Семинар #1: Основы C++

Пространства имён, ссылки, перегрузка функций, `std::string` и `std::vector`.

Язык C++ создан на основе языка C и одним из приоритетов языка является обратная совместимость с C. Поэтому почти любая программа на языке C будет работать и на языке C++. Однако, нужно помнить, что это разные языки. В C++ было добавлено огромное количество новых возможностей, что сделало этот язык, вероятно, языком с самым большим количеством возможностей.

Для файлов исходного кода на языке C++ будем использовать расширение `.cpp`, а для заголовочных файлов расширение `.hpp`. Для компиляции программ на C++ будем использовать компилятор `g++`:

```
g++ main.cpp
```

Все библиотеки языка C доступны и в C++. Но для каждой библиотеки языка C в языке C++ есть своя (немного изменённая) версия этой библиотеки. Подключить эти библиотеки можно, используя те же названия, что и при подключении этих библиотек в C. Но лучше использовать новые имена для этих библиотек, чтобы подчеркнуть, что это именно новая версия библиотеки из языка C. Для библиотеки `stdio.h` этим новым именем является `cstdio`. Для библиотеки `math.h` – `cmath`, для `stdlib.h` – `cstdlib` и так далее.

```
#include <cstdio>
int main()
{
    printf("Hello World\n");
}
```

Нужно отметить, что работа директивы `#include` не отличается в языках C и C++. А `cstdio` является просто текстовым файлом (без расширения), лежащим в системной папке на диске.

Пространство имён (англ. *namespace*)

Можем создавать пространство имён и писать внутри него код. Все объявленные внутри пространства имён сущности (например, переменные, функции, структуры) не видны напрямую вне пространства имён. Обратиться к ним можно по другому имени, состоящему из названия пространства имён, двух двоеточий и имени сущности:

```
#include <cstdio>
namespace mipt
{
    int a = 10;
    int square(int x)
    {
        return x * x;
    }
    struct point
    {
        int x, y;
    };
}

int main()
{
    mipt::point p = {20, 30};
    printf("%i\n", mipt::square(mipt::a + p.x));
}
```

Кстати, обратите внимание, что в языке C++, в отличии от C, перед именем типа структуры не обязательно писать слово `struct`:

```
struct point
{
    int x, y;
};

int main()
{
    struct point p1;  // В C нужно было обязательно писать слово struct
    point p2;         // В C++ можно не писать
}
```

Зачем нужны пространства имён?

Представьте, что вы создаёте большую программу, исходный код который содержит миллионы строк кода. Конечно, большая часть кода написана не вами, так как вы используете библиотеки, написанные другими программистами. Библиотекой можно назвать совокупность файлов исходного кода, нацеленных на решение какой-либо задачи. Например, есть библиотека для работы с графикой в которой содержатся функции/структуры/классы для работы с графикой. Или библиотека для работы с аудио, которая содежит функции/структуры/классы для обработки звука.

Если вы подключаете несколько библиотек, то существует высокая вероятность, что название чего-либо из одной библиотеки совпадёт с названием чего-то из другой библиотеки. Произойдёт так называемый конфликт имён. Конфликт имён, конечно, является ошибкой. Чтобы этого избежать и используются пространства имён.

```
#include <cstdio>

namespace audio
{
    int a = 10;
    int calculate(int x)
    {
        return x + 1;
    }
}

namespace graphics
{
    int a = 20;
    int calculate(int x)
    {
        return x * 2;
    }
}

int main()
{
    printf("%i\n", audio::a);     // 10
    printf("%i\n", graphics::a);  // 20
    printf("%i\n", graphics::calculate(audio::calculate(graphics::a)));  // 42
}
```

Директива `using namespace`

Если вам очень не хочется постоянно писать названия пространства имён, то вы можете использовать директиву `using namespace`, которые добавляет все имена из некоторого пространства имён в текущее пространство имён:

```
#include <cstdio>

namespace mipt
{
    int a = 10;
    int square(int x)
    {
        return x * x;
    }
}

using namespace mipt;  // После этой строки mipt:: писать больше не нужно

int main()
{
    printf("%i\n", square(a));
}
```

Использование такой возможности, конечно, полностью уничтожает всю пользу, которую приносят пространства имён. Поэтому, `using namespace` в своих программах лучше не использовать.

`using`-объявления

Как правило пространство имён содержит большое количество различных сущностей. Использование `using namespace` сделает видимыми сразу все имена из пространства имён и сильно увеличит вероятность возникновения конфликта имён. Но есть возможно сделать видимыми не все имена, а только несколько имён, используя ключевое слово `using` (без слова `namespace`):

```
#include <cstdio>

namespace mipt
{
    int a = 10;
    int b = 20;
    int square(int x)
    {
        return x * x;
    }
}

using mipt::square, mipt::a; // После этой строки mipt:: не нужно писать для a и square

int main()
{
    printf("%i\n", square(a + mipt::b));
}
```

Ключевое слово `using` можно использовать как вне всех функций, глобально, так и локально, внутри функций.

Пространство имён `std`

Все переменные/функции/структуры/классы стандартной библиотеки языка C++ содержатся в пространстве имён `std`.

Печать на экран с помощью `std::cout`

Рассмотрим глобальную переменную `cout` (сокращение от ***c****haracter* ***out****put*), определённую в библиотеке `iostream` в пространстве имён `std`. Эта переменная и оператор `<<` используется для печати значения какого-либо объекта на экран:

```
std::cout << объект;
```

В результате этой операции объект напечатается на экран (если он может напечататься). Результатом этой операции является всё тот же `std::cout`, поэтому можно применять `<<` несколько раз:

```
std::cout << объект1 << объект2 << объект3;
```

Использование `cout` и оператора `<<` является самым распространённым способом вывода текста на экран в C++.

```
#include <iostream>
int main()
{
    std::cout << "Hello, " << 123 << "\n";
}
```

В библиотеке `iostream` также есть такой объект, как `std::endl`. Если передать его объекту `cout` через оператор `<<` то напечатается перенос строки:

```
#include <iostream>
int main()
{
    std::cout << "Hello, " << 123 << std::endl;
}
```

Нужно отметить, что использование `std::endl` работает медленней, чем `\n`, при печати в файл, так как вывод `std::endl` помимо печати переноса строки также сбрасывает всё содержимое буфера в файл.

