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

Introduction, History, and More

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.

2026-01-06

Outline

Course Syllabus

From the official senate-approved calendar...

The main objective of this course is to explore advanced topics of the object oriented paradigm through the use of the C++ programming language. Topics covered include: advanced object oriented design, the use of abstraction to manage complexity, objects and classes, inheritance and class hierarchies, multiple inheritance, operator and method overloading, namespaces and visibility, templates, dynamic binding and virtual functions, exception handling, multi-threading and C++ standard library. In addition, the course will include a practical project, solving a real-life problem, implemented in C++, involving the client/server methodology, and an interface to a database using a graphics toolkit. (Prerequisites: COMP-2120, COMP-2560.) (3 lecture hours a week)

Course Syllabus (con't)

The instructor also adds:

- The course syllabus was written years ago and C++ has evolved considerably since then.
- While C++ is an "object-oriented language" and does support "object-oriented programming":
  - C++ supports *many* additional programming paradigms —not just object-oriented programming.
  - Don't think only object-oriented programming.
  - In C++, generic programming and "concepts" are far more important than using object-oriented programming.
  - C++ programs will often only make use of object-based (no inheritance) programming rather than use object-oriented (has inheritance) programming.

Course Syllabus (con't)

- This course aims to reasonably cover the current C++ standard and parts of the next C++ standard and their underlying notions.
  - It is not possible in a 12 week semester to cover everything.
- There are no ISO C++ Standard (SQL) database, networking, or graphics APIs:
  - for "database" IOStreams and `<filesystem>` will be used
  - "networking" and "graphics" will likely not be covered for time reasons
    - Outside of this course should you want to use networking, consider using the Boost.ASIO library. That said, there are many third-party networking libraries for C and C++that can be used.
    - There are many third-party graphics libraries for C and C++that can be used.

Creation of C++

The C++ programming language was:

- created by **Bjarne Stroustrup** at Bell Labs in 1979
- started as an **enhancement** to the C programming language
- originally called **C with Classes**
- renamed to **C++** in 1983

The C++ programming language is a **statically-typed**, compiled, free-form, **general-purpose**, and **multi-paradigm** programming language.

ISO Standards

The C++ language is defined by these ISO/IEC 14882 standards:

ISO Year Released	Name	ISO Name	Remarks
1998	C++98	ISO/IEC 14882:1998	first standard
2003	C++03	ISO/IEC 14882:2003	bug fixes
2011	C++11	ISO/IEC 14882:2011	concurrency, ...
2014	C++14	ISO/IEC 14882:2014	bug fixes and small improvements
2017	C++17	ISO/IEC 14882:2017	attributes, folds, parallel algorithms, ...
2020	C++20	ISO/IEC 14882:2020	concepts, coroutines, modules, ...
2024	C++23	ISO/IEC 14882:2024	`<mdspan>`, `<stacktrace>`, `<stdfloat>`, ...

ISO Standards (con't)

C++ also relies on parts of ISO/IEC 9899 standards for the C programming language, e.g.,

ISO Year Released	Name	ISO Name	Remarks
1989	C89	ISO/IEC 9899:1990	first standard
1999	C99	ISO/IEC 9899:1999	restricted pointers, ...
2011	C11	ISO/IEC 9899:2011	concurrency, ...
2018	C17	ISO/IEC 9899:2018	bug fixes
2024	C23	ISO/IEC 9899:2024	decimal floating-point, bit precise integers, ...

ISO Standards (con't)

Details concerning C++'s use of C are specified in the C++ standards, for example in these locations:

- §1 Scope [intro.scope],
- §2 Normative references [intro.refs], and,
- Annex C: Compatibility [diff]

as well as other locations.

NOTE: C++ does not "import" all of C —only specific components of the C standard are "imported".

Language Design Objectives

General aims (from [3, §4.2, p. 110], emphasis added):

- C++'s **evolution** must be **driven by real problems**.
- **Don't get involved** in a **sterile quest** for **perfection**.
- C++ must be useful **now**.
- **Every feature** must have a **reasonably obvious implementation**.
- Always **provide a transition** path.
- C++ is a **language, not a complete system**.
- Provide **comprehensive support** for each **supported style**.
- **Don't** try to **force people** [to use constructs they don't want or need].

Language Design Objectives (con't)

Design support rules (from [3, §4.3, p. 114], emphasis added):

- Support **sound** design notions.
- Provide **facilities** for **program organization**.
- **Say** what you **mean**.
- **All features** must be **affordable**.
- It is **more important** to **allow** a **useful feature** than to **prevent every misuse**.
- Support **composition** of software from **separately developed parts**.

Language Design Objectives (con't)

Language-technical rules (from [3, §4.4, p. 117], emphasis added):

- **No implicit violations** of the **static type system**.
- Provide **as good support** for **user-defined** types as for **built-in types**.
- **Locality** is good.
- **Avoid order dependencies.**
- **If in doubt**, pick the variant of a feature that is the **easiest to teach**.
- **Syntax matters** (often in perverse ways).
- **Preprocessor usage** should be **eliminated**.

