Returning Multiple Values from Functions in C++

https://dzone.com/articles/returning-multiple-values-from-functions-in-c

Eli Bendersky

Returning Multiple Values from Functions in C++

In this day and age, programmers often need to return multiple values from functions in C++. Author Eli Bendersky provides an overview of some of the options available to accomplish this feat, along with a peek at what's in store for this necessity in C++.


by
Eli Bendersky
·
Mar. 09, 16 · Performance Zone · Analysis

Since C++ has no built-in syntax for returning multiple values from functions and methods, programmers have been using a number of techniques to simulate this when needed, and the number has grown since the introduction of C++11. In this post, I want to provide an overview of some of the options we have today for returning multiple values from functions, and possible future directions in the language.

Introduction - Why Multiple Return Values?

Multiple return values from functions are not a new concept in programming - some old and venerable languages like Common Lisp have had them since the early 1980s.

There are many scenarios where multiple return values are useful:

First and foremost, for functions that naturally have more than one value to compute. For example, the Common Lisp floor function computes the quotient *and* the remainder of its two operands, and returns both. Another example is std::minmax in C++11, that finds the minimal and the maximal value in a container simultaneously.

Second, multiple return values are helpful when the data structure the function operates on contains multiple values per entry. For example, Python 3's dict.items is an iterator over key / value pairs, and each iteration returns both, which is frequently useful. Similarly, in C++, the mapping family of containers provides iterators that hold key / value pairs, and methods like std::map::find logically return a pair, even though it's encapsulated in an iterator object. Another related, but slightly different example is Python's enumerate, which takes any sequence or iterator and returns index / value pairs - very useful for writing some kinds of for loops.

Third, the multiple return values may signal different "paths" - like error conditions or "not found" flags, in addition to actual values. In Go, map lookup returns a value / found pair, where "found" is a boolean flag saying whether the key was found in the map. In general, in Go, it's idiomatic to return a value / error pair from functions. This method is useful in C++ as well, and I'll cover an example in the next section.

Multiple return values are so convenient that programmers usually find ways to simulate them even in languages that don't support them directly. As for new programming languages, most of them come with this feature natively supported. Go, Swift, Clojure, Rust and Scala all support multiple return values.

Multiple Return Values in C++ with Output Parameters

Back to C++, let's start our quest with the oldest and possibly still most common method - using some of the function's parameters as "out" parameters. This method is made possible by C++ (based on C before it) making a strict distinction between parameters passed by value and by reference (or pointer) into functions. Parameters passed by pointers can be used to "return" values to the caller.

This technique has old roots in C, where it's used in many places in the standard library; for example fgets and fscanf. Many POSIX functions adopt the conventions of returning an integer "error code" (0 for success) while writing any output they have into an output parameter. Examples abound - gettimeofday, pthread_create... there are hundreds (or thousands). This has become such a common convention that some code-bases adopt a special marker for output parameters, either with a comment or a dummy macro. This is to distinguish by-pointer input parameters from output parameters in the function signature, thus signaling to the user which is which:

```
#define OUT
int myfunc(int input1, int* input2, OUT int* out) {
   ...
}
```

C++ employs this technique in the standard library as well. A good example is the std::getline function. Here's how we read everything from stdin and echo every line back with a prefix:

```
#include <iostream>
#include <string>
int main(int argc, const char** argv) {
  std::string line;
  while (std::getline(std::cin, line)) {
    std::cout << "echo: " << line << "\n";
  }
  return 0;
}
```

std::getline writes the line it has read into its second parameter. It returns the stream (the first parameter) since a C++ stream has interesting behavior in boolean context. It's true so long as everything is OK, but flips to false once an error occurs, or an end-of-file condition is reached. The latter is what the sample above uses to concisely invoke std::getline in the condition of a while loop.

C++'s introduction of reference types adds a choice over the C approach. Do we use pointers or references for output parameters? On one hand

references result in simpler syntax (if the line would have to be passed by the pointer in the code above, we'd have to use &line in the call) and also cannot be nullptr, which is important for output parameters. On the other hand, with references it is very hard to look at a call and discern which parameters are input and which are output. Also, the nullptr argument works both ways - occasionally it is useful to convey to the callee that some output is not needed and a nullptr in an output parameter is a common way to do this.

As a result, some coding guidelines recommend only using pointers for output parameters, while using const references for input parameters. But as with all issues of style, YMMV.

