How to use the PI constant in C++

I want to use the PI constant and trigonometric functions in some C++ program. I get the trigonometric functions with `include <math.h>`. However, there doesn't seem to be a definition for PI in this header file.

How can I get PI without defining it manually?

c++ trigonometry

edited Apr 4 '14 at 9:08 — Yu Hao — 91.2k 22 146 196

asked Nov 13 '09 at 8:24 — Etan — 7,741 14 67 120

3 @tiwo, are you asking what's the difference between `3.14`, `3.141592` and `atan(1) * 4`? – Nikola Malešević Sep 6 '12 at 16:09

10 As a side note, cmath should be used in C++ instead of math.h, which is for C. – juzzlin Nov 20 '14 at 17:13

Loosely related: see cise.ufl.edu/~manuel/obfuscate/pi.c on how to calculate value of PI directly from definition. – lorro Jul 26 '16 at 18:20

15 Answers

On some (especially older) platforms (see the comments below) you might need to

```
#define _USE_MATH_DEFINES
```

and then include the necessary header file:

```
#include <math.h>
```

and the value of pi can be accessed via:

```
M_PI
```

In my `math.h` (2014) it is defined as:

```
# define M_PI           3.14159265358979323846  /* pi */
```

but check your `math.h` for more. An extract from the "old" `math.h` (in 2009):

```
/* Define _USE_MATH_DEFINES before including math.h to expose these macro
 * definitions for common math constants.  These are placed under an #ifdef
 * since these commonly-defined names are not part of the C/C++ standards.
 */
```

However:

1. on newer platforms (at least on my 64 bit Ubuntu 14.04) I do not need to define the `_USE_MATH_DEFINES`
2. On (recent) Linux platforms there are `long double` values too provided as a GNU Extension:

```
# define M_PIl          3.141592653589793238462643383279502884L /* pi */
```

edited Apr 25 '14 at 6:54

answered Nov 13 '09 at 8:28 — fritzone — 17.9k 10 57 107

33 `#define _USE_MATH_DEFINES` followed by `#include <math.h>` defines `M_PI` in visual c++. Thanks. – Etan Nov 13 '09 at 8:30

2 Works with cygwin headers as well. – Rob Mar 4 '11 at 5:35

1 works in xcode aswell – madoke Mar 5 '12 at 3:05

13 You can always include `cmath` instead of `math.h`. – Richard J. Ross III Apr 15 '12 at 20:34

9 Even after defining `_USE_MATH_DEFINES` if GCC complains that's because `__STRICT_ANSI__` is defined (perhaps you passed `-pedantic` or `-std=c++11`) which disallows `M_PI` to be defined, hence

Pi can be calculated as `atan(1)*4` . You could calculate the value this way and cache it.

answered Nov 13 '09 at 8:26

Konamiman
39.4k 14 88 120

59 For c++11 users: `constexpr double pi() { return std::atan(1)*4; }` – matiu Sep 3 '12 at 16:17

19 -1: Works only if `atan(1)*4 == 3.14159265358979323846264338327950288` (roughly speaking). I wouldn't bet on it. Be normal and use a raw literal to define the constant. Why lose precision when you don't need to? – Thomas Eding Oct 23 '12 at 23:49

26 One can avoid the multiplication operation with `atan2(0, -1);` . – legends2k May 29 '13 at 21:18

23 @matiu `atan` is not `constexpr` . – R. Martinho Fernandes Sep 5 '13 at 15:28

27 Try `acos(-1)` instead, no need for `atan2` . – Mehrdad Jul 9 '14 at 11:52

You could also use boost, which defines important math constants with maximum accuracy for the requested type (i.e. float vs double).

```
const double pi = boost::math::constants::pi<double>();
```

Check out the boost documentation for more examples.

answered Nov 13 '09 at 12:33

BuschnicK
2,636 4 25 41

135 Boost: Boosting the already unnecessary complexity of C++ since 1999! – Dan Moulding Jul 28 '10 at 18:22

37 Catchy and partly true. On the other hand boost can be phenomenally useful at times... – BuschnicK Jul 29 '10 at 14:52

41 @DanMoulding: Uhm. Is C the only other language you know? Because all other languages I know, except C, have a standard library which is magnitudes bigger than C++' (e.g. Python, Haskell, C#, PHP, Delphi, Erlang, Java, ...). From personal experience, that elitist `not gonna use libs` -opinion is a pest and probably the number one reason for bad software written in C++. – Sebastian Mach Jul 9 '13 at 6:15

9 @Gracchus: Yup. C++ without libraries (or without the new C++11 libraries) is, as much as I like that language and as much as I would like to code everything myself, not very productive. – Sebastian Mach Aug 11 '13 at 10:22

