12.15 — std::optional

learncpp.com/cpp-tutorial/stdoptional/

In lesson 9.4 -- Detecting and handling errors, we discussed cases where a function encounters an error that it cannot reasonably handle itself. For example, consider a function that calculates and returns a value:

```
int doIntDivision(int x, int y)
{
    return x / y;
}
```

If the caller passes in a value that is semantically invalid (such as `y` = `0`), this function cannot calculate a value to return (as division by 0 is mathematically undefined). What do we do in that case? Because functions that calculate results should have no side effects, this function cannot reasonably resolve the error itself. In such cases, the typical thing to do is have the function detect the error, but then pass the error back to the caller to deal with in some program-appropriate way.

In the previously linked lesson, we covered two different ways to have a function return an error back to the caller:

- Have a void-returning function return a bool instead (indicating success or failure).
- Have a value-returning function return a sentinel value (a special value that does not occur in the set of possible values the function can otherwise return) to indicate an error.

As an example of the latter, the `reciprocal()` function that follows returns value `0.0` (which can never otherwise occur) if the user passes in a semantically invalid argument for `x`:

```
// The reciprocal of x is 1/x, returns 0.0 if x=0
double reciprocal(double x)
{
    if (x == 0.0) // if x is semantically invalid
       return 0.0; // return 0.0 as a sentinel to indicate an error occurred

    return 1.0 / x;
}
```

While this is a fairly attractive solution, there are a number of potential downsides:

- The programmer must know which sentinel value the function is using to indicate an error (and this value may differ for each function returning an error using this method).
- A different version of the same function may use a different sentinel value.

- This method does not work for functions where all possible sentinel values are valid return values.

Consider our `doIntDivision()` function above. What value could it return if the user passes in `0` for `y`? We can't use `0`, because `0` divided by anything yields `0` as a valid result. In fact, there are no values that we could return that cannot occur naturally.

So what are we to do?

First, we could pick some (hopefully) uncommon return value as our sentinel and use it to indicate an error:

```
#include <limits> // for std::numeric_limits

// returns std::numeric_limits<int>::lowest() on failure
int doIntDivision(int x, int y)
{
    if (y == 0)
        return std::numeric_limits<int>::lowest();
    return x / y;
}
```

In the above function, we use `std::numeric_limits<int>::lowest()` (which resolves to the largest negative integral value) to indicate that the function failed. While this mostly works, it has two downsides:

- Every time we call this function, we need to test the return value for equality with `std::numeric_limits<int>::lowest()` to see if it failed. That's verbose and ugly.
- It is an example of a semipredicate problem: if the user calls `doIntDivision(std::numeric_limits<int>::lowest(), 1)`, the returned result `std::numeric_limits<int>::lowest()` will be ambiguous as to whether the function succeeded or failed. That may or may not be a problem depending on how the function is actually used, but it's another thing we have to worry about and another potential way that errors can creep into our program.

Second, we could abandon using return values to return errors and use some other mechanism (e.g. exceptions). However, exceptions have their own complications and performance costs, and may not be appropriate or desired. That's probably overkill for something like this.

Third, we could abandon returning a single value and return two values instead: one (of type `bool`) that indicates whether the function succeeded, and the other (of the desired return type) that holds the actual return value (if the function succeeded) or an indeterminate value (if the function failed). This is probably the best option of the bunch.

Prior to C++17, choosing this latter option required you to implement it yourself. And while C++ provides multiple ways to do so, any roll-your-own approach will inevitably lead to inconsistencies and errors.