Language Design Objectives (con't)

Low-level programming support rules (from [3, §4.5, p. 120], emphasis added):

- **Use traditional** (dumb) **linkers**.
- **No gratuitous incompatibilities** with C.
- **Leave no room** for a **lower-level language** below C++ (except assembler).
- *What you **don't use**, you don't **pay for** (zero-overhead rule).*
- **If in doubt**, provide means for **manual control**.

WARNING!

- C++ is a very rich, powerful language.
- Know you **won't** learn it the day before a test or an assignment! People that have tried to do this, i typically find themselves dropping the course.
- You will **need to keep up** with the readings, assignments, studying, and **practice programming** throughout this course to be successful.

Modern C++ (con't)

Modern C++ features:

- compatible with most things from the **C language**
- full **static-type checking**
- the ability to **overload functions** including for operators, e.g., ==, <
- **generic programming** and **static polymorphism**
- **object-oriented programming** and **dynamic polymorphism**
  - encapsulation, inheritance, and virtual member functions in addition to references, data, and function pointers
- exception handling; namespaces
- value, pointer, and reference (lvalue and **rvalue**) semantics
- the C Standard Library and the C++ Standard Library
- **concurrency**, **coroutines**, and ... much more

In This Course

NOTICE

- Unless you are explicitly given otherwise in writing, you **must** use C++ and the C++ Standard Library in all assessed work in this course.
  - Failure to do this will (very likely) result in a zero (0) being assigned.
- You are also **explicitly prohibited** from using the C Standard Library when there is an **equivalent** in the C++ Standard Library.
- This is a C++ course —not a course with other programming languages.

Outline

Imperative Paradigm

The **imperative** programming paradigm defines computation in terms of programming statements that describe changes in state.

- i.e., program statements describe *how* something is to be done instead of describing only *what* is to be done
- e.g., the C subset of C++ is imperative

The imperative paradigm includes the **procedural** paradigm.

Procedural Paradigm

"Decide which procedures you want;
use the best algorithms you can find."
[5, §2.3, p. 23]

"[...] the idea of composing a program
out of functions operating on arguments. [...]
Explicit abstraction mechanisms [...] are not used."
[4, §22.1.3, p. 781]

Procedural Paradigm (con't)

The **procedural** programming paradigm is an **imperative** paradigm originally represented by the C language subset which provides:

- a set of **scalar data types**,
- the ability to define and manipulate **arrays**,
- the ability to define and manipulate **aggregate data types**,
- a set of **conditional constructs**,
- a set of **looping constructs**,
- a set of **built-in operators**,
- the ability to define, reference, and call **functions**,
- changing the values of **variables**, and,
- the ability to query, set, and manipulate **addresses**.

Modular Paradigm

"Decide which modules you want;
partition the program so that data is hidden within modules."
[5, §2.4, p. 26]

"[...] compose our systems out of 'components' [...]
that we can build, understand, and test in isolation. [...]
so that they can be used in more than one program ('reused')."
[4, §22.1.2.5, p. 779]

Modular Paradigm (con't)

The **modular** programming paradigm enables **encapsulation**: the ability to expose or hide functions/types defined in different modules.

- In C++ a module can be defined by the contents of a source file, or, "translation unit".
- In C++ a module can also be defined by a `namespace`s, `struct`s, `union`s, or `class`es;

Modules can be used to provide a **simple but limited** way to **abstract** data.

Modular Paradigm (con't)

A module using a **namespace** to implement a stack:

w01/tcxxpl-2p5p1.hxx

```
1  #ifndef code_tcxxpl_2p5p1_hxx_
2  #define code_tcxxpl_2p5p1_hxx_
3
4  namespace Stack {
5    struct Rep;
6    typedef Rep& stack; // definition of stack layout is elsewhere
7    stack create();     // make a new stack
8    void destroy(stack s);  // delete s
9    void push(stack s, char c); // push c onto s
10   char pop(stack s);  // pop s
11 } // namespace Stack
12
13 #endif // #ifndef code_tcxxpl_2p5p1_hxx_
```

–Code is from [5, §2.5.1, p. 30]

Modular Paradigm (con't)

Equivalent C subset code:

w01/tcxxpl-2p5p1-c.hxx

```
1  #ifndef code_tcxxpl_2p5p1_c_hxx_
2  #define code_tcxxpl_2p5p1_c_hxx_
3
4  struct Stack_Rep;
5  typedef Stack_Rep* stack; // def'n of stack layout is elsewhere
6
7  extern stack Stack_create();     // make a new stack
8  extern void Stack_destroy(stack s);  // delete s
9  extern void Stack_push(stack s, char c); // push c onto s
10 extern char Stack_pop(stack s);  // pop s
11
12 #endif // #ifndef code_tcxxpl_2p5p1_c_hxx_
```

–Code converted by Paul Preney.

