Back to Basics: The Structure of a Program

Bob Steagall
CppCon 2020

KEWB COMPUTING

Overview

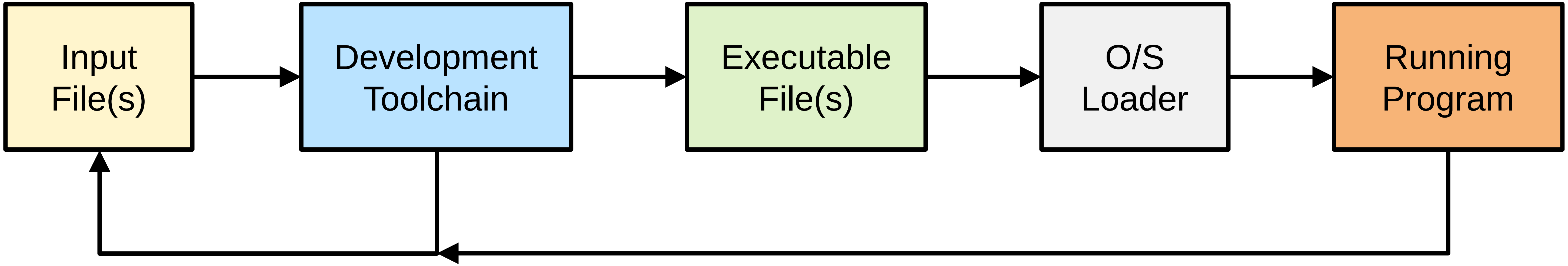
- The process of building a program
- What a translation unit (TU) is, and its relationship to the code you write
- The phases of translation
- Declarations, definitions, and linkage
- The one-definition rule (ODR)
- Storage duration
- ABIs and name-mangling
- Linking and loading

Goals

- Describe a few important "global" concepts
- Clarify terminology that can be confusing
- Shine some light on how C++ compilation works

Building and Running a C++ Program

- 50,000 ft view



Input Files

- User-Defined Code
  - Header files (.h, .hpp, .hh, etc.)
  - Source files (.c, .cpp, .cxx, .C, etc.)
  - Resource files (.res, .qrc, .rcc, etc.)

- Dependencies (libraries)
  - Header files (<vector>, <boost/text/text.hpp>)
  - Precompiled files (libstdc++.a, libc++.so, msvcrt.lib, ws2_32.dll, crt1.o)
  - Source files (Boost, Catch2)
  - Resources (icons, images, translations)

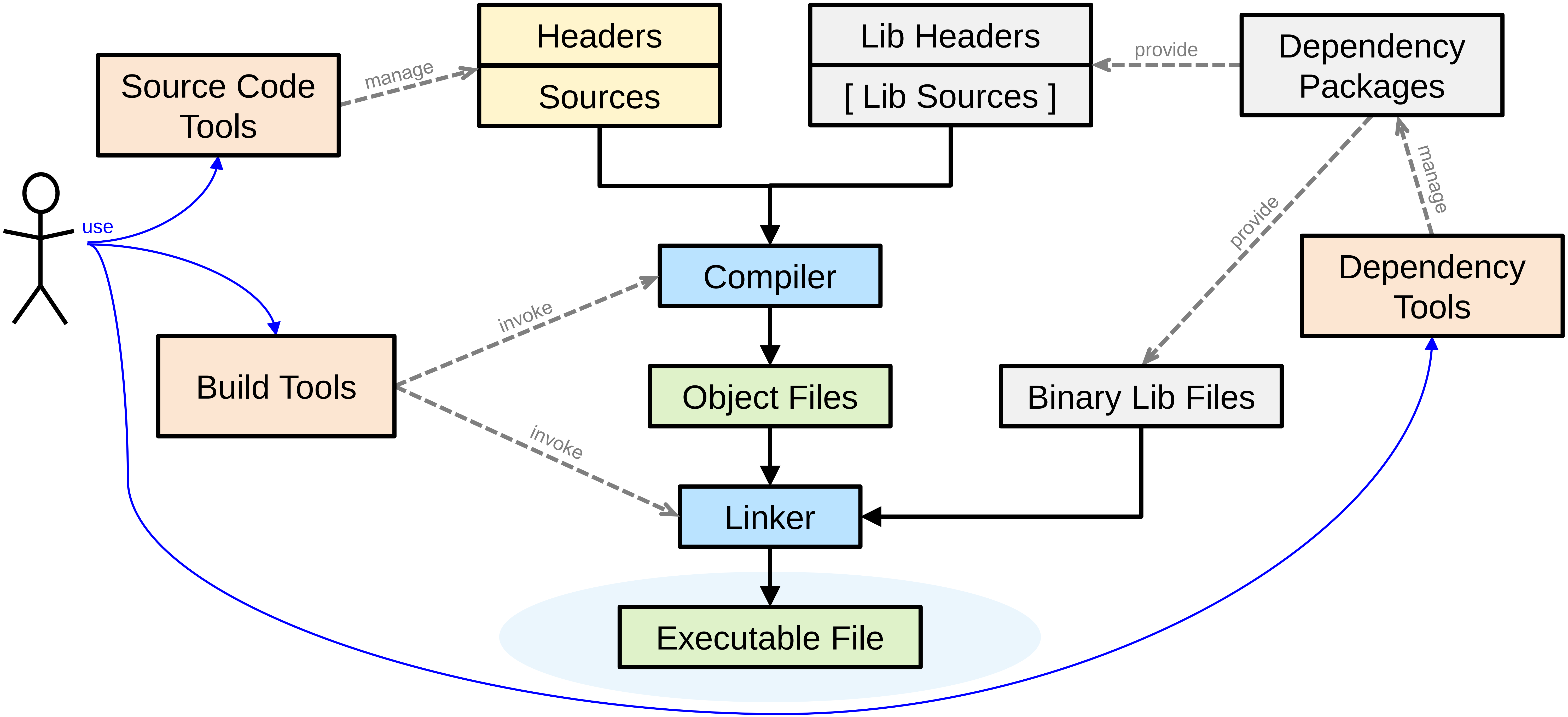
Input Files

- User-Defined Code
  - Header files (.h, .hpp, .hh, etc.)
  - Source files (.c, .cpp, .cxx, .C, etc.)
  - Resource files (.res, .qrc, .rcc, etc.)

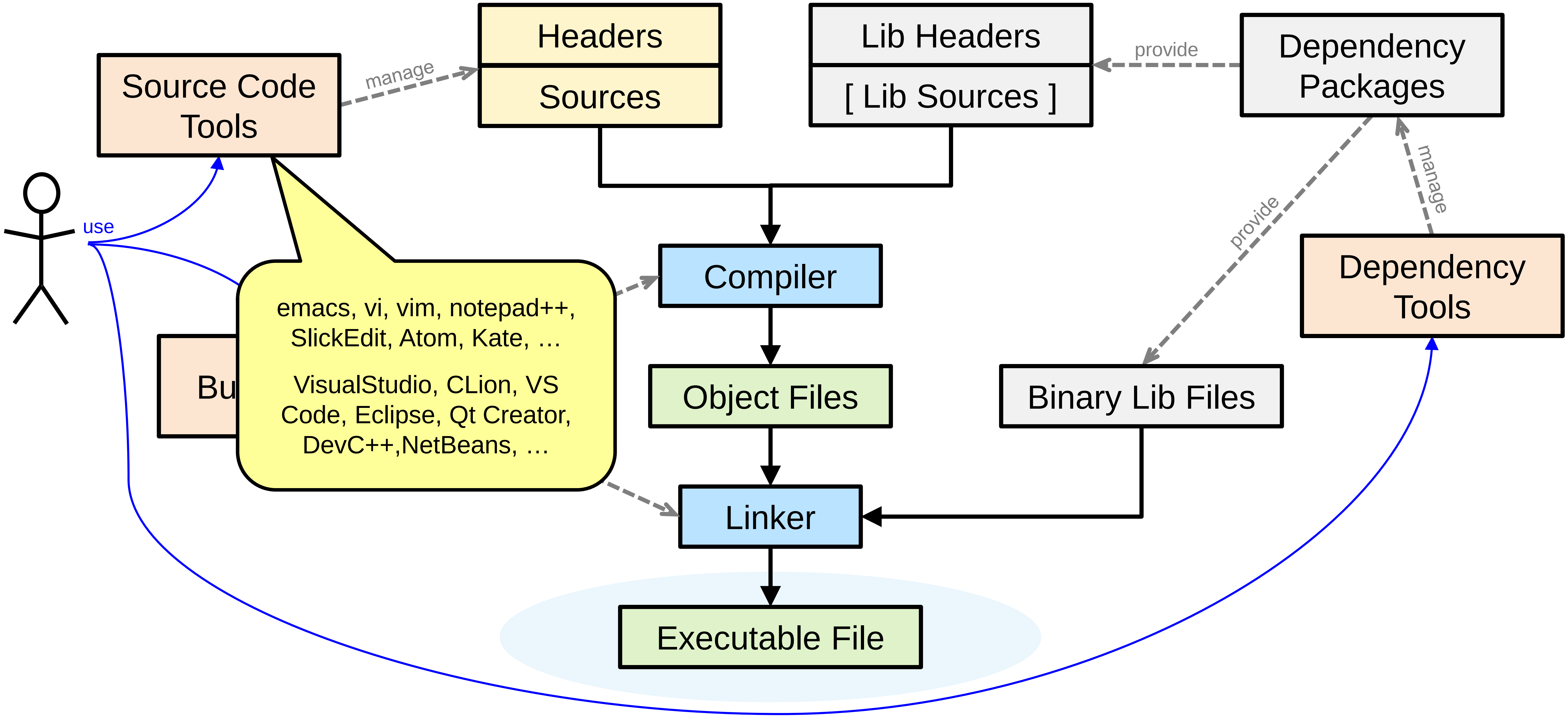
Why do we have this division?

- Dependencies (libraries)
  - Header files (`<vector>`, `<boost/text/text.hpp>`)
  - Precompiled files (`libstdc++.a`, `libc++.so`, `msvcrt.lib`, `ws2_32.dll`, `crt1.o`)
  - Source files (Boost, Catch2)
  - Resources (icons, images, translations for I18N)