Returning a `std::optional`

C++17 introduces `std::optional`, which is a class template type that implements an optional value. That is, a `std::optional<T>` can either have a value of type `T`, or not. We can use this to implement the third option above:

```
#include <iostream>
#include <optional> // for std::optional (C++17)

// Our function now optionally returns an int value
std::optional<int> doIntDivision(int x, int y)
{
    if (y == 0)
        return {}; // or return std::nullopt
    return x / y;
}

int main()
{
    std::optional<int> result1 { doIntDivision(20, 5) };
    if (result1) // if the function returned a value
        std::cout << "Result 1: " << *result1 << '\n'; // get the value
    else
        std::cout << "Result 1: failed\n";

    std::optional<int> result2 { doIntDivision(5, 0) };

    if (result2)
        std::cout << "Result 2: " << *result2 << '\n';
    else
        std::cout << "Result 2: failed\n";

    return 0;
}
```

This prints:

```
Result 1: 4
Result 2: failed
```

Using `std::optional` is quite easy. We can construct either construct a `std::optional<T>` with a value, or not:

```
std::optional<int> o1 { 5 };            // initialize with a value
std::optional<int> o2 {};               // initialize with no value
std::optional<int> o3 { std::nullopt }; // initialize with no value
```

To see if a `std::optional` has a value, we can choose one of the following:

```
if (o1.has_value()) // call has_value() to check if o1 has a value
if (o2)             // use implicit conversion to bool to check if o2 has a value
```

To get the value from a `std::optional`, we can choose one of the following:

```
std::cout << *o1;             // dereference to get value stored in o1 (undefined
behavior if o1 does not have a value)
std::cout << o2.value();      // call value() to get value stored in o2 (throws
std::bad_optional_access exception if o2 does not have a value)
std::cout << o3.value_or(42); // call value_or() to get value stored in o3 (or value
`42` if o3 doesn't have a value)
```

Note that `std::optional` has a usage syntax that is essentially identical to a pointer:

Behavior	Pointer	`std::optional`
Hold no value	initialize/assign `{}` or `std::nullptr`	initialize/assign `{}` or `std::nullopt`
Hold a value	initialize/assign an address	initialize/assign a value
Check if has value	implicit conversion to bool	implicit conversion to bool or `has_value()`
Get value	dereference	dereference or `value()`

However, semantically, a pointer and a `std::optional` are quite different.

- A pointer has reference semantics, meaning it references some other object, and assignment copies the pointer, not the object. If we return a pointer by address, the pointer is copied back to the caller, not the object being pointed to. This means we can't return a local object by address, as we'll copy that object's address back to the caller, and then the object will be destroyed, leaving the returned pointer dangling.
- A `std::optional` has value semantics, meaning it actually contains its value, and assignment copies the value. If we return a `std::optional` by value, the `std::optional` (including the contained value) is copied back to the caller. This means we can return a value from the function back to the caller using `std::optional`.

With this in mind, let's look at how our example works. Our `doIntDivision()` now returns a `std::optional<int>` instead of an `int`. Inside the function body, if we detect an error, we return `{}`, which implicitly returns a `std::optional` containing no value. If we have a value, we return that value, which implicit returns a `std::optional` containing that value.

Within `main()`, we use an implicit conversion to bool to check if our returned `std::optional` has a value or not. If it does, we dereference the `std::optional` object to get the value. If it doesn't, then we execute our error condition. That's it!

Pros and cons of returning a `std::optional`

Returning a `std::optional` is nice for a number of reasons:

- Using `std::optional` effectively documents that a function may return a value or not.
- We don't have to remember which value is being returned as a sentinel.
- The syntax for using `std::optional` is convenient and intuitive.

Returning a `std::optional` does come with a few downsides:

- We have to make sure the `std::optional` contains a value before getting the value. If we dereference a `std::optional` that does not contain a value, we get undefined behavior.
- `std::optional` does not provide a way to pass back information about why the function failed.

Unless your function needs to return additional information about why it failed (either to better understand the failure, or to differentiate different kinds of failure), `std::optional` is an excellent choice for functions that may return a value or fail.

Best practice

Return a `std::optional` (instead of a sentinel value) for functions that may fail, unless your function needs to return additional information about why it failed.

Related content

`std::expected` (introduced in C++23) is designed to handle the case where a function can return either an expected value or an unexpected error code. See the std::expected reference for more information.

Using `std::optional` as an optional function parameter

In lesson 12.11 -- Pass by address (part 2), we discussed how pass by address can be used to allow a function to accept an "optional" argument (that is, the caller can either pass in `nullptr` to represent "no argument" or an object). However, one downside of this approach is that a non-nullptr argument must be an lvalue (so that its address can be passed to the function).

