Class

Object-Oriented Programming in C++
Weng Kai

reference

Declaring references

- Reference is a new way to manipulate objects in C++
  - `char c;        // a character`
  - `char* p = &c; // a pointer to a character`
  - `char& r = c;  // a reference to a character`
- Local or global variables
  - *type*& *refname* = name;
  - For ordinary variables, the initial value is required
- In parameter lists and member variables
  - *type*& *refname*
  - Binding defined by caller or constructor

References

- Declares a new *name* for an *existing* object

```
int  X = 47;

int& Y = X; // Y is a reference to X



// X and Y now refer to the same variable

cout << "Y = " << y;  // prints Y = 47

Y = 18;

cout << "X = " << x;  // prints X = 18
```

Rules of references

- References must be initialized when defined
- Initialization establishes a binding
  - In declaration

    ```
    int x = 3;
    int& y = x;
    const int& z = x;
    ```

  - As a function argument

    ```
    void f ( int& x );
    f(y);     // initialized when function is called
    ```

1. const字面值常量只能用来初始化const引用
2. const引用可以绑定到不同类型或初始化为右值（const int &b = 1.5;）；而非const引用只能绑定与该引用同类型的对象
3. const引用可以读取但不可以被修改引用对象

Rules of references

- Bindings don't change at run time, unlike pointers
- Assignment changes the object referred-to

```
int& y = x;

y = 12; // Changes value of x
```

- The target of a reference must have a location!

```
void func(int &);

func (i * 3);       // Warning or error!
```

Pointers vs. References

- References
  - can't be null
  - are dependent on an existing variable, they are an alias for an variable
  - can't change to a new "address" location

- Pointers
  - can be set to null
  - pointer is independent of existing objects
  - can change to point to a different address

Restrictions

- No references to references
- No pointers to references

```
int&* p;            // illegal
```

  - Reference to pointer is ok

```
void f(int*& p);
```

- No arrays of references

Point

```
typedef struct point {
    float x;
    float 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 {
     float x;
     float 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) const;
    void print() const;

private:
    int x;
    int y;
} ;
```