Modular Paradigm (con't)

Now let's use the stack in C++:

w01/tcxxpl-2p5p1-f.cxx

```
1  #include "tcxxpl-2p5p1.hxx"
2
3  struct Bad_pop { };
4
5  void f()
6  {
7    Stack::stack s1 = Stack::create(); // make a new stack
8    Stack::stack s2 = Stack::create(); // make another new stack
9
```

Modular Paradigm (con't)

w01/tcxxpl-2p5p1-f.cxx

```
10    Stack::push(s1, 'c');
11    Stack::push(s2, 'k');
12
13    if (Stack::pop(s1) != 'c') throw Bad_pop();
14    if (Stack::pop(s2) != 'k') throw Bad_pop();
15
16    Stack::destroy(s1);
17    Stack::destroy(s2);
18  }
```

–Code is from [5, §2.5.1, p. 30]

Modular Paradigm (con't)

Or if using the C subset code:

w01/tcxxpl-2p5p1-f-c.cxx

```
1  #include "tcxxpl-2p5p1-c.hxx"
2
3  extern void handle_error();
4
5  void f()
6  {
7    stack s1 = Stack_create(); // make a new stack
8    stack s2 = Stack_create(); // make another new stack
9
```

Modular Paradigm (con't)

w01/tcxxpl-2p5p1-f-c.cxx

```
10    Stack_push(s1, 'c');
11    Stack_push(s2, 'k');
12
13    if (Stack_pop(s1) != 'c') handle_error();
14    if (Stack_pop(s2) != 'k') handle_error();
15
16    Stack_destroy(s1);
17    Stack_destroy(s2);
18  }
```

–Code converted by Paul Preney.

Modular Paradigm (con't)

All that is left is to write definitions in single file:

- e.g., `tcxxpl-2p5p1-defs-cxx.cxx` for the C++ version, and,
- e.g., `tcxxpl-2p5p1-defs-c.cxx` for the C subset version.

Modular Paradigm (con't)

In the C subset version any file-scoped local variables must be declared as follows to make then "private" or "internal" to the module:

- `static` if the code is actual C code
  - If true, then `extern "C"` and using the preprocessor macro `__cplusplus` will also need to be used in the header file to ensure compilation compatibility with both C and C++.
  - If true, then the file extension above should be `.c` not `.cxx`.
- within an **anonymous namespace** if the code is C++ code
  - The above C use of `static` shouldn't be used in C++98: use an anonymous namespace.
  - An anonymous namespace is a namespace without a name.

Modular Paradigm (con't)

With the above stack code, one can only have a stack of **`char`**s.

To make a stack of **`double`**s one must:

- create a new module,
- a new stack struct, and
- functions implementing the new type of stack.

This is tedious, time-consuming, error-prone, and hard-to-maintain.

Modular Paradigm (con't)

Possible solutions to avoid having to rewrite the stack just to change the data type used:

- use object-based/oriented programming, or,
- use object-based/oriented programming with generic/template programming.

NOTE: These solutions do not make modular programming obsolete!

Data Abstraction

"*Data abstraction:* the idea of first providing a set of types suitable for an application area and then writing the program using those. [...] This is called 'abstraction' because a type is used through an interface [...]."
[4, §22.1.3, p. 781]

"Decide which types you want;
provide a full set of operations for each type."
[5, §2.5.2, p. 32]

Object-Based Paradigm

The **object-based** programming paradigm employs data structures called "objects" that are composed of state (i.e., represented by variables) and methods where object state is used to determine what is executed next.

- e.g., C++ code using `struct` or `class` or objects represented as modules is object-based.

Object-Oriented Paradigm

The **object-oriented** programming paradigm is object-based programming with **inheritance** added.

- e.g., C++ code using `struct` or `class` *employing inheritance* is object-oriented.

Inheritance enables expressing **hierarchical relationships** directly in code.

Object-Oriented Paradigm (con't)

"*Object-oriented programming:* the idea of organizing types into hierarchies to express their relationships directly in code."
[4, §22.1.3, p. 781]

"Decide which classes you want;
provide a full set of operations for each class;
make commonality explicit by using inheritance."
[5, §2.6.2, p. 39]

Generic Paradigm

"Decide which algorithms you want;
parameterize them so that they work for
a variety of suitable types and data structures."
[5, §2.5.2, p. 32]

Generic Paradigm (con't)

In C++ generics enable very powerful forms of **compile-time polymorphism**; exploiting them relies focusing on **algorithms** and **type abstractions** as **patterns**.

The **generic** programming paradigm allows code to be written using special placeholders representing as-yet unknown **types** and **constant values**. These placeholders are called **parameters**.

Generic code is turned into real code by **substituting** real types and constant values for each parameter. This process in C++ is called **template instantiation**.

Generic Paradigm (con't)

Since C++ is a **statically-typed** language, any and all use of generics **must be fully evaluated and determined at compile-time**.

In C++ generic use requires defining and using **`template`**s. Just as functions have arguments, templates have parameters.

However, function arguments are substituted at **run-time** whereas template parameters are **instantiated** (i.e., substituted) at **compile-time**.

Generic Paradigm (con't)

w01/generic1-eg.cxx

```
1  template <typename T>
2  T min(T a, T b)
3  {
4    return a < b ? a : b;
5  }
6
7  int main()
8  {
9    min(2,2); // OK: match for min(int,int)
10   min(2.3,13.1); // OK: match for min(double,double)
11   min(3.3,1); // ERROR: no match for min(double,int)
12 }
```

This code shows that `min<T>()` is a **pattern** using the (type) **placeholder** `T`. In `main()` the pattern is **instantiated** when `min()` is called with two arguments having **exactly** the same type.

