Separate Compilation Separate Compilation

Slides

Building C++ Programs

Problems

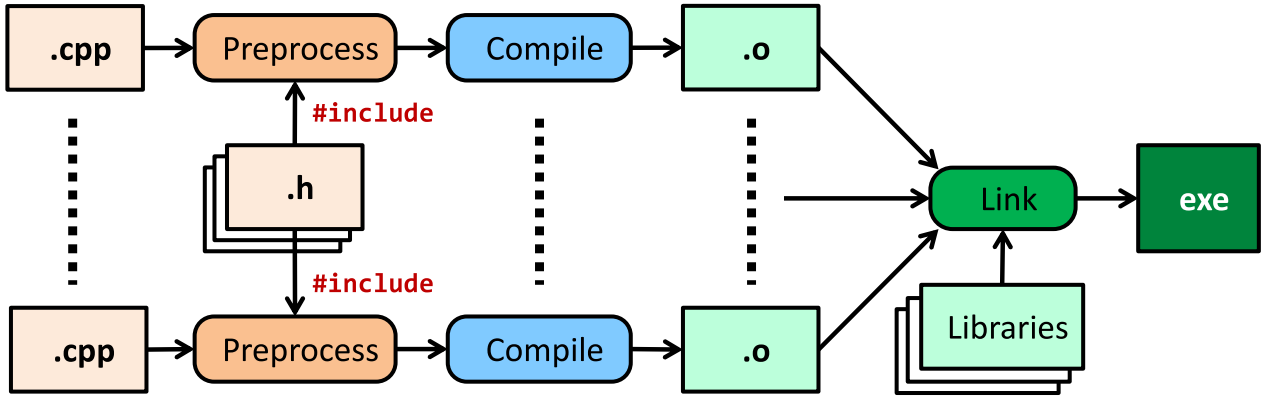
- large projects: **a lot** of source code
- many developers on the same project
- need to keep compilation times low
- some functionality should be shared between different projects

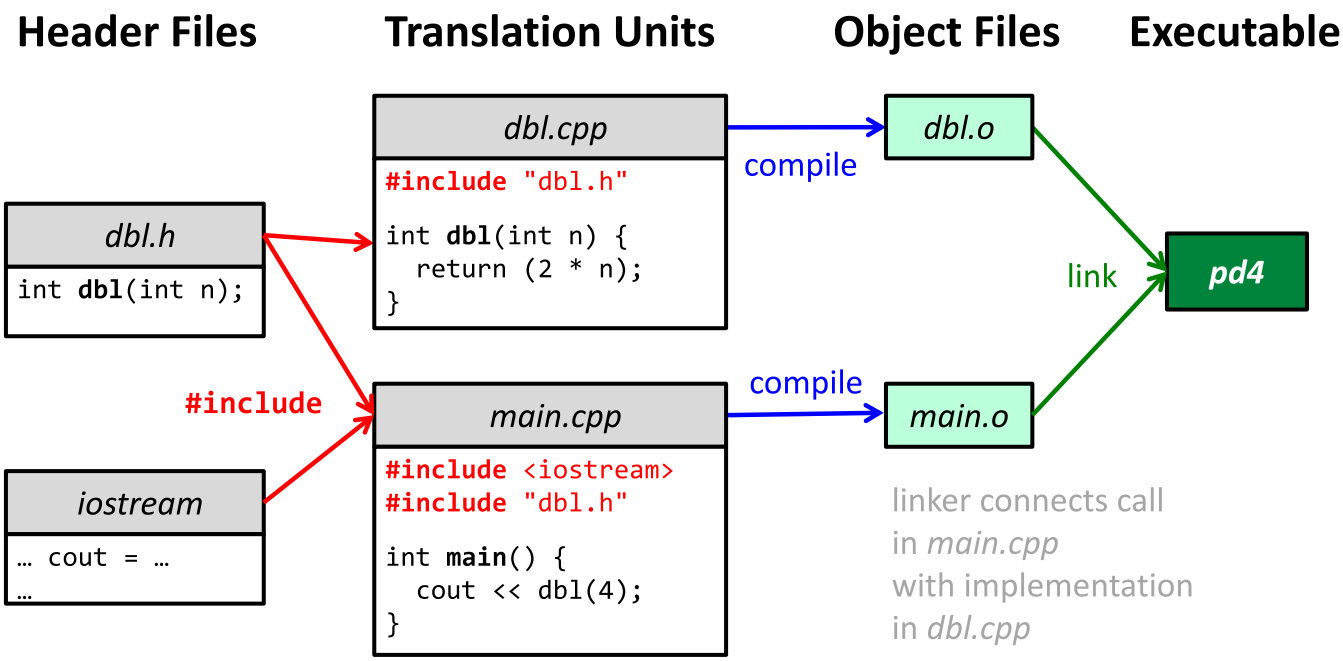
Solution

- divide program code into many separate files
- compile files separately
- only recompile files when modified

