COMP-3400-30-R-2026W C++, Week 2

Overloading, Relations, Pointers, Iterators, and References

Mr. Paul Preney

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

Not all slides will necessarily be presented in the lecture. It is your responsibility to read and learn all slide content and related materials in the textbooks and in this course.

2026-01-13

Outline

Values and References: Overview

In the C programming language **all** declarations are values: even pointers are values.

- C **only** supports pass/return-by-value.
- In C pass/return-by-reference is **manually managed** through the use of **pointers** by **manually** using the **address-of** and **indirection** operators.
  - i.e., since in C each pointer **value** is pass/return-by-value and everything is **manually managed**, C only supports pass/return-by-value.

Values and References: Overview (con't)

In the Java programming language:

- value semantics are used for all built-in types
  - e.g, **`short`**, **`int`**, **`long`**, **`double`**, etc.
- reference semantics are used otherwise, e.g., classes

NOTE: C++ **always** uses value semantics unless the type involved is actually a reference.

Values and References: Overview (con't)

The C++ programming language supports:

- value and pointers, i.e., as found in C
- reference semantics:
    - lvalue references since C++98
    - rvalue references since C++11
    - with declarations, function argument passing, return values
    - with no ability to change what the reference refers to once set.

Values

In C++:

- unless `&` or `&&` appears in a type, the type is a value
  - The appearance of `&` or `&&` implies the type is a reference, so its absence implies a value.
  - Due to C++ grammar rules, `&` or `&&` will typically appear as the **rightmost** part of a type, e.g.,
    - `int&`
    - `int&&`
    - `std::string const&`
    - `std::list<int>&&`
    - Exception: `int(&)[100]`, i.e., reference to array
    - Exception: `int(&&)[100]`, i.e., reference to array
    - Exception: `int(&)(float)`, i.e., reference to function
    - Exception: `void(&&)(char)`, i.e., reference to function

Values (con't)

- There are **fundamental** and **compound** types.
- **Fundamental types** include `char`, `short`, `int`, `long`, etc. [1, [basic.fundamental]].
- **Compound types** include arrays, functions, pointers, references, classes, unions, enumerations, and pointers-to-members [1, [basic.compound]].
- Every fundamental and compound type in C++ can be decorated with `const` and/or `volatile`.
  - `const` renders an object's value read-only [1, [basic.type.qualifier]]
  - `volatile` renders an object's value as possibly asynchronously modified [1, [basic.type.qualifier]]
    - Except as mandated by the standard when using `std::signal()`, *do not use* `volatile` with/for concurrency. Instead use `std::atomic`.

Pointers

Pointers are **explicitly manipulated address** values and are used and behave as they do in C.

The **invalid memory address** is referred to as the **null pointer**.

Pointers (con't)

In C++98, null pointers were represented:

- by the number `0` (zero) cast to the pointer type
- C's `NULL` but know using such in C++ is strongly discouraged

There were issues with C++98's use of `0` to represent the null pointer with some overloading needs so C++11 fixed this by:

- defining a new keyword **`nullptr`** to represent the null pointer value
- defining a new type `std::nullptr_t` as the type of the value **`nullptr`**
- allowing specific null pointer values to be obtained by casting, often done implicitly, **`nullptr`** to the desired pointer type
  - e.g., `int *p = nullptr;`

Pointers (con't)

Recall from C:

Pointer Declaration	Interpretation
`T*`	read-write address referring to a read-write value of type T
`T const*`	read-write address referring to a read-only value of type T
`T * const`	read-only address referring to a read-write value of type T
`T const * const`	read-only address referring to a read-only value of type T

References

IMPORTANT!

References are not pointers!

References (con't)

The C++ standard states:

- "[...] Lvalue references and rvalue references are distinct types. [...]" [1, [dcl.ref]]
- "It is unspecified whether or not a reference requires storage" [1, [dcl.ref]]
  - This implies that the compiler **does not have to use pointers to implement references**.
  - Tip: Avoid explicitly finding/using a reference's referent address as this can have a significant impact on generated code.
- "There shall be no references to references, no arrays of references, and no pointers to references. [...] a null reference cannot exist in a well-defined program, because the only way to create such a reference would be to bind it to the 'object' obtained by dereferencing a null pointer, which causes undefined behavior. [...] a reference cannot be bound directly to a bit-field." [1, [dcl.ref]]

References (con't)

Internally a reference might be implemented as a **read-only pointer** to some value:

Pointer Declaration	Reference Declaration
`T*`	invalid —pointer not read-only
`T const*`	invalid —pointer not read-only
`T * const`	`T&`
`T const * const`	`T const&`

i.e., a reference can be thought of as being internally represented as **constant pointer** to the referent which implies:

- the referent **must be provided** when the reference variable is **declared**, and,
- the referent **cannot be changed** once it has been initialized.

References (con't)

C++ reference variables have the same properties and semantics as what they refer to.

References (con't)

In C++ object-oriented code:

- `this` is **always** a constant pointer to the object
- `this` is **not** a reference since the introduction of `this` in early C++ *predated* the introduction of references to the language —so `this` was defined to be a constant pointer instead.

C++ Expression Category Taxonomy

$$\begin{tikzcd}
 & & \text{expression} \arrow[dl] \arrow[dr] & & \\
 & \text{glvalue} \arrow[dl] \arrow[dr] & & \text{rvalue} \arrow[dl] \arrow[dr] & \\
\text{lvalue} & & \text{xvalue} & & \text{prvalue}
\end{tikzcd}$$

[1, [basic.lval]]

C++ Expression Category Taxonomy (con't)

- an **lvalue**:
  - designates a **function** or an **object**
  - is historically a value that could appear on the left-hand side of an assignment expression
- an **xvalue** is an **expiring object** value
  - i.e., it refers to an object usually near the end of its lifetime
- a **prvalue** is an **rvalue** that is **not an xvalue**
- a **glvalue** is a **generalized** lvalue, i.e., it designates an lvalue or an xvalue
- an **rvalue**:
  - is an **xvalue**, a **temporary object or a subobject of such**, or a **value not associated** with an object
  - is historically a value that could appear on the right-hand side of an assignment expression

[1, [basic.lval]]

Lvalue References

In practice one can think of an lvalue as a **variable/object** that has a **name**:

- `int i=3; // i is an lvalue`
- `int(3); // int(3) is not an lvalue`

Lvalue References (con't)

Lvalue references must be **initialized** when declared:

- by an **lvalue**, or,
- by an **rvalue iff its referent is `const`**
  - This may cause the compiler to create a temporary variable to hold the value.
  - e.g., `int& i = 2; // ERROR`
  - e.g., `int const& i = 2; // OK`

Lvalue References (con't)

The following C code:

w02/lvalue-ref-1a.c

```c
void foo(int * const v, int const * const k)
{
  *v += *k;
}

int main()
{
  int i=2, j=45;
  foo(&i, &j);
}
```

Lvalue References (con't)

can be thought of as equivalent to this C++ code:

w02/lvalue-ref-1b.cxx

```cpp
void foo(int& v, int const& k)
{
  v += k;
}

int main()
{
  int i=2, j=45;
  foo(i,j);
}
```

Rvalue References

In practice one can think of an rvalue as a **value** that does not have a **name**:

- `int&& i=3; // 3 is an rvalue; i is an lvalue`
- `int&& j=int(2); // int(2) is an rvalue; j is an lvalue`

Rvalue references must be **initialized** by an **rvalue**.

Why Rvalue References?

Rvalue references allow one to define **move semantics** for types.

