When are C++ macros beneficial? [closed]

Asked 16 years, 3 months ago Modified 2 years, 4 months ago Viewed 105k times

183

Closed. This question needs to be more focused. It is not currently accepting answers.

Want to improve this question? Update the question so it focuses on one problem only by editing this post.

Closed 6 years ago.

Improve this question

The **C** preprocessor is justifiably feared and shunned by the C++ community. In-lined functions, consts and templates are usually a safer and superior alternative to a `#define`.

The following macro:

```
#define SUCCEEDED(hr) ((HRESULT)(hr) >= 0)
```

is in no way superior to the type safe:

```
inline bool succeeded(int hr) { return hr >= 0; }
```

But macros do have their place, please list the uses you find for macros that you *can't* do without the preprocessor.

Please put each use-cases in a seperate answer so it can be voted up and if you know of how to achieve one of the answers without the preprosessor point out how in that answer's comments.

c++ c-preprocessor

Share

Improve this question

Follow

edited Jun 3, 2016 at 17:28

community wiki
8 revs, 4 users 75%
Motti

I once took a C++ application full of macros that took 45 minutes to build, replaced the macros with inline functions, and got the build down to less than 15 minutes. – endian Oct 23, 2008 at 7:37

Static Assert – Özgür Mar 8, 2009 at 0:42

The thread is about contexts in which macros are beneficial, not contexts in which they are suboptimal. – underscore_d May 25, 2017 at 21:46

@Özgür What do you intend to say? – John Sep 24, 2021 at 2:49

38 Answers

Sorted by: Highest score (default)

1 2 Next

131

As wrappers for debug functions, to automatically pass things like `__FILE__`, `__LINE__`, etc:

```
#ifdef ( DEBUG )
#define M_DebugLog( msg )  std::cout << __FILE__ << ":" << __LINE__ << ": " << msg
#else
#define M_DebugLog( msg )
#endif
```

Since C++20 the magic type `std::source_location` can however be used instead of `__LINE__` and `__FILE__` to implement an analogue as a normal function (template).

Share

Improve this answer

Follow

edited Aug 3, 2022 at 5:49

community wiki
4 revs, 4 users 82%
Frank Szczerba

15 Actually, the original snippet: << **FILE** ":" << is fine, **FILE** generates a string constant, which will be concatenated with the ":" into a single string by the pre-processor. – Frank Szczerba Mar 19, 2009 at 23:59

12 This only requires the preprocessor because `__FILE__` and `__LINE__` **also** require the preprocessor. Using them in your code is like an infection vector for the preprocessor. – T.E.D. Apr 30, 2012 at 18:53

1 @T.E.D. Why "Using them in your code is like an infection vector for the preprocessor. "? Could you explain that in more detail for me? – John Sep 24, 2021 at 2:53

1 @John - 10 year later Q. Wow. Well, one example I remember was an old logging facility designed to have these passed in that I wanted to simplify/modernize to be stream based instead. The problem I ran into was that I had to make the stream objects macros too, so they could automagically fill in those values. If you try it with straight code, every log message gets the file and line number of the inside of the log stream object. – T.E.D. Sep 24, 2021 at 3:32

97

Methods must always be complete, compilable code; macros may be code fragments. Thus you can define a foreach macro:

```
#define foreach(list, index) for(index = 0; index < list.size(); index++)
```

And use it as thus:

```
foreach(cookies, i)
    printf("Cookie: %s", cookies[i]);
```

Since C++11, this is superseded by the range-based for loop.

Share

Improve this answer

Follow

edited Mar 2, 2015 at 15:10

community wiki
2 revs, 2 users 88%
jdmichal

6 +1 If you are using some ridiculously complex iterator syntax, writing a foreach style macro can make your code much easier to read and maintain. I've done it, it works. – postfuturist Oct 25, 2008 at 21:01

10 Most comments completely irrelevant to the point that macros may be code fragments instead of complete code. But thank you for the nitpicking. – jdmichal Nov 12, 2008 at 20:15

13 This is C not C++. If you are doing C++, you should be using iterators and std::for_each. – chrish Jun 11, 2009 at 19:19

23 I disagree, chrish. Before lambda's, `for_each` was a nasty thing, because the code each element was runnnig through was not local to the calling point. `foreach`, (and I highly recommend `BOOST_FOREACH` instead of a hand-rolled solution) let's you keep the code close to the iteration site, making it more readable. That said, once lambda's roll out, `for_each` might once again be the way to go. – GManNickG Aug 18, 2009 at 23:50

8 And it's worth noting that BOOST_FOREACH is itself a macro (but a very well-thought-out one) – Tyler McHenry Aug 18, 2009 at 23:56

61

Header file guards necessitate macros.

