'printf' vs. 'cout' in C++

What is the difference between `printf()` and `cout` in C++?

c++ c printf cout

edited May 22 at 21:27

Peter Mortensen
11.3k ● 16 ● 78 ● 110

asked May 20 '10 at 9:42

hero
1,116 ● 2 ● 8 ● 6

protected by Yu Hao Feb 3 '15 at 9:05

This question is protected to prevent "thanks!", "me too!", or spam answers by new users. To answer it, you must have earned at least 10 reputation on this site (the association bonus does not count).

28 The question in its nature is equivalent to "What is the difference between a car and an airplane?". – AnT May 20 '10 at 18:14

same wonder here!! – Muhammad Hewedy May 21 '10 at 2:20

@Amarghosh perhaps because of rep points? – AJMansfield Nov 16 '12 at 16:08

As a courtesy please consider accepting one of the provided answers an an accepted one if it helped you solve your problem. – RBT Oct 31 '16 at 4:07

16 Answers

From the C++ FAQ:

> **[15.1] Why should I use `<iostream>` instead of the traditional `<cstdio>`?**
>
> Increase type safety, reduce errors, allow extensibility, and provide inheritability.
>
> `printf()` is arguably not broken, and `scanf()` is perhaps livable despite being error prone, however both are limited with respect to what C++ I/O can do. C++ I/O (using `<<` and `>>`) is, relative to C (using `printf()` and `scanf()`):
>
> - More type-safe: With `<iostream>`, the type of object being I/O'd is known statically by the compiler. In contrast, `<cstdio>` uses "%" fields to figure out the types dynamically.
> - Less error prone: With `<iostream>`, there are no redundant "%" tokens that have to be consistent with the actual objects being I/O'd. Removing redundancy removes a class of errors.
> - Extensible: The C++ `<iostream>` mechanism allows new user-defined types to be I/O'd without breaking existing code. Imagine the chaos if everyone was simultaneously adding new incompatible "%" fields to `printf()` and `scanf()`?!
> - Inheritable: The C++ `<iostream>` mechanism is built from real classes such as `std::ostream` and `std::istream`. Unlike `<cstdio>`'s `FILE*`, these are real classes and hence inheritable. This means you can have other user-defined things that look and act like streams, yet that do whatever strange and wonderful things you want. You automatically get to use the zillions of lines of I/O code written by users you don't even know, and they don't need to know about your "extended stream" class.

On the other hand, `printf` is significantly faster, which may justify using it in preference to `cout` in *very* specific and limited cases. Always profile first. (See, for example, http://programming-designs.com/2009/02/c-speed-test-part-2-printf-vs-cout/)

edited Sep 19 '15 at 11:51

community wiki
8 revs, 8 users 26%
Marcelo Cantos

1 On the other other hand, there's the FastFormat library (fastformat.org), offering type-safety, expressivity and performance at once. (Not that I tried it yet...) – xtofl May 20 '10 at 10:07

How does such a poorly presented piece of copy-pasting, with no consideration for escaping (hence a rash of inexplicable omissions from the text) get *nine* upvotes? – Marcelo Cantos May 20 '10 at 11:15

2 There's also Boost.Format, wonder how the two compare. – Matthieu M. May 20 '10 at 11:21

@Marcelo Cantos: I suspect a backlash to the accepted answer. – Gorpik May 20 '10 at 11:44

@xtofl yeah, I looked at it a while ago. Seemed promising, but rated dangerously high on my annoy-o-meter when I realized it depends on environment variables, which is one thing I really do not care to set up just for my code to compile. – jalf May 20 '10 at 11:50

@jalf: I knew there should have been something.... would you mind adding that to this, old, question? stackoverflow.com/questions/446276/… – xtofl May 20 '10 at 12:15

2 @Marcelo probably because it's a good summary, with everything cited. The formatting... yeah, that's pretty bad. I should have fixed that myself, but it appears that others (yourself included) took care of it, which, of course, is more constructive than just whining. – Mikeage May 21 '10 at 7:59

1 As of late `printf()` is also supposed to be extensible. See "printf hooks" at udrepper.livejournal.com/20948.html – Maxim Egorushkin Nov 25 '11 at 8:32

2 @MaximYegorushkin: Standard `printf` has no such ability. Non-portable library mechanisms are hardly on the same level as fully standardized extensibility of iostreams. – Ben Voigt Jan 15 '14 at 18:09

2 "On the other hand, printf is significantly faster" printf is also cleaner and easier to use, which is why i avoid cout when possible. – FluorescentGreen5 Nov 11 '15 at 13:09

"On the other hand, printf is significantly faster, which may justify using it in preference to cout". Is this true? The equivalent claim with cin/scanf is *not* true. See stackoverflow.com/a/12762166/462335 – nibot May 30 at 21:17

I'm surprised that everyone in this question claims that `std::cout` is way better than `printf`, even if the question just asked for differences. Now, there is a difference - `std::cout` is C++, and `printf` is C (however, you can use it in C++, just like *almost* anything else from C). Now, I'll be honest here; both `printf` and `std::cout` have their advantages.

