libpqxx

@version 7.7.5 @author Jeroen T. Vermeulen @see http://pqxx.org @see https://github.com/jtv/libpqxx

Welcome to libpqxx, the C++ API to the PostgreSQL database management system.

Compiling this package requires PostgreSQL to be installed – including the C headers for client development. The library builds on top of PostgreSQL's standard C API, libpq. The libpq headers are not needed to compile client programs, however.

For a quick introduction to installing and using libpqxx, see the README.md file. The latest information can be found at http://pqxx.org/.

Some links that should help you find your bearings:

- @ref getting-started
- @ref thread-safety
- @ref connections
- @ref transactions
- @ref escaping
- @ref performance
- @ref transactor
- @ref datatypes Accessing results and result rows {#accessing-results} ================================

A query produces a result set consisting of rows, and each row consists of fields. There are several ways to receive this data.

The fields are "untyped." That is to say, libpqxx has no opinion on what their types are. The database sends the data in a very flexible textual format. When you read a field, you specify what type you want it to be, and libpqxx converts the text format to that type for you.

If a value does not conform to the format for the type you specify, the conversion fails. For example, if you have strings that all happen to contain numbers, you can read them as `int`. But if any of the values is empty, or it's null (for a type that doesn't support null), or it's some string that does not look like an integer, or it's too large, you can't convert it to `int`.

So usually, reading result data from the database means not just retrieving the data; it also means converting it to some target type.

Querying rows of data

The simplest way to query rows of data is to call one of a transaction's "query" functions, passing as template arguments the types of columns you want to get back (e.g. `int`, `std::string`, `double`, and so on) and as a regular argument the query itself.

You can then iterate over the result to go over the rows of data:

```
for (auto [id, value] :
    tx.query<int, std::string>("SELECT id, name FROM item"))
{
    std::cout << id << '\t' << value << '\n';
}
```

The "query" functions execute your query, load the complete result data from the database, and then as you iterate, convert each row it received to a tuple of C++ types that you indicated.

There are different query functions for querying any number of rows (`query()`); querying just one row of data as a `std::tuple` and throwing an error if there's more than one row (`query1()`); or querying

Streaming rows

There's another way to go through the rows coming out of a query. It's usually easier and faster if there are a lot of rows, but there are drawbacks.

One, you start getting rows before all the data has come in from the database. That speeds things up, but what happens if you lose your network connection while transferring the data? Your application may already have processed some of the data before finding out that the rest isn't coming. If that is a problem for your application, streaming may not be the right choice.

Two, streaming only works for some types of query. The `stream()` function wraps your query in a PostgreSQL `COPY` command, and `COPY` only supports a few commands: `SELECT`, `VALUES`, or an `INSERT`, `UPDATE`, or `DELETE` with a `RETURNING` clause. See the `COPY` documentation here: https://www.postgresql.org/docs/current/sql-copy.html.

Three, when you convert a field to a "view" type (such as `std::string_view` or `std::basic_string_view<std::byte>`), the view points to underlying data which only stays valid until you iterate to the next row or exit the loop. So if you want to use that data for longer than a single iteration of the streaming loop, you'll have to store it somewhere yourself.

Now for the good news. Streaming does make it very easy to query data and loop over it:

```
for (auto [id, name, x, y] :
    tx.stream<int, std::string_view, float, float>(
        "SELECT id, name, x, y FROM point"))
  process(id + 1, "point-" + name, x * 10.0, y * 10.0);
```

The conversion to C++ types (here `int`, `std::string_view`, and two `float`s) is built into the function. You never even see `row` objects, `field` objects, iterators, or conversion methods. You just put in your query and you receive your data.

Results with metadata

Sometimes you want more from a query result than just rows of data. You may need to know right away how many rows of result data you received, or how many rows your `UPDATE` statement has affected, or the names of the columns, etc.

For that, use the transaction's "exec" query execution functions. Apart from a few exceptions, these return a `pqxx::result` object. A `result` is a container of `pqxx::row` objects, so you can iterate them as normal, or index them like you would index an array. Each `row` in turn is a container of `pqxx::field`, Each `field` holds a value, but doesn't know its type. You specify the type when you read the value.

For example, your code might do:

```
pqxx::result r = tx.exec("SELECT * FROM mytable");
for (auto const &row: r)
{
   for (auto const &field: row) std::cout << field.c_str() << '\t';
   std::cout << '\n';
}
```

But results and rows also support other kinds of access. Array-style indexing, for instance, such as `r[rownum]`:

```
std::size_t const num_rows = std::size(r);
for (std::size_t rownum=0u; rownum < num_rows; ++rownum)
{
  pqxx::row const row = r[rownum];
  std::size_t const num_cols = std::size(row);
  for (std::size_t colnum=0u; colnum < num_cols; ++colnum)
  {
    pqxx::field const field = row[colnum];
    std::cout << field.c_str() << '\t';
  }

  std::cout << '\n';
}
```

Every row in the result has the same number of columns, so you don't need to look up the number of fields again for each one:

```
std::size_t const num_rows = std::size(r);
std::size_t const num_cols = r.columns();
```

```
for (std::size_t rownum=0u; rownum < num_rows; ++rownum)
{
  pqxx::row const row = r[rownum];
  for (std::size_t colnum=0u; colnum < num_cols; ++colnum)
  {
    pqxx::field const field = row[colnum];
    std::cout << field.c_str() << '\t';
  }

  std::cout << '\n';
}
```

You can even address a field by indexing the `row` using the field's *name:*

```
std::cout << row["salary"] << '\n';
```

But try not to do that if speed matters, because looking up the column by name takes time. At least you'd want to look up the column index before your loop and then use numerical indexes inside the loop.

