7.14 — Ellipsis (and why to avoid them)

BY ALEX ON FEBRUARY 22ND, 2008 | LAST MODIFIED BY ALEX ON JANUARY 23RD, 2020

In all of the functions we've seen so far, the number of parameters a function will take must be known in advance (even if they have default values). However, there are certain cases where it can be useful to be able to pass a variable number of parameters to a function. C++ provides a special specifier known as ellipsis (aka "…") that allow us to do precisely this.

Because ellipsis are rarely used, potentially dangerous, and we recommend avoiding their use, this section can be considered optional reading.

Functions that use ellipsis take the form:

```
return_type function_name(argument_list, ...)
```

The *argument_list* is one or more normal function parameters. Note that functions that use ellipsis must have at least one non-ellipsis parameter. Any arguments passed to the function must match the argument_list parameters first.

The ellipsis (which are represented as three periods in a row) must always be the last parameter in the function. The ellipsis capture any additional arguments (if there are any). Though it is not quite accurate, it is conceptually useful to think of the ellipsis as an array that holds any additional parameters beyond those in the argument_list.

An ellipsis example

The best way to learn about ellipsis is by example. So let's write a simple program that uses ellipsis. Let's say we want to write a function that calculates the average of a bunch of integers. We'd do it like this:

```
#include <iostream>
#include <cstdarg> // needed to use ellipsis

// The ellipsis must be the last parameter
// count is how many additional arguments we're passing
double findAverage(int count, ...)
{
    double sum = 0;

    // We access the ellipsis through a va_list, so let's declare one
    va_list list;

    // We initialize the va_list using va_start.  The first parameter is
    // the list to initialize.  The second parameter is the last non-ellipsis
    // parameter.
    va_start(list, count);

    // Loop through all the ellipsis arguments
    for (int arg=0; arg < count; ++arg)
         // We use va_arg to get parameters out of our ellipsis
         // The first parameter is the va_list we're using
         // The second parameter is the type of the parameter
         sum += va_arg(list, int);

    // Cleanup the va_list when we're done.
    va_end(list);

    return sum / count;
}

int main()
{
    std::cout << findAverage(5, 1, 2, 3, 4, 5) << '\n';
    std::cout << findAverage(6, 1, 2, 3, 4, 5, 6) << '\n';
}
```

This code prints:

```
3
3.5
```

As you can see, this function takes a variable number of parameters! Now, let's take a look at the components that make up this example.

First, we have to include the cstdarg header. This header defines va_list, va_arg, va_start, and va_end, which are macros that we need to use to access the parameters that are part of the ellipsis.

We then declare our function that uses the ellipsis. Remember that the argument list must be one or more fixed parameters. In this case, we're passing in a single integer that tells us how many numbers to average. The ellipsis always comes last.

Note that the ellipsis parameter has no name! Instead, we access the values in the ellipsis through a special type known as va_list. It is conceptually useful to think of va_list as a pointer that points to the ellipsis array. First, we declare a va_list, which we've called "list" for simplicity.

The next thing we need to do is make list point to our ellipsis parameters. We do this by calling va_start(). va_start() takes two parameters: the va_list itself, and the name of the *last* non-ellipsis parameter in the function. Once va_start() has been called, va_list points to the first parameter in the ellipsis.

To get the value of the parameter that va_list currently points to, we use va_arg(). va_arg() also takes two parameters: the va_list itself, and the type of the parameter we're trying to access. Note that va_arg() also moves the va_list to the next parameter in the ellipsis!

Finally, to clean up when we are done, we call va_end(), with va_list as the parameter.

Note that va_start() can be called again any time we want to reset the va_list to point to the first parameter in the ellipses again.

Why ellipsis are dangerous: Type checking is suspended

Ellipsis offer the programmer a lot of flexibility to implement functions that can take a variable number of parameters. However, this flexibility comes with some downsides.

With regular function parameters, the compiler uses type checking to ensure the types of the function arguments match the types of the function parameters (or can be implicitly converted so they match). This helps ensure you don't pass a function an integer when it was expecting a string, or vice versa. However, note that ellipsis parameters have no type declarations. When using ellipsis, the compiler completely suspends type checking for ellipsis parameters. This means it is possible to send arguments of any type to the ellipsis! However, the downside is that the compiler will no longer be able to warn you if you call the function with ellipsis arguments that do not make sense. When using the ellipsis, it is completely up to the caller to ensure the function is called with ellipsis arguments that the function can handle. Obviously that leaves quite a bit of room for error (especially if the caller wasn't the one who wrote the function).

