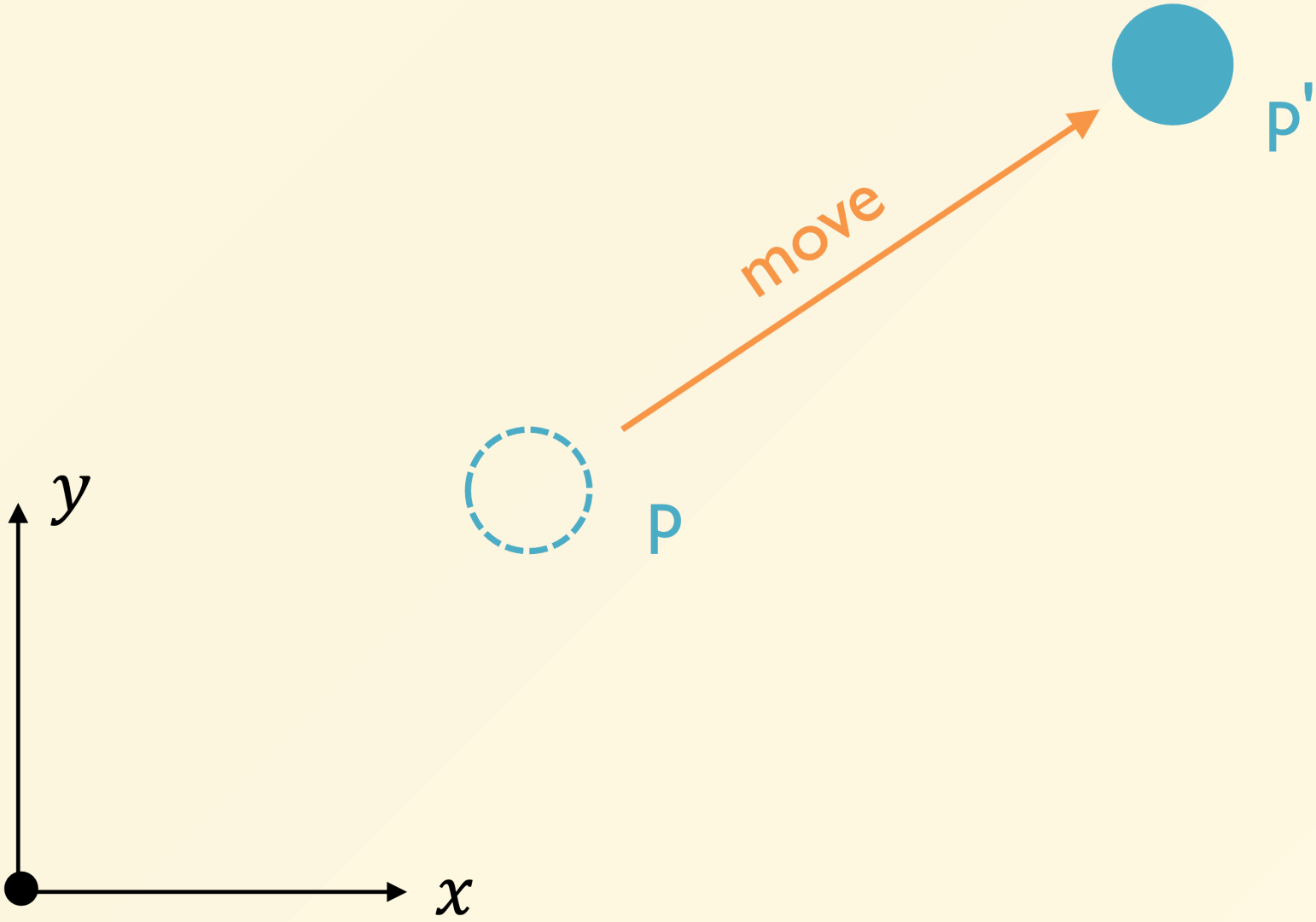
Class

Object-Oriented Programming with C++


P'
move
P
y
x

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;
}
```

C vs. C++

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);
```

