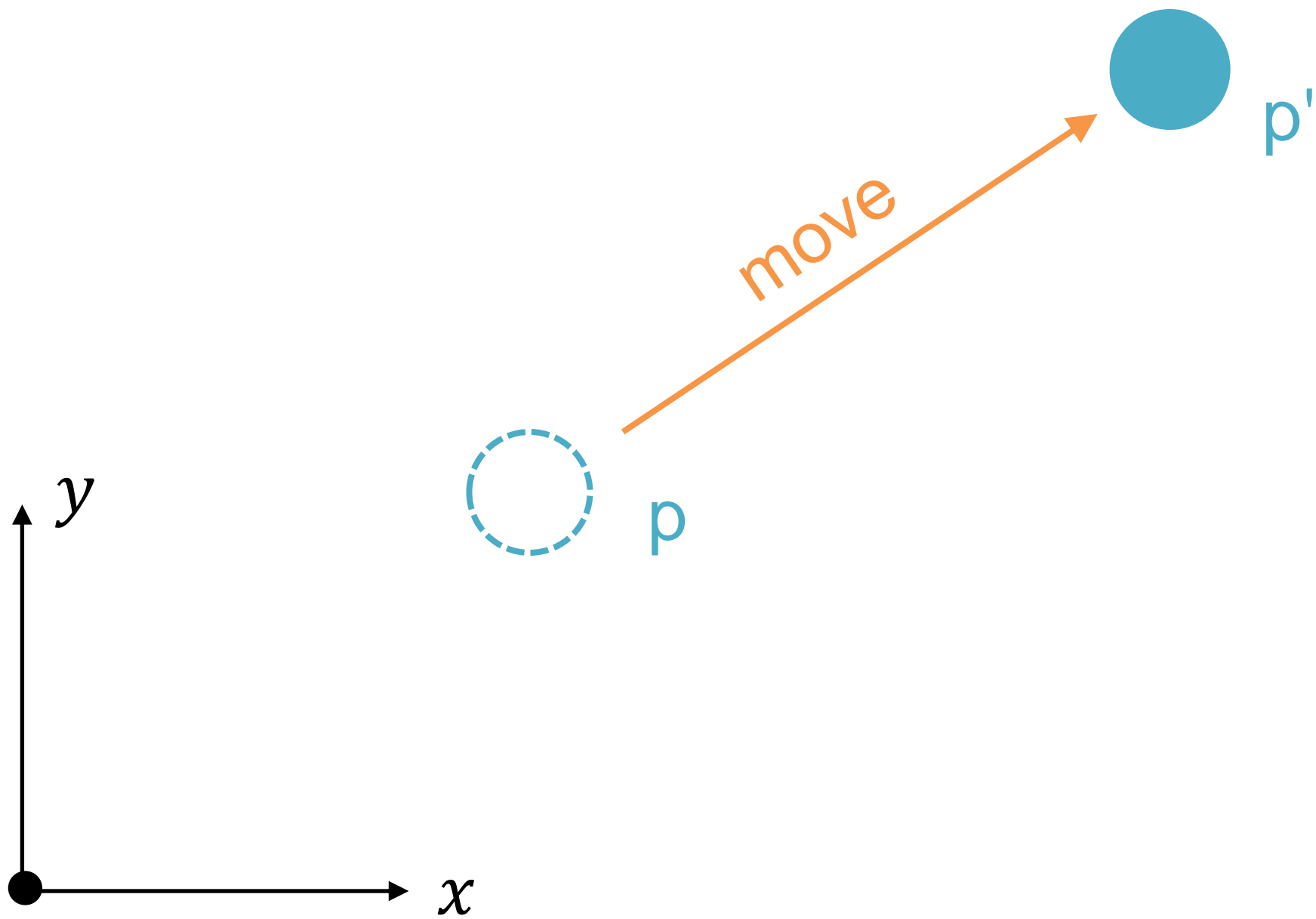


Class

Object-Oriented Programming with C++

Zhaopeng Cui



Point

```
typedef struct point {  
    int x;  
    int y;  
} Point;  
  
Point a;  
a.x = 1; a.y = 2;  
  
void print(const Point* p) {  
    printf("%d %d\n", p->x, p->y);  
}  
  
print(&a);
```

move (dx,dy)?

```
void move(Point* p, int dx, int dy) {  
    p->x += dx;  
    p->y += dy;  
}
```

Prototypes

```
typedef struct point {  
    int x;  
    int y;  
} Point;  
  
void print(const Point* p);  
void move(Point* p, int dx, int dy);
```

Usage

```
Point a;  
Point b;  
  
a.x = b.x = 1;  
a.y = b.y = 1;  
  
move(&a, 2, 2);  
  
print(&a);  
print(&b);
```

C++ version

```
class Point {  
public:  
    void init(int x, int y);  
    void move(int dx, int dy);  
    void print() const;  
  
private:  
    int x;  
    int y;  
};
```

implementations

```
void Point::init(int ix, int iy) {  
    x = ix; y = iy;  
}  
  
void Point::move(int dx, int dy) {  
    x += dx; y += dy;  
}  
  
void Point::print() const {  
    cout << x << ' ' << y << endl;  
}
```


:: resolver

- <Class Name>::<function name>
- ::<function name>

```
void S::f() {  
    ::f(); // Would be recursive otherwise  
    ::a++; // Select the global a  
    a--; // The a at class scope  
}
```

