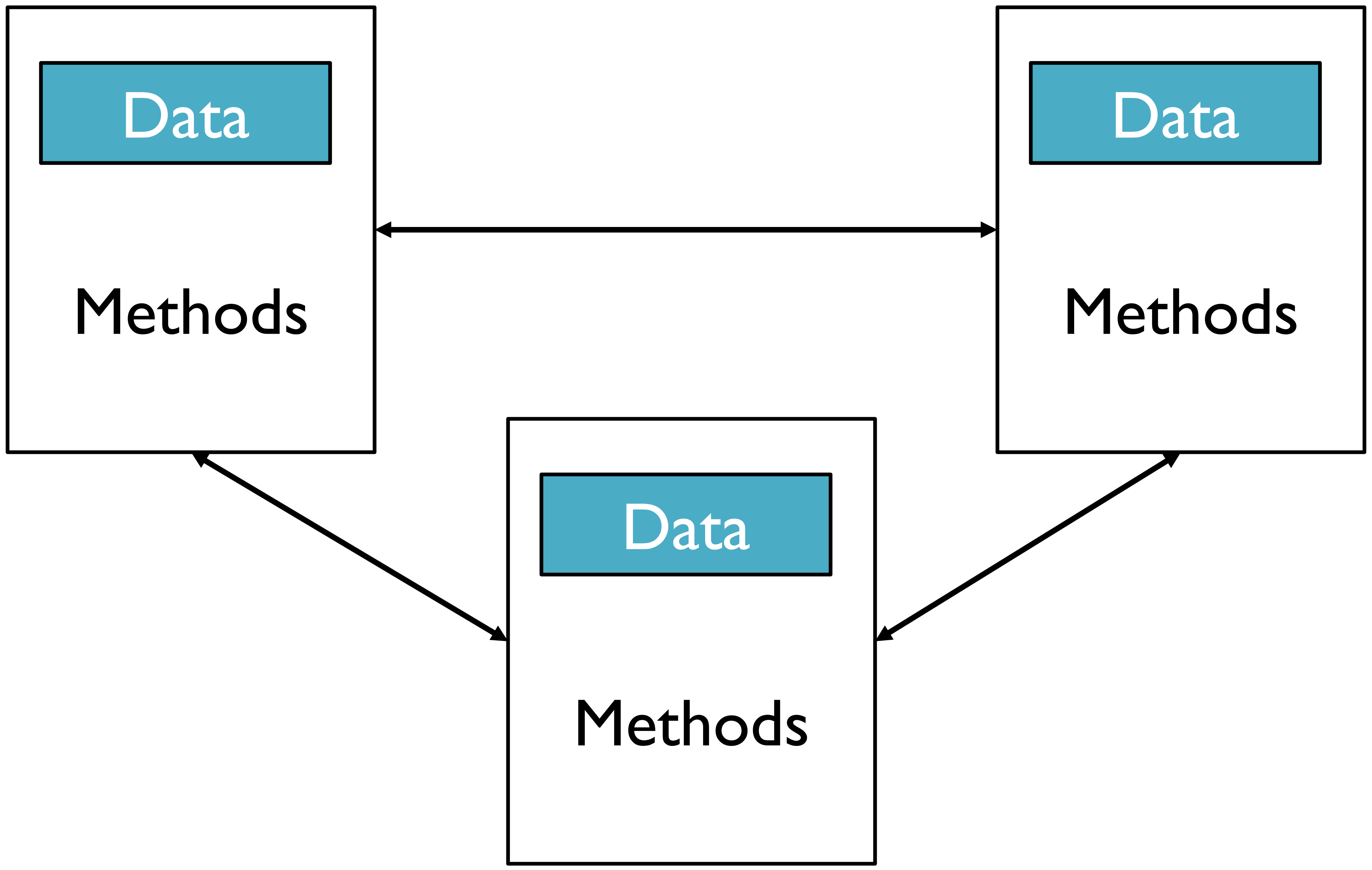
Object Interaction

Object-Oriented Programming with C++

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

Clock display

Abstract

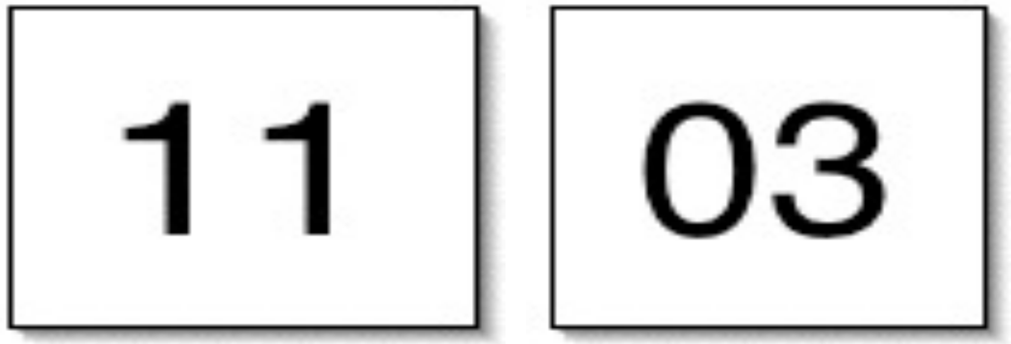
- Abstraction is the ability to ignore details of parts to focus attention on a higher level of a problem.

- 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.

