

Глубины С (и С++)

авторы: Ольве Маудал (Olve Maudal) и Джон Джаггер (Jon Jagger)

перевод на русский: oscar.ru

оригинал: <https://olvemaudal.com/2011/10/10/deep-c/>



Программировать - сложно. А хорошо программировать на С и С++ ещё сложнее. Что и говорить, редко увидишь хотя бы один листинг на С/С++ размером в экран монитора, содержащий исключительно хороший и не вызывающий вопросов код. Почему же такой сомнительный код пишут профессиональные программисты? Дело в том, что многие из них не имеют представления о внутреннем устройстве языка, который они используют. Да, в некоторых ситуациях они могут понять, что результатом выполнения кода будет неопределённое или неуточнённое поведение, но почему это поведение является таковым - едва ли смогут сказать. На этих слайдах мы покажем небольшие фрагменты кода на С/С++ и используем их для обсуждения внутреннего устройства, ограничений и парадигм этих замечательных, но опасных языков программирования.



Предположим, вы собираетесь провести собеседование на должность embedded-разработчика. В ходе собеседования вы хотели бы узнать, насколько хорошо кандидат разбирается в языке, на котором пишет. Вот отличный пример для начала разговора:

```
int main()  
{  
    int a = 42;  
    printf(“%d\n”, a);  
}
```

Что произойдет, если вы выполните компиляцию, линковку и запуск этой программы?

Что произойдет, если вы выполните компиляцию, линковку и запуск этой программы?

```
int main()
{
    int a = 42;
    printf(“%d\n”, a);
}
```

Один из кандидатов может сказать:

Вы должны добавить `#include <stdio.h>` и `return 0`, и тогда компиляция и линковка пройдет успешно. После запуска на экран будет выведено число 42.



© www.ClipProject.info

и, казалось бы, это совершенно правильный ответ...

Что произойдет, если вы выполните компиляцию, линковку и запуск этой программы?

```
int main()
{
    int a = 42;
    printf(“%d\n”, a);
}
```

Но другой кандидат может проявить более глубокие знания. Она скажет что-то вроде:



Вероятно, вы хотели бы добавить `#include <stdio.h>`, которая содержит явное объявление функции `printf()`. Компиляция и линковка пройдут успешно, и после запуска в стандартный поток вывода будет отправлено число 42 и переход на новую строку.

Что произойдет, если вы выполните компиляцию, линковку и запуск этой программы?

```
int main()
{
    int a = 42;
    printf("%d\n", a);
}
```

ВОЗМОЖНО, она добавит:

Компилятор C++ не сможет выполнить компиляцию исходного варианта кода, так как требует явного объявления всех функций.

Однако правильный компилятор C сгенерирует неявное объявление для функции printf() и скомпилирует этот код в объектный файл.

И после линковки с библиотекой стандартного ввода-вывода в ней будет найдена реализация функции printf(), которая будет соответствовать этому неявному объявлению.

Так что компиляция, линковка и запуск пройдут успешно.

Хотя, возможно, компилятор выдаст предупреждение.



Что произойдет, если вы выполните компиляцию, линковку и запуск этой программы?

```
int main()
{
    int a = 42;
    printf("%d\n", a);
}
```

и, пока она в ударе, продолжит комментировать:

Если мы говорим про C99 (или C++ 98), то возвращаемое значение используется для индикации успешного выполнения, но в более ранних версиях C (например, ANSI C и K&R) возвращаемое значение является неопределенным - вы получите мусор.

Но так как возвращаемые значения часто передаются через регистры, я не удивлюсь, если мы получим число 3... Так как сама `printf()` вернет 3, то есть число символов, отправленных в стандартный поток вывода.



Что произойдет, если вы выполните компиляцию, линковку и запуск этой программы?

```
int main()  
{  
    int a = 42;  
    printf(“%d\n”, a);  
}
```

и, пока она в ударе, продолжит комментировать:



И говоря о стандартах C... Если вы стремитесь их соблюдать, то должны использовать `int main(void)` как точку входа - так гласит стандарт.

Важно использовать `void` для указания функций, не подразумевающих аргументов, потому что `'int f();'` означает, что функция `f` может принять произвольное число аргументов. Вероятно, в нашем случае имелось в виду `"int f(void);"`. Явное указание отсутствия аргументов через `void` точно не повредит.

Что произойдет, если вы выполните компиляцию, линковку и запуск этой программы?

```
int main()
{
    int a = 42;
    printf("%d\n", a);
}
```



Что произойдет, если вы выполните компиляцию, линковку и запуск этой программы?

```
int main(void) ←  
{  
    int a = 42;  
    printf(“%d\n”, a);  
}
```



Что произойдет, если вы выполните компиляцию, линковку и запуск этой программы?

```
int main(void)
{
    int a = 42;
    printf(“%d\n”, a);
}
```

и чтобы совсем сразить вас, она скажет:

И если вы позволите мне немного позанудствовать... Программа еще не до конца соответствует требованиям стандарта, так как исходный код должен завершаться символом перехода на новую строку.



Что произойдет, если вы выполните компиляцию, линковку и запуск этой программы?

```
int main(void)
{
    int a = 42;
    printf("%d\n", a);
}
```



Что произойдет, если вы выполните компиляцию, линковку и запуск этой программы?

```
int main(void)
{
    int a = 42;
    printf("%d\n", a);
}
```




Что произойдет, если вы выполните компиляцию, линковку и запуск этой программы?

```
int main(void)
{
    int a = 42;
    printf("%d\n", a);
}
```

Да, и не забудем добавить `#include` для явного объявления `printf()`.




Что произойдет, если вы выполните компиляцию, линковку и запуск этой программы?



```
int main(void)
{
    int a = 42;
    printf("%d\n", a);
}
```




Что произойдет, если вы выполните компиляцию, линковку и запуск этой программы?



```
int main(void)
{
    int a = 42;
    printf("%d\n", a);
}
```



Что произойдет, если вы выполните компиляцию, линковку и запуск этой программы?



```
#include <stdio.h>

int main(void)
{
    int a = 42;
    printf(“%d\n”, a);
}
```



Что произойдет, если вы выполните компиляцию, линковку и запуск этой программы?

```
#include <stdio.h>

int main(void)
{
    int a = 42;
    printf(“%d\n”, a);
}
```

Вот теперь это чертовски милая небольшая программа на C! Вы со мной согласны?



Что произойдет, если вы выполните компиляцию, линковку и запуск этой программы?

```
#include <stdio.h>

int main(void)
{
    int a = 42;
    printf("%d\n", a);
}
```

И вот что я получу, выполнив компиляцию, линковку и запуск этой программы на своей машине:

```
$ cc -std=c89 -c foo.c
$ cc foo.o
$ ./a.out
42
$ echo $?
3
```

```
$ cc -std=c99 -c foo.c
$ cc foo.o
$ ./a.out
42
$ echo $?
0
```

Есть ли разница между этими двумя кандидатами?



© www.ClipProject.info



© www.ClipProject.info

Не то чтобы большая, но пока мне очень нравятся ее ответы.

Теперь представим, что эти кандидаты - не конкретные люди. Возможно, это персонификации типичных черт характера, присущих инженерам вашей компании?

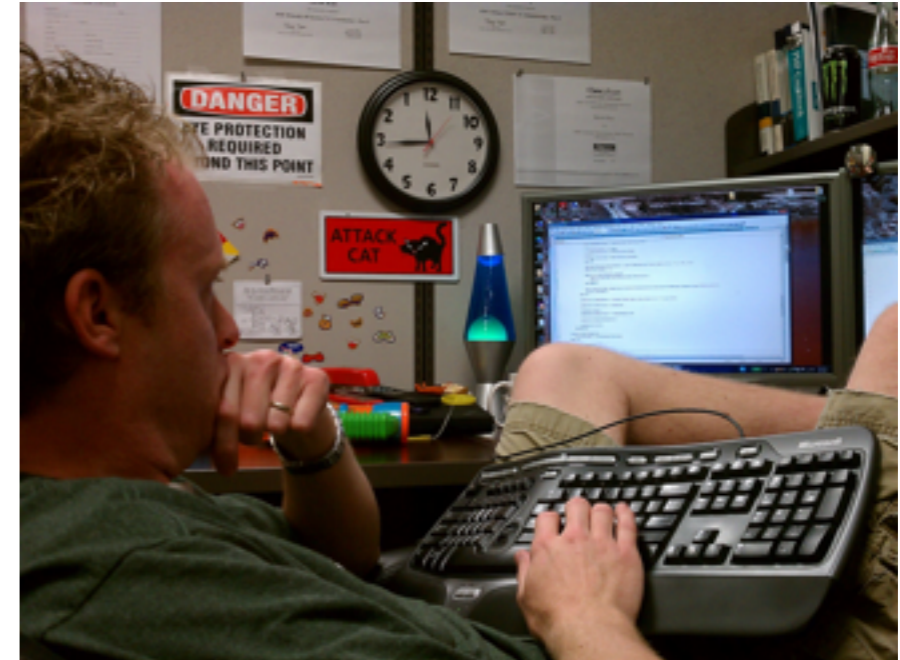
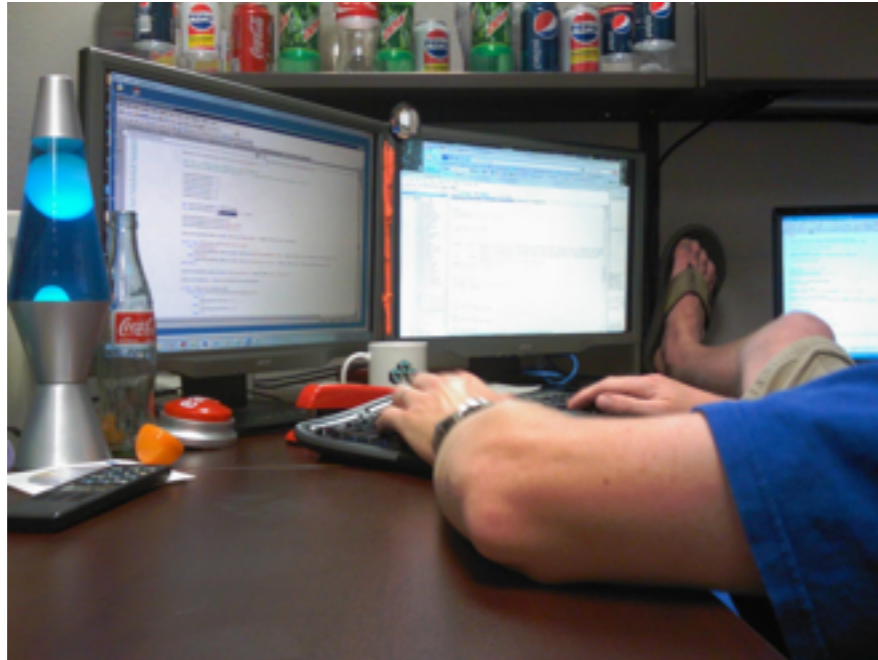