- When assigning variables, calling functions, and returning locally created variables by value unnecessary copies of variables are made.
- This is **inefficient** when the variable is an **rvalue** as it will be typically be **soon destroyed**.
- It was recognized that there was **no need to copy** rvalues.
- Instead **moving something from an old** variable **into a new** variable is all that needs to be done when the **old variable will (soon) be destroyed**.
- Think of **moving** as an **optimized copy** operation.

Why Rvalue References? (con't)

The complexity of moving structs/classes **with data members**:

- $O(\text{moving data}) = o(\text{copying data})$
  - Non-dynamically allocated objects typically must copy.
- $\Omega(\text{moving data}) = \Omega(\text{copying pointers to the data})$
  - Dynamically allocated objects, e.g., `std::string`, greatly benefit from moves.
- $\Omega(\text{moving data}) \neq 0$
  - If the cost of copying or moving ever appears to be zero, then **copy elison** (which includes **return value optimization (RVO)**) [1, [class.copy]] was performed by the compiler **instead of** copying or moving.

Consider an Example

Consider a `math_vector` that uses dynamic memory allocation to store its values:

w02/move-eg-1.cxx

```cpp
// Given ...
math_vector a{1.0, 1.1, 1.2};
math_vector b{2.0, 2.1, 2.2};
math_vector c{3.0, 3.1, 3.2};
math_vector d{4.0, 4.1, 4.2};

// The expression ...
auto e = a + b + c + d;
```

Consider an Example (con't)

In most OOP languages, `e=a+b+c+d;` executes inefficiently, i.e., something like this:

- `a + b` returns temp1 (copy local variable on return, destroy local variable)
- temp1 `+ c` returns temp2 (copy local variable on return, destroy local variable)
- temp2 `+ d` returns temp3 (copy local variable on return, destroy local variable)
- temp3 becomes `e`

e.g., if `math_vector` uses dynamic memory, then RAM will be allocated, copied, and deallocated many times inefficiently.

Consider an Example (con't)

In C++, assuming `math_vector` is move aware with the code and the compiler does *not* exploit various optimizations involving RVO, **`constexpr`**, perfect forwarding, and other coding techniques such as **expression templates**, etc., then the result of `e=a+b+c+d;` is much better:

- a + b returns temp1 (move local variable on return, destroy local variable)
- temp1 + c returns temp2 (move local variable on return, destroy local variable)
- temp2 + d returns temp3 (move local variable on return, destroy local variable)
- temp3 becomes e

e.g., if `math_vector` uses dynamic memory, then RAM will be "moved" by copying a pointer. This is *way* more efficient.

Templates, References, and Type Deduction

When using templates C++ has the ability to perform full type deduction with a variety of patterns. [1, [temp.deduct.type]]

Perhaps the most commonly used pattern is T&&, e.g.,

w02/t-ref-ref.cxx

```cpp
#include <iostream>

template <typename T>
void print(T&& t) // T&& here matches ALL types incl. any ref and/or cv-qualifiers!
{
  std::cout << t << std::endl;
}

int main()
{
  int i = 65;
  print(5.5); print("Hello"); print(i); print(char('A'));
}
```

Templates, References, and Type Deduction (con't)

There is a *finite* list of template patterns where the C++ compiler will do full type deduction. FYI, these patterns are:

- `T`, `T cv-qual`, `T*`, `T&`, `T&&`, `T[integer-constant]`, `template-name<T>`, `type(T)`, `T()`, `T(T)`, `T type::*`, `type T::*`, `T T::*`, `T (type::*)()`, `type (T::*)()`, `type (type::*)(T)`, `type (T::*)(T)`, `T (type::*)(T)`, `T (T::*)()`, `T (T::*)(T)`, `type[i]`, `template-name<i>`, `TT<T>`, `TT<i>`, `TT<>`
- where:
  - `T` is a template type argument; `TT` is a template template argument; `i` is a template non-type argument; `cv-qual` are cv-qualifiers; `integer-constant` is an integer constant; `template-name` is a class template; `type` is a type;
  - `(T)` is a parameter-type-list where at least one parameter contains a `T`
  - `()` is a parameter-type-list where no parameter type contains a `T`
  - `<T>` is a template argument list where at least one argument contains a `T`
  - `<i>` is a template argument list where at least one argument contains an `i`
  - `<>` is a template argument list where no argument contains a `T` or an `i`

`std::move` and `std::forward`

`std::move(value)`

- **Casts** `value` into an **rvalue**.
- Is used by the programmer to force an lvalue to be an rvalue.
  - Given: `int&& r=13;`
  - Since `r` is an lvalue (it has a name!), to pass it to a function, `foo()`, requiring an rvalue, one would write: `foo(std::move(r));`.

[1, [forward]].

`std::move` and `std::forward` (con't)

`std::forward<Type>(value)`

- Used with **templates** to **enable perfect-forwarding** of **function arguments**.
- Essentially, `std::forward` ensures `value` remains an rvalue if it is, otherwise, it remains as an lvalue.

[1, [forward]].

Perfect Forwarding Definition

The Forwarding Problem Addressed by ISO C++ [1]

For a given expression $E(a_1, a_2, ..., a_n)$ that depends on the (generic) parameters $a_1, a_2, ..., a_n$, [...][perfect forwarding makes it] possible to write a function (object) f such that $f(a_1, a_2, ..., a_n)$ is equivalent to $E(a_1, a_2, ..., a_n)$.

Which implies:

1. f must be able to forward $a_1, a_2, ..., a_n$ to E **unmodified**, and,
2. f must be able to return the result of $E(a_1, a_2, ..., a_n)$ back to the caller **unmodified**.

C++ implements perfect forwarding with:

- rvalue references, move semantics, `std::move`, `std::forward`, and some adjustments to other language features.

Exploiting Copy and Move Semantics

Q. How do you write code to exploit copy and move semantics? Answers:

1. Write appropriate copy and move constructors and assignment operators for all **`struct`**s, **`class`**es, and **`union`**s.
2. Write code in a straight-forward manner, e.g.,
   - **Avoid direct pointer manipulation** by encapsulating pointers within classes.
   - **Prefer references over pointers.**
   - **Prefer values over references.**
   - **Prefer returning by value.**
   - **Use pass-by-value** over pass-by-const-reference **if** a function **must always modify a local copy** of that argument —otherwise seriously consider using pass-by-const-reference.
   - **Use `const`** when a variable/argument will **never** be modified.
   - **Optimize later only if necessary,** e.g., explicitly using `std::move()` or `std::forward<T>()`.

Outline

From Pointers to Ranges

Iterators in C++ are based on the syntax and rules of **pointers**.

Thus, let's review some important particulars about pointers.

