

Основы программирования. Теормин

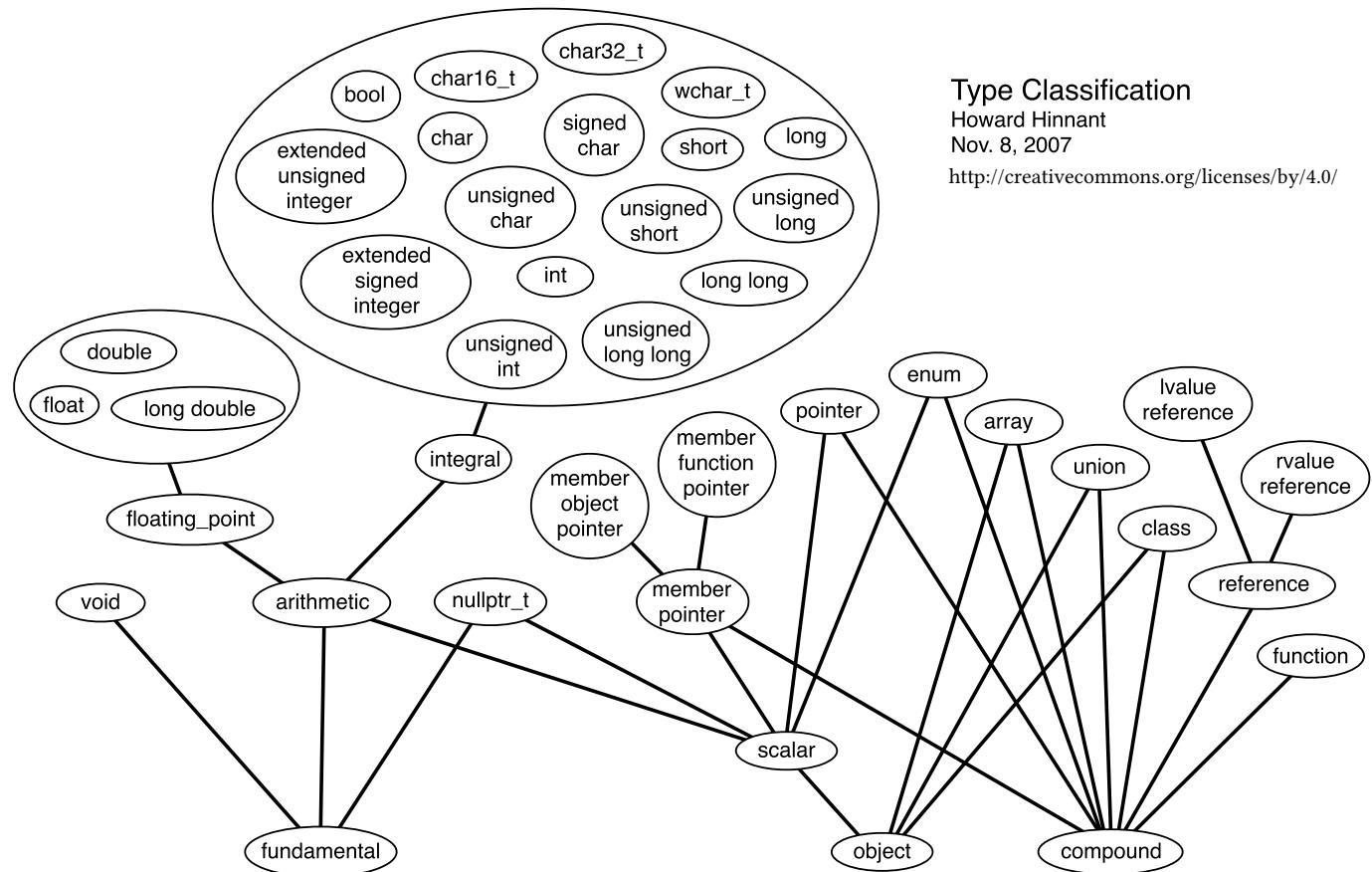
Содержание

Встроенные типы данных, представление чисел в памяти, строки	1
Целочисленные типы	1
Числа с плавающей точкой	2
Строки	2
Преобразования типов	2
Неявные преобразования	2
Явные преобразования	3
Структуры и объединения	3
Структура	3
Объединение	4
Выравнивание	5
Указатели	6
Массивы	6
Устройство памяти	7
Виртуальная и физическая память	7
Сегменты памяти	8
Страницы физической памяти	9
Стек вызова функций	9
Ссылки	9
Перегрузка функций	10
namespace	11
TODO Компиляция, этапы, ошибки компиляции	12
OOP	12
Абстракция	12
Инкапсуляция	12
TODO Полиморфизм	12
TODO Наследование	12
TODO Классы. Специальные методы	12
TODO Перегрузка операторов	13
TODO Шаблоны функций, классов. Частичная и полная специализация	13
RAII	13
Всякие мелочи	14
Linkage Duration	14
Storage Duration	14



