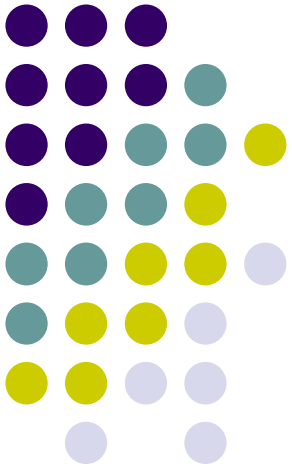
Advanced C++

April 15, 2021

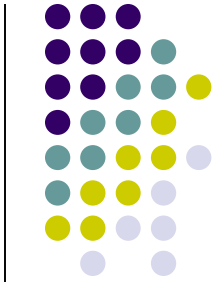
Mike Spertus

mike@spertus.com

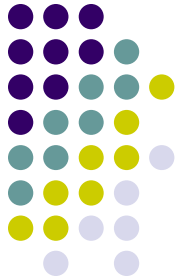


MASTERS PROGRAM IN COMPUTER SCIENCE

THE UNIVERSITY OF CHICAGO



STREAMS, RANGES, AND ITERATORS



IOSTREAMS

IO Streams are also templates

- ostream is for 8-bit characters
- That misses a lot of the world
- In reality, ostream is a typedef for basic_ostream, which is templatized by the character type
  - using ostream=basic_ostream<char>;
    // Use wostream for wide characters
    using wostream=basic_ostream<wchar_t>;
- basic_ostream has another template parameter giving more info about the character set, but this can usually be ignored because it has a good default
  - template<class CharT,
    class Traits=char_traits<charT>>
    class basic_ostream;

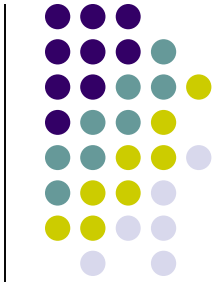
Sidebar: How wide is a wchar_t?

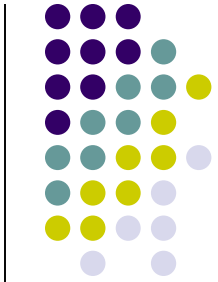
- 32 bits on Linux and 16 bits on Windows
- Ugh!
- It took a while to realize that 16 bits aren't enough
- But backwards-compatibility prevents fixing
- This means that a single wchar_t might not hold a character
- Better to avoid `wchar_t` and stick with `char`, `utf8_t`, or `utf32_t`
  - Even though Unicode support is still lacking…

I/O Manipulators

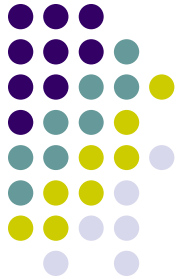
- Recall that `endl` not only inserts a `'\n'` in an `ostream`, it also flushes it.
- An object that actively manipulates a stream when inserted (or extracted) like `endl` is called an I/O manipulator.
- Some iomanipulators
  - `std::endl` of course
  - `std::hex` for hexadecimal i/o
  - `std::setw` to set a fixed width for the next insertion
  - `std::setfill` allows you to set the fill characters if the insertion is smaller than the width

Defining `endl`



- C++ makes it easy to create an I/O manipulator.
  - Simply define it as a function that takes and returns a stream
- If we only care about `ostream` (i.e., ordinary characters), it could be defined as follows
- ```
  ostream &
  endl(ostream &os)
  {
      os << '\n';
      os.flush();
      return os;
  }
  ```
  ```

How endl gets called



- The standard library defines an overload of operator<< for inserting a function in an ostream that calls the function with the ostream as argument

```
ostream &
operator<<
 (ostream&os,
  ostream&(*manip)(ostream &))
{
    return manip(os);
}
```

More advanced

- Really, `endl` should work for any output stream, regardless of character type.
- Recall from a few slides ago, that the stream class are really templates.
- Here's how `endl` is really defined.
- ```
  template<typename charT, typename Traits>
  basic_ostream<charT, Traits> &
  endl(basic_ostream<charT, Traits> &os)
  {
      os << '\n';
      os.flush();
      return os;
  }
  ```
  ```

