



Fall 2022

compscicenter.ru

Башарин Егор

eaniconer@gmail.com

<https://t.me/egorbasharin>

Лекция VI

Classes: Alignment, Padding, new/delete, Inheritance,
Namespaces

Alignment & Padding

Alignment

- Выравнивание - положительное число, степень двойки (минимальное значение = 1)
- К объектам предъявляются требования по выравниванию в памяти: адрес объекта должен нацело делиться на выравнивание

```
#include <iostream>
#include <type_traits>

int main() {
    std::cout << alignof(int) << std::endl;
    std::cout << std::alignment_of<int>::value << std::e
}
```

Alignment

- процессор вычитывает данные по размерам, кратным машинным словам (32 or 64 bits)
- рассмотрим два расположения `int32_t` в памяти

| stack | |
|---------|---------|
| | <-- 104 |
| 4 | |
| +-----+ | <-- 103 |
| 3 | |
| +-----+ | <-- 102 |
| 2 | |
| +-----+ | <-- 101 |
| 1 | |
| +-----+ | <-- 100 |

| stack | |
|---------|---------|
| 4 | <-- 104 |
| +-----+ | |
| 3 | <-- 103 |
| +-----+ | |
| 2 | <-- 102 |
| +-----+ | |
| 1 | <-- 101 |
| +-----+ | |
| - | <-- 100 |

– В клетках указан порядковый номер бита

В каком случае чтение эффективнее? (32 bit system)

Alignment

- компилятор по умолчанию делает эффективнее
 - выравнивает данные
 - платформно- и компиляторо-зависимо
 - пример: на ARM допустимы только выровненные по 4 int'ы

Alignment & Padding

```
#include <iostream>
```

```
struct T {  
    char a;  
    int32_t b;  
};
```

```
int main() {  
    std::cout << sizeof(T) << " " << alignof(T);  
}
```

```
clang++ -Xclang -fdump-record-layouts
```

Make alignment stricter

```
struct alignas(8) T {  
    char a;  
    int32_t b;  
};
```

```
struct T {  
    alignas(8) char a;  
    int32_t b;  
};
```

```
alignas(64) char cacheline[64];
```


Object and Value Representations

- Object Representation: `sizeof(T)` последовательных объектов типа `unsigned char`
- Value Representation: биты, хранящие значение объекта
 - NB: `sizeof(struct_type) >= sum(sizeof(member_i_type))`

new/delete Expression

- **new**:
 1. memory allocation
 2. object construction
 3. address returning
- **delete**
 1. object destruction
 2. memory deallocation
- **godbolt** с иллюстрацией этапов

operators

- `new/delete` *выражения* используют соответствующие **операторы** для выделения памяти
- их можно переопределять:
 - глобально
 - для отдельного класса (как static-методы)
 - \Rightarrow в коде можно увидеть `::operator new,`
`::operator delete`

in-class overloading example

```
#include <iostream>
```

```
struct Node {  
    Node* child_ = nullptr;  
  
    Node() : child_(nullptr) {}  
    Node(Node* child) : child_(child) {}  
    ~Node() { delete child_; }  
  
    static void* operator new(size_t size) {  
        std::cout << "My new operator with size: "  
                    << size << std::endl;  
        return ::operator new(size);  
    }  
  
    static void operator delete(void *p) {
```

Inheritance

Наследование позволяет:

- расширять уже существующие классы
- работать с объектами разных типов однородно (через базовый класс)
- ...

Syntax

```
class|struct derived-class-name:  
    { access-specifier [virtual] base-class-name, ... }  
{ member-specification }
```

access-specifier – public, protected, private

влияет на доступ к открытым членам класса base-class-name в наследнике

Пример

```
struct GameObject{ Point position; };
```

```
class Car: private GameObject {  
    Point vel;  
    double orien;  
    double omega;  
};
```

```
struct Prize: GameObject {  
    int value;  
};
```

Object Repr

GameObject:

| |
|----------|
| +-----+ |
| position |
| +-----+ |

Prize:

| | |
|------------|---------|
| +-----+ | +-----+ |
| position | value |
| +-----+ | +-----+ |
| GameObject | |