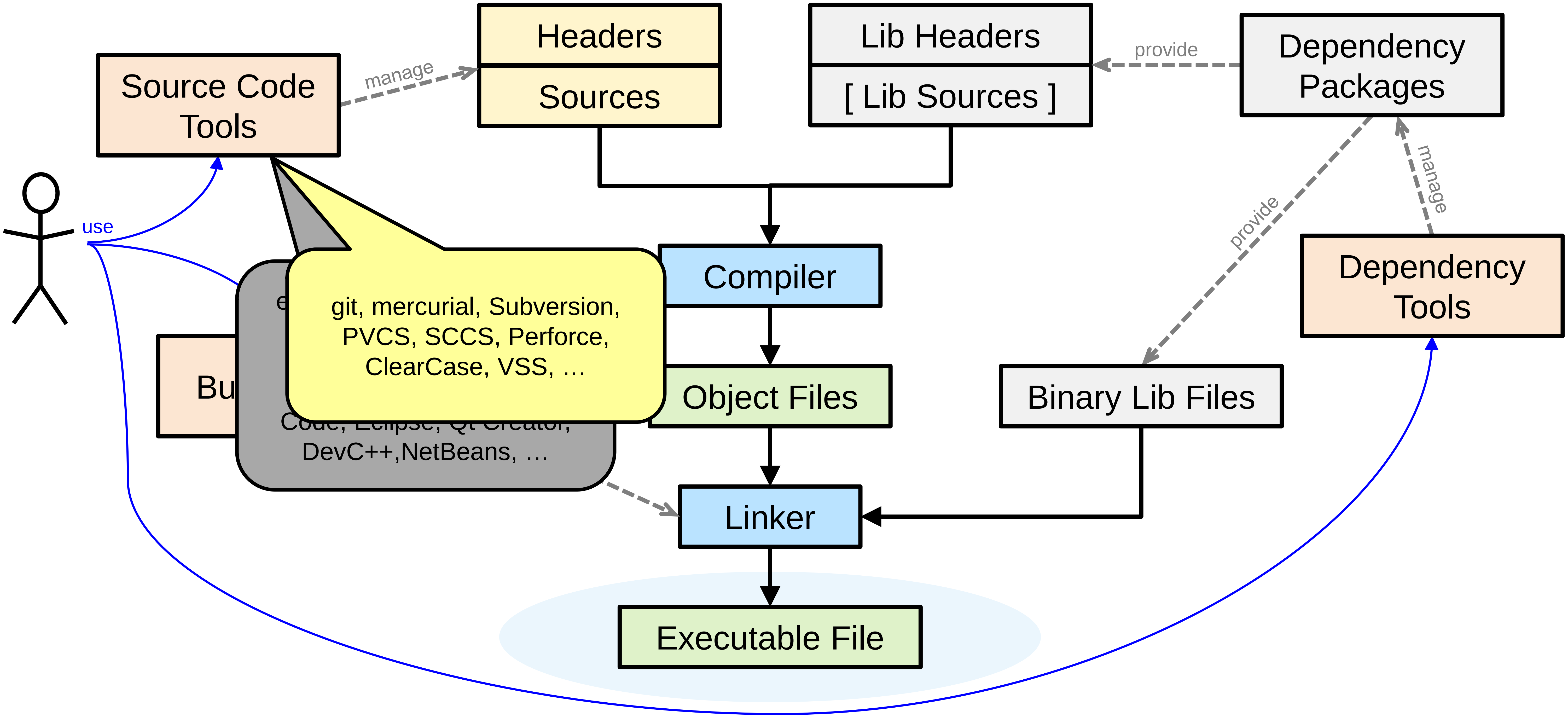
C++ Programming Ecosystem – Building Executables



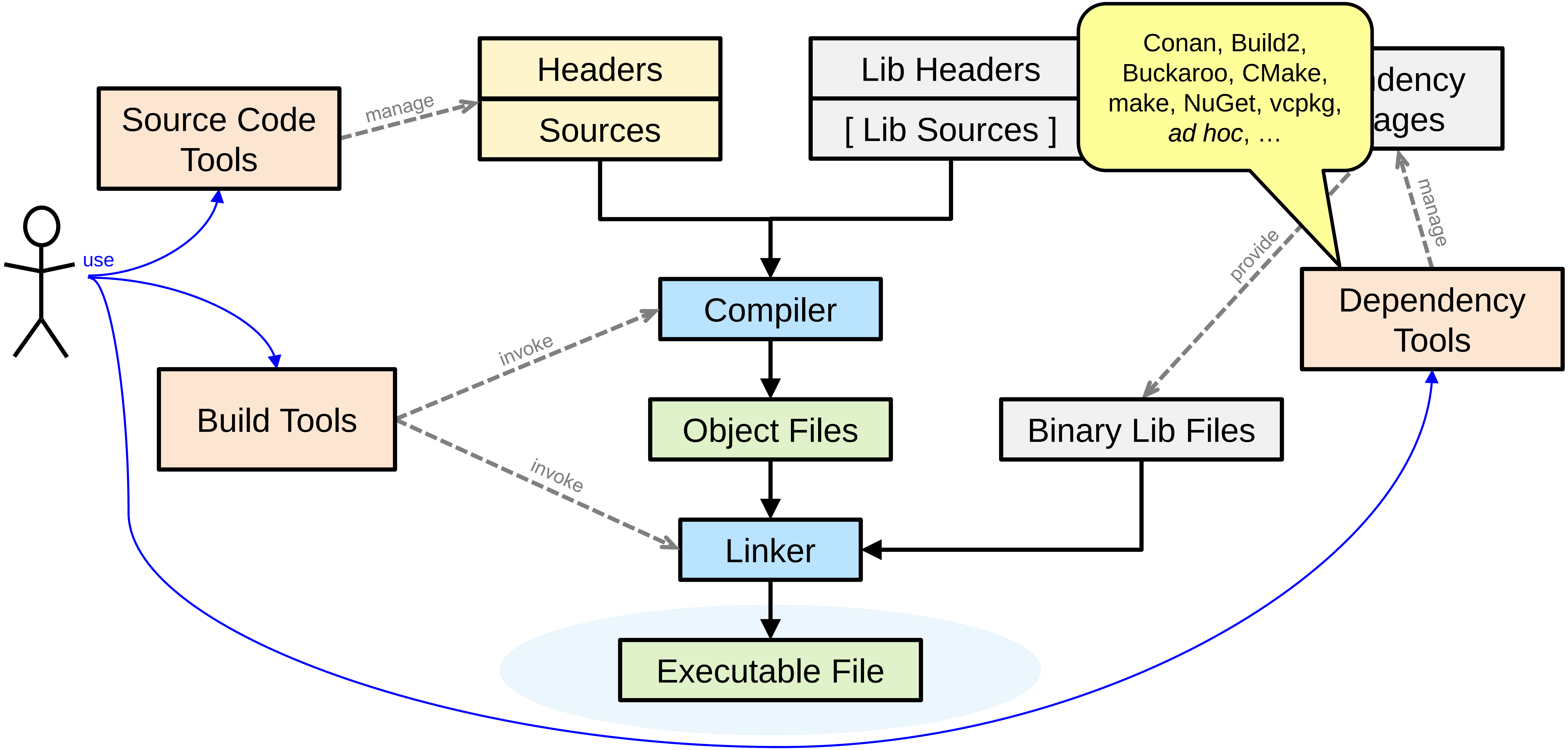
C++ Programming Ecosystem – Building Executables



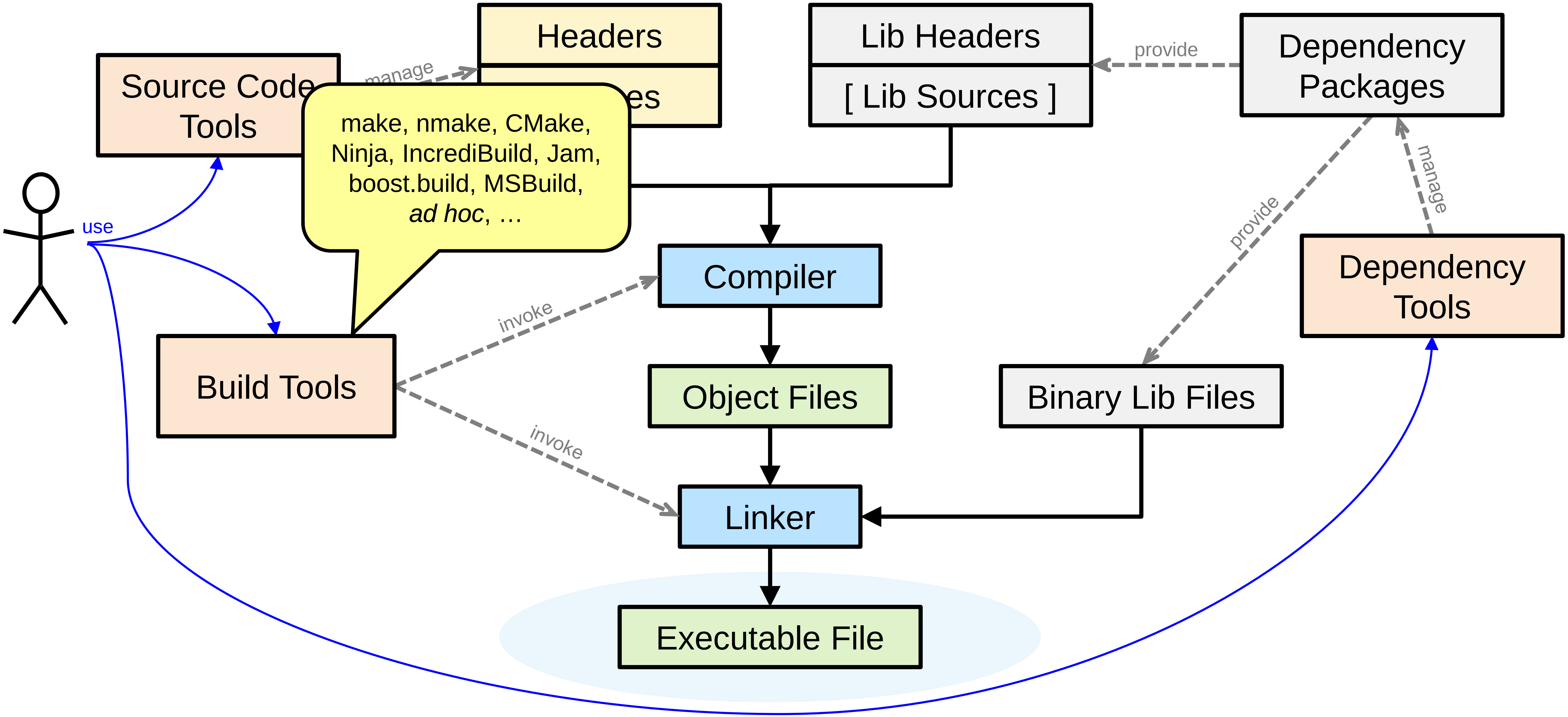
C++ Programming Ecosystem – Building Executables



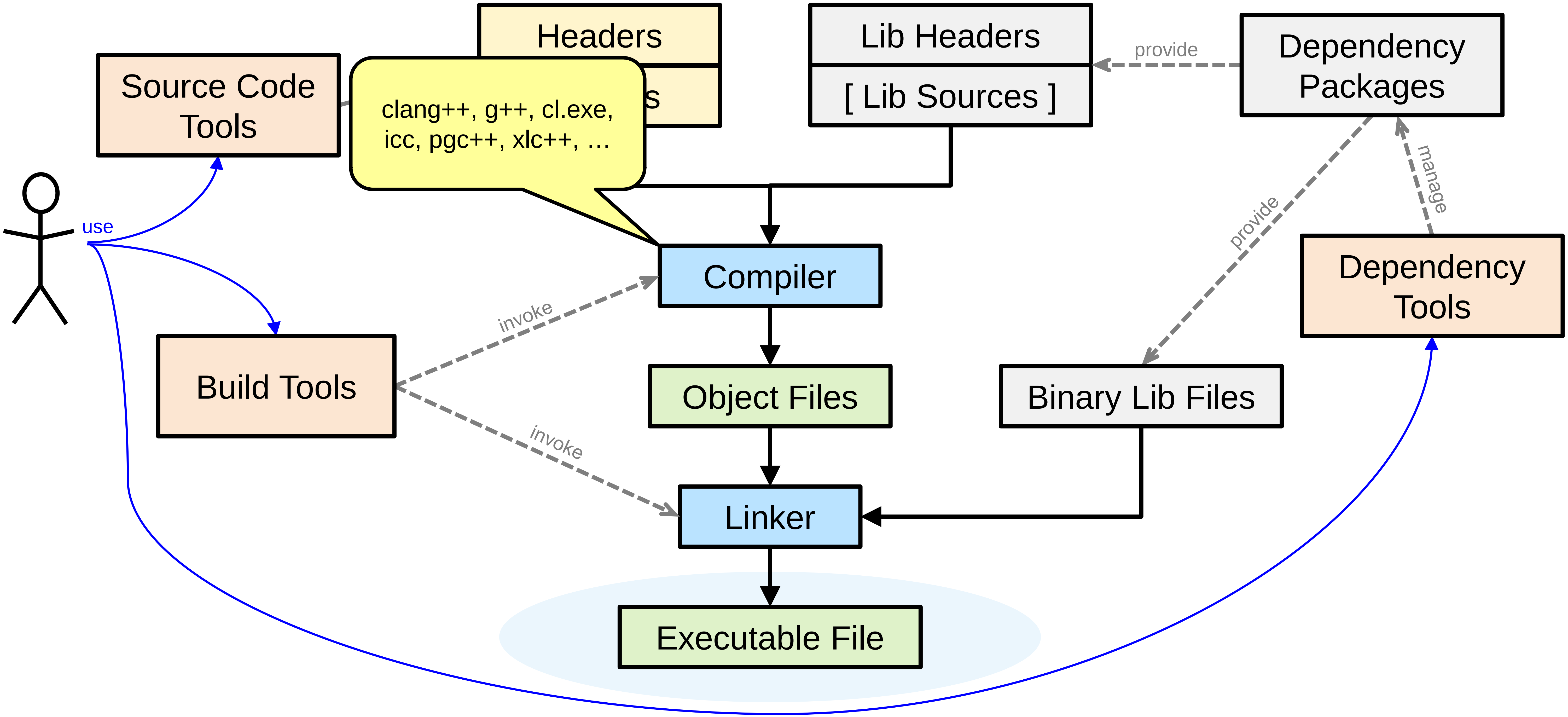
C++ Programming Ecosystem – Building Executables