I/O manipulators with arguments

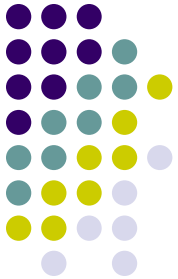


```
struct setw {
  setw(size_t s) : width(s) {}
  size_t width;
}

template<typename char_t, typename traits>
basic_ostream<char_t, traits> &
operator<<(basic_ostream<char_t, traits> os,
                    setw sw) { … }
```

Stream buffers

- Stream classes do formatted I/O (i.e., they let you insert and extract arbitrary types from a character stream).
- Once it is time to actually output characters to (or input from) a device, a stream buffer is used to perform the I/O.
- Every stream has a streambuf member that can be set using `basic_ios::rdbuf`. Recall that `basic_ios` is the base class for all streams.

Stream buffers



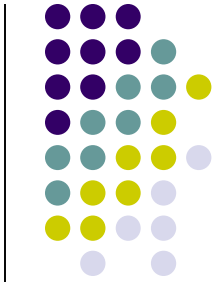
- Let's look at http://www.angelikalanger.com/IOStreams/Excerpt/excerpt.htm to go deeper into stream buffers

Customizing streams

- Basically, different types of streams are created by customizing a streambuf type
  - Stream types have no virtual functions
  - While stream buffers do
- However, we need to create a new stream type just to attach our stream buffer in the constructor. For example, the implementation of file streams is (logically)

```
struct ofstream : public ostream {
  ofstream() : ostream(mybuf) {…}
  … // Other constructors and methods
private:
  filebuf mybuf;
};
```

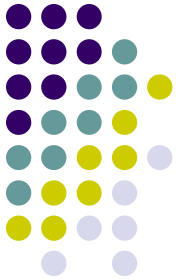
Now with templates



- Of course, since ostream is really a typedef for basic_ostream<charT, traits>, the above code could really be:

```
template
  <class charT,
   class traits=char_traits<charT> >
struct basic_ofstream
  : public basic_ostream<charT, traits> {
  basic_ofstream() : basic_ostream(mybuf) {…}
  // Other constructors and methods
private:
  filebuf mybuf;
};
```

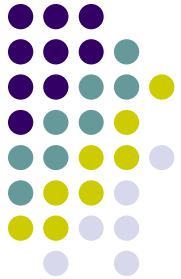
Customizing stream buffers



- A stream buffer can have an underlying memory buffer that can buffer characters
  - Set it with _Init() method
  - Calling _Init() with no arguments leads to unbuffered I/O
- To customize a stream buffer, we override the overflow (for output) or underflow (for input) methods
  - These are called when the buffer (if it exists) cannot buffer any more characters

Storing state in streams

- Inserters/IO manipulators may need this
  - For example, std::hex has to store somewhere on a per-stream basis that the radix (base) of a stream is 16
  - Likewise, the inserters for numeric types will need to check the radix of the stream to print them
  - std::ios_base::xalloc() can allocate a slot in each stream for you to store your data
- New stream types may need this
  - E.g., fstreams need to store a file handle
  - stringstreams need to store a string
  - This can be done either with (1) xalloc, or (2) by adding a field to the stream, or (3) adding a field to the streambuf
    - Which do you think is best?



