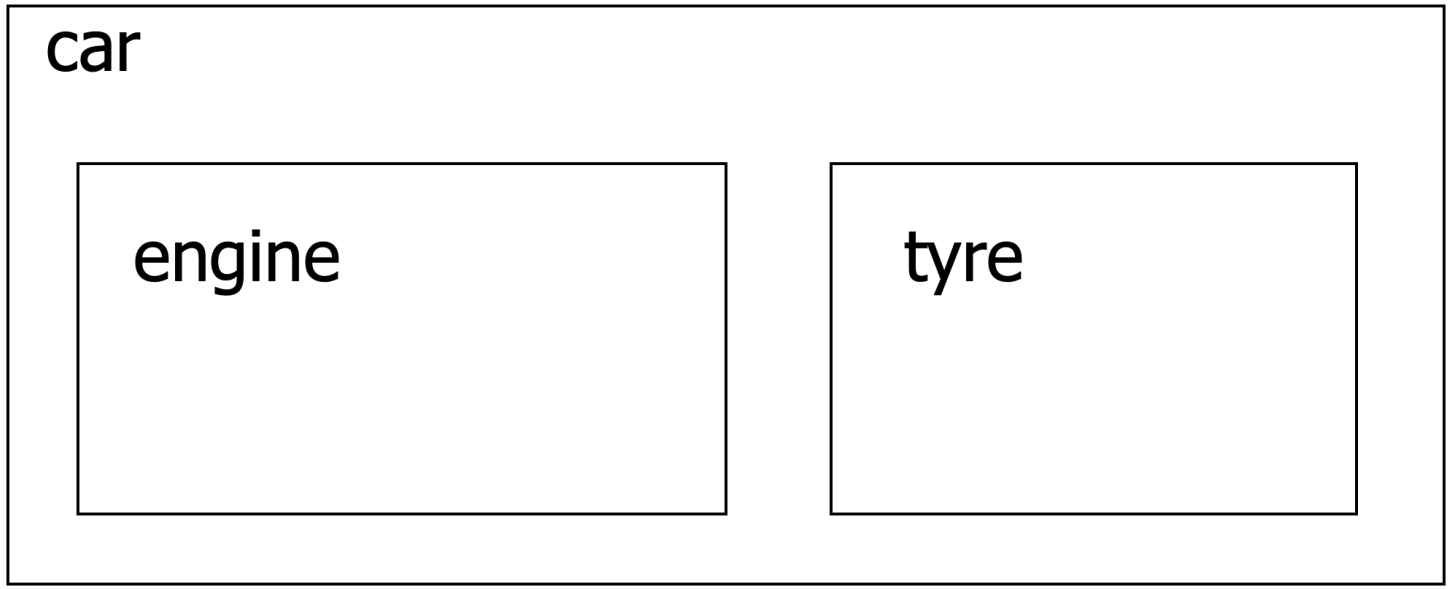
Composition

Reusing the implementation

- Composition: construct new object with existing objects
- It is the relationship of “has-a”



Each object has its own memory consists of other objects. -- by Alan Kay

Composition

- Objects can be used to build up other objects
- Ways of inclusion
    - Fully
    - By reference (Inclusion by reference allows sharing)
- For example, an Employee has a

```
Name
Address
Health Plan
Salary History: Collection of Raise objects
Supervisor: Another Employee object!
```