© www.ClipProject.info

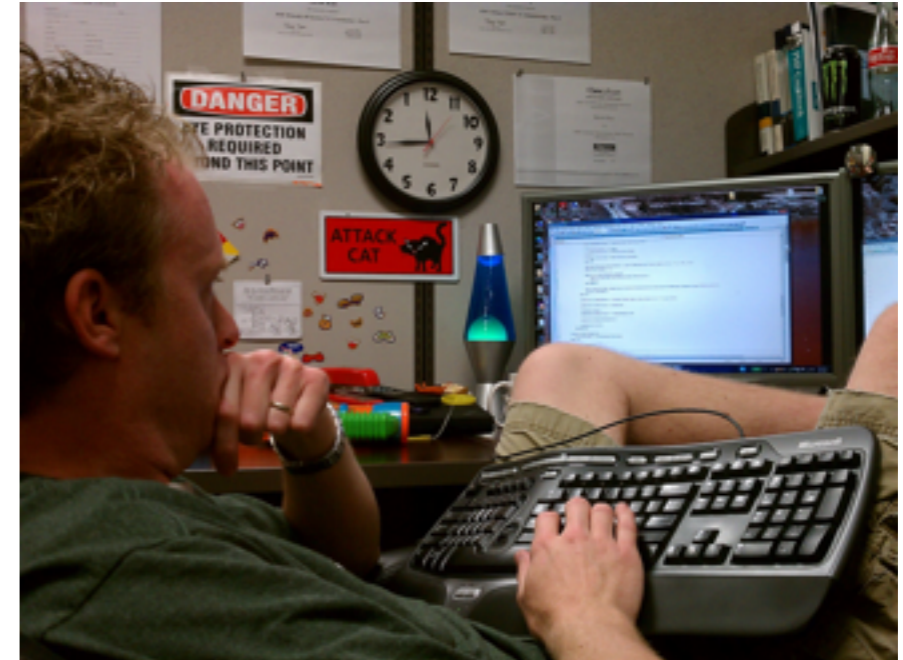
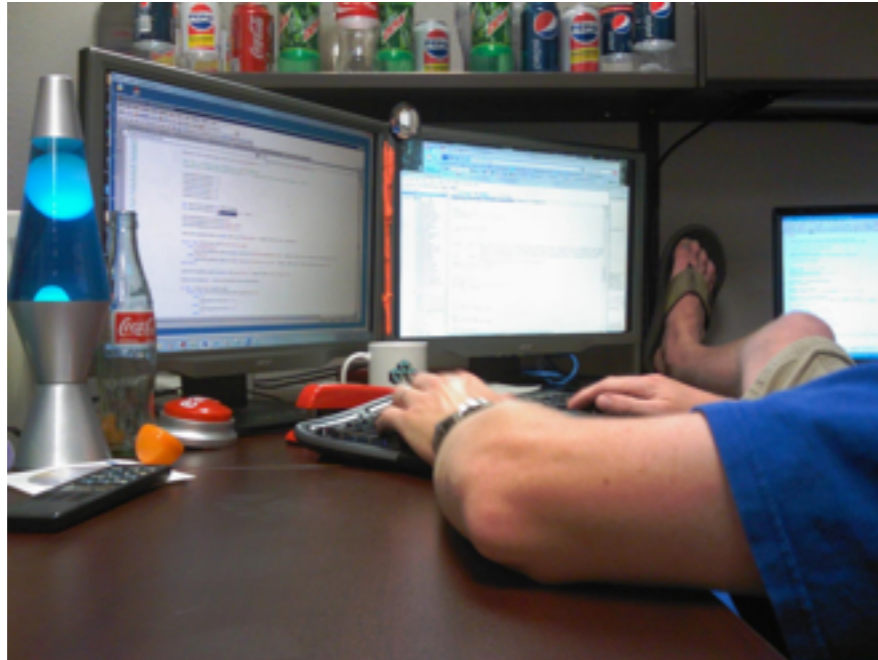


© www.ClipProject.info

Теперь представим, что эти кандидаты - не конкретные люди. Возможно, это персонификация типичных черт характера, присущих инженерам вашей компании?



Если бы ваши коллеги имели глубокие знания о языке программирования, который используют - принесло бы вам это серьезную пользу?



Давайте проверим, насколько глубоко наши кандидаты знают C и C++ ...



© www.ClipProject.info



© www.ClipProject.info

Давайте проверим, насколько глубоко наши кандидаты знают С и С++ ...


```
#include <stdio.h>

void foo(void)
{
    int a = 3;
    ++a;
    printf("%d\n", a);
}

int main(void)
{
    foo();
    foo();
    foo();
}
```

Мы получим 4, 4, 4.




Мы получим 4, 4, 4.



```
#include <stdio.h>

void foo(void)
{
    int a = 3;
    ++a;
    printf("%d\n", a);
}


int main(void)
{
    foo();
    foo();
    foo();
}
```



```
#include <stdio.h>

void foo(void)
{
    static int a = 3;
    ++a;
    printf("%d\n", a);
}

int main(void)
{
    foo();
    foo();
    foo();
}
```



```
#include <stdio.h>

void foo(void)
{
    static int a = 3;
    ++a;
    printf("%d\n", a);
}

int main(void)
{
    foo();
    foo();
    foo();
}
```

Мы получим 4, 5, 6.




Мы получим 4, 5, 6.



```
#include <stdio.h>

void foo(void)
{
    static int a = 3;
    ++a;
    printf("%d\n", a);
}


int main(void)
{
    foo();
    foo();
    foo();
}
```



```
#include <stdio.h>

void foo(void)
{
    static int a;
    ++a;
    printf("%d\n", a);
}

int main(void)
{
    foo();
    foo();
    foo();
}
```



```
#include <stdio.h>

void foo(void)
{
    static int a;
    ++a;
    printf("%d\n", a);
}

int main(void)
{
    foo();
    foo();
    foo();
}
```

Эм, это же неопределенное поведение?
Мы получим мусор?

Нет, мы получим 1, 2, 3.

Ладно... Но почему?

Потому что статические переменные
инициализируются нулями.


По стандарту статические переменные
инициализируются нулями, так что мы
получим 1, 2, 3.



```
#include <stdio.h>

void foo(void)
{
    static int a;
    ++a;
    printf("%d\n", a);
}

int main(void)
{
    foo();
    foo();
    foo();
}
```




```
#include <stdio.h>

void foo(void)
{
    int a;
    ++a;
    printf("%d\n", a);
}

int main(void)
{
    foo();
    foo();
    foo();
}
```

```
#include <stdio.h>

void foo(void)
{
    int a;
    ++a;
    printf("%d\n", a);
}

int main(void)
{
    foo();
    foo();
    foo();
}
```



Мы получим 1, 1, 1.

Эм, почему ты так думаешь?

Вы же сами сказали про инициализацию нулями.

Но это не статическая переменная.

А, ну тогда мы получим мусор.

Значение переменной `a` будет неопределенным, так что в теории мы получим мусор. Однако на практике локальные переменные обычно хранятся в стеке вызовов, и `a` при каждом вызове будет размещаться в одной и той же ячейке памяти, так что мы получим три значения, каждое из которых на 1 больше предыдущего. При условии, что ваш компилятор не выполнит оптимизацию...

На своей машине я действительно получил 1, 2, 3.

Я не удивлена... Если вы компилируете в отладочном режиме, то рантайм любезно сделает для памяти стека `memset` в 0.



```
#include <stdio.h>

void foo(void)
{
    int a;
    ++a;
    printf("%d\n", a);
}

int main(void)
{
    foo();
    foo();
    foo();
}
```

Как ты думаешь, почему статические переменные инициализируются нулем, а локальные - не инициализируются?



А?..

Установка значений локальных переменных в 0 увеличит время выполнения пропорционально числу функций в программе. А язык С ориентирован на минимизацию времени выполнения кода.



Инициализация нулями всех глобальных данных - это однократная операция, которая происходит при запуске - видимо, поэтому она считается допустимой в С.

И, если быть точным, то в С++ статические переменные инициализируется не нулем, а значениями по умолчанию... Которые для встроенных типов данных равны нулю.

```
#include <stdio.h>
```

```
void foo(void)
```

```
{
```

```
    int a;
```

```
    ++a;
```

```
    printf("%d\n", a);
```

```
}
```

```
int main(void)
```

```
{
```

```
    foo();
```

```
    foo();
```

```
    foo();
```

```
}
```

```
#include <stdio.h>
```




```
void foo(void)
{
    int a;
    ++a;
    printf("%d\n", a);
}
```

```
int main(void)
{
    foo();
    foo();
    foo();
}
```

```
#include <stdio.h>
```

```
void foo(void)
{
    int a;
    ++a;
    printf("%d\n", a);
}
```



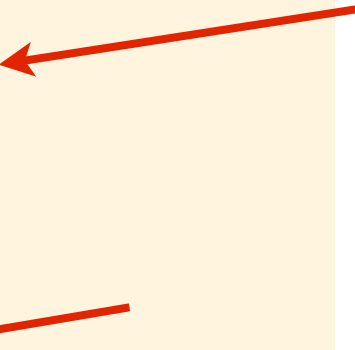
```
int main(void)
{
    foo();
    foo();
    foo();
}
```

```
#include <stdio.h>

static int a;

void foo(void)
{
    int a;
    ++a;
    printf("%d\n", a);
}

int main(void)
{
    foo();
    foo();
    foo();
}
```




```
#include <stdio.h>

static int a;


void foo(void)
{
    ++a;
    printf("%d\n", a);
}

int main(void)
{
    foo();
    foo();
    foo();
}
```




```
#include <stdio.h>

static int a;

void foo(void)
{
    ++a; 
    printf("%d\n", a);
}

int main(void)
{
    foo();
    foo();
    foo();
}
```

```
#include <stdio.h>

static int a;

void foo(void)
{
    ++a;
    printf("%d\n", a);
}

int main(void)
{
    foo();
    foo();
    foo();
}
```



Мы получим 1, 2, 3.

Так, а почему?

Потому что а - статическая переменная...

Согласен...

Здорово!

```
#include <stdio.h>
static int a;
void foo(void)
{
    ++a;
    printf("%d\n", a);
}
int main(void)
{
    foo();
    foo();
    foo();
}
```

```
#include <stdio.h>
```

```
int a;
```

```
void foo(void)
```

```
{  
    ++a;  
    printf("%d\n", a);  
}
```

```
int main(void)
```

```
{  
    foo();  
    foo();  
    foo();  
}
```

```
#include <stdio.h>

int a;

void foo(void)
{
    ++a;
    printf("%d\n", a);
}

int main(void)
{
    foo();
    foo();
    foo();
}
```



Мы получим мусор.

Почему ты так решил?

Ох, тут тоже будет инициализация нулем?

Да.

Наверное, мы получим 1, 2, 3?

Да...