ITERATORS

Iterators

- Iterators are the C++ lightweight abstractions generalizing C pointer arithmetic
  - In fact, C pointers are iterators, but avoid them because it is easy to overrun a buffer
    - This is what caused HeartBleed! http://nakedsecurity.sophos.com/2014/04/08/anatomy-of-a-data-leak-bug-openssl-heartbleed/
  - Also, what if you are iterating a list instead of an array?
- All STL containers can produce iterators that let you run through their elements without running off the end

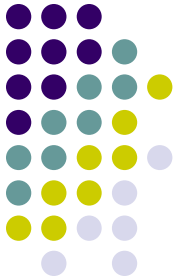
Before we understand iterators, let's look at pointers

- A pointer stores the address of an object
- `A *` is type "pointer to `A`"
  - `A *ap; // Note ap is not initialized. Don't use yet!`
- `new` expressions return a pointer to the newly created object
  - `A *ap2 = new A();`
- & takes the address of an object
  - ```
    A a;
    ap = &a; // & takes address of obj. Now we can use ap
    ```
- `->` is short for `(* ).`
  - ```
    ap2->x = 3; // Sets obj's x member to 3. (Assume A has 'int x' member)
    (*ap2).x = 3; // Same as above line
    ```
- `*` "dereferences" a pointer, returning a reference to the object it is pointing to
  - ```
    // a gets copy of obj pointed to by ap2
    a = *ap2;
    a.x = 5; // Doesn't modify ap2->x
    // ar is reference to obj pointed to by ap2
    A &ar = *ap2;
    ar.x = 7; // modifies obj pointed to by ap2
    ```
    ```

Pointers into arrays

- If a pointer points to an element of an array, then you can also use it to access other elements of the array via “pointer arithmetic”
  - ```
    A *arp = new A[10]; // arp is addr of 0th elt
    A *arp2 = arp+2; // arp2 is addr of 2nd elt
    arp[3] = 2; // Short for *(arp + 3) = 2;
    A *arp3 = &(*arp)[3]; // arp3 is addr of 3rd elt
    A &arr3 = (*arp)[3] // arr3 is ref to 3rd elt
    A as[10]; // Array of 10 A objects
    A *asp = as; // Name of array is addr of 0th elt
    A *asp2 = as+2; // asp2 points 2 objects past as
    asp2 = &as[2]; // Does the same thing
    as++;  // as now points to the 1st elt
    ```
    ```

Understanding iterators

- Iterators are an abstraction of “pointer to C-style array element” for any container or sequence, not just C-style arrays
- Based on Section 6.3 of Josuttis’ *The C++ Standard Library, 2nd edition*
- A type behaves as an iterator by supporting the following operators
  - `operator *` gives the element currently being iterator
  - `operator++` causes the iterator to advance to the next element
  - `operator==` and `operator!=` to compare iterators
  - `operator=` to assign iterators
  - Some iterators define additional operators like `operator--`

Iterator categories



- Iterators come in various flavors
- Forward Iterators
  - Can advance these but cannot decrement them. `std::forward_list<T>::iterator` is an example of forward iterators
- Bidirectional iterators
  - Can increment or decrement. E.g., `std::list<T>::iterator.`
- Random access iterators
  - Can add or subtract and integer to advance or retreat by a specific amount. E.g., pointer arithmetic
- Input iterators
  - Can get values from them but not assign to them. E.g., istream iterators
- Output iterators
  - Can assign to them but not get values from them. E.g., `ostream_iterator.`

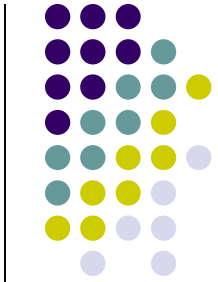
Knowing what kind of iterator you have matters

- If `vec` is a `std::vector<int>`, then you can sort it with
  - `std::sort(vec.begin(), vec.end());`
- If `lst` is a `std::list<int>`, you get a long horribly confusing error message if you try to sort it with
  - `std::sort(lst.begin(), lst.end());`
- The problem is that `std::sort()` expects random-access iterators and linked lists only have bidirectional iterators

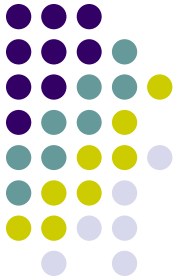
How can you find out about your iterator?

