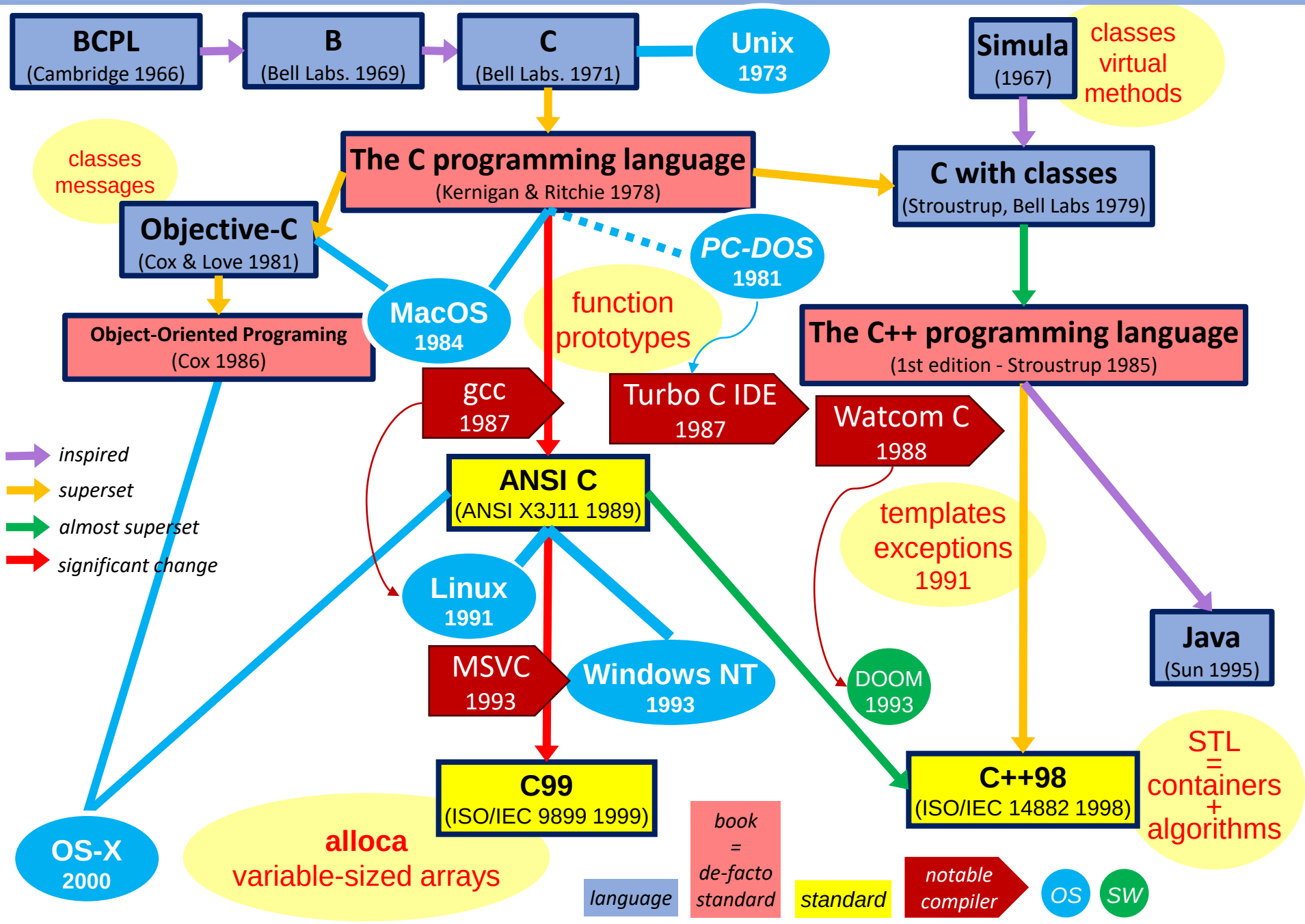
# Programming in C++

David Bednárek

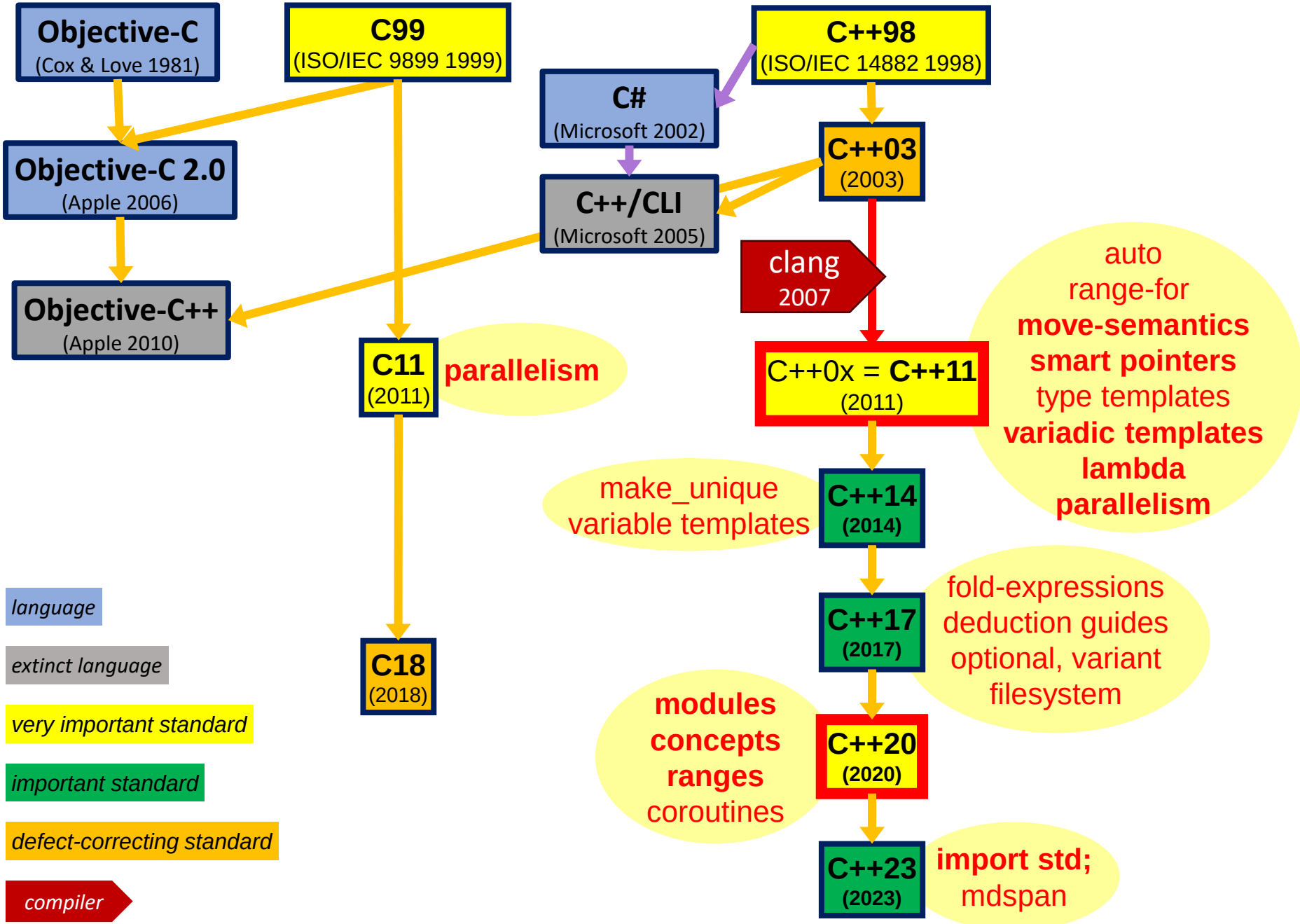
2022/2023

# History and Literature

# Ancient history of C and C++



# Modern history of C++ and related languages



# Books

- http://stackoverflow.com/questions/388242/the-definitive-c-book-guide-and-list
- Be sure that you have (at least) the C++11 versions of the books

- Introduction to programming (using C++)
  - Stanley B. Lippman, Josée Lajoie, Barbara E. Moo: C++ Primer (5th Edition)
    - Addison-Wesley 2012 (976 pages)
  - Bjarne Stroustrup: Programming: Principles and Practice Using C++ (2nd Edition)
    - Addison-Wesley 2014 (1312 pages)
- Introduction to C++
  - Bjarne Stroustrup: A Tour of C++ (2nd Edition)
    - Addison-Wesley 2018 (256 pages)
- Reference
  - Bjarne Stroustrup: The C++ Programming Language - 4th Edition
    - Addison-Wesley 2013
  - Nicolai M. Josuttis: The C++ Standard Library: A Tutorial and Reference (2nd Edition)
    - Addison-Wesley 2012

# Books

- http://stackoverflow.com/questions/388242/the-definitive-c-book-guide-and-list
- Be sure that you have the C++11 versions of the books

- Best practices
  - Scott Meyers: Effective Modern C++
    - O'Reilly 2014 (334 pages)
- Advanced [not in this course]
  - David Vandevoorde, Nicolai M. Josuttis, Douglas Gregor: C++ Templates: The Complete Guide (2nd Edition)
    - Addison-Wesley 2017 (832 pages)
  - Anthony Williams: C++ Concurrency in Action: Practical Multithreading
    - Manning Publications 2012 (528 pages)

- On-line materials
  - Bjarne Stroustrup, Herb Sutter: C++ Core Guidelines
    - github.com/isocpp/CppCoreGuidelines
  - Nate Kohl et al.: C++ reference [C++98, C++03, C++11, C++14, C++17, C++20]
    - cppreference.com

# The C++ Programming Language

- C/C++ can live alone
  - No need for an interpreter or JIT compiler at run-time
  - Run-time support library contains only the parts really required
  - Restricted environments may run with less-than-standard support
    - Dynamic allocation and/or exceptions may be stripped off
    - Code may work with no run-time support at all
  - Compilers allow injection of system/other instructions within C/C++ code
    - Inline assembler or intrinsic functions
  - Code may be mixed with/imported to other languages

- There is no other major language capable of this
  - All current major OS kernels are implemented in C
    - C was designed for this role as part of the second implementation of Unix
    - C++ would be safer but it did not exist
  - Almost all run-time libraries of other languages are implemented in C/C++
    - If C/C++ dies, all the other languages will die too