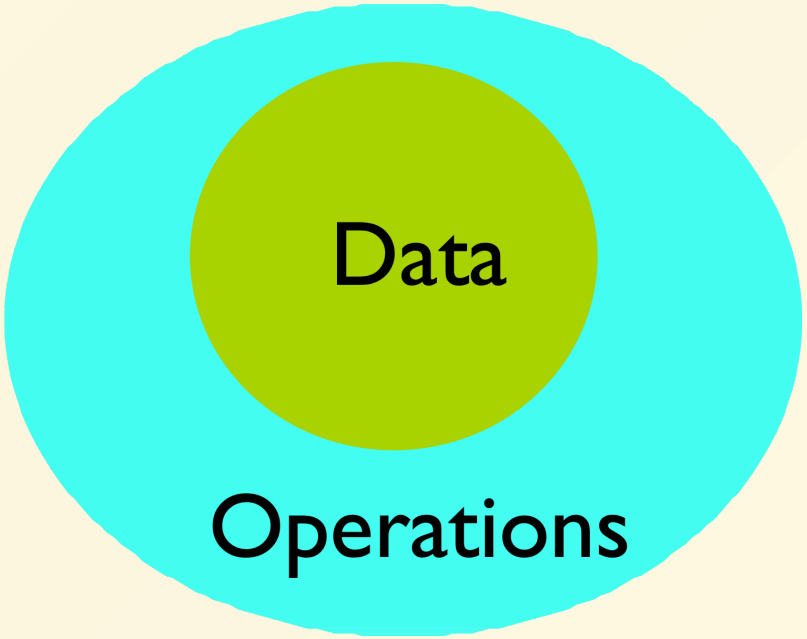
C++

```cpp
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



Ticket Machine

Something is there

PRICE

balance

total

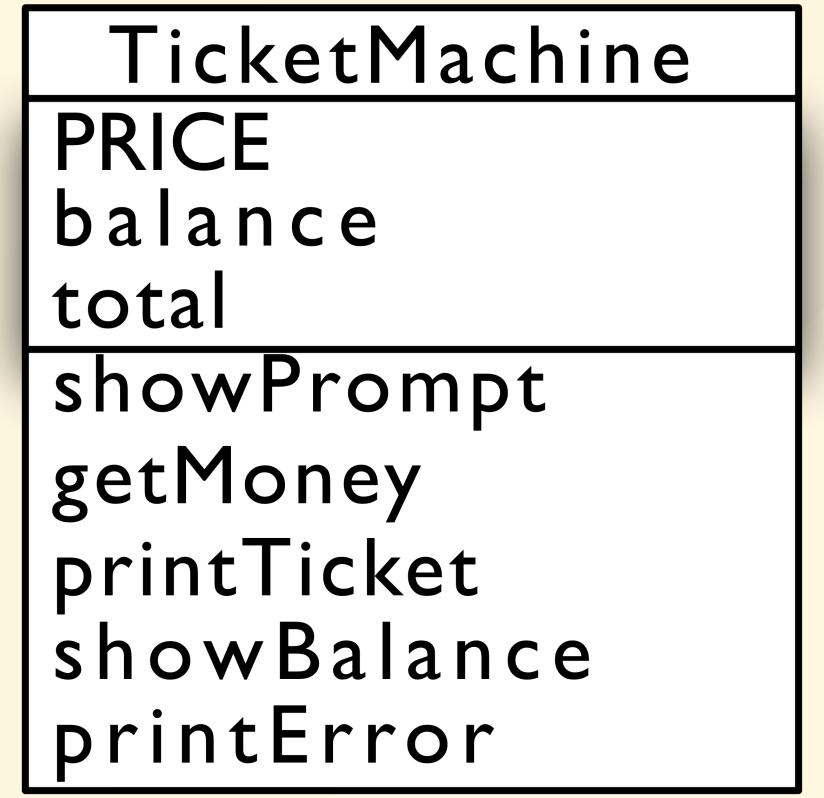
show prompt

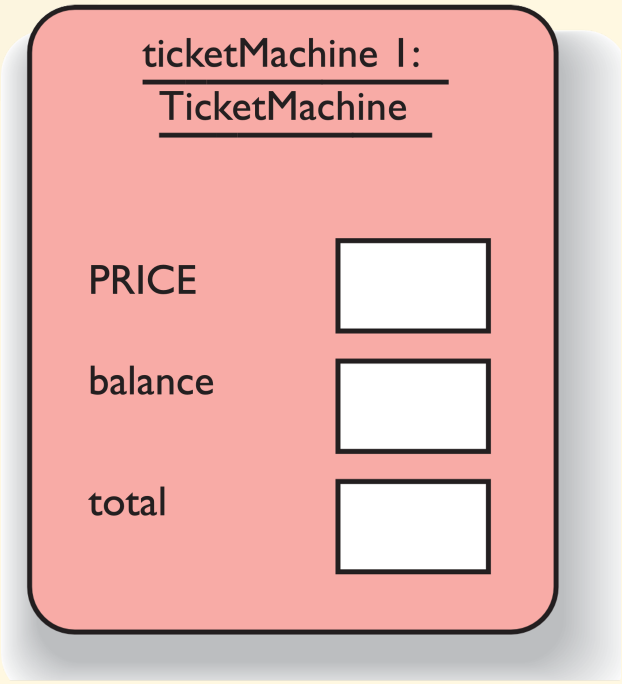
print balance

insert money

print ticket

Something is there



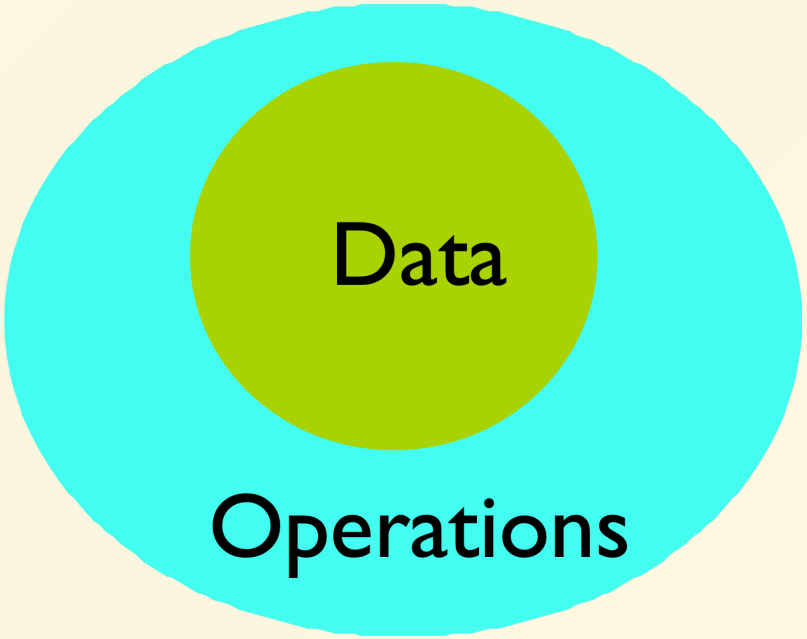


Turn it into code

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

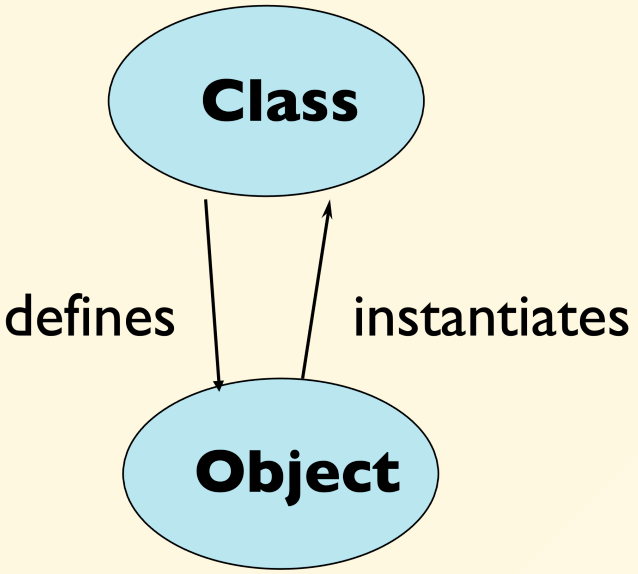
Objects = Attributes + Services

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



Object vs. Class

- Objects (cat)
  - Represent things, events
  - Respond to messages at run-time
- Classes (cat class)
  - Define properties of instances
  - Act like native-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 header and source files are used to define one class.
- Class declaration and member function prototypes are in the `.h` header file.
- All the function bodies (implementations) are in the `.cpp` source file.
- PImpl technique: debatable, hides private members and removes compilation dependency.

:: 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
}
```