For C++23 or better, there's also a two-dimensional array access operator:

```
for (std::size_t rownum=0u; rownum < num_rows; ++rownum)
{
    for (std::size_t colnum=0u; colnum < num_cols; ++colnum)
        std::cout result[rownum, colnum].c_str() << '\t';
    std::cout << '\n';
}
```

And of course you can use classic "begin/end" loops:

```
for (auto row = std::begin(r); row != std::end(r); row++)
{
  for (auto field = std::begin(row); field != std::end(row); field++)
    std::cout << field->c_str() << '\t';
  std::cout << '\n';
}
```

Result sets are immutable, so all iterators on results and rows are actually `const_iterator`s. There are also `const_reverse_iterator` types, which iterate backwards from `rbegin()` to `rend()` exclusive.

All these iterator types provide one extra bit of convenience that you won't normally find in C++ iterators: referential transparency. You don't need to dereference them to get to the row or field they refer to. That is, instead of `row->end()` you can also choose to say `row.end()`. Similarly, you may prefer `field.c_str()` over `field->c_str()`.

This becomes really helpful with the array-indexing operator. With regular C++ iterators you would need ugly expressions like `(*row)[0]` or `row->operator[](0)`. With the iterator types defined by the result and row classes you can simply say `row[0]`. Binary data {#binary} ===========

The database has two ways of storing binary data: `BYTEA` is like a string, but containing bytes rather than text characters. And *large objects* are more like a separate table containing binary objects.

Generally you'll want to use `BYTEA` for reasonably-sized values, and large objects for very large values.

That's the database side. On the C++ side, in libpqxx, all binary data must be either `std::basic_string<std::byte>` or `std::basic_string_view<std::byte>`; or if you're building in C++20 or better, anything that's a block of contiguous `std::byte` in memory.

So for example, if you want to write a large object, you'd create a `pqxx::blob` object. And you might use that to write data in the form of `std::basic_string_view<std::byte>`.

Your particular binary data may look different though. You may have it in a `std::string`, or a `std::vector<unsigned char>`, or a pointer to `char` accompanied by a size (which could be signed or unsigned, and of any of a few different widths). Sometimes that's your choice, or sometimes some other library will dictate what form it takes.

So long as it's *basically* still a block of bytes though, you can use `pqxx::binary_cast` to construct a `std::basic_string_view<std::byte>` from it.

There are two forms of `binary_cast`. One takes a single argument that must support `std::data()` and `std::size()`:

```
std::string hi{"Hello binary world"};
my_blob.write(pqxx::binary_cast(hi);
```

The other takes a pointer and a size:

```
char const greeting[] = "Hello binary world";
char const *hi = greeting;
my_blob.write(pqxx::binary_cast(hi, sizeof(greeting)));
```

Caveats

There are some restrictions on `binary_cast` that you must be aware of.

First, your data must of a type that gives us *bytes*. So: `char`, `unsigned char`, `signed char`, `int8_t`, `uint8_t`, or of course `std::byte`. You can't feed in a vector of `double`, or anything like that.

Second, the data must be laid out as a contiguous block in memory. If there's no `std::data()` implementation for your type, it's not suitable.

Third, `binary_cast` only constructs something like a `std::string_view`. It does not make a copy of your actual data. So, make sure that your data remains alive and in the same place while you're using it. Supporting additional data types {#datatypes} ==================================

Communication with the database mostly happens in a text format. When you include an integer value in a query, you use `to_string` to convert it to that text format. When you get a query result field "as a float," it converts from the text format to a floating-point type. These conversions are everywhere in libpqxx.

The conversion sydstem supports many built-in types, but it is also extensible. You can "teach" libpqxx (in the scope of your own application) to convert additional types of values to and from PostgreSQL's string format.

This is massively useful, but it's not for the faint of heart. You'll need to specialise some templates. And, **the API for doing this can change with any major libpqxx release.**

Converting types

In your application, a conversion is driven entirely by a C++ type you specify. The value's SQL type has nothing to do with it, nor is there anything in the string that would identify its type.

So, if you've SELECTed a 64-bit integer from the database, and you try to convert it to a C++ "short," one of two things will happen: either the number is small enough to fit in your `short` (and it just works), or else it throws a conversion exception.

Or, your database table might have a text column, but a given field may contain a string that *looks* just like a number. You can convert that value to an integer type just fine. Or to a floating-point type. All that matters to the conversion is the actual value, and the type.