Generic Paradigm (con't)

w01/generic2-eg.cxx

```
1  template <typename T>
2  struct ListNode
3  {
4    ListNode<T> *prev;
5    ListNode<T> *next;
6    T datum;
7  };
8
```

Generic Paradigm (con't)

w01/generic2-eg.cxx

```
9  int main()
10 {
11   // Generate ListNode where T is an int ...
12   ListNode<int> a;
13   a.prev = a.next = nullptr;
14   a.datum = 34;
15
16   // Generate ListNode where T is a ListNode<char> ...
17   ListNode< ListNode<int> > b;
18   b.prev = b.next = nullptr;
19   b.datum = a;
20 }
```

If writing C++98 code, replace **`nullptr`** with `0` (zero).

The code above shows that **`struct ListNode`** is a **pattern** using the (type) **placeholder** `T`. In `main()` the pattern is **instantiated** (twice) when suitable `T`s are **provided** to `ListNode<T>`.

Template Metaprogramming

The **template metaprogramming** programming paradigm is a generic programming paradigm where the representation of the generics is **Turing-complete** and such is being **exploited**.

C++ template metaprogramming **was discovered by accident** by Erwin Unruh and was found to be Turing-complete [7].

This means **any computation** that can be computed by a **computer program** can be performed using template metaprogramming.

Templates are **evaluated at compile-time** –not run-time!

Template Metaprogramming (con't)

w01/tmp-eg.cxx

```
1  template <bool Cond, typename T, typename F>
2  struct if_;
3
4  template <typename T, typename F>
5  struct if_<true,T,F>                  // Partial Specialization: True case
6  {
7    typedef T result;
8  };
9
10 template <typename T, typename F>
11 struct if_<false,T,F>                 // Partial Specialization: False case
12 {
13   typedef F result;
14 };
15
16 // If sizeof(short) == sizeof(int)
17 // then choose long, otherwise choose int.
18 typedef
19   typename if_<
```

Template Metaprogramming (con't)

```
20      sizeof(short) == sizeof(int), // TEST
21      long,                         // True result
22      int                           // False result
23    >::result,                      // "Return" of if_
24    MY_INT                          // typedef name
25  ;
26
27  int main()
28  {
29    MY_INT i;
30  }
```

Note that the above code **computes** the type of `MY_INT` at compile-time.

Functional Paradigm

The **functional** programming paradigm describes the evaluation of code using **pure functions**.

Template metaprogramming (and C++11's **`constexpr`**) functions utilize the functional paradigm.

The `<type_traits>` header provides a number of template metaprogramming operations.

That one can even use this paradigm in C++ was not by design.

Multiparadigm Programming Language

A programming language that allows one to freely **mix** multiple programming paradigms can be said to be a multiparadigm programming language.

C++ is a multiparadigm programming language.

The mixing of any of the aforementioned paradigms in C++ is **seamless** (i.e., no special keywords are required to mix them together in code).

Outline

Assumptions

Given this course's prerequisites, it is assumed:

- you already know how to program in Java, and,
- you already know how to program in C.

Assumptions (con't)

C is mostly a subset of C++ as most valid C programs are also valid C++ programs, however:

- not all valid C programs are valid C++ programs,
- in C++ the C Standard Library in the `std` namespace,
- for namespace and differences between C and C++, using a standard C include header requires:
  1. dropping the `.h` from its file name, and,
  2. prefixing a 'c' to the file name
  - For example, "`<stdio.h>`" becomes "`<cstdio>`" in C++.
  - Do **not** directly include a standard C header file in C++.
  - If you must include a C header in C++ not part of the C Standard Library that was not designed for use with C++ but whose contents can be compiled by a C++ compiler, then enclose such within an `extern "C"` block so all symbols within that header have C linkage.

Hello World!

The traditional minimal program that outputs “Hello World!” using the C Standard Library in C++ looks like:

w01/hello-1.cxx

```
1  #include <cstdio>
2
3  int main()
4  {
5    std::printf("Hello World!\n");
6  }
```

Hello World! (con't)

... and the version using the C++ Standard Library looks like:

w01/hello-2.cxx

```
1  #include <iostream>
2
3  int main()
4  {
5    std::cout << "Hello World!\n";
6  }
```

To compile this program, SSH into `cs3400.cs.uwindsor.ca` and run:

- `g++ -o hello hello-2.cpp`

Using the std Namespace

When first learning C++, you are best to tell the compiler to **search** the std **namespace** for all (as yet unresolved) **symbols** by writing "**using namespace std;**" at file scope **after** all include directives:

w01/hello-3.cxx

```
1  #include <iostream>
2
3  using namespace std;
4
5  int main()
6  {
7    cout << "Hello World!\n";
8  }
```

Doing so allows one to **avoid explicitly writing** std:: in front of all symbol names that are in the std namespace.

Using the std Namespace (con't)

As your C++ code skills improve, move your use of "**using namespace std;**" to smaller scopes (e.g., inside functions only). For example:

w01/hello-4.cxx

```
1  #include <iostream>
2
3  int main()
4  {
5    using namespace std;
6    cout << "Hello World!\n";
7  }
```

Amongst other things, this will help **reduce ambiguous symbol compiler errors**.

