Динамическая идентификация типа

В языке C++ для поддержки объектно-ориентированного программирования используется **динамическая идентификация типа (RTTI – Run-Time Type Identification)** и четыре дополнительных **оператора приведения типов** (плюс 5-й оператор приведения типов cast из C).

Система RTTI позволяет идентифицировать тип объекта при выполнении программы. Дополнительные операторы приведения типов обеспечивают более безопасный способ приведения. Поскольку один из этих операторов приведения (оператор dynamic_cast) непосредственно связан с системой динамической идентификации типов имеет смысл рассмотреть их вместе.

Механизм динамической идентификации типа состоит из трех составляющих:

- оператора динамического преобразования типа dynamic_cast<>;
- оператора идентификации точного типа объекта typeid();
- класса type_info.

Динамическая идентификация типов не характерна для таких неполиморфных языков, как язык C. В этих языках нет необходимости определять тип объекта при выполнении программы, поскольку тип каждого объекта известен еще на этапе компиляции. Однако в полиморфных языках, таких как C++, возникают ситуации, в которых тип объекта на этапе компиляции неизвестен, поскольку природа этого объекта уточняется только в ходе выполнения программы. Язык C++ реализует полиморфизм с помощью иерархий классов, виртуальных функций и указателей на объекты базового класса. Поскольку указатели на объекты базового класса могут ссылаться и на объекты производных классов, не всегда можно предсказать, на объект какого типа они будут ссылаться в тот или иной момент. Эту идентификацию приходится осуществлять в ходе выполнения программы.

Динамическая идентификация типа

Для идентификации типа объекта используется оператор typeid, определенный в заголовке #include <typeinfo>.

Его формат: typeid (object)

Здесь, параметр object является объектом, тип которого мы хотим идентифицировать. Он может иметь любой тип, в том числе встроенный или определенный пользователем.

Оператор typeid возвращает ссылку на объект типа type_info, описывающий тип объекта. Стандарт "ISO/IEC 14882:2003(E), Programming languages - C++" определяет класс type_info так:

```
class type_info
 {
  public:
    virtual ?type_info();
      bool operator == (const type_info& rhs) const;
      bool operator != (const type_info& rhs) const;
      bool before(const type_info& rhs) const;
     const char* name() const;
  private:
    type_info(const type_info& rhs);
    type_info& operator=(const type_info& rhs);
 };
```

Как видно, объекты типа type_info невозможно ни создать, ни скопировать – конструкторы и операция присваивания закрыты. Объекты типа type_info можно сравнивать между собой, используя перегруженные операторы "==" и "!=", – и это основные операции, которые используются совместно с оператором typeid(); функция name() возвращает указатель на имя заданного типа.

Метод before() позволяет сортировать информацию о типе type_info. Нет никакой связи между отношениями упорядочения, определяемыми before(), и отношениями наследования. Если вызывающий объект предшествует объекту, использованному как параметр, функция before() возвращает значение true.

Динамическая идентификация типа

Пример использования typeid

```
#include <iostream>
#include <typeinfo>
using namespace std;
  class myclass1 {   /* ... */ };
  class myclass2 {  /* … */ };
int main()
{   int i, j;   float f;  char *p;
     myclass1 ob1;  myclass2 ob2;
cout <<"Тип объекта i: "<< typeid(i).name();
  cout << endl;
cout <<"Тип объекта f: "<< typeid(f).name();
  cout << endl;
cout <<"Тип объекта p: "<< typeid(p).name();
  cout << endl;
cout <<"Тип объекта ob1: "<< typeid(ob1).name();
  cout << endl;
cout <<"Тип объекта ob2: "<< typeid(ob2).name();
  cout << "\n\n";
if(typeid(i) == typeid(j))
  cout <<"Типы объектов i и j совпадают\n";
if(typeid(i) != typeid(f))
  cout <<"Типы объектов i и f не совпадают\n";
if(typeid(ob1) != typeid(ob2))
  cout <<"ob1 и ob2  имеют разные типы\n";
return 0;  }
```

Результаты работы этой программы приведены ниже.

- Тип объекта i: int
- Тип объекта f: float
- Тип объекта p: char *
- Тип объекта ob1: class myclass1
- Тип объекта ob2: class myclass2

- Типы объектов i и j совпадают
- Типы объектов i и f не совпадают
- Объекты ob1 и ob2 имеют разные типы

Динамическая идентификация типа

Пример применения typeid к иерархии полиморфных классов

Самое важное свойство оператора typeid проявляется, когда он применяется к указателю на объект базового класса. В этом случае он автоматически возвращает тип реального объекта, на который ссылается указатель, причем этот объект может быть экземпляром как базового, так и производного классов. (Напомним, что указатель на объект базового класса может ссылаться на объекты производного класса.)