Compilation unit

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

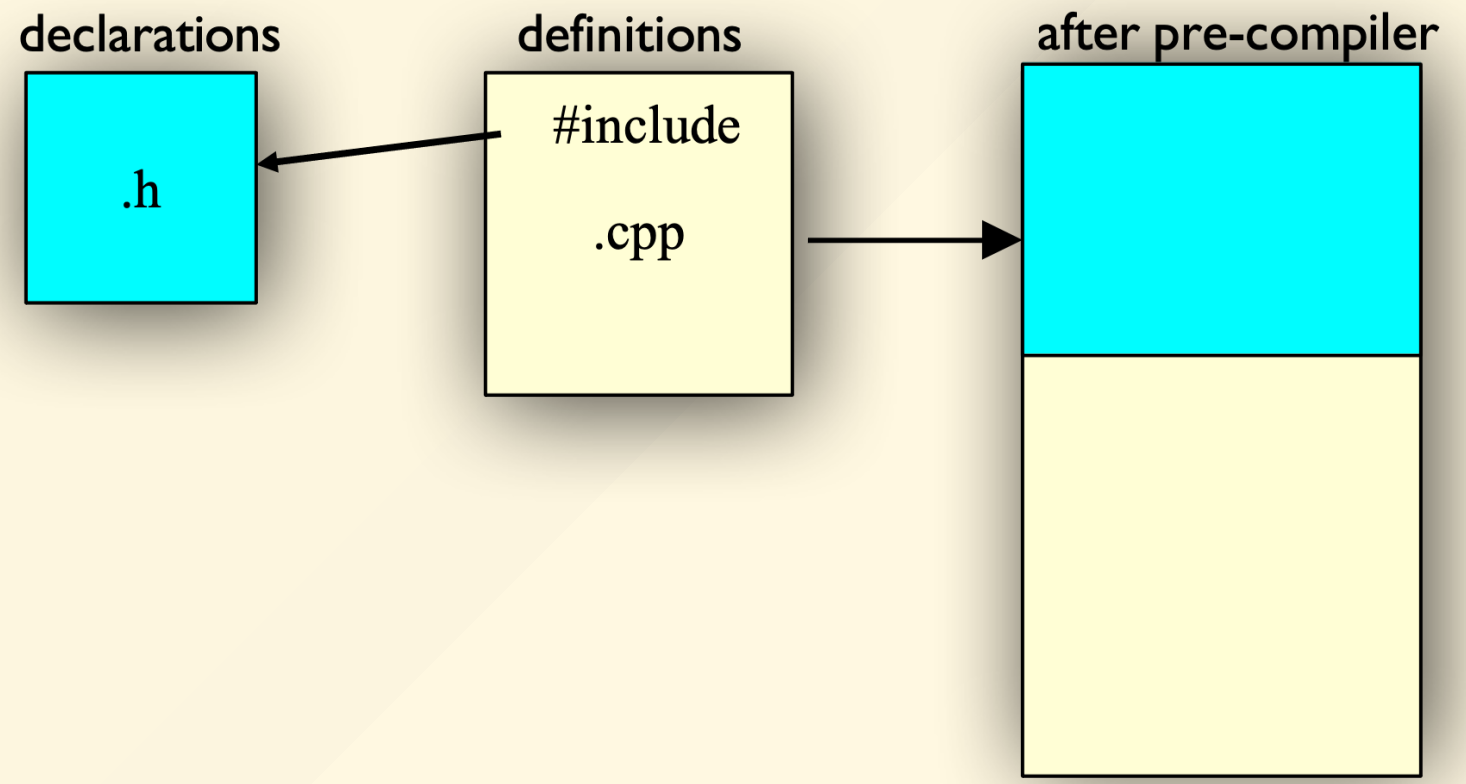
The header files

- 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.