In some cases the templates for these conversions can tell the type from the arguments you pass them:

```
auto x = to_string(99);
```

In other cases you may need to instantiate template explicitly:

```
auto y = from_string<int>("99");
```

Supporting a new type

Let's say you have some other SQL type which you want to be able to store in, or retrieve from, the database. What would it take to support that?

Sometimes you do not need *complete* support. You might need a conversion *to* a string but not *from* a string, for example. The conversion is defined at compile time, so don't be too afraid to be incomplete. If you leave out one of these steps, it's not going to crash at run time or mess up your data. The worst that can happen is that your code won't build.

So what do you need for a complete conversion?

First off, of course, you need a C++ type. It may be your own, but it doesn't have to be. It could be a type from a third-party library, or even one from the standard library that libpqxx does not yet support.

You also specialise the `pqxx::type_name` variable to specify the type's name. This is important for all code which mentions your type in human-readable text, such as error messages.

Then, does your type have a built-in null value? You specialise the `pqxx::nullness` template to specify the details.

Finally, you specialise the `pqxx::string_traits` template. This is where you define the actual conversions.

Let's go through these steps one by one.

Your type

You'll need a type for which the conversions are not yet defined, because the C++ type is what determines the right conversion. One type, one set of conversions.

The type doesn't have to be one that you create. The conversion logic was designed such that you can build it around any type. So you can just as easily build a conversion for a type that's defined somewhere else. There's no need to include any special methods or other members inside it. That's also how libpqxx can support converting built-in types like `int`.

By the way, if the type is an enum, you don't need to do any of this. Just invoke the preprocessor macro `PQXX_DECLARE_ENUM_CONVERSION`, from the global namespace near the top of your translation unit, and pass the type as an argument.

The library also provides specialisations for `std::optional<T>`, `std::shared_ptr<T>`, and `std::unique_ptr<T>`. If you have conversions for `T`, you'll also have conversions for those.

Specialise `type_name`

When errors happen during conversion, libpqxx will compose error messages for the user. Sometimes these will include the name of the type that's being converted.

To tell libpqxx the name of each type, there's a template variable called `pqxx::type_name`. For any given type `T`, it should have a specialisation that provides that `T`'s human-readable name:

```
namespace pqxx
{
template<> std::string const type_name<T>{"T"};
}
```

(Yes, this means that you need to define something inside the pqxx namespace. Future versions of libpqxx may move this into a separate namespace.)

Define this early on in your translation unit, before any code that might cause libpqxx to need the name. That way, the libpqxx code which needs to know the type's name can see your definition.

Specialise `nullness`

A struct template `pqxx::nullness` defines whether your type has a natural "null value" built in. If so, it also provides member functions for producing and recognising null values.

The simplest scenario is also the most common: most types don't have a null value built in. In that case, derive your nullness traits from `pqxx::no_null`:

```
namespace pqxx
{
template<> struct nullness<T> : pqxx::no_null<T> {};
}
```

(Here again you're defining this in the pqxx namespace.)

If your type does have a natural null value, the definition gets a little more complex:

```
namespace pqxx
{
template<> struct nullness<T>
{
  static constexpr bool has_null{true};
  static constexpr bool always_null{false};

  static bool is_null(T const &value)
  {
    // Return whether "value" is null.
    return ...;
  }

  [[nodiscard]] static T null()
  {
    // Return a null value.
    return ...;
  }
```

```
};
}
```

You may be wondering why there's a function to produce a null value, but also a function to check whether a value is null. Why not just compare the value to the result of `null()`? Because two null values may not be equal. `T` may have several different null values. Or it may override the comparison operator, similar to SQL where NULL is not equal to NULL.

As a third case, your type may be one that *always* represents a null value. This is the case for `std::nullptr_t` and `std::nullopt_t`. In that case, you set `nullness<TYPE>::always_null` to `true` (as well as `has_null` of course), and you won't need to define any actual conversions.

Specialise `string_traits`

This part is more work. (You can skip it for types that are *always* null, but those will be rare.) Specialise the `pqxx::string_traits` template:

```
namespace pqxx
{
template<> struct string_traits<T>
{
  static T from_string(std::string_view text);
  static zview to_buf(char *begin, char *end, T const &value);
  static char *into_buf(char *begin, char *end, T const &value);
  static std::size_t size_buffer(T const &value) noexcept;
};
}
```

You'll also need to write those member functions, or as many of them as needed to get your code to build.

`from_string`

We start off simple: `from_string` parses a string as a value of `T`, and returns that value.

The string may not be zero-terminated; it's just the `string_view` from beginning to end (exclusive). In your tests, cover cases where the string does not end in a zero byte.

It's perfectly possible that the string isn't actually a `T` value. Mistakes happen. In that case, throw a `pqxx::conversion_error`.

(Of course it's also possible that you run into some other error, so it's fine to throw different exceptions as well. But when it's definitely "this is not the right format for a `T`," throw `conversion_error`.)

`to_buf`

In this function, you convert a value of `T` into a string that the postgres server will understand.

