Преобразования типов в С

В языке С явное приведение типов осуществлялось при заключении типа в круглые скобки, так называемое "приведение в круглых скобках"

Например, преобразование double --> int:

```
int i = (int) 2.5;
```

1) такая форма приведения не наглядна
2) при приведении пользовательских типов могут возникать ошибки

```
Cls1 * a = ...;
Cls2 * b = (Cls2 *) a;
```

Для некоторых типов могут возникать неопределенности. Cls1 и Cls2 могут быть никак не связаны, и тогда операция присвоения не правомочна. В целом надо различать преобразования *связанных* типов и *несвязанных* типов.

Преобразования типов в C++

В C++ дополнительно ввели именованные операторы преобразования типов:

- **const_cast < > ( )**
- **static_cast < > ( )**
- **reinterpret_cast < > ( )**
- **dynamic_cast < > ( )**

Преобразование const_cast <> ()

Оператор **const_cast**, как и следует из его имени, позволяет убрать или добавить константность:

```
A const * ac = ...;
A * a = const_cast<A *>(ac);
```

const_cast не применим:

Для тех объектов, которые объявлены как **const**, нельзы отменять константность. Будет *undefined behaviour*.

Если в функцию, например, передана константная ссылка, то внутри мы можем снять константность, хотя это не хорошо.

Для приведения констант применяется лишь оператор const_cast. Применение любого другого оператора приведения типов в данном случае привело бы к ошибке при компиляции.

Ошибка при компиляции произойдет в случае использования оператора **const_cast** в записи, которая осуществляет любые другие преобразования типов, отличные от создания или удаления константности.

Преобразование const_cast <> () - пример

Предположим, что мы хотим в массиве типа A определена функция **Get**, возвращающая в виде константы i-ый объект из массива A и описываемая так:

```
EX const * Get(int i) const;
```

А теперь мы хотели бы получить еще один вариант функции **Get**, но без константного возврата и с использованием первого варианта:

```
EX & A::Get (int i)
{
  return const_cast<EX &>(const_cast<A const *>(this)->Get(i));
}
```

Нельзя написать Get(i), т.к. уйдем в рекурсию

Надо привести this к const A, а потом убрать константность.

Преобразование **reinterpret**_cast <> ()

Оператор **reinterpret_cast** предназначен для приведения друг к другу указателей, которые друг от друга не зависят:

```
int * p = ...;
double * d = reinterpret_cast<int *>(p);
```

кроме того , с его помощью можно преобразовать любой целочисленный тип в любой ссылочный и наоборот.

Фактически данный оператор просто позволяет взглянуть на одно и то же содержимое памяти, интерпретируя его в соответствии с желаниями программиста.

Преобразование **reinterpret**_cast <> () - 2

Оператор **reinterpret_cast** не меняет константности.

Оператор **reinterpret_cast** является жестко машинно-зависимым. Чтобы безопасно его использовать, надо хорошо понимать, как именно реализованы используемые типы, а также то, как компилятор осуществляет приведение.

Этот оператор ни во что не компилируется, как и **const_cast**. Здесь не происходит вызова никакой функции. Это просто указание системе типов, что теперь у нас используется другой тип.

Проблема безопасности заключается в том, что при изменении типа компилятор не выдаст никаких предупреждений или сообщений об ошибке. Компилятор не способен выяснить, адрес какого именно значения фактически хранит указатель. Отследить такие ошибки иногда чрезвычайно трудно, особенно если приведение указателя к другому типу происходит в одном файле, а используется указатель - в другом.

Преобразование **static**_cast <> ()

Оператор **static_cast** предназначен для проведения преобразований между связанными значениями. Связанность проверяется на этапе компиляции, поэтому и называется *static*.