Т.о, в ходе выполнения программы, используя оператор typeid, можно идентифицировать тип объекта, на который ссылается указатель базового класса.

Результаты работы этой программы приведены ниже.

- **Указатель p ссылается на объект типа class Mammal,**
- **Указатель p ссылается на объект типа class Cat,**
- **Указатель p ссылается на объект типа class Platypus.**

```
#include <iostream>
#include <typeinfo>
using namespace std;
 class Mammal {public: //класс Mammal – полиморфный
   virtual  bool  lays_eggs() { return false; }
   // ...    };
 class Cat: public Mammal {  public:    /* ... */  };
 class Platypus: public Mammal {  public:
  bool  lays_eggs() { return true; } /* ... */  };

int main()
{   Mammal *p, AnyMammal;
     Cat cat;
     Platypus platypus;
  p = &AnyMammal;
  cout << "Указатель p ссылается на объект типа ";
  cout << typeid(*p).name() << endl;
  p = &cat;
    cout << "Указатель p ссылается на объект типа ";
    cout << typeid(*p).name() << endl;
  p = &platypus;
    cout << "Указатель p ссылается на объект типа ";
    cout << typeid(*p).name() << endl;
 return 0;
}
```

Динамическая идентификация типа

Разбор примера применения typeid к иерархии полиморфных классов

Если оператор typeid применяется к указателю на объект полиморфного базового класса, тип объекта, на который ссылается указатель, можно определить в ходе выполнения программы. В любом случае, если оператор typeid применяется к указателю на объект неполиморфного базового класса, возвращается базовый тип указателя. Иначе говоря, невозможно определить, на объект какого типа ссылается этот указатель на самом деле.

Закомментируем, например, ключевое слово virtual, стоящее перед именем функции lays_eggs() в классе Mammal, перекомпилируем программу и запустим ее вновь:

Указатель p ссылается на объект типа class Mammal
Указатель p ссылается на объект типа class Mammal
Указатель p ссылается на объект типа class Mammal

Класс Mammal больше не является полиморфным, и каждый объект теперь имеет тип Mammal, т.е. тип указателя p.

Поскольку оператор typeid обычно применяется к разыменованным указателям, следует предусмотреть обработку исключительной ситуации bad_typeid, которая может возникнуть, если указатель будет нулевым.

Оператор typeid имеет вторую форму, получающую в качестве аргумента имя типа. Ее общий вид приведен ниже:

typeid(<имя_типа>)

Например, следующий оператор совершенно правилен: `cout << typeid(int).name();`

Данная форма оператора typeid позволяет получить объект класса type_info, описывающий заданный тип. Благодаря этому ее можно применять в операторах сравнения типов.

Например:

```
void WhatMammal(Mammal  &ob)
{  cout << "Параметр ob ссылается на объект типа ";
   cout << typeid(ob).name()  << endl;
   if(typeid(ob)  == typeid(Cat))
       cout << "Кошки боятся воды.\n";
}
```

Динамическая идентификация типа

Применение динамической идентификации типа

```
#include <iostream>
using namespace std;
 class Mammal {  public:
    virtual bool lays_eggs() {return false;}
    // ...   };
 class Cat: public Mammal { public: /* … /* };
 class Platypus: public Mammal { public:
     bool lays_eggs() { return true; }    /* ... */ };
     class Dog: public Mammal { public: /* … /* };
// Фабрика объектов, производных от класса Mammal
 Mammal *factory()
  { switch(rand() % 3 ) { case 0: return new Dog;
    case 1: return new Cat; case 2: return new Platypus; }
   return 0;      }
int main()
 {  Mammal *ptr;  // Указатель на базовый класс
       int i;     int c=0, d=0, p=0;
 // Создаем и подсчитываем объекты
 for(i=0; i<10; i++)   // Создаем и подсчитываем объекты
    {  ptr = factory(); // Создаем объект
       cout<<"Тип объекта: "<< typeid(*ptr).name();
       cout<< endl;
   // Подсчёт
    if(typeid(*ptr) == typeid(Dog))  d++;
    if(typeid(*ptr) == typeid(Cat))   c++;
    if(typeid(*ptr) == typeid(Platypus))  p++;    }
 cout << endl;
 cout << " Созданные животные:\n";
 cout << " Собаки: " << d << endl;
 cout << " Кошки: " << c << endl;
 cout << " Утконосы: " << p << endl;
return 0;
}
```

