Misc C++ Topics

Introduction

These notes cover topics I didn't otherwise find a natural place to include.

Type Aliases

C++ allows a programmer to create an alias for a data type. There are two ways to do this, one using the `typedef` keyword, the other using the, well, `using` keyword.

The general form for a `typedef` alias is:

```
typedef [type] [alias];
```

Where:

- `typedef` is a required keyword.
- `[type]` is an already existing type, or a type declaration in this position.
- `[alias]` is the alias name used to refer to the [type].

A couple simple examples using `typedef`:

```
typedef int MyInt;
typedef std::string MyString;
```

Given these two `typedef`s, they can be used in code as:

```
MyInt a = 10;
MyString s = "Hi Mom!";
```

The same two aliases can be defined with the using keyword like:

```
using MyInt = int;
using MyString = std::string;
```

What is the difference between the two? According to the C++ standard, nothing. Since the C++ 11 standard where the `using` approach was introduced, it is now the recommended idiom (per Scott Meyers in Effective Modern C++). The reason for the using technique is to follow other assignment patterns, where the expression on the right is assigned to the value on the left.

There is one thing using can do, that typedef can't.

```
template <typename T>
class Original {};

template <typename T>
using Alias = Original<T>;
```

`Alias<T>` is a type alias for `Original<T>`.

Random Number Generation

C++ has a sophisticated built-in capability for generating random numbers. It is possible to easily generate uniformly distributed random numbers, or generate random numbers to a distribution, such as normal, binomial, exponential, and others. The `<random>` library contains all of the random number generation capabilities.

You may be familiar with legacy C random library functions `rand` and `srand`, forget about them, don't ever use them!

Random Device

The first concept to understand is that of the `std::random_device`. This is a non-deterministic uniform random number generator; specifically it is a non-deterministic uniform random *bit* generator. The `std::random_device` generates numbers based on a hardware device on the computer (and I don't know what component(s) is used). The C++ standard says a device can implement a pseudo random number generator if no hardware support is available; plan on it being a pseudo-random number generator unless you discover otherwise. This class implements the `()` operator, which updates its internal state and returns a generated value. Let's take a look at some code to understand it:

```
std::random_device rd;

for (int i = 0; i < 10; i++)
{
	std::cout << rd() << std::endl;
}
```

This code produces ten random numbers, pretty simple.

While this is great, this isn't what you want to use to generate random numbers, for example, uniform random numbers over some range. You might be tempted to use the old divide and multiply approach to get a random number in a range, but that isn't the purpose of this device.

Random Number Engines

The random number library includes three types of random number engines, *linear congruential*, *Mersenne Twister*, and *subtract with carry*. Which engine to choose is application dependent and beyond the scope of this class. At a glance, however, linear congruential is fast and "good enough" for most things, whereas the Mersenne Twister is slower but has a very long non-repeating sequence.

An engine produces pseudo-random numbers, from an initial seed. This is different from the `std::random_device` which (can) generates non-deterministic random numbers. Similar to the `std::random_device`, an engine implements the `()` operator which updates its internal state and returns the next value in the sequence. What to seed the engine with? How about using `std::random_device` to generate a value! Let's take a look at some code to see how this all works:

```
std::random_device rd;
std::default_random_engine engine{rd()};

for (int i = 0; i < 10; i++)
{
	std::cout << engine() << std::endl;
}
```

This code produces ten random numbers. Remember, the difference between the engine and the device is that the engine will produce the exact same sequence of random numbers given the same seed. There is no seed for a device, it generates random numbers based on a hardware component(s). If we use a device to seed an engine, we'll get a different pseudo-random sequence each time.

While a random number engine produces random numbers, you still don't want to use it and the good old fashioned divide and multiply approach. Instead, you want to use a random number distribution object.

Random Number Distributions

The next component is to generate numbers according to a distribution. For example, it might be desired to generate uniformly distributed numbers between 0.0 and 1.0. To do this, we'll use the `std::uniform_real_distribution`, well, distribution. A distribution requires a random number engine to generate a random sequence, which it then uses in turn to produce a distribution according to various parameters. For the `std::uniform_real_distribution` the code looks like:

```
std::random_device rd;
std::default_random_engine engine{rd()};
std::uniform_real_distribution<double> dist(0.0, 1.0);

for (int i = 0; i < 10; i++)
{
	std::cout << dist(engine) << std::endl;
}
```

This code produces ten uniformly distributed random numbers in the range of [0.0, 1.0).

The `<random>` library includes a good number of distributions with several types in each of the following categories: *Uniform*, *Bernoulli*, *Poisson*, *Normal*, and *Sampling*. The specific parameters for each distribution differ according to their needs. When using a distribution refer to its documentation.

Random Shuffle

In addition to generating random numbers, it is possible to use the same generators to randomly order elements in a container. The `<algorithm>` library includes an `std::shuffle` that can be used on a container. consider the following code:

```
std::vector<int> v = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
std::array<int, 10> v2 = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

std::random_device rd;
std::mt19937 generator{rd()};

std::shuffle(v.begin(), v.end(), generator);
std::shuffle(v2.begin(), v2.end(), generator);
// Alternatively using ranges
std::ranges::shuffle(v, generator);
std::ranges::shuffle(v2, generator);

for (auto number : v)
{
	std::cout << number << " ";
}
std::cout << std::endl;

for (auto number : v2)
{
	std::cout << number << " ";
}
std::cout << std::endl;
```

Date & Time Handling

C++ provides an excellent date and time handling library `<chrono>`. There are four main concepts to understand when speaking of date and time handling: durations, time points, clocks, and claendars. A duration is a span of time, such as hours, or minutes. A time point refers to a specific point in time, such as the 4th of July, or the start of a race (in milliseconds, for example). A duration is the difference between two time points. A clock is necessary in order to obtain the current time. Finally, a calendar is used to represent a day, month, year, or weekday (among others).

