11.12 — Ellipsis (and why to avoid them)

ALEX JULY 15, 2021

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:

```cpp
#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:

```
1 | 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:

```
1 | int value{ 7 };
  | std::cout << findAverage(6, 1.0, 2, "Hello, world!", 'G', &value, &findAverage) <<
2 | '\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:

```
1 | 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:

```
1 | 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:

```cpp
#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{ static_cast<double>(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 (true)
 {
  // 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.

```cpp
#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 (true)
 {
  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.

For advanced readers

To improve upon ellipses-like functionality, C++11 introduced `parameter packs` and `variadic templates`, which offers functionality similar to ellipses, but with strong type checking. However, significant usability challenges impeded adoption of this feature.

In C++17, `fold expressions` were added, which significantly improves the usability of parameter packs, to the point where they are now a viable option.

We hope to introduce lessons on these topics in a future site update.

Next lesson
11.13 Introduction to lambdas (anonymous functions)

Back to table of contents

Previous lesson
11.11 Command line arguments

Notify me about replies: POST COMMENT

91 COMMENTS

Newest