Lets look at an example of a mistake that is pretty subtle:

```
    std::cout << findAverage(6, 1.0, 2, 3, 4, 5, 6) << '\n';
```

Although this may look harmless enough at first glance, note that the second argument (the first ellipsis argument) is a double instead of an integer. This compiles fine, and produces a somewhat surprising result:

```
1.78782e+008
```

which is a REALLY big number. How did this happen?

As you have learned in previous lessons, a computer stores all data as a sequence of bits. A variable’s type tells the computer how to translate that sequence of bits into a meaningful value. However, you just learned that the ellipsis throw away the variable’s type! Consequently, the only way to get a meaningful value back from the ellipsis is to manually tell va_arg() what the expected type of the next parameter is. This is what the second parameter of va_arg() does. If the actual parameter type doesn’t match the expected parameter type, bad things will usually happen.

In the above findAverage program, we told va_arg() that our variables are all expected to have a type of int. Consequently, each call to va_arg() will return the next sequence of bits translated as an integer.

In this case, the problem is that the double we passed in as the first ellipsis argument is 8 bytes, whereas va_arg(list, int) will only return 4 bytes of data with each call. Consequently, the first call to va_arg will only read the first 4 bytes of the double (producing a garbage result), and the second call to va_arg will read the second 4 bytes of the double (producing another garbage result). Thus, our overall result is garbage.

Because type checking is suspended, the compiler won’t even complain if we do something completely ridiculous, like this:

```cpp
    int value = 7;
    std::cout << findAverage(6, 1.0, 2, "Hello, world!", 'G', &value, &findAverage) << '\n';
```

Believe it or not, this actually compiles just fine, and produces the following result on the author’s machine:

```
1.79766e+008
```

This result epitomizes the phrase, “Garbage in, garbage out”, which is a popular computer science phrase “used primarily to call attention to the fact that computers, unlike humans, will unquestioningly process the most nonsensical of input data and produce nonsensical output” (**Wikipedia**).

So, in summary, type checking on the parameters is suspended, and we have to trust the caller to pass in the right type of parameters. If they don’t, the compiler won’t complain -- our program will just produce garbage (or maybe crash).

Why ellipsis are dangerous: ellipsis don’t know how many parameters were passed

Not only do the ellipsis throw away the *type* of the parameters, it also throws away the *number* of parameters in the ellipsis. This means we have to devise our own solution for keeping track of the number of parameters passed into the ellipsis. Typically, this is done in one of three ways.

Method 1: Pass a length parameter

Method #1 is to have one of the fixed parameters represent the number of optional parameters passed. This is the solution we use in the findAverage() example above.

However, even here we run into trouble. For example, consider the following call:

For example:

```cpp
    std::cout << findAverage(6, 1, 2, 3, 4, 5) << '\n';
```

On the author’s machine at the time of writing, this produced the result:

```
699773
```

What happened? We told findAverage() we were going to give it 6 values, but we only gave it 5. Consequently, the first five values that va_arg() returns were the ones we passed in. The 6th value it returns was a garbage value somewhere in the stack. Consequently, we got a garbage answer. At least in this case it was fairly obvious that this is a garbage value.

A more insidious case:

```
    std::cout << findAverage(6, 1, 2, 3, 4, 5, 6, 7) << '\n';
```

This produces the answer 3.5, which may look correct at first glance, but omits the last number in the average, because we only told it we were going to provide 6 parameters (and then provided 7). These kind of mistakes can be very hard to catch.