Ты понимаешь разницу между этим примером кода и предыдущим (с ключевым словом `static` для `int a`)?

Не особо... Хотя секунду...
Это как-то связано с областью видимости?

Да, связано...

```
#include <stdio.h>

int a;

void foo(void)
{
    ++a;
    printf("%d\n", a);
}

int main(void)
{
    foo();
    foo();
    foo();
}
```

Мы получим 1, 2, 3, ведь переменная все еще размещена в статическом хранилище и инициализируется нулем.



Ты понимаешь разницу между этим примером кода и предыдущим (с ключевым словом `static` для `int a`)?

Конечно, все дело в области видимости линкера. В этом примере переменная будет доступна из других модулей. Но если сделать ее статической (как в предыдущем примере), то она становится локальной для модуля и перестает быть видимой для линкера.

Теперь я хочу показать тебе полный отвал башки!

```
#include <stdio.h>

void foo(void)
{
    int a;
    printf("%d\n", a);
}

void bar(void)
{
    int a = 42;
}

int main(void)
{
    bar();
    foo();
}
```

```
$ cc foo.c && ./a.out
42
```

Можешь объяснить это?

А?..



Возможно, у этого компилятора есть пул именованных переменных для переиспользования. Например, если переменная `a` сначала объявлена в функции `bar()`, а потом в функции `foo()` объявляется переменная с тем же именем и типом, то для ее хранения используется та же область памяти. Если вы измените имя переменной, например, на `b`, то, думаю, вы уже не получите 42.

Мда...

Теперь я хочу показать тебе полный отвал башки!

```
#include <stdio.h>

void foo(void)
{
    int a;
    printf("%d\n", a);
}

void bar(void)
{
    int a = 42;
}

int main(void)
{
    bar();
    foo();
}
```

```
$ cc foo.c && ./a.out
42
```

Классно!

Видимо, я должна рассказать о стеке вызовов и фреймах активации?

Ты уже показала, что понимаешь, о чем речь. Как ты думаешь, что произойдет, если мы включим оптимизацию или используем другой компилятор?



Во время оптимизации могут произойти разные вещи. Я могу предположить, что вызов `bar()` будет опущен, так как он не имеет побочных эффектов (функция ничего не возвращает и не меняет значения глобальных переменных). И я не удивлюсь, если код `foo()` будет встроен в `main()`, то есть вызова функции не будет (но так как `foo()` находится в области видимости линкера, то объектный код для нее должен быть сгенерирован на тот случай, если она будет использоваться в другом модуле). В любом случае, я думаю, что после оптимизации выводимое значение будет уже другим.


```
#include <stdio.h>

void foo(void)
{
    int a;
    printf("%d\n", a);
}

void bar(void)
{
    int a = 42;
}

int main(void)
{
    bar();
    foo();
}
```

```
$ cc -O foo.c && ./a.out
1606415608
```



Да, мусор!

А что ты скажешь об этом примере?

```
#include <stdio.h>

void foo(void)
{
    int a = 41;
    a = a++;
    printf("%d\n", a);
}

int main(void)
{
    foo();
}
```

```
#include <stdio.h>

void foo(void)
{
    int a = 41;
    a = a++;
    printf("%d\n", a);
}

int main(void)
{
    foo();
}
```



Я никогда не написал бы такой код.

Рад это слышать!

Думаю, что мы получим 42.

Почему?

Ну а какие еще варианты?

Вообще, я действительно получил 42, запустив этот код на своей машине.

Я же говорил!

Но вообще-то это является неопределенным поведением.

Ну да, говорю же, я никогда не написал бы такой код.

Примечание насчет числа 42, которое объясняет смайлик в последней фразе:

https://ru.wikipedia.org/wiki/Ответ_на_главный_вопрос_жизни,_вселенной_и_всего_такого

```
#include <stdio.h>

void foo(void)
{
    int a = 41;
    a = a++;
    printf("%d\n", a);
}

int main(void)
{
    foo();
}
```



Мы получим неопределенное значение.

Но у меня нет ошибок при компиляции, и я получаю 42.

Значит, вам нужно повесить уровень предупреждений в настройках компилятора. Значение `a` однозначно не определено после присваивания и инкремента, потому что здесь нарушается одно из фундаментальных правил C (и C++) - "между двумя точками следования значение переменной может изменяться не более одного раза". В данном примере это происходит дважды, и поэтому значение `a` является неопределенным.

Ты говоришь, значение `a` может быть любым? Но я же получил именно 42.

Да, любым! Вы могли получить 42, 41, 43, 0, 1099, или что-нибудь еще... Я не удивлена, что вы получили 42... Что же еще мы могли увидеть в данной ситуации? Или, возможно, компилятор присваивает 42 всем неопределенным значениям ;-)

А что ты скажешь об этом примере?

```
#include <stdio.h>

int b(void) { puts("3"); return 3; }
int c(void) { puts("4"); return 4; }

int main(void)
{
    int a = b() + c();
    printf("%d\n", a);
}
```

```
#include <stdio.h>

int b(void) { puts("3"); return 3; }
int c(void) { puts("4"); return 4; }

int main(void)
{
    int a = b() + c();
    printf("%d\n", a);
}
```



О, это легко - получим 3, 4 и 7.

Вообще, мы можем также получить 4, 3 и 7.

А? Порядок оценки выражений не определен?

Он не неопределен, он не уточнен.

Да неважно. Глупые компиляторы. Думаю, мы хотя бы должны получить предупреждение?

Предупреждение о чем?

```
#include <stdio.h>

int b(void) { puts("3"); return 3; }
int c(void) { puts("4"); return 4; }

int main(void)
{
    int a = b() + c();
    printf("%d\n", a);
}
```

Порядок оценки для большинства выражений в С и С++ не уточнен, и компилятор сам определяет оптимальный порядок для конкретной платформы. И тут мы опять возвращаемся к точкам следования.

Этот код соответствует стандарту. Мы можем получить 3, 4, 7 или 4, 3, 7 - это зависит от компилятора.

Моя жизнь была бы проще, если бы мои коллеги тоже об этом знали.





© www.ClipProject.info



© www.ClipProject.info

На данный момент я считаю, что парень продемонстрировал только поверхностные знания о программировании на C, а вот ответы девушки были превосходны.

В каких аспектах она ориентируется лучше большинства кандидатов?



- Объявление и определение переменных
- Соглашение о вызове и фреймы активации
- Точки следования
- Модели памяти
- Оптимизация от компилятора
- Знание различных стандартов C

Давайте поговорим о:



Точках следования
Различных стандартах С

Что мы получим в результате запуска этих примеров?

1

```
int a=41; a++; printf("%d\n", a);
```

42

2

```
int a=41; a++ & printf("%d\n", a);
```

неопределенное
значение

3

```
int a=41; a++ && printf("%d\n", a);
```

42

4

```
int a=41; if (a++ < 42) printf("%d\n", a);
```

42

5

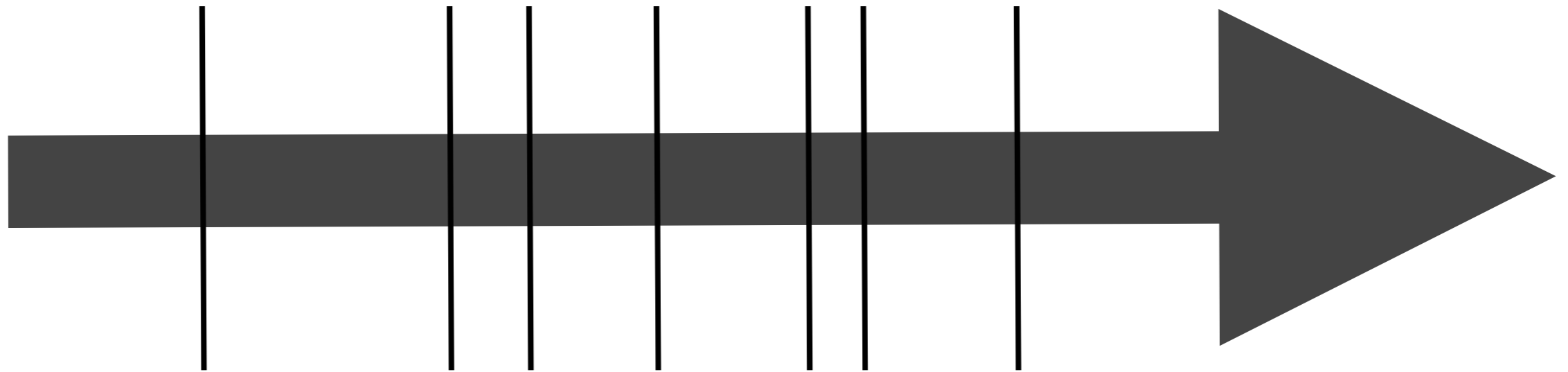
```
int a=41; a = a++; printf("%d\n", a);
```

неопределенное
значение

Когда именно в C и C++ проявляются побочные эффекты?

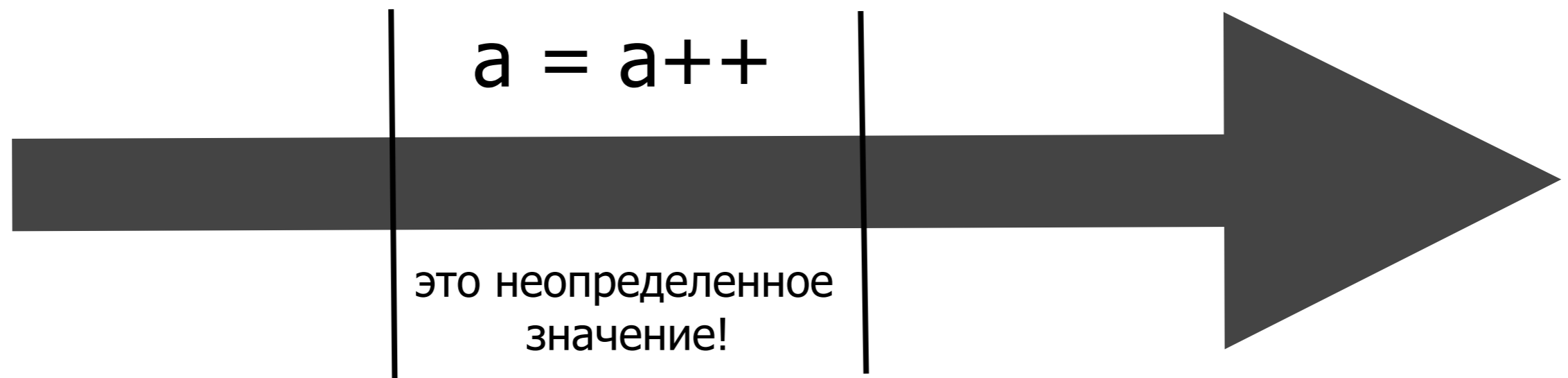
Точка следования

- это любая точка программы, в которой гарантируется, что все побочные эффекты предыдущих вычислений уже проявились, а побочные эффекты последующих ещё отсутствуют. (5.1.2.3)