Duration

A duration is composed of two pieces of information, a *period* (a *fraction*) and a *count*. The period is a unit of time expressed as a fraction of a second; where the fraction can be less than, equal to, or greater than 1. The count is the number of periods in the duration. For example, a duration might have a period of 1 and a count of 10, representing 10 seconds. Another duration might have a period of .001 (1 / 1000) and a count of 10,000, which also represents 10,000 milliseconds, or 10 seconds.

Let's look at a few examples:

```
std::chrono::duration<std::uint32_t, std::ratio<1, 1>> seconds(10);
std::chrono::duration<std::uint32_t, std::ratio<1, 1000>> milliseconds(10000);
std::chrono::duration<std::uint32_t, std::ratio<60, 1>> minutes(4);
```

An `std::chrono::duration` is defined by an underlying data type as the first template parameter, followed by the ratio defining the period. The first line of code defines a duration with an underlying unsigned integer type and a ratio of 1:1, which is a second. The second line of code defines a duration with an underlying unsigned integer type and a ratio of 1:1000, which is a millisecond. Finally, the third example using a ratio of 60:1, which is a minute. The underlying data type doesn't have to be integral (or unsigned), it can be floating point to allow for fractional representations of the period, but is generally not used.

As you have already noticed, the declaration of a duration type is quite long. We can create type aliases (the `using` keyword) and use them instead:

```
using seconds = std::chrono::duration<std::uint32_t, std::ratio<1, 1>>;
using milliseconds = std::chrono::duration<std::uint32_t, std::ratio<1, 1000>>;
using minutes = std::chrono::duration<std::uint32_t, std::ratio<60, 1>>;


seconds s(10);
milliseconds ms(10000);
minutes min(4);
```

Also, the `<chrono>` library provides types for common durations, such as hours, minutes, seconds, etc.

```
std::chrono::seconds s(10);
std::chrono::milliseconds ms(10000);
std::chrono::minutes min(4);
```

By defining variables in terms of durations, it is possible to convert between them automatically. In other words, a duration expressed as seconds can be added to a duration of minutes and the minutes duration adds that many seconds. Consider the following code:

```
std::chrono::seconds seconds(30);
std::chrono::minutes minutes(1);
std::chrono::seconds minutesToSeconds = minutes;

std::cout << seconds.count() << " seconds" << std::endl;
std::cout << minutes.count() << " minutes" << std::endl;
```

```
std::cout << minutesToSeconds.count() << " seconds" << std::endl;

seconds += minutes;
std::cout << seconds.count() << " seconds" << std::endl;
```

When run, this code produces the following output:

```
30 seconds
1 minutes
60 seconds
90 seconds
```

Notice the third line of code, where a duration of minutes is assigned to a duration of seconds. When reported in the sixth line of code, it reports as 60 seconds. Also, note the seventh line of code where minutes is added to seconds. When reported in the next line of code, it correctly reports as 90 seconds.

Duration Cast

Duration conversions from a less precise type to a more precise type happen automatically, in the same way C++ automatically converts from an `int` to a `float` without saying anything. However, conversions from a precise type to a less precise type do not happen automatically. To overcome the compile error, it is necessary to use an `std::chrono::duration_cast`:

```
std::chrono::seconds seconds(30);
std::chrono::minutes minutes(1);

minutes += std::chrono::duration_cast<std::chrono::minutes>(seconds);
std::cout << minutes.count() << " minutes" << std::endl;
```

Consider the following code:

```
std::chrono::duration<float, std::ratio<60, 1>> minutes(1);
std::chrono::duration<std::uint32_t, std::ratio<1, 1>> seconds(30);

minutes += seconds;
std::cout << minutes.count() << " minutes" << std::endl;
```

The reason this code compiles is because the minutes duration can handle fractional seconds. However, if the underlying data type for minutes is an integral type, a compiler error occurs.

Clock

A clock is defined by some starting point (also known as an epoch) and a rate at which time passes, a tick rate. Clocks are used to capture a time point (discussed next). There are many clocks provided in the `<chrono>` library, these are described next.

- `std::chrono::system_clock` – This is the (computer) system real time clock, also called a *wall clock*. The epoch for this clock is undefined, but usually set to January 1st, 1970 (same as the Unix Time).
- `std::chrono::steady_clock` – A monotonic clock. By monotonic it means that the time represented by the clock cannot decrease with subsequent calls to the `::now` function. This clock does not (necessarily) have a relationship to the wall clock time of the system. This clock is recommended for use in measuring time intervals.
- `std::chrono::high_resolution_clock` – This is a clock that provides the shortest possible tick, or period, possible for the system (or implementation). This clock might be an alias of the system or steady clock.
- `std::chrono::utc_clock` – Used to represent Coordinated Universal Time (UTC). This clock measures time since 00:00:00 UTC, Thursday, January 1st, 1970; it also includes leap seconds.

- `std::chrono::tai_clock` – This clock represents International Atomic Time (TAI). It measures time since 00:00:00 January 1st, 1958, and offset 10 seconds ahead of UTC at that date (to account for leap seconds?).
- `std::chrono::gps_clock` – This clock represents Global Positioning System (GPS) time, measured since 00:00:00 January 6th, 1980 UTC.
- `std::chrono::file_clock` – This is an alias for the clock used to represent filesystem times, `std::filesystem::file_time_type`, no epoch is defined for this clock.