A Review Of Pointers

Pointers are permitted:

- to point to a valid object,
  - e.g., `int ar[5]; int *p=&ar[4]; int *q=ar+3;`
  - e.g., `int i=4; int *p=&i;`
- be null (i.e., invalid), or,
  - e.g., `int *p=0; int *q=nullptr;`
    - The integer `0` is cast to the null pointer in pointer contexts.
    - With C++11 or newer use `nullptr` instead of `0`.
- point to one “position” past the valid object — as long as one never accesses the datum pointed to.
  - e.g., `int ar[5]; int *p=&ar[4]+1; int *q=ar+5;`
  - e.g., `int i=4; int *p=&i+1;`

A Review Of Pointers (con't)

Pointers are not permitted:

- to point to any address before a valid object,
  - e.g., given `int ar[5];` then `int *p=&ar[-1];` and `int *q=ar-1;` are illegal
  - e.g., given `int i=4;` then `int *p=&i-1;` is illegal
- to point to any address **after** the one-past-the-valid-object position, or,
  - e.g., given `int ar[5];` then `int *p=ar+6;` is illegal
  - e.g., given `int i=4;` then `int *p=&i+2;` is illegal
- to access an address that does not point to a valid object.
  - e.g., given `int ar[5]; int *p=ar+5;` then `int i=*p;` is illegal
  - e.g., given `int *p=nullptr;` then `int i=p[0];` and `*p=17;` are illegal

A Review Of Pointers (con't)

In C and C++ array and pointer syntax are equivalent in this sense:

- `array[index]` $\equiv$ `*(array+index)`
    - `array` is an array or a pointer
    - `index` is an integer

Applying the address-of operator, `&`, to both sides yields this equivalence:

- `&array[index]` $\equiv$ `array+index`

A Review Of Pointers (con't)

Consequently, these expressions are all the same:

- `array[0]` $\equiv$ `*(array+0)` $\equiv$ `*array`
- `&array[0]` $\equiv$ `array+0` $\equiv$ `array`
- `array[3]` $\equiv$ `*(array+3)`
- `&array[3]` $\equiv$ `array+3`

A Review Of Pointers (con't)

More examples:

- Given `int i=3; int *p=&i;` then `p[0] = 2;` is the same as writing `*p = 2;`.
- Given `int ar[5];` then `*(ar+3) = 2;` is the same as writing `ar[3] = 2;`.
- Given `int ar[5];` then `ar+3` is the same as writing `&ar[3]`.

A Review Of Pointers (con't)

Know pointer values can only be meaningfully compared using `<`, `<=`, `>=`, and `>` with:

- pointers pointing to the same underlying valid object having the same type (subject to **`const-volatile`**-qualification rules), and,
  - e.g., `int ar[5]; int *p=ar; int *q=ar+3; bool b=(p < q);`
- pointers pointing to the one-past-the-valid-object position for the same underlying valid object and type.
  - e.g., `int i; int *p=&i; int *q=&i+1; bool b=(p < q);`
  - e.g., `int ar[5]; int *p=ar; int *q=ar+5; bool b=(p < q);`

A Review Of Pointers (con't)

If pointers can be meaningfully compared using `<`, `<=`, `>=`, and `>` then they can be meaningfully subtracted with the result being the type `std::ptrdiff_t`:

- e.g., `int ar[5]; int *p=ar; int *q=ar+5; auto length=(p - q); // length == 5`

With C++ iterators:

- When subtracting two random-access iterators the result type is represented by the iterator member `typedef difference_type`.
- Prefer using `std::distance(from_pos,to_pos)` over subtraction.

Evolving Code From Pointers To Iterators To Ranges

w02/p2i2r-1.cxx

```cpp
#include <iostream>

int main()
{
  using namespace std;

  int ar[] = { 1, 2, 3, 4, 5 };

  for (int i=0; i != 5; ++i)
    cout << ar[i] << ' ';
  cout << '\n';
}
```

Evolving Code From Pointers To Iterators To Ranges (con't)

w02/p2i2r-2.cxx

```
#include <iostream>

int main()
{
  using namespace std;

  int ar[] = { 1, 2, 3, 4, 5 };
  int *start = &ar[0];     // start points to a[0]
  int *stop = start + 5;   // start points to a[5] which is one past the last element

  for (int *i=start; i != stop; ++i)
    cout << *i << ' ';
  cout << '\n';
}
```

Evolving Code From Pointers To Iterators To Ranges (con't)

w02/p2i2r-3.cxx

```
#include <iostream>

int ar[] = { 1, 2, 3, 4, 5 };

using myiterator = int*;                     // same as: typedef int* myiterator;
myiterator start() { return &ar[0]; }   // hard-coded to ar array for now
myiterator stop() { return start()+5; } // hard-coded to ar array for now

int main()
{
  using namespace std;
  for (myiterator i=start(); i != stop(); ++i) // now use myiterator, start(), and stop()
    cout << *i << ' ';
  cout << '\n';
}
```

Evolving Code From Pointers To Iterators To Ranges (con't)

w02/p2i2r-4.cxx

```
#include <iostream>

int ar[] = { 1, 2, 3, 4, 5 };

using myiterator = int*;
myiterator mybegin() { return &ar[0]; }      // renamed start() to mybegin()
myiterator myend() { return mybegin()+5; }  // renamed stop() to myend()

int main()
{
  using namespace std;

  for (myiterator i=mybegin(); i != myend(); ++i)
    cout << *i << ' ';
  cout << '\n';
}
```

Evolving Code From Pointers To Iterators To Ranges (con't)

w02/p2i2r-5.cxx

```
#include <iostream>

using myiterator = int*;
myiterator mybegin(int* startpos) { return startpos; }  // no longer hard-coded to ar
myiterator myend(int* stoppos) { return stoppos+5; }    // hard-coded to ar's size

int main()
{
  using namespace std;

  int ar[] = { 1, 2, 3, 4, 5 };
  for (myiterator i=mybegin(ar); i != myend(ar); ++i) // pass in ar array
    cout << *i << ' ';
  cout << '\n';
}
```

Evolving Code From Pointers To Iterators To Ranges (con't)

w02/p2i2r-6.cxx

```
#include <algorithm>
#include <iostream>

using myiterator = int*;
myiterator mybegin(int* startpos) { return startpos; }
myiterator myend(int* stoppos) { return stoppos+5; }

int main()
{
  using namespace std;
  int ar[] = { 1, 2, 3, 4, 5 };
  for_each( mybegin(ar), myend(ar), // begin at ar[0] and process up to but not including ar[5]
    [](int i) { cout << i << ' '; } // lambda function
  );
  cout << '\n';
}
```

Evolving Code From Pointers To Iterators To Ranges (con't)

w02/p2i2r-7.cxx

```
#include <algorithm>
#include <iostream>

using myiterator = int*;
myiterator mybegin(int* startpos) { return startpos; }
myiterator myend(int* stoppos) { return stoppos+5; }

int main()
{
  using namespace std;

  int ar[] = { 2, 7, -3, 1, 4 };
  sort(mybegin(ar), myend(ar));
  for_each(mybegin(ar), myend(ar), [](int i) { cout << i << ' '; });
  cout << '\n';
}
```

Evolving Code From Pointers To Iterators To Ranges (con't)

w02/p2i2r-8.cxx

```cpp
#include <algorithm>
#include <iostream>
#include <vector>

int main()
{
  using namespace std;

  vector<int> ar{ 2,7,-3,1,4 }; // replace array literal with std::vector
  sort(begin(ar), end(ar)); // use std::begin() and std::end()
  for_each(begin(ar), end(ar), [](int i) { cout << i << ' '; }); // likewise
  cout << '\n';
}
```

Evolving Code From Pointers To Iterators To Ranges (con't)

w02/p2i2r-9.cxx

```cpp
#include <algorithm>
#include <execution> // For C++17 parallel algorithm's execution policies.
#include <vector>
#include <iostream>

using namespace std;

int main()
{
  vector<int> ar{ 2, 7, -3, 1, 4 };
  sort(execution::par, begin(ar), end(ar)); // parallel
  for_each(execution::par, begin(ar), end(ar), [](int i) { cout << i << ' '; }); // parallel
  cout << '\n';
}

```

Evolving Code From Pointers To Iterators To Ranges (con't)

w02/p2i2r-10.cxx

```cpp
// program is p2i2r-7.cpp modified to support ranges...
#include <algorithm>
#include <iostream>
#include <ranges>     // For C++20 range-related items

struct mycontainer { int ar[5]; }; // a trivial "container" type to hold array
using myiterator = int*;
myiterator begin(mycontainer& c) { return c.ar; }
myiterator end(mycontainer& c) { return c.ar+5; }
static_assert(std::ranges::range<mycontainer>); // i.e., mycontainer is a (minimal) C++20 range

int main()
{
  mycontainer ar{ 2, 7, -3, 1, 4 };
  std::ranges::sort(ar);
  std::ranges::for_each(ar, [](int i) { std::cout << i << ' '; });
  std::cout << '\n';
}
```

Evolving Code From Pointers To Iterators To Ranges (con't)

w02/p2i2r-11.cxx

```
#include <algorithm>
#include <iostream>
#include <ranges>
#include <vector>

int main()
{
  using namespace std;
  vector ar{ 2, 7, -3, 1, 4 };
  ranges::sort(ar);
  ranges::for_each(ar, [](int i) { cout << i << ' '; });
  cout << '\n';
}
```

Examples Using Iterators

Generics, i.e., templates, are used extensively in C++.