Встроенные типы данных, представление чисел в памяти, строки

Начну сию книжку с запугивания. Вот так классифицируются типы в C++:



Но в данном разделе теормина нас только раздел «fundamental». Оттуда выделим следующие типы:

- Целочисленные: `char`, `int`, `short`, `long`
- Числа с плавающей точкой: `float`, `double`
- `bool`
- `void`
- `nullptr_t`

Целочисленные типы

Для некоторых арифметических типов данных есть модификаторы:

- Размерность: `short`, `long`
- Наличие знака: `signed`, `unsigned`

С помощью оператора `sizeof` можно получить размер типа в байтах. Стандарт C++ гарантирует для целочисленных данных следующую цепочку:

```
1 == sizeof(char) <= sizeof(short) <= sizeof(int) <= sizeof(long) <= sizeof(long long)
```

Для нас это значит, что:

- `char` всегда 1 байт
- Какие-то типы данных могут иметь одинаковый размер. Это особенно заметно на 64-битных системах: размеры `int` и `long` совпадают на Windows, но отличаются на macOS или Linux.
 - Чтобы гарантировать размерность типов данных, можно использовать `typedef`'ы из хедера `cstdint` (например, `int16_t` или `uint32_t`).

На современных компьютерах знаковые целые числа хранятся в *дополнительном коде*. Это позволяет легко складывать два целых числа, учитывая переполнение. Для преобразования в дополнительный код (т.е. взятия отрицания), все биты исходного числа инвертируются, а затем прибавляется единица.

Числа с плавающей точкой

К данным типам относятся `float` и `double` (а также `long double`). Стандарт C++ не диктует их представление в памяти, но все современные системы придерживаются стандарту IEEE 754, где вещественное число записано следующим образом:

$$\begin{array}{c} 1 & \underbrace{10000001}_{\text{Знак}} & \underbrace{010000000000000000000000_2}_{\text{Экспонента}} \\ & & \underbrace{\hspace{1cm}}_{\text{Мантисса}} \end{array}$$

Например, для числа -1.2345 :

- Знак равен 1 («минус»)
- Мантисса равна 12345
- Экспонента равна 10^{-4}

То есть, число хранится в форме, знакомой еще со школьной физики: $-12345 \cdot 10^{-4}$

Помимо обычных вещественных чисел, стандартом IEEE 754 определены:

- $+0$ и -0
- $+inf$ и $-inf$
- NaN — «not a number» (при неопределенностях типа $\frac{0}{0}$)

Размеры чисел с плавающей точкой фиксированы:

- `float`: 32 бита
- `double`: 64 бита

Строки

По сути дела, строки представлены в виде последовательности `char`'ов. Со временем языка Си принято, чтобы строки оканчивались на знак '`\0`', что означает конец строки. Это сделано потому, что для «сырых» массивов нельзя узнать их длину, только если ее не хранить как-то самостоятельно. На нулевой символ опираются многие функции из стандартной библиотеки: `strlen`, `strcmp` и тому подобные.

В современном C++ лучше использовать `std::string`, но это уже другая история.

TODO литералы? enum?

Преобразования типов

Так называемые касты.