Method 2: Use a sentinel value

Method #2 is to use a sentinel value. A **sentinel** is a special value that is used to terminate a loop when it is encountered. For example, with strings, the null terminator is used as a sentinel value to denote the end of the string. With ellipsis, the sentinel is typically passed in as the last parameter. Here's an example of findAverage() rewritten to use a sentinel value of -1:

```
#include <iostream>
#include <cstdarg> // needed to use ellipsis

// The ellipsis must be the last parameter
double findAverage(int first, ...)
{
    // We have to deal with the first number specially
    double sum = first;

    // We access the ellipsis through a va_list, so let's declare one
    va_list list;

    // We initialize the va_list using va_start.  The first parameter is
    // the list to initialize.  The second parameter is the last non-ellipsis
    // parameter.
    va_start(list, first);

    int count = 1;
    // Loop indefinitely
    while (1)
    {
        // We use va_arg to get parameters out of our ellipsis
        // The first parameter is the va_list we're using
        // The second parameter is the type of the parameter
        int arg = va_arg(list, int);

        // If this parameter is our sentinel value, stop looping
        if (arg == -1)
            break;

        sum += arg;
        count++;
    }

    // Cleanup the va_list when we're done.
    va_end(list);

    return sum / count;
}

int main()
{
    std::cout << findAverage(1, 2, 3, 4, 5, -1) << '\n';
    std::cout << findAverage(1, 2, 3, 4, 5, 6, -1) << '\n';
}
```

Note that we no longer need to pass an explicit length as the first parameter. Instead, we pass a sentinel value as the last parameter.

However, there are a couple of challenges here. First, C++ requires that we pass at least one fixed parameter. In the previous example, this was our count variable. In this example, the first value is actually part of the numbers to be averaged. So instead of treating the first value to be averaged as part of the ellipsis parameters, we explicitly declare it as a normal parameter. We then need special handling for it inside the function (in this case, we set sum to first instead of 0 to start).

Second, this requires the user to pass in the sentinel as the last value. If the user forgets to pass in the sentinel value (or passes in the wrong value), the function will loop continuously until it runs into garbage that matches the sentinel (or crashes).

Finally, note that we've chosen -1 as our sentinel. That's fine if we only wanted to find the average of positive numbers, but what if we wanted to include negative numbers? Sentinel values only work well if there is a value that falls outside the valid set of values for the problem you are trying to solve.

Method 3: Use a decoder string

Method #3 involves passing a "decoder string" that tells the program how to interpret the parameters.

```
#include <iostream>
#include <string>
#include <cstdarg> // needed to use ellipsis

// The ellipsis must be the last parameter
double findAverage(std::string decoder, ...)
{
    double sum = 0;

    // We access the ellipsis through a va_list, so let's declare one
    va_list list;

    // We initialize the va_list using va_start.  The first parameter is
    // the list to initialize.  The second parameter is the last non-ellipsis
    // parameter.
    va_start(list, decoder);

    int count = 0;
    // Loop indefinitely
    while (1)
    {
        char codetype = decoder[count];
        switch (codetype)
        {
        default:
        case '\0':
            // Cleanup the va_list when we're done.
            va_end(list);
            return sum / count;

        case 'i':
            sum += va_arg(list, int);
```

```
                count++;
                break;

            case 'd':
                sum += va_arg(list, double);
                count++;
                break;
            }
        }
}


int main()
{
    std::cout << findAverage("iiiii", 1, 2, 3, 4, 5) << '\n';
    std::cout << findAverage("iiiiii", 1, 2, 3, 4, 5, 6) << '\n';
    std::cout << findAverage("iiddi", 1, 2, 3.5, 4.5, 5) << '\n';
}
```

In this example, we pass a string that encodes both the number of optional variables and their types. The cool thing is that this lets us deal with parameters of different types. However, this method has downsides as well: the decoder string can be a bit cryptic, and if the number or types of the optional parameters don't match the decoder string precisely, bad things can happen.

For those of you coming from C, this is what printf does!

Recommendations for safer use of ellipsis

First, if possible, do not use ellipsis at all! Oftentimes, other reasonable solutions are available, even if they require slightly more work. For example, in our findAverage() program, we could have passed in a dynamically sized array of integers instead. This would have provided both strong type checking (to make sure the caller doesn't try to do something nonsensical) while preserving the ability to pass a variable number of integers to be averaged.

Second, if you do use ellipsis, do not mix expected argument types within your ellipsis if possible. Doing so vastly increases the possibility of the caller inadvertently passing in data of the wrong type and va_arg() producing a garbage result.

Third, using a count parameter or decoder string as part of the argument list is generally safer than using a sentinel as an ellipsis parameter. This forces the user to pick an appropriate value for the count/decoder parameter, which ensures the ellipsis loop will terminate after a reasonable number of iterations even if it produces a garbage value.