Perhaps unsurprisingly (given the name), `std::optional` is an alternative way for a function to accept an optional argument (that is used as an in-parameter only). Instead of this:

```cpp
#include <iostream>

void printIDNumber(const int *id=nullptr)
{
    if (id)
        std::cout << "Your ID number is " << *id << ".\n";
    else
        std::cout << "Your ID number is not known.\n";
}

int main()
{
    printIDNumber(); // we don't know the user's ID yet

    int userid { 34 };
    printIDNumber(&userid); // we know the user's ID now

    return 0;
}
```

You can do this:

```cpp
#include <iostream>
#include <optional>

void printIDNumber(std::optional<const int> id = std::nullopt)
{
    if (id)
        std::cout << "Your ID number is " << *id << ".\n";
    else
        std::cout << "Your ID number is not known.\n";
}

int main()
{
    printIDNumber(); // we don't know the user's ID yet

    int userid { 34 };
    printIDNumber(userid); // we know the user's ID now

    printIDNumber(62); // we can also pass an rvalue

    return 0;
}
```

There are two advantages to this approach:

1. It effectively documents that the parameter is optional.
2. We can pass in an rvalue (since `std::optional` will make a copy).

However, because `std::optional` makes a copy of its argument, this becomes problematic when `T` is an expensive-to-copy type (like `std::string`). With normal function parameters, we worked around this by making the parameter a `const lvalue reference`, so that a copy would not be made. Unfortunately, as of C++23 `std::optional` does not support references.

Therefore, we recommend using `std::optional<T>` as an optional parameter only when `T` would normally be passed by value. Otherwise, use `const T*`.

For advanced readers

Although `std::optional` doesn't support references directly, you can use `std::reference_wrapper` (which we cover in lesson 17.5 -- Arrays of references via std::reference_wrapper) to mimic a reference. Let's take a look a look at what the above program looks like using a `std::string` id and `std::reference_wrapper`:

```
#include <functional>  // for std::reference_wrapper
#include <iostream>
#include <optional>
#include <string>

struct Employee
{
    std::string name{}; // expensive to copy
    int id;
};

void printEmployeeID(std::optional<std::reference_wrapper<Employee>> e=std::nullopt)
{
    if (e)
        std::cout << "Your ID number is " << e->get().id << ".\n";
    else
        std::cout << "Your ID number is not known.\n";
}

int main()
{
    printEmployeeID(); // we don't know the Employee yet

    Employee e { "James", 34 };
    printEmployeeID(e); // we know the Employee's ID now

    return 0;
}
```

And for comparison, the pointer version:

```
#include <iostream>
#include <string>

struct Employee
{
    std::string name{}; // expensive to copy
    int id;
};

void printEmployeeID(const Employee* e=nullptr)
{
    if (e)
        std::cout << "Your ID number is " << e->id << ".\n";
    else
        std::cout << "Your ID number is not known.\n";
}

int main()
{
    printEmployeeID(); // we don't know the Employee yet

    Employee e { "James", 34 };
    printEmployeeID(&e); // we know the Employee's ID now

    return 0;
}
```

These two programs are nearly identical. We'd argue the former isn't more readable or maintainable than the latter, and isn't worth introducing two additional types into your program for.

In many cases, function overloading provides a superior solution:

```cpp
#include <iostream>
#include <string>

struct Employee
{
    std::string name{}; // expensive to copy
    int id;
};

void printEmployeeID()
{
    std::cout << "Your ID number is not known.\n";
}

void printEmployeeID(const Employee& e)
{
    std::cout << "Your ID number is " << e.id << ".\n";
}

int main()
{
    printEmployeeID(); // we don't know the Employee yet

    Employee e { "James", 34 };
    printEmployeeID(e); // we know the Employee's ID now

    printEmployeeID( { "Dave", 62 } ); // we can even pass rvalues

    return 0;
}
```

Best practice

Prefer `std::optional` for optional return types.

Prefer function overloading for optional function parameters (when possible). Otherwise, use `std::optional<T>` for optional arguments when `T` would normally be passed by value. Favor `const T*` when `T` is expensive to copy.