C vs. C++

C

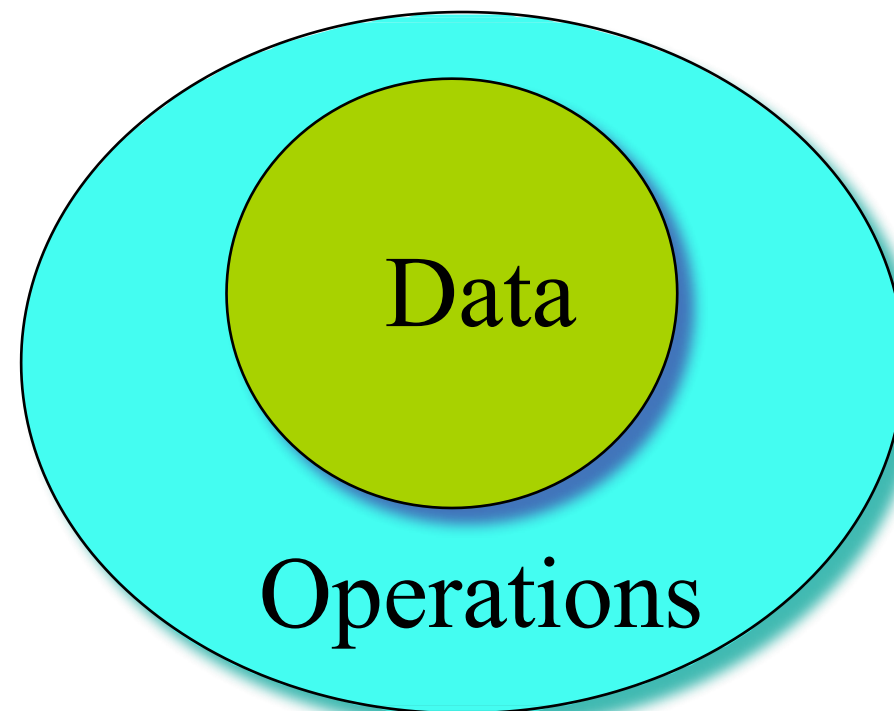
```
typedef struct point {  
    int x;  
    int y;  
} Point;  
  
void print(const Point* p);  
void move(Point* p,  
          int dx, int dy);  
  
Point a;  
a.x = 1;  
a.y = 2;  
move(&a, 2, 2);  
print(&a);
```

C++

```
class Point {  
public:  
    void init(int x, int y);  
    void print() const;  
    void move(int dx, int dy);  
private:  
    int x;  
    int y;  
};  
  
Point a;  
a.init(1, 2);  
a.move(2, 2);  
a.print();
```

Objects = Attributes + Services

- Data: the properties or status
- Operations: the functions



Objects

- In C++, an object is just a variable, and the purest definition is “a region of storage”.

Ticket Machine

- Ticket machines print a ticket when a customer inserts the correct money for their fare.
- Our ticket machines work by customers' inserting money into them, and then requesting a ticket to be printed. A machine keeps a running total of the amount of money it has collected throughout its operation.



Procedure-Oriented

- Step to the machine
- Insert money into the machine
- The machine prints a ticket
- Take the ticket and leave



Procedure-Oriented

- Step to the machine
- Insert money into the machine
- We make a program simulate the procedure of buying tickets. It works. But there is no such machine. There's nothing left for the further development.