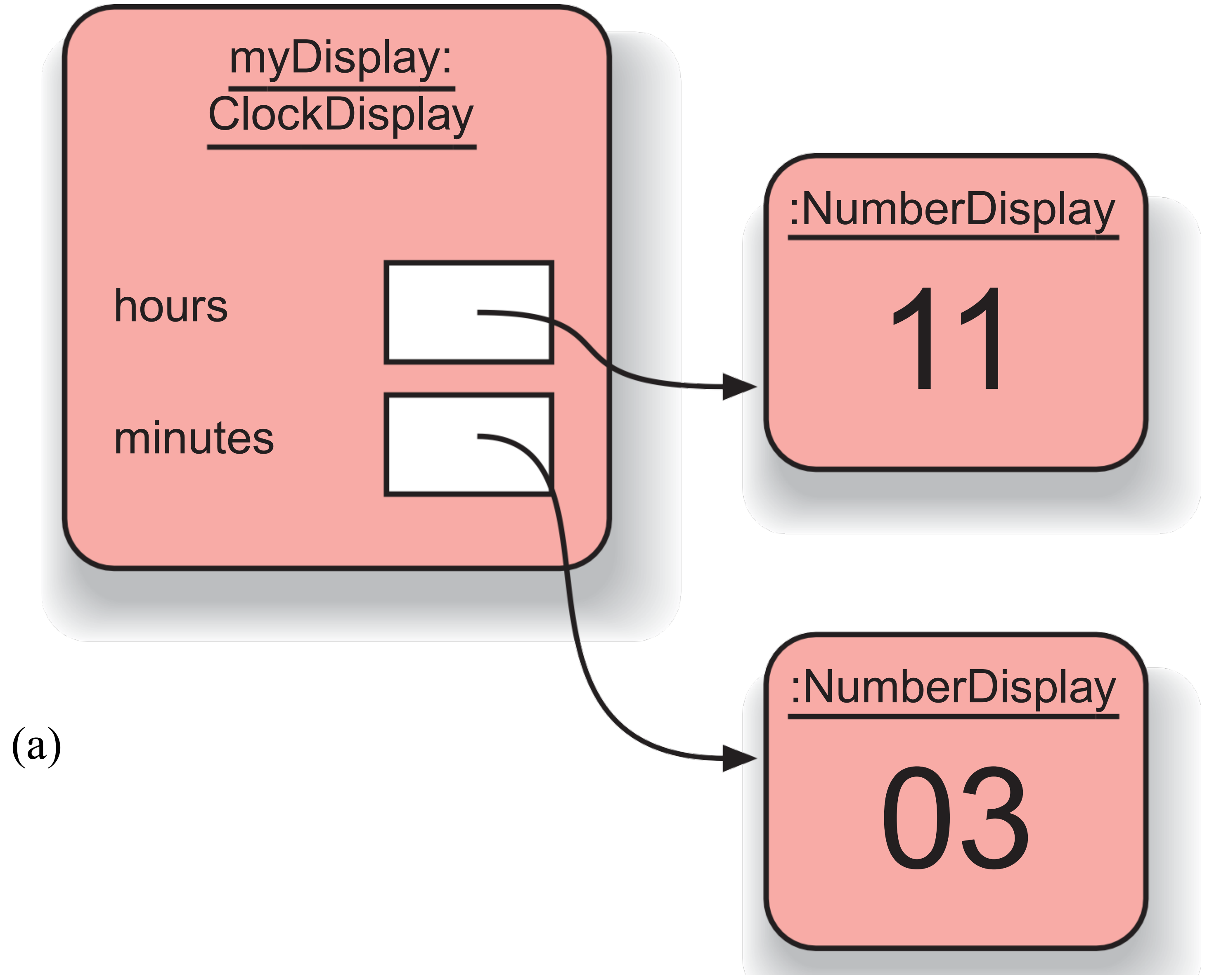
Modularizing the clock display