The caller will provide you with a buffer where you can write the string, if you need it: from `begin` to `end` exclusive. It's a half-open interval, so don't access `*end`.

If the buffer is insufficient for you to do the conversion, throw a `pqxx::conversion_overrun`. It doesn't have to be exact: you can be a little pessimistic and demand a bit more space than you need. Just be sure to throw the exception if there's any risk of overrunning the buffer.

You don't *have* to use the buffer for this function though. For example, `pqxx::string_traits<bool>::to_buf` returns a compile-time constant string and ignores the buffer.

Even if you do use the buffer, your string does not *have* to start at the beginning of the buffer. For example, the integer conversions start by writing the *least* significant digit to the *end* of the buffer, and then writes the more significant digits before it. It was just more convenient.

Return a `pqxx::zview`. This is basically a `std::string_view`, but with one difference: a `zview` guarantees that there will be a valid zero byte right after the `string_view`. The zero byte is not counted as part of its size, but it will be there.

Expressed in code, this rule must hold:

```
void invariant(zview z)
{
  assert(z[std::size(z)] == 0);
}
```

Make sure you write your trailing zero *before* the `end`. If the trailing zero doesn't fit in the buffer, then there's just not enough room to perform the conversion.

Beware of locales when converting. If you use standard library features like `sprintf`, they may obey whatever locale is currently set on the system. That means that a simple integer like 1000000 may come out as "1000000" on your system, but as "1,000,000" on mine, or as "1.000.000" for somebody else, and on an Indian system it may be "1,00,000". Values coming from or going to the database should be in non-localised formats. You can use libpqxx functions for those conversions: `pqxx::from_string`, `pqxx::to_string`, `pqxx::to_buf`.

`into_buf`

This is a stricter version of `to_buf`. All the same requirements apply, but in addition you must write your string into the buffer provided, starting *exactly* at `begin`.

That's why this function returns just a simple pointer: the address right behind the trailing zero. If the caller wants to use the string, they can find it at `begin`. If they want to write a different value into the rest of the buffer, they can start at the location you returned.

`size_buffer`

Here you estimate how much buffer space you need for converting a `T` to a string. Be precise if you can, but pessimistic if you must. It's usually better to waste a few unnecessary bytes than to spend a lot of time computing the exact buffer space you need. And failing the conversion because you under-budgeted the buffer is worst of all.

Include the trailing zero in the buffer size. If your `to_buf` takes more space than just what's needed to store the result, include that too.

Make `size_buffer` a `constexpr` function if you can. It can allow the caller to allocate the buffer on the stack, with a size known at compile time.

Optional: Specialise `is_unquoted_safe`

When converting arrays or composite values to strings, libpqxx may need to quote values and escape any special characters. This takes time.

Some types though, such as integral or floating-point types, can never have any special characters such as quotes, commas, or backslashes in their string representations. In such cases, there's no need to quote or escape such values in arrays or composite types.

If your type is like that, you can tell libpqxx about this by defining:

```
namespace pqxx
{
template<> inline constexpr bool is_unquoted_safe<MY_TYPE>{true};
}
```

The code that converts this type of field to strings in an array or a composite type can then use a simpler, more efficient variant of the code. It's always safe to leave this out; it's *just* an optimisation for when you're completely sure that it's safe.

Do not do this if a string representation of your type may contain a comma; semicolon; parenthesis; brace; quote; backslash; newline; or any other character that might need escaping.

Optional: Specialise `param_format`

This one you don't generally need to worry about. Read on if you're writing a type which represents raw binary data, or if you're writing a template where *some specialisations* may contain raw binary data.

When you call parameterised statements, or prepared statements with parameters, libpqxx needs to your parameters on to libpq, the underlying C-level PostgreSQL client library.

There are two formats for doing that: *text* and *binary.* In the first, we represent all values as strings, and the server then converts them into its own internal binary representation. That's what the string conversions are all about, and it's what we do for almost all types of parameters.

But we do it differently when the parameter is a contiguous series of raw bytes and the corresponding SQL type is `BYTEA`. There is a text format for those, but we bypass it for efficiency. The server can use the binary data in the exact same form, without any conversion or extra processing. The binary data is also twice as compact during transport.

(People sometimes ask why we can't just treat all types as binary. However the general case isn't so clear-cut. The binary formats are not documented, there are no guarantees that they will be platform-independent or that they will remain stable, and there's no really solid way to detect when we might get the format wrong. But also, the conversions aren't necessarily as straightforward and efficient as they sound. So, for the general case, libpqxx sticks with the text formats. Raw binary data alone stands out as a clear win.)

Long story short, the machinery for passing parameters needs to know: is this parameter a binary string, or not? In the normal case it can assume "no," and that's what it does. The text format is always a safe choice; we just try to use the binary format where it's faster.

The `param_format` function template is what makes the decision. We specialise it for types which may be binary strings, and use the default for all other types.

"Types which *may* be binary"? You might think we know whether a type is a binary type or not. But there are some complications with generic types.

Templates like `std::shared_ptr`, `std::optional`, and so on act like "wrappers" for another type. A `std::optional<T>` is binary if `T` is binary. Otherwise, it's not. If you're building support for a template of this nature, you'll probably want to implement `param_format` for it.