Disclaimer: I'm more experienced with C than C++, so if there is a problem with my answer, feel free to edit or comment.

Real differences

Extensibility

`std::cout` is extensible. I know that people will say that `printf` is extensible too, but such extension is not mentioned in the C standard (so you would have to use non-standard features - but not even common non-standard feature exists), and such extensions are one letter (so it's easy to conflict with an already-existing format).

Unlike `printf`, `std::cout` depends completely on operator overloading, so there is no issue with custom formats - all you do is define a subroutine taking `std::ostream` as the first argument and your type as second. As such, there are no namespace problems - as long you have a class (which isn't limited to one character), you can have working `std::ostream` overloading for it.

However, I doubt that many people would want to extend `ostream` (to be honest, I rarely saw such extensions, even if they are easy to make). However, it's here if you need it.

Syntax

As it could be easily noticed, both `printf` and `std::cout` use different syntax. `printf` uses standard function syntax using pattern string and variable-length argument lists. Actually, `printf` is a reason why C has them - `printf` formats are too complex to be usable without them. However, `std::cout` uses a different API - the `operator <<` API that returns itself.

Generally, that means the C version will be shorter, but in most cases it won't matter. The difference is noticeable when you print many arguments. If you have to write something like `Error 2: File not found.`, assuming error number, and its description is placeholder, the code would look like this. Both examples work identically (well, sort of, `std::endl` actually flushes the buffer).

```
printf("Error %d: %s.\n", id, errors[id]);
std::cout << "Error " << id << ": " << errors[id] << "." << std::endl;
```

While this doesn't appear too crazy (it's just two times longer), things get more crazy when you actually format arguments, instead of just printing them. For example, printing of something like `0x0424` is just crazy. This is caused by `std::cout` mixing state and actual values. I never saw a language where something like `std::setfill` would be a type (other than C++, of course). `printf` clearly separates arguments and actual type. I really would prefer to maintain the `printf` version of it (even if it looks kind of cryptic) compared to `iostream` version of it (as it contains too much noise).

```
printf("0x%04x\n", 0x424);
std::cout << "0x" << std::hex << std::setfill('0') << std::setw(4) << 0x424 <<
std::endl;
```

Translation

This is where the real advantage of `printf` lies. The `printf` format string is well... a string. That makes it really easy to translate, compared to `operator <<` abuse of `iostream`. Assuming that the `gettext()` function translates, and you want to show `Error 2: File not found.`, the code to get translation of the previously shown format string would look like this:

```
printf(gettext("Error %d: %s.\n"), id, errors[id]);
```

Now, let's assume that we translate to Fictionish, where the error number is after the description. The translated string would look like `%2$s oru %1$d.\n`. Now, how to do it in C++? Well, I have no idea. I guess you can make fake `iostream` which constructs `printf` that you can pass to `gettext`, or something, for purposes of translation. Of course, `$` is not C standard, but it's so common that it's safe to use in my opinion.

Not having to remember/look-up specific integer type syntax

C has lots of integer types, and so does C++. `std::cout` handles all types for you, while `printf` requires specific syntax depending on an integer type (there are non-integer types, but the only non-integer type you will use in practice with `printf` is `const char *` (C string, can be obtained using `to_c` method of `std::string`)). For instance, to print `size_t`, you need to use `%zd`, while `int64_t` will require using `%"PRIu64"d`. The tables are available at http://en.cppreference.com/w/cpp/io/c/fprintf and http://en.cppreference.com/w/cpp/types/integer.

You can't print the NUL byte, `\0`

Because `printf` uses C strings as opposed to C++ strings, it cannot print NUL byte without specific tricks. In certain cases it's possible to use `%c` with `'\0'` as an argument, although that's clearly a hack.

Differences nobody cares about

Performance

Update: It turns out that `iostream` is so slow that it's usually slower than your hard drive (if you redirect your program to file). Disabling synchronization with `stdio` may help, if you need to output lots of data. If the performance is a real concern (as opposed to writing several lines to STDOUT), just use `printf`.

Everyone thinks that they care about performance, but nobody bothers to measure it. My answer is that I/O is bottleneck anyway, no matter if you use `printf` or `iostream`. I think that `printf` *could* be faster from a quick look into assembly (compiled with clang using the `-O3` compiler option). Assuming my error example, `printf` example does way fewer calls than the `cout` example. This is `int main` with `printf`:

```
main:                                   @ @main
@ BB#0:
        push    {lr}
        ldr     r0, .LCPI0_0
        ldr     r2, .LCPI0_1
        mov     r1, #2
        bl      printf
        mov     r0, #0
        pop     {lr}
        mov     pc, lr
        .align  2
@ BB#1:
```