Функция factory() создает объекты различных классов, производных от класса Mammal. (Функции, создающие объекты, иногда называются фабриками объектов (object factory)). Конкретный тип создаваемого объекта задается функцией rand(), которая представляет собой генератор случайных чисел в языке C++. Т.е., тип создаваемого объекта заранее не известен. Программа создаст 10 объектов и подсчитывает количество объектов каждого типа. Поскольку все объекты генерируются функцией factory(), для идентификации фактического типа объекта используется оператор typeid.

Результаты работы этой программы приведены ниже.

```
Тип объекта:  class Platypus
Тип объекта:  class Platypus
Тип объекта:  class Cat
Тип объекта:  class Cat
Тип объекта:  class Platypus
Тип объекта:  class Cat
Тип объекта:  class Dog
Тип объекта:  class Dos
Тип объекта:  class Cat
Тип объекта:  class Platypus
  Созданные животные:
Собаки: 2
Кошки: 4
Утконосы: 4
```

Динамическая идентификация типа

Применение оператора typeid к шаблонным классам

```
#include <iostream>
using namespace std;
  template <class T> class myclass {
    T a;
     public:
       myclass(T i) { a = i; }      // ...
   };
int main()
{  myclass<int> o1(10), o2(9);
   myclass<double> o3(7.2);
  cout << "Тип объекта o1: ";
  cout << typeid(o1).name() << endl;
  cout << "Тип объекта o2: ";
  cout << typeid(o2).name() << endl;
  cout <<  "Тип объекта o3: ";
  cout << typeid(o3).name() << endl;
  cout << endl;
   if(typeid(o1) == typeid(o2))
        cout << "Объекты o1 и o2 имеют одинаковый
тип\n";
   if(typeid(o1) == typeid(o3))
    cout << "Ошибка\n";
  else
   cout << "Объекты o1 и o3 имеют разные типы\n";
  return 0;
}
```

Тип объекта, являющегося объектом шаблонного класса, определяется, в частности, тем, какие данные используются в качестве обобщенных при создании конкретного объекта. Если при создании двух экземпляров шаблонного класса используются данные разных типов, считается, что эти объекты имеют разные типы.

Результаты работы этой программы приведены ниже.

```
Тип объекта o1: class myclass<int>
Тип объекта o2: class myclass<int>
Тип объекта o3: class myclass<double>

Объекты ol и o2 имеют одинаковый типы
Объекты ol и o3 имеют разные типы
```

Как видим, хотя два объекта представляют собой экземпляры одного и того же шаблонного класса, если параметры не совпадают, их типы считаются разными. В данной программе объект o1 имеет тип myclass<int>, а объект o3 –myclass<double>. Таким образом, их типы не совпадают.

Динамическая идентификация типов применяется не во всех программах. Однако при работе с полиморфными типами механизм RTTI позволяет распознавать типы объектов в любых ситуациях.

Операторы приведения типа

В языке С++ существуют пять операторов приведения типов. Первый оператор является вполне традиционным и унаследован от языка С.

Операция приведения типов в стиле С:

Она может записываться в двух формах:

- **тип (выражение)**
- **(тип) выражение**

Результатом операции является значение заданного типа, например:

```
int a = 2;
float b = 6.8;
printf ("%lf %d", double (a), (int) b);
```

Величина a преобразуется к типу double, а переменная b – к типу int с отсечением дробной части, в обоих случаях внутренняя форма представления результата операции преобразования иная, чем форма исходного значения.

Остальные четыре оператора приведения типов были добавлены впоследствии, в С++.

К ним относятся операторы: dynamic_cast,
const_cast,
reinterpret_cast,
static_cast.

Эти операторы позволяют полнее контролировать процессы приведения типов.

Операторы приведения типа

Оператор dynamic_cast

Оператор dynamic_cast осуществляет динамическое приведение типа с последующей проверкой корректности приведения. Если приведение оказалось некорректным, оно не выполняется.

Общий вид оператора dynamic_cast:

dynamic_cast <target_type> (expr);

Здесь - параметр target_type задает результирующий тип,
- параметр expr – выражение, которое приводится к новому типу.

Результирующий тип должен быть указательным или ссылочным, а приводимое выражение – вычислять указатель или ссылку. Таким образом, оператор dynamic_cast можно применять для приведения типов указателей или ссылок.