此处的x与y均为声明，而非定义（类内的成员变量均为声明）

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 {
    float x;
    float 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);
```

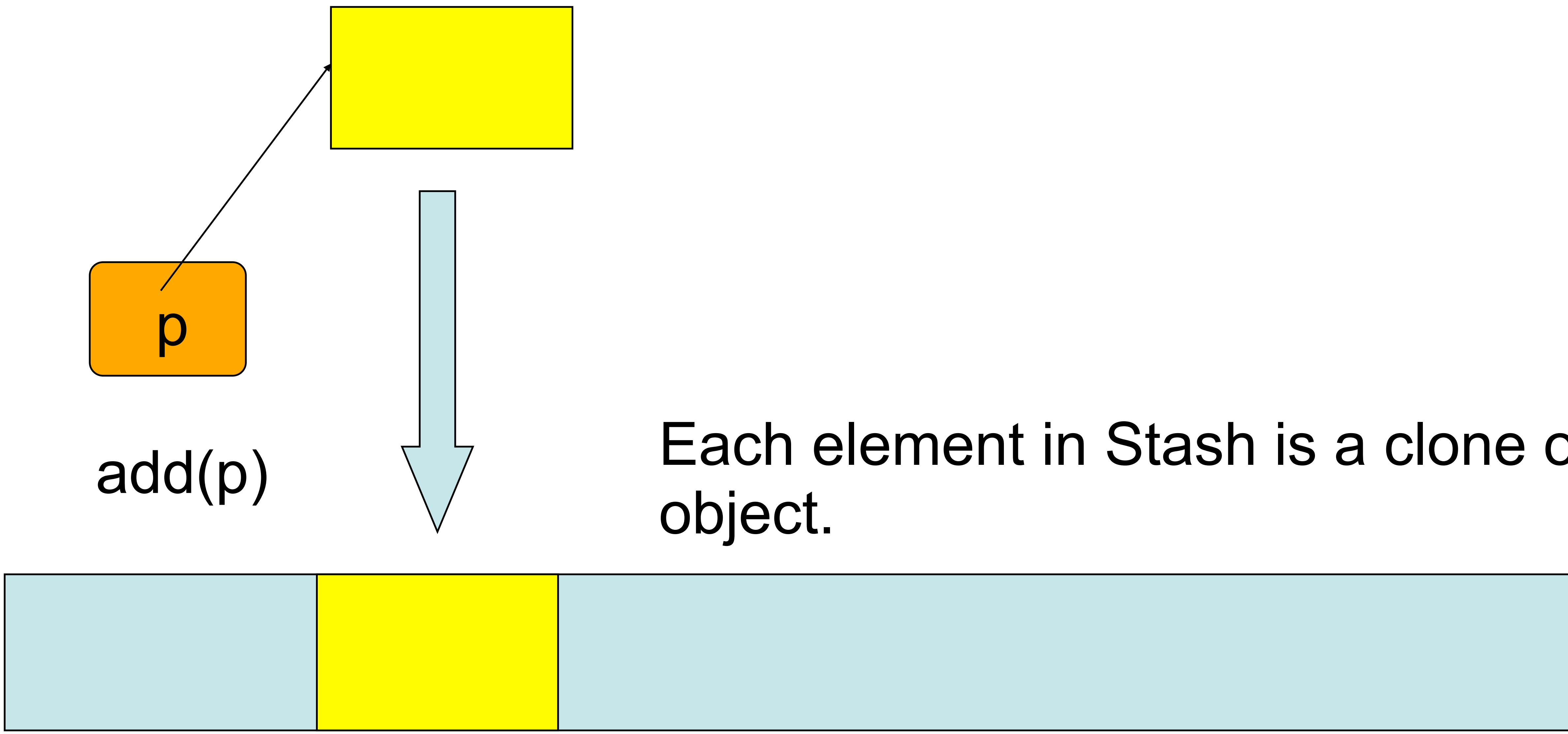
```cpp
class Point {
public:
    void init(int x,int y);
    void print() const;
    void move(int dx,int dy);

private:
    int x;
    int y;
} ;

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

Stash

Container

- Container is an object that holds other objects.
- For most kinds of containers, the common interface is put() and get().
- Stash is a container that stores objects and can be expanded during running.

Stash



Each element in Stash is a clone of the object.

Stash

- Typeless container.
- Stores objects of the same type.
  - Initialized w/ the size of the type
  - Doesn't care the type but the size
- add() and fetch()
- Expanded when needed

- See: CStash.h, CStash.cpp, CStashTest.cpp

Functions in struct

```
struct Stash {
   int size; // Size of each space
   int quantity; // Number of storage spaces
   int next; // Next empty space
   // Dynamically allocated array of bytes:
   unsigned char* storage;
   // Functions!
   void initialize(int size);
   void cleanup();
   int add(const void* element);
   void* fetch(int index);
   int count();
   void inflate(int increase);
};
```

See: Stash.h

Implementation of the functions

- We just defined in the header file that there will be these functions in this struct.
- All the bodies of these functions will be in a source file.

See: Stash.cpp

Call the functions in a struct

```
Stash a;
a.initialize(10);
```

- There is a relationship with the function be called and the variable to call it.
- The function itself knows it is doing something w/ the variable.
- Example: [StashTest.cpp]

this: the hidden parameter

- **this** is a hidden parameter for all member functions, with the type of the struct
  `void Stash::initialize(int sz)`
  ➔ (can be regarded as)
  `void Stash::initialize(Stash*this, int sz)`
- To call the function, you must specify a variable
  ```
  Stash a;
  a.initialize(10);
  ```
  ➔ (can be regarded as)
  `Stash::initialize(&a,10);`
- Example: this.cpp