11 I believe he said *complexity* not *size*. Presumably referring to a) the 3 nested namespaces, and b) defining pi as a templated function rather than just a normal constant. – Timmmm Apr 3 '14 at 15:05

I would recommend just typing in pi to the precision you need. This would add no calculation time to your execution, and it would be portable without using any headers or #defines. Calculating acos or atan is always more expensive than using a precalculated value.

```
const double PI  =3.14159265358979323863;
const float  PI_F=3.14159265358979f;
```

edited Jan 19 '14 at 5:40

Francisco Presencia
5,923 4 29 64

answered Nov 12 '11 at 20:51

Alex
499 4 3

16 This is a great example why we should not take this approach, we people make mistakes, rounding, copy&pasting, etc. I think using M_PI is the right approach. – nacho4d Jan 21 '14 at 1:47

7 If one is doing this in C++11, make the `const a constexpr` . – legends2k Jan 23 '14 at 10:51

1 @nacho4d I too prefer M_PI if it's available, but not all systems are POSIX compliant. I think this approach is better than the 4*atan(1) method for the cases where M_PI is not available. – m24p Feb 20 '14 at 16:17

2 "Calculating acos or atan is always more expensive" is not true. Any modern optimizing compiler knows all about standard math functions and can constant-propagate through them. See e.g. goo.gl/BvdJyr – Nemo Jan 23 '16 at 5:37

1 @Nemo, Counter example: godbolt.org/g/DsAern As has been said elsewhere, it appears only GCC does this currently and that's likely because it has declared the basic math functions as `constexpr` . – Parker Coates Jan 4 at 20:19

```
double get_PI()
{
    double pi;
    __asm
    {
        fldpi
        fstp pi
    }
    return pi;
}

double PI = get_PI();
```

answered Jun 4 '15 at 15:01

Henrik
401 4 2

11 :-) probably not that platform independent, but a nice additional exotic solution! – Etan Jun 4 '15 at 18:25

Rather than writing

```
#define _USE_MATH_DEFINES
```

I would recommend using `-D_USE_MATH_DEFINES` or `/D_USE_MATH_DEFINES` depending on your compiler.

This way you are assured that even in the event of someone including the header before you do (and without the #define) you will still have the constants instead of an obscure compiler error that you will take ages to track down.

answered Nov 13 '09 at 12:0

Matthieu M.
179k 24 223 445

Good tip. If "you" are a compilation unit then of course you can ensure the macro is defined before anything is included. But if "you" are a header file, it's out of your control. – Steve Jessop Nov 13 '09 at 19:18

3 In fact even if "you" are a compilation unit... depending on the ordering of the headers is a the shortest path toward maintenance nightmare... – Matthieu M. Nov 13 '09 at 19:37

1 You don't have to depend on the ordering of the headers, though. It doesn't matter whether headers include each other, provided that you do the #define before you #include anything at all (at least, assuming that nothing #undefs it). Same applies to NDEBUG. – Steve Jessop Nov 14 '09 at 3:13

1 The very common issue in a project is that if you're compiling with Visual Studio for example you don't know in which order the compiler is going to go through your files so if you use `<cmath>` in different places it becomes a big pain (especially if it is included by another library you are including). It would have been much better if they put that part outside of the header guards but well can't do much about that now. The compiler directive works pretty well indeed. – meneldal May 18 '15 at 2:18

Since the official standard library doesn't define a constant PI you would have to define it yourself. So the answer to your question "How can I get PI without defining it manually?" is "You don't -- or you rely on some compiler-specific extensions.". If you're not concerned about portability you could check your compiler's manual for this.

C++ allows you to write

```
const double PI = std::atan(1.0)*4;
```

but the initialization of this constant is not guaranteed to be static. The G++ compiler however handles those math functions as intrinsics and is able to compute this constant expression at compile-time.

answered Nov 13 '09 at 8:37

sellibitze
20.4k 2 54 82

5 The standard does not define pi? You got to be kidding me... – Navin Mar 30 '14 at 5:14

4 I usually use acos(-1), as you say, they are compile-time evaluated. When I tested M_PI, acos(-1) and atan(1)*4, I got identical values. – Micah Sep 9 '14 at 19:14

From the Posix man page of math.h:

> The <math.h> header shall provide for the following constants. The values are of type double and are accurate within the precision of the double type.
>
> M_PI Value of pi

```
M_PI_4 Value of pi/4

M_1_PI Value of 1/pi

M_2_PI Value of 2/pi

M_2_SQRTPI
       Value of 2/ sqrt pi
```

edited Jun 14 '13 at 21:26

yiati
698 9 22

answered Mar 12 '13 at 11:35

Joakim
261 3 3

This worked on Linux. – Kemin Zhou Aug 4 '16 at 19:16

Standard C++ doesn't have a constant for PI.