7.15 -- Introduction to lambdas (anonymous functions)

Index

7.13 -- Command line arguments

C++ TUTORIAL | PRINT THIS POST

56 comments to 7.14 — Ellipsis (and why to avoid them)

Vir1oN
June 27, 2019 at 11:53 pm · Reply

Hi

In your site`s index section (where there`s a list of key words and related chapter`s links) it`s written "Sentinel value, 7.13".
However, sentinel values are only mentioned here for the first time, so you might want to fix that to "Sentinel value, 7.14"

Alex
June 28, 2019 at 3:04 pm · Reply

Fixed, thanks!

cdecde57
June 13, 2019 at 2:22 pm · Reply

I am just curious. The lesson explains that this is an optional lesson and that using dynamically allocated arrays or things like vectors are a good substitute for ellipses. With that, I am looking into learning c++ to the best of my ability so I can like develop 2d games and maybe even 3d ones someday. I am just wondering if this lesson / learning about ellipses is important to this type of development.

nascardriver
June 14, 2019 at 1:04 am · Reply

No, you don't need this anymore unless you want to learn C at some point.
Varargs can be replaced by parameter packs. Have a look at those after you learned about templates (Parameter packs are not covered by learncpp).

cdecde57

June 14, 2019 at 6:48 am · Reply

Ok, thanks!

NXPY
March 26, 2019 at 12:25 am · Reply

I think you should mention that the pointer set through va_start() doesn't reset itself .

For example, in this code :

```
#include<iostream>
#include<cstdarg>

void func(int count , ...)
{
    va_list list ;
    va_start(list,count) ; /// Set pointer to first element
    for(int i{0} ; i<count ; ++i)
        {
            std::cout << va_arg(list,int) ;


        }
        /// Pointer doesn't reset , Garbage value will be output here
    for(int i{0} ; i<count ; ++i)
        {
            std::cout << va_arg(list,int) ;


        }
        va_end(list) ;

}
int main()
{
    func(4,2,3,4,5) ;
}
```

The second loop prints garbage value .
So instead of that, we should reset the pointer ourselves by calling va_start()again .

```
#include<iostream>
#include<cstdarg>

void func(int count , ...)
{
    va_list list ;
    va_start(list,count) ; /// Set pointer to first element
    for(int i{0} ; i<count ; ++i)
        {
            std::cout << va_arg(list,int) ;


        }
      ///Reseting here ====================================================
    va_start(list,count) ; /// Reset the pointer to the first element again
     ///Reseting here =====================================================

    for(int i{0} ; i<count ; ++i)
```

```
        {
            std::cout << va_arg(list,int) ;


        }
        va_end(list) ;

}
int main()
{
    func(4,2,3,4,5) ;
}
```

Alex
March 26, 2019 at 5:08 pm · Reply

Post updated to note that va_start can be used in this way.

David
March 8, 2019 at 9:32 pm · Reply

Thanks for these tutorials Alex they're a huge help!
Nascardriver, if you happen to see this, do you have a website or somewhere that you publish programming tips or tutorials? You seem very experienced in C++, thanks for taking the time to answer so many questions.

nascardriver
March 9, 2019 at 9:19 am · Reply

Hi David!

I don't publish any tips or tutorials myself. I'll gladly read your quiz solutions and questions here on learncpp to make sure you have a solid base to work with.
Once you get to the end, there shouldn't be too many things left for you to do wrong :)

Jeroen P. Broks
February 18, 2019 at 11:45 am · Reply

And then to think that Go and Lua encourage the use of these (Although Lua has no types (well, not entirely true, but you cannot declare a type and force the user to use that), and Go can allow you to give up a type. And Go did not want to allow this syntax "func myfunc(a int=2)" but on some Go forums they actually advice to use the "..." method in stead if you want arguments to be optional (one of the things I hate about Go).