Точки следования - правило 1

Между соседними точками следования значение объекта должно изменяться не более одного раза. (6.5)



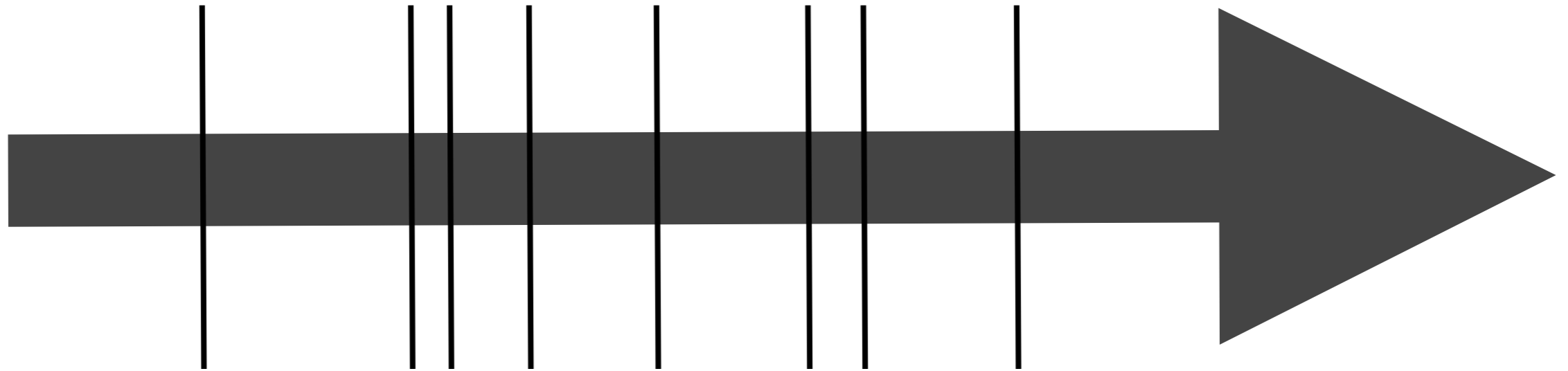
Точки следования - правило 2

Кроме того, к исходному значению объекта разрешен доступ только для чтения - для вычисления его нового значения. (6.5)



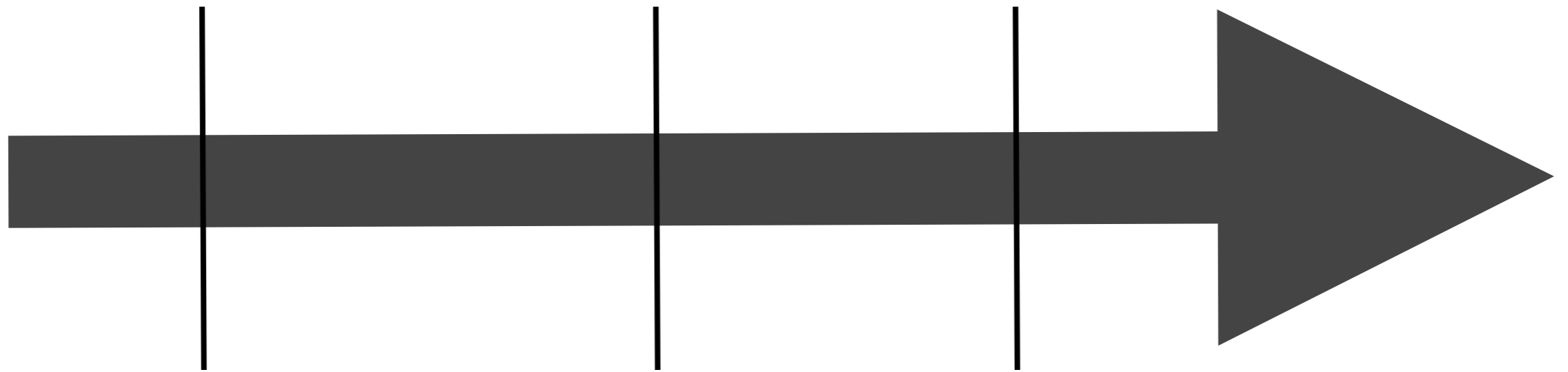
Точки следования

Масса разработчиков думает, что в С много точек следования.



Точки следования

Но на самом деле их не так уж и много.



Это позволяет максимизировать возможности компилятора по оптимизации.


```

/* K&R C */

void say_it(a, s)
    int a;
    char s[];
{
    printf("%s %d\n", s, a);
}

main()
{
    int a = 42;
    puts("Welcome to classic C");
    say_it(a, "the answer is");
}

```

```

/* C89 */

void say_it(int a, char * s)
{
    printf("%s %d\n", s, a);
}

main()
{
    int a = 42;
    puts("Welcome to C89");
    say_it(a, "the answer is");
}

```

```

// C++ (C++98)

#include <cstdio>

struct X {
    int a;
    const char * s;
    explicit X(const char * s, int a = 42)
        : a(a), s(s) {}
    void say_it() const {
        std::printf("%s %d\n", s, a);
    }
};

int main()
{
    X("the answer is").say_it();
}

```

```

// C99

struct X
{
    int a;
    char * s;
};

int main(void)
{
    puts("Welcome to C99");
    struct X x = { .s = "the answer is", .a = 42 };
    printf("%s %d\n", x.s, x.a);
}

```

Давайте вернемся к нашим кандидатам...

Что ты скажешь об этом примере?

```
#include <stdio.h>

struct X { int a; char b; int c; };

int main(void)
{
    printf("%d\n", sizeof(int));
    printf("%d\n", sizeof(char));
    printf("%d\n", sizeof(struct X));
}
```

```
#include <stdio.h>

struct X { int a; char b; int c; };

int main(void)
{
    printf("%d\n", sizeof(int));
    printf("%d\n", sizeof(char));
    printf("%d\n", sizeof(struct X));
}
```

Мы получим 4, 1 и 12.

Да, на своей машине я получил именно это.

Еще бы, ведь `sizeof` возвращает размер переменной в байтах. В C `int` занимает 32 бита или 4 байта, `char` занимает 1 байт, а размер структуры всегда кратен 4.

Ясно.

Он хочет печеньеку за ответ?..



```
#include <stdio.h>

struct X { int a; char b; int c; };

int main(void)
{
    printf("%d\n", sizeof(int));
    printf("%d\n", sizeof(char));
    printf("%d\n", sizeof(struct X));
}
```

Хм... Для начала давайте исправим код. Возвращаемое значение для `sizeof` имеет тип `size_t`, и это не тоже самое, что и `int`, так что `%d` неподходящий спецификатор форматирования для `printf` в данном случае.



Ладно, и какой спецификатор нам нужен?

Это непростой вопрос. `size_t` является целочисленным беззнаковым типом, и на 32-битных машинах ему соответствует `unsigned int`, а на 64-битных - обычно `unsigned long`. В C99 добавили специальный спецификатор `%zu` для типа `size_t` - по возможности следует использовать его.

Ладно, давай это поправим и ты начнешь отвечать на вопрос.

```
#include <stdio.h>

struct X { int a; char b; int c; };

int main(void)
{
    printf("%d\n", sizeof(int));
    printf("%d\n", sizeof(char));
    printf("%d\n", sizeof(struct X));
}
```



```
#include <stdio.h>

struct X { int a; char b; int c; };


int main(void)
{
    printf("%zu\n", sizeof(int));
    printf("%zu\n", sizeof(char));
    printf("%zu\n", sizeof(struct X));
}
```



```
#include <stdio.h>

struct X { int a; char b; int c; };

int main(void)
{
    printf("%zu\n", sizeof(int));
    printf("%zu\n", sizeof(char));
    printf("%zu\n", sizeof(struct X));
}
```



Теперь все зависит от конкретной платформы и опций компилятора. Единственное, что мы знаем точно - sizeof для char вернет 1. Вы запускаете этот код на 64-битной машине?

Да, у меня 64-битная машина, запущенная в режиме совместимости с 32-битными приложениями.

Тогда, полагаю, мы получим 4, 1 и 12 (последнее - из-за выравнивания памяти)

Но, конечно, это зависит от опций компилятора. Мы могли бы получить 4, 1 и 9, если бы компилятор упаковывал структуры - например, в gcc это можно сделать с помощью флага `fpack-struct`.


```
#include <stdio.h>

struct X { int a; char b; int c; };

int main(void)
{
    printf("%zu\n", sizeof(int));
    printf("%zu\n", sizeof(char));
    printf("%zu\n", sizeof(struct X));
}
```

Действительно, на своей машине я получил 4, 1 и 12. Почему 12?



Доступ к памяти, размер которой меньше машинного слова - очень ресурсоемкая операция. Поэтому компилятор оптимизирует размещение переменных, выравнивая их по границам машинных слов.

Почему без выравнивания доступ к памяти стал бы ресурсоемким?

Набор инструкций для большинства процессоров оптимизирован для обмена данными между памятью и CPU. Если бы нужно было записать значение в переменную, которая пересекает границу машинного слова, то потребовалось бы прочитать оба слова, выделить значение с помощью маски, изменить значение, опять собрать два слова с помощью маски и записать их. Возможно, это было бы раз в 10 медленнее, чем при использовании выравнивания. А язык C ориентирован на минимизацию времени выполнения кода.

```
#include <stdio.h>

struct X { int a; char b; int c; };

int main(void)
{
    printf("%zu\n", sizeof(int));
    printf("%zu\n", sizeof(char));
    printf("%zu\n", sizeof(struct X));
}
```

Что произойдет, если я добавлю в структуру char d?

Если вы добавите его в конец структуры - я думаю, на вашей машине она будет занимать 16 байт. Потому что размер в 13 байт явно не является оптимальным - а что если у вас будет массив таких структур? Но если вы добавите ее сразу после char b - я думаю, вероятно размер структуры останется равным 12 байтам.

А почему тогда компилятор не меняет порядок элементов структуры для уменьшения используемой памяти и повышения скорости выполнения?

В принципе, некоторые языки позволяют это - но не C/C++. Идеология C/C++ подразумевает "ручное" управление памятью.



```
#include <stdio.h>

struct X { int a; char b; int c; };

int main(void)
{
    printf("%zu\n", sizeof(int));
    printf("%zu\n", sizeof(char));
    printf("%zu\n", sizeof(struct X));
}
```

А что будет, если я добавлю `char * d` в конец структуры?

Вы говорили, что у вас 64-битная машина, так что указатели, вероятно, занимают 8 байт... Возможно, структура займет 20 байт? Хотя, наверное, 64-битные указатели тоже должны выравниваться? Возможно, мы получим 4, 1 и 24?

Прекрасный ответ! Даже не важно, что я получу на своей машине - мне нравятся твои аргументы и проницательность.



В каких аспектах она ориентируется лучше большинства кандидатов?