Are there any other areas that **necessitate** macros? Not many (if any).

Are there any other situations that benefit from macros? YES!!!

One place I use macros is with very repetitive code. For example, when wrapping C++ code to be used with other interfaces (.NET, COM, Python, etc...), I need to catch different types of exceptions. Here's how I do that:

```
#define HANDLE_EXCEPTIONS \
catch (::mylib::exception& e) { \
    throw gcnew MyDotNetLib::Exception(e); \
} \
catch (::std::exception& e) { \
    throw gcnew MyDotNetLib::Exception(e, __LINE__, __FILE__); \
} \
catch (...) { \
    throw gcnew MyDotNetLib::UnknownException(__LINE__, __FILE__); \
}
```

I have to put these catches in every wrapper function. Rather than type out the full catch blocks each time, I just type:

```
void Foo()
{
```

```
    try {
        ::mylib::Foo()
    }
    HANDLE_EXCEPTIONS
}
```

This also makes maintenance easier. If I ever have to add a new exception type, there's only one place I need to add it.

There are other useful examples too: many of which include the `__FILE__` and `__LINE__` preprocessor macros.

Anyway, macros are very useful when used correctly. Macros are not evil -- their **misuse** is evil.

Share

Improve this answer

Follow

edited Feb 7, 2014 at 7:48

community wiki
4 revs, 4 users 96%
Kevin

8 Most compilers support `#pragma once` these days, so I doubt guards are really necessary – 1800 INFORMATION Sep 18, 2008 at 20:10

15 They are if you're writing for all compilers instead of only most ;-) – Steve Jessop Sep 18, 2008 at 20:13

36 So instead of portable, standard preprocessor functionality, you recommend using a preprocessor extension to avoid using the preprocessor? Seems sort of ridiculous to me. – Logan Capaldo Mar 8, 2009 at 0:49

6 There is a solution for that that doesn't require macros: `void handleExceptions(){ try { throw } catch (::mylib::exception& e) {....} catch (::std::exception& e) {...} ... }`. And on the function side: `void Foo(){  try {::mylib::Foo()  } catch (...) {handleExceptions(); } }` – MikeMB Jul 2, 2016 at 23:59

3 I don't see the point of the macro here. You could define a function template similar to `std::invoke` and place exception handling inside. Basically a higher-order function that calls supplied function and handles any errors. – Xeverous Mar 11, 2022 at 12:16

57

Mostly:

1. Include guards
2. Conditional compilation
3. Reporting (predefined macros like `__LINE__` and `__FILE__`)
4. (rarely) Duplicating repetitive code patterns.
5. In your competitor's code.

Share

Improve this answer

Follow

edited Aug 13, 2009 at 20:12

community wiki
2 revs, 2 users 92%
David Thornley

2 Looking for some help on how to realize number 5. Can you guide me towards a solution? – Max Feb 14, 2020 at 13:49

1 @David Thornley Could you please show me an example on "Conditional compilation"? – John Sep 24, 2021 at 3:05

4 @John • `#ifdef _WIN32 /* do the Windows thing */ #elif __linux__ /* do the Linux thing */ #elif __APPLE__ /* do the Macintosh thing */ #else #error Unsupported #endif` – Eljay Jun 20, 2022 at 13:55

50

Inside conditional compilation, to overcome issues of differences between compilers:

```
#ifdef WE_ARE_ON_WIN32
#define close(parm1)            _close (parm1)
#define rmdir(parm1)            _rmdir (parm1)
#define mkdir(parm1, parm2)     _mkdir (parm1)
#define access(parm1, parm2)    _access(parm1, parm2)
#define create(parm1, parm2)    _creat (parm1, parm2)
#define unlink(parm1)           _unlink(parm1)
#endif
```

Share

Improve this answer

Follow

edited Sep 24, 2021 at 3:15

community wiki
2 revs, 2 users 96%
Andrew Stein

16 In C++, the same could be obtained through the use of inline functions: <code>#ifdef ARE_WE_ON_WIN32
inline int close(int i) { return _close(i) ; }
 #endif</code> – paercebal Oct 22, 2008 at 22:31

3 That removes the #define's, but not the #ifdef and #endif. Anyway, I agree with you. – Gorpik Oct 23, 2008 at 8:16

23 NEVER EVER define lower case macros. Macros to alter functions are my nightmare (thank you Microsoft). Best example is in first line. Many libraries have `close` functions or methods. Then when you include header of this library and header with this macro than you have a big problem, you are unable to use library API. – Marek R May 28, 2015 at 14:56