- A lot of times (especially when in a template), you may have an object that you know is an iterator
- But you might want to know more about its iterator behavior
  - What type of object does it iterate?
  - Is it a random access iterator?
  - ...
- Let's try to imagine how we would solve this as class designers

We could store this "metadata" in the iterator class

- Iterators can have "member types" with the info
- Here is a simplified view of vector<int>::iterator
- ```
  template<typename T>
  class vector {
  public:
    struct iterator {
      using value_type        = T;
      using iterator_category = random_access_iterator_tag;
      using difference_type   = ptrdiff_t;
      using pointer           = T *;
      using reference         = T &;
      /* ... */
    };
    /* ... */
  };
  ```
  ```

What is `random_access_iterator_tag`?

- It's just an empty "indicator" class with no members used to tag what kind of iterator we have

  ```
  struct random_access_iterator_tag {};
  ```

- Our `TT` class also used this popular idiom
- At first glance, you might have thought it should just be an `enum` value indicating what kind of iterator we have
  - Let's see why the class approach works better

Bad way to tell if an iterator is random access



- This naïve approach sort of works
-
```
If constexpr (is_same_v<vector<int>::iterator_category,
                        random_access_iterator_tag>)
  cout << ”Random access :)\n”;
else
  cout << ”Not random access :(\n”;
```

Problem 1: Improperly rejects more specialized iterators

- Suppose we needed a forward iterator
- We would be happy with a random access iterator because they satisfy all the requirements for a forward iterator
- But testing if the `iterator_category` is `forward_iterator_tag` would reject it

Solution via inheritance

- To reflect that random access iterators are more specialized than forward iterators, it seems natural to use inheritance
- Let's change our previous definition a little
- `struct random_access_iterator_tag`
  `: public forward_iterator_tag {};`
- This is why we didn't use an `enum`

Iterator tags

- Here are the ones defined by the standard library
- Like we've seen before, these classes have empty bodies because we are really just looking at their type name
- Note that use of inheritance to reflect "isA" relationship
- ```
  struct input_iterator_tag {};
  struct output_iterator_tag {};
  struct forward_iterator_tag
    : public input_iterator_tag {};
  struct bidirectional_iterator_tag
    : public forward_iterator_tag {};
  struct random_access_iterator_tag
    : public bidirectional_iterator_tag {};
  struct contiguous_iterator_tag
    : public random_access_iterator_tag {};
  ```
  ```

Now our test works (sort of)



```
// Don't do this. We'll provide better versions later
template<typename T>
auto
I_need_a_forward_iterator(T t,
  forward_iterator_tag = T::iterator_category())
{ /* ... */ }

vector v = {1, 2, 3};

// Now the following line is ok
auto x = I_need_a_forward_iterator(v.begin());
```

Problem 2: What if our iterator isn't a class?

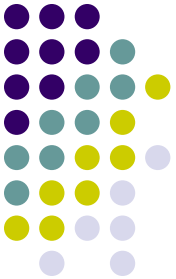
- Pointers should qualify as iterators
- Indeed, iterators can be thought of as "generalized pointers"
- But pointers aren't classes, so they can't have member type aliases
- `int *::iterator_category` is nonsense
- So we can't call `I_need_a_forward_iterator` with an `int *` ☹

Solution: "Every problem can be solved by an extra indirection"

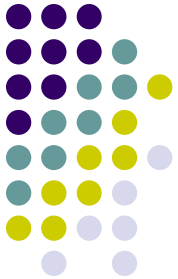
- Let's solve by "lifting" the iterator metadata into a different class called `iterator_traits`
- Now we define the iterator properties by specializing `iterator_traits`

```
namespace std {
   template <class T>
   struct iterator_traits<T*> {
     using value_type        = T;
     using difference_type   = ptrdiff_t;
     using iterator_category = contiguous_iterator_tag;
     using pointer = T *;
     using reference = T &;
  };
}
```

