Returning multiple values from a C++ function

Asked 16 years ago Modified 1 year, 11 months ago Viewed 614k times

385

Is there a preferred way to return multiple values from a C++ function? For example, imagine a function that divides two integers and returns both the quotient and the remainder. One way I commonly see is to use reference parameters:

```
void divide(int dividend, int divisor, int& quotient, int& remainder);
```

A variation is to return one value and pass the other through a reference parameter:

```
int divide(int dividend, int divisor, int& remainder);
```

Another way would be to declare a struct to contain all of the results and return that:

```
struct divide_result {
    int quotient;
    int remainder;
};

divide_result divide(int dividend, int divisor);
```

Is one of these ways generally preferred, or are there other suggestions?

Edit: In the real-world code, there may be more than two results. They may also be of different types.

c++

Share
Improve this question
Follow

edited Nov 26, 2008 at 15:38

asked Nov 26, 2008 at 15:19
Fred Larson
62k ● 18 ● 116 ● 177

23 Answers

Sorted by: Highest score (default)

354

In C++11 you can:

```
#include <tuple>

std::tuple<int, int> divide(int dividend, int divisor) {
    return  std::make_tuple(dividend / divisor, dividend % divisor);
}

#include <iostream>

int main() {
    using namespace std;

    int quotient, remainder;

    tie(quotient, remainder) = divide(14, 3);

    cout << quotient << ',' << remainder << endl;
}
```

In C++17:

```
#include <tuple>

std::tuple<int, int> divide(int dividend, int divisor) {
    return  {dividend / divisor, dividend % divisor};
}

#include <iostream>

int main() {
    using namespace std;

    auto [quotient, remainder] = divide(14, 3);

    cout << quotient << ',' << remainder << endl;
}
```

or with structs:

```
auto divide(int dividend, int divisor) {
    struct result {int quotient; int remainder;};
    return result {dividend / divisor, dividend % divisor};
}

#include <iostream>

int main() {
    using namespace std;

    auto result = divide(14, 3);

    cout << result.quotient << ',' << result.remainder << endl;

    // or

    auto [quotient, remainder] = divide(14, 3);

    cout << quotient << ',' << remainder << endl;
}
```

Share
Improve this answer
Follow

edited Dec 8, 2018 at 14:57

answered May 13, 2013 at 6:51
oblitum
11.9k ● 6 ● 58 ● 121

8 I have one concern with functions returning tuples. Say the above function prototype is in a header, then how do I know what does the first and second returned values mean without understanding the function definition ? quotient-remainder or remainder-quotient. – Uchia Itachi Nov 4, 2016 at 7:22

22 @UchiaItachi Same concern for function parameters, you can give names to them, but the language doesn't even enforce that, and the parameters names have no value at call site when reading. Also, on a single return, you just have a type, but having the name could also be useful, with tuples you just double the issue, so imo, the language just lacks regarding being self-documented in several ways, not only this. – oblitum Jan 20, 2017 at 15:21

1 how would the last example look if I wanted to explicitely specify return type of divide()? Shall I then define result somewhere else, or I can define it right in return type specification? – Slava Jul 11, 2017 at 13:30

4 @Slava you can't define a type right at the function signature, you would have to declare the type outside and use it as return type, like it's normally done (just move the `struct` line outside the function body and replace `auto` function return with `result`. – oblitum Jul 11, 2017 at 15:43

4 @pepper_chico What if want to put the function definition of `divide` into a separate cpp file? I get the error `error: use of ‘auto divide(int, int)’ before deduction of ‘auto’`. How do I solve this? – Adriaan Feb 23, 2018 at 20:26

286

For returning two values I use a `std::pair` (usually typedef'd). You should look at `boost::tuple` (in C++11 and newer, there's `std::tuple`) for more than two return results.

With introduction of structured binding in C++ 17, returning `std::tuple` should probably become accepted standard.

