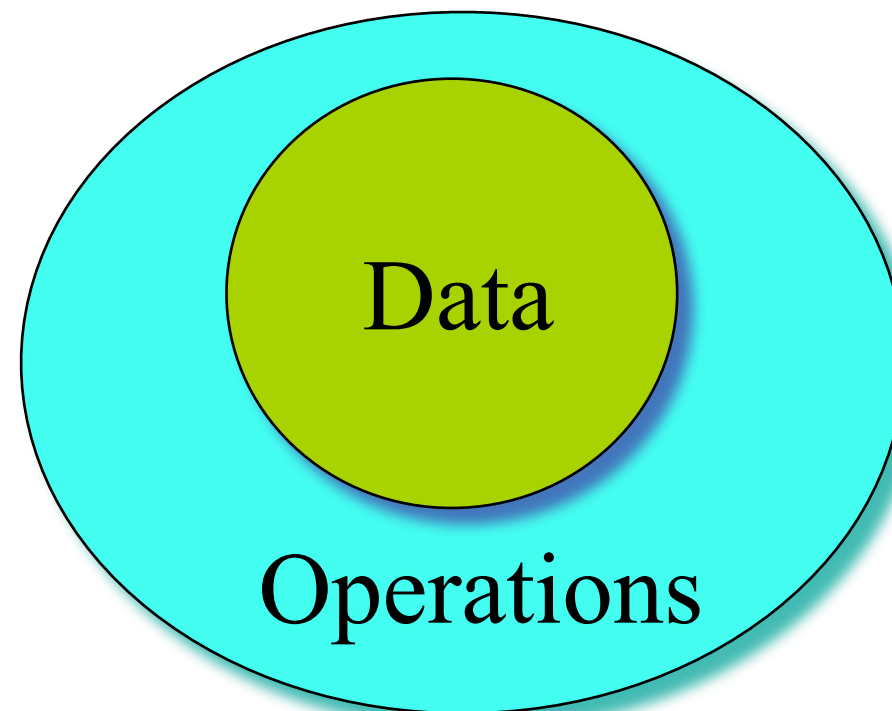


Class and Object

Weng Kai

Objects = Attributes + Services

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



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

1. 无参数即为default，并非一定是系统自动生成的
2. 当有构造器，就必须通过构造器初始化（否则，可以用C语言的结构赋值方式

- 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.
- Example: `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) };`

Definition of a class

- In C++, separated .h and .cpp files are used to define one class.
- Class declaration and prototypes in that class are in the header file (.h).
- All the bodies of these functions are in the source file (.cpp).