- C/C++ is fast
  - Only FORTRAN can currently match C/C++
  - C++ is exactly as fast as C
    - But programming practices in C++ often trade speed for safety
- Why?
  - The effort spent by FORTRAN/C/C++ compiler teams on optimization
    - 40 years of development
  - Strongly typed language with minimum high-level features
    - No garbage-collection, reflexion, introspection, …
  - The language does not enforce any particular programming paradigm
    - C++ is not necessarily object-oriented
  - The programmer controls the placement and lifetime of objects
  - If necessary, the code may be almost as low-level as assembly language
- High-Performance Computing (HPC) is done in FORTRAN and C/C++
- python/R/matlab may also work in HPC well…
  - …but only if most work is done inside library functions (implemented in C)

# Major features specific for C++
# (compared to other modern languages)

# Major distinguishing features of C++ (for beginners)

- Archaic text-based system for publishing module interfaces
  - Will be (gradually) replaced by true modules defined in C++20
    - All major compilers (as of 2023) implement the modules in the language
    - The standard library implementations are not yet ready for the module interface
- No 100%-reliable protections
  - Programmer's mistakes may result in uncontrolled crashes
  - Hard crashes (invalid memory accesses) cannot be caught as exceptions
    - Some compilers can do it in some cases
- Preference for value types
  - Similar to old languages, unlike any modern (imperative) language
  - Objects are often manipulated by copying/moving instead of sharing references to them
  - No implicit requirement for dynamic allocation
- No garbage collector
  - Approximated by smart pointers since C++11
    - Safety still dependent on programmer's discipline

# Major distinguishing features of C++ (for beginners)

- C makes it easy to shoot yourself in the foot;
  C++ makes it harder, but when you do it blows your whole leg off.
  - Bjarne Stroustrup, creator of C++

## java/C#/...

```
void f(/*...*/)
{
   T v = new T(/*...*/);
       // v is a reference

   do_it(v);
       // the reference is passed
}
```

- Do we really need dynamic allocation here?
  - Probably not, but...
  - ... what if do_it stores a copy of the reference somewhere
- Programmers don't care
  - The language enforces the use of **new**
  - Advanced compilers (escape analysis) may sometimes detect that dynamic allocation is not needed
    - The code is then converted into an equivalent of the C++ value style

## modern C++

- Value-based approach

```
void f(/*...*/)
{
   T v(/*...*/);
       // v is the object

   do_it(v);
       // usually passed by reference
}
```

- do_it shall not store the reference to **v** anywhere
  - if it does, the program will probably crash later
  - see *"Shooting in one's foot"*
- C++ conventions include this:
  - *If an object is passed by reference to a function, the function must stop using the reference upon its exit*
  - technically, do_it *can* store the reference (e.g. in a static variable), but it requires ugly code

## java/C#/...

```
void f(/*...*/)
{
   T v = new T(/*...*/);
       // v is a reference

   do_it(v);
       // the reference is passed
}
```

- Do we really need dynamic allocation here?
  - Probably not, but...
  - ... what if do_it stores a copy of the reference somewhere
- Programmers don't care
  - The language enforces the use of **new**
  - Advanced compilers (escape analysis) may sometimes detect that dynamic allocation is not needed
    - The code is then converted into an equivalent of the C++ value style

## modern C++

- Smart pointers

```
void f(/*...*/)
{
    auto v =
        std::make_unique<T>(/*...*/);
        // v is a smart pointer

   do_it(std::move(v));
        // ownership of the object
        // transferred to do_it
}
```

- If we really need to store a reference to **v** forever
  - Dynamic allocation required
  - Wrapped into smart-pointers
- Passing smart pointers around often requires special syntax
  - It acts as a warning to readers
  - It is far more complex than java etc.

## Value-based approach

- Suitable function declaration

```
void do_it(T & p);
```

  - or

```
void do_it(const T & p);
```

- Usage

```
void f(/*...*/)
{
   T v(/*...*/);
      // v is the object

   do_it(v);
      // usually passed by reference
}
```

- C++ conventions include this:
  - *If an object is passed by reference to a function, the function must stop using the reference upon its exit*
- This is NOT enforced by the language itself
  - technically, do_it *can* store the reference (e.g. in a static variable), but it requires unusual code

```
T * g = nullptr;
void do_it(T & p) { g = &p; }
```

## Smart pointers

- Suitable function declaration

```
void do_it(std::unique_ptr<T> p);
```

  - or

```
void do_it(std::unique_ptr<T> && p);
```

- Usage

```
void f(/*...*/)
{
    auto v =
       std::make_unique<T>(/*...*/);
       // v is a smart pointer

   do_it(std::move(v));
       // ownership of the object
       // transferred to do_it
}
```

- Passing smart pointers around often requires special syntax
  - std::move(v), &*v, etc.
  - It acts as a warning to readers
- There are other smart pointers
  - std::shared_ptr<T>
- There are *observer* pointers
  - T *
  - const T *

## java/C#/...

- **Programmers don't care** about the lifetime of objects
  - The have no choice anyway
  - Advanced compilers may optimize
- *Shouldn't a programmer have an idea of what will happen to their object?*