The decision to use binary format is made based on a given object, not necessarily based on the type in general. Look at `std::variant`. If you have a `std::variant` type which can hold an `int` or a binary string, is that a binary parameter? We can't decide without knowing the individual object.

Containers are another hard case. Should we pass `std::vector<T>` in binary? Even when `T` is a binary type, we don't currently have any way to pass an array in binary format, so we always pass it as text. String escaping {#escaping} ===============

Writing queries as strings is easy. But sometimes you need a variable in there: `"SELECT id FROM user WHERE name = '" + name + "'"`.

This is dangerous. See the bug? If `name` can contain quotes, you may have an SQL injection vulnerability there, where users can enter nasty stuff like "`.'; DROP TABLE user`". Or if you're lucky, it's just a nasty bug that you discover when `name` happens to be "d'Arcy".

So, you'll need to *escape* the `name` before you insert it. This is where quotes and other problematic characters are marked as "this is just a character in the string, not the end of the string." There are several functions in libpqxx to do this for you.

SQL injection

To understand what SQL injection vulnerabilities are and why they should be prevented, imagine you use the following SQL statement somewhere in your program:

```
TX.exec(
    "SELECT number,amount "
    "FROM account "
    "WHERE allowed_to_see('" + userid + "','" + password + "')");
```

This shows a logged-in user important information on all accounts he is authorized to view. The userid and password strings are variables entered by the user himself.

Now, if the user is actually an attacker who knows (or can guess) the general shape of this SQL statement, imagine getting following password:

```
x') OR ('x' = 'x
```

Does that make sense to you? Probably not. But if this is inserted into the SQL string by the C++ code above, the query becomes:

```
SELECT number,amount
FROM account
WHERE allowed_to_see('user','x') OR ('x' = 'x')
```

Is this what you wanted to happen? Probably not! The neat `allowed_to_see()` clause is completely circumvented by the "`OR ('x' = 'x')`" clause, which is always `true`. Therefore, the attacker will get to see all accounts in the database!

Using the esc functions

Here's how you can fix the problem in the example above:

```
TX.exec(
    "SELECT number,amount "
    "FROM account "
    "WHERE allowed_to_see('" + TX.esc(userid) + "', "
    "'" + TX.esc(password) + "')");
```

Now, the quotes embedded in the attacker's string will be neatly escaped so they can't "break out" of the quoted SQL string they were meant to go into:

```
SELECT number,amount
FROM account
WHERE allowed_to_see('user', 'x'') OR (''x'' = ''x')
```

If you look carefully, you'll see that thanks to the added escape characters (a single-quote is escaped in SQL by doubling it) all we get is a very strange-looking password string–but not a change in the SQL statement.

Getting started

The most basic three types in libpqxx are the *connection*, the *transaction*, and the *result*.

They fit together as follows:

- You connect to the database by creating a `pqxx::connection` object (see @ref connections).
- You create a transaction object (see @ref transactions) operating on that connection. You'll usually want the `pqxx::work` variety.

  Once you're done you call the transaction's `commit` function to make its work final. If you don't call this, the work will be rolled back when the transaction object is destroyed.
- Until then, use the transaction's `exec`, `query_value`, and `stream` functions (and variants) to execute SQL statements. You pass the statements themselves in as simple strings. (See @ref streams for more about data streaming).
- Most of the `exec` functions return a `pqxx::result` object, which acts as a standard container of rows: `pqxx::row`.

  Each row in a result, in turn, acts as a container of fields: `pqxx::field`. See @ref accessing-results for more about results, rows, and fields.
- Each field's data is stored internally as a text string, in a format defined by PostgreSQL. You can convert field or row values using their `as()` and `to()` member functions.
- After you've closed the transaction, the connection is free to run a next transaction.

Here's a very basic example. It connects to the default database (you'll need to have one set up), queries it for a very simple result, converts it to an `int`, and prints it out. It also contains some basic error handling.

```cpp
#include <iostream>
#include <pqxx/pqxx>

int main()
{
  try
  {
    // Connect to the database.  In practice we may have to pass some
    // arguments to say where the database server is, and so on.
    // The constructor parses options exactly like libpq's
    // PQconnectdb/PQconnect, see:
    // https://www.postgresql.org/docs/10/static/libpq-connect.html
    pqxx::connection c;

    // Start a transaction.  In libpqxx, you always work in one.
    pqxx::work w(c);

    // work::exec1() executes a query returning a single row of data.
    // We'll just ask the database to return the number 1 to us.
    pqxx::row r = w.exec1("SELECT 1");

    // Commit your transaction.  If an exception occurred before this
    // point, execution will have left the block, and the transaction will
    // have been destroyed along the way.  In that case, the failed
    // transaction would implicitly abort instead of getting to this point.
    w.commit();

    // Look at the first and only field in the row, parse it as an integer,
    // and print it.
    //
    // "r[0]" returns the first field, which has an "as<...>()" member
    // function template to convert its contents from their string format
    // to a type of your choice.
    std::cout << r[0].as<int>() << std::endl;
  }
  catch (std::exception const &e)
  {
    std::cerr << e.what() << std::endl;
    return 1;
  }
}
```

This prints the number 1. Notice that you can keep the result object around after you've closed the transaction or even the connection. There are situations where you can't do it, but generally it's fine. If you're interested: you can install your own callbacks for receiving error messages from the database, and in that case you'll have to keep the connection object alive. But otherwise, it's nice to be able to "fire and forget" your connection and deal with the data.