Оператор dynamic_cast предназначен для приведения полиморфных типов. Dynamic_cast применяется для приведения типов в иерархической структуре наследования; он может приводить базовый тип к производному, производный к базовому или один производный тип – к другому производному типу.

Допустим, даны два полиморфных класса B и D, причем класс D является производным от класса B. Тогда оператор dynamic_cast может привести указатель типа D* к типу B*. Это возможно благодаря тому, что указатель на объект базового класса может ссылаться на объект производного класса. Однако обратное динамическое приведение указателя типа B* к типу D* возможно лишь в том случае, если указатель действительно ссылается на объект класса D.

Оператор dynamic_cast достигает цели, если указатель или ссылка, подлежавшие приведению, ссылаются на объект результирующего класса или объект класса, производного от результирующего. В противном случае приведение типов считается неудавшимся. В случае неудачи оператор dynamic_cast, примененный к указателям, возвращает нулевой указатель. Если оператор dynamic_cast применяется к ссылкам, в случае ошибки генерируется исключительная ситуация bad_cast.

Операторы приведения типа

Оператор dynamic_cast

Пример.

Класс Base является полиморфным, а Derived – производным от него:

```
Base *bp, b_ob;
Derived *dp, d_ob;
bp = &d_ob;  // указатель базового типа ссылается на объект производного класса
dp=dynamic_cast<Derived*>(bp); //приведение указателя производного типа выполнено успешно
if(dp) cout << "Приведение выполнено успешно";
```

Приведение указателя bp, имеющего базовый тип, к указателю dp, имеющему производный тип, выполняется успешно, поскольку указатель bp на самом деле ссылается на объект класса Derived. Т.о, этот фрагмент выводит на экран сообщение "Приведение выполнено успешно".

Однако в следующем фрагменте приведение не выполняется, потому что указатель bp ссылается на объект класса Base, а приведение базового объекта к производному невозможно:

```
bp = &b_ob; // указатель базового типа ссылается на объект класса Base
dp = dynamic_cast<Derived *> (bp);  // Ошибка
if(!dp) cout << "Приведение не выполняется";
```

Поскольку приведение невозможно, фрагмент выводит на экран сообщение "Приведение не выполняется".

Операторы приведения типа

Оператор dynamic_cast

Пример. Программа демонстрирует разные ситуации применения 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 << "Приведение типа Derived * к типу
Derived * выполнено успешно\n";
    dp->f();     } else       cout << "Ошибка\n";
 cout << endl;
 bp = dynamic_cast<Base *> (&d_ob);
  if(bp) { cout << "Приведение типа Derived * к
типу Base * выполнено успешно\n";
    bp->f();    } else     cout << "Ошибка\n";
  cout << endl;
  bp = dynamic_cast<Base *> (&b_ob);
   if(bp) { cout << "Приведение типа Base * к типу
                  Base * выполнено успешно\n";
    bp->f();  } else   cout << "Ошибка\n";
  cout << endl;
  dp = dynamic_cast<Derived *> (&b_ob);
                                  // см. продолжение
```