Типы, к которым применим **static_cast**:

- числовые типы,
- связанные наследованием,
- пользовательские преобразования,
- к void *

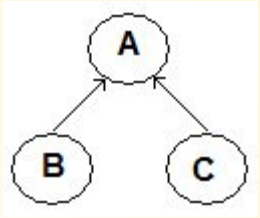
Преобразование **static**_cast <> () – 2

Числовые типы :

```
static_cast<int>(5.5);
```

Типы, связанные наследованием, а также ссылки и указатели на них:

предположим, что классы A,B,C связаны наследованием




```
B * b = ...;
A * a = static_cast<A *>(b);
b = static_cast<B *>(a);
```

В том месте, где пишем это преобразование, должно быть известно, что B - наследник A

Если просто объявить эти классы:

```
class A;
class B;
```

то этот код не пойдет.

А вот **const_cast** и **reinterpret_cast** могли бы сработать – им неважно, как устроены приводимые данные

Преобразование **static**_cast <> () – 3

Пользовательские преобразования : `static_cast<string>("Hello");`

Это эквивалентно `string("Hello");  // Вызов конструктора`

Следует отметить, что проведение подобных преобразований не всегда безопасно, поскольку при выполнении контроль отсутствует

```
class B { };
class D : public B { };
void f(B* pb, D* pd)
{
   D* pd2 = static_cast<D*>(pb);   // pb может указывать не только на B
   B* pb2 = static_cast<B*>(pd);   // безопасно
}
```

Преобразование к void*: можно преобразовать любой указатель к void * и обратно.

Преобразование **static**_cast <> () – пример

Только объявим классы

```
struct A;

struct B;

struct D;

A * a = ...;

B * b = static_cast<B *>(a); // Не скомпилируется

B * b = (B *)a; // Это сработает как reinterpret_cast

                // Мы не сдвинем указатель, но компилятор не выдаст ошибки
```

Если же помимо объявления будут определения классов, то **static_cast** сработает правильно.

Замечание: Если приводим встроенные типы, то это можно делать по старому, в стиле C, в остальных случаях надо использовать C++ операторы преобразования типов.

RTTI - Run-Time Type Information

RTTI позволяет программам, которые используют указатели или ссылки на базовые классы, выяснять фактические типы объектов производных классов, к которым относятся эти указатели и ссылки.

Большинство случаев использования RTTI можно избежать, от этого архитектура станет только лучше.

RTTI обеспечивает два следующих оператора:

- Оператор **typeid**, которые возвращает фактический тип объекта, к которому относится указатель или ссылка.
- Оператор **dynamic_cast**, который осуществляет безопасное преобразование указатели или ссылки на базовый класс в указатель или сссылку на произвольный класс

Эти операторы возвращают информацию о динамическом типе только для тех классов, которые обладают одной или несколькими виртуальными функциями. Для всех остальных классов возвращается информация о статическом типе (т. е. типе времени компиляции)

Оператор typeid

Оператор typeid имеет следующий формат: **typeid(e)**

здесь **e** - это любое выражение или имя класса

Результатом операнда **typeid** является ссылка на объект библиотечного типа **type_info**.

Чтобы использовать класс **type_info**, необходимо подключить библиотечный заголовок **typeinfo**:

```
#include <typeinfo>
```

type_info - структура, позволяющая получить некоторую информацию о типе

Операции с типом typeinfo

Действия, которые можно производить с объектами типа type_info:

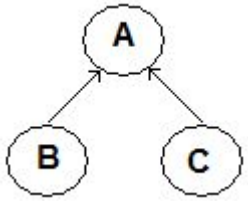
Сравнение на равенство **==**

Сравнение на неравенство **!=**

name() - возвращает символьную строку в стиле C, содержащую отображаемую версию имени типа. Может вернуть пустую строку, что не очень хорошо.

before() - возвращает логическое значение, указывающее на то, следует ли один тип прежде другого. Порядок следования зависит от компилятора. Т.е. с помощью этого метода типы можно упорядочить.