AndrewStein, do you see any benefit to the use of macros in this context over @paercebal's suggestion? If not, it seems macros are actually gratuitous. – einpoklum Apr 20, 2016 at 11:10

2 `#ifdef WE_ARE_ON_WIN32` plz :) – Lightness Races in Orbit Aug 3, 2018 at 14:27

37

When you want to make a string out of an expression, the best example for this is `assert` (`#x` turns the value of `x` to a string).

```
#define ASSERT_THROW(condition) \
if (!(condition)) \
     throw std::exception(#condition " is false");
```

Share

Improve this answer

Follow

edited Feb 4, 2010 at 13:34

community wiki
3 revs
Motti

8 Just a nitpick, but I personally would leave the semicolon off. – Michael Myers ♦ Sep 18, 2008 at 21:05

13 I agree, in fact I would put it in a do {} while(false) (to prevent else highjacking) but I wanted to keep it simple. – Motti Oct 5, 2008 at 18:24

35

String constants are sometimes better defined as macros since you can do more with string literals than with a `const char *`.

e.g. String literals can be easily concatenated.

```
#define BASE_HKEY "Software\\Microsoft\\Internet Explorer\\"
// Now we can concat with other literals
RegOpenKey(HKEY_CURRENT_USER, BASE_HKEY "Settings", &settings);
RegOpenKey(HKEY_CURRENT_USER, BASE_HKEY "TypedURLs", &URLs);
```

If a `const char *` were used then some sort of string class would have to be used to perform the concatenation at runtime:

```
const char* BaseHkey = "Software\\Microsoft\\Internet Explorer\\";
RegOpenKey(HKEY_CURRENT_USER, (string(BaseHkey) + "Settings").c_str(),
&settings);
RegOpenKey(HKEY_CURRENT_USER, (string(BaseHkey) + "TypedURLs").c_str(), &URLs);
```

Since C++20 it is however possible to implement a string-like class type that can be used as a non-type template parameter type of a user-defined string literal operator which allows such concatenation operations at compile-time without macros.

Share

Improve this answer

Follow

edited Aug 3, 2022 at 5:57

community wiki
3 revs, 2 users 97%
Motti

4 In C++11, I'd consider this to be the most important part (other than include guards). Macros are really the best thing that we have for compile-time string processing. That is a feature that I hope we get in C++11++ – David Stone Nov 16, 2012 at 3:34

3 This is the situation that led to me wishing for macros in C#. – Rawling Dec 5, 2012 at 15:25

3 I wish I could +42 this. A very important, though not often remembered aspect of string literals. – Daniel Kamil Kozar Mar 25, 2016 at 11:22

24

When you want to change the program flow (`return`, `break` and `continue`) code in a function behaves differently than code that is actually inlined in the function.

```
#define ASSERT_RETURN(condition, ret_val) \
if (!(condition)) { \
    assert(false && #condition); \
    return ret_val; }

// should really be in a do { } while(false) but that's another discussion.
```

Share

Improve this answer

Follow

edited Nov 17, 2010 at 15:11

community wiki
2 revs
Motti

Throwing an exception seems to me like a better alternative. – einpoklum Apr 20, 2016 at 11:11

When writing python C(++) extensions, exceptions are propagated by setting an exception string, then returning `-1` or `NULL`. So a macro can greatly reduce boilerplate code there. – black_puppydog Oct 14, 2016 at 12:59

20

The obvious include guards

```
#ifndef MYHEADER_H
#define MYHEADER_H

...

#endif
```

Share

Improve this answer

Follow

answered Sep 18, 2008 at 19:53

community wiki
Kena

Let's say we'll ignore obvious things like header guards.

Sometimes, you want to generate code that needs to be copy/pasted by the precompiler:

```
#define RAISE_ERROR_STL(p_strMessage)
\
do
\
{
\
   try
\
   {
\
      std::tstringstream strBuffer ;
\
      strBuffer << p_strMessage ;
\
      strMessage = strBuffer.str() ;
\
      raiseSomeAlert(__FILE__, __FUNCSIG__, __LINE__, strBuffer.str().c_str())
\
   }
\
   catch(...){}
\
   {
\
   }
\
}
\
while(false)
```

which enables you to code this:

```
RAISE_ERROR_STL("Hello... The following values " << i << " and " << j << " are
wrong") ;
```

And can generate messages like:

```
Error Raised:
====================================
File : MyFile.cpp, line 225
Function : MyFunction(int, double)
Message : "Hello... The following values 23 and 12 are wrong"
```

Note that mixing templates with macros can lead to even better results (i.e. automatically generating the values side-by-side with their variable names)

Other times, you need the __FILE__ and/or the __LINE__ of some code, to generate debug info, for example. The following is a classic for Visual C++:

```
#define WRNG_PRIVATE_STR2(z) #z
#define WRNG_PRIVATE_STR1(x) WRNG_PRIVATE_STR2(x)
#define WRNG __FILE__ "("WRNG_PRIVATE_STR1(__LINE__)") : ------------ : "
```

As with the following code:

```
#pragma message(WRNG "Hello World")
```

it generates messages like:

```
C:\my_project\my_cpp_file.cpp (225) : ------------ Hello World
```

Other times, you need to generate code using the # and ## concatenation operators, like generating getters and setters for a property (this is for quite a limited cases, through).