Неявные преобразования

При выполнении данных операций, C++ выполнит неявные преобразования в следующем порядке:

1. Если какой-то из операндов является `long double`, то другой operand приводится к `long double`
2. Аналогичным образом приведение к `double`
3. Аналогичным образом приведение к `float`
4. Операнды типа `char` или `short` приводятся к `int`
5. Наконец, если какой-то operand — `long`, то второй приводится к `long`

Если обобщить, то:

1. Сначала кастуем к наибольшим вещественным типам
2. Затем кастуем к наибольшим целочисленным типам

Пример:

```
int a = 42;
float b = 3.14;
long c = a + b;
```

В третьей строке произойдет следующее:

1. a приведется к float:

```
a = 42.0f
```

2. Выполнится сложение:

```
a + b = 45.14f
```

3. Результат преобразуется в long (дробная часть отбрасывается) и запишется в с:

```
c = 45
```

Явные преобразования

В C++ есть 5 различных оператора преобразований

1. `static_cast` — выполняет преобразования типов во время компиляции
2. `dynamic_cast` — выполняет преобразования указателей или ссылок на полиморфные объекты
3. `const_cast` — добавляет или убирает модификатор `const`
4. `reinterpret_cast` — указывает компилятору, чтобы битовое представление операнда интерпретировалось как новый тип.
5. `(new_type)value` — C-style преобразование. В данном касте компилятор пытается выполнить преобразование в следующем порядке:
 - 1) `const_cast`
 - 2) `static_cast`
 - 3) `static_cast + const_cast`
 - 4) `reinterpret_cast`
 - 5) `reinterpret_cast + const_cast`

Пример сценария, когда может быть полезен `reinterpret_cast` — преобразование указателя в число (чтобы узнать адрес):

```
int value = 1;
int *pointer = &value;
uint64_t address = reinterpret_cast<uint64_t>(pointer);
```

В современном C++:

- Обычно достаточно `static_cast` или `dynamic_cast` (но об этом виде каста в следующем семестре)
- Использование `reinterpret_cast` или `const_cast` рекомендовано лишь в крайних случаях, так как есть большой риск UB
- C-style каст не рекомендуется из-за чрезмерной неявности

Структуры и объединения

Структура

Структура — последовательная группа переменных.

Базовый пример:

```
struct Point {  
    int x;  
    int y;  
};  
  
struct Rect {  
    Point pt1;  
    Point pt2;  
};  
  
int main() {  
    // Различные виды инициализации  
    Point p1 = {1, 2};  
    Point p2 = {.x = 3, .y = 4};  
  
    Rect r1{p1, {3, 4}};  
    Rect r2 = {.pt1 = {1, 2}, .pt2 = p2};  
}
```

Пример анонимных структур:

```
struct Button {  
    struct {  
        int x, y;  
    };  
  
    struct {  
        int w, h;  
    };  
};  
  
int main() {  
    Button btn = {  
        // Обращаемся к полям напрямую!  
        .x = 0, .y = 100,  
        .w = 200, .h = 50  
    };  
}
```

Пример синтаксиса -> для разыменовывания полей структуры:

```
struct Vector {  
    int x, y;  
};  
  
void Scale(Vector* vec, int scale) {  
    // Обе строки делают одно и то же.  
    // Но первая строка красивее :)  
    vec->x *= scale;  
    (*vec).y *= scale;  
}
```

Объединение

Объединение — переменная, содержащая поля, лежащие в одной и той же области памяти (в отличие от структур, где для каждого поля предназначены разные ячейки памяти).

Базовый пример:

```
union MyUnion {  
    // Оба поля ссылаются на одну и ту же  
    // область памяти  
    double d;  
    char c[8];  
};
```

Пример вариативного объекта (паттерн Tagged Union):

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

struct Circle { Point center; int radius; };
struct Rect { Point a, b; };
struct Triangle { Point a, b, c; };

union ShapeU {
    Circle circle;
    Rect rect;
    Triangle triangle;
};

enum ShapeType {
    kCircle, kRect, kTriangle
};

struct Shape {
    ShapeType type;
    ShapeU shape;
};
```

Выравнивание

Современные процессоры умеют быстро обращаться к полям структур, если они *выровнены*. Это значит, что адрес поля `int` должен быть кратен 4, адрес `double` должен быть кратен 8, и так далее. Для этого компилятор вставляет между полями и в конце структуры *padding* — пустые байты.

Помимо этого, размер всей структуры должен быть кратным размеру самого большого поля, чтобы можно было быстро обращаться к массиву самих структур.

Рассмотрим пример:

```
struct Foo {
    char a; // 1 байт
    // [7 байт padding, чтобы выровнять b]
    double b; // 8 байт
    int c; // 4 байта
    // [4 байта padding, чтобы добить структуру до кратности 8]
};
```

Итого структура занимает 24 байта, хотя из них полезной информации на 13 байт!

Переупорядочим поля:

```
struct Foo {
    double b; // 8 байт
    int c; // 4 байта
    char a; // 1 байт
    // [3 байта padding для кратности 8]
};
```

Теперь размер структуры равен 16 байт.

Для наиболее «плотной» компоновки структуры, поля нужно размещать по порядку убывания их размера.

Подробнее про упаковку структур можно почитать в [этой замечательной статье](#).

Указатели

Указатель — переменная, хранящая в себе адрес на ячейку памяти.

Особый случай — указатель на нулевой адрес (null pointer). Используется для обозначения того, что переменная никуда не указывает.

- Для взятия указателя на объекта используется оператор &
- Для обращения к переменной по указателю (т.н. *разыменовывания*) используется оператор *

Пример — обмен значений переменных:

```
void swap(int* a, int* b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

int main() {
    int x = 2;
    int y = 7;

    swap(&x, &y);
    std::cout << x << " " << y; // 7 2

    swap(&a, nullptr); // Segmentation fault!
}
```

Указатели поддерживают арифметику: их можно складывать, вычитать. Зачем? Ответ тому — массивы.

Массивы

Массив — упорядоченная последовательность однотипных элементов. Размер массива фиксирован.

Пример:

```
int main() {
    // Различные определения массива
    int arr1[10]; // Без инициализации, заполнен мусором!
    int arr2[10] = {0}; // 10 нулей
    int arr3[] = {1, 2, 3}; // Перечисление элементов без явного количества
    int arr4[4] = {1, 2, 3, 4}; // Перечисление с явным количеством
    int arr5[2][3] = {
        {1, 2, 3},
        {4, 5, 6}
    }; // Многомерный массив

    // Обращение к элементам массива
    assert(arr2[4] == 2);
    assert(arr5[1][2] == 6)
}
```

Массив и указатель тесно связаны! По сути, массив в C++ — это указатель на его первый элемент. Более того, у указателя можно обращаться по индексу, будто перед нами массив.

```
int main() {
    int arr[] = {1, 2, 3};

    assert(arr[0] == *arr);
    assert(arr[2] == *(arr + 2));
    assert(arr == &arr[0]);

    int* p = arr;
    assert(p[1] == 2);
}
```

Отсюда заметим, что при сложении указателя и числа, число умножается на размер типа, а не тупо прибавляется как количество байт.