One *4-digits* display?

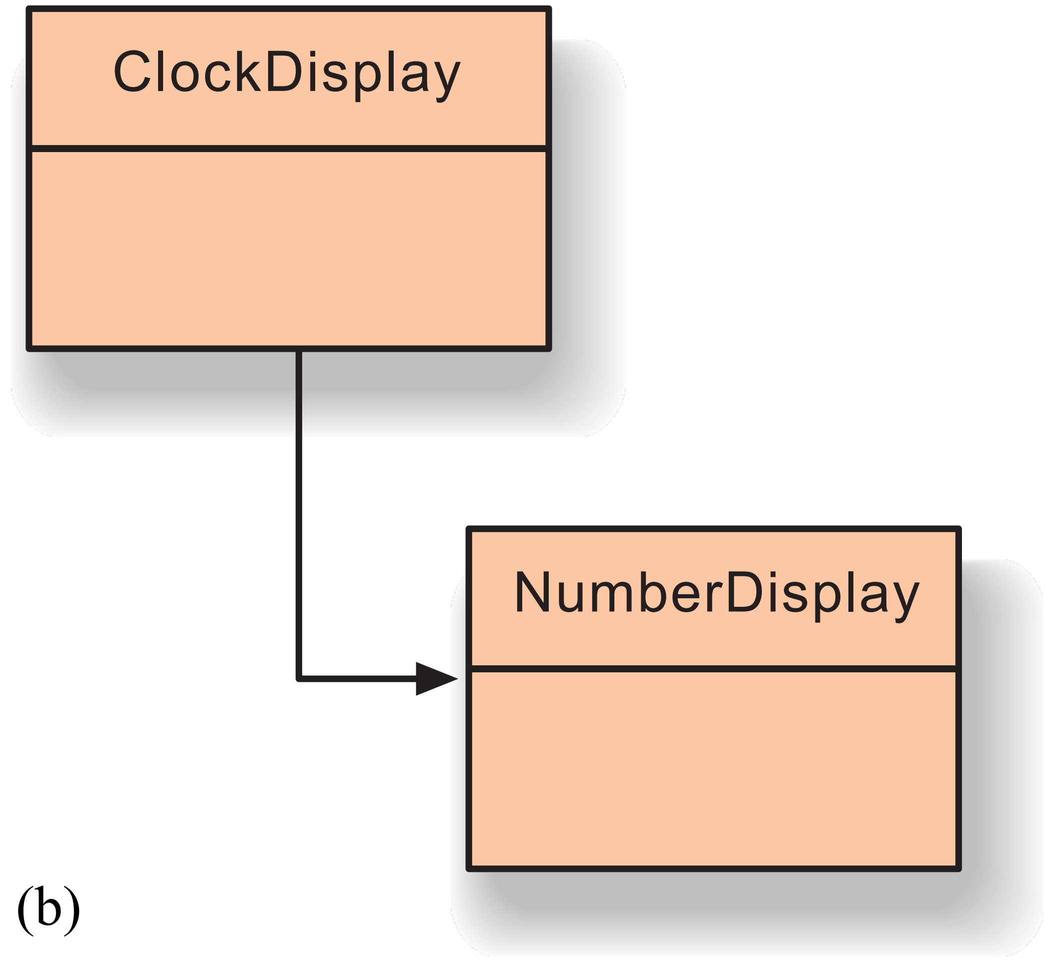
Or two *2-digits* displays?



