Основы Программирования

Содержание

Лекция 1. Типы данных, идентификаторы, операторы, операторы ветвления, циклы, функции

Идентификаторы

- **Идентификаторы** — это имена, используемые для обозначения переменных, типов, функций, шаблонов и т.д.
- Должны начинаться с буквы или подчёркивания, могут содержать цифры.
- Зарезервированные слова языка использовать нельзя.

Типы данных

- Для определения границ типа удобно использовать `numeric_limits<type>::param` (заголовок `<limits>`). Например: `std::numeric_limits<int>::max()`, `std::numeric_limits<`
- Соотношение размеров целочисленных типов:

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

- Стандарт **не фиксирует** конкретные размеры в байтах, только их соотношение.

Целочисленные литералы

Префикс	Система счисления
(нет)	десятичная (dec)
`0`	восьмеричная (oct)
`0x`	шестнадцатеричная (hex)
`0b`	двоичная (bin)

Вещественные литералы

Суффикс	Тип
(пусто)	`double`
`f` / `F`	`float`
`l` / `L`	`long double`

Также используется экспоненциальная запись: `1.5e10`, `2.3E-4`.

Declaration и Definition

- **Declaration (объявление)** — сообщает компилятору о существовании сущности (её имени и атрибутах).
- **Definition (определение)** — полностью описывает сущность, выделяет под неё память (для переменных) или задаёт реализацию (для функций).
- Любое определение является одновременно и объявлением.

Оператор switch

- Константы в `case` — **только целочисленные** (или приводимые к целочисленным: `enum`, `char`).
- Вычисления начинаются с первой совпавшей с константой ветки.
- Все константы должны быть **разными**.
- Если совпадения не нашлось — выполняется `default`.
- `break` вызывает выход из `switch`.

- **Сквозное выполнение (fall-through)** — без `break` выполнение «провалится» в следующий case.
- Возможен один исход для нескольких `case` (перечисляются подряд без `break`).

Функции

- **Вызов функции без определения — compile-time error** (точнее, ошибка линковки, если объявление есть, а определения нет).

Лекция 2. Указатели, массивы и строки

Указатель

- **Указатель (pointer)** — переменная, диапазон значений которой состоит из адресов ячеек памяти и специального значения — нулевого адреса.
- Значение нулевого адреса используется только для обозначения того, что указатель в данный момент не указывает ни на какую ячейку памяти.
- Зануляем либо `nullptr` (предпочтительно, C++11+), либо `0` / `NULL` (устаревший C-стиль; `NULL` — это `0`, что одновременно и `int`, и указатель — отсюда возможные неоднозначности при перегрузке функций).

void*

- `void*` — указатель на любой тип.
- Может быть явно приведён к указателю на другой тип (в C-style: `(int*)ptr`, в C++: `static_cast<int*>(ptr)`).
- Нельзя разыменовать напрямую — сначала нужно привести к конкретному типу.

Представление в виде байтов

- **Любой указатель может быть представлен в виде массива байт!**

```cpp
#include <iostream>
#include <format>

void printBytes(void* ptr, size_t size) {
    uint8_t* bytes = (uint8_t*)ptr;
    for (size_t i = 0; i < size; ++i) {
        std::cout << std::format("{:08b} ", *bytes);
        ++bytes;  // увеличивает адрес на 1, смещая на след. байт
    }
    std::cout << std::endl;
}
```

`std::format` — ещё один способ организовать форматированный вывод (C++20).

Указатель на функцию

- Синтаксис: `return_type (*name)(arg_type, arg_type, ...)`
- Присваивание: `name = func;` или `name = &func;` (эквивалентно).
- Вызов: `name(args);` или `(*name)(args);`

```
int add(int a, int b) { return a + b; }
int (*ptr)(int, int) = add;
int result = ptr(2, 3);  // 5
```

Лекция 3. Структуры, объединения

Структура (struct)

- **Структура** — это одна или несколько переменных (возможно, различных типов), которые для удобства работы с ними сгруппированы под одним именем.
- В C++ структура практически идентична классу (отличие — модификатор доступа по умолчанию: в `struct` — `public`, в `class` — `private`).

Объединение (union)

- **Объединение** — это переменная, которая может содержать (в разные моменты времени) объекты различных типов и размеров.
- Все требования относительно размеров и выравнивания выполняет компилятор.
- Объединения позволяют хранить разнородные данные в одной и той же области памяти без включения в программу машинно-зависимой информации.
- Размер объединения равен размеру самого большого его поля.

```
union Data {
    int i;
    float f;
    char str[4];
};  // sizeof(Data) == 4
```

Лекция 4. Работа с памятью

Архитектура: фон Неймана vs Гарвардская

- **2 потока**: поток данных и поток команд-вычислений.
  - В архитектуре фон Неймана они идут через одну шину памяти.
  - В Гарвардской — данные и команды разделены.
- **2 блока**: блок процессора и блоки, отвечающие за память:
  - **SSD** (сотни ГБ) и **HDD** (ГБ–ТБ) — жёсткие диски.
  - **Оперативная память** (~16 ГБ).
  - **Кэш процессора** (3 уровня: L1 — ~24 КБ, L2 — ~112 КБ, L3 — ~4 МБ) — сверхоперативная память, используемая микропроцессором для уменьшения среднего времени доступа к компьютерной памяти.

