Introduction to Programming (C/C++)

05: Object-Oriented Programming

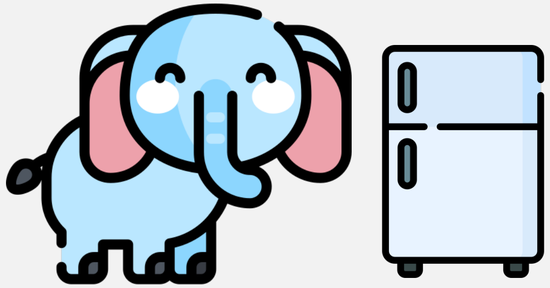
Huanchen Zhang

清華大學
Tsinghua University

交叉信息研究院
Institute for Interdisciplinary Information Sciences

Top-Down Thinking

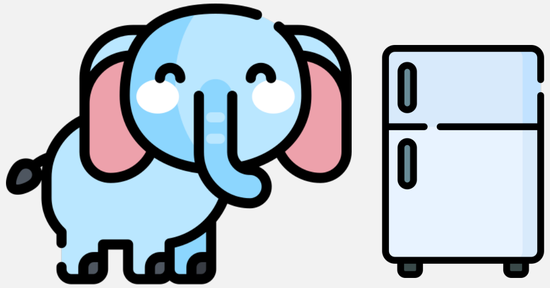
Task: pack an elephant into a fridge



1. Get an elephant
2. Create a large-enough fridge
3. Open the fridge door
4. Put the elephant in
5. Close the fridge door

Top-Down Thinking

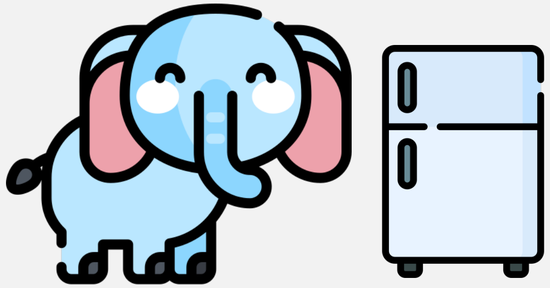
Task: pack an elephant into a fridge



1. Get an elephant
2. Create a large-enough fridge
3. Open the fridge door
4. Put the elephant in
   - **4.1** Sit the elephant
   - **4.2** Move the elephant
   - **4.3** Roll up the nose
5. Close the fridge door

Top-Down Thinking

Task: pack an elephant into a fridge

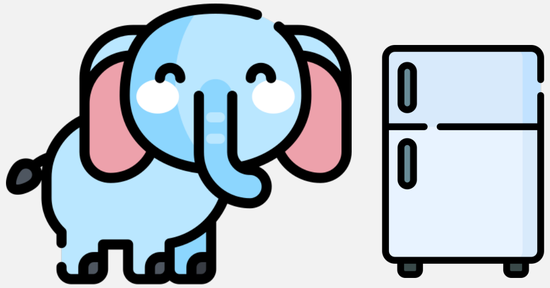


1. Get an elephant
2. Create a large-enough fridge
3. Open the fridge door
4. Put the elephant in
   - **4.1** Sit the elephant
   - **4.2** Move the elephant
   - **4.3** Roll up the nose
5. Close the fridge door

Subtasks → **Functions**

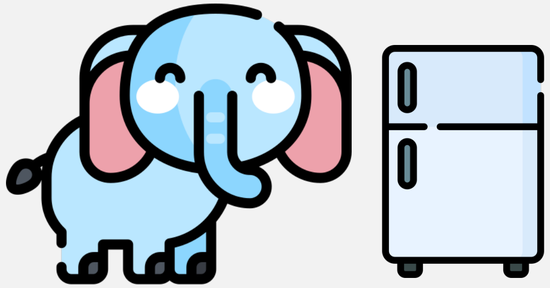
Procedural Programming

```
CreateElephant()
CreateFridge()
OpenFridge()
SitElephant()
MoveElephant()
RollupNose()
CloseFridge()
```



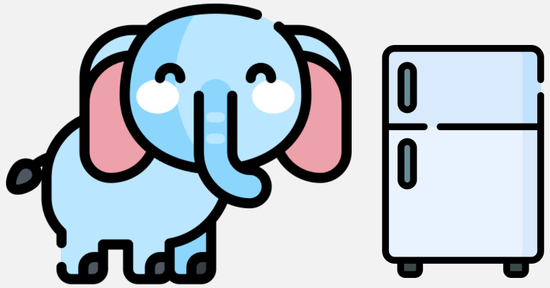
Procedural Programming

```
size, weight, CreateElephant(size, weight)
CreateFridge()
OpenFridge()
SitElephant()
MoveElephant()
RollupNose()
CloseFridge()
```



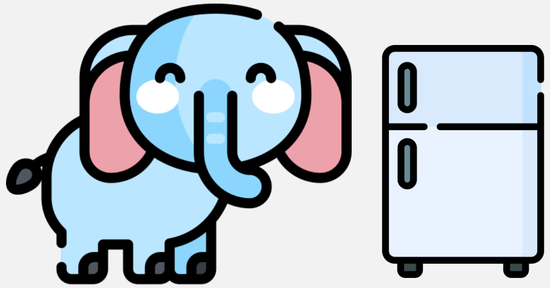
Procedural Programming

```
size, weight, CreateElephant(size, weight)
area, depth, CreateFridge(area, depth)
OpenFridge()
SitElephant()
MoveElephant()
RollupNose()
CloseFridge()
```



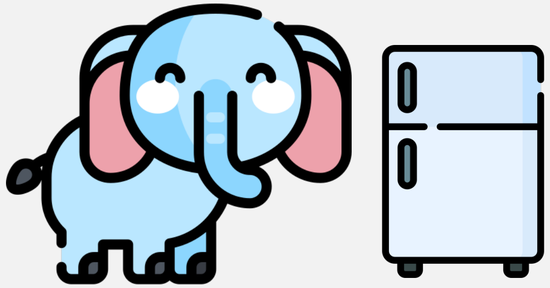
Procedural Programming

```
size, weight, CreateElephant(size, weight)
area, depth, CreateFridge(area, depth)
door_state, OpenFridge(door_state)
SitElephant()
MoveElephant()
RollupNose()
CloseFridge()
```



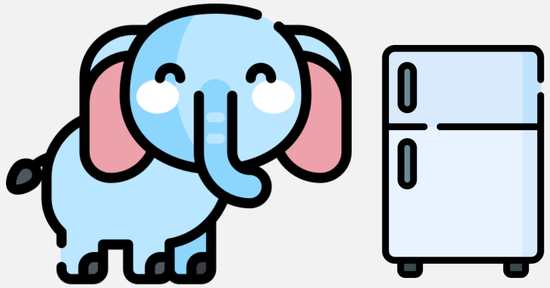
Procedural Programming

```
size, weight, CreateElephant(size, weight)
area, depth, CreateFridge(area, depth)
door_state, OpenFridge(door_state)
e_state, SitElephant(e_state)
MoveElephant()
RollupNose()
CloseFridge()
```



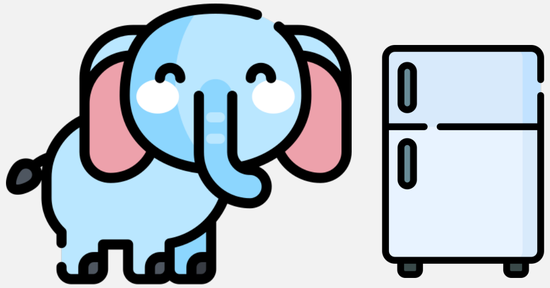
Procedural Programming

```
size, weight, CreateElephant(size, weight)
area, depth, CreateFridge(area, depth)
door_state, OpenFridge(door_state)
e_state, SitElephant(e_state)
e_pos, MoveElephant(e_pos)
RollupNose()
CloseFridge()
```



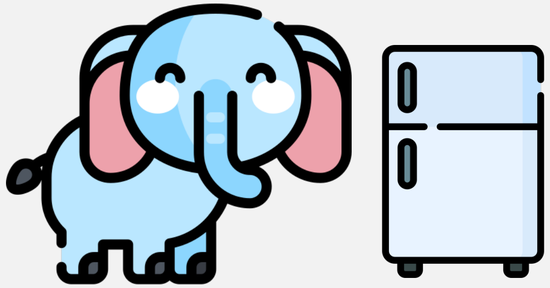
Procedural Programming

```
size, weight, CreateElephant(size, weight)
area, depth, CreateFridge(area, depth)
door_state, OpenFridge(door_state)
e_state, SitElephant(e_state)
e_pos, MoveElephant(e_pos)
nose_state, RollupNose(nose_state)
CloseFridge()
```



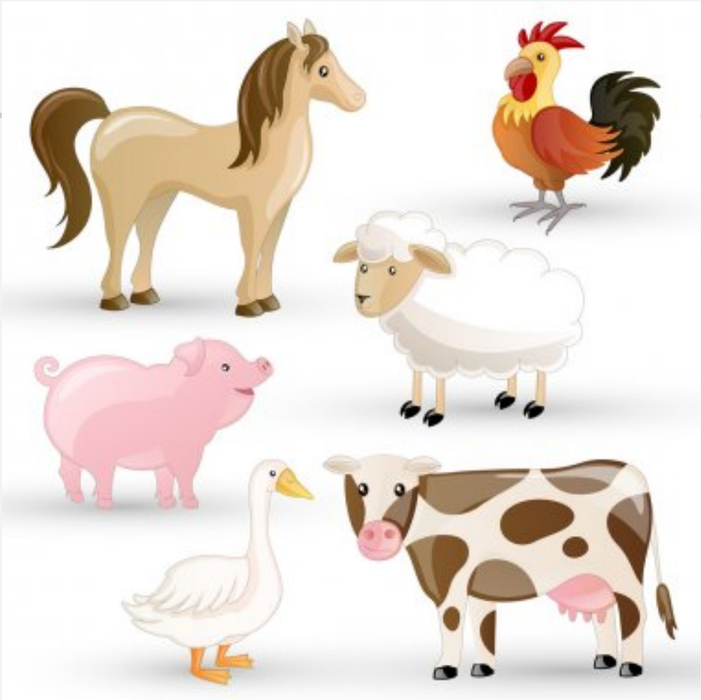
Procedural Programming

```
size, weight, CreateElephant(size, weight)
area, depth, CreateFridge(area, depth)
door_state, OpenFridge(door_state)
e_state, SitElephant(e_state)
e_pos, MoveElephant(e_pos)
nose_state, RollupNose(nose_state)
CloseFridge(door_state)
```

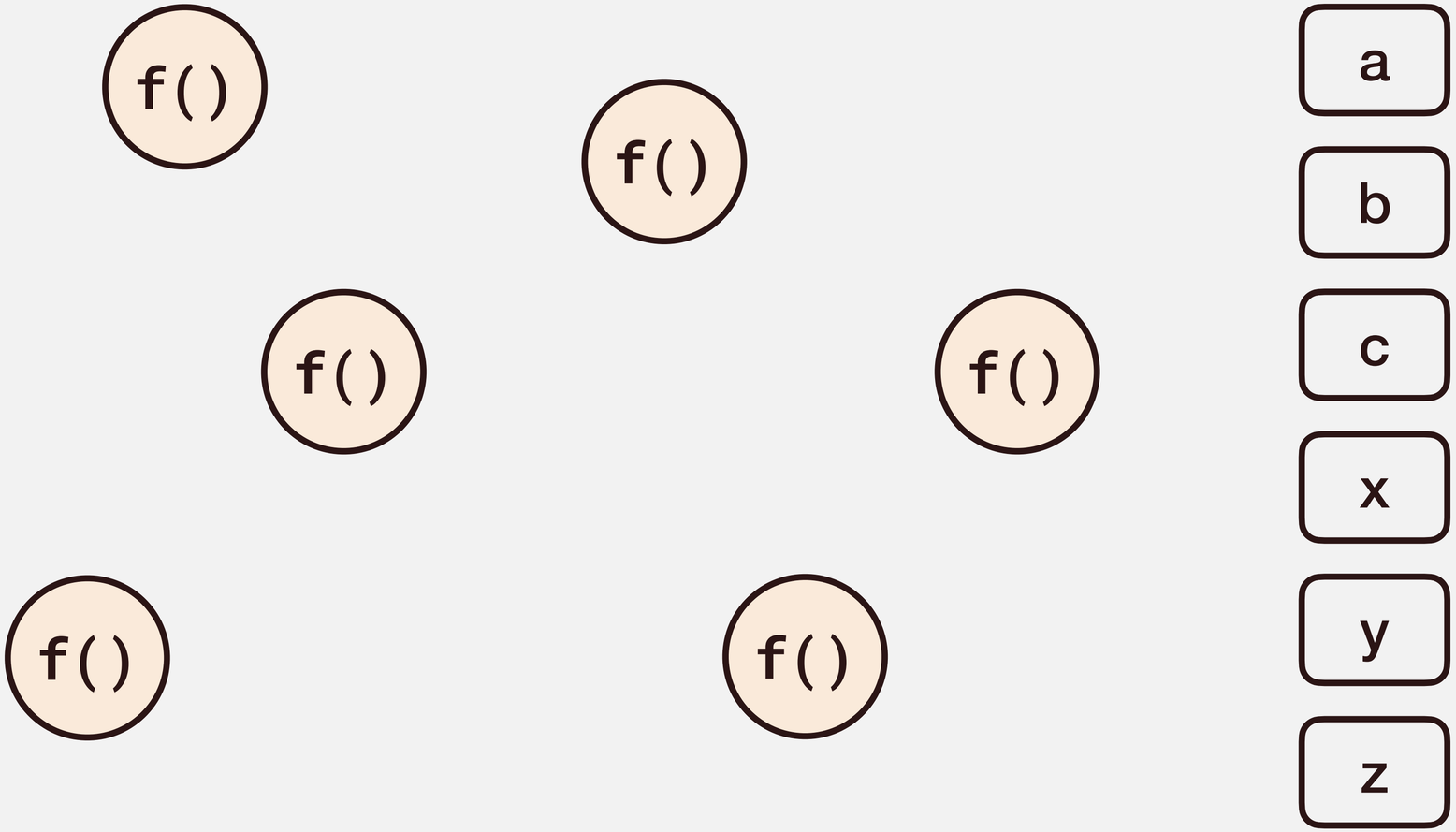


Procedural Programming

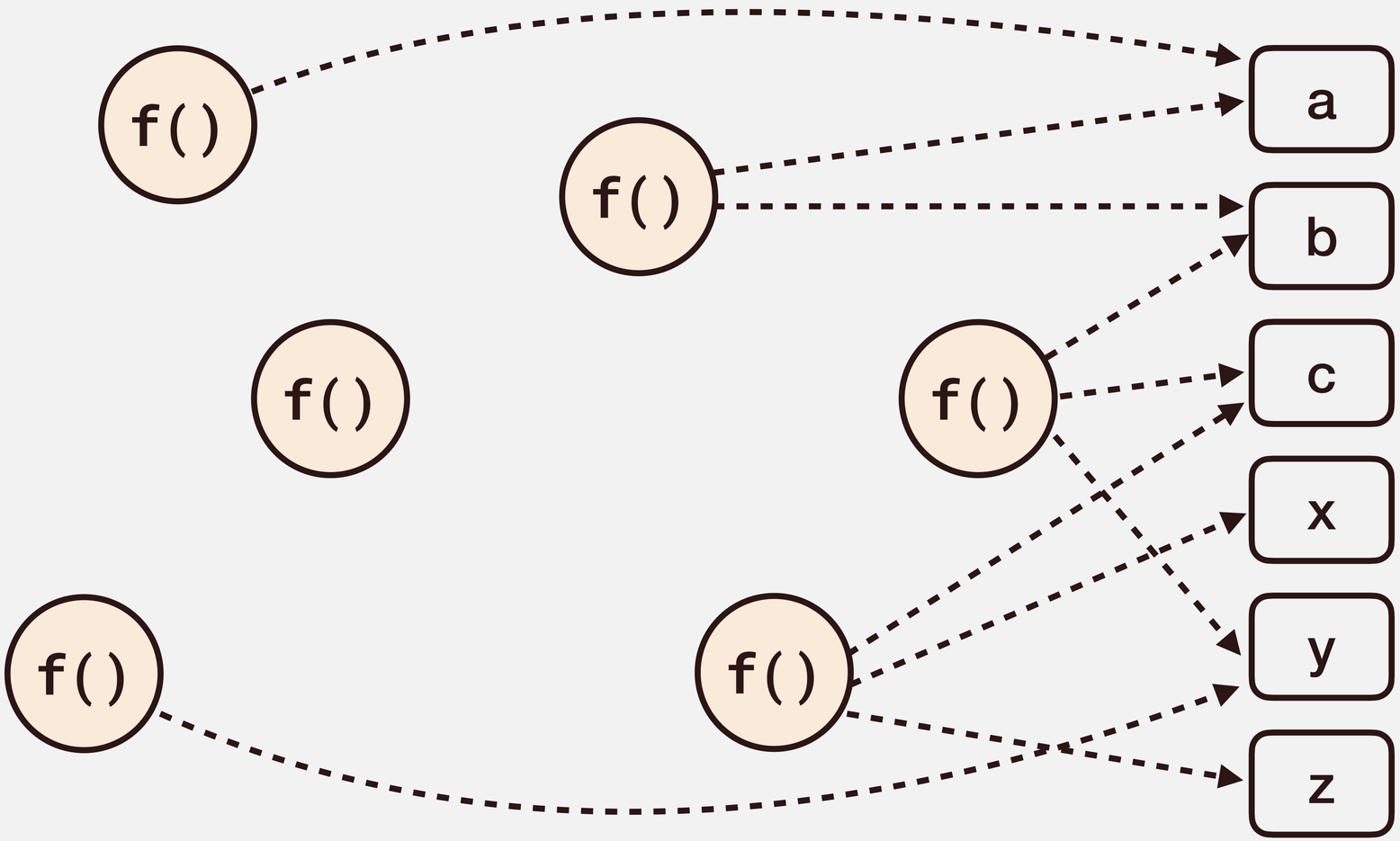
```
size, weight, CreateElephant(size, weight)
area, depth, CreateFridge(area, depth)
door_state, OpenFridge(door_state)
e_state, SitElephant(e_state)
e_pos, MoveElephant(e_pos)
nose_state, RollupNose(nose_state)
CloseFridge(door_state)
```



