Пособие предназначено для студентов, изучающих язык С++. Является дополнением к существующему методическому пособию[1].
Может быть также полезно преподавателям, ведущим практические занятия по языку С++.

Приведение типов в С++.

В С++ выделены логически различные виды приведения типов. В совокупности они позволяют делать все те преобразования, которые допускает синтаксис Си-преобразований, кроме одного. С помощью старого синтаксиса можно было приводить от производного класса к закрытому базовому классу (что опасно), новый синтаксис этого не позволяет.

Си-приведение типов в С++ полностью поддерживается.

1. dynamic_cast (преобразование + проверка)

Оператор dynamic_cast реализует приведение полиморфных типов (указателей или ссылок) в динамическом режиме.

Замечание. *Преобразования между объектами разных классов (в том числе и родственных) осуществляются с помощью преобразований, определенных пользователем - конструкторов и операторов преобразования.*
Далее, когда говорим о приведении между классами, имеем в виду работу со ссылками или указателями на классы.

dynamic_cast <T*>(p) - преобразует **p** в тип **T*,** если ***p** действительно является объектом типа **T**, или типа класса, производного от **T**, иначе возвращает 0.
dynamic_cast <T&>(p) - преобразует **p** в тип **T&,** если **p** действительно является объектом типа **T**, или типа класса, производного от **T**, иначе генерируется исключительная ситуация **bad_cast**.

Замечание. Возможно и преобразование в обратную сторону, из производного в базовый, но это преобразование выполняется и неявно.

*Замечание. Попытка использовать преобразование **dynamic_cast** для приведений типа между неродственными полиморфными классами допустима. Результат: всегда фиксируется ошибка приведения (**0** либо **bad_cast**).*

*Замечание. **dynamic_cast** позволяет делать приведение **из виртуального полиморфного** (т.е. содержащего виртуальные функции) базового класса к производному, чего обычным приведением сделать нельзя. Для неполиморфных виртуальных базовых классов и **dynamic_cast** не поможет.*

Пример:

```
class  Base{…virtual …};
class Derived : public Base {…};
Base *bp, b_ob;
```

Derived * dp, d_ob;
bp=&d_ob; // указывает на объект производного класса
dp=dynamic_cast <Derived*>(bp); // OK
dp=dynamic_cast <Derived*>(&b_ob); // **0**, т.к. &b_ob не указывает на объект производного класса.

При неудачной попытке приведения типов результатом выполнения dynamic_cast является 0, если в операторе использовались указатели. Если же использовались ссылки, генерируется исключительная ситуация **bad_cast**.

Указатель может иметь нулевое значение, поэтому при динамическом приведении указателя в случае возникновения ошибки может возвращаться это нулевое значение, которое затем может быть проверено в программе. Ошибка при приведении ссылки всегда приводит к возбуждению исключительной ситуации **bad_cast**, так как никакого выделенного значения для ссылок не существует. Проверка правильности динамического приведения ссылок всегда выполняется перехватом исключительной ситуации.

Пример:

```
class A { ... };
class B: public A { ... };
class C: public A { ... };
void f (A * p, A & r)
{ B * pp ;
    if (pp = dynamic_cast<B *> (p))
       { ... /* использование указателя pp    */          }
    else {.../*указатель pp не принадлежит нужному типу*/  }
  B & pr = dynamic_cast<B &> (r);
... /*использование ссылки pr */
}
void g ()
{ try { f (new B, * new B); // правильный вызов
        f (new C, * new C); // выход в перехватчик (C – из
        другой  иерархии, основанной на том же базовом классе)
      }
catch (bad_cast)
     { ... // обработка исключительной ситуации
       }
}
```

2. static_cast (преобразование без проверки)

Статическое приведение типов возможно:

- для указателей (ссылок) на объекты родственных (не обязательно полиморфных) классов, относящихся к одной иерархии классов.
- для свободных указателей (*void**), которые могут преобразовываться в значения указателей любых типов,
- для преобразований между арифметическими типами.