You can easily notice that two strings, and `2` (number) are pushed as `printf` arguments. That's about it; there is nothing else. For comparison, this is `iostream` compiled to assembly. No, there is no inlining; every single `operator <<` call means another call with another set of arguments.

```
main:                                   @ @main
@ BB#0:
        push    {r4, r5, lr}
        ldr     r4, .LCPI0_0
        ldr     r1, .LCPI0_1
        mov     r2, #6
        mov     r3, #0
        mov     r0, r4
        bl
_ZSt16__ostream_insertIcSt11char_traitsIcEERSt13basic_ostreamIT_T0_ES6_PKS3_l
        mov     r0, r4
        mov     r1, #2
        bl      _ZNSolsEi
        ldr     r1, .LCPI0_2
        mov     r2, #2
        mov     r3, #0
        mov     r4, r0
        bl
_ZSt16__ostream_insertIcSt11char_traitsIcEERSt13basic_ostreamIT_T0_ES6_PKS3_l
        ldr     r1, .LCPI0_3
        mov     r0, r4
        mov     r2, #14
        mov     r3, #0
        bl
_ZSt16__ostream_insertIcSt11char_traitsIcEERSt13basic_ostreamIT_T0_ES6_PKS3_l
        ldr     r1, .LCPI0_4
        mov     r0, r4
        mov     r2, #1
        mov     r3, #0
        bl
_ZSt16__ostream_insertIcSt11char_traitsIcEERSt13basic_ostreamIT_T0_ES6_PKS3_l
        ldr     r0, [r4]
        sub     r0, r0, #24
        ldr     r0, [r0]
        add     r0, r0, r4
        ldr     r5, [r0, #240]
```

```
        cmp     r5, #0
        beq     .LBB0_5
@ BB#1:                                 @
%_ZSt13__check_facetISt5ctypeIcEERKT_PS3_.exit
        ldrb    r0, [r5, #28]
        cmp     r0, #0
        beq     .LBB0_3
@ BB#2:
        ldrb    r0, [r5, #39]
        b       .LBB0_4
.LBB0_3:
        mov     r0, r5
        bl      _ZNKSt5ctypeIcE13_M_widen_initEv
        ldr     r0, [r5]
        mov     r1, #10
        ldr     r2, [r0, #24]
        mov     r0, r5
        mov     lr, pc
        mov     pc, r2
.LBB0_4:                                @ %_ZNKSt5ctypeIcE5widenEc.exit
        lsl     r0, r0, #24
        asr     r1, r0, #24
        mov     r0, r4
        bl      _ZNSo3putEc
        bl      _ZNSo5flushEv
        mov     r0, #0
        pop     {r4, r5, lr}
        mov     pc, lr
.LBB0_5:
        bl      _ZSt16__throw_bad_castv
        .align  2
@ BB#6:
```

However, to be honest, this means nothing, as I/O is the bottleneck anyway. I just wanted to show that `iostream` is not faster because it's "type safe". Most C implementations implement `printf` formats using computed goto, so the `printf` is as fast as it can be, even without compiler being aware of `printf` (not that they aren't - some compilers can optimize `printf` in certain cases - constant string ending with `\n` is usually optimized to `puts`).

Inheritance

I don't know why you would want to inherit `ostream`, but I don't care. It's possible with `FILE` too.

```
class MyFile : public FILE {}
```

Type safety

True, variable length argument lists have no safety, but that doesn't matter, as popular C compilers can detect problems with `printf` format string if you enable warnings. In fact, Clang can do that without enabling warnings.

```
$ cat safety.c

#include <stdio.h>

int main(void) {
    printf("String: %s\n", 42);
    return 0;
}

$ clang safety.c

safety.c:4:28: warning: format specifies type 'char *' but the argument has type
'int' [-Wformat]
    printf("String: %s\n", 42);
                    ~~     ^~
                    %d
1 warning generated.
$ gcc -Wall safety.c
safety.c: In function ‘main’:
safety.c:4:5: warning: format ‘%s’ expects argument of type ‘char *’, but argument
2 has type ‘int’ [-Wformat=]
     printf("String: %s\n", 42);
     ^
```

edited May 25 at 17:32 — Peter Mortensen — 11.3k ● 16 ● 78 ● 110

answered Nov 27 '13 at 9:23 — xfix — 5,594 ● 1 ● 25 ● 52

7 You say I/O is the bottleneck anyway. Obviously you never tested that assumption. I quote myself: "On the other hand, the iostreams version, at 75.3 MB/s, can't buffer data fast enough to keep up with a hard disk. That's bad, and it's not even doing any real work yet. I don't think I have too high expectations when I say my I/O library should be able to be able to saturate my disk controller." – Ben Voigt Jan 15 '14 at 18:15

1 @BenVoigt: I admit, I try to avoid C++ when possible. I tried using it a lot, but it was more annoying, and less maintainable than other programming language I used. This is yet another reason for me to avoid C++ - this isn't even fast (it's not even iostream - entire C++ library is slow in most implementations, perhaps with exception for `std::sort`, which is somehow surprisingly fast compared to `qsort` (2 times), at cost of executable size). – xfix Jan 15 '14 at 20:36

