Объектно-ориентированное программирование

Object-oriented programming

VII. Наследование

Inheritance

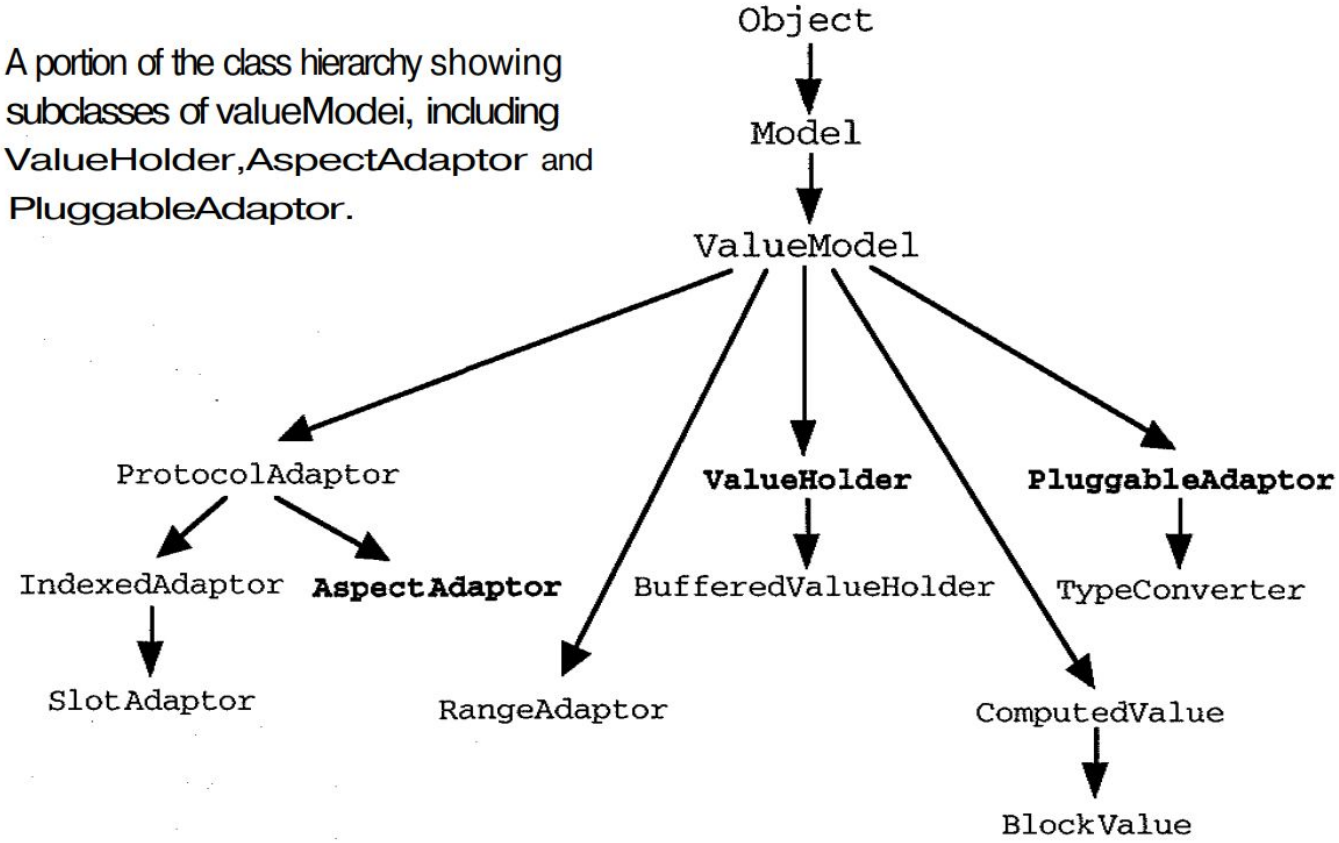
"**Inheritance** programming shows the power of **differential description**. A **generic** 'object' is displayed as a cloud. One can make a rectangle from the undifferentiated object by saying, in effect, '*I want something just like that except...*'"

A. C. Kay

"Computer Software (1984)"

VII. Inheritance

A portion of the class hierarchy showing subclasses of valueModei, including ValueHolder,AspectAdaptor and PluggableAdaptor.



https://rmod-files.lille.inria.fr/FreeBooks/Art/artAdded174186187Final.pdf

Наследование в Smalltalk-76

```
Class new title: 'Window';
    fields='frame';
    asFollows!
Scheduling
startup
    [frame contains: stylus =>
        self enter.
        repeat:
            [frame contains: stylus loc =>
                [keybard active => [self keyboard]
                stylus down => [self pendown]]
            self outside => []
            stylus down => [^self leave]]]
        ^false]
Default Event Responses
enter [self show]
leave
outside [^false]
pendown
keyboard [keyboard next. frame flash]
Image
show
```

```
Class new title= 'DocWindow';
    subclassof: Window;
    fields='document scrollbar edit Menu';
    asFollows!
Event Responses
enter [self show.editMenu show.scrollbar show]
leave [document hideselection.editMenu
hide.scrollbar hide]
outside
    [editMenu startup => []
    scrollbar startup => [self showDoc]
    ^false]
pendown [document pendown]
keyboard [document keyboard]
Image
show [super show.self showDoc]
showDoc [doucument showin: frame at: scrollbar
position]
title [^document title]
```

"We needed a better theory about inheritance entirely (and still do). For example, **inheritance** and **instancing** (which is a kind of inheritance) muddles both **pragmatics** (such as factoring code to save space) and **semantics** (used for way too many tasks such as: specialization, generalization, speciation, and so forth)."

