

Преобразование указателей

Оператор `dynamic_cast<>`, Run Time Type Information

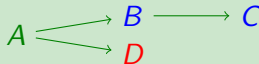
- преобразует указатель (ссылку) базового типа на указатель (ссылку) производного класса:

```
Derivative* pd = dynamic_cast<Derivative*>(pb);
```

```
Derivative& rd = dynamic_cast<Derivative&>(rb);
```

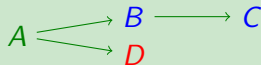
- если преобразование невозможно — возвращает нулевой указатель, а для ссылок возбуждает исключение
- `dynamic_cast<>` опирается на RTTI, то есть на информацию о каждом полиморфном классе **во время выполнения программы**

Иерархия классов для тестов:



👉 В классах `B` и `C` имеется функция `m()`, которой нет ни в `A` ни в `D`

Реализация иерархии:



```
struct A {  
    virtual void Iam() { cout << "I am A" << endl;}  
};  
  
struct B : public A {  
    virtual void Iam() { cout << "I am B" << endl;}  
    virtual void m() { cout << "fun m() from B" << endl;}  
};  
  
struct C : public B {  
    virtual void Iam() { cout << "I am C" << endl;}  
    virtual void m() { cout << "fun m() from C" << endl;}  
};  
  
struct D : public A {  
    virtual void Iam() { cout << "I am D" << endl;}  
};
```

- Объекты всех классов хранятся как набор указателей на базовый класс **A**

... вызываем функцию `Iam()` для всех объектов

✓ Механизм виртуальных функций:

```
vector<A*> pa { new A, new B, new C, new D };  
for(auto p: pa) {  
    p->Iam();  
}
```

```
I am A  
I am B  
I am C  
I am D
```

- Как вызвать `m()` для подходящих объектов наиболее простым способом?

... вызываем функцию `m()` только для ветки **B**

✓ Оператор `dynamic_cast<>`:

```
for(auto p: pa) {  
    B* pb = dynamic_cast<B*>(p);  
    if( pb ) pb->m();  
}
```

```
fun m() from B  
fun m() from C
```

Оператор typeid(), Run Time Type Information


- typeid(X) возвращает объект с информацией о фактическом типе X, тип возвращаемого объекта: std::type_info
- необходим заголовочный файл <typeinfo>

Пример использования typeid()

```
#include <typeinfo> // for typeid()
for(auto p: pa) {
    cout << typeid(*p).name() << " ";
    if(typeid(*p) == typeid(B)) ((B*)(p))->m();
    else if(typeid(*p) != typeid(D)) p->Iam();
    else cout << endl;
}
```

Output

```
1A I am A
1B fun m() from B
1C I am C
1D
```

 Наиболее часто typeid используют для сравнения с typeid() тестируемого класса

Обратите внимание

- В `typeid()` можно использовать имя класса: `typeid(*p)==typeid(A)`
- Константность игнорируется: `typeid(const T) == typeid(T)`
- Тип указатель на класс и тип самого класса различаются:
`typeid(*p).name()` вернет `1A`
`typeid(p).name()` вернет `P1A`
- `typeid(p)` для нулевого указателя возбуждает исключение `bad_typeid`
- Функция `name()` возвращает текст включающий имя класса, но содержимое текста зависит от компилятора

Замечания к использованию `dynamic_cast<>` и `typeid()`

- RTTI может не поддерживаться: `-fno-rtti`
- В случае сложного разветвленного наследования их вызовы «дороги»
- Вместо RTTI рекомендуется использовать полиморфизм и шаблоны

Преобразование с помощью `static_cast<>`

`static_cast<тип данных>` (переменная или выражение)

☞ Позволяет проводить «хорошо определенные» преобразования типов:

- преобразования типов для которых определены операторы преобразования или конструкторы с одним аргументом

☞ Для указателей:

- от указателя на производный класс к указателю на базовый
- от указателя на базовый класс к указателю на производный

