Interaction & Construction

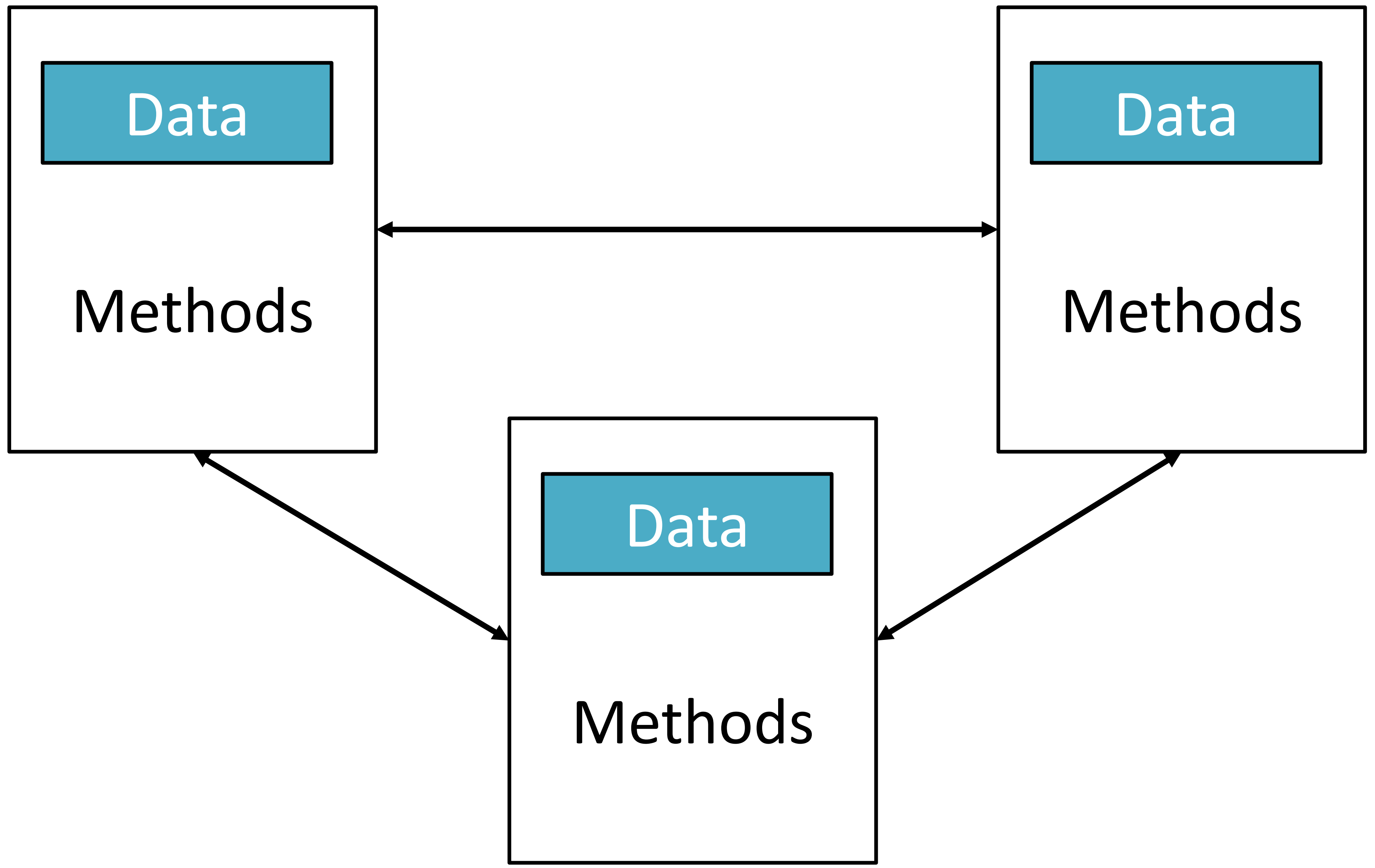
Object-Oriented Programming in C++

Zhaopeng Cui

Object Interaction

Object oriented programming

- Objects send and receive messages (object do things!)



Object send messages

- 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 dealing with these data together in an object
- Hide the details of the data and the action
- Restrict access only to the publicized methods

Abstraction

- Abstraction is the ability to ignore details of parts to focus attention on a higher level of a problem.

