Contemporary C++: Learning Modern C++ in a Modern Way

الماس فناوری ابری پاسارگاد- آلفا

مدرس: سعید امراللهی بیوکی

Agenda 6/24

Session 6. Introduction to user-defined types: structures, enumerations and introduction to separate compilations

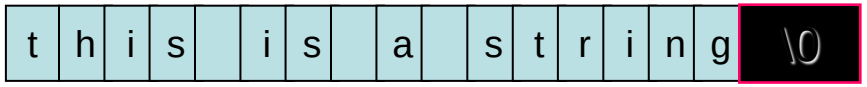
- Structures
- Enumeration and C++11 enumerations classes
- The C preprocessor and header files
- Separate compilation: one-definition rule, header files and source files
- Separate compilation: The stack example
- Q&A

150 min (incl. Q & A)

String literals

• A *string literal* is a character sequence enclosed within double quotes:

"*t h i s i s a s t r i n g*"




```
char v[17];      int *v = "this is a string";
```

• The type of a string literal is "array of the appropriate number of const characters."

• A string literal can be assigned to a char*.

```
char* p = "Hello";
"" // empty string: const char[1]
```

```
wchar_t* p = L"Hello";
L"" // empty string: const wchar_t[1]
```

• Escape characters:

Name	ASCII name	C++ name
newline	NL (LF)	\n
horizontal tab	HT	\t
vertical tab	VT	\v
backspace	BS	\b
alert	BEL	\a
question mark	?	\?
single quote	‘	\’
double quote	“	\”

آلفا ALPHA

Writing simple functions

```
/* atoi: convert s to integer */
int atoi(const char s[])
{
  int i, n;
  n = 0;
  for (i = 0; s[i] >= '0' && s[i] <= '9'; ++i)
    n = 10 * n + (s[i] - '0');

  return n;
}
```

```
/* strlen: return length of string s */
int strlen(const char *s)
{
  int n;
  for (n = 0; *s != '\0', s++)
    n++;

  return n;
}
```

Writing simple functions cont.

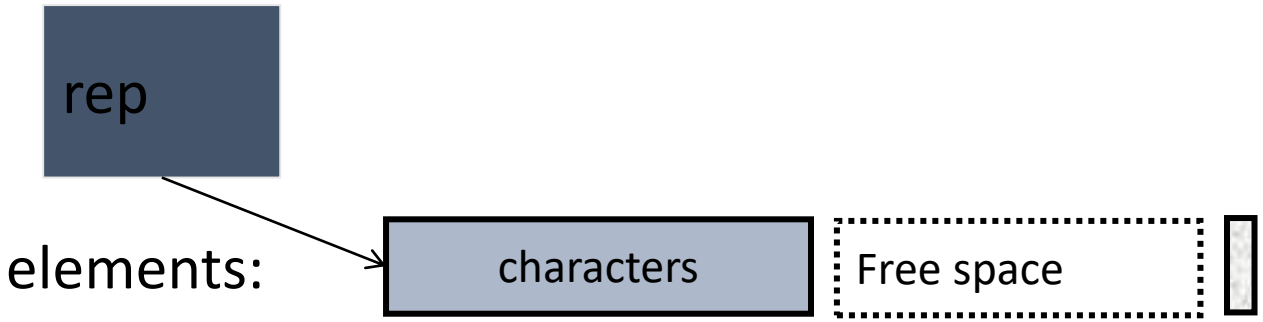
```
/* strcpy: copy t to s; array subscript version */
void strcpy(char *s, const char *t)
{
  int i;
  i = 0;
  while ((s[i] = t[i]) != '\0')
  i++;
}
```

```
/* strcpy: copy t to s; pointer version */
void strcpy(char *s, char *t) // average
{
  int i;
  i = 0;
  while ((*s = *t) != '\0') {
    s++;
    t++;
  }
}
```

- String concatenation

String

- A string might be implemented:
  - For short strings that characters are stored in the string handle itself, and
  - For longer strings the elements are stored contiguously on the free-store (like vector elements)



String manipulation: writing simple classes

- String concatenation

- String comparison

- Delete repeated words

C-style String literal vs. string

آلفا
ALPHA

C-style String literal vs. string

C-style character array

C++ standard string

آلفا
ALPHA

C-style String literal vs. string

C-style character array

From day one

C++98

C++ standard string

C-style String literal vs. string

C-style character array

From day one

Zero-terminated string literal

std::string object

C++98

C++ standard string

آلفا
ALPHA

C-style String literal vs. string

C-style character array

From day one

Zero-terminated string literal

The language core

Standard library: <string>

std::string object

C++98

C++ standard string

آلفا ALPHA

C-style String literal vs. string

C-style character array

From day one

Zero-terminated string literal

The language core

Compile-time constant size

Run-time variable size

Standard library: <string>

std::string object

C++98

C++ standard string

آلفا ALPHA

C-style String literal vs. string

C-style character array

From day one

Zero-terminated string literal

The language core

Compile-time constant size

Size unaware

[ ], pointer arithmetic

[ ]

Size-aware

Run-time variable size

Standard library: <string>

std::string object

C++98

C++ standard string

آلفا ALPHA

C-style String literal vs. string

C-style character array

From day one

Zero-terminated string literal

The language core

Compile-time constant size

Size unaware

[ ], pointer arithmetic

Automatic storage

Dynamic storage

[ ]

Size-aware

Run-time variable size

Standard library: <string>

std::string object

C++98

C++ standard string

آلفا ALPHA

C-style String literal vs. string

C-style character array

From day one

Zero-terminated string literal

The language core

Compile-time constant size

Size unaware

[ ], pointer arithmetic

Automatic storage

Fixed size/no
grow and shrink

Variable size/grow
and shrink

Dynamic storage

[ ]

Size-aware

Run-time variable size

Standard library: <string>

std::string object

C++98

C++ standard string

آلفا ALPHA

C-style String literal vs. string

C-style character array

From day one

Zero-terminated string literal

The language core

Compile-time constant size

Size unaware

[ ], pointer arithmetic

Automatic storage

Fixed size/no grow and shrink

Implicit decay of name to pointer/pointer arithmetic

C++ standard string

No implicit decay of name to pointer/pointer arithmetic

Variable size/grow and shrink

Dynamic storage

[ ]

Size-aware

Run-time variable size

Standard library: <string>

std::string object

C++98

آلفا ALPHA

C-style String literal vs. string

C-style character array

From day one

Zero-terminated string literal

The language core

Compile-time constant size

Size unaware

[ ], pointer arithmetic

Automatic storage

Fixed size/no grow and shrink

Implicit decay of name to pointer/pointer arithmetic

C++ standard string

No implicit decay of name to pointer/pointer arithmetic

Variable size/grow and shrink

Dynamic storage

[ ]

Size-aware

Run-time variable size

Standard library: <string>

std::string object

C++98

آلفا ALPHA

C-style array of chars vs. standard library string : comparison

- char string is defined in core language, but string is defined in standard library.
- The size of char string can be determined using null character/strlen() function, but the size of string can be determined using size() member function.
- char string is not generic, but standard string are generic.
- In array of char, we may be need to pointer arithmetic but in string we don’t.
- In array of char, we may be need to memory management but in string we don’t.