☞ нет проверки правильности преобразования

- от `void*` к любому другому указателю

☞ правильность гарантируется только если это обратное преобразование

❶ указатели: производный → базовый

```
A* pa1 = static_cast<A*>(new B);  
pa1->Iam(); // I am B
```

❷ указатели: базовый → производный

```
B* pb1 = static_cast<B*>(new A);  
pb1->Iam(); // I am A  
pb1->m();    // Ошибка сегментирования
```

❸ указатели: void* → произвольный указатель

```
void *vp = new B; // B* -> void*, всегда возможно сделать!  
static_cast<A*>(vp)->Iam(); // I am B  
static_cast<D*>(vp)->funD(); // Ошибка сегментирования
```

произвольный* → произвольный* ?

```
static_cast<D*>(new B)->Iam(); // invalid static_cast from 'B*' to 'D*'
```

Преобразование с помощью `const_cast`

`const_cast<тип данных>(const переменная)`

☞ «снятие константности» для указателей или ссылок:
разрешает запись в `const переменная`

☞ `const_cast` это только указание компилятору!

Типичное применение: передача константного указателя в функцию

```
void print(char* t) {  
    cout << t << endl;  
}  
  
const char* text = "This is test"; // const-object  
print(text); // compilation error  
print(const_cast<char*>(text)); // This is test
```


Оператор reinterpret_cast<>

reinterpret_cast<тип данных> (переменная)

- ☞ средство чтобы «обмануть» компилятор: меняет один тип данных на другой (**type punning**)
- ☞ никакие проверки не проводятся и это может легко привести к ошибке времени исполнения

Пример

```
struct fourchar {  
    char a,b,c,d;  
};  
  
unsigned int tmp = 0x33445566;  
fourchar& pt = reinterpret_cast<fourchar &>(tmp);  
cout<<"access fields: "<<pt.a<<" "<<pt.c<< endl; // access fields: f D
```

Исключения (exceptions)

C++ имеет специальный механизм для обработки ошибок

- можно «возбудить исключение» (`raise the exception`)
- можно перехватить исключение (`catch the exception`)

Основные сведения

- Исключение возбуждается с помощью оператора `throw`
- Наблюдение за исключениями происходит внутри блока `try{...}`
- Блоки `catch(type){...}` «отлавливают» исключения в соответствии с **типом выражения**
- Если исключение произошло вне блока `try` или ни один из `catch` блоков не подошел, происходит аварийное завершение программы

Пример

```
double my_sqrt(double x) {  
    if (x < 0) throw(x); // 1) raise the exeption  
    return sqrt(x);  
}  
  
cout << "Enter number (negative will cause exception)" << endl;  
cin >> x;  
try { // 2) check for exceptions in this block  
    cout << " my_sqrt(" << x << ")= " << my_sqrt(x) << endl;  
}  
catch(double a) { 3) do something if an exception occur  
    cerr << " catch exception: double a= " << a << endl;  
}
```

```
Enter number (negative will cause exception)  
-9  
my_sqrt(-9)=  catch exception: double a= -9
```

Вызов `throw x` передает двоякую информацию

- 1 тип исключения: `x` – это `double`, поэтому срабатывает блок `catch(double a)`
- 2 значение переменной `x` (в блоке `catch(double)` значение `x` недоступно)

- 📖 после `try` может быть несколько `catch` для исключений разного типа
- 📖 блок `catch(...)` перехватывает исключения любого типа

Пример с несколькими `catch` блоками

```
try {...}  
catch (const char* msg) {  
    cerr << "const char* exception handled" << msg << endl;  
}  
catch (int error_number) {  
    cerr << "int exception handled" << error_number << endl;  
}  
catch(...) {  
    cerr << "Handler for any type of exception called" << endl;  
}
```

Некоторые стандартные исключения

Исключения в `dynamic_cast<>`

- 👉 В случае работы с ссылками `dynamic_cast<>` возбуждает исключение `std::bad_cast`