C++ Build Model

- **Headers** (*.h) + **Translation Units** (*.cpp) contain source code
- **Preprocessor** performs text substitutions
- **Compiler** translates TUs into *object files*
- **Linker** links object files and *external libraries* into an executable



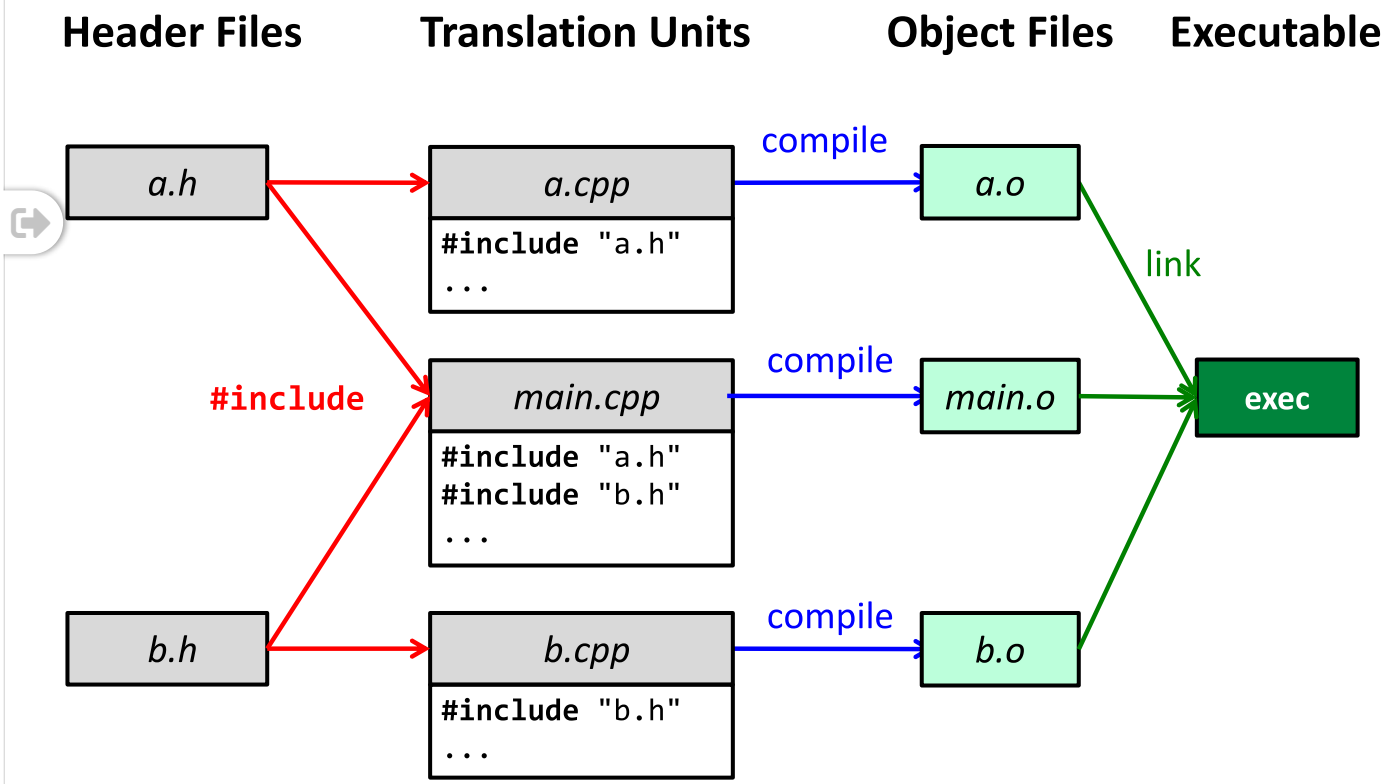



```
$ g++ -c main.cpp
$ g++ -c dbl.cpp
$ g++ main.o dbl.o -o pd4
```