Преемущество вывода `std::cout` перед `printf`

Большим преемуществом использования `std::cout` перед функцией `printf` является то, что для `std::cout` не нужно указывать спецификатор типа, компилятор сам распознает тип передаваемого выражения.

```
#include <iostream>
int main()
{
    int a = 10;
    float b = 1.5;
    char c = 'A';

    std::cout << a << " " << b << " " << c << std::endl; // Напечатает 10 1.5 A
}
```

При этом, переменные типа `char` будут всегда печататься как символы. Если нужно напечатать переменную типа `char` как число, то нужно просто привести её тип к другому целочисленному типу:

```
#include <iostream>
int main()
{
    char c = 65;
    std::cout << c << " " << (int)c << std::endl;  // Напечатает A 65
}
```

Считывание с экрана с помощью `std::cin`

Глобальная переменная `cin` (сокращение от ***c****haracter* ***in****put*), определённая в библиотеке `iostream` в пространстве имён `std` используется для считывания с экрана. Пример использования `std::cin`:

```
#include <iostream>
int main()
{
    int a, b;
    std::cin >> a >> b;                // Считали a и b
    std::cout << a + b << std::endl;   // Напечатали их сумму
}
```

Корректное использование функций из языка C в языке C++

Вся стандартаная библиотека языка C++ лежит в пространстве имён `std`, поэтому любое имя из стандартной библиотеки начинается с `std::`. Это очень удобно, так как позволяет сразу увидеть в коде программы, какая сущность является стандартной, а какая нет. Однако, это не относится к сущностям пришедшим из языка C. Так как из-за обратной совместимости (C++ должен уметь компилировать программы на языке C) эти сущности должны работать без указания пространства имён `std`. Тем не меннее, даже для стандартных сущностей пришедших из языка C можно и нужно указывать пространство имён `std`.

Скомпилируется, но некорректно

```
#include <stdio.h>
#include <math.h>

int main()
{
    printf("%f\n", sqrt(3));
}
```

Корректно

```
#include <cstdio>
#include <cmath>

int main()
{
    std::printf("%f", std::sqrt(3));
}
```

Пространства имён и отступы

Внутри пространств имён НЕ следует делать дополнительный отступ. Использование отступа внутри пространства имён является примером плохого стиля кода. В примерах этой главы отступ делался, чтобы код был более понятным. В реальности код пространства имён должен выглядеть как-то так:

```
#include <iostream>
namespace mipt
{
int a = 10;
int square(int x)
{
    return x * x;
}
struct point
{
    int x, y;
};
}

int main()
{
    std::cout << mipt::square(5) << std::endl;
}
```

Ссылки

В C++ вводится понятие нового типа под названием ссылка. Ссылку можно представить себе как такой необычный указатель, который всегда указывает в одно место и который сам автоматически разыменовывается при использовании. Ссылки заменяют указатели в некоторых ситуациях, так как ими удобнее пользоваться.

Тип ссылки состоит из названия типа, на который указывает ссылка + значок амперсанда (`&`). Например, ссылка на `int` будет иметь тип `int&`.

Пусть есть переменная `a`. Давайте создадим указатель и ссылку на эту переменную и увеличим её на 1 с помощью указателя/ссылки:

Используем указатель

```
int a = 10;
int* p = &a;
*p += 1;
```

Используем ссылку

```
int a = 10;
int& r = a;
r += 1;
```

! Не стоит путать `&` используемый при объявлении ссылки и `&` используемый для нахождения адреса переменной. Это разные `&`, никак друг с другом не связанные.

Ссылками удобно пользоваться потому что:

1. При создании ссылки нам не нужно передавать ей адрес. Просто передаём ей некоторый объект, а ссылка уже сама находит его адрес.
2. Ссылку не нужно разыменовывать, она разыменовывается сама.

При работе ссылками важно помнить, что все (почти) операции применяемые к ссылке, на самом деле применяются к объекту на который эта ссылка указывает. Это происходит потому, что ссылка всегда (почти) сама разыменуется.

```
#include <iostream>
using std::cout, std::endl;
int main()
{
    int a = 10;
    int& r = a;

    r += 5;                     // Прибавим к a число 5
    r *= 2;                     // Умножим a на 2
    cout << r << endl;          // Напечатаем a
    cout << sizeof(r) << endl;  // Напечатаем размер a
    cout << &r << endl;         // Напечатаем адрес a
}
```

Также, как и указатели, ссылки могут указывать не только на переменные, но и на другие объекты, такие как элементы массива или структуры.

```
#include <iostream>
int main()
{
    int a[5] = {10, 20, 30, 40, 50};

    int& r = a[1];
    r += 1;

    std::cout << a[1] << std::endl;  // Напечатает 21
}
```

Отличия ссылок и указателей

Несмотря на то, что ссылки и указатели во многом похожи, у них есть и много существенных отличий.

1. **Указатель можно создать без инициализации, а ссылку обязательно нужно инициализировать**

   То есть, указатель можно создать так:

   ```
   int* p;
   ```

   В этом случае в `p` будет храниться произвольный адрес. Разыменовывать такой указатель, не задав его значение адресом какого-либо объекта, очень опасно – это приведёт к неопределённому поведению.

   Ссылку же нельзя создать без инициализации, то есть такая строка будет ошибкой:

   ```
   int& r;  // Ошибка компиляции
   ```

   При создании ссылки нужно обязательно указать то, на что она будет указывать

2. **Указатель может быть нулевым, а ссылка нет**

   Указатель можно приравнять нулевому значению, равному `NULL` или `nullptr`. Разыменование такого нулевого указателя приведёт к ошибке.

   Ссылку же нельзя инициализировать никаким нулевым значением. То есть не существует никакой константы, которая бы обозначала нулевую ссылку.

3. **Указатель можно переприсвоить, а ссылку нет**

   Если указатель `p` сначала указывал в одно место, например, на переменную `a`, то можно просто написать:

   ```
   p = &b;
   ```

   и указатель станет указывать на переменную `b`.

   Со ссылками такое не пройдёт, они всегда указывают на тот объект, который был указан при создании ссылки. При попытке изменить это и написать что-то вроде:

   ```
   r = b;
   ```

   ссылка автоматически разыменуется и присваивание произойдёт к тому, на что указывает ссылка.

4. **У указателей есть арифметика указателей, а у ссылок нет**

   К указателю можно прибавлять/отнимать целые числа. Можно вычесть 2 указателя. Можно применить `[]` к указателю.

   Ничего такого со ссылками сделать нельзя. При попытке прибавить к ссылке число, оно прибавится к той переменной, на которую указывает ссылка, так как ссылка автоматически разыменуется.

5. **Ссылки это не совсем обычный объект, некоторые операции с ними запрещены**

   Нельзя создать массив из ссылок. Нельзя получить адрес ссылки (если применим `&` к ссылке то вернётся адрес того объекта на который указывет ссылка). Нельзя создать указатель на ссылку. Нельзя создать ссылку на ссылку.

   ```
   int x = 1, y = 2, z = 3;

   int* a[3] = {&x, &y, &z};  // OK, массив из указателей
   int& b[3] = {x, y, z};     // Ошибка, создать массив из ссылок нельзя


   int* p = &x;
   int** q = &p;  // OK, указатель на указатель

   int& r = x;
   int&& s = r;  // Ошибка, нельзя создать ссылку на ссылку
   int& t = r;   // OK, ссылка r сама разыменуется и t будет указывать на x
   ```

Константные ссылки

Используя константную ссылку нельзя изменить объект на который она указывает:

```
int a = 10;
const int& r = a;
r += 1;             // Ошибка, нельзя изменить a, так как ссылка константная
```

Константные ссылки могут привязываться как к константам, так и не константам:

```
int a = 10;
const int b = 20;

int& r1 = a;        // OK
int& r2 = b;        // Ошибка, нельзя привязать обычную ссылку к константе

const int& c1 = a;  // OK
const int& c2 = b;  // OK
```

Удивительной особеностью константных ссылок является то, что они могут привязываться не только к объектам, лежащим в памяти, но и к "временным" объектам у которых нет адреса:

```
int a = 10;

int& r1 = a + 1;        // Ошибка, выражению a + 1 не соответствует никакой объект в памяти
int& r2 = 20;           // Ошибка, выражению 20 не соответствует никакой объект в памяти

const int& c1 = a + 1;  // OK
const int& c2 = 20;     // OK
```

После этого можно корректно пользоваться ссылками `c1` и `c2`, их значения будут равны `11` и `20`.