2 No one here has mentioned issues in parallel environment when using cout. – Nicholas Hamilton May 19 '14 at 23:35

4 Your performance argument makes no sense whatsoever. More assembly in your program doesn't mean that program will be slower, because you're *not* accounting for all the code that makes the printf function,

which is a lot of code. In my opinion, it is possible to optimize cout with << operator a lot better than the printf, because compiler can make better sense of variables and formatting. – Ignas2526 Jan 26 '15 at 12:22

3 I like a lot of things about this answer, but perhaps my favorite part is "Everyone thinks that they care about performance, but nobody bothers to measure it." – Kyle Strand Oct 15 '15 at 17:48

|

And I quote:

> In high level terms, the main differences are type safety (cstdio doesn't have it), performance (most iostreams implementations are slower than the cstdio ones) and extensibility (iostreams allows custom output targets and seamless output of user defined types).

answered May 20 '10 at 9:45

Kyle Rozendo
20.1k ● 5 ● 62 ● 106

Especially on unix where with POSIX you never know what size one of the typedefs really has so you need lots of casts or as 99% of the programs you just risk it with %d. Took even a long time before %z came with C99. But for time_t/off_t the quest for the correct format instruction continues. – Lothar Feb 5 '15 at 21:40

People often claim that `printf` is much faster. This is largely a myth. I just tested it, with the following results:

```
cout with only endl                    1461.310252 ms
cout with only '\n'                     343.080217 ms
printf with only '\n'                    90.295948 ms
cout with string constant and endl     1892.975381 ms
cout with string constant and '\n'      416.123446 ms
printf with string constant and '\n'    472.073070 ms
cout with some stuff and endl          3496.489748 ms
cout with some stuff and '\n'          2638.272046 ms
printf with some stuff and '\n'        2520.318314 ms
```

Conclusion: if you want only newlines, use `printf`; otherwise, `cout` is almost as fast, or even faster. More details can be found on my blog.

To be clear, I'm not trying to say that `iostream`s are always better than `printf`; I'm just trying to say that you should make an informed decision based on real data, not a wild guess based on some common, misleading assumption.

Update: Here's the full code I used for testing. Compiled with `g++` without any additional options (apart from `-lrt` for the timing).

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

class TimedSection {
    char const *d_name;
    timespec d_start;
    public:
        TimedSection(char const *name) :
            d_name(name)
        {
            clock_gettime(CLOCK_REALTIME, &d_start);
        }
        ~TimedSection() {
            timespec end;
            clock_gettime(CLOCK_REALTIME, &end);
            double duration = 1e3 * (end.tv_sec - d_start.tv_sec) +
                              1e-6 * (end.tv_nsec - d_start.tv_nsec);
            std::cerr << d_name << '\t' << std::fixed << duration << " ms\n";
        }
};

int main() {
    const int iters = 10000000;
    char const *text = "01234567890123456789";
    {
        TimedSection s("cout with only endl");
        for (int i = 0; i < iters; ++i)
            std::cout << std::endl;
    }
    {
        TimedSection s("cout with only '\\n'");
        for (int i = 0; i < iters; ++i)
            std::cout << '\n';
    }
    {
        TimedSection s("printf with only '\\n'");
        for (int i = 0; i < iters; ++i)
            printf("\n");
    }
    {
        TimedSection s("cout with string constant and endl");
        for (int i = 0; i < iters; ++i)
            std::cout << "01234567890123456789" << std::endl;
```

```
    }
    {
        TimedSection s("cout with string constant and '\\n'");
        for (int i = 0; i < iters; ++i)
            std::cout << "01234567890123456789\n";
    }
    {
        TimedSection s("printf with string constant and '\\n'");
        for (int i = 0; i < iters; ++i)
            printf("01234567890123456789\n");
    }
    {
        TimedSection s("cout with some stuff and endl");
        for (int i = 0; i < iters; ++i)
            std::cout << text << "01234567890123456789" << i << std::endl;
    }
    {
        TimedSection s("cout with some stuff and '\\n'");
        for (int i = 0; i < iters; ++i)
            std::cout << text << "01234567890123456789" << i << '\n';
    }
    {
        TimedSection s("printf with some stuff and '\\n'");
        for (int i = 0; i < iters; ++i)
            printf("%s0123456789012345678 9%i\n", text, i);
    }
}
```

edited May 20 '10 at 12:23

answered May 20 '10 at 11:35

Thomas

97.1k ● 28 ● 222 ● 335

2 In your scores printf beats cout easily (majority cases). I wonder why you recommend using cout when it comes to perf. Though I agree perf is not too different in realistic cases.. – mishal153 May 20 '10 at 12:07

2 @mishal153: I'm just trying to say that the performance is not too different, so the commonly-heard advice of "never use cout because it's waaay slow" is plain stupid. Note that cout has the obvious advantage of type-safety, and often readability as well. (Floating-point formatting with iostreams is horrible...) – Thomas May 20 '10 at 12:18

2 @Thomas: "reproducing the code" wouldn't necessarily reproduce your results. In order to know if your benchmarks are valid, we need to be able to look at the code. But yeah, thanks for adding the code. :) – jalf May 20 '10 at 12:30

26 The important difference between `printf()` and `std::ostream` is that **the former outputs all arguments in one single call** whereas `std::ostream` incurs a separate call for each `<<` . The test only outputs one argument and a new-line, that's why you can't see the difference. – Maxim Egorushkin Nov 25 '11 at 8:36

9 The compiler should be able to inline these calls. Also, `printf` might make a lot of calls under the covers to helper functions for various formatting specifiers... that, or it's a monstrous monolithic function. And again, because of inlining, it shouldn't make a difference in speed at all. – Thomas Nov 26 '11 at 11:05

|

One is a function that prints to stdout. The other is an object that provides several member functions and overloads of `operator<<` that print to stdout. There are many more differences that I could enumerate, but I'm not sure what you are after.

edited Feb 4 '13 at 3:51

answered May 20 '10 at 9:46

Marcelo Cantos

130k ● 27 ● 255 ● 305

For me, the real differences which would make me go for 'cout' rather than 'printf' are:

1) **<<** operator can be overloaded for my classes.

2) Output stream for cout can be easily changed to a file : (: copy paste :)