Putting member type aliases in our classes still works

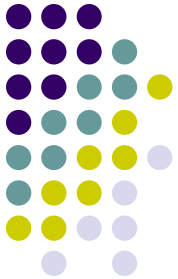
- The iterator_traits primary template looks for member types in the iterator class
- ```
  template<typename T>
  struct iterator_traits {
    using iterator_category = typename T::iterator_category;
    using value_type        = typename T::value_type;
    using difference_type   = typename T::difference_type;
    using pointer           = typename T::pointer;
    using reference         = typename T::reference;
  };
  ```
- So it still finds the member traits in vector iterators
  ```

Fixing our test: C++17



```
// Better
template<typename T>
auto
I_need_a_forward_iterator(T t,
  forward_iterator_tag
    = typename iterator_traits<T>::iterator_category())
{ /* ... */ }

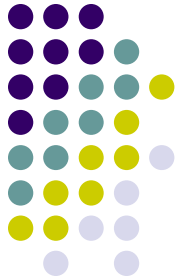
vector v = {1, 2, 3};

// Now the following line is ok
auto x = I_need_a_forward_iterator(v.begin());
```

Replacing tag types with concepts (C++20 only)

- C++20 has iterator concepts that are less "hacky" than using the tags
- E.g., `std::random_access_iterator`
- **Note (advanced)**
  - When implementing ranges (see later this lecture), we learned that our existing notions of iterators weren't quite right and took the opportunity to fix them
  - So saying a type satisfying the `std::random_access_iterator` concept is slightly different from saying its iterator_category is std::random_access_iterator_tag
  - The old notions are called *LegacyRandomAccessIterator*, etc.
  - These difference are highly obscure, and you shouldn't worry too much about them, but no need to freak out if someone mentions legacy iterators

Fixing our test: C++20



```
// Best
template<forward_iterator T>
auto
I_need_a_forward_iterator(T t)
{ /* ... */ }


// Equally good
auto
I_need_a_forward_iterator(forward_iterator auto t)
{ /* ... */ }


vector v = {1, 2, 3};


// Now the following line is ok
auto x = I_need_a_forward_iterator(v.begin());
```

Let's look at another example

- IteratorOverload.cpp on Canvas

Iterators and CTAD



- vectors are commonly constructed from iterator pairs
- ```
  list li = { 1, 2, 3};
  vector vi(li.begin(), li.end());
  ```
- How does C++ know that vi is a vector<int>?
- A deduction guide!
- ```
  // Allocator omitted for simplicity
  template<class InputIt>
  vector(InputIt, InputIt)
    -> vector<typename iterator_traits<InputIt>::value_type>;
  ```

How do I make my own iterator

- Define all the member type aliases
- Define all the relevant operators
- ```
  template<typename T>
  struct vector<T>::iterator {
    // Member type aliases from above
    T &operator*();
    iterator operator+=(const difference_type &);
    // Many other operators -=, ++, --,...
  };
  // Symmetric operators
  template<typename T>
  bool operator==(vector<T>::iterator, vector<T>::iterator);
  template<typename T>
  bool operator!=(vector<T>::iterator, vector<T>::iterator);
  ```
  ```



RANGES

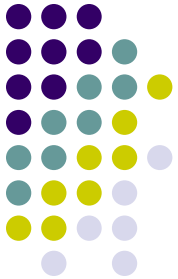
Introducing ranges

- Ranges are a work in process replacement for the C++ standard library algorithms that are
  - Composable so you can feed one algorithm into another
  - Work directly on ranges rather than iterator pairs for natural usage
  - Lazy for efficiency because you don’t have to create intermediate containers holding all of the elements from each step
  - Fully conceptized for improved safety and error messages
- **Note:**
  - C++20 ranges are too limited to be useful
  - C++23 ranges will be better but are not yet properly supported
  - We will often use the ranges v3 library that is the testbed