compile unit

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

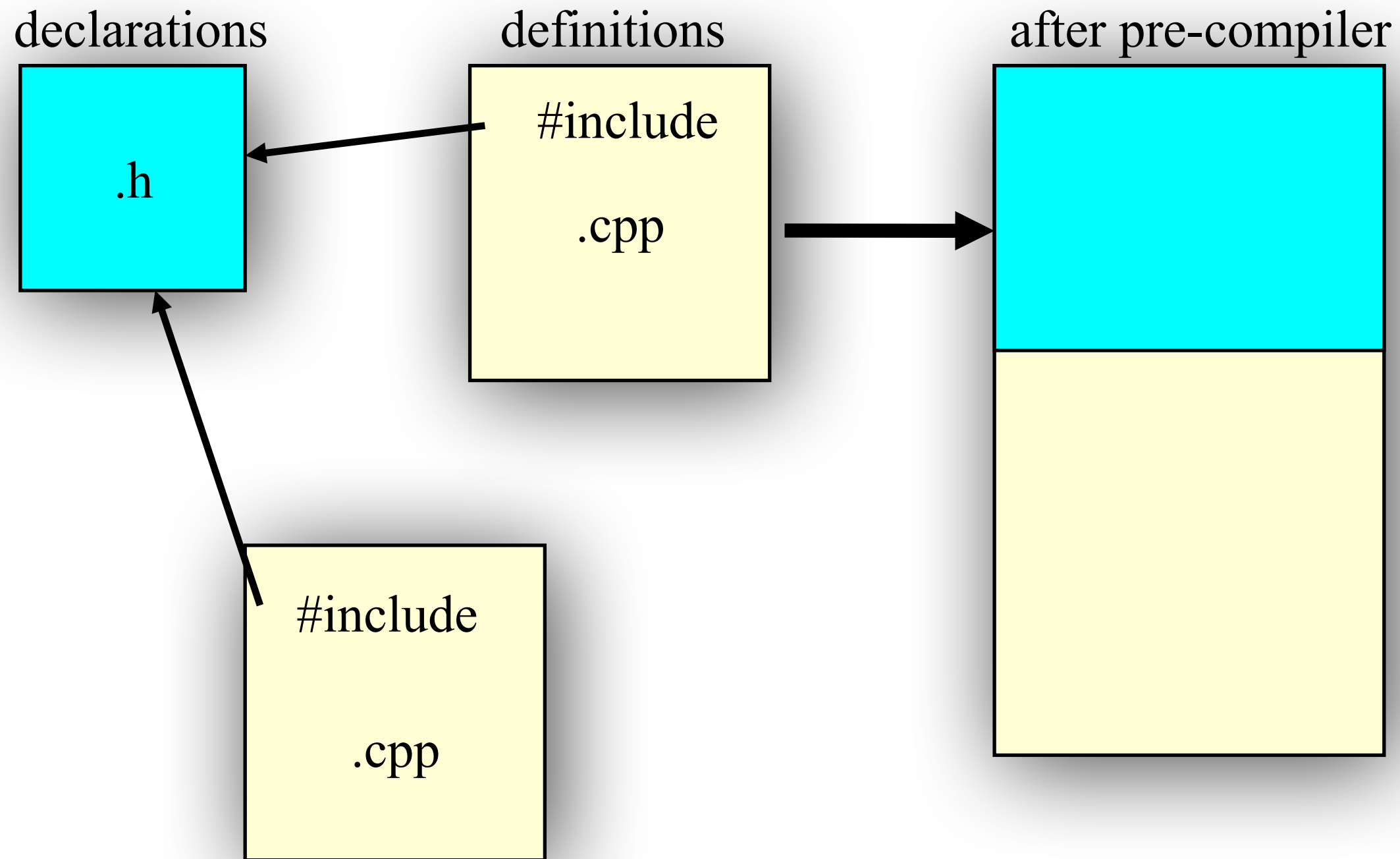
The header files

- If a function is declared in a header file, you *must* include the header file everywhere the function is used and where the function is defined.
- If a class is declared in a header file, you *must* include the header file everywhere the class is used and where class member functions are defined.

Header = interface

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

Structure of C++ program



Other modules that use the functions

Declarations vs. Definitions

- A .cpp file is a compile unit
- Only declarations are allowed to be in .h
 - extern variables
 - function prototypes
 - class/struct declaration

#include

- #include is to insert the included file into the .cpp file at where the #include statement is.
- #include "xx.h":first search in the current directory, then the directories declared somewhere
- #include <xx.h>:search in the specified directories
- #include <xx>:same as #include <xx.h>

Standard header file structure

```
#ifndef HEADER_FLAG  
#define HEADER_FLAG  
// Type declaration here...  
#endif // HEADER_FLAG
```

Tips for header

1. One class declaration per header file
2. Associated with one source file in the same prefix of file name.
3. The contents of a header file is surrounded with `#ifndef #define #endif`