```
if(dp)                          // продолжение
   cout << "Ошибка\n";       else
    cout <<"Приведение типа Base * к типу Derived *
                  невозможно\n";
cout << endl;
bp = &d_ob;  // Указатель bp ссылается на объект
                  класса Derived
dp = dynamic_cast<Derived *> (bp);
if(dp) {
  cout<<"Приведение указателя bp к типу Derived *
выполнено успешно поскольку bp действительно ссылается
на объект класса Derived\n";
    dp->f();  } else   cout << "Ошибка\n";
cout << endl;
bp=&b_ob;//Указатель bp ссылается на объект класса Base
  dp = dynamic_cast<Derived *> (bp);
  if(dp)  cout << "Error";  else
  {cout<<"Теперь приведение указателя bp к типу Derived *
невозможно, так как указатель bp на самом деле ссылается на
объект класса Base\n";    }
 cout << endl;
 dp = &d_ob; // Указатель dp ссылается на объект
                  класса Derived
  bp = dynamic_cast<Base *> (dp);
  if(bp) {
        cout<<"Приведение указателя dp к типу Base *
выполнено успешно\n"; bp->f(); }
    else  cout << "Error\n";
  return 0;
}
```

Операторы приведения типа

Оператор dynamic_cast

Пример. Программа демонстрирует разные ситуации применения оператора dynamic_cast

Результаты работы этой программы приведены ниже:

Приведение типа Derived * к типу Derived * выполнено успешно.
Внутри класса Derived

Приведение типа Derived * к типу Base * выполнено успешно.
Внутри класса Derived

Приведение типа Base * к типу Base * выполнено успешно.
Внутри класса Base

Приведение типа Base * к типу Derived * невозможно

Приведение указателя bp к классу Derived * выполнено успешно, поскольку указатель bp действительно ссылается на объект класса Derived
Внутри класса Derived

Теперь приведение указателя bp к типу Derived невозможно, так как указатель bp на самом деле ссылается на объект класса Base.

Приведение указателя dp к классу Base * выполнено успешно.
Внутри класса Derived

Операторы приведения типа

Замена оператора typeid оператором dynamic_cast

Иногда оператор dynamic_cast можно использовать вместо оператора typeid.

Допустим, что класс Base является полиморфным, а Derived – производным от него. В следующем фрагменте указателю dp присваивается адрес объекта, на который ссылается указатель bp, только если этот объект действительно является экземпляром класса Derived:

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

В этом фрагменте используется традиционный оператор приведения. Это вполне безопасно, поскольку оператор if проверяет легальность приведения с помощью оператора typeid. Однако в этой ситуации лучше заменить оператор typeid и условный оператор if оператором dynamic_cast:

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

Поскольку оператор dynamic_cast выполняется успешно, только если приводимый объект является либо экземпляром результирующего класса, либо объектом класса, производного от результирующего, указатель dp содержит либо нулевой указатель, либо адрес объектов типа Derived.

Поскольку оператор dynamic_cast выполняется успешно, только если приведение является легальным, его применение иногда упрощает ситуацию.

Операторы приведения типа

Оператор dynamic_cast

Пример. В этой программе одна и та же операция выполняется дважды: сначала – с помощью оператора typeid, а затем – оператора dynamic_cast

```
#include <iostream>
#include <typeinfo>
using namespace std;
  class Base {  public:  virtual void f() {}  };
  class Derived : public Base {  public:
    void derivedOnly()
      { cout << "Объект класса Derived \n"; }  };
int main()
{   Base *bp, b_ob;
    Derived *dp, d_ob;
  // ************************************
  // Применение оператора typeid
  // ************************************
  bp = &b_ob;
  if(typeid(*bp) == typeid(Derived))
   { dp = (Derived *) bp;  dp->derivedOnly(); }
  else     cout << "Приведение типа Base к типу
                    Derived невозможно\n";
  bp = &d_ob;
  if(typeid(*bp) == typeid(Derived))
   { dp = (Derived *) bp;  dp->derivedOnly();  }
  else
    cout << "Ошибка,  приведение невозможно!\n";
                                  // см. продолжение
```

```
                                  // продолжение
  // ************************************
  // Применение оператора dynamic_cast
  // ************************************
  bp = &b_ob;
  dp = dynamic_cast<Derived *> (bp);
  if(dp) dp->derivedOnly();
  else   cout << "Приведение типа Base к типу
                    Derived невозможно\n";
  bp = &d_ob;
  dp = dynamic_cast<Derived *> (bp);
  if(dp) dp->derivedOnly();
  else
    cout<<"Ошибка, приведение невозможно!\n";
  return 0;
}
```

Как видим, оператор dynamic_cast упрощает логику приведения указателя базового типа к указателю производного типа.

- **Приведение типа Base к типу Derived невозможно.**
- **Объект класса Derived.**
- **Приведение типа Base к типу Derived невозможно.**
- **Объект класса Derived.**

Операторы приведения типа

Применение dynamic_cast к шаблонным классам