All clocks provide the same type and functional information. A clock exposes the type used to track the number of periods in a duration (rep), the `std::ratio` used to represent a period, the duration type, and time point type. For example, the following code shows how to report the period ratio:

```
auto periodSystem = std::chrono::system_clock::period();
std::cout << "System Clock Period          : " << periodSystem.num << " / "
      << periodSystem.den << std::endl;

auto periodSteady = std::chrono::steady_clock::period();
std::cout << "Steady Clock Period          : " << periodSteady.num << " / "
      << periodSteady.den << std::endl;

auto periodHigh = std::chrono::high_resolution_clock::period();
std::cout << "High Resolution Clock Period : " << periodHigh.num << " / "
      << periodHigh.den << std::endl;
```

When executed on my computer, the steady and high resolution clock uses the same, higher resolution, period (`std::ratio`), whereas the system clock uses a lower resolution period.

Time Points

A time point represents a point in time, stored as a duration from the epoch of a clock. Therefore, a time point records both the clock it is based upon and the duration from the clock epoch. There are conversions available to convert time points between clocks, but that is left for the reader to investigate (it is pretty simple).

All clocks have a `::now` method, that return the current time as an `std::time_point`, based on the clock type. For example, `::now` on the system clock is an `std::chrono::time_point<std::chrono::system_clock>` type. Consider the following code:

```
auto systemNow = std::chrono::system_clock::now();
std::cout << "System clock periods since clock epoch          : " <<
systemNow.time_since_epoch().count() << std::endl;

auto steadyNow = std::chrono::steady_clock::now();
std::cout << "Steady clock periods since clock epoch          : " <<
steadyNow.time_since_epoch().count() << std::endl;

auto highResolutionNow = std::chrono::high_resolution_clock::now();
std::cout << "High resolution clock periods since clock epoch : " <<
highResolutionNow.time_since_epoch().count() << std::endl;

std::chrono::hours systemHours =
std::chrono::duration_cast<std::chrono::hours>(systemNow.time_since_epoch());
std::cout << "System clock hours since epoch          : "
      << systemHours.count() << std::endl;

std::chrono::hours steadyHours =
std::chrono::duration_cast<std::chrono::hours>(steadyNow.time_since_epoch());
std::cout << "Steady clock hours since epoch          : "
      << steadyHours.count() << std::endl;
```

```
std::chrono::hours highResolutionHours =
std::chrono::duration_cast<std::chrono::hours>(highResolutionNow.time_since_epoch());
std::cout << "High resolution clock hours since epoch : "
      << highResolutionHours.count() << std::endl;
```

The first three segments report the number of periods since each clock's epoch. That may not be a lot of use, but maybe something like the number of hours is more useful. The next three segments of code perform a duration cast from the epoch counts to hours and reports those hours. With the duration cast, you don't have to know the period for the source, it is embedded in the type, you only need to provide the duration type conversion.

Timing Things

Now that you understand the different clocks, how to obtain the current time (point), and durations, all those can be put together to time how long something takes. For example, let's time how long it takes to recursively compute some Fibonacci numbers. Consider the following code:

```
std::uint64_t fibonacci(std::uint16_t n)
{
    if (n == 0)
        return 0;
    if (n == 1)
        return 1;

    return fibonacci(n - 2) + fibonacci(n - 1);
}

auto start = std::chrono::steady_clock::now();
auto result = fibonacci(24);
auto end = std::chrono::steady_clock::now();
auto diff = std::chrono::duration_cast<std::chrono::microseconds>(end - start);

std::cout << "Fibonacci(24) = " << result << std::endl;
std::cout << "fibonacci(24) took " << diff.count() << " microseconds" << std::endl;
```

The function is an old fashioned recursive approach to computing the Fibonacci number. Following the function, the next three lines of code are used to capture the starting and stopping time (points) before and after computing the Fibonacci number. The fourth line takes the difference between `start` and `end` time points, resulting in a duration, then casts it to microseconds. The duration cast didn't have to be microseconds, it could be anything else we want or need. The last two lines report the Fibonacci number along with the length of time it took to compute.

Calendar

Beginning with C++ 20 the language (standard library) has robust support for calendar data types and operations. Unless otherwise specified, the calendar is a Gregorian calendar; this matters for years, weekdays, and leap years.

- `std::chrono::day` – Represents a day of a month
- `std::chrono::month` – Represents a month of a year; `std::chrono::January`, `std::chrono::February`, etc.
- `std::chrono::year` – Represents a year
- `std::chrono::weekday` – Represents a day of the week; `std::chrono::Sunday`, `std::chrono::Monday`, etc.
- `std::chrono::month_day` – Represents a specific day of a month
- `std::chrono::year_month_day` – Represents a specific year, month, and day
- and much more…

Let's take a look at an interesting use of the / operator to declare a year_month_day variable. Without using the `/` operator one can be declared as…

```
auto ymd = std::chrono::year_month_day(
       std::chrono::year(2001),
       std::chrono::month(1),
       std::chrono::day(1));
```

Where this is the date January 1st, 2001; the first day of the 21st century.

An alternative syntax for this declaration is…

```
auto ymd = std::chrono::January / 1 / 2001;
```

These aren't the only two ways to form a date, but they are probably the ones most commonly used. Similar alternatives are provided for creating a `month_day`, `month_weekday`, or `month_weekday_last`, and others.