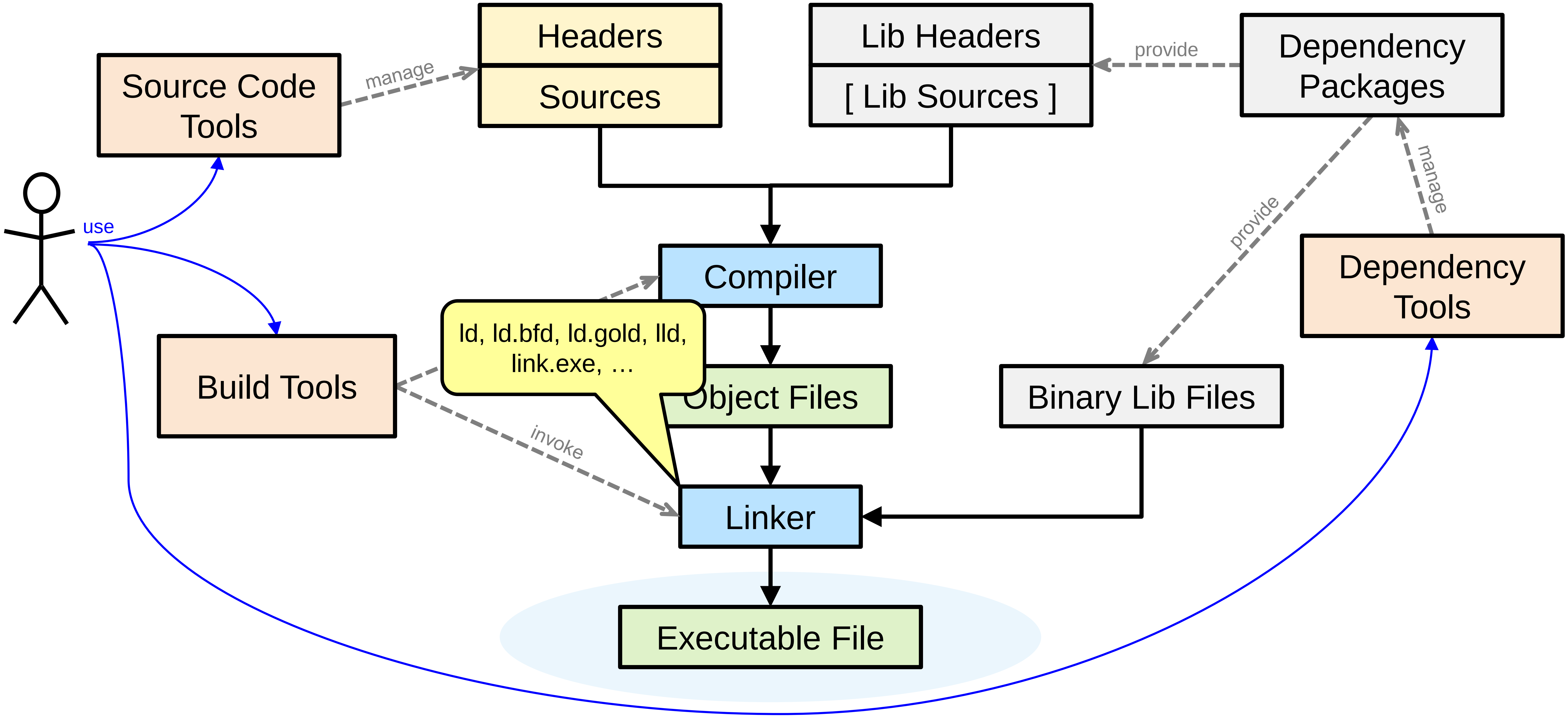
C++ Programming Ecosystem – Building Executables



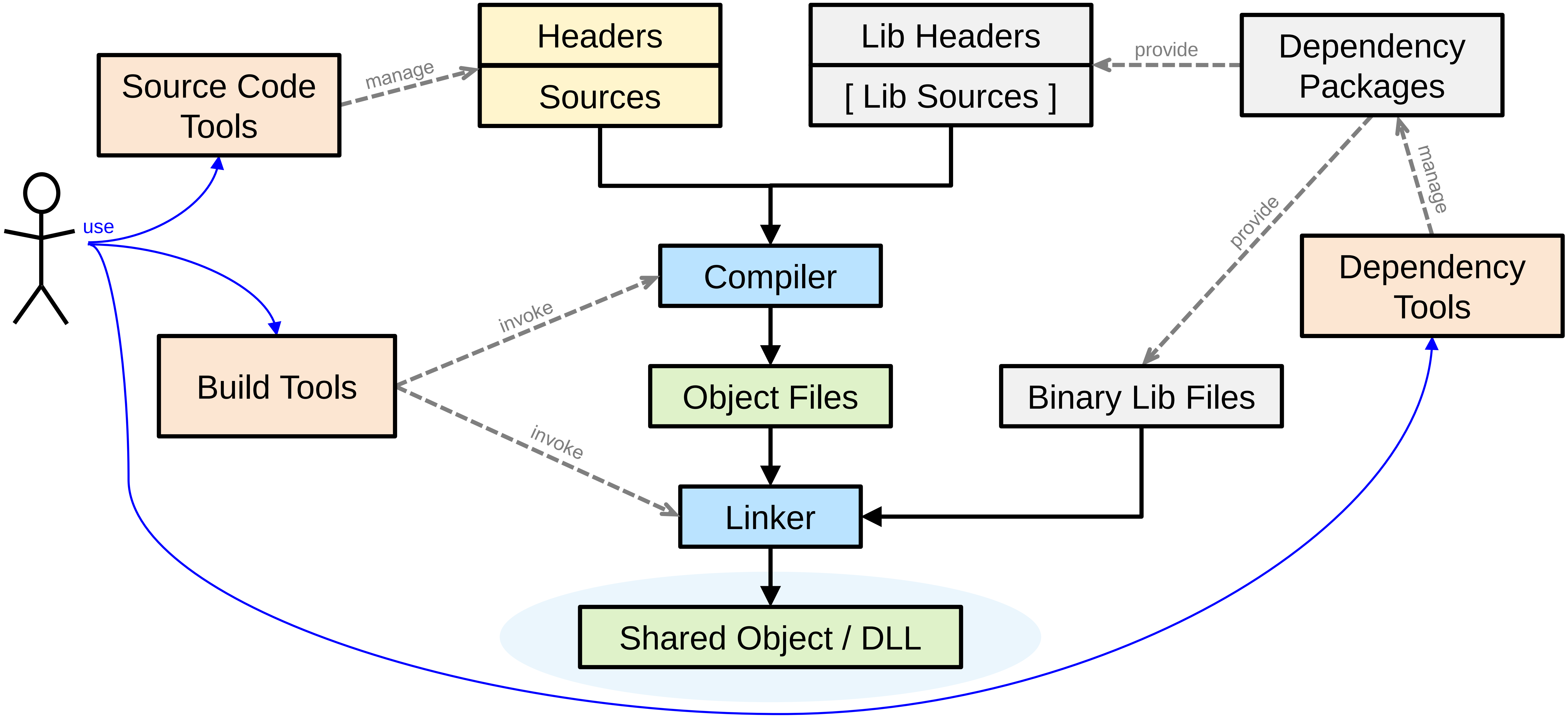
C++ Programming Ecosystem – Building Executables



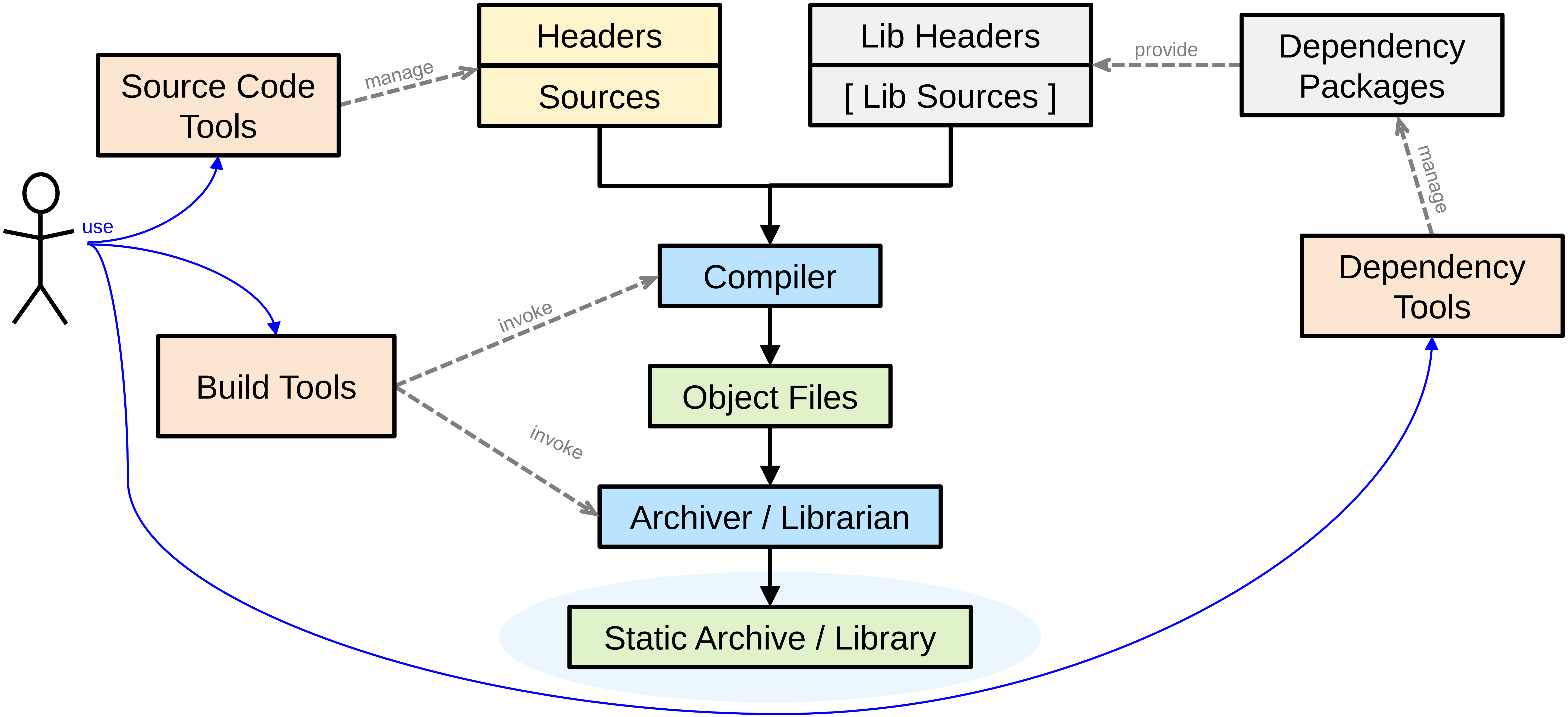
C++ Programming Ecosystem – Building Executables



