COMP-3400-30-R-2026W C++, Week 1

Post-Lecture Slides

Mr. Paul Preney

School of Computer Science, University of Windsor, Windsor, Ontario

2026-01-06

Outline

1. Overview
   - Purpose of These Slides

Purpose of These Slides

The purpose of these slides is to follow up on the live code done in the first class.

Purpose of These Slides (con't)

The presented code attempts to compute π in two ways:

1. "first form" is numerically unstable
2. "second form" is numerically stable

In general, one wants to compute using numerically stable routines.

Purpose of These Slides (con't)

Additional background concerning these forms can be found at this Wikipedia URL:

- `https://en.wikipedia.org/wiki/Floating-point_arithmetic%23Minimizing_the_effect_of_accuracy_problems`

Outline

2 p1.cpp
    Code as Presented

p1.cpp: Code as Presented

This program is the program as presented live in class.

The issue with this program was that the output did not output the value of π.

The purpose of these slides is to correct such and to also explain some additional items concerning enhancements to the code.

Outline

3 p2.cpp
- Overview
- Program Output
- Use of C++ Coroutines

p2.cpp: Overview

This program fixes `p1.cpp` as follows:

- Changed `double value = sqrt(3.0);` to be `double value = 1.0 / sqrt(3.0);`.
- `form_to_pi()` is called (see `std::println()` call at the end of `main()`)

p2.cpp: Program Output

Notice that the first column's π values increase in accuracy and then as the steps continue the accuracy of π decreases until all significance is lost.

The second column's π output increases in accuracy until it no longer can increase.

p2.cpp: Program Output (con't)

Notice `std::cout` use was replaced in `p2.cpp` by a call to `std::println()`.

The format specification string used with `std::println()`, `std::print()`, `std::format()`, `std::format_to()`, `std::format_to_n()`, `std::vformat()`, and `std::vformat_to()` were all based on the format specification in Python.

- `https://en.cppreference.com/w/cpp/utility/format/spec.html`

Except for the `std::vformat*()` calls, the format string is evaluated at compile-time and must be a compile-time constant.

p2.cpp: Use of C++ Coroutines

- C++ coroutines are (special) functions that can be suspended and resumed.
- From outside the coroutine, one cannot distinguish a call to a coroutine from a call to a normal function.
  - What makes a C++ coroutine a coroutine is the use of any one (or more) of **`co_await`**, **`co_yield`**, or **`co_return`** in the coroutine function body.
  - NOTE: C++ coroutines cannot use **`return`**.

p2.cpp: Use of C++ Coroutines (con't)

- C++ coroutines are "stackless".
- When a coroutine suspends, the execution returns back to the caller (regardless of the call-depth of the suspension in the coroutine).
- While coroutines when actually running will appear on the function call stack they do not live on the function call stack. Realize that coroutines implicitly involve dynamic memory allocation use (in most cases).

p2.cpp: Use of C++ Coroutines (con't)

- In most cases, C++ coroutines have a special return type that enables (usually implicitly) controlling coroutine suspension and resumption.
  - This return type is (internally) tied to the coroutine itself.
  - In this example `std::generator<T>` is a class that models as an **input range** whose iterators trigger resuming the coroutine to obtain the next value obtained from the next call to **`co_yield`**.
    - If nothing is iterated over then the coroutine is never called.
    - If **`co_return`** is called or if a coroutine function ends (normally or via an exception) then the coroutine ends, e.g., iteration over the `std::generator<T>` range ends.
    - Should a coroutine end via an exception `std::generator<T>`'s iterator captures such and rethrows it with the caller.

p2.cpp: Use of C++ Coroutines (con't)

- C++ coroutines cannot:
  - use variadic arguments, i.e., function argument ellipsis, `...` , from C
    - NOTE: C++ parameter packs ellipses are okay since they are not variadic function arguments. Variadic function arguments are an unknown number of arguments at run-time. (C++ parameter pack expansions are always known at compile-time.)
  - use **`return`** statements
  - have placeholder, e.g., **`auto`** (or a concept), return types
  - be **`constexpr`** or **`consteval`**
  - be constructors, destructors, or `main()`
- So use coroutines for executing things at run-time only. If compile-time execution of a coroutine is needed then consider implementing the equivalent as an iterator class using **`constexpr`** constructors, functions, etc. to achieve compile-time execution.

Outline

4 p3.cpp
- Overview
- Program Output
- Use of C++ Coroutines
- Iterators

p3.cpp: Overview

This program changes `p2.cpp` as follows:

- A header file, `p3-ansi-terminal.hpp`, has been written to hold ANSI terminal code used to output colour.
- The `p3.cpp` program outputs the three columns of numbers with the first colour in bright blue.
- The correct digits of π are obtained from the `<numbers>` header (which defines π and other constants for built-in floating-point types).

p3.cpp: Program Output

Notice that the first column's π values increase in accuracy and then as the steps continue the accuracy of π decreases until all significance is lost.

