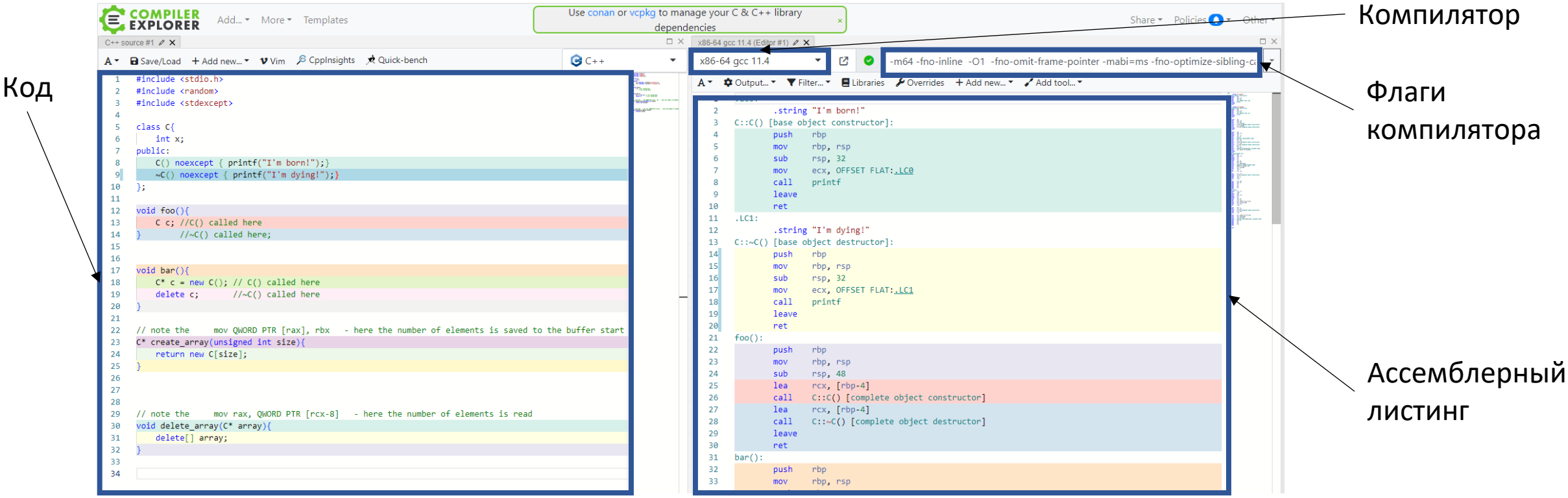
Низкоуровневое программирование

Лекция 4

Основы декомпиляции программ на C/C++

Compiler Explorer

Сайт https://godbolt.org позволяет посмотреть результат компиляции кода на C/C++ (и некоторых других языков) одним из компиляторов.



Правило as-if

Компиляторы C/C++ имеют право проводить любые преобразования с кодом, которые сохраняют итоговый порядок возникновения наблюдаемых эффектов (side effects) выполнения программы, к которым относятся:

- операции ввода-вывода,
- чтение/запись переменных, помеченных как *volatile.*

Эти операции гарантированно производятся в том же порядке, что и в программе без оптимизаций (as if the program is executed as written).

Типовые оптимизации:

- переупорядочивание кода, в т.ч удаление недостижимого кода и встраивание функций (inlining);
- предварительные вычисления (`a = 1; b = a++;  => a = 2; b = 1;`).
- замена инструкций на эквивалентные им последовательности инструкций;

Включение/отключение отдельных оптимизаций контролируется флагами компилятора.

Границы оптимизаций

Поскольку каждый .cpp-файл компилируется отдельно от остальных, информация о коде в других файлах недоступна. Как следствие, компилятор не может делать никаких предположений о работе этих функций => не имеет права оптимизировать их вызовы.

Как следствие, порядок вызова внешних функций (т.е. функций, определение которых отсутствует в текущем файле) производится в том же порядке, что и в исходном файле*.

Отдельным случаем является т.н. link-time optimization, при включении которой несколько изменяется процесс работы компилятора и итоговая оптимизация производится над всем доступным кодом программы.

*за исключением некоторых функций из стандартной библиотеки, реализуемых на уровне компилятора

О volatile

Ключевое слово *volatile* используется при объявлении переменных, которые могут изменить свое значение без видимых (с точки зрения отдельного потока) причин.

По прямому назначению такие переменные используются в системном программировании для взаимодействия с различными устройствами, и, зачастую некорректно – в многопоточном программировании для объявления некоторых общих переменных.

Поскольку значение в переменной может быть изменено внезапно, компилятор не может проводить оптимизации над этой переменной. Более того, он сохранит порядок операций чтения/записи всех volatile-переменных (но переупорядочить чтение/запись обычной и volatile-переменной компилятор может).

За пределами системного программирования volatile-переменные используются для подавления оптимизации.

Замечания о локальных переменных

- Если компилятор может поместить переменную в регистр – он поместит ее в регистр.
- Если компилятор обнаружит, что переменную можно убрать – он ее уберет.
- Одно и то же место на стеке на разных этапах работы функции может быть занято разными переменными. К примеру, компилятор может сбросить переменную на стек для того, чтобы взять ее адрес, а когда переменная станет не нужна – на это же место сбросить другую переменную. Исключение – переменные, помеченные как volatile (обязаны иметь уникальный адрес).
- *Примечание о константах:* дизассемблер не всегда может понять, как следует декодировать тот или иной набор байтов. Как следствие, вещественные константы могут быть декодированы, как целые числа, и положительные целые числа – как отрицательные (или наоборот).
- Тип переменной можно определить по инструкциям, которые с ней работают (усложняется эквивалентными заменами).

Переупорядочивание кода и встраивание функций

Компилятор имеет право переупорядочить расположение кода программы, например, расположить метку *else* до *if* или разбить тело функции на несколько фрагментов, переупорядочить их и связать инструкциями перехода.

Целью подобных решений является повышение локальности кода – если код сконцентрирован в одной области, ЦП сможет выполнять его быстрее по причине кэширования (см. лекцию о микроархитектуре). Поэтому компилятор стремится расположить код, вероятность выполнения которого более вероятна, близко.