☞

Use string rather than zero-terminated arrays of char.

Structures

- An array is an aggregate of elements of the same type.

  - Homogeneous

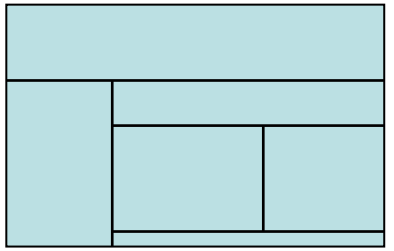
V:

- A struct is an aggregate of elements of arbitrary types.

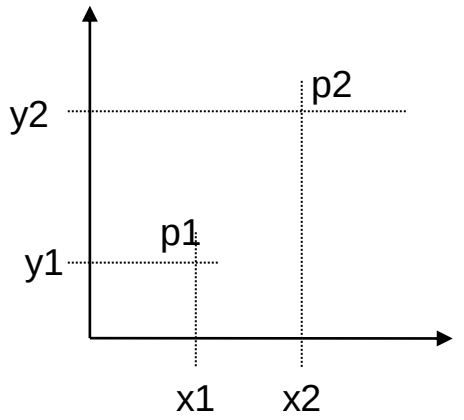
  - Heterogeneous

- Data abstraction: Ignore irrelevant and unimportant things.

S:



```
// represent point with two coordinates
struct Point {
  int x, y; // members
};
```



- A struct is a scope.

```
struct Point {
  int x, y; // local scope
};
int x; // another scope
```

Structures cont.

```
// type definition
struct Point {
  int x, y;
};
```

```
// object creation: memory allocation
struct Point {  int x, y;} p1, p2; // two point objects
int                        a, b;
Point p3, p4; // two other points
```

- Struct initializer:

```
Point Center = {0, 0};
Point Max = { 1000, 1000 };
```

- Member selection operator or . (dot) operator.

structure-name . member

```
Point Center;
Center.x = 0;
Center.y = 0;

Point Max;
Max.x = Max.y = 1000;
```

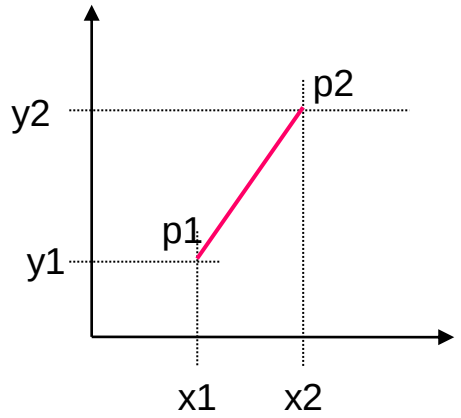
```
struct address {
  char* name; // "Ali Hasani"
  int number; // 61
  char* street; // "Valiasr"
  char* town; // "Tehran"
  long zip; // 79745
};
```

```
address ah;
ah.name = "Ali Hasani";
ah.number = 61;
// other members are uninitialized
```

Structures cont.

- Structures can be nested.

```
struct Line {
  Point pt1;
  Point pt2;
};
Line L;
```



- Simple functions:

```
/* makepoint: make a point from x and y components */
Point MakePoint(int x, int y)
{
  Point temp;
  temp.x = x;
  temp.y = y;

  return temp;
}
const int XMAX = 1000;
const int YMAX = 1000;
Point p1 = MakePoint(0, 0);
Point p2 = MakePoint(XMAX, YMAX);
Point Middle = MakePoint((p1.x + p2.x) / 2, (p1.y + p2.y) / 2);
```

Pointer to structures

```
Point p = MakePoint(2, 3);
Point* pp = &p;
int xx = (*pp).x, yy = (*pp).y; // good
```

it is p

- Structure pointer dereference operator: ->

structure-name -> member

```
Point p = MakePoint(2, 3);
Point* pp = &p;
int xx = pp->x, yy = pp->y; // better
```

```
void print_name_and_family(address* p)
{
  // just print name and family
  cout << p->name << '\n' << p->family << '\n';
}
```

- Another example:

```
struct student_info {
  string name;
  double midterm, final;
  vector<double> homework;
};
student_info myInfo;
```

Enumeration

- An enumeration is a type that can hold a set of values specified by the user.

$$enum\ Tag\_name_{op}\ \{\ enumerator_1 = init_{op},\ enumerator_2 = init_{op},\ \ldots,\ enumerator_n = init_{op}\ \};$$

```
enum { dark, light };
enum Season { WINTER, SPRING, SUMMER, FALL };
enum Color { RED, GREEN, BLUE };
enum Bool { False, True }; // redundant
```

- The elements of an enumeration called enumerator. They are symbolic integral constants. They are r-value. Unlike constants, they are not addressable.

- By default, enumerator values are assigned increasing from 0.

```
enum { dark, light }; // dark == 0, light = 1
enum Season { WINTER = 1, SPRING, SUMMER, FALL }; // SPRING == 2, …
enum Color { RED = 0, GREEN = 2, BLUE = 4 };
```

- Each enumeration is distinct type.

```
Season mySeason = SUMMER;
Color c = YELLOW; // error
Color c = SUMMER; // error
Bool b = false; // error: can't convert from bool to Bool
```

- Enumerations are converted to integers for arithmetic operations.

Enumeration classes

- Conventional enumerations

```
enum Season { WINTER, SPRING, SUMMER, FALL };
enum TrafficLight { RED, YELLOW, GREEN };
enum Color { Red, Green, Blue };
Color c = RED; //error: two different enumerations
Season s = 3; // I mean FALL! error: no conversion from int to enum
int color_as_int = Blue; // implicit conversion
```

- Conventional enumerations have three problems:
  1. implicit conversion to int
  2. conventional enums export their enumerators to the surrounding scope, causing name clashes.
  3. the underlying type of an enum cannot be specified.
- enumeration classes are *strongly-typed* and *scoped*.

```
enum class TrafficLight { Red, Yellow, Green };
enum class Color { Red, Green, Blue };
TrafficLight CrossStreet = TrafficLight::Red;
Color RGB = Color::Red;
int c = RGB; // error
```

Enumeration classes continued

- Underlying type
- conventional enums and enumeration classes: compatibility

```
enum class TrafficLight { RED, YELLOW, GREEN }; // the underlying type is int, by default
enum class Color : char { red, blue }; // compact representation
enum class E : long long { E1= 1LL, E2 = 2LL, Ebig = 0xFFFFFFFFFFFFFFFF };
```

- Very important feature in embedded system programming

Preprocessing and Preprocessor

- Preprocessor: the part of a C++ implementation that *removes comments*, performs *macro substitution* and *file inclusion*.