Object & Classes

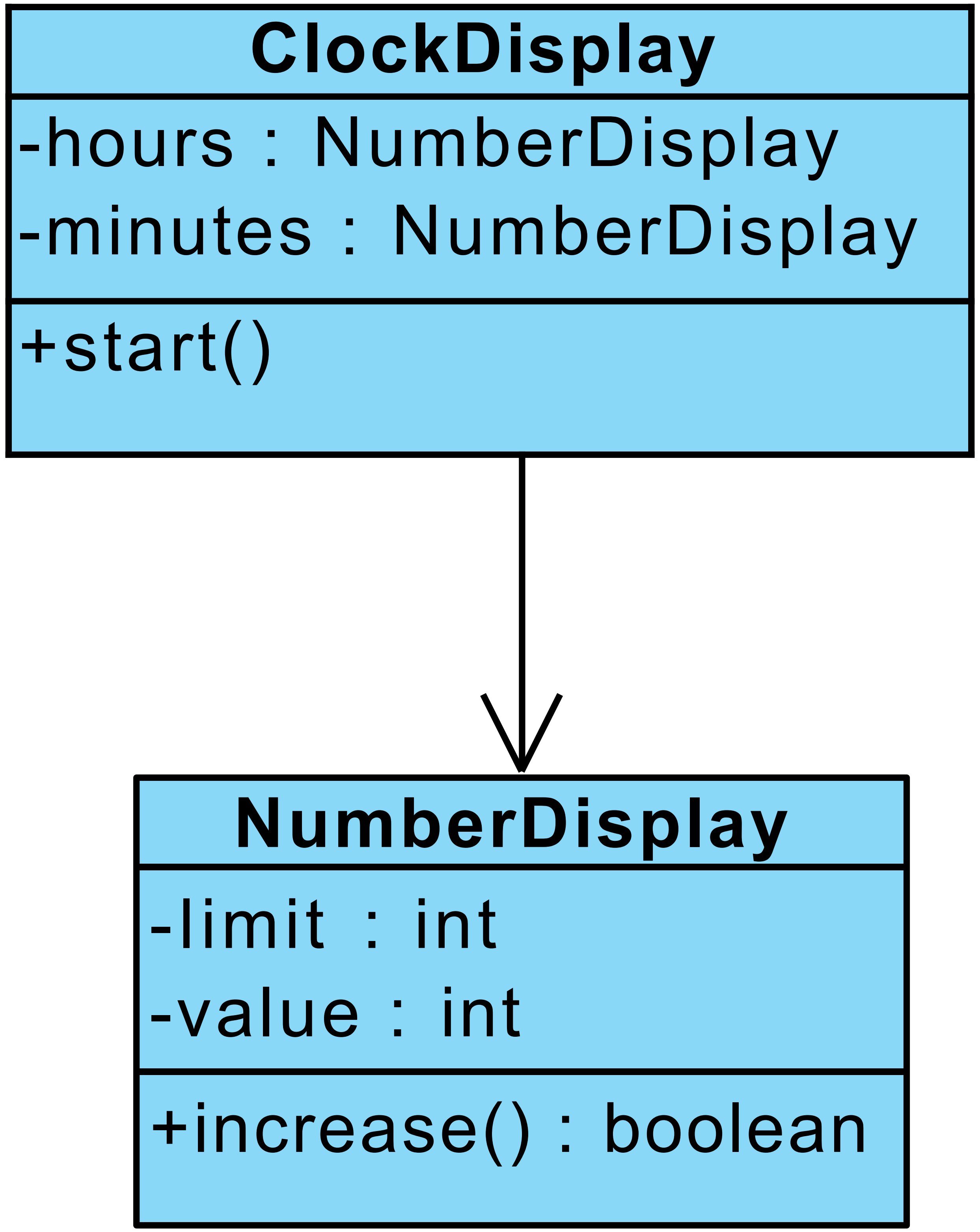


(a)



(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 YEA

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 does a constructor work?