```
int main() {
    int arr[] = {1, 2, 3};
    assert(arr[1] == 1[arr]);
}
```

Это компилируется и работает. Думайте.

Устройство памяти

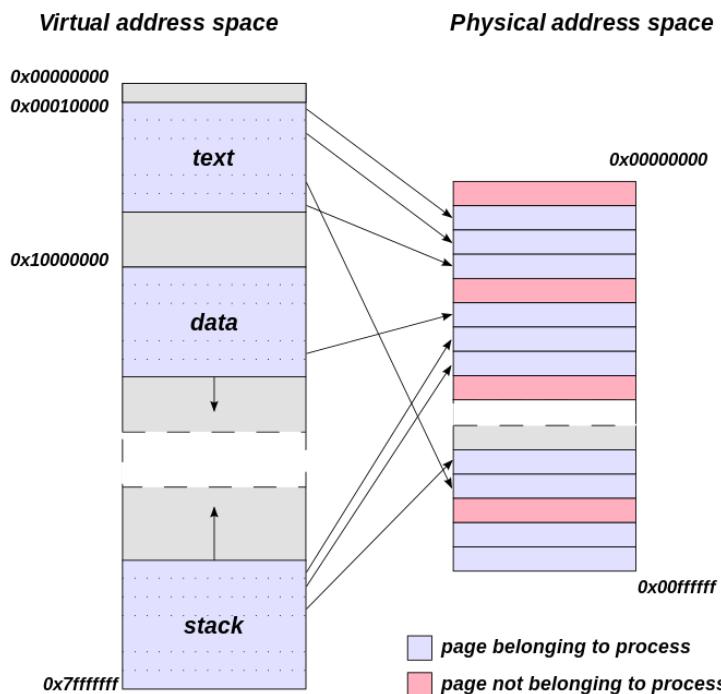
Виртуальная и физическая память

Оперативная память в компьютере представлена в виде последовательности байтов, доступных по адресу. Архитектура Фон-Неймана, используемая на современных компьютерах, также размещает сами программы в оперативной памяти. Обобщенно назовем все это *физическими адресным пространством*.

Однако современные операционные системы не дают процессам свободный доступ ко всей памяти. Вместо этого, каждому процессу предоставляется *виртуальное адресное пространство*. Программа думает, будто обращается к какой-то ячейке памяти, но вместо этого ОС решает что сделать: отобразить данный виртуальный адрес в физический, выдать содержимое файла, крашнуть прогу к чертям.

Полезностей виртуальной памяти полно, среди них:

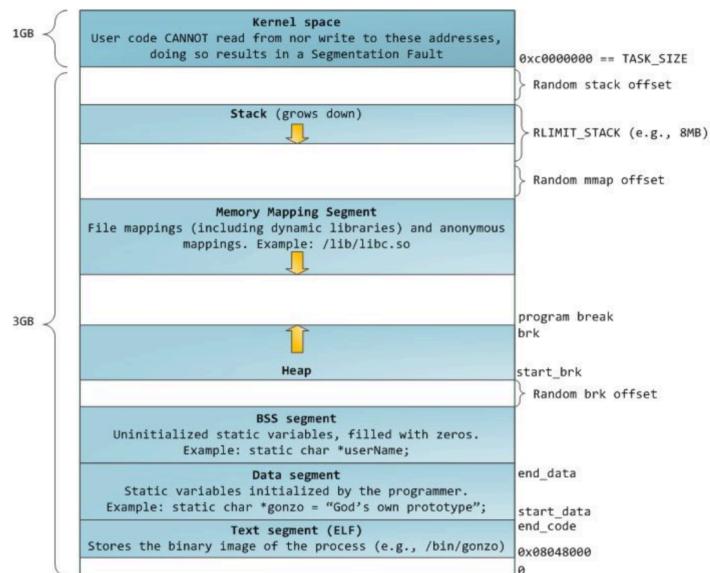
- **Изоляция:** не хотелось бы, чтобы программа могла читать (и уж тем более записывать) память других процессов. Для этого ОС создает отдельное адресное пространство для каждого процесса.
- **Безопасность:** для виртуальной памяти можно настроить права доступа. В какие-то адреса можно читать и писать, какие-то read-only, а куда-то вообще нельзя обращаться, иначе ОС просто убьет процесс.
- **Удобство:** процессы постоянно запускаются и завершаются, из-за чего физическое пространство сегментируется. С точки зрения процесса, ячейки памяти под виртуальными адресами 0x01 и 0x02 находятся рядом друг с другом, однако на физической памяти они могут находиться очень «далеко» друг от друга. ОС сама заботится о распределении физической памяти.



Отображение из виртуальной в физическую память наглядно

Сегменты памяти

Благодаря виртуальной памяти, все адресное пространство можно разделить на *сегменты* — разделы, каждая имеющая какую-то свою роль.



Сегменты памяти наглядно

В зависимости от ОС, сегменты могут варьироваться, но на Unix-системах (macOS, Linux и т.п.) можно выделить следующие:

- **Text** — машинный код программы. Read-only!
- **Rodata** — константы (например, строковые литералы). Read-only!
- **Data** — инициализированные статические переменные.
- **BSS** — неинициализированные статические переменные. Заполняется нулями.
- **Memory Mapping** — отображения файлов в память. Например, динамические библиотеки.
- **Stack** — локальные переменные, вызовы функций. *Растет вниз*.
- **Heap** — динамически выделяемая память (`malloc`, `new`). *Растет вверх*.
- **Kernel space** — всякие внутренние данные ОС. Чтение и запись запрещены!

Пример:

```

// Литерал "Hello World" в Rodata, указатель на него в Data
static const char* message = "Hello World";

// BSS
int global_number;

int main() {
    // Stack
    int local_number = 123;

    // Массив float[100] в Heap, указатель на него в Stack
    float* float_array = new float[100];

    global_number = 42; // OK
    message[0] = 'a' // Segmentation fault!
    message = "Other message" // OK
}

```

Страницы физической памяти

Операционная система делит физическую память на куски, называемые *страницами*. Обычно одна страница занимает 4 Кб, но это число может варьироваться в зависимости от ОС. Поэтому когда программа запрашивает, скажем, 100 байт на куче, на самом деле выделяется больше. *Обычно это не страшно — аллокатор сам разберется с избытком. Но это уже другая история.*

Стек вызова функций

Как было сказано ранее, на стеке хранится информация для вызываемой функции: ее аргументы, локальные переменные и адрес возврата. В общем это называется *стеком фреймом*.

«Под капотом» процессор использует два регистра:

- esp — хранит верхушку стека (незанятый байт!)
- ebp — хранит начало стека относительно функции

В C++ стек организован по соглашению cdecl.