C Preprocessor

- C Provides certain language facilities by means of a *preprocessor*, which is conceptually a separate first step in compilation.
- Actually C preprocessor is a system software which is usually called *Macro processor*. Briefly It is called *Cpp*.
- Cpp is character and file oriented.
- Cpp has three fundamental features:
  - File inclusion
  - Macro substitution: Expand Inline
  - Conditional Inclusion

```
#include <stdio.h>
main()
{
  printf("hello, world\n");
}
```

#include "*filename*"

#include <*filename*>

#define *name replacement text*

```
#define PI 3.14159
```

```
#define MAX(A, B) ((A) > (B) ? (A) : (B))
```

```
#ifndef HEADER_FILE
#define HEADER_FILE

#endif // HEADER_FILE
```

```
#ifndef HEADER_FILE
#define HEADER_FILE

#endif // HEADER_FILE
```

Preprocessing considered Harmful

☞

Prefer the compiler to the preprocessor.

Language-technical rule:

Preprocessor usage should be eliminated.

Preprocessor Elimination

- Constant types

```
#define ASPECT_RATIO 1.653
```

Initialization

- Constant must be initialized.

```
const float ASPECT_RATIO = 1.653; // symbolic constant
```

```
ASPECT_RATIO = ASPECT_RATIO + 1.0;
// error: ASPECT_RATIO is not modifiable l-value
```

- Inline functions

```
#define MAX(A, B) ((A) > (B) ? (A) : (B))
```

```
int Max(int a, int b); //decl.
```

```
int a = 1; b = 0;
MAX(a++, b);
```

```
inline int Max(int a, int b) { return a > b ? a : b; } //definition
```

Efficient

Type-safe

C++ Program

- One Definition Rule (ODR): there must be exactly one definition of each entity in a program. If more than one definition appears, say because of replication through header files, the meaning of all such duplicates must be identical.

```
// file1.c:
struct S1 { int a; char b; };
struct S1 { int a; char b; }; // error: double definition
```

```
// file1.c:
struct S2 { int a; char b; };
// file2.c:
struct S2 { int a; charb bb; }; // error
```

- A (C/C++) program is a collection of separately compiled units combined by linker. Every function, object, type, etc. used in this collection must have a unique definition. The program must contain exactly one function called main()

Physical structure of C++ programs

Physical structure of C++ programs

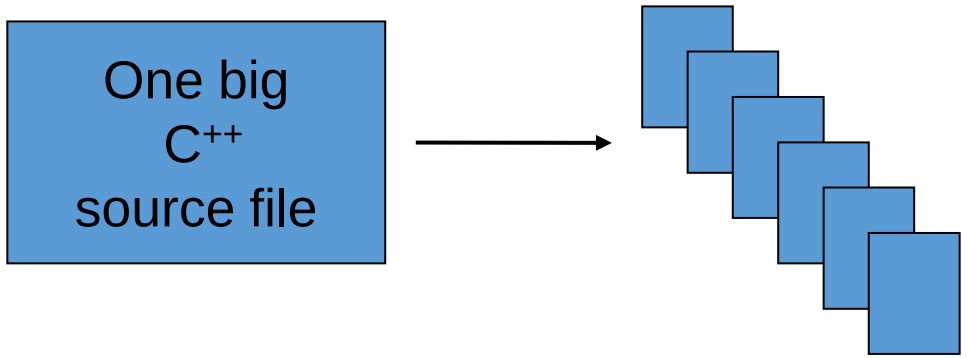
- A file is the traditional unit of storage (in a file system) and the traditional unit of compilation.

Physical structure of C++ programs

- A file is the traditional unit of storage (in a file system) and the traditional unit of compilation.
- Having a complete program in one file is usually impossible.
    - Standard library code
    - Operating system code

Physical structure of C++ programs

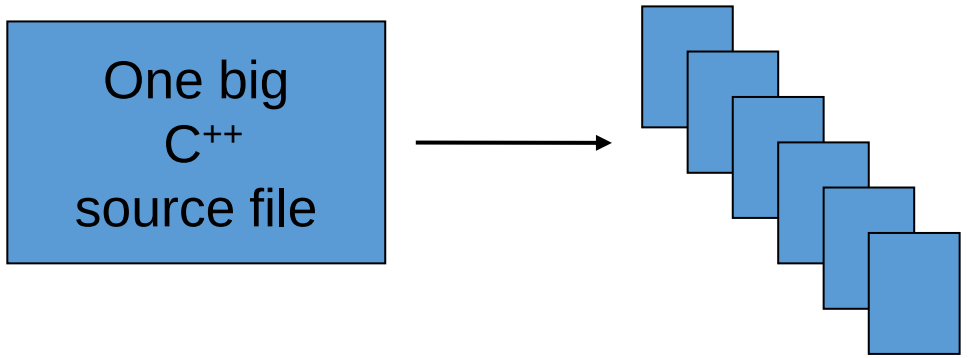
- A file is the traditional unit of storage (in a file system) and the traditional unit of compilation.
- Having a complete program in one file is usually impossible.
  - Standard library code
  - Operating system code



several smaller C++ source files

Physical structure of C++ programs

- A file is the traditional unit of storage (in a file system) and the traditional unit of compilation.
- Having a complete program in one file is usually impossible.
  - Standard library code
  - Operating system code

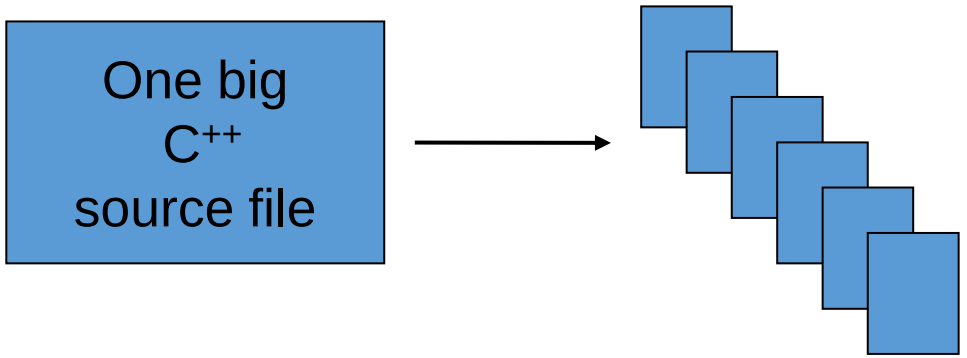


- Reduced compilation time
- More readability

several smaller C++ source files

Physical structure of C++ programs

- A file is the traditional unit of storage (in a file system) and the traditional unit of compilation.
- Having a complete program in one file is usually impossible.
  - Standard library code
  - Operating system code