A. C. Kay

"Early History of Smalltalk (1993)"

“A number of **inheritance ideas** were tried out in Smalltalk-72, **none of which I liked** all that much. My favorite were the "slot inheritance" experiments that were done by Larry Tesler. **Slot inheritance** was invented for AI and other **reasoning systems** – and also for providing protection of **polymorphic** meanings. For many objects, the contents of the instance variables can be thought of as "parts". Thus, one could imagine a lookup strategy that would search down through the instance variables for ideas and info about the object.”

A. C. Kay

"One problem with almost all **inheritance** mechanisms is that they **compromise data abstraction** to an extent. In languages with inheritance, a data abstraction implementation (i.e., a class) has two kinds of users. There are the “outsiders” who simply use the objects by calling the operations. But in addition there are the “insiders.” These are the **subclasses**, which are typically permitted to **violate encapsulation**. There are three ways that **encapsulation can be violated**: the subclass might **access** an instance variable of its superclass, **call** a private operation of its superclass, or **refer** directly to superclasses of its superclass."

B. Liskov *(1987)*

“Data Abstraction and Hierarchy”

https://www.cs.tufts.edu/~nr/cs257/archive/barbara-liskov/data-abstraction-and-hierarchy.pdf

"[**Smalltalk**] encourages people to see inheritance as the sole or at least primary way of organizing programs and to organize **classes** into **single-rooted hierarchies**. In **C++**, **classes are types** and inheritance is by no means the only means of organizing programs."

B. Stroustrup

VII.a. Язык C++

C++

Опять Simula

“Such **partial similarity** fairly often applied to processes in different simulation models, indicating that programming effort could be saved by somehow preprogramming the **common properties**”

K. Nygaard

(1978)

Иерархии классов в Simula

```
link class car (license number, weight);
    integer license number; real weight; …
car class truck (load);
    ref (list) load; …
car class bus (capacity);
    integer capacity;
    begin ref (person) array passenger
        [1 : capacity] …
    end;

list class bridge;
   begin real load; …
   end;
```

Влияние **Simula** на **C++**

- Классы ведут себя как сопрограммы (легко писать симуляции многопоточности)
- Конструкторы, оператор **new**
- Статическая система типов[1] данных (компилятор выявляет ошибки программиста еще до запуска программы)
- Программы легко организовываются в иерархию подпрограмм[2]
- Сам язык оставлял желать лучшего (linking time, связывание классов занимало очень много времени[3], большие программы писать было тяжело; run-time type checking; garbage collection etc.)
- **Классы позволяют мыслить о сущностях в программах напрямую**

https://dl.acm.org/doi/pdf/10.1145/234286.1057836

https://www.youtube.com/watch?v=ZXc_z1sNbfA

Классы позволяют мыслить о сущностях в программах напрямую

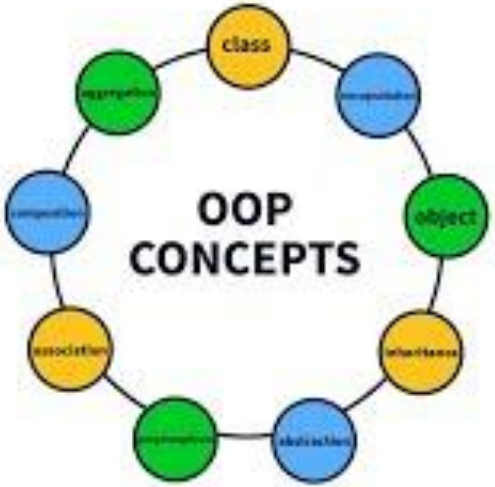
“A compile-time* **hierarchy**[1] that matches the **domain model**.”

M. “Mahk” LeBlanc

*Не хватает уточнения, что она инкапсулирована

C++ Object-oriented hierarchy of encapsulation

https://www.youtube.com/watch?v=04ksL1hf_p8



Что такое ООР?

1. Объект – это описание процесса, представляющего собой инкапсулированную систему, все отношения которой с другими системами устанавливаются посредством какого-то протокола и определяют комплексные системы в какой-то предметной области (domain).
   a. ООП – это программирование цепочек сообщений от одних объектов к другим согласно общим правилам.
2. Объект – это модель сущности из предметной области (domain), которая описывает все отношения этой модели с другими моделями из той же области.
   a. ООП – это программирование иерархий объектов для их последующего использования в виде основных компонентов, которые взаимодействуют с программой.

“The most **treacherous** metaphors are the ones that seem to work for a time – because they can **keep** more **powerful insights** from bubbling up. As a result, progress is slow.”