The second column's π output increases in accuracy until it no longer can increase.

p3.cpp: Program Output (con't)

Notice `std::cout` use was replaced in `p2.cpp` by a call to `std::println()`.

The format specification string used with `std::println()`, `std::print()`, `std::format()`, `std::format_to()`, `std::format_to_n()`, `std::vformat()`, and `std::vformat_to()` were all based on the format specification in Python.

- `https://en.cppreference.com/w/cpp/utility/format/spec.html`

Except for `std::vformat*()` calls, the format string is evaluated at compile-time and must be a compile-time constant. (This program only uses `std::println()`.)

p3.cpp: Use of C++ Coroutines

- C++ coroutines are (special) functions that can be suspended and resumed.
- From outside the coroutine, one cannot distinguish a call to a coroutine from a call to a normal function.
  - What makes a C++ coroutine a coroutine is the use of any one (or more) of **`co_await`**, **`co_yield`**, or **`co_return`** in the coroutine function body.
  - NOTE: C++ coroutines cannot use **`return`**.

p3.cpp: Use of C++ Coroutines (con't)

- C++ coroutines are "stackless".
- When a coroutine suspends, the execution returns back to the caller (regardless of the call-depth of the suspension in the coroutine).
- While coroutines when actually running will appear on the function call stack they do not live on the function call stack. Realize that coroutines implicitly involve dynamic memory allocation use (in most cases).

p3.cpp: Use of C++ Coroutines (con't)

- In most cases, C++ coroutines have a special return type that enables (usually implicitly) controlling coroutine suspension and resumption.
  - This return type is (internally) tied to the coroutine itself.
  - In this example `std::generator<T>` is a class that models as an **input range** whose iterators trigger resuming the coroutine to obtain the next value obtained from the next call to **`co_yield`**.
    - If nothing is iterated over then the coroutine is never called.
    - If **`co_return`** is called or if a coroutine function ends (normally or via an exception) then the coroutine ends, e.g., iteration over the `std::generator<T>` range ends.
    - Should a coroutine end via an exception `std::generator<T>`'s iterator captures such and rethrows it with the caller.

p3.cpp: Use of C++ Coroutines (con't)

- C++ coroutines cannot:
  - use variadic arguments, i.e., function argument ellipsis, `...` , from C
    - NOTE: C++ parameter packs ellipses are okay since they are not variadic function arguments. Variadic function arguments are an unknown number of arguments at run-time. (C++ parameter pack expansions are always known at compile-time.)
  - use **`return`** statements
  - have placeholder, e.g., **`auto`** (or a concept), return types
  - be **`constexpr`** or **`consteval`**
  - be constructors, destructors, or `main()`
- So use coroutines for executing things at run-time only. If compile-time execution of a coroutine is needed then consider implementing the equivalent as an iterator class using **`constexpr`** constructors, functions, etc. to achieve compile-time execution.

p3.cpp: Iterators

The `std::generator<T>` interface is that of a container object. Its `begin()` and `end()` member functions allow one to obtain iterators to the “container”.

p3.cpp: Iterators (con't)

C++'s iterators are based on pointer syntax:

- A **`nullptr`**-valued pointer is a pointer with an invalid location. This is very similar to the "end" iterator.
  - A **`nullptr`**-valued pointer should never be indirected to access a value, e.g., **`operator`**`*` and **`operator`**`->` should never be used. The same is true for iterators.
  - A **`nullptr`**-valued pointer iterator can be checked if it is null or not using **`operator`**`==` or **`operator`**`!=`. With iterators the "null" value is the position of the end iterator itself. (Do not check an iterator against the value of **`nullptr`**.)

p3.cpp: Iterators (con't)

- Like pointers, using `operator++` goes to the next position.
- Like pointers, using `operator--` goes to the previous position.
- Similarly like pointers, one can use `operator+=`, `operator-=`, `operator+`, and `operator-` with iterators to change the position.
- Like pointers to access what is pointed to, one can use `operator*`, `operator->`, and `operator[]`.

p3.cpp: Iterators (con't)

When processing values in a container:

- one accesses elements inside the container via iterators
- the "first" iterator position is returned by the `begin()` member function
- the one-past-the-last element position is returned by the `end()` member function
- i.e., iterators represent the half-open interval [`begin()`,`end()`)
  - Although sometimes not mentioned by others, this is exactly the design of pointers in C and C++.
  - If you were not aware of this, you will be getting used to it in C++ as iterator use requires exploiting this.

