Неявное преобразование типов данных

Значение переменной хранится в виде последовательности бит, а тип переменной указывает компилятору, как интерпретировать эти биты в соответствующие значения. Разные типы данных могут представлять одно значение по-разному, например, значение `4` типа `int` и значение `4.0` типа `float` хранятся как совершенно разные двоичные шаблоны, то есть обычное копирование из `int` в `float` без преобразования будет ошибочно.

`float f = 4;` // инициализация переменной типа с плавающей точкой целым числом 4 с преобразованием

Процесс конвертации значений из одного типа данных в другой называется преобразованием типов. Преобразование типов может выполняться в следующих случаях:

1) Присваивание или инициализация переменной значением другого типа данных:

```
double k(4);  // инициализация переменной типа double целым числом 4
k = 7;  // присваиваем переменной типа double целое число 7
```

2) Передача значения в функцию, где другой тип параметра:

```
void doSomething(long l){}
doSomething(4);  // передача числа 4 (тип int) в функцию с параметром типа long
```

3) Возврат из функции, где другой тип возвращаемого значения:

```
float doSomething(){
    return 4.0;  // 4.0 (тип double) из функции, которая возвращает float
}
```

4) Использование бинарного оператора с операндами разных типов:

```
double division = 5.0 / 4;  // операция деления со значениями типов double и int
```

Есть 2 основных способа преобразования типов:

1) Неявное преобразование типов, когда компилятор автоматически конвертирует один фундаментальный тип данных в другой.

2) Явное преобразование типов, когда разработчик использует один из операторов явного преобразования для выполнения конвертации объекта из одного типа данных в другой.

Неявное преобразование типов (или «автоматическое преобразование типов») выполняется всякий раз, когда требуется один фундаментальный тип данных, но предоставляется другой, и пользователь не указывает компилятору, как выполнить конвертацию (не использует явное преобразование типов через операторы явного преобразования).

Есть 2 основных способа неявного преобразования типов: числовое расширение и числовая конверсия.

Числовое расширение возникает когда значение из одного типа данных конвертируется в другой тип данных побольше (по размеру и по диапазону значений). Например, тип `int` может быть расширен в тип `long`, а тип `float` может быть расширен в тип `double`. Числовые расширения делятся на два типа:

- Интегральное расширение (или «целочисленное расширение»). Включает в себя преобразование целочисленных типов, меньших, чем `int` (`bool`, `char`, `unsigned char`, `signed char`, `unsigned short`, `signed short`) в `int` (если это возможно) или `unsigned int`.

- Расширение типа с плавающей точкой. Конвертация из типа `float` в тип `double`.

Числовые расширения всегда безопасны и не приводят к потере данных.

Числовые конверсии возникают когда преобразуется значение из более крупного типа данных в аналогичный, но более мелкий тип данных, или конвертация происходит между разными типами данных. В отличие от расширений, которые всегда безопасны, конверсии могут (но не всегда) привести к потере данных. Поэтому в любой программе, где выполняется неявная конверсия, компилятор будет выдавать предупреждение.

```
double d = 4; // конвертирование 4 (тип int) в double
short s = 3; // конвертирование 3 (тип int) в short
```

Во всех случаях, когда происходит конвертация значения из одного типа данных в другой, который не имеет достаточного диапазона для хранения конвертируемого значения, результаты будут неожиданные. Поэтому делать так не рекомендуется.

Например:

```
int i = 30000;
char c = i;
cout << static_cast<int>(c) << endl; // >> 48
```

В примере происходит присвоения огромное целочисленное значение типа int переменной типа `char` (диапазон которого составляет от `-128` до `127`). Это приведет к переполнению и выводу числа `48`. Если число подходит по диапазону, конвертация пройдет успешно. Например:

```
int i = 3;
short s = i; // конвертируем значение типа int в тип short
cout << s << endl; // >> 3

double d = 0.1234;
float f = d; // конвертируем значение типа double в тип float
cout << f << endl; // >> 0.1234
```

В случаях со значениями типа с плавающей точкой могут произойти округления из-за худшей точности в меньших типах. В этом случае возникает потеря в точности, так как точность типа `float` меньше, чем типа `double`. Например:

```
#include <iomanip> // для std::setprecision()

float f = 0.123456789; // значение типа double - 0.123456789 имеет 9 значащих цифр, но float может хранить только 7
cout << std::setprecision(9) << f << endl; // >>0.123456791
```

При обработке выражений компилятор разбивает каждое выражение на отдельные подвыражения. Арифметические операторы требуют, чтобы их операнды были одного типа данных. Чтобы это гарантировать, компилятор использует следующие правила:

- Если операндом является целое число меньше (по размеру/диапазону) типа `int`, то оно подвергается интегральному расширению в `int` или в `unsigned int`.
- Если операнды разных типов данных, то компилятор вычисляет операнд с наивысшим приоритетом и неявно конвертирует тип другого операнда в такой же тип, как у первого.