- In many cases if you don't see a template being used it is because you are likely using a type alias, e.g., `typedef` or an appropriate `using`, to the template, e.g.,
  - `std::string` is actually `std::basic_string<char,std::char_traits<char>>`
  - `std::cout` is actually an instance of `std::basic_ostream<char,std::char_traits<char>>`

Examples Using Iterators (con't)

Since C++20 there are six (6) kinds of iterators:

- **input**, i.e., `==`, `!=`, `*`, `->`, `++`, but not reachable
- **output**, i.e., `==`, `!=`, `*`, `=` (assignment), `++`, but not (necessarily) reachable
- **forward**, i.e., input iterator interface, `*`, `->`, `++`, and reachable
- **bidirectional**, i.e., forward iterator interface and `--`
- **random-access**, i.e., bidirectional iterator interface, relational operations (i.e., `==`, `!=`, `<`, `<=`, `>=`, `>`, `<=>`), `+`, `-`, and `[]`
- **contiguous**, i.e., random-access iterator interface and elements are stored contiguously

Examples Using Iterators (con't)

In C++ a range is a type that provides **two member functions** as follows:

- `begin()` which returns an iterator interface object to the start of the range
- `end()` which either returns an iterator interface object to **one-position-past-the-end of the range**, or, it is a sentinel value, e.g., `std::default_sentinel` or a suitable custom "end" value that works as such with the range type

Examples Using Iterators (con't)

"Interface checking" is done using **concepts**.

- Before C++20, the programmer was (largely) responsible to ensure the proper interfaces existed.
- C++20 introduced **concepts** (syntax-only) into the language.
  - Semantic checks cannot be done with syntax checks so the programmer must ensure semantic requirements are met.
  - Concepts as a design idea (syntax and semantics) has existed in C++ since before the first C++ standard.
- analogously, "function prototypes (in C and C++) are to functions as concepts are to templates"
  - e.g., being able to check the interface(s) of other types
  - defining concepts requires using template parameters

Examples Using Iterators (con't)

Consider some concepts......

w02/sorts.cxx

```
// std::permutable<I> is defined as ...
//   template <typename I>
//   concept permutable =
//     std::forward_iterator<I> &&
//     std::indirectly_movable_storable<I, I> &&
//     std::indirectly_swappable<I, I>
//   ;
```

(These concepts are defined in the `<iterator>` header.)

Examples Using Iterators (con't)

w02/sorts.cxx

```
template <typename I>
concept forward_permutable_c = std::permutable<I>;

template <typename I>
concept bidirectional_permutable_c = std::bidirectional_iterator<I> &&
  forward_permutable_c<I>;

template <typename I>
concept random_access_permutable_c = std::random_access_iterator<I> &&
  bidirectional_permutable_c<I>;

template <typename I>
concept contiguous_permutable_c = std::contiguous_iterator<I> &&
  random_access_permutable_c<I>;
```

Examples Using Iterators (con't)

A function that returns an iterator that is one position before the end iterator would be:

w02/sorts.cxx

```
constexpr auto one_position_before_end(std::ranges::range auto&& r)
{
  return one_position_before_end(r.begin(),r.end());
}
```

but isn't this cheating —it just calls another function?!!

Examples Using Iterators (con't)

Overloading and (concept) subsumption comes to the rescue:

w02/sorts.cxx

```cpp
template <std::bidirectional_iterator It>
constexpr auto one_position_before_end(It first, It last)
{
  if (first == last) return first;
  else return --last;
}

template <std::forward_iterator It, std::sized_sentinel_for<It> Sent>
constexpr auto one_position_before_end(It first, Sent last)
{
  if (first == last) return first;
  std::ranges::advance(first,last-first-1,last);
  return first;
}

template <std::forward_iterator It, std::sentinel_for<It> Sent>
constexpr auto one_position_before_end(It first, Sent last)
{
  auto prev{ first };
  for (; first != last; ++first)
    prev = first;
  return prev;
}
```

Examples Using Iterators (con't)

But maybe that isn't impressive, let's compute the midpoint iterator position given a range...

w02/sorts.cxx

```cpp
template <
  std::forward_iterator FwdIter,
  std::sized_sentinel_for<FwdIter> Sentinel
>
constexpr auto midpoint(FwdIter first, Sentinel last) ->
  FwdIter
{
  using std::ranges::distance;
  using std::ranges::next;
  return next(first, distance(first,last)/2, last);
}
```

Examples Using Iterators (con't)

But what if we don't know the size of the range involved? No problem:...

w02/sorts.cxx

```
template <std::ranges::forward_range R>
constexpr auto midpoint(R&& r) ->
  std::ranges::iterator_t<R>
{
  using std::ranges::advance;

  auto const last{ r.end() };
  auto slow_iter{ r.begin() };
  auto fast_iter{ slow_iter };
  if (fast_iter != last)
  {
    advance(fast_iter,1);
    while (fast_iter != last)
    {
      advance(slow_iter,1);
      advance(fast_iter,1);
      if (fast_iter == last)
        break;
      advance(fast_iter,1);
    }
  }
  return slow_iter;
}
```

Examples Using Iterators (con't)

Still not impressed? Consider this Bubble Sort...

w02/sorts.cxx

```cpp
template <std::ranges::forward_range R, typename Compare = std::ranges::less>
requires std::sortable<std::ranges::iterator_t<R>,Compare>
constexpr void bubble_sort(R&& r, Compare cmp = {})
{
  bubble_sort(r.begin(),r.end(),cmp);
}
```

w02/sorts.cxx

```cpp
template <forward_permutable_c FwdIt, std::sentinel_for<FwdIt> ItSent,
  typename Compare = std::ranges::less>
requires std::sortable<FwdIt,Compare>
constexpr void bubble_sort(FwdIt first, ItSent last, Compare comp = {})
{
  using std::ranges::iter_swap;
  using std::ranges::next;

  if (is_range_size_zero_or_one(first,last))
    return;
```

Examples Using Iterators (con't)

```

  // [first,last) has at least 2 elements...
  //   * this means prev_i and i are initially valid
  bool swapped;
  do
  {
    swapped = false;
    for (auto prev_i{ first }, i{ next(first) }; i != last; (void)++prev_i, (void)++i)
    {
      if (comp(*i,*prev_i))
      {
        iter_swap(i,prev_i);
        swapped = true;
      }
    }
  }
  while (swapped);
}
```

Examples Using Iterators (con't)

Want shorter? Consider this (two-line) Selection Sort...

w02/sorts.cxx

```cpp
template <forward_permutable_c FwdIt, std::sentinel_for<FwdIt> ItSent,
  typename Compare = std::ranges::less>
requires std::sortable<FwdIt,Compare>
constexpr void selection_sort(FwdIt first, ItSent last, Compare cmp = {})
{
  using std::ranges::iter_swap;
  using std::ranges::min_element;

  for (; first != last; ++first)
    iter_swap(first, min_element(first,last,cmp));
}
```