Частным случаем переупорядочивания является встраивание функций. Компилятор может проводить встраивание любых функций, определение которых ему доступно на этапе компиляции, *если он сочтет это выгодным*.

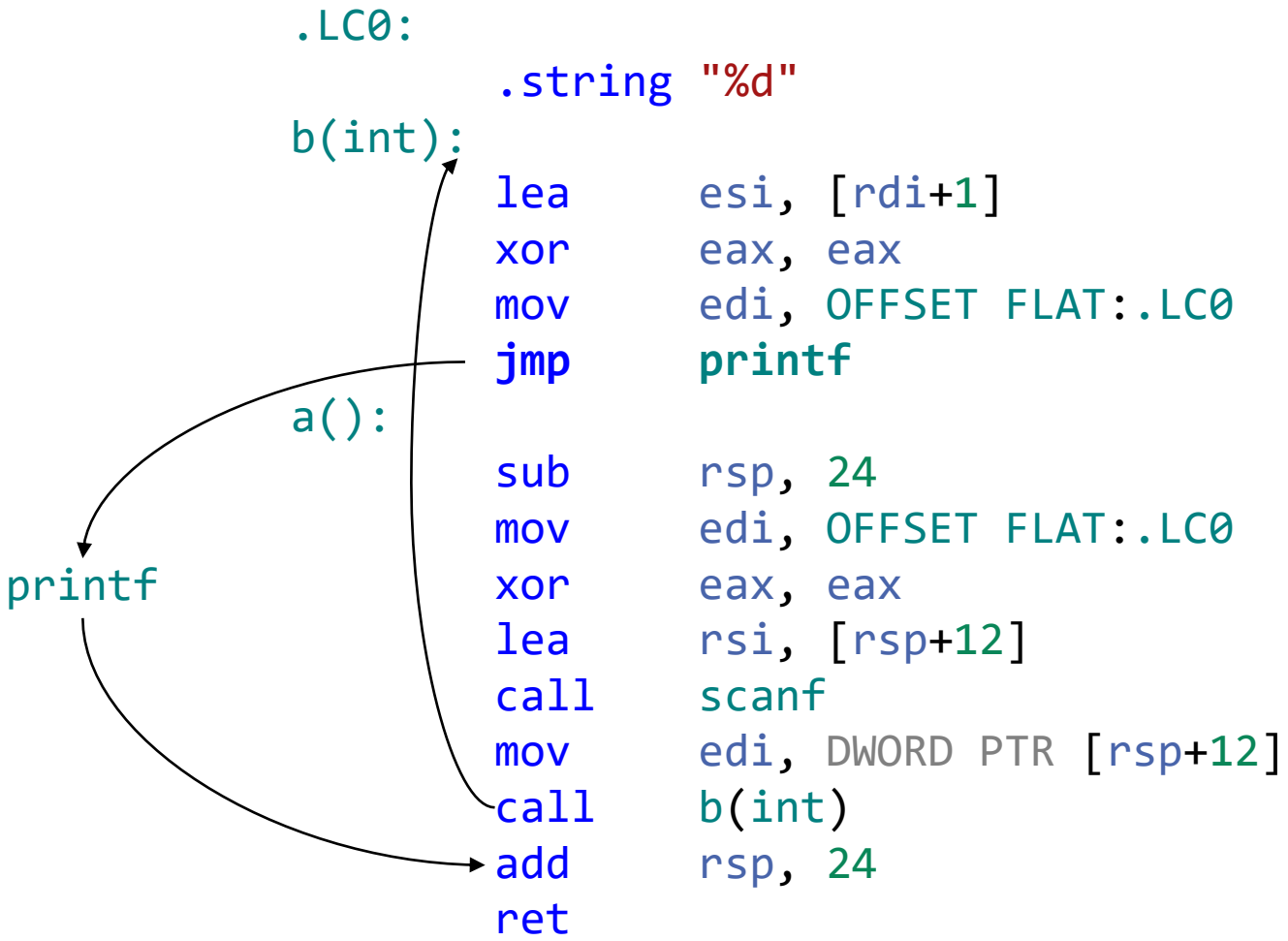
При встраивании компилятор может анализировать межпроцедурные зависимости и адаптировать код исходной функции к конкретному вызову.

При этом, если вызываемая функция не помечена, как static, компилятор все равно обязан сгенерировать код функции.

Простые оптимизации: хвостовой вызов

Если функция оканчивается вызовом другой функции, то компилятор может заменить call на jmp. При этом на стеке остается исходный адрес возврата, который используется вызванной функцией.

```cpp
#include <cstdio>

void b(int x){
    printf("%d", x+1);
}

void a(){
    int x;
    scanf("%d", &x);
    b(x);
}
```



https://godbolt.org/z/sKTdaW9cf

Простые оптимизации: хвостовая рекурсия

Отдельным случаем оптимизации хвостовых вызовов является хвостовая рекурсия.

Если функция оканчивается рекурсивным вызовом, и при этом возвращаемый результат не модифицируется, то цепочка рекурсивных вызовов может быть оптимизирована компилятором в цикл.

При этом, в отличие от полноценной рекурсии, на стеке не создается новый кадр стека, а используется текущий.

```
const int*  search(const int* start, const int* end,
                                          int value){
    while(start != end){
        if(*start == value)
            return start;
        ++start;
    }
    return nullptr;
}
```

```
const int*  search_rec(const int* start, const int* end,
                                              int value){
    if(start == end)
        return nullptr;
    if(*start == value)
        return start;
    return search(++start, end, value);
}
```

https://godbolt.org/z/59h3TMfzT

Простые оптимизации: таблица переходов

Часто в программах встречаются условные операторы switch или цепочки if-else.

Вместо того, чтобы честно выполнять N сравнений, компилятор может создать таблицу переходов (**jump table**) в которой находятся адреса кода соответствующего case или тела if, и сгенерировать код, который вычисляет индекс в таблице переходов,

https://godbolt.org/z/hMf9ozPdE

```
int foo(int c, int x, int y){
    switch(c){
        case 1:
            x+=y; break;
        case 2:
            x+=y; break;
        case 3: x-=y
    }
    return x;
}
```

```
int foo(int c, int x, int y){
    int* table[] = {&&end, &&case1, &&case2, &&case3};
    if(c < 4) goto *table[c];
    goto end;

    case1:  x+=y; goto end;
    case2:  x+=y; goto end;
    case3:  x-=y;
    end:    return x;
}
```

Простые оптимизации: развертка циклов

На уровне ассемблера в начале/конце тела цикла for находятся инструкции уменьшения счетчика, сравнения и условного перехода.