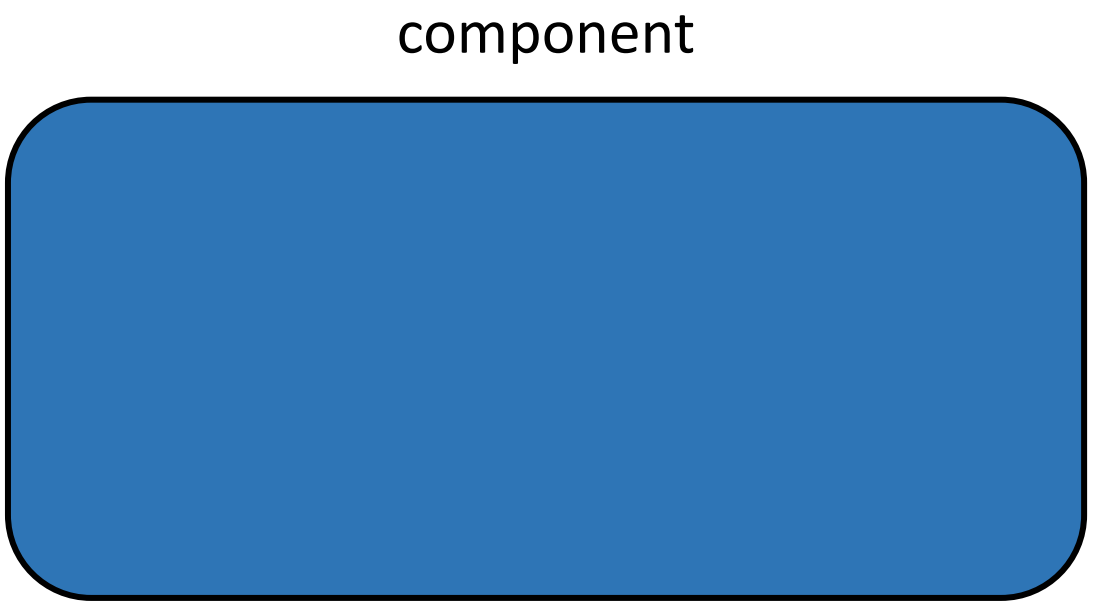
- Reduced compilation time
- More readability