“Процесс”

- Под каждый процесс отводится своя память.
- Своё адресное пространство.
- Свои потоки для каждого процесса.
- ОС для каждого процесса создаёт иллюзию того, что он может пользоваться всей виртуальной памятью, хотя на самом деле она разделена между процессами.
- ОС хранит **взаимно-однозначное соответствие** между виртуальной памятью и физической памятью с помощью **Page Table** (выдаёт страницы памяти, забирает, отдаёт дальше).
- Для каждого процесса создаётся **своя виртуальная таблица** — одним и тем же виртуальным адресам в разных процессах могут соответствовать разные места физической памяти.
- ОС может **swap-ать** часть информации на жёсткий диск, если она используется редко.

Сегменты памяти

- Сегмент ОС (виртуальная таблица) — доступа у пользователя нет.
- **3 сегмента фиксированного размера**:
  - **Text** — исполняемый код.
  - **Data** — инициализированные глобальные/статические переменные.
  - **BSS** — неинициализированные глобальные/статические переменные.
- **3 сегмента, меняющих размер**:

- **Stack** — стек вызовов.
- **Heap** — куча.
- **Memory mapping segment** — для маппинга файлов.

```
Process id: 18636
Data segment:    00007ff6cc079000
BSS  segment:    00007ff6cc078000
Text segment:    00007ff6cc079004
Code segment:    00007ff6cc071430
Stack segment:   0000007 8c0bff884
```

> Первые сегменты расположены близко друг к другу, а Stack — где-то далеко (растёт в обратном направлении).

Stack (стек)

- Чтение и запись.
- Локальные переменные функций.
- Почему размер стека такой небольшой? — Чтобы обеспечить возможность быть достаточно близким к кэшу процессора, но при этом достаточно большим для типичных задач.
- Растёт в сторону уменьшения адресов.
- Управляется автоматически (LIFO).

Heap (куча)

- В отличие от стека, позволяет создавать динамические структуры **большого размера**.
- Управление жизнью объектов в куче — **«ручное»**.

Семейство C-функций: - `malloc` / `free` — оперируют байтами. - `malloc` **не гарантирует** выделение памяти (может вернуть `NULL`). - Не забывать **выставлять указатель в `NULL` после освобождения** (избегаем dangling pointer). - `free(NULL)` ничего не делает (безопасно). - `calloc` / `realloc` — оперируют не байтами, а массивами: - `calloc` дополнительно зануляет память. - `realloc` изменяет размер ранее выделенного блока. - Если не освобождать память — объём используемой памяти будет раздуваться (**memory leak**).

Операторы C++: - `new` / `delete`: - При нехватке памяти **выкинет исключение** (`std::bad_alloc`), в отличие от C-функций. - Занимается **инициализацией и вы-**

зовом конструктора классов (в отличие от C-функций). - Для массивов используется `new[]` / `delete[]`.

Segmentation fault

Возникает при: - Обращении к несуществующему адресу. - Обращении к сегменту, прав для которого нет. > Если объявить массив `char` через `[]` (`char s[] = "abc"`) — ему можно менять элементы. А если через * (`char* s = "abc"`) — это указатель на глобальный строковый литерал, и его уже **нельзя менять**. - Попытке поменять данные в read-only сегменте. - Обращении по нулевому указателю. - Обращении по указателю на удалённый объект (dangling pointer). - Переполнении стека (stack overflow). - Переполнении буфера (buffer overflow).

Memory mapping segment

- Для физических файлов, которые мы можем загружать на компьютер (через `mmap`).
- Используется также для подгрузки разделяемых библиотек.

Text

- Тексты, строчки (строковые литералы), исполняемый код.
- **Не доступен для записи** (read-only).

Data

- Статические переменные и константы (инициализированные).
- В случае констант — только для чтения.

BSS

- Глобальные/статические **неинициализированные** переменные.
- Запись и чтение.
- Инициализируются нулями при старте программы.

Регистры процессора

- **Регистр** — поле заданной длины во внутрипроцессорной сверхбыстрой оперативной памяти (СОЗУ).

- Используется самим процессором; может быть как доступным, так и не доступным программно.
- Например, при выборке из памяти очередной команды она помещается в регистр команд, обращение к которому программист прописать не может.
- Виды регистров:
  - Регистры общего назначения.
  - Специальные регистры для приложений.
  - Сегментные регистры.
  - Специальные регистры режима ядра.