Передача по ссылке в функции

Чаще всего ссылки используются для того чтобы передать что либо в функцию. Часто нам хочется передать переменную в функцию и изменить её внутри функции. Это можно делать и с помощью указателей, но с помощью ссылок это делать гораздо удобней. Рассмотрим две программы:

Передаём по указателю:

```
#include <iostream>
void square(int* p)
{
    *p = *p * *p;
}

int main()
{
    int a = 5;
    square(&a);
    std::cout << a << std::endl;
}
```

Передаём по ссылке:

```
#include <iostream>
void square(int& a)
{
    a = a * a;
}

int main()
{
    int a = 5;
    square(a);
    std::cout << a << std::endl;
}
```

Обратите внимание на 2 вещи:

1. Ссылку не нужно разыменовывать внутри функции, это происходит автоматически.
2. При передаче в функцию, не нужно передавать адрес переменной. Нужно передать саму переменную, её адрес вычислится автоматически. При этом копирования объекта в функцию не происходит, ссылки работают также быстро как и указатели.

Передача по константной ссылке в функции

Даже если мы не хотим менять объект внутри функции, мы всё-равно можем захотеть передать его по ссылке, так как передача по ссылке не копирует объект, следовательно гораздо более эффективна для больших объектов, чем передача по значению. В этом случае очень важно передавать объект именно по константной ссылке, так как это будет явно показывать, что объект не изменится внутри функции.

```
#include <iostream>
struct Book
{
    char title[100];
    int pages;
    float price;
};

void printTitleValue(Book b)
{
    std::cout << b.title << std::endl;
}

void printTitleRef(Book& b)
{
    std::cout << b.title << std::endl;
}

void printTitleCRef(const Book& b)
{
    std::cout << b.title << std::endl;
}

int main()
{
    Book b = {"Harry Potter", 100, 200};

    printTitleValue(b);  // Медленно, так как структура будет копироваться

    printTitleRef(b);    // Быстро, но мы не можем быть уверены, что структура
                         // не изменится в функции

    printTitleCRef(b);   // Самый лучший способ в этом случае
}
```

Три способа передачи объекта в функцию в языке C++

1. Передача по значению: `void func(Object a)`

   Используйте этот способ если объект маленький и вы не хотите изменять его внутри функции.

2. Передача по ссылке: `void func(Object& a)`

   Используйте этот способ если вы хотите изменить объект внутри функции.

3. Передача по константной ссылке: `void func(const Object& a)`

   Используйте этот способ если объект большой и вы не хотите изменять его внутри функции.

Возврат ссылок из функций

Ссылки можно и возвращать из функции. Например, в данном примере, функция `get` возвращает ссылку на глобальную переменную `x`:

```
#include <iostream>
int x = 10;

int& get()
{
    return x;
}

int main()
{
    get() = 20;
    std::cout << x << std::endl;      // Напечатает 20
}
```

Необычное свойство функций, которые возвращают ссылки, заключается в том, что выражение вызова функции может стоять слева от знака присваивания.

Как это работает:

- Функция `get` возвращает ссылку на глобальную переменную `x`. То есть за место `get()` подставится ссылка на `x`.
- Так как ссылки автоматически разыменуются и так как ссылка, возвращённая из функции, указывает на `x`, то на место `get()` по сути подставится сам `x`.

Возврат ссылок на локальные переменные. Висячие ссылки.

При возврате ссылки из функции нужно следить за тем, чтобы функция не вернула ссылку на локальную переменную, как это происходит в данном примере. После завершения функции `get`, переменная `x` удалится, так как она была определена внутри функции. В результате, внутри функции `main` мы попробуем доступиться к области памяти, в которой раньше лежала переменная `x`. Это приведёт к неопределённому поведению.

```
#include <iostream>

int& get()
{
    int x = 10;
    int& r = x;
    return r;
}

int main()
{
    int& r = get();
    r += 1;  // Пытаемся изменить переменную x, которая уже удалена, это UB
}
```

Ссылки, которые указывают на уже удалённый объект в памяти называются висячими ссылками (англ. *dangling references*).

Функции, которые принимают ссылку и возвращают её же

Интересный и часто встречающийся случай – это когда функция принимает ссылку, а потом возвращает её же. Например, в данном примере, функция `inc` принимает ссылку, увеличивет то, на что указывает эта ссылка на `1`, а затем возвращает эту ссылку. При этом никакого копирование самой переменной `a` в функцию и из функции не происходит.

Также это безопасно, ведь теперь возвращаемая ссылка будет указывать не на локальную переменную функции `inc`, а туда же, куда указывала ссылка `r`, пришедшая на вход функции. И при завершении функции `inc`, объект, на который указывает возвращаемая ссылка (переменная `a`), не удалится.

```
#include <iostream>
int& inc(int& r)
{
    r += 1;
    return r;
}
int main()
{
    int a = 10;
    inc(a);
    std::cout << a << std::endl;  // Напечатает 11

    inc(a) += 5;
    std::cout << a << std::endl;  // Напечатает 17

    inc(inc(inc(a)));
    std::cout << a << std::endl;  // Напечатает 20
}
```

Перегрузка функций

Пусть у нас есть задача – нужно написать функцию, которая бы вычисляла модуль числа. Такую функцию несложно написать, но только с числами какого типа будет работать эта функция? Ведь если мы напишем такую функцию для чисел типа `int`, она не будет работать с числами типа `double`. По хорошему, придётся писать такую функцию для каждого числового типа, чтобы мы могли бы находить модуль любого числа.

В языке C придётся писать функции для каждого типа и давать каждой функции уникальное имя:

```
#include <stdio.h>
int abs(int a)
{
    if (a < 0)
        return -a;
    return a;
}
double fabs(double a)
{
    if (a < 0)
        return -a;
    return a;
}
int main()
{
    double x = abs(-1.5);   // Ошиблись - используем не ту функцию!
    printf("%lf\n", x);     // Напечатает 1.0
}
```

Такой подход имеет следующие проблемы:

1. Придётся придумывать и запоминать разные странные названия функций для разных входных типов. Например, в стандартной библиотеке языка C для нахождения модуля чисел типа `float` используется функция `fabsf`, а для чисел типа `long long` – функция `llabs`.

2. Легко случайно ошибиться и вызвать функцию для другого типа. Потом такую ошибку может быть непросто обнаружить.

3. Приходится писать много похожих функций. Грубо говоря, каждую функцию придётся повторить несколько раз - по одному разу для каждого числового типа. Получается очень много кода.