Something is there

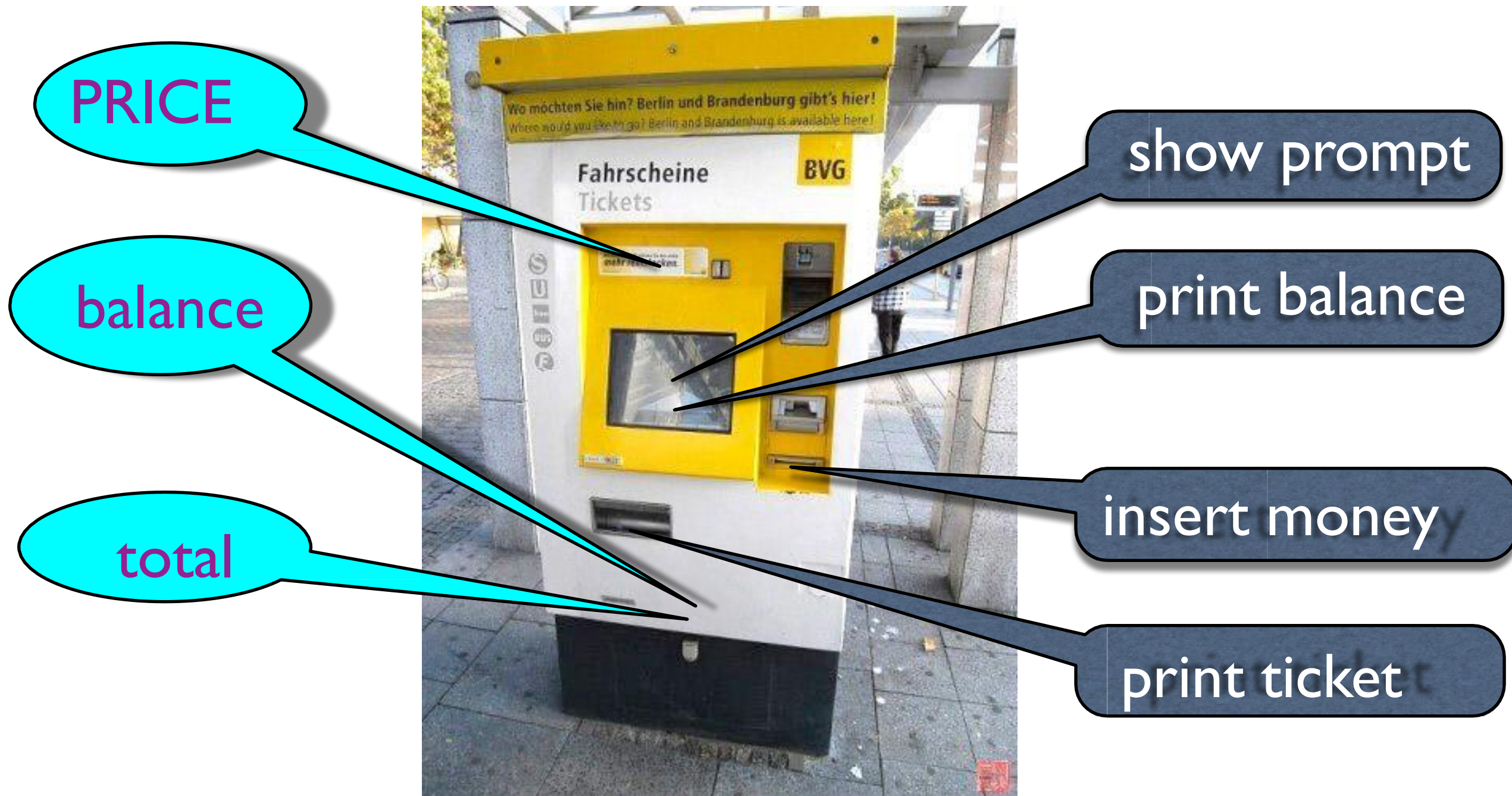
PRICE

balance

total



Something is there



Something is there

TicketMachine	
PRICE	
Balance	
Total	
showPrompt	
getMoney	
printTicket	
showBalance	
printError	

ticketMachine 1:
TicketMachine

price

balance

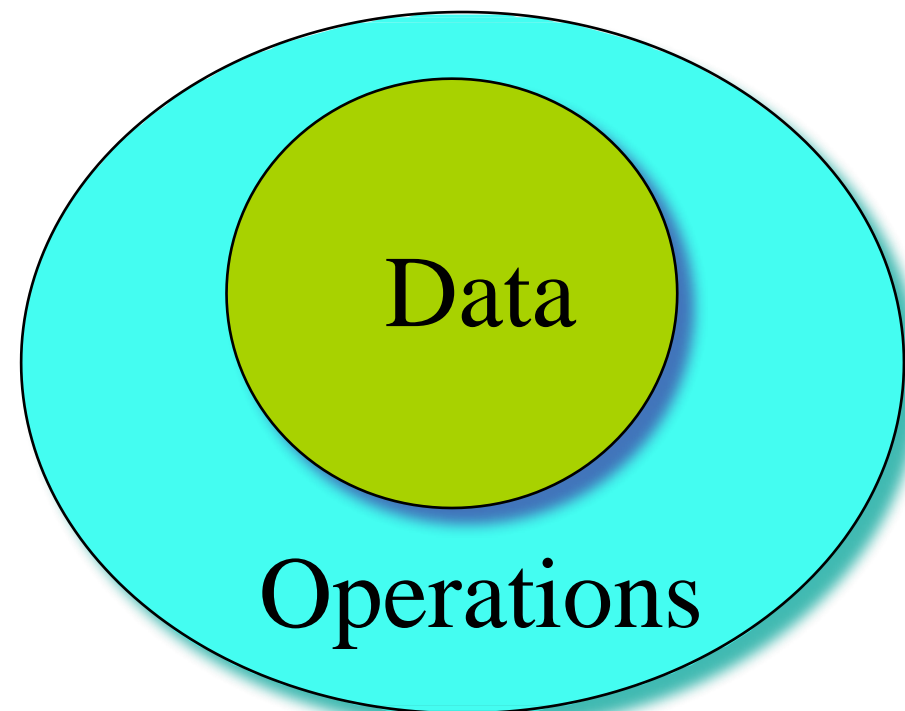
total

Turn it into code

```
class TicketMachine {  
public:  
    void showPrompt();  
    void getMoney();  
    void printTicket();  
    void showBalance();  
    void printError();  
  
private:  
    const int PRICE;  
    int balance;  
    int total;  
};
```

