **reinterpret_cast** следует использовать для выполнения операции приведения внутренне несовместимых типов указателей.

Атрибута **const** объект можно лишить только с помощью оператора **reinterpret_cast**. С помощью оператора **dynamic_cast**, **static_cast** или **reinterpret_cast** этого сделать нельзя.

Примеры

1. В следующей программе демонстрируется использование оператора **reinterpret_cast**.

```
// Пример использования оператора reinterpret_cast
#include <iostream>
using namespace std;

int main()
{
    int i;
    char *p = "Это строка";

    // приведение типа указателя к типу целого
    i = reinterpret_cast<int>(p);

    cout << i;

    return 0;
}
```

В данной программе с помощью оператора **reinterpret_cast** указатель на строку превращён в целое. Это фундаментальное преобразование типа и оно хорошо отражает возможности оператора **reinterpret_cast**.

2. В следующей программе демонстрируется оператор **const_cast**.

```
// Пример использования оператора const_cast
#include <iostream>
using namespace std;

void f(const int *p)
{
    int *v;

    // преобразование типа лишает указатель p атрибута const
    v = const_cast<int *>(p);

    *v = 100; // теперь указатель v может изменить объект
}

int main()
{
    int x = 99;

    cout << "Объект x перед вызовом функции равен: " << x << endl;
    f(&x);
    cout << "Объект x после вызова функции равен: " << x << endl;

    return 0;
}
```

Ниже представлен результат выполнения программы:

```
Объект x перед вызовом функции равен: 99
Объект x перед вызовом функции равен: 100
```

Как видите, несмотря на то что параметром функции **f()** задан постоянный указатель, вызов этой функции с объектом **x** в качестве параметра изменил значение объекта.

3. Оператор **static_cast**, по существу, предназначен для замены прежнего оператора приведения типов. Он просто выполняет операцию приведения типов над объектами не полиморфных классов. Например, в следующей программе тип **float** приводится к типу **int**.

```
// Пример использования оператора static_cast
#include <iostream>
using namespace std;

int main()
{
    int i;
    float f;

    f = 199.22;

    i = static_cast<int>(f);

    cout << i;

    return 0;
}
```