КОНСТРУИРОВАНИЕ ПРОГРАММ

Лекция № 19 — Преобразование типов. Исключения.

Преподаватель: Поденок Леонид Петрович, 505а-5

+375 17 293 8039 (505а-5)

+375 17 320 7402 (ОИПИ НАНБ)

prep@lsi.bas-net.by

ftp://student:2ok*uK2@Rwox@lsi.bas-net.by/

Кафедра ЭВМ, 2021

2021.12.15

Оглавление

Преобразование типов

Неявное преобразование

Когда значение копируется в совместимый тип, автоматически выполняются неявные преобразования. Например:

```
short a = 2000;
int b;
b = a;
```

Здесь значение **a** повышается от **short** до **int** без необходимости какого-либо явного оператора. Этот процесс известен как стандартное преобразование (преобразование по-умолчанию).

Стандартные преобразования действуют на фундаментальные типы данных и позволяют выполнять преобразования между числовыми типами (от **short** до **int**, от **int** до **float**, от **double** до **int** ...), в или из **bool** и некоторые преобразования указателей.

Преобразование в **int** из некоторого целочисленного типа меньшего размера или в **double** из **float** называется повышением и гарантирует получение точно такого же значения в целевом типе.

Преобразования между арифметическими типами, которые не являются повышениями, не всегда могут точно представлять одно и то же значение:

- Если отрицательное целочисленное значение преобразуется в тип без знака, результирующее значение соответствует его побитовому дополнению до 2 (то есть -1 становится наибольшим значением, представляемым типом, -2 — вторым по величине, ...).

- Преобразования из/в **bool** считают **false** эквивалентным нулю (для числовых типов) и нулевому указателю (для типов указателей). Значение **true** эквивалентно всем другим значениям и преобразуется в эквивалент 1.

- Если преобразование происходит от типа с плавающей запятой к целочисленному типу, значение усекается (десятичная часть удаляется).

Если результат находится вне диапазона представимых значений по типу, преобразование вызывает неопределенное поведение.

- Если преобразование выполняется между числовыми типами одного и того же класса (целое в целое или плавающее в плавающее), преобразование является допустимым, но значение зависит от реализации (и может не быть переносимым).

Некоторые из этих преобразований могут означать потерю точности, о чем компилятор *может* сигнализировать с помощью предупреждений[1].

Такого предупреждения можно избежать с помощью явного преобразования.

Что касается неосновных типов, массивы и функции неявно преобразуются в указатели, а указатели в целом допускают следующие преобразования:

- нулевые указатели могут быть преобразованы в указатели любого типа;
- указатели на любой тип могут быть преобразованы в безтиповые указатели **`void`**.
- повышение указателя — указатели на производный класс могут быть преобразованы в указатель на доступный и однозначный базовый класс, без изменения его cv-квалификации.

1 -W -Wall -Wextra — gcc, clang; /W4 — MSVC

Неявные преобразования с классами

В мире классов неявными преобразованиями можно управлять с помощью трех функций-членов:

конструкторы с одним аргументом — позволяют неявное преобразование из определенного типа для инициализации объекта;

оператор присваивания — разрешить неявное преобразование из определенного типа в присваиваниях;

оператор приведения типа — разрешить неявное преобразование в определенный тип.

Например:

```
// implicit conversion of classes:
#include <iostream>

class A {};

class B {
public:
  B(const A&) {}                              // приведение из A (конструктор)
  B& operator= (const A&) {return *this;} // приведение из A (присвоение)
  operator A() { return A(); }  // приведение в A (оператор приведения типа)
};
```

```
int main () {

  A foo;

  B bar = foo;    // вызывается constructor
  bar   = foo;    // вызывается оператор присваивания
  foo   = bar;    // вызывается оператор приведения типа (B к A)

  return 0;
}
```

Оператор приведения типов использует особый синтаксис — он использует ключевое слово **operator**, за которым следует тип назначения и пустой набор скобок.

```
operator A() { return A(); }
```

Обратите внимание, что тип возвращаемого значения является типом назначения и поэтому не указывается перед ключевым словом **operator**.