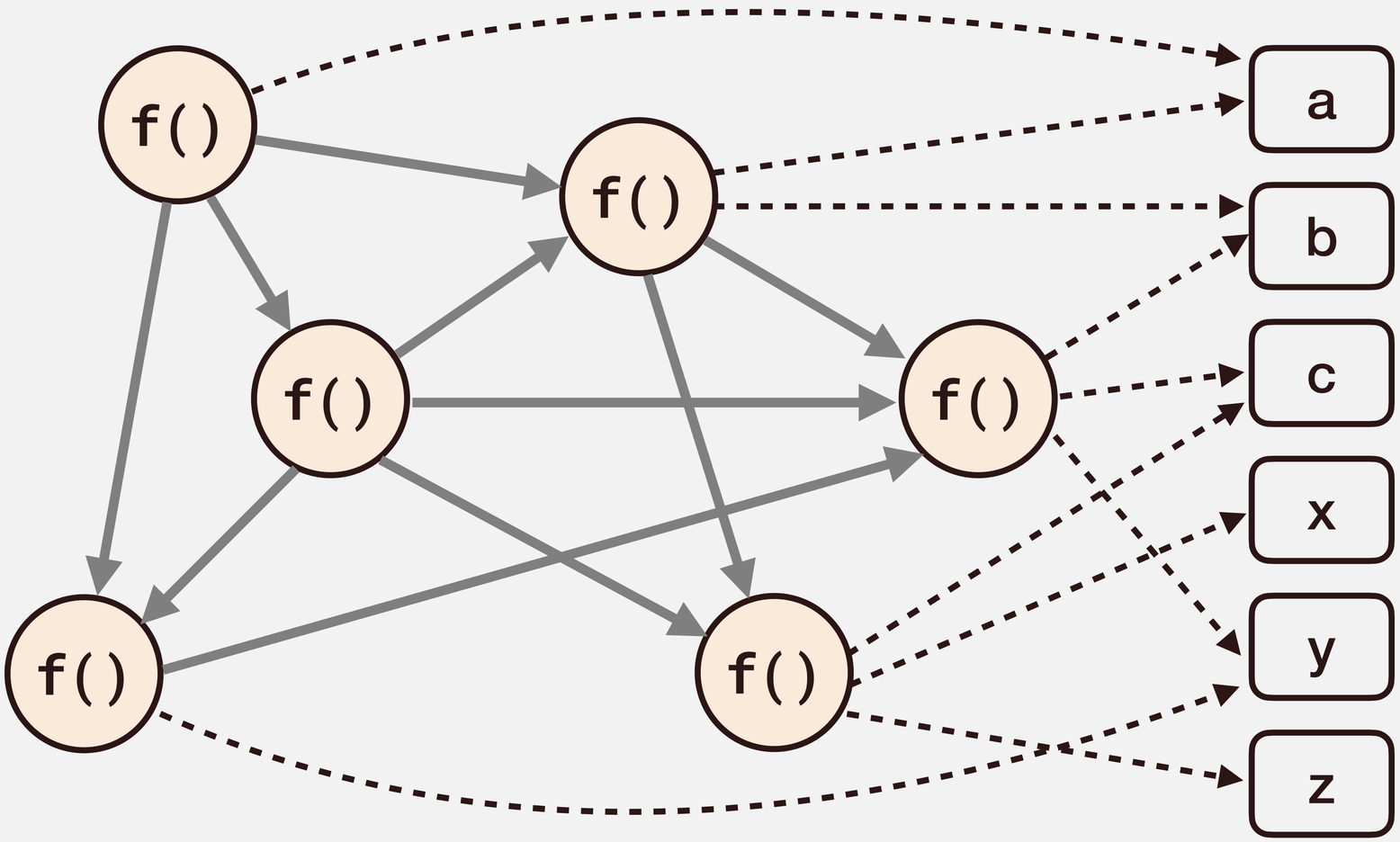
Procedural Programming



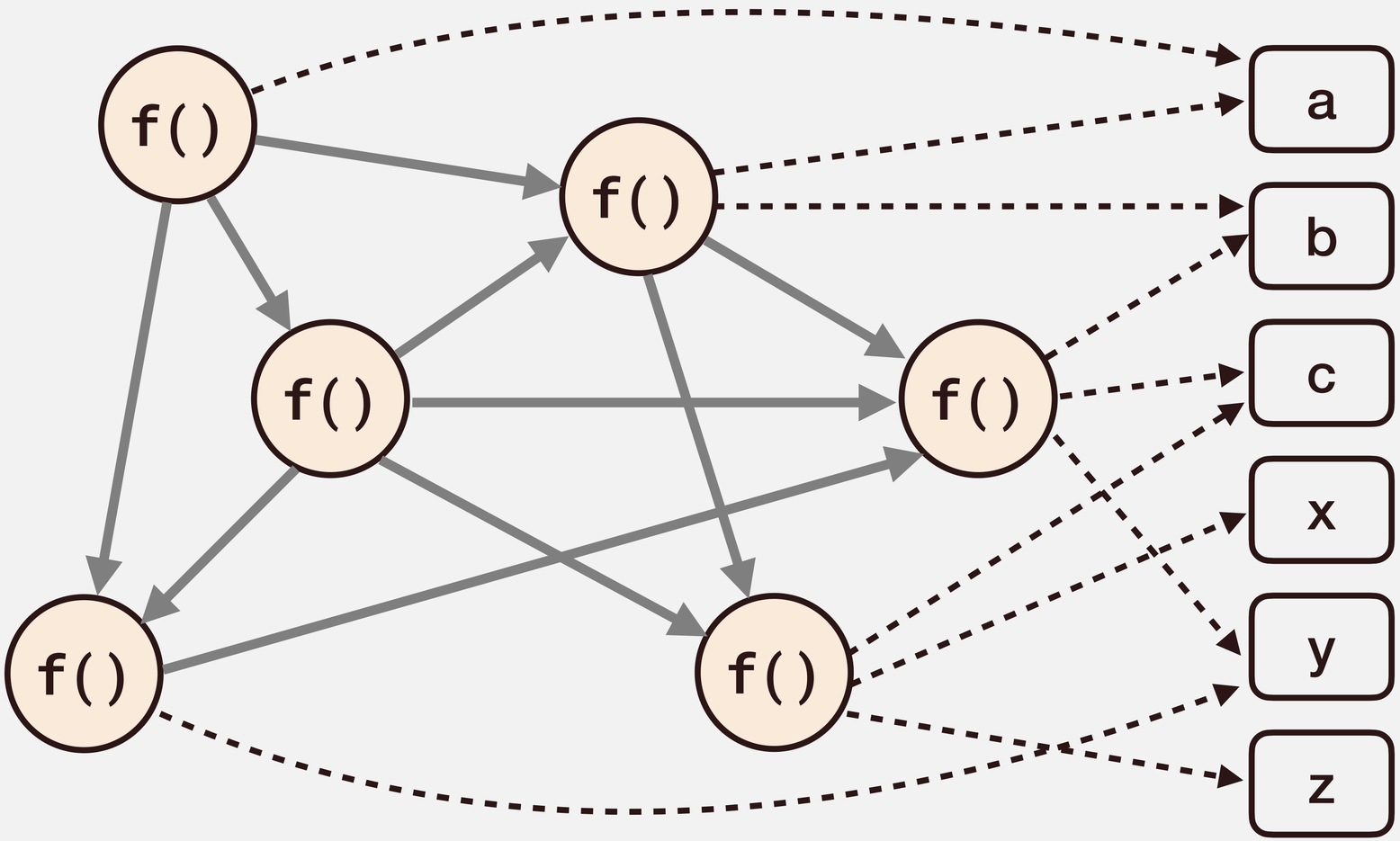
Procedural Programming



Procedural Programming

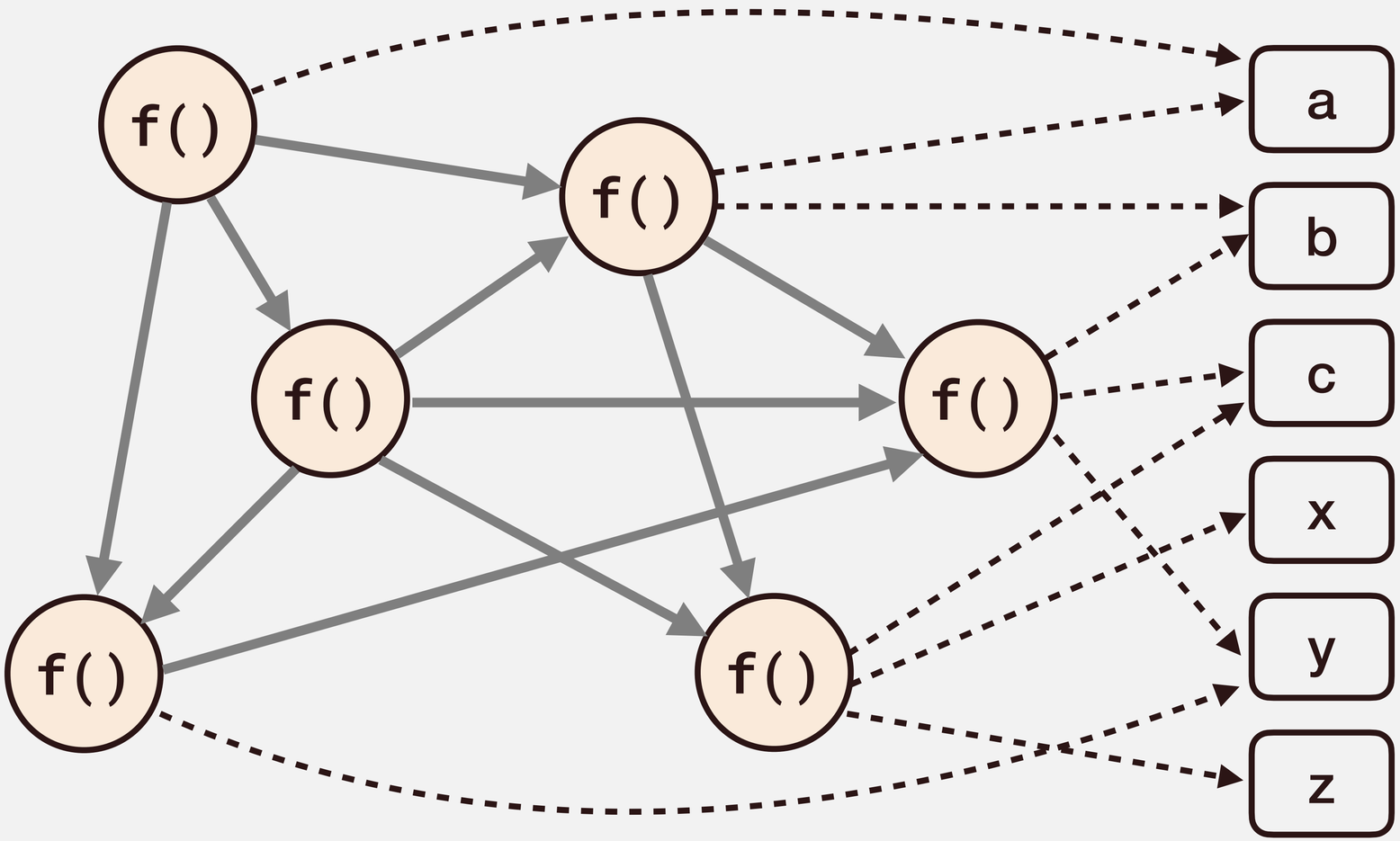


Procedural Programming



Spaghetti Code

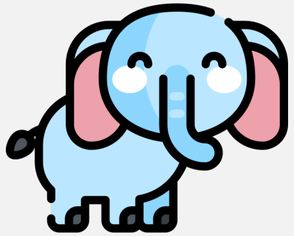
Procedural Programming

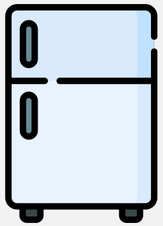


Spaghetti Code

Hard to manage Complexity

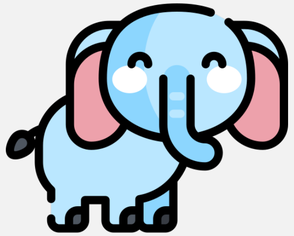
Object-Oriented Thinking

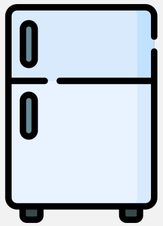




Properties

Object-Oriented Thinking

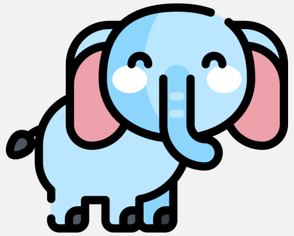


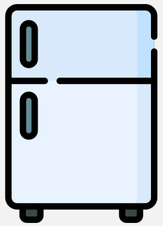


Properties

`size`	`area`
`weight`	`depth`
`pos`	`pos`
`nose_state`	`door_state`

Object-Oriented Thinking





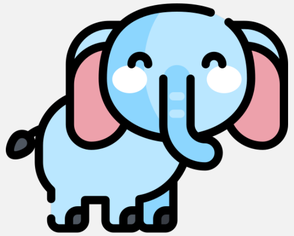
Properties

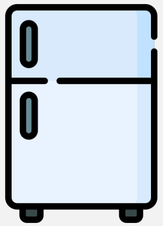
```
size
weight
pos
nose_state
```

```
area
depth
pos
door_state
```

Behaviors

Object-Oriented Thinking





Properties

```
size
weight
pos
nose_state
```

```
area
depth
pos
door_state
```

Behaviors

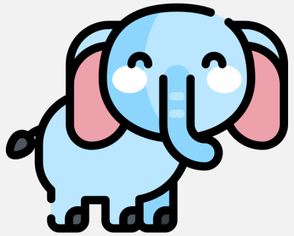
```
Sit()
Move()
Rollup()
```

```
Open()
Close()
```

Object-Oriented Thinking

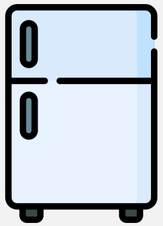
Properties

Behaviors



```
size
weight
pos
nose_state

Sit()
Move()
Rollup()
```



```
area
depth
pos
door_state

Open()
Close()
```

