Приведение типов

МГТУ им. Н.Э. Баумана

April 4, 2016

Приведение типов

Необходимость операторов приведения, отличных от (type)v

- лучшая практика приведения типа в программе: не выполнять приведение

Приведение в стиле C:

```
1 (NewType)expression
```

- при преобразовании указателей нет проверки допустимости преобразования;
- можно производить и платформонезависимые, и платформозависимые преобразования типов;
- можно приводить константный тип к неконстантному;
- при чтении программы сложно обнаружить конструкцию T(v)

необходимость в лучших операторах приведения типов

Чем плох C-style cast

преобразования с потерей константности

```
1 const char* str = "Hello world";
2 char* str1 = (char*) str;
3 str1[0] = 'a'; // uv
```

платформо(не)зависимые преобразования типов

```
1 Base* b = new Base;
2 uint32_t ptrval = (uint32_t)b; // ok for x86, platform-dependent
3 uint32_t floatRounded = (uint32_t)1.2345; // ok, platform-independent
```

Чем плох C-style cast

при преобразовании указателей нет проверки допустимости преобразования

```
1  struct Base {
2      virtual ~Base() {}
3  };
4  struct Derived: Base {
5      virtual void name() {}
6  };
7  int main() {
8    Base* b = new Base();
9    Derived* d = (Derived*) b;
10   d->name(); // undefined behavior
11 }
```

Главный недостаток:

Все перечисленные преобразования имеют одинаковый синтаксис и сложно различить, какую угрозу таит в себе конкретное применение преобразования типа.

Контроль на этапе компиляции

static_cast

Оператор преобразования связанных между собой типов, таких как указатели на классы в одной иерархии, преобразование целого типа к перечислению, вещественного типа к целому. Контроль корректности преобразования осуществляется на этапе компиляции.

```
1    NewType static_cast<NewType>(expression)
```

```
1    int a = static_cast<int>(1,2345678);
```

Инициализирующее преобразование

Если возможно объявление и инициализация временного объекта типа NewType в виде `NewType tmp(expression)` (то есть существует подходящий конструктор или заданный пользователем оператор преобразования типа), `static_cast<NewType>(expression)` выполняет создание такого объекта.

```
1 int n = static_cast<int>(3.14);
2 std::cout << "n = " << n << '\n';
```

```
1 struct Vector {
2   Vector(size_t size) {mSize = size;}
3 };
4 // ...
5 Vector v = static_cast<Vector>(10);
```

Применение

lvalue -> xvalue

```
1  struct Vector {
2    Vector(Vector&& v) {
3      size = v.size;
4      buf = v.buf;
5      v.buf = nullptr;
6      v.size = 0;
7    }
8  private:
9    size_t size;
10   T* buf;
11 };
12 //...
13 Vector v2 = static_cast<Vector&&>(v);
14 std::cout << "v size " << v.size() << '\n'; // 0
```

Применение

Нисходящее преобразование по иерархии типов

При нисходящем преобразованиии не происходит проверки допустимости преобразования. Допустимо для указателей и ссылок.

```
1 struct B {};
2 struct D : B {};
3 // ...
4 D d;
5 B& br = d; // upcast via implicit conversion
6 D& another_d = static_cast<D&>(br); // downcast
```

Игнорирование возвращаемого значения

```
1 struct Notifier {
2   size_t notify() {
3     // ...
4     return notified_count;
5   }
6 };
7 // ...
8 static_cast<void>(notifier.notify());
```

Применение

Инверсия стандартного преобразования типа

```
1  int n = 0;
2    void* nv = &n;
3    int* ni = static_cast<int*>(nv);
4    std::cout << "*ni = " << *ni << '\n';
```

Преобразование массива к указателю

Оператор преобразования связанных между собой типов, таких как указатели на классы в одной иерархии, преобразование целого типа к перечислению, вещественного типа к целому. Контроль корректности преобразования осуществляется на этапе компиляции.

```
1 struct B {};
2 struct D : B {};
3 // ...
4 D a[10];
5 B* dp = static_cast<B*>(a);
```

Применение

Преобразования перечислений

```
enum class E { ONE, TWO, THREE };
enum EU { ONE, TWO, THREE };
// ...
// scoped enum to int or float
E e = E::ONE;
int one = static_cast<int>(e);

// int to enum, enum to another enum
E e2 = static_cast<E>(one);
EU eu = static_cast<EU>(e2);
```

void* к указателю на любой тип

```
int e = 2;
void* voidp = &e;
Vector* p = static_cast<Vector*>(voidp);
```

Константность

const_cast

Приведение типов, позволяющее убрать квалификаторы const и volatile.

```
NewType const_cast<NewType>(expression)
```

```
int a = 1;
const int* b = &a;
// *b = 2; // error: *b is const
*const_cast<int*>(b) = 2; // ok
```

Применение допустимо, если точно известно, что на самом деле объект не константный!