Приоритет типов операндов:

`long double` (самый высокий)
`double`
`float`
`unsigned long long`
`long long`
`unsigned long`
`long`
`unsigned int`
`int` (самый низкий)

Чтобы узнать решающий тип в выражении можно использовать оператор `typeid` (находится в заголовочном файле `typeinfo`). Например:

```
#include <typeinfo> // для typeid()

short x(3);
short y(6);
cout << typeid(x + y).name() << endl; // решающий тип данных в выражении x + y будет i ( int )
```

Поскольку значениями переменных типа `short` являются целые числа и тип `short` меньше (по размеру/диапазону) типа `int`, то он подвергается интегральному расширению в тип `int`. Результатом сложения двух `int`-ов будет тип `int`.

Ещё пример:

```
double a(3.0);
short b(2);
cout << typeid(a + b).name() << endl; // итоговый тип double
```

Здесь `short` подвергается интегральному расширению в `int`. Однако `int` и `double` по-прежнему не совпадают. Поскольку `double` находится выше в иерархии типов, то целое число `2` преобразовывается в `2.0` (тип `double`), и сложение двух чисел типа `double` дадут число типа `double`.

Также могут возникать неожиданное поведение, например:

```
cout << 5u - 10 << endl; // 5u означает значение 5 типа unsigned int
```

Ожидается, что результатом выражения `5u - 10` будет `-5`, поскольку `5 - 10 = -5`, но результат: `4294967291`
Здесь значение `signed int (10)` подвергается расширению в `unsigned int`, которое имеет более высокий приоритет, и выражение вычисляется как `unsigned int`. А поскольку `unsigned` — это только положительные числа, то происходит переполнение.

Ключевое слово explicit

Иногда выполнение неявных преобразований может иметь смысл, а иногда может быть крайне нежелательным и генерировать неожиданные результаты:

```
class SomeString{
    std::string m_string;
public:
    SomeString(int a){  // выделяется строка размером a
        m_string.resize(a);
    }

    SomeString(const char *string){  // выделяется строка для хранения значения типа string
        m_string = string;
    }

    friend std::ostream& operator<<(std::ostream& out, const SomeString
&s);
};

std::ostream& operator<<(std::ostream& out, const SomeString &s){
    out << s.m_string;
    return out;
}
…
SomeString mystring = ‘a'; // выполняется копирующая инициализация
cout << mystring << endl;
```

В примере, приводится попытка инициализировать строку одним символом типа `char`. Поскольку переменные типа `char` являются частью семейства целочисленных типов, то компилятор будет использовать конструктор преобразования `SomeString(int)` для неявного преобразования символа типа `char` в тип `SomeString`. В результате переменная типа `char` будет конвертирована в тип `int`. А это не совсем то, что ожидается.

Один из способов решения этой проблемы — сделать конструктор явным, используя ключевое слово `explicit` (которое пишется перед именем конструктора). Явные конструкторы (с ключевым словом `explicit`) не используются для неявных конвертаций:

Пример с использованием ключевого слова `explicit:`

```
class SomeString{
    std::string m_string;
public:
    explicit SomeString(int a){  // выделяется строка размером a
        m_string.resize(a);
    }

    SomeString(const char *string){
        m_string = string;
    }

    friend std::ostream& operator<<(std::ostream& out, const
SomeString &s);
};

std::ostream& operator<<(std::ostream& out, const SomeString &s){
    out << s.m_string;
    return out;
}
…
SomeString mystring = 'a'; // ошибка компиляции
cout << mystring << endl;
```

Вышеприведенная программа не скомпилируется, так как `SomeString(int)` явный, а другого конструктора преобразования, который выполнил бы неявную конвертацию `'a'` в `SomeString`, компилятор просто не нашел. Однако использование явного конструктора только предотвращает выполнение неявных преобразований. Явные конвертации (через операторы явного преобразования) по-прежнему разрешены:

```
cout << static_cast<SomeString>('a'); // разрешено: явное преобразование 7 в SomeString через оператор static_cast
```