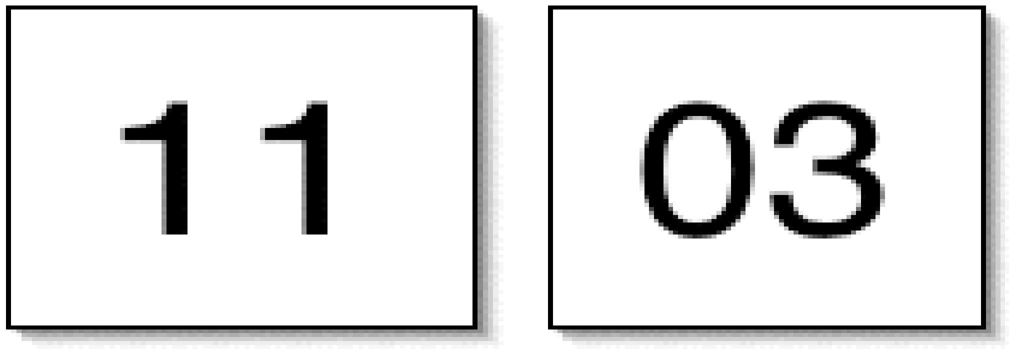
- Modularization is the process of dividing a whole into well-defined parts, which can be built and examined separately, and which interact in well-defined ways.

Clock display

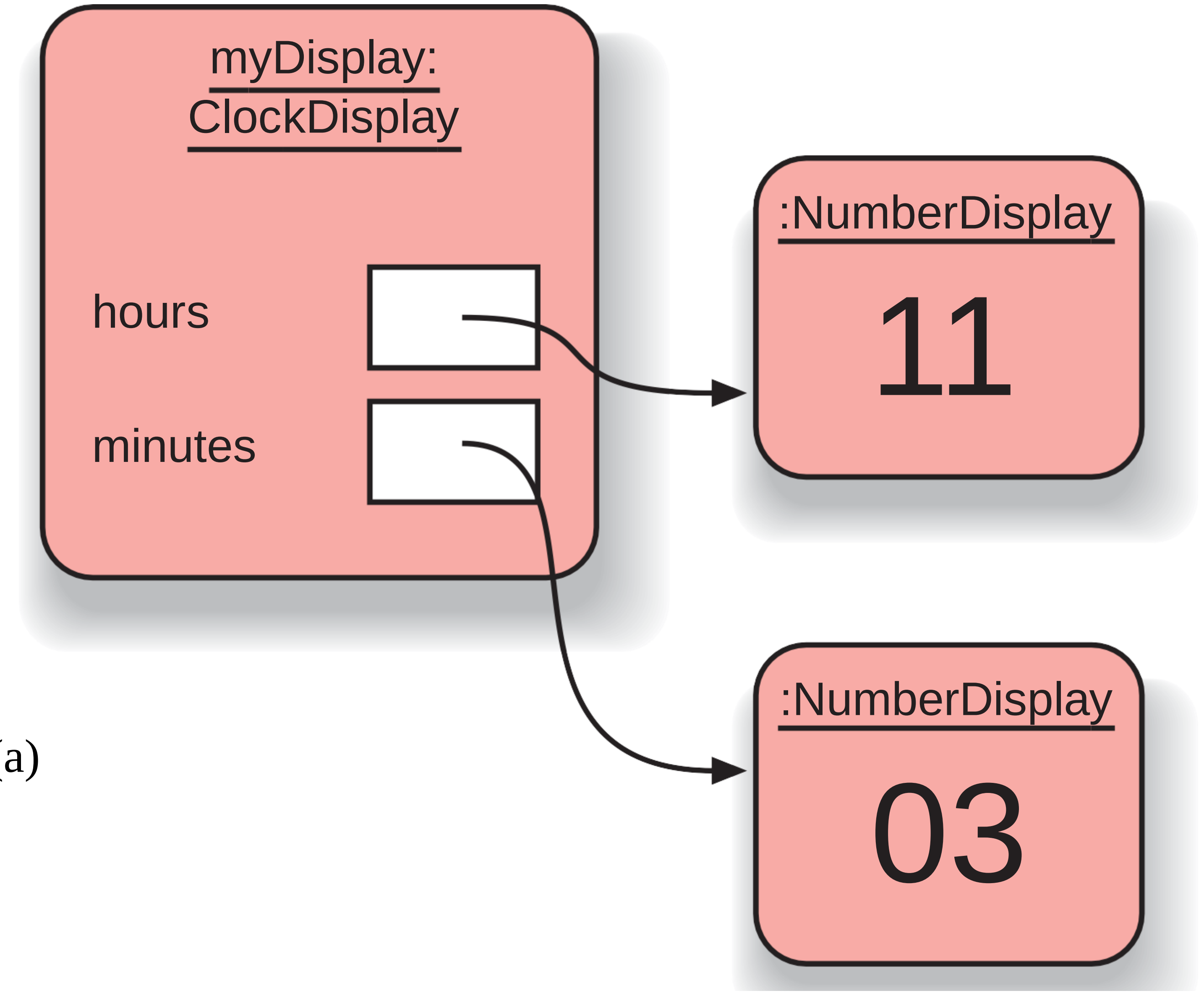
Modularizing the clock display

One *4-digits* display?