Применение

В константном методе

```
struct type {
    type() :i(3) {}
    void m1(int v) const {
        // this->i = v; // compile error: this is a pointer to const
        const_cast<type*>(this)->i = v; // OK as long as the type object
            isn't const
    }
    int i;
};
// ...
type t;
t.m1(); // ok, t is not const
const type t1;
t1.m1(); // UNDEFINED BEHAVIOR!
```

Еще undefined behavior

```
const char *str = "hello";
char *str1 = const_cast<char*>(str);
str1[0] = 'a'; // UB
```

```
const int j = 3; // j is declared const
int* pj = const_cast<int*>(&j);
*pj = 4; // undefined behavior!
```

Преобразование между любыми типами

reinterpet_cast

Допускает практически любое преобразование типа. Использование reinterpret_cast не добавляет никаких инструкций в машинный код программы – это лишь сообщение компилятору, что память в конкретной области следует считать имеющий указанный тип.

```
1    NewType reinterpret_cast<NewType>(expression)
```

```
1    char c[4] = {1, 2, 3, 4};
2    int a = reinterpret_cast<int>(*c);
```

Любой тип T1 может быть приведен при использовании reinterpet_cast к типу T2, при этом никакие конструкторы не будут вызваны.

Примеры

Указатель к целому и обратно

```
1 // pointer to integer and back
2 uintptr_t v1 = reinterpret_cast<uintptr_t>(&i); // static_cast is an
      error
3 std::cout << "The value of &i is 0x" << std::hex << v1 << '\n';
4 int* p1 = reinterpret_cast<int*>(v1);
5 assert(p1 == &i);
```

Преобразования указателей на функции

```
1 int f() { return 42; }
2 // ...
3
4 // pointer to function to another and back
5 void(*fp1)() = reinterpret_cast<void(*)()>(f);
6 // fp1(); undefined behavior
7 int(*fp2)() = reinterpret_cast<int(*)()>(fp1);
8 std::cout << std::dec << fp2() << '\n'; // safe
```

Преобразование типа с контролем во время выполнения

dynamic_cast

Безопасное преобразование указателей и ссылок на полиморфные классы вверх и вниз по иерархии наследования.

```
1    NewType dynamic_cast<NewType>(expression) throw(std::bad_cast)
```

Поведение для указателей (NewType == Type*):

- возвращает значение типа NewType, если объект-значение expression действительно имеет тип Type* или тип DerivedType*, где DerivedType – производный от Type (downcast, sidecast);
- возвращает nullptr в противном случае.

Поведение для ссылок (NewType == Type&):

- условия приведения аналогичны приведению указателей;
- в случае невозможности приведения выбрасывается исключение std::bad_cast.

```
1    Base* b = new Derived();
2    Derived* d = dynamic_cast<Derived*>(b); // ok
```

Примеры

downcast, sidecast

```
1  struct V {
2    virtual void f() {}; // must be polymorphic to use runtime-checked
         dynamic_cast
3  };
4  struct A : virtual V {};
5  struct B : virtual V {
6    B(V* v, A* a) {
7      // casts during construction (see the call in the constructor of D
           below)
8      dynamic_cast<B*>(v); // well-defined: v of type V*, V base of B,
           results in B*
9      // dynamic_cast<B*>(a); // undefined behavior: a has type A*, A not a
           base of B
10   }
11 };
12 struct D : A, B {
13     D() : B((A*)this, this) { }
14 };
15 int main()
16 {
17     D d; // the most derived object
18     A& a = d; // upcast, dynamic_cast may be used, but unnecessary
19     D& new_d = dynamic_cast<D&>(a); // downcast
```

Примеры

downcast – pointers

```
1  struct Base {
2      virtual ~Base() {}
3  };
4  struct Derived: Base {
5      virtual void name() {}
6  };
7  int main() {
8      Base* b1 = new Base;
9      if(Derived* d = dynamic_cast<Derived*>(b1)) // ???
10     {
11         std::cout << "downcast from b1 to d successful\n";
12         d->name(); // safe to call
13     }
14
15     Base* b2 = new Derived;
16     if(Derived* d = dynamic_cast<Derived*>(b2)) // ???
17     {
18         std::cout << "downcast from b2 to d successful\n";
19         d->name(); // safe to call
20     }
21     delete b1; delete b2;
22 }
```

Примеры

downcast – pointers

```
1  struct Base {
2      virtual ~Base() {}
3  };
4  struct Derived: Base {
5      virtual void name() {}
6  };
7  int main() {
8      Base* b1 = new Base;
9      if(Derived* d = dynamic_cast<Derived*>(b1)) // ???
10     {
11         std::cout << "downcast from b1 to d successful\n";
12         d->name(); // safe to call
13     }
14
15     Base* b2 = new Derived;
16     if(Derived* d = dynamic_cast<Derived*>(b2)) // ???
17     {
18         std::cout << "downcast from b2 to d successful\n";
19         d->name(); // safe to call
20     }
21     delete b1; delete b2;
22 }
```

```
1  downcast from b2 to d successful
```