Пусть тело цикла – 1 инструкция. Тогда в теле цикла полезная нагрузка приходится лишь на 1/4 инструкций, а 3/4 инструкций нужны только для организации цикла.

Если цикл имеет фиксированную длину N, то можно повторить тело цикла N раз (развернуть цикл) и избавиться от сравнений вовсе.

```
int pow5(int x){
    int r = 1;
    for(int i=0; i<5;++i)
        r*=x;
    return r;
}
```



```
int pow5(int x){
    int r = x;
    r*=x;
    r*=x;
    r*=x;
    return r*x;

}
```

Простые оптимизации: развертка циклов

Если количество итераций цикла неизвестно заранее, но известно в момент входа в цикл, то можно частично развернуть цикл, разделив его на 2 части.

Первая часть обрабатывает «голову» цикла, если общее число итераций не кратно K.

Вторая часть выполняет итерации группами по K (т.е. содержит тело цикла, повторенное K раз.)

```
int pow(int x, int c){
    int r = 1;
    for(c; c >0; c--)
        r*=x;
    return r;
}
```

```
int pow(int x, int c){
    int r = 1;
    switch(c % 5){
        case 4: r*=x;
        case 3: r*=x;
        case 2: r*=x;
        case 1: r*=x;
    }
    for(; c > 4; c-=5)
        {r*=x;r*=x;r*=x;r*=x;r*=x;}

    return r;
}
```

https://godbolt.org/z/nT97bKbz8

Duff’s Device

Особенность реализации switch в языке C позволяет не делить цикл на 2 части для обработки основной части и «хвоста». Можно вместо этого использовать химерную конструкцию под названием Duff’s Device.

Устройство Даффа в начале выполнения осуществляет *прыжок в тело цикла*, пропуская часть лишних итераций. После этого цикл продолжается, как положено.

```
int pow_duff(int x, int c){
    int r = 1;
    if (c < 1) return r;

    switch(c%5){
      do{
        case 0: r*=x;
        case 4: r*=x;
        case 3: r*=x;
        case 2: r*=x;
        case 1: r*=x;
      }while((c-=5) > 0 );
    }

    return r;
}
```

https://godbolt.org/z/nT97bKbz8

Эквивалентные замены

Каждая инструкция выполняется целое число тактов, при этом это число отличается для разных инструкций. Как следствие, имеет смысл заменять «дорогие» инструкции на «дешевые» (strength reduction).

Типовыми примерами являются:

- замена умножения/деления на степень 2 на сдвиги;
- замена умножений на сдвиги и сложения (в т.ч. с помощью `LEA`);
- замена вещественных операций целочисленными;
- замена целочисленного деления (*самая дорогая целочисленная операция!*).

Дороговизна/дешевизна инструкций определяется точнее, если указать целевую микроархитектуру при компиляции (флаги -march/-mtune на GCC/clang).

См. также: таблицы времени выполнения инструкций (англ.)

Неопределенное поведение

Неопределенное поведение (Undefined behaviour, UB) – поведение программы, которое возникает при нарушении требований стандарта C++ и ведет к непредсказуемому результату работы программы.

Обычно проблемы «неопределенного поведения» возникают при сборке с оптимизациями, т.к. компилятор имеет право считать, что *ситуации неопределенного поведения не происходят*. Как следствие, компилятор может, в результате оптимизаций, изменить или убрать важную с точки зрения логики работы часть кода, приводящую к неопределенному поведению (UB не происходит -> код приводящий к UB никогда не выполняется –> код можно убрать).

К ситуациям неопределенного поведения стандарт C/C++ относит чаще всего или явные ошибки программиста, или ситуации, которые на разных архитектурах обрабатываются по-разному, и потому не могут быть сведены к 1 стандарту.

Современные компиляторы детектируют большую часть источников неопределенного поведения => **предупреждения компилятора надо читать и принимать во внимание**.

В языке ассемблера нет неопределенного поведения – поведение программы жестко задано поведением инструкций.

Примеры неопределенного поведения

- знаковое переполнение;
- вечный цикл без *наблюдаемых извне* действий;
- использование неинициализированных переменных;
- разыменование нулевого указателя;
- доступ к элементу за границами массива;
- доступ к переменной через указатель другого типа;
- и т.д.

О расположении локальных переменных и полей структур

- Отличить группу переменных от структуры/объекта можно только по семантике - структура копируется целиком, но используется по частям. Например, если структура передавалась в регистре, то потом ее поля будут извлекаться битовыми операциями.
- Компилятор имеет право переупорядочивать порядок локальных переменных, но не порядок полей структуры – они идут в том же порядке.
- Локальные переменные, если они располагаются не в регистре, могут располагаться непоследовательно в памяти – между ними могут быть небольшие «пустоты» в несколько байт.
- Теоретически размер структуры должен быть равен сумме размеров полей, на практике он может отличаться в большую сторону.

```
struct A{
    char x;
    long long y;
};
```

```
std::cout << sizeof(A)
Output: 16 (???)
```

Последние 2 явления можно объяснить требованиями на выравнивание данных.

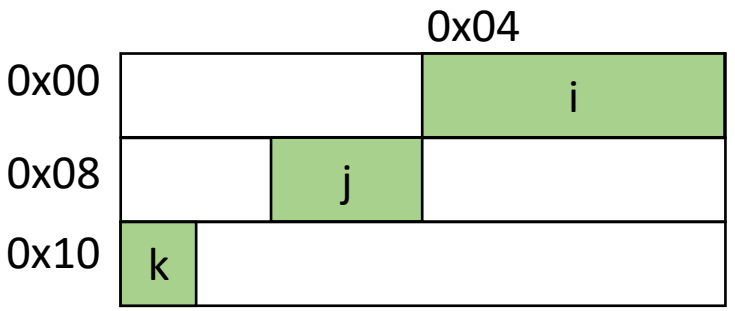
Выравнивание адреса

Говорят, что данные **выровнены** по границе в N байт, если адрес данных делится без остатка на N.