this: the pointer to the variable

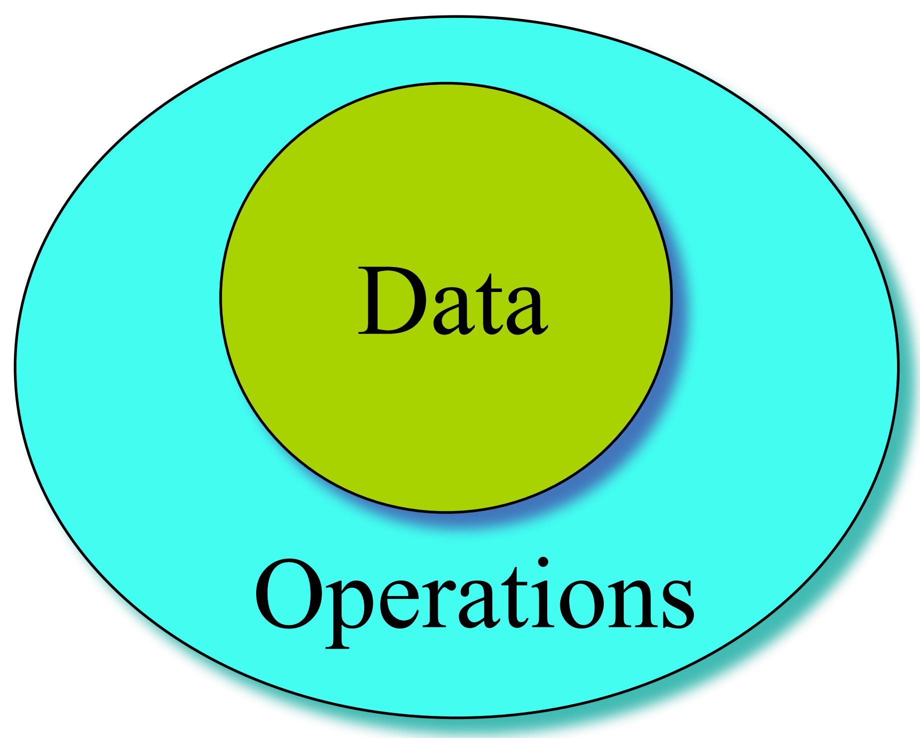
- Inside member functions, you can use **this** as the pointer to the variable that calls the function.

- **this** is a natural local variable of all structs member functions that you can not define, but can use it directly.

- Example: Integer.h, Integer.cpp

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”.

- The struct variables mentioned before are just objects in C++.