Other times, you will generate code than won't compile if used through a function, like:

```
#define MY_TRY      try{
#define MY_CATCH    } catch(...) {
#define MY_END_TRY  }
```

Which can be used as

```
MY_TRY
   doSomethingDangerous() ;
MY_CATCH
   tryToRecoverEvenWithoutMeaningfullInfo() ;
   damnThoseMacros() ;
MY_END_TRY
```

(still, I only saw this kind of code rightly used **once**)

Last, but not least, the famous `boost::foreach` !!!

```
#include <string>
#include <iostream>
#include <boost/foreach.hpp>

int main()
{
    std::string hello( "Hello, world!" );

    BOOST_FOREACH( char ch, hello )
    {
        std::cout << ch;
    }
```

```
    return 0;
}
```

(Note: code copy/pasted from the boost homepage)

Which is (IMHO) way better than `std::for_each`.

So, macros are always useful because they are outside the normal compiler rules. But I find that most the time I see one, they are effectively remains of C code never translated into proper C++.

Share

Improve this answer

Follow

edited Aug 18, 2009 at 20:14

community wiki
4 revs, 2 users 97%
paercebal

3 Use the CPP only for what the compiler cannot do. For example, RAISE_ERROR_STL should use the CPP only to determine file, line, and function signature, and pass those to a function (possibly inline) that does the rest. – Rainer Blome Feb 4, 2016 at 14:20

Please update your answer to reflect C++11 and address @RainerBlome's comment. – einpoklum Apr 20, 2016 at 11:13

@RainerBlome : We do agree. The RAISE_ERROR_STL macro is pre-C++11, so in that context, it is fully justified. My understanding (but I have never had the occasion to deal with those specific features) is that you can use variadic templates (or macros?) in Modern C++ to solve the problem more elegantly. – paercebal Apr 24, 2016 at 12:10

@einpoklum : "Please update your answer to reflect C++11 and address RainerBlome's comment" No. :-) . . . I believe, at best, I will add a section for Modern C++, with alternative implementations reducing or eliminating the need for macros, but the point stands: Macros are ugly and evil, but when you need to do something the compiler doesn't understand, you do it via macros. – paercebal Apr 24, 2016 at 12:14

Even with C++11, a lot of what your macro does can be left for a function to do: `#include <sstream> #include <iostream> using namespace std; void trace(char const * file, int line, ostream & o) { cerr<<file<<":"<<line<<": "<< static_cast<ostringstream & >(o).str().c_str()<<endl; } struct Oss { ostringstream s; ostringstream & lval() { return s; } }; #define TRACE(ostreamstuff) trace(__FILE__, __LINE__, Oss().lval()<<ostreamstuff) int main() { TRACE("Hello " << 123); return 0; }` That way, the macro is way shorter. – Rainer Blome Apr 29, 2016 at 13:28

18

Unit test frameworks for C++ like UnitTest++ pretty much revolve around preprocessor macros. A few lines of unit test code expand into a hierarchy of classes that wouldn't be fun at all to type manually. Without something like UnitTest++ and it's preprocessor magic, I don't know how you'd efficiently write unit tests for C++.

Share

edited Apr 29, 2017 at 19:29

community wiki

Improve this answer

Follow

3 revs, 2 users 88%
Joe

Unittests are perfectly possible to write without a framework. In the end, it only really depends on what kind of output you want. If you don't care, a simple exit value indicating success or failure should be perfectly fine. – Clearer Jun 19, 2018 at 13:44

16

You can't perform short-circuiting of function call arguments using a regular function call. For example:

```
#define andm(a, b) (a) && (b)

bool andf(bool a, bool b) { return a && b; }

andm(x, y) // short circuits the operator so if x is false, y would not be evaluated
andf(x, y) // y will always be evaluated
```

Share

Improve this answer

Follow

edited Sep 18, 2008 at 20:09

community wiki
2 revs
1800 INFORMATION

4 Maybe a more general point: functions evaluate their arguments exactly once. Macros can evaluate arguments more times or fewer times. – Steve Jessop Sep 18, 2008 at 20:06