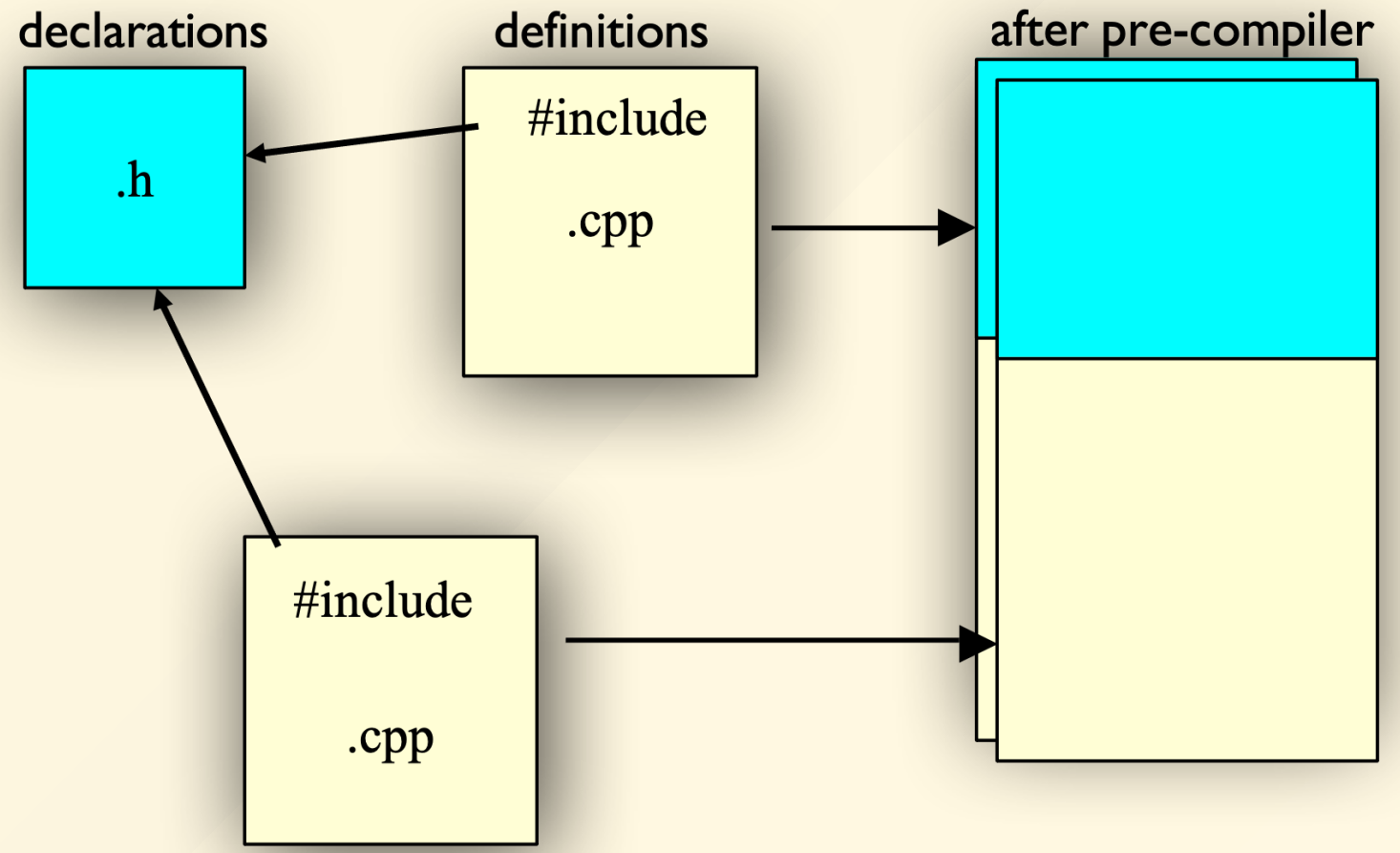
Header = interface

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

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 content of header file into the .cpp file, right 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, depending on your development environment.

Standard header file structure

```
#ifndef HEADER_FLAG
#define HEADER_FLAG

// all kinds of declarations here...

#endif // HEADER_FLAG
```

Tips for header

- One class declaration per header file
- Same name with the .cpp file.
- The declaration contents are surrounded with the safe guard.

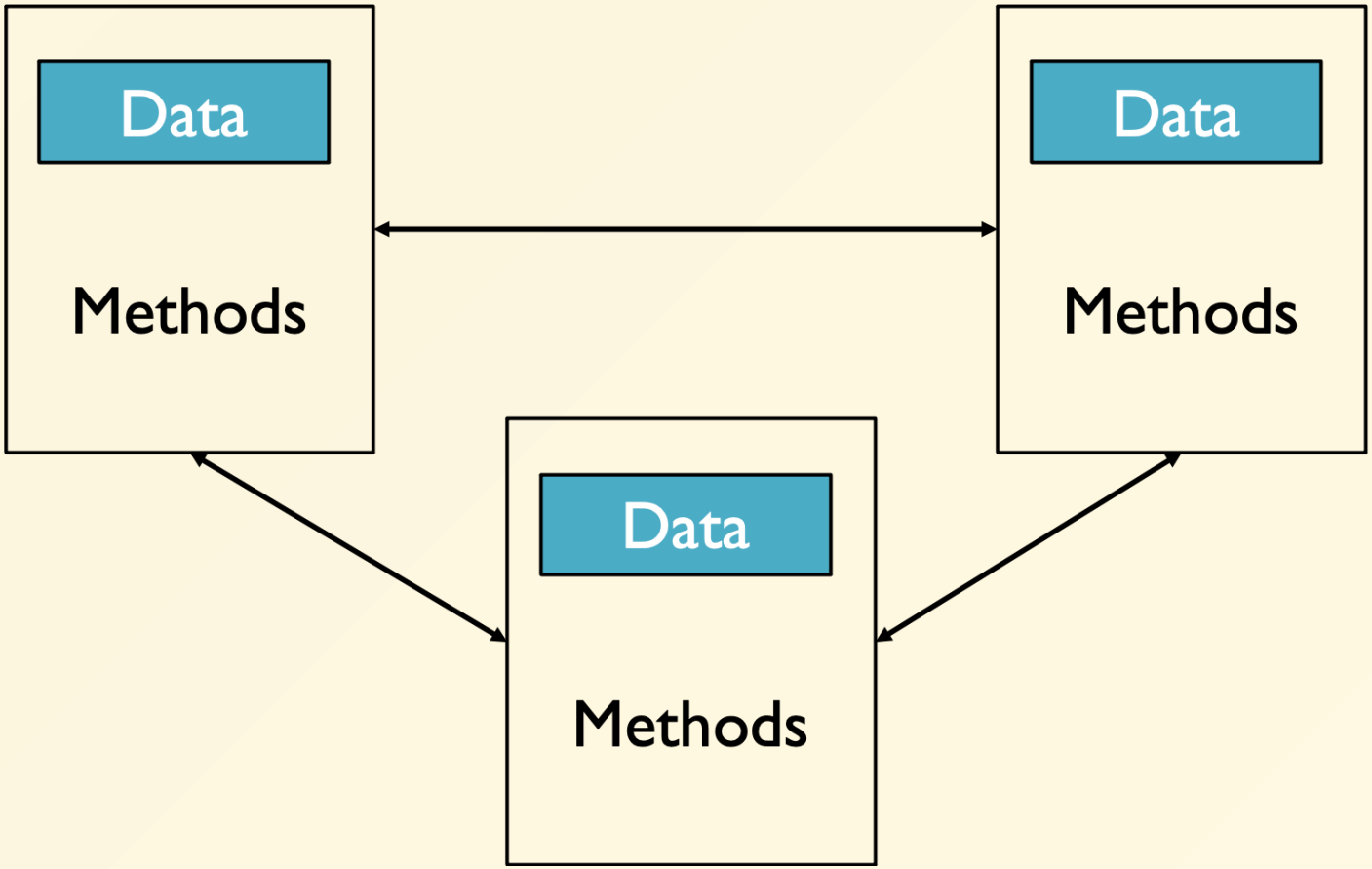
Build automation tools

- The CMake utility
  - cross-platform, free, and open-source
  - build automation, testing, packaging and installation
  - it is not a build system itself; it generates another system's build files.