Using the std Namespace (con't)

As your skills further improve, avoid using "**using namespace std;**" in favour of using the **using** directive to tell the compiler where to look for specific symbols explicitly, e.g.,

w01/hello-5.cxx

```
1  #include <iostream>
2
3  int main()
4  {
5    // When cout is used w/o qualification, get it from std ...
6    using std::cout;
7    cout << "Hello World!\n";
8  }
```

Default Streams

C++ defines the following always-open stream objects in the `std` namespace:

Stream	Header	C Equivalent	Remarks
`cin`	`<istream>`	`stdin`	standard input
`cout`	`<ostream>`	`stdout`	standard output (buffered)
`cerr`	`<ostream>`	`stderr`	standard error (unbuffered)
`clog`	`<ostream>`	`stderr`	standard error (buffered)

Unlike their C equivalents, in C++ `cin`, `cout`, `cerr`, and `clog` are instances of the `std::istream` and `std::ostream` classes with additional capabilities.

IOStream Reading and Writing Operators

The capabilities of C++ stream classes are extensive. Their capabilities can be easily used using overloaded bit-shift operators:

- use **`operator`**>> to read data from a stream, and,
- use **`operator`**<< to write data to a stream.

The "Hello World!" program demonstrated output using `cout`.

For more information see [2, §15 to §15.3].

IOStream Reading and Writing Operators (con't)

To read in data, use `cin` and the `>>` bit-shift operator:

w01/cin-1.cxx

```
1  #include <iostream>
2  #include <string>
3
4  using namespace std;
5
6  int main()
7  {
8    int i; double d; string s;
9    cin >> i >> d >> s;
10   cout << s << ',' << i << ',' << d << '\n';
11 }
```

See [6, §4.3 and §38] and [2, §15].

IOStream Reading and Writing Operators (con't)

The C++ Standard defines many overloaded << and >> operators including:

- **bool**, **char**, **char*** and std::string
- **short** and **unsigned short**
- **int** and **unsigned int**
- **long** and **unsigned long**
- **long long** and **unsigned long long** (C++11)
- **float**, **double**, **long double** **void***
- std::string

I/O For User-Defined Types

Suppose you have a type called MYTYPE, then you can read it in by writing:

w01/read-in-1.cxx

```
1  #include <istream>
2  std::istream& operator >>(std::istream& is, MYTYPE& t)
3  {
4    // Code to read in type here ...
5    is >> t.whatever; // i.e., write necessary code here
6    return is;
7  }
```

I/O For User-Defined Types (con't)

... and write it out by writing:

w01/write-out-1.cxx

```
1  #include <ostream>
2  std::ostream& operator <<(std::ostream& os, MYTYPE const& t)
3  {
4    // Code to write out type here ...
5    os << t.whatever; // i.e., write necessary code here
6    return os;
7  }
```

Error Detection For IOStreams

If a stream is implicitly or explicitly **cast** to a `bool`, then the result is:

- `true` iff there have been no errors on the stream,
- `false` otherwise.

If **any** error occurs on a stream in C++, **no further I/O** occurs on that stream object **until** the error is cleared.

Error Detection For IOStreams (con't)

The stream errors are:

ios_base Constant	Member Function	Remarks
goodbit	good()	I/O works. Not an error.
eofbit	eof()	The end-of-file was encountered.
badbit	bad()	An operation failed and is unrecoverable.
failbit	fail()	An operation failed and is possibly recoverable.

See [2, §15.4] for the details.

Compiling and Linking Code

When using GNU GCC's g++ compiler, use the following options to compile your C++ code, e.g. if the source code is in `file.cxx` use:

ISO Level	g++ Command Line Options
C++98	`-std=c++98 -Wall -Wextra -Werror -Wold-style-cast file.cxx`
C++11	`-std=c++11 -Wall -Wextra -Werror -Wold-style-cast file.cxx`
C++14	`-std=c++14 -Wall -Wextra -Werror -Wold-style-cast file.cxx`
C++17	`-std=c++17 -Wall -Wextra -Werror -Wold-style-cast file.cxx`
C++20	`-std=c++20 -Wall -Wextra -Werror -Wold-style-cast file.cxx`
C++23	`-std=c++23 -Wall -Wextra -Werror -Wold-style-cast file.cxx`

NOTE: C++23 is not yet an official standard but has already been approved in 2023 and will be official very soon. Many compilers regularly release experimental support for upcoming standards.

Compiling and Linking Code (con't)

Additionally when linking C++ code compiled with GCC:

- Link with `-pthread` when using any of these headers:
  - `<thread>`, `<future>`, `<mutex>`, or `<condition_variable>`
- Link with `-latomic` when using `<atomic>`.
- Link with `-ltbb` when using parallel standard algorithms and the `<execution>` header.

Compiling and Linking Code (con't)

When using a debugger, remember to compile the code with debug symbols, e.g., use one of these compiler options:

- `-g`, `-g1`, `-g2`, `-g3`, `-ggdb`

NOTE: This course will use GCC version 13.2 or higher and will focus on C++ code conforming to the C++20 standard as well as newer upcoming C++ standard features.

Outline

`auto`: Variable Declarations

C++11 *redefined* the existing **`auto`** keyword from C/C++.