Кадр (stack frame)

- **Кадр** — множество данных: аргументы функции, все локальные переменные функции и адрес, куда функция должна вернуть результат.
- Размер кадра известен **на этапе компиляции**.

Важно

- **Указатель находится на стэке!** (Сам указатель — локальная переменная; объект, на который он указывает, может быть в куче.)
- Из-за того, что процессор отдаёт процессам отдельные страницы виртуальной таблицы, при завершении процесса данные участки памяти будут очищены и отданы другим процессам.

Лекция 5. Компиляция

Раздельная компиляция — зачем

- Уменьшение количества строк в одном файле.
- Ускорение компиляции (пересобираются только изменённые модули).
- Разделение программы на логические модули, которые можно переиспользовать.
- Проще читать код.
- Разделение на `.h`- и `.cpp`-файлы нужно для разных этапов компиляции.

Этапы трансляции (на самом деле их около 9)

1. **Препроцессор** — создаёт блоки (единицы трансляции), включая `.h`-файлы в `.cpp`-файлы.
2. **Компиляция** — получает из каждой единицы трансляции объектные файлы (`.o`) — скомпилированный код, независимый для каждой единицы трансляции.
3. **Линковщик (Linker)** — собирает объектные файлы в одну программу.

```
a.hpp  b.hpp  c.hpp  d.hpp
  ↓      ↓      ↓
a.cpp  b.cpp  c.cpp
  ↓      ↓      ↓
── Preprocessor ──
  ↓      ↓      ↓
d.hpp  c.hpp  d.hpp
b.hpp  b.hpp  c.hpp
a.hpp  a.cpp  b.cpp
a.cpp  b.cpp  c.cpp
  ↓      ↓      ↓
─── Compiler ───
  ↓      ↓      ↓
 a.o    b.o    c.o
       ↓
     Linker
       ↓
    a.exe
```

- Раздельная компиляция позволяет перекомпилировать только изменённые

части.

Запуск этапов трансляции отдельно (на примере clang)

Флаг	Действие
`clang++ -E`	только препроцессор
`clang++ -S`	препроцессор + компиляция
`clang++ -c`	препроцессор + компиляция + ассемблирование
`clang++ --Xlinker`	запуск линковщика

Declaration vs Definition

- **Declaration** — задаёт имя и прочие атрибуты для сущностей (например, сигнатуру функции). Может встречаться **сколько угодно раз**.
- **Definition** — полностью определяет сущность; является одновременно и объявлением. Должно быть **ровно одно** (ODR — One Definition Rule).

Заголовочные файлы (`.h` / `.hpp`)

Содержат: - Объявления (иногда определения) функций. - Переменные. - Типы. - Статические объявления и определения. - Шаблоны (только определения, т.к. они требуются для инстанциации).

Этап препроцессора

- Анализирует код и ищет части, соответствующие языку препроцессора (лексический анализ).
- Исполняет директивы (команды языка препроцессора).
- **Препроцессор не знает ничего о C++**, он знает только свой язык.

Директива `#include` — включает весь контекст файла в другой файл: - `<...>` — для файлов из системных директорий. - `"..."` — для файлов пользователя (поиск начинается с текущей директории).

Директива `#define` — определяет имя, которое может быть переменной или функцией: - Просто делает контекстную замену. - Ищет знакомые слова в коде и выполняет вокруг них определённые инструкции. - **Опасность**: на этом моменте ничего не известно о типах данных (препроцессор «тупенький» в плане синтак-

сиса C++), поэтому, например, сравнения макросов могут давать неожиданные результаты.

Стражи (include guards) — помогают препроцессору не включать повторно один и тот же файл:

```
#ifndef HEADER_MATH_H
#define HEADER_MATH_H
// ... содержимое ...
#endif
```

Альтернатива — `#pragma once` (не входил в стандарт до C++17, но поддерживается большинством компиляторов).

`#pragma pack` — позволяет кучнее располагать в памяти данные структуры (контроль выравнивания).

Этап компиляции

- Смотрит на код, переданный препроцессором.
- Код компилируется отдельно, независимо файл от файла.
- Компилируются только `.cpp`-файлы (заголовочные уже в их составе).
- На выходе получаем объектные файлы (`.o`) — бинарные файлы со скомпилированным кодом.
- Промежуточный результат — ассемблерный код.

Этап линковки

- Все объектные файлы объединяются в один исполняемый.
- При этом происходит **подстановка адресов функций в места их вызова** (поэтому для использования функции достаточно лишь её объявления — определение нужно линковщику).
- По каждому объектному файлу строится таблица всех функций, которые в нём определены.

Linkage (связывание)

Свойство идентификатора, позволяющее компилятору создавать для нескольких одинаковых имён в разных единицах трансляции одну и ту же сущность. - **External linkage** — доступность из всех единиц трансляции. - **Internal linkage**

— доступность только из текущей единицы трансляции. - **No linkage** — только текущий скоуп.