Object Interaction

Object-oriented programming

- Objects send and receive messages!



Message communication

- Messages are
  - *Composed* by the sender
  - *Interpreted* by the receiver
  - *Implemented* by methods
- Messages
  - May return results
  - May cause receiver to change state, i.e., *side effects*

Encapsulation

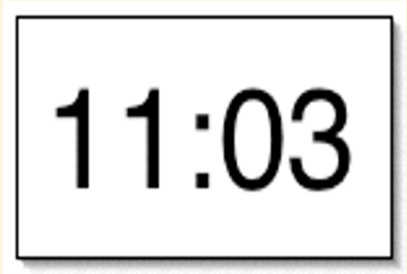
- Bundle data and methods together
- Hide the details of dealing with the data
- Restrict access only to the publicized methods

Abstraction

- Abstraction is the ability to ignore details of parts to focus on high-level problems.
- Modularization is the process of dividing a whole into parts that can be built separately and interact in well-defined ways.

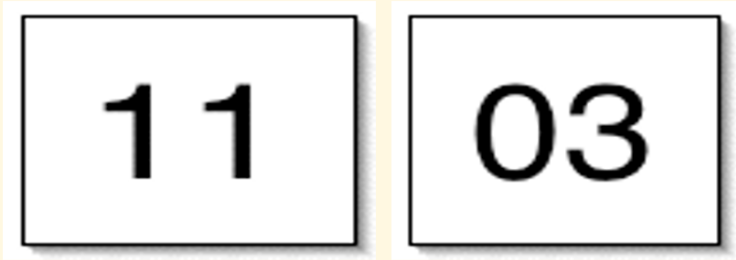
Clock display

Modularizing the clock display

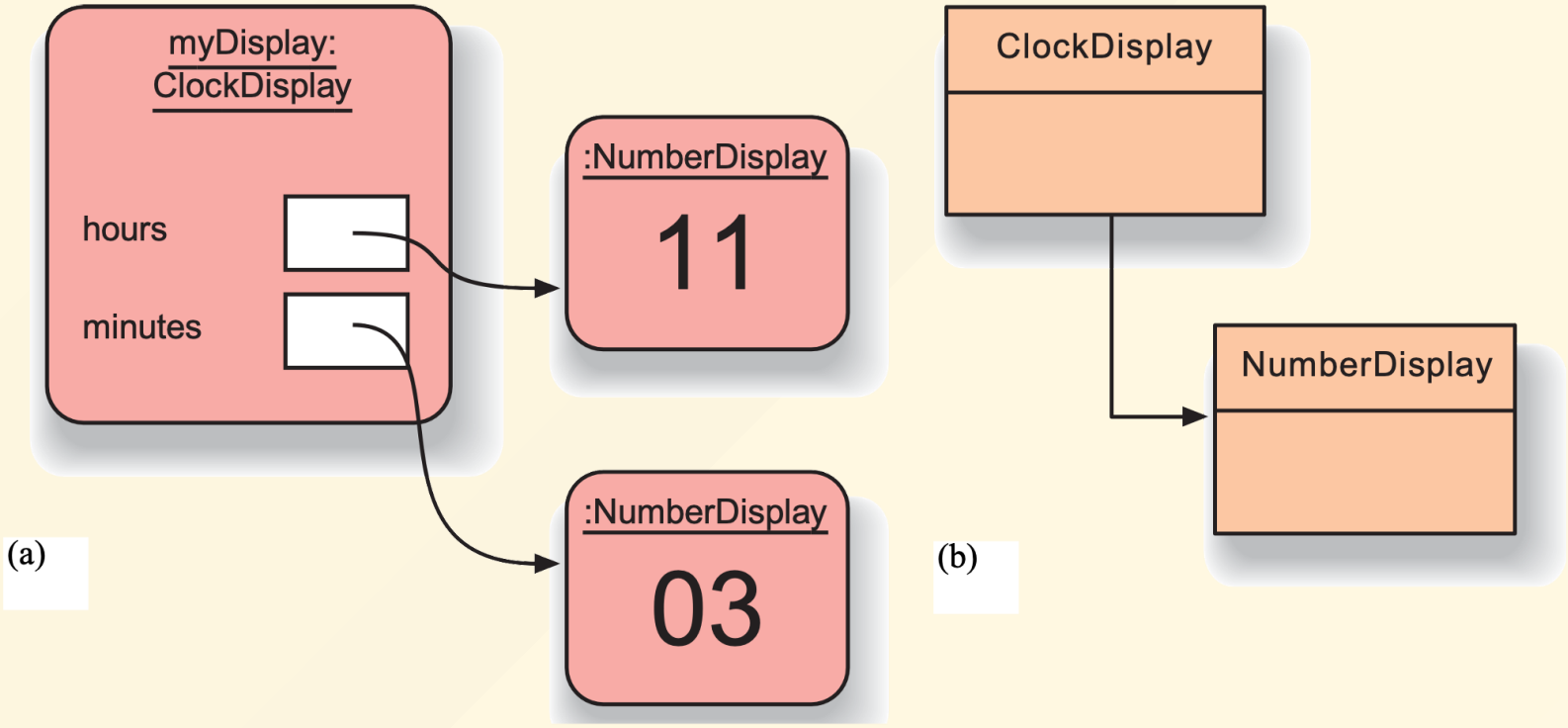


One *4-digits* display?