При прямой или `uniform` инициализации неявная конвертация также будет выполняться:

```
SomeString str('a'); // разрешено
```

Еще одним способом запретить конвертацию `'a'` в `SomeString` (неявным или явным способом) является добавление закрытого конструктора `SomeString(char)`:

```
class SomeString{
    std::string m_string;
    SomeString(char){} // объекты типа SomeString(char) не могут быть созданы вне класса
public:
    SomeString(int a){ // выделяется строка размером a
        m_string.resize(a);
    }

    SomeString(const char *string){ // выделяется строка для хранения значения типа string
        m_string = string;
    }

    friend std::ostream& operator<<(std::ostream& out, const SomeString &s);
};

std::ostream& operator<<(std::ostream& out, const SomeString &s){
    out << s.m_string;
    return out;
}

…

SomeString mystring('a'); // ошибка компиляции
cout << mystring << endl;
```

Лучшее решение — использовать ключевое слово `delete` (добавленное в C++11) для удаления этого конструктора:

```
class SomeString{
    std::string m_string;
public:
    SomeString(char) = delete; // использование этого конструктора приведет к ошибке
    SomeString(int a){ // выделяется строка размером a
        m_string.resize(a);
    }

    SomeString(const char *string){ // выделяется строка для хранения значения типа string
        m_string = string;
    }

    friend std::ostream& operator<<(std::ostream& out, const SomeString &s);
};

std::ostream& operator<<(std::ostream& out, const SomeString &s){
    out << s.m_string;
    return out;
}
…
SomeString mystring('a'); // ошибка компиляции
cout << mystring << endl;
```

Или ещё один пример — можно объявить класс `Rational`:

```
class Rational {
    int m_num = 0;
public :
    Rational (int num): m_num(num){}
    const Rational operator*(const Rational& rhs)
const
    {
        return Rational(m_num * rhs.m_num);
    }
};
```

```
Rational r1(5);
Rational r2 = temp * 2;// неявное преобразование к Rational(2)
```

Если конструктор `Rational` пометить как `explicit`, то неявное преобразование вызовет ошибку времени компиляции:

```
class Rational {
    int m_num = 0;
public :
    explicit Rational (int num): m_num(num){}
    const Rational operator*(const Rational& rhs)
const
    {
        return Rational(m_num * rhs.m_num);
    }
};


Rational r1(5);
Rational r2 = temp * 2; // ошибка, невозможное преобразование
```

Полиморфизм

Полиморфизм (происходит от греческих слов «poly» - много и «morphos» - форма) – это свойство программного кода изменять свое поведение в зависимости от ситуации, возникающей при выполнении программы. В C++ полиморфизм возможен в нескольких случаях: использование виртуальных функций, перегрузки функций, перегрузки операторов.

В рамках концепции полиморфизма, в C++ доступна перегрузка (`overload`) функций и методов. Компилятор различает «перегруженные» функции только по списку полученных параметров а не по возвращаемому значению. Списки параметров перегруженных функций должны отличаться по количеству параметров или если количество параметров одинаковое, то по типам параметров. Например:

```
// функция Average с двумя параметрами типа int
int Average(int a, int b){ … }

// функция Average с тремя параметрами типа int
int Average(int a, int b, int c){ … }

// функция Max с двумя параметрами типа double
int Average(double a, double b){ … }
```

Перегрузка функций/методов это статический полиморфизм. Компилятор распознает перегруженные функции только по получаемым параметрам. Если две функции имеют одинаковые имена, одинаковое количество и типы параметров но возвращают разные значения, то такие функции считаются одинаковыми. В этом случае компилятор выдаст ошибку:

```
// функция Average с двумя параметрами типа int
int Average(int a, int b){ … }

// функция Max с двумя параметрами типа int, возвращающая double
double Average(int a, int b){ … }

…

// error: functions that differ only in their return type cannot be
```

Наследование

Наследование позволяет задействовать функциональность базового класса (`base class`) в другом - производном классе (`subclass`). Наследование полезно, поскольку оно позволяет структурировать и повторно использовать код, что, в свою очередь, может значительно ускорить процесс разработки. В C++ наследование достигается специальным синтаксисом:

```
// базовый класс
class Person {
    public:
        string name;
        int age;
};

// производный класс
class Student : public Person{
    public:
        int yearOfStudy;
};
```

В языке C++ есть третий спецификатор доступа, о который еще не упоминался, так как он полезен только в контексте наследования. Спецификатор доступа protected открывает доступ к членам класса дочерним классам. Доступ к protected-члену вне тела класса закрыт.

```
// базовый класс
class Person {
    public:
              string name;
    private:
              int age;
   protected:
              float height;
};

// производный класс
class Student : public Person{
      public:
            Student(){
                name = "Student"; // разрешено
                age = 18;         // запрещено
                height = 1.75f;   // разрешено
            }
};

int main(){
    Person person;
    person.name = "Student"; // разрешено
    person.age = 18;         // запрещено
    person.height = 1.75f;   // запрещено
}
```

В C++ есть несколько типов наследования:

Публичный (`public`) тип наследования - публичные (`public`) и защищенные (`protected`) данные наследуют без изменения уровня доступа к ним.