However if an 'infinite' number of values is needed like in an average calculation routine, is a vector acceptable in stead:

```
#include <iostream>
#include <vector>
int average(std::vector<int> numbers) {
   int total{0};
   int avg;
   for (int i : numbers) total +=i;
   avg = total/numbers.size();
   return avg;
}
```

```
10  // Now I chose ints, although doubles could be a better choice, but that was not my prime c
11
12  int main(){
13      std::cout << average({1,2,3,4,5,6,7,8,9,10}) << '\n';
14  }
```

Or is this considered 'dirty code'?

nascardriver
February 19, 2019 at 7:50 am · Reply

* Line 5: Initialize your variables with brace initializers.
* Line 10: Limit your lines to 80 characters in length for better readability on small displays.
* Enable compiler warnings, read them, fix them.
* @main: Missing return statement.
* @numbers should be a const reference.

That's fine.
You might later learn about parameter packs (I'm not sure if they're covered by learncpp). Those are similar to ellipsis, but they're ok to use:

```
template <class... T>
int average(T... numbers)
{
  return static_cast<int>((numbers + ...) / static_cast<int>(sizeof...(numbers)));
}

// Call like before
average(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
```

Alex
February 19, 2019 at 7:59 pm · Reply

Parameter packs aren't covered yet. I'll add a lesson when I update chapter 7.

Jeroen P. Broks
February 21, 2019 at 2:11 am · Reply

Ah, yeah, that code was just typed too quickly (a rather bad habit of mine in all languages I used so far, especially with demo code), but you answered my prime concern... And those parameter packs look interesting...

hellmet
February 6, 2020 at 10:26 am · Reply

Wow... Wow those parameter packs... I saw them last when I asked a question on perfect forwarding. Any chance they are related. God where do I learn these (pls send book names). This is some mind bending code in the headers using variadic templates.
God I think it's going to take a lifetime to learn C++ and actually use it well.

I think this can be used to implement a typesafe printf?

Digging deeper, damn using constexpr might make this 'recursive descent' into the base case a single final template function call! That is ... that's mind-blowing!

I'll try to test this and update.

nascardriver
February 7, 2020 at 8:36 am · Reply

Parameter packs are great. Perfect forwarding uses them too.

> where do I learn these
Read the article on cppreference, look at examples, play around.

> I think this can be used to implement a typesafe printf?
You probably could, but formatting strings is easier if you can place the variables right where they're printed, and not after the string.

```
template <class... T>
void print(const T&... args)
{
  // Fold expressions make parameter packs even better
  ((std::cout << args), ...);
}

print("hello", ' ', 123); // hello 123
```

On the same note, C++20 has `std::format`.

pete
June 25, 2018 at 11:10 pm · Reply

Hi Alex,
once again, thank you for these amazing tutorials!
you've mantiond that "va_arg() also moves the va_list to the next parameter in the ellipsis!"
yet va_arg() does not seem to use any itrator or something of this kind.
can you give a short explantion on how does it moves after each iteration?
thanks!

nascardriver
June 26, 2018 at 8:50 am · Reply

Hi pete!

"Each call to this macro modifies the state of ap so that the next call to this macro will expand to the argument that follows the one it evaluates to."
Where "ap" is the list.

The @va_list knows what the next argument is. Since you're passing the list to @va_arg, @va_arg knows the next argument too.

References
* va_arg - http://www.cplusplus.com/reference/cstdarg/va_arg/

pete
June 27, 2018 at 9:33 pm · Reply

ok, got it.
thanks nascardriver!

Ishak
July 8, 2018 at 11:55 pm · Reply

```
for (int arg=0; arg < count; ++arg)
```

```
sum += va_arg(list, int);
```

can you please explain why we use ++arg ? I thought it was used to push the list forward?

nascardriver
July 9, 2018 at 2:04 am · Reply

If you have n arguments you can call @va_arg n times and it will return the next argument every time.
In this case there are @arg arguments and you want to have all of them, so you have to call @va_arg @arg times.

Trevor
February 20, 2018 at 2:18 am · Reply

You wrote:
First, we have to include the cstdarg header. This header defines va_list, va_start, and va_end, which are macros that we need to use to access the parameters that are part of the ellipsis.

Is va_arg also a macro that is needed?

Trevor

Alex
February 22, 2018 at 9:10 am · Reply

Yes, lesson updated to include va_arg in the list of macros that cstdarg includes.