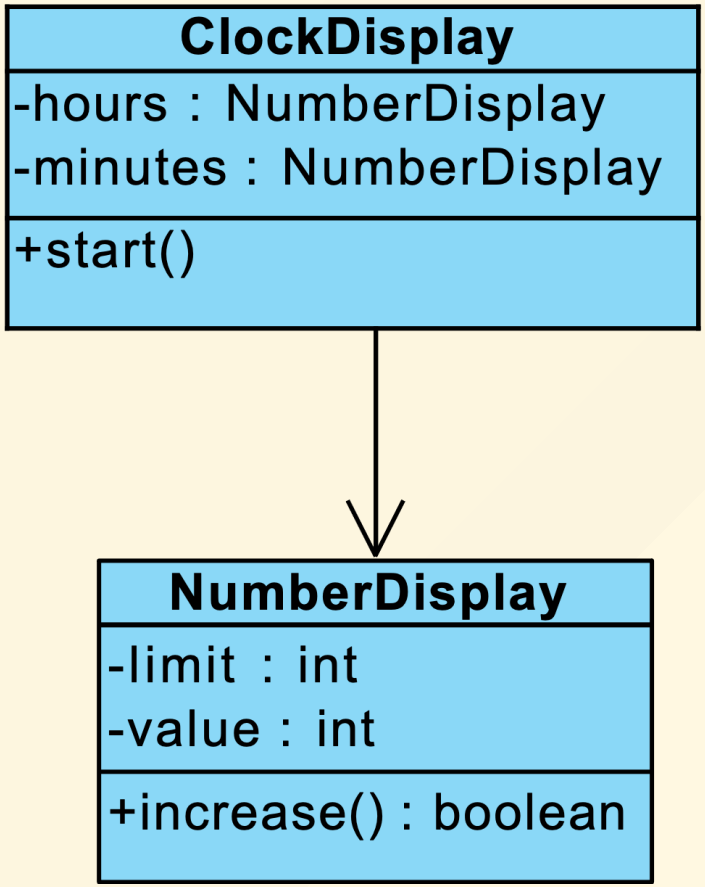
Or two *2-digits* displays?



Objects & Classes



Class diagram



Ctor and Dtor

Point::init()

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

Point a;
a.init(1,2);
a.move(2,2);
a.print();
```


功成到码
年年有余屯屯
蒸蒸日上烫烫
APPY NEW YE

Guaranteed initialization

- With a *constructor* (ctor)
  - the compiler ensures its call when an object is created.
  - use the name as the class.

How does a ctor work?

```
class X {
  int i;
public:
  X();
};
```

constructor

```
void f() {
  X a;
  // ...
}
```

a.X();

Ctors with arguments

- The constructor can have arguments
  - specify how an object is created
  - give it initialization values
  - ...

```
Tree(int i) {…}
Tree t(12);

// see Constructor1.cpp
```

Guaranteed cleanup

- With the destructor (dtor)
  - the compiler ensures its call when an object is about to end its life-cycle.
  - the name of the class with a leading tilde (~)
  - no arguments at all

```
class Y {
public:
  ~Y();
};
```

Aggregate initialization

```
int a[5] = {1,2,3,4,5};
int b[6] = {5};
int c[] = {1,2,3,4}; // sizeof(c) / sizeof(*c)

struct X {
  int i; float f; char c;
};
X x1 = {1, 2.2, 'c'};
X x2[3] = { {1, 1.1, 'a'}, {2, 2.2, 'b'} }

struct Y {
  float f; int i; Y(int a);
};
Y y1[] = { Y(1), Y(2), Y(3) };
```

The default constructor

- A *default constructor* is one that can be called with no arguments.