Storage duration

Свойство объекта, описывающее, когда тот попадает в память и когда её освобождает. - **Automatic** — время жизни ограничено скоупом объявления. - **Static** — время жизни от запуска программы до её окончания. - **Thread** — время жизни ограничено потоком. - **Dynamic** — `new` / `delete`.

Storage class specifier

- `static` — static duration + internal linkage.
- `extern` — static duration + external linkage.
- `thread_local` — thread storage duration.

`mutable`

Можно добавить к переменным-членам класса для указания того, что данная переменная может изменяться даже в константном контексте:

```
const struct {
    int n1;
    mutable int n2;
} x = {0, 0};  // const object with mutable member
x.n2 = 4;  // OK: mutable member of a const object isn't const
```

Итого по компиляции

1. Взяли весь текст программы, включили хедеры, обработали препроцессором код, вставили макросы — создали единицы трансляции.
2. Единицы трансляции скормили компилятору, который проверил их на корректность языку C++, получили объектные файлы.
3. Линковщик связал объектные файлы, подставил адреса функций в места их вызова — получили один исполняемый файл.

Ошибки компиляции

- Текст программы после препроцессора не соответствует синтаксису языка.
- Определение функции с сигнатурой, отличной от объявления.

- Не все используемые конструкции хотя бы объявлены.

Ошибки линковки

- Не получилось связать место использования с местом определения.
- Отсутствие определения функции.
- Определение функции дважды (нарушение ODR).

  На этапе препроцессора ошибок практически быть не может (исключение — `#include` несуществующего файла).

Лекция 6. Ссылки, инициализация, перегрузка функций и именованные скоупы

Ссылки (lvalue ref)

- **«Псевдоним»** для уже существующего объекта.
- **Обязательно инициализирована** при объявлении.
- Не занимает дополнительную память (формально; на практике может реализовываться через указатель).
- **Нельзя сделать указатель на ссылку**, нельзя сделать массив ссылок.
- **Продлевают «жизнь» временным переменным** (если ссылка константная и инициализируется временным объектом).
- Скорее всего, компилятор сам передаёт адреса при передаче аргументов по ссылке, но нас это не заботит — на уровне языка это просто другое имя.

Опасности и правила

- **Возвращать ссылку на локальную переменную при выходе из функции — ошибка** (объект уничтожен, ссылка «висячая»).
- Можно возвращать ссылку, которую мы изначально передали в аргументе функции (переменная не локальна для функции).
- В случае **больших объектов нужно передавать по ссылке** (предпочтительно `const&`), чтобы не забивать стек вызова функции лишними копиями.

Перегрузка функций

- Функции с одинаковым именем, но разными списками параметров.
- **Нельзя перегружать функции с одинаковыми аргументами, но разными возвращаемыми типами** — компилятор не сможет однозначно выбрать функцию.

Инициализация — виды

Вид	Пример
Default initialization	`std::string s1;`
Value initialization	`int l{};`
Direct initialization	`std::string s4("hello");`
Copy initialization	`std::string s3 = "hello";`
List initialization	`std::string str{'a', 'b', 'c'};`

Вид	Пример
Aggregate initialization	`char a[3] = {'a', 'b'};`

Namespace

- **Предотвращают конфликт имён**.
- Могут состоять из нескольких блоков (могут быть «раскрыты» в разных местах).
- Упрощают читабельность кода.

Unnamed namespace:

```
namespace {
    void helper() { /* ... */ }
}
```

- Помогает в линковке.
- Замена `static`-функции на уровне файла (внутри неименованного namespace всё имеет internal linkage).

Namespace alias:

```
namespace SN = Foo::SomeLongNamespaceName;
```

Лекция 7. ООП. Абстракция. Инкапсуляция

Абстракция

- **Абстракция** — придание объекту характеристик, которые чётко определяют его концептуальные границы, отличая от всех других объектов.
- Основная идея: отделить **способ использования** составных объектов данных от **деталей их реализации** в виде более простых объектов.

Инвариант

- **Инвариант в ООП** — выражение, определяющее непротиворечивое внутреннее состояние объекта.
- Должен сохраняться между вызовами публичных методов класса.

Class

- **Класс** — универсальный, комплексный тип данных, состоящий из тематически единого набора «полей» (переменных) и «методов» (функций для работы с этими полями).
- Является моделью информационной сущности с внутренним и внешним интерфейсами для оперирования своим содержимым.

Инкапсуляция

- **Механизм языка**, позволяющий ограничить доступ одних компонентов программы к другим.
- **Языковая конструкция**, позволяющая связать данные с методами, предназначенными для обработки этих данных.

Access Modifiers (модификаторы доступа)

- **Public** — доступно всем.
- **Protected** — доступно классу и его наследникам.
- **Private** — доступно только классу.