- Различия 32- и 64-битных платформ
- Выравнивание памяти
- Оптимизации при работе с памятью
- Дух С

Давайте поговорим о:



Моделях памяти
Оптимизациях
Духе С

Модели памяти

Статическое хранилище

Объект, который имеет внутреннее или внешнее связывание или объявлен с ключевым словом **static** хранится в сегменте памяти **data** или **bss**, и имеет *статическое время хранения*. Время его жизни - на протяжении выполнения всей программы. (6.2.4)

```
int * immortal(void)
{
    static int storage = 42;
    return &storage;
}
```

Модели памяти

Автоматическое хранилище

Объект без связывания и объявленный без ключевого слова **static** хранится в стеке и имеет *автоматическое время хранения*. Время его жизни - на протяжении выполнения блока кода, с которым он связан. (6.2.4)

```
int * zombie(void)
{
    auto int storage = 42;
    return &storage;
}
```

Модели памяти

Динамическое хранилище

Объект, созданный вызовом `calloc`, `malloc` или `realloc`, хранится в куче (heap) и имеет *динамическое время хранения*. Время его жизни - от вызова функции выделения памяти до вызова функции освобождения памяти. (7.20.3)

```
int * finite(void)
{
    int * ptr = malloc(sizeof *ptr);
    *ptr = 42;
    return ptr;
}
```


Оптимизации от компилятора

По умолчанию вы должны компилировать с включенной оптимизацией. Это позволит обнаружить больше потенциальных проблем.

opt.c

```
#include <stdio.h>

int main(void)
{
    int a;
    printf("%d\n", a);
}
```

```
>cc -Wall opt.c
```

предупреждений нет!

opt.c

```
#include <stdio.h>

int main(void)
{
    int a;
    printf("%d\n", a);
}
```

```
>cc -Wall -O opt.c
warning: 'a' is uninitialized
```

Дух С

"У духа С много граней, но его сутью является понимание фундаментальных принципов, которые заложены в основу языка."

(из вступления "Rationale for International Standard — Programming Language — C")

- ✱ Доверять программисту
- ✱ Сохранять простоту и лаконичность языка
- ✱ Предоставлять один способ сделать что-то
- ✱ Скорость выполнения важнее переносимости
- ✱ Поддерживать концептуальную чистоту
- ✱ Не мешать программисту делать то, что ему нужно

Давайте поговорим с нашими кандидатами о C++

По шкале от 1 до 10: насколько хорошо ты знаешь C++?

Я оцениваю себя на 8 или 9.



На 4, может на 5. Мне еще многое нужно узнать о C++.



7



Примечание: на фото - Бьёрн Страуструп, создатель языка C++

Что ты скажешь об этом примере?

```
#include <iostream>

struct X
{
    int a;
    char b;
    int c;
};

int main(void)
{
    std::cout << sizeof(X) << std::endl;
}
```

```
#include <iostream>

struct X
{
    int a;
    char b;
    int c;
};

int main(void)
{
    std::cout << sizeof(X) << std::endl;
}
```

Эта структура является POD (Plain Old Data, простой структурой данных) и стандарт C++ гарантирует, что она будет размещена в памяти также, как в C.

Так что на вашей машине, полагаю, мы опять получим 12.


И, кстати, странно использовать `func(void)` вместо `func()`, потому что в C++ `void` используется по умолчанию. Это справедливо и для функции `main()`. Конечно, хуже от этого не будет, но просто код выглядит так, как будто его писал упертый C программист, пытающийся освоить C++.



```
#include <iostream>

struct X
{
    int a;
    char b;
    int c;
};


int main(void)
{
    std::cout << sizeof(X) << std::endl;
}
```



```
#include <iostream>

struct X
{
    int a;
    char b;
    int c;
};

int main()
{
    std::cout << sizeof(X) << std::endl;
}
```




```
#include <iostream>

struct X
{
    int a;
    char b;
    int c;
};

int main()
{
    std::cout << sizeof(X) << std::endl;
}
```

Мы получим 12.

Ладно.

А если мы добавим туда метод?



```
#include <iostream>

struct X
{
    int a;
    char b;
    int c;
};

int main()
{
    std::cout << sizeof(X) << std::endl;
}
```



```
#include <iostream>

struct X
{
    int a;
    char b;
    int c;
};

int main()
{
    std::cout << sizeof(X) << std::endl;
}
```



```
#include <iostream>

struct X
{
    int a;
    char b;
    int c;
};

int main()
{
    std::cout << sizeof(X) << std::endl;
}
```



```
#include <iostream>

struct X
{
    int a;
    char b;
    int c;
    void set_value(int v) { a = v; }
};

int main()
{
    std::cout << sizeof(X) << std::endl;
}
```



```
#include <iostream>

struct X
{
    int a;
    char b;
    int c;

    void set_value(int v) { a = v; }
};

int main()
{
    std::cout << sizeof(X) << std::endl;
}
```



Эээ, а разве в C++ так можно? Я думаю, нужно объявить класс.

В чем отличие между классами и структурами в C++?

Ну, в классе можно объявить методы, а в структуре вроде нет.
Или все-таки можно? Будут какие-то отличия в области видимости?

```
#include <iostream>

struct X
{
    int a;
    char b;
    int c;

    void set_value(int v) { a = v; }
};

int main()
{
    std::cout << sizeof(X) << std::endl;
}
```



В любом случае, мы получим 16. Потому что в структуру будет добавлен указатель на метод.

Так.

А если я добавлю еще два метода?

```
#include <iostream>

struct X
{
    int a;
    char b;
    int c;

    void set_value(int v) { a = v; }
};

int main()
{
    std::cout << sizeof(X) << std::endl;
}
```




```
#include <iostream>

struct X
{
    int a;
    char b;
    int c;

    void set_value(int v) { a = v; }
};

int main()
{
    std::cout << sizeof(X) << std::endl;
}
```



```
#include <iostream>

struct X
{
    int a;
    char b;
    int c;

    void set_value(int v) { a = v; }
    int get_value() { return a; }
    void increase_value() { a++; }
};

int main()
{
    std::cout << sizeof(X) << std::endl;
}
```



```
#include <iostream>

struct X
{
    int a;
    char b;
    int c;

    void set_value(int v) { a = v; }
    int get_value() { return a; }
    void increase_value() { a++; }
};

int main()
{
    std::cout << sizeof(X) << std::endl;
}
```

Видимо, мы получим 24? Ведь добавятся еще два указателя?

На моей машине я получаю меньшее число.

А, точно, там же таблица с указателями и достаточно только одного указателя на эту таблицу! Я все это знаю, просто забыл.

Вообще, я получил 12.

Э? Возможно, какая-то странная оптимизация от компилятора, раз эти методы нигде не вызываются.



```
#include <iostream>

struct X
{
    int a;
    char b;
    int c;

    void set_value(int v) { a = v; }
    int get_value() { return a; }
    void increase_value() { a++; }
};

int main()
{
    std::cout << sizeof(X) << std::endl;
}
```

Как думаешь, что мы получим после запуска этого кода?

На вашей машине? Видимо, опять 12.

Так, а почему?

Потому что методы не влияют на размер структуры. Объекты ничего не знают о своих методах, а вот методы знают об объектах.

Это станет очевидным, если переписать этот же код на С.



C++

```
struct X
{
    int a;
    char b;
    int c;

    void set_value(int v) { a = v; }
    int get_value() { return a; }
    void increase_value() { a++; }
};
```

C++

```
struct X
{
    int a;
    char b;
    int c;

    void set_value(int v) { a = v; }
    int get_value() { return a; }
    void increase_value() { a++; }
};
```

C

```
struct X
{
    int a;
    char b;
    int c;
};

void set_value(struct X * this, int v) { this->a = v; }
int get_value(struct X * this) { return this->a; }
void increase_value(struct X * this) { this->a++; }
```

Вот так?

Да, именно, и теперь очевидно, что методы не изменяют размер объекта.



```
#include <iostream>

struct X
{
    int a;
    char b;
    int c;

    void set_value(int v) { a = v; }
    int get_value() { return a; }
    void increase_value() { a++; }
};

int main()
{
    std::cout << sizeof(X) << std::endl;
}
```



```
#include <iostream>

struct X
{
    int a;
    char b;
    int c;

    void set_value(int v) { a = v; }
    int get_value() { return a; }
    void increase_value() { a++; }
};

int main()
{
    std::cout << sizeof(X) << std::endl;
}
```




```
#include <iostream>

struct X
{
    int a;
    char b;
    int c;

    virtual void set_value(int v) { a = v; }
    int get_value() { return a; }
    void increase_value() { a++; }
};

int main()
{
    std::cout << sizeof(X) << std::endl;
}
```



```
#include <iostream>

struct X
{
    int a;
    char b;
    int c;

    virtual void set_value(int v) { a = v; }
    int get_value() { return a; }
    void increase_value() { a++; }
};

int main()
{
    std::cout << sizeof(X) << std::endl;
}
```

Что произойдет теперь?

Размер структуры, вероятно, увеличится. Стандарт C++ не упоминает, как именно должны быть реализованы виртуальные классы и перегрузка методов, но типовой подход заключается в использовании таблицы виртуальных методов, на которую потребуется указатель. В нашем случае это добавит 8 байт к размеру. Значит, будет 20 байт?

Я получил 24.

А, понятно. Вероятно, это связано с выравниванием памяти.



```
#include <iostream>

struct X
{
    int a;
    char b;
    int c;

    virtual void set_value(int v) { a = v; }
    int get_value() { return a; }
    void increase_value() { a++; }
};

int main()
{
    std::cout << sizeof(X) << std::endl;
}
```



```
#include <iostream>

struct X
{
    int a;
    char b;
    int c;

    virtual void set_value(int v) { a = v; }
    virtual int get_value() { return a; }
    virtual void increase_value() { a++; }
};

int main()
{
    std::cout << sizeof(X) << std::endl;
}
```



```
#include <iostream>

struct X
{
    int a;
    char b;
    int c;

    virtual void set_value(int v) { a = v; }
    virtual int get_value() { return a; }
    virtual void increase_value() { a++; }
};

int main()
{
    std::cout << sizeof(X) << std::endl;
}
```

А что мы получим теперь?

Думаю, опять 24 - потому что достаточно одной vtable на класс.

Так, а что такое vtable?

Это механизм для реализации одного из видов полиморфизма в C++. В сущности, это таблица с адресами методов, она позволяет осуществлять перегрузку методов при наследовании классов.



Давайте обсудим еще один пример...

```
#include "B.hpp"

class A {
public:
    A(int sz) { sz_ = sz; v = new B[sz_]; }
    ~A() { delete v; }
    // ...
private:
    // ...
    B * v;
    int sz_;
};
```

Взгляни на этот фрагмент кода. Представь, что я джуниор, которого ты собеседуешь в свою команду, и это мой код. Пожалуйста, будь придирчивым и попытайся дружелюбно рассказать мне о подводных камнях C++ и, возможно, научить меня, как надо писать на этом языке.