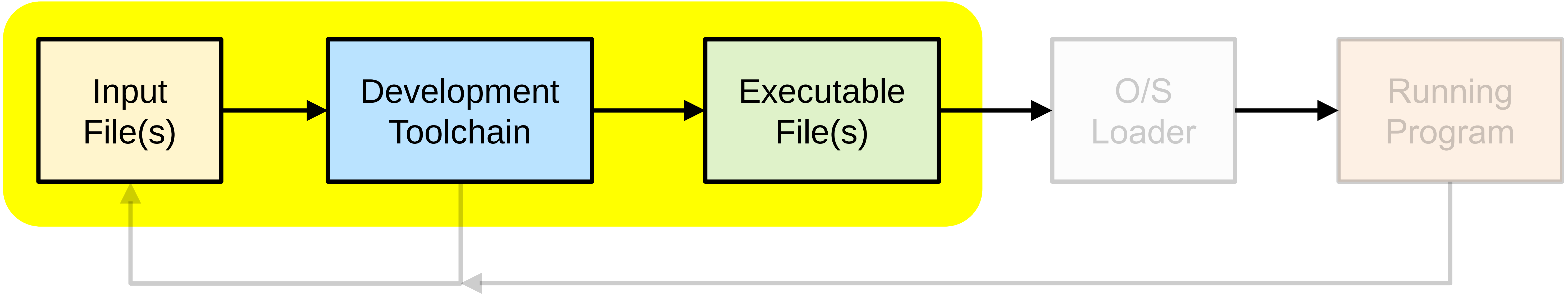
C++ Programming Ecosystem – Building Dynamic Libraries



C++ Programming Ecosystem – Building Static Libraries



Our Focus – Building C++ Executables



How do We Represent C++ Programs?

- Our programs are represented by source code
  - Source code is expressed in human-readable text files

- We typically have three kinds of source code
  - Header files (headers) – generally used more than once when building an executable
  - Source files (source) – generally used only once
  - Resource files (resources) – used only once to represent special non-executable information

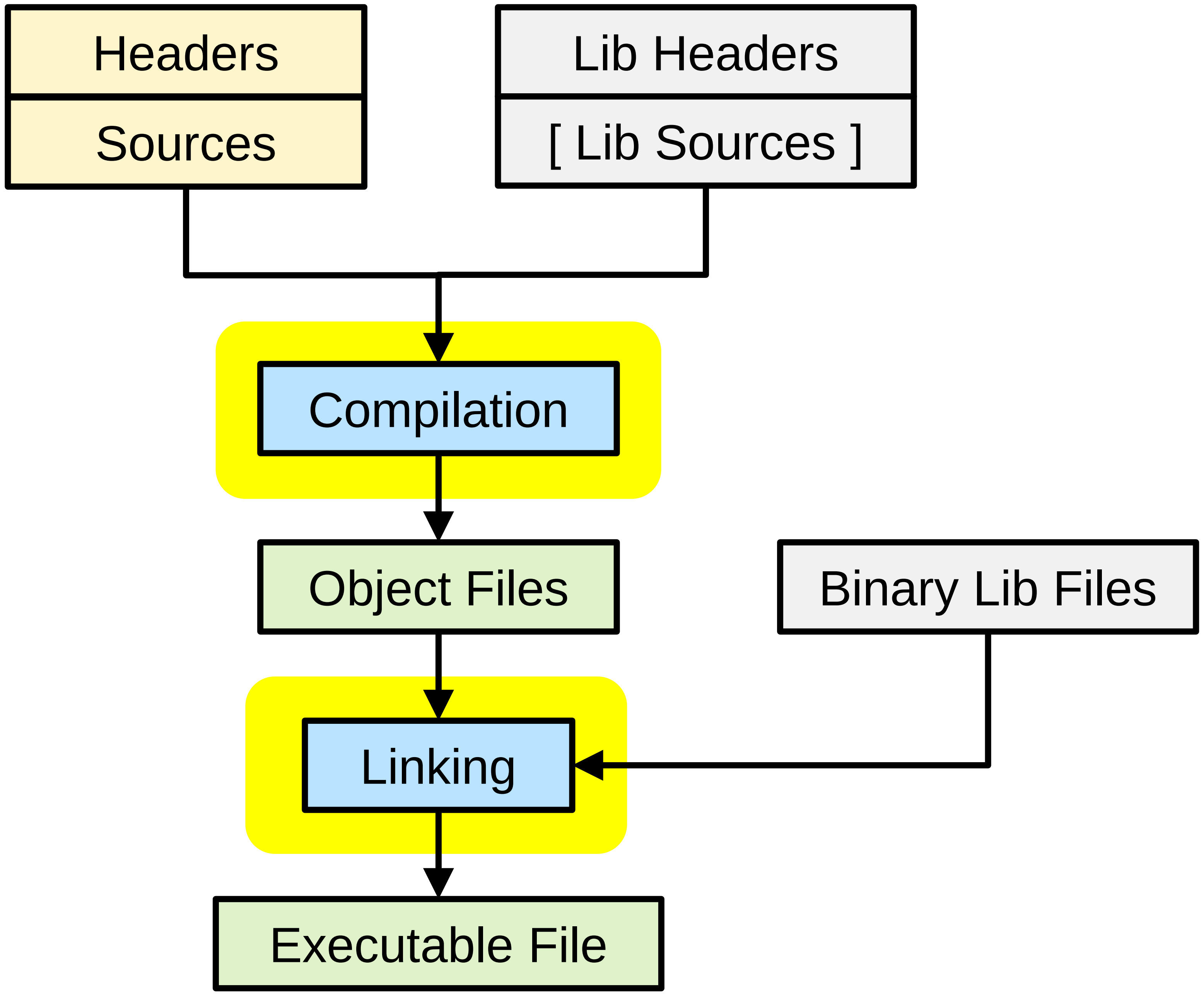
```cpp
//  hello.h
//=========
#ifndef HELLO_H_INC
#define HELLO_H_INC

#include <iostream>

void print_hello();

#endif
```

```cpp
//  hello.cpp
//===========
#include "hello.h"

void print_hello()
{
    std::cout << "Hello!" << std::endl;
}
```

```cpp
//  main.cpp
//==========
#include "hello.h"

int main()
{
    print_hello();
    return 0;
}
```

Building a C++ Executable

- Builds are accomplished by compilation and linking
  - Why?
  - Our human-readable text files must be converted to binary machine language



Building a C++ Executable

- Compilation
  - The process of converting human-readable source code into binary object files
  - From a high-level perspective, there are four stages of compilation:
    - Lexical analysis
    - Syntax analysis
    - Semantic analysis
    - Code generation
  - In C++, we typically generate one object file for each source file

- Linking
  - The process of combining object files and binary libraries to make a working program

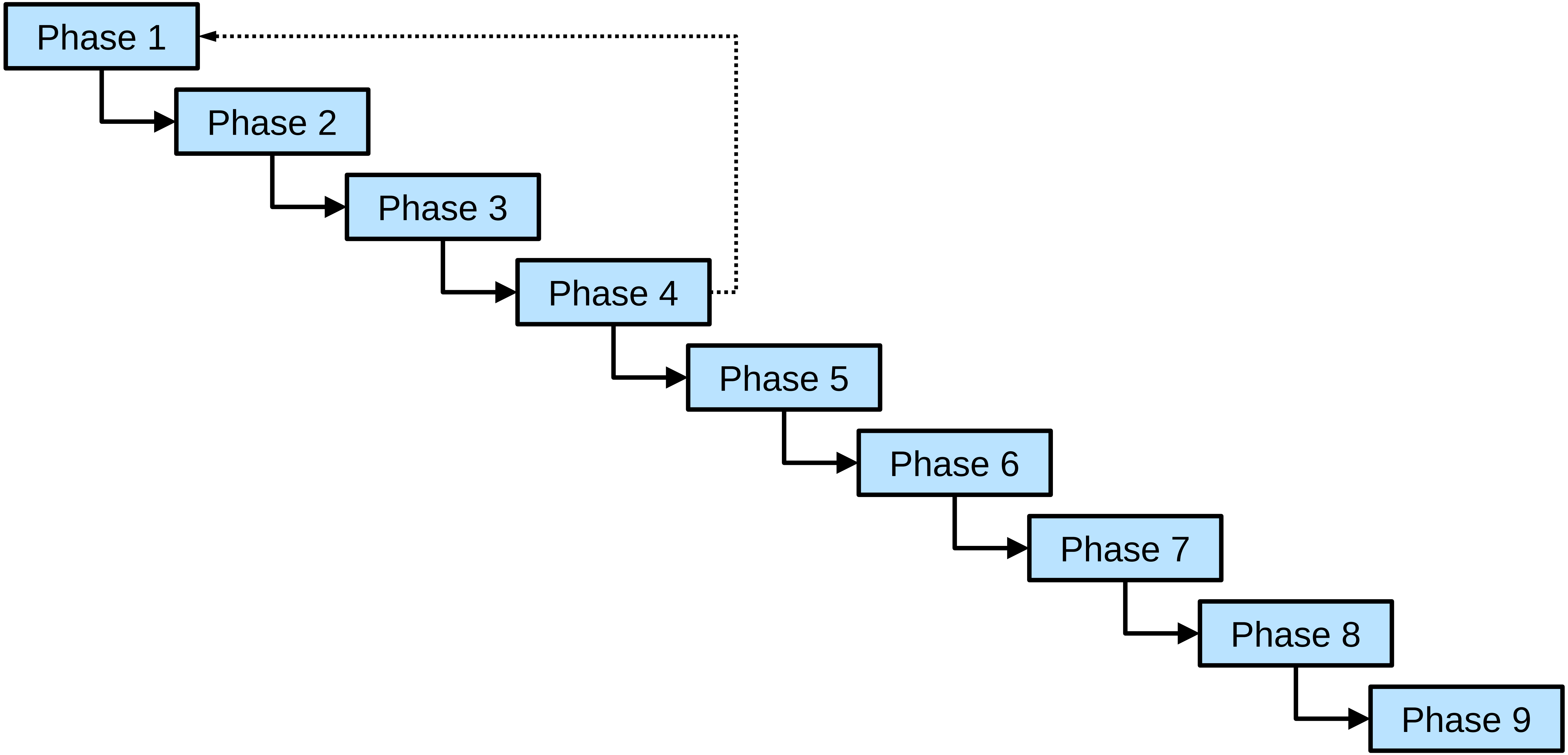
Compilation – From Source File to Object File