В языке C++ добавлена возможность, которая поможет нам решить первые две проблемы. Эта новая возможность языка называется *перегрузка функций* и она позволяет нам писать разные функции с одинаковыми именами, если у этих функций различаются типы параметров и/или их количество. Функции, которые используют механизм перегрузки функций, называются перегруженными функциями или просто перегрузками.

Теперь, если мы будем использовать C++ и перегрузку функций, пример выше перепишется так:

```
#include <iostream>
int abs(int a)
{
    if (a < 0)
        return -a;
    return a;
}

double abs(double a)
{
    if (a < 0)
        return -a;
    return a;
}

int main()
{
    double x = abs(-1.5);         // Всё правильно, abs можно передавать и int и double
                                  // компилятор сам выберет нужную перегрузку
    std::cout << x << std::endl;  // Напечатает 1.5
}
```

Как можно перегружать функции

- Функции можно перегружать по типу аргументов:

```
#include <iostream>

void func(int a)    {std::cout << "Int"    << std::endl;}
void func(double a) {std::cout << "Double" << std::endl;}

int main()
{
    func(1);    // Напечатает Int
    func(1.5);  // Напечатает Double
}
```

- Функции можно перегружать по количеству аргументов:

```
#include <iostream>

void func()             {std::cout << "Zero" << std::endl;}
void func(int a)        {std::cout << "One"  << std::endl;}
void func(int a, int b) {std::cout << "Two"  << std::endl;}

int main()
{
    func();        // Напечатает Zero
    func(10);      // Напечатает One
    func(10, 20);  // Напечатает Two
}
```

- Можно перегрузить по обычной ссылке и константной ссылке:

```
#include <iostream>

void func(int& a)        {std::cout << "Ref"  << std::endl;}
void func(const int& a)  {std::cout << "CRef" << std::endl;}

int main()
{
    int a = 10;
    const int b = 20;

    func(a);       // Напечатает Ref
    func(b);       // Напечатает CRef

    func(30);      // Напечатает CRef
    func(a + 1);   // Напечатает CRef
}
```

Как нельзя перегружать функции

- Функции нельзя перегружать по типу возвращаемого значения:

```
#include <iostream>

int func(int a)     {std::cout << "Int"    << std::endl;}
double func(int a)  {std::cout << "Double" << std::endl;}

int main()
{
    func(1);  // Невозможно выбрать
}
```

Неоднозначности при выборе перегрузки

Так как в языке C++ также есть такая вещь, как неявное приведение типов, то может так оказаться, что в качестве перегрузки подойдут несколько функций. Какая из функций будет при этом выбираться? Рассмотрим это на нескольких примерах:

- Известно, что числа типа `float` можно присваивать числам типа `double` и наоборот. При этом происходит неявное приведение типов. Соответственно, числа типа `float` также можно передавать в функции, которые принимают `double`. Но что если есть две перегрузки, одна принимает `float`, а другая – `double` и мы передаём число типа `float`. Оба варианта подходят. Какая из перегрузок выберется?

```
#include <iostream>

void func(float a)  {std::cout << "Float"  << std::endl;}
void func(double a) {std::cout << "Double" << std::endl;}

int main()
{
    float x = 1.5f;
    func(x);  // Напечатает Float
}
```

  В этом случае выберется перегрузка с `float`, так как она лучше подходит.

- Что если у нас есть две перегрузки: одна с `int`, а другая с `double`, а мы передаём в функию число `float`. Опять обе функции подходят. Какая из перегрузок выберется на этот раз?

```
#include <iostream>

void func(int a)    {std::cout << "Int"    << std::endl;}
void func(double a) {std::cout << "Double" << std::endl;}

int main()
{
    float x = 1.5f;
    func(x);  // Напечатает Double
}
```

  В этом случае выберется перегрузка с `double`. Она лучше подходит, так как при преобразовании `float` в `double` не происходит потери точности числа, а при преобразовании `float` в `int` такая потеря точности может произойти. Поэтому тут выбирается перегрузка с `double`.

- Что если у нас есть две перегрузки: одна с `int`, а другая с `float`, а мы передаём в функцию число `double`. Опять обе функции подходят. Но в обоих вариантах будет происходить потеря точности. Какая из перегрузок выберется на этот раз?

```
#include <iostream>

void func(int a)   {std::cout << "Int"   << std::endl;}
void func(float a) {std::cout << "Float" << std::endl;}

int main()
{
    double x = 1.5;
    func(x);  // Ошибка компиляции
}
```

  В этот раз компилятор не сможет выбрать правильную перегрузку, так как в этом случае обе функции подходят одинаково хорошо. Интересно, что если в этом примере одну из функции просто удалить, то программа скомпилируется без ошибки.

Перегрузка на более сложных типах

Перегрузка часто применяется к функциям принимающим объекты пользовательских типов, таких как структуры или классы.

```
#include <iostream>
struct Book
{
    char title[100];
    int numPages;
};

struct Movie
{
    char name[100];
    float rating;
};

void printTitle(const Book& b)
{
    std::cout << b.title << std::endl;
}

void printTitle(const Movie& m)
{
    std::cout << m.name << std::endl;
}

int main()
{
    Book b = {"Harry Potter", 500};
    Movie m = {"The Matrix", 9.0};

    printTitle(b);  // Напечатает Harry Potter
    printTitle(m);  // Напечатает The Matrix
}
```

Строка языка C++. Класс `std::string`.

Как известно, строки в языке C являются просто массивами символов (элементов типа `char`). Использовать такие строки не очень удобно, так как нужно постоянно следить, чтобы строка вмещалась в массив в котором она хранится. При этом легко ошибиться и выйти за границы этого массива, что приведёт к неопределённому поведению. Также строки в языке C неудобно присваивать, конкатенировать и сравнивать – для этих операций требуется использовать функции из библиотеки `string.h`.

В языке C++ появляется гораздо более удобный способ работы со строками – использование класса `std::string` из библиотеки `string` (не путайте эту библиотеку с библиотекой `string.h`, также известной как `cstring`). Работая со строкам `std::string` можно не задумываться о выделении/освобождении памяти – всё будет сделано автоматически. Также такие строки удобно присваивать, складывать и сравнивать.