Share
Improve this answer
Follow

edited Mar 26, 2018 at 19:06
SergeyA
62.5k ● 5 ● 84 ● 143

answered Nov 26, 2008 at 15:22
Rob
78.5k ● 57 ● 161 ● 199

18 +1 for tuple. Keep in mind the performance ramifications of large objects returning in a structure vs. passing by reference. – Marcin Nov 26, 2008 at 15:40

15 If you're going to use tuples, why not use them for pairs as well. Why have a special case? – Ferruccio Nov 26, 2008 at 15:44

4 Fred, yes boost::tuple can do that :) – Johannes Schaub - litb Nov 26, 2008 at 15:51

55 In C++11, you can use `std::tuple`. – Ferruccio Oct 20, 2011 at 10:32

17 If you want to accept multiple values from a function, a convenient way of doing this is to use `std::tie` stackoverflow.com/a/2573822/502144 – fdermishin May 12, 2014 at 8:44

157

Personally, I generally dislike return parameters for a number of reasons:

- it is not always obvious in the invocation which parameters are ins and which are outs
- you generally have to create a local variable to catch the result, while return values can be used inline (which may or may not be a good idea, but at least you have the option)
- it seems cleaner to me to have an "in door" and an "out door" to a function -- all the inputs go in here, all the outputs come out there
- I like to keep my argument lists as short as possible

I also have some reservations about the pair/tuple technique. Mainly, there is often no natural order to the return values. How is the reader of the code to know whether `result.first` is the quotient or the remainder? And the implementer could change the order, which would break existing code. This is especially insidious if the values are the same type so that no compiler error or warning would be generated. Actually, these arguments apply to return parameters as well.

Here's another code example, this one a bit less trivial:

```
pair<double,double> calculateResultingVelocity(double windSpeed, double
windAzimuth,
                                               double planeAirspeed, double
planeCourse);

pair<double,double> result = calculateResultingVelocity(25, 320, 280, 90);
cout << result.first << endl;
cout << result.second << endl;
```

Does this print groundspeed and course, or course and groundspeed? It's not obvious.

Compare to this:

```
struct Velocity {
    double speed;
    double azimuth;
};
Velocity calculateResultingVelocity(double windSpeed, double windAzimuth,
                                    double planeAirspeed, double planeCourse);

Velocity result = calculateResultingVelocity(25, 320, 280, 90);
```

```
cout << result.speed << endl;
cout << result.azimuth << endl;
```

I think this is clearer.

So I think my first choice, in general, is the struct technique. The pair/tuple idea is likely a great solution in certain cases. I'd like to avoid the return parameters when possible.

Share

Improve this answer

Follow

edited Aug 19, 2021 at 13:46

answered Nov 26, 2008 at 17:05

Fred Larson

62k ● 18 ● 116 ● 177

1 The suggestion to declare a `struct` like `Velocity` is a nice one. However, one concern is that it pollutes the namespace. I suppose that with C++11, the `struct` can have a long type name, and one can use `auto result = calculateResultingVelocity(...)`. – Hugues Feb 12, 2013 at 5:06

9 +1. A function should return *one* "thing", not a somehow-ordered "tuple of things". – DevSolar May 13, 2013 at 7:12

1 I prefer structs over std::pairs/std::tuples for the reasons described in this answer. But I don't like the namespace "pollution" also. The ideal solution for me would be returning anonymous struct like `struct { int a, b; } my_func();`. This could be used as this: `auto result = my_func();`. But C++ doesn't allow this: "new types may not be defined in a return type". So I have to create structs like `struct my_func_result_t` ... – anton_rh Apr 21, 2016 at 6:14

3 @anton_rh : C++14 allows returning local types with `auto`, so `auto result = my_func();` is trivially obtainable. – ildjarn Jan 15, 2017 at 1:11

15 Some 15 years ago when we discovered boost we used tuple a lot since it is quite handy. Overtime we experienced the disadvantage in readability especially for tuples having the same type (e.g. tuple<double, double>; which one is which). So lately we are more in the habit of introducing a small POD structure where at least the name of the member variable indicates something sensible. – gast128 Aug 14, 2017 at 15:19

41

There are a bunch of ways to return multiple parameters. I'm going to be exhastive.

Use reference parameters:

```
void foo( int& result, int& other_result );
```

Use pointer parameters:

```
void foo( int* result, int* other_result );
```

which has the advantage that you have to do a `&` at the call-site, possibly alerting people it is an out-parameter.