Whichever style you pick, this approach has obvious downsides:

- The output values are not uniform - some are returned, some are parameters, and it's not easy to know which parameters are for output. std::getline is simple enough, but when your function takes 4 and returns 3 values, things start getting hairy.
- Calls require declarations of output parameters beforehand (such as the line in the example above). This bloats the code.
- Worse, the separation of parameter declaration from its assignment within the function call can result in uninitialized variables in some cases. To analyze whether the line is initialized in the example above, one has to carefully understand the semantics of std::getline.

On the other hand, prior to the introduction of move semantics in C++11, this style had serious performance advantages over the alternatives, since it can avoid extra copying. I'll discuss this a bit later on in the article.

Pairs and Tuples

The std::pair type is a veteran in C++. It's used in a bunch of places in the standard library to do things like hold keys and values of mappings, or to hold "status, result" pairs. Here's an example that demonstrates both:

```
#include <iostream>
#include <unordered_map>
using map_int_to_string = std::unordered_map<int, std::string>;
void try_insert(map_int_to_string& m, int i, const std::string& s) {
  std::pair<map_int_to_string::iterator, bool> p = m.insert({i, s});
```

```
  if (p.second) {
    std::cout << "insertion succeeded. ";
  } else {
    std::cout << "insertion failed. ";
  }
  std::cout << "key=" << p.first->first << " value=" << p.first->second << "\
n";
}
int main(int argc, const char** argv) {
  std::unordered_map<int, std::string> mymap;
  mymap[1] = "one";
  try_insert(mymap, 2, "two");
  try_insert(mymap, 1, "one");
  return 0;
}
```

The std::unordered_map::insert method returns two values: an element iterator and a boolean flag saying whether the requested pair was inserted or not (it won't be inserted if the key already exists on the map). What makes the example really interesting is that there are *nested* multiple values being returned here. insert returns a std::pair. But the first element of the pair, the iterator, is just a thin wrapper over another pair - the key/value pair - hence the first->first and first->second accesses we use when printing the values out.

Thus, we also have an example of a shortcoming of std::pair - the obscureness of first and second, which requires us to always remember the relative positions of values within the pairs. p.first->second gets the job done but it's not exactly a paragon of readable code.

With C++11, we have an alternative - std::tie:

```
void try_insert_with_tie(map_int_to_string& m, int i, const std::string& s) {
  map_int_to_string::iterator iter;
  bool did_insert;
  std::tie(iter, did_insert) = m.insert({i, s});
  if (did_insert) {
    std::cout << "insertion succeeded. ";
  } else {
    std::cout << "insertion failed. ";
  }
  std::cout << "key=" << iter->first << " value=" << iter->second << "\n";
}
```

Now we can give the pair members readable names. The disadvantage of this approach is, of course, that we need the separate declarations that take extra space. Also, while in the original example we could use auto to infer the type of the pair (useful for really hairy iterators), here we have to declare them fully.

Pairs work for two return values, but sometimes we need more. C++11's introduction of variadic templates finally made it possible to add a generic tuple type into the standard library. A std::tuple is a generalization of a std::pair for multiple values. Here's an example:

```
std::tuple<int, std::string, float> create_a_tuple() {
  return std::make_tuple(20, std::string("baz"), 1.2f);
}
int main(int argc, const char** argv) {
  auto data = create_a_tuple();
  std::cout << "the int: " << std::get<0>(data) << "\n"
            << "the string: " << std::get<1>(data) << "\n"
            << "the float: " << std::get<2>(data) << "\n";
  return 0;
}
```

The std::get template is used to access tuple members. Again, this is not the friendliest syntax but we can alleviate it somewhat with std::tie:

```
int i;
std::string s;
float f;
std::tie(i, s, f) = create_a_tuple();
std::cout << "the int: " << i << "\n"
          << "the string: " << s << "\n"
          << "the float: " << f << "\n";
```

Another alternative is to use even more template metaprogramming magic to create a "named" tuple (similar to the Python namedtuple type). Here's an example. There are no standard solutions for this, though.