J. Raskin (1994?)

VII.b. Механизм наследования в C++

Inheritance in C++

Три основных элемента дизайна C++

- Организационная структура программы как в Simula (иерархии классов, многопоточность, статическая система типов)

> "Support for program organization--that is, **classes**, some form of class hierarchies, some form of support for concurrency, and **strong** (that is, **static**) checking of a type system based on classes. This I saw as **support for the process of inventing programs**"
>
> ***B. Stroustrup***

Три основных элемента дизайна **C++**

- Организационная структура программы как в Simula (иерархии классов, многопоточность, статическая система типов)
- Высокая производительность[1] (как при сборке программ, так и в работе)

> “Produce programs that ran as **fast** as BCPL programs and shared BCPL's ability to **easily combine** separately compiled **units** into a program”
>
> ***B. Stroustrup***

Три основных элемента дизайна **C++**

- Организационная структура программы как в Simula (иерархии классов, многопоточность, статическая система типов)
- Высокая производительность (как при сборке программ, так и в работе)
- Портативность (“железо”, операционные системы)

> “The "good" implementation I needed would typically **not** be **available** until "next year" and only on a machine I **couldn't afford**. This implied that a tool must have multiple sources of implementations.”
>
> ***B. Stroustrup***

Организационная структура программы

1. Классы (“абстрактные типы данных”)
2. Подклассы
3. Контроль доступа (**public**/**private**)
4. Конструкторы и деструкторы
5. Функции-обертки (**call**/**return** wrappers)
6. Дружественные классы (**friend**)
7. Статическая система типов и приведение типов для аргументов функций
8. “Инлайнинг” функций[1] (**inline**)
9. Аргументы по умолчанию
10. Перегрузка оператора присваивания

Основные инструменты **C++**

1. Виртуальные функции[1].
2. Перегрузка функций и операторов.
3. Ссылочные переменные.
4. Константные переменные - **const**.
5. Классы как **интерфейсы**.

Абстрактные типы данных против классов

```
struct date { int day, month, year; };
struct date today;
extern void set_date();
extern void next_date();
extern void print_date();
extern void next_today();
```

```
class date {
    int day, month, year;
    friend void set_date(date*, int, int, int),
                next_date(date*),
                print_date(date*), next_today();
};
```

Зачем нужны виртуальные[1] функции

```
enum kind { circle, triangle, square };

class shape {
    point center;
    color col;
    kind k; // необходимо дополнительное поле для конкретизации вида фигуры
public:
    // реализация открытых методов здесь
    void draw() {
        switch(k) {
        case circle: // логика, которая рисует круг
            break;
        case triangle: // логика, которая рисует треугольник
            break;
        case square: // логика, которая рисует квадрат
        }
    }
};
```

Виртуальные функции в **C++**

```
class shape {
    point center;
    color col;
    // дополнительное поле не нужно
public:
    virtual void draw();
    // реализация открытых методов здесь
};

class circle : public shape {
    int radius;
public:
    void draw () { /* логика, которая рисует круг */ }
};

void draw_all(shape** v, int size) {
    for (size_t i = 0; i < size; ++i) v[i].draw();
}
```

Поддержка ООП в C++

1. Наследование (1979)
2. Виртуальные функции (1983)
3. Инкапсуляция (1983)
4. Абстрактные классы и множественное наследование (1987)

- Большинство классов не предназначены для включения в иерархии
- Большинство иерархий не нуждается во множественном наследовании
- Множественное наследование интерфейсов необходимо в контексте статических типов
- Единого универсального базового класса быть не может без удара по производительности
- Виртуальные вызовы должны быть эффективными
- Модель наследования C++ имеет сильное теоретическое обоснование[1]

Что такое ООР?

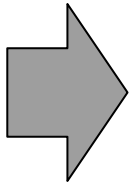
"*Object-oriented* programming is **programming using inheritance**. *Data abstraction* is **programming using user-defined types**. With few exceptions, object-oriented programming can and ought to be a *superset of data abstraction*."

B. Stroustrup

Association of Simula Users Conference, 1986

Наследование в C++

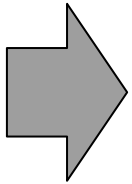
```
class A {
    char c{0}; // 1 байт
};

class B : A {
    char c{0}; // 1 байт
};
```



```
Class A
   size=1 align=1
   base size=1 base align=1
A (0x0x3b26480) 0

Class B
   size=2 align=1
   base size=2 base align=1
B (0x0x2971340) 0
  A (0x0x3b264e0) 0
```

Множественное наследование в C++

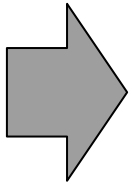
```
class A {
    char c{0};
};

class B {
    char c{0};
};

class C : A, B {
    char c{0};
};
```



```
Class A
   size=1 align=1
   base size=1 base align=1
A (0x0x3916480) 0

Class B
   size=1 align=1
   base size=1 base align=1
B (0x0x39164e0) 0

Class C
   size=3 align=1
   base size=3 base align=1
C (0x0x3929000) 0
  A (0x0x3916540) 0
  B (0x0x39165a0) 1
```

Ромбовидная иерархия наследования в C++

```
class I {
    char c{0};
};

class A : I {
    char c{0};
};

class B : I {
    char c{0};
};

class C : A, B {
    char c{0};
};
```



```
Class I
   size=1 align=1
   base size=1 base align=1
I (0x0x3906480) 0

Class A
   size=2 align=1
   base size=2 base align=1
A (0x0x2971340) 0
  I (0x0x39064e0) 0

Class B
   size=2 align=1
   base size=2 base align=1
B (0x0x29713a8) 0
  I (0x0x3906540) 0
```

```
Class C
   size=5 align=1
   base size=5 base align=1
C (0x0x391a000) 0
  A (0x0x2971410) 0
    I (0x0x39065a0) 0
  B (0x0x2971478) 2
    I (0x0x3906600) 2
```