several smaller C++ source files

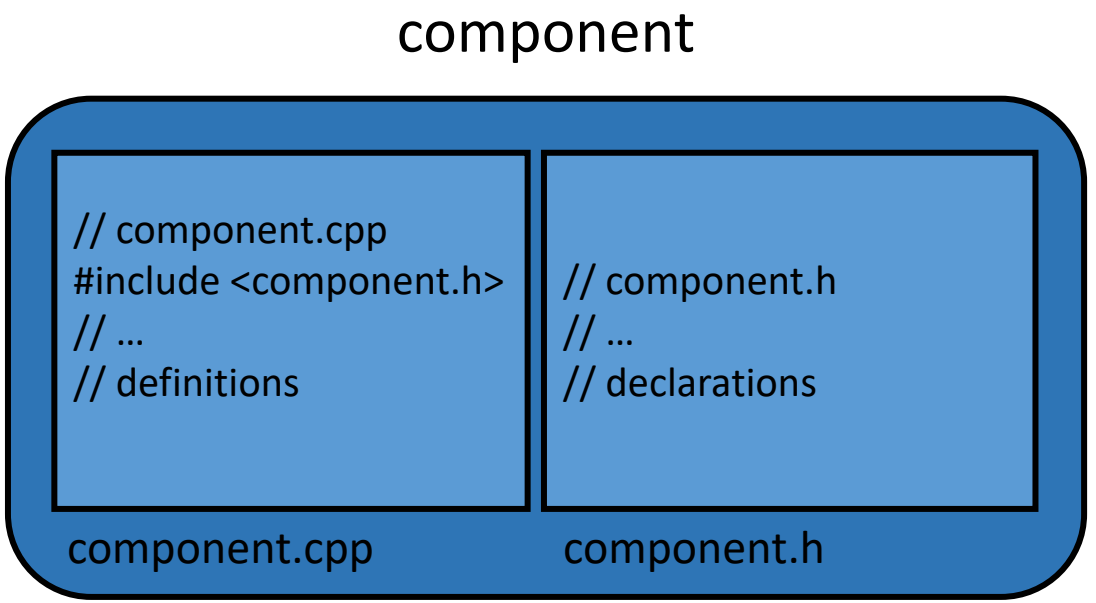
Logical structure ←→ Physical structure

Components

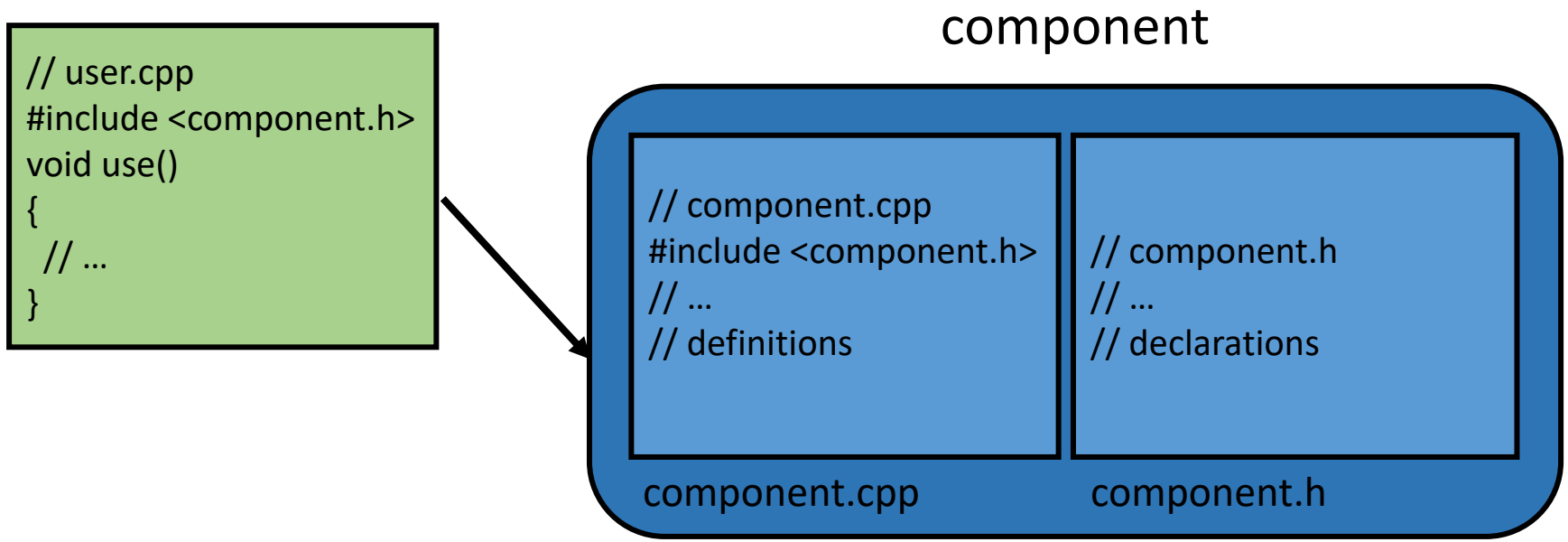
Components



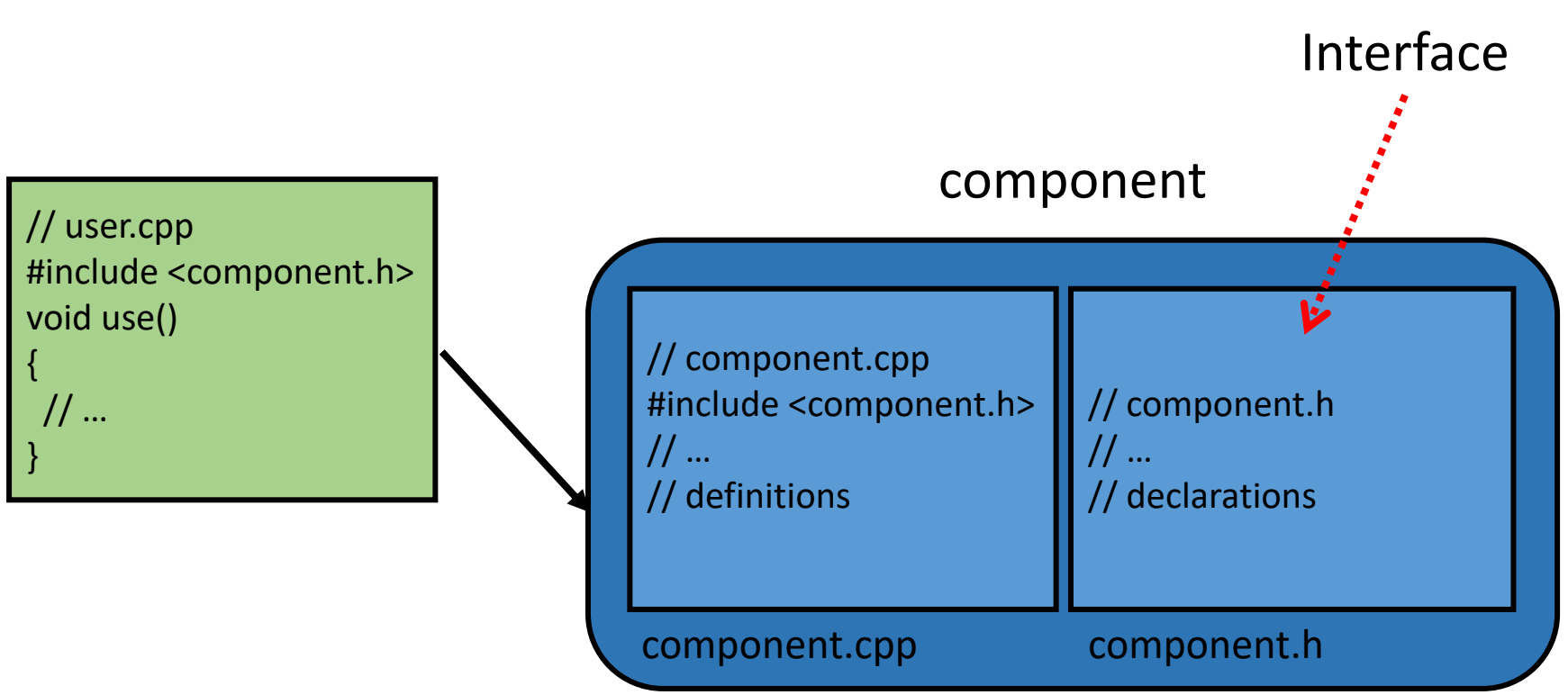
Components


component
// component.cpp
#include <component.h>
// …
// definitions
// component.h
// …
// declarations
component.cpp
component.h

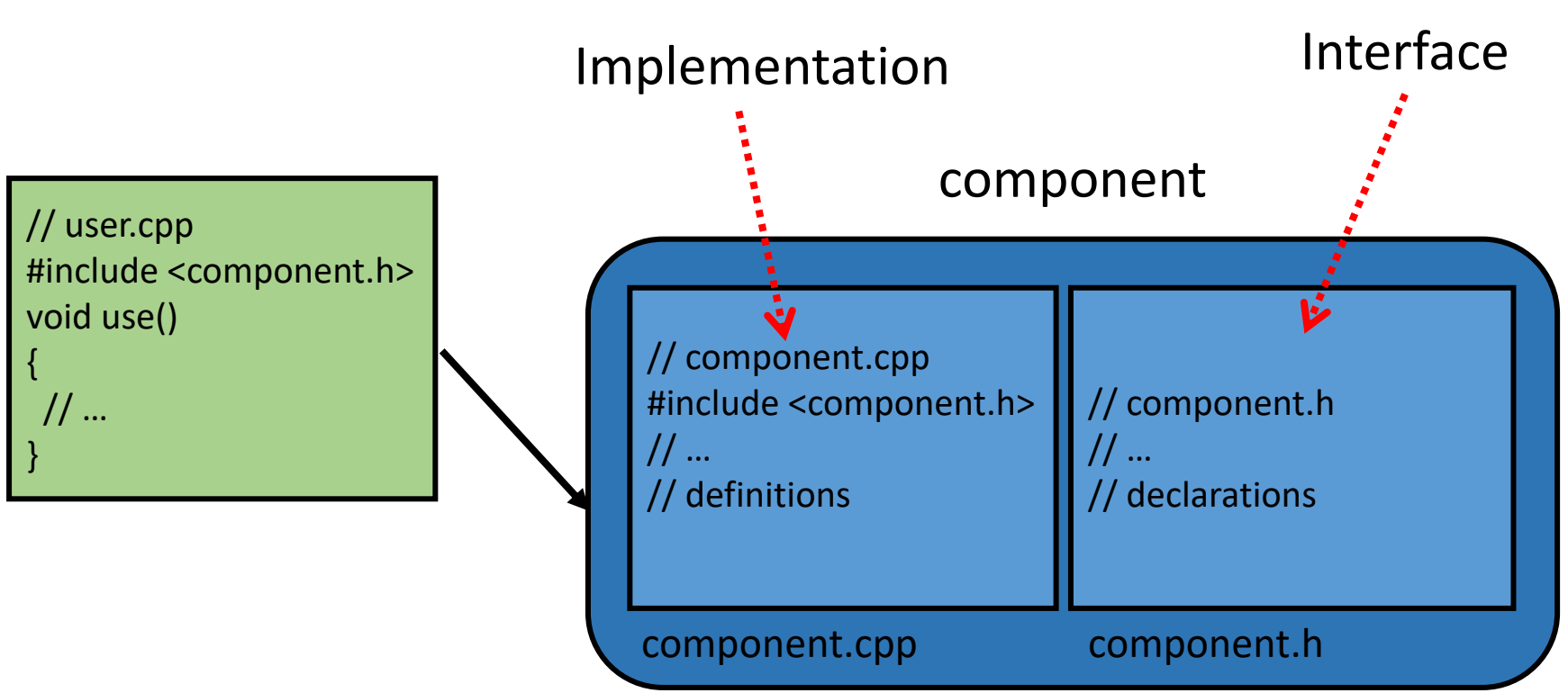
Components


component
// user.cpp
#include <component.h>
void use()
{
// …
}
// component.cpp
#include <component.h>
// …
// definitions
// component.h
// …
// declarations
component.cpp
component.h