При этом использовать старые строки из языка C тоже можно, в языке C++ такие строки называются *C-строками*.

```
#include <iostream>
#include <cstring>  // Библиотека для работы с C-строками. Содержит strlen, strcpy и т. д.
#include <string>   // Библиотека, содержащая класс std::string

int main()
{
    // Работаем с C-строками в языке C++
    char a[50] = "Cat";
    std::strcpy(a, "Mouse");
    std::strcat(a, "Dog");

    if (std::strcmp(a, "MouseDog") == 0)
        std::cout << a << std::endl;


    // Проделаем аналогичные операции, используя класс std::string
    std::string b = "Cat";
    b = "Mouse";
    b += "Dog";

    if (b == "MouseDog")
        std::cout << b << std::endl;
}
```

Манипуляция с отдельными символами в строках `std::string`

Работа с отдельными символами в строках `std::string` происходит также, как и в C-строках. Элемент строки `std::string` тоже является символом типа `char`. Выход за границы также является неопределённым поведением.

```
#include <iostream>
#include <string>
int main()
{
    std::string a = "Mouse";
    std::cout << a[0] << std::endl;  // Напечатает M

    a[4] = 'X';
    std::cout << a << std::endl;     // Напечатает MousX

    a[10] = 'X'; // UB
}
```

Для работы с символами можно использовать библиотеку `ctype.h` из языка C, в C++ она называется `cctype`:

```
#include <iostream>
#include <string>
#include <cctype>
int main()
{
    std::string s = "Mouse123!";
    for (std::size_t i = 0; i < s.size(); ++i)
    {
        if (std::isalpha(s[i]))
            s[i] += 1;
    }
    std::cout << s << std::endl;  // Напечатает Npvtf123!
}
```

Методы класса `std::string`

Класс – это пользовательский тип данных, который позволяет объединять данные и функции, работающие с этими данными, в одну сущность.

Методы класса – это функции, определенные внутри класса, которые описывают поведение объектов этого класса. Вызов методов класса осуществляется на объектах класса с помощью оператора точка.

Рассмотрим методы класса `std::string`:

- Метод `push_back` – добавляет один символ в конец строки.
- Метод `pop_back` – удаляет один символ из конца строки.

```
#include <iostream>
#include <string>
int main()
{
    std::string s = "Elephant";
    s.push_back('X');
    std::cout << s << std::endl;  // Напечатает ElephantX

    s.pop_back();
    s.pop_back();
    std::cout << s << std::endl;  // Напечатает Elephan
}
```

- Метод `insert` – добавляет символы в строку.
- Метод `clear` – очищает строку.

```
#include <iostream>
#include <string>
int main()
{
    std::string s = "Elephant";
    s.insert(2, "Cat");           // Начиная с символа с индексом 2, добавь Cat
    std::cout << s << std::endl;  // Напечатает ElCatephant

    s.clear();
    std::cout << s << std::endl;         // Напечатает пустую строку
    std::cout << s.size() << std::endl;  // Напечатает 0
}
```

- Метод `erase` – удаляет символы из строки.

```
#include <iostream>
#include <string>
int main()
{
    std::string s = "Elephant";
    s.erase(2, 4);                // Начиная с символа с индексом 2, удали 4 символа
    std::cout << s << std::endl;  // Напечатает Elnt
}
```

- Метод `substr` – получает подстроку из строки. При этом создаёт новую строку и копирует туда часть изначальной строки.

```
#include <iostream>
#include <string>
int main()
{
    std::string s = "Elephant";
    std::string a = s.substr(2, 4);  // Начиная с символа с индексом 2,
                                     // возьми подстроку длины 4
    std::cout << a << std::endl;     // Напечатает epha
}
```

- Метод `find` – ищет подстроку в строке. Возвращает индекс первого символа первого вхождения подстроки (тип `std::size_t`). В случае, если найти подстроку не удалось, возвращает числовую константу `std::string::npos`.

```
#include <iostream>
#include <string>
using std::cout, std::endl, std::size_t;
int main()
{
    std::string s = "Elephant";
    size_t k = s.find("phan");

    if (k != std::string::npos)
        cout << "Found, index = " << k << endl;
    else
        cout << "Not found" << endl;
}
```

Данная программа напечатает `Found, index = 3`.

- Метод `starts_with` – проверяет, начинается ли строка данного префикса. Метод `ends_with` – проверяет, кончается ли строка на данный суффикс. Методы возращают значение типа `bool`.

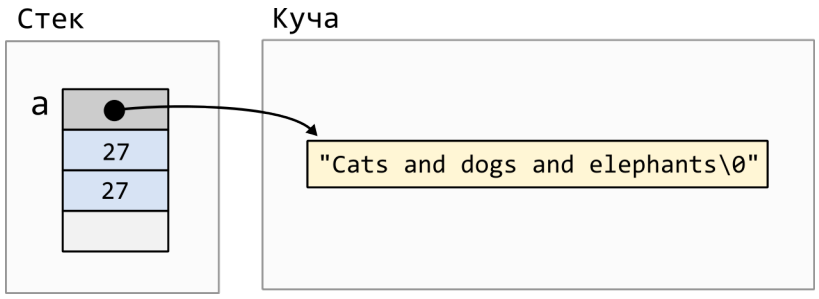
```
#include <iostream>
#include <string>
int main()
{
    std::string s = "Elephant";

    if (s.starts_with("Ele"))
        std::cout << "Yes" << std::endl;
}
```

Внутреннее устройство объектов типа `std::string`

Строки типа `std::string` представляют собой не что иное, как динамический массив из элементов типа `char`. Реализация строки может отличаться в стандартных библиотеках разных компиляторов, но в большинстве реализации строка имеет примерно следующее строение. В самом объекте строки хранятся указатель на данные, размер строки и её вместимость. Указатель хранит адрес на массив элементов типа `char`, который хранится в куче.
Рассмотрим строку:

```
std::string a = "Cats and dogs and elepnants";
```

Её можно представить в памяти следующим образом:



Обратите внимание, на следующие моменты:

- Нулевой символ `\0` также хранится на конце строки, несмотря на то что размер строки мы храним и могли бы обойтись без нулевого символа. Это делается для того, чтобы можно было легко передавать содержимое `std::string` в функцию, которая принимает C-строку. В такую функцию достаточно будет передать указатель на данные в куче.
- Размер и вместимость хранятся без учёта нулевого символа. То есть, если вместимость равна 27, то реально в куче выделяется 28 байт.
- Размер самого объекта строки равен 32 байта, последние 8 байт могут быть не заняты. Эти 8 байт нужны, для так называемой *оптимизации малой строки*. Об этой оптимизации будет рассказано в следующем семинаре.

Дополнительные методы класса `std::string`

- Метод `c_str()` – возвращает указатель на данные в куче.
- Метод `size()` – возвращает размер строки.
- Метод `capacity()` – возвращает вместимость строки.
- Метод `reserve(size_t n)` – увеличивает вместимость строки.
- Метод `resize(size_t n)` – изменяет размер строки.
- Метод `shrink_to_fit()` – делает вместимость строки, равной её размеру.

```
#include <iostream>
#include <string>
int main()
{
    std::string a = "Cats and dogs and elepnants";
    std::cout << a.size() << " " << a.capacity() << std::endl;  // Напечатает 27 27

    a += "!";
    std::cout << a.size() << " " << a.capacity() << std::endl;  // Напечатает 28 54

    a.shrink_to_fit();
    std::cout << a.size() << " " << a.capacity() << std::endl;  // Напечатает 28 28
}
```