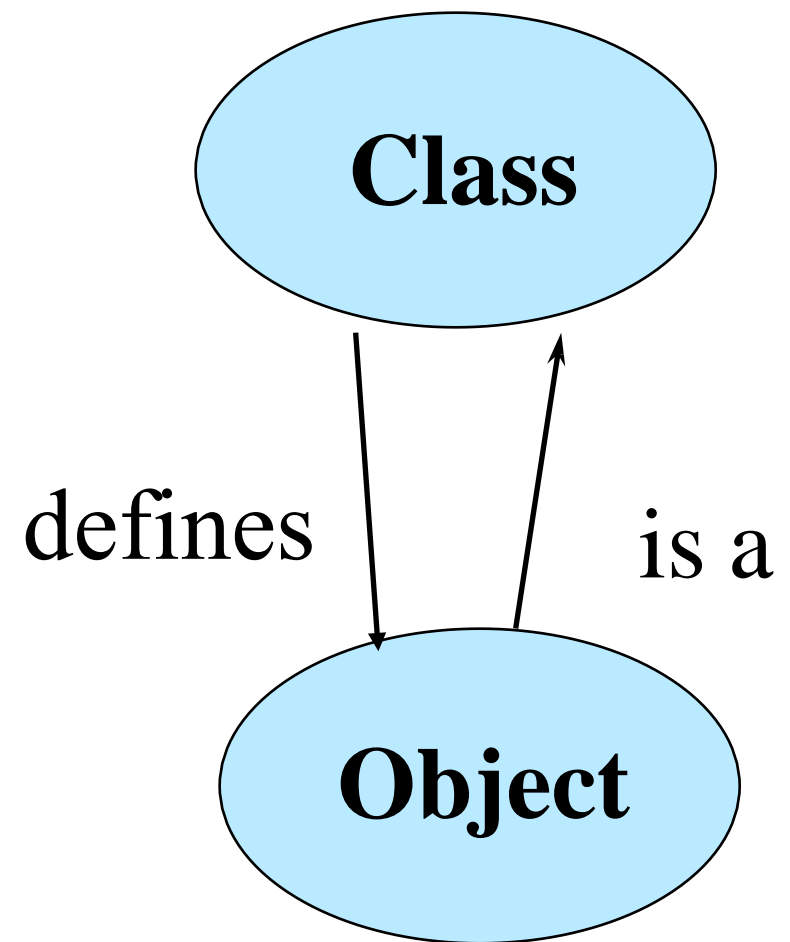
Objects = Attributes + Services

- Data: the properties or status
- Operations: the functions



Object vs. Class

- **Objects (cat)**
 - Represent things, events, or concepts
 - Respond to messages at run-time
- **Classes (cat class)**
 - Define properties of instances
 - Act like types in C++



OOP Characteristics

- Everything is an object.
- A program is a bunch of objects telling each other what to do by sending messages.
- Each object has its own memory made up of other objects.
- Every object has a type.
- All objects of a particular type can receive the same messages.

Definition of a class

- In C++, separated .h and .cpp files are used to define one class.
- Class declaration and prototypes in that class are in the header file (.h).
- All the bodies of these functions are in the source file (.cpp).

Compile unit

- The compiler sees only one .cpp file, and generates .obj file
- The linker links all .obj into one executable file
- To provide information about functions in other .cpp files, use .h

The header files

- If a function is declared in a header file, you must include the header file everywhere the function is used and where the function is defined.
- If a class is declared in a header file, you must include the header file everywhere the class is used and where class member functions are defined.

Header = interface

- The header is a contract between you and the user of your code.
- The compiler enforces the contract by requiring you to declare all structures and functions before they are used.