Обычно данные выравниваются по границе 2/4/8/16 байт.

На рисунке:

i выровнена по границе в 2 и 4 байта,
j выровнена по границе 2 байта,
k выровнена по границе 2,4, 8 и 16 байт.



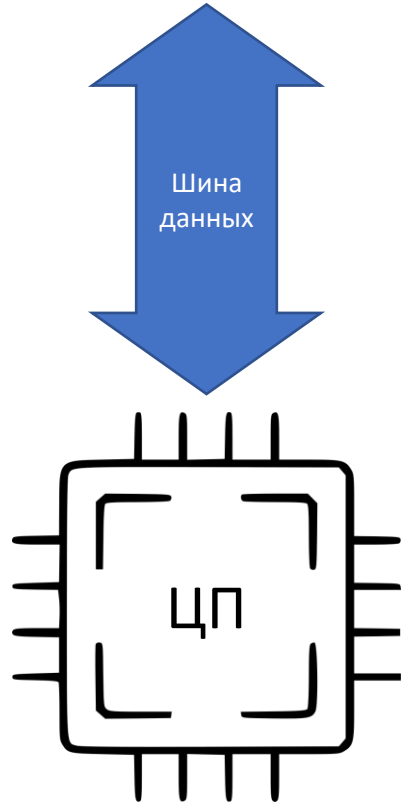
Аппаратные особенности ОЗУ

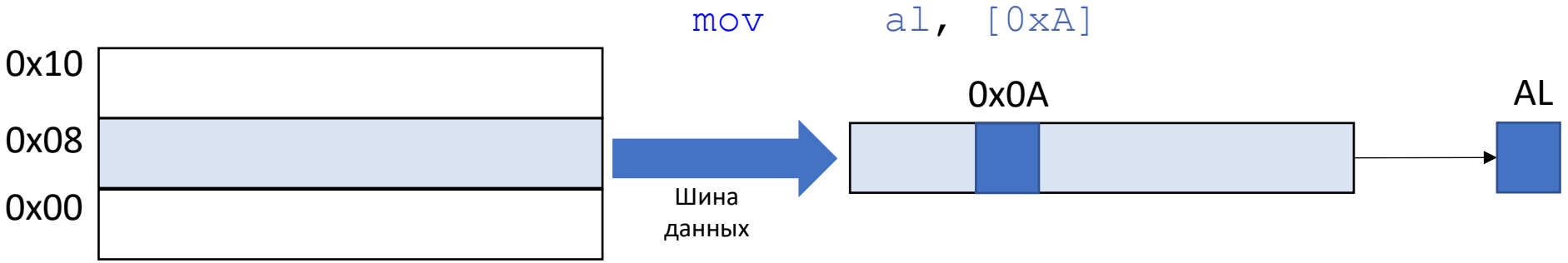
Байт – минимальная адресуемая единица информации.

Формально, каждый байт ОЗУ имеет свой адрес. Это верно только на программном уровне.

На аппаратном уровне процессор соединен с ОЗУ шиной данных. В современных системах x86-64 ширина шины данных равна 64 битам1. (32 битам для DDR5).

Оперативная память делится на блоки, каждый из которых равен ширине шины данных. Чтение данных выполняется блоками (лишние данные не используются). Операции записи тоже оперируют блоками, но могут выполнять частичное изменение блока.





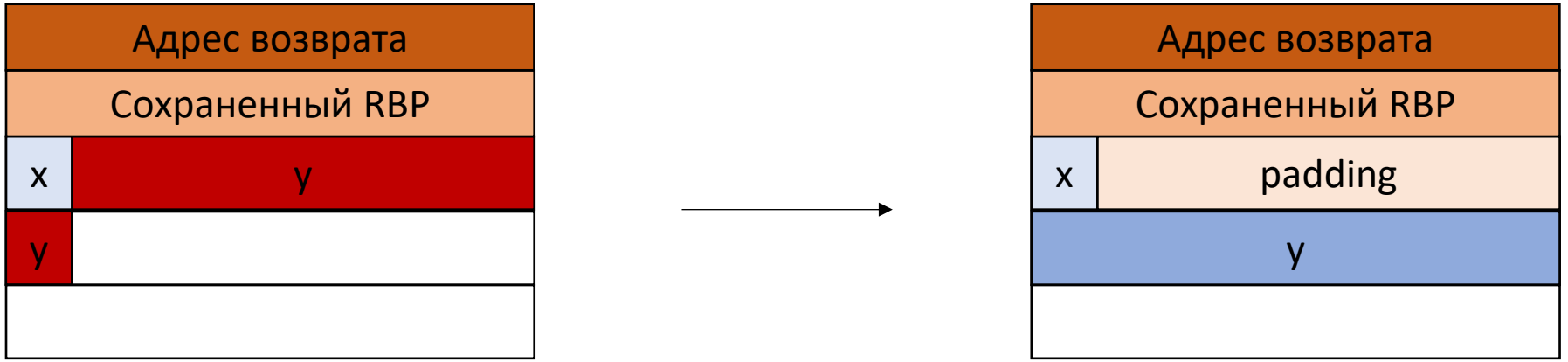
Выравнивание переменных компилятором

Пусть дан код справа и компилятор решает расположить обе переменные на стеке.

Если компилятор расположит переменные рядом, то чтение переменной Y потребует 2-х аппаратных чтений.

Если компилятор выровняет переменную, то она будет прочитана за 1 чтение, но между переменными возникнет неиспользуемое пространство.

```
void main(){
    char x;
    long long y;
    /*…*/
};
```



Выравнивание

Переменные и поля базовых типов выравниваются по своему размеру.

Структуры выравниваются так, чтобы доступ к ее полям был выровненным (выравнивание структуры равно максимальному выравниванию среди ее полей).