```
struct Y {
    float f;
    int i;
    Y(int a);
};
```

```
Y y1[] = { Y(1), Y(2), Y(3) };
```

```
Y y2[2] = { Y(1) };
```

```
Y y3[7];
```

```
Y y4;
```

auto default constructor

- Be careful:
  - If (and only if) there are no constructors for a class, the compiler will automatically create one for you.

Initializer list

```
class Point {
private:
  const float x, y;
public:
  Point(float xa, float ya)
    : y(ya), x(xa) {}
};
```

- Do not perform assignment within ctor's body
- Order of initialization is order of declaration
  - Not the order in the initializer list!
  - Destroyed in the reverse order.

Initialization vs. assignment

- `Student::Student(string s) : name(s) {}`

  initialization

  before constructor body

- `Student::Student(string s) { name = s; }`

  assignment

  inside constructor body

  string must have a default constructor

Local variable vs. Field

```
int TicketMachine::refundBalance() {
  int amountToRefund;
  amountToRefund = balance;
  balance = 0;
  return amountToRefund;
}
```

- Lifetime:
  - `amountToRefund` is with the function call
  - `balance` is with the object, i.e., object state

Local variable vs. Field

```
int TicketMachine::refundBalance() {
  int amountToRefund;
  amountToRefund = balance;
  balance = 0;
  return amountToRefund;
}
```

- But how is the access to `balance` achieved?
- A local variable of the same name as a field will prevent the field from being accessed within a method.

Fields

- Fields, or data members, are defined outside constructors and methods.
- Fields are used to store data that persists throughout the life of an object. In other words, they maintain the current state of an object.
- Fields have class scope: their accessibility extends throughout the whole class.

`this`: the hidden parameter

- `this` is a hidden parameter for all member functions, with the type of the class.

```
void Point::print()
```

➔ (can be regarded as)

```
void Point::print(Point *this)
```

`this`: the hidden parameter

- To call the function, you must specify a variable.

```
Point a;
a.print();
```

➔ (can be regarded as)

```
Point::print(&a);
```

`this`: the pointer to the caller

- Inside member functions, you can use `this` as the pointer to the variable that calls the function.
- `this` is a natural parameter of all class member functions that you cannot define, but can use directly.

Constant objects

```
const Currency the_raise(42, 38);
```

- What member functions can access the internals?
- How can the object be protected from change?

Const member functions

- Declare safe member functions `const`

```
void Date::set_day(int d) {
  day = d; // ok, non-const so can modify
}

int Date::get_day() const {
  return day; // ok
}
```

Const member functions

- Cannot modify the objects

```
void Date::set_day(int d) {
  day = d; // ok, non-const so can modify
}

int Date::get_day() const {
  day++; // ERROR: modifies data member
  set_day(12); // ERROR: calls non-const member
  return day;
}
```

Const member function usage

- Repeat the const keyword in the definition as well as the declaration.

```
int get_day() const;
```

```
int get_day() const { return day };
```

- Function members that do not modify data should be declared const.
- const member functions are safe for const objects.

Const and non-const objects

```
// non-const object
Date when(1,1,2001); // not a const
int day = when.get_day(); // OK
when.set_day(13); // OK

// const object
const Date birthday(12,25,1994); // const
int day = birthday.get_day(); // OK
birthday.set_day(14); // ERROR
```

Constant members in class

```
class A {

  const int i;

};
```

- has to be initialized in the ctor's *initializer list*.

Compile-time constants in class

```
class HasArray {
  const int size;
  int array[size]; // ERROR!
  ...
};
```

- Make the const value static (one per class, not one per-object):

```
static const int size = 100;
```

Overhead for a function call

- The extra processing time required:
  - Push parameters
  - Push return address
  - Prepare return values
  - Pop all pushed

Overhead for a function call

```
int f(int i) {
  return i*2;
}
int main() {
  int a = 4;
  int b = f(a);
}
```

Inline

- An *inline* function is expanded in place, like a preprocessor macro in C, so the overhead of the function call is eliminated.
- Much safer than macro. It checks the types of the parameters, and has no dangerous *side effect*.

Inline

original

```
inline int f(int i) {
  return i * 2;
}
int main() {
  int a = 4;
  int b = f(a);
}
```

after expansion

```
int main() {
  int a = 4;
  int b = a + a;
}
```

inline vs. macro

macro

```
#define unsafe(i) \
 ((i)>=0?(i):-(i))

int f();

int main() {
  ans = unsafe(x++);
  ans = unsafe(f());
}
```

inline

```
inline int safe(int i)
{ return i>=0 ? i:-i; }

int f();

int main() {
  ans = safe(x++);
  ans = safe(f());
}
```

Tradeoff

- Expand the code size but deduces the function call overhead, so it gains speed at the expenses of space.

Inline inside class

- Any function you define inside a class declaration is automatically an inline.

```
class Cup {
  rgb color;
public:
  rgb getColor() { return color; }
  void setColor(rgb color) { this->color = color; }
};
```

Inline or not?

- inline
  - small functions, 2 or 3 lines
  - frequently called functions, e.g. inside loops
- not inline?
  - very large functions, say, more than 20 lines
  - recursive functions

Inline may not in-line

- The compiler does not have to honor your request to make a function inline.
- It might decide the function is too large or notice that it calls itself, or the feature might not be implemented for your particular compiler.
- Nowadays, the keyword `inline` for functions comes to mean "*multiple definitions are permitted*" rather than "*inlining is preferred*".

Inline functions in header file

- Put inline function's body in *header file*. Then `#include` it where the function is needed.
- Never be afraid of multi-definition of inline functions.