Constructor (конструктор)

- **Специальный, не статический метод**, используемый для инициализации объекта (обеспечивает инвариант).

- Название этого метода совпадает с именем класса.
- Не имеет возвращаемого типа.

Виды конструкторов: - **Default constructor** (без аргументов): - Если **нет ни одного конструктора**, то дефолтный = умолчательный (компилятор сгенерирует сам). - Если объявлен хотя бы один конструктор (любой), а дефолтный — нет, **компилятор его уже не смастерит** (даже не скомпилится конструирование без аргументов). - Если в любом converting-конструкторе все аргументы сделать аргументами по умолчанию — используется как дефолтный в случае необходимости. - **Converting constructor** (из других типов). - **Copy constructor** — создаёт такой же объект (передача объекта через константную ссылку: `T(const T&)`). - **Move constructor** — передаёт значение, а сам зануляется/инвалидируется; используются rvalue ref (`T(T&&)`) — будут разбираться во втором семестре подробно.

> **Порядок инициализации полей класса** — в том же порядке, что и их **декларация (объявление)**. Сделано это для того, чтобы деструктор работал однозначно (а работает он строго **противоположно** конструктору).

Method

- **`const`-метод** — гарантия, что переменные-члены не изменятся внутри метода:

```
int getValue() const;
```

- В `const`-методе нельзя менять состояние объекта; нельзя вызывать неконстантные методы для константных объектов.

Destructor (деструктор)

- Имя: `~ClassName()`.
- Вызывается при уничтожении объекта.
- Обычно используется для освобождения ресурсов.

`operator=` (оператор присваивания)

- Принимает `const T&`, возвращает `T&`.
- **Важна проверка на самоприсваивание** (`if (this == &other) return *this;`).

- **Важно освобождение данных** в объекте, которому присваиваем, если это необходимо (new/delete).

```
MyClass& operator=(const MyClass& other) {
    if (this == &other) return *this;
    // освободить старые данные
    // скопировать новые данные
    return *this;
}
```

Ключевое слово default

- Указывает компилятору **самостоятельно сгенерировать** соответствующую функцию класса.
- Применимо к: конструктору по умолчанию, конструктору копирования, конструктору перемещения, операторам присваивания (копированием и перемещением), деструктору.

```
MyClass() = default;
```

Ключевое слово delete

- Позволяет **явно отказаться** от использования стандартного конструктора/метода.
- Введено с C++11.

```
MyClass(const MyClass&) = delete;  // запрет копирования
```

Геттеры

- Методы, возвращающие значения каких-то полей класса.
- Обычно объявляются как const.

Ключевое слово explicit

- Используется для **предотвращения неявных преобразований** при вызовах функций.

```
class MyClass {
public:
    explicit MyClass(int x);
```

```
};
MyClass obj = 5;  // ошибка: неявное преобразование запрещено
MyClass obj(5);   // OK
```

Special Members — таблица генерации компилятором

Объявлено пользователем	default constructor	destructor	copy constructor	copy assignment	move constructor	move assignment
Ничего	defaulted	defaulted	defaulted	defaulted	defaulted	defaulted
Любой конструктор	not declared	defaulted	defaulted	defaulted	defaulted	defaulted
Default constructor	user declared	defaulted	defaulted	defaulted	defaulted	defaulted
Destructor	defaulted	user declared	defaulted	defaulted	not declared	not declared
Copy constructor	not declared	defaulted	user declared	defaulted	not declared	not declared
Copy assignment	defaulted	defaulted	defaulted	user declared	not declared	not declared
Move constructor	not declared	defaulted	deleted	deleted	user declared	not declared
Move assignment	defaulted	defaulted	deleted	deleted	not declared	user declared

Rule of Three

Если определён хотя бы один из трёх методов, то надо определить все три: - Destructor - Copy constructor - Copy assignment operator

Rule of Five (C++11+)

Если определён хотя бы один из пяти методов, то надо определить все пять: - Destructor - Copy constructor - Copy assignment operator - Move constructor - Move assignment operator

Rule of Zero: лучше всего проектировать классы так, чтобы не определять ни одного из этих методов вручную, а полагаться на корректные члены-классы (умные указатели, контейнеры и т.п.).