@[Greg Rogers] all the macro preprocessor does is substitute text. Once you understand that, there should be no more mystery about it. – 1800 INFORMATION Sep 18, 2008 at 20:08

You could get the equivalent behavior by templatizing andf instead of forcing the evaluation to bool before calling the function. I wouldn't have realized what you said was true without trying it for myself though. Interesting. – Greg Rogers Sep 18, 2008 at 20:11

How exactly could you do that with a template? – 1800 INFORMATION Sep 18, 2008 at 21:47

9 Hiding short-circuiting operations behind a function style macro is one of the things I really don't want to see in production code. – MikeMB Jul 3, 2016 at 0:08

14

To fear the C preprocessor is like to fear the incandescent bulbs just because we get fluorescent bulbs. Yes, the former can be {electricity | programmer time} inefficient. Yes, you can get (literally) burned by them. But they can get the job done if you properly handle it.

When you program embedded systems, C uses to be the only option apart form assembler. After programming on desktop with C++ and then switching to smaller, embedded targets, you learn to stop worrying about “inelegancies” of so many bare C

features (macros included) and just trying to figure out the best and safe usage you can get from those features.

Alexander Stepanov says:

> When we program in C++ we should not be ashamed of its C heritage, but make full use of it. The only problems with C++, and even the only problems with C, arise when they themselves are not consistent with their own logic.

Share
Improve this answer
Follow

answered Oct 17, 2008 at 15:46

community wiki
VictorH

I think this is the wrong attitude. Just because you can learn to "properly handle it" doesn't mean it's worth anyone's time and effort. – Neil G Apr 7, 2010 at 23:20

10

Some very advanced and useful stuff can still be built using preprocessor (macros), which you would never be able to do using the c++ "language constructs" including templates.

Examples:

Making something both a C identifier and a string

Easy way to use variables of enum types as string in C

Boost Preprocessor Metaprogramming

Share
Improve this answer
Follow

edited May 23, 2017 at 12:26

community wiki
3 revs
Suma

The third link is broken fyi – Robin Hartland Jun 28, 2015 at 21:01

Take a look `stdio.h` and `sal.h` file in `vc12` for better understand. – Elshan May 17, 2016 at 10:54

9

We use the `__FILE__` and `__LINE__` macros for diagnostic purposes in information rich exception throwing, catching and logging, together with automated log file scanners in our QA infrastructure.

For instance, a throwing macro `OUR_OWN_THROW` might be used with exception type and constructor parameters for that exception, including a textual description. Like this:

```
OUR_OWN_THROW(InvalidOperationException, (L"Uninitialized foo!"));
```

This macro will of course throw the `InvalidOperationException` exception with the description as constructor parameter, but it'll also write a message to a log file consisting of the file name and line number where the throw occured and its textual description. The thrown exception will get an id, which also gets logged. If the exception is ever caught somewhere else in the code, it will be marked as such and the log file will then indicate that that specific exception has been handled and that it's therefore not likely the cause of any crash that might be logged later on. Unhandled exceptions can be easily picked up by our automated QA infrastructure.

Share
Improve this answer
Follow

answered Sep 18, 2008 at 20:05

community wiki
Johann Gerell

8

Code repetition.

Have a look to boost preprocessor library, it's a kind of meta-meta-programming. In topic->motivation you can find a good example.

Share
Improve this answer
Follow

edited Aug 1, 2014 at 23:28

community wiki
2 revs, 2 users 91%
Ruggero Turra

I almost all, if not all, cases - code repetition can be avoided with function calls. – einpoklum Apr 20, 2016 at 11:13

@einpoklum: I don't agree. Have a look to the link – Ruggero Turra Apr 20, 2016 at 13:19

7

One common use is for detecting the compile environment, for cross-platform development you can write one set of code for linux, say, and another for windows when no cross platform library already exists for your purposes.

So, in a rough example a cross-platform mutex can have

```
void lock()
{
    #ifdef WIN32
    EnterCriticalSection(...)
    #endif
    #ifdef POSIX
    pthread_mutex_lock(...)
    #endif
}
```

For functions, they are useful when you want to explicitly ignore type safety. Such as the many examples above and below for doing ASSERT. Of course, like a lot of C/C++ features you can shoot yourself in the foot, but the language gives you the tools and lets you decide what to do.

Share

Improve this answer

Follow

edited Sep 18, 2008 at 19:58

community wiki
2 revs
Doug T.