Examples Using Iterators (con't)

What about the Insertion Sort?

w02/sorts.cxx

```cpp
template <bidirectional_permutable_c BidiIt, std::sentinel_for<BidiIt> ItSent,
  typename Compare = std::ranges::less>
requires std::sortable<BidiIt,Compare>
constexpr void insertion_sort(BidiIt first, ItSent last, Compare cmp = {})
{
  using std::ranges::iter_swap;
  using std::ranges::prev;

  for (auto i{ first }; i != last; ++i)
    for (auto j{ i }; j != first && cmp(*j,*prev(j)); --j)
      iter_swap(prev(j),j);
}
```

Examples Using Iterators (con't)

Isn't the Shell Sort definable in terms of the Insertion Sort?

w02/sorts.cxx

```cpp
template <std::ranges::bidirectional_range R, typename Compare = std::ranges::less>
requires std::sortable<std::ranges::iterator_t<R>,Compare>
constexpr void shell_sort(R&& r, Compare cmp = {})
{
  using namespace std::ranges;

  if (is_range_size_zero_or_one(r))
    return;

  std::initializer_list<std::size_t> const il{
    701, 301, 132, 57, 23, 10, 4, 1
  };

  // Set skip_by to the first position in il that is equal to or smaller
  // than the size of the range only if size is O(1). Otherwise set to
  // il.begin().
  auto skip_by =
    [&]()
```

Examples Using Iterators (con't)

```
      {
        if constexpr(sized_range<R>)
          return lower_bound(il, std::ranges::size(r), greater{});
        else
          return il.begin();
      }()
    ;

  for (; skip_by != il.end(); ++skip_by)
  {
    auto gap_view{ r | std::views::stride(*skip_by) };
    insertion_sort(gap_view.begin(), gap_view.end(), cmp);
  }
}
```

Examples Using Iterators (con't)

Notice Shell Sort was defined to work on ranges. To allow one to pass in iterators instead, one can make such a range like this:

w02/sorts.cxx

```cpp
template <
  bidirectional_permutable_c It,
  std::sentinel_for<It> ItSent,
  typename Compare = std::ranges::less
>
requires std::sortable<It,Compare>
constexpr void shell_sort(It first, ItSent last, Compare cmp = {})
{
  shell_sort(std::ranges::subrange(first,last), cmp);
}
```

Examples Using Iterators (con't)

What about the Heap Sort?

w02/sorts.cxx

```
template <random_access_permutable_c It, std::sentinel_for<It> Sent,
  typename Compare = std::ranges::less>
requires std::sortable<It,Compare>
constexpr void heap_sort(It first, Sent last, Compare = {})
{
  using namespace std::ranges;
  make_heap(first,last);
  sort_heap(first,last);
}

template <std::ranges::random_access_range R, typename Compare = std::ranges::less>
requires std::sortable<std::ranges::iterator_t<R>,Compare>
constexpr void heap_sort(R& r, Compare cmp = {})
{
  heap_sort(r.begin(),r.end(),cmp);
}
```

Examples Using Iterators (con't)

Look at the functions available in the `<algorithm>` header —it is easy to implement an *in-place* merge sort.

Examples Using Iterators (con't)

What about the Quick Sort?

w02/sorts.cxx

```
template <forward_permutable_c It, std::sentinel_for<It> ItSent,
  typename Compare = std::ranges::less>
requires std::sortable<It,Compare>
constexpr void quick_sort(It first, ItSent last, Compare cmp = {})
{
  using namespace std::ranges;

  if (first == last) return;

  // use the last element as the pivot ...
  auto pivot{ one_position_before_end(first,last) };
  auto mid{ midpoint(first,next(pivot)) };
  if (pivot != mid)
    iter_swap(pivot,mid); // swap elements

  auto second_part{
    // partition [first, prev(last)) using *pivot ...
    partition(first, pivot, [&](auto const& v) { return cmp(v,*pivot); })
```

Examples Using Iterators (con't)

```
    };

    // place the pivot value at second_part.begin()...
    // (ASIDE: the pivot value is between the two halves.)
    iter_swap(pivot,second_part.begin());

    quick_sort(first, second_part.begin(), cmp);
    quick_sort(next(second_part.begin()), last, cmp);
}
```

Examples Using Iterators (con't)

What about a different version of the Quick Sort for bidirectional iterators?

w02/sorts.cxx

```
template <bidirectional_permutable_c It, std::sentinel_for<It> ItSent,
  typename Compare = std::ranges::less>
requires std::sortable<It,Compare>
constexpr void quick_sort(It first, ItSent last, Compare cmp = {})
{
  using namespace std::ranges;

  if (first == last) return;

  // NOTE: This requires bidirectional iterators since the previous
  //       position to partition's returned iterator is needed.
  auto mid{ midpoint(first,last) };
  iter_swap(first,mid);
  auto second_part{ partition(next(first),last,
    [&](auto const& v) { return cmp(v,*first); }) };
  auto previous{ prev(second_part.begin()) };
  if (first != previous)
    iter_swap(first,previous);
```

Examples Using Iterators (con't)

```

  quick_sort(first, previous, cmp);
  quick_sort(second_part.begin(), last, cmp);
}
```

Examples Using Iterators (con't)

Doesn't C++ now support three-way relational comparisons? Yes. Consider this three-way partitioning algorithm...

w02/sorts.cxx

```cpp
template <bidirectional_permutable_c It, std::sentinel_for<It> ItSent>
requires std::three_way_comparable<std::iter_value_t<It>, std::weak_ordering>
constexpr auto three_way_partition(
  It first, ItSent last, std::iter_value_t<It> const& value
) -> std::tuple<It,It>
{
  using namespace std::ranges;

  if (first == last)
    return { first, first };

  auto lo{ first };
  auto hi{ position_at_end(first,last) };
  for (auto i{ lo }; i != hi; )
  {
    auto const cmp_result{ *i <=> value };
    if (cmp_result < 0)
```

Examples Using Iterators (con't)

```
        iter_swap(lo++,i++);
      else if (cmp_result > 0)
        iter_swap(i,hi--);
      else
        ++i;
    }
    return { lo, hi };
  }
```

Outline

Operator Overloading: Overview

Except for the:

- `.` (member access)
- `.*` (member access through pointer to member)
- `::` (scope access)
- `?:` (ternary operator), and,
- the preprocessor operators

all other operators in C++ can be overloaded.

Operator Overloading: Overview (con't)

Some operators can *only* be overloaded as non-**`friend`** member functions:

- `=` (assignment operator)
- `( )` (function call operator)
- `[ ]` (array operator)
- `->` (struct/class/union indirection operator)

Operator Overloading: Overview (con't)

Overloading an operator *must* involve at least one user-defined type: operator overloading cannot redefine any language-defined operator definitions.

New operators cannot be created.

It is not possible to change the precedence, grouping, or the number of operands of an operator.

Any operators that have short-circuiting or sequencing features, e.g., `&&`, `||`, and `,`, don't exhibit such when user-defined overloads are called.

Operator Overloading: Overview (con't)

Any overload of `operator->()` must return a raw pointer or an object that overloads `operator->()`.

Avoid overloading operators unless such is necessary. When overloading operators, preserve the conventional meaning of such to avoid confusing other programmers (and yourself) as to what that operator does when used with a specific type.