```

void foo(int x) {
    // Кладем старый ebp на стек
    // Сохраняем текущий адрес esp в регистр ebp

    int a = 42; // Адрес: [ebp-4]
    long b = 69; // Адрес: [ebp-12]

    int d = b + x; // Сложить [ebp-12] и [ebp+8], положить по адресу [ebp-16]

    // Переходим по адресу возврата [ebp+4]
}

int main() {
    // Кладем старый ebp на стек (но ebp нулевой, так как это начало программы)
    // Сохраняем текущий адрес esp в регистр ebp

    int x = 5; // Адрес: [ebp-4]

    // Кладем на стек адрес возврата в main
    // Кладем на стек значение аргумента x,
    foo(x); // Переходим по адресу функции foo
}

```

Ссылки

Ссылка — алиас на объект. Семантически схож на указатель, но имеет ряд отличий:

- Всегда ведет на уже существующий объект
- Обязана быть инициализированной
- Не имеет арифметики
- Нельзя сделать указатель на ссылку

Базовый пример:

```
int main() {
    int num = 1;

    int& ref = num;
    ref = 123;

    std::cout << num; // 123
}
```

Пример ссылки как аргумента функции:

```
void swap(int& a, int& b) {
    int t = a;
    a = b;
    b = t;
}

int main() {
    int a = 2;
    int b = 7;
    swap(a, b);

    std::cout << a << " " << b; // 7 2
}
```

Примеры ошибок:

```
int main() {
    int& a; // Не инициализировано

    int b = 1;
    const int& c = b;
    c = 42; // Нельзя менять const ссылку
}
```

Пример UB:

```
int& foo() {
    int x = 20;
    return x;
}

int main() {
    int a = foo(); // UB
}
```

Перегрузка функций

Перегрузка функций — механизм, позволяющий объявлять несколько функций с одинаковым именем и типом возврата, но разными аргументами.

```
void foo(int x) {}
void foo(double x, int y) {}
void foo(const char* s) {}

// А вот так нельзя:
// float foo() {}

int main() {
    foo(1);
    foo(1.23, 4);
    foo("Hello");
}
```

Важно понимать, что при передаче аргументов функции, они могут неявно приводиться к другому типу. Компилятор не всегда в силах определить, какую именно функцию вызвать:

```

int foo(int x) {}
int foo(float x) {}

int main() {
    double num = 1.23;
    foo(num); // Ошибка! Компилятор не знает к чему привести num: к float или к int?
}

```

namespace

Пространства имен (namespace) — абстрактное хранилище идентификаторов: классов, функций, глобальных переменных и т.п. Используется для логической группировки идентификатора, что упрощает читаемость кода и предотвращает конфликт одинаковых имен.

- С помощью `using namespace` можно «импортировать» все идентификаторы из пространства имен в скопу.
- Можно делать алиасы на неймспейсы (например, для сокращения имени).
- Разрешены вложенные неймспейсы (но рекомендуется не больше 2-3).
- Можно не указывать имя неймспейса. Тогда его идентификаторы будут доступны только в рамках единицы трансляции (что эквивалентно ключевому слову `static`).

Базовый пример:

```

namespace Foo {
    void print() {
        std::cout << "Foo\n";
    }
}

namespace Bar {
    void print() {
        std::cout << "Bar\n";
    }
}

int main() {
    Foo::print(); // Foo
    Bar::print(); // Bar
}

```

Пример глобального `using namespace`:

```

namespace Foo {
    void print() {
        std::cout << "Foo\n";
    }
}

using namespace Foo;

int main() {
    print(); // Foo
}

```

Но так делать не рекомендуется.

Пример локального `using namespace`:

```

namespace Foo {
    void print() {
        std::cout << "Foo\n";
    }
}

int main() {
    using namespace Foo;
    print(); // Foo
}

```

Пример алиаса + вложенные неймспейсы