Ключевое слово explicit

При вызове функции C++ допускает одно неявное преобразование для каждого аргумента.
Но это не всегда то, что требуется.
Например, мы в последний пример добавим следующую функцию:

```
void fn(B arg) {}
```

Эта функция принимает аргумент типа **B**, но ее также можно вызывать с объектом типа **A** в качестве аргумента:

```
#include <iostream>

class A {};

class B {
public:
//  B(const A&) {}             // приведение из A (конструктор)
    explicit B(const A& x) {} // требуется указывать явное преобразование
    B& operator= (const A& x) {return *this;}
    operator A() {return A();}
};

void fn(B x) {}
```

```
int main () {

  A foo;
  B bar(foo);
  bar = foo;
  foo = bar;

  B bar = foo;    // conversion from «A» to non-scalar type «B» requested
  bar   = foo;    // вызывается оператор присваивания
  foo   = bar;    // вызывается оператор приведения типа

  fn(foo);        // could not convert «foo» from «A» to «B»
  fn(bar);

  return 0;
}
```

Конструкторы, помеченные как **explicit**, не могут быть вызваны с помощью синтаксиса, подобного присваиванию.

В приведенном выше примере **bar** не может быть сконструирован именно таким образом:

```
B bar = foo;
```

Функции-члены приведения типов (описанные в предыдущем разделе) также могут быть указаны как **explicit**.

Это предотвращает неявные преобразования таким же способом, как это делают для целевого типа явно заданные конструкторы.

Приведение типа

C++ является языком со строгой типизацией. Многие преобразования, особенно те, которые подразумевают различную интерпретацию значения, требуют явного преобразования, известного в C++ как приведение типов.

Существует два основных синтаксиса для приведения типов: функциональный и C-образный:

```
double x = 10.3;
int y;
y = int(x);    // функциональная нотация
y = (int)x;    // c-нотация
```

Функциональность этих универсальных форм приведения типов достаточна для большинства потребностей с основными типами данных. Однако эти операторы могут бездумно применяться к классам и указателям на классы, что может привести к коду, который, будучи синтаксически правильным, может вызвать ошибки времени выполнения.

Например, следующий код компилируется без ошибок:

Например, следующий код без ошибок компилируется:

```
#include <iostream>

class Dummy {
    double u, v;
};

class Addition {
    int i, j;
public:
    Addition(int a, int b) { i = a; j = b; }
    int result() { return i + j;}
};

int main () {

  Dummy d;

  Addition *padd;

  padd = (Addition *)&d;  // преобразование указателей на несовместимые объекты

  cout << padd->result();

  return 0;
}
```

Программа объявляет указатель на **Addition**, но затем присваивает ему ссылку на объект другого совершенно не имеющего никакого отношения к указанному типа, используя явное приведение типа:

```
padd = (Addition *)&d;
```

Ни чем неограниченное явное приведение типов позволяет преобразовывать любой указатель в любой другой тип указателя, независимо от типов, на которые они указывают.

Последующий вызов функции-члена **result** приведет к ошибке времени выполнения или другим неожиданным результатам.

Чтобы управлять преобразованиями такого типа между классами, в C++ есть четыре конкретных оператора приведения:

```
dynamic_cast;
reinterpret_cast;
static_cast;
const_cast.
```

Синтаксис их следующий:

```
dynamic_cast < new_type > ( expression )
reinterpret_cast < new_type > ( expression )
static_cast < new_type > ( expression )
const_cast < new_type > ( expression )
```

Традиционные эквиваленты приведения типов для этих выражений будут:

```
( new_type ) expression
new_type ( expression )
```

но каждый из них имеет своими особые характеристики.

dynamic_cast

dynamic_cast может использоваться только с указателями и ссылками на классы (или с **void ***).

Его цель — быть уверенным, что результат преобразования типа указывает на действительный завершенный объект целевого типа указателя.

Процесс включает повышение указателя (преобразование из указателя на производный в указатель на базовый) таким же образом, как это допускается и при неявном преобразовании.

Но **dynamic_cast** также может понижать (преобразовывать из указателя на базовый в указатель на производный) полиморфные классы (классы с виртуальными элементами), но если и только если указанный объект является действительным завершенным объектом целевого типа.

Примечание

dynamic_cast требует информация о типах времени выполнения (RTTI — Real Time Type Info) для отслеживания динамических типов. Некоторые компиляторы поддерживают данную функцию как опцию, которая по умолчанию отключена.

Чтобы проверка типов во время выполнения с использованием **dynamic_cast** была доступна, ее необходимо включить.