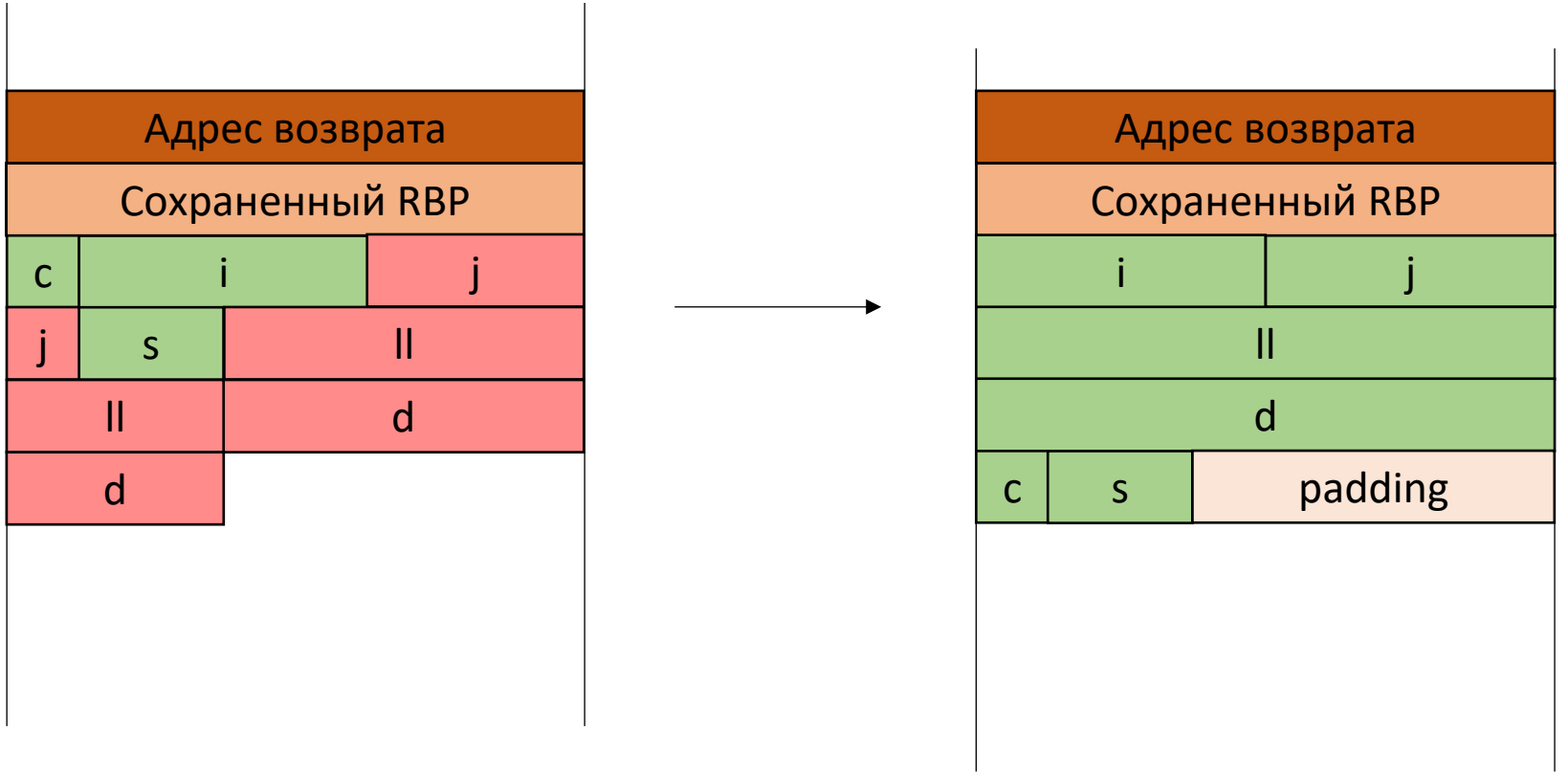
Чтобы узнать выравнивание типа или поля, можно использовать ключевое слово **alignof**.

Существует специальное ключевое слово **alignas**, позволяющее вручную *увеличить* выравнивание поля или структуры в целом.

```
struct A{
    char x;
    long long y;
};

std::cout << sizeof(A)
Output: 16


std::cout << alignof(A)
Output: 8


std::cout << alignof(A::x)
Output: 1

std::cout << alignof(char)
Output: 1

std::cout << alignof(int)
Output: 4
```

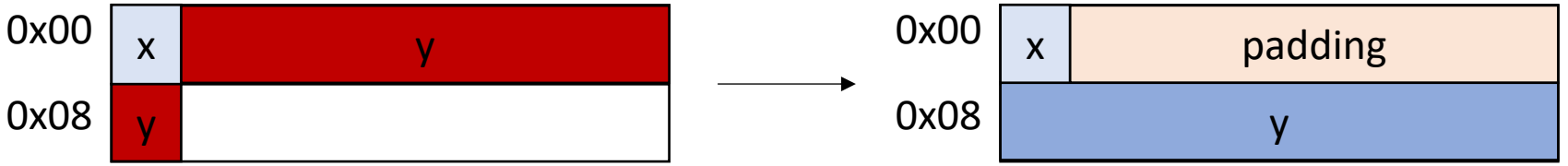
Переупорядочивание переменных

```
void foo(){
    char c;
    int i, j;
    short s;
    long long ll;
    double d;
    …
}
```

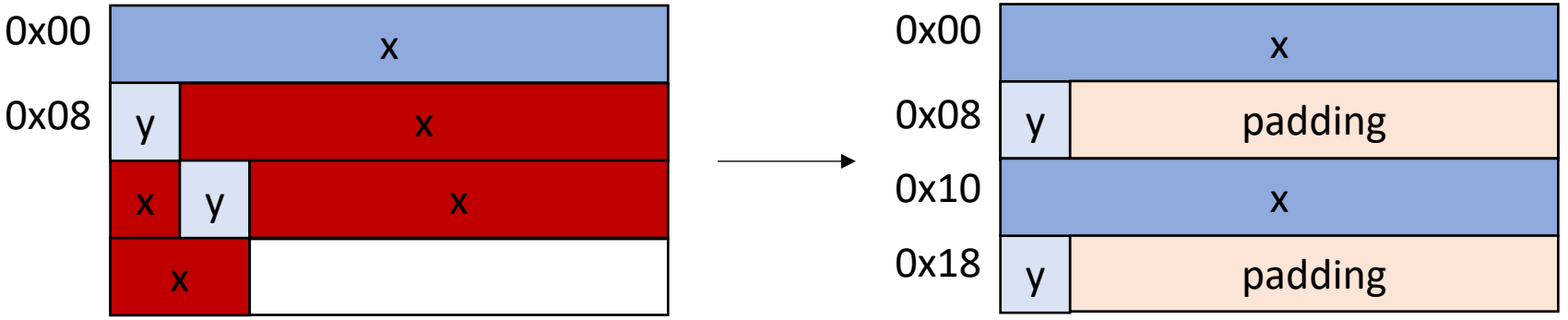


Выравнивание полей

```
struct A{
    char x;
    char padding[7];
    long long y;
};
```




```
struct B{
    long long x;
    char y;
    char padding[7];
};

B array[2];
```