```

namespace SomeLongNamespace {
    namespace Foo {
        void print() {
            std::cout << "Foo\n";
        }
    }
}

int main() {
    namespace NS = SomeLongNamespace::F00;
    NS::print(); // Foo
}

```

Пример безымянного неймспейса:

```
namespace {
    void print() { std::cout << "Foo\n"; }
}

int main() {
    print(); // Foo
}
```

TODO Компиляция, этапы, ошибки компиляции

ООП

Объектно-ориентированное программирование — парадигма программирования, в которой мы описываем объекты: их свойства, поведение и взаимодействие между другими объектами.

ООП держится на нескольких принципах: абстракция, инкапсуляция, полиморфизм, наследование.

Абстракция

Придание объекту характеристик, которые определяют его концептуальные границы, отличая от остальных объектов.

Суть в том, чтобы использовать абстрактные объекты как составные детали для более сложных. Скажем, система обработки заявок скорой помощи состоит из приоритетной очереди и еще каких-нибудь абстрактных объектов.

В C++ это можно выразить в виде структуры с функциями, которые ее как-то модифицируют:

```
struct Stack {
    int len;
    int values[100];
};

void push(Stack& stack, int value);
int pop(Stack& stack);
```

Инкапсуляция

Инкапсуляция — объединение данных и поведения в единую логическую единицу, а также скрытие внутренних деталей реализации от внешнего мира:

В C++ для этого используются модификаторы доступа, а также вложение методов внутри класса/структурь:

```
class Stack {
public:
    void push(int value);
    int pop();

private:
    int len;
    int values[100];
};
```

TODO Полиморфизм

TODO Наследование

TODO Классы. Специальные методы

TODO также рассказать про ключевое слово `explicit`

TODO Перегрузка операторов

TODO Шаблоны функций, классов. Частичная и полная специализация.

RAII

RAII — Resource Acquisition Is Initialization. Идея в том, что если у нас есть какой-то ресурс, который имеет сложный жизненный цикл. Например, файл:

```
#include <cstdio>

int main() {
    FILE* f = fopen("output.txt", "w");
    if (f == nullptr) {
        std::cout << "Failed to open file\n";
        return 1;
    }

    fprintf(f, "Hello World");
    fclose(f);
}
```

Мы вручную открыли файл, проверили успех этой операции, а потом еще руками закрыли. А если мы забудем закрыть? А что, если это не файл, а указатель? Утечка памяти! А если это соединение с базой данных? Упс, положили БД.

В общем, мы хотим, чтобы жизненный цикл ресурса управлялся автоматически. Именно в этом и суть RAII. Мы оборачиваем ресурс в класс, который при конструировании инициализируется, а при деструкте закрывается.

```
class RaiiFile {
public:
    RaiiFile(const char* name, const char* mode) {
        f_ = fopen(name, mode);
        if (f_ == nullptr) {
            // handle error
        }
    }

    ~RaiiFile() {
        fclose(f_);
    }

    FILE* operator*() {
        return f_;
    }

private:
    FILE* f_;
};

int main() {
    RaiiFile f("output.txt", "w");
    fprintf(*f, "Hello World");

    // Файл закроется автоматически!
}
```

Всякие мелочи

Linkage Duration

- External linkage — символ доступен для других единиц трансляций. Переменные помечаются ключевым словом `extern`, либо `const`. Функции по умолчанию `external`.
- Internal linkage — символ доступен только внутри единицы трансляции. Помечается ключевым словом `static`.
- No linkage — символ недоступен для линковки. Пример — локальная переменная (в т.ч. статическая).

Пример:

```
extern int global = 42; // External linkage
static int static_global = 123; // Internal linkage

// External linkage (by default)
void foo() {
    int local_num = 69; // No linkage
    printf("Меня можно вызвать из других единиц трансляции\n");
}

// Internal linkage
static void bar() {
    static int hhost = 239; // No linkage
    printf("Я доступен только внутри текущего файла\n");
}
```

Storage Duration

- Static duration — переменные, живущие на протяжении всей программы. Пример — глобальные переменные, переменные с модификатором `static`.
- Automatic duration — переменные внутри блока, создающиеся в начале блока и разрушающиеся в конце. Пример — локальные переменные.
- Dynamic duration — переменные, за жизненный цикл которой отвечает пользователь. Пример — переменные, созданные через `new/malloc`
- Thread duration — переменные с модификатором `thread_local`. Но подробнее об этом поговорим в следующем семестре...