Structs

When faced with sophisticated "named tuple" implementations, old-timers snort and remind us that in the olden days of C, this problem already had a perfectly valid solution - a struct. Here's the last example rewritten using a struct:

```
struct RetVal {
  int inumber;
  std::string str;
  float fnumber;
};
RetVal create_a_struct() {
  return {20, std::string("baz"), 1.2f};
}
// ... usage
{
  // ...
  auto retvaldata = create_a_struct();
  std::cout << "the int: " << retvaldata.inumber << "\n"
```

```
            << "the string: " << retvaldata.str << "\n"
            << "the float: " << retvaldata.fnumber << "\n";
}
```

When the returned value is created, the syntax and nice an concise. We could even omit some of the fields if their default values are good enough (or the struct has constructors for partial field initialization). Also note how natural the access to the returned value's fields is: all fields have descriptive names - this is perfect! C99 went a step further here, allowing named initialization syntax for struct fields:

```
RetVal create_a_struct_named() {
  return {.inumber = 20, .str = std::string("baz"), .fnumber = 1.2f};
}
```

This is very useful for self-documenting code that doesn't force you to go peek into the RetVal type every time you want to decode a value. Unfortunately, even if your C++ compiler supports this, it's *not standard C++*, because C++ did not adopt the feature. Apparently there [was an active proposal to add it](), but it wasn't accepted; at least not yet.

The rationale of the C++ committee, AFAIU, is to prefer constructors to initialize struct fields. Still, since C++ functions don't have a named parameter ("keyword argument" in Python parlance) syntax, using ctors here wouldn't be more readable. What it would allow, though, is convenient non-zero default initialization values.

For example:

```
struct RetValInitialized {
  int inumber = 17;
  std::string str = "foobar";
  float fnumber = 2.24f;
};
RetValInitialized create_an_initialized_struct() {
  return {};
}
```

Or even fancier initialization patterns with a constructor:

```
struct RetValWithCtor {
  RetValWithCtor(int i)
    : inumber(i), str(i, 'x'), fnumber(i) {}
  int inumber;
  std::string str;
  float fnumber;
};
RetValWithCtor create_a_constructed_struct() {
  return {10};
}
```

This would also be a good place to briefly address the performance issue I mentioned earlier. In C++11, it's almost certain that structs returned by value will not actually be copied due to the return-value optimization mechanism. Neither will the std::string held by value within the struct be copied. For even more details, see section 12.8 of the C++11 standard, in the paragraph starting with:

> **When certain criteria are met, an implementation is allowed to omit the copy/move construction of a class object, even if the copy/move constructor and/or destructor for the object have side effects. In such cases, the implementation treats the source and target of the omitted copy/move operation as simply two different ways of referring to the same object, and the destruction of that object occurs at the later of the times when the two objects would have been destroyed without the optimization**

This mechanism is called *copy elision* by the standard.

Structured Bindings: A New Hope for C++17

Luckily, the C++ standard committee consists of brilliant folks who have already recognized that even though C++ has many ways to do multiple return values, none is really perfect. So there's a new proposal making the rounds now for the C++17 edition of the language, called Structured bindings.

In brief, the idea is to support a new syntax that will make tying results of tuple-returning functions easier. Recall from the discussion above that while tuples have a fairly convenient syntax returning them from functions, the situation on the receiving side is less than optimal with a choice between clunky std::get calls or pre-declaration and std::tie.

What the proposal puts forward is the following syntax for receiving the tuple returned by create_a_tuple:

```
auto {i, s, f} = create_a_tuple();
// Note: proposed C++17 code, doesn't compile yet
```

The types of i, s, and f are "auto"-inferred by the compiler from the return type of create_a_tuple. Moreover, a different enhancement of C++17 is to permit a shorter tuple creation syntax as well, removing the need for std::make_tuple and making it as concise as struct creation:

```
std::tuple<int, std::string, float> create_a_tuple() {
  return {20, std::string("baz"), 1.2f};
}
// Note: proposed C++17 code, doesn't compile yet
```

The structured bindings proposal is for returned struct values as well, not just tuples, so we'll be able to do this:

```
auto {i, s, f} = create_a_struct();
```

I sure hope this proposal will get accepted. It will make simple code pleasant to write and read, at no cost to the compiler and runtime.

Conclusion

So many possibilities, what to choose? Personally, since I believe code readability is more important than making it quick to compose, I like the explicit approach of wrapping multiple values in structs. When the returned values logically belong together, this is a great way to collect them in a natural self-documenting way. So this would be the approach I'd use most often.

That said, sometimes the two values returned really don't belong together in any logical sense - such as a stream and a string in the getline example. Littering the source code with one-off struct types named StreamAndResult or OutputAndStatus is far from ideal, so in these cases I'd actually consider a std::pair or a std::tuple.

It goes without saying that the proposed structured bindings in C++17 can make all of this even easier to write, making folks less averse to the current verboseness of tuples.