Классы и объекты C++

В языке C++ помимо структур возникли понятия классов и объектов. Тем не менее, объявление класса и объявление структуры отличается только уровнем доступа по умолчанию => *расположение объектов и структур в памяти не отличается*.

Более того, *модификаторы доступа действуют только на уровне языка* => на уровне ассемблера вполне можно изменить приватное поле или вызвать приватный метод.

Порядок полей объекта *одного уровня видимости* совпадает с порядком объявления. Порядок полей разных уровней видимости не определен.

Конструкторы и деструкторы

Конструкторы и деструкторы являются методами класса, следовательно, их механизм вызова производится согласно соответствующему соглашению о вызовах.

Компилятор автоматически генерирует вызов конструктора при объявлении объекта. Обратите внимание, что *конструктор вызывается после выделения памяти для объект* (не важно, на стеке или в куче).

Аналогичная ситуация происходит с деструктором – компилятор вызывает его автоматически перед освобождением памяти, выделенной для объекта.

Если выделение памяти производится на стеке или в секциях данных, компилятор самостоятельно учитывает выравнивание.

Если выделение памяти производится в куче, обычно дополнительных действий не требуется, т.к. malloc() и new() возвращают адрес, выровненный по границе 16 байт. Если для структуры требуется большее выравнивание, то за него отвечает программист, а не компилятор.

Конструкторы и деструкторы

https://godbolt.org/z/6cWroda4n

```
class C{
    int x;
public:
    C(){ printf("I'm born!");}
    ~C(){ printf("I'm dying!");}
};

void foo(){
    C c; // C() called here
}        //~C() called here;
```

```
foo():
        push    rbp
        mov     rbp, rsp
        sub     rsp, 48
        lea     rcx, [rbp-4]
        call    C::C()
        lea     rcx, [rbp-4]
        call    C::~C()
        leave
        ret
```

Конструкторы и деструкторы

https://godbolt.org/z/6cWroda4n

```
class C{
    int x;
public:
    C(){ printf("I'm born!");}
    ~C(){ printf("I'm dying!");}
};

void bar(){
    C* c = new C(); // C() called here
    delete c;       //~C() called here
}
```

```
bar():
        push    rbp
        mov     rbp, rsp
        push    rbx
        sub     rsp, 32
        mov     ecx, 4
        call    operator new(unsigned long)
        mov     rbx, rax
        mov     rcx, rax
        call    C::C()
        mov     rcx, rbx
        call    C::~C()
        mov     edx, 4
        mov     rcx, rbx
        call    operator delete(void*, unsigned long)
        add     rsp, 32
        pop     rbx
        pop     rbp
        ret
```

Операторы new[] и delete[]

https://godbolt.org/z/6cWroda4n

```
C* create_array(unsigned int size)
{
    return new C[size];
}
```

Оператор new[] сводится к выделению буфера достаточного объема, *записи количества объектов в начало буфера* и вызову конструктора для каждого из объектов массива.

При этом адрес, возвращаемый new[], подбирается так, что значение (адрес + 8) выровнено по 16 байтам -> область с объектами выровнена по 16 байтам.

Выделение буфера

Запись количества элементов массива

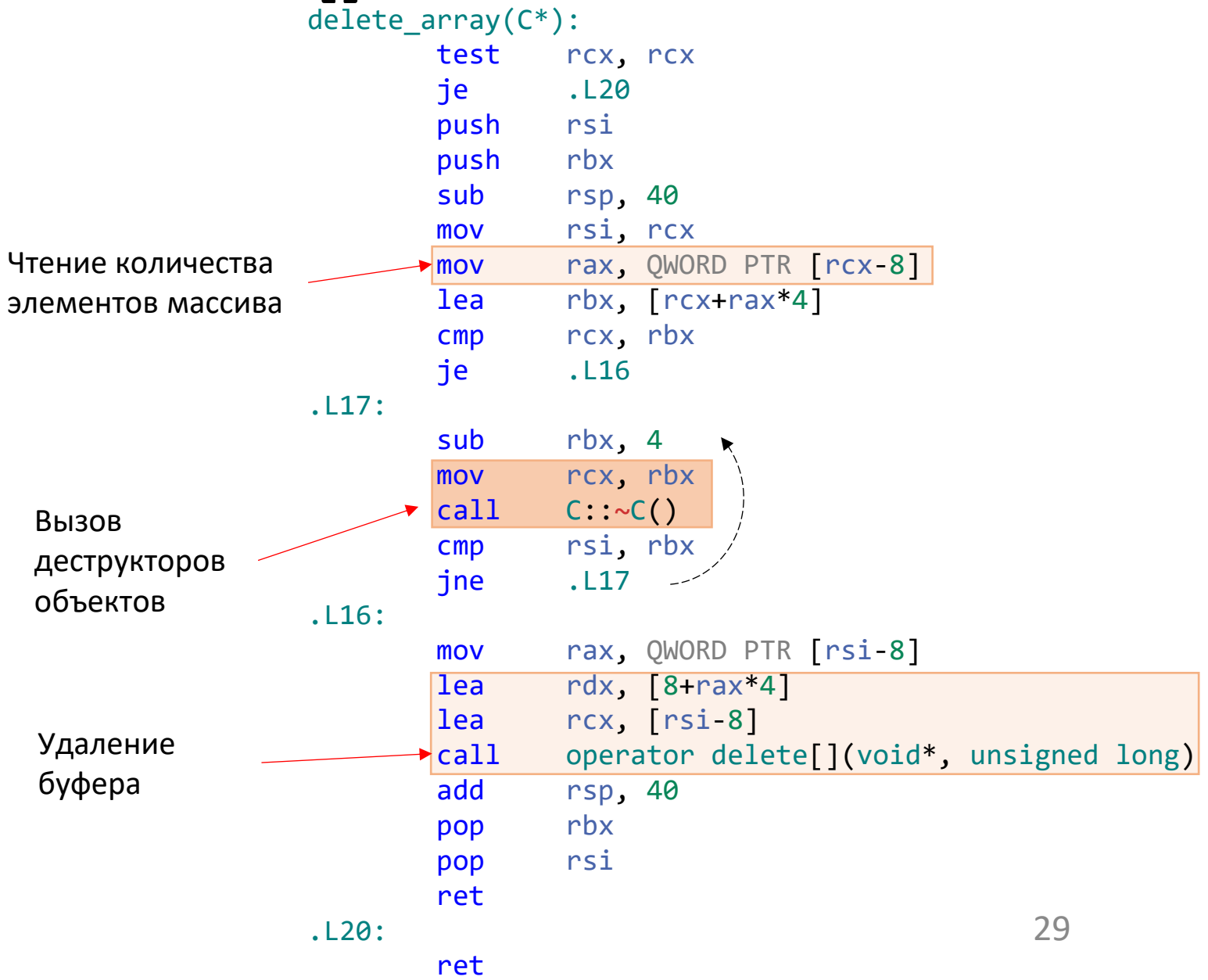
Вызов конструкторов объекта

```
create_array(unsigned int):  ;size в RCX
        push    rdi
        push    rsi
        push    rbx
        sub     rsp, 32 ; PROLOGUE END
        mov     ebx, ecx
        lea     rcx, [8+rbx*4] ; общий размер в байтах
        call    operator new[](unsigned long)
        mov     QWORD PTR [rax], rbx
        lea     rdi, [rax+8]
        sub     rbx, 1
        js      .L9
        mov     rsi, rdi
.L11:   mov     rcx, rsi
        call    C::C()
        add     rsi, 4
        sub     rbx, 1
        jns     .L11
.L9:    mov     rax, rdi
        add     rsp, 32 ; EPILOGUE START
        pop     rbx
        pop     rsi
        pop     rdi
        ret
```