```
#include <iostream>
#include <fstream>
using namespace std;

int main ()
{
    cout << "This is sent to prompt" << endl;
    ofstream file;
    file.open ("test.txt");
    streambuf* sbuf = cout.rdbuf();
    cout.rdbuf(file.rdbuf());
    cout << "This is sent to file" << endl;
    cout.rdbuf(sbuf);
    cout << "This is also sent to prompt" << endl;
    return 0;
}
```

3) I find cout more readable, especially when we have many parameters.

One **problem** with `cout` is the formatting options. Formatting the data (precision, justificaton, etc.) in `printf` is easier.

edited Jun 28 '10 at 17:28

Peter Mortensen
11.3k • 16 • 78 • 110

answered May 20 '10 at 12:01

mishal153
812 • 1 • 15 • 32

it's nice. How can I know nobody modify global cout this way in some foreign library thread? – vp_arth Dec 29 '16 at 17:04

Two points not otherwise mentioned here that I find significant:

1) `cout` carries a lot of baggage if you're not already using the STL. It adds over twice as much code to your object file as `printf`. This is also true for `string`, and this is the major reason I tend to use my own string library.

2) `cout` uses overloaded `<<` operators, which I find unfortunate. This can add confusion if you're also using the `<<` operator for its intended purpose (shift left). I personally don't like to overload operators for purposes tangential to their intended use.

Bottom line: I'll use `cout` (and `string`) if I'm already using the STL. Otherwise, I tend to avoid it.

answered Dec 5 '14 at 21:53

Bill Weinman
897 • 8 • 10

With primitives, it probably doesn't matter entirely which one you use. I say where it gets usefulness is when you want to output complex objects.

For example, if you have a class,

```
#include <iostream>
#include <cstdlib>

using namespace std;

class Something
{
public:
        Something(int x, int y, int z) : a(x), b(y), c(z) { }
        int a;
        int b;
        int c;

        friend ostream& operator<<(ostream&, const Something&);
};

ostream& operator<<(ostream& o, const Something& s)
{
        o << s.a << ", " << s.b << ", " << s.c;
        return o;
}

int main(void)
{
        Something s(3, 2, 1);

        // output with printf
        printf("%i, %i, %i\n", s.a, s.b, s.c);

        // output with cout
        cout << s << endl;

        return 0;
}
```

Now the above might not seem all that great, but let's suppose you have to output this in multiple places in your code. Not only that, let's say you add a field "int d." With cout, you only have to change it in once place. However, with printf, you'd have to change it in possibly a lot of places and not only that, you have to remind yourself which ones to output.

With that said, with cout, you can reduce a lot of times spent with maintenance of your code and not only that if you re-use the object "Something" in a new application, you don't really have to worry about output.

answered May 20 '10 at 18:02

Daniel
682 • 4 • 5

Also, to add about the performance thing, I'd say that you shouldn't output anything at all if your application is made for performance. Any sort of output to std is rather expensive and slow. I say you should avoid it and only output when it is absolutely necessary to do so. – Daniel May 20 '10 at 18:04

keep in mind that your class might have private members you cannot access so easily from the outside. With the output operator, you have exactly one location that needs to be friend to your class, and now you can output it anywhere, even in code you didn't know about. – hochl Aug 4 at 14:02

More differences: "printf" returns an integer value (equal to the number of characters printed) and "cout" does not return anything

And.

`cout << "y = " << 7;` is not atomic.

`printf("%s = %d", "y", 7);` is atomic.

cout performs typechecking, printf doesn't.

There's no iostream equivalent of `"% d"`

edited Dec 27 '12 at 16:43
Keith Thompson
168k ● 23 ● 232 ● 414

answered Dec 5 '12 at 13:34
skan
2,154 ● 6 ● 24 ● 48

2 `cout` doesn't return anything because it's an object, not a function. `operator<<` does return something (normally its left operand, but a false value if there's an error). And in what sense is the `printf` call "atomic"? – Keith Thompson Dec 27 '12 at 16:44

8 It is like an atomic bomb. `printf("%s\n",7);` – artless noise Mar 14 '13 at 2:25

@artlessnoise wait why segmentation fault? `%s` is ? – Abhinav Gauniyal Jan 18 '16 at 13:14

1 That is the point of the 'atomic bomb' statement. A `printf` *%s* argument must have a valid pointer to a null terminated string. The memory range '7' (a pointer) is not usually valid; a segmentation fault could be lucky. On some systems, '7' might print a lot of garbage to a console and you would have to look at it for a day before the program stops. In other words, this is a bad thing about `printf`. Static analysis tools can catch many of these issues. – artless noise Jan 18 '16 at 14:36

I'd like to point out that if you want to play with threads in C++, if you use `cout` you can get some interesting results.

Consider this code:

```
#include <string>
#include <iostream>
#include <thread>

using namespace std;

void task(int taskNum, string msg) {
    for (int i = 0; i < 5; ++i) {
        cout << "#" << taskNum << ": " << msg << endl;
    }
}

int main() {
    thread t1(task, 1, "AAA");
    thread t2(task, 2, "BBB");
    t1.join();
    t2.join();
    return 0;
}

// g++ ./thread.cpp -o thread.out -ansi -pedantic -pthread -std=c++0x
```

Now, the output comes all shuffled. It can yield different results too, try executing several times:

```
##12::  ABABAB

##12::  ABABAB

##12::  ABABAB

##12::  ABABAB

##12::  ABABAB
```

You can use `printf` to get it right, or you can use `mutex`.

```
#1: AAA
#2: BBB
#1: AAA
#2: BBB
#1: AAA
#2: BBB
#1: AAA
#2: BBB
#1: AAA
#2: BBB
```

Have fun!

edited Jul 28 at 13:34

answered Nov 17 '15 at 17:52
Apollo
561 ● 1 ● 8 ● 22

2

wtf `thread`s don't make output go nuts. I just reproduced and found both `xyz` and `ABC` in the output. There was not mangling b/w `ABC` as `ABABAB`. – Abhinav Gauniyal Jan 18 '16 at 13:10

I don't know how `cout` works with threads, but I know for sure that the code you are showing isn't the one that you used to get those outputs. Your code passes the string `"ABC"` for thread 1 and `"xyz"` for thread 2, but your output shows `AAA` and `BBB`. Please fix it, because right now it's confusing. – Fabio Turati Feb 8 '16 at 21:28

TL;DR: Always do your own research, in regard of **generated machine code size**, **performance**, **readability** and **coding time** before trusting random comments online, including this one.

I'm no expert. I just happened to overhear two co-workers talking about how we should avoid using C++ in embedded systems because of performance issues. Well, interesting enough, I did a benchmark based on a real project task.

In said task, we had to write some config to RAM. Something like:

> coffee=hot
> sugar=none
> milk=breast
> mac=AA:BB:CC:DD:EE:FF

Here's my benchmark programs (Yes, I know OP asked about printf(), not fprintf(). Try to capture the essence and by the way, OP's link points to fprintf() anyway.)

C program:

```
char coffee[10], sugar[10], milk[10];
unsigned char mac[6];

/* Initialize those things here. */

FILE * f = fopen("a.txt", "wt");

fprintf(f, "coffee=%s\nsugar=%s\nmilk=%s\nmac=%02X:%02X:%02X:%02X:%02X:%02X\n",
coffee, sugar, milk, mac[0], mac[1],mac[2],mac[3],mac[4],mac[5]);

fclose(f);
```

C++ program:

```
//Everything else is identical except:

std::ofstream f("a.txt", std::ios::out);

f << "coffee=" << coffee << "\n";
f << "sugar=" << sugar << "\n";
f << "milk=" << milk << "\n";
f << "mac=" << (int)mac[0] << ":"
    << (int)mac[1] << ":"
    << (int)mac[2] << ":"
    << (int)mac[3] << ":"
    << (int)mac[4] << ":"
    << (int)mac[5] << endl;
f.close();
```

I did my best to polish them before I looped them both 100,000 times. Here are the results:

C program:

```
real    0m 8.01s
user    0m 2.37s
sys     0m 5.58s
```

C++ program:

```
real    0m 6.07s
user    0m 3.18s
sys     0m 2.84s
```

Object file size:

```
C   - 2,092 bytes
C++ - 3,272 bytes
```

Conclusion: On my very specific **platform**, with a very specific **processor**, running a very specific version of **Linux kernel**, to run a program which is compiled with a very specific version of **GCC**, in order to accomplish a very specific **task**, I would say the C++ approach is more suitable because it runs significantly faster and provide much better readability. On the other hand, C offers small footprint, in my opinion, means nearly nothing because program size is not of our concern.

Remeber, YMMV.

answered Apr 1 at 5:08

Wesley
304 • 3 • 16

I would like say that extensibility lack of `printf` is not entirely true:
In C, it is true. But in C, there are no real classes.
In C++, it is possible to overload cast operator, so, overloading a `char*` operator and using `printf` like this:

```
Foo bar;
...;
printf("%s",bar);
```

can be possible, if Foo overload the good operator. Or if you made a good method. In short, `printf` is as extensible as `cout` for me.

Technical argument I can see for C++ streams (in general... not only cout.) are:

- Typesafety. (And, by the way, if I want to print a single `'\n'` I use `putchar('\n')`... I will not use a nuke-bomb to kill an insect.).
- Simpler to learn. (no "complicated" parameters to learn, just to use `<<` and `>>` operators)
- Work natively with `std::string` (for `printf` there is `std::string::c_str()`, but for `scanf` ?)

For `printf` I see:

- Easier, or at least shorter (in term of characters written) complex formatting. Far more readable, *for me* (matter of taste I guess).
- Better control of what the function made (Return how many characters where written and there is the `%n` formatter: "Nothing printed. The argument must be a pointer to a signed int, where the number of characters written so far is stored." (from printf - C++ Reference)
- Better debugging possibilities. For same reason as last argument.

My personal preferences go to `printf` (and `scanf`) functions, mainly because I love short lines, and because I don't think type problems on printing text are really hard to avoid. The only thing I deplore with C-style functions is that `std::string` is not supported. We have to go through a `char*` before giving it to `printf` (with the `std::string::c_str()` if we want to read, but how to write?)

edited Jan 11 '12 at 10:20

Bertrand Marron
13.6k ● 6 ● 37 ● 77

answered Jan 11 '12 at 10:09

bmorel
29 ● 1

3 The compiler has no type information for varargs functions, so it won't convert the actual parameter (except *default argument promotions*, such as standard integral promotions). See 5.2.2p7. A user-defined conversion to `char*` won't be used. – Ben Voigt Jul 31 '12 at 2:42

Even if this worked, it wouldn't be an example of sprintf extensibility, just a clever hack to give sprintf what it expects, and it ignores some serious issues such as where the `char*` lives and for how long, and the dangers of user-defined implicit casts. – Marcelo Cantos Jun 7 '15 at 3:40

Of course you can write "something" a bit better to keep maintenance:

```
#include <iostream>
#include <cstdlib>

using namespace std;

class Something
{
    public:
        Something(int x, int y, int z) : a(x), b(y), c(z) { }
        int a;
        int b;
        int c;

        friend ostream& operator<<(ostream&, const Something&);

        void print() const { printf("%i, %i, %i\n", a, b, c); }
};

ostream& operator<<(ostream& o, const Something& s)
{
    o << s.a << ", " << s.b << ", " << s.c;
    return o;
}

int main(void)
{
    Something s(3, 2, 1);

    // Output with printf
    s.print(); // Simple as well, isn't it?

    // Output with cout
    cout << s << endl;

    return 0;
}
```

And a bit extended test of cout vs. printf, added a test of 'double', if anyone wants to do more testing (Visual Studio 2008, release version of the executable):

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

class TimedSection {
    char const *d_name;
    //timespec d_start;
    clock_t d_start;

    public:
        TimedSection(char const *name) :
            d_name(name)
        {
            //clock_gettime(CLOCK_REALTIME, &d_start);
            d_start = clock();
        }
        ~TimedSection() {
            clock_t end;
            //clock_gettime(CLOCK_REALTIME, &end);
            end = clock();
            double duration = /*1e3 * (end.tv_sec - d_start.tv_sec) +
                              1e-6 * (end.tv_nsec - d_start.tv_nsec);
                              */
                              (double) (end - d_start) / CLOCKS_PER_SEC;

            std::cerr << d_name << '\t' << std::fixed << duration * 1000.0 << "
ms\n";
        }
};


int main() {
    const int iters = 1000000;
    char const *text = "01234567890123456789";
    {
        TimedSection s("cout with only endl");
        for (int i = 0; i < iters; ++i)
            std::cout << std::endl;
    }
    {
        TimedSection s("cout with only '\\n'");
        for (int i = 0; i < iters; ++i)
            std::cout << '\n';
    }
    {
        TimedSection s("printf with only '\\n'");
        for (int i = 0; i < iters; ++i)
            printf("\n");
    }
    {
        TimedSection s("cout with string constant and endl");
        for (int i = 0; i < iters; ++i)
            std::cout << "01234567890123456789" << std::endl;
    }
    {
        TimedSection s("cout with string constant and '\\n'");
        for (int i = 0; i < iters; ++i)
            std::cout << "01234567890123456789\n";
    }
    {
        TimedSection s("printf with string constant and '\\n'");
        for (int i = 0; i < iters; ++i)
            printf("01234567890123456789\n");
    }
    {
        TimedSection s("cout with some stuff and endl");
        for (int i = 0; i < iters; ++i)
            std::cout << text << "01234567890123456789" << i << std::endl;
    }
    {
        TimedSection s("cout with some stuff and '\\n'");
        for (int i = 0; i < iters; ++i)
            std::cout << text << "01234567890123456789" << i << '\n';
    }
    {
        TimedSection s("printf with some stuff and '\\n'");
        for (int i = 0; i < iters; ++i)
            printf("%s01234567890123456789%i\n", text, i);
    }
    {
        TimedSection s("cout with formatted double (width & precision once)");
        std::cout << std::fixed << std::scientific << std::right << std::showpoint;
        std::cout.width(8);
        for (int i = 0; i < iters; ++i)
            std::cout << text << 8.315 << i << '\n';
    }
    {
        TimedSection s("cout with formatted double (width & precision on each
call)");
        std::cout << std::fixed << std::scientific << std::right << std::showpoint;

        for (int i = 0; i < iters; ++i)
            { std::cout.width(8);
              std::cout.precision(3);
              std::cout << text << 8.315 << i << '\n';
            }
    }
    {
        TimedSection s("printf with formatted double");
        for (int i = 0; i < iters; ++i)
```

```
            printf("%8.3f%i\n", 8.315, i);
    }
}
```