Problems with standard library algorithms

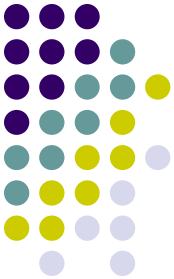
- Require iterator pairs
- Not composable
- Intermediate stages must be materialized
- Compute values that will be thrown away
- Must be finite
- Excessive copying
- Play badly with inference
- Ironically, C++ algorithms were a selling point of C++ in C++98
  - Shows how much languages have advanced

An example



- For example, suppose we want to take a vector<int>
  - Sort it
  - Filter out the even numbers
  - Multiply them by 10
  - Take the first 3 elements
  - Then print to the console
- https://godbolt.org/z/rKK169bMe

Annoying iterator pairs

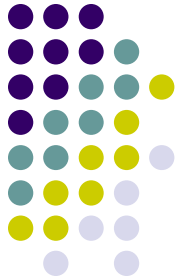


- We can't directly sort/transform/copy/etc. a container
- Instead, we always have to give it a pair of iterators
- For example, in the example, we have to say
  - `sort(vi.begin(), vi.end())`
  - `copy_if(vi.begin(), vi.end(),...)`
  - `transform(filtered.begin(), filtered.end(),...)`
- Sure, it's just an "annoyance," but one nearly every program runs into repeatedly

Standard library algorithms are not composable

- One often wants to use more than one algorithm
- Unfortunately, it is very hard to combine standard library algorithms
- Note that at each stage, we need to create a temporary container to hold the intermediate result
- Distracting from the logic of the algorithm
- Forcing us to think about what kind of intermediate container to use
- Forcing us to think about what kind of output iterator to use
  - Should we use a back_insert_iterator or resize an use begin()? Does it depend if the value type is default constructible? Are we more concerned about safety or speed?...

Each step in a computation must be materialized



- Since algorithms can't be composed, we had to create intermediate containers to hold the results
- Suppose there were a million elements in our vector
- Those intermediate containers will be very big and take up a lot of memory
- In spite of the fact that they have no independent value. They just hold intermediate steps
- In such a case, "windowing" several at a time would be much better

Computing values that will be thrown away

- This algorithm did a lot of multiplying of numbers
- That were just going to be thrown out in the last step
- Wouldn't it be great if values could be pulled on demand, so it knew it didn't need to compute any more?

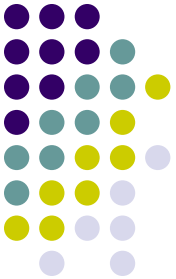
Only works for finite sequence

- Suppose we wanted to apply algorithms to infinite streams
- Unfortunately, since algorithms aren't composable, we can't start on the second step of the algorithm until the first is finished
- Oops

Excessive copying

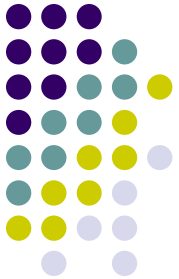
- While ints are quick to copy, not all types are
- Also, created the `filtered` vector will probably require a memory allocation, which will dwarf other costs
- Wouldn't it be better if `filtered` were just a "filtered view" on `vi`?

Plays badly with inference



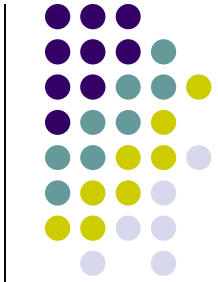
- While we could deduce that `vi` is a `vector<int>`
- `vector vi{9, 3, 2, 8, 5, 6, 7, 4,1, 10};`
- All of the intermediate values need to be manually fully specified
- `vector<int> filtered;`
- While this isn't terrible here, imagine more complicated transformations