> Since the questioner asked: this can be done without macros by including different headers via different include paths per platform. I'm inclined to agree though that macros are often more convenient. – Steve Jessop Sep 18, 2008 at 19:59

> I second that. If you start using macros for that purpose, the code can quickly become much less readable – Nemanja Trifunovic Sep 18, 2008 at 20:38

7

I occasionally use macros so I can define information in one place, but use it in different ways in different parts of the code. It's only slightly evil :)

For example, in "field_list.h":

```
/*
 * List of fields, names and values.
 */
FIELD(EXAMPLE1, "first example", 10)
FIELD(EXAMPLE2, "second example", 96)
FIELD(ANOTHER, "more stuff", 32)
...
#undef FIELD
```

Then for a public enum it can be defined to just use the name:

```
#define FIELD(name, desc, value) FIELD_ ## name,

typedef field_ {

#include "field_list.h"

    FIELD_MAX
```

```
} field_en;
```

And in a private init function, all the fields can be used to populate a table with the data:

```
#define FIELD(name, desc, value) \
    table[FIELD_ ## name].desc = desc; \
    table[FIELD_ ## name].value = value;

#include "field_list.h"
```

Share

Improve this answer

Follow

answered Sep 18, 2008 at 20:50

community wiki
Andrew Johnson

2 Note: similar technique can be implemented even without a separate include. See: stackoverflow.com/questions/147267/… stackoverflow.com/questions/126277/… – Suma Oct 23, 2008 at 21:54

6

Something like

```
void debugAssert(bool val, const char* file, int lineNumber);
#define assert(x) debugAssert(x,__FILE__,__LINE__);
```

So that you can just for example have

```
assert(n == true);
```

and get the source file name and line number of the problem printed out to your log if n is false.

If you use a normal function call such as

```
void assert(bool val);
```

instead of the macro, all you can get is your assert function's line number printed to the log, which would be less useful.

Share

Improve this answer

Follow

answered Sep 18, 2008 at 20:03

community wiki
Keshi

Why would you reinvent the wheel when implementations of the Standard Library already provide via `<cassert>` the `assert()` macro, which dumps the file/line/function info? (in all implementations I've seen, anyway) – underscore_d May 25, 2017 at 21:39

4

```
#define ARRAY_SIZE(arr) (sizeof arr / sizeof arr[0])
```

Unlike the 'preferred' template solution discussed in a current thread, you can use it as a constant expression:

```
char src[23];
int dest[ARRAY_SIZE(src)];
```

Share

Improve this answer

Follow

edited Oct 5, 2008 at 18:53

community wiki
2 revs
fizzer

2 This can be done with templates in a safer way (which won't compile if passed a pointer rather than an array) stackoverflow.com/questions/720077/calculating-size-of-an-array/... – Motti Aug 13, 2009 at 20:17

1 Now that we have constexpr in C++11, the safe (non-macro) version can also be used in a constant expression. `template<typename T, std::size_t size> constexpr std::size_t array_size(T const (&)[size]) { return size; }` – David Stone Sep 9, 2015 at 2:02

3

You can use #defines to help with debugging and unit test scenarios. For example, create special logging variants of the memory functions and create a special memlog_preinclude.h:

```
#define malloc memlog_malloc
#define calloc memlog calloc
#define free memlog_free
```

Compile you code using:

```
gcc -Imemlog_preinclude.h ...
```

An link in your memlog.o to the final image. You now control malloc, etc, perhaps for logging purposes, or to simulate allocation failures for unit tests.

Share

answered Sep 18, 2008 at 21:03

community wiki

Improve this answer

Follow

Andrew Johnson

3

You can `#define` constants on the compiler command line using the `-D` or `/D` option. This is often useful when cross-compiling the same software for multiple platforms because you can have your makefiles control what constants are defined for each platform.

Share

Improve this answer

Follow

answered Sep 19, 2008 at 3:44

community wiki

bk1e

3

When you are making a decision at compile time over Compiler/OS/Hardware specific behavior.

It allows you to make your interface to Comppiler/OS/Hardware specific features.

```
#if defined(MY_OS1) && defined(MY_HARDWARE1)
#define   MY_ACTION(a,b,c)      doSothing_OS1HW1(a,b,c);}
#elif define(MY_OS1) && defined(MY_HARDWARE2)
#define   MY_ACTION(a,b,c)      doSomthing_OS1HW2(a,b,c);}
#elif define(MY_SUPER_OS)
          /* On this hardware it is a null operation */
#define   MY_ACTION(a,b,c)
#else
#error  "PLEASE DEFINE MY_ACTION() for this Compiler/OS/HArdware configuration"
#endif
```

Share

Improve this answer

Follow

edited Oct 24, 2008 at 3:14

community wiki

2 revs

Loki Astari

2

Compilers can refuse your request to inline.

Macros will always have their place.