- The standard calls the compilation process **translation**

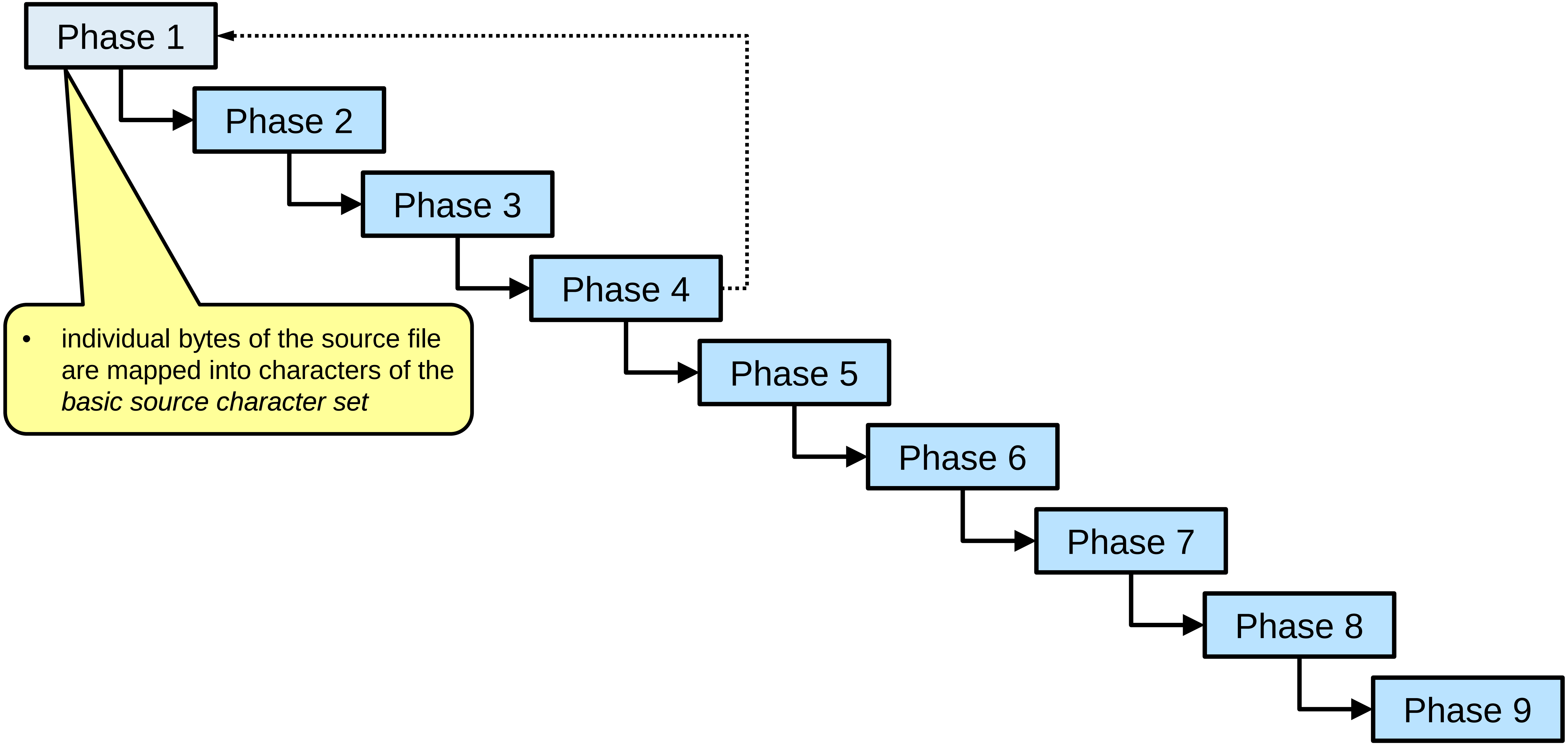
- In C++, translation is performed upon a **translation unit** (**TU**) in nine well-defined stages
  - Evocatively named Phases 1 through 9

- A translation unit is defined (roughly) as
  - A source file,
  - Plus with all the headers and source files included via the `#include` directive,
  - Minus any source lines skipped by conditional inclusion preprocessing directives (#ifdef),
  - And all macros expanded

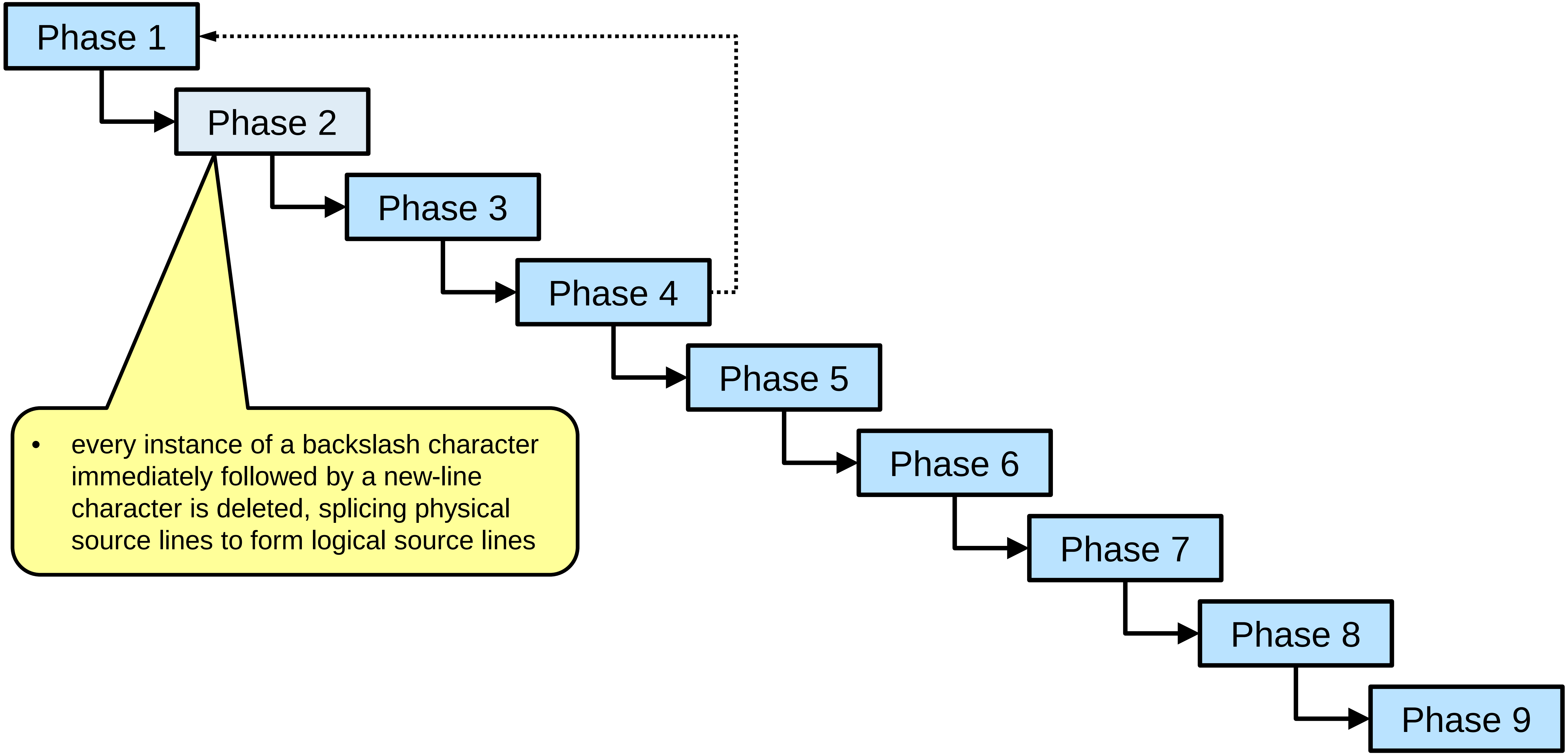
Phases of Translation



Phases of Translation (1)

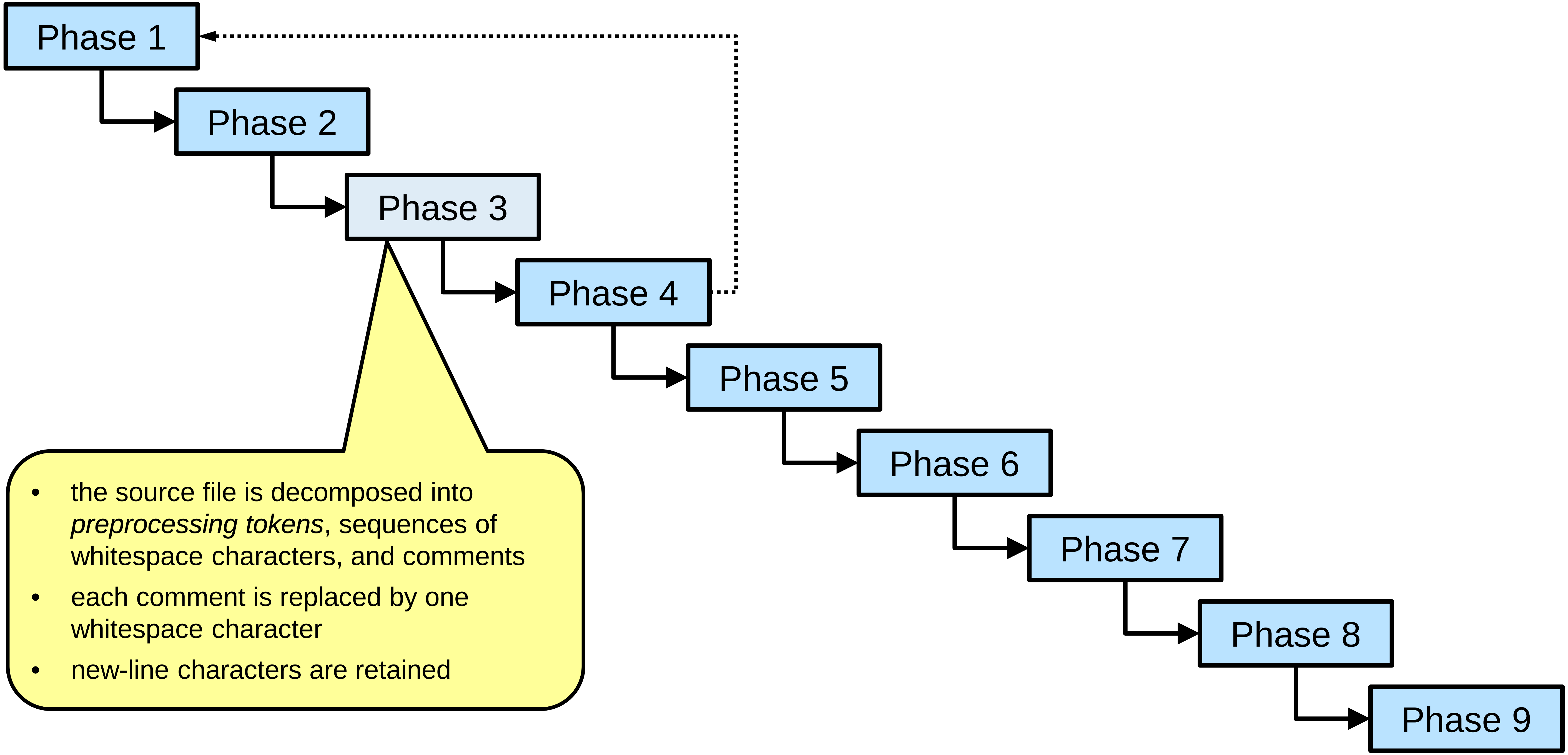


Phases of Translation (2)