Encapsulation

g

b3

a1

x1

a2

c1

f

y2

x2

b1

c3

b2

c2

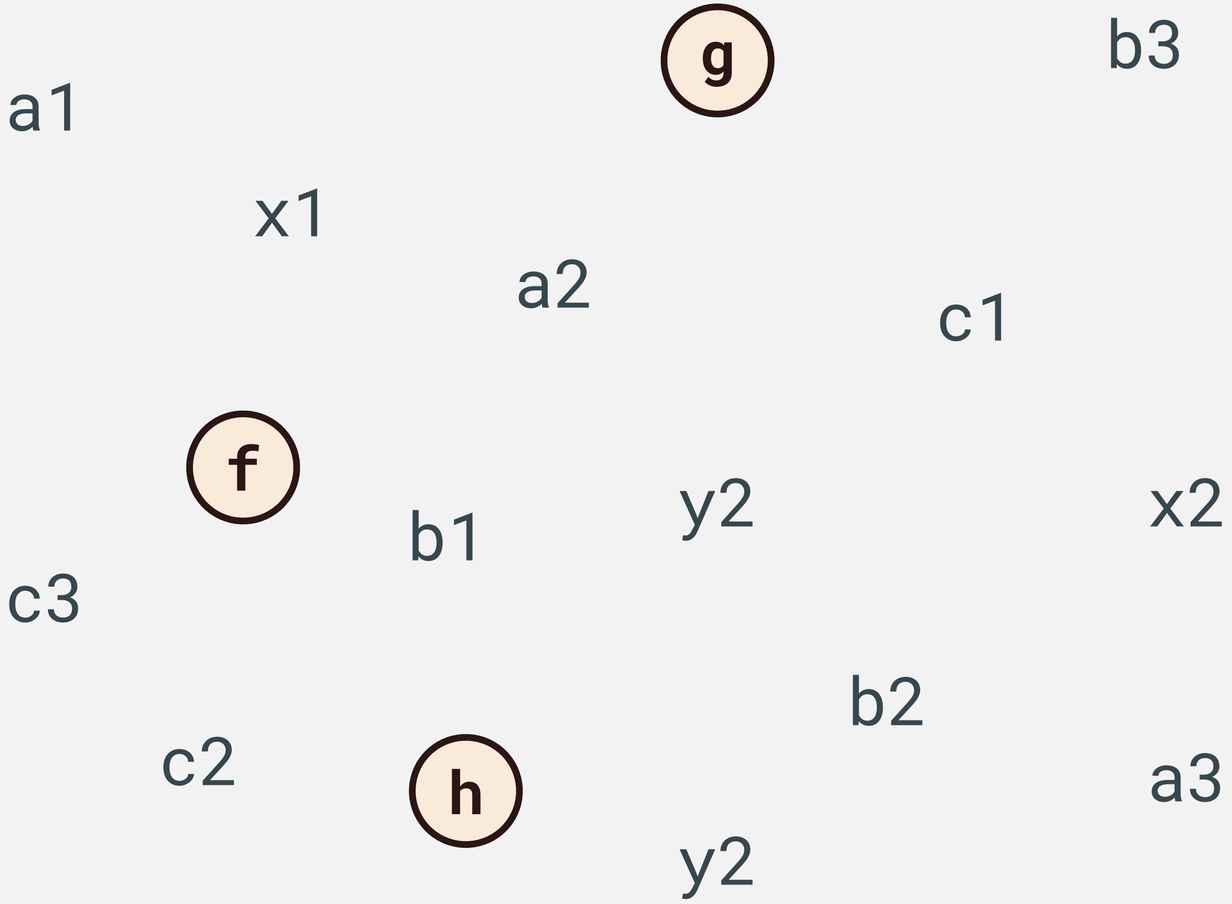
h

a3

y2

Encapsulation

1 **Group**



Encapsulation

1 **Group**

g

a2
b2
c2

a1
b1
c1

x2
y2

f

x1
y1

a3
b3
c3

h

Encapsulation

1. **Group**
2. **Hide**

g

a2
b2
c2

a1
b1
c1

x2
y2

f

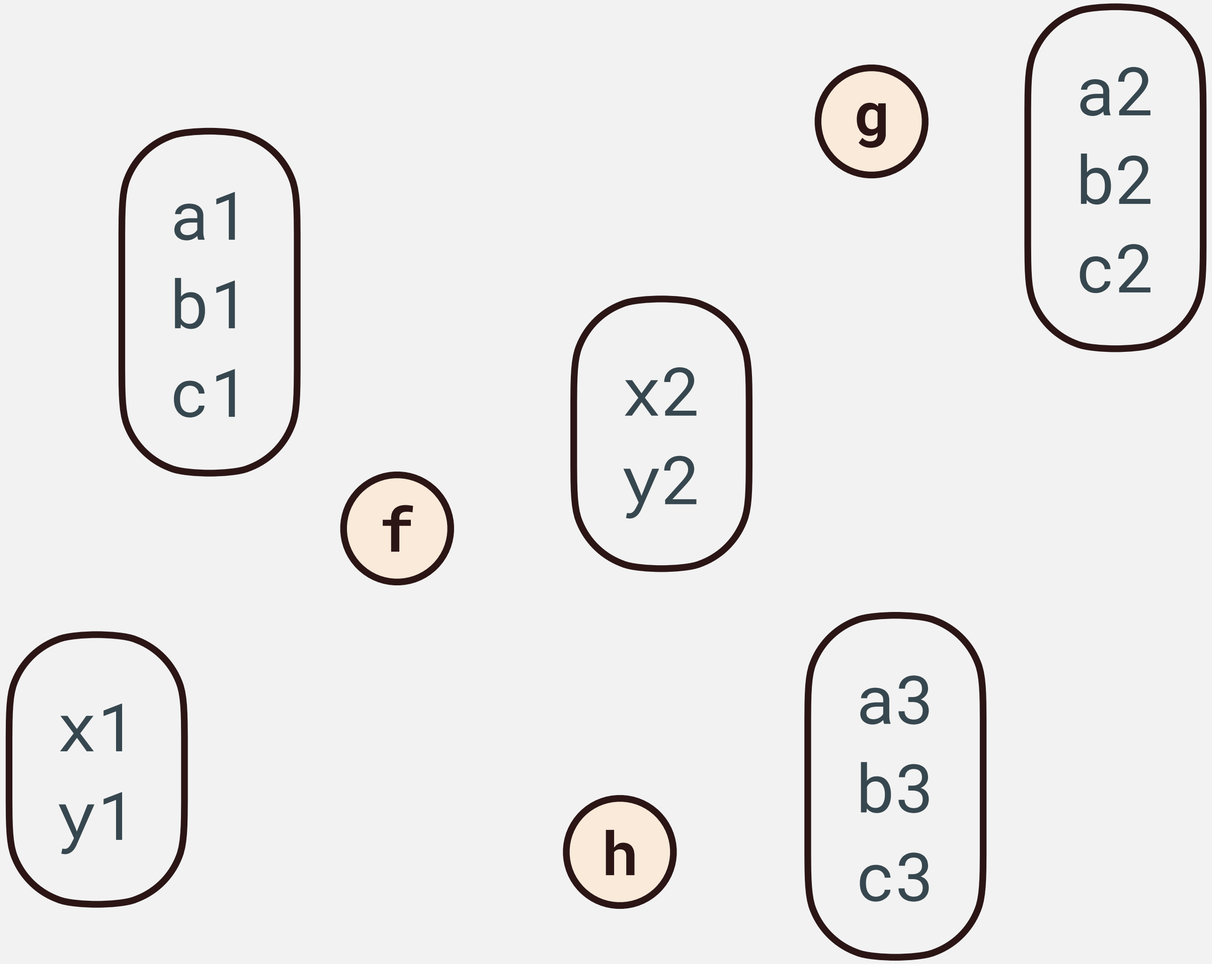
x1
y1

a3
b3
c3

h

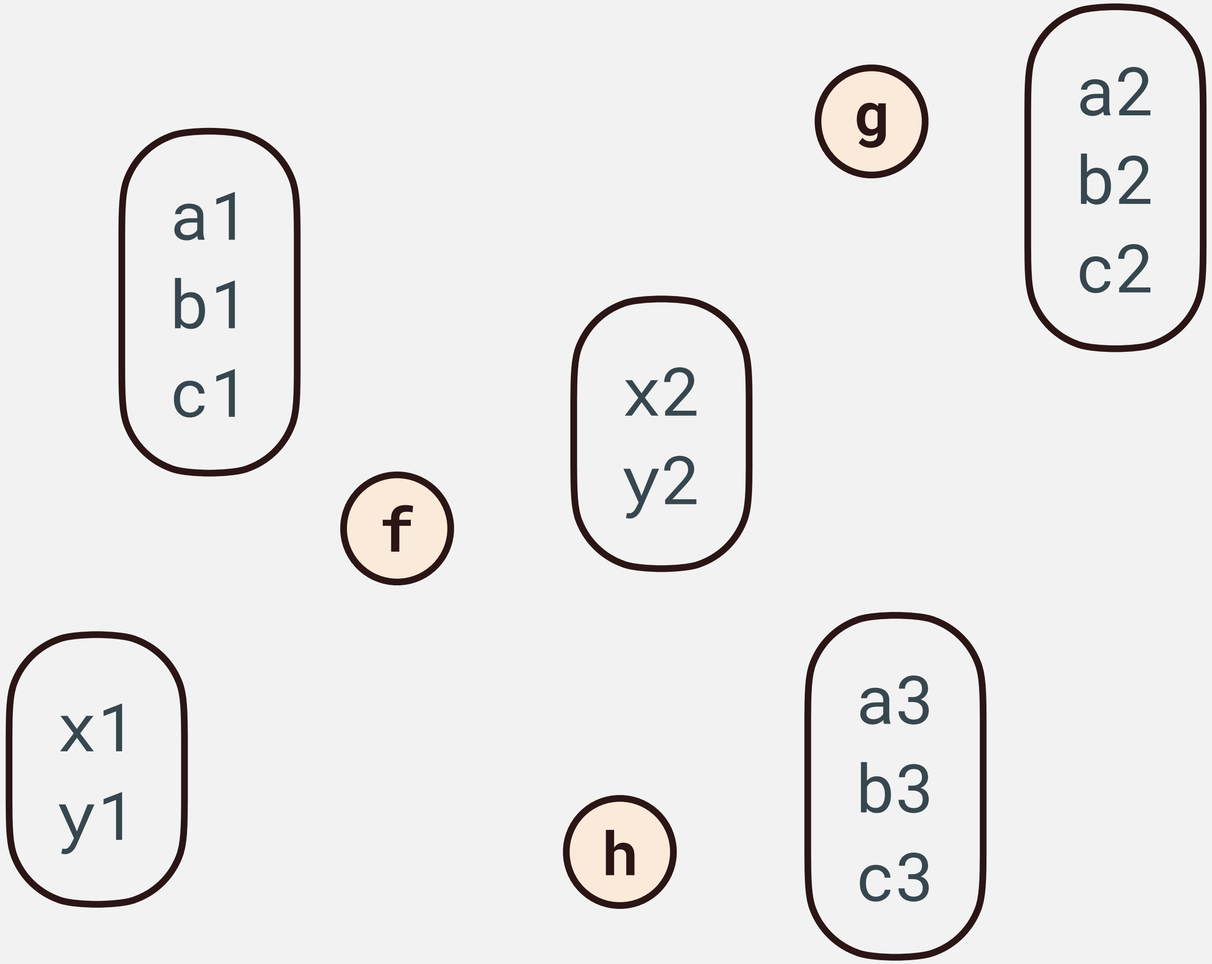
Encapsulation

1. **Group**
2. **Hide**



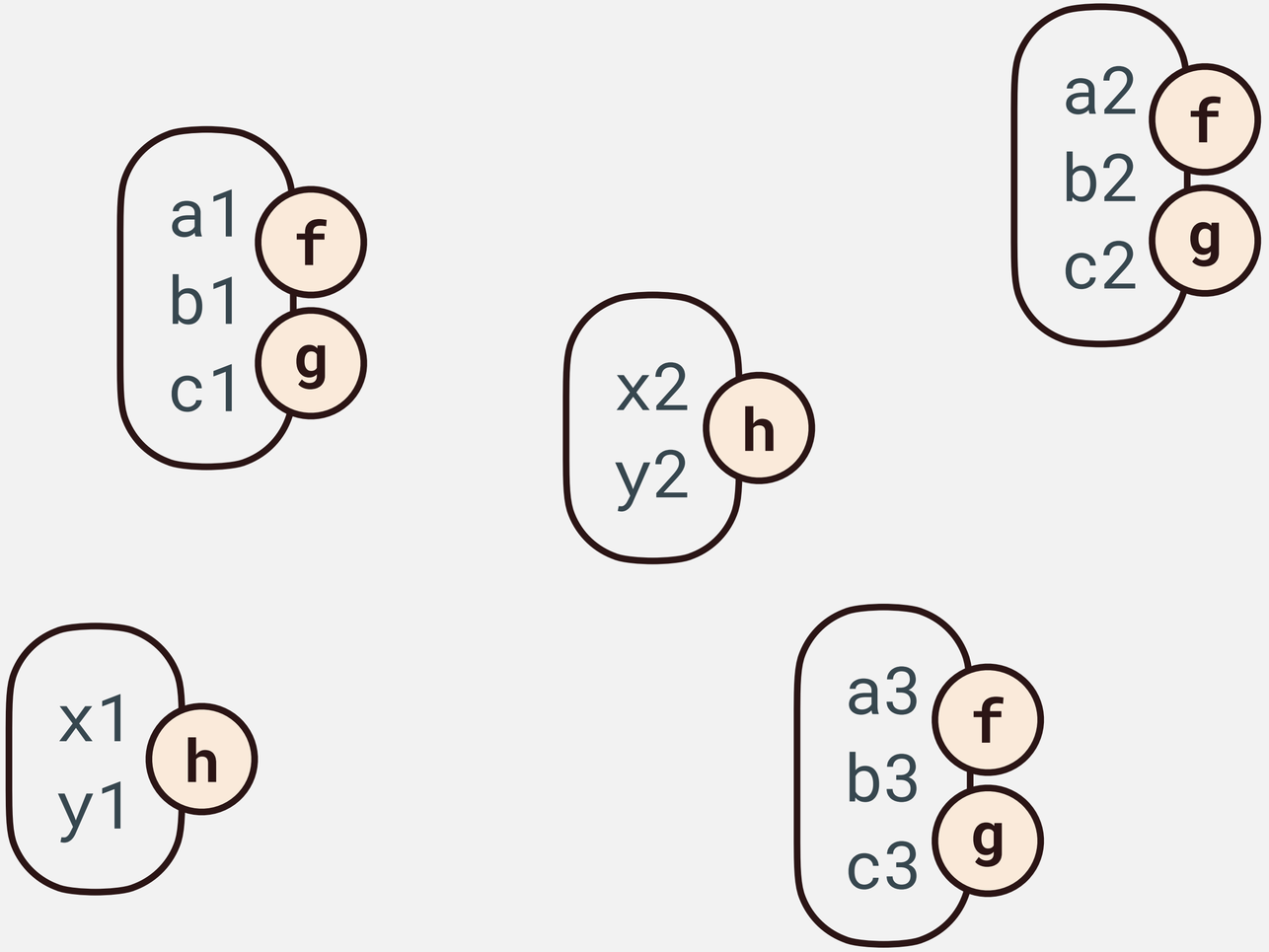
Encapsulation

1. **Group**
2. **Hide**
3. **Regulate Access**

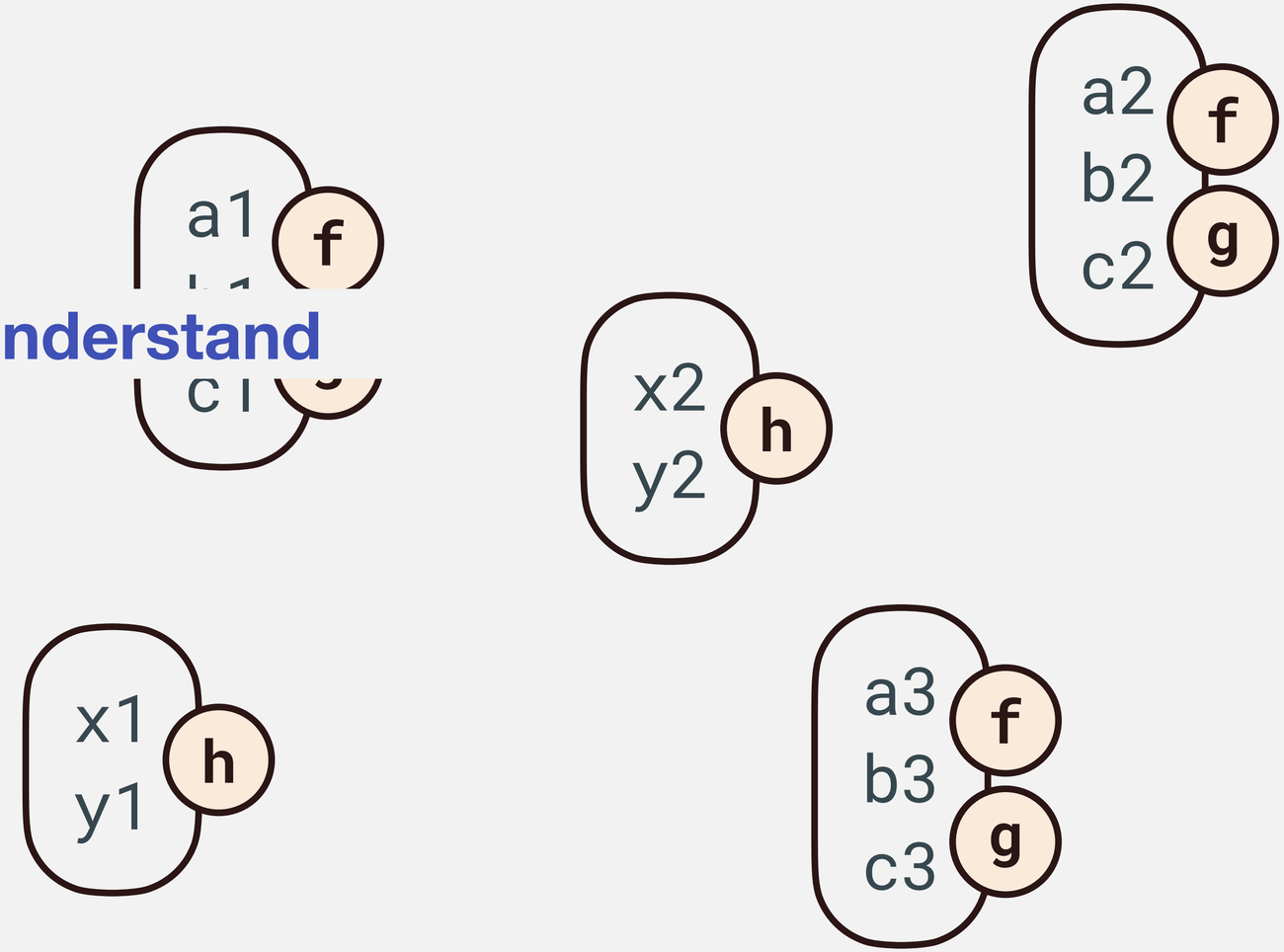


Encapsulation

1. **Group**