Это выглядит очень хреново. Это ваш код? Во-первых...

Никогда не используйте 2 пробела для отступов.

Фигурную скобку после имени класса нужно перенести на новую строку.

sz_? Никогда не видел такой нотации, следует использовать `_sz` (как у "Банды четырех") или `m_sz` (это стандарт Microsoft).



```
#include "B.hpp"

class A {
public:
    A(int sz) { sz_ = sz; v = new B[sz_]; }
    ~A() { delete v; }
    // ...
private:
    // ...
    B * v;
    int sz_;
};
```

Что-нибудь еще?

A?

Вы про использование `delete[]` вместо `delete` при высвобождении памяти массива объектов? Ну, я достаточно опытен и знаю, что современные компиляторы и так это поймут.

Так. А что насчет "правила трех"? Нужно ли поддерживать или запрещать возможность копирования этих объектов?

Да без разницы... Никогда не слышал о таком правиле, но, конечно, если копировать эти объекты, то могут возникнуть проблемы. Думаю, это и есть дух C++... Разработчики должны страдать.




```
#include "B.hpp"

class A {
public:
    A(int sz) { sz_ = sz; v = new B[sz_]; }
    ~A() { delete v; }
    // ...
private:
    // ...
    B * v;
    int sz_;
};
```

И, кстати, думаю, вы знаете, что в C++ все деструкторы всегда следует объявлять как виртуальные. Я прочел в какой-то книге, что очень важно избегать нарезки объектов при высвобождении памяти для объектов-наследников.



Или что-то в таком духе...

Он хочет еще одну печенюшку?

```
#include "B.hpp"

class A {
public:
    A(int sz) { sz_ = sz; v = new B[sz_]; }
    ~A() { delete v; }
    // ...
private:
    // ...
    B * v;
    int sz_;
};
```

Взгляни на этот фрагмент кода. Представь, что я джуниор, которого ты собеседуешь в свою команду, и это мой код. Пожалуйста, будь придирчивым и попытайся дружелюбно рассказать мне о подводных камнях C++ и, возможно, научить меня, как надо писать на этом языке.

Так, с чего бы мне начать... Давайте сфокусируемся на главном.

Насчет деструктора. При использовании оператора `new[]` вы должны освобождать память оператором `delete[]`. В этом случае после вызова деструктора память будет освобождена для **каждого** объекта в массиве. Сейчас же конструктор будет вызван `sz` раз, а деструктор - только один. Если `B` выделит ресурсы, которые нужно будет освободить в этом деструкторе - то случится что-то плохое.



```
#include "B.hpp"

class A {
public:
    A(int sz) { sz_ = sz; v = new B[sz_]; }
    ~A() { delete v; }
    // ...
private:
    // ...
    B * v;
    int sz_;
};
```

Следующая вещь связана с "правилом трех". Если вам нужен деструктор - то, вероятно, вы должны реализовать или отключить конструктор копирования и оператор присваивания копированием - те, которые будут созданы по умолчанию компилятором скорее всего не подойдут.


Далее. Думаю, это менее значимая проблема - но все равно важно использовать списки инициализации для членов объекта. В нашем примере это не слишком принципиально, но когда вложенные объекты являются более сложными - имеет смысл инициализировать их через списки инициализации, вместо того, чтобы позволить им инициализироваться значениями по умолчанию, а уже **ПОТОМ** присвоить им корректные значения.

Пожалуйста, исправьте код, и я скажу вам больше...




```
#include "B.hpp"

class A {
public:
    A(int sz) { sz_ = sz; v = new B[sz_]; }
    ~A() { delete v; }
    // ...
private:
    // ...
    B * v;
    int sz_;
};
```




```
#include "B.hpp"

class A {
public:
    A(int sz) { sz_ = sz; v = new B[sz_]; }
    ~A() { delete[] v; }
    // ...
private:
    // ...
    B * v;
    int sz_;
};
```




```
#include "B.hpp"

class A {
public:
    A(int sz) { sz_ = sz; v = new B[sz_]; }
    ~A() { delete[] v; }
    // ...
private:
    // ...
    B * v;
    int sz_;
};
```



```
#include "B.hpp"


class A {
public:
    A(int sz) { sz_ = sz; v = new B[sz_]; }
    ~A() { delete[] v; }
    // ...
private:
    // ...
    B * v;
    int sz_;
};
```



```
#include "B.hpp"

class A {
public:
    A(int sz) { sz_ = sz; v = new B[sz_]; }
    ~A() { delete[] v; }
    // ...
private:

    // ...
    B * v;
    int sz_;
};
```




```
#include "B.hpp"

class A {
public:
    A(int sz) { sz_ = sz; v = new B[sz_]; }
    ~A() { delete[] v; }
    // ...
private:
    A(const A &);
    A & operator=(const A &);
    // ...
    B * v;
    int sz_;
};
```

```
#include "B.hpp"


class A {
public:
    A(int sz) { sz_ = sz; v = new B[sz_]; }
    ~A() { delete[] v; }
    // ...
private:
    A(const A &);
    A & operator=(const A &);
    // ...
    B * v;
    int sz_;
};
```

Уже лучше...



```
#include "B.hpp"

class A {
public:
    A(int sz) { sz_ = sz; v = new B[sz_]; }
    ~A() { delete[] v; }
    // ...
private:
    A(const A &);
    A & operator=(const A &);
    // ...
    B * v;
    int sz_;
};
```



```
#include "B.hpp"
```

```
class A {
```

```
public:
```

```
    A(int sz) { sz_ = sz; v = new B[sz_]; }
```

```
    virtual ~A() { delete[] v; }
```

```
    // ...
```

```
private:
```

```
    A(const A &);
```

```
    A & operator=(const A &);
```

```
    // ...
```

```
    B * v;
```

```
    int sz_;
```

```
};
```

```
#include "B.hpp"

class A {
public:
    A(int sz) { sz_ = sz; v = new B[sz_]; }
    virtual ~A() { delete[] v; }
    // ...
private:
    A(const A &);
    A & operator=(const A &);
    // ...
    B * v;
    int sz_;
};
```



Так, так, так... Попридержите коней.

Какой смысл объявлять виртуальный деструктор в таком классе? В нем нет виртуальных методов, так что в наследовании не будет смысла. Я знаю, что есть программисты, которые наследуют от неvirtуальных классов, но, думаю, они просто не понимают основную концепцию ООП. Я предлагаю удалить спецификатор `virtual` из объявления деструктора, потому что он указывает на то, что данный класс может использоваться как базовый - а это, очевидно, не так.

Вместо этого лучше добавить список инициализации.

```
#include "B.hpp"
```

```
class A {
```

```
public:
```

```
    A(int sz) { sz_ = sz; v = new B[sz_]; }
```

```
    virtual ~A() { delete[] v; }
```

```
    // ...
```

```
private:
```

```
    A(const A &);
```

```
    A & operator=(const A &);
```

```
    // ...
```


```
    B * v;
```

```
    int sz_;
```

```
};
```


```
#include "B.hpp"

class A {
public:
    A(int sz) { sz_ = sz; v = new B[sz_]; }
    ~A() { delete[] v; }
    // ...
private:
    A(const A &);
    A & operator=(const A &);
    // ...
    B * v;
    int sz_;
};
```



```
#include "B.hpp"

class A {
public:
    A(int sz) { sz_ = sz; v = new B[sz_]; }
    ~A() { delete[] v; }
    // ...
private:
    A(const A &);
    A & operator=(const A &);
    // ...
    B * v;
    int sz_;
};
```




```
#include "B.hpp"


class A {
public:
    A(int sz) : sz_(sz) { sz_ = sz; v = new B[sz_]; }
    ~A() { delete[] v; }
    // ...
private:
    A(const A &);
    A & operator=(const A &);
    // ...
    B * v;
    int sz_;
};
```

```
#include "B.hpp"

class A {
public:
    A(int sz) : sz_(sz) { v = new B[sz_]; }
    ~A() { delete[] v; }
    // ...
private:
    A(const A &);
    A & operator=(const A &);
    // ...
    B * v;
    int sz_;
};
```

```
#include "B.hpp"

class A {
public:
    A(int sz) : sz_(sz) { v = new B[sz_]; }
    ~A() { delete[] v; }
    // ...
private:
    A(const A &);
    A & operator=(const A &);
    // ...
    B * v;
    int sz_;
};
```



```
#include "B.hpp"

class A {
public:
    A(int sz) : sz_(sz), v(new B[sz_]) { v = new B[sz_]; }
    ~A() { delete[] v; }
    // ...
private:
    A(const A &);
    A & operator=(const A &);
    // ...
    B * v;
    int sz_;
};
```

```
#include "B.hpp"

class A {
public:
    A(int sz) : sz_(sz), v(new B[sz_]) {}
    ~A() { delete[] v; }
    // ...
private:
    A(const A &);
    A & operator=(const A &);
    // ...
    B * v;
    int sz_;
};
```

```
#include "B.hpp"

class A {
public:
    A(int sz) : sz_(sz), v(new B[sz_]) {}
    ~A() { delete[] v; }
    // ...
private:
    A(const A &);
    A & operator=(const A &);
    // ...
    B * v;
    int sz_;
};
```

Теперь у нас есть список инициализации...



```
#include "B.hpp"

class A {
public:
    A(int sz) : sz_(sz), v(new B[sz_]) {}
    ~A() { delete[] v; }
    // ...
private:
    A(const A &);
    A & operator=(const A &);
    // ...
    B * v;
    int sz_;
};
```



Ой... Вы видите проблему, которую мы создали?

Вы компилируете с ключом `-Wall`? Нужно также подумать о ключах `-Wextra` `-pedantic` и `-Wffc++`

Без этих ключей вы можете не заметить здесь ошибку. Но с ними компилятор сразу расскажет вам обо всем...

Хороший подход - всегда записывать инициализируемые объекты в том порядке, в котором они определены. В нашем случае во время инициализации `v(new B[sz_])` значение `sz_` еще не определено, ведь `sz_` инициализируется значением `sz` **только после этого**. Вообще, эта ошибка часто встречается к коду на C++.

```
#include "B.hpp"

class A {
public:
    A(int sz) : sz_(sz), v(new B[sz_]) {}
    ~A() { delete[] v; }
    // ...
private:
    A(const A &);
    A & operator=(const A &);
    // ...
    B * v;
    int sz_;
};
```





```
#include "B.hpp"

class A {
public:
    A(int sz) : sz_(sz), v(new B[sz]) {}
    ~A() { delete[] v; }
    // ...
private:
    A(const A &);
    A & operator=(const A &);
    // ...
    B * v;
    int sz_;
};
```




```
#include "B.hpp"
```

```
class A {  
public:  
    A(int sz) : sz_(sz), v(new B[sz]) {}  
    ~A() { delete[] v; }  
    // ...  
private:  
    A(const A &);  
    A & operator=(const A &);  
    // ...  
    B * v;  
    int sz_;  
};
```



```
#include "B.hpp"
```

```
class A {  
public:  
    A(int sz) : v(new B[sz]), sz_(sz) {}  
    ~A() { delete[] v; }  
    // ...  
private:  
    A(const A &);  
    A & operator=(const A &);  
    // ...  
    B * v;  
    int sz_;  
};
```



```
#include "B.hpp"

class A {
public:
    A(int sz) : v(new B[sz]), sz_(sz) {}
    ~A() { delete[] v; }
    // ...
private:
    A(const A &);
    A & operator=(const A &);
    // ...
    B * v;
    int sz_;
};
```



Теперь все стало гораздо лучше! Что же мы еще можем сделать? Наверное, я хотела бы добавить пару мелких замечаний...