## modern C++

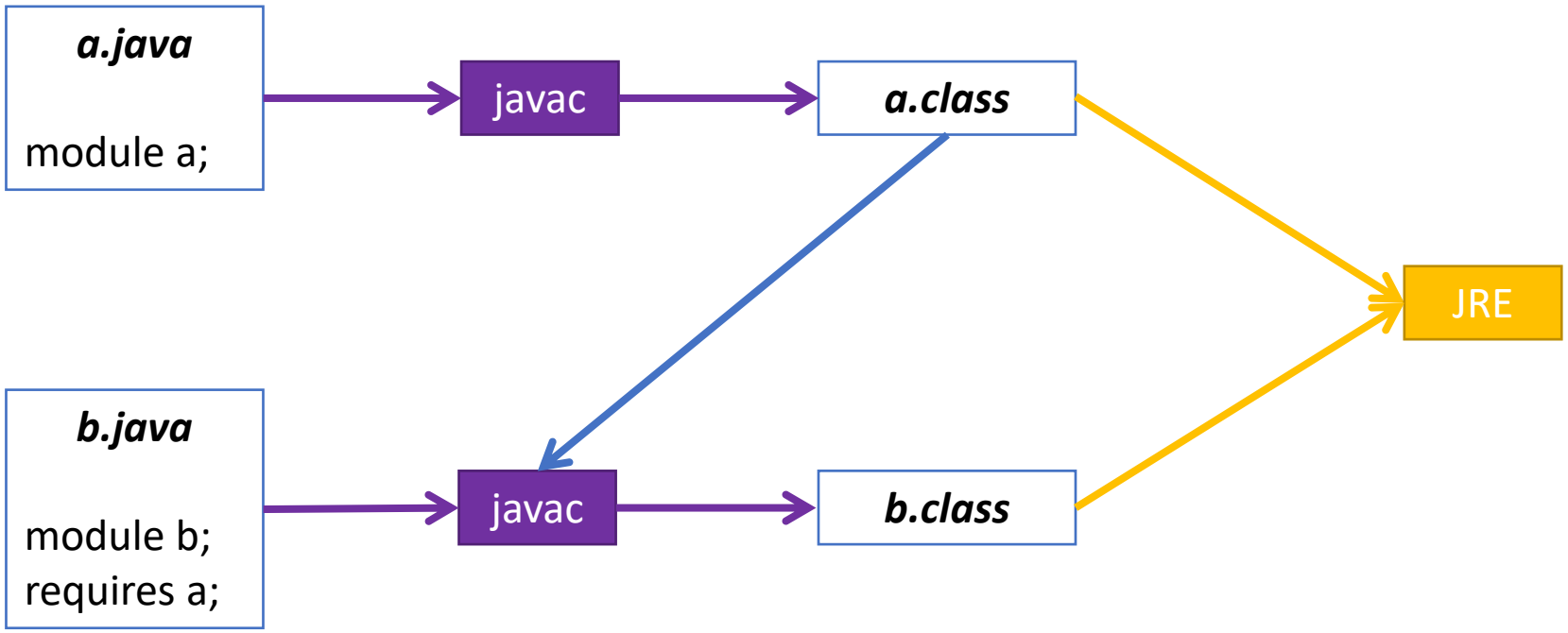
- **Programmers must think** about the lifetime of objects
  - It kills beginners
  - It helps in large projects
- You have to select from a variety of pointer/reference types
- You sometimes have to use some operators when passing pointer/references around
- This acts as a documentation!
  - If you adhere to conventions
- Details later…

# Major distinguishing features of C++ (for advanced programmers)

- User-defined operators
  - Pack sophisticated technologies into symbolic interfaces
  - C and the standard library of C++ define widely-used conventions
- Extremely strong generic-programming mechanisms
  - Turing-complete compile-time computing environment for meta-programming
  - No run-time component – zero runtime cost of being generic
- C++ is now more complex than any other general programming language ever created