The result is:

```
cout with only endl    6453.000000 ms
cout with only '\n'    125.000000 ms
printf with only '\n'    156.000000 ms
cout with string constant and endl    6937.000000 ms
cout with string constant and '\n'    1391.000000 ms
printf with string constant and '\n'    3391.000000 ms
cout with some stuff and endl    9672.000000 ms
cout with some stuff and '\n'    7296.000000 ms
printf with some stuff and '\n'    12235.000000 ms
cout with formatted double (width & precision once)    7906.000000 ms
cout with formatted double (width & precision on each call)    9141.000000 ms
printf with formatted double    3312.000000 ms
```

edited May 22 at 21:29

Peter Mortensen
11.3k • 16 • 78 • 110

answered Nov 25 '11 at 8:10

LuP
44 • 3

Wow, why is `endl` so much less efficient than `'\n'`? – Nicholas Hamilton May 19 '14 at 23:38

I believe it's because `endl` flushes the buffer, and `\n` does not, although I'm not sure this is definitively the reason why. – Caleb Xu Mar 5 '15 at 2:18

This is not an asnwer to the question, it's more like an answer to Daniel's and Thomas's. – Fabio Turati Feb 8 '16 at 21:35

```
cout<< "Hello";
printf("%s", "Hello");
```

Both are used to print values. They have completely different syntax. C++ has both, C only has printf.

edited Sep 4 '10 at 20:23

Federico klez Culloca
10.7k • 9 • 36 • 67

answered May 20 '10 at 9:47

scatman
5,722 • 17 • 56 • 88

18 ... what? did you mixup something? – xtofl May 20 '10 at 10:08

30 Sometimes SO depresses me. – anon May 20 '10 at 10:16

3 The function names had been reversed: cout was used with the printf syntax, and printf was used with the cout syntax. Shouldn't have even been accepted! – Mahmoud Al-Qudsi May 20 '10 at 11:12

51 There should be a badge for worst answer but selected as correct. – Ed S. Jun 28 '10 at 17:28

8 Although this is certainty not the best answer, I don't understand how scatman is being punished for his answer only because it was picked as the best answer. xbit has a way worse answer IMO but has -1 vote. I'm not saying xbit should be down voted any more, but I don't see it being fair to down vote scatman for the OP's mistake anymore than it has to be... – Jesse Sep 4 '10 at 20:35

|

I'm not a programmer, but I have been a human factors engineer. I feel a programming language should be easy to learn, understand and use, and this requires that it have a simple and consistent linguistic structure. Although all the languages is symbolic and thus, at its core, arbitrary, there are conventions and following them makes the language easier to learn and use.

There are a vast number of functions in C++ and other languages written as function(parameter), a syntax that was originally used for functional relationships in mathematics in the pre-computer era. `printf()` follows this syntax and if the writers of C++ wanted to create any logically different method for reading and writing files they could have simply created a different function using a similar syntax.

In Python we of course can print using the also fairly standard `object.method` syntax, i.e. variablename.print, since variables are objects, but in C++ they are not.

I'm not fond of the cout syntax because the << operator does not follow any rules. It is a method or function, i.e. it takes a parameter and does something to it. However it is written as though it were a mathematical comparison operator. This is a poor approach from a human factors standpoint.

edited May 25 at 17:53

Peter Mortensen
11.3k • 16 • 78 • 110

answered Mar 7 '16 at 5:09

Daniel Woodard
5 • 2

`printf()` is a function whereas `cout` is a variable.

edited Jul 15 '14 at 16:25

Mark

2,069 ● 7 ● 22 ● 43

answered Jan 22 '14 at 20:35

john

25 ● 1

4 I did a roll-back because, although the answer itself may be wrong, it is still a genuine answer. If you (correctly) think the answer is wrong, you have two options: 1) add a comment or 2) add a new answer (or do both). Don't change someone's answer to such that it says something completely different from what was intended by the author. – Mark Jul 15 '14 at 16:27

`printf` is a function, but `printf()` is a function call =) – vp_arth Dec 29 '16 at 17:15