Adding Days, Months, Years

For the most part, modifying a calendar type is how you think it should be done, add some number of days, months, or years to the object.

```
auto ymd = std::chrono::January / 1 / 2001;
ymd += std::chrono::months{2};
ymd += std::chrono::years{1};
```

But interestingly, you can't add days in the same way, which is frustrating. I have read a Stack Overflow discussion that talks about a "date" being a `time_point` and that has something to do with the reason. To my understanding, there is no "date" type in `std::chrono`, there are `time_points` and then the various calendar types. A `year_month_day` sure sounds like a `time_point` (with day as its precision) when you think about it. However, when I look at the code in the standard library it is not a `time_point`, it is a class with year, month, day attributes and various methods available on the type. I don't know the **real** reason adding of days isn't directly supported, but it seems like someone did some mental gymnastics to justify not providing it (the Stack Overflow claims to provide a justification, one that I don't accept as valid). In any case, here is how to do it…

Step 1: Convert the `year_month_day` to a number of days

```
auto days = std::chrono::sys_days{ymd};
```

Step 2: Add the desired number of days to this object

```
days += std::chrono::days{7};
```

Step 3: Convert the number of days back into a `year_month_day` object

```
ymd = std::chrono::year_month_day{days};
```

Structured Binding

Structured binding allows the sub-objects (of a complex type) or elements (of an array) to be bound (by value or by reference) to multiple variables in an identifier list. This is easier to explain by example, let's start with the elements of an array.

```
std::array<int, 4> primes{2, 3, 5, 7};
auto [one, two, three, four] = primes;
std::cout << one << ", " << two << ", " << three << ", " << four << std::endl;
```

The variables `one`, `two`, `three`, and `four` are placed into the current scope.

If desired, it is possible to bind these values by reference, as:

```
auto& [one, two, three, four] = primes;
```

The number of elements in the identifier-list must match the number of elements in the array.

Another type of binding is to the sub-objects of a complex data type. Consider the following code:

```
class MyClass
{
  public:
    MyClass(int a, double b) :
        m_a(a),
        m_b(b)
    {
    }

    int m_a;
    double m_b;
};

...

MyClass data(1, 2.2);
auto [x, y] = data;
std::cout << x << ", " << y << std::endl;
```

This is also called *destructuring*, as the code is separating the items in the structure into individual items.

The order of declaration in the class is the order of binding to the items in the identifier-list. Only public members are bound. In fact, the source for the destructuring can have only public members. If there are private members an error happens during compilation, either not enough items in the initializer-list or private members in the source object.

Another use for this is when iterating through collections, particularly from the standard library. Consider the following collection:

```
std::map<int, std::string> lookup{
       {1, "Logan"},
       {2, "Smithfield"},
       {3, "Hyrum"},
       {4, "Nibley"},
       {5, "Hyde Park"}};
```

Each element in the `std::map` is an `std::pair`. `std::pair` exposes `.first` and `.second` members, which are used as the key and value, respectively, for the `std::map`.

Without structured binding, iteration looks like:

```
for (const auto& item : lookup)
{
```

```
        std::cout << item.first << " - " << item.second << std::endl;
}
```

With structured binding, iteration looks like:

```
for (const auto& [key, value] : lookup)
{
        std::cout << key << " - " << value << std::endl;
}
```

Returning Multiple Values

A common need is to return multiple values from a function. There are three common ways this has been done historically in C++.

1. Use reference parameters. Problem is this (may) requires passing in uninitialized values to the function, needing to declare them before use.
2. Use a standard library object like `std::tuple` or using `std::tie`.
3. Use a custom data structure that contains public only members.

With the addition of structured binding, this can be used as a way to cleanly return multiple items from a function. Consider the following code:

```
auto returnMultipleItems()
{
    return std::make_tuple(1, 2.0, std::string("Hi Mom"));
}
```

… used as …

```
auto [a, b, c] = returnMultipleItems();
std::cout << a << ", " << b << ", " << c << "" << std::endl;
```

You don't know about `std::tie`?!

```
int d;
double e;
std::string f;
std::tie(d, e, f) = returnMultipleItems();
std::cout << d << ", " << e << ", " << f << "" << std::endl;
```

Now you do.

Optional Values

There are times when it is desired to perform a computation or lookup a value that may or may not have a valid result. One way of handling this is to write a function that returns a `bool` `true`/`false` if the operation was successful or not and then use an *out reference* parameter for the result. Another, ***shamefully horrible***, alternative is to return a *sentinel* value that indicates failure, and if it isn't the sentinel, then it is a valid result. Another, not too bad, idea is to return an `std::pair` where one of the values in the pair is a `bool` indicating success or failure. None of these approaches are necessary in C++, instead a helper class may be used to *optionally* hold a value. The class is called `std::optional`.

The idea of `std::optional` is that it is used to indicate the presence of a value or not, and if a value is present, provide access to the value. There are a number of ways to construct an `std::optional` with or without a value, they are demonstrated in the code sample below. `std::optional` exposes an `.has_value()` method that returns `true`/`false` indicating presence of a value or not, and then a `.value()` method that returns the contained value. The use of `std::optional` is explained quite easily through demonstration.