Лекция 8. Перегрузка функций, методов и операторов

Операторы (классификация)

1. Арифметические: - Унарные префиксные: `+`, `-`, `++`, `--` - Унарные постфиксные: `++`, `--` - Бинарные: `+`, `-`, `*`, `/`, `%`, `+=`, `-=`, `*=`, `/=`, `%=`

2. Битовые: - Унарные: `~` - Бинарные: `&`, `|`, `^`, `&=`, `^=`, `|=`, `>>`, `<<`, `>>=`, `<<=`

3. Логические: - Унарные: `!` - Бинарные: `&&`, `||` - Сравнения: `==`, `!=`, `>`, `<`, `>=`, `<=`

Прочие: 1. Оператор присваивания: `=` 2. Специальные: - Префиксные: `*`, `&` - Постфиксные: `->`, `->*` - Особые: `,`, `::` 3. Скобки: `[]`, `()` 4. Оператор приведения: `(type)` 5. Тернарный оператор: `x ? y : z` 6. Работа с памятью: `new`, `new[]`, `delete`, `delete[]`

Перегрузка операторов — правила

- **Не должна противоречить здравой логике** (`+` должен делать что-то похожее на сложение).
- Может быть **членом класса** или **глобальной функцией**.
- **`[]`, `()`, `->`, `=` — всегда члены класса**.
- **Ввод (`>>`) и вывод (`<<`) — всегда глобальные функции**.
- Операторы **`::`, `.*`, `.`, `?:` — перегружать нельзя**.
- **Новые операторы создать нельзя**.

Способы перегрузки

Оператор	Как член класса	Не как член класса
`@a`	`(a).operator@()`	`operator@(a)`
`a@b`	`(a).operator@(b)`	`operator@(a, b)`
`a=b`	`(a).operator=(b)`	—
`a(b...)`	`(a).operator()(b...)`	—
`a[b]`	`(a).operator[](b)`	—
`a->`	`(a).operator->()`	—
`a@` (постфикс)	`(a).operator@(0)`	`operator@(a, 0)`

Операторы ввода / вывода

- `operator<<` и `operator>>`.

- **Не как члены класса** (если бы были — поток был бы типом, к которому применяется оператор, т.к. первый аргумент — это объект, к которому применяется метод).

```
std::ostream& operator<<(std::ostream& stream, const Type& value);
std::istream& operator>>(std::istream& stream, Type& value);
```

Ключевое слово `friend`

Иногда называют **«костылём, который ломает инкапсуляцию»**.

- **Дружественные функции** — это функции, которые не являются членами класса, однако имеют доступ к его закрытым членам (`private` и `protected`).
- Определение этих функций производится **вне класса**.
- Дружественные функции могут определяться в другом классе.
- Если нужно много функций из другого класса — можно целый класс объявить дружественным (`friend class Foo;`).

Functor (функтор) / Function object

- **Объект функции** — объект, который может вызываться как функция.
- Для этого применяется **`operator()`** (его можно перегрузить).

```
struct Adder {
    int x;
    int operator()(int y) const { return x + y; }
};
Adder add5{5};
add5(10);  // 15
```

- Имеет смысл, когда в функции должно отражаться какое-то **состояние** (которое хранится в полях функтора).

RAII (Resource Acquisition Is Initialization)

- Захват ресурса неразрывно совмещается с инициализацией объекта, а освобождение — с уничтожением объекта.
- В основе RAII лежит идея **связывания жизненного цикла ресурса** (памяти, файлового дескриптора и т.п.) с **жизненным циклом объекта в C++**.
- Это означает, что ресурсы выделяются и освобождаются **автоматически** при создании и уничтожении объектов.

Преимущества: - Автоматическое управление ресурсами. - Безопасность (включая при исключениях). - Повышение читаемости кода. - Повышение производительности. - Поддержка стандартных контейнеров.

> Когда объект RAII создаётся, он гарантирует, что ресурсы будут правильно инициализированы. Когда объект RAII выходит из области видимости, его деструктор гарантирует, что ресурсы будут корректно освобождены, **даже в случае исключения**.

Принципы: - **Тот, кто ресурс создал — тот этим ресурсом и владеет!** - **Если ресурс нужно передать — делаем это явно!** (хвост передачи владения)

Операторы разыменования и доступа

```
Foo& operator*();
Foo* operator->();
```

Часто используются в умных указателях и итераторах.

Префиксный vs постфиксный инкремент

- **Префиксный** (++x) — просто меняет данный объект, возвращает ссылку на себя. **Дешевле**.
- **Постфиксный** (x++) — сначала делает копию, возвращает копию. **Работает дольше**.
- Для постфиксного нужен фиктивный `int` в аргументе (передавать не обязательно):

```
MyClass& operator++();        // префиксный
MyClass  operator++(int);     // постфиксный
```

Приведение типов

```
operator Type() const;  // оператор приведения к типу Type
```

Проблема `malloc`

- `malloc` **не вызывает конструктор объекта**, поэтому не соблюдается инвариант.

- **`new` рулит!** (соответственно и `delete` против `free`; `realloc` тоже не зовёт конструкторы).

Class vs Structure

- По синтаксису **почти ничем не отличаются** (даже наследование работает одинаково).
- По умолчанию в `struct` всё `public`, а в `class` — `private`.
- **Идейно**: `struct` — просто набор данных (методы обычно не определяются), а `class` имеет кучу идиом и принципов ООП.