You can also convert an entire row to a series of C++-side types in one go, using the @c as member function on the row:

```
pqxx::connection c;
pqxx::work w(c);
pqxx::row r = w.exec1("SELECT 1, 2, 'Hello'");
auto [one, two, hello] = r.as<int, int, std::string>();
std::cout << (one + two) << ' ' << std::strlen(hello) << std::endl;
```

Here's a slightly more complicated example. It takes an argument from the command line and retrieves a string with that value. The interesting part is that it uses the escaping-and-quoting function `quote` to embed this string value in SQL safely. It also reads the result field's value as a plain C-style string using its `c_str` function.

```
#include <iostream>
#include <stdexcept>
#include <pqxx/pqxx>

int main(int argc, char *argv[])
{
  try
  {
    if (!argv[1]) throw std::runtime_error("Give me a string!");

    pqxx::connection c;
    pqxx::work w(c);

    // work::exec() returns a full result set, which can consist of any
    // number of rows.
    pqxx::result r = w.exec("SELECT " + w.quote(argv[1]));

    // End our transaction here.  We can still use the result afterwards.
    w.commit();

    // Print the first field of the first row.  Read it as a C string,
    // just like std::string::c_str() does.
    std::cout << r[0][0].c_str() << std::endl;
  }
  catch (std::exception const &e)
  {
    std::cerr << e.what() << std::endl;
    return 1;
  }
}
```

You can find more about converting field values to native types, or converting values to strings for use with libpqxx, under @ref stringconversion. More about getting to the rows and fields of a result is under @ref accessing-results.

If you want to handle exceptions thrown by libpqxx in more detail, for example to print the SQL contents of a query that failed, see @ref exception. libpqxx {#mainpage} =======

@version @PQXXVERSION@ @author Jeroen T. Vermeulen @see http://pqxx.org @see https://github.com/jtv/libpqxx

Welcome to libpqxx, the C++ API to the PostgreSQL database management system.

Compiling this package requires PostgreSQL to be installed – including the C headers for client development. The library builds on top of PostgreSQL's standard C API, libpq. The libpq headers are not needed to compile client programs, however.

For a quick introduction to installing and using libpqxx, see the README.md file. The latest information can be found at http://pqxx.org/.

Some links that should help you find your bearings:

- @ref getting-started
- @ref thread-safety
- @ref connections
- @ref transactions
- @ref escaping
- @ref performance
- @ref transactor
- @ref datatypes Statement parameters {#parameters} =====================

When you execute a prepared statement (see @ref prepared), or a parameterised statement (using functions like `pqxx::connection::exec_params`), you may write special *placeholders* in the query text. They look like `$1`, `$2`, and so on.

If you execute the query and pass parameter values, the call will respectively substitute the first where it finds `$1`, the second where it finds `$2`, et cetera.

Doing this saves you work. If you don't use statement parameters, you'll need to quote and escape your values (see `connection::quote()` and friends) as you insert them into your query as literal values.

Or if you forget to do that, you leave yourself open to horrible SQL injection attacks. Trust me, I was born in a town whose name started with an apostrophe!

Statement parameters save you this work. With these parameters you can pass your values as-is, and they will go across the wire to the database in a safe format.

In some cases it may even be faster! When a parameter represents binary data (as in the SQL `BYTEA` type), libpqxx will send it directly as binary, which is a bit more efficient. If you insert the binary data directly in your query text, your CPU will have some extra work to do, converting the data into a text format, escaping it, and adding quotes.

Dynamic parameter lists

In rare cases you may just not know how many parameters you'll pass into your statement when you call it.

For these situations, have a look at `params`. It lets you compose your parameters list on the fly, even add whole ranges of parameters at a time.

You can pass a `params` into your statement as a normal parameter. It will fill in all the parameter values it contains into that position of the statement's overall parameter list.

So if you call your statement passing a regular parameter `a`, a `params` containing just a parameter `b`, and another regular parameter `c`, then your call will pass parameters `a`, `b`, and `c`. Or if the params object is empty, it will pass just `a` and `c`. If the params object contains `x` and `y`, your call will pass `a, x, y, c`.

You can mix static and dynamic parameters freely. Don't go overboard though: complexity is where bugs happen!

Generating placeholders

If your code gets particularly complex, it may sometimes happen that it becomes hard to track which parameter value belongs with which placeholder. Did you intend to pass this numeric value as `$7`, or as `$8`? The answer may depend on an `if` that happened earlier in a different function.

(Generally if things get that complex, it's a good idea to look for simpler solutions. But especially when performance matters, sometimes you can't avoid complexity like that.)