Как работает typeid




```
A * a = new C();
type_info ti = typeid(* a);  // Передается ссылка на объект
ti.name();                   // "C"
```

typeid(int) - определяется в момент компиляции

typeid(A) - определяется в момент компиляции // "A"

typeid(a) - определяется в момент компиляции // "A * "

typeid(* a) - действие времени выполнения.
Для того чтобы это действие действительно было временем выполнения класс A должен обладать полиморфизмом (должно быть наличие виртуальной функции), иначе вернет "A"

```
if (typeid(*a)== typeid(A))…    // Пример использования
```

Преобразование dynamic_cast < > ( )

```
if (Circle * c = dynamic_cast<Circle>(a))
{
... // Первая фигура - круг
}
```

Использование оператора dynamic_cast при программировании

```
A * a = new C();
B * b = dynamic_cast<B *>(a);  // Вернет 0
```

Преобразование указателей с помощью dynamic_cast

```
B & b = dynamic_cast<B &>(* a);
// В случае неудачи бросит исключение
```

Преобразование ссылок с помощью dynamic_cast

dynamic_cast обеспечивает механизм, который проверяет правомерность преобразований в процессе выполнения.

Этот тип преобразований называют «upcast» (подброс), поскольку он перемещает указатель по иерархии классов от производного до его предка.

dynamic_cast <void*> - проверка при выполнении делается для того, чтобы определить реальный тип преобразуемого выражения.

Преобразование dynamic_cast < > - пример 1

```
#include <iostream>
using namespace std;
class B { };
class C : public B { };
class D : public C { };

void f(D* pd) {
   C* pc = dynamic_cast<C*>(pd);   // ok: C is a direct base class
                                   // pc points to C subobject of pd
   B* pb = dynamic_cast<B*>(pd);   // ok: B is an indirect base class
                                   // pb points to B subobject of pd
}
void main(int argc, char * argv[])
{
    D* pd;
    f(pd);
}
```

Преобразование dynamic_cast < > - пример 2

```
#include <iostream>
using namespace std;
struct B {virtual void f();};
struct A : public B {};

void B::f() {
   A* pa = new A;
   B* pb = new B;
   void* pv = dynamic_cast<void*>(pa);   // pv now points to an object of type A
   pv = dynamic_cast<void*>(pb);         // pv now points to an object of type B
}

void main(int argc, char * argv[])
{
  A a;
  a.f();
}
```

Преобразование dynamic_cast < > - пример 3

```
#include <iostream>
using namespace std;
class B {public: virtual void f();};
class D : public B {public: virtual void f();};

void B::f() {
   B* pb = new D;   // unclear but ok
   B* pb2 = new B;

   D* pd = dynamic_cast<D*>(pb);   // ok: pb actually points to a D
   D* pd2 = dynamic_cast<D*>(pb2); // pb2 points to a B not a D
}
void D::f() {}

void main(int argc, char * argv[])
{
  B b;
  b.f();
}
```

Контейнеры в C++

Контейнерами называются классы, предназначенные для размещения большого количества индивидуальных элементов.

Множество операций с контейнерными классами включает в себя возможность легко получить доступ к индивидуальным элементам. Классы также являются средствами достижения абстракции в C++. Абстракция – это возможность игнорирования подробностей. Абстракция и инкапсуляция – близнецы-братья.

Классы скрывают подробности и обеспечивают внешние удобные способы обработки вычислительных задач.

Контейнеры – основные операции

Несмотря на разнообразие внутренней организации контейнерных классов, можно определить группу базовых операций, обеспечивающую общую функциональность контейнеров. К ним относятся:

- Операция организации контейнера (конструктор)
- Операция внесения в контейнер нового элемента (set)
- Операция чтения элемента из контейнера (get)
- Операция пополнения контейнера новым элементом (add)
- Операция удаления заданного элемента из контейнера (remove)
- Операция очистки всего содержимого контейнера (clear)

Именно эти операции чаще всего должны быть переопределены при использовании некоторых «стандартных» базовых классов или определены при создании собственных контейнерных классов.