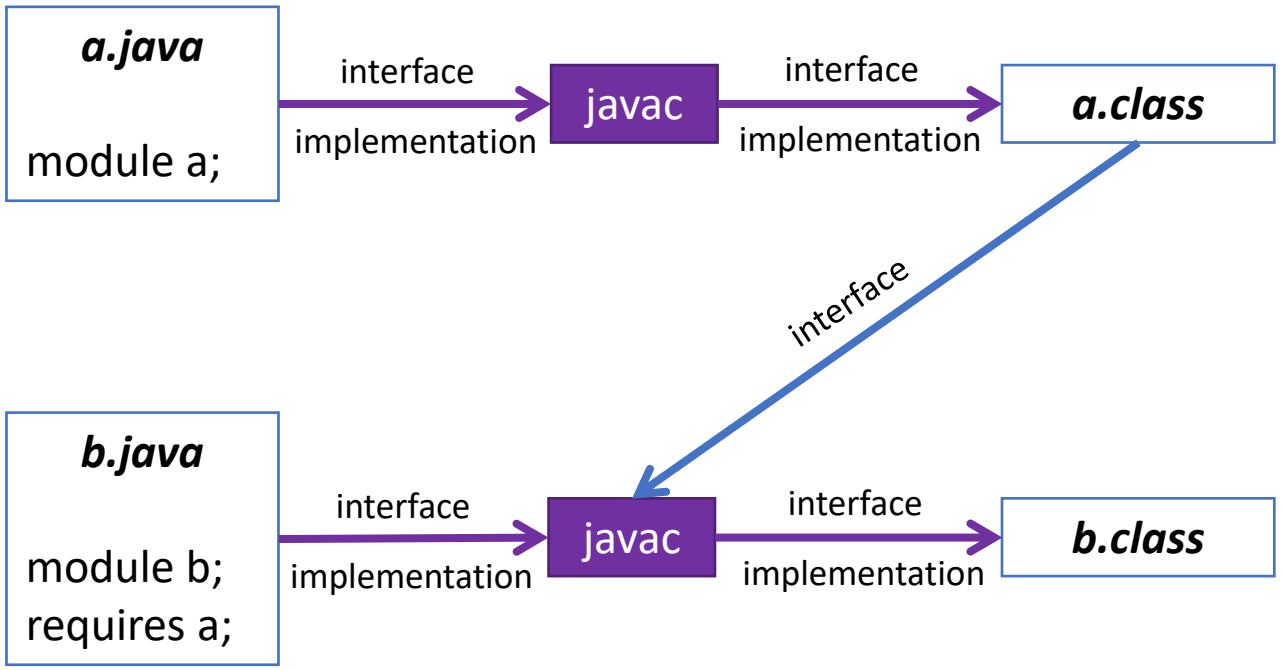
# Programming languages and compilers

# Compilation in modern languages



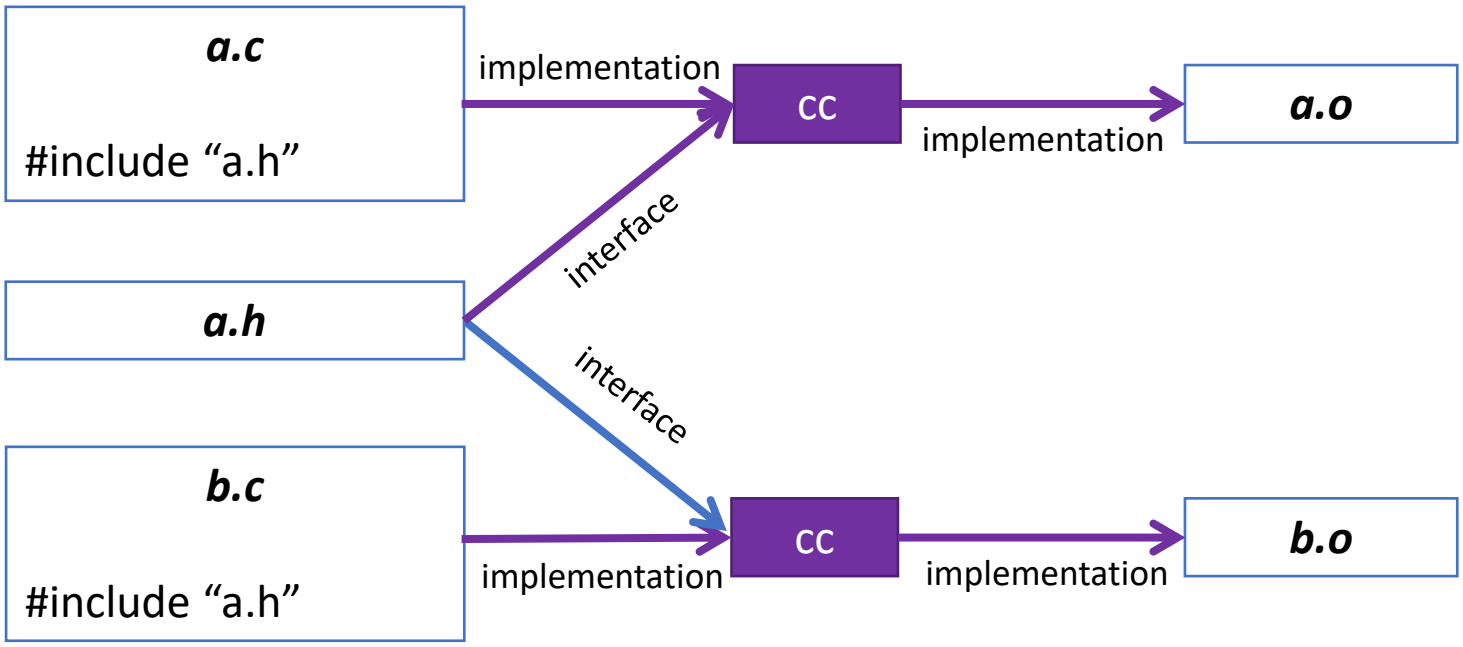

- Compilers produce binary packages from source code
  - These packages are also read by the compiler when referenced
    - All languages created after 1990 use something like import/require clauses
  - But not in C/C++ before C++20
    - C++20 has modules and module interfaces, more complex than in java