2. **Hide**
3. **Regulate Access**



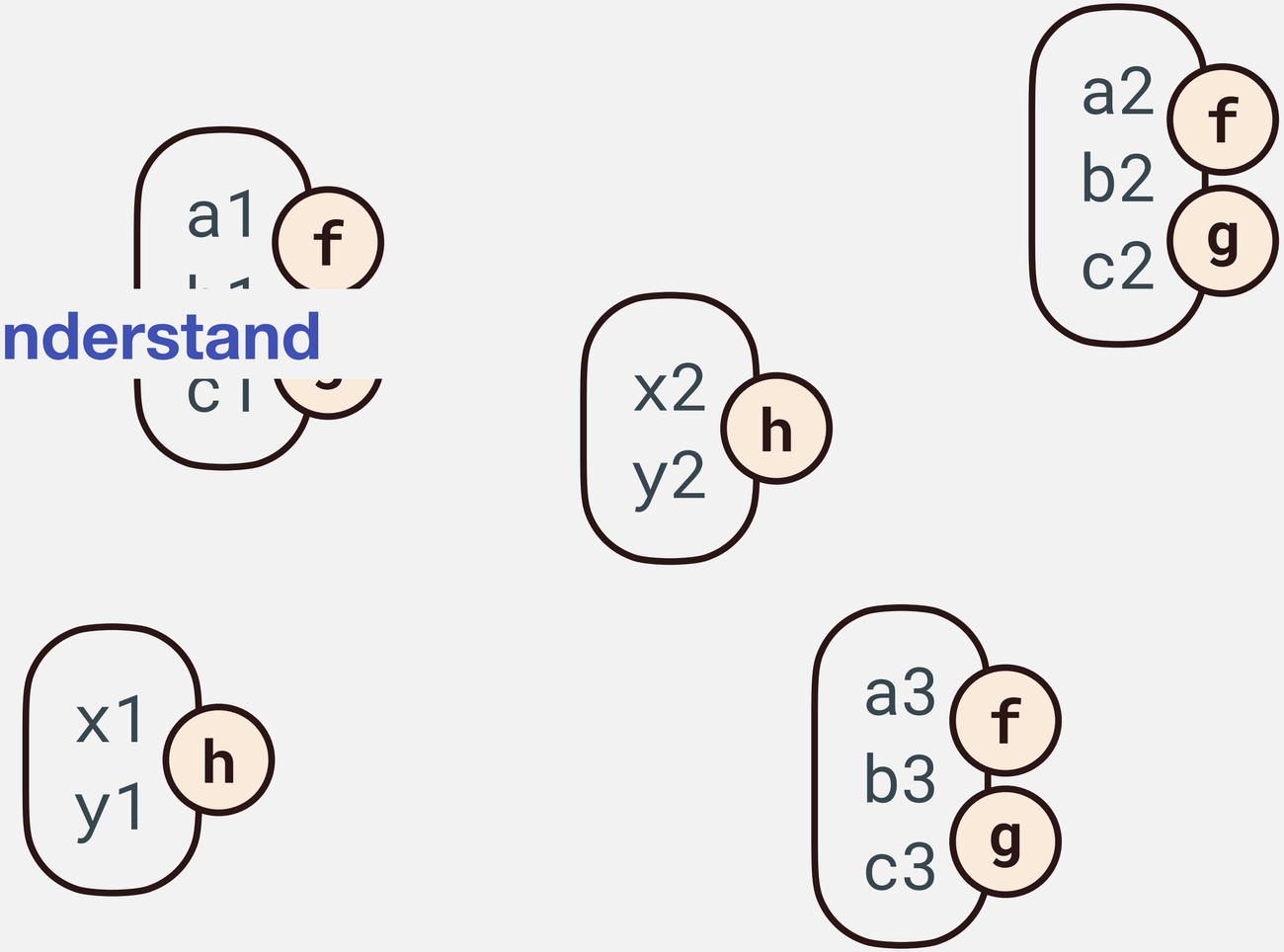
Encapsulation



Easier to maintain and understand

2. **Hide**
3. **Regulate Access**

Encapsulation



Easier to maintain and understand

Hide complex details

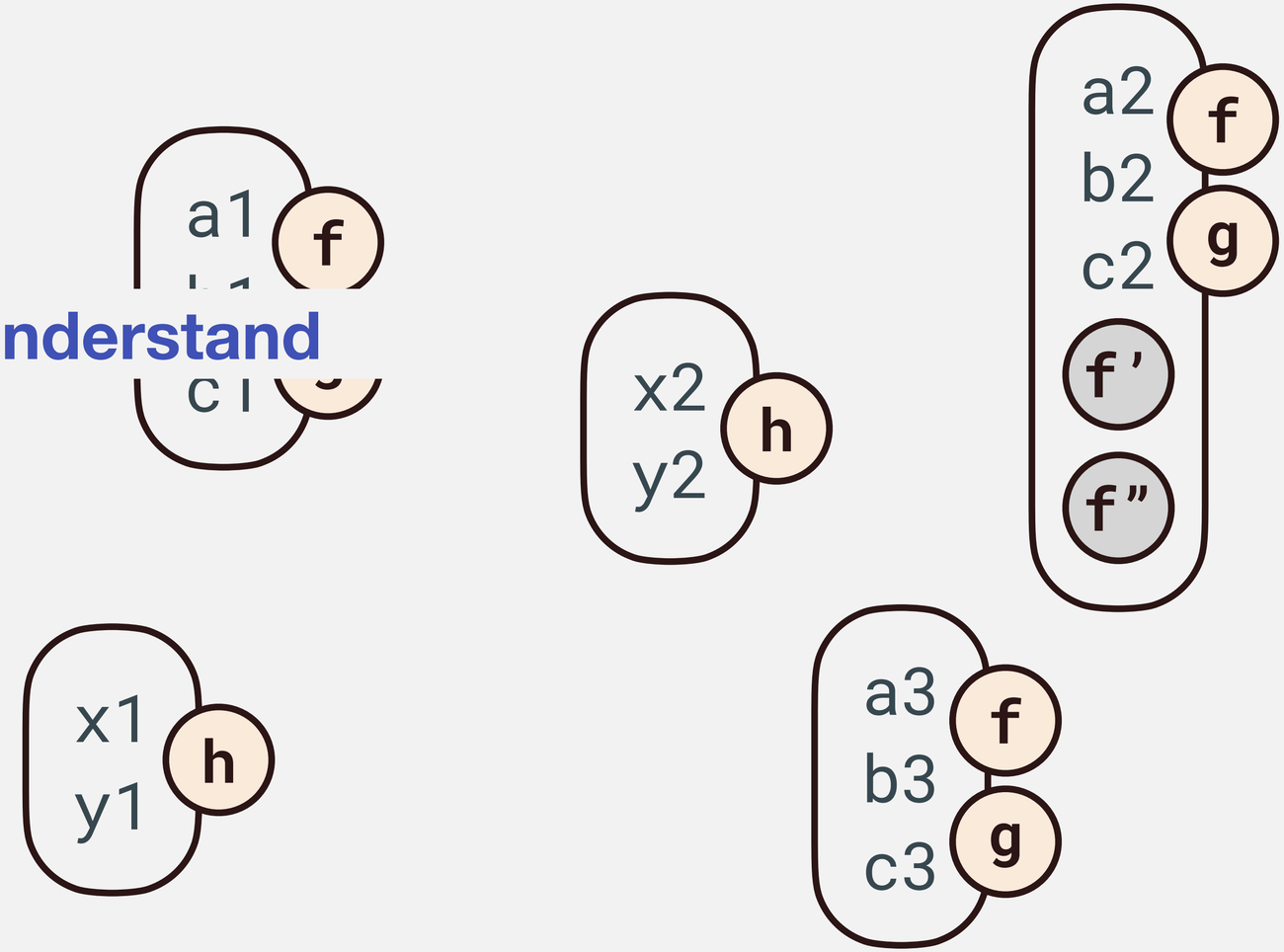
3 **Regulate Access**

Encapsulation

Easier to maintain and understand

Hide complex details

3 Regulate Access

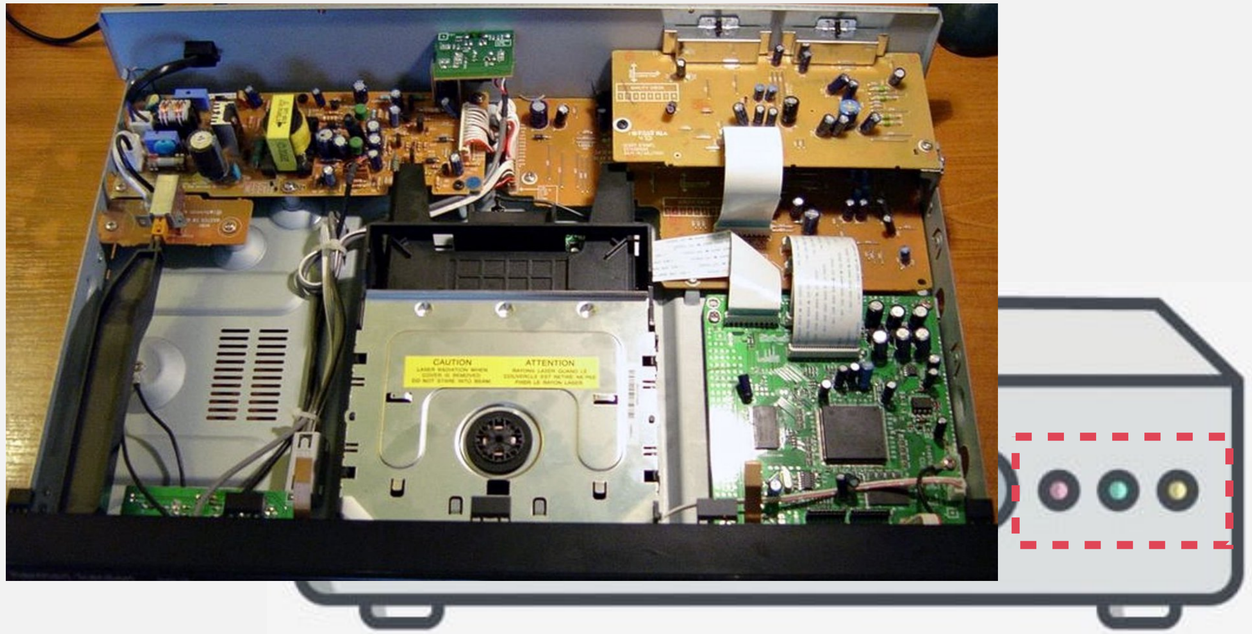


Hiding Complexity

Hiding Complexity

Interface

Hiding Complexity

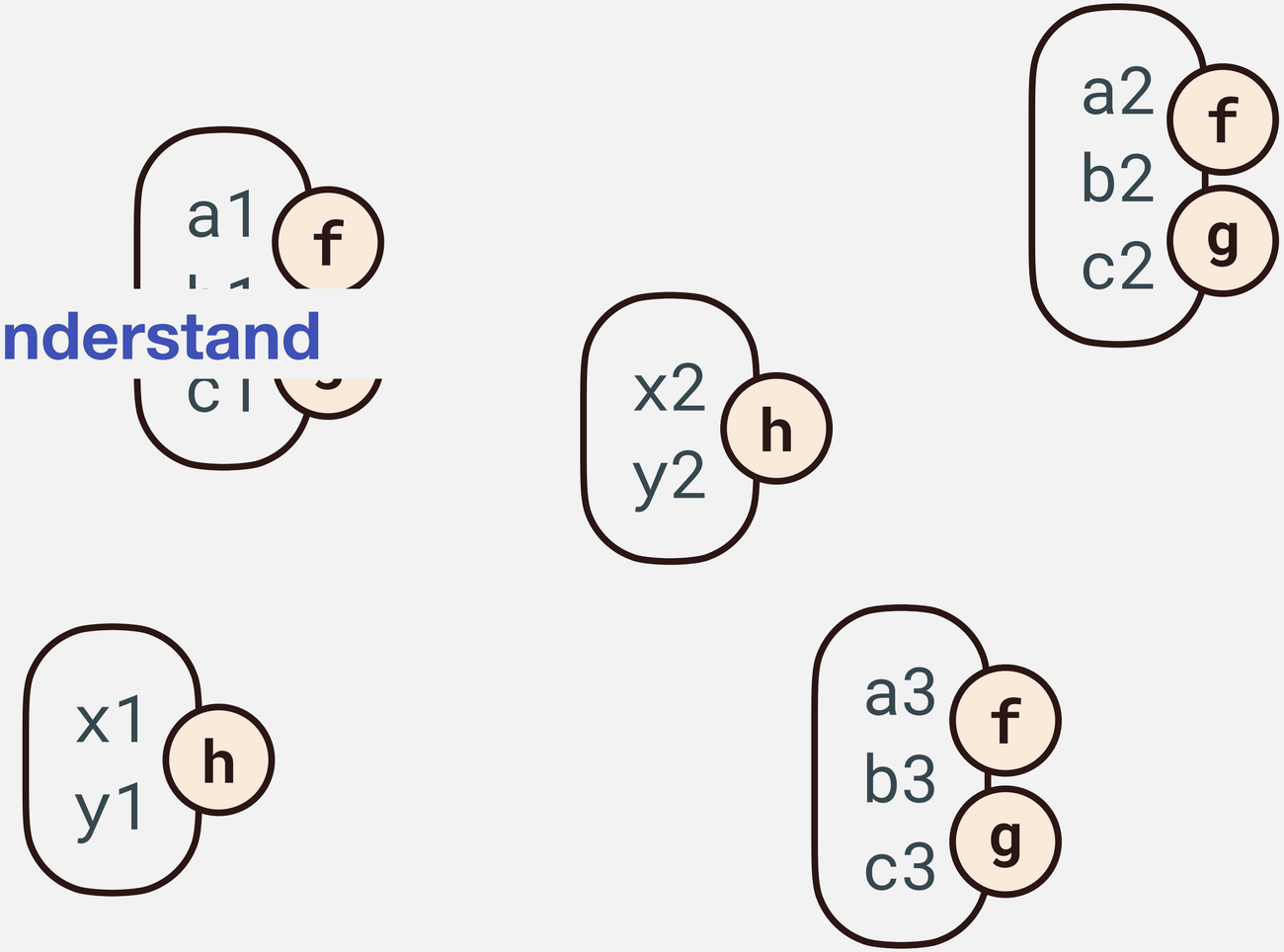


Interface

Hidden from outside

(Internal state, functionalities, …)