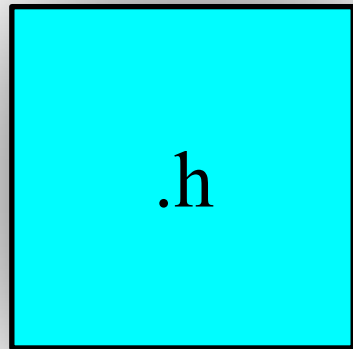
Structure of C++ program



.h

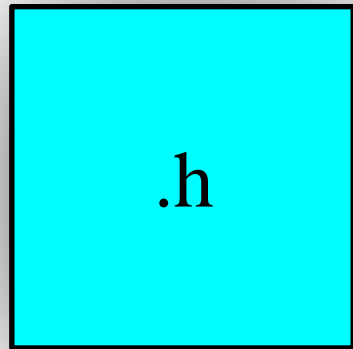
Structure of C++ program

declarations

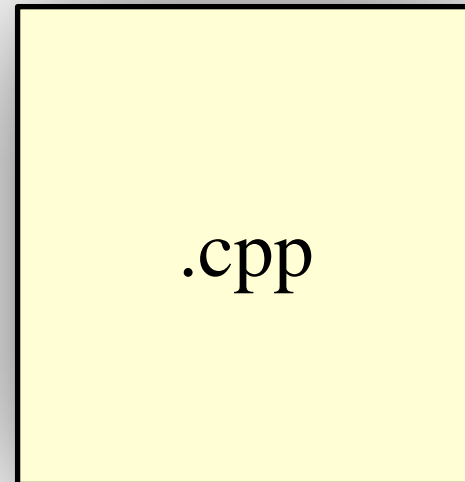


Structure of C++ program

declarations

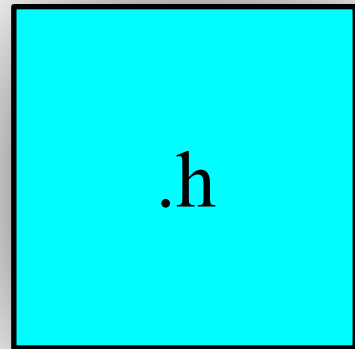


definitions

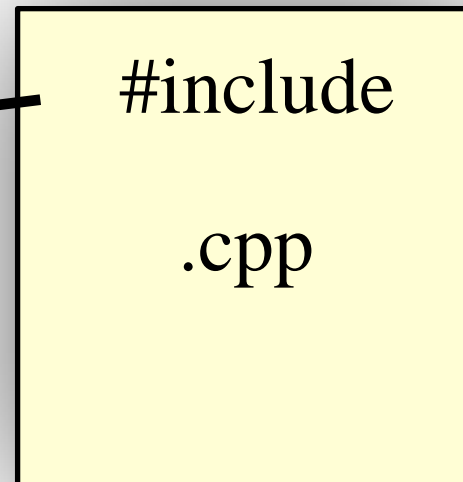


Structure of C++ program

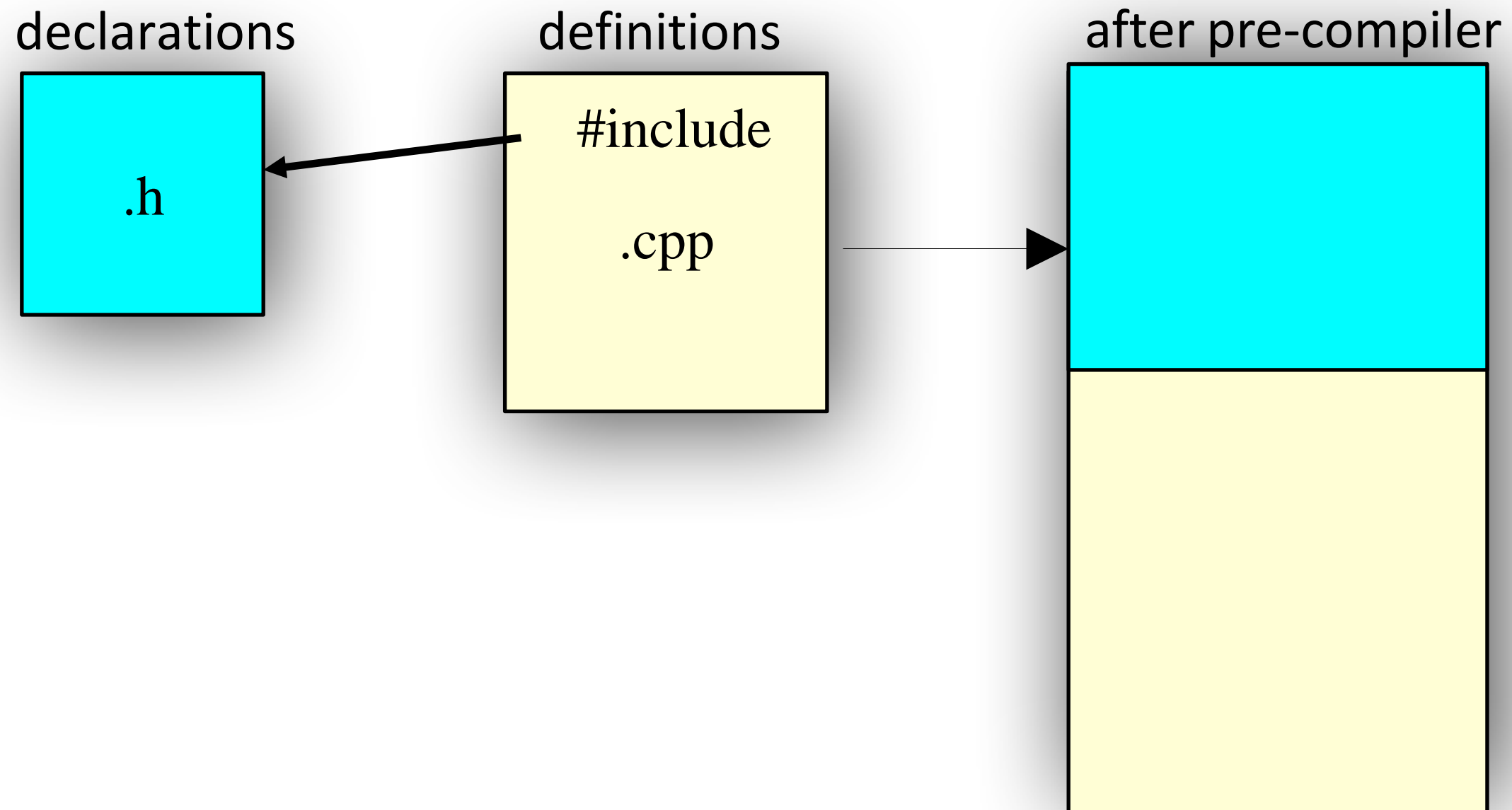