Например:

```
// dynamic_cast
#include <iostream>
#include <exception>

class Base { virtual void dummy() {} };
class Drvd : public Base { int a; };

int main () {

    Base *pba = new (std::nothrow) Drvd; //
    Base *pbb = new (std::nothrow) Base; //
    Drvd *pd;

    pd = dynamic_cast<Drvd*>(pba);
    if (pd == 0)     // удалось ли преобразование?
        std::cout << "Null pointer on first type-cast.\n";

    pd = dynamic_cast<Drvd*>(pbb);
    if (pd == 0)     // удалось ли преобразование?
        std::cout << "Null pointer on second type-cast.\n";

    return 0;
}
```

Приведенный выше код пытается выполнить два динамических приведения из объектов-указателей типа **Base *** (**pba** и **pbb**) к объекту-указателю типа **Drvd ***, но только первое будет успешным (см. на их соответствующие инициализации):

```
Base *pba = new Derived;
Base *pbb = new Base;
```

Несмотря на то, что оба являются указателями типа **Base ***, **pba** фактически указывает на объект типа **Drvd**, а **pbb** — на объект типа **Base**. Поэтому, когда соответствующие приведения типов выполняются с использованием оператора **dynamic_cast**, **pba** указывает на полный объект класса **Drvd**, тогда как **pbb** указывает на объект класса **Base**, который является неполным объектом класса **Drvd**.

Когда **dynamic_cast** не может привести указатель, поскольку он не является завершенным объектом требуемого класса, как это имеет место во втором преобразовании предыдущего примера, он возвращает нулевой указатель, указывающий на ошибку.

Если **dynamic_cast** используется для преобразования в ссылочный тип и преобразование невозможно, инициируется исключение типа **bad_cast**.

dynamic_cast также может выполнять другие неявные приведения, разрешенные для указателей — приведение нулевых указателей между типами указателей (даже между несвязанными классами) и приведение любого указателя любого типа к указателю **void ***.

static_cast

static_cast может выполнять преобразования между указателями на родственные классы, не только повышающие (от указателя на производный к указателю на базовый), но также и понижающие (от указателя на базовый к указателю на производный).

Во время выполнения не выполняется никаких проверок с целью убедиться, что преобразуемый объект фактически является полным объектом целевого типа. Поэтому программист должен убедиться самостоятельно, что преобразование безопасно. С другой стороны, это не приводит к накладным расходам на проверку безопасности типов, как это имет место в случае **dynamic_cast**.

```
class Base {};
class Derived : public Base {};
Base *a    = new Base;
Derived *b = static_cast<Derived *>(a);
```

С точки зрения компилятора это будет правильный код, хотя **b** будет указывать на неполный объект класса, что может привести к ошибкам во время выполнения при разыменовании.

Следовательно, **static_cast** может выполнять над указателями на классы не только преобразования, допустимые неявно, но также им противоположные.

static_cast также может выполнять все неявные преобразования (не только те, которые имеют дело с указателями на классы), а также может выполнять противоположные действия.

Он может:

- приводить из **void *** в любой тип указателя. В этом случае гарантируется, что если значение **void *** было получено путем преобразования из того же типа указателя, то полученное значение указателя будет таким же.

- приводить целые числа, значения с плавающей точкой, а также типы перечисления в типы перечисления.

Кроме того, **static_cast** также может выполнять следующее:

- явный вызов конструктора с одним аргументом или оператора преобразования;
- преобразования в rvalue ссылки;
- преобразования значения типов **enum** в целые числа или в значения с плавающей точкой;
- преобразования любого типа в **void**, оценивая и отбрасывая значение.

reinterpret_cast

reinterpret_cast преобразует любой тип указателя в любой другой тип указателя, даже из неродственных классов. Результатом операции является простая двоичная копия значения из одного указателя в другой. Разрешены все преобразования указателя — ни указанный контент, ни сам тип указателя не проверяются.

Оператор также может приводить указатели к целочисленным типам или от них. Формат, в котором целочисленное значение представляет указатель, зависит от платформы. Единственная гарантия состоит в том, что указатель, приведенный к целочисленному типу, достаточно большому, чтобы полностью его содержать (например, **intptr_t**), гарантированно сможет быть приведен обратно к допустимому указателю.