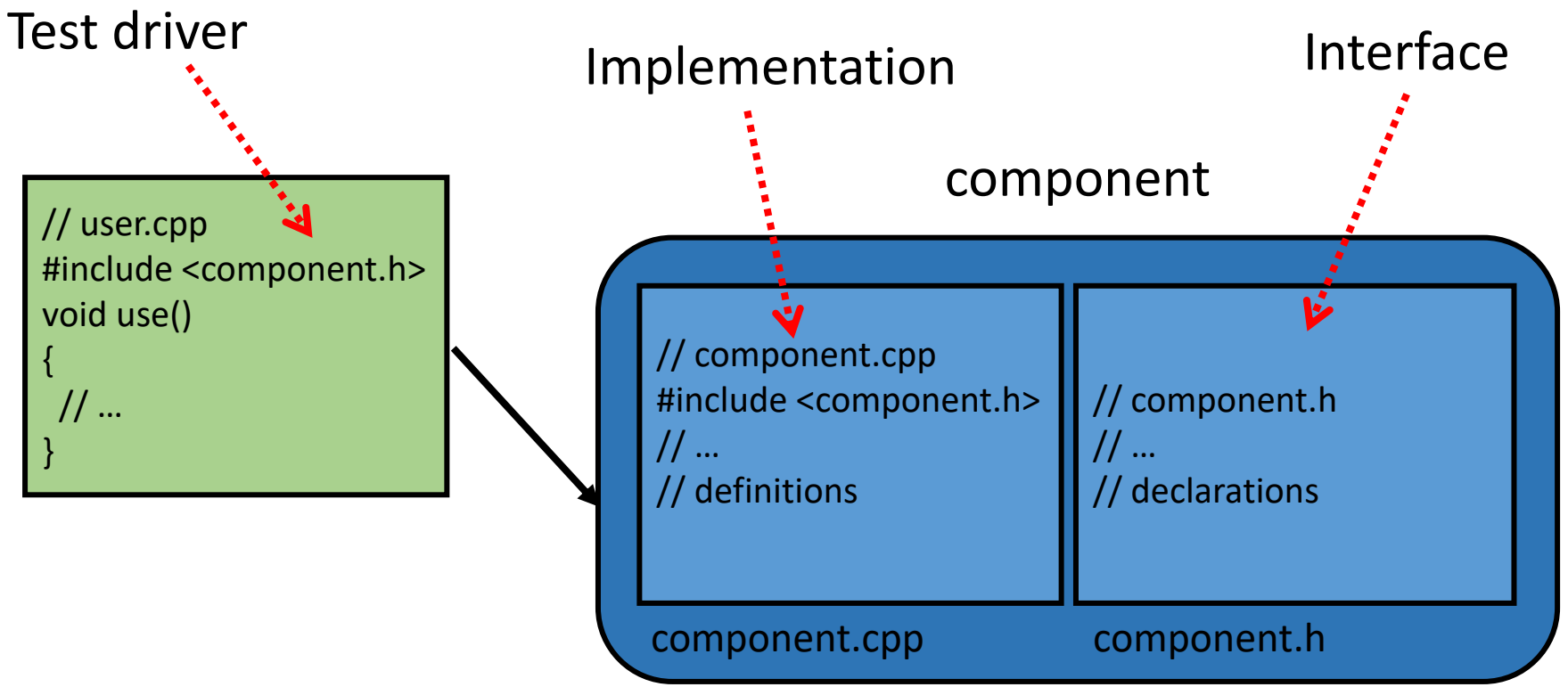
Components


Interface
component
// user.cpp
#include <component.h>
void use()
{
// …
}
// component.cpp
#include <component.h>
// …
// definitions
// component.h
// …
// declarations
component.cpp
component.h

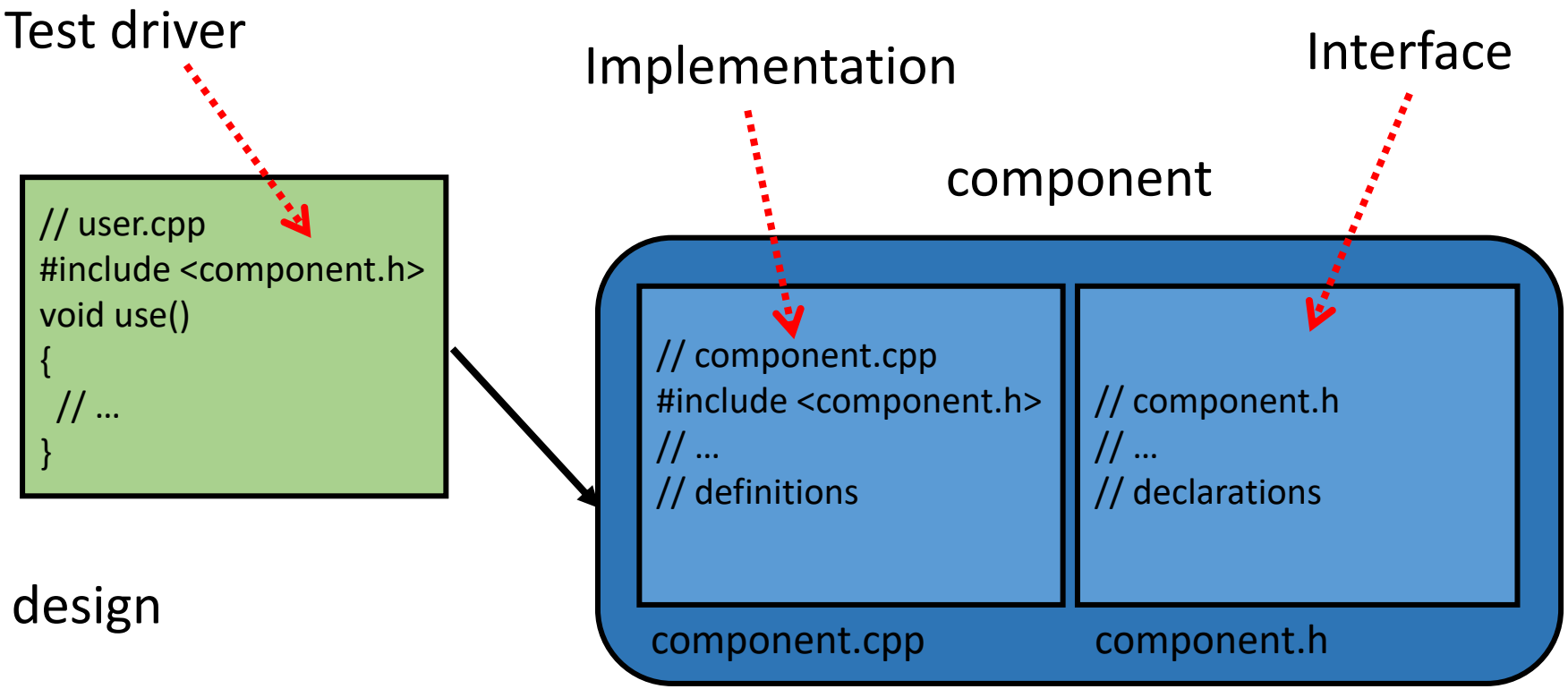
Components


Implementation
Interface
component
// user.cpp
#include <component.h>
void use()
{
// …
}
// component.cpp
#include <component.h>
// …
// definitions
// component.h
// …
// declarations
component.cpp
component.h