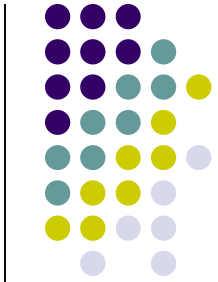
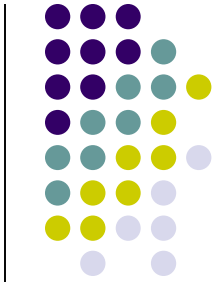
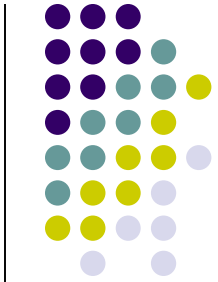
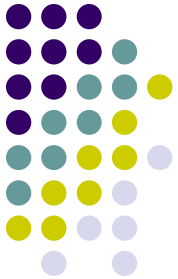
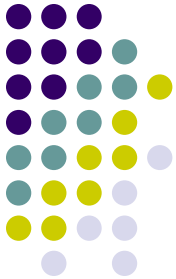
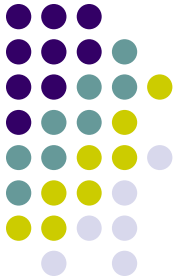
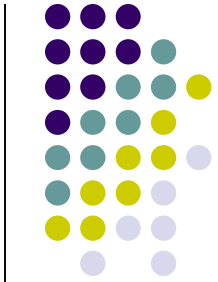
Same example with ranges



- https://godbolt.org/z/1f3G7MEhs
- Every one of the above aspects is better
- Let's look at it

iota



- `views::iota` is a "counting range"
- ```
  copy(iota(1, 5),
       ostream_iterator<int>(std::cout, "\n"));
  ```
- Prints 1 through 4
  - Remember C++ ranges include their beginning but not their end?
- https://godbolt.org/z/5z44ez5fG
  ```

Infinite ranges

- If you give iota a single argument, it counts from that argument forever
- This example shows how you don't need to finish the first algorithm before starting the next one
- https://godbolt.org/z/M4v8Kv5xK

Truncating infinite ranges and `views::take`

- Since values will be generated on demand, you can truncate an infinite range wherever you know how many you need
- Here's a program that generates the first 10 triangular numbers
- https://godbolt.org/z/3Wjh97jxv

`ranges::to` (C++23)



- Working with views rather than materializing everything can be great
- But at the end, you likely want your result in a container
- This is where `ranges::to` comes in
- To store the triangular numbers in a vector at the end
- Just pipe it into `ranges::to<vector>`
- https://godbolt.org/z/8Wzx3f77j

Ranges and inference

- Since I no longer need intermediate containers, I no longer need to declare them to be `vector<int>`
- For the output, I still need a vector, but `ranges::to<vector>` performs CTAD, so I don't need to say `ranges::to<vector<int>>`
  - Although that would also be legal

ranges::views::zip



- We can splice multiple ranges together like a zipper into a range of tuples
- By zipping an iota together with our triangular numbers, we can create a map for looking up the nth triangular number
- 1 2 3 4 5
  1 3 6 10 15
- https://godbolt.org/z/5EPzs5bf7

HW 16-1

- Improve our optimized copy from our previous version (on Canvas)
- Currently it only considers doing a memcpy if the arguments are actually pointers
- But a memcpy might make sense for other contiguous iterators beyond pointers
- You can use either a C++20 iterator concept or the iterator category

HW 16-2



- Redo HW 2-1 from last quarter with ranges
  - I've copied the slides here for reference, don't get confused by their saying HW 2.1.
- **Note:** You'll want to use `ranges::to`, which won't be available until C++23, so use Ranges v3 (https://ericniebler.github.io/range-v3/). Either with your own compiler or on Godbolt like https://godbolt.org/z/8Wzx3f77j

HW 2.1 (Part 1)



- Last week we mentioned std::copy. There is a related function in the <algorithm> header called std::transform, which lets you apply a function to each element before copying. Look up or Google for std::transform to understand the precise usage.
- Write a program that puts several doubles in a vector and then uses std::transform to produce a new vector with each element equal to the square of the corresponding element in the original vector and print the new vector (If you use ostream_iterator to print the new vector, you will likely get an extra comma at the end of the output. Don't worry if that happens).

HW 2.1 (Part 2)



- We will extend the program in part 1 to calculate and print the distance of a vector from the origin.
- There is also a function in the <numeric> header called `accumulate` that can be used to add up all the items in a collection.
  - (Googling for "accumulate numeric example" gives some good examples of using `accumulate`. We're interested in the 3 argument version of `accumulate`).
- After squaring each element of the vector as in part 1, use accumulate to add up the squares and then take the square root. (That this is the distance from the origin is the famous Pythagorean theorem, which works in any number of dimensions).

HW 2.1 (Part 3)



- In real-life, you'd probably use `std::inner_product` to find the Euclidean length of a vector
- Learn about `inner_product` and use it to find a better way to accomplish part 2

HW 2.1 (Extra credit part 4)

- As yet another way to calculate the Euclidean length of a vector, is also a four argument version of `accumulate` that can combine `transform` and `accumulate` in a single step. Use this to provide another solution to part 2 of this problem

HW 16-3: Extra Credit

- We could have done https://godbolt.org/z/3Wjh97jxv equally well with `views::iota(1, 6)` instead of `views::take`
  - https://godbolt.org/z/ohTYqcr6W
- Can you construct a problem where truncating an infinite range is genuinely better?
- **Hint:** A simple modification should give one solution

HW 16-4

- Create a new stream `IndentStream` and I/O manipulators `indent` and `unindent` with the following properties.
  - Each line output to an `IndentStream` is indented by a given amount.
  - Inserting `indent` or `unindent` into a stream increases or decreases the indent level by 4.

HW 16-4 (Continued)

- For example,