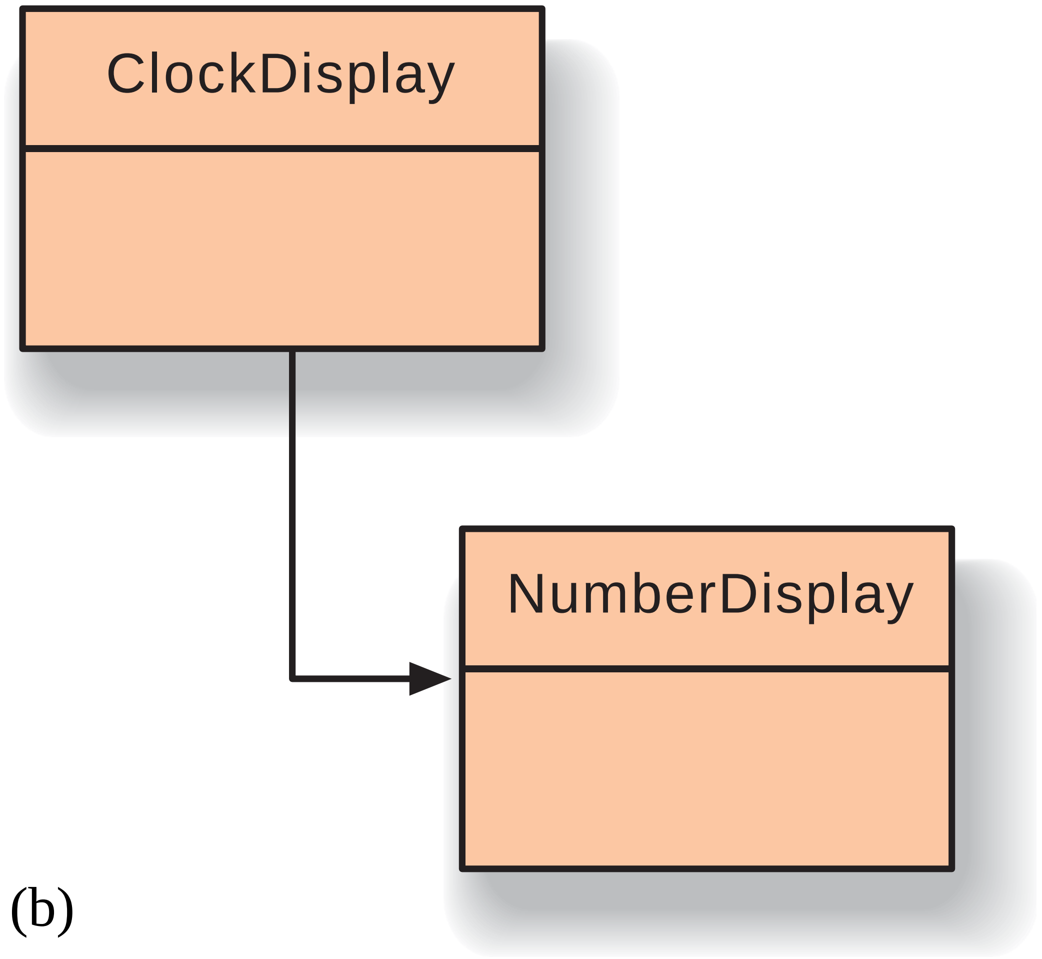
Or two *2-digits* displays?