Operator Overloading: Unary, As Free Function

An example of overloading the unary + operator as a free function:

w02/op-overload-free.cxx

```
struct foo { int i; };

auto operator+(foo const& a)
{
  return foo{ a.i+3 };
}
```

Operator Overloading: Unary, As Member Function

An example of overloading the unary + operator as a member function:

w02/op-overload-class.cxx

```cpp
struct foo
{
  int i;

  auto operator+() const // notice there is no argument
                         // the object itself is the first argument
  {
    return foo{ this->i+3 };
  }
};
```

Operator Overloading: Unary, As Member Function (con't)

An example of overloading the unary + operator as a **friend** function defined when it is declared as a **friend**:

w02/op-overload-friend.cxx

```cpp
struct foo
{
  int i;

  friend auto operator+(foo const& a)
  {
    return foo{ a.i+3 };
  }
};
```

NOTE: **friend** functions are free functions that have access to all protected and private scoped names in a class declaring that function as its friend.

Operator Overloading: Unary, As Member Function (con't)

An example of overloading the unary + operator as a **friend** function (using a prototype and the function is defined elsewhere):

w02/op-overload-friend2.cxx

```
struct foo; // forward declaration
auto operator+(foo const&); // function prototype

struct foo
{
  int i;
  friend auto operator+(foo const&); // friend declaration
};

auto operator+(foo const& a) // definition
{
  return foo{ a.i+3 };
}
```

Operator Overloading: Binary, As Free Function

An example of overloading the binary + operator as a free function:

w02/op-overload-free-binary.cxx

```cpp
struct foo { int i; };

auto operator+(foo const& a, foo const& b)
{
  return foo{ a.i + b.i };
}
```

Operator Overloading: Binary, As Member Function

An example of overloading the binary + operator as a member function:

w02/op-overload-class-binary.cxx

```cpp
struct foo
{
  int i;

  auto operator+(foo const& rhs) const // argument is the second operand
                                // this object is the first operand
  {
    return foo{ this->i + rhs.i };
  }
};
```

Operator Overloading: Binary, As Member Function (con't)

An example of overloading the binary + operator as a **friend**:

w02/op-overload-friend-binary.cxx

```
struct foo
{
  int i;

  friend auto operator+(foo const& a, foo const& b)
  {
    return foo{ a.i + b.i };
  }
};
```

Operator Overloading: Prefix and Postfix ++ and -- Operators

The postfix operators, `++` and `--`, are special:

- the postfix operator has an additional unused **`int`** parameter that the compiler will internally optimize away.

This allows the function definitions of the two variants of each postfix operator to be distinguished.

Operator Overloading: Prefix and Postfix ++ and -- Operators (con't)

By convention, prefix operator overloads should return lvalues.

By convention, postfix operator overloads should return rvalues.

Operator Overloading: Prefix and Postfix ++ and -- Operators (con't)

Often the postfix operator overload can be defined in terms of the prefix overload:

w02/op-overload-prepostfix.cxx

```cpp
struct foo
{
  int i;

  foo& operator++()
  {
    ++this->i;
    return *this;
  }

  foo operator++(int) // postfix operator
  {
    foo const retval{*this}; // make a copy
    this->operator++(); // call prefix operator on *this
    return retval; // return the copy
  }
};
```

Operator Overloading: IOStream Operator Overloads

The IOStreams hierarchy makes use of operator overloads to produce formatted output, e.g.,

w02/op-overload-io.cxx

```cpp
#include <iostream>

struct foo { int i; };

std::ostream& operator<<(std::ostream& os, foo const& f) // i.e., notice const&
{
  os << f.i;
  return os;
}

std::istream& operator>>(std::istream& is, foo& f) // i.e., notice & and no const
{
  is >> f.i;
  return is;
}

int main()
```

Operator Overloading: IOStream Operator Overloads (con't)

```
{
  foo a;
  std::cin >> a;
  std::cout << "read in: " << a << '\n';
}

```

Operator Overloading: IOStream Operator Overloads (con't)

This version uses some unformatted I/O calls to check some characters to ensure input is valid (and writes such out when outputting):

w02/op-overload-io2.cxx

```
#include <istream>
#include <ostream>
#include <iostream>
#include <string>

struct foo { int i; };

std::ostream& operator<<(std::ostream& os, foo const& f) // i.e., notice const&
{
  os << '[' << f.i << ']';
  return os;
}

std::istream& operator>>(std::istream& is, foo& f) // i.e., notice & and no const
{
  if (is.get() != '[')
  {
```

Operator Overloading: IOStream Operator Overloads (con't)

```
      is.unget(); // put character read in back in stream buffer
      is.setstate(std::ios_base::failbit); // fail the stream
      return is;
    }

    int temp;
    if (is >> temp && is.get() == ']')
      f.i = temp;
    else
      is.setstate(std::ios_base::badbit); // make stream bad
    return is;
  }

int main()
{
  foo a;
  if (std::cin >> a)
    std::cout << "read in: " << a << '\n';
}

```

Operator Overloading: IOStream Operator Overloads (con't)

This version uses some inserted whitespace on output to make it easier to skip whitespace when reading in data to process each component. This makes it much easier to write I/O code to have some more extra formatting:

w02/op-overload-io3.cxx

```
#include <istream>
#include <ostream>
#include <iostream>
#include <string>

struct foo { int i; };

std::ostream& operator<<(std::ostream& os, foo const& f) // i.e., notice const&
{
  os << "foo{ " << f.i << " }";
  return os;
}

std::istream& operator>>(std::istream& is, foo& f) // i.e., notice & and no const
{
```

Operator Overloading: IOStream Operator Overloads (con't)

```
  std::istream::sentry s(is, true); // true ensures whitespace is skipped
  if (s) // i.e., if the stream is in a good state
  {
    std::string before, after;
    int temp;
    if ((is >> before >> temp >> after) && before == "foo{" && after == "}")
      f.i = temp; // set f since temp was properly read in
    else
      is.setstate(std::ios_base::badbit); // temp not properly read in; make stream bad
  }
  return is;
}

int main()
{
  foo a;
  if (std::cin >> a)
    std::cout << "read in: " << a << '\n';
}

```

Outline

Relational Operators: Before C++20

w02/rel-before-cxx20.cxx

```
// for some user-defined type, e.g., ...
struct foo { int i; };
// one needs to write all relevant/needed relational operator overloads which
// must return bool and need const& parameters, e.g.,
bool operator==(foo const& a, foo const& b) { return a.i == b.i; }
bool operator!=(foo const& a, foo const& b) { return a.i != b.i; }
bool operator <(foo const& a, foo const& b) { return a.i < b.i; }
bool operator<=(foo const& a, foo const& b) { return a.i <= b.i; }
bool operator>=(foo const& a, foo const& b) { return a.i >= b.i; }
bool operator >(foo const& a, foo const& b) { return a.i > b.i; }

int main()
{
  foo a{1}, b{2};
  if (a < b) { /* ... */ }
}
```

i.e., one had to write each and every desired relational operator manually.

Relational Operators: C++20

w02/rel-since-cxx20.cxx

```cpp
#include <compare>      // needed for std::*_ordering definitions

// for some user-defined type, e.g., ...
struct foo
{
  int i;

  // One can instead write a friend operator⟺(const&, const&) "= default"
  // definition which will (lexicographically) define ⟺, ═, ≠, <, ⩽, ⩾,
  // and >
  friend constexpr auto operator<=>(foo const&, foo const&) noexcept = default;
};

int main()
{
  foo a{1}, b{2};
  if (a < b) { /* ... */ }
}
```

Relational Operators: C++20 (con't)

In C++20, the language's treatment of *all* of the relational operators was *significantly changed*, and, a new *three-way-operator*, **operator** `<=>`, was introduced.