Marcus
February 14, 2018 at 7:55 am · Reply

But what about using it like this?
https://docs.microsoft.com/da-dk/cpp/cpp/ellipses-and-variadic-templates/

nascardriver
February 14, 2018 at 11:45 am · Reply

Hi Marcus!
Those are a different story, I don't think there's a tutorial about them yet.
Those are good, the ones here are bad.

Alex
February 15, 2018 at 5:44 pm · Reply

Correct, that's a different use of the ellipses for a different purpose. The main difference is the ellipses used in variadic templates are type safe, whereas these are not.

Mattermonkey
November 28, 2017 at 2:21 pm · Reply

So, this line right here has me wondering what's possible in this language:

```
sum += va_arg(list, int);
```

Can you pass types as arguments to functions? Can I define my own functions that take types as arguments? I could do it with templates and a dummy variable (just pass a var of the desired type), but that's weird and memory inefficient.

Alex
November 29, 2017 at 5:03 pm · Reply

> Can you pass types as arguments to functions?

No, you can only pass values. However, if you have a way to map values to types, then you can pass a value and translate it into a type.

Mattermonkey
December 4, 2017 at 9:10 am · Reply

But then what does the function prototype for va_arg look like?
Is int technically just the name of a variable declared in the cstdarg header somewhere, which it maps to a value?

Alex
December 6, 2017 at 9:18 pm · Reply

I see what you're getting at now.

va_arg isn't a real function. It's a function-like macro. With function-like macros, the arguments from the call are text substituted by the preprocessor in the function-like macro body, and then that code is compiled. In this case, the "type" parameter is substituted into the return value for the expression that the macro expands to.

InsideCoder
June 18, 2017 at 1:30 am · Reply

First of all, I wanted to thank for these awesome tutorials. They have definitely helped me a lot.
Second, I was wondering how the switch statement is able to terminate the while loop by returning

```
return sum/count;
```

.

Alex
June 18, 2017 at 8:22 pm · Reply

A return statement always terminates the entire function immediately, regardless of whether you're inside a loop or nested block.

Mike
February 27, 2017 at 2:27 pm · Reply

Hi Alex,
can you tell wheter the use of the "auto" type in ellipsis is a) allowed and b) useful/recommended?
Thank and props for your great tutorial!

Alex
February 27, 2017 at 4:55 pm · Reply

a) No. b) No. :)

Urio
February 14, 2017 at 7:35 pm · Reply

I'm amazed now that I understand how printf works, thanks a lot for mentioning that. It's fuckin geinus!
I've known the site for a long time, yet I've just now got time to actually commit to it.
Thank you very much, the tutorial is definitely the best one I've encountered anywhere (even better than at my university). I've learned so much, so well, so fast.
Keep up the good work!

Christopher
January 10, 2017 at 11:39 am · Reply

Big thanks for creating this comprehensive tutorial! I've learnt a lot in quite a short amount of time!
I was just wondering when we begin to learn things that can help with video game design, like creating levels and what not.
Thanks! :)

Alex
January 10, 2017 at 5:43 pm · Reply

This isn't a video game design tutorial, it's a C++ fundamentals tutorial. I don't (currently) cover anything about UI or game design, but the topics you learn here would certainly be useful in building your own gaming engines.

Christopher
January 11, 2017 at 11:46 am · Reply

Alright, thanks!

J3ANP3T3R
May 23, 2016 at 4:25 pm · Reply

what are those va_ ?

Alex
May 23, 2016 at 4:41 pm · Reply

va_list is a data type, and va_start and va_arg are functions.

"va" standards for "variable arguments".

Yueiqng Zhang
May 20, 2016 at 4:51 pm · Reply

You need std::cout

Alex
May 21, 2016 at 11:10 am · Reply

Updated. Thanks for pointing that out.

Shiva
May 12, 2016 at 2:35 am · Reply

Typos:

* Title - not that it matters, but is it correct to say why to in English?
* We then declare our function that uses the ellipse. (ellipsis)
* For example, with strings, the null terminal (terminator) is used...
* 'Wikipedia' link text - 'W' is usually capitalised. (Funny paragraph and nice article, BTW :D)

Nice lesson! Thanks for the note about printf too. I've always wondered how printf-like functions were implemented. :)

Alex
May 12, 2016 at 4:41 pm · Reply

Yes, "why to" is proper English, if not particularly common. Other errors fixed! Thanks for pointing them out.