# Compilation in modern languages



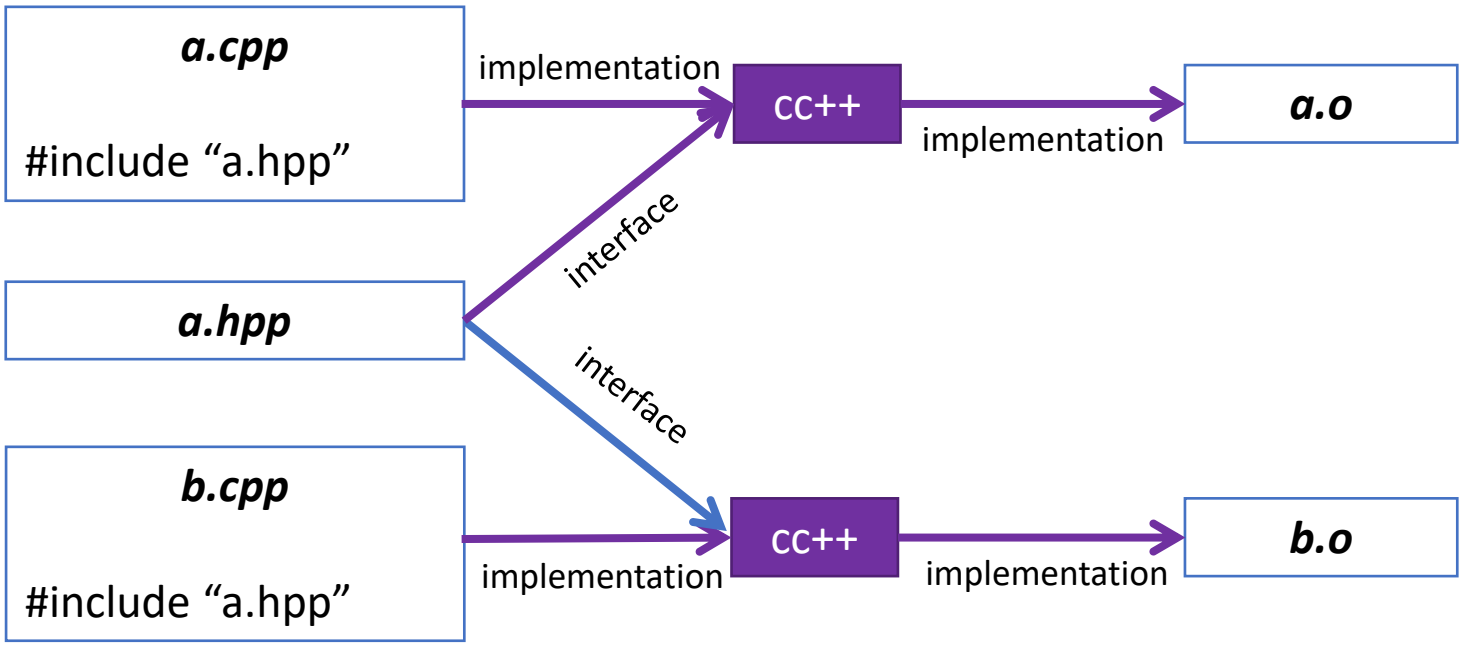

- Why not in C/C++? There are disadvantages:
  - When anything inside a.java changes, new timestamp of a.class induces recompilation of b.java
    - Even if the change is not in the public interface
  - How do you handle cyclic references?

# Compilation in C



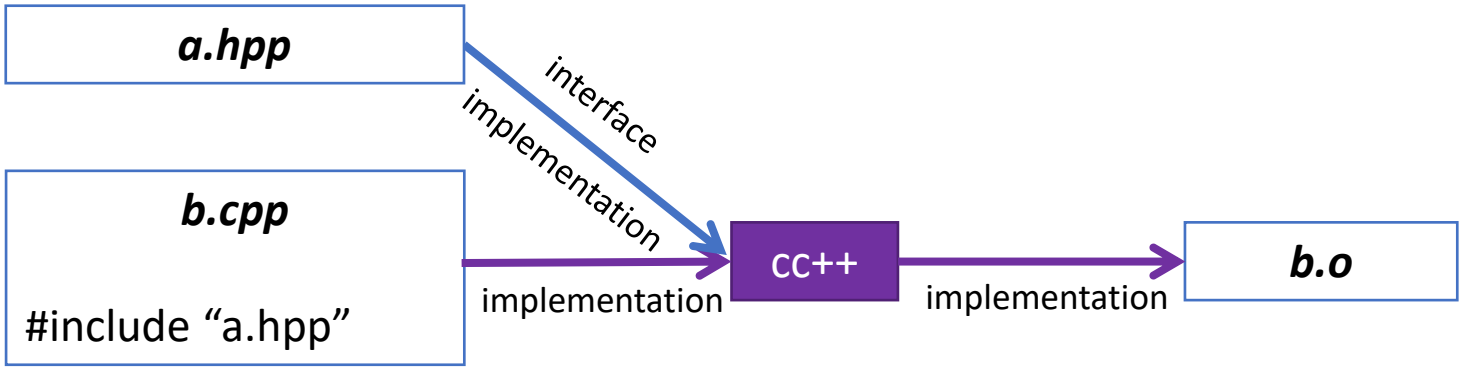

- In C, the situation was simple
  - Interface = function headers in „header files“
    - Typically small
  - Implementation = function bodies in “C files”
    - Change of a.c does not require recompilation of b.c