Before C++20:

- one had to write each and every relational operator for a type to support all desired comparisons
  - e.g., any/all of `operator==`, `operator!=`, `operator<`, `operator<=`, `operator>=`, `operator>`
- such definitions were, clearly, based on either or both `operator==` and `operator<` definitions
  - leading to a more code clutter and maintenance
  - leading to unintented mistakes/bugs

Relational Operators: C++20 (con't)

Since C++20:

- If there is a definition of `operator==` then the compiler will generate a definition of `operator!=` that performs the negation of `operator==`
  - This assumes the user did not write a definition for `operator!=`.
- in cases where the arguments passed to `operator==` or `operator<=>` are *not* the same, the compiler will automatically generate the symmetric variants of either/both of these operators
  - This assumes the user did not write explicit symmetric definition(s).
- if `operator<=>` is `= default`ed, then such implies a compiler-generated `= default`ed `operator==`
  - If the user does not explicitly provide a `operator==` definition.
  - The reverse is not true: if `operator<=>` is not defaulted, then the compiler will *not* generate `operator==`.

Relational Operators: C++20 (con't)

- defining an `= default`ed `operator<=>` function will automatically generate all of the relational operators
  - unless the user explicitly wrote / deleted specific relational operator definitions
  - i.e., the compiler will favour using user-defined definitions / deletions over generating definitions
- either or both `operator==` and `operator<=>` can be `= default`ed
  - when done, the compiler generates its definitions processing each data member in the order of appearance in the type definition, i.e., using so-called lexicographical ordering (of data members)
  - if `operator<=>` has an `auto` return type, then the compiler will determine the strongest ordering possible based on the ordering of each of the data members in order to select one of `std::strong_ordering`, `std::weak_ordering`, or `std::partial_ordering`

Relational Operators: C++20 (con't)

- if there is no `operator<=>` then there will be no auto-generation of `operator<`, `operator<=`, `operator>=`, `operator>`
- when using `operator<=>` one must include the `<compare>` header file
- sometimes it is possible that automatically-generated code will be inefficient
    - e.g., this can happen if there is a more efficient algorithm than lexicographically computing `operator==`
        - e.g., if you were implementing a string class that stores the string and the size of the string, you would write `operator==` to *first* compare the sizes of the two strings *before* comparing each element of the strings as that is more efficient; —the compiler-generated version (if it was even correct —it might not be if pointers are involved) would be lexicographical and therefore dependent on the *order data members were declared* possibly causing all elements to be compared before the sizes of the two strings.
    - Solution: Write an explicit definition of `operator==` (and any other operator) when there is a more efficient definition that should be used instead.

Relational Operators: C++20 (con't)

- if the compiler cannot automatically-generate definitions, then compile-time errors will result and one will have to address such or write such definitions manually
  - Ensure each of these overloads' two parameters are passed as `const&`.
  - Ensure each relational operator returns `bool`.
  - Ensure `<=>` returns one of the three orderings.
- It is possible to explicitly tell the compiler which ordering to use with `operator<=>` by explicitly setting the return type of such to one of the three orderings.
- While automatically generated code is nice, if there is a more efficient way to implement an operator than lexicographically, implement that more efficient operator.
  - In general when this is done, `operator<=>` will not be able to have an `auto` return type —explicitly define the ordering instead.

Relational Operators: C++20 (con't)

So, yes, while this looks like it can be easy to use, the specifics can become quite complicated. The ISO C++ committee put a lot of work in to this in order to get the details correct.

Usually one would define `operator==` (explicitly or defaulted) and `operator<=>` (defaulted).

If a class' use of all data members are as values, those data members all have relational operators already properly defined on them, and, all of those data members participate in determining the computed results of the relational operators, then one can easily have the compiler generate relational operator and/or three-way-operator code.

Otherwise one *will* need to write some definitions.

This is an example where the `operator<=>` function is explicitly written —not merely `= default`ed, e.g.

Relational Operators: C++20 (con't)

w02/rel-since-cxx20b.cxx

```cpp
#include <compare>
#include <iostream>
struct foo
{
  int i;
  // operator== must be written since <=> is not =default...
  friend bool operator==(foo const& lhs, foo const& rhs) = default;
  friend constexpr auto operator<=>(foo const& lhs, foo const& rhs) noexcept
  {
#if 1
    return lhs.i <=> rhs.i; // use built-in operator<=>(int,int) definition
#else // or do this more manually...
    if (lhs.i < rhs.i)
      return std::strong_ordering::less;
    else if (lhs.i == rhs.i)
      return std::strong_ordering::equal;
    else
      return std::strong_ordering::greater;
#endif
```

Relational Operators: C++20 (con't)

```
  }
};

int main()
{
  foo a{1}, b{2};
  std::cout << (a == b) << (a != b) << (a < b) << (a <= b) << (a >= b) << (a > b) << '\n'
    // besides directly comparing ordering values, orderings can be compared
    // relationally to 0, e.g.,
    << ((a <=> b) == 0) << ' ' << ((a <=> b) < 0) << '\n';
}
```

Relational Operators: Relational Orderings

In mathematics, a *predicate* is a function that returns a boolean value.

In C++ predicates return **`bool`** and are either unary or binary. They are used extensively in the Standard Library.

Relational Operators: Relational Orderings (con't)

The C++ Standard Library **defines key-based sorting/ordering** using **strict weak ordering** using a *single* predicate.

Some of the Standard Library items such applies to include:

- all associative, i.e., sorted, containers,
- all container member functions requiring a key-based sorting/ordering criterion, and,
- all algorithms requiring a key-based sorting/ordering criterion.

NOTE: If the predicate is *not* passed/set explicitly, then the Standard Library definition of concern will use the `operator<` definition for that type.

Relational Operators: Relational Orderings (con't)

Strict Weak Order [1, §25.8 [alg.sorting]] [2, §7.7, pp.314–315]

Given an operator, e.g., $<$, the following properties must hold:

- Irreflexive: $\neg(a < a)$
- Antisymmetric: $(a < b) \Rightarrow \neg(b < a)$
- Transitive: $(a < b) \land (b < c) \Rightarrow (a < c)$
- Transitivity of equivalence:

$$\begin{aligned} &\neg(a < b) \land \neg(b < a) \land \neg(b < c) \land \neg(c < b) \\ &\qquad \Rightarrow \neg(a < c) \land \neg(c < a) \end{aligned}$$

- Using this definition of equivalence: $\neg(a < b) \land \neg(b < a)$

Relational Operators: Relational Orderings (con't)

IMPORTANT!

- The term **strict** refers to the irreflexive requirement.
- The term **weak** refers to requirements that are less strong than a total order, but, stronger than those for a partial order.

Using:

- the above definition of equivalence for `operator==`,
- the strict weak ordering requirements for `operator<`, and,
- defining the remaining relational operators in terms of such,

then such is *not* what one is typically used to:

- i.e., if $\neg(a < b) \equiv \top$ does not imply $(a >= b) \equiv \top$.

So when calling `a < b` returns `false`, one may/will need to call `b < a` under strict weak order semantics.

Relational Operators: Relational Orderings (con't)

Total Orders

Total orders have the following properties under an operator, e.g., R:

- Irreflexive: $\neg(aRa)$, if and only if *strict*
- Reflexive: (aRa), if and only if *non-strict*
- Antisymmetric: $(aRb) \Rightarrow \neg(bRa)$
- Transitive: $(aRb) \wedge (bRc) \Rightarrow (aRc)$
- Totality: $(aRb) \vee (bRa)$
    - i.e., all elements are comparable

ASIDE: Total ordering is what one is typically used to, e.g., if $\neg(a < b)$ then such implies $a >= b$.