Преобразования, которые могут быть выполнены с помощью **reinterpret_cast**, но не могут быть выполнены с помощью **static_cast**, являются операциями низкого уровня, основанными на реинтерпретации двоичных представлений типов, что в большинстве случаев приводит к коду, специфичному для системы, и, следовательно, непереносимому. Например:

```
class A { /* ... */ };
class B { /* ... */ };
A *a = new A;
B *b = reinterpret_cast<B*>(a);
```

Этот код компилируется, хотя и не имеет особого смысла, поскольку теперь **b** указывает на объект совершенно не родственного и, вероятно, несовместимого класса. Разыменование **b** небезопасно.

const_cast

Этот тип приведения манипулирует константностью объекта, на который указывает указатель. Он может как устанавливать, так и удалять квалификатор. Например, чтобы передать **const** указатель на функцию, которая ожидает неконстантный аргумент:

```
// const_cast
#include <iostream>

void print(char *str) {
  std::cout << str << '\n';
}

int main () {

  const char *c = "sample text"; // .rodata
  print(const_cast<char *> (c));

  return 0;
}
```

После компиляции и запуска получаем:

```
sample text
```

Приведенный выше пример гарантированно отработает правильно, потому что функция **print** не записывает в указанный объект. Следует отметить, что удаление константности указанного объекта с целью фактической записи в него, приводит к неопределенному поведению.

typeid — позволяет проверить тип выражения

typeid (*expression* **)**

Этот оператор возвращает ссылку на постоянный объект типа **type_info**, который определен в стандартном заголовке **<typeinfo>**.

Значение, возвращаемое **typeid**, можно сравнить с другим значением, возвращенным **typeid**, используя операторы **==** и **!=** .

Также значение, возвращаемое **typeid**, может служить для получения последовательности символов с нулевым символом в конце, представляющей тип данных или имя класса, с помощью члена **name()**.

```
#include <iostream>
#include <typeinfo>

int main () {

  int *a, b;
  a = 0; b = 0;
  if (typeid(a) != typeid(b)) {
    std::cout << "a and b are of different types:\n";
    std::cout << "a is: " << typeid(a).name() << '\n';
    std::cout << "b is: " << typeid(b).name() << '\n';
  }
  return 0;
}
```

```
a and b are of different types:
a is: Pi
b is: i
```

Когда **typeid** применяется к классам, **typeid** использует технику RTTI для отслеживания типа динамических объектов. Когда **typeid** применяется к выражению, тип которого является полиморфным классом, результатом является тип самого производного завершенного объекта:

```
// typeid, polymorphic class
#include <iostream>
#include <typeinfo>
#include <exception>

class Base { virtual void f() {} };
class Derived : public Base {};

int main () {
  try {
    Base *a = new Base;
    Base *b = new Derived;
    std::cout << "a is: " << typeid(a).name() << '\n';
    std::cout << "b is: " << typeid(b).name() << '\n';
    std::cout << "*a is: " << typeid(*a).name() << '\n';
    std::cout << "*b is: " << typeid(*b).name() << '\n';
  } catch (exception& e) { std::cout << "Exception: " << e.what() << '\n'; }
  return 0;
}
```

```
a is: P4Base
b is: P4Base
*a is: 4Base
*b is: 7Derived
```

Примечание.

Строка, возвращаемая именем члена **type_info**, зависит от конкретной реализации компилятора и библиотеки.

Это не обязательно простая строка с типичным именем типа, как в компиляторе, используемом для создания этого вывода.

Следует обратить внимание, что тип, который **typeid** рассматривает для указателей, является самим типом указателя (и **a**, и **b** относятся к классу типов **Base ***). Однако когда **typeid** применяется к объектам (например, ***a** и ***b**), **typeid** возвращает их динамический тип (то есть тип их наиболее производного завершенного объекта).

Если **typeid** оценивает указатель, которому предшествует оператор разыменования (*), и этот указатель имеет нулевое значение, **typeid** генерирует исключение **bad_typeid**.

Исключения

Исключения предоставляют способ реагировать на исключительные обстоятельства (например, ошибки времени выполнения) в программах, передавая управление специальным функциям, называемым *обработчиками*.

Чтобы отловить (catch) исключения, часть кода подвергается проверке на исключения.

Это делается путем помещения этой части кода в блок **try**.

Когда в этом блоке возникает исключительная ситуация, генерируется (throw) исключение, которое передает управление обработчику исключения. Если исключение не сгенерировано, код продолжается нормально выполняться и все обработчики игнорируются.