Prior to C++11, the **`auto`** keyword was used to declare variables with **automatic storage duration**, e.g., on the call stack. Since such was (and remains) the default storage duration for variables declared at block scope or in function parameter lists, **`auto`** was almost completely unused (in C and in C++).

auto: Variable Declarations (con't)

Starting in C++11, **auto** is used as a placeholder for a type:

- provided the type can be *deduced* by the compiler, and,
- the type can be deduced using the rules for template argument deduction.
  - Such is true in all cases when **auto** appears not as **decltype(auto)**. For example, when used to declare a variable [C++11] and when used as a return type [C++14].

Examples:

- `auto i = 5; // i's type deduced to be int`
- `auto const& j = i; // j's type is int const&`
- `auto k = 3.2F; // k's type is float`

auto: Variable Declarations (con't)

Know that Template Argument Deduction (TAD) type deduction is not simple overall, e.g., see [1, Template Argument Deduction]. The design of TAD's rules are to do the "right thing" (most of the time) —so one doesn't have to become a language expert to make use of such.

auto when used *must always be initialized* by an expression whose resulting type is the type that is deduced —or a compiler error results.

This is true for all uses of **auto** —not just variable declarations.

auto: Variable Declarations (con't)

Type-deduced **auto** can appear in many contexts:

- C++11: in a variable declaration
- C++11: as a function's return type provided the return type is specified as a suffix return type
- C++14: as a function's return type without a provided suffix return type
- C++14: as a function parameter type of a lambda function (implies the lambda function is a function template)
- C++17: as a template parameter for a non-type template parameter
- C++17: in structured binding declarations
- C++20: as a placeholder for a type after a concept constraint, e.g., as a function argument (implies the function is a function template)
- C++23: as a simple type specifier of an explicit type conversion, e.g., **auto**`(expr)` and **auto**`{expr}`.

auto: With and Without a return Type

auto is permitted to be used as the return type of a function —provided the function provided the return type at the end of the function "prototype" portion, e.g.,

w01/auto-suffix-return.cxx

```
1  auto func1() -> double; // function prototype w/suffix return
2
3  auto func2(int i, double d) -> float
4  {
5    return i + d;
6  }
```

auto: With and Without a return Type (con't)

Suffix return types were add to C++11 to allow code to be written that was extremely difficult or impossible to write, e.g.,

w01/func-template-suffix-return.cxx

```
1 template <typename T, typename U>
2 auto my_func(T t, U u) ->
3   decltype(t+u) // return type is whatever type t+u is
4 {
5   return t+u;
6 }
```

auto: With and Without a return Type (con't)

Lambda functions in C++11 have an **auto** return type with an optional suffix return type if the compiler can deduce the return type, e.g.,

w01/lambda1.cxx

```
1 [](int a, float f) // this is a lambda function with two parameters
2 {
3   return a + f + 5.3; // return type is deduced to be double
4 }
```

If there is no **return** statement, then the lambda return type is deduced to be **void**.

auto: With and Without a return Type (con't)

A suffix return type can be specified with a lambda function by writing such after the function parameter list, e.g.,

w01/lambda2.cxx

```
1 [](int a, float f) -> float // set return type to be float
2 {
3   // no type deduction for return type occurs
4   return a + f + 5.3; // double result is implicitly cast to float
5 }
```

auto: With and Without a return Type (con't)

Similarly to lambda functions in C++11, starting with C++14, **auto** is permitted to be used as the return type of any function *without* a specified suffix return type, e.g.,

w01/auto-return.cxx

```
1  auto func2(int i, double d)
2  {
3    return i + d; // deduced return type is double
4  }
```

decltype(auto)

decltype(auto) sometimes needs to be used instead of **auto**.

Instead of using template argument deduction rules to determine the type, **decltype(auto)** uses the type determined by **decltype(**`expression`**)** where `expression` is the expression associated with **decltype(auto)**.

`decltype(auto)` (con't)

`decltype(expression)` Type

Given an expression, E, associated with an **`decltype(auto)`** variable, or, **`decltype(auto)`**-return with no suffix return type, the type is determined as the resulting type of E.

`decltype(auto)` (con't)

Template Argument Deduction v. `decltype(auto)`

Template argument deduction (TAD) differs from **`decltype(auto)`** as follows:

- TAD is a deductive process; **`decltype(auto)`** is not
- with TAD, if the deduced type is a forwarding reference (i.e., an rvalue reference), the deduced type will be adjusted from an rvalue reference to be a value type (i.e., a type with no reference), e.g., T&& is changed to T

In some special cases one will wish to preserve the **exact** type of an expression requiring the use of **`decltype(expression)`** or **`decltype(auto)`**. (More will be said later in the course when this is relevant. Most of the time, one will not need to use this.)

Outline

Storage Durations

In C++ an object's storage duration also determines its linkage.

Storage Durations (con't)

Automatic Storage Duration

Refers to objects that are allocated at the start of the enclosing code block it is defined in and deallocated at the end of that block.

All local objects, except those declared **`static`**, **`extern`**, or **`thread_local`**, have automatic storage duration.

Also all objects declared **`register`** have automatic storage duration. **`register`** hints to the compiler to store the variable in a CPU register.

- **`register`** hints to the compiler to store the variable in the processor's register if possible
- **`register`** was deprecated in C++17
  - Why? Compilers do a much better job of assigning variables to registers than humans do. Most compilers ignored the use of **`register`** because of this.

Storage Durations (con't)

Static Storage Duration

Static storage duration is for objects allocated when the program begins and deallocated when the program ends. If this duration applies to an object, only one instance of the object will exist.