Object & Classes

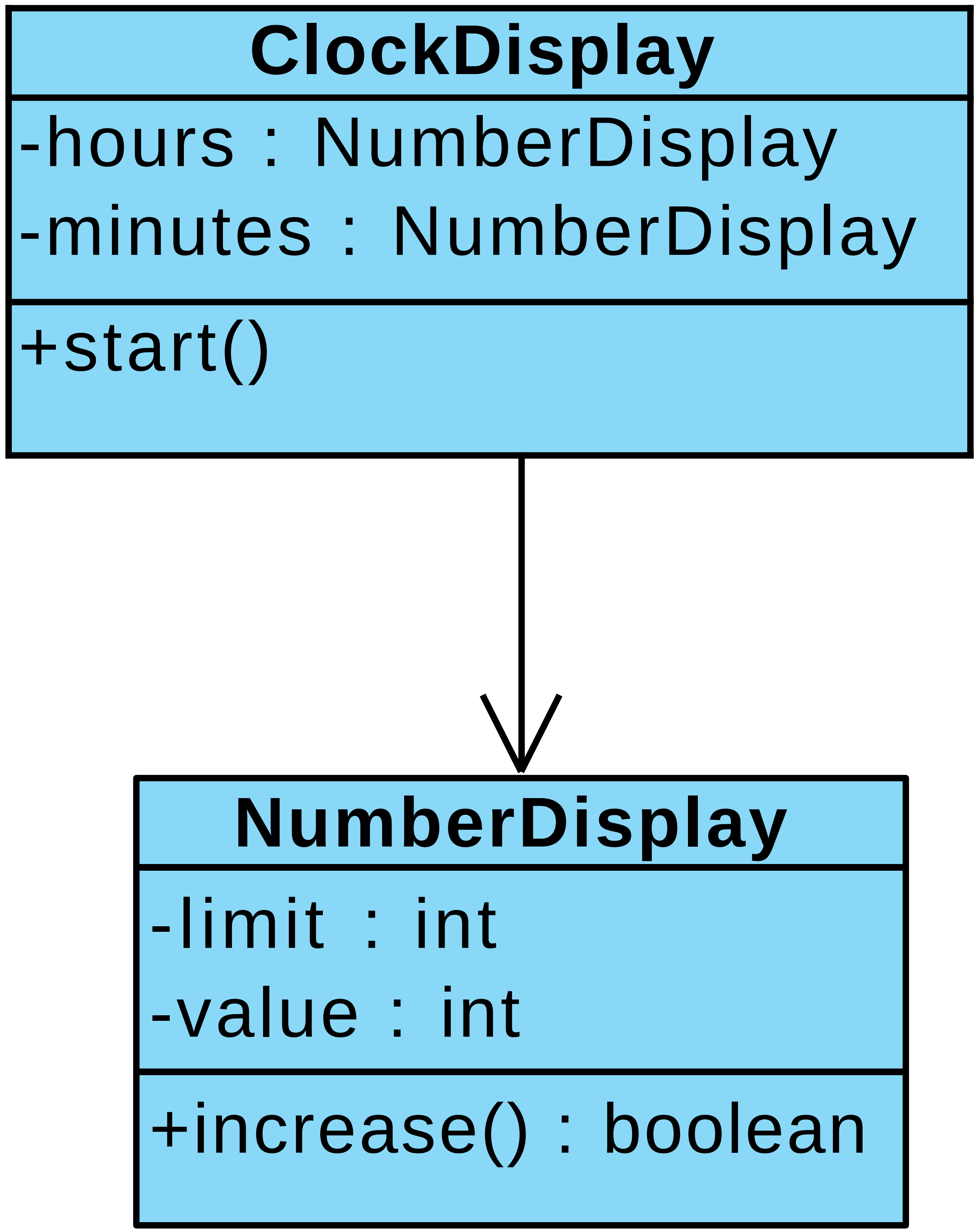


(a)



(b)

Class diagram



Implementation - ClockDisplay

```
class ClockDisplay {
    NumberDisplay hours;
    NumberDisplay minutes;

   //Constructor and methods omitted.
}
```

Implementation - ClockDisplay

```
class NumberDisplay {
    int limit;
    int value;

    //Constructor and methods omitted.

}
```

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();
```

功成到码
年年有余屯屯屯
蒸蒸日上烫烫烫
APPY NEW YE
火灾时
电梯

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 construcor 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);
```

The default constructor

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

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.

- See Constructor1.cpp

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.
- Example: Nojump.cpp

Aggregate initialization

```cpp
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(2) };
```

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];
```

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

Initialization

Member Init

- Directly initialize a member
- benefit: for all ctors
- C++ 11 works

Initializer list

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

- Can initialize any type of data
  - pseudo-constructor calls for built-ins
  - No need to perform assignment within body of ctor
- –Order of initialization is order of ***declaration***
  - Not the order in the list!
  - Destroyed in the reverse order.

Initialization vs. assignment

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

initialization

before constructor

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

assignment

inside constructor

string must have a default constructor

Local variable

- Local variables are defined inside a method, have a scope limited to the method to which they belong.

Local variable

```
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

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

- But how is the access to **balance** achieved?

Field

- Fields (or member variables) are defined outside constructors and methods.