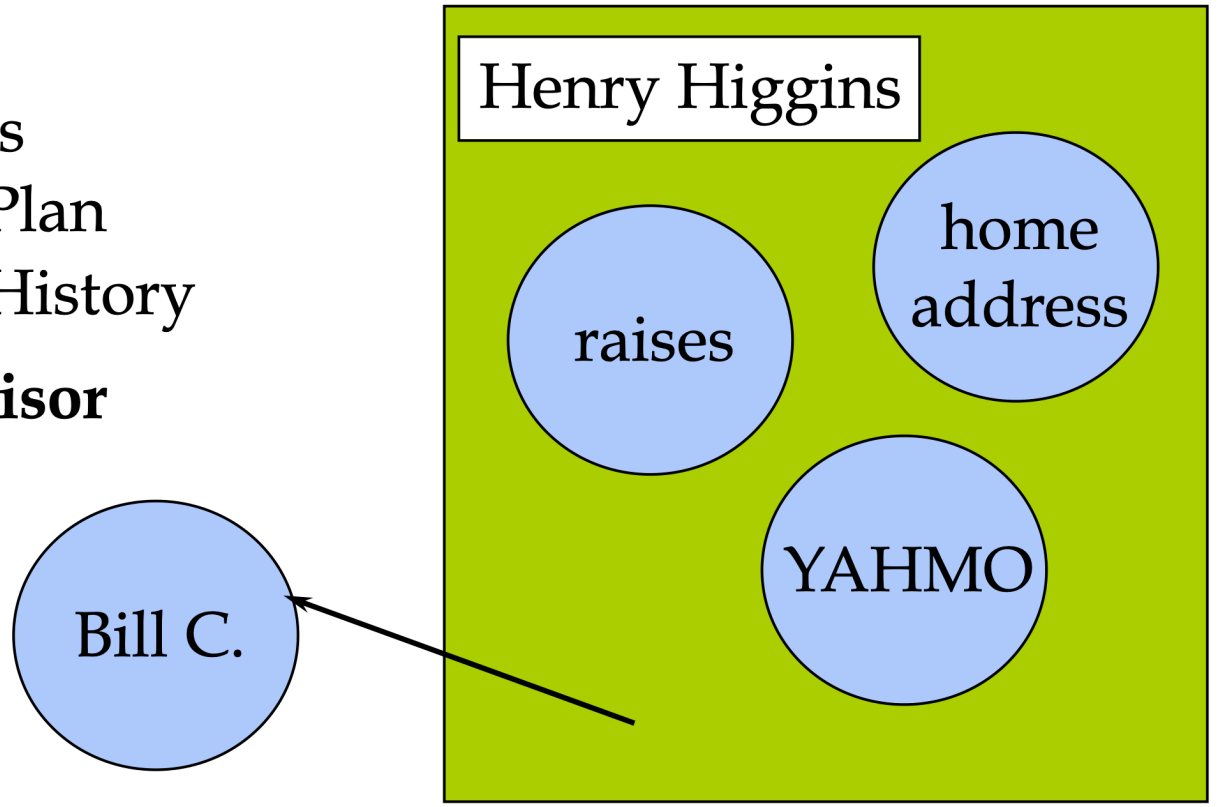
Composition in action

Classes

Employee

- Name
- Address
- HealthPlan
- Salary History
- **Supervisor**

Instances



Example

```
class Person { ... };
class Currency { ... };
class SavingsAccount {
public:
    SavingsAccount(
        const char* name,
        const char* address,
        int cents );
    ~SavingsAccount();
    void print();
private:
    Person m_saver;
    Currency m_balance;
};
```

Example...

```
SavingsAccount::SavingsAccount (
    const char* name,
    const char* address,
    int cents ) : m_saver(name, address),
    m_balance(0, cents) {}

void SavingsAccount::print() {
    m_saver.print();
    m_balance.print();
}
```

Embedded objects

- All embedded objects are initialized
  - The default constructor is called if
    - you don’t supply the arguments, and there is a default constructor (or one can be built)
- Constructors can have initialization list
  - any number of objects separated by commas
  - is optional
  - provide arguments to sub-constructors
- The destructors will be called automatically

Remember

- If we wrote the constructor as (assuming we have the set accessors for the sub-objects):

```
SavingsAccount::SavingsAccount (
    const char* name,
    const char* address,
    int cents ) {
    m_saver.set_name( name );
    m_saver.set_address( address );
    m_balance.set_cents( cents );
}
```

Default constructors would be called

`public` vs. `private`

- It is common to make embedded objects private:
  - they are part of the underlying implementation
  - the new class only has part of the public interface of the old class
- Can embed as a public object if you want to have the entire public interface of the subobject available in the new object:

```
class SavingsAccount {
public:
    Person m_saver;  ...
}; // assume Person class has set_name()

SavingsAccount  account;
account.m_saver.set_name("Fred" );
```