Write an `out<?>` template and use it:

```
template<class T>
struct out {
  std::function<void(T)> target;
  out(T* t):target([t](T&& in){ if (t) *t = std::move(in); }) {}
  out(std::optional<T>* t):target([t](T&& in){ if (t) t-
>emplace(std::move(in)); }) {}
  out(std::aligned_storage_t<sizeof(T), alignof(T)>* t):
    target([t](T&& in){ ::new( (void*)t ) T(std::move(in)); } ) {}
  template<class...Args> // TODO: SFINAE enable_if test
  void emplace(Args&&...args) {
    target( T(std::forward<Args>(args)...) );
  }
  template<class X> // TODO: SFINAE enable_if test
  void operator=(X&&x){ emplace(std::forward<X>(x)); }
  template<class...Args> // TODO: SFINAE enable_if test
  void operator()(Args...&&args){ emplace(std::forward<Args>(args)...); }
};
```

then we can do:

```
void foo( out<int> result, out<int> other_result )
```

and all is good. `foo` is no longer able to read any value passed in as a bonus.

Other ways of defining a spot you can put data can be used to construct `out`. A callback to emplace things somewhere, for example.

We can return a structure:

```
struct foo_r { int result; int other_result; };
foo_r foo();
```

whick works ok in every version of C++, and in `c++17` this also permits:

```
auto&&[result, other_result]=foo();
```

at zero cost. Parameters can even not even be moved thanks to guaranteed elision.

We could return a `std::tuple`:

```
std::tuple<int, int> foo();
```

which has the downside that parameters are not named. This permits the `c++17`:

```
auto&&[result, other_result]=foo();
```

as well. Prior to `c++17` we can instead do:

```
int result, other_result;
std::tie(result, other_result) = foo();
```

which is just a bit more awkward. Guaranteed elision doesn't work here, however.

Going into stranger territory (and this is after `out<>`!),

We can use continuation passing style:

```
void foo( std::function<void(int result, int other_result)> );
```

and now callers do:

```
foo( [&](int result, int other_result) {
  /* code */
} );
```

a benefit of this style is you can return an arbitrary number of values (with uniform type) without having to manage memory:

```
void get_all_values( std::function<void(int)> value )
```

the `value` callback could be called 500 times when you `get_all_values( [&](int value){} )`.

For pure insanity, you could even use a continuation on the continuation.

```
void foo( std::function<void(int, std::function<void(int)>)> result );
```

whose use looks like:

```
foo( [&](int result, auto&& other){ other([&](int other){
  /* code */
}) });
```

which would permit many-one relationships between `result` and `other`.

Again with uniforn values, we can do this:

```
void foo( std::function< void(span<int>) > results )
```

here, we call the callback with a span of results. We can even do this repeatedly.

Using this, you can have a function that efficiently passes megabytes of data without doing any allocation off the stack.

```
void foo( std::function< void(span<int>) > results ) {
  int local_buffer[1024];
  std::size_t used = 0;
  auto send_data=[&]{
    if (!used) return;
    results({ local_buffer, used });
    used = 0;
  };
  auto add_datum=[&](int x){
    local_buffer[used] = x;
    ++used;
    if (used == 1024) send_data();
  };
  auto add_data=[&](gsl::span<int const> xs) {
    for (auto x:xs) add_datum(x);
  };
  for (int i = 0; i < 7+(1<<20); ++i) {
    add_datum(i);
  }
  send_data(); // any leftover
}
```

Now, `std::function` is a bit heavy for this, as we would be doing this in zero-overhead no-allocation environments. So we'd want a `function_view` that never allocates.

Another solution is:

```
std::function<void(std::function<void(int result, int other_result)>)> foo(int
input);
```

where instead of taking the callback and invoking it, `foo` instead returns a function which takes the callback.

```
foo(7)([&](int result, int other_result){ /* code */ });
```

this breaks the output parameters from the input parameters by having separate brackets.

Use a Generator:

With `variant` and **c++20** coroutines, you could make `foo` a generator of a variant of the return types (or just the return type). The syntax is not yet fixed, so I won't give examples.

Use signals/slot style:

In the world of signals and slots, a function that exposes a set of signals:

```
template<class...Args>
struct broadcaster;

broadcaster<int, int> foo();
```

allows you to create a `foo` that does work async and broadcasts the result when it is finished.