Encapsulation



Easier to maintain and understand

Hide complex details

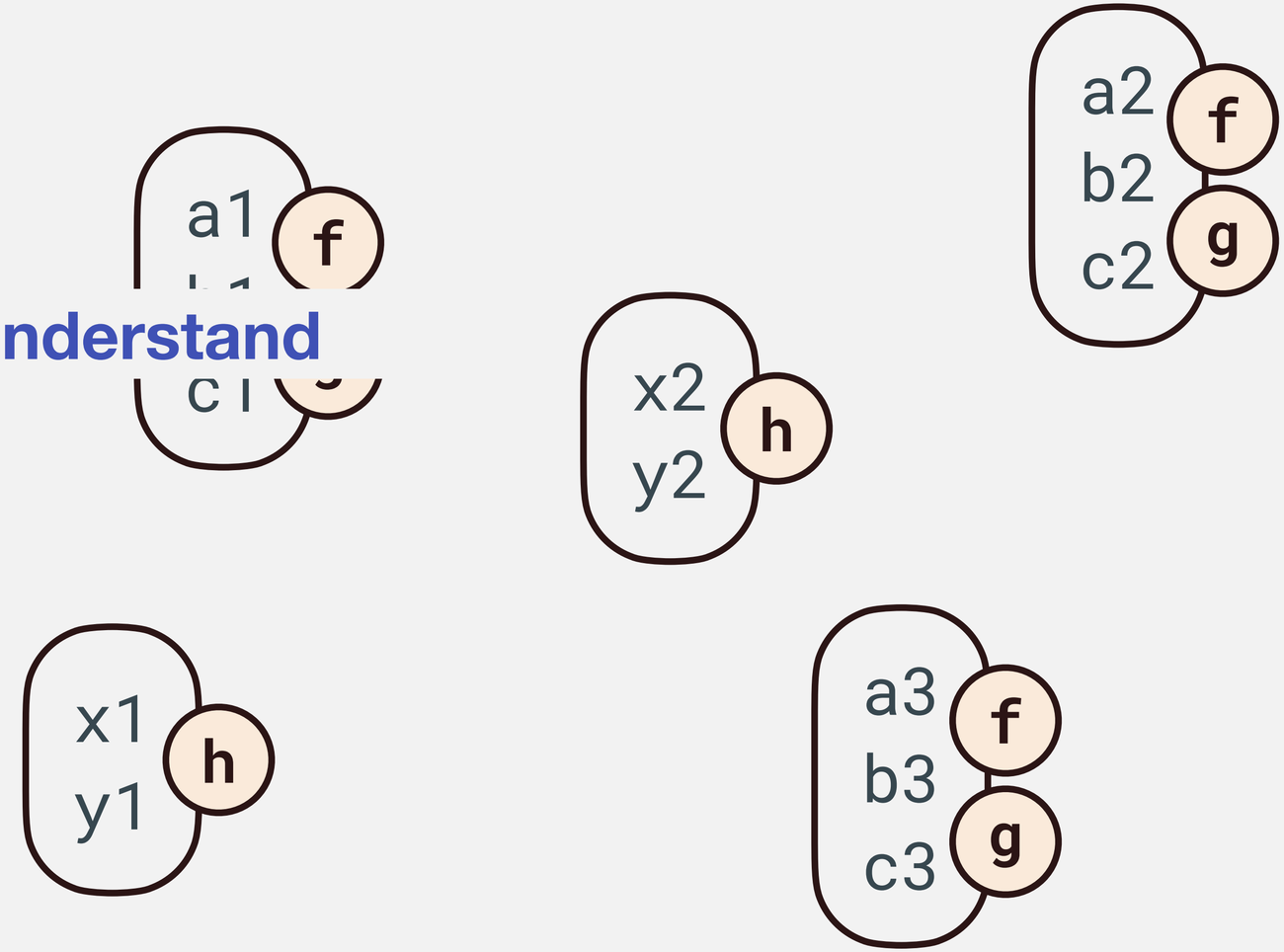
3 **Regulate Access**

Encapsulation

Easier to maintain and understand

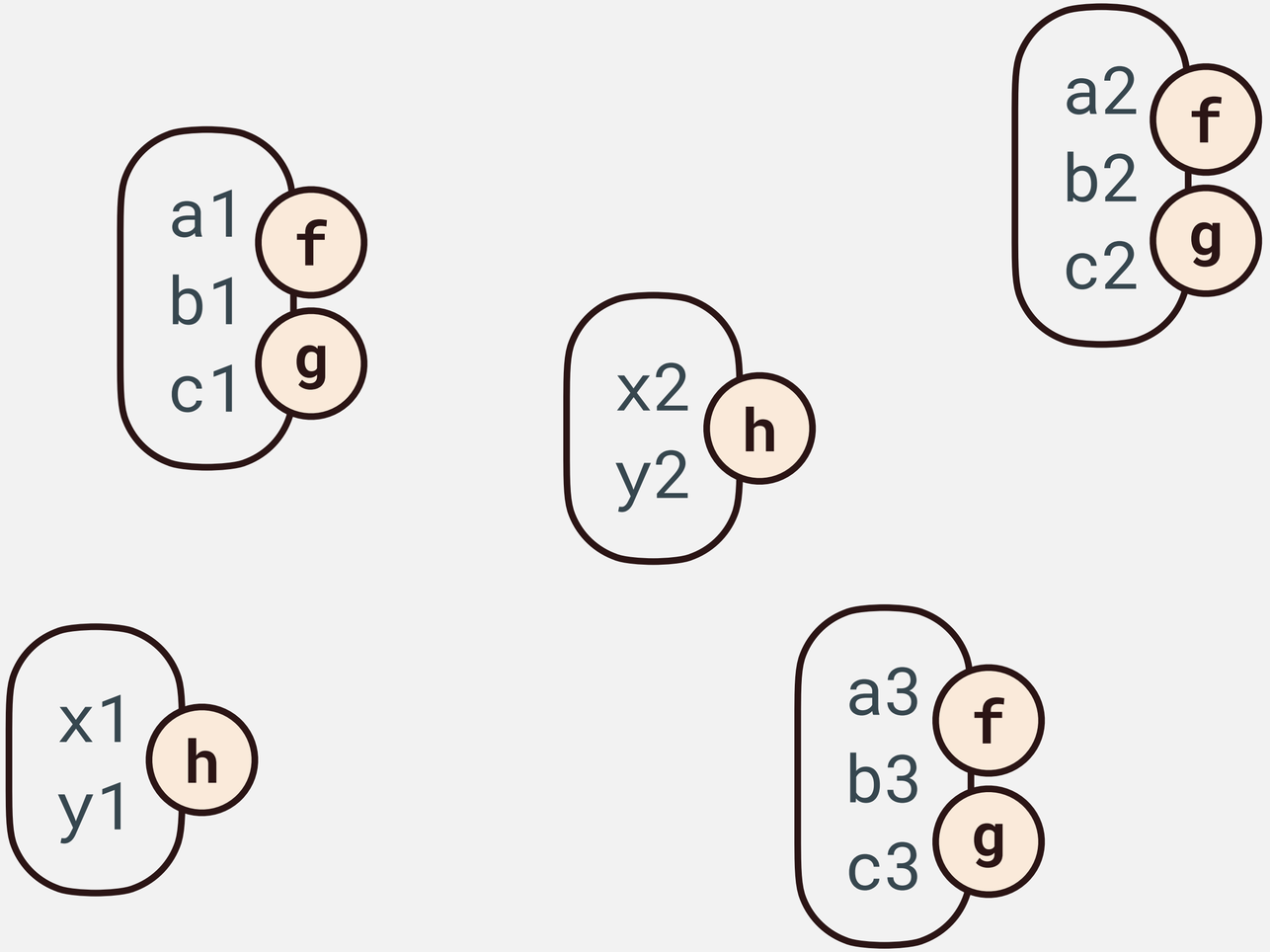
Hide complex details

Reduce human errors



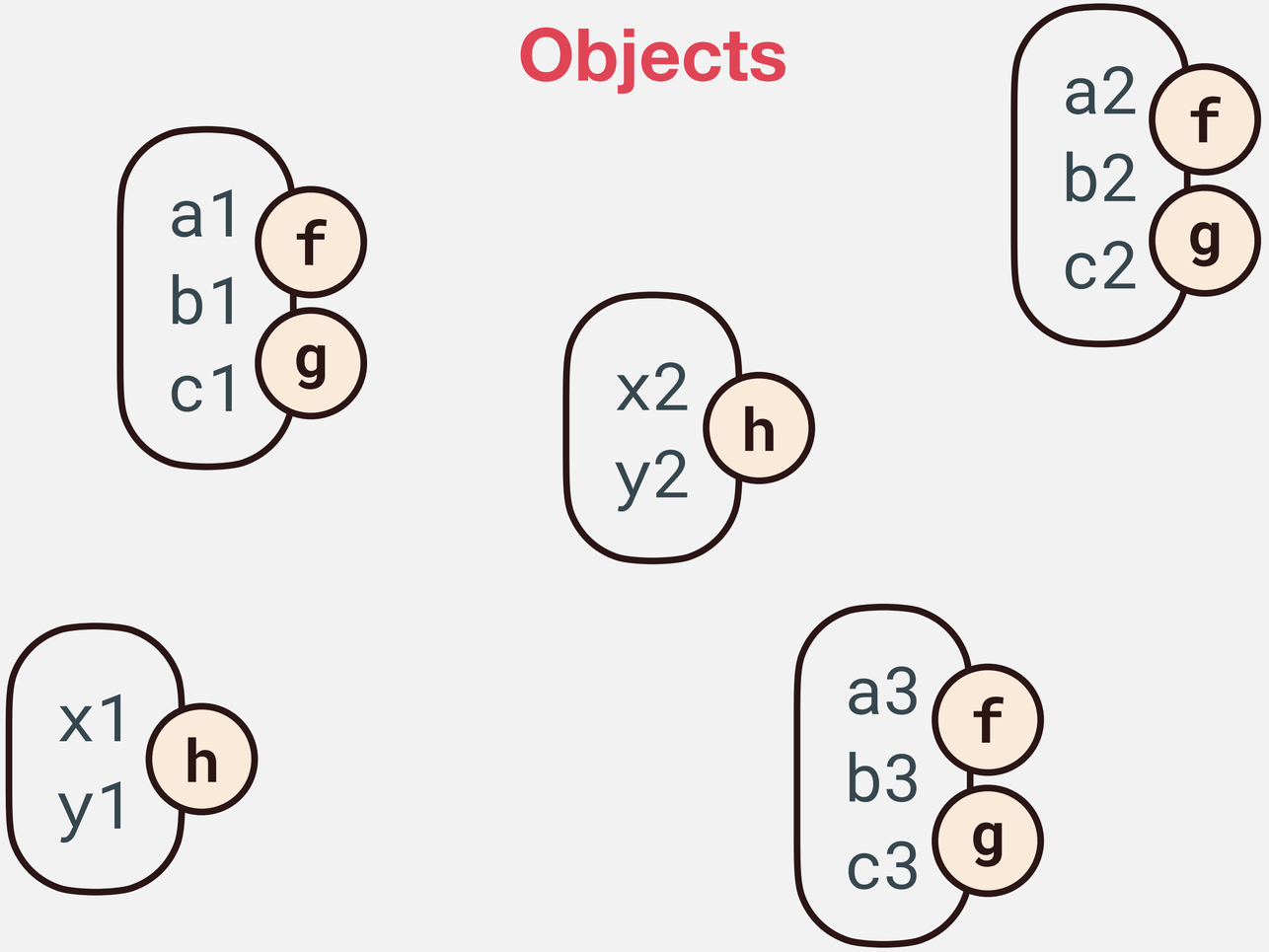
Encapsulation

Complexity ↓



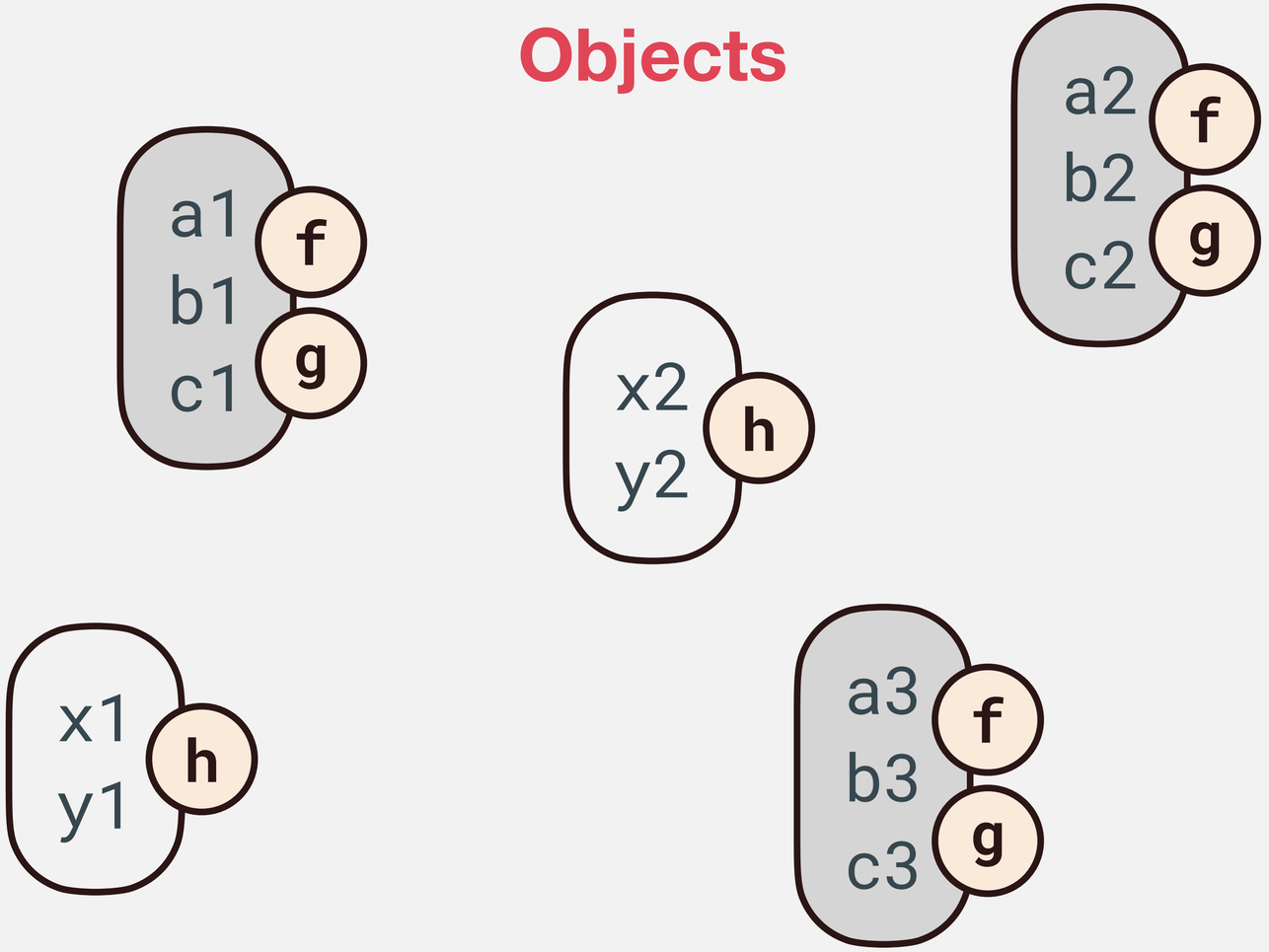
Encapsulation

Complexity ↓



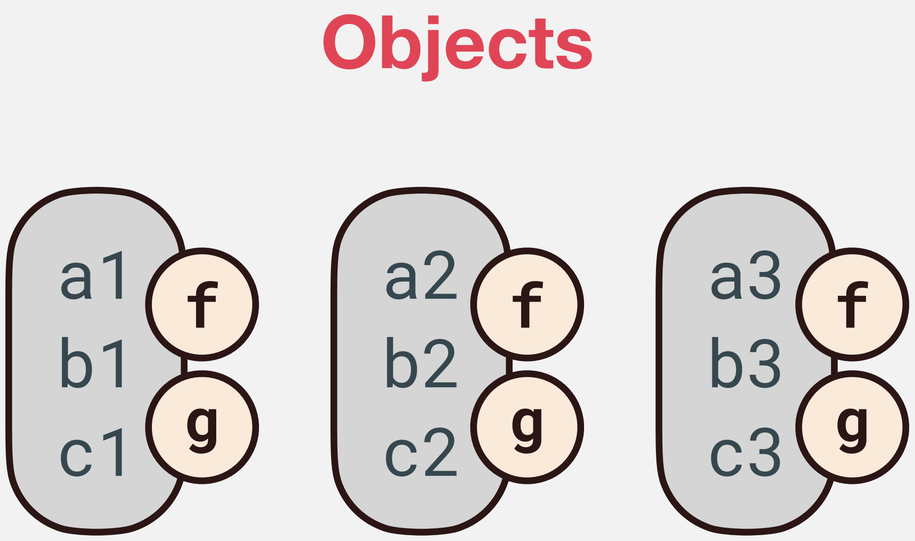
Encapsulation

Complexity ↓



Abstraction

➜ Create a concept/template to describe a **class** of objects



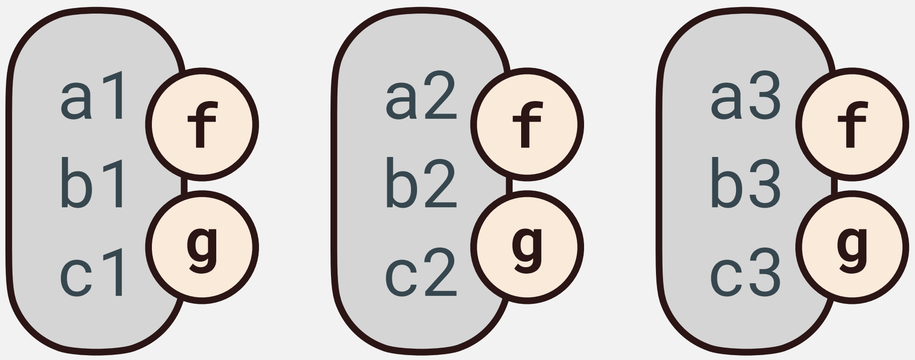
Abstraction

➜ Create a concept/template to describe a **class** of objects

Properties

Behaviors

Objects



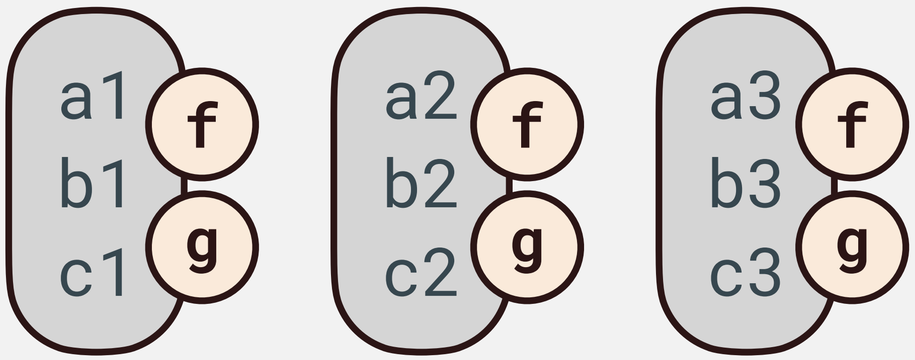
Abstraction

➜ Create a concept/template to describe a **class** of objects

Properties `a, b, c`

Behaviors

Objects



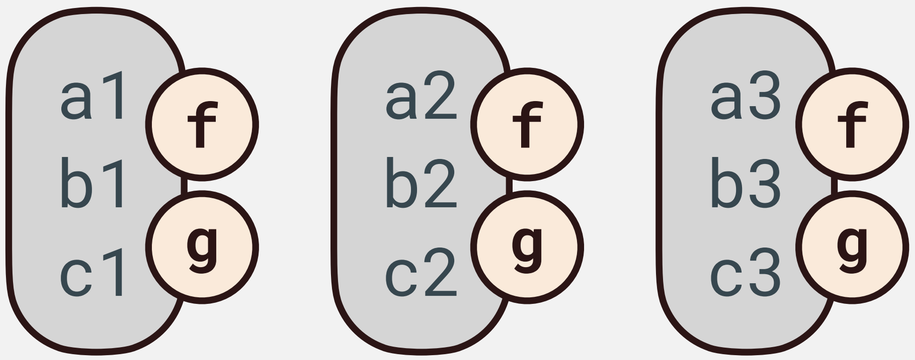
Abstraction

➜ Create a concept/template to describe a **class** of objects

Properties a, b, c

Behaviors f g f' f"

Objects



Abstraction

Class	Object
Abstracted Concept	Actual Thing

Abstraction

Class	Object
Abstracted Concept	Actual Thing
“Elephant”	

Abstraction

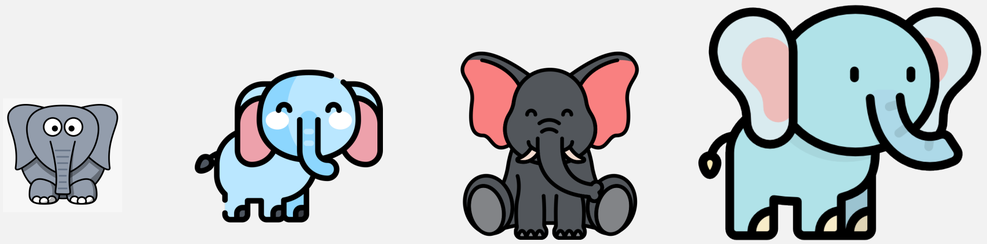
Class

Abstracted Concept

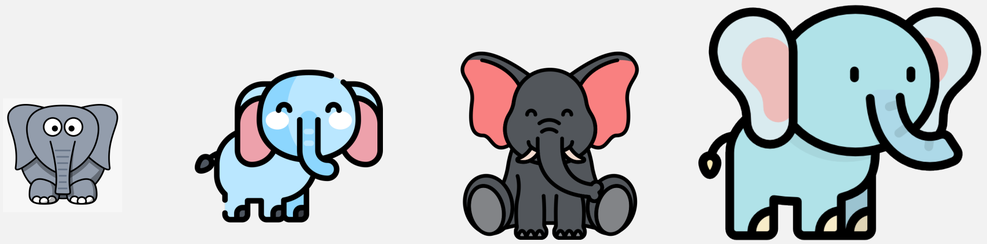
“Elephant”

Object

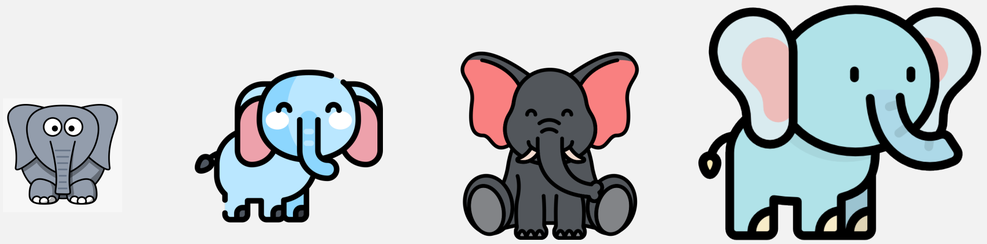
Actual Thing



Abstraction

Class	Object
Abstracted Concept	Actual Thing
“Elephant”	
“Triangle”	

Abstraction

Class	Object
Abstracted Concept	Actual Thing
“Elephant”	
“Triangle”	

C Struct

- Only the “data group” step of Encapsulation
  - data members can be accessed and modified **freely** from outside

```
struct Student {
  long id;
  char name[64];
  int department_id;
};
```

C Struct

- Only the “data group” step of Encapsulation
  - data members can be accessed and modified **freely** from outside

```c
struct Student {
  long id;
  char name[64];
  int department_id;
};
```

```c
int main() {
  struct Student s1;
  s1.id = 10001;
  strcpy(s1.name, "Juan Wang");
  s1.department_id = 47;
}
```

C Struct

➜ Only the “data group” step of Encapsulation

- data members can be accessed and modified **freely** from outside

```c
typedef struct Student {
  long id;
  char name[64];
  int department_id;
} student;
```

```c
int main() {
  struct Student s1;
  s1.id = 10001;
  strcpy(s1.name, "Juan Wang");
  s1.department_id = 47;
}
```

C Struct

➜ Only the “data group” step of Encapsulation

- data members can be accessed and modified **freely** from outside

```c
typedef struct Student {
  long id;
  char name[64];
  int department_id;
} student;
```

```c
int main() {
  student s1;
  s1.id = 10001;
  strcpy(s1.name, "Juan Wang");
  s1.department_id = 47;
}
```

C Struct

➜ Only the “data group” step of Encapsulation

- data members can be accessed and modified **freely** from outside

```c
typedef struct Student {
  long id;
  char name[64];
  int department_id;
} student;
```

```c
int main() {
  student s1;
  s1.id = 10001;
  strcpy(s1.name, "Juan Wang");
  s1.department_id = 47;
}
```

C Struct

➜ Only the “data group” step of Encapsulation

- data members can be accessed and modified **freely** from outside

```c
typedef struct Student {
  long id;
  char name[64];
  int department_id;
} student;
```

```c
int main() {
  student *s1 = (student*)
    malloc(sizeof(student));
  s1.id = 10001;
  strcpy(s1.name, "Juan Wang");
  s1.department_id = 47;
}
```

C Struct

➜ Only the “data group” step of Encapsulation

- data members can be accessed and modified **freely** from outside

```c
typedef struct Student {
  long id;
  char name[64];
  int department_id;
} student;
```

```c
int main() {
  student *s1 = (student*)
    malloc(sizeof(student));
  s1->id = 10001;
  strcpy(s1->name, "Juan Wang");
  s1->department_id = 47;
}
```

C++

- C with Object-Oriented Extension (initial goal)
  - “C with classes”
  - Created by Bjarne Stroustrup in 1980s
- Standardization
  - **C++98**
  - C++03
  - **C++11**
  - C++14
  - C++17
  - C++20

Class Members

```
class Date {
  Date(int d, int m, int y);
  int ToDaysInYear();
  void AddMonth(int m);

  int day;
  int month;
  int year;
};
```

Class Members

```
class Date {