p3.cpp: Iterators (con't)

- To process all elements in a container, one needs to loop over all iterator elements, e.g.,

  -
    ```
    auto it{ mycontainer.begin() };
    auto it_end{ mycontainer.end() };
    for (; it != it_end; ++it)
      std::cout << *it << '\n'; // e.g., output what *it points to
    ```

  -
    ```
    for (auto& elem : mycontainer)
      std::cout << elem << '\n'; // e.g., output each element value
    ```

p3.cpp: Iterators (con't)

There are six (6) kinds of iterators in C++:

- input iterators
  - supports pre- and post-increment and accessing values
  - remembering positions (reachability) is not supported
  - e.g., streams and `std::generator<T>`
- output iterators
  - supports pre- and post-increment and writing values
- forward iterators
  - all forward iterators meet the requirements of input and output iterators
  - e.g., ++ (increment operators)

p3.cpp: Iterators (con't)

- bidirectional iterators
  - all bidirectional iterators meet the requirements of forward iterators
  - e.g., `--` (decrement operators)
- random-access iterators
  - all random-access iterators meet the requirements of bidirectional iterators
  - e.g., `+=`, `-=`, `+`, `-`, and `[]`
  - random-access iterator required operations all have O(1) time complexity
- contiguous iterators
  - all contiguous iterators meet the requirements of random-access iterators
  - all contiguous iterators guarantee that the elements in the container are stored adjacently in memory
  - e.g., elements in an array

p3.cpp: Iterators (con't)

So this is why the program has code like this:

```
auto f1 = first_form();
auto f1it = f1.begin();
```

i.e.,

- `f1` holds a `std::generator<double>` instance returned from `first_form()`
- Using `f1`, calling `f1.begin()` yields the iterator to the first value.
- Internal to the iterator, the coroutine is appropriately suspended and resumed.
- Accessing the iterator value, e.g., `*f1it`, yields the last **`co_yield`**ed value.
- Incrementing the iterator, e.g., `++f1it`, causes the (cached) last **`co_yield`**ed value to be invalidated/replaced with the next **`co_yield`**ed value (if any) from the coroutine, i.e., it resumes coroutine execution.

Outline

5 p4.cpp
Overview

p4.cpp: Overview

This program changes `p3.cpp` as follows:

- Instead of `p3-ansi-terminal.hpp` the same has been called `p4-ansi-terminal.hpp`.
- `p4.cpp` adds the following functions:
  - `pi_coloured_diff_string()`
  - `coloured_string()`
- `p4.cpp` changes `main()` as follows:
  - Three (3) columns of formatted output.
  - The first column colour is in bright blue and is the number of steps taken.
  - The second and third column colours are a mix of bright yello and red.
    - Bright yellow represents digits of π that are correct.
    - Red represents digits of π that are incorrect.
    - Notice the second column initially increases in accuracy and then loses all accuracy as the number of steps increases.
    - Notice the third column increases in accuracy as the number of steps increases.

Outline

6 p4.cpp
    Overview

p4.cpp: Overview

This program changes `p4.cpp` as follows:

- Instead of `p4-ansi-terminal.hpp` the same has been called `p5-ansi-terminal.hpp`.
  - While the operations ultimately perform the same, there are significant changes to the way the header file outputs colour.
  - `reset()` is defined to output the ANSI reset code to an output stream.
  - `set` is a (helper) class whose constructor accepts a colour that will later be written to an output stream.
    - See the **`friend`** function defined inside the class.
    - A **`friend`** to a class has access to the class' **`private`** and **`protected`** members.
    - Notice the use of the unary $+$ in the **`friend`** function. Unary $+$ applied to a built-in char will promote its value to be that of an integer (which is what is wanted) otherwise when outputting the value the char would appear as a character (which wouldn't be correct in this case).
    - Class enumerations in C++ need to be cast to their underlying values in order to access those values. (This is why **`static_cast`** is used.)

p4.cpp: Overview (con't)

- `reset_string()` is defined to call the `reset()` function to write the reset code to an output string stream which is then converted into a string.
- `set_string()` converts a colour to a string by using `set` and then that output string stream is converted to a `std::string`.

p4.cpp: Overview (con't)

The changes to the header allowed the following:

- ANSI reset and ANSI colours can be directly output to any output stream or string.
- The numbers are output first to strings to allow them to be compared and as they are compared colours are added to the final output stream.
- Instead of directly writing to the final output stream, output is generated in output string streams to allow conversion to a string and later that is written out to the final output stream.
- `std::ostream_iterator<char>(buf)` is an anonymously constructed output stream iterator associated with the stream `buf`. This works nicely since `std::format_to()` requires an output iterator to be passed to it.