Ticket Machine

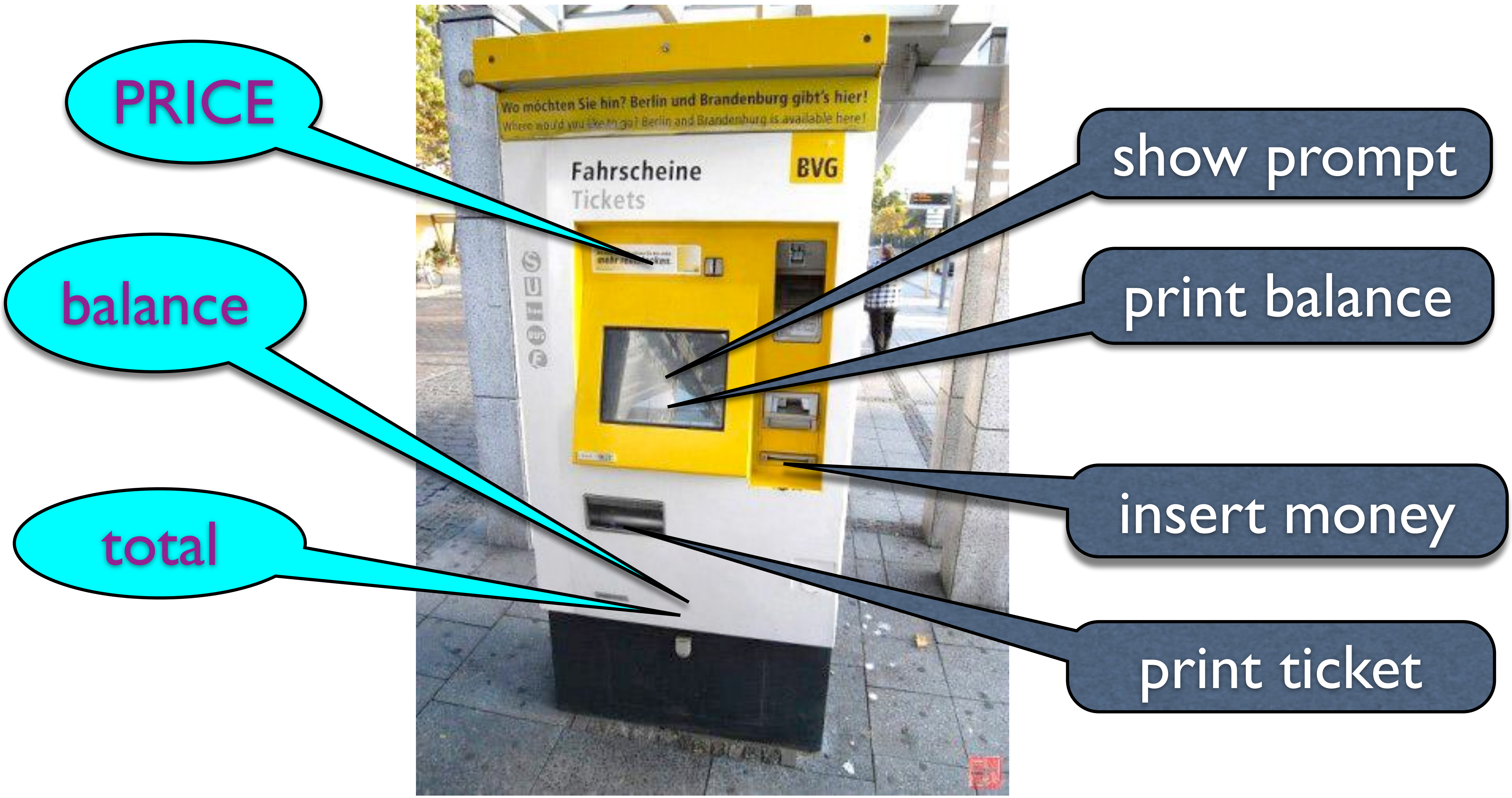
- Ticket machines print a ticket when a customer inserts the correct money for their fare.
- Our ticket machines work by customers ‘inserting’ money into them, and then requesting a ticket to be printed. A machine keeps a running total of the amount of money it has collected throughout its operation.