Пример, сравните с примером для указателей

```
vector<A*> pa { new A, new B, new C, new D };
```

```
for(auto p: pa) {  
    A& ref = *p;  
    try {  
        dynamic_cast<B&>(ref).m();  
    } catch (std::bad_cast) {  
        continue;  
    }  
}
```

```
// dynamic_cast для указателей  
for(auto p: pa) {  
    B* pb = dynamic_cast<B*>(p);  
    if( pb ) pb->m();  
}
```

Проверка допустимости диапазона в `vector<>`

👉 Функция `at(idx)` возвращает элемент вектора если индекс `idx` лежит в диапазоне `[0,size())` или вызывает исключение `out_of_range`

Пример: `at(idx)`

```
#include <stdexcept>                // set of std-exceptions
vector<int> vec {1,2,3,4,5};
try {
    cerr << " vec.at(4)= " << vec.at(4) << endl;
    cerr << " vec.at(6)= " << vec.at(6) << endl;
} catch(out_of_range& e) {
    cerr << e.what() << endl;
}
```

```
vec.at(4)= 5
vec.at(6)= vector::_M_range_check:
__n (which is 6) >= this->size() (which is 5)
```

Исключения в операторе new

👉 Если `new` не может выделить память, возбуждается исключение `std::bad_alloc`

Пример

```
const unsigned int Size = 100*1024*1024;
int i = 0;
try {
    for(i = 1; i < 100; i++) {
        int* a = new int[Size]; // 400Mb
    }
} catch(std::bad_alloc&) {
    cerr << " iter# " << i << " catch exeption: stop " << endl;
}
```

Output for 1Gb virtual memory limit

```
iter# 3 catch exeption: stop
```

`new(nothrow)`

👉 `new` с параметром `std::nothrow` возвращает нулевой указатель, а исключение не возбуждается

Пример

```
#include <new>           // std::nothrow defined in header <new>
const unsigned int Size = 100*1024*1024;
for(int i = 1; i < 100; i++) {
    int* a = new(nothrow) int[Size];
    if( a==0 ) {
        cerr << " iter# " << i << " new return 0: stop" << endl;
        exit(EXIT_FAILURE);
    }
}
```

Output for 1Gb virtual memory limit

```
iter# 3 new return 0: stop
```


- `void f() noexcept`; – указание, что `f()` не возбуждает исключений
 - `void f() noexcept(false)`; – `f()` может возбуждать исключения
 - начиная с C++17 `noexcept` входит в спецификацию функции
- 👉 вместо `noexcept(false)` можно `throw()`, но в C++20 это удалено

```
void fun() noexcept { // function that does not throw
    f();               // valid, even if f() throws
    throw 42;          // valid, effectively a call to std::terminate
}
void fun(); // function that might throw
// ERROR: нельзя переопределять если отличие только в noexcept
void fun1(); // OK! other function that might throw
void (*fp)() noexcept; // pointer to function that does not throw
fp = fun;             // OK
fp = fun1;             // ERROR since C++17
```

Замечания к использованию исключений

- Использование исключений создает скрытые потоки управления которые трудно понимать, подобно оператору `goto`
- Генерация исключений отрицательно сказывается на производительности системы
 - ✓ если исключения предусмотрены, но не срабатывают, то производительность практически не снижается

☞ Неаккуратное использование ведет к утечке ресурсов

```
#include <memory> // smart pointers
```

Используются для автоматизации управлением памятью:

- «умные указатели» работают для объектов, которые создаются с помощью `new` и в целом подобны обычным указателям
- объект автоматически удаляется когда исчезает последний «умный указатель» на него – объект становится не нужен
- типы указателей:
 - ✓ `unique_ptr<>` — объекту соответствует **единственный** умный указатель
 - ✓ `shared_ptr<>` — возможно несколько указателей и объект удаляется, когда удаляется **последний** `shared_ptr<>` на этот объект
 - ✓ `weak_ptr<>` — не владеет объектом, но позволяет следить за `shared_ptr` и за соответствующим ему объектом
 - ✗ `auto_ptr<>` — умные указатели из C++98: неудачные и теперь устаревшие