Защищенный (`protected`) - все унаследованные данные становятся защищенными.

Приватный (`private`) - все унаследованные данные становятся приватными.

В C++ конструкторы и деструкторы не наследуются. Однако они вызываются, когда дочерний класс инициализирует свой объект. Конструкторы вызываются один за другим иерархически, начиная с базового класса и заканчивая последним производным классом. Деструкторы вызываются в обратном порядке.

```
// базовый класс
class Person {
    public:
            Person(){ cout << "Person" << endl; }
            ~Person(){ cout << "delete Person" << endl; }

            string name;
            int age;
};

// производный класс
class Student : public Person{
      public:
            Student(){ cout << "Student" << endl; }
            ~Student(){ cout << "delete Student" << endl; }

            int yearOfStudy;
};
```

Output:

```
>> Person
>> Student
>> delete Student
>> delete Person
```

Виртуальные функции

Виртуальные функции — специальный вид методов класса. Виртуальная функция отличается об обычной функции тем, что для обычной функции связывание вызова функции с её определением осуществляется на этапе компиляции. Для виртуальных функций это происходит во время выполнения программы (динамическое связывание). Виртуальные функции можно отнести к динамическому полиморфизму, так как компилятор заранее не имеет данных о том, какую функцию вызовет программа.

Виртуальная функция определяется в базовом классе, а любой порожденный класс может её переопределить. Вызов виртуальной функции возможен только через указатель или ссылку на базовый класс. Для объявления виртуальной функции используется ключевое слово virtual. Метод класса может быть объявлен как виртуальный, если класс базовый в иерархии порождения и если реализация метода зависит от класса и будет различной в каждом порожденном классе. Пример:

```
class Person {
    public:
          string name;
          int age;
          virtual void say() { cout << "Im Person" << endl; }
};


class Student : public Person{
     public:
          int yearOfStudy;
          virtual void say() override {
              cout << "Im Student" << endl;
          }
};
```

Модификатор `override` появился в стандарте C++11 и используется при описании методов дочернего класса. Модификатор `override` следует писать для тех методов, которые по задумке являются переопределенными методами базового класса. В реализации (т. е. в `cpp`-файле), модификатор `override` не пишется. Модификатор `override` позволяет компилятору следить за тем, чтобы метод, помеченный этим модификатором действительно переопределял метод базового класса. В языке C++ метод дочернего класса будет переопределять метод базового класса только в том случае, если его сигнатура полностью совпадает с сигнатурой базового класса. Можно совершить ошибку, например, в одной сигнатуре используется обычный указатель, а в другой константный указатель, таким образом не совершив переопределения желаемого метода.

Существует одно исключение из правила, когда тип возврата переопределения может не совпадать с типом возврата виртуального метода родительского класса, но при этом оставаться переопределением. В переопределенных методах в качестве возвращаемого типа можно вместо указателя на родительский класс возвращать указатель на дочерний класс. Это называется ковариантным типом возврата:

```
class Person {
    public:
        string name;
        int age;
        virtual Person* getPerson() { return this; }
};

// производный класс
class Student : public Person{
    public:
        int yearOfStudy;
        virtual Student* getPerson() override { return
        this; }
}
```

Модификатор `final` используется в случае, если необходимо запретить дальнейшее переопределение виртуальной функции или наследование определенного класса. Если пользователь пытается переопределить метод или наследовать класс с модификатором final, то компилятор выдаст ошибку.
Модификатор `final` указывается в том же месте, что и `override`, например:

```
class Student : public Person{
    public:
        int yearOfStudy;
        virtual Student* getPerson() override final
        { return this; }
}

class Programmer : public Student{
    public:
        // ошибка, переопределение запрещено !
        virtual Programmer* getPerson() override
        { return this; }
}
```