Procedure-Oriented

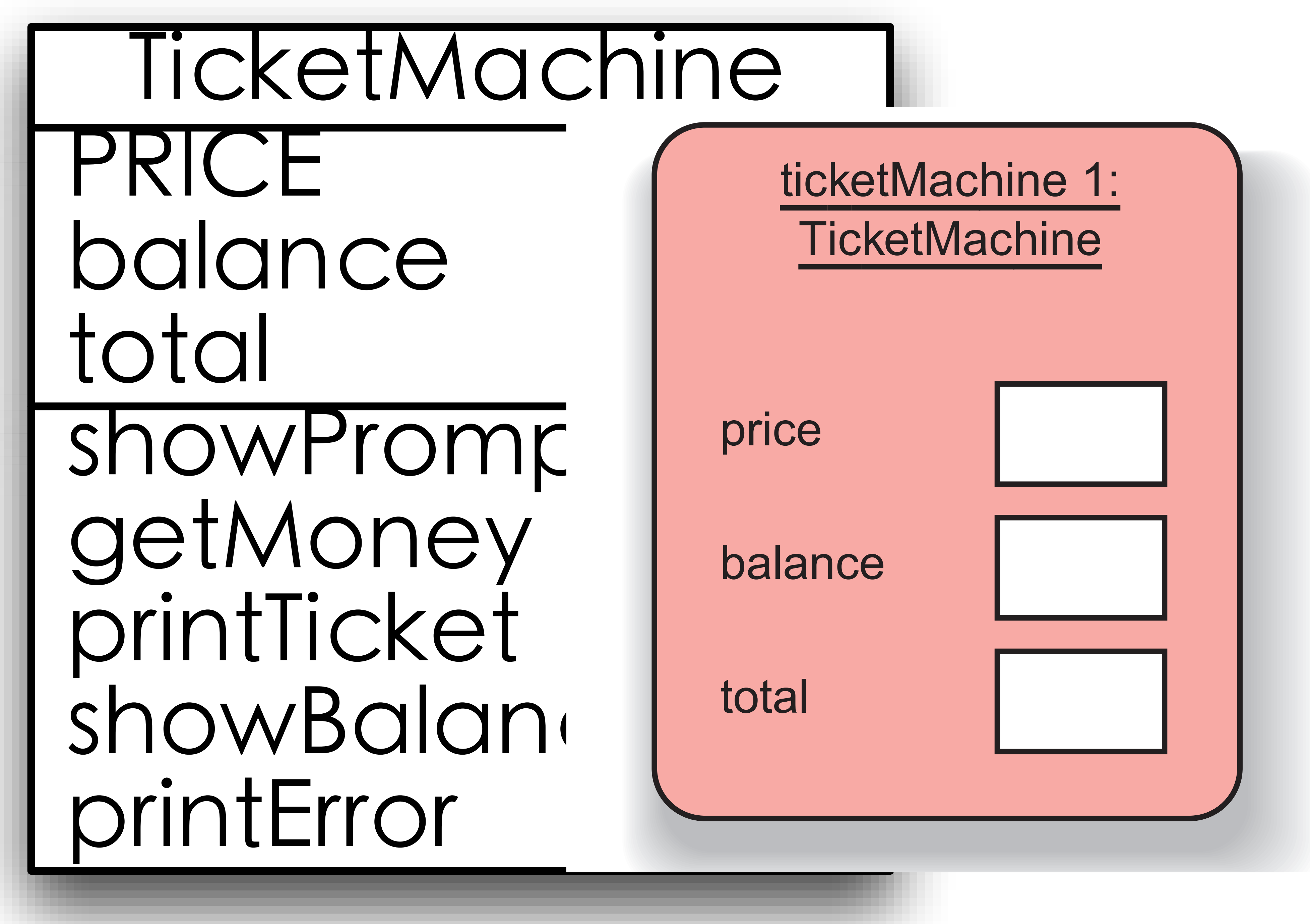
- Step to the machine
- Insert money into the machine
- The machine prints a ticket
- Take the ticket and leave

We make a program simulates the procedure of buying tickets. It works. But there is no such machine. There’s nothing left for the further development.