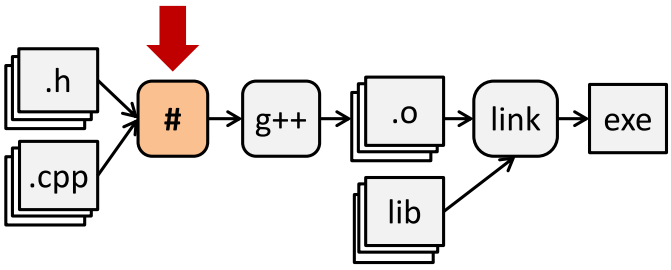
compiling *main.cpp* yields *main.o*

compiling *dbl.cpp* yields *dbl.o*

linking *main.o* + *dbl.o* yields executable *pd4*



Preprocessor



- text processing engine
- runs before compilation
- *directives* start with **#**

Usage in C++ (and you should limit it to that)

- combining source code (`#include`)
- conditional compilation
- obtaining platform information during compilation

Macros = Preprocessor Text Substitutions

- `#define MACRO [TEXT]`
- `#ifdef MACRO / #ifndef MACRO … #else … #endif`
- `#undef MACRO`

```
#define PLATFORM intelx86

#ifdef _WIN32                 only compiled for Windows
   …
#endif

#ifndef PLATFORM              compiled if PLATFORM is not defined
  …
#endif

#undef PLATFORM               delete definition of macro PLATFORM
```

Special Macros

- *__LINE__, __FILE__, __DATE__, __TIME__*
- *__cplusplus*
  - *C++98: #define 199711L*
  - *C++11: #define 201103L*
  - *C++14: #define 201402L*
  - *C++17: #define 201703L*

```
cout << "This is the line number " << __LINE__
     << " of file " << __FILE__ << ".\n"
     << "Its compilation began " << __DATE__
     << " at " << __TIME__ << ".\n"
     << "C++ version support: " << __cplusplus;
```

#include Problems

a.h	b.h	main.cpp

`struct foo {`
`  int member;`
`};`
 	
`void f(foo&);`
 	
…
	`#include "a.h"`
	
	`struct bar {`
	`  float x;`
	`};`
	
	`void g(foo&,bar&);`
	
	…
		`#include "a.h"`
		`#include "b.h"`
		
		`int main()`
		`{`
		`  foo f;`
		
		`  bar b;`
		
		`  …`
		`}`

#include Problems

- main.cpp after Preprocessing
- **foo** and **f** **defined twice**

```
struct foo {int member;};
void f(foo&);

struct foo {int member; };
void f(foo&);

struct bar {float x; };
void g(foo&,bar&);

int main() {
   foo f;
   bar b;
   …
}
```

```
// main.cpp: #include "a.h"

// main.cpp: #include "b.h"

// b.h:      #include "a.h"
```

Solution: #include Guards

a.h

```
#ifndef A_H
#define A_H



struct foo {
    int member;
};


void f(foo&);

…


#endif
```

b.h

```
#ifndef B_H
#define B_H

#include "a.h"

struct bar {
    float x;
};


void g(foo&,bar&);

…


#endif
```

main.cpp

```
#include "a.h"
#include "b.h"


int main()
{
   foo f;
   bar b;

   …
}
```

Header/Source Separation for Classes

a.h

```
class A {
public:

   void bar();

   double square(double);

   int bar(int a, int count);

private:

   //member variables
   int m_;
};
```

a.cpp

```
#include "a.h"

void A::bar() {
   cout << m_ << '\n';
}


double A::square(double x) {
   return (x * x);
}


int A::bar(int a, int c) {
   return ((a + m_) * c);
}
```

Namespace Pollution

Avoid in headers

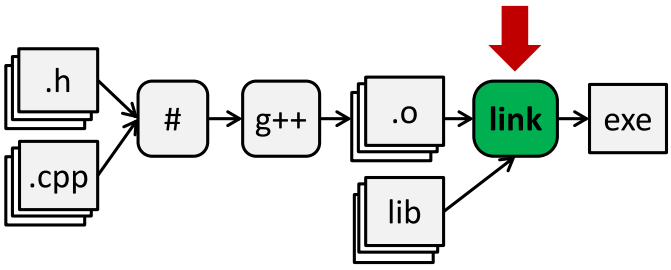
- using namespaces — e.g.: `using namespace std;`
- using symbols of a namespace — e.g.: `using std::cout;`

Why?