Something I find useful is #define DEBUG for debug tracing -- you can leave it 1 while debugging a problem (or even leave it on during the whole development cycle) then turn it off when it is time to ship.

Share

Improve this answer

Follow

answered Sep 18, 2008 at 19:54

community wiki

unwieldy

11 If the compiler refuses your request to inline, it might have a very good reason. A good compiler will be better at inlining properly than you are, and a bad one will give you more performance problems than this. – David Thornley Oct 23, 2008 at 19:22

@DavidThornley Or it might not be a great optimising compiler like GCC or CLANG/LLVM. Some compilers are just crap. – mrr Jun 3, 2014 at 4:08

2

In my last job, I was working on a virus scanner. To make thing easier for me to debug, I had lots of logging stuck all over the place, but in a high demand app like that, the expense of a function call is just too expensive. So, I came up with this little Macro, that still allowed me to enable the debug logging on a release version at a customers site, without the cost of a function call would check the debug flag and just return without logging anything, or if enabled, would do the logging... The macro was defined as follows:

```
#define dbgmsg(_FORMAT, ...)  if((debugmsg_flag  & 0x00000001) ||
(debugmsg_flag & 0x80000000))     { log_dbgmsg(_FORMAT, __VA_ARGS__);  }
```

Because of the VA_ARGS in the log functions, this was a good case for a macro like this.

Before that, I used a macro in a high security application that needed to tell the user that they didn't have the correct access, and it would tell them what flag they needed.

The Macro(s) defined as:

```
#define SECURITY_CHECK(lRequiredSecRoles)
if(!DoSecurityCheck(lRequiredSecRoles, #lRequiredSecRoles, true)) return
#define SECURITY_CHECK_QUIET(lRequiredSecRoles)
(DoSecurityCheck(lRequiredSecRoles, #lRequiredSecRoles, false))
```

Then, we could just sprinkle the checks all over the UI, and it would tell you which roles were allowed to perform the action you tried to do, if you didn't already have that role. The reason for two of them was to return a value in some places, and return from a void function in others...

```
SECURITY_CHECK(ROLE_BUSINESS_INFORMATION_STEWARD | ROLE_WORKER_ADMINISTRATOR);

LRESULT CAddPerson1::OnWizardNext()
{
   if(m_Role.GetItemData(m_Role.GetCurSel()) == parent->ROLE_EMPLOYEE) {
      SECURITY_CHECK(ROLE_WORKER_ADMINISTRATOR |
ROLE_BUSINESS_INFORMATION_STEWARD ) -1;
   } else if(m_Role.GetItemData(m_Role.GetCurSel()) == parent->ROLE_CONTINGENT)
{
      SECURITY_CHECK(ROLE_CONTINGENT_WORKER_ADMINISTRATOR |
```

```
ROLE_BUSINESS_INFORMATION_STEWARD | ROLE_WORKER_ADMINISTRATOR) -1;
   }
...
```

Anyways, that's how I've used them, and I'm not sure how this could have been helped with templates... Other than that, I try to avoid them, unless REALLY necessary.

Share

Improve this answer

Follow

edited Feb 27, 2009 at 20:02

community wiki
3 revs, 2 users 93%
LarryF

2

I use macros to easily define Exceptions:

```
DEF_EXCEPTION(RessourceNotFound, "Ressource not found")
```

where DEF_EXCEPTION is

```
#define DEF_EXCEPTION(A, B) class A : public exception\
  {\
  public:\
    virtual const char* what() const throw()\
    {\
      return B;\
    };\
  }\
```

Share

Improve this answer

Follow

answered Jun 15, 2011 at 21:13

community wiki
MrBeast

1

If you have a list of fields that get used for a bunch of things, e.g. defining a structure, serializing that structure to/from some binary format, doing database inserts, etc, then you can (recursively!) use the preprocessor to avoid ever repeating your field list.

This is admittedly hideous. But maybe sometimes better than updating a long list of fields in multiple places? I've used this technique exactly once, and it was quite helpful that one time.

Of course the same general idea is used extensively in languages with proper reflection -- just instrospect the class and operate on each field in turn. Doing it in the C preprocessor is fragile, illegible, and not always portable. So I mention it with some trepidation. Nonetheless, here it is...