Передача строк в функции

При передаче строки в функцию по значению происходит глубокое копирование строки, что может быть медленно, если строка большая. Поэтому лучше строки передавать по ссылке. Следуют передавать строку по обычной ссылке, если вы хотите изменить строку внутри функции и передавать строку по константной ссылке, если не хотите изменять строку.

```
#include <iostream>
#include <string>
#include <cctype>
using std::cout, std::endl, std::size_t;

// Переводим все буквы строки в верхний регистр. Передаём по ссылке.
void convertToUpper(std::string& s)
{
    for (size_t i = 0; i < s.size(); ++i)
    {
        s[i] = std::toupper(s[i]);
    }
}
// Считаем количество букв в строке. Обязательно нужно передавать по константной ссылке.
size_t countLetters(const std::string& s)
{
    size_t result = 0;
    for (size_t i = 0; i < s.size(); ++i)
    {
        if (std::isalpha(s[i]))
            result += 1;
    }
    return result;
}

// При передаче строки в эту функцию, строка будет копироваться, это плохо если строка длинная
void badFunction(std::string s)
{
    std::cout << s << std::endl;
}

int main()
{
    std::string a = "Mouse123Cat!";
    cout << a << endl;                // Напечатает Mouse123Cat!

    convertToUpper(a);
    cout << a << endl;                // Напечатает MOUSE123CAT!
    cout << countLetters(a) << endl;  // Напечатает 8

    badFunction(a);  // Произойдёт копирование строки a в функцию badFunction
}
```

Передача строк `std::string` в функции языка C

Метод `c_str` класса `std::string` возвращает указатель на данные в куче. Этот метод можно использовать если нужно передать содержимое строки `std::string` в функцию, которая принимает C-строку.

```
#include <iostream>
#include <cstring>
#include <string>

int main()
{
    std::string a = "Cat";

    char b[10];
    std::strcpy(b, a.c_str());

    std::cout << b << std::endl;  // Напечатает Cat
}
```

Конвертация чисел в строки и наоборот

Для конвертации чисел в строки можно использовать функцию `std::to_string`. Эта функция может принимать числа любых типов, так как тут используется механизм перегрузки функций.

```
#include <iostream>
#include <string>

int main()
{
    int a = 123;
    std::string sa = std::to_string(a);
    std::cout << sa << std::endl;  // Напечатает 123

    float x = 123.456;
    std::string sx = std::to_string(x);
    std::cout << sx << std::endl;  // Напечатает 123.456
}
```

Для конвертации из строки в число следует использовать функции `std::stoi` (для чисел типа `int`), `std::stof` (для чисел типа `float`), `std::stod` (для чисел типа `double`) и так далее. Тут нельзя было применить перегрузку функций, так как функции различаются только по возвращаемому значению.

```
#include <iostream>
#include <string>

int main()
{
    std::string sa = "123";
    int a = std::stoi(sa);
    std::cout << a << std::endl;  // Напечатает 123

    std::string sx = "123.456";
    float x = std::stof(sx);
    std::cout << x << std::endl;  // Напечатает 123.456
}
```

Передача C-строк в функции, которые принимают `std::string`

Что, если мы передаём C-строку в функцию, которая принимает строку типа `std::string`? Кажется, это работать не должно, так как типы не совпадают. Но, на самом деле это работает автоматически, если функция принимает строку по значению или по константной ссылке. Рассмотрим это на примере:

```
#include <iostream>
#include <string>
using std::cout, std::endl;

void func1(std::string str)        {cout << str << endl;}
void func2(std::string& str)       {cout << str << endl;}
void func3(const std::string& str) {cout << str << endl;}

int main()
{
    char a[10] = "Cat";
    func1(a);  // OK
    func2(a);  // Ошибка
    func3(a);  // OK
}
```

Литералы типа `std::string`

Строковые литералы в языках C и C++ не отличаются. Это простые массивы элементов типа `char`, хранящиеся в сегменте данные. Строковые литералы имеют тип `const char[n]`, где `n` – это размер строки включая `\0`.

```
#include <iostream>
#include <string>

void func(const char* str)        {std::cout << "C-string" << std::endl;}
void func(const std::string& str) {std::cout << "std::string" << std::endl;}

int main()
{
    func("Cat");  // Напечатает C-string
}
```

Тут компилятор выберет первую перегрузку, так как тип выражения `"Cat"` – это `const char[4]`.

Хотелось бы передавать куда-либо строковой литерал так, чтобы он имел тип `std::string`. Можно конечно создать временный объект `std::string` и сразу передать в функцию, но это может быть слишком многословно. Поэтому в язык была добавлена возможность создавать литерал типа `std::string`. Чтобы их подключить нужно добавить пространство имён `std::string_literals`. После этого можно будет создавать литералы типа `std::string`, просто добавив в конце обычного литералы букву `s`.

```
#include <iostream>
#include <string>
using namespace std::string_literals;

void func(const char* str)        {std::cout << "C-string" << std::endl;}
void func(const std::string& str) {std::cout << "std::string" << std::endl;}

int main()
{
    func(std::string("Cat"));  // Напечатает std::string (передаём временный объект)
    func("Cat"s);              // Напечатает std::string (передаём литерал типа std::string)
}
```

Чтение в строку типа `std::string`. Функция `std::getline`.

Для считывания строк можно использовать:

- Объект `std::cin`:

  ```
  std::cin >> str;
  ```

  В этом случае происходит следующее:

  1. Пропускаются все пробельные символы до первого непробельного символа.
  2. Затем происходит считывание символов в строку `str` до первого пробельного символа.

- Функция `std::getline`:

  ```
  std::getline(std::cin, str);
  ```

  В этом случае происходит следующее:

  1. Пробельные символы в начале НЕ пропускаются.
  2. Происходит считывание символов в строку `str` до символа переноса строки (`\n`).

В обоих случаях можно использовать возвращаемое значение как булевое значение конца ввода. Например, если нужно считывать из стандартного входа по словам, то можно написать:

```
#include <iostream>
#include <string>
int main()
{
    std::string a;
    while (std::cin >> a)
        std::cout << a << std::endl;
}
```

Если же нужно считывать из стандартного входа по строкам, то можно написать:

```
#include <iostream>
#include <string>
int main()
{
    std::string a;
    while (std::getline(std::cin, a))
        std::cout << a << std::endl;
}
```

Динамический массив C++. Класс `std::vector`

Класс `std::vector`, пожалуй, самый главный, самый часто используемый класс языка C++. Это класс не имеет никакого отношения к векторам из математики, а представляет собой динамический массив с настраиваемым типом хранимых элементов.

```
#include <iostream>
#include <vector>
int main()
{
    std::vector<int> v {10, 20, 30, 40, 50};

    // Напечатать вектор можно только поэлементно
    for (std::size_t i = 0; i < v.size(); ++i)
        std::cout << v[i] << " ";
    std::cout << std::endl;
}
```

В треугольных скобочках передаётся тип элемента вектора. Тут используется механизм шаблонов, который будет пройден в следующих семинарах.

Доступ к отдельным элементам в векторе

Доступ к отдельному элементу у вектора не отличается от доступ к элементу у обычного массива. Он также осуществляется через квадратные скобки. Выход за границы вектора также является неопределённым поведением.