declarations



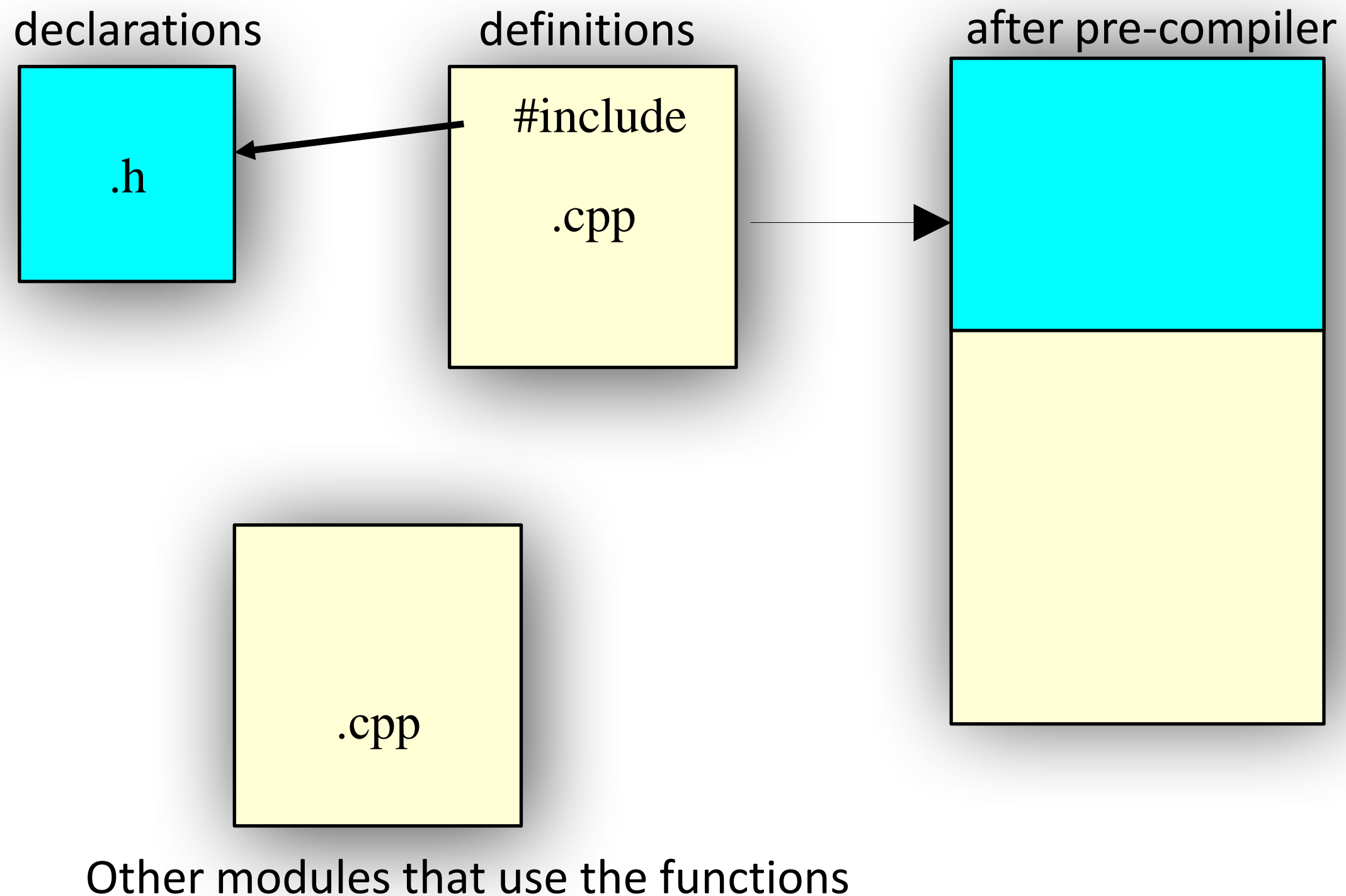
definitions



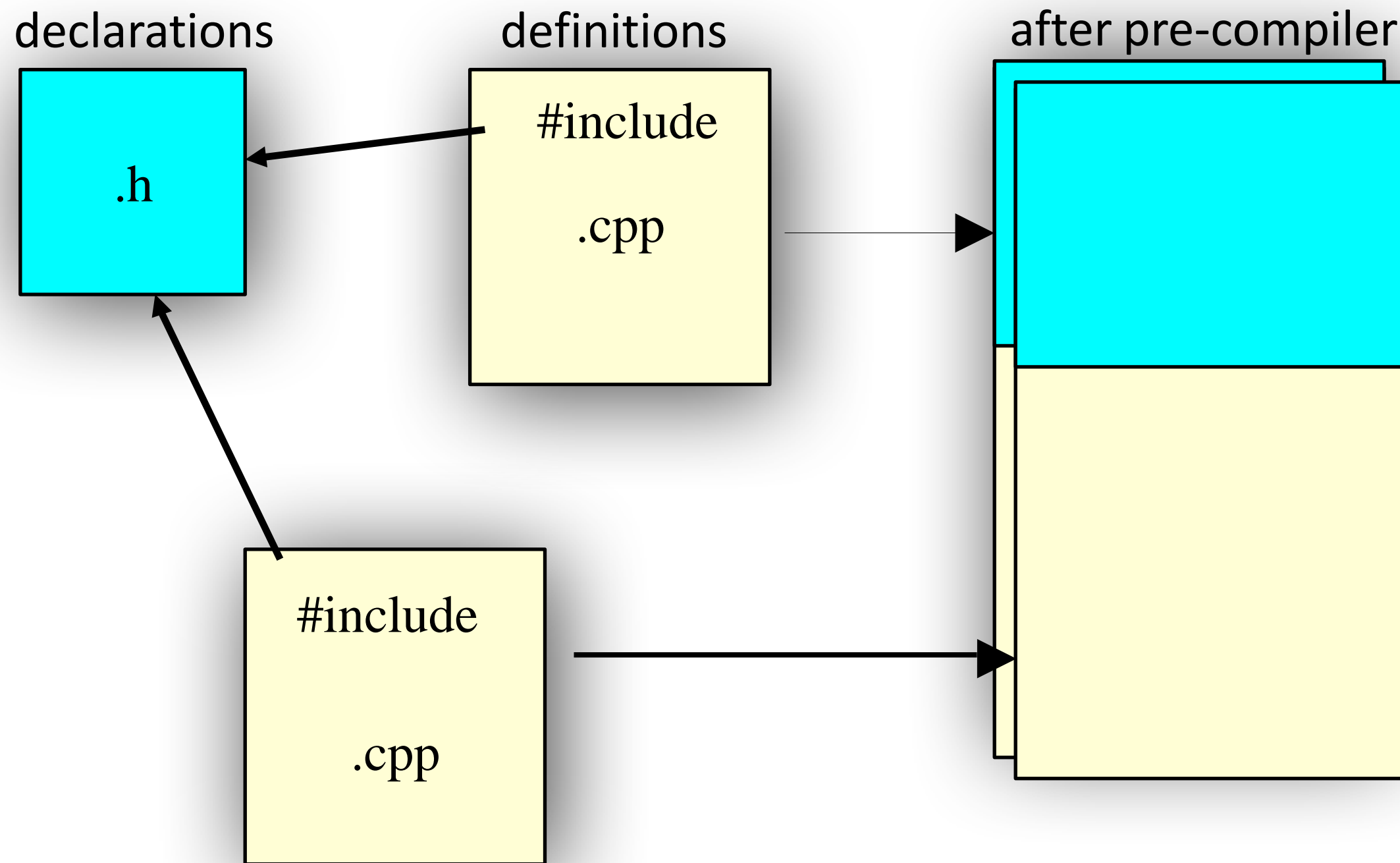
Structure of C++ program



Structure of C++ program



Structure of C++ program



Declarations vs. Definitions

- A `.cpp` file is a compile unit
- Only declarations are allowed to be in `.h`
 - extern variables
 - function prototypes
 - class/struct declaration