Relational Operators: Relational Orderings (con't)

Partial Orders

Partial orders have the following properties under an operator, e.g., R:

- Irreflexive: $\neg(aRa)$, if and only if *strict*
- Reflexive: (aRa), if and only if *non-strict*
- Antisymmetric: $(aRb) \Rightarrow \neg(bRa)$
- Transitive: $(aRb) \vee (bRc) \Rightarrow (aRc)$

Where not all elements, a and b, are comparable under R.

- i.e., $\exists a, b \in T, \neg(aRb) \wedge \neg(bRa)$

Relational Operators: Relational Orderings (con't)

In C++20:

- `std::strong_ordering` corresponds to a *total order*
- `std::weak_ordering` corresponds to a *weak order*
- `std::partial_ordering` corresponds to a *partial order*
  - Notice the definition of `std::partial_ordering` permits `std::partial_ordering::unordered` results.
  - In terms of **`operator`**`<` semantics each of the above must be *strict*.

Relational Operators: Relational Orderings (con't)

Effectively, the C++20 orderings are mapped to strict weak order (SWO) semantics for `operator<` using the SWO definition of equivalence to determine equivalence using `operator<`.

- Prior to C++20 this was (always) true for key-based sorting/ordering.
- Such is still true in C++20 except implementations might deal with orderings in more efficient ways since the type of ordering can be determined at compile-time and be exploited by code.

Additional notes:

- A *weak order* doesn't have a notion of *equality*, instead it has the notion of *equivalence*.

Relational Operators: Relational Orderings (con't)

- In C++20, if an ordering is not a `std::strong_ordering`, then the definition of **`operator`**`==` must be using the SWO definition of *equivalence* which means, in the worst case, that **`operator`**`<` has to be called twice to know if two elements are equivalent.
- When a *partial order* has incomparable/unordered elements such is treated as equivalent unto itself which permits unordered comparisons under SWO semantics. (If this wasn't done, then clearly it would be impossible to compare those elements.)
- With the introduction of **`operator`**`<=>` it is possible for code to determine at compile-time which kind of ordering is being used for a type. This, in turn, can enable more efficient code.
  - e.g., a `std::strong_ordering` allows using the fact that $(\neg(a < b) \equiv \top) \implies ((a >= b) \equiv \top)$.

Relational Operators: Relational Orderings (con't)

Having said the above, realize that C++ key-based ordering *only* needs **`operator`**`<` to be defined to sort/order objects.

For example, if a class only has **`operator`**`<` defined, it is possible to fully use `std::sort`, `std::map`, etc.

Outline

5. Attributes
    - `[[deprecated]]`
    - `[[fallthrough]]`
    - `[[likely]]` and `[[unlikely]]`
    - `[[maybe_unused]]`
    - `[[nodiscard]]`
    - `[[noreturn]]`
    - `[[no_unique_address]]`

Attributes: Overview

C++11 introduced standard syntax for *implementation-defined language extensions* to the C++ language called **attributes**:

- can be used almost everywhere in a C++ program
- can be applied to almost everything
- only valid where the implementation permits it to be valid
  - implementations may ignore unknown attributes without causing an error
- normally apply to the *preceding entity*
  - With *declarations*, attributes may appear *before* or *after* the *name*.
- all non-standard attributes are in an attribute namespace for that implementation, e.g., `[[gnu::packed]]`, `[[clang::trivial_abi]]`, `[[msvc::intrinsic]]`

Attributes: [[deprecated]]

This attribute indicates the name/entity being declared is permitted but is discouraged for some reason, e.g., it is expected to be removed/significantly changed later.

This attribute has two forms:

- `[[deprecated]]`
- `[[deprecated("reason")]]`

w02/deprecated.cxx

```cpp
[[deprecated]] void foo() { }
[[deprecated("bar() will be removed in next release")]] void bar() { }
void foo(int) { }
void bar(int) { }
```

Attributes: [[fallthrough]]

This attribute indicates that code falls through from the previous (**switch**) case label into the next, and, informs the compiler not to issue a warning.

w02/fallthrough.cxx

```cpp
#include <iostream>
int main()
{
  if (int i; std::cin >> i)
  {
    switch (i)
    {
      case 1:
      case 2:
        std::cout << "1 or 2\n";
        [[fallthrough]];
      case 3: // no warning given here
        std::cout << "3\n";
    }
  }
}
```

Attributes: [[likely]] and [[unlikely]]

This attribute informs the compiler that a label/statment is likely or unlikely to occur, e.g.,

w02/likely-unlikely1.cxx

```
int foo(int i)
{
  switch (i)
  {
    case 1:
      [[fallthrough]];

    case 2: [[likely]]
      return 1;
  }
  return 0;
}
```

Attributes: [[likely]] and [[unlikely]] (con't)

w02/likely-unlikely2.cxx

```cpp
constexpr unsigned long long fib(unsigned long long n) noexcept
{
  if (n == 0) [[unlikely]]
    return 0;
  else if (n == 1) [[unlikely]]
    return 1;
  else [[likely]]
    return fib(n-1) + fib(n-2);
}

constexpr unsigned long long factorial(unsigned long long n) noexcept
{
  unsigned long long retval{1};
  for (; n > 1; --n) [[likely]]
    retval *= n;
  return retval;
}
```

Attributes: [[maybe_unused]]

This attribute suppresses compiler warnings on an unused entity, e.g.,

w02/maybe_unused.cxx

```
int main()
{
  [[maybe_unused]] int i;
}
```

Attributes: [[nodiscard]]

This attribute asks the compiler to issue a warning if the return value is discarded. There are two forms:

- `[[nodiscard]]`
- `[[nodiscard("some text")]]`

code/w02/nodiscard.cxx

```cpp
#include <iostream>
struct [[nodiscard]] super_important { };
struct foo {
  int i;
  [[nodiscard]] bool operator==(foo const& f) const { return i == f.i; }
  static super_important bar() { return {}; }
};
int main() {
  foo a{1}, b{2};
  a == b;         // warning issued since bool is discarded
  foo::bar();     // warning issued since super_important is discarded
}
```

Attributes: `[[nodiscard]]` (con't)

A number of C++ functions are declared with the `[[nodiscard]]` attribute including:

- **`operator new`** and **`operator new`**`[]`
- various `allocate()` member functions
- `std::launder()`
- `std::assume_aligned()`
- various `empty()` member functions
- `std::async()`

Attributes: [[noreturn]]

This attribute informs the compiler that the function does not return, e.g.,

w02/noreturn.cxx

```cpp
#include <cstdlib>
[[noreturn]] void foo() { throw "something"; }
[[noreturn]] void bar() { for (;;) ; }
[[noreturn]] void quit() { std::exit(10); }
```

Attributes: `[[noreturn]]` (con't)

A number of C++ functions are declared with the `[[noreturn]]` attribute including:

- `std::abort()`
- `std::exit()`
- `std::quick_exit()`
- `std::terminate()`
- `std::rethrow_exception()`
- `std::rehtrow_nested()`
- `std::throw_with_nested()`
- `std::longjmp()`

Attributes: [[no_unique_address]]

This attribute indicates that a non-static data member need not have an address distinct from all other non-static data members in the same class. This allows such members to be overlapping with other non-static data members or base class subobjects. For example, an empty class data member can be optimized to use no space. e.g.,

code/w02/no_unique_address.cxx

```
struct foo { }; // i.e., foo is an empty class
struct bar1
{
  int i;
  [[no_unique_address]] foo f;
  int j;
};
struct bar2 { int i; int j; };

static_assert(sizeof(foo) >= 1); // the size of any object must at least be one
static_assert(sizeof(bar1) == sizeof(bar2));
```