```
IndentStream ins(cout);
ins << "int" << endl;
ins << fib(int n) {" << indent << endl;
ins << "if (n == 0) return 0;" << endl;
ins << "if (n == 1) return 1;" << endl;
ins << "return fib(n-2) + fib(n-1);" << unindent << endl;
ins << "}" << endl;
```

will output

```
int
fib(int n) {
    if (n == 0) return 0;
    if (n == 1) return 1;
    return fib(n-2) + fib(n-1);
}
```

HW 16-4 (continued)



- As described in class, almost all the work will be done by creating an `IndentStreamBuf` class and redefining its `overflow` method. Just about all `IndentStream` will do is set its stream buffer to an appropriate `IndentStreamBuf` in the constructor.
  - You will not want to have an actual memory buffer in your `IndentStreamBuf`, so your `overflow` method will get called on each character
- Useful link
  - http://www.angelikalanger.com/IOStreams/Excerpt/excerpt.htm

HW 16-5: Extra Credit

- Create an `ostream_joiner` that acts just like `ostream_iterator` except that the delimiter only goes between elements (and not after the final element).
- This is better than `ostream_iterator` because you don't get a trailing comma
  - ```
    vector v = {1, 2, 3};
    copy(v, ostream_iterator<int>(cout, ", "); // 1, 2, 3,
    copy(v, ostream_joiner(cout, ", ");  // 1, 2, 3
    ```
- Use this to easily create an "`operator<<`" to print vectors in `ostreams`.
- For "regular extra credit", just do this with ostreams.
- For "extra extra credit"
  - Make this fully generic to work with any basic_ostream (e.g., wide characters). You may need to learn a little about C++17 Class Template Argument Deduction to fully handle the extra credit case
- Note: Since `ostream_joiner` is in the process of standardization, there are implementations available on the web. Please don't look at them
    ```

HW 16-5: Hints

- Hint: You'll need to define iterator traits for `ostream_joiner`
  - For output iterators, the `value_type`, `difference_type`, `pointer`, and `reference` can all be `void`
  - They just aren't really meaningful for output iterators
- Hint: The real work will be in `operator*`
  - You write to an output iterator `oi` by assigning to *oi
  - This means that you will have to give `*oi` a type that you have defined an `operator=` for
  - The `operator=` will do the actually work of embedding the delimiter and sending to the wrapped ostream