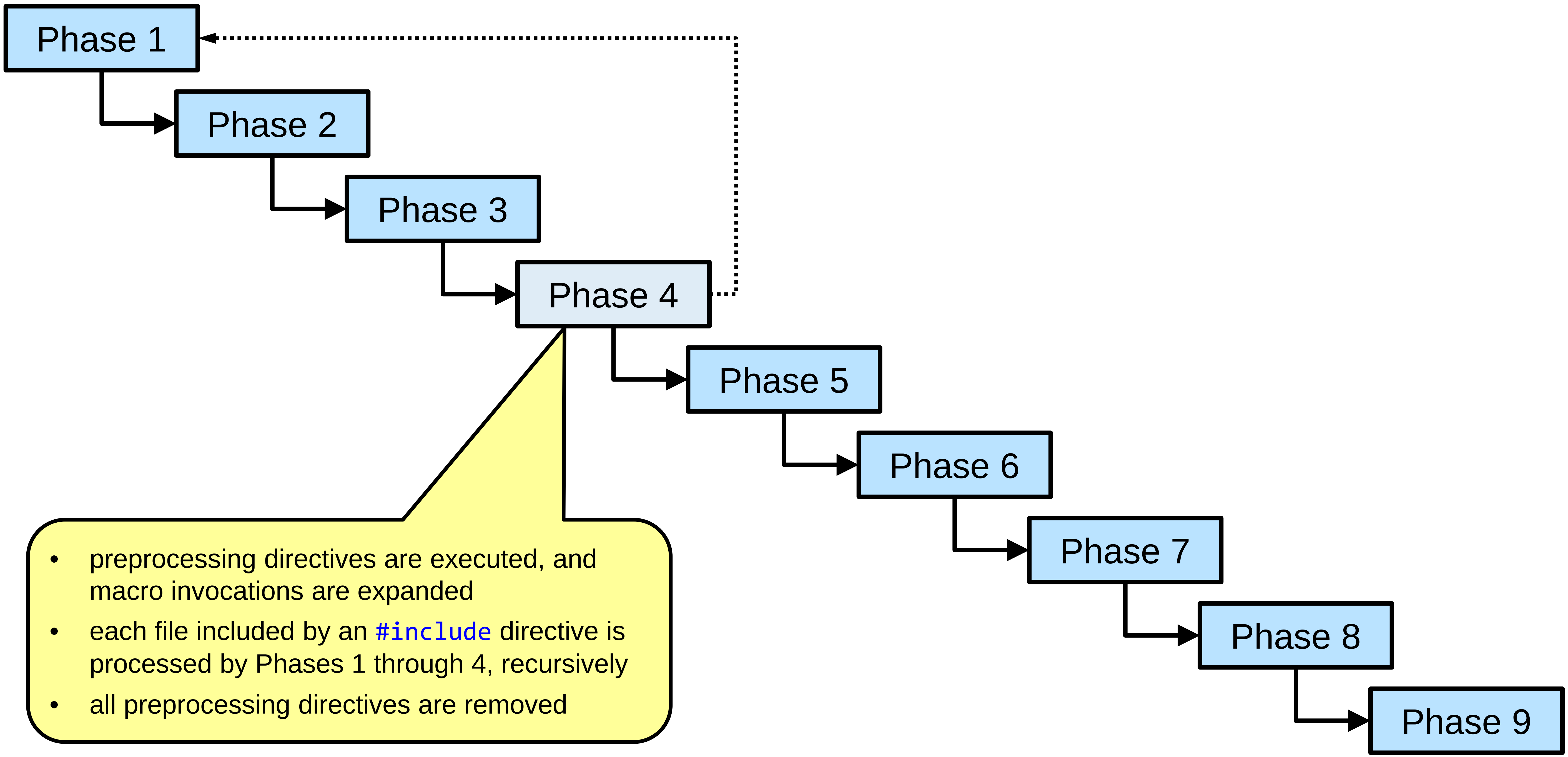


- every instance of a backslash character immediately followed by a new-line character is deleted, splicing physical source lines to form logical source lines

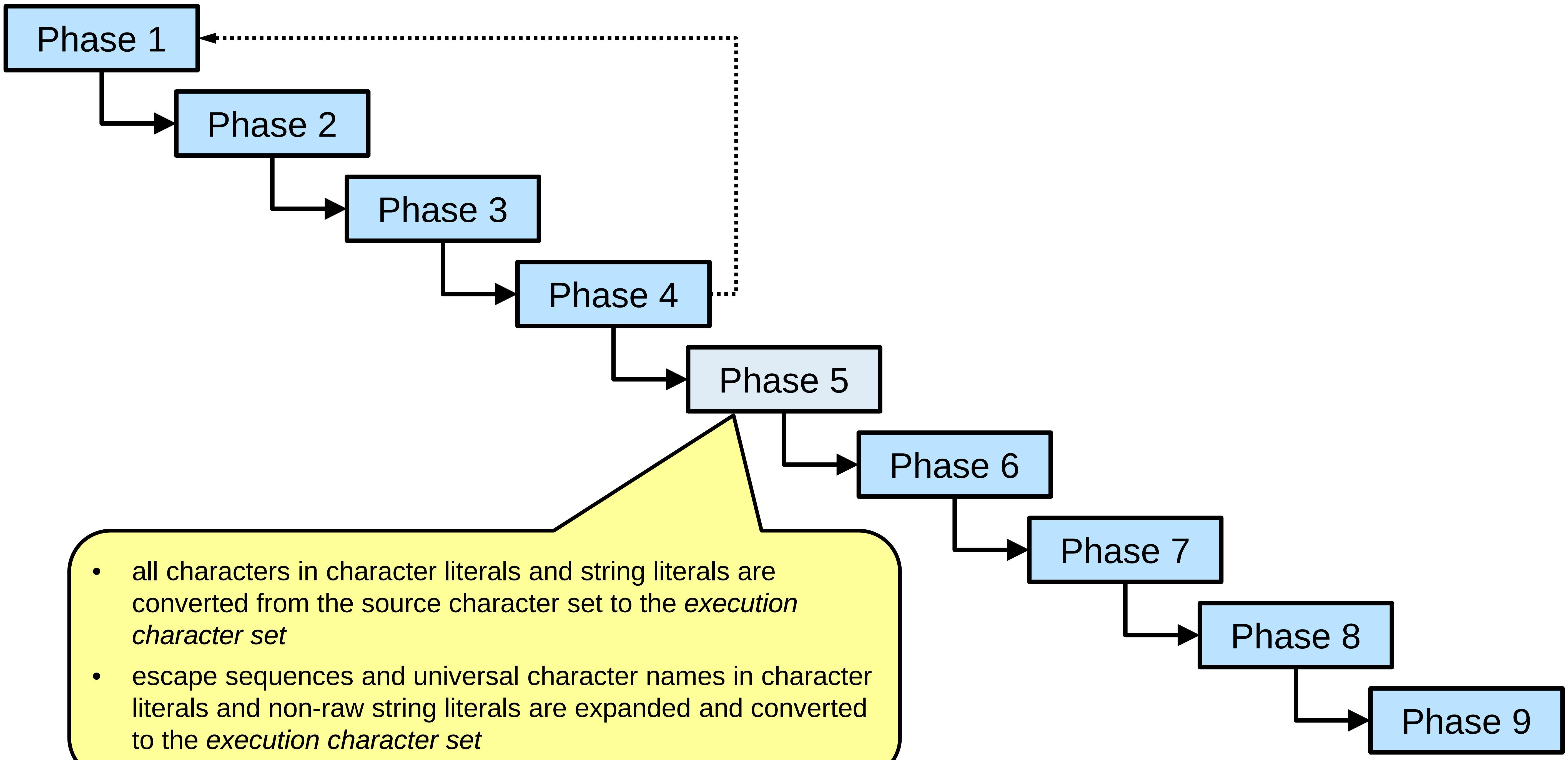
Phases of Translation (3)



Phases of Translation (4)

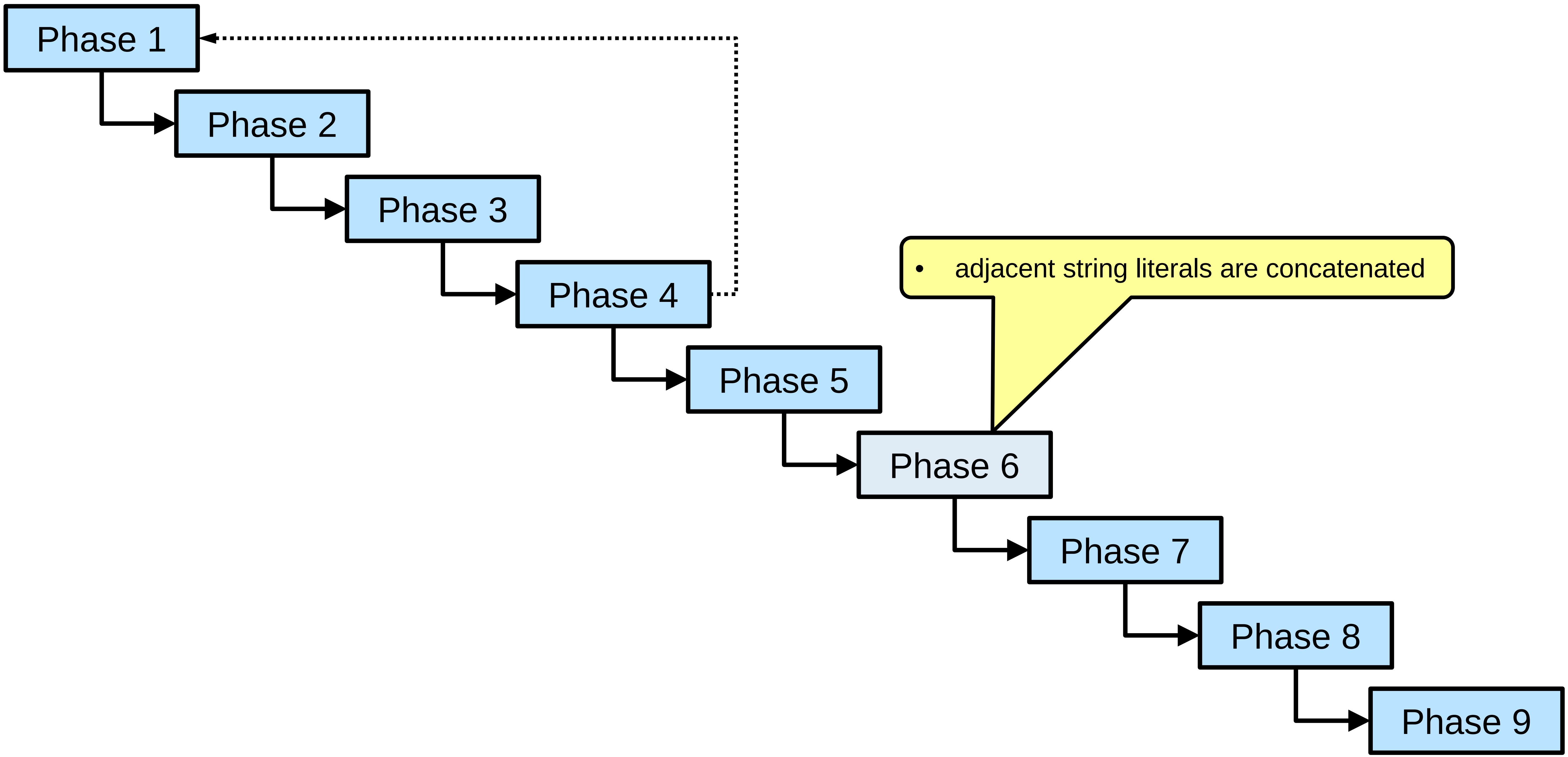


Phases of Translation (5)