Starting with a `Person` class, as follows...

```
class Person
{
  public:
    Person(std::string name, std::uint8_t age, std::string id) :
        m_name(name),
        m_age(age),
        m_id(id)
    {}

    const std::string& getName() { return m_name; }
    std::uint8_t getAge() { return m_age; }
    const std::string& getId() { return m_id; }

  private:
    std::string m_name;
    std::uint8_t m_age;
    std::string m_id;
};
```

We'll use this class to create a database of students. The database has the following type…

```
using Database = std::unordered_map<std::string, std::shared_ptr<Person>>;
```

Create a simple database with…

```
Database createDatabase()
{
    Database db;
    db["a111"] = std::make_shared<Person>("Student One", 18, "a111");
    db["a222"] = std::make_shared<Person>("Student Two", 19, "a222");
    db["a333"] = std::make_shared<Person>("Student Three", 20, "a333");
    db["a444"] = std::make_shared<Person>("Student Four", 21, "a444");
    db["a555"] = std::make_shared<Person>("Student Five", 22, "a555");
    return db;
}
```

This database should be searchable by student id. If the id is found, the student is returned, otherwise nothing is returned. This is a perfect use-case for `std::optional`. We'll write a find function as follows…

```
std::optional<std::shared_ptr<Person>> findStudent(const std::string& id, const Database& db)
{
    if (db.find(id) != db.end())
    {
        //return std::optional<std::shared_ptr<Person>>(db.at(id));
        //return std::make_optional<std::shared_ptr<Person>>(db.at(id));
        //return std::make_optional(db.at(id));
        //return {db.at(id)};
        return db.at(id);
    }

    // return std::nullopt;
    return {};
}
```

The lines of code you see commented are alternatives for how to return `std::optional`, with a value or not.

After searching the database, we need a simple function to report the student details…

```
void reportStudent(std::shared_ptr<Person> student)
{
    std::cout << "--- " << student->getName() << " ---" << std::endl;
    std::cout << "\tage: " << static_cast<unsigned int>(student->getAge()) << std::endl;
    std::cout << "\tid : " << student->getId() << std::endl;
}
```

Finally, let's demonstrate the use of returning and testing the optional result…

```
    Database db = createDatabase();

    std::optional find1 = findStudent("a222", db);
    std::optional find2 = findStudent("a666", db);

    if (find1)
    {
        reportStudent(find1.value());
    }
    if (find2)
    {
        reportStudent(find2.value());
    }
```

It is possible to directly test `find1` and `find2`, because `std::optional` overloads `operator bool`. Alternatively, the `.has_value()` method can be called to test for the presence of a value. If the value exists, the `.value()` method can be called to obtain the value. If `.value()` is called with no value present, an exception is thrown. Another possibility, is to use the `.value_or()` method to return a default value (that you specify) in the case no value exists in the optional.

```
auto notFound = std::make_shared<Person>("not found", 0, "n/a");
reportStudent(find1.value_or(notFound));
reportStudent(find2.value_or(notFound));
```

“Any” Value Type

There are times when it is needed for an object to hold values of many different or arbitrary types. For example, the cells in a spreadsheet may contain strings, numbers, currency, formulas, references to other cells, images, sounds, and possibly many other data types. For components like this, `std::any` can be used. It is a type-safe container that can hold a single value of any type. The value stored in `std::any` must be a copy constructable type, having either default or code-defined copy/move constructor(s).

Following on from the `std::optional` code samples from above, `std::any` can be used in the following ways…

```
std::any v = 1;
std::cout << v.type().name() << " : " << std::any_cast<int>(v) << std::endl;
```

The variable `v` is declared and the integer value `1` is assigned. In order to use the value stored in `std::any`, an `std::any_cast<>` to the type is required. If an incorrect cast is specified, an exception is thrown. In other words, while `std::any` can hold any type, the programmer still has the responsibility to know the type at the time of its use.

```
v = 2.0;
std::cout << v.type().name() << " : " << std::any_cast<double>(v) << std::endl;
```

The value is changed to 2.0, and the type is now a `double`.

```
v = std::string("This is string value");
std::cout << v.type().name() << " : " << std::any_cast<std::string>(v) << std::endl;

v = Person("Student Six", 23, "a666");
std::cout << std::any_cast<Person>(v).getName() << std::endl;
```

Complex user-defined data types may be assigned to `std::any`.

```
v = std::shared_ptr<Person>(new Person("Student Seven", 24, "a777"));
reportStudent(std::any_cast<std::shared_ptr<Person>>(v));
```

Again, any (copy constructable) type may be assigned, including an `std::shared_ptr`. In this example, the s`td::shared_ptr` is created through its own constructor, rather than through `std::make_shared`. When the value goes out of scope or a new one assigned, the `std::shared_ptr` will also go out of scope as expected.

When to use `std::any`, some thoughts:

- Parsing command line parameters.
- In parsing files, if you can’t specify the needed types.
- Passing messages in a type-safe manner.
- User interface controls that need to hold many/any value types.

I don’t have a strong recommendation to use `std::any` in many places, instead I point you towards `std::variant` which has much stronger type guarantees.

TODO: Here is some std::variant code

```
#include <iostream>
#include <variant>
#include <vector>

class MyVisitor
{
    public:
        void operator()(int a)
        {
            std::cout << "int: " << a << std::endl;
        }

        void operator()(double a)
        {

            std::cout << "double: " << a << std::endl;
        }

        void operator()(bool a)
        {
            std::cout << "bool: " << a << std::endl;
        }
};

int main()
{
    std::vector<std::variant<int, double, bool>> data;

    data.push_back(42);
    data.push_back(3.14159);
    data.push_back(true);
```

```
    for (auto&& value : data)
    {
        std::visit(MyVisitor(), value);
    }

    std::variant<int, double, bool> value = 6;
    std::cout << std::get<int>(value) << std::endl;

    return 0;
}
```

Constant, or Compile-Time, Expressions

You already understand the concept of an expression. This is something the compiler evaluates and translates it into code that is executed at run-time. For most expressions, it makes sense for them to be evaluated at run-time because the data used in the computation isn't known until the program is running. The data may come from a file, database, network stream, or user input. However, there are times when the data for an expression is known at the time the code is being compiled. Therefore, rather than generating instructions to evaluate the expression at run-time, the compiler can evaluate the expression at compile-time and place the result directly in the executable, no run-time cost necessary! C++ calls these c*onstant expressions*, and uses the `constexpr` keyword to define them.