Преобразования Base \leftarrow Derived

Определены автоматически:

```
Prize p{ Point{...}, 100 };  
GameObject &go = p;  
GameObject *goPtr = &p;
```

Следовательно, родитель копируем от объекта-наследника:

```
Prize p{ Point{42, 24}, 100 };  
GameObject go = p;    // GameObject(const GameObject&);  
                      // GameObject{Point{42, 24}}
```

- $\text{Base\&} \leftarrow \text{Derived\&}$ И $\text{Base*} \leftarrow \text{Derived*}$ АВТОМАТИЧЕСКИ
- \Rightarrow срезка (создание копии): поля только базового класса, утрата других значений/инвариантов и т.д.

Особенности

- Базовый класс должен быть определен до наследования
- Из наследника нет доступа к `private` полям базового класса, есть к `public` и `protected`
 - `protected` наследование: `public`-поля `Base` доступны только в `Derived` [*]

Derived::Derived

- конструкторы не наследуются (и не бывают виртуальными)
- сконструировать Base-часть до Derived **необходимо**
 - явно или через конструктор по-умолчанию
 - до выполнения списка инициализации полей Derived
- порядок конструирования: Base1, Base2 (в порядке объявления наследования), Derived
 - вызовы деструкторов — в обратном порядке

Derived::Derived

```
struct GameObject {  
    // no default constructor  
    GameObject(Point position) : position{position} {}  
  
    Point position;  
};  
  
struct Prize: GameObject {  
    Prize(Point pos, int val)  
        : GameObject{pos}  
        , val{val}  
    {}  
  
    Prize(Point p)  
        : position(p) // error! not in init-list  
        // error! no default ctor for base GameObject  
    ..  
};
```

Methods overriding

```
struct GameObject {
    /* ... */
    void CalcShift() { /* ... */ }
};

struct RoadSign: GameObject {
    /* ... */
    void CalcShift() {
        if (HitByCar()) { InitShifting(); }
        GameObject::CalcShift(); // явный вызов метода
    }
};

int main() {
    GameObject go; go.CalcShift();
    RoadSign    rs; rs.CalcShift();
}
```

- методы совпадают по сигнатуре
 - откуда компилятор знает, какой позвать?
- такой перегрузкой покрываются не все случаи

Virtual methods motivation

```
struct GameObject {  
    void CalcShift() { /* ... */ }  
};  
  
struct RoadSign: GameObject {  
    void CalcShift() { /* ... */ }  
};  
  
std::vector<GameObject *go> objects;  
  
int main() {  
    GameObject *go = new RoadSign{...};  
    objects.push_back(go);  
    objects[0]->CalcShift();  
}
```

Virtual methods syntax

```
virtual member-function [override] [final] [= 0;]
```

`override` — компилятор проверит, что функция с такой сигнатурой есть в предке

`final` — запрет переопределения в потомках

`= 0;` — pure virtual function (class -> abstract class, нельзя создавать объекты)

Virtual methods example

```
struct GameObject: VisibleObject {  
    virtual void CalcShift() { /* ... */ }  
};  
  
struct RoadSign: GameObject {  
    virtual void CalcShift() override {  
        /* ... */  
        GameObject::CalcShift();  
    }  
};
```


Abstract classes example

```
struct VisibleObject {  
    virtual void Render() = 0;  
}  
  
// допустима своя реализация вне класса  
void VisibleObject::Render() { /* ... */ }  
  
struct RoadSign: VisibleObject {  
    virtual void Render() final { // метод *надо* переопределить  
        /* ... */  
  
        // реализацию можно заиспользовать явно  
        VisibleObject::Render();  
    }  
};
```

Virtual methods

- реализация — vtable
 - таблица виртуальных функций (в начале класса)
 - создание объекта в т.ч. подставляет адрес на правильный vtable
- **важен** виртуальный деструктор при наследовании
- виртуальные методы **не стоит** использовать в конструкторах и деструкторах

GameObject Repr

GameObject:

| | |
|---------|----------|
| vtable* | position |
|---------|----------|

| | |
|-----------|------|
| GoVtable: | addr |
|-----------|------|

| | |
|------------|------|
| CalcShift: | 0x42 |
|------------|------|

Prize Repr

Prize:

| | | |
|---------|-------------|---------|
| +-----+ | +-----+ | +-----+ |
| vtable* | position | value |
| +-----+ | +-----+ | +-----+ |
| | GameObject* | |

| | |
|--------------|---------|
| PrizeVtable: | +-----+ |
| | addr |
| | +-----+ |
| CalcShift: | 0x24 |
| | +-----+ |

Namespaces

Зачем нужны: * Позволяют различать одинаковые имена ** до
неймспейсов — префиксы имен: `struct XML_Parser,`
`int XML_Get...` * Лучше структурируется код * ...

names

Имена в C++ — это обозначение конкретных сущностей:

- variables, constants
- functions
- structs, classes, enums, unions
- templates
- typedefs, usings
- namespaces

namespace + inner example

```
namespace ns {  
    int A = 42;  
  
    namespace ns_inner {  
        void foo(int) { ... }  
    }  
}  
  
// использование  
int B = ns::A;    // B — имя из _глобального_ пространства  
ns::ns_inner::foo(ns::A);
```

namespace extension

```
namespace ns {  
    namespace ns_inner {  
        int A = 42;  
    }  
  
    namespace ns_inner_other {  
        int A = 42;  
    }  
  
    namespace ns_inner {  
        int B = 24;  
    }  
}
```


usage syntax

- `operator::` — ищет имя в соответствующем пространстве имен
 - `my_namespace::my_func`
 - `::func` — поиск в глобальном пространстве имен
- `std::string::npos` — классы и структуры задают свое пространство имен

алгоритм поиска имен

В процессе компиляции, когда нужно *разрешить* имя:

1. Если оно есть в текущем неймспейсе — остановиться и выдать все одноименные сущности
2. Если текущий неймспейс глобальный - выдать ошибку
3. Перейти к родительскому неймспейсу
4. Повторить сначала

example

```
int foo(int i) { return 1; }

namespace ns {
    int foo(float f) { return 2; }
    int foo(double a, double b) { return 3; }
    namespace ns_inner {
        int global = foo(5);
    }
}
```

- когда какое-либо имя найдено — остановка
- выбор функции из перегрузок — из найденных имен

Koenig lookup (ADL)

```
namespace ns {  
    struct Point { ... };  
    Point operator+(Point a, Point const& b);  
}  
  
int main() {  
    ns::Point a(1,2);  
    ns::Point b(3,4);  
    b = ns::operator+(a, b); // ОК  
    b = a + b;               // ТОЖЕ ОК  
    return 0;  
}
```

- для имен функций (\Rightarrow и операторов)
- на первой фазе алгоритма поиска
- дополнительно рассматриваем пространства имен, из которых аргументы

using namespace <smth>

включает все имена из неймспейса в текущий

```
namespace my_global { namespace ns {  
    int A = 42;  
    namespace ns_inner {  
        int foo(int i) { return i; }  
    }  
} // namespace ns  
  
using namespace ns;  
int B = A; // -> ns::A  
int C = ns::ns_inner::foo(B); // -> ns::ns_inner::foo  
  
using namespace ns::ns_inner;  
int D = foo(B); // -> ns::ns_inner::foo  
  
} // namespace my_global
```

using <smth>

включает одно имя из неймспейса в текущий

```
namespace ns {  
    int A = 42;  
  
    namespace ns_inner {  
        int foo(int i) { return i; }  
    }  
} // namespace ns  
  
using ns::A;  
int B = A; // -> ns::A  
int C = ns::ns_inner::foo(B);
```

anonymous namespace

Имя безымянного пространства имен уникально (генерируется компилятором)

```
namespace {  
    int A = 42;  
    /* ... */  
}
```

```
// translated to:  
namespace $uniq {  
    int A = 42;  
    /* ... */  
}
```

```
using namespace $uniq;
```

- Полезно для сокрытия деталей
- Зачастую заменяет static-переменные (в заголовочных файлах) — создает уникальные сущности