Оператор **static_cast** позволяет выполнять преобразование из типа S в тип T, если обратное преобразование из T в S может быть выполнено неявно.

Замечание: *Допустимость преобразования проверяется на этапе компиляции. В случае явной недопустимости (например, преобразования между указателями на неродственные классы) происходит ошибка компиляции.*
Если указатель на базовый класс не был настроен на объект производного класса, преобразование из указателя на базовый класс к указателю на производный класс будет все же выполнено, что делает опасным использование полученного результата для доступа к данным из производного класса.

Замечание: *При преобразовании* ***static_cast*** *константность сохраняется, с его помощью нельзя убрать атрибут* ***const*** *в выражении.*

Пример:

```
class  Base{…};
class Derived : public Base {…};
Base *bp, b_ob;            Derived  * dp,   d_ob;
      bp=&d_ob; // указывает на объект производного класса
      dp=static_cast <Derived*>(bp);  // OK
      dp=static_cast <Derived*>(&b_ob); // преобразование
    опасное!
      // реально dp указывает на объект базового класса!
```

Пример:

```
int i; double d=3.1415; i=static_cast<int>(d); //OK
```

3. const_cast (отбрасывание const или volatile)

Оператор **const_cast** используется **только**, чтобы снять атрибут const (или volatile) у объекта.

Замечание: *Исходный объект не должен быть константным. Если же исходный объект был объявлен как константный, результат снятия константности с помощью* ***const_cast*** *зависит от компилятора. Обычно это ошибка времени выполнения при попытке изменения константного объекта через неконстантный указатель.*

Пример:

```
class  X{…};
void fun(X* p);
X  x;
const X & sc = x;  // const ссылка на не const объект

fun(&sc); //Error! нельзя передать указатель на константу
```

```
fun(const_cast <X*>(&sc));  // OK! сняли const, можно менять
sc и x в функции
```

4. reinterpret_cast

Оператор **reinterpret_cast** используется для небезопасных преобразований, зависящих от реализации. С его помощью проводят преобразования между указателями разных типов, между интегральными типами и указателями. Действует как Си-приведение.

Пример:

```
char* s=….;
long *lp = reinterpret_cast<long*>(s);
```

Замечание: *Для случая множественного наследования по результату действия* ***reinterpret_cast*** *не совпадает со* ***static_cast*** *(совпадает с Си-приведением).*

Пример:

```
class A { int a; ...};
class B { int b; ...};
class C: public A, public B { int c;...};

C c;
C* pc=&c;
B* pbs=static_cast<B*>(pc);//указывает на B в C

B* pbr = reinterpret_cast<B*>(pc);// указывает на начало
                                  // объекта C, т.е. на A

C* pcs = static_cast<C*>(pbs);  //  указывает на начало
                                // объекта C, т.е. на A

C* pcr = reinterpret_cast<C*>(pbs);//указывает на B в C
```

Объект класса C

pc, pbr, pcs -→

pbs, pcr-→

A	int a; …
B	int b; …
C	int c; …

Список литературы.

1. Волкова И. А., Иванов А. В., Карпов Л. Е. Основы объектно-ориентированного программирования. Язык программирования C++. Учебное пособие для студентов 2 курса. – М.: Издательский отдел факультета ВМК МГУ, 2011 – 112 с.
2. Standard for the C++ Programming Language ISO/IEC 14882, 1998.
3. Страуструп Б. Язык программирования C++. Специальное изд./Пер. с англ. - М.: "Бином", 2005.
4. Страуструп Б. Дизайн и эволюция C++. Пер. с англ. – М.:ДМК Пресс, 2000.
5. Мейерс С. Эффективное использование C++. 35 новых рекомендаций по улучшению ваших программ и проектов: Пер. с англ. – М.:ДМК Пресс; Спб.: Питер, 2006.
6. Элджер Дж. C++: библиотека программиста – Спб.: Питер, 2001.