(EDIT: I see now that this is similar to what @Andrew Johnson said on 9/18; however the idea of recursively including the same file takes the idea a bit further.)

```
// file foo.h, defines class Foo and various members on it without ever
repeating the
// list of fields.

#if defined( FIELD_LIST )
   // here's the actual list of fields in the class.  If FIELD_LIST is defined,
we're at
   // the 3rd level of inclusion and somebody wants to actually use the field
list.  In order
   // to do so, they will have defined the macros STRING and INT before
including us.
   STRING( fooString )
   INT( barInt )
#else // defined( FIELD_LIST )

#if !defined(FOO_H)
#define FOO_H

#define DEFINE_STRUCT
// recursively include this same file to define class Foo
#include "foo.h"
#undef DEFINE_STRUCT

#define DEFINE_CLEAR
// recursively include this same file to define method Foo::clear
#include "foo.h"
#undef DEFINE_CLEAR

// etc ... many more interesting examples like serialization

#else // defined(FOO_H)
// from here on, we know that FOO_H was defined, in other words we're at the
second level of
// recursive inclusion, and the file is being used to make some particular
// use of the field list, for example defining the class or a single method of
it

#if defined( DEFINE_STRUCT )
#define STRING(a)  std::string a;
#define INT(a)     long a;
   class Foo
   {
      public:
#define FIELD_LIST
// recursively include the same file (for the third time!) to get fields
// This is going to translate into:
//    std::string fooString;
//    int barInt;
#include "foo.h"
#endif

      void clear();
   };
#undef STRING
#undef INT
#endif // defined(DEFINE_STRUCT)
```

```
#if defined( DEFINE_ZERO )
#define STRING(a) a = "";
#define INT(a) a = 0;
#define FIELD_LIST
   void Foo::clear()
   {
// recursively include the same file (for the third time!) to get fields.
// This is going to translate into:
//    fooString="";
//    barInt=0;
#include "foo.h"
#undef STRING
#undef int
   }
#endif // defined( DEFINE_ZERO )

// etc...


#endif // end else clause for defined( FOO_H )

#endif // end else clause for defined( FIELD_LIST )
```

Share

Improve this answer

Follow

answered Oct 3, 2008 at 2:16

community wiki
Eric

1

I've used the preprocesser to calculate fixed-point numbers from floating point values used in embedded systems that cannot use floating point in the compiled code. It's handy to have all of your math in Real World Units and not have to think about them in fixed-point.

Example:

```
// TICKS_PER_UNIT is defined in floating point to allow the conversions to
compute during compile-time.
#define TICKS_PER_UNIT  1024.0


// NOTE: The TICKS_PER_x_MS will produce constants in the preprocessor.  The
(long) cast will
//       guarantee there are no floating point values in the embedded code and
will produce a warning
//       if the constant is larger than the data type being stored to.
//       Adding 0.5 sec to the calculation forces rounding instead of
truncation.
#define TICKS_PER_1_MS( ms ) (long)( ( ( ms * TICKS_PER_UNIT ) / 1000 ) + 0.5 )
```

Share

Improve this answer

Follow

answered Aug 18, 2009 at 23:46

community wiki
dwj

1 This can be done with an inlined function – Motti Aug 19, 2009 at 7:01

Will inline functions use other inline functions and prevent floating point values from getting in the final code? The example above is quite simple but I've used this method for computing rotational velocity of a wheel through several gears with different ratios based on the counts per revolution of a motor. The macros define each level of conversion. – dwj Aug 19, 2009 at 15:57

@dwj • Yes, and yes. – Eljay Jun 20, 2022 at 14:12

1

Yet another foreach macros. T: type, c: container, i: iterator

```
#define foreach(T, c, i) for(T::iterator i=(c).begin(); i!=(c).end(); ++i)
#define foreach_const(T, c, i) for(T::const_iterator i=(c).begin(); i!=
(c).end(); ++i)
```

Usage (concept showing, not real):

```
void MultiplyEveryElementInList(std::list<int>& ints, int mul)
{
    foreach(std::list<int>, ints, i)
        (*i) *= mul;
}

int GetSumOfList(const std::list<int>& ints)
{
    int ret = 0;
    foreach_const(std::list<int>, ints, i)
        ret += *i;
    return ret;
}
```

Better implementations available: Google **"BOOST_FOREACH"**

Good articles available: **Conditional Love: FOREACH Redux** (Eric Niebler)
http://www.artima.com/cppsource/foreach.html

Share

Improve this answer

Follow

answered Feb 4, 2010 at 11:33

community wiki
Notinlist

1

Maybe the greates usage of macros is in platform-independent development. Think about cases of type inconsistency - with macros, you can simply use different header files -- like: --WIN_TYPES.H

```
typedef ...some struct
```

--POSIX_TYPES.h

```
typedef ...some another struct
```

--program.h

```
#ifdef WIN32
#define TYPES_H "WINTYPES.H"
#else
#define TYPES_H "POSIX_TYPES.H"
#endif

#include TYPES_H
```

Much readable than implementing it in other ways, to my opinion.

Share

Improve this answer

Follow

answered Apr 18, 2010 at 12:37

community wiki
rkellerm