"Голые" указатели в C++ коде обычно являются плохим знаком. Хорошие разработчики стараются избегать их. В нашем случае, кажется, `v` является чем-то вроде вектора из STL или типа того.

Похоже, вы смешиваете разные нотации для приватных переменных класса, но пока над кодом работаете только вы - я думаю, это ваше личное дело. Я полагаю, постфикс `_` вполне подходит для этого, как и префикс `m_`, но вы должны избегать префикса `__`, так как он используется в именах, зарезервированных в стандарте C, POSIX и/или компиляторах.

В каких аспектах она ориентируется лучше большинства кандидатов?



- связи между C и C++
- вариантах полиморфизма
- грамотной инициализации объектов
- "правиле трех"
- операторах `new[]` и `delete[]`
- общеизвестных нотациях именованя

Давайте поговорим о:



Времени жизни объектов
Правиле трех
Таблице виртуальных методов (vtable)

Грамотная инициализация объектов

Присваивание - не то же самое, что инициализация.

```
struct A
{
    A() { puts("A()"); }
    A(int v) { puts("A(int)"); }
    ~A() { puts("~A()"); }
};

struct X
{
    X(int v) { a=v; }
    X(long v) : a(v) { }
    A a;
};

int main()
{
    puts("bad style");
    { X slow(int(2)); }
    puts("good style");
    { X fast(long(2)); }
}
```

```
bad style
A()
A(int)
~A()
~A()
good style
A(int)
~A()
```

Время жизни объектов

Основной принцип C++ по работе с объектами заключается в том, что когда время жизни объекта заканчивается, выполняются вызовы деструкторов подобъектов, причем в обратном порядке по сравнению с операциями их инициализации.

```
struct A
{
    A() { puts("A()"); }
    ~A() { puts("~A()"); }
};

struct B
{
    B() { puts("B()"); }
    ~B() { puts("~B()"); }
};

struct C
{
    A a;
    B b;
};
```

```
int main()
{
    C obj;
}
```

```
A()
B()
~B()
~A()
```


Время жизни объектов

Основной принцип C++ по работе с объектами заключается в том, что когда время жизни объекта заканчивается, выполняются вызовы деструкторов подобъектов, причем в обратном порядке по сравнению с операциями их инициализации.

```
struct A
{
    A() : id(count++)
    {
        printf("A(%d)\n", id);
    }
    ~A()
    {
        printf("~A(%d)\n", id);
    }
    int id;
    static int count;
};
```

```
int main()
{
    A array[4];
}
```

```
A (0)
A (1)
A (2)
A (3)
~A (3)
~A (2)
~A (1)
~A (0)
```

Время жизни объектов

Основной принцип C++ по работе с объектами заключается в том, что когда время жизни объекта заканчивается, выполняются вызовы деструкторов подобъектов, причем в обратном порядке по сравнению с операциями их инициализации.

```
struct A
{
    A() : id(count++)
    {
        printf("A(%d)\n", id);
    }
    ~A()
    {
        printf("~A(%d)\n", id);
    }
    int id;
    static int count;
};
```

```
int main()
{
    A * array = new A[4];
    delete[] array;
}
```

```
A (0)
A (1)
A (2)
A (3)
~A (3)
~A (2)
~A (1)
~A (0)
```

```
int main()
{
    A * array = new A[4];
    delete array;
}
```

```
A (0)
A (1)
A (2)
A (3)
~A (0)
```

Правило трех

Если класс или структура определяет деструктор, конструктор копирования или оператор присваивания копированием, то они должны явным образом определить все три этих метода.

```
class wibble_ptr {
public:
    wibble_ptr()
        : ptr(new wibble), count(new int(1)) {
    }
    wibble_ptr(const wibble_ptr & other)
        : ptr(other.ptr), count(other.count) {
        (*count)++;
    }
    wibble_ptr & operator=(const wibble_ptr & rhs) {
        wibble_ptr copy(rhs);
        swap(copy);
        return *this;
    }
    ~wibble_ptr() {
        if (--(*count) == 0)
            delete ptr;
    }
    ...
private:
    wibble * ptr;
    int * count;
};
```

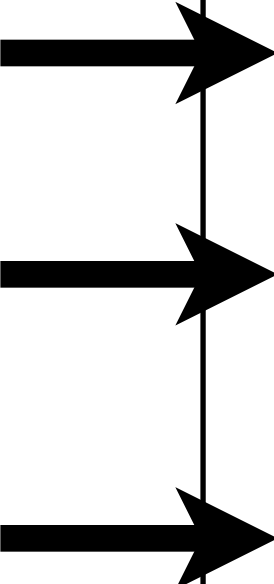


Таблица виртуальных методов (vtable)

```
struct base
{
    virtual void f();
    virtual void g();
    int a,b;
};

struct derived : base
{
    virtual void g();
    virtual void h();
    int c;
};

void poly(base * ptr)
{
    ptr->f();
    ptr->g();
}

int main()
{
    poly(&base());
    poly(&derived());
}
```