Пример использования модификатора final, касательно наследования:

```
class Student final : public Person{
    public:
        int yearOfStudy;
        virtual Student* getPerson() override
        final { return this; }
}
```

```
// ошибка, нельзя наследовать final-класс
class Programmer : public Student{
}
```

В языке программирования C++ деструктор полиморфного базового класса должен быть виртуальным. Только так обеспечивается корректное разрушение объекта производного класса через указатель на соответствующий базовый класс.

```
class Person {
public:
    virtual ~Person(){ }
};
```

```
class Student : public Person{
public:
    ~Student(){ }
};

Person *person = new Student;
delete person;
```

Дружественные функции и классы

Доступ к данным класса обычно скрываются с помощью `private`. Однако может возникнуть ситуация, когда у вас есть класс и функция, которая работает с этим классом, но которая не находится в его теле. Например, есть класс, в котором хранятся данные, и функция (или другой класс), которая выводит эти данные на экран. Хотя код класса и код функции вывода разделены для упрощения поддержки кода, код функции вывода тесно связан с данными класса. Следовательно, сделав члены класса private, получить к ним доступ из другого места будет невозможно.

В таких ситуациях есть два варианта:

1) Сделать открытыми методы класса и через них функция будет взаимодействовать с классом (так называемые геттеры (getter) и сеттеры (setter)). Однако здесь есть несколько нюансов. Во-первых, эти открытые методы нужно будет определить, на что потребуется время, и они будут загромождать интерфейс класса. Во-вторых, в классе нужно будет открыть методы, которые не всегда должны быть открытыми и предоставляющими доступ извне.

2) Использовать дружественные классы и дружественные функции, с помощью которых можно будет предоставить функции вывода доступ к закрытым данным класса. Это позволит функции вывода напрямую обращаться ко всем закрытым переменным-членам и методам класса, сохраняя при этом закрытый доступ к данным класса для всех остальных функций вне тела класса.

Дружественная функция — это функция, которая имеет доступ к закрытым членам класса, как если бы она сама была членом этого класса. Во всех других отношениях дружественная функция является обычной функцией. Ею может быть, как обычная функция, так и метод другого класса. Для объявления дружественной функции используется ключевое слово friend перед прототипом функции, которую вы хотите сделать дружественной классу. Неважно, объявляетется ли она в `public` или в `private` зоне класса.

```
class Anything{
    int m_value;
public:
    Anything() { m_value = 0; }
    void add(int value) { m_value += value; }
     // функцию reset() дружественная классу Anything
    friend void reset(Anything &anything);
};

// Функция reset() теперь является дружеской классу Anything
void reset(Anything &anything){
    anything.m_value = 0;  // доступ к закрытым членам объектов класса Anything
}

…

Anything one;
one.add(4);
reset(one);
```

Функцию `reset()` принимает объект класса `Anything` и устанавливает `m_value` значение `0`. Поскольку `reset()` не является членом класса `Anything`, то в обычной ситуации функция `reset()` не имела бы доступ к закрытым членам `Anything`. Однако, поскольку эта функция является дружественной классу `Anything`, она имеет доступ к закрытым членам `Anything`.

Функция может быть другом сразу для нескольких классов, например:

```
class Humidity;

class Temperature
{
    int m_temp;
public:
    Temperature(int temp=0) { m_temp = temp; }

    friend void outWeather(const Temperature &temperature, const Humidity &humidity);
};

class Humidity
{
    int m_humidity;
public:
    Humidity(int humidity=0) { m_humidity = humidity; }

    friend void outWeather(const Temperature &temperature, const Humidity &humidity);
};

void outWeather(const Temperature &temperature, const Humidity &humidity)
{
    std::cout << "The temperature is " << temperature.m_temp <<
       " and the humidity is " << humidity.m_humidity << '\n';
}
```

….

Temperature temp(15);
Humidity hum(11);
outWeather(temp, hum);

Один класс может быть дружественным другому классу. Это откроет всем членам первого класса доступ к закрытым членам второго класса, например:

```
class Values
{
    int m_intValue;
    double m_dValue;
public:
    Values(int intValue, double dValue)
    {
        m_intValue = intValue;
        m_dValue = dValue;
    }
    friend class Display;   // класс Display друг класса Values
};

class Display
{
    bool m_displayIntFirst;
public:
    Display(bool displayIntFirst) { m_displayIntFirst = displayIntFirst; }

    void displayItem(Values &value)
    {
        if (m_displayIntFirst)
            std::cout << value.m_intValue << " " << value.m_dValue << '\n';
        else
            std::cout << value.m_dValue << " " << value.m_intValue << '\n';
    }
};

…

Values value(7, 8.4);
Display display(false);
display.displayItem(value);
```

Во-первых, даже несмотря на то, что `Display` является другом `Values`, `Display` не имеет прямой доступ к указателю `*this` объектов `Values`.
Во-вторых, даже если `Display` является другом `Values`, это не означает, что `Values` также является другом `Display`. Если вы хотите сделать оба класса дружественными, то каждый из них должен указать в качестве друга противоположный класс.
Наконец, если класс `A` является другом `B`, а `B` является другом `C`, то это не означает, что `A` является другом `C`.