Clock display

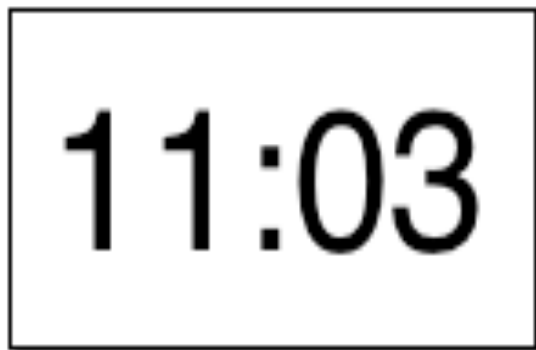


11:03

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



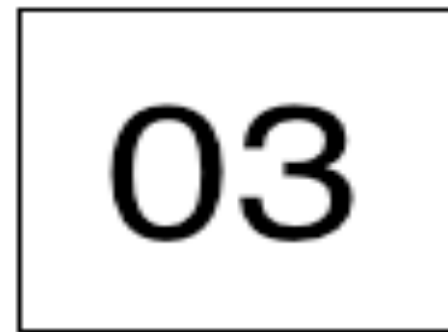
11:03

One four-digit display?

Or two two-digit displays?

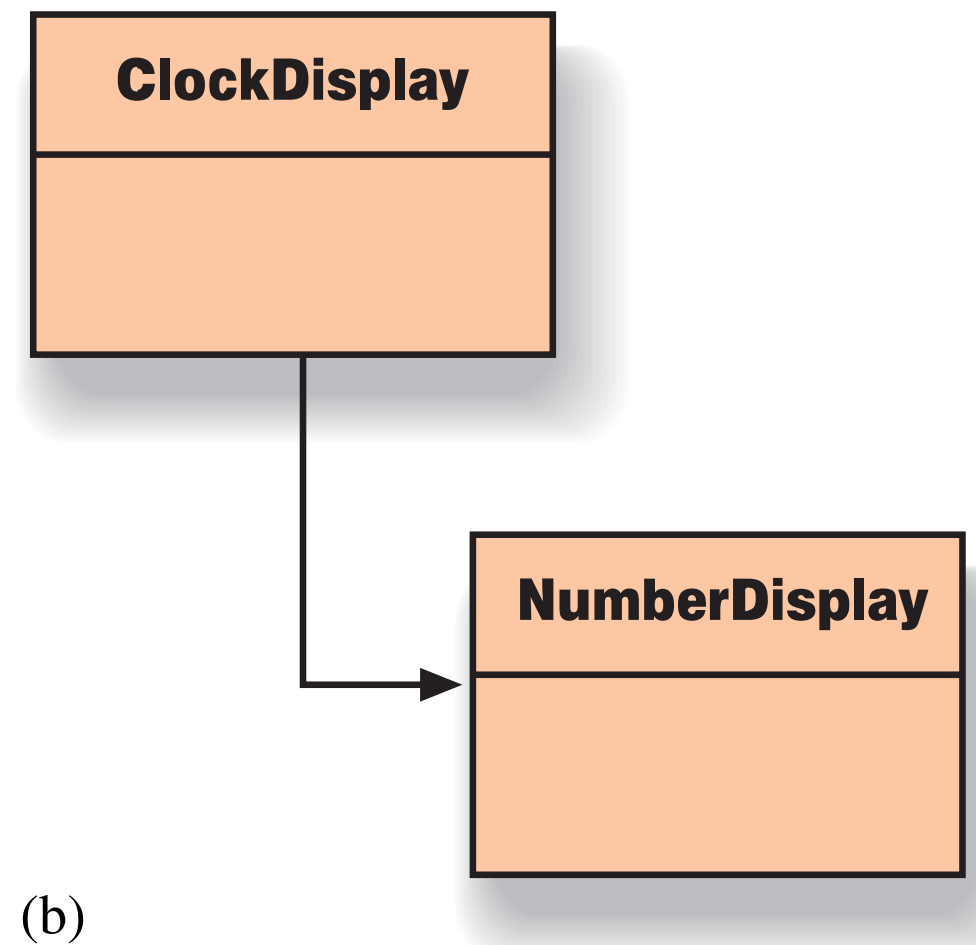
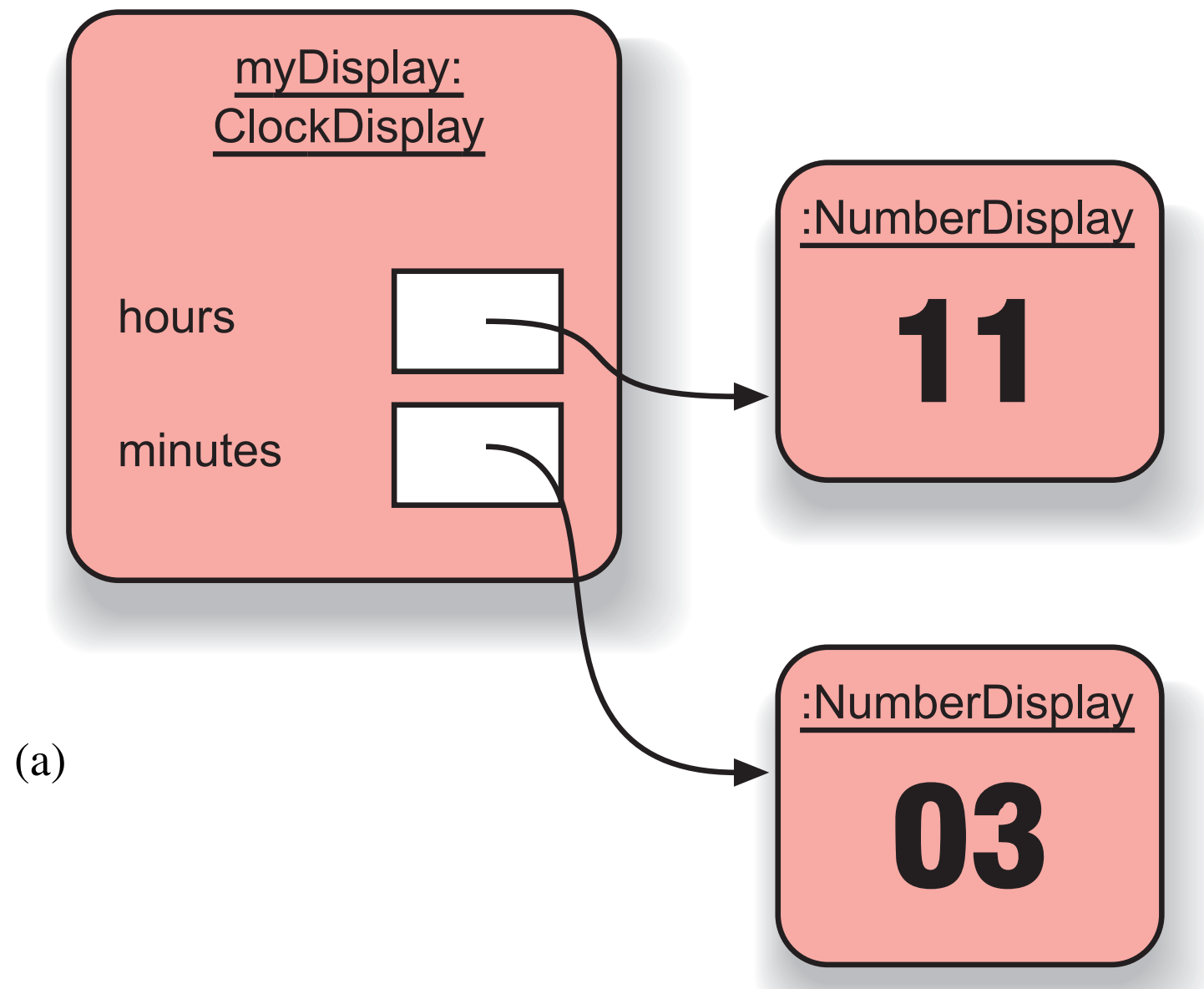