All objects declared at **`namespace`** scope including the global namespace as well as those declared with **`static`** or **`extern`** have this storage duration.

All **`static`**ally declared objects have *internal* linkage, unless they are **`static`** class data members *not* in an anonymous **`namespace`** (in which case they have *external* linkage).

Storage Durations (con't)

Thread Storage Duration

Thread storage duration is for objects allocated when the thread begins and deallocated when the thread ends. If this duration applies to an object, each thread has its own instance of the object.

Only objects declared with the **`thread_local`** keyword have this storage duration.

All **`thread_local`** variables are implicitly **`static`**.

`thread_local` can be used with **`static`** or **`extern`** to affect linkage.

Can only be used with objects declared at namespace, block, and static data members.

Storage Durations (con't)

Dynamic Storage Duration

Dynamic storage duration is for objects allocated and deallocated upon request using dynamic memory allocation/deallocation functions, e.g., **`new`** and **`delete`**.

Linkage

Linkage determines how one can or cannot refer to names within a program's translation units as well as to other programs'/library names in their respective modules/translation units.

Linkage (con't)

No Linkage

A name that can only be referred to from the scope it is in.

These names defined at block scope have no linkage:

- variables not explicitly declared `extern`
- local classes and their member functions
- all other names declared at block scope

All names not specified as having external, module, or internal linkage have no linkage.

Linkage (con't)

Internal Linkage

A name that can only be referred to from all scopes in the current translation unit.

These names declared at namespace scope have internal linkage:

- variables, variable templates [C++14], functions, **`static`** function templates
- data members of anonymous `unions`
- non-**`volatile`** non-**`template`** non-**`inline`** non-`exported` **`const`**-qualified/**`constexpr`** variables not declared **`extern`** and don't otherwise have external linkage

All names in unnamed **`namespace`**s or within namespaces within an unnamed **`namespace`** even if explicitly declared **`extern`**, have internal linkage.

Linkage (con't)

External Linkage

A name that can be referred to from the scopes in other translation units.

Variables and functions with external linkage can be linked to from other programming languages' translation units.

Unless in an unnamed `namespace`, any of the following names declared at `namespace` scope:

- functions and function templates not declared `static`
- non-`const` variables not declared `static`
- variables declared `extern`
- enumerations
- class names, member functions, `static` data members, nested classes, enumerations, functions first introduced via a `friend` declaration inside a class

If first declared at block scope, all variables declared `extern` and all names of functions.

Linkage (con't)

Module Linkage

Names that can only be referred to from scopes in the same module unit or in other translation units of the same named module.

All names declared at namespace scope have module linkage if their declarations are attached to a named module, are not **`export`**ed, and don't have internal linkage.

(This linkage was created to implement C++20 modules.)

Outline

Digit Separators

C++14 allows all *integer* and *floating-point* literal values to use single quotation marks as digit separators that are *ignored* by the C++ compiler.

The purpose of such is to help human readability of such numbers.

Such separators need not be every three digits —the programmer can choose where to put them.

w01/digit-separator.cxx

```
1 auto d = 42'000'000; // decimal
2 auto o = 042'00'00'00; // octal
3 auto x = 0x9FC2'FACE'CAFE; // hexadecimal
4 auto b = 0b1110'1101'1101'1000'0111'0110; // binary (C++14)
5 auto fp1 = 3.141'592'653'589'793'238'462'643'383'279'50; // floating-point
6 auto fp2 = 2.99'792'458e8; // floating-point, decimal, scientific notation
7 auto fp3 = 0x1FF'0B21p734; // floatiing-point, hexadecimal (C++17)
```

Outline

C++ Language Literal Prefixes and Suffixes

The C++ language defines a number of prefixes and suffixes associated with integer and floating-point literal values:

Literal	Std	Kind	Type	Description
`0b`	C++14	prefix	integer	is binary (base 2)
`0`		prefix	integer	is octal (base 8)
`0x`, `0X`		prefix	integer	is hexadecimal (base 16)
`l`, `L`		suffix	`long int`	
`ul`, `UL`		suffix	`unsigned long int`	
`lu`, `LU`		suffix	`unsigned long int`	
`ll`, `LL`		suffix	`long long int`	
`llu`, `LLU`		suffix	`unsigned long long int`	

C++ Language Literal Prefixes and Suffixes (con't)

Literal	Std	Kind	Type	Description
`0x`, `0X`	C++17	prefix	floating-point	hexadecimal mantissa
`f`, `F`		suffix	`float`	
`L`, `L`		suffix	`long double`	
`bf16`, `BF16`		suffix	`std::bfloat16_t`	"brain float" format
`f16`, `F16`		suffix	`std::float16_t`	half precision
`f32`, `F32`		suffix	`std::float32_t`	single precision
`f64`, `F64`		suffix	`std::float64_t`	double precision
`f128`, `F128`		suffix	`std::float128_t`	quad precision

Note: C++17 hexadecimal floating-point numbers use a "p" to separate the mantissa and exponent portions —not an "e". (The mantissa is in hexadecimal and the exponent is an integer in base 10.)