Операторы new[] и delete[]

```
void delete_array(C* array){
    delete[] array;
}
```

Оператор delete[] функционирует обратно оператору new[] (считывает количество элементов – вызывает деструкторы – удаляет буфер).



Наследование

В результате наследования один класс может расширять поведение другого класса.

С точки расположения в памяти *объект класса-предка является частью объекта класса-потомка* (“матрешка”).

В случае создания объекта класса-потомка конструкторы всех его классов-предков вызываются по цепочке в порядке наследования.

Вызов деструкторов происходит в порядке, обратном порядку наследования.

P.S. Множественное или виртуальное наследование существенно усложняет положение дел, но здесь рассматриваться не будет

Наследование

```
class A{
public:
    int x=0;
    A(){printf("A");};
    ~A(){printf("~A");};
};

class B: public A{
public:
    int y=1;
    B() {printf("B");}
    ~B() {printf("~B");}
};
```

https://godbolt.org/z/dzKrz5rnv

Виртуальные методы

Виртуальные методы необходимы для реализации полиморфизма.

При вызове обычного метода компилятор вызывает метод *декларируемого* типа объекта.

В ходе вызова виртуального метода компилятор вызывает метод, принадлежащий *реальному* типу объекта.

```
class A{
public:
    int x=0;
    void non_virt(){puts("A::non_virt");}
    virtual void virt(){puts("A::virt");}
};

class B:public A{
public:
    int y=1;
    void non_virt(){puts("B::non_virt");}
    void virt() override {puts(“B::virt”);}
};

void bar(A& a){
    a.non_virt();
    a.virt();
}
```

Виртуальные методы

Если в классе определен хотя бы 1 виртуальный метод, компилятор создает для класса **таблицу виртуальных методов** (vtable).

В данной таблице содержатся адреса реализаций методов для конкретного класса.

Помимо непосредственно адресов методов, в таблице содержатся служебная информация, обычно используемая при множественном наследовании.

Доступ к таблице виртуальных методов осуществляется через скрытое приватное поле **_vptr**, содержащее указатель на таблицу виртуальных методов.

```
class A{
private:
    void* _vptr=(vtable+16);
public:
    int x=0;
    void non_virt(){puts("A::non_virt");}
    virtual void virt(){puts("A::virt");}
};



vtable for A:
        qword   0
        qword   typeinfo for A
        qword   A::virt()
```

Виртуальные методы

https://godbolt.org/z/Er77xqEWj

```
void bar(A& a){
    a.non_virt();
    a.virt();
}
```

```
A a;
bar(a);
```

```
bar(A&):
        push    rbp
        mov     rbp, rsp
        push    rbx
        sub     rsp, 40
        mov     rbx, rcx
        call    A::non_virt()
        mov     rax, [rbx]
        mov     rcx, rbx
        call    [rax]
        mov     rbx, [rbp-8]
        leave
        ret
```

```
vtable for A:
        qword   0
        qword   typeinfo for A
        qword   A::virt()
```

```
A::virt():
        push    rbp
        …
```

Виртуальные методы

https://godbolt.org/z/Er77xqEWj

```
void bar(A& a){
    a.non_virt();
    a.virt();
}
```

```
B b;
bar(b);
```

```
bar(A&):
        push    rbp
        mov     rbp, rsp
        push    rbx
        sub     rsp, 40
        mov     rbx, rcx
        call    A::non_virt()
        mov     rax, [rbx]
        mov     rcx, rbx
        call    [rax]
        mov     rbx, [rbp-8]
        leave
        ret
```

```
vtable for B:
        qword   0
        qword   typeinfo for B
        qword   B::virt()
```

```
B::virt():
        push    rbp
        …
```

Исключения в C++

Для обработки ошибок в языке C++ был введен механизм исключений.

Исключительная ситуация – ситуация, при которой дальнейшее выполнение программы невозможно.

Исключение – объект, содержащий информацию об исключительной ситуации.