# Compilation in C++ (before C++20 modules)



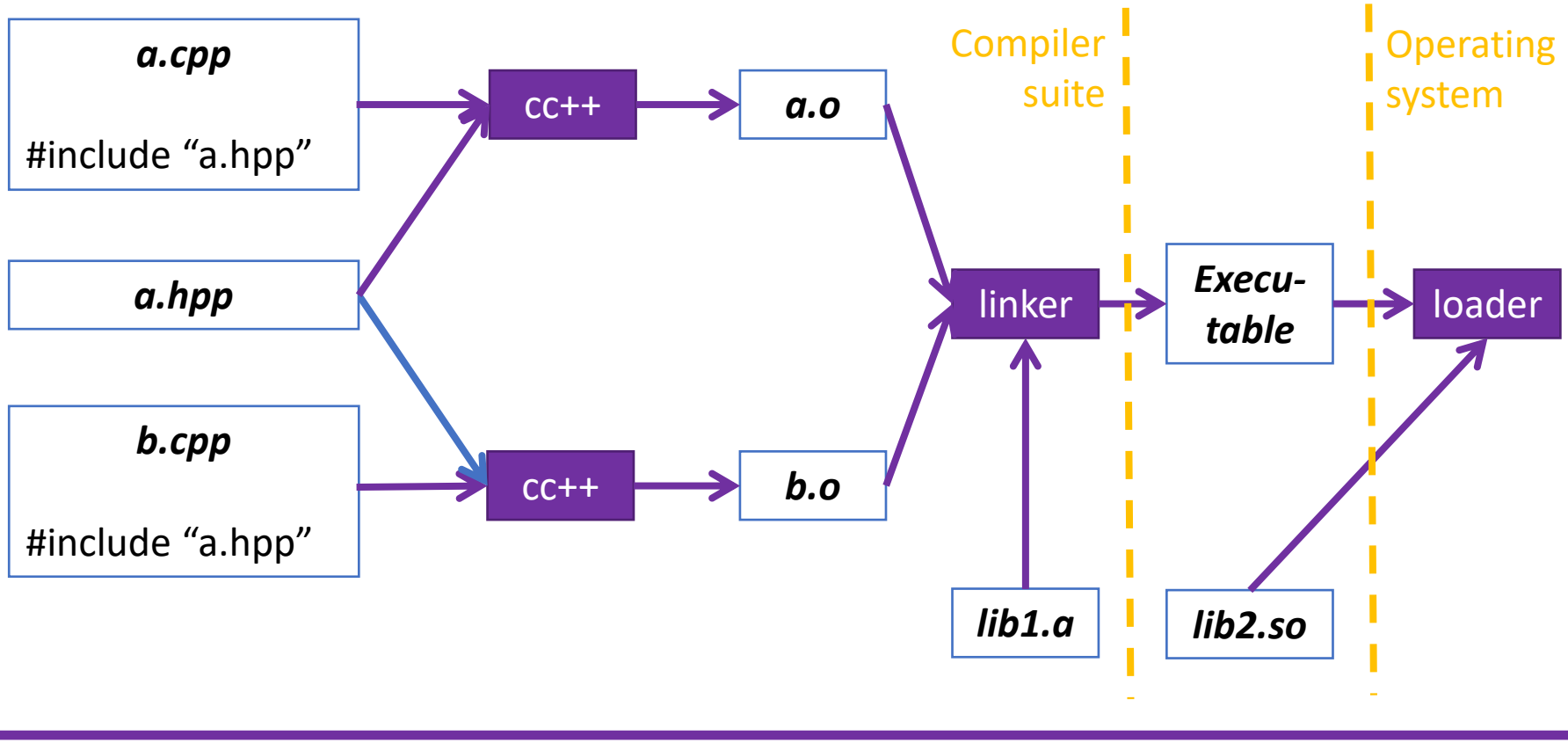

- In modern C++, the separate compilation is no longer an advantage
  - Interface (classes etc.) is often larger than implementation (function bodies)
  - Changes often affect the interface, not (only) the body
- The purely textual behavior of #include is anachronism

