Объектно-ориентированное программирование

Object-oriented programming

VII. Наследование

Inheritance

"I strongly felt then, as I still do, that there is no one right way of writing every program, and a language designer has no business trying to *force* programmers to use a particular **style**. The language designer does, on the other hand, have an obligation to encourage and **support a variety of styles and practices** that have proven effective and to provide language features and tools to help programmers avoid the well-known traps and pitfalls."

B. Stroustrup

"A History of C++: 1979-1991"

Влияние **Simula** на C++

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

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

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)

“C++ was designed to provide Simula’s **facilities for program organization** together with C’s **efficiency** and **flexibility** for systems programming.”

B. Stroustrup

Базовые элементы реализации C++

- Распределенная операционная система на базе UNIX
- Иерархическая система классов для C
- Библиотека для создания сопрограмм
- Zero-cost abstractions

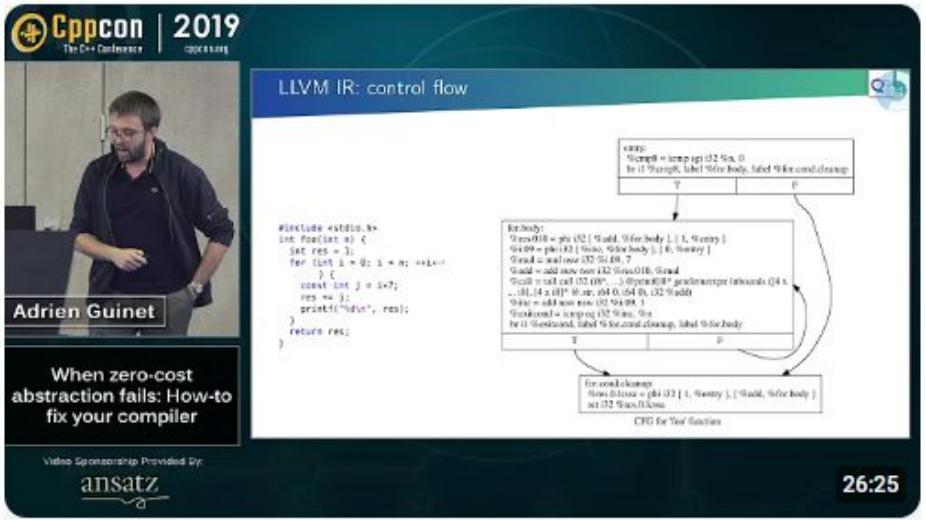
“What you don’t use, you don’t pay for. What you do use, you couldn’t hand code any better.”

B. Stroustrup

“Zero-cost” abstractions

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

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



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.

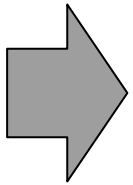
Anders Lindqvist @anders_breakin · Sep 13
On my 13 year old home computer I mostly avoid them due to DEBUG performance. I often start out using them and then removing them over time. I'm waiting for a new computer, maybe my feelings will change :)