Лекция 9. ООП. Наследование

Наследование

- Позволяет описать новый класс на основе уже существующего с частично или полностью заимствующейся функциональностью.
- Класс, от которого производится наследование, называется **базовым**, родительским или **суперклассом**.
- Новый класс — **потомком**, наследником, дочерним или **производным классом**.
- Позволяет использовать **полиморфизм**.

Is-a relationship

- **«Является объектом типа»** — при `public`-наследовании объект производного класса является также и объектом базового класса.
- Возможность манипулирования объектами по ссылкам/указателям на базовые классы.

Наследник

- **Хранит в себе родителя** (физически содержит подобъект базового класса).
- **Сохраняет методы родителя** (за исключением некоторых случаев с переопределением).
- Возможно **приведение к базовому классу** (slicing — при копировании по значению теряется «лишняя» часть наследника).
- Учитывает **модификаторы доступа** при наследовании.

Порядок вызова конструкторов и деструкторов

1. **Base constructor** — конструктор базового класса.
2. **Derived constructor** — конструктор производного класса.
3. **Derived destructor** — деструктор производного класса.
4. **Base destructor** — деструктор базового класса.

   То есть деструкторы вызываются в обратном порядке относительно конструкторов.

Спецификаторы наследования

Specifier	Within Same Class	In Derived Class	Outside the Class
Private	Yes	No	No
Protected	Yes	Yes	No
Public	Yes	Yes	Yes

При наследовании: - `public` наследование — public → public, protected → protected. - `protected` наследование — public → protected, protected → protected. - `private` наследование — public → private, protected → private.

Устройство в памяти

- Выравнивание такое же, как и в структурах.
- Сначала располагаются **поля базовых классов**, а потом наследуемых.

Опасные ситуации

- Не всегда логика внешнего мира применима вполне к ООП (что-то может не выполняться — классический пример с прямоугольником и квадратом, нарушающий принцип LSP).

Множественное наследование

Если классу достаётся 2 метода или переменные одного наименования при множественном наследовании: - Можно **явно указать используемую переменную/функцию** через `BaseClass::method()`. - Можно создать ещё один метод с таким же именем и внутри вызывать тот, который нужно, из базовых (возможно, с дополнительными действиями).

Ключевое слово `final` у класса

- Говорит о том, что **наследование запрещено**.
- Пример: нет виртуального деструктора → удобно сделать `final`, чтобы предотвратить наследование.

```
class MyClass final { /* ... */ };
```

Лекция 10. ООП. Полиморфизм

Полиморфизм

- **Свойство системы**, позволяющее использовать объекты с одинаковым интерфейсом без информации о типе и внутренней структуре объекта.

Динамический полиморфизм

- **Позднее (dynamic) и раннее (static) связывание**.
- Реализуется через **виртуальные функции**.

Виртуальные функции

- Без знания реального типа класса позволяют вызывать метод того класса, **чем переменная фактически является**.
- Ключевое слово **`virtual`** — говорит о том, что в классах-потомках функция может быть переопределена.
- Ключевое слово **`override`** (необязательно, но рекомендуется) — явно указывает, что метод переопределяет функцию родителя (помогает на уровне компиляции отслеживать ошибки изменения сигнатуры).
- Ключевое слово **`final`** (у виртуальной функции) — переопределять метод в классах-потомках **нельзя**.
- Если метод **не виртуальный** — берётся метод того типа, через который вызываем (статический тип). Дёшево и сердито!

```
class Base {
public:
    virtual void foo();
    virtual ~Base() = default;
};

class Derived : public Base {
public:
    void foo() override;  // переопределение
};
```

Таблица виртуальных функций (vtable)

- **Таблица заводится для любого класса с виртуальной функцией.**

- Вызов виртуального метода — это вызов метода **по адресу из таблицы** (немного долговато из-за дополнительной косвенности).
- Каждый объект класса с виртуальными функциями хранит указатель на vtable (vptr).

  Стандарт не определяет механизм реализации виртуальных функций, однако большинство компиляторов реализуют именно таблицу виртуальных функций.

Виртуальный деструктор

- **Нужен в любом базовом классе**, от которого есть хотя бы один наследник.
- Иначе при использовании полиморфизма (например, `Base* p = new Derived(); delete p;`) будет вызван **только деструктор базового класса** → утечка ресурсов / UB.

  «Никогда не знаешь, будет ли класс наследоваться...» Варианты решения: - Можем делать **все деструкторы виртуальными** (но это проигрыш в памяти из-за vtable). - Тогда можем явно указывать `final` для тех классов, которые точно не будут наследоваться. - Либо жёстко следить за всем и вся (открыть третий глаз — опционально □).