Контейнерные классы в MFC

Контейнерные классы в MFC называются коллекциями. Все коллекции по способу внутренней организации поделены на три вида:

list (список) – этот класс обеспечивает упорядоченный, неиндексируемый список элементов, базирующийся на двунаправленном списке. Список имеет «голову» и «хвост» и добавляет и удаляет элементы с головы или хвоста списка. Часто используются операции вставки и удаления элементов из середины списка.

array (массив) – этот класс обеспечивает динамически изменяемый по размеру , упорядоченный и индексируемый целыми числами массив некоторых объектов.

map (отображение) – это класс, в котором каждый из объектов ассоциируется с некоторым ***ключевым*** значением. Этот класс еще иногда называют словарем (dictionary)

Эти базовые коллекции в рамках MFC имеют различные способы реализации, отличающиеся по крайней мере одним свойством – использованы ли для реализации класса шаблоны или нет.

Классы коллекций в MFC

Классы коллекций, не использующих шаблоны, приведены в таблице:

Массивы	*Списки*	*Словари*
CObArray	**CObList**	**CMapPtrToWord**
CByteArray	**CPtrList**	**CMapPtrToPtr**
CDWordArray	**CStringList**	**CMapStringToOb**
CPtrArray		**CMapStringToPtr**
CStringArray		**CMapStringToString**
CWordArray		**CMapWordToOb**
CUIntArray		**CMapWordToPtr**

Классы коллекций в MFC

Классы коллекций с использованием шаблонов, приведены в таблице:

Содержимое	**Массивы**	**Списки**	**Отображения**
Объекты любых типов	**CArray**	**CList**	**CMap**
Указатели на объекты любых типов	**CTypedPtrArray**	**CTypedPtrList**	**CTypedPtrMap**

Классы коллекций в MFC – общие свойства

Основные свойства коллекций, определяемые их видом:

Вид	*Упорядо-чивание?*	*Индек-сация?*	*Вставка элемента*	*Поиск заданного элемента*	*Возможно ли дублирование?*
List	**Да**	**Нет**	**Быстро**	**Медленно**	**Да**
Array	**Да**	**Целым**	**Медленно**	**Медленно**	**Да**
Map	**Нет**	**Ключом**	**Быстро**	**Быстро**	**Нет(ключи) Да (значения)**

Коллекция – пример (начало)

Рассмотрим класс vct, предназначенный для хранения целых чисел в форме индексируемого массива.

```
#include <iostream>
using namespace std;
class vct {
    int *p;          // базовый указатель
    int size;        // число элементов
    int maxsize;  // размер занятой памяти
    int crt;         // текущий элемент
public:
    vct(int n);                  // Конструктор
    ~vct() { delete [] p; }  // Деструктор
    inline int getsize()        // Взять размер массива
    { return size;}
    int& operator[] (int i);   // Индексатор  a[i]
    int& get(int i);             // Доступ для чтения
    void set(int i, int val);   // Доступ для записи
    int add(int val);            // Пополнение
    int remove(int i);          // Удаление
    void clear();                // Очистка
    int& next();                // Читать с переходом на следующий
    int& prev();                // Читать  с переходом на предыдущий
};
```

Коллекция – пример

Продолжение 1 …

```
vct::vct(int n=10)   //------------------------------ Конструктор
{
  maxsize =n; crt = size=0; p = new int [n];
}


int&  vct::operator[] (int i)   //---------------------- Индексатор  a[i]
{
    if ( i < 0 || i >= size) {
        cout << "Size???\n";
        exit(1);
    }
    return p[i];
}
```