Наследуемые функции против виртуальных функций

```
class Base {
protected:
   int x;
public:
   void fbase() { x = 0; }
};

class Derived : public Base {
public:
   void fderived() {
       fbase();
   }
};
```

```
void f(Base& b) {
   b.fbase();
}

void g(Derived d) {
   d.fbase();
}
```

Наследуемые функции против виртуальных функций

```
class Base {
protected:
   int x;
public:
   void fbase() { x = 0; }
};

class Derived : public Base {
public:
   void fbase() {
       Base::x = 1;
   }
};
```

```
void f(Base& b) {
   b.fbase();
}

void g(Derived d) {
   d.fbase();
}
```

```
Base x;
f(x);

Derived y;
f(y);
```

Какая функция из блока слева будет вызвана?

Наследуемые функции против виртуальных функций

```
class Base {
protected:
   int x;
public:
   void fbase() { x = 0; }
};

class Derived : public Base {
public:
   void fbase() {
       Base::x = 1;
   }
};
```

```
void f(Base& b) {
   b.fbase();
}

void g(Derived d) {
   d.fbase();
}
```

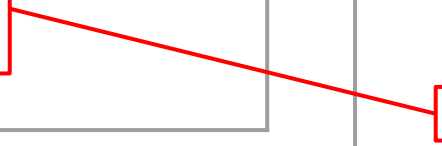
```
   Base x;
   f(x);

   Derived y;
   f(y);
```

Наследуемые функции против виртуальных функций

```
class Base {
protected:
   int x;
public:
   virtual void fbase() {
       x = 0;
   }
};

class Derived : public Base {
public:
   void fbase() {
       Base::x = 1;
   }
};
```

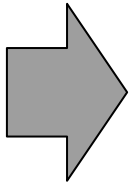
```
void f(Base& b) {
   b.fbase();
}

void g(Derived d) {
   d.fbase();
}
```

```
   Base x;
   f(x);

   Derived y;
   f(y);
```



Наследование в C++ с виртуальными функциями

```
class A {
    char c{0};
virtual void f();
};

class B : A {
    char c{0};
};
```



```
Vtable for A
A::_ZTV1A: 3 entries
0     (int (*)(...))0
8     (int (*)(...))(& _ZTI1A)
16    (int (*)(...))A::f

Class A
   size=16 align=8
   base size=9 base align=8
A (0x0x3a0e480) 0
    vptr=((& A::_ZTV1A) + 16)
```

```
Vtable for B
B::_ZTV1B: 3 entries
0     (int (*)(...))0
8     (int (*)(...))(& _ZTI1B)
16    (int (*)(...))A::f

Class B
   size=16 align=8
   base size=10 base align=8
B (0x0x3801340) 0
    vptr=((& B::_ZTV1B) + 16)
  A (0x0x3a0e4e0) 0
      primary-for B
(0x0x3801340)
```

Высокая производительность

“What you don’t use, you don’t pay for. What you do use, you couldn’t hand code any better.”

B. Stroustrup

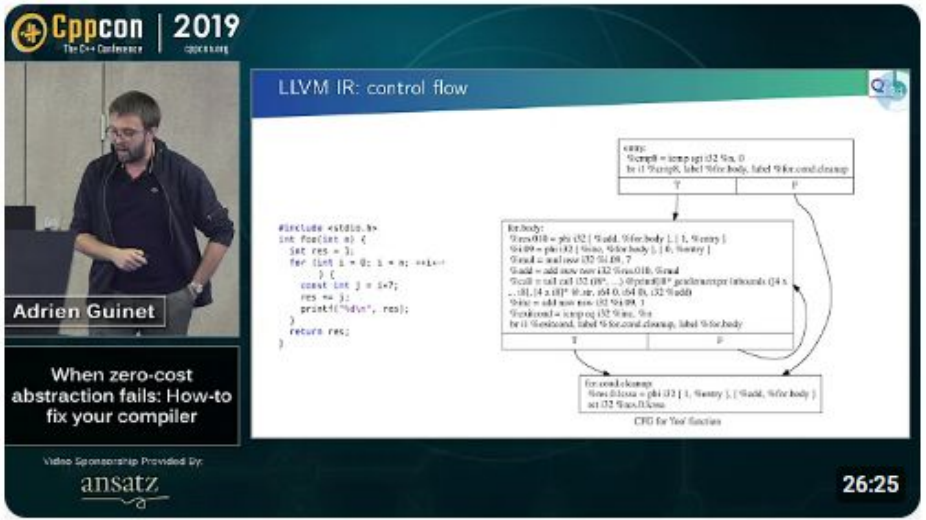
VII.c. Высокая производительность в C++

Zero-cost abstractions

“Zero-cost” abstractions

https://www.youtube.com/watch?v=rHIkrotSwcc

https://www.youtube.com/watch?v=EPI7dW5CUfc



leon.brinzan@iis.utm.md

Dennis Gustafsson
@voxagonlabs

Game developers, are you using STL containers (std::vector, std::unordered_map, etc)? If no, what's the main reason (runtime perf or debug perf/compile times/error messages/etc)?