C++ Language Literal Prefixes and Suffixes (con't)

The C++ language also defines a number of prefixes associated with string literal values:

Literal	Std	Description
L		wide string (`wchar_t`)
u8	C++11	UTF-8 string (`char`; `char8_t` in C++20)
u	C++11	UTF-16 string (`char16_t`)
U	C++11	UTF-32 string (`char32_t`)

+ raw string literals (see next slide)...

C++ Language Literal Prefixes and Suffixes (con't)

C++11 defined a new kind of string literal: a raw string literal:

- all other string literals use the \ character as an escape character
- raw string literals do *not* support the escaping of characters, hence they are raw since they are used as-is

C++ Language Literal Prefixes and Suffixes (con't)

Raw string literals:

- start with `R"delim(`,
- then the characters in the string appear, and,
- end with `)delim`
- where `delim` is a possibly empty delimeter token that does *not* appear in within the string literal

w01/raw-string-lit.cxx

```
1  // NOTE: None of the raw strings below have any escape characters in them.
2  //       Everything in the raw string is as-is.
3  char const* s1 = R"(This some as-is text.\nNo newlines here!)";
4  wchar_t const* s2 = LR"(Some more \t\v\n\r as-is text.)";
5  char8_t const* s3 = u8R"nice(Another raw string \x0A\x0D as-is.)nice";
6  char16_t const* s4 = uR"okay(More stuff.)okay";
7  char32_t const* s5 = UR"qwerty(Even more stuff.)qwerty";
```

C++ Language Literals

C++11 also allows the integer, floating-point, character, and string literals to produce objects of user-defined type by using a user-defined suffix.

- C++ reserves all literal suffixes *that do not start with an underscore*.
- All user-defined literal suffixes *must start with an underscore*.

Such suffixes are defined by defining an operator overload of the literal operator, e.g.,

w01/userdef-lit1.cxx

```
1  struct si_metre  { long double value; }; // type representing the SI unit metre
2  si_metre operator""_mm(long double value) { return si_metre{value / 1000.L}; }
3  si_metre operator""_cm(long double value) { return si_metre{value / 100.L}; }
4  si_metre operator""_m(long double value) { return si_metre{value}; }
5  si_metre operator""_km(long double value) { return si_metre{value * 1000.L}; }
6
7  si_metre m1 = 29.5_mm; // 29.5 / 1000 so it is in metres
8  si_metre m2 = 55.321_cm; // 55.321 / 100 so it is in metres
9  si_metre m3 = 22.1234e17_m; // 22.1234e17 metres
10 si_metre m4 = 3156.23e4_km; // 3156.23e4 * 1000 so it is in metres
```

C++ Language Literals (con't)

`operator""` can have one parameter if it is one of these types:

- `char const*`
- `char`, `wchar_t`, `char8_t` (C++20), `char16_t`, `char32_t`
- `unsigned long long int`
- `long double`

or two parameters with any of these types:

- `(char const*, std::size_t)`
- `(wchar_t const*, std::size_t)`
- `(char8_t const*, std::size_t)` (C++20)
- `(char16_t const*, std::size_t)`
- `(char32_t const*, std::size_t)`

C++ Language Literals (con't)

Additionally:

- `operator""` definitions for integer and floating-point literals may be function templates where the `char` values in the literal are passed to the function template
  - To process such requires writing advanced "template metaprogramming" code.
- `operator""` definitions for string literals are permitted to be function templates if the function template parameter is a non-type template parameter of class type
  - The string literal is then passed to the `constexpr` constructor of the (structural) class type, e.g., see the example on the next slide.
- otherwise `operator""` should return the type it is responsible for constructing.

C++ Language Literals (con't)

For those curious about advanced uses of **operator""** function templates for string literals:

w01/userdef-lit2.cxx

```
#include <algorithm>  // for std::copy_n
#include <cstddef>    // for std::size_t

template <std::size_t N>
struct crazy_string_literal_type
{
  char value[N];
  constexpr crazy_string_literal_type(char const (&str)[N])
  {
    std::copy_n(str,N,value);
  }
};
template <crazy_string_literal_type c>
constexpr auto operator""_hmmm() { return c; }

auto cool = "This is some text."_hmmm;
```

C++ Language Literals (con't)

C++ defines a number of literal suffixes in the Standard Library that are useful in programs:

Literal	Std	Class	Description
`if`	C++14	`std::complex<float>`	complex imaginary number
`i`	C++14	`std::complex<double>`	complex imaginary number
`il`	C++14	`std::complex<long double>`	complex imaginary number

Defined in namespace: `std::literals::complex_literals`

C++ Language Literals (con't)

Literal	Std	Class	Description
`y`	C++20	`std::chrono::duration`	years
`d`	C++20	`std::chrono::duration`	days
`h`	C++14	`std::chrono::duration`	hours
`min`	C++14	`std::chrono::duration`	minutes
`s`	C++14	`std::chrono::duration`	seconds
`ms`	C++14	`std::chrono::duration`	milliseconds
`us`	C++14	`std::chrono::duration`	microseconds
`ns`	C++14	`std::chrono::duration`	nanoseconds

Defined in namespace: `std::literals::chrono_literals`

C++ Language Literals (con't)

Literal	Std	namespace	Class
s	C++14	`std::literals::string_literals`	`std::string`
sv	C++17	`std::literals::string_view_literals`	`std::string_view`