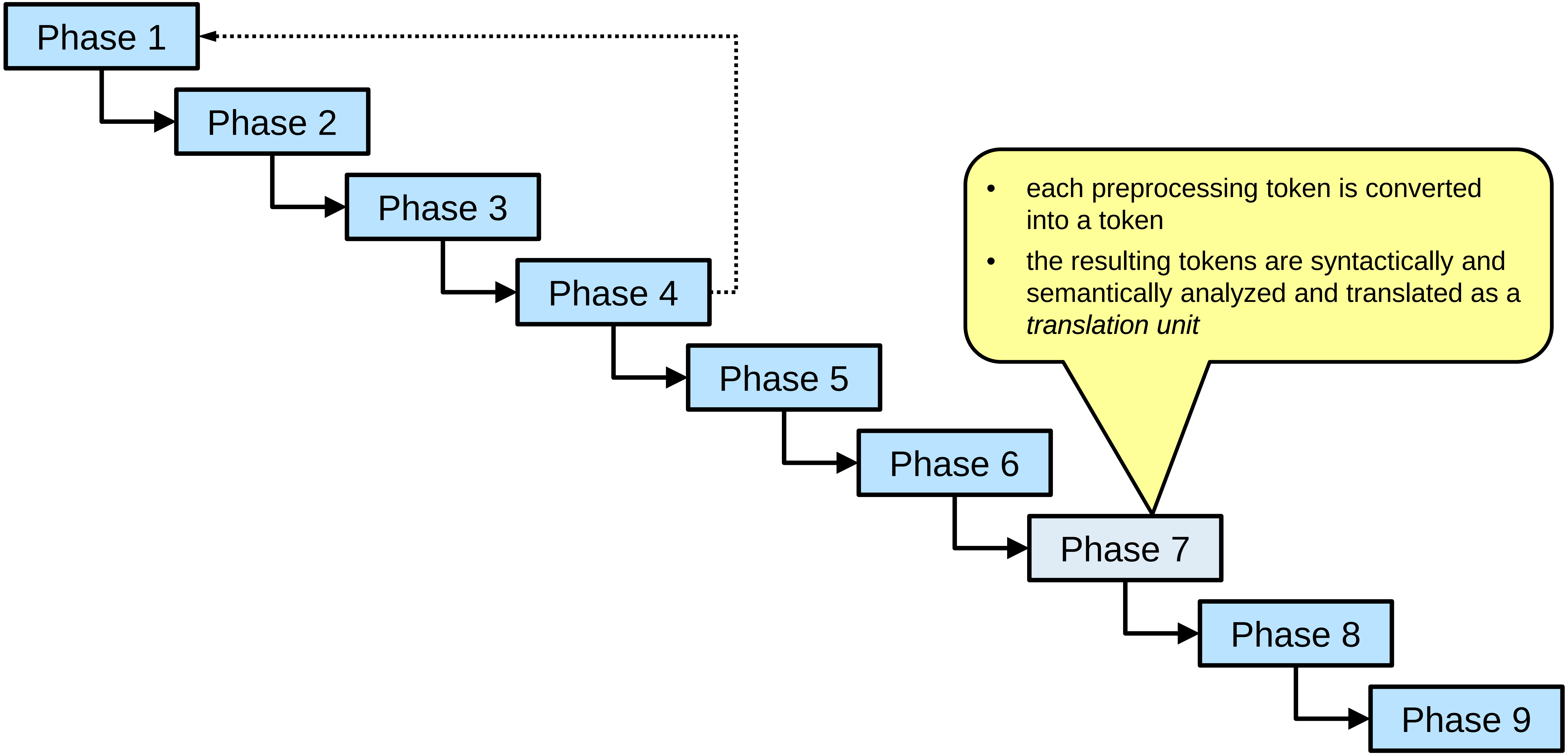


- all characters in character literals and string literals are converted from the source character set to the *execution character set*
- escape sequences and universal character names in character literals and non-raw string literals are expanded and converted to the *execution character set*

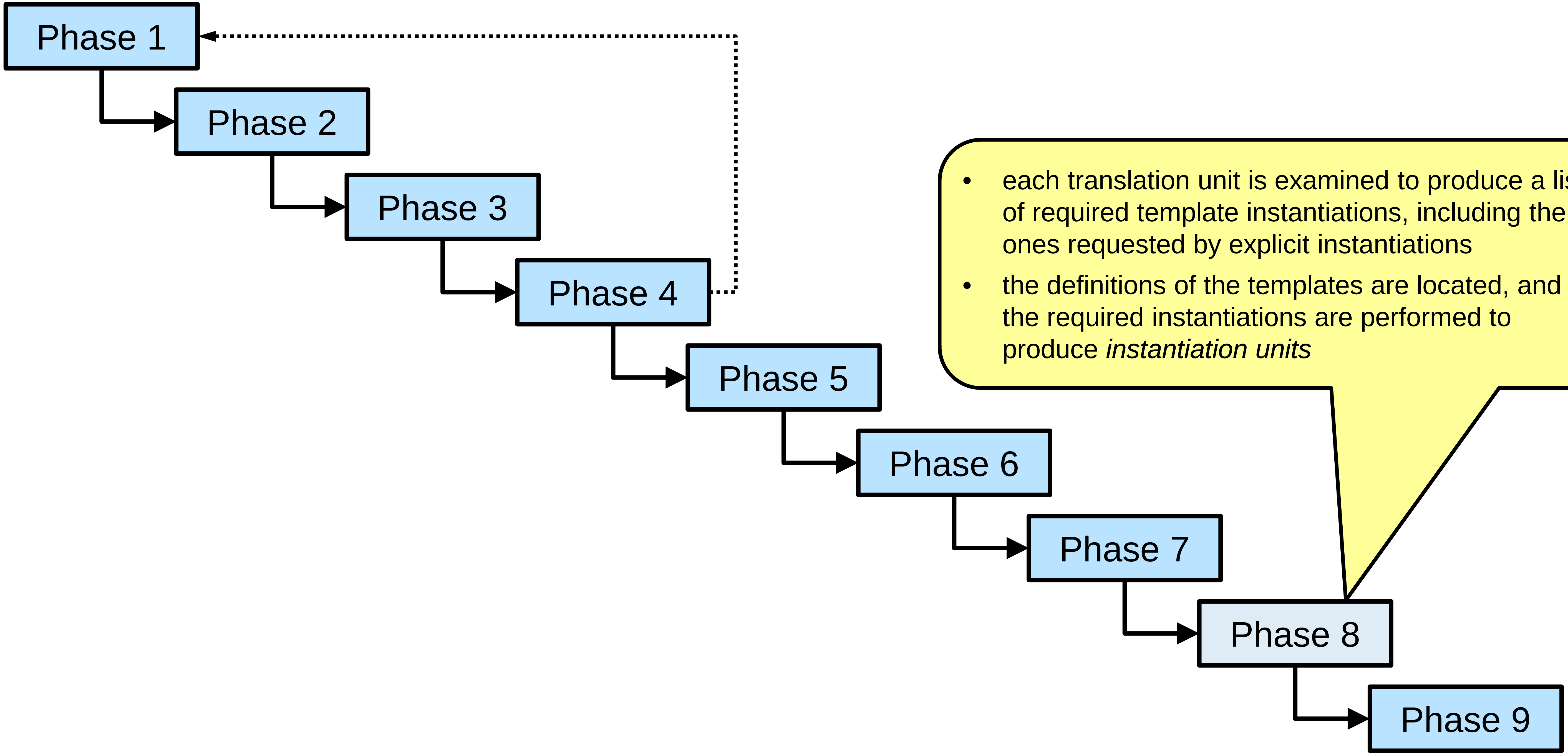
Phases of Translation (6)



Phases of Translation (7)

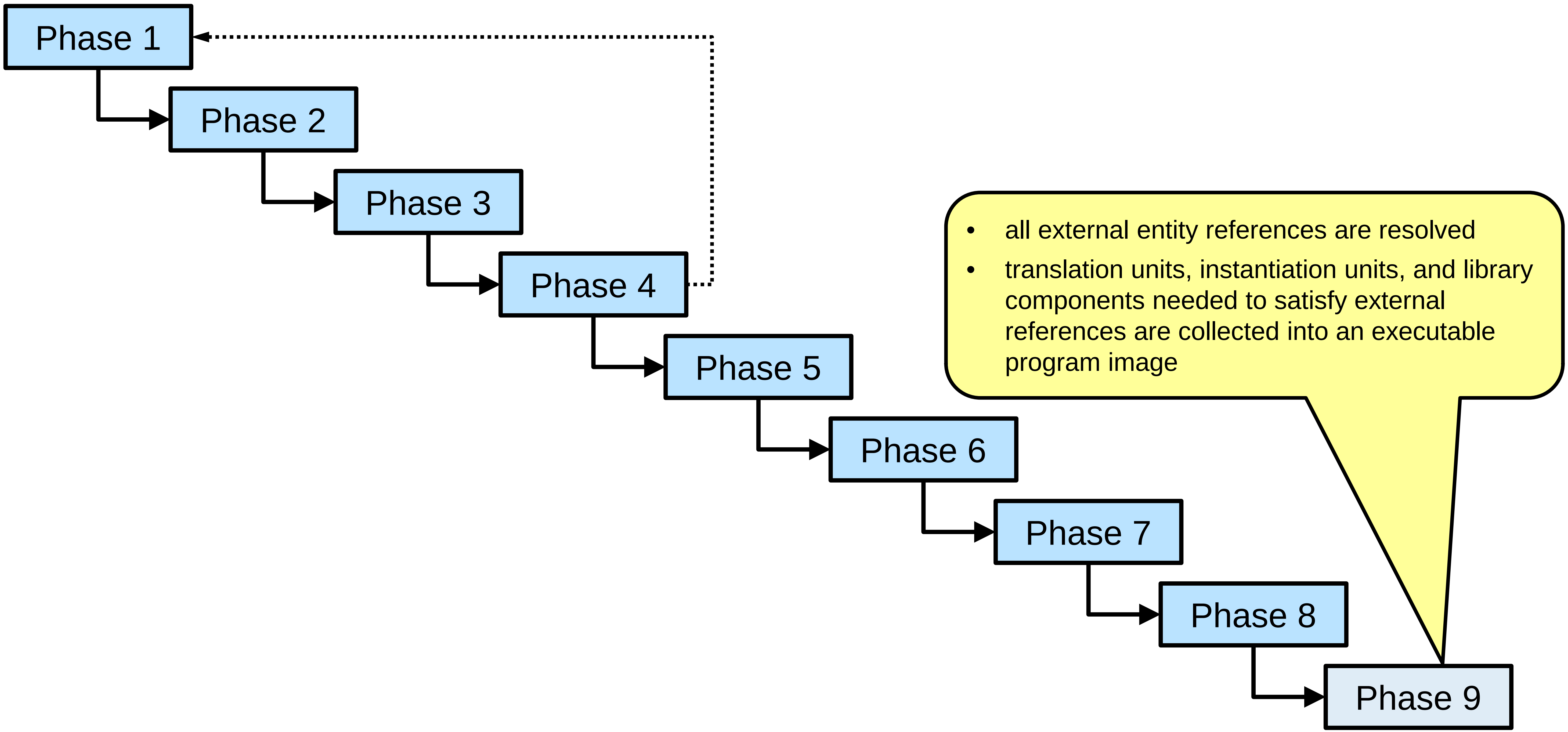


Phases of Translation (8)



- each translation unit is examined to produce a list of required template instantiations, including the ones requested by explicit instantiations
- the definitions of the templates are located, and the required instantiations are performed to produce *instantiation units*

Phases of Translation (9)



Compilation – From Source File to Object File

- Phases 1 through 6 perform lexical analysis
  - These are what we usually think of as *pre-processing*
  - The output of Phase 6 is a translation unit

- Phases 7 and 8 perform syntax analysis, semantic analysis, and codegen
  - These are what we usually think of as *compilation*
  - The output is called a *translated translation unit* (e.g., object code)

- Phase 9 performs program image creation
  - This is what we usually think of as *linking*
  - The output is an executable image suitable for the intended execution environment

Phases of Translation (6) – a Sample TU For Fun

```
//  hello.h
//=========
#ifndef HELLO_H_INC
#define HELLO_H_INC

#include <iostream>

void print_hello();

#endif
```

```
//  hello.cpp
//===========
#include "hello.h"

void print_hello()
{
    std::cout << "Hello!" << std::endl;
}
```

```
//  main.cpp
//==========
#include "hello.h"

int main()
{
    print_hello();
    return 0;
}
```

- Compilers provide a way to inspect TU contents (or something close to it)
  - With GCC, you can use the `–E` flag:

```
$ g++ -std=c++20 -E main.cpp  | egrep -v '#' | tee main.i
$ g++ -std=c++20 -E hello.cpp | egrep -v '#' | tee hello.i
```

- How many lines in `main.i`? `hello.i`?
  - 41,625 / 41,624 {with GCC 10.2 on Ubuntu 18.04}