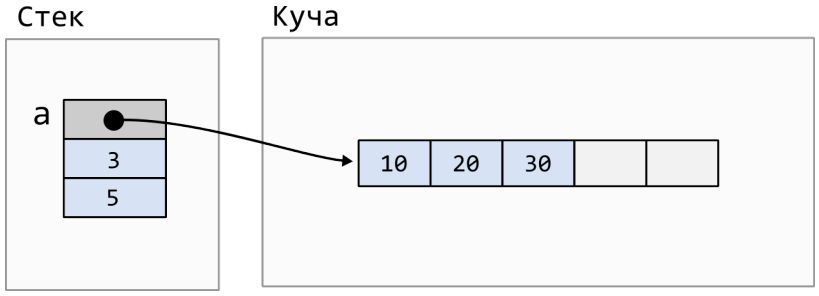
```
#include <iostream>
#include <vector>
int main()
{
    std::vector<int> v {10, 20, 30, 40, 50};
    std::cout << v[1]  << std::endl;  // Напечатает 20
    std::cout << v[10] << std::endl;  // UB
}
```

Внутренее устройство вектора

Объекты типа `std::vector` представляют собой, динамический массив из элементов типа, задаваемого в треугольных скобках. Реализация вектора может отличаться в стандартных библиотеках разных компиляторов, но в большинстве реализации вектор имеет примерно следующее строение. В самом объекте вектора хранятся указатель на данные, размер вектора и его вместимость. Указатель хранит адрес на массив элементов, который хранится в куче. Рассмотрим вектор:

```
std::vector<int> a {10, 20, 30};
a.reserve(5);
```

Его можно представить в памяти следующим образом:



Методы класса `std::vector`

- Метод `push_back` – добавляет элемент в конец вектора.
  Метод `pop_back` – удаляет последний элемент вектора.

```
#include <iostream>
#include <vector>

void printVector(const std::vector<int>& v)
{
    for (std::size_t i = 0; i < v.size(); ++i)
        std::cout << v[i] << " ";
    std::cout << std::endl;
}

int main()
{
    std::vector<int> v {10, 20, 30, 40};

    v.push_back(50);
    printVector(v);  // Напечатает 10 20 30 40 50

    v.pop_back();
    printVector(v);  // Напечатает 10 20 30 40
}
```

- Метод `front` – возвращает первый элемент вектора, то же самое, что и `v[0]`. Метод `back` – возвращает последний элемент вектора, то же самое, что и `v[v.size() - 1]`.

```
#include <iostream>
#include <vector>

int main()
{
    std::vector<int> v {10, 20, 30, 40};
    std::cout << v.front() << " " << v.back() << std::endl;  // Напечатает 10 40
}
```

- Метод `clear` – очищает вектор, удаляет все элементы.
- Метод `empty` – возращает `true`, если вектор пуст и `false` иначе.
- Метод `size` – возвращает размер вектора.
- Метод `capacity` – возвращает вместимость вектора.

```
#include <iostream>
#include <vector>

int main()
{
    std::vector<int> v {10, 20, 30, 40, 50};
    std::cout << v.size() << " " << v.capacity() << std::endl;  // Напечатает 5 5
    v.push_back(60);
    std::cout << v.size() << " " << v.capacity() << std::endl;  // Напечатает 6 10
}
```

- Метод `data` – возвращает указатель на данные в куче. Можно использовать этот метод, если нужно передать вектор в функцию, которая принимает указатель на элемент массива.

```
#include <iostream>
#include <vector>
#include <cstring>

int main()
{
    std::vector<int> v {10, 20, 30, 40, 50};

    int a[5];
    std::memcpy(a, v.data(), 5 * sizeof(int));

    for (std::size_t i = 0; i < 5; ++i)
        std::cout << a[i] << " ";
    std::cout << std::endl;
}
```

Операторы, которые можно применять к вектору

Вектора можно присваивать. Также вектора можно сравнивать. Сравнение происходит в лексикографическом порядке.

```
#include <iostream>
#include <vector>

int main()
{
    std::vector<int> v {10, 20, 30, 40, 50};
    std::vector<int> u;

    u = v;
    if (u >= v)
        std::cout << "Yes" << std::endl;
}
```

Передача вектора в функции

Также как и для строки, при передаче вектора в функцию, которая принимает по значению, происходит глубокое копирование вектора, что может быть очень медленно. Поэтому вектор желательно передавать по ссылке. Вектор следует передавать по обычной ссылке, если вектор нужно изменить внутри функции, и по константной ссылке, если в функции вектор меняться не будет.

```
#include <iostream>
#include <vector>
#include <string>
using std::cout, std::endl, std::size_t;

void printVector(const std::vector<std::string>& v)
{
    for (size_t i = 0; i < v.size(); ++i)
        cout << v[i] << " ";
    cout << endl;
}
```

```cpp
std::vector<std::string> getAnimals()
{
    std::vector<std::string> result {"Cat", "Dog", "Spider", "Lion", "Snake", "Shark"};
    return result;
}

void eraseLastLetters(std::vector<std::string>& v)
{
    for (size_t i = 0; i < v.size(); ++i)
    {
        if (!v[i].empty())
            v[i].pop_back();
    }
}

std::vector<std::string> getAnimalsByLetter(const std::vector<std::string>& v, char c)
{
    std::vector<std::string> result;
    for (size_t i = 0; i < v.size(); ++i)
    {
        if (v[i].starts_with(c))
            result.push_back(v[i]);
    }
    return result;
}

int main()
{
    std::vector<std::string> a = getAnimals();
    printVector(a);  // Cat Dog Spider Lion Snake Shark

    eraseLastLetters(a);
    printVector(a);  // Ca Do Spide Lio Snak Shar

    std::vector<std::string> b = getAnimalsByLetter(a, 'S');
    printVector(b);  // Spide Snak Shar
}
```

Приведение типов. Оператор `static_cast`

Опасность приведения в стиле C

Язык C++ унаследовал от языка C приведение типов через круглые скобочки. Такое приведение через круглые скобочки в C++ называется как *приведение в стиле C* (англ. *C-style cast*). Однако приведение в стиле C может делать слишком много и оно может быть небезопасно в некоторых ситуациях. Рассмотрим эти ситуации:

1. **Приведение типов в стиле C может убирать `const` у указателей и ссылок**
   Когда мы передаём объект в функцию, которая принимает по константному указателю, то мы предполагаем, что внутри функции этот объект не изменится. Почти всегда это так, но, в теории, константный указатель внутри функции может быть приведён к обычному указателю с помощью приведения в стиле C.

   ```
   #include <iostream>
   using std::cout, std::endl;

   void printByAddress(const int* p)
   {
       cout << *p << endl;

       int* q = (int*)p;
       *q += 10;
   }

   int main()
   {
       int a = 10;
       printByAddress(&a); // Передаём по const int* и думаем, что a внутри не поменяется
       cout << a << endl;  // Напечатает 20
   }
   ```

2. **Приведение типов в стиле C может конвертировать типы указателей без ограничений**
   Можно, например, конвертировать указатель `int*` в указатель `float*`. Если после этого конвертированный указатель будет разыменован, то это приведёт к неопределённому поведению из-за нарушения строгих правил алиасинга.