Yes	34.2%
No, because runtime perf	15.1%
No, bc everything else	24.7%
Sometimes / some of them	26%

1,335 votes · Final results

8:27 AM · Sep 13, 2024 · **29.7K** Views

M @mfukar · 19h
stdlib containers are not fit for any purpose, including teaching because their implementation is what i can only leniently call fucking stupid

kinjal kishor @kinjalkishor · Sep 13
std::pmr containers are quite good though

Andre Weissflog @FlohOfWoe · 23h
IMHO the most convincing argument is this though:

godbolt.org/z/dWxhGhrMd

Kingsley Hopking @BeardyKing · 10h
I often use the STL when writing offline tools & prototyping a feature i.e when memory fragmentation & breakneck perf is less of a concern

In engine code I mainly use C / custom types along with custom allocators as I know the lifetime & perf requirements of the code I'm writing

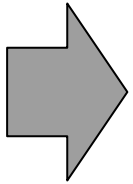
Steve Verreault @sveroverr · 23h
I do, usually wrapped in other interfaces, but C++ deserved a better standard library. They feel academic, as though designed by someone divorced from any real day to day practice, but with a theoretical understanding. Also they're not OO friendly in an OO language.

Anders Lindqvist @anders_breakin · Sep 13
On my 13 year old home computer I mostly avoid them due to DEBUG performance. I often start out using them and then removing them over time. I'm waiting for a new computer, maybe my feelings will change :)

Stefano Cristiano @pagghiu_ · 20h
- Terrible compile time and debug performance
- Bad runtime performance in some cases
- No custom inline-storage buffers (SmallVector<T,N> acting as Vector<T>)
- Exceptions instead of return values for failures

https://www.godbolt.org/z/dWxhGhrMd

```
#include <vector>
```



x86-64 clang 18.1.0 | -E -std=c++23

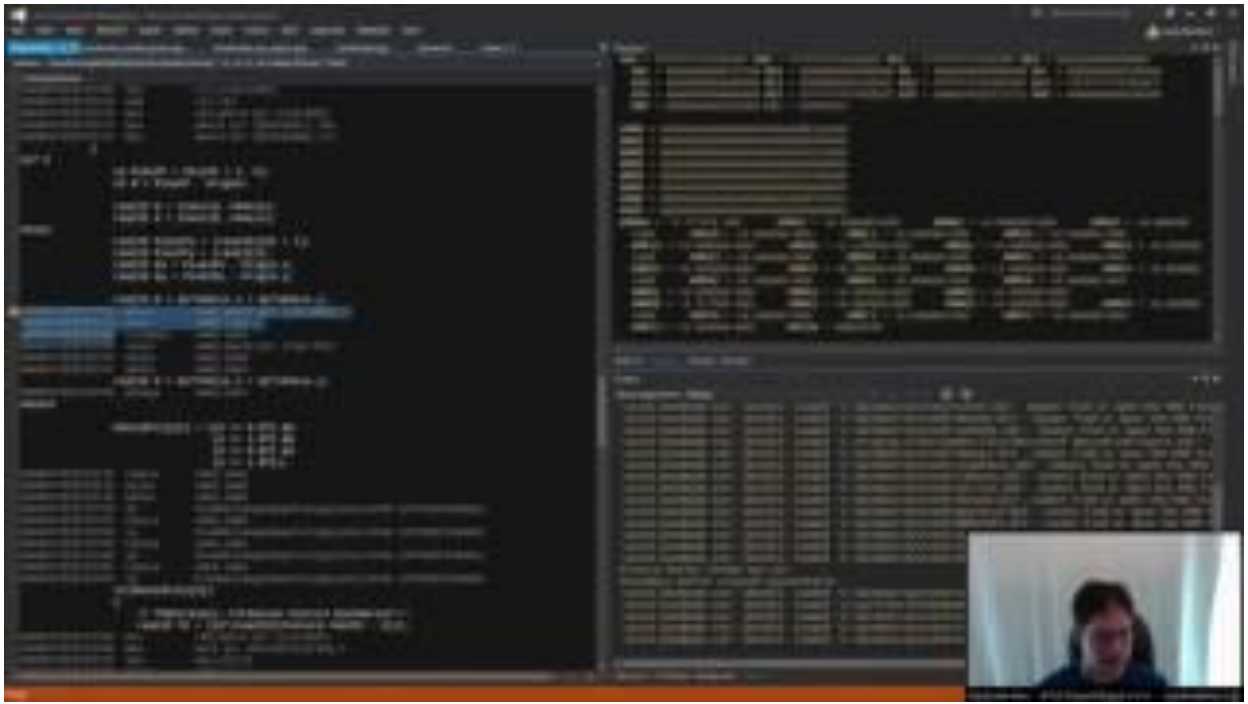
Output... | Filter... | Libraries | Overrides | Add new... | Add tool...

```
24207         const auto __end = __ucont.end();
24208         auto __removed = std::__remove_if(__ucont.begin(), __end,
24209        __ops::__pred_iter(std::ref(__pred)));
24210         if (__removed != __end)
24211     {
24212       __cont.erase(__niter_wrap(__cont.begin(), __removed),
24213            __cont.end());
24214       return __osz - __cont.size();
24215     }
24216
24217         return 0;
24218       }
24219
24220      template<typename _Tp, typename _Alloc, typename _Up>
24221        constexpr
24222        inline typename vector<_Tp, _Alloc>::size_type
24223        erase(vector<_Tp, _Alloc>& __cont, const _Up& __value)
24224        {
24225          using namespace __gnu_cxx;
24226          std::vector<_Tp, _Alloc>& __ucont = __cont;
24227          const auto __osz = __cont.size();
24228          const auto __end = __ucont.end();
24229          auto __removed = std::__remove_if(__ucont.begin(), __end,
24230         __ops::__iter_equals_val(__value));
24231          if (__removed != __end)
24232     {
24233       __cont.erase(__niter_wrap(__cont.begin(), __removed),
24234            __cont.end());
24235       return __osz - __cont.size();
24236     }
24237
24238         return 0;
24239       }
24240
24241   }
24242   # 1 "/app/example.cpp" 2
```