Shiva
May 13, 2016 at 1:29 am · Reply

Er, I just noticed, there are seven more *ellipse-instead-of-ellipsis* typos on this page. Just search for *ellipse*. ;)

Alex
May 13, 2016 at 2:17 pm · Reply

I only found six. :) Thanks for the heads up, they're fixed now.

Reaversword
December 16, 2015 at 6:38 am · Reply

I'd like to share my point of view about ellipsis.

In my opinion, its absolutely necessary, but only in the case when you don't know what kind (type) of data you are about to receive.
The "other" case, don't know how much elements of the same type you gonna receive, can be smartly solved with an array or a vector, cause they haves a fixed type.

I used it in the past, in other language (action script 3, flash), but AS3 didn't "lose" the type of the element (and as a consecuence, number of elements either), so it was just a matter of compare possible (expected) types with some "ifs" sentences and act consecuently.

So, in C++ the big, unsafe problem is that the complete ellipsis itself is a long bunch of bits, ready to be correctly interpreted, or awfully mistaken! ...so its a real "oh-shit-now-what". And this fact makes ellipsis, with no doubt, absolutely dangerous. More even if we realize some types haves variable sizes (as strings, for example), and if I'm not wrong, they not even have a terminator (''). So irremediably we need to include some way to interpret the next "bunch of bits", maybe using some "characters coding" as in third example, or perhaps interleaving it between values ('i', 35, 'd', 56.7, etc...).

I still am unable to figure out how to "cut" the "rope of bits" to be able to get a string, since I don't know how to say "va_arg(list, 16 bytes string)". I can include its size in sending, but I don't know how to use this information, to be honest.

Besides, you can't access previous "bunch of bits" once we've used "va_arg(list, int)", you need to use va_end(list) to re-start, among other things, because there is no "array" or "vector" class able to contain "multi-type" elements.

Almost beautiful how something so wonder, flexible and useful as ellipsis can turn in a so evil problem.

To finish, I must say that an ellipsis function is not for the final user, but for very controlled environment, usually sending and receiving our own classes types, of very internal hidden-to-the-user processes as this example:

In this program, the function takes objects (classes instances) and creates a bokeh (https://en.wikipedia.org/wiki/Bokeh) shape depending of some factors, so "fixed" parameters of that function was things as "cameraDof", "focalLengh", "bokehAnisotropy", "numberOfBlades", etc... but finally, in the ellipsis arrived all objects able to provoke a bokeh, different type ones, with different parameters to consider.

So perhaps a single shape (as circle or square) haves only color, depth, and other ones, as images was a compound of several bokehs for the brightest pixels of that image the program needs to decompose, and something similar did happen with letters, the pink ones showed pink hearts instead of pink circles as a "bokeh shape"!.

You never know how many and what types there was in screen, but thanks to ellipsis it was possible to manage them all with a single function.

As you imagine, this is what I mean to say with "internal processes".

Of course, if a parter gonna touch the code, its mandatory to document the ellipsis function properly (although studying cases to catch the types can provide some idea).

The Welder
February 26, 2015 at 9:38 am · Reply

I totally disagree with this entire discussion. For instance, I wrote a log function in which you can specify a string containing placeholders and a variable number of arguments of different types. (I guess it's exactly like a printf). Now since I was forced to use Visual Studio 2012, variadic templates weren't an option for me. This function had to be as flexible as possible.

Now whilst I understand the point of using a number to define the number of arguments or even a sentinel at the end of the arguments, that really doesn't have any real meaning when you're dealing with varying argument types, apart from knowing how many arguments you're expecting. (There are two caveats here, firstly why would you want to place that kind of added restriction on the programmer of having to pass an extra argument and secondly and most importantly, there's no way to be sure that this count value actually *IS* the count of the following arguments. In fact, it's as flimsy and error prone as accidently passing the incorrect variable type to a placeholder)

Of course, I appreciate the concept of type safety, but when using my Log function, if you specify the wrong type you'll just get garbage out. So it's down to the programmer to get it right. Afterall, he'd also have written the formatted string also.

If you're creating a library with a function call like I was, it's not exactly sensible write a function that takes say a vector reference (if you're lucky enough to have a collection of the same type) or even tuple. It's about using ellipsis properly and understanding the implications. Everybody at one time or another gets their arguments in the wrong order and it's less of a problem than having to use sentinels and add extra parameters etc.