#include

- **#include** is to insert the included file into the .cpp file at where the **#include** statement is.
 - `#include "xx.h"`: usually search in the current directory, implementation defined
 - `#include <xx.h>`: search in the specified directories
 - `#include <xx>`: same as `#include <xx.h>`

Standard header file structure

```
#ifndef HEADER_FLAG  
#define HEADER_FLAG  
// all kinds of declarations here...  
#endif // HEADER_FLAG
```

Tips for header

1. One class declaration per header file
2. Associated with one source file in the same prefix of file name.
3. The contents of a header file is surrounded with the safe guard, e.g., `#ifndef #define... #endif`
4. `#pragma once` equivalent to `#ifndef...#endif`

The Makefile utility

Motivation

- Small programs —→ single file
- “Not so small” programs :
 - Many lines of code
 - Multiple components
 - More than one programmer

Motivation - continued

- Problems:

- Long files are harder to manage
(for both programmers and machines)
- Every change requires long compilation
- Many programmers can not modify
the same file simultaneously
- Division to components is desired

Motivation - continued

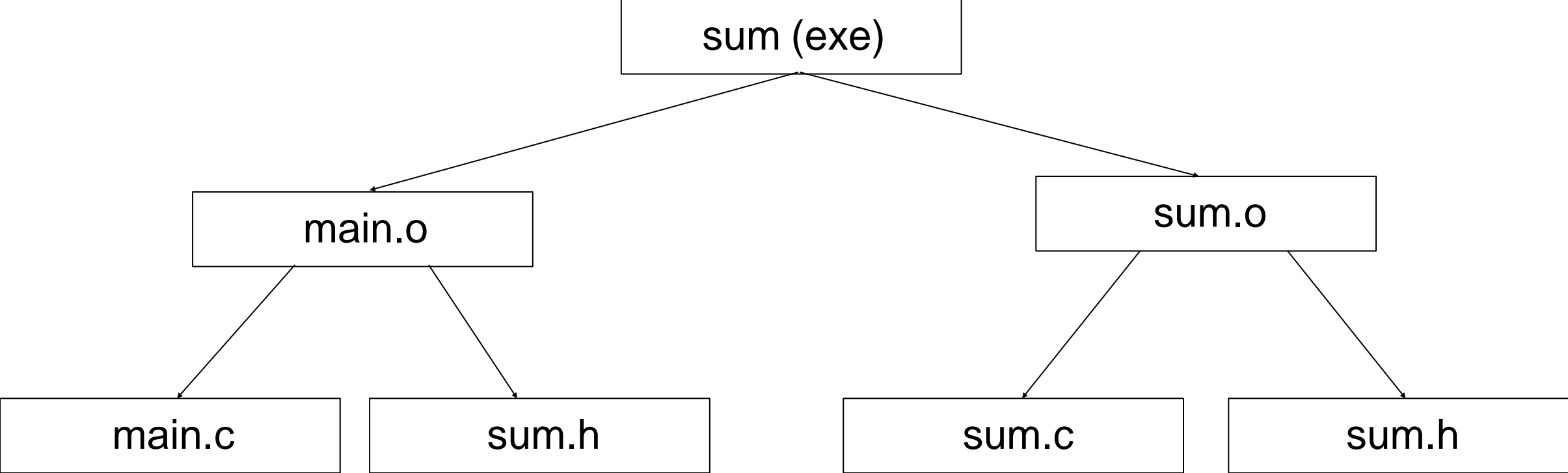
- Solution : divide project to multiple files
- Targets:
 - Good **division** to components
 - **Minimum compilation** when something is changed
 - **Easy maintenance** of project structure, dependencies and creation

Project maintenance

- Done in Unix by the Makefile mechanism
- A **makefile** is a file (script) containing :
 - Project **structure** (files, **dependencies**)
 - **Instructions** for files creation
- The **make** command reads a makefile, understands the project structure and makes up the executable
- Note that the Makefile mechanism is **not limited to C programs**

Project structure

- Project structure and dependencies can be represented as a DAG (= Directed Acyclic Graph)
- Example :
 - Program contains 3 files
 - main.c, sum.c, sum.h
 - sum.h included in both .c files
 - Executable should be the file sum