Следует быть внимательным при использовании дружественных функций и классов, поскольку это может нарушать принципы инкапсуляции. Если детали одного класса изменятся, то детали класса-друга также будут вынуждены измениться. Следовательно, лучше ограничивать количество и использование дружественных функций и классов.

Вместо того, чтобы делать дружественным целый класс, можно сделать дружественными только определенные методы класса. Их объявление аналогично объявлениям обычных дружественных функций, за исключением имени метода с префиксом `ClassName::` в начале, например `Display::displayItem()`.

```
class Display;  // предварительное объявление класса Display

class Values
{
	int m_intValue;
	double m_dValue;
public:
	Values(int intValue, double dValue)
	{
		m_intValue = intValue;
		m_dValue = dValue;
	}

	// метод Display::displayItem() друг класса Values
	friend void Display::displayItem(Values& value);
};

class Display
{
	bool m_displayIntFirst;
public:
	Display(bool displayIntFirst) { m_displayIntFirst = displayIntFirst; }

	void displayItem(Values &value)
	{
		if (m_displayIntFirst)
			std::cout << value.m_intValue << " " << value.m_dValue << '\n';
		else
			std::cout << value.m_dValue << " " << value.m_intValue << '\n';
	}
};
```

Инициализация в C++

В C++ многое унаследовано от C. В C есть несколько способов инициализации переменных. Их можно вообще не инициализировать, и это называется инициализация по умолчанию, в таком случае данные, которые будут лежать в переменной не будут детерминированы. При обращении к такой переменной возникает неопределённое поведение. То же касается пользовательских типов: если в некотором `struct` есть неинициализированные поля, то при обращении к ним также возникает неопределённое поведение.

В C++ было добавлено множество новых конструкций: классы, конструкторы, public, private, методы, но ничто из этого не влияет на только что описанное поведение. Если в классе некоторый элемент не инициализирован, то при обращении к нему возникает неопределённое поведение. В C++98 можно инициализировать переменные при помощи member initializer list (списки инициализации). Объявление списков происходит в конструкторе после скобок с набором аргументов.

```
// C++98: member initialiser list

class Widget {
  public:
    Widget() : i(0), j(0) {} // member initialiser list

  private:
    const int i;
    int j;
};

…

Widget widget;
```

Инициализаторы элементов по умолчанию

В C++11 были добавлены инициализаторы элементов по умолчанию (direct member initializers), которыми пользоваться значительно удобнее. Они позволяют инициализировать все переменные одновременно.

```
// C++11: default member initialisers

class Widget {
  public:
    Widget() {}

  private:
    const int i = 0; // default member initialisers
    int j = 0;
};

…

Widget widget;
```

Копирующая инициализация

При копирующей инициализации переменной через знак равенства указывается её значение.

```
int i = 2; // copy initialization
```

Копирующая инициализация также используется, когда аргумент передаётся в функцию по значению, или когда происходит возврат объекта из функции по значению.

```
int square(int i) {
  return i * i;
}
int s = square(10);
```

Знак равенства может создать впечатление, что происходит присвоение значения, но это не так. Копирующая инициализация — это не присвоение значения.

Важное свойство копирующей инициализации: если типы значений не совпадают, то выполняется последовательность преобразования (`conversion sequence`). У последовательности преобразования есть определенные правила, например, она не вызывает explicit конструкторов, поскольку они не являются преобразующими конструкторами. Поэтому, если выполнить копирующую инициализацию для объекта, конструктор которого отмечен как explicit, происходит ошибка компиляции.

```
class Widget {
    int val;
    explicit Widget(int a_val): val(a_val) {}
};

Widget w1 = 1; // ERROR
```

Агрегатная инициализация

Агрегатная инициализация выполняется, когда массив инициализируется рядом значений в фигурных скобках:

```
int i[4] = {0, 1, 2, 3};
```

Если при этом не указать размер массива, то он выводится из количества значений, заключённых в скобки:

```
int j[] = {0, 1, 2, 3}; // array size deduction
```