Let's begin with a simple example…

```
constexpr auto sum(int a, int b)
{
    return a + b;
}

int main()
{
    constexpr auto total = sum(2, 6);
    std::cout << total << std::endl;

    return 0;
}
```

The function `sum` is defined as a `constexpr` and then used in main to compute the `total` of 2 and 6. When this program is run, the expected total of 8 is printed. The interesting part is the code generated by the compiler. I've used Visual Studio to generate the annotated assembly code output for this code, it is shown next…

```
; 13   :     constexpr auto total = sum(2, 6);
; 14   :
; 15   :     std::cout << total << std::endl;

      mov    rcx, QWORD PTR __imp_?cout@std@@3V?$basic_ostream@DU?
$char_traits@D@std@@@1@A
      mov  edx, 8
      call   QWORD PTR __imp_??6?$basic_ostream@DU?
$char_traits@D@std@@@std@@QEAAAEAV01@H@Z
      mov    rcx, rax
      lea    rdx, OFFSET FLAT:??$endl@DU?$char_traits@D@std@@@std@@YAAEAV?
$basic_ostream@DU?$char_traits@D@std@@@0@AEAV10@@Z ;
std::endl<char,std::char_traits<char> >
      call   QWORD PTR __imp_??6?$basic_ostream@DU?
$char_traits@D@std@@@std@@QEAAAEAV01@P6AAEAV01@AEAV01@@Z@Z
```

The first three lines, each beginning with semi-colons, show the original C++ source code, the remainder of the code is the assembly generated by the compiler. I've highlighted the interesting part. Notice the value of 8 is directly moved into a CPU register. What you are seeing here is the compiler evaluated the `constexpr sum` at compile time and determined the value of total was 8 and directly placed that into the executable. No run-time calling of the `sum` function occurs, the computation occurred at compile-time.

Let’s try another example using recursion, a Fibonacci sequence generator…

```
constexpr auto fibonacci(unsigned int n)
{
    if (n == 0 || n == 1)
        return 1;

    return fibonacci(n - 1) + fibonacci(n - 2);
}


constexpr auto fibN = fibonacci(11);
std::cout << fibN << std::endl;
```

When executed, this code shows the number 144, which is the correct value. Inspection of the compiler generated assembly shows the value of 144 directly placed into the executable, rather than a call to the function and run-time evaluation.

```
; 23   :
; 24   :     constexpr auto fibN = fibonacci(11);
; 25   :     std::cout << fibN << std::endl;

      mov    rcx, QWORD PTR __imp_?cout@std@@3V?$basic_ostream@DU?
$char_traits@D@std@@@1@A
     mov  edx, 144                ; 00000090H
```

In addition to this function being available for compile-time evaluation, it can also be used for run-time evaluation. Consider the following code…

```
std::cout << "Enter a Fibonacci number to compute: ";
std::uint16_t input;
std::cin >> input;
std::cout << "The value is: " << fibonacci(input) << std::endl;
```

Improving The switch Statement

Languages like Java and C# provide a native ability to switch on string values. Let's add a similar capability to C++.

The switch statement in C++ must switch on integral values, and the case statement values must be known at compile time. We can write a function to take an `std::string` (or similar) and compute a hash value based on the contents of the string. The key to this hash function is to write it in a way that can be evaluated at compile-time for string literals and run-time for data only known at run-time. We'll use `constexpr` to write the hash function, as follows…

```
// Hash function based on reference here:
//     https://stackoverflow.com/questions/8317508/hash-function-for-a-string
constexpr unsigned int string_hash(const char* s)
{
    constexpr auto A = 54059;
    constexpr auto B = 76963;
    constexpr auto C = 86969;
    constexpr auto FIRSTH = 37;

    unsigned h = FIRSTH;
    while (*s)
    {
        h = (h * A) ^ (s[0] * B);
        s++;
    }
    return h;
}
```

With this hash function available, it can be used as…

```
void switchOnString(const std::string name)
{
    // Names taken from: https://en.wikipedia.org/wiki/Alice_and_Bob
    switch (string_hash(name.c_str()))
    {
        case string_hash("Bob"):
            std::cout << "It was Bob" << std::endl;
            break;
        case string_hash("Alice"):
            std::cout << "It was Alice" << std::endl;
            break;
        case string_hash("Carol"):
            std::cout << "It was Carol" << std::endl;
            break;
    }
}
```

Pretty cool!

Should I declare constant literals as `const` or `constexpr`?