  Date(int d, int m, int y);
  int ToDaysInYear();
  void AddMonth(int m);

  int day;
  int month;
  int year;
};
```

Data Members

Class Members

```
class Date {
  Date(int d, int m, int y);
  int ToDaysInYear();
  void AddMonth(int m);

  int day;
  int month;
  int year;
};
```

Member Functions

- Declared within a class
- Can be invoked only by objects of this class

Data Members

Access Control

```
class Date {
  Date(int d, int m, int y);
  int ToDaysInYear();
  void AddMonth(int m);

  int day;
  int month;
  int year;
};
```

- Members are private by default

Access Control

```
class Date {
  Date(int d, int m, int y);
  int ToDaysInYear();
  void AddMonth(int m);

  int day;
  int month;
  int year;
};
```

- Members are private by default
- Use `public:` / `private:` as a switch

Access Control

```
class Date {
 public:
  Date(int d, int m, int y);
  int ToDaysInYear();
  void AddMonth(int m);

  int day;
  int month;
  int year;
};
```

- Members are private by default
- Use `public:` / `private:` as a switch

Access Control

```
class Date {
 public:
  Date(int d, int m, int y);
  int ToDaysInYear();
  void AddMonth(int m);
 private:
  int day;
  int month;
  int year;
};
```

- Members are private by default
- Use `public:` / `private:` as a switch

Access Control

```
class Date {
 public:
  Date(int d, int m, int y);
  int ToDaysInYear();
  void AddMonth(int m);
 private:
  int day_;
  int month_;
  int year_;
};
```

- Members are private by default
- Use `public:` / `private:` as a switch

Class Members

```
class Date {
 public:
  Date(int d, int m, int y);
  int ToDaysInYear();
  void AddMonth(int m);
 private:
  int day_;
  int month_;
  int year_;
};
```

```
void func(Date *today) {
  today->AddMonth(1);
}
```

Class Members

```
class Date {
 public:
  Date(int d, int m, int y);
  int ToDaysInYear();
  void AddMonth(int m);
 private:
  int day_;
  int month_;
  int year_;
};
```

```
void func(Date *today) {
  today->AddMonth(1);
  (today->month_)++;
}
```

Implementing Member Functions

```
class Date {
 public:
  Date(int d, int m, int y);
  int ToDaysInYear();
  void AddMonth(int m);
 private:
  int day_;
  int month_;
  int year_;
};
```

```
void func(Date *today) {
  today->AddMonth(1);
}
```

Implementing Member Functions

```
class Date {
 public:
  Date(int d, int m, int y);
  int ToDaysInYear();
  void AddMonth(int m);
 private:
  int day_;
  int month_;
  int year_;
};
```

```
void func(Date *today) {
  today->AddMonth(1);
}

void Date::AddMonth(int m) {
  month_ += (m - 1);
  year_ += (month_ / 12);
  month_ = (month_ % 12) + 1;
}
```

Implementing Member Functions

```
class Date {
 public:
  Date(int d, int m, int y);
  int ToDaysInYear();
  void AddMonth(int m);
 private:
  int day_;
  int month_;
  int year_;
};
```

```
void func(Date *today) {
  today->AddMonth(1);
}

void Date::AddMonth(int m) {
  month_ += (m - 1);
  year_ += (month_ / 12);
  month_ = (month_ % 12) + 1;
}
```

Implementing Member Functions

```
class Date {
 public:
  Date(int d, int m, int y);
  int ToDaysInYear();
  void AddMonth(int m);
 private:
  int day_;
  int month_;
  int year_;
};
```

```
void func(Date *today) {
  today->AddMonth(1);
}

void Date::AddMonth(int m) {
  month_ += (m - 1);
  year_ += (month_ / 12);
  month_ = (month_ % 12) + 1;
}
```

Constructors

```
class Date {
 public:
  Date(int d, int m, int y);
  int ToDaysInYear();
  void AddMonth(int m);
 private:
  int day_;
  int month_;
  int year_;
};
```

Constructors

```
class Date {
 public:
  Date(int d, int m, int y);
  int ToDaysInYear();
  void AddMonth(int m);
 private:
  int day_;
  int month_;
  int year_;
};
```

- To initialize the data members

Constructors

```
class Date {
 public:
  Date(int d, int m, int y);
  int ToDaysInYear();
  void AddMonth(int m);
 private:
  int day_;
  int month_;
  int year_;
};
```

- To initialize the data members
- Function name = class name

Constructors

```
class Date {
 public:
  Date(int d, int m, int y);
  int ToDaysInYear();
  void AddMonth(int m);
 private:
  int day_;
  int month_;
  int year_;
};
```

- To initialize the data members
- Function name = class name
- Can take arbitrary number of arguments
- No return value