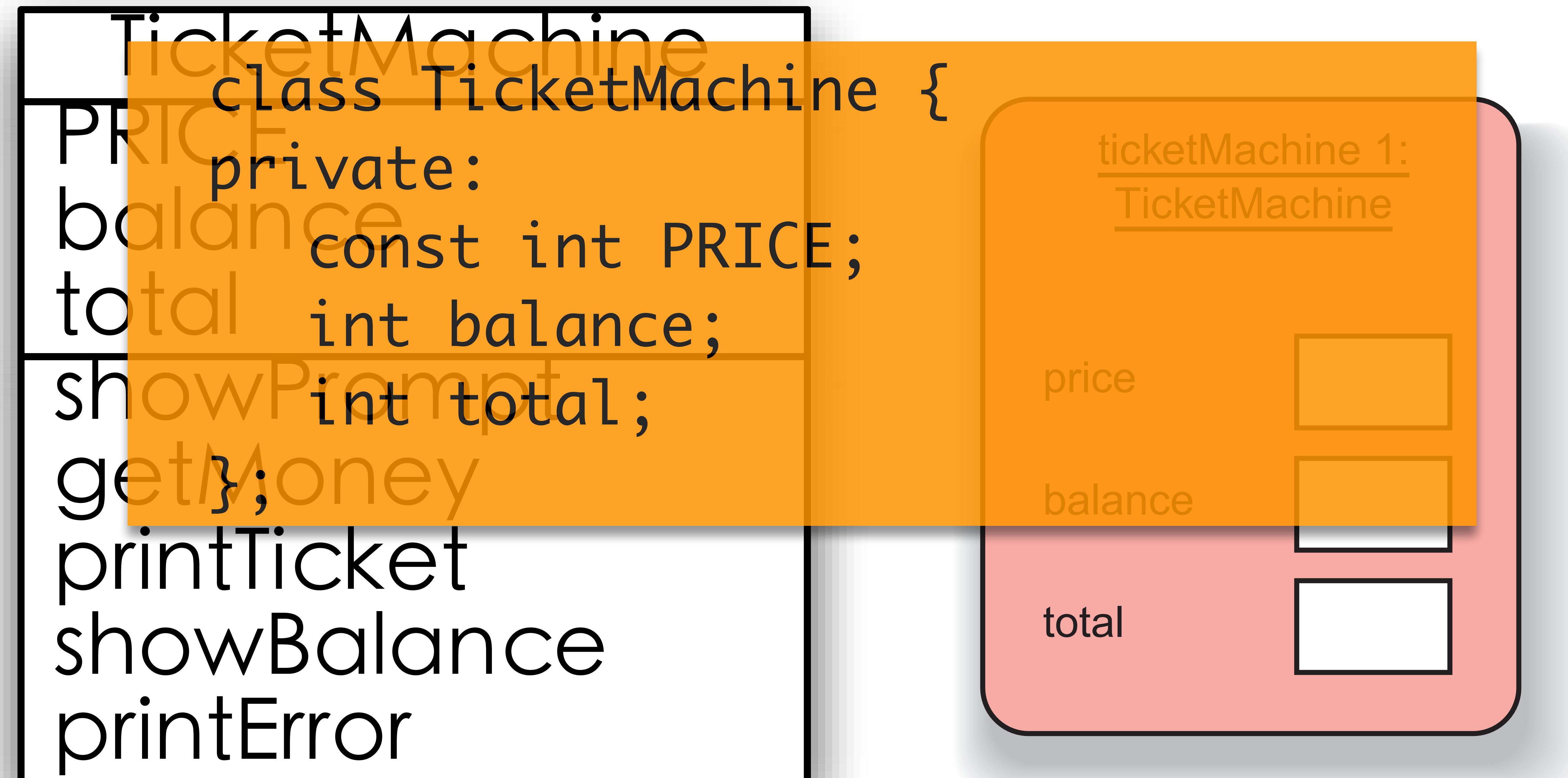
Something is there



Something is here



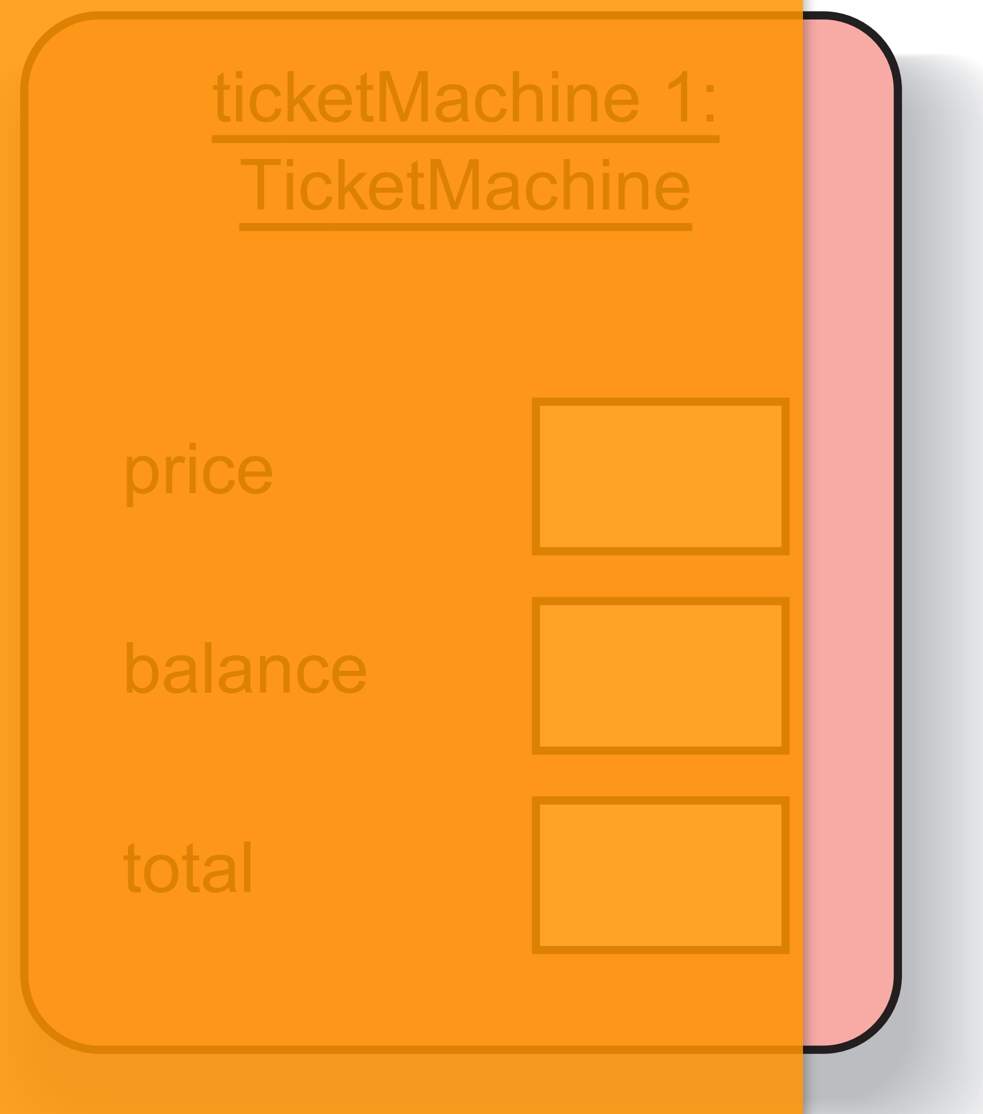
Turn it into code




```
class TicketMachine {
private:
    const int PRICE;
    int balance;
    int total;
};
```