- every include of the header
  pulls in **all** the symbols that are used
- might cause conflicts
  if multiple namespaces are implicitly used
  by inclusion of multiple headers

Linker

- combines machine code
- inserts jump addresses
  for function calls

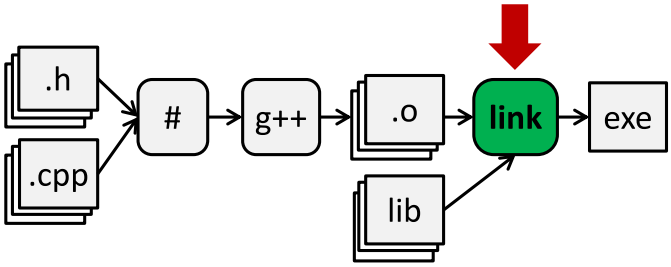


main.cpp	dbl.cpp	dbl.h

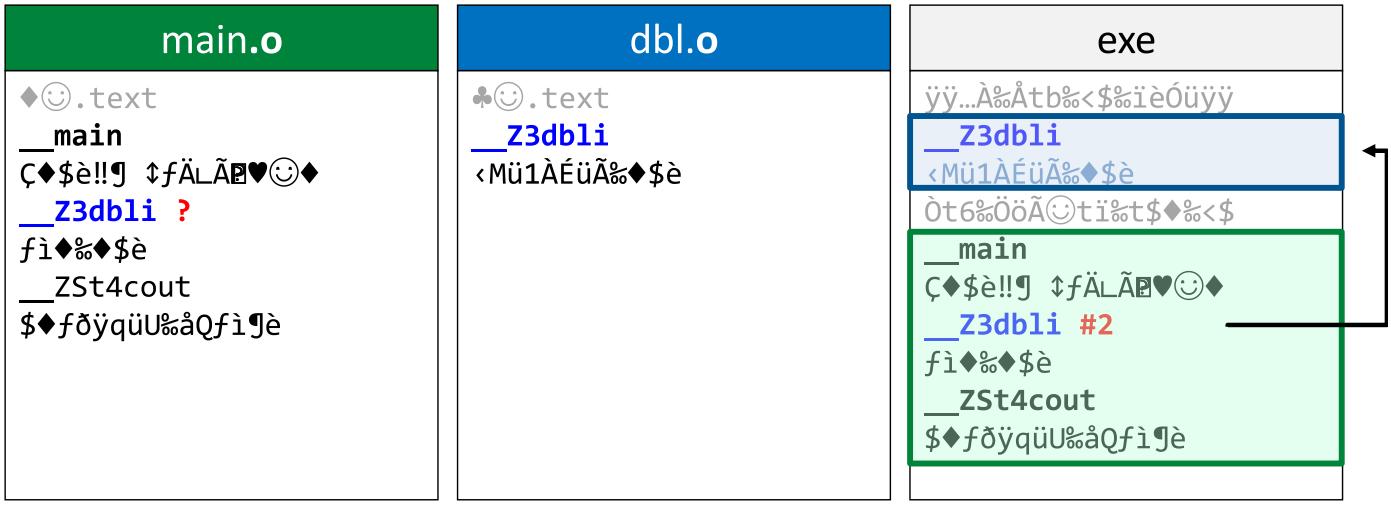
`#include "dbl.h"`
 	
`int main()`
`{`
`  cout << dbl(4)`
`  << '\n';`
`}`
	`#include "dbl.h"`
	
	`int dbl(int n)`
	`{`
	`  return (2 * n);`
	`}`
		`int dbl(int n);`

Linker

- combines machine code
- inserts jump addresses for function calls



Corresponding object files (conceptual):



Typical Linker Error

```
$ g++ -c main.cpp
$ g++ -c dbl.cpp
$ g++ main.o -o prog
    (.text.startup+0x1e): undefined reference to `dbl(int)'
    collect2: error: ld returned 1 exit status
```

- linker sees a reference to `dbl(int)` in the code of `main.o`
- but can't find any object code for it
- `dbl.o` contains the object code for `dbl(int)`

➔ **we need to link `dbl.o` as well**

Linking

```
$ g++ -c main.cpp
$ g++ -c dbl.cpp
$ g++ main.o dbl.o -o prog
```

`dbl.o` contains

- object code for `dbl(int)`

`main.o` contains

- object code for `main()`
- object code for iostream stuff (`cout`)