Reference: https://stackoverflow.com/questions/14116003/difference-between-constexpr-and-const

Consider that we define the golden ratio two different ways:

```
const auto GOLDEN_RATIO1 = 1.6180339887498948482;
constexpr auto GOLDEN_RATIO2 = 1.6180339887498948482;
```

Both of the declarations ensure the value can not be modified. But the `constexpr` version means `GOLDEN_RATIO2` is known at compile-time, and therefore, can be used during compile-time evaluation.

```
static_assert(GOLDEN_RATIO1 ==  1.6180339887498948482);     // Results in a failure
static_assert(GOLDEN_RATIO2 ==  1.6180339887498948482);     // Evaluates to true
```

For most applications, `const` is fine, but if the value is needed for compile-time evaluation, then `constexpr` is necessary.

std::format

Prior to C++ 20 there were two techniques for formatting strings (output) in C++. The first is the `printf/puts/putchar` family, with its origin in the C language, and the `<iostream>` library with its origin with C++. There are other ways of doing output to the console, but `printf` and `<iostream>` and the official language standard techniques. A new library is introduced with C++ 20, `<format>`, that introduces a robust formatting sequence, is type-safe, and is extensible with user-defined types and other *goodies*.

`printf` has long been popular due to its compact formatting syntax, so why not use it? The main reason, it isn't type safe, has undefined conversion behavior which leads to well-known security vulnerabilities (e.g., uncontrolled format string). Furthermore, it doesn't support user-defined data types. Here is a blog post about security vulnerabilities associated `printf`.

...I've long since lost the post, unfortunately…

Here is a link to an article that describes a `printf` format string attack…

> https://owasp.org/www-community/attacks/Format_string_attack

`<iostream>` is a robust replacement for `printf`, but its syntax is quite verbose, leading developers to bemoan its use and dangerously revert back to using `printf`.

Some years ago, Victor Zverovich created a new C++ library that solves the problems with both `printf` and `<iostream>`, it is called fmt(https://github.com/fmtlib/fmt). This library was proposed for C++ and after some changes was accepted into the C++ 20 standard. As of the time of writing this document, only the MSVC C++ compiler/library has support for it, neither the gcc or llvm compiler/libraries have it included.

Basic Features

- Simple format syntax; in fact, based on Python's `format`
- Support for positional arguments
- Support for localization
- Compile-time validation of of formatting use
- Extensibility with support for user-defined types
- Better performance than both `printf` and `<iostream>`
- Smaller code size compared to using `printf` and `<iostream>`

Formatting a String

The `<format>` library formats strings, then those strings can be used however you want. If you want the output to go to the console, use `std::cout`, if you want the output to go to a file use `std::ofstream`, if you want the output to go to a database, use the database API, and so on.

The basic use of `std::format` looks like the following…

```
std::cout << std::format("{}, the value of Pi is {:6.4f}\n",
       "Hello World",
       3.1415926);
```

A look at what is being used above:

- The first parameter is the format string itself, the remaining parameters are values to be used by the formatting string.
- A replacement field begins with { and ends with }. In-between these characters are optional formatting specifiers, including an argument id if you want to identify which argument to use for replacement, otherwise they go in increasing order. The formatting specification is found at this link: https://en.cppreference.com/w/cpp/utility/format/formatter#Standard_format_specification
    - Within this documentation is a link to the Python format specification which is basically the same

Format Specifier : [fill][align][sign][#][0][width][.precision][type]

- **fill** : specify fill character
- **align** : < (left) > (right) ^ (center)
    - right justify within a field: `std::format("{:>12s}","right");`
    - center (finally!) justify within a field: `std::format("{:^12s}","center");`
- **sign** : -, +, space (show – for negative numbers, space for positive)
    - Not the word ‘space’, an actual single space character
- **#** : enable alternate formatting rules (hex, binary, octal, etc)
- **width** : min field width
- **precision** : number of digits or characters of a string
- **type** : b (binary), d (decimal), o (octal), x (hex), and more

Misc Items

- Can set the localization to be the user-specified computer locale with the following
    - `std::locale::global(std::locale(""));`
    - `std::cout.imbue(std::locale(""));`
    - Then, in the format string use `L` after the `:` character to cause locale-specific formatting: `std::format("{:Ld}", 100000);`
    - This will result in numbers being delineated/grouped using commas/decimals based on the computer’s setting. Otherwise, no commas/decimals are used.
- Dynamically specify field width and precision using replacement fields (very nice): std::format(`"{:{}.{}f}", 3.14159, 6, 4);`

std::filesystem

Prior to C++ 17 there was no C++ language/library support for performing file system operations, such as creating a directory, listing the files in a directory, etc. Instead, a programming had to use the operating system API or a third party library that abstracted the OS specific operations. The Boost library provided `boost.filesystem` that performs all of these kinds of operations in a platform independent manner. As C++ 17 was being considered, `boost.filesystem` was proposed for including with the C++ standard library, and was accepted with only few changes. As of C++ 17 standard now specifies `std::filesystem` in the `<filesystem>` library.

Basic Features

- Information (status) about files can be queried. File manipulation (reading/writing) is still performed using the `<iostream>` library.
- Folders (directories) and files can be queried (which folders/files are in a specified folder), along with creating or removing folders.
- The concept of a path is abstracted, along with operations for concatenating pathnames, getting filename extensions, getting the parent of a path, and more.
- Additionally, support for reading/changing permissions along with support for creating or removing symbolic or hard links exists, among many other capabilities.

Pathname

A `std::filesystem::path` represents a file system path, a folder or a file (or symbolic link). A path is composed of the following items:

- (optional) ***root-name*** : Depending on the operating system and/or file system the root-name could be something like "`C:`", "`//server`", "`//mnt`" (not including the quotes).
- (optional) ***root-directory*** : The folder name for the path. The root-directory can be absolute (starting from a root-name) or relative (not starting from a root-name and possibly other root-directories). If the path is relative, it might require another path to correctly resolve.
- It might include zero or more of the following:
  - ***file-name*** : The name of a file, which might have a filename extension.
  - ***directory-separators*** : A forward slash character `/` (or whatever is specified in `path::preferred_separator`).
  - ***dot*** : A filename of `.` refers to the current folder.
  - ***dot-dot*** : a filename of `..` refers to the parent folder.