- Fields are used to store data that persists throughout the life of an object. As such, they maintain the current state of an object. They have a lifetime that lasts as long as their object lasts.

- Fields have class scope: their accessibility extends throughout the whole class, and so they can be used within any of the constructors or methods of the class in which they are defined.

Local variable

```
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 vs. parameters vs. local variables

- All three kinds of variable are able to store a value that is appropriate to their defined type.
- As long as they are defined as private, fields cannot be accessed from anywhere outside their defining class.
- Formal parameters and local variables persist only for the period that a constructor or method executes. Their lifetime is only as long as a single call, so their values are lost between calls. As such, they act as temporary rather than permanent storage locations.
- Formal parameters are defined in the header of a constructor or method. They receive their values from outside, being initialized by the actual parameter values that form part of the constructor or method call.

Fields vs. parameters vs. local variables

- Formal parameters have a scope that is limited to their defining constructor or method.
- Local variables are defined inside the body of a constructor or method. They can be initialized and used only within the body of their defining constructor or method.
- Local variables must be initialized before they are used in an expression – they are not given a default value.
- Local variables have a scope that is limited to the block in which they are defined. They are not accessible from anywhere outside that block.

this: the hidden parameter

- **this** is a hidden parameter for all member functions, with the type of the struct

```
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 local variable of all class member functions that you cannot define, but can use it directly.

const object

Constant objects

- What if an object is const?

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

- What members can access the internals?

- How can the object be protected from change?

Constant objects

- What if an object is const?

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

- What members can access the internals?
- How can the object be protected from change?
- Solution: declare member functions const
  - Programmer declares member functions to be safe

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 their 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; // OK
}
```

Const member functions 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

```cpp
// 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 initializer list of the constructor

Compile-time constants in classes

```
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;
```

- Or use “anonymous enum” hack:

```
Class HasArray{
    enum { size = 100 };
    int array[size];  // OK!
    …
}
```

Inline function

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 Functions

```
inline  int plusOne(int x) { return ++x; };
```

- The "definition" of an inline function should be put in a header file.
- An inline function definition may not generate any code in .obj file.
- It is declaration rather than definition.

Inline functions in header file

- So you can put inline functions' bodies in header file. Then **#include** it where the function is needed.

- Never be afraid of multi-definition of inline functions.

- Definitions of inline functions are just declarations.

Tradeoff of inline functions

- Body of the called function is to be inserted into the caller.
- This may expand the code size, but deduces the overhead of calling time.
- So it gains speed at the expenses of space.
- It is much better than macro in C. It checks the types of the parameters, and has no dangerous side effect.

Inline vs. macro

inline

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

int f();

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

macro

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

int f();

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

Inline inside classes

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

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

Access functions

- They are small functions that allow you to read or change part of the state of an object – that is, an internal variable or variables.

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

Reducing clutter

- Member functions defined within classes use the Latin in situ (in place) and maintains that all definitions should be placed outside the class to keep the interface clean.
- Example: Noinsitu.cpp

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 (recursion is not allowed or indeed possible for inline functions), 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".

Type of function parameters and return value

way in

- void f(Student i);
  - a new object is to be created in f
- void f(Student *p);
  - better with const if no intend to modify the object
- void f(Student& i);
  - better with const if no intend to modify the object

way out

- Student f();
  - a new object is to be created at returning
- Student* f();
  - what should it points to?
- Student& f();
  - what should it refers to?

hard decision

```
char *foo()
{
    char *p;
    p = new char[10];
    strcpy(p, "something");
    return  p;
}

void bar()
{
    char *p = foo();
    printf("%s", p);
    delete  p;
}
```

define a pair functions of alloc and free

Better to ask user to take resp., pass pointers in & out

tips

- Pass in an object if you want to store it
- Pass in a const pointer or reference if you want to get the values
- Pass in a pointer or reference if you want to do something to it
- Pass out an object if you create it in the function
- Pass out pointer or reference of the passed in only
- Never new something and return the pointer

Basic operations for vector

- Constructors

```
vector<Elem> c;
vector<Elem> c1(c2);
```

- Simple methods

```
V.size( )        // num items
V.empty( )       // empty?
==, !=, <, >, <=, >=
v1.swap(v2)      // swap
```

- Iterators

```
I.begin( )       // first position
I.end( )         // last position
```

- Element access

```
V.at(index)
V[index]
V.front( )       // first item
V.back( )        // last item
```

- Add/Remove/Find

```
V.push_back(e)
V.pop_back( )
V.insert(pos, e)
V.erase(pos)
V.clear( )
```

~~V.find(first, last, item)~~

`find(first, last, item)`