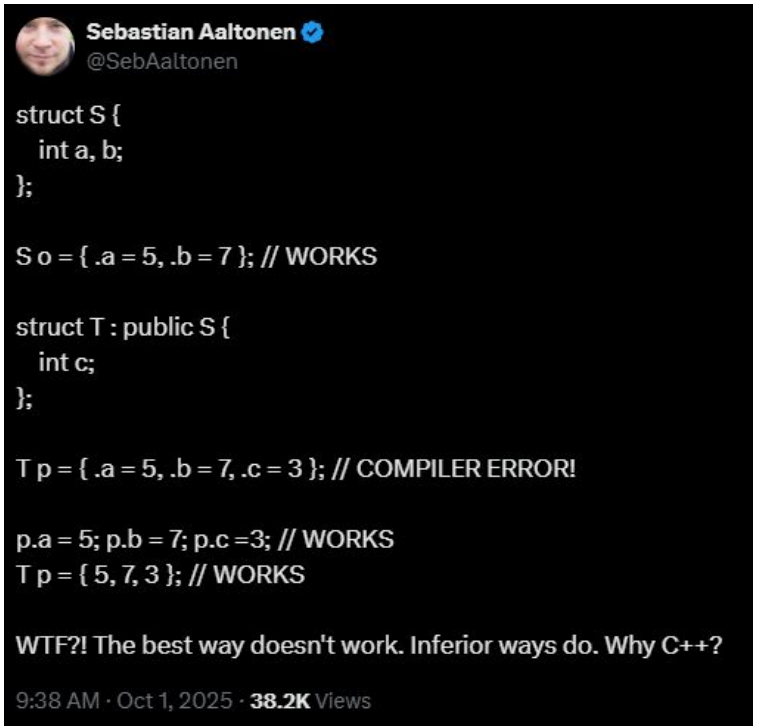
https://www.godbolt.org/z/dWxhGhrMd

VII. Inheritance

https://www.youtube.com/watch?v=B2BFbs0DJzw



Наследование в C++




```c
struct S {
    int a, b;
};

struct S o = { .a = 5, .b = 7 }; // WORKS

struct T : public struct S {
    int c;
};

struct T p = { .a = 5, .b = 7, .c = 3 };

p.a = 5; p.b = 7; p.c = 3;       // WORKS
struct T p = { 5, 7, 3 };        // WORKS
```

https://x.com/SebAaltonen/status/1973276342218268881

Наследование в C++

Наследование в C++

NeoMallchad
@Mallchad

glares.
I thought you guys didn't like inheritance

10:28 PM · Oct 1, 2025 · **29.2K** Views

Sebastian Aaltonen
@SebAaltonen

The biggest problem with inheritance are:
1. People deriving real-world concepts instead of functionality causing messy diamond inheritance issues. (you want to derive from flying instead of bird)
2. Base classes not being concrete objects. Need to use pointers instead of concrete data members. Leads to single object allocation pattern, which results in lots of locks (allocate, free) -> MT contention, fragmentation and cache trashing (pointer walks).
3. Heavy use of virtual functions trashes instruction caches and prevents compiler inlining. Also makes code hard to refactor and multithread, since you never know what data is modified by who (knowing your data writes are the key in multithreading).
4. Composition is more flexible than inheritance and you can also do external composition to have full flexibility of your data layout. For example split data by system, which is a significant improvement for code base maintainability and performance.

POD struct inheriting another POD struct is completely fine. It doesn't have any of these issues.

Наследование в C++

People deriving **real-world concepts** instead of functionality causing messy **diamond inheritance** issues (you want to derive from “flying” instead of “bird”).

Наследование в C++

People deriving real-world concepts instead of functionality causing messy diamond inheritance issues (you want to derive from “flying” instead of “bird”).

Base classes not being concrete objects. Need to **use pointers** instead of concrete data members. Leads to single object allocation pattern, which **results in lots of locks** (allocate, free) -> MT contention, **fragmentation** and **cache thrashing** (pointer walks).

Наследование в C++

People deriving real-world concepts instead of functionality causing messy diamond inheritance issues (you want to derive from “flying” instead of “bird”).

Base classes not being concrete objects. Need to use pointers instead of concrete data members. Leads to single object allocation pattern, which results in lots of locks (allocate, free) -> MT contention, fragmentation and cache thrashing (pointer walks).

Heavy use of **virtual functions** thrashes instruction caches and prevents compiler inlining. Also makes code hard to refactor and multithread, since you never know what data is modified by who (**knowing** your **data writes** is the **key in multithreading**).

Наследование в C++

People deriving real-world concepts instead of functionality causing messy diamond inheritance issues (you want to derive from “flying” instead of “bird”).