makefile

sum: main.o sum.o

gcc -o sum main.o sum.o

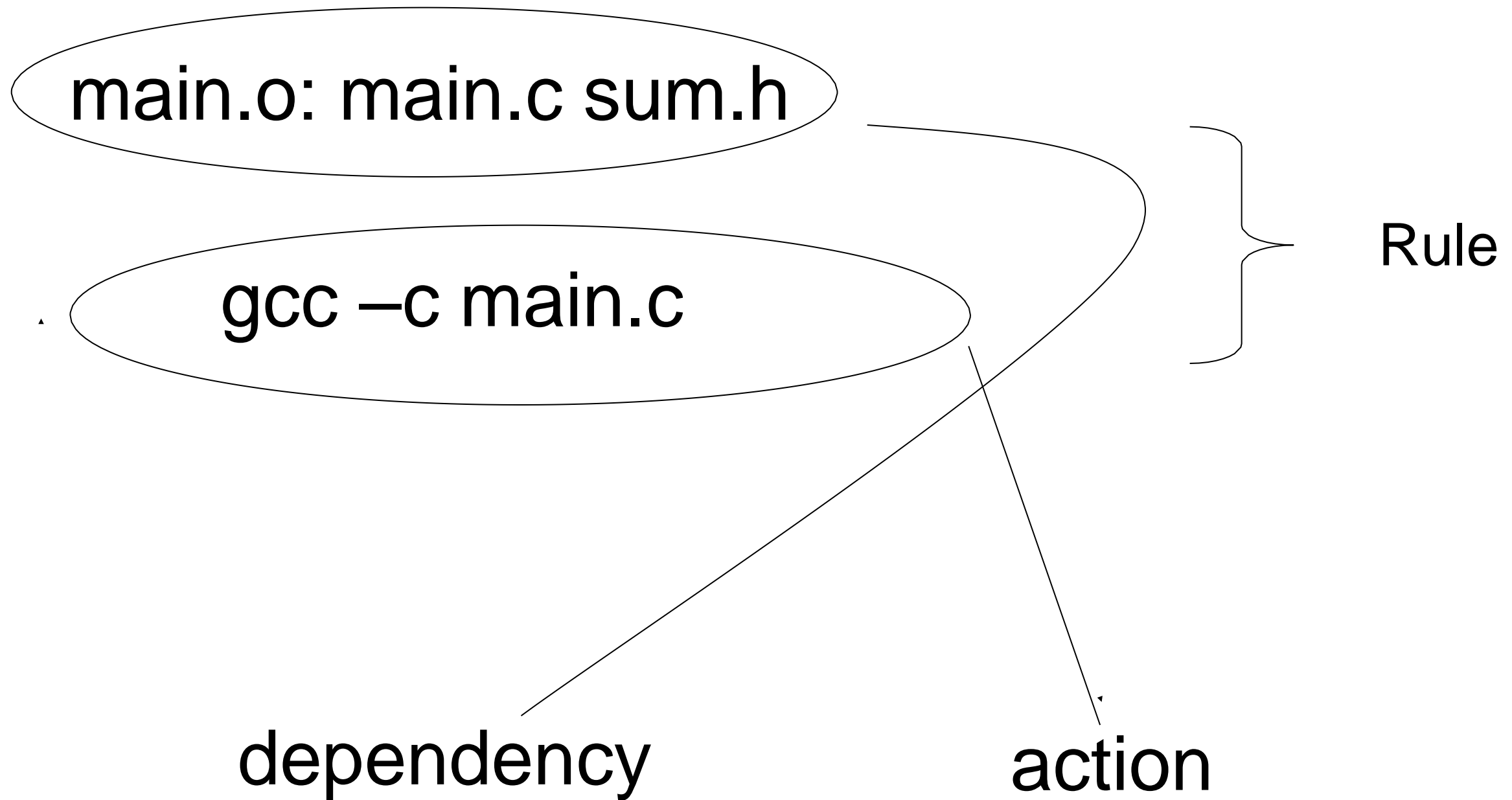
main.o: main.c sum.h

gcc -c main.c

sum.o: sum.c sum.h

gcc -c sum.c

Rule syntax



Equivalent makefiles

- .o depends (by default) on corresponding .c file. Therefore, equivalent makefile is:

```
sum: main.o sum.o
```

```
gcc -o sum main.o sum.o
```

```
main.o: sum.h
```

```
gcc -c main.c
```

```
sum.o: sum.h
```

```
gcc -c sum.c
```

Equivalent makefiles - continued

- We can compress identical dependencies and use built-in macros to get another (shorter) equivalent makefile :

```
sum: main.o sum.o
```

```
gcc -o $@ main.o sum.o
```

```
main.o sum.o: sum.h
```

```
gcc -c $*.c
```


make operation

- Project dependencies tree is constructed
- Target of first rule should be created
- We go down the tree to see if there is a target that should be recreated. This is the case when the target file is older than one of its dependencies
- In this case we recreate the target file according to the action specified, on our way up the tree. Consequently, more files may need to be recreated
- If something is changed, linking is usually necessary

make operation - continued

- make operation ensures **minimum compilation**, when the project structure is written properly

- **Do not write** something like:

prog: main.c sum1.c sum2.c

gcc -o prog main.c sum1.c sum2.c

which requires **compilation of all project** when something is changed

Make operation - example

<u>File</u>	<u>Last Modified</u>
sum	10:03
main.o	09:56
sum.o	09:35
main.c	10:45
sum.c	09:14
sum.h	08:39

Make operation - example

- Operations performed:

```
gcc -c main.c
```

```
gcc -o sum main.o sum.o
```

- `main.o` should be `recompiled` (`main.c` is newer).
- Consequently, `main.o` is newer than `sum` and therefore `sum` should be `recreated` (by re-linking).

Reference

- Good tutorial for makefiles

<https://www.gnu.org/software/make/manual/make.html>

The CMake utility

CMake

- A cross-platform, open-source build system
- Cmake is used to control software compilation process using simple platform and compiler independent configuration files.
- Cmake generates native makefiles and workspaces that can be used in the compiler environment of your choice.

CMake

- A cross-platform, open-source build system
- A family of tools designed to build, test and package software.
- CMake is used to control software compilation process using simple platform and compiler independent configuration files.

CMake

- You write a single configuration that CMake understands.
- CMake takes care that it works on all compilers and platforms.
- Don't make any assumption about the platform or compiler!

A Simple Example

- 1. Set the minimum required version of Cmake
`cmake_minimum_required (VERSION 2.8)`
- 2. Set the project information
`project (point_design)`
- 3. Add an executable target
`add_executable(point_design main.cpp
Point.cpp)`