Components

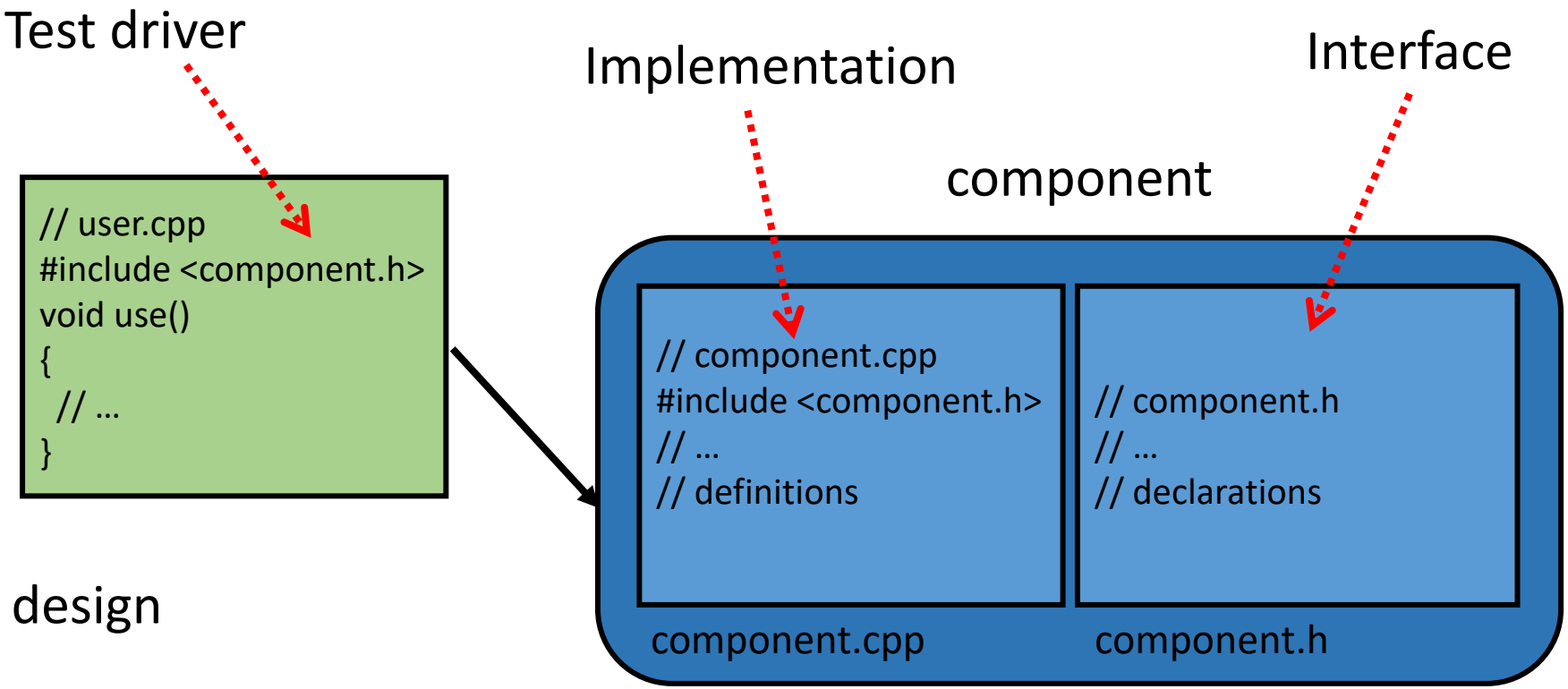

Test driver
Implementation
Interface
component
// user.cpp
#include <component.h>
void use()
{
// …
}
// component.cpp
#include <component.h>
// …
// definitions
// component.h
// …
// declarations
component.cpp
component.h