There's a little helper class called `placeholders`. You can use it as a counter which produces those placeholder strings, `$1`, `$2`, `$3`, et cetera. When you start generating a complex statement, you can create both a `params` and a `placeholders`:

```
pqxx::params values;
pqxx::placeholders name;
```

Let's say you've got some complex code to generate the conditions for an SQL "WHERE" clause. You'll generally want to do these things close together in your, so that you don't accidentally update one part and forget another:

```
if (extra_clause)
{
  // Extend the query text, using the current placeholder.
  query += " AND x = " + name.get();
  // Add the parameter value.
  values.append(my_x);
  // Move on to the next placeholder value.
  name.next();
}
```

Depending on the starting value of `name`, this might add to `query` a fragment like " `AND x = $3` " or " `AND x = $5` ". Performance features {#performance} ====================

If your program's database interaction is not as efficient as it needs to be, the first place to look is usually the SQL you're executing. But libpqxx has a few specialized features to help you squeeze more performance out of how you issue commands and retrieve data:

- @ref streams. Use these as a faster way to transfer data between your code and the database.
- `std::string_view` and `pqxx::zview`. In places where traditional C++ worked with `std::string`, see whether `std::string_view` or `pqxx::zview` will do. Of course that means that you'll have to look at the data's lifetime more carefully, but it'll save the computer a lot of copying.
- @ref prepared. These can be executed many times without the server parsing and planning them anew each time. They also save you having to escape string parameters.
- `pqxx::pipeline` lets you send queries to the database in batches, and continue other processing while they are executing.
- `pqxx::connecting` lets you start setting up a database connection, but without blocking the thread.

As always of course, don't risk the quality of your code for optimizations that you don't need! Prepared statements {#prepared} ===================

Prepared statements are SQL queries that you define once and then invoke as many times as you like, typically with varying parameters. It's basically a function that you can define ad hoc.

If you have an SQL statement that you're going to execute many times in quick succession, it may be more efficient to prepare it once and reuse it. This saves the database backend the effort of parsing complex SQL and figuring out an efficient execution plan. Another nice side effect is that you don't need to worry about escaping parameters. Some corporate coding standards require all SQL parameters to be passed in this way, to reduce the risk of programmer mistakes leaving room for SQL injections.

Preparing a statement

You create a prepared statement by preparing it on the connection (using the `pqxx::connection::prepare` functions), passing an identifier and its SQL text.

The identifier is the name by which the prepared statement will be known; it should consist of ASCII letters, digits, and underscores only, and start with an ASCII letter. The name is case-sensitive.

```
void prepare_my_statement(pqxx::connection &c)
{
  c.prepare(
      "my_statement",
      "SELECT * FROM Employee WHERE name = 'Xavier'");
}
```

Once you've done this, you'll be able to call `my_statement` from any transaction you execute on the same connection. For this, use the `pqxx::transaction_base::exec_prepared` functions.

```
pqxx::result execute_my_statement(pqxx::transaction_base &t)
{
  return t.exec_prepared("my_statement");
}
```

Parameters

Did I mention that prepared statements can have parameters? The query text can contain `$1`, `$2` etc. as placeholders for parameter values that you will provide when you invoke the prepared satement.

See @ref parameters for more about this. And here's a simple example of preparing a statement and invoking it with parameters:

```
void prepare_find(pqxx::connection &c)
{
  // Prepare a statement called "find" that looks for employees with a
  // given name (parameter 1) whose salary exceeds a given number
  // (parameter 2).
  c.prepare(
    "find",
    "SELECT * FROM Employee WHERE name = $1 AND salary > $2");
}
```

This example looks up the prepared statement "find," passes `name` and `min_salary` as parameters, and invokes the statement with those values:

```
pqxx::result execute_find(
  pqxx::transaction_base &t, std::string name, int min_salary)
{
```

```
  return t.exec_prepared("find", name, min_salary);
}
```

A special prepared statement

There is one special case: the *nameless* prepared statement. You may prepare a statement without a name, i.e. whose name is an empty string. The unnamed statement can be redefined at any time, without un-preparing it first.

Performance note

Don't assume that using prepared statements will speed up your application. There are cases where prepared statements are actually slower than plain SQL.

The reason is that the backend can often produce a better execution plan when it knows the statement's actual parameter values.

For example, say you've got a web application and you're querying for users with status "inactive" who have email addresses in a given domain name X. If X is a very popular provider, the best way for the database engine to plan the query may be to list the inactive users first and then filter for the email addresses you're looking for. But in other cases, it may be much faster to find matching email addresses first and then see which of their owners are "inactive." A prepared statement must be planned to fit either case, but a direct query will be optimised based on table statistics, partial indexes, etc.

Zero bytes

@warning Beware of "nul" bytes!

Any string you pass as a parameter will end at the *first char with value zero*. If you pass a string that contains a zero byte, the last byte in the value will be the one just before the zero.

So, if you need a zero byte in a string, consider that it's really a *binary string*, which is not the same thing as a text string. SQL represents binary data as the `BYTEA` type, or in binary large objects ("blobs").