Constructors

```
class Date {
 public:
  Date(int d, int m, int y);
  int ToDaysInYear();
  void AddMonth(int m);
 private:
  int day_;
  int month_;
  int year_;
};
```

```
void func() {
  Date today(11, 10, 2022);
  today.AddMonth(3);
}
```

Constructors

```
class Date {
 public:
  Date(int d, int m, int y);
  int ToDaysInYear();
  void AddMonth(int m);
 private:
  int day_;
  int month_;
  int year_;
};
```

```
void func() {
  Date today(11, 10, 2022);
  today.AddMonth(3);
  Date tomorrow(12, 10);
  Date yesterday;
}
```

Constructors

```
class Date {
 public:
  Date();
  Date(int d, int m);
  Date(int d, int m, int y);
  int ToDaysInYear();
  void AddMonth(int m);
 private:
  int day_, month_, year_;
};
```

```
void func() {
  Date today(11, 10, 2022);
  today.AddMonth(3);
  Date tomorrow(12, 10);
  Date yesterday;
}
```

Constructors

```
class Date {
 public:
  Date();
  Date(int d, int m);
  Date(int d, int m, int y);
  int ToDaysInYear();
  void AddMonth(int m);
 private:
  int day_, month_, year_;
};
```

```
void func() {
  Date today(11, 10, 2022);
  today.AddMonth(3);
  Date tomorrow(12, 10);
  Date yesterday;
}
```

Function Overloading

- Functions have the same name but different parameters

Constructors

```
class Date {
 public:
  Date();
  Date(int d, int m);
  Date(int d, int m, int y);
  int ToDaysInYear();
  void AddMonth(int m);
 private:
  int day_, month_, year_;
};
```

```
void func() {
  Date today(11, 10, 2022);
  today.AddMonth(3);
  Date tomorrow(12, 10);
  Date yesterday;
}
```

Function Overloading

- Functions have the same name but different parameters

Constructors

```
class Date {
 public:
  Date();
  Date(int d, int m);
  Date(int d, int m, int y);
  int ToDaysInYear();
  void AddMonth(int m);
 private:
  int day_, month_, year_;
};
```

Default Constructor

```
void func() {
  Date today(11, 10, 2022);
  today.AddMonth(3);
  Date tomorrow(12, 10);
  Date yesterday;
}
```

Function Overloading

- Functions have the same name but different parameters

Constructors

```
class Date {
 public:
  int ToDaysInYear();
  void AddMonth(int m);
 private:
  int day_, month_, year_;
};
```

```
void func() {
  Date yesterday;
}
```

Constructors

```
class Date {
 public:
  int ToDaysInYear();
  void AddMonth(int m);
 private:
  int day_, month_, year_;
};
```

```
void func() {
  Date yesterday;
}
```

Constructors

```
class Date {
 public:
  int ToDaysInYear();
  void AddMonth(int m);
 private:
  int day_, month_, year_;
};
```

```
void func() {
  Date yesterday;
}
```

- Default constructor is automatically generated by the compiler

Constructors

```
class Date {
 public:
  int ToDaysInYear();
  void AddMonth(int m);
 private:
  int day_, month_, year_;
};
```

```
void func() {
  Date yesterday;
}
```

- Default constructor is automatically generated by the compiler

  only if no constructor is declared in the class

Implementing Constructors

```
Date::Date(int d, int m, int y) {
  day_ = d;
  month_ = m;
  year_ = y;
};
```

Implementing Constructors

```
Date::Date(int d, int m, int y)
    : day_(d), month_(m), year_(y) {}
```

Implementing Constructors

```
Date::Date(int d, int m, int y)
    : day_(d), month_(m), year_(y) {}
```

Initializer List

Implementing Constructors

```
Date::Date(int d, int m, int y)
    : day_(d), month_(m), year_(y) {}

Date::Date(int d, int m)
    : day_(d), month_(m), year_(0) {}
```

Initializer List

Prefer Initializer List

```
class A {
 public:
  A(int n) { x = n; }
  int x;
};
class B {
 public:
  B(int n) { a_.x = n; }
 private:
  A a_;
};
```

Prefer Initializer List

```
class A {
 public:
  A(int n) { x = n; }
  int x;
};
class B {
 public:
  B(int n) { a_.x = n; }
 private:
  A a_;
};
```

Prefer Initializer List

```
class A {
 public:
  A(int n) { x = n; }
  int x;
};
class B {
 public:
  B(int n) : a_(n) {}
 private:
  A a_;
};
```

Prefer Initializer List

```
class A {
 public:
  A(int n) { x = n; }
  int x;
};
class B {
 public:
  B(int n) : a_(n) {}
 private:
  A a_;
};
```

- To avoid unnecessary calls to default constructors
- To initialize base class members

Copy Constructor

```
Date::Date(Date &date) {
  day_ = date.day_;
  month_ = date.month_;
  year_ = date.year_;
}
```

```
void func() {
  Date today(24, 10, 2023);
  Date tomorrow = today;
}
```

Copy Constructor

```
Date::Date(Date &date) {
  day_ = date.day_;
  month_ = date.month_;
  year_ = date.year_;
}
```

```
void func() {
  Date today(24, 10, 2023);
  Date tomorrow = today;
}
```

Reference Type in C++

lvalue, rvalue

`lvalue`	`rvalue`
➜ Associated with a specific memory location	

lvalue, rvalue

`lvalue`	`rvalue`
➜ Associated with a specific memory location	
➜ Persists beyond a single expression	

lvalue, rvalue

`lvalue`	`rvalue`
➜ Associated with a specific memory location	
➜ Persists beyond a single expression	
➜ Can be on the left side of =	

lvalue, rvalue

`lvalue`

- Associated with a specific memory location
- Persists beyond a single expression
- Can be on the left side of =

`rvalue`

- Anything that is not an lvalue
  Does not point to anywhere
- Temporary within an expression
- CANNOT be on the left side of =

lvalue, rvalue

```
int a = 666;
```

lvalue, rvalue

```
int a = 666;
```

lvalue

lvalue, rvalue

```
int a = 666;
```

lvalue rvalue

lvalue, rvalue

```
int a = 666;
```

lvalue rvalue

```
int *p = &a;
```

lvalue, rvalue

```
int a = 666;
```

lvalue **rvalue**

```
int *p = &a;
```

```
int 666 = a;
```

lvalue, rvalue

```
int a = 666;
```

lvalue rvalue

```
int *p = &a;
```

```
int 666 = a;
```

lvalue, rvalue

```
int a = 666;
```

lvalue rvalue

```
int *p = &a;
```

```
int 666 = a;
```

```
int *p = &666;
```

lvalue, rvalue

```
int a = 666;
```

lvalue rvalue

```
int *p = &a;
```

```
int 666 = a;
```

```
int *p = &666;
```

lvalue, rvalue

```
int a = 666;
```

lvalue rvalue

```
int *p = &a;
```

```
int 666 = a;
```

```
int *p = &666;
```

```
int a = b + c;
```

lvalue, rvalue

```
int a = 666;
```

lvalue rvalue

```
int *p = &a;
```

```
int 666 = a;
```

```
int *p = &666;
```

```
int a = b + c;
```

rvalue

lvalue, rvalue

```
int a = 666;
```

lvalue rvalue

```
int *p = &a;
```

```
int 666 = a;
```

```
int *p = &666;
```

```
int a = b + c;
```

rvalue

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

int main() {
  int a = 1;
  Increment(a) = 5;
}
```

lvalue, rvalue

```
int a = 666;
```

lvalue rvalue

```
int *p = &a;
int 666 = a;
int *p = &666;
int a = b + c;
```

rvalue

rvalue

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

int main() {
  int a = 1;
  Increment(a) = 5;
}
```

lvalue, rvalue

```
int a = 666;
```

lvalue rvalue

```
int *p = &a;
int 666 = a;
int *p = &666;
int a = b + c;
```

rvalue

rvalue →

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

int main() {
  int a = 1;
  Increment(a) = 5;
}
```

lvalue Reference vs. Pointers

```
int a = 666;
int &r = a;
int *p = &a;
```

lvalue Reference vs. Pointers

```
int a = 666;
int &r = a;
int *p = &a;
```

a	0x1234	666

lvalue Reference vs. Pointers

```
int a = 666;
int &r = a;
int *p = &a;
```

a	0x1234	666
p	0x5678	0x1234

lvalue Reference vs. Pointers

```
int a = 666;
int &r = a;
int *p = &a;
```

r	a	0x1234	666
	p	0x5678	0x1234

lvalue Reference vs. Pointers

```
int a = 666;
int &r = a;
int *p = &a;

(*p)++;

r++;
```

r	a	0x1234	666
	p	0x5678	0x1234

lvalue Reference vs. Pointers

- A reference must be initialized, and cannot be re-assigned

```
int a = 666;
int b = 888;
int *p;
p = &a;
p = &b;
```

lvalue Reference vs. Pointers

- A reference must be initialized, and cannot be re-assigned

```
int a = 666;
int b = 888;
int *p;
p = &a;
p = &b;
```

```
int a = 666;
int b = 888;
int &r;
```

lvalue Reference vs. Pointers

- A reference must be initialized, and cannot be re-assigned

```
int a = 666;
int b = 888;
int *p;
p = &a;
p = &b;
```

```
int a = 666;
int b = 888;
int &r;
```

☒

lvalue Reference vs. Pointers

➜ A reference must be initialized, and cannot be re-assigned

```
int a = 666;
int b = 888;
int *p;
p = &a;
p = &b;
```

```
int a = 666;
int b = 888;
int &r;
int &r = a;
```

☒

☑

lvalue Reference vs. Pointers

➜ A reference must be initialized, and cannot be re-assigned

```
int a = 666;
int b = 888;
int *p;
p = &a;
p = &b;
```

```
int a = 666;
int b = 888;
int &r;
int &r = a;
int &r = nullptr;
```

lvalue Reference vs. Pointers

➜ A reference must be initialized, and cannot be re-assigned

```
int a = 666;
int b = 888;
int *p;
p = &a;
p = &b;
```

```
int a = 666;
int b = 888;
int &r;
int &r = a;
int &r = nullptr;
```

lvalue Reference vs. Pointers

➜ A reference must be initialized, and cannot be re-assigned

```
int a = 666;
int b = 888;
int *p;
p = &a;
p = &b;
```

```
int a = 666;
int b = 888;
int &r;                ☒
int &r = a;            ☑
int &r = nullptr;      ☒
&r = b;
```

lvalue Reference vs. Pointers

➜ A reference must be initialized, and cannot be re-assigned

```
int a = 666;
int b = 888;
int *p;
p = &a;
p = &b;
```

```
int a = 666;
int b = 888;
int &r;                ☒
int &r = a;            ☑
int &r = nullptr;      ☒
&r = b;                ☒
```

lvalue Reference vs. Pointers

- A reference is just an alias. Its own address and size are invisible

```
int a = 666;
int *p = &a;
int &r = a;
```

lvalue Reference vs. Pointers

- A reference is just an alias. Its own address and size are invisible

```
int a = 666;
int *p = &a;
int &r = a;
assert(&p != &a);
```

lvalue Reference vs. Pointers

- A reference is just an alias. Its own address and size are invisible

```
int a = 666;
int *p = &a;
int &r = a;
assert(&p != &a);
assert(&r == &a);
```

lvalue Reference vs. Pointers

- A reference is just an alias. Its own address and size are invisible

```
int a = 666;
int *p = &a;
int &r = a;
assert(&p != &a);
assert(&r == &a);
assert(sizeof(p) == sizeof(int *));
```

lvalue Reference vs. Pointers

- A reference is just an alias. Its own address and size are invisible

```
int a = 666;
int *p = &a;
int &r = a;
assert(&p != &a);
assert(&r == &a);
assert(sizeof(p) == sizeof(int *));
assert(sizeof(r) == sizeof(int));
```

Pass by Reference (C++ ONLY)

```
void swap(int *a, int *b) {
  int tmp = *a;
  *a = *b;
  *b = tmp;
}

int main() {
  int a = 1;
  int b = 2;
  swap(&a, &b);
}
```

```
void swap(int &a, int &b) {
  int tmp = a;
  a = b;
  b = tmp;
}

int main() {
  int a = 1;
  int b = 2;
  swap(a, b);
}
```

Pass by Reference (C++ ONLY)

```
void swap(int *a, int *b) {
  int tmp = *a;
  *a = *b;
  *b = tmp;
  a = b;
  a++;
}
```

```
void swap(int &a, int &b) {
  int tmp = a;
  a = b;
  b = tmp;
  &a = &b;
  (&a)++;
}
```

Pass by Reference (C++ ONLY)

```
void swap(int *a, int *b) {
  int tmp = *a;
  *a = *b;
  *b = tmp;
  a = b;
  a++;
}
```

```
void swap(int &a, int &b) {
  int tmp = a;
  a = b;
  b = tmp;
  &a = &b;
  (&a)++;
}
```

Copy Constructor

```
Date::Date(Date &date) {
  day_ = date.day_;
  month_ = date.month_;
  year_ = date.year_;
}
```

```
void func() {
  Date today(24, 10, 2023);
  Date tomorrow = today;
}
```

Copy constructor is invoked automatically

Objects and Functions

```
Date func(Date date) {
  date.AddMonth();
  return date;
}

int main() {
  Date today(24, 10, 2023);
  Date next_month = func(today);
}
```

today	next_month
?	?

Objects and Functions

pass by value

```
Date func(Date date) {
  date.AddMonth();
  return date;
}

int main() {
  Date today(24, 10, 2023);
  Date next_month = func(today);
}
```

today	next_month
?	?

Objects and Functions

pass by value

```
Date func(Date date) {
  date.AddMonth();
  return date;
}

int main() {
  Date today(24, 10, 2023);
  Date next_month = func(today);
}
```

today	next_month
24,10,2023	24,11,2023

Objects and Functions

pass by reference

```
Date func(Date &date) {
  date.AddMonth();
  return date;
}

int main() {
  Date today(24, 10, 2023);
  Date next_month = func(today);
}
```

today	next_month
?	?

Objects and Functions

pass by reference

```
Date func(Date &date) {
  date.AddMonth();
  return date;
}

int main() {
  Date today(24, 10, 2023);
  Date next_month = func(today);
}
```

today	next_month
24,11,2023	24,11,2023

lvalue Reference

```
int &r = 666;
```

lvalue Reference

```
int &r = 666;
```

lvalue Reference

```
int &r = 666;  ☒

void Func(int &x) {
   ...
}

int main() {
  Func(5);
}
```

lvalue Reference

```
int &r = 666;  ☒

void Func(int &x) {
   ...
}

int main() {
  Func(5);     ☒
}
```

lvalue Reference

```
void Func(BigObject x) {
  ...
}

int main() {
  Func(BigObject());
}
```

lvalue Reference

```
void Func(BigObject x) {
  ...
}

int main() {
  Func(BigObject());
}
```

← Extra memcpy when pass by value

lvalue Reference

```
void Func(BigObject &x) {
  ...
}

int main() {
  Func(BigObject());
}
```

lvalue Reference

```
void Func(BigObject &x) {
  ...
}

int main() {
  Func(BigObject());
}
```

rvalue

Const lvalue Reference

```
void Func(const BigObject &x) {
  ...
}

int main() {
  Func(BigObject());
}
```

x is immutable

Const lvalue Reference

```
void Func(const BigObject &x) {
  ...
}

int main() {
  Func(BigObject());
}
```

x is immutable

```
const BigObject &x = BigObject();
```

Const lvalue Reference

```
void Func(const BigObject &x) {
  ...
}

int main() {
  Func(BigObject());
}
```

x is immutable

```
const BigObject &x = BigObject();
```

⇩

```
BigObject internal_name = BigObject();
const BigObject &x = internal_name;
```

rvalue Reference (C++11)

```
int &r = 666;
```

```
const int &r = 666;
```

rvalue Reference (C++11)

```
int &r = 666;
```

```
const int &r = 666;
```

← But you can’t alter the value

rvalue Reference (C++11)

```
int &r = 666;
```

```
const int &r = 666;
```

But you can’t alter the value

```
int &&r = 666;
```

rvalue Reference (C++11)

```
int &r = 666;
```

```
const int &r = 666;
```

But you can’t alter the value

```
int &&r = 666;
```

rvalue Reference (C++11)

```
int &r = 666;
```

```
const int &r = 666;
```

But you can’t alter the value

```
int &&r = 666;
r += 888;
```

rvalue Reference (C++11)