```
#include <iostream>
using namespace std;
template <class T> class Num { protected: T val;
  public:   Num(T x) { val = x; }
    virtual T getval() { return val; }   /* ... */  };
template <class T> class SqrNum : public  Num<T>
  { public: SqrNum(T x) : Num<T>(x) { }
  T getval() { return val * val; }  };
int main()
{   Num<int> *bp, numInt_ob(2);
    SqrNum<int> *dp, sqrInt_ob(3);
    Num<double> numDouble_ob(3.3);
  bp = dynamic_cast<Num<int> *> (&sqrInt_ob);
  if(bp) { cout << "Приведение типа SqrNum<int>* к
        типу Num<int>*\n"<< " выполнено успешно";
 cout<<"Значение равно "<<bp->getval()<<endl; }
       else  cout << "Ошибка\n";   cout << endl;
 dp = dynamic_cast<SqrNum<int> *> (&numInt_ob);
   if(dp)   cout << "Ошибка\n";   else { cout<<"Приве-
дение типа Num<int>* к типу SqrNum<int>*
невозможно\n";
   cout<<"Приведение указателя на объект базового \n";
cout<<" класса к указателю на объект производного типа
невозможно\n";   }   cout << endl;
bp = dynamic_cast<Num<int> *> (&numDouble_ob);
   if(bp)   cout << "Ошибка\n";   else  cout<<"Приведе-
ние типа Num<double>* к типу Num<int>* невоз-
можно\n"; cout<<"Это два разных типа\n";
  return 0;
}
```

Результаты работы этой программы:

Приведение типа SqrNum<int>* к типу Num<int>* выполнено успешно
Значение равно 9

Приведение типа Num<int>* к типу SqrNum<int>* невозможно
Приведение указателя на объект базового класса к указателю на объект производного класса невозможно.

Приведение типа Num<double>* к типу Num<int>* невозможно.
Это два разных типа.

Основной смысл этого фрагмента заключается в том, что с помощью оператора dynamic_cast нельзя привести указатель на объект одной шаблонной специализации к указателю на экземпляр другой шаблонной специализации.

Напоминание: точный тип объекта шаблонного класса определяется типом данных, которые используются при его создании. Таким образом, типы Num<double> и Num<int> различаются.

Операторы приведения типа

Оператор const_cast

Оператор const_cast используется для явного замещения модификаторов const и/или volatile. Операция служит для удаления модификатора const. Как правило, она используется при передаче в функцию константного указателя на место формального параметра, не имеющего модификатора const. Результирующий тип должен совпадать с исходным, за исключением атрибутов const и volatile.

Общий вид оператора const_cast:

const_cast <type> (expr);

Здесь - параметр type задает результирующий тип приведения,
- параметр expr – выражение, которое приводится к новому типу.

```
#include <iostream>
using namespace std;
  void sqrval(const int *val)  {  int *p;
  // Удаление модификатора const
  p = const_cast<int *> (val);
  *p = *val * *val;                    }
int main()
{ int x = 10;
  cout<<"Значение x перед вызовом: "<<x<<endl;
  sqrval(&x);
  cout<<"Значение x  после вызова: "<<x<<endl;
  return 0;
}
```

Эта программа выводит на экран следующие результаты:

```
Значение x перед вызовом: 10
Значение x после вызова: 100
```

Как видим, функция sqrval() изменяет значение переменной x, даже если ее параметр является константным указателем.

Применение оператора const_cast для удаления атрибута const небезопасно. Его следует использовать осторожно.

Атрибут const можно удалить только с помощью оператора const_cast.

Операторы dynamic_cast, static_cast и reinterpret_cast на атрибут const не влияют.

Операторы приведения типа

Оператор static_cast

Оператор static_cast выполняет неполиморфное приведение. Его можно применять для любого стандартного преобразования типов. Проверка приведения в ходе выполнения программы не производится.

Оператор static_cast имеет следующий вид:

static_cast<type> (expr)

Здесь - параметр type задает результирующий тип приведения,
- параметр expr – выражение, которое приводится к новому типу.

Оператор static_cast, по существу, заменяет исходный оператор приведения. Просто он выполняет неполиморфное приведение.

Например, следующая программа приводит переменную типа int к типу double:

```
#include <iostream>
using namespace std;
int main()
{
  int i;
  for(i=0; i<10; i++)
    cout << static_cast<double> (i) / 3 << " ";
  return 0;
}
```

Операторы приведения типа

Оператор reinterpret_cast

Оператор reinterpret_cast преобразует один тип в совершенно другой. Например, он может преобразовать указатель на целое число в целое число, а целое число — в указатель. Кроме того, его можно использовать для приведения несовместимых типов указателей.

Оператор reinterpret_cast имеет следующий вид:

reinterpret_cast<fype> (expr)

Здесь - параметр type задает результирующий тип приведения,
- параметр expr – выражение, которое приводится к новому типу.

Например, эта программа иллюстрирует применение оператора reinterpret_cast:

```
#include <iostream>
using namespace std;
int main()
{
  int i;
  char *p = "This is a string";
  i = reinterpret_cast<int> (p); // Приведение указателя к целому числу
  cout << i;
  return 0;
}
```

Здесь оператор reinterpret_cast преобразует указатель p в целое число, т.е. один основной тип – в другой. Такое преобразование является типичным для оператора reinterpret_cast.