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

How does a constructor work?

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

constructor

How does a constructor work?

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

constructor

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

How does a constructor work?

```
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 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.

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.

Storage allocation

Storage allocation

- The compiler allocates all the storage for a scope at the opening brace of that scope.

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

Aggregate initialization

- `int a[5] = {1,2,3,4,5};`

Aggregate initialization

- `int a[5] = {1,2,3,4,5};`
- `int b[6] = {5};`

Aggregate initialization

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

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'};`

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'} }`

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); };`

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.

The default constructor

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

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

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) };
```

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) };
```

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

“auto” default constructor

- If you have a constructor, the compiler ensures that construction always happens.

“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.

Local variable vs. Field

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

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?

Local variable vs. Field

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

A local variable of the same name as a field will prevent the field from being accessed within a method.

Fields, parameters, local variables

- All three kinds of variable are able to store a value that is appropriate to their defined type.
- Fields 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.

Call the functions in a class

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

- 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.

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.

Initialization

Initializer list

```
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 initializer list!
  - Destroyed in the reverse order.

Initialization vs. assignment

Initialization vs. assignment

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

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

Initialization vs. assignment

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

initialization

before constructor body

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

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

Function overloading

- Same functions with different arguments list.

```
void print(char * str, int width); // #1
void print(double d, int width); // #2
void print(long l, int width); // #3
void print(int i, int width); // #4
void print(char *str); // #5

print("Pancakes", 15);
print("Syrup");
print(1999.0, 10);
print(1999, 12);
print(1999L, 15);
```

Function overloading

- Same functions with different arguments list.

```
void print(char * str, int width); // #1
void print(double d, int width); // #2
void print(long l, int width); // #3
void print(int i, int width); // #4
void print(char *str); // #5

print("Pancakes", 15); // #1
print("Syrup");        // #5
print(1999.0, 10);     // #2
print(1999, 12);       // #4
print(1999L, 15);      // #3
```

Overload and auto-cast

```
void f(int i);
void f(double d);

f('a');
f(2);
f(2L);
f(3.2f);
```

Default arguments

- A default argument is a value given in the declaration that the compiler automatically inserts if you don't provide a value in the function call.

```
Stash(int size, int initQuantity = 0);
```

- To define a function with an argument list, defaults must be added from right to left.

```
int harpo(int n, int m = 4, int j = 5);
int chico(int n, int m = 6, int j); //illegal
int groucho(int k = 1, int m = 2, int n = 3);

beeps = harpo(2);
beeps = harpo(1,8);
beeps = harpo(8,7,6);
```

Constant objects

Constant objects

- What if an object is const?

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

- What member functions can access the internals?

Constant objects

- What if an object is const?

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

- What member functions 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 member functions 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

- Cannot modify their objects

```
int Date::set_day(int d){
    //...error check d here...
    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 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 objects

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 in class

```
class A {
  const int i;
 };
```

- has to be initialized in initializer list of the constructor

Compile-time constants *in classes*

Compile-time constants *in classes*

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

Compile-time constants *in classes*

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

- Make the const value static:
  - `static const int size = 100;`
  - `static indicates only one per class (not one per object)`

Compile-time constants *in classes*

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

- Make the const value static:
  ```
  - static const int size = 100;
  - static indicates only one per class (not one per object)
  ```
- Or use "anonymous enum" hack :

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

Inline function

Overhead for a function call

- The processing time required by a device prior to the execution of a command
    - Push parameters
    - Push return address
    - Prepare return values
    - Pop all pushed

Inline

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

Inline Functions

- An inline function is expanded in place, like a preprocessor macro, so the overhead of the function call is eliminated.

Inline

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

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

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

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

```
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.

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

int f();

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

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

int f();

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

Inline inside classes

- Any function you define inside a class declaration is automatically an inline.
  - Example: Inline.cpp

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.

```
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 or not?

- Inline:
  - Small functions, 2 or 3 lines
  - Frequently called functions, e.g. inside loops

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.

Inline may not in-line

- Nowadays, the keyword inline for functions comes to mean "multiple definitions are permitted" rather than "inlining is preferred".