Declarations and Definitions

- An **entity** is one of these things:
  - value
  - object
  - reference
  - structured binding
  - function
  - enumerator
  - type
  - class member
  - bit-field
  - template
  - template specialization
  - namespace
  - pack

Declarations and Definitions

- A **name** is the use of an identifier (several forms are defined) that denotes an entity (or label)

- Every name that denotes an entity is introduced by a **declaration**

- A **declaration** introduces one or more **names** into a translation unit
  - A declaration may also *re-introduce* a name into a translation unit

- A **definition** is a declaration that fully defines the entity being introduced

- A **variable** is an entity introduced by the declaration of an object (or of a reference other than a non-static data member)

Declarations and Definitions

- Every declaration *is also a definition*, unless:
  - It declares a function without specifying the function's body
  - It contains the `extern` specifier
  - It is a declaration of a class name without a corresponding definition
  - It is a function declaration without a corresponding definition
  - It is a parameter declaration in a function declaration that is not a definition
  - It is a template parameter
  - It is a `typedef` declaration
  - It is a `using` declaration
  - And several other cases…

- The set of definitions is a proper subset of the set of declarations

Declarations and Definitions

Declarations

```
extern int       a;
extern const int c;
```

Definitions

```
extern int       a = 0;
extern const int c = 37;
```

Declarations and Definitions

Declarations

```
extern int       a;
extern const int c;

int f(int);
```

Definitions

```
extern int       a = 0;
extern const int c = 37;

int f(int x)
{
    return x + 1;
}
```

Declarations and Definitions

Declarations

```
extern int       a;
extern const int c;

int f(int);




class Foo;
```

Definitions

```
extern int       a = 0;
extern const int c = 37;

int f(int x)
{
    return x + 1;
}

class Foo
{
    int mval;

  public:
    Foo(int x) : mval(x) {}
};
```

Declarations and Definitions

Declarations

```
extern int       a;
extern const int c;

int f(int);




class Foo;







using N::d;
```

Definitions

```
extern int       a = 0;
extern const int c = 37;

int f(int x)
{
    return x + 1;
}

class Foo
{
    int mval;

  public:
    Foo(int x) : mval(x) {}
};

namespace N { int d; }
```

Declarations and Definitions

Declarations

```
extern int       a;
extern const int c;

int f(int);




class Foo;







using N::d;

enum color : int;
```

Definitions

```
extern int       a = 0;
extern const int c = 37;

int f(int x)
{
    return x + 1;
}

class Foo
{
    int mval;

  public:
    Foo(int x) : mval(x) {}
};

namespace N { int d; }

enum color : int { red, green, blue };
```

Declarations and Definitions

Declarations

```
struct Bar
{
    int compute_x(int y, int z);
};


using bar_vec = std::vector<Bar>;

typedef int Int;
```

Definitions

```
int  Bar::compute_x(int y, int z)
{
    return (y + z)*3;
}
```

Linkage

- C++ allows certain declarations of an entity to occur more than once in the same program, translation unit, or scope.

- If a name denotes (refers to) one of these things, it may have **linkage**
  - object
  - reference
  - function
  - type
  - template
  - namespace
  - value

- If a name has linkage, then it refers to the same entity as when that name was introduced by a declaration in a different scope

Linkage

- A name can have one of three kinds of linkage – external, internal, no(ne)

- **External linkage**
  - An entity denoted by a name with external linkage can be referred to by names from scopes of other translation units, or from other scopes in the same translation unit

- **Internal linkage**
  - An entity denoted by a name with internal linkage can be referred to by names from scopes in the same translation unit

- **No linkage**
  - An entity denoted by a name with no linkage can be referred to only by names from the scope where it is declared

External Linkage

```
// my_header.h
//------------
int f(int i);        // declares function f

extern int const x; // declares x

using Int = int;

// my_source.cpp
//--------------
#include "my_header.h"

int f(int i)         // defines f (and i)
{
    return i + x;
}
```

```
// my_other_source.cpp
//---------------------
#include "my_header.h"

extern int const x = 17; // defines x

int g(int i)             // defines g (and i)
{
    return f(i) % 3;
}


// f, x, and Int have external linkage
```

Internal Linkage

```
// my_header.h
//------------
int f(int i);        // declares function f

extern int const x; // declares x

using Int = int;

// my_source.cpp
//--------------
#include "my_header.h"

int f(int i)         // defines f (and i)
{
    return i + x;
}
```

```
// my_other_source.cpp
//---------------------
#include "my_header.h"

extern int const x = 17; // defines x

namespace {
    int const y = 1000;   // defines y
    struct foo {          // defines foo
        int fuzzy();
        ...
    };
    using fubar = foo;
}

int g(int i)          // defines g and i
{
    foo my_foo;
    ...
    return f(i) + y - my_foo.fuzzy();
}

// y, foo, fubar have internal linkage
```

Internal Linkage

```
// my_header.h
//------------
int f(int i);       // declares function f

extern int const x; // declares x

using Int = int;

// my_source.cpp
//--------------
#include "my_header.h"

int f(int i)        // defines f (and i)
{
    return i + x;
}
```

```
// my_other_source.cpp
//---------------------
#include "my_header.h"

extern int const x = 17; // defines x

namespace {
    int const y = 1000;  // defines y
    struct foo {         // defines foo
        int fuzzy();
        ...
    };
    using fubar = foo;
}

int g(int i)             // defines g and i
{
    typedef foo FUBAR;
    foo my_foo;
    int z = 64;          // defines z
    ...
    return f(i) + y - my_foo.fuzzy();
}
// FUBAR, my_foo, and z have no linkage
```

The One-Definition Rule (ODR)

- A given translation unit can contain at most one definition of any:
  - variable
  - function
  - class type
  - enumeration type
  - template
  - default argument for a parameter for a function in a given scope
  - default template argument

- There may be multiple declarations, but there can only be one definition

The One-Definition Rule (ODR)

- A given program must contain exactly one definition of every non-`inline` variable or function that is used in the program
  - Again, multiple declarations are OK, but only one definition

- For an `inline` variable or an `inline` function, a definition is required in every translation unit that uses it
  - `inline` was originally a suggested optimization made to the compiler
  - It has now evolved to mean "multiple definitions are permitted"

- Exactly one definition of a class must appear in any translation unit that uses it in such a way that the class must be complete
  - Like construction or calling a member function

The One-Definition Rule (ODR) - OK

```
// TU-1
//-----

extern int const x;
```

```
// TU-2
//-----

extern int const x = 0;
```

The One-Definition Rule (ODR) - Invalid

```
// TU-1
//-----

extern int const x = 0;
```

```
// TU-2
//-----

extern int const x = 0;
```

The One-Definition Rule (ODR) - OK

```
// TU-1
//-----

void f(int i);
```

```
// TU-2
//-----

void f(int i)
{
    return i+1;
}
```

The One-Definition Rule (ODR) - Invalid

```cpp
// TU-1
//-----

void f(int i)
{
    return i+1;
}
```

```cpp
// TU-2
//-----

void f(int i)
{
    return i+1;
}
```

The One-Definition Rule (ODR) - OK

```cpp
// TU-1
//-----

struct foo
{
    int val;
    foo() : val(-1){}
};
```

```cpp
// TU-2
//-----

struct foo
{
    int val;
    foo() : val(-1){}
};
```

The One-Definition Rule (ODR) - OK

```cpp
// TU-1
//-----

struct foo
{
    int val;
    foo();
};
```

```cpp
// TU-2
//-----

struct foo
{
    int val;
    foo();
};

foo::foo(int i) : val(i) {}
```

The One-Definition Rule (ODR) - Invalid

```cpp
// TU-1
//-----

struct foo
{
    int val;
    foo();
};

foo::foo(int i) : val(i) {}
```

```cpp
// TU-2
//-----

struct foo
{
    int val;
    foo();
};

foo::foo(int i) : val(i) {}
```

The One-Definition Rule (ODR)

- My simple guidelines:

- For an `inline` thing (variable or function) that get used in a translation unit, make sure it is defined at least once somewhere in that translation unit

- For everything else that gets used, make sure it is defined exactly once in across all translation units

Storage Duration and Objects

- Every object has a **storage duration**

- **automatic** storage duration
  - Object storage is allocated at the beginning of the enclosing block and deallocated at the end of the block
  - Applies to all local objects except those declared `thread_local`, `static`, or `extern`

- **dynamic** storage duration
  - Object storage is allocated and deallocated by the program using functions that perform dynamic memory allocation
  - Objects with this duration can be created using *new-expressions* and destroyed using *delete-expressions*

Storage Duration and Objects

- **static** storage duration
  - Object storage is allocated at the beginning of the program and deallocated at the end of the program
  - Applies to all objects declared at namespace scope, including the global namespace
  - Applies to all objects declared `static` or `extern`
  - There is only one instance of an object with static duration in the program

- **thread** storage duration
  - Object storage is allocated when the thread creating the object begins and deallocated when that thread ends
  - Applies only to objects declared `thread_local`
  - Each thread has its own instance of an object with thread duration

What is an ABI?

- ABI (application binary interface) is the platform-specific specification of how entities *defined* in one TU interact with entities *defined* in another TU

- For C++, this includes things like
  - Name mangling for functions
  - Name mangling for non-template types
  - Name mangling for template instantiations
  - The size, layout, and alignment of objects of any given type
  - How the bytes in an object's binary representation are interpreted
  - Calling conventions for passing arguments to functions and receiving a returned object
  - Calling conventions for making system calls

- On Linux, GCC and Clang use the Itanium ABI; On Windows, MSVC uses its own ABI

Name Mangling

- **Name mangling** refers to the way in which entity names in a TU are transformed into symbol names in object code

- C++ deliberately maintains binary compatibility with C
  - C++ object files can use functions and data in C object files, and vice versa

- The C language does not provide overloading or namespaces
  - Given a C function whose declaration is `void fubar(int)`,
  - The corresponding symbol name in object code is `_fubar`
  - The symbol name will be the same no matter the number of parameters or their types

- Re-using names in C can be tricky, if not impossible

Name Mangling

- C++ **does** support overloading and namespaces – names can be reused
  - A C++ entity name is mapped into a name that can co-exist with C entity names
  - Each overloaded use of a name is mapped into a unique symbol name
  - Extra information about the entity is encode into its symbol name

```
// From GCC 10.2 on Ubuntu 18.04
namespace wikipedia
{
    class article
    {
      public:
        std::string format();
        // symbol: _ZN9wikipedia7article6formatB5cxx11Ev
    };
}

std::string format(std::string const& fmt, int64_t val);
// symbol: _Z6formatRKNSt7__cxx1112basic_stringIcSt11char_traitsIcESaIcEEEl
```

Linking

- **Linking** is the final stage of compilation, performed by the linker

- The linker combines object files and libraries to produce an executable file
  - It examines all the object files and libraries to find their symbols
  - It determines a set of all symbols that are used by the program
  - It resolves references to internal and external symbols
  - It assigns addresses to the symbols representing functions and variables
  - It revises code and data to reflect these addresses
  - It emits executable code

- The compiler and the linker must agree on the ABI
  - Both must understand object code in exactly the same way

Loading and Running

- The operating system loader validates the executable image file
  - Checks permissions and resource requirements
  - Checks that the file is executable and that the instructions in it can be executed

- The operating system loads and runs the image
  - The executable file is copied into memory
    - Relocations and symbol fixups are performed, if needed (e.g., when using shared objects or DLLs)
  - Command-line parameters are copied onto the stack
  - Registers are initialized
    - Stack pointer set to point to the top of the stack; other registers are cleared
  - Control jumps to the start routine, which
    - Performs program initialization
    - Calls `main()` initialization with the command-parameters

Thank You for Attending!

Talk: https://github.com/BobSteagall/CppCon2020

Blog: https://bobsteagall.com