I love ellipsis and the flexibility they give you far out-weighs any problems you might get my accidently fluffing up your argument order.

Alex
December 3, 2015 at 5:18 pm · Reply

You disagree with the entire discussion (including all of the points describing why they are dangerous) because you found a use case where using ellipsis was probably the best option? That seems silly. You should agree with the discussion and then use them anyway. :)

earl_k
April 24, 2014 at 10:39 am · Reply

So, do we conclude that the use of ellipsis in creating functions with variable list of parameters really dangerous? Personally, if I will base my conclusion on the author's article, I will still use the ellipsis as far as it offers me convenience and flexibility because the dangers cited here are all due to the recklessness of the user calling the function (and stupidity if you are the one who created that specific function and you don't know what you are doing).

```
cout << FindAverage(6, 1, 2, 3, 4, 5) << endl;
```

And if I was the one who created this function and let the public use it without me explaining how to correctly use it, then that's irresponsibility.

Calling prinf and its cousins without really understanding it, well, I no longer know what you are.

But anyway, with or without ellipsis, misuse of functions really has its perils.

Alex
December 3, 2015 at 5:15 pm · Reply

Yes, it's really dangerous. That doesn't mean you shouldn't do it, especially if it's the best solution for the challenge you're facing! Being aware of the potentially dangers, and where they typically occur can be helpful in avoiding them.

AndresJak
March 29, 2011 at 9:51 am · Reply

Why is it when I use big numbers, for example FindAvarage(71,73,85), or any other big numbers it gives me bizarre answers every time(first i thought it was my programs code was wrong, but than i used your code and it still gave me weird answers) i run the program or, or i missed something you mentioned.

abhishekchauhan
October 27, 2011 at 11:53 pm · Reply

I know this is a very late reply, but here's the answer for benefit of those who are visiting this page for the first time...

The first argument to your function FindAverage() should have been the number of variable list arguments. You've passed 71 there.

This means, the program will keep on looking at 71*sizeof(int) bytes of memory and will do your job, the exact issue pointed out in this article.

Really guys, don't use "ellipsed" functions as far as possible. Try to use the concept of passing an array of pointers instead(the way they are passed in main()).

Ben
April 2, 2009 at 2:16 am · Reply

I think it might be worth mentioning the ellipses' value in formatted-string functions such as the printf() family.
It would be ideal for implementing a date() function similar to that in PHP, for instance, where the number and type of parameters is explicitly defined in the format string. Obviously this is also dangerous (possibly even more so) but it would demonstrate handling mixed-type variable arguments. Of course it might also demonstrate How To Break Your Stack (TM).

Alex
May 1, 2009 at 8:37 pm · Reply

Yeah, I didn't want to encourage people to try to clone printf().

Tom
March 24, 2008 at 9:59 am · Reply

Hello Alex -

Interesting. Could you elaborate on what is meant by:

"Finally, to clean up when we are done, we call va_end(), with va_list as the parameter." ??

What does va_end() actually do, does it just reset the pointer into the list to the default value? (Wouldn't that be the same as resetting the pointer va_list to 0? Or, is there more to it?) What is the practical effect of calling va_end()? If we reference the list pointed to by va_list after calling va_end(), do we just get the first element in the ellipses again?

For example:

```
va_end();
va_start(list, nCount);
int nX = va_arg(list, int);
```

Would that set nX equal to the first element of the ellipses?

Alex
March 24, 2008 at 2:34 pm · Reply

My understanding (and I may be wrong about this) is that the implemention of va_start() and va_args() is left up to the compiler. If that's actually the case, then va_end() could do any necessary cleanup.

I looked at how va_end() was implemented in Microsoft Visual Studio and this is how it is defined:

```
#define va_end(ap) ap = (va_list)0
```

As you can see, it's actually a macro function that just sets the ap parameter to 0. So, at least with Microsoft Visual Studio, there's no real practical effect of calling va_end(), outside of maybe NULLing your list in case you inadvertently try to use it again without calling va_start().

In your example, you'd have to pass list into va_end(), but va_start() should cause the list to start at the beginning of the ellipses again -- so yes, nX would be the first element of the ellipses.