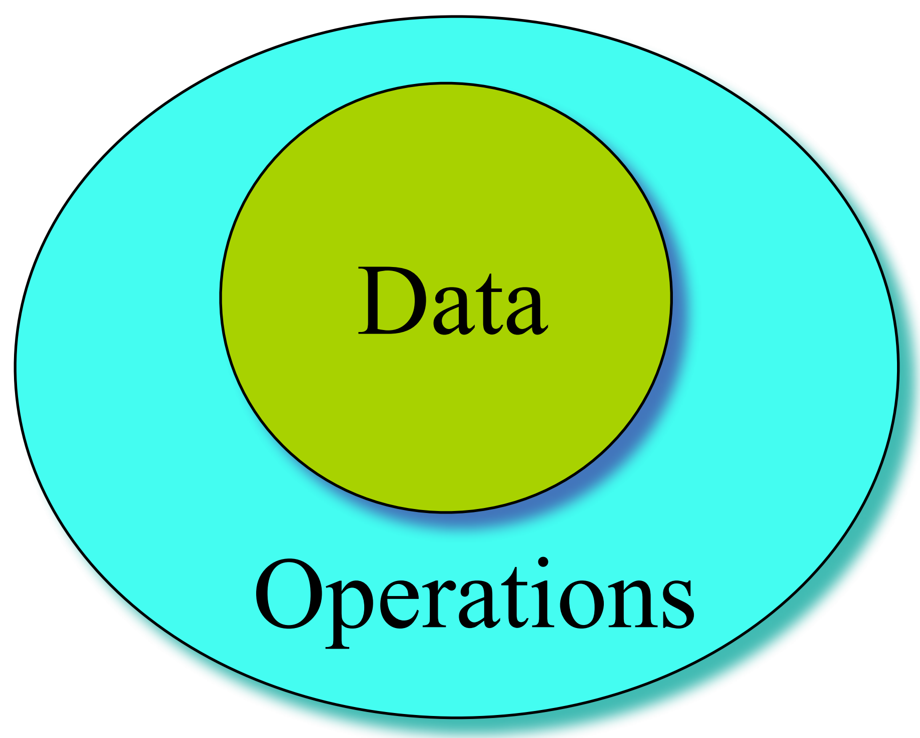
Turn it into code

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



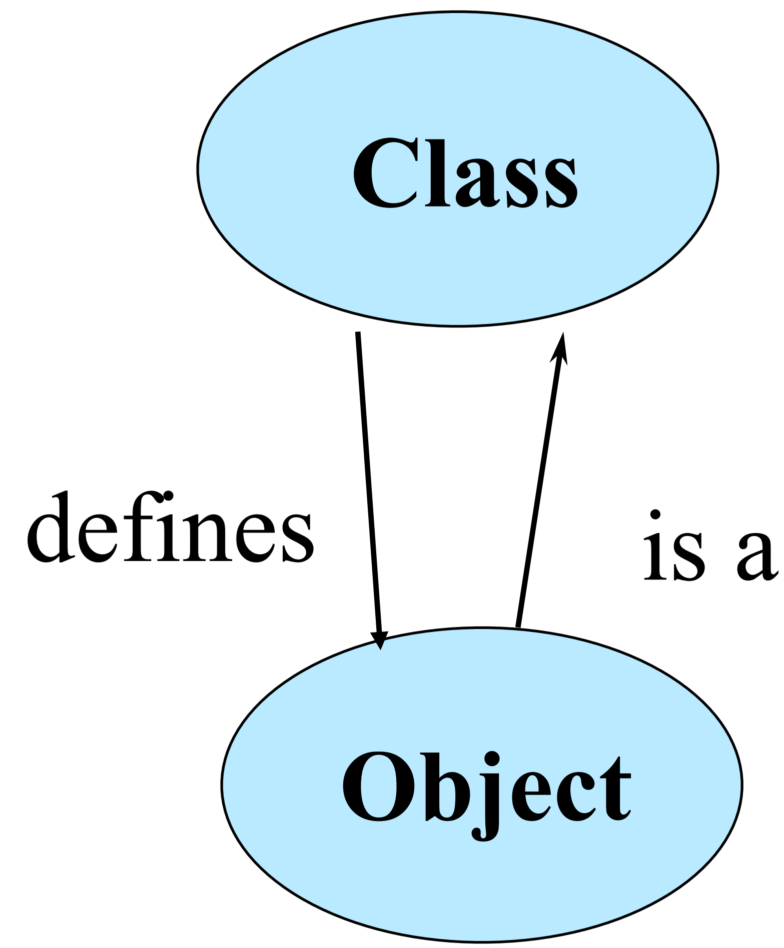
Objects = Attributes + Services

- 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

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

C’tor and D’tor

Point::init()

```
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();
```

Guaranteed initialization with the constructor

- If a class has a constructor, the compiler automatically calls that constructor at the point an object is created, before client programmers can get their hands on the object.
- The name of the constructor is the same as the name of the class.

How a constructor does?

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

constructor

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

a.X();

Constructors with arguments

- The constructor can have arguments to allow you to specify how an object is created, give it initialization values, and so on.

```
Tree(int i) {…}
```

```
Tree t(12);
```

- Constructor1.cpp

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

- If you have a constructor, the compiler ensures that construction *always* happens.
- *If* (and only if) there are no constructors for a class (**struct** or **class**), the compiler will automatically create one for you.
  - Example:AutoDefaultConstructor.cpp

The destructor

- In C++, cleanup is as important as initialization and is therefore guaranteed with the destructor.
- The destructor is named after the name of the class with a leading tilde (~). The destructor never has any arguments.

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

When is a destructor called?

- The destructor is called automatically by the compiler when the object goes out of scope.
- The only evidence for a destructor call is the closing brace of the scope that surrounds the object.

Storage allocation

- The compiler allocates all the storage for a scope at the opening brace of that scope.
- The constructor call doesn't happen until the sequence point where the object is defined.
  - Examlpe: Nojump.cpp

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) };`