Use pipelines:

Down this line we have a variety of pipeline techniques, where a function doesn't do something but rather arranges for data to be connected in some way, and the doing is relatively independant.

```
foo( int_source )( int_dest1, int_dest2 );
```

then this code doesn't *do* anything until `int_source` has integers to provide it. When it does, `int_dest1` and `int_dest2` start recieving the results.

Share

Improve this answer

Follow

edited Dec 25, 2022 at 2:55

answered Oct 16, 2018 at 18:49

Yakk - Adam Nevraumont

275k ● 30 ● 350 ● 554

2 This answer contains more information than other answers! in particular, information about `auto&&[result, other_result]=foo();` for functions returning both tuples and structures. Thanks! – jjmontes Apr 1, 2019 at 19:33

2 I appreciate this exhaustive answer, especially since I'm still stuck with C++11 and therefore can't use some of the more modern solutions that other people propose. – Bri Bri Jun 6, 2019 at 18:51

Why do you use `auto&&` instead of `auto` in `auto&&[result, other_result]=foo();` ? – starriet 차주녕 May 29, 2023 at 4:16

@starriet `auto&` means "modifiable reference", `auto const&` means "non-modifiable reference, possibly an extended temporary", `auto` means "make a copy". I use `auto&&` for "I don't care, solve it for me, could be temporary with lifetime extension, could be reference, not my problem, I'm good with anything". – Yakk - Adam Nevraumont May 29, 2023 at 13:49

1 @Yakk-AdamNevraumont Thank you. I thought `T&&` means rvalue reference, so I thought using `auto&` or `auto&&` instead of `auto` (copy) makes a dangling reference. Is it ok to use `auto&` or `auto&&` to capture a return value from a function? Also, I don't understand how "I don't care, solve it for me" can be fine. Please let me know if I misunderstood something, I'm trying to learn these concepts. Thanks :) – starriet 차주녕 May 29, 2023 at 22:58

30

```
std::pair<int, int> divide(int dividend, int divisor)
{
   // :
   return std::make_pair(quotient, remainder);
}

std::pair<int, int> answer = divide(5,2);
 // answer.first == quotient
 // answer.second == remainder
```

std::pair is essentially your struct solution, but already defined for you, and ready to adapt to any two data types.

Share Improve this answer Follow

answered Nov 26, 2008 at 15:22

James Curran
103k ●37 ●185 ●262

3 That'll work for my simple example. In general, however, there may be more than two values returned. – Fred Larson Nov 26, 2008 at 15:27

6 Also not self-documenting. Can you remember which x86 register is the remainder for DIV? – Mark Nov 26, 2008 at 15:35

2 @Mark - I agree that positional solutions can be less maintainable. You can run into the "permute and baffle" problem. – Fred Larson Nov 26, 2008 at 15:42

It's entirely dependent upon the actual function and the meaning of the multiple values, and their sizes:

17

- If they're related as in your fraction example, then I'd go with a struct or class instance.
- If they're not really related and can't be grouped into a class/struct then perhaps you should refactor your method into two.
- Depending upon the in-memory size of the values you're returning, you may want to return a pointer to a class instance or struct, or use reference parameters.

Share Improve this answer Follow

answered Nov 26, 2008 at 15:21
Stewart Johnson
14.4k ● 7 ● 63 ● 70

1 I like your answer and your last bullet reminds me of something I just read that passing by value has gotten much faster depending on circumstances making this more complicated... cpp-next.com/archive/2009/08/want-speed-pass-by-value – sage Dec 26, 2012 at 19:29

15

With C++17 you can also return one ore more unmovable/uncopyable values (in certain cases). The possibility to return unmovable types come via the new guaranteed return value optimization, and it composes nicely with *aggregates*, and what can be called *templated constructors*.

```
template<typename T1,typename T2,typename T3>
struct many {
  T1 a;
  T2 b;
  T3 c;
};

// guide:
template<class T1, class T2, class T3>
many(T1, T2, T3) -> many<T1, T2, T3>;

auto f(){ return many{string(),5.7, unmovable()}; };

int main(){
   // in place construct x,y,z with a string, 5.7 and unmovable.
   auto [x,y,z] = f();
}
```

The pretty thing about this is that it is guaranteed to not cause *any* copying or moving. You can make the example `many` struct variadic too. More details:

[Returning variadic aggregates (struct) and syntax for C++17 variadic template 'construction deduction guide']