External Linkage

- **Symbol:** function or variable
- **External Linkage:** symbol *declared* in current TU, but possibly *defined* elsewhere
- functions have external linkage by default

foo.h	tu1.cpp	tu2.cpp

`//declaration`
`void foo();`
	`//foo declaration`
	`#include "foo.h"`
	
	`//definition`
	`void foo() {`
	`  …`
	`}`
		`//foo declaration`
		`#include "foo.h"`
		
		`void bar() {`
		`  …`
		`  //call`
		`  foo();`
		`  …`
		`}`

External Linkage

foo *defined* in header included by 2 TUs
⇒ **foo** appears in 2 TUs
⇒ machine code for **foo** is generated **2 times**
⇒ **linker error** (if tu1.o and tu2.o shall be linked together)

foo.h	tu1.cpp	tu2.cpp

`//external linkage`
 	
`//definition`
`void foo() {`
 	
`  //code…`
 	
`}`
	`#include "foo.h"`
	
	`void bar() {`
	`  …`
	`  foo();`
	`  …`
	`}`
		`#include "foo.h"`
		
		`void baz() {`
		`  …`
		`  foo();`
		`  …`
		`}`

Internal Linkage

= symbol defined and only visible within current TU

- ***unnamed* namespace** or keyword **inline**
- compiler generates individual **foo** for every TU
- 2 different functions ⇒ tu1.o and tu2.o can be linked together!

a.h	tu1.cpp	tu2.cpp

`//internal linkage`
`namespace {`
 	
`void foo() {`
`  …`
`}`
 	
`}`
	`#include "a.h"`
	
	`void bar() {`
	`  …`
	`  foo();`
	`  …`
	`}`
		`#include "a.h"`
		
		`void baz() {`
		`  …`
		`  foo();`
		`  …`
		`}`

`static const` Data Members

have **external** linkage

⇒ you should provide separate declarations and definitions

"widget.h"	"widget.cpp"

`class Widget {`
`public:`
`  …`
 	
`  //declaration`
 	
`  static int const x;`
 	
`  static std::string const s;`
 	
`private:`
`  …`
`};`
	`//definition`
	
	`int const Widget::x = 42;`
	
	`std::string const s = "47";`

`static inline` Data Members

C++17

have **internal** linkage

```
class Widget {
public:
   …

   static inline int const x = 42;

   static inline std::string const s = "47";

private:
   …
};
```

`static constexpr` Members

C++11

- can be used to provide **compile-time** constant values
- no linkage hassle

```
class Widget {
public:
   //OK
   static constexpr int x = 42;

   //probably a more robust interface than a named constant:
   static constexpr int y() { return 42; }
   …
private:
   …
};
```

Make

- used to automate the build process
- processes **makefiles**
- default: looks for a file named "Makefile"

```
> make prog



> ./prog
```

- read makefile
- compile & link → program *prog*
- execute *prog*

Makefiles

- contain instructions for compiling, linking and more
- encode dependencies between files like source and object files
- dependencies are checked via edit date/time