Эта же инициализация используется для агрегатных (`aggregate`) классов, то есть таких классов, которые являются просто набором публичных элементов:

```
struct Widget {
   int i;
   float j;
};

Widget widget = {1, 3.14159};
```

Такой синтаксис работал ещё в C и C++98, начиная с C++11, в нём можно пропускать знак равенства:

```
Widget widget{1, 3.14159};
```

Агрегатная инициализация на самом деле использует копирующую инициализацию для каждого элемента. Поэтому, если попытаться использовать агрегатную инициализацию (как со знаком равенства, так и без него) для нескольких объектов с explicit конструкторами, то для каждого объекта выполняется копирующая инициализация и произойдёт ошибка компиляции:

```
struct Widget {
private:
  int val;
public:
  explicit Widget(int a_val): val(a_val) {}
};

struct Thingy {
  Widget w1, w2;
};

…

Thingy thingy = {3, 4}; // ERROR
Thingy thingy {3, 4}; // ERROR
```

Если для этих объектов есть другой конструктор, не-explicit, то вызывается он, даже если он хуже подходит по типу:

```
struct Widget {
private:
  int val;
public:
  explicit Widget(int a_val): val(a_val) {}
  Widget(double a_val): val(a_val) {}
};

struct Thingy {
  Widget w1, w2;
};

…

Thingy thingy = {3, 4}; // вызывает Widget(double)
Thingy thingy{3, 4}; // вызывает Widget(double)
```

Если при агрегатной инициализации пропустить некоторые элементы в массиве значений, то соответствующим переменным присваивается значение нуль. Это очень полезное свойство, потому что благодаря нему никогда не может быть неинициализированных элементов. Такое свойство работает как с классами, так и с массивами.

```
struct Widget {
  int i;
  int j;
};

…

Widget widget = {1};
std::cout << widget.j << std::endl;

…

int[100] = {}; // все элементы инициализируются нулями
```

При агрегатной инициализации возможен пропуск скобок (brace elision). Данные будут разложены в порядке объявления. Структура Widget в данном случае является подагрегатом (subaggregate), то есть с вложенным агрегатным классом.

```
struct Widget {
  int i;
  int j;
};

struct Thingy {
  Widget w;
  int k;
};

…

Thingy t_12_3 = {{1, 2}, 3}; // i: 1, j: 2, k: 3
Thingy t123 = {1, 2, 3}; // i: 1, j: 2, k: 3
Thingy t12 = {1, 2}; // i: 1, j: 2, k: 0
```

Статическая инициализация

От C также унаследована статическая инициализация: статические переменные всегда инициализируются. Это может быть сделано несколькими способами. Статическую переменную можно инициализировать выражением-константой. В этом случае инициализация происходит во время компиляции. Если же переменной не присвоить никакого значения, то она инициализируется значением нуль:

```
static int i = 3;     // инициализация константой
static int j;         // инициализация нулем

int main() {
  std::cout << i << std::endl; // 3
  std::cout << j << std::endl; // 0
}
```

Прямая инициализация

Наиболее важная возможность, отличающая C++ от C — это конструкторы. При помощи этого же синтаксиса можно инициализировать встроенные типы вроде int и float. Этот синтаксис называется прямой инициализацией. Она выполняется всегда, когда есть аргумент в круглых скобках. Для встроенных типов (int, bool, float) никакого отличия от копирующей инициализации здесь нет. Если же речь идёт о пользовательских типах, то, в отличие от копирующей инициализации, при прямой инициализации можно передавать несколько аргументов.

```
Widget widget(1, 2);
int i_var(3);
float f_var(10.0f);
double d_var(10.0);
```

При прямой инициализации не выполняется последовательность преобразования. Вместо этого происходит вызов конструктора при помощи разрешения перегрузки (overload resolution). У прямой инициализации тот же синтаксис, что и у вызова функции, и используется та же логика, что и в других функциях C++. В ситуации с explicit конструктором прямая инициализация работает нормально, хотя копирующая инициализация выдаёт ошибку:

```
struct Widget {
  explicit Widget(int) {}
};

Widget w1 = 1; // ошибка
Widget w2(1);  // происходит вызов конструктора
```

В ситуации с двумя конструкторами, один из которых explicit, а второй хуже подходит по типу, при прямой инициализации вызывается первый, а при копирующей — второй. В такой ситуации изменение синтаксиса приведёт к вызову другого конструктора, такое неочевидное поведение может стать причиной ошибки:

```
struct Widget {
  explicit Widget(int) {}
  Widget(double) {}
};

…

Widget w2(1);   // вызывает Widget(int)
Widget w1 = 1; // вызывает Widget(double)
```