Если исключение было брошено, и в текущей функции для него определен обработчик, то на стеке уничтожаются только переменные, принадлежащие текущему блоку кода.

Если в текущей функции нет подходящего обработчика, то начинается процесс раскрутки стека.

До введения исключений единственным способом узнать об ошибке было сочетание специальных возвращаемых значений и системного макроса errno.

```
void foo(){
    S c{"C"};
    if(rand())
        throw std::logic_error("");
}

void bar(){
    S a{"A"};
    try{
        S b{“B"};
        foo();
    }catch(std::logic_error& e){
        printf("Caught!\n");
    }
}
```

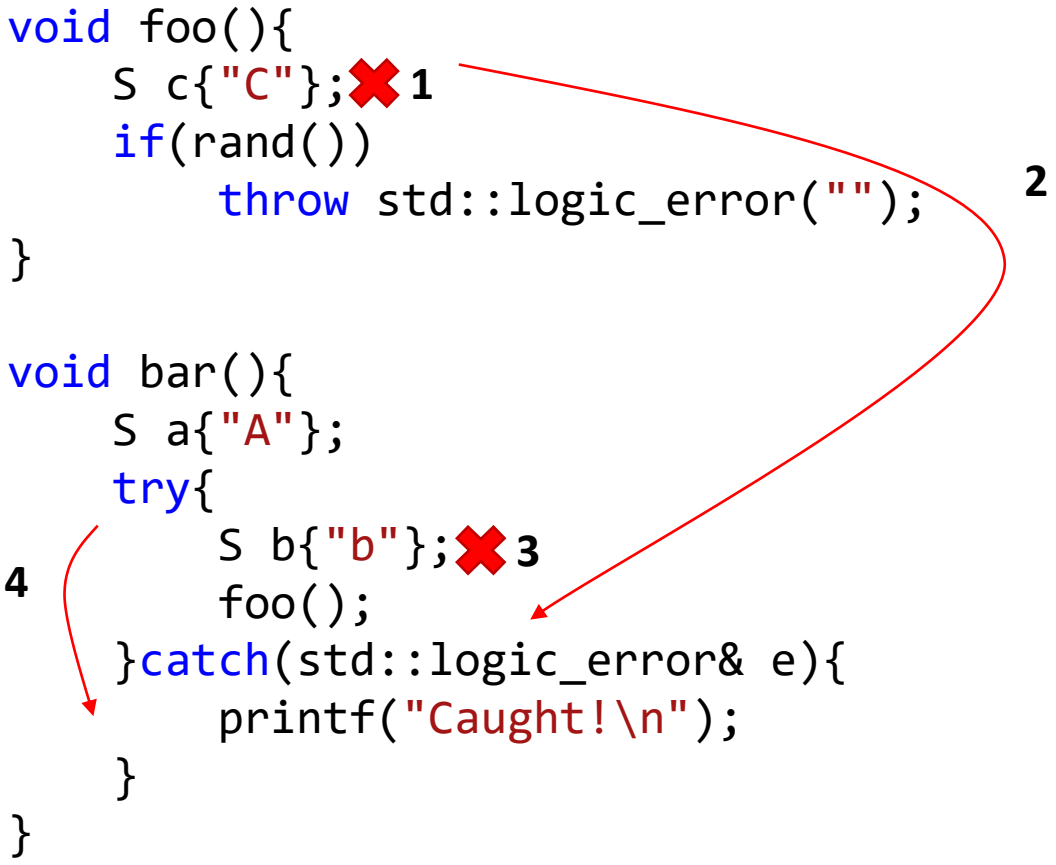
Раскрутка стека

В ходе раскрутки стека (**stack unwinding**) сначала уничтожаются все объекты в текущем кадре стека в порядке их создания.

Затем из стека достается адрес возврата, проверяется наличие обработчика в вызывающей функции.

Если обработчик есть – уничтожаются локальные переменные в текущем блоке кода и происходит прыжок на обработчик.

Если обработчика нет – раскрутка продолжается.



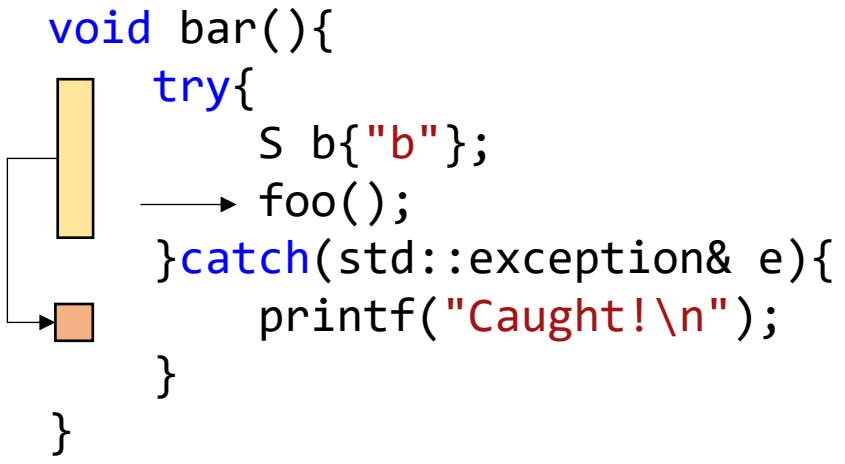
Идентификация обработчика

В приближенном виде процесс поиска обработчика описывается следующим образом.

Существует глобальная таблица, в которой хранится информация обо всех функциях, *не помеченных как noexcept*.

В этой структуре для каждой функции есть информация о коде раскрутки стека и обработки исключений: адрес начала/конца покрываемого кода и адрес кода раскрутки.

StartAddress	EndAddress	TargetAddress
…	…	…
0x00000FF1	0x00000FF9	0x00000FFA
…	…	…



Идентификация обработчика

Когда возникает исключение, в таблице производится поиск записи о текущей функции.

Если соответствующая запись найдена, то управление передается по соответствующему адресу. Код по адресу производит очистку стека (этот код есть, даже если в функции нет блока catch), проверку типа исключения и, если тип подходит, обработку исключения.

Если запись не найдена, вызывается std::terminate.

https://godbolt.org/z/6Kn56GWxc

https://habr.com/ru/post/279111/