In libpqxx, you represent binary data as a range of `std::byte`. They must be contiguous in memory, so that libpqxx can pass pointers to the underlying C library. So you might use `std::basic_string<std::byte>`, or `std::basic_string_view<std::byte>`, or `std::vector<std::byte>`. Streams {#streams} =======

Most of the time it's fine to retrieve data from the database using `SELECT` queries, and store data using `INSERT`. But for those cases where efficiency matters, there are two classes to help you do this better: `stream_from` and `stream_to`. They're less flexible than SQL queries, and there's the risk of losing your connection while you're in mid-stream, but you get some speed and memory efficiencies in return.

Both stream classes do data conversion for you: `stream_from` receives values from the database in PostgreSQL's text format, and converts them to the C++ types you specify. Likewise, `stream_to` converts C++ values you provide to PostgreSQL's text format for transfer. (On its end, the database of course converts values to and from their SQL types.)

Null values

So how do you deal with nulls? It depends on the C++ type you're using. Some types may have a built-in null value. For instance, if you have a `char const *` value and you convert it to an SQL string, then converting a `nullptr` will produce a NULL SQL value.

But what do you do about C++ types which don't have a built-in null value, such as `int`? The trick is to wrap it in `std::optional`. The difference between `int` and `std::optional<int>` is that the former always has an `int` value, and the latter doesn't have to.

Actually it's not just `std::optional`. You can do the same thing with `std::unique_ptr` or `std::shared_ptr`. A smart pointer is less efficient than `std::optional` in most situations because they allocate their value on the heap, but sometimes that's what you want in order to save moving or copying large values around.

This part is not generic though. It won't work with just any smart-pointer type, just the ones which are explicitly supported: `shared_ptr` and `unique_ptr`. If you really need to, you can build support for additional wrappers and smart pointers by copying the implementation patterns from the existing smart-pointer support.

stream_from

Use `stream_from` to read data directly from the database. It's faster than the transaction's `exec` functions if the result contains enough rows. But also, you won't need to keep your full result set in memory. That can really matter with larger data sets.

And, you can start processing your data right after the first row of data comes in from the server. With `exec()` you need to wait to receive all data, and then you begin processing. With `stream_from` you can be processing data on the client side while the server is still sending you the rest.

You don't actually need to create a `stream_from` object yourself, though you can if you want to. Two shorthand functions, @ref pqxx::transaction_base::stream and @ref pqxx::transaction_base::for_stream, can each create the streams for you with a minimum of overhead.

Not all kinds of queries will work in a stream. Internally the streams make use of PostgreSQL's `COPY` command, so see the PostgreSQL documentation for `COPY` for the exact limitations. Basic `SELECT` and `UPDATE ... RETURNING` queries should just work.

As you read a row, the stream converts its fields to a tuple type containing the value types you ask for:

```
auto stream pqxx::stream_from::query(
    tx, "SELECT name, points FROM score");
std::tuple<std::string, int> row;
while (stream >> row)
  process(row);
stream.complete();
```

As the stream reads each row, it converts that row's data into your tuple, goes through your loop body, and then promptly forgets that row's data. This means you can easily process more data than will fit in memory.

stream_to

Use `stream_to` to write data directly to a database table. This saves you having to perform an `INSERT` for every row, and so it can be significantly faster if you want to insert more than just one or two rows at a time.

As with `stream_from`, you can specify the table and the columns, and not much else. You insert tuple-like objects of your choice:

```
pqxx::stream_to stream{
    tx,
    "score",
    std::vector<std::string>{"name", "points"}};
for (auto const &entry: scores)
    stream << entry;
stream.complete();
```

Each row is processed as you provide it, and not retained in memory after that.

The call to `complete()` is more important here than it is for `stream_from`. It's a lot like a "commit" or "abort" at the end of a transaction. If you omit it, it will be done automatically during the stream's destructor. But since destructors can't throw exceptions, any failures at that stage won't be visible in your code. So, always call `complete()` on a `stream_to` to close it off properly! Thread safety {#thread-safety} =============

This library does not contain any locking code to protect objects against simultaneous modification in multi-threaded programs. Therefore it is up to you, the user of the library, to ensure that your threaded client programs perform no conflicting operations concurrently.

Most of the time this isn't hard. Result sets are immutable, so you can share them between threads without problem. The main rule is:

@li Treat a connection, together with any and all objects related to it, as a "world" of its own. You should generally make sure that the same "world" is never accessed by another thread while you're doing anything non-const in there.

That means: don't issue a query on a transaction while you're also opening a subtransaction, don't access a cursor while you may also be committing, and so on.

In particular, cursors are tricky. It's easy to perform a non-const operation without noticing. So, if you're going to share cursors or cursor-related objects between threads, lock very conservatively!

Use `pqxx::describe_thread_safety` to find out at runtime what level of thread safety is implemented in your build and version of libpqxx. It returns a `pqxx::thread_safety_model` describing what you can and cannot rely on. A command-line utility `tools/pqxxthreadsafety` prints out the same information.