Many C++ compilers define `M_PI` in `cmath` (or in `math.h` for C) as a non-standard extension. You may have to `#define _USE_MATH_DEFINES` before you can see it.

answered Nov 13 '09 at 8:27

RichieHindle
174k 38 287 357

I generally prefer defining my own: `const double PI = 2*acos(0.0);` because not all implementations provide it for you.

The question of whether this function gets called at runtime or is static'ed out at compile time is usually not an issue, because it only happens once anyway.

answered Nov 18 '09 at 4:03

Sumudu Fernando
1,528 1 8 15

8 acos(-1) is also pi. – Roderick Taylor Aug 5 '11 at 2:55

3 It's often less CPU instructions and/or less latency to load an immediate operand than read an operand from a memory location. Also, only expressions that are known at compile-time could be pre-computed (I mean `double x = pi * 1.5;` and the like). If you ever intend to use PI in crunchy math in tight loops, you better make sure the value is known to the compiler. – Eugene Ryabtsev Aug 19 '14 at 7:55

I would do

```
template<typename T>
T const pi = std::acos(-T(1));
```

or

```
template<typename T>
T const pi = std::arg(-std::log(T(2)));
```

I would **not** *typing in π to the precision you need*. What is that even supposed to mean? The *precision you need* is the precision of `T`, but we know nothing about `T`.

You might say: *What are you talking about?* `T` *will be* `float`, `double` *or* `long double`. *So, just type in the precision of* `long double`, *i.e.*

```
template<typename T>
T const pi = static_cast<T>(/* long double precision π */);
```

But do you really know that there won't be a new floating point type in the standard in the future with an even higher precision than `long double`? You don't.

And that's why the first solution is beautiful. You can be quite sure that the standard would overload the trigonometric functions for a new type.

And please, don't say that the evaluation of a trigonometric function at initialization is a performance penalty.

edited Mar 14 '16 at 14:23

answered Feb 26 '16 at 12:25

0xbadf00d
5,202 12 49 77

1 Note that `arg(log(x)) == π` for all `0 < x < 1`. – 0xbadf00d Feb 26 '16 at 12:45

```
#define _USE_MATH_DEFINES
#include <cmath>

#ifndef M_PI
#define M_PI (3.14159265358979323846)
#endif

#ifndef M_PIl
#define M_PIl (3.14159265358979323846264338327950288)
#endif
```

On a side note, all of below compilers define M_PI and M_PIl constants if you include `<cmath>` . There is no need to add `#define _USE_MATH_DEFINES which is only required for VC++.

```
x86 GCC 4.4+
ARM GCC 4.5+
x86 Clang 3.0+
```

edited Feb 13 at 15:16

Chris Nolet
4,317 2 37 75

answered May 28 '16 at 11:42

ShitalShah
16.8k 4 70 68

Can the downvoter comment on what is wrong with this answer. This is well researched and tested and being in use in real system. I had definitely like to improve it if something is wrong. – ShitalShah Jul 19 '16 at 7:02

On windows (cygwin + g++), I've found it necessary to add the flag `-D_XOPEN_SOURCE=500` for the preprocessor to process the definition of `M_PI` in `math.h` .

edited Dec 15 '14 at 12:36

A.L
6,318 8 33 58

answered Dec 15 '14 at 12:11

Papa Smurf
99 1 3

1 Does this work on mingw? – Jerfov2 Dec 27 '15 at 18:34

2 This is not an answer, but a comment to fritzone's answer. – 0xbadf00d Feb 26 '16 at 12:47

2 @0xbadf00d: It is a completely standalone answer that provides the steps needed to get `M_PI` working on a particular platform. That isn't a comment on an answer for some other platform any more that an answer for some other platform is a comment on this one. – Ben Voigt Jun 9 '16 at 19:18

C++14 lets you do `static constexpr auto pi = acos(-1);`

answered Mar 21 '16 at 20:09

Willy Goat
471 2 15

`std::acos` is not a `constexpr` . So, your code won't compile. – 0xbadf00d May 5 '16 at 17:05

@0xbadf00d I compiled it with g++ – Willy Goat May 6 '16 at 3:00

4 @WillyGoat: Then g++ is wrong, because `acos` is not `constexpr` in C++14, and is not proposed to become `constexpr` even in C++17 – Ben Voigt Jun 9 '16 at 19:22

You can do this:

```
#include <cmath>
#ifndef M_PI
#define M_PI (3.14159265358979323846)
#endif
```

If `M_PI` is already defined in `cmath` , this won't do anything else than include `cmath` . If `M_PI` isn't defined (which is the case for example in Visual Studio), it will define it. In both cases, you can use `M_PI` to get the value of pi.

This value of pi comes from Qt Creator's qmath.h.

answered Nov 9 '16 at 11:53

Donald Duck
3,252 11 30 45