Base classes not being concrete objects. Need to use pointers instead of concrete data members. Leads to single object allocation pattern, which results in lots of locks (allocate, free) -> MT contention, fragmentation and cache thrashing (pointer walks).

Heavy use of virtual functions thrashes instruction caches and prevents compiler inlining. Also makes code hard to refactor and multithread, since you never know what data is modified by who (knowing your data writes is the key in multithreading).

Composition is more flexible than **inheritance** and you can also do **external** composition to have full flexibility of your **data layout**. For example, split data by system, which is a significant improvement for code base maintainability and performance.

Наследование против композиции

```
class stack {
    int *first;
    int *last;
    int *total;
public:
    // здесь работа с ресурсами

    void push();
    void pop();
};
```

```
class queue : public stack {
public:
    // здесь работа с ресурсами

    void push() { stack::push(); }
    void pop();
};
```

```
class deque : public queue {
public:
    // здесь работа с ресурсами

    void push_back() { stack::push(); }
    void pop_back() { stack::pop(); }
    void pop_front() { queue::pop(); }
    void push_front();
};
```

Наследование против композиции

```
class deque {
    int *first;
    int *last;
    int *total;
public:
    // здесь работа с ресурсами

    void push_front();
    void push_back();
    void pop_front();
    void pop_back();
};
```

```
class stack {
    deque data;
public:
    void push() { data.push_back(); }
    void pop() { data.pop_back(); }
};
```

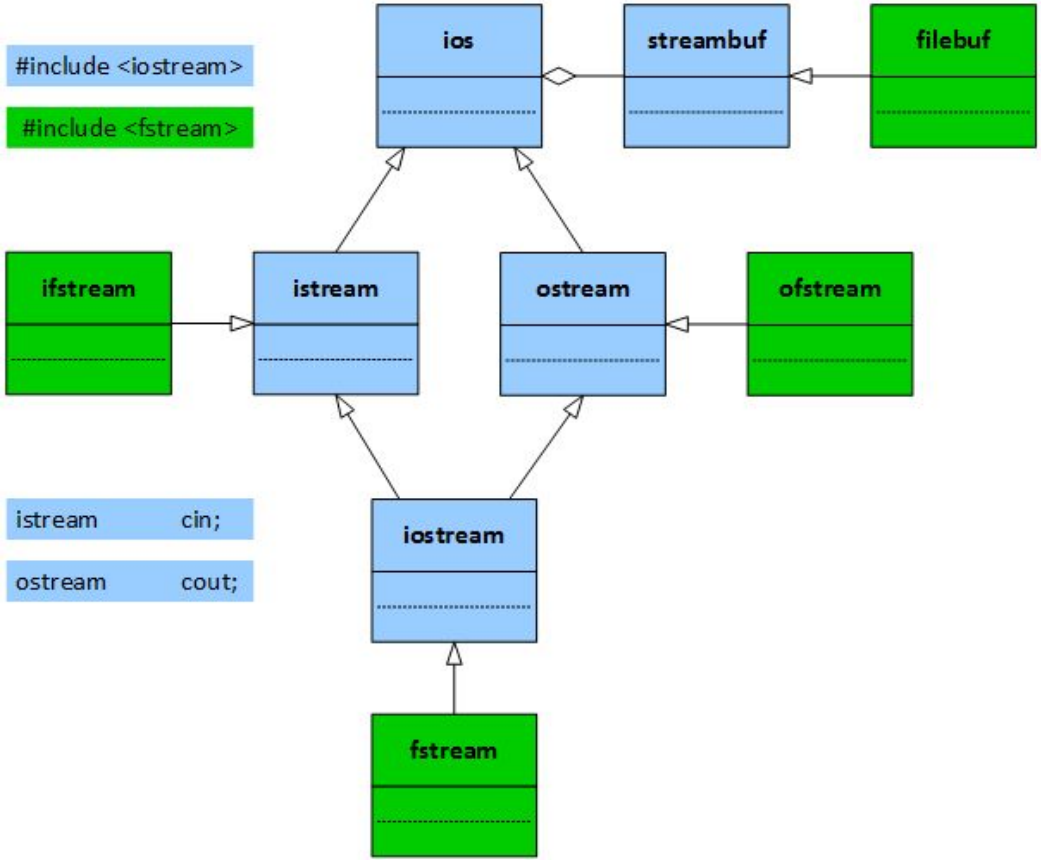
```
class queue {
    deque data;
public:
    void push() { data.push_back(); }
    void pop() { data.pop_front(); }
};
```

Как наследование используется в библиотеках

```
* Don't be clever.
* Don't be stupid.
* Naming matters.
* Generic components should be aware of move-only types.
* We are thread-compatible `[res.on.data.races]`.
* Exceptions are used for error conditions (there are some exceptions).
* Do not gratuitously overload operators.
* Classes allocating memory get an allocator (inconsistently applied).
* Containers get allocators unless they don't allocate.
* Containers use allocator through allocator traits.
* Allocators are part of type unless there is already type-erasure for other reasons.
* `const`-correctness is observed and is used as a proxy for thread-safety in the standard library.
* Class signatures want to be near minimal (the obvious counter-example is std::basic_string).
* Destructors shall not throw.
* Things should be `constexpr` where reasonable.
* Avoid inheritance and virtual functions where possible.
* Prefer function objects (i.e., deduced templates) to function pointers.
```