Share
Improve this answer

edited May 23, 2017 at 11:47
Community Bot

answered Jul 22, 2016 at 16:55
Johan Lundberg

Follow 1 ● 1 27k ● 13 ● 75 ● 98

14

C++17, using `std::make_tuple`, structured binding and as much `auto` as possible:

```
#include <tuple>

#include <string>
#include <cstring>

auto func() {
    // ...
    return std::make_tuple(1, 2.2, std::string("str"), "cstr");
}

int main() {
    auto [i, f, s, cs] = func();
    return i + f + s.length() + strlen(cs);
}
```

With `-O1` this optimizes out completely: https://godbolt.org/z/133rT9Pcq
`-O3` needed only to optimize out std::string: https://godbolt.org/z/Mqbez73Kf

And here: https://godbolt.org/z/WWKvE3osv you can see GCC storing all the returned values packed together in a single chunk of memory (`rdi+N`), POD-style, proving there is no performance penalty.

Share
Improve this answer
Follow

edited Jan 19, 2022 at 10:59

answered Jan 19, 2022 at 10:38
Piotr Henryk Dabrowski
881 ● 9 ● 14

13

The OO solution for this is to create a ratio class. It wouldn't take any extra code (would save some), would be significantly cleaner/clearer, and would give you some extra refactorings letting you clean up code outside this class as well.

Actually I think someone recommended returning a structure, which is close enough but hides the intent that this needs to be a fully thought-out class with constructor and a few methods, in fact, the "method" that you originally mentioned (as returning the pair) should most likely be a member of this class returning an instance of itself.

I know your example was just an "Example", but the fact is that unless your function is doing way more than any function should be doing, if you want it to return multiple values you are almost certainly missing an object.

Don't be afraid to create these tiny classes to do little pieces of work--that's the magic of OO--you end up breaking it down until every method is very small and simple and every class small and understandable.

Another thing that should have been an indicator that something was wrong: in OO you have essentially no data--OO isn't about passing around data, a class needs to manage and manipulate it's own data internally, any data passing (including accessors) is a sign that you may need to rethink something..

Share Improve this answer Follow

answered Nov 26, 2008 at 17:45

Bill K
62.8k ● 18 ● 112 ● 158

Can you expand more on *'hides the intent'* please? It would be great if you could give an example of how you would implement OP's code using OO – Sabito Aug 19, 2021 at 5:26

1 @Sabito錆兎 Fred Larson's example is a good one. OO isn't always the best solution for truly general utility methods like "Divide", but when you are solving an actual business problem like he did, the advantage becomes obvious. – Bill K Aug 26, 2021 at 17:44

10

There is precedent for returning structures in the C (and hence C++) standard with the `div`, `ldiv` (and, in C99, `lldiv`) functions from `<stdlib.h>` (or `<cstdlib>`).

The 'mix of return value and return parameters' is usually the least clean.

Having a function return a status and return data via return parameters is sensible in C; it is less obviously sensible in C++ where you could use exceptions to relay failure information instead.

If there are more than two return values, then a structure-like mechanism is probably best.

Share Improve this answer Follow

answered Nov 26, 2008 at 17:39

Jonathan Leffler
752k ● 145 ● 946 ● 1.3k

6

Here is the link to the "core guidelines" (by Bjarne Stroustrup and Herb Sutter) on this topic.

https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rf-out-multi

Partial Quote:

> F.21: To return multiple “out” values, prefer returning a struct or tuple
>
> **Reason** A return value is self-documenting as an “output-only” value. Note that C++ does have multiple return values, by convention of using a tuple (including pair), possibly with the extra convenience of tie or structured

> bindings (C++17) at the call site. Prefer using a named struct where there are semantics to the returned value. Otherwise, a nameless tuple is useful in generic code.