Fully vs by reference

- Fully means "It is here, as part of this object", while by reference means "It is there"
- For fully, the constructors and destructors will be called automatically, while for by reference, it is your job to init and destroy the objects
- By reference usually is used at:
  - The logical relationship is not a fully
  - The size-of is not known at the beginning
  - The resource is to be allocated/connected at run-time
- Other OOP languages use by reference only

Clock display

Modularization

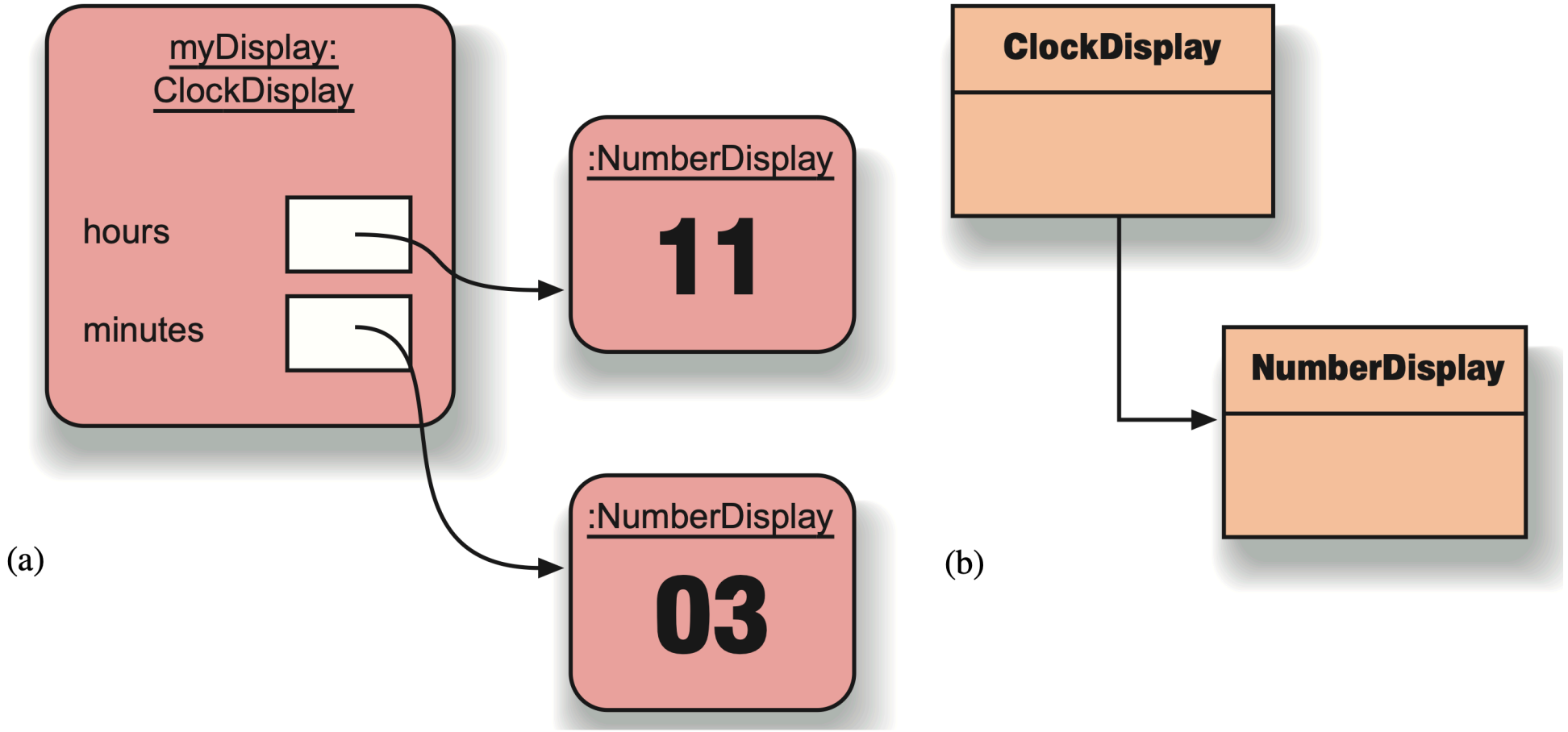
- is the process of dividing a whole into well-defined parts, which can be built and examined separately, and which interact in well- defined ways.