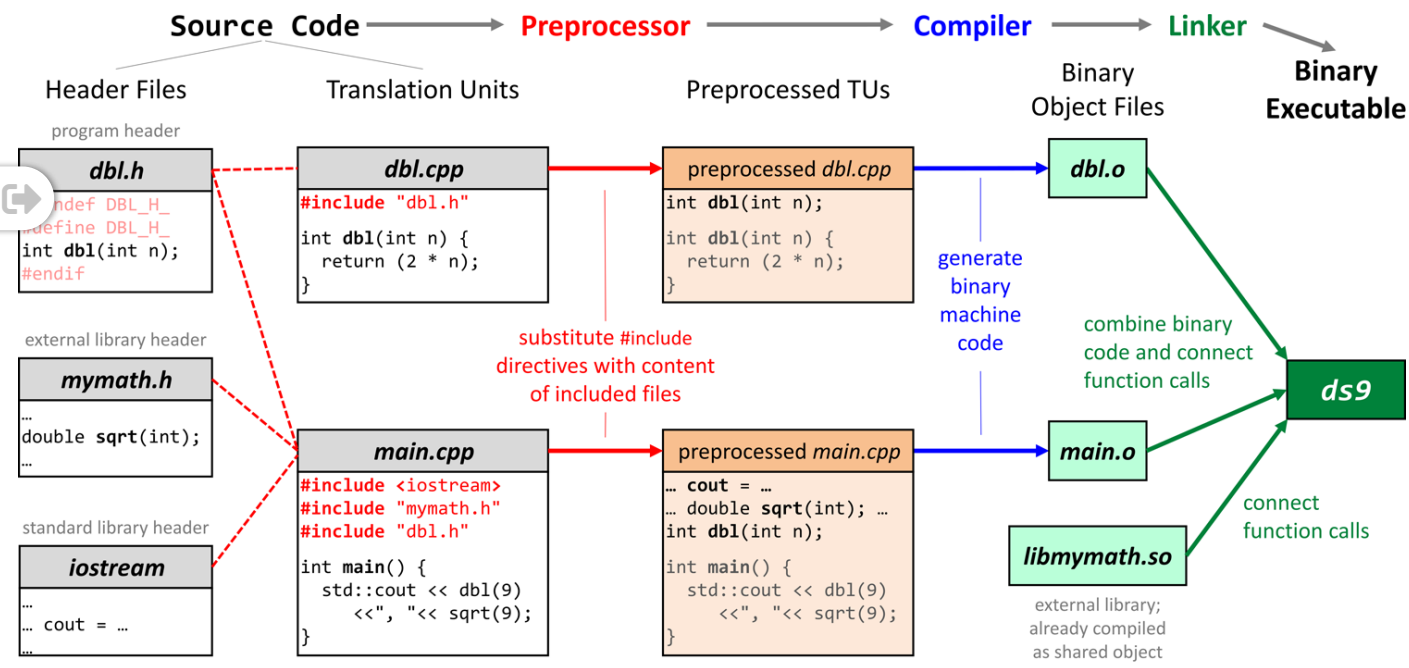
Makefile	Explanation

`#comment`
 	
`dbl.o` `: dbl.cpp`
`<TAB>g++ -c dbl.cpp`
 	
`main.o` `: main.cpp`
`<TAB>g++ -c main.cpp`
 	
`prog:` `main.o dbl.o`
`<TAB>g++ main.o dbl.o -o prog`
	`dbl.o older than dbl.cpp ?`
	`→ compile dbl.cpp → dbl.o`
	
	`main.o older than main.cpp ?`
	`→ compile main.cpp → main.o`
	
	`prog` `older than main.o` **`or`** `dbl.o?`
	`→ link main.o and dbl.o`
	`→ executable` *`prog`*

Useful *g++* Flags

Flag	Meaning
`-s`	omit symbol table from executable
`-D MACRO`	define macro "MACRO"
`-I path`	add include path
`-l lib`	link external library "lib"
`-L path`	path for external libraries
`-S`	print assembly code to file
`-pg`	add profiling information for *gprof*

Diagram




```
$ g++ -c main.cpp
$ g++ -c dbl.cpp
$ g++ main.o dbl.o -l mymath -o ds9
$ ./ds9
$ 18, 3
```

preprocessing & compiling ***main.cpp*** yields ***main.o***
preprocessing & compiling ***dbl.cpp*** yields ***dbl.o***
linking ***main.o***, ***dbl.o*** and library ***mymath*** yields executable ***ds9***
run program ***ds9***
program output

Build Systems

← Random Beginner's Guide / Organization Namespaces →

Related

- Macro __cplusplus
- Beginner's Guide to Linkers
- Toolchains (Compilers, Linkers, etc.) Explained
- Compiling and Linking (by Ben Saks)

Linkage (CopperSpice C++)

Inline Namespaces (CopperSpice C++)

C++ article beginner-level header-files linker toolchain

Last updated: 2019-10-20

Found this useful? Share it:

Comments

Tags

algorithms allocators arrays article beginner-level blogs books build-systems C++ C++-standardization C++11 C++14 C++17 C++20 C++98 C-style C-vs-C++ casts classes code-editors code-formatters code-formatting command-line community comparisons compilers concepts conferences const constexpr containers control-flow CUDA custom-types data-structures debugging design diagnostics exceptions file-io find find_if functional-prog functions gallery generic-prog groups guide guidelines hash-map hash-set hashing header-files heap ides idiom initialization input io iostreams iterators lambda language-mechanism language-references learning libraries library linker list low-level map memory modern-C++ move-semantics news oop organizations output package-manager paradigm pattern people performance podcasts pointers preprocessor profiling Python randomness ranges recipe references set social-media stack standardization std-algorithms std-containers std-library std-macros std-vector STL strings style taste templates testing toolchain tools traversal types user version-control views VIM VIM-plugins warnings websites