Share Improve this answer Follow

answered Apr 29, 2022 at 15:19

Xiangming Hu

355 ● 4 ● 7

this should be the accepted answer. Lets hope it finds its way to the top, fast
– Markus Dutschke Aug 16, 2022 at 11:44

5

Here, i am writing a program that is returning multiple values(more than two values) in c++. This program is executable in c++14 (G++4.9.2). program is like a calculator.

```
#  include <tuple>
# include <iostream>

using namespace std;

tuple < int,int,int,int,int >   cal(int n1, int n2)
{
    return  make_tuple(n1/n2,n1%n2,n1+n2,n1-n2,n1*n2);
}

int main()
{
    int qut,rer,add,sub,mul,a,b;
    cin>>a>>b;
    tie(qut,rer,add,sub,mul)=cal(a,b);
    cout << "quotient= "<<qut<<endl;
    cout << "remainder= "<<rer<<endl;
    cout << "addition= "<<add<<endl;
    cout << "subtraction= "<<sub<<endl;
    cout << "multiplication= "<<mul<<endl;
    return 0;
}
```

So, you can clearly understand that in this way you can return multiple values from a function. using std::pair only 2 values can be returned while std::tuple can return more than two values.

Share

Improve this answer

Follow

edited Jan 23, 2017 at 8:34
M.M
141k ● 25 ● 220 ● 387

answered Feb 4, 2015 at 10:41
PRAFUL ANAND
405 ● 4 ● 7

4 With C++14 you can also use `auto` return type on `cal` to make this even cleaner. (IMO).
– sfjac Apr 19, 2015 at 14:53

5

Use a struct or a class for the return value. Using `std::pair` may work for now, but

1. it's inflexible if you decide later you want more info returned;
2. it's not very clear from the function's declaration in the header what is being returned and in what order.

Returning a structure with self-documenting member variable names will likely be less bug-prone for anyone using your function. Putting my coworker hat on for a moment, your `divide_result` structure is easy for me, a potential user of your function, to immediately understand after 2 seconds. Messing around with ouput parameters or mysterious pairs and tuples would take more time to read through and may be used incorrectly. And most likely even after using the function a few times I still won't remember the correct order of the arguments.

Share
Improve this answer
Follow

edited Jul 24, 2017 at 17:08
dif
2,493 ● 6 ● 38 ● 51

answered Nov 26, 2008 at 17:43
Michel
1,494 ● 11 ● 16

4

If your function returns a value via reference, the compiler cannot store it in a register when calling other functions because, theoretically, the first function can save the address of the variable passed to it in a globally accessible variable, and any subsecuently called functions may change it, so the compiler will have (1) save the value from registers back to memory before calling other functions and (2) re-read it when it is needed from the memory again after any of such calls.

If you return by reference, optimization of your program will suffer

Share Improve this answer Follow

answered Nov 26, 2008 at 16:00
dmityugov
4,480 ● 24 ● 18

3

I tend to use out-vals in functions like this, because I stick to the paradigm of a function returning success/error codes and I like to keep things uniform.

Share Improve this answer Follow

answered Nov 26, 2008 at 15:22
John Dibling
101k ● 32 ● 191 ● 331

2

Alternatives include arrays, generators, and inversion of control, but none is appropriate here.

Some (e.g. Microsoft in historical Win32) tend to use reference parameters for simplicity, because it's clear who allocates and how it will look on the stack, reduces the proliferation of structures, and allows a separate return value for success.

"Pure" programmers prefer the struct, assuming it *is* the function value (as is the case here), rather than something that's touched incidentally by the function. If you had a more complicated procedure, or something with state, you'd probably use references (assuming you have a reason for not using a class).

Share

Improve this answer

Follow

edited Nov 26, 2008 at 15:39

answered Nov 26, 2008 at 15:20

Mark

6,295 ●2 ●37 ●34

2

I'd say there is no preferred method, it all depends on what you're going to do with the response. If the results are going to be used together in further processing then structures make sense, if not I'd tend to pass then as individual references unless the function was going to be used in a composite statement:

`x = divide( x, y, z ) + divide( a, b, c );`

I often choose to pass 'out structures' by reference in the parameter list rather than having the pass by copy overhead of returning a new structure (but this is sweating the small stuff).

`void divide(int dividend, int divisor, Answer &ans)`

Are out parameters confusing? A parameter sent as reference suggests the value is going to change (as opposed to a const reference). Sensible naming also removes confusion.

Share Improve this answer Follow

answered Nov 26, 2008 at 15:40

Patrick

8,290 ●7 ●58 ●72

1 I think it's a bit confusing. Someone reading code that calls it sees "divide(a,b,c);". There's no indication that c is an outval until they look up the signature. But that's a general fear of non-const reference params, rather than particular to this question. – Steve Jessop Nov 26, 2008 at 21:51

2

Why do you insist on a function with multiple return values? With OOP you can use a class offering a regular function with a single return value, and any number of additional "return values" like below. The advantage is that the caller has a choice of looking at the extra data members, but is not required to do this. This is the preferred method for complicated data base or networking calls, where lots of additional return info may be needed in case errors occur.