Modularizing the clock display

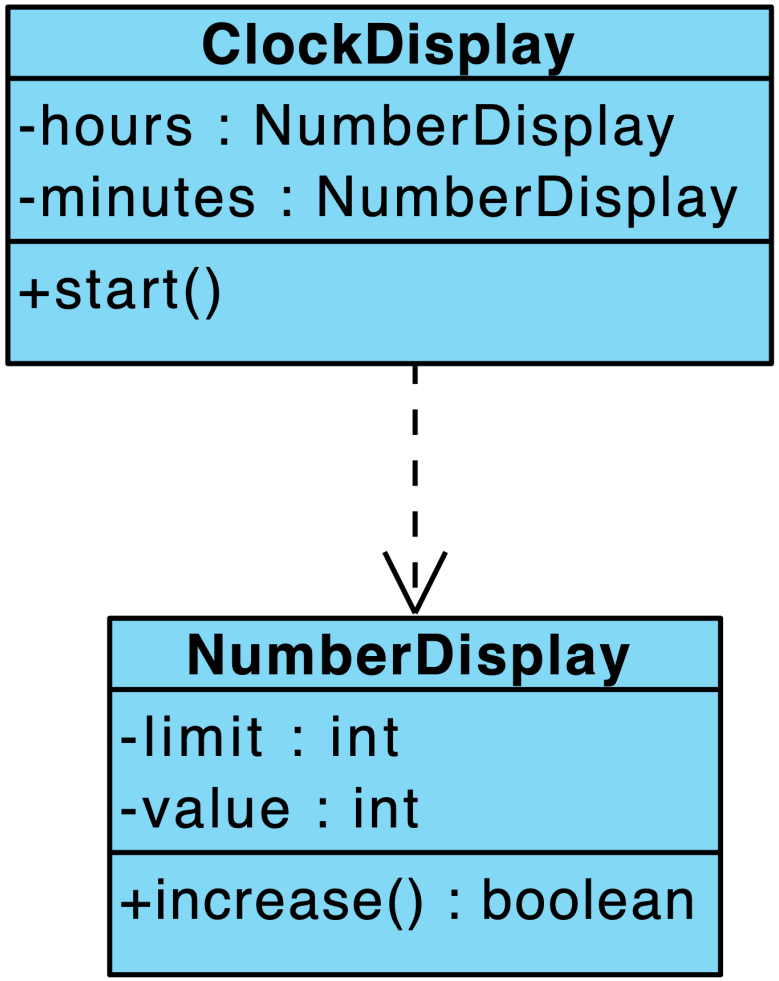
One four-digit display?

Or two two-digit displays?

Objects & Classes



Class diagram



Implementation -- ClockDisplay

```
class ClockDisplay {
    NumberDisplay hour;
    numberDisplay minute;
    ...
}
```

Implementation -- NumberDisplay

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

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

Implementation -- Clock

```
class Clock {
    NumberDisplay hour;
    numberDisplay minute;
    ...
}
```

- What should be the constructor of the Clock?

Namespace

Controlling names:

- Controlling names through scoping
- We've done this kind of name control:

```
class Marbles {
   enum Colors { Blue, Red, Green };
   //...
};
class Candy {
   enum Colors { Blue, Red, Green };
   //...
};
```

Avoiding name clashes

- Including duplicate names at global scope is a problem:

```
// old1.h
void f();
void g();
```

```
// old2.h
void f();
void g();
```

Avoiding name clashes (cont)

- Wrap declarations in namespaces.

```
// old1.h
namespace old1 {
   void f();
   void g();
}
```

```
// old2.h
namespace old2 {
   void f();
   void g();
}
```

Namespace

- Expresses a logical grouping of classes, functions, variables, etc.
- A namespace is a scope just like a class
- Preferred when only name encapsulation is needed

```
namespace Math {
   double abs(double );
   double sqrt(double );
   int trunc(double);
}// Note: No terminating end colon!
```

...

Defining namespaces

- Place namespaces in include files:

```
// Mylib.h
namespace MyLib {
   void foo();
   class Cat {
   public:
      void Meow();
   };
}
```

Defining namespace functions

- Use normal scoping to implement functions in namespaces.

```
// MyLib.cpp
#include "MyLib.h"
void MyLib::foo() { cout << "foo\n