Абстрактный класс

- **Класс, экземпляр которого не может быть создан**.
- Обычно используется в качестве **базового класса**.
- Содержит хотя бы 1 **pure virtual function** (чисто виртуальную функцию): `virtual void f() = 0;` — не определена в базовом классе, но может быть (точнее, должна быть) переопределена в потомке.

```
class AbstractShape {
public:
    virtual double area() const = 0;  // pure virtual
    virtual ~AbstractShape() = default;
};
```

NVI Idiom (Non-Virtual Interface)

Идиома невиртуального интерфейса — подход, позволяющий клиентам вызывать закрытые виртуальные функции опосредованно, через открытые невирту-

альные функции-члены.

Применение: - Перегрузка оператора вывода **только для базового класса**, где будет вызываться виртуальная функция базового класса. - Во всех наследниках переопределяем эту виртуальную функцию.

```
class Base {
public:
    void print(std::ostream& os) const { doPrint(os); }  // public non-virtual
private:
    virtual void doPrint(std::ostream& os) const;  // private virtual
};
```

Стоимость виртуальных функций

- **Лишнее обращение к таблице** vtable вместо явного адреса.
- **Невозможно сделать inline optimization** (при вызове из метода другой функции — не очень большой — код может подставиться напрямую, а виртуальный вызов так оптимизировать сложно).
- Для **коллекций объектов** — они всегда оказываются в куче (т.к. полиморфные коллекции хранят указатели).
- **Порядок объектов** в памяти также может влиять на скорость (кэш-локальность).

Коллекция объектов

- **Контейнер (массив) указателей** на базовый класс — стандартный способ хранить разнотипные объекты иерархии.

```
std::vector<Base*> shapes;
shapes.push_back(new Circle());
shapes.push_back(new Square());
```

Лекция 11. Шаблоны классов и функций

Шаблоны

> **«Способ объяснить компилятору, как мы хотим решить ту или иную шаблонную задачу.»**

- Шаблоны определяют **семейство функций, классов, типов и переменных**.
- Шаблон **параметризуется** одним или несколькими параметрами, которые могут быть:
  - **Тип** (`typename T` / `class T`)
  - **Константные выражения** (целочисленных типов, enum'ов)
  - **Указатели** (на объект, функцию, член класса)
  - **`std::nullptr_t`**
  - **Вещественные числа, литеральные типы** (с C++20)
- Реализуют **статический полиморфизм** (полиморфизм времени компиляции).
- **Не требуют дополнительных расходов** по сравнению с «прямыми» реализациями (zero-overhead).

Пример:

```
template <class T>
const T& max(const T& a, const T& b) {
    return a > b ? a : b;
}
```

> Сам по себе шаблон ещё **не является функцией** — это «рецепт» для генерации функций.

Template argument deduction (TAD)

- **Вывод аргументов шаблона** из типов фактических аргументов (подбор шаблонного параметра).
- Перед инстанциацией **все параметры шаблона должны быть известны**.
- Компилятор может вывести эти параметры из аргументов, **если это можно сделать однозначно**.

Пример:

```
int main() {
    int a = 10;
    int b = 20;
    printMe<int>(max<int>(a, b));        // явное указание типа
    printMe(max(a, b));                   // компилятор выводит сам
    printMe<CRational>(max<double>(a, b)); // явное приведение/указание
}
```

Инстанциация

- **Генерация кода** функции или класса по шаблону для конкретных параметров.
- **Без инстанциации не происходит генерации** конкретного кода шаблона.
- При использовании шаблонной функции или класса требуется **полное определение**, поэтому для использования в других единицах трансляции шаблоны требуется **определять в заголовочных файлах**.
- Шаблон генерирует «настоящий» класс или функцию.
- **Явная и неявная инстанциация:**
  - Можно явно написать тип: `func<int>(a, b);`
  - Компилятор сам может понять тип, если это однозначно: `func(a, b);`

Class Template Argument Deduction (CTAD)

- То же самое, что TAD, но для классов (с C++17).
- Позволяет не указывать шаблонные параметры при создании объекта:

```
std::pair p(1, 2.5);  // вместо std::pair<int, double> p(1, 2.5);
std::vector v{1, 2, 3};
```

Non-type template parameter

Параметры шаблона, не являющиеся типами: - Константные выражения целочисленных типов, enum'ов. - Указатели (на объект, функцию, член класса). - `std::nullptr_t`. - Вещественные числа, литеральные типы (с C++20).

```
template <int N>
class Array {
    int data[N];
```

```
};
Array<10> arr;
```

Специализация шаблонов

- Делаем то же самое, что и обычный шаблон, но рядом с `template` ставим пустые `<>`, а возле самого объекта `<тип>`, а дальше — другое определение (полное).
- Используется, если нужны **какие-то специальные действия для определённых типов**.
- Есть возможность **запретить какой-то метод**, если сделать специализацию и не писать в ней этот метод (или вообще оставить пустое определение).

Пример полной специализации:

```
template <class T>
class MyClass {
    void doSomething() { /* общая реализация */ }
};

template <>
class MyClass<bool> {
    void doSomething() { /* специальная реализация для bool */ }
};
```