```
int &r = 666;
```

```
const int &r = 666;
```

But you can't alter the value

```
int &&r = 666;
```

Super useful in "move semantics" (will discuss later)

```
r += 888;
```

Destructor

```
class Date {
 public:
  Date();
  Date(int d, int m, int y);
  int ToDaysInYear();
  void AddMonth(int m);

 private:
  int day_, month_, year_;
};
```

```
void func() {
  Date today(21, 3, 2022);
  today.AddMonth(3);
}
```

Destructor

```
class Date {
 public:
  Date();
  Date(int d, int m, int y);
  int ToDaysInYear();
  void AddMonth(int m);

 private:
  int day_, month_, year_;
};
```

```
void func() {
  Date today(21, 3, 2022);
  today.AddMonth(3);
}
```

`today` is reclaimed by the default destructor automatically

Destructor

```
class Date {
 public:
  Date();
  Date(int d, int m, int y);
  int ToDaysInYear();
  void AddMonth(int m);

 private:
  int day_, month_, year_;
  char *msg_;
};
```

```
void func() {
  Date today(21, 3, 2022);
  today.AddMonth(3);
}

Date::Date(int d, int m, int y) :
  day_(d), month_(m), year_(y) {
  msg_ = (char *)malloc(100);
}
```

Destructor

```
class Date {
 public:
  Date();
  Date(int d, int m, int y);
  ~Date();
  int ToDaysInYear();
  void AddMonth(int m);

 private:
  int day_, month_, year_;
  char *msg_;
};
```

```
Date::Date(int d, int m, int y) :
  day_(d), month_(m), year_(y) {
  msg_ = (char *)malloc(100);
}

Date::~Date() {
  if (msg_)
    free(msg_);
}
```

Dynamic Memory Allocation in C++

C Style

```
int *p = (int *)malloc(sizeof(int));
free(p);
```

Dynamic Memory Allocation in C++

C Style

```
int *p = (int *)malloc(sizeof(int));
free(p);
```

C++ Style

```
int *p = new int;
delete p;
```

Dynamic Memory Allocation in C++

C Style

```
int *p = (int *)malloc(sizeof(int));
free(p);

char *msg = (char *)malloc(100);
free(msg);
```

C++ Style

```
int *p = new int;
delete p;

char *msg = new char[100];
delete[] msg;
```

Dynamic Memory Allocation in C++

C Style

```c
int *p = (int *)malloc(sizeof(int));
free(p);

char *msg = (char *)malloc(100);
free(msg);
```

C++ Style

```cpp
int *p = new int;
delete p;

char *msg = new char[100];
delete[] msg;

Date *date = new Date(11,10,2022);
date->AddMonth(1);
delete date;
```

Destructor

```
class Date {
public:
  Date();
  Date(int d, int m, int y);
  ~Date();
  int ToDaysInYear();
  void AddMonth(int m);

private:
  int day_, month_, year_;
  char *msg_;
};
```

```
Date::Date(int d, int m, int y) :
  day_(d), month_(m), year_(y) {
  msg_ = (char *)malloc(100);
}

Date::~Date() {
  if (msg_)
    free(msg_);
}
```

Destructor

```cpp
class Date {
public:
  Date();
  Date(int d, int m, int y);
  ~Date();
  int ToDaysInYear();
  void AddMonth(int m);

private:
  int day_, month_, year_;
  char *msg_;
};
```

```cpp
Date::Date(int d, int m, int y) :
  day_(d), month_(m), year_(y) {
  msg_ = new char[100];
}

Date::~Date() {
  if (msg_)
    delete[] msg_;
}
```

Getter and Setter

```
class Date {
public:
  Date();
  Date(int d, int m, int y);
  void AddMonth(int m);

private:
  int day_, month_, year_;
};
```

Getter and Setter

```
class Date {
public:
  Date();
  Date(int d, int m, int y);
  void AddMonth(int m);

private:
  int day_, month_, year_;
};
```

- Default data members to private
- Provide getters & setters for the ones intended to be exposed

Getter and Setter

```
class Date {
public:
  Date();
  Date(int d, int m, int y);
  int GetMonth();
  bool SetMonth(int m);
  void AddMonth(int m);

private:
  int day_, month_, year_;
};
```

Getter and Setter

```cpp
class Date {
public:
  Date();
  Date(int d, int m, int y);
  int GetMonth();
  bool SetMonth(int m);
  void AddMonth(int m);

private:
  int day_, month_, year_;
};
```

```cpp
int Date::GetMonth() {
  return month_;
}

bool Date::SetMonth(int m) {
  if (m < 1 || m > 12)
    return false;
  month_ = m;
  return true;
}
```

Static Members

```
class Date {
  static int date_count = 0;
public:
  Date();
  Date(int d, int m, int y);
  int GetMonth();
  bool SetMonth(int m);
  static int GetDateCount();

private:
  int day_, month_, year_;
};
```

Static Members

```
class Date {
  static int date_count = 0;
public:
  Date();
  Date(int d, int m, int y);
  int GetMonth();
  bool SetMonth(int m);
  static int GetDateCount();

private:
  int day_, month_, year_;
};
```

- is part of the class, but is not a part of any object

Static Members

```
class Date {
  static int date_count = 0;
public:
  Date();
  Date(int d, int m, int y);
  int GetMonth();
  bool SetMonth(int m);
  static int GetDateCount();

private:
  int day_, month_, year_;
};
```

- is part of the class, but is not a part of any object
- Only one copy of the static member no matter how many objects of the class are created

Static Members

```
class Date {
  static int date_count = 0;
public:
  Date();
  Date(int d, int m, int y);
  int GetMonth();
  bool SetMonth(int m);
  static int GetDateCount();

private:
  int day_, month_, year_;
};
```

```
Date::Date(int d, int m, int y) :
  day_(d), month_(m), year_(y) {
  date_count++;
}

int main() {
  Date today(21, 3, 2022);
  Date tomorrow(22, 3, 2022);
  Date yesterday(20, 3, 2022);
  int num_dates = Date::GetDateCount();
}
```

Static Members

```
class Date {
  static int date_count = 0;
public:
  Date();
  Date(int d, int m, int y);
  int GetMonth();
  bool SetMonth(int m);
  static int GetDateCount();

private:
  int day_, month_, year_;
};
```

```
Date::Date(int d, int m, int y) :
  day_(d), month_(m), year_(y) {
  date_count++;
}

int main() {
  Date today(21, 3, 2022);
  Date tomorrow(22, 3, 2022);
  Date yesterday(20, 3, 2022);
  int num_dates = Date::GetDateCount();
}
```

Static Members

```
class Date {
  static int date_count = 0;
public:
  Date();
  Date(int d, int m, int y);
  int GetMonth();
  bool SetMonth(int m);
  static int GetDateCount();

private:
  int day_, month_, year_;
};
```

```
Date::Date(int d, int m, int y) :
  day_(d), month_(m), year_(y) {
  date_count++;
}

int main() {
  Date today(21, 3, 2022);
  Date tomorrow(22, 3, 2022);
  Date yesterday(20, 3, 2022);
  int num_dates = Date::GetDateCount();
}
```

3

Concepts Recap

Encapsulation & Abstraction

Class vs. Object

Data member

Function member

Access Control

Scope Operator

Static member

- Constructor
- Copy Constructor
- Destructor
- Getter & Setter

Concepts Recap

Encapsulation & Abstraction

Class vs. Object

Data member

Function member

Access Control

Scope Operator

Static member

Constructor

Copy Constructor

Destructor

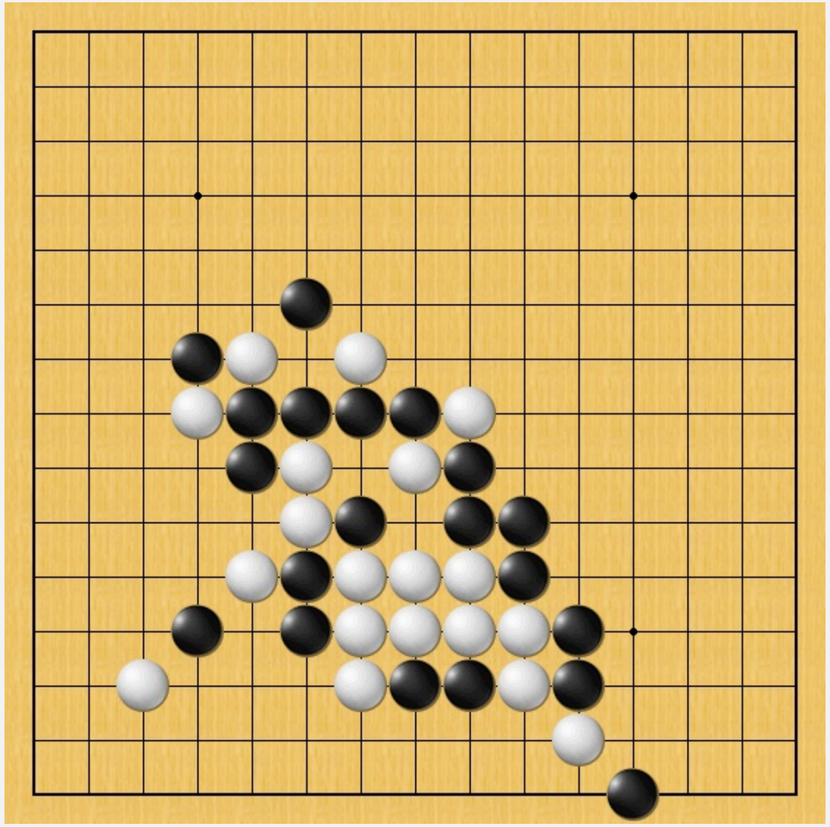
Getter & Setter

Other C++ Features

Function Overloading

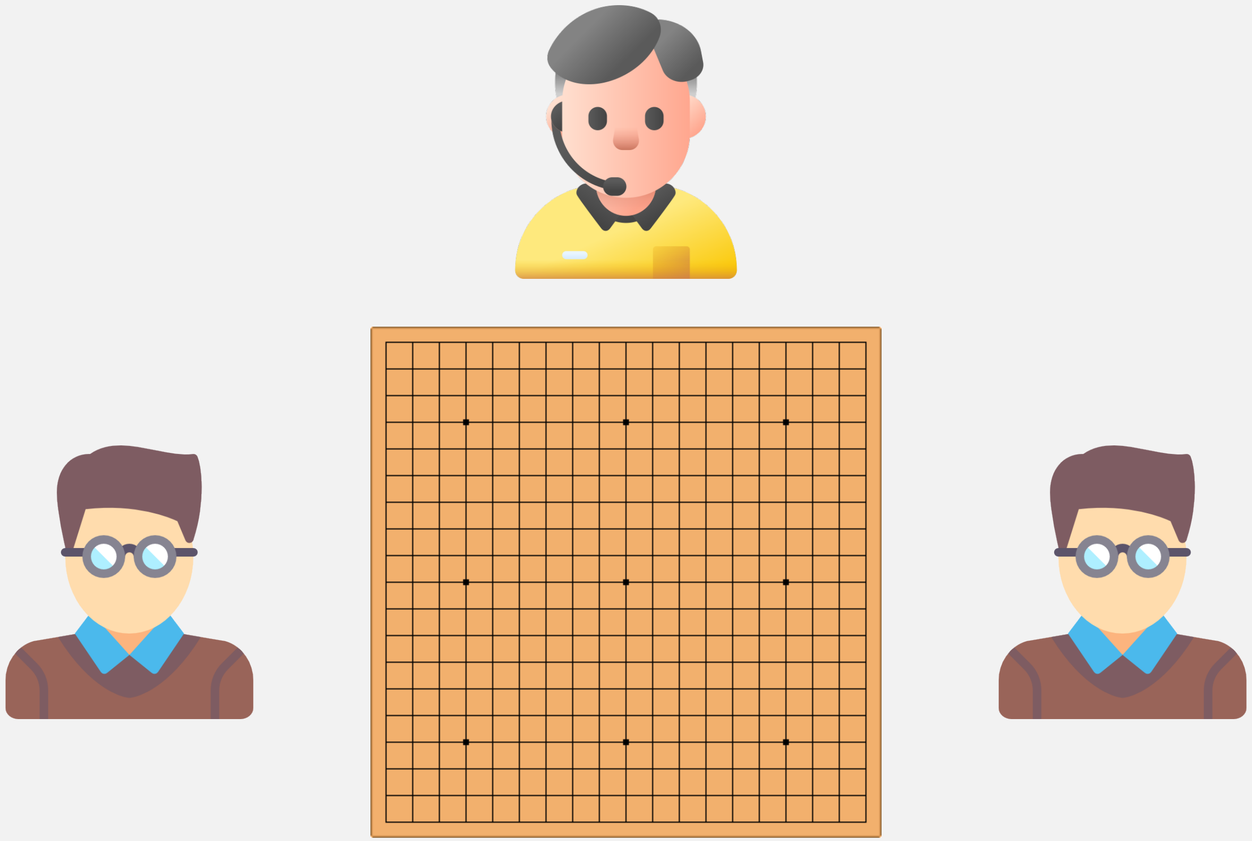
C++ reference type

`new, delete`

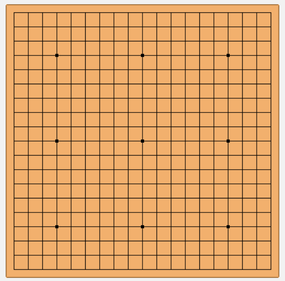


五子棋

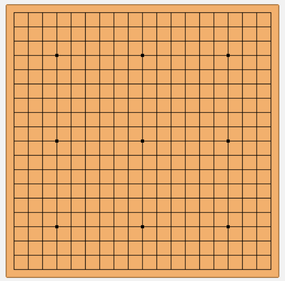
Roles in Gobang



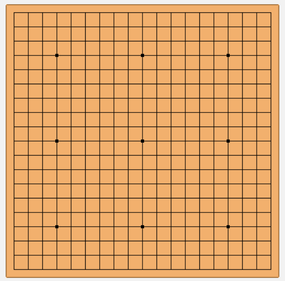
Roles in Gobang

	Board	Player	Game
			
Properties			
Behaviors			

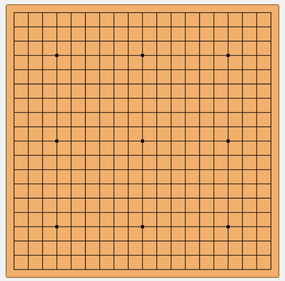
Roles in Gobang

	Board	Player	Game
			
Properties	size intersection states		
Behaviors			

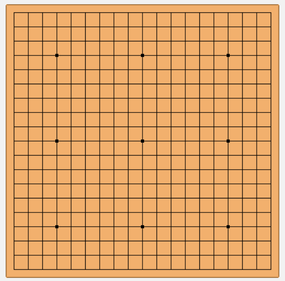
Roles in Gobang

	Board	Player	Game
			
Properties	size intersection states		
Behaviors	draw get/set i-state		

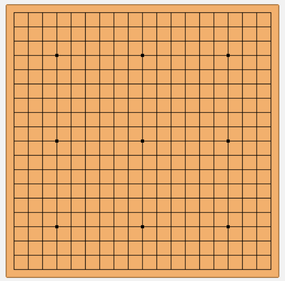
Roles in Gobang

	Board	Player	Game
			
Properties	size intersection states	name, role winning state	
Behaviors	draw get/set i-state		

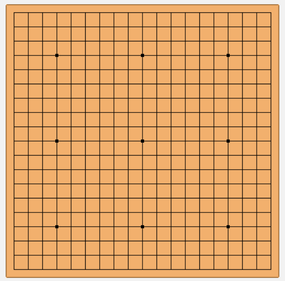
Roles in Gobang

	Board	Player	Game
			
Properties	size intersection states	name, role winning state	
Behaviors	draw get/set i-state	play a move get/set w-state	

Roles in Gobang

	Board	Player	Game
			
Properties	size intersection states	name, role winning state	1 board 2 players
Behaviors	draw get/set i-state	play a move get/set w-state	

Roles in Gobang

	Board	Player	Game
			
Properties	size intersection states	name, role winning state	1 board 2 players
Behaviors	draw get/set i-state	play a move get/set w-state	start game judge win/lose

Demo