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

Object-oriented programming

```
#include <vector>
```

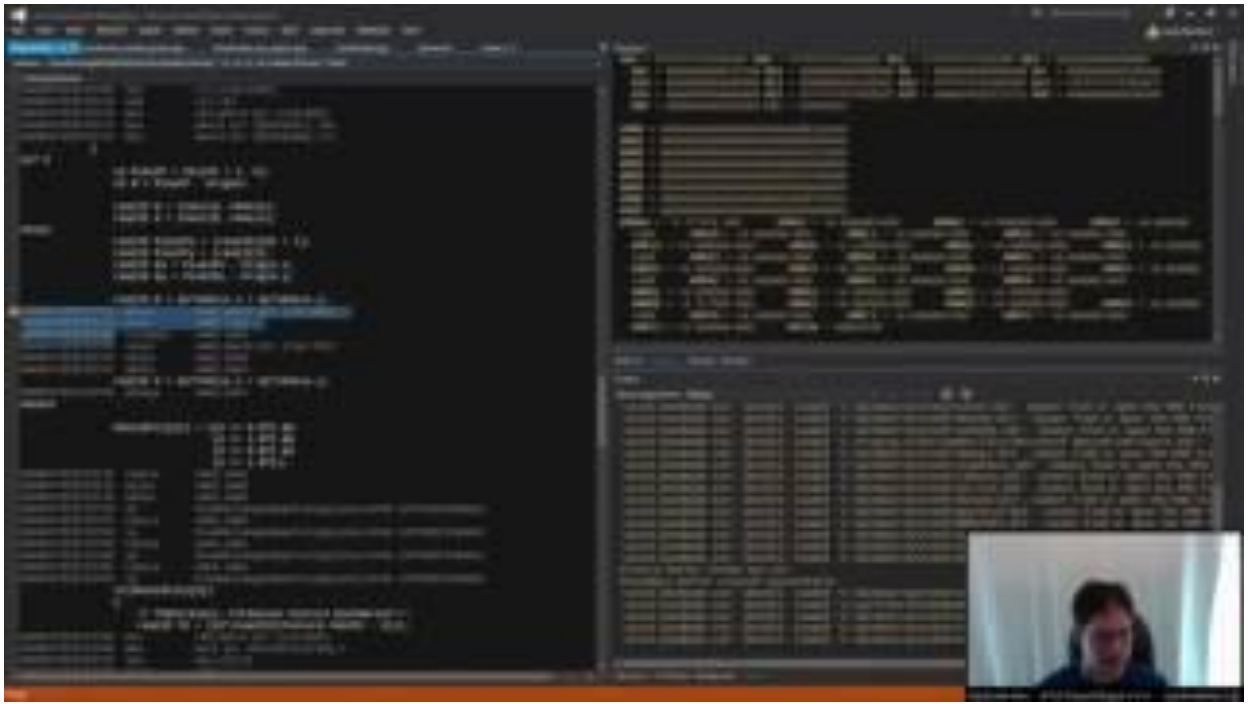


x86-64 clang 18.1.0 | -E -std=c++23

A | 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.youtube.com/watch?v=B2BFbs0DJzw



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();
};
```

Основные инструменты C++

1. Виртуальные функции (наследование).
2. Перегрузка функций и операторов.
3. Ссылочные переменные.
4. Константные переменные - **const**.
5. Ручное управление динамической памятью.
6. Строгая система статической проверки типов данных.

Зачем нужны виртуальные 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++

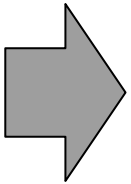
```
class shape {
    point center;
    color col;
    // дополнительное поле не нужно
public:
    virtual void draw();
    // реализация открытых методов здесь
};

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();
}
```

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

Наследование в C++

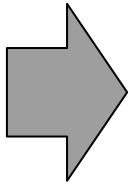
```
class A {
    char c{0}; // 1 байт
};

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
```

Наследование в C++ с абстрактными классами

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

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

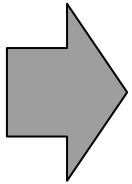


```
Vtable for A
A::_ZTV1A: 3 entries
0     (int (*)(...))0
8     (int (*)(...))(& _ZTI1A)
16    (int (*)(...))A::f

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)
```

Множественное наследование в C++

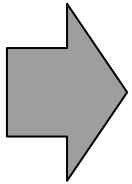
```
class A {
    char c{0};
};

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
```

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

Наследование против композиции

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

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

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

    void push_back() { stack::push_back(); }
    void pop_front();
};
```

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

    void push_back() { stack::push_back(); }
    void pop_back() { stack::pop_back(); }
    void pop_front() { queue::pop_front(); }
    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_back() { data.push_back(); }
    void pop_back() { data.pop_back(); }
};
```

```
class queue {
    deque data;
public:
    void push_front() { data.push_front(); }
    void pop_front() { data.pop_front(); }
};
```

"*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

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