# Compilation of generic code in C++ (before C++20 modules)



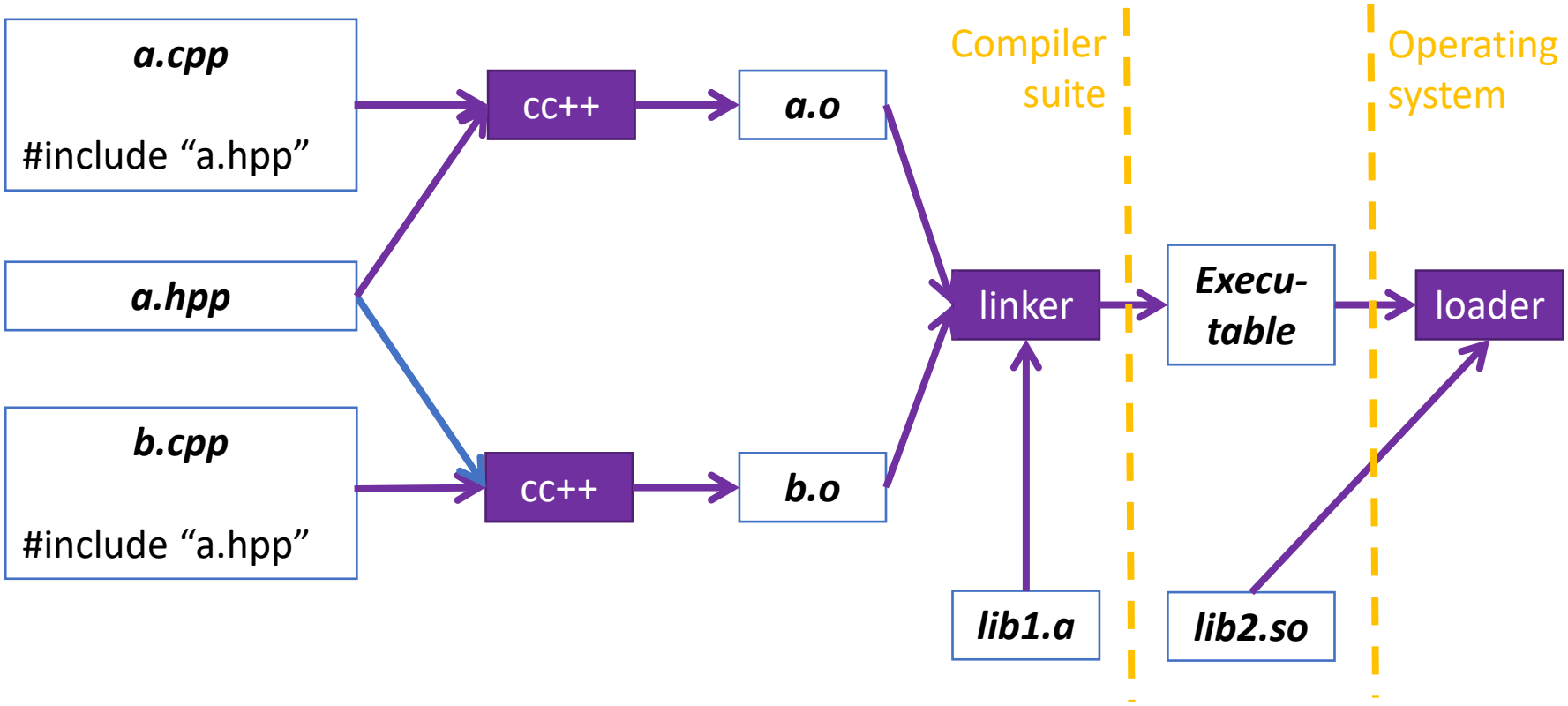

- Implementation of generic functions (templates) must be visible where called
  - Explanation later…
- Generic code often comprises of header files only

# Compilation in C++ (before C++20 modules)



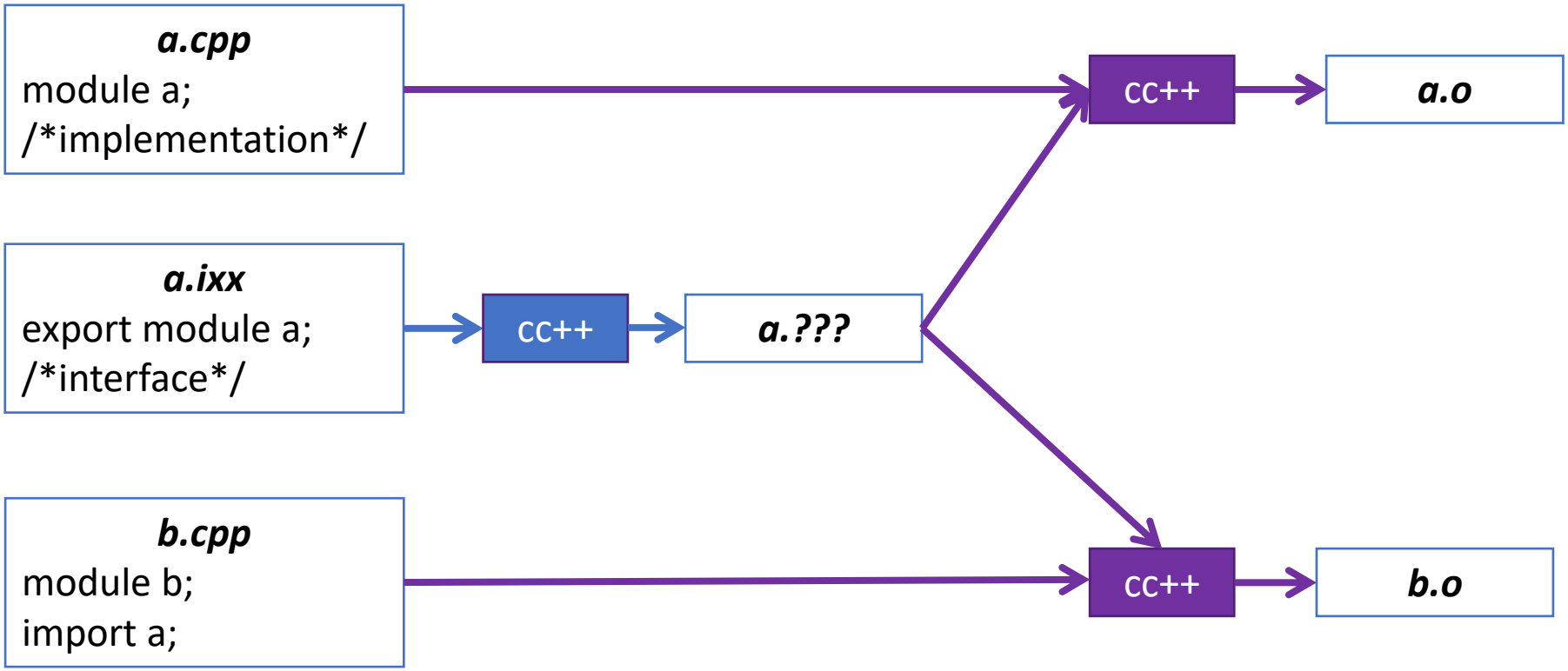

- Object files (.o, .obj) contain binary code of target platform
  - They are incomplete – not executable yet
- Linker/loader merges them together with library code
  - Static/dynamic libraries. Details later…

# Compilation in C++ (before C++20 modules)




- The (contents of) a .o [unix] or .obj [windows] file is called a module
  - also applied to the corresponding .c or .cpp file
  - one module = one independent run of the compiler
    - if more .cpp files specified at compiler command-line, they are still independent
- This is related but not the same meaning as in C++20 modules

# Compilation with C++20 modules (preview)




- Problems
  - The files can no longer be compiled in arbitrary order
  - New build system required
    - Module interface files must be compiled before module implementation files
      - The suffix .ixx of *module interface files* is Microsoft-specific solution of this problem
    - There may be dependences between different *module interface files*