Components



- The Fundamental unit of design

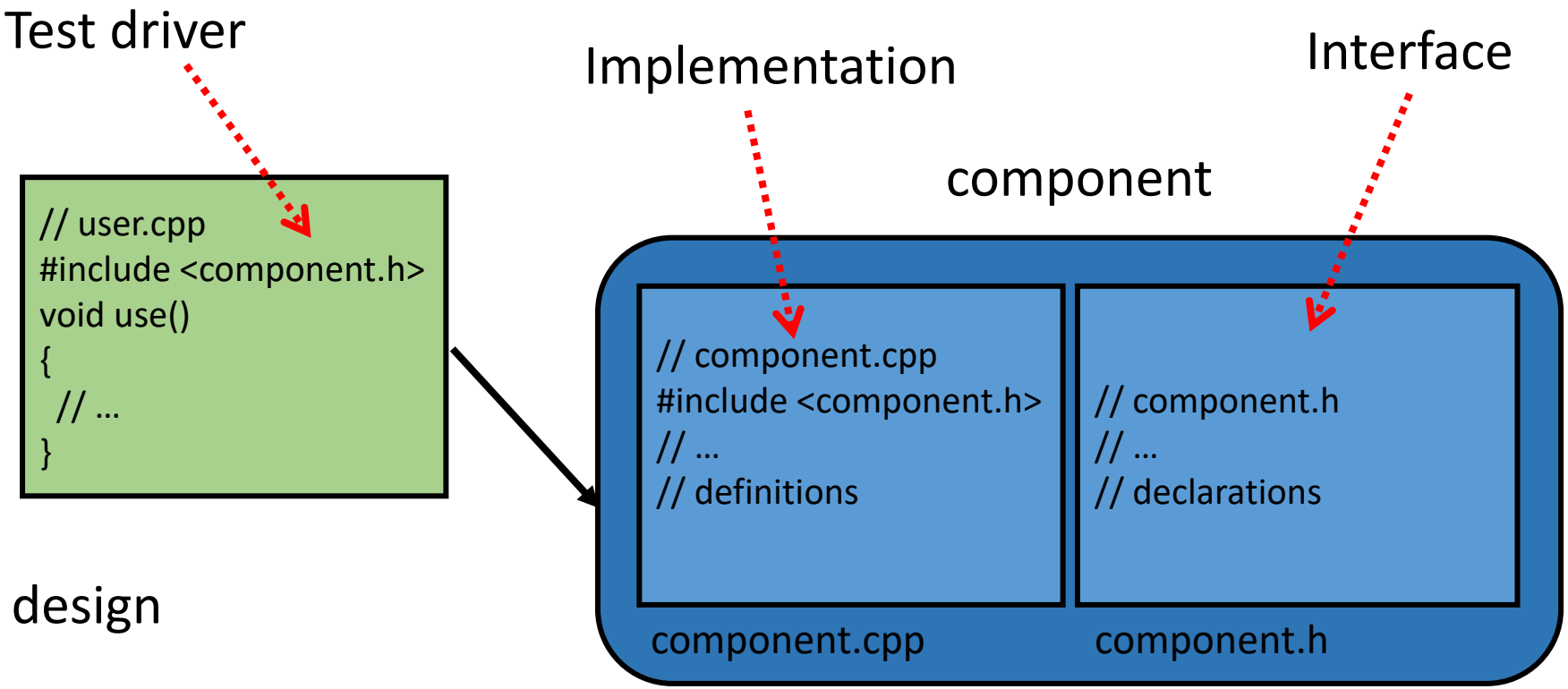
Components



- The Fundamental unit of design

- Physical component vs. Logical component
- Logical vs. physical aspects

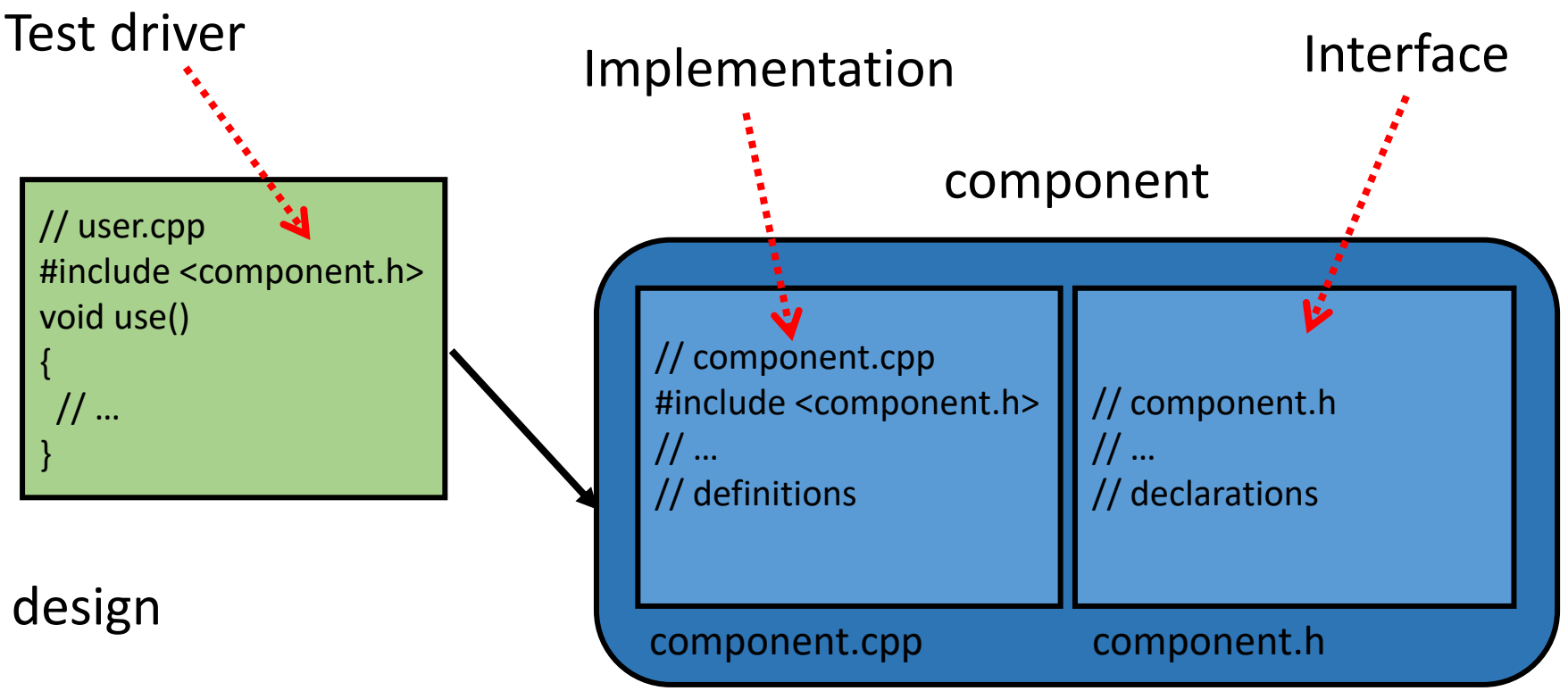
Components



- The Fundamental unit of design

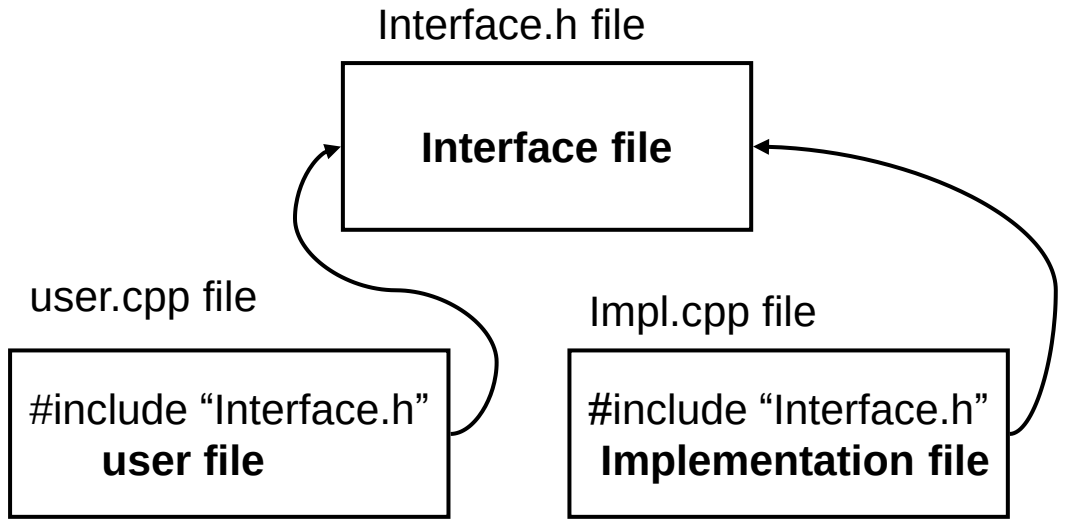
- Physical component vs. Logical component
- Logical vs. physical aspects
- Large-scale C++ software development

Components



- The Fundamental unit of design

- Physical component vs. Logical component
- Logical vs. physical aspects
- Large-scale C++ software development



Thanks for your patience …

A man who asks a question is a fool for minute,
The man who does not ask, is a fool for a life.
- Confucius

Learning to ask the right (often hard) questions is an essential part of learning to think as a programmer.
- Bjarne Stroustrup *programming Principles and Practice Using C++, page 4.*

There is no stupid question, but there is stupid answer.
- Howard Hinnant