Коллекция – пример

Продолжение 2 …

```
int& vct::get(int i)          // ------------------------------ Доступ для чтения
{
    if ( i<0 || i>=size) {
        cout << "Size???\n";  exit(1);
    }
    return p[i];
}
void  vct::set(int i, int val)  //------------------------------ Доступ для записи
{
    if (i < 0 || i>=10000) {            // так решили
        cout << "Size???\n";  exit(1);
    }
    if (i >= maxsize) {
      maxsize = i+10;
      int * pnew= new int[maxsize+10];
      memcpy(pnew, p, size* sizeof(int));
      delete [] p;
      p = pnew;
    }
    if (i < size) size = i;
    p[i] = val;
}
```

Коллекция – пример

Продолжение 3 …

```
int vct::add(int val)          //-------------------------------- Пополнение
{
   if (size >= maxsize) {
      if (maxsize>=10000) {      // Мы так решили
        cout << "Size???\n";
        exit(1);
      }
      maxsize = size+10;
      int * pnew= new int[maxsize];
      memcpy(pnew, p, size* sizeof(int));
      delete [] p;
      p = pnew;
    }
    p[size++] = val;
    return size;
}
```

Коллекция – пример

Продолжение 4 …

```
int vct::remove(int i)          //------------------------------ Удаление
{
    if (i<0 || i> size-1)
      return -1;                // Игнорируем выполнение
    if (i < size-1)
        memcpy(&p[i],&p[i+1], (size-i+1) * sizeof(int));
    size--;
    return i;
}
void vct::clear()               //--------------------------------- Очистка
{
  if (maxsize >10) {
    maxsize = 10;
    delete [] p;
    p = new int[10];
  }
  crt = size = 0;
}
```

Коллекция – пример

Продолжение 5 …

```
int& vct::next()              //------------------- Читать С ПЕРЕХОДОМ НА следующий
{
    if (size==0) {
        cout << "Size???\n";   exit(1);
    }
    if (crt >= size)     return (p[crt=0]);
    else                 return (p[crt++]);
}

int& vct::prev()              //-------------------- Читать  С ПЕРЕХОДОМ НА предыдущий
{
    if (size==0) {
        cout << "Size???\n"; exit(1);
    }
    if (crt == 0) {
        crt=size-1;
       return (p[0]);
    }
    else
        if (crt > size-1) crt = size-1;
        return (p[crt--]);
}
```

Коллекция – пример

Продолжение 6 …

```
void main() {
  vct  v;
  int i;
  for (i=0; i < 9; i++)     v.add(i);

  v.remove(5);
  cout<<v[5]<<endl;

  for (i=0; i < v.getsize(); i++)
      cout << v.next()<<"  ";
  cout << endl;

  for (i=v.getsize()-1; i > -1; i--)
      cout << v.prev()<<"  ";
  cout << endl;

  char s; std::cin>>s;
}
```

Результат:

```
6
0  1  2  3  4  6  7  8
8  7  6  4  3  2  1  0
```

Новые методы:
позиционирование…,
покажи текущий…
Разделение текущих по next и prev

Коллекция – итераторы

На практике может потребоваться организовывать перебор элементов контейнера одновременно и разными способами. Решение этой задачи состоит в создании отдельного связанного с контейнером класса, называемого итератором (iterator class), в функции которого входит обращение и восстановление элементов контейнера.

```
. . .
friend vct_iter;
. . .
class vct_iter {
private:
   vct* pv;    // Базовый указатель на контейнер
   int crt;    // Индекс обращения
public:
    vct_iter(vct& v): crt(0), pv(&v) { }
    int& next();
};
```

Коллекция – итераторы -продолжение

```
int& vct::next()
{
   if (crt == pv -> size)
     return (pv -> p[crt = 0]); // на 1-й элемент
   else
     return (pv -> p[crt++]);  // на следующий
}
```

Эти два класса имеют дружественное отношение. Итерация, как фундаментальная операция должна быть эффективна. Главное, что для этого нужно – быстрый доступ к частным деталям реализации контейнера.

Полное отсоединение обращения от агрегированного объекта позволяет делать столько итераторов, сколько нужно для проведения вычислений.