To answer your original question, this example has a method to return the quotient, which is what most callers may need, and additionally, after the method call, you can get the remainder as a data member.

```
class div{
   public:
      int remainder;
      int quotient(int dividend, int divisor){
         remainder = ...;
         return ...;
      }
};
```

Share

Improve this answer

Follow

edited Nov 21, 2014 at 10:28

answered Nov 21, 2014 at 10:20

Roland
5,146 ● 7 ● 57 ● 86

1 I think there are cases where this is inefficient. E.g. you have a single for loop that generates several return values. If you split those values into separate functions you'd need to run through the loop once for each value. – jiggunjer Mar 8, 2015 at 17:42

1 @jiggunjer You could run the loop once and store the several return values in separate class data members. This underscores the flexibility of the OOP concept. – Roland Mar 9, 2015 at 12:50

2

Boost tuple would be my preferred choice for a generalized system of returning more than one value from a function.

Possible example:

```
include "boost/tuple/tuple.hpp"

tuple <int,int> divide( int dividend,int divisor )

{
  return make_tuple(dividend / divisor,dividend % divisor )
}
```

Share

Improve this answer

Follow

edited Feb 4, 2015 at 11:26

PRAFUL ANAND
405 ● 4 ● 7

answered Nov 26, 2008 at 15:43

AndyUK
3,993 ● 7 ● 45 ● 47

2

rather than returning multiple values,just return one of them and make a reference of others in the required function for eg:

```
int divide(int a,int b,int quo,int &rem)
```

Share

Improve this answer

Follow

edited Nov 16, 2017 at 18:58

Machavity ♦
31.6k ● 27 ● 95 ● 105

answered Nov 16, 2017 at 18:52

user8953567

Did I not mention this in the question itself? Also, see my objections in my answer.
– Fred Larson Nov 16, 2017 at 20:21

1

We can declare the function such that, it returns a structure type user defined variable or a pointer to it . And by the property of a structure, we know that a structure in C can hold multiple values of asymmetrical types (i.e. one int variable, four char variables, two float variables and so on…)

Share Improve this answer Follow

answered Oct 20, 2015 at 11:02
Rohit Hajare
135 ●1 ●7

0

I would just do it by reference if it's only a few return values but for more complex types you can also just do it like this :

```
static struct SomeReturnType {int a,b,c; string str;} SomeFunction()
{
  return {1,2,3,string("hello world")}; // make sure you return values in the right order!
}
```

use "static" to limit the scope of the return type to this compilation unit if it's only meant to be a temporary return type.

```
 SomeReturnType st = SomeFunction();
 cout << "a "   << st.a << endl;
 cout << "b "   << st.b << endl;
 cout << "c "   << st.c << endl;
 cout << "str " << st.str << endl;
```

This is definitely not the prettiest way to do it but it will work.

Share
Improve this answer
Follow

edited May 21, 2019 at 20:32

answered May 21, 2019 at 18:11
Carsten
9 ●2

`struct SomeReturnType {int a,b,c; string str;} SomeFunction()` error: new types may not be defined in a return type – MatG Dec 12, 2022 at 14:42

Quick answer:

-5

```
#include <iostream>
using namespace std;

// different values of [operate] can return different number.
int yourFunction(int a, int b, int operate)
{
    a = 1;
    b = 2;

    if (operate== 1)
    {
        return a;
    }
    else
    {
        return b;
    }
}

int main()
{
    int a, b;

    a = yourFunction(a, b, 1); // get return 1
    b = yourFunction(a, b, 2); // get return 2

    return 0;
}
```

Share Improve this answer Follow

answered Oct 3, 2020 at 18:17
myworldbox
1

I recommend looking at some of the other solutions such as `auto&&[result, other_result]=foo();`. This has the benefit of not doing extra work if `foo` has to do some heavy lifting before calculating `a` or `b`, and it's a standard solution instead of passing `operate` in, which might confuse other programmers. – user904963 Dec 22, 2021 at 19:49