Уникальное владение с `unique_ptr`

- Создаем указатель `unique_ptr` вместе с созданием объекта
- `unique_ptr` нельзя присвоить никакому другому указателю, но можно «передвинуть», move-ctor
- Окончание жизни `unique_ptr` вызывает уничтожение объекта

Тестовая функция с `unique_ptr`

```
void test_unique_ptr() {  
    unique_ptr<int> up1(new int {11}); // OK!  
    cout << "*up1= " << *up1 << endl;      // *up1= 11  
  
    // the direct assignment is forbidden:  
    // unique_ptr<int> up2 = new int {11};  
    // no conversion from int* to unique_ptr  
  
    // unique_ptr<int> up2 = up1; // copy ctor is deleted
```

... продолжение

```
unique_ptr<int> up2 = std::move(up1);    // move: up1 is nullptr now
cout << " up1= " << up1.get() << endl; // up1= 0
cout << "*up2= " << *up2 << endl;      // *up2= 11
```


```
// Array form of unique_ptr:
```

```
unique_ptr<int[]> up3(new int[5] {1,2,3,4,5}); // array
for(int i = 0; i < 5; i++) cout << up3[i] << " ";
cout << endl; // 1 2 3 4 5
```

```
// make_unique<> - function to creates a unique pointer C++14
```

```
auto up4 = make_unique<Stack>(); // one Stack
auto up5 = make_unique<Stack>(5); // array of Stack[5]
```

```
}
```

 По завершению функции все объекты на которые указывали уникальные указатели автоматически удалятся

Умные указатели `shared_ptr`

- `shared_ptr` обычно создается вместе с созданием объекта
- `shared_ptr` можно перенаправить на другой объект методом `reset`
- объект удаляется когда «исчезает» последний `shared_ptr` на него

```
shared_ptr<int> sp1(new int {5}); // one int with value 5
shared_ptr<int> sp2= new int {5}; // no conv. from int* to shared_ptr
cout << "sp1= " << *sp1 << endl; // *sp1= 5

// reset() method: allocates the new memory object;
//          old memory block is released automatically
sp1.reset(new int {3});
cout << "sp1=" << *sp1 << endl; // *sp1=3

shared_ptr<int> sp0;          // an uninitialized shared pointer
sp0.reset(new int {0});      // point to new obj with reset()
cout<<"sp0="<<*sp0<<endl; // *sp0=0
```

... продолжение

```
auto sp2 = sp1; // one more pointer on the same memory
if( sp2 == sp1 ) cout << "sp2=" << *sp2 << endl; // *sp2=3
sp2.reset(new int {1}); // *sp1 memory block still exist
cout << "*sp2=" << *sp2 << " *sp1=" << *sp1 << endl; // *sp2=1 *sp1=3

// C++17: shared_ptr can be used for dynamic arrays
shared_ptr<int[]> spa(new int[5] {1,2,3,4,5}); // array of 5 elements
// elements can be access using operator[]
cout<<"spa[3]= "<<spa[3]<<endl; // spa[3]= 4
```

☞ Как только на объект не указывает ни один из `shared_ptr` он уничтожается

Проверка (assertion) при компиляции

(C++11)

```
static_assert(bool_constexpr, message)
```

👉 Проверка логического выражения **во время компиляции** и остановка компиляции если выражение ложно

`bool_constexpr` — константное выражение

`message` — текст выводимый при остановке (в C++17 необязателен)

Успех:

```
// компилятор проверит, что -3/2 == -1 и остановится если это не так
static_assert(-3/2==-1, "negative values rounds away from zero");
static_assert(int(-3./2)==-1,"negative values rounds away from zero");
```

Неудача:

```
static_assert(sizeof(void*)<=sizeof(int), "can not store void* in int");
error: static assertion failed: can not store void* in int
```