To concatenate two paths together, use the `/` or `/=` operators. The `+` and `+=` operators also concatenate, but do not place a folder separator character when doing so. You usually want to use the `/` or `/=` operators. Regardless of the operating system or file system, a path correctly handles the directory separator character, forward or backward slash. As a programmer, you do not need to worry about this, just use the library correctly and it with *just work*.

Various operations are possible on a path, some of these include:

- Obtaining the `root_name()`, `root_directory()`, `root_path()`, `relative_path()`, and `parent_path()`.

- Extract the `filename()`, `stem()`, and `extension()`.
- Can iterate over the path elements using `begin()` and `end()` iterator access.

Current Folder

When a process (program) runs, it has the concept of the *current folder*. At startup, unless OS specific settings are applied, the current folder is the same folder as where the executable was run from. Note, where the executable was run from, might be different from the location it exists. This can happen when running from the command line.

The current folder can be obtained with the following function: `std::filesystem::current_path()`

The current folder can be changed by passing a new `path` into this same function, like the following…

```
auto parent = std::filesystem::current_path().parent_path();
std::filesystem::current_path(parent);
```

Enumerating Folders

It is possible to enumerate the items in a folder using a `directory_iterator.` To create a directory iterator, pass in a `directory_entry` (which can be a `path` object) and it will construct the iterator. With the iterator, any kind of iterator can be performed, using a for loop, for each loop, or even views (C++ 20). The following are two examples of using this iterator…

```
auto entries = std::filesystem::directory_iterator(std::filesystem::current_path());
for (auto&& entry : entries)
{
    std::cout << entry.path() << std::endl;
}

auto entries = std::filesystem::directory_iterator(std::filesystem::current_path());
for (auto entry = std::filesystem::begin(entries); entry != std::filesystem::end(entries); ++entry)
{
    std::cout << (*entry).path() << std::endl;
}
```

The display of these paths includes the full pathname of the entry, whether it is a folder or file. To report the folders/files as just the name in the folder, the various path manipulation methods would be used, such as `.filename()`, `.stem()`, and `.extension()`. To determine if it is a folder or file, methods like `.is_directory()` and `.is_regular_file()` are used.

Parallel Standard Library

References:

- https://www.modernescpp.com/index.php/parallel-algorithm-of-the-standard-template-library
- http://www.modernescpp.com/index.php/parallel-algorithms-of-the-stl-with-gcc
- https://stackoverflow.com/questions/51031060/are-c17-parallel-algorithms-implemented-already

The C++ standard library provides parallel execution options for a large number of algorithms. This was added as part of the C++ 17 standard. To use these algorithms is quite simple, it is a matter is specifying an execution policy as the first parameter to the supported algorithms. The algorithms are found in the `<algorithm>` library and the execution policies are found in the `<execution>` library.

Execution Policies

There are four execution policies defined for use; three from C++ 17 and one added in C++ 20:

- `std::execution::seq` (`sequenced_policy`) : Execution of the algorithm occurs sequentially over the range of elements.
- `std::execution::par` (`parallel_policy`) : Execution of the algorithm occurs in parallel over the range of elements.
- `std::execution::par_unseq` (`parallel_unsequenced_policy`) : Execution of the algorithm occurs in parallel and using vectorized (SIMD) operations, if possible.
- `std::execution::unseq` (`unsequenced_policy`) : Execution of the algorithm may occur in any order over the range of elements.

An important note about parallel operations: The algorithms do not protect against race conditions or deadlocks that may occur as a result of elements in the range being modified during the execution of the algorithm. It is the responsibility of the programmer to ensure no race conditions or deadlocks can occur during the execution of the algorithm.

Sorting Algorithm

An easy way to illustrate the use of parallel algorithms is by using the `std::sort` algorithm. Consider the following code…

```
auto sequential1 = generateData();
auto sequential2 = generateData();
auto parallel1 = generateData();
auto parallel2 = generateData();

std::sort(sequential1.begin(), sequential1.end());
std::sort(std::execution::seq, sequential2.begin(), sequential2.end());
std::sort(std::execution::par, parallel1.begin(), parallel1.end());
std::sort(std::execution::par_unseq, parallel2.begin(), parallel2.end());
```

The variables `sequential1`, `sequential2`, `parallel1`, and `parallel2` are `std::vector` of integers, created by the `generateData()` function.

The first call to `std::sort` uses the algorithm as it has been prior to C++ 17, a single threaded sorting algorithm. The second call passes in `std::execution::seq` as the first parameter, indicating to use the sequential (single threaded) sorting algorithm, which should be the same as the first call. The third call passes in `std::execution::par` as the first

parameter, indicating to use the parallel sorting algorithm. The fourth call passes in `std::execution::par_unseq` as the first parameter, indicating to use the parallel sorting algorithm and vectorized operations (if possible).

When timing the execution of these four options on my computer, the first two sorts take the same amount of time, as expected. The third sort is much faster (more than 5 times faster on an 4-core hyper-threaded CPU), and the fourth sort is incrementally faster than the third. These timing results follow what is expected. Other algorithms may have different results, due to the nature of the algorithm itself, but in general, the parallel execution is expected to be faster than the sequential execution.