Исключение выдается с помощью ключевого слова **throw** из блока **try**. Обработчики исключений *объявляются* с ключевым словом **catch**, которое должно быть помещено сразу после блока **try**:

```
#include <iostream>

int main () {
  try {
    throw 20;
  } catch (int e) {
    std::cout << "An exception occurred. Exception Nr. " << e << '\n';
  }
  return 0;
}
An exception occurred. Exception Nr. 20
```

Код, подлежащий проверке на исключения заключен в блок **try**. В этом примере этот код просто вызывает исключение:

```
throw 20;
```

Выражение **throw** принимает один параметр (в данном случае целочисленное значение 20), который передается в качестве аргумента обработчику исключений.

Обработчик исключений объявляется с ключевым словом **catch** сразу после закрывающей скобки блока **try**.

Синтаксис для **catch** аналогичен обычной функции с одним параметром. Тип этого параметра очень важен, так как тип аргумента, передаваемого выражением **throw**, проверяется на соответствие типу параметра, и только в том случае, если они совпадают, исключение перехватывается данным обработчиком.

Несколько обработчиков (то есть, выражений **catch**) могут быть сцеплены — каждый с другим типом параметра. Выполняется только тот обработчик, тип аргумента которого соответствует типу исключения, указанному в операторе **throw**.

Если в качестве параметра **catch** используется многоточие (**...**), данный обработчик будет перехватывать любое исключение, независимо от того, какой тип параметра у этого исключения. Это можно использовать в качестве обработчика по умолчанию, который перехватывает все исключения, не обнаруженные другими обработчиками:

Пример

```
try {
  // code here
}
catch (int param) { std::cout << "int exception"; }
catch (char param) { std::cout << "char exception"; }
catch (...) { std::cout << "default exception"; }
```

В данном случае последний обработчик будет перехватывать любое исключение, тип которого не является ни **int**, ни **char**.

После обработки исключительной ситуации выполнение программы возобновляется после блока **try-catch**, а не после оператора **throw** !.

Также возможно вложить блоки **try-catch** в более внешние блоки **try**. В таких случаях у нас есть возможность перенаправить исключение из внутреннего блока **catch** на его внешний уровень. Это делается с помощью выражения **throw;** без аргументов. Например:

```
try {
    try {
        // code here
    } catch (int n) {
        throw;
    }
} catch (...) {
  cout << "Exception occurred";
}
```

Стандартные исключения

Стандартная библиотека C++ предоставляет базовый класс, специально разработанный для объявления объектов, которые будут сгенерированы в качестве исключений. Он называется **std::exception** и определен в заголовке **<exception>**. В этом классе есть виртуальная функция-член, называемая **what()**, которая возвращает последовательность символов с нулевым символом в конце (C-строка **char** *). Эта функция может быть переопределена в производных классах с целью содержать какое-то описание исключения.

```
#include <iostream>
#include <exception>

class myexception : public exception {
  virtual const char* what() const throw() {
    return "My exception happened";
  }
} myex;

int main () {
    try {
        throw myex;
    } catch (exception& e) {
        cout << e.what() << '\n';
    }
    return 0;
}
My exception happened.
```

В данном примере обработчик перехватывает объекты исключений по ссылке, поэтому он может отлавливать классы, производные от **exception**, как это имеет место для объекта **myex** типа **myexception**.

Все исключения, создаваемые компонентами стандартной библиотеки C++, генерируют исключения, полученные из данного класса исключений:

Исключение	Описание
bad_alloc	генерируется операторами **new** on allocation failure
bad_cast	генерируется оператором **dynamic_cast** когда тот терпит неудачу при динамическом приведении
bad_exception	генерируется определенными спецификаторами динамических исключений
bad_typeid	генерируется оператором **typeid**
bad_function_call	генерируется пустыми функциональными объектами
bad_weak_ptr	генерируется оператором **shared_ptr** при передаче плохого **weak_ptr**

Также заголовок **<exception>**, выведенный из **exception**, определяет два общих типа исключений, которые могут наследоваться пользовательскими исключениями для сообщений об ошибках:

Исключение	Описание
logic_error	ошибка, связанная с внутренней логикой программы
runtime_error	ошибка, обнаруженная во время выполнения

Типичный пример, когда нужно проверять стандартные исключения, относится к распределению памяти:

```
#include <iostream>
#include <exception>

int main () {

  try   {
    int* myarray= new int[1000];
  } catch (exception& e) {
    cout << "Standard exception: " << e.what() << endl;
  }
  return 0;
}
```

Исключением, которое может быть перехвачено обработчиком исключений в этом примере, является **bad_alloc**. Поскольку **bad_alloc** является производным от стандартного исключения базового класса, его можно перехватить (захват по ссылке, захват всех связанных классов).