Универсальная инициализация

C++ очень много различных синтаксисов инициализации с разным поведением, которое может доставить не мало трудностей в том или ином случае. Например агрегатную инициализацию можно было использовать только с массивами, но не с контейнерами вроде `std::vector`. Создатели языка попытались решить проблему в C++11, введя синтаксис с фигурными скобками. Используемый для инициализации список называется `braced-init-list`. Этот список не является объектом, у него нет типа. Переход на C++11 с более ранних версий не создаёт никаких проблем с агрегатными типами, так что это изменение не является критическим.

```
Widget widget{1, 2};  // direct-list-initialization
Widget widget = {1, 2};  // copy-list-initialization
```

Теперь у списка в фигурных скобках появились новые возможности. Хоть у него и нет типа, он может быть скрыто преобразован в `std::initializer_list`, это такой специальный новый тип. Если есть конструктор, принимающий на вход `std::initializer_list`, то вызывается именно этот конструктор:

```
template <typename T>
class vector {
    vector(std::initializer_list<T> init); // конструктор с initializer_list
};

…

std::vector<int> vec{0, 1, 2, 3, 4}; // вызывает этот конструктор
```

Тип `std::initializer_list` — это вектор фиксированного размера с элементами `const`. То есть это тип, у него есть функции `begin` и `end`, которые возвращают итераторы. У `std::initializer_list` есть собственный тип итератора, и чтобы его использовать, нужно включить специальный заголовок `<initializer_list>`:

```
#include <iostream>
#include <initializer_list>

struct Widget{
    S( std::initializer_list<int> lt ){
        for( auto n = lt.begin() ; n < lt.end() ; n++ ){
            std::cout << *n << std::endl;
        }
    }
};

int main() {
    S s{1, 2, 3, 4};
}
```

Синтаксис с `std::initializer_list` не редко становится источником ошибок и недопонимания. Например:

```
std::vector<int> v(3, 0); // вектор содержит 0, 0, 0
std::vector<int> v{3, 0}; // вектор содержит 3, 0

std::string s(48, 'a'); // «aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa"
std::string s{48, 'a'}; // "0a", так как ASCII число 48 код символа "0"
```

В итоге в новых стандартах инициализация списком выполняется следующим образом.

- Для агрегатных типов при такой инициализации выполняется агрегатная инициализация.
- Для встроенных типов выполняется прямая инициализация ({a}) или копирующая инициализация (= {a});
- Для классов выполняется такая последовательность:

1) Вначале выполняется вызов конструктора, который принимает `std::initializer_list`.
2) Если для этого вызова необходимо сделать неочевидные преобразования — они выполняются.
3) Если подходящего конструктора нет, выполняется обычный
4) вызов конструктора () при помощи разрешения перегрузки.

Для второго шага есть пара исключений:
1) При использовании = {a}, когда в списке один элемент a, может быть использована инициализация копированием.
2) Пустые фигурные скобки, {}.
Если имеется тип с конструктором по умолчанию и конструктором, который принимает `initializer_list`, то при инициализации с пустыми фигурными скобами предпочтение будет отдано конструктору по умолчанию.

```
template Typename<T>
struct Widget {
  Widget();
  Widget(std::initializer_list<T>);
};

int main() {
  Widget<int> w{}; // вызов конструктора по умолчанию
}
```

У инициализации списком есть полезное свойство: не допускаются преобразования, сужающие диапазон значений (`narrowing conversions`). Если для инициализации int использовать double, это является сужающим преобразованием, и такой код не компилируется:

```
int main() {
  int i{2.0}; // ошибка!
}
```

То же самое происходит, если агрегатный объект, содержащий в себе переменные типа `int`, инициализировать списком элементов `double`:

```
struct Widget {
   int i;
   int j;
};
```

```
int main() {
   Widget widget = {1.0, 0.0}; // ошибка в C++11 в отличие от C++98/03
}
```

При инициализации списком можно использовать вложенные фигурные скобки, но, в отличие от агрегатной инициализации, с ними не работает пропуск скобок (`brace elision`). Например, такой синтаксис будет работать для контейнера `map`. Внешние фигурные скобки инициализируют этот `map`, а внутренние фигурные скобки — его элементы:

```
std::map<std::string, std::int> m_map {{"abc",
0}, {"def", 1}};
```