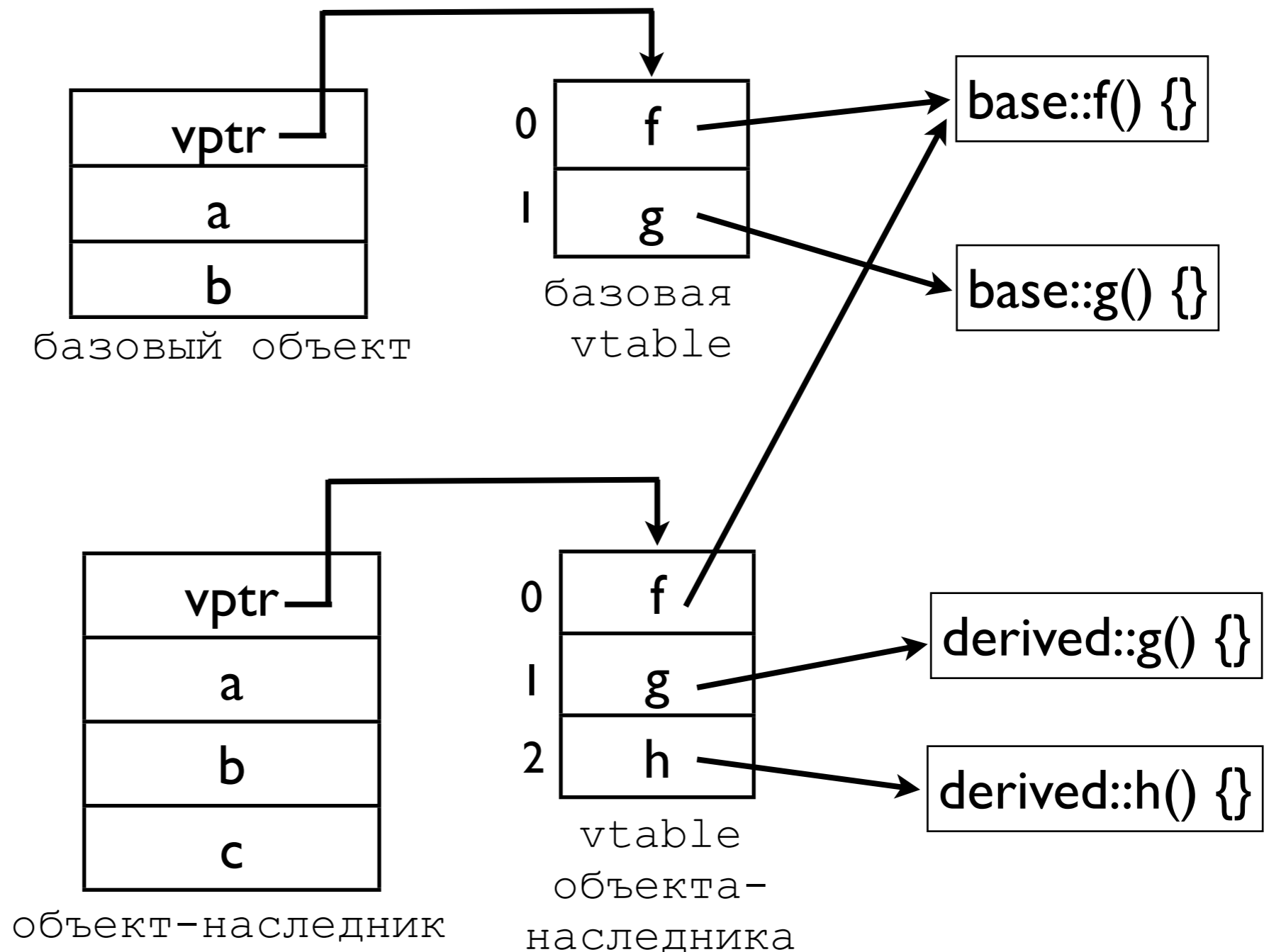


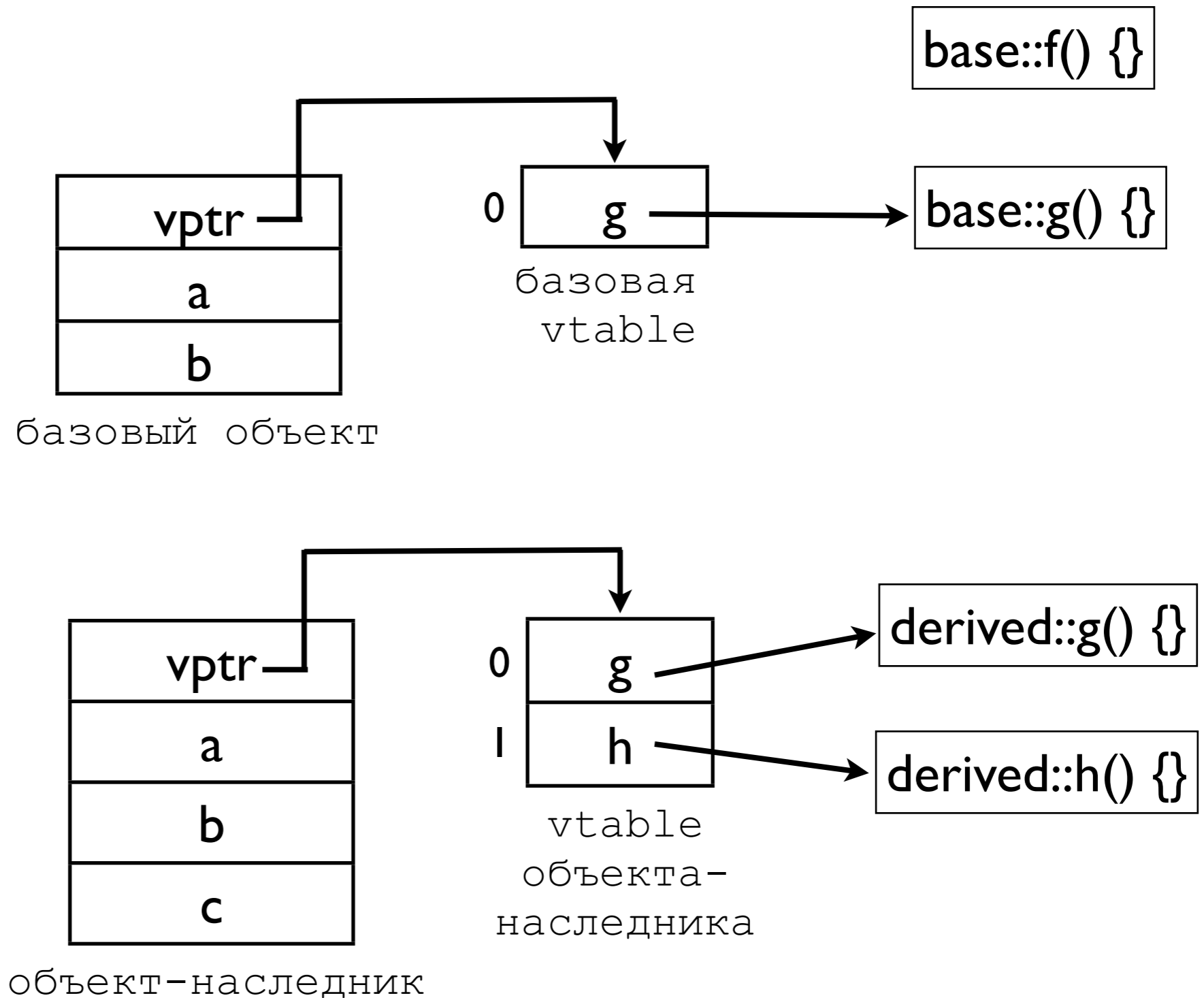
Таблица виртуальных методов (vtable)

```
struct base
{
    void f();
    virtual void g();
    int a,b;
};

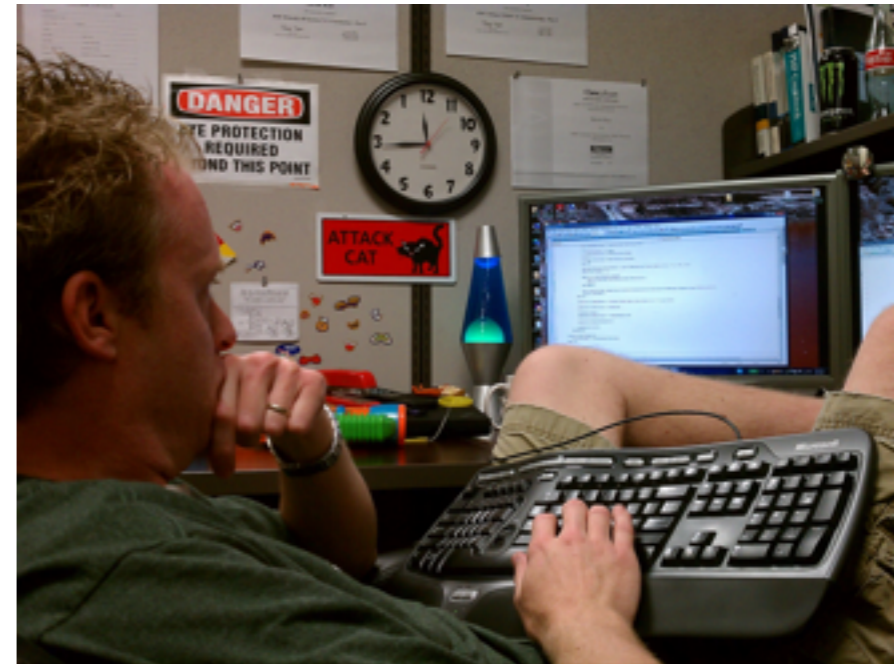
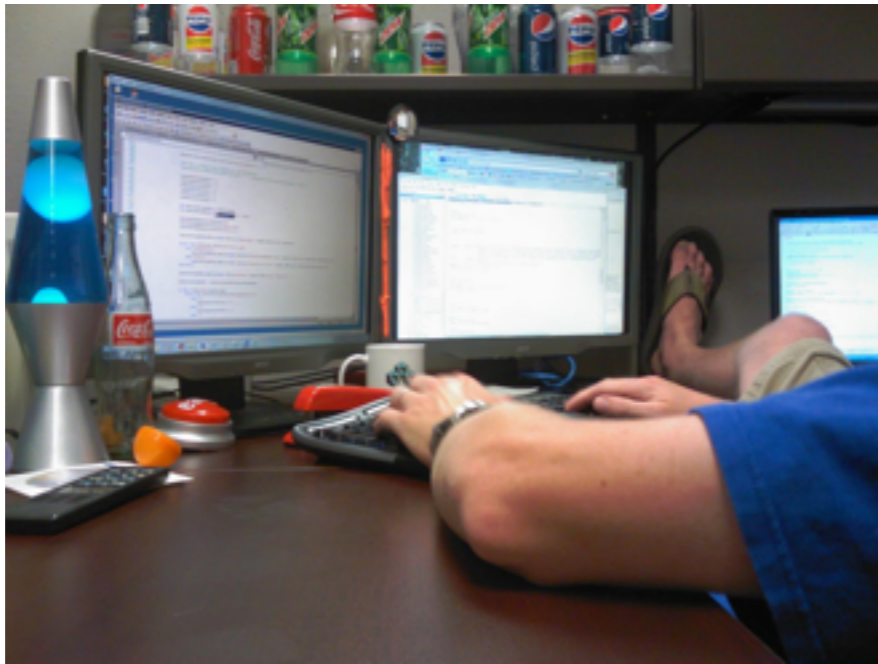
struct derived : base
{
    virtual void g();
    virtual void h();
    int c;
};

void poly(base * ptr)
{
    ptr->f();
    ptr->g();
}

int main()
{
    poly(&base());
    poly(&derived());
}
```



Стала бы жизнь лучше, если бы больше ваших коллег глубоко знали язык, на котором они программируют?



Мы не считаем, что **всем** C/C++ разработчикам вашей компании требуются такие глубокие познания. Но вы определенно должны иметь критическую массу таких людей, которые заботятся о своем профессионализме, не перестают развиваться и желают лучше изучить язык, который используют.

Давайте вернемся к нашим кандидатам...



© www.ClipProject.info



© www.ClipProject.info

Так в чем же разница между этими кандидатами?
В их текущем знании языка? Нет!

В их отношении к обучению!



Когда ты последний раз посещал курсы по программированию?

В смысле? Я изучал программирование в университете, а потом получал практический опыт на работе. Разве нужно еще что-то делать?

Какие книги по программированию ты прочел в последнее время?

Книги? Я не читаю книги. Если что-то надо узнать - можно просто погуглить.

Ты общаешься о техниках программирования со своими коллегами?

Они тупицы, что я могу от них узнать...

Похоже, ты много знаешь о С и С++. Как ты пришла к этому?

Я узнаю что-то новое каждый день, и мне это очень нравится.

Иногда я читаю треды о С/С++ на stack overflow, comp.lang.c и comp.lang.c++

Я являюсь участником местной группы разработчиков на С/С++, иногда мы устраиваем встречи.

Я читаю книги. Много книг. Вы знаете, что у Джеймса Греннинга недавно вышла замечательная книга "Test Driven Development for Embedded C"?

Надо сказать, иногда я участвую в WG14 и WG21.

Я состою в ACCU, потому что забочусь о своем профессионализме. Я читаю Overload, C Vu и дискуссии на accu-general

И каждый раз, когда у меня есть возможность, я записываюсь на обучающие курсы по С и С++. Далеко не всегда я узнаю что-то новое от преподавателя, но часто - от других учащихся.

Но, думаю, лучший источник моих знаний - тесное общение с коллегами в процессе работы над проектами.



Мы поговорили о:

- компиляторах и линкерах
- объявлении и определении
- фреймах активации
- сегментации памяти
- выравнивании памяти
- точках следования
- порядке оценки выражений
- неопределенном и неупреждаемом поведении
- оптимизациях от компилятора
- немного о C++
- грамотной инициализации объектов
- времени жизни объектов
- таблице виртуальных методов
- правиле трех
- ...а также о профессионализме и отношении к обучению



Эм...

Да?

Мне действительно нравится программировать, но теперь я понимаю, что еще далек от того, чтобы стать профессионалом. Вы можете дать мне советы, как начать углубленно изучать С и С++?

Прежде всего ты должен осознать, что программирование неразрывно связано с постоянным обучением. Неважно, сколько ты знаешь, всегда есть то, чему стоит поучиться. Следующее, что нужно понять - программирование является командной работой, и ты должен трудиться вместе со своими коллегами. Думай о программировании как о виде спорта, где никто не может выиграть матч в одиночку.

Хорошо, я поразмыслю над этим...

Также возьми за привычку изучать код, сгенерированный транслятором. Это позволит узнать много интересных вещей. Используй отладчик в пошаговом режиме и наблюдай, как используется память и как процессор выполняет те или иные инструкции.



Какие книги, онлайн-ресурсы, обучающие курсы и конференции по C/C++ вы можете посоветовать?

Для ознакомления с современными методиками разработки ПО я рекомендую "Test-Driven Development for Embedded C", написанную Джеймсом Грэннингом. Для глубокого изучения C хорошо подойдет "Expert C programming" Петера ван дер Линдена. Она была написана 20 лет назад, но ее материал до сих пор актуален. По C++ я советую начать с "Effective C++" Скотта Майерса и "C++ coding standards", написанной Гербом Саттером и Андреем Александреску.

Кроме того, при каждой возможности записывайся на учебные курсы. Если ты настроен серьезно - то найдешь чему поучиться и у преподавателя, и других обучающихся.

И, наконец, я бы советовал тебе вступить в одно из сообществ программистов. В частности, я рекомендую ACCU (accu.org), они фокусируются именно на C и C++. Ты знал, что каждую весну они проводят в Оксфорде недельную конференцию, на которую съезжаются разработчики со всего мира? Может, увидимся там в апреле?

Спасибо!

Удачи!





<http://www.sharpshirter.com/assets/images/sharkpunchashgrey1.jpg>