   ```
   #include <iostream>

   int main()
   {
       int a = 10;
       int* p = &a;

       float* q = (float*)p;
       *q = 1.0;  // UB
   }
   ```

   Но в редких случаях такое приведение указателей может быть полезно. Во-первых, есть исключения из строгих правил алиасинга, например можно указателем `char*` указывать на объект типа `int`. Во-вторых, строгие правила алиасинга можно просто отключить.

Из-за того, что C++ должен быть обратно совместим с языком C, в языке C++ осталось небезопасное приведение через круглые скобочки. Но использование приведение в стиле C считается плохим стилем кода. Его не следует использовать. Вместо этого следует использовать операторы приведения языка C++: `static_cast` `const_cast`, `reinterpret_cast`.

Оператор приведения `const_cast`

Оператор приведения `const_cast` нужен только для того, чтобы убирать или добавлять `const` у типов указателей и ссылок. Этот оператор приведения считается небезопасным и его использование запрещено.

```
#include <iostream>
using std::cout, std::endl;

void print(const int& a)
{
    cout << a << endl;
    int& b = const_cast<int&>(a);
    b += 10;
}

int main()
{
    int a = 10;
    print(a);           // Передаём по константной ссылке и думаем, что a внутри не поменяется
    cout << a << endl;  // Напечатает 20
}
```

Оператор приведения `reinterepret_cast`

Оператор приведения `reinterepret_cast` нужен для того, чтобы конвертировать разные типы указателей и ссылок. Например, он может конвертировать `int*` в `float*`. Этот оператор приведения считается небезопасным, так как можно легко нарушить правила строгого алиасинга, но иногда его необходимо использовать.

```
#include <iostream>

int main()
{
    int a = 10;
    int* p = &a;

    char* qc = reinterpret_cast<char*>(p);
    *qc = 0xaa;  // OK, так как char* является исключением из строгих правил алиасинга

    float* qf = reinterpret_cast<float*>(p);
    *qf = 1.0;   // UB, нарушены строгие правил алиасинга
}
```

Оператор приведения `static_cast`

Оператор приведения `static_cast` нужен для всего остального. Он считается относительно безопасным и его следует использовать всегда, когда нужно сделать приведение типов. Остальные типы приведений следует использовать только если `static_cast` не подходит.

```
#include <iostream>

int main()
{
    int a = 10;
    float b = static_cast<float>(a);
}
```

Другие различия между C и C++

В C++ нет неявного приведения из `void*` в указатель другого типа

В обоих языках нет неявного приведения одного типа указателя к другому в общем случае:

```
int main()
{
    int a = 10;
    int* p = &a;
    float* q = p;  // Ошибка в обоих языках
}
```

Но есть неявное приведение в некоторых ситуациях:

- Из обычного указателя в константный указатель того же типа:

  ```
  int main()
  {
      int a = 10;
      int* p = &a;
      const int* q = p;  // OK в обоих языках
  }
  ```

- Если один тип является `typedef`-ом другого:

  ```
  #include <stdio.h>
  int main()
  {
      unsigned long long a = 10;
      unsigned long long* p = &a;
      size_t* q = p;  // OK в обоих языках, на 64-х битных системах
  }
  ```

- Из числового литерала, равного нулю:

  ```
  int main()
  {
      int* p = 0;  // OK в обоих языках
  }
  ```

- Из типа указателя к типу `void*`:

  ```
  int main()
  {
      int a = 10;
      int* p = &a;
      void* q = p;  // OK в обоих языках
  }
  ```

- Из указателя типа `void*` к типу `int*`:

  ```
  int main()
  {
      int a = 10;
      void* p = &a;
      int* q = p;  // OK в C, Ошибка в C++
  }
  ```

  В языке C++ запрещён такое неявное приведение для безопасности. Такое неявное приведение может привести к ошибкам.

В C++ нужно использовать константу nullptr для нулевых указателей

В языке C для нулевого указателя используется константа `NULL`. В языке C++ её тоже можно использовать, но нежелательно, вместо этого лучше использовать новую константу `nullptr`.

Причина этого в том, что `NULL` является просто макро-константой определённой в стандартной библиотеке, например, так:

```
#define NULL 0
```

В разных компиляторах это определение может быть разным, то есть `NULL` может быть разного типа. В языке C++ это может привести к различным проблемам:

```
#include <iostream>
void func(int a)  {std::cout << "Int"     << std::endl;}
void func(int* p) {std::cout << "Pointer" << std::endl;}

int main()
{
    int* p = NULL;
    func(p);        // Напечатает Pointer
    func(NULL);     // Ошибка, в зависимости от компилятора может выбрать первую перегрузку
                    // или привести к ошибке компиляции
}
```

В языке C++ сделали новый указатель `nullptr`, который имеет уникальный тип `nullptr_t`. При этом `nullptr` может неявно приводится к любому типу указателя. Это избавляет от проблемы, когда компилятор мог спутать `NULL` с нулём.

```
#include <iostream>
void func(int a)  {std::cout << "Int"     << std::endl;}
void func(int* p) {std::cout << "Pointer" << std::endl;}

int main()
{
    int* q = nullptr;
    func(q);        // Напечатает Pointer
    func(nullptr);  // Напечатает Pointer
}
```

Параметры функций со значением по умолчанию

Язык C++ поддерживает параметры функций со значением по умолчанию. Для это нужно просто задать значения по умолчанию в объявлении функции:

```
#include <iostream>
void func(int a, int b = 10, std::string prefix = "Hello: ")
{
    std::cout << prefix;
    std::cout << a + b << std::endl;
}

int main()
{
    func(10, 20, "Hi: ");  // Напечатает Hi: 30
    func(10, 20);          // Напечатает Hello: 30
    func(10);              // Напечатает Hello: 20
}
```

В целом, это неудачное нововведение C++. Использование таких функций является плохим стилем кода.

Форматированный вывод с использованием `std::cout`

Печать в шестнадцатеричной системе счисления

```
#include <iostream>
using std::cout, std::endl;

int main()
{
    int a = 200;

    cout << std::hex;   // Переключить печать в 16-ричную систему
    cout << a << endl;  // Напечатает c8
    cout << a << endl;  // Напечатает c8
    cout << std::dec;   // Переключить печать в 10-ичную систему
    cout << a << endl;  // Напечатает 200
}
```

Печать чисел с плавающей точкой с заданием количества знаков после точки

Используем `std::setprecision` из библиотеки `iomanip`.

```
#include <iostream>
#include <iomanip>
#include <cmath>
using std::cout, std::endl;

int main()
{
    double x = std::sqrt(2);
    cout << x << endl;  // Напечатает 1.41421

    cout << std::setprecision(20);
    cout << x << endl;  // Напечатает 1.4142135623730951455
}
```

Печать чисел с фиксированной шириной и с замощением другими символами

Используем `std::setw` и `std::setfill` из библиотеки `iomanip`.

```
#include <iostream>
#include <iomanip>
using std::cout, std::endl;

int main()
{
    int a = 123;

    // Следующая строка напечатает |  123|
    cout << "|" << std::setw(5) <<   a << "|" << endl;

    // Следующая строка напечатает |00123|
    cout << "|" << std::setw(5) <<  std::setfill('0') << a << "|" <<  endl;
}
```