11

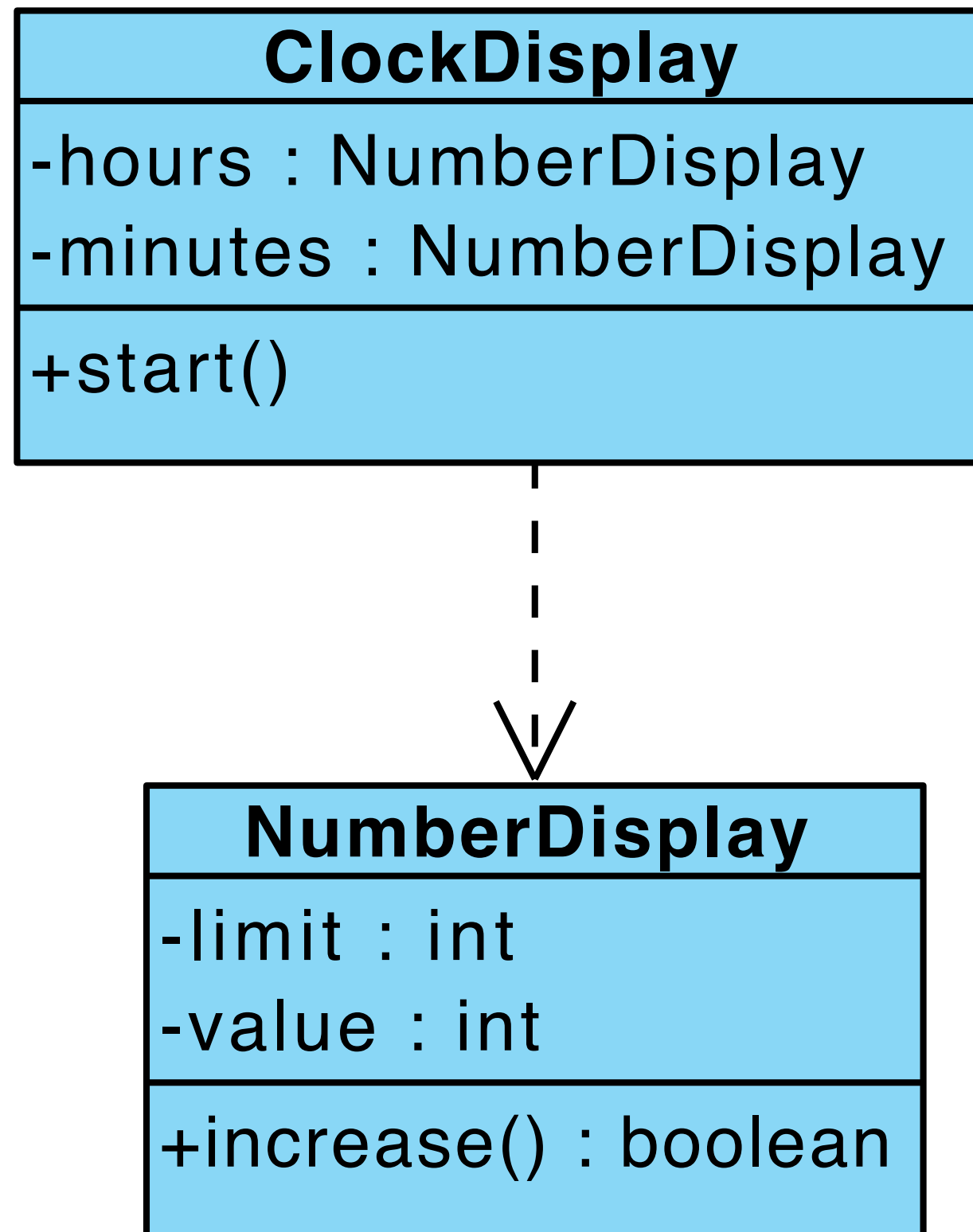


03

Objects & Classes



Class Diagram



Implementation - ClockDisplay

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

*Constructor and
methods omitted.*

```
}
```

Implementation - NumberDisplay

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

*Constructor and
methods omitted.*

```
}
```

local variable

```
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 being accessed from 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.

- 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.

- 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.

Initialization

Member Init

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

Initializer list

```
class Point {  
private:  
    const float x, y;  
    Point(float xa = 0.0, float ya = 0.0)  
        : 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

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);
```

Example: leftover.cpp

Overload and auto-cast

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

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

Example: overload.cpp

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); //illegale  
int groucho(int k = 1, int m = 2, int n = 3);
```

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

Example: left.cpp