https://github.com/cplusplus/LEWG/blob/archive/library-design-guidelines.md

VII. Inheritance



https://icarus.cs.weber.edu/~dab/cs1410/textbook/14.Streams/introduction.html

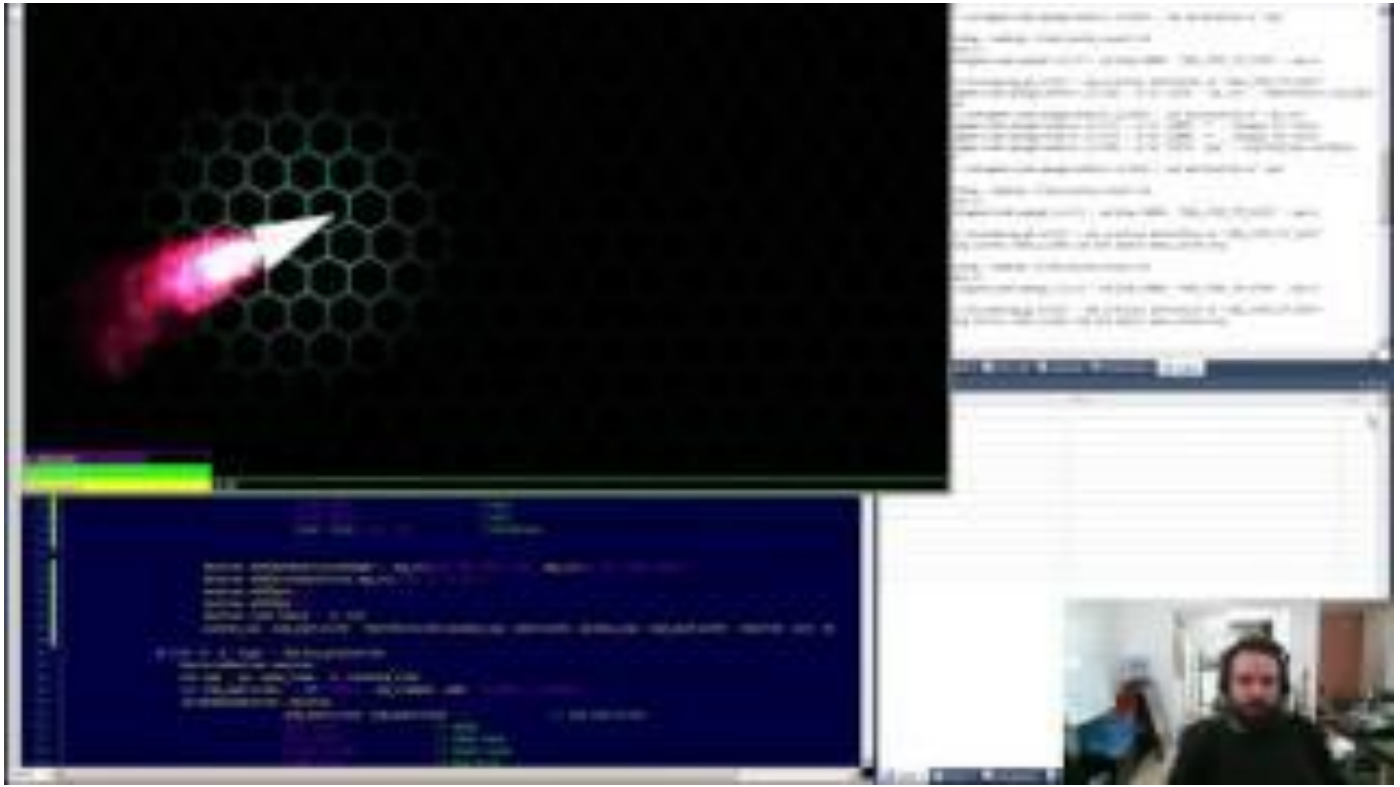
Особенности наследования

- Implementation inheritance vs. interface inheritance
  - private vs. public
- Множественное наследование
  - Комбинирование классов в один таким образом, чтобы дочерний класс описывал **объекты**, которые могут себя вести как **любой из своих базовых классов**
- Абстрактные базовые классы
  - Позволяют изменять реализацию без компилирования всей иерархии
  - Позволяют явно выделить интерфейс в отдельный класс
- Inheritance vs. containment
  - Явное включение базового класса в большинстве* случаев эквивалентно

Inheritance is the base class of evil

https://www.youtube.com/watch?v=bIhUE5uUFOA

https://youtu.be/q4nUK0EBzmI?si=gctz7CzkZxtcdU4l&t=12079



Что такое OOP?

- инкапсуляция (абстракция состояния и избавление от манипуляции состоянием в коде)
- наследование? (иерархическая система типов согласно классификации)
- ~~полиморфизм~~
- делегация? (делегирование сообщений другим объектам, **forwarding**)
- динамическое связывание? (**late binding**, dynamic dispatch)

Что такое OOP?

- инкапсуляция (абстракция состояния и избавление от манипуляции состоянием в коде, **public** и **private**)
- ~~наследование~~
- ~~полиморфизм~~
- делегация? (**forwarding**, **move** semantics)
- динамическое связывание (**виртуальные функции**)
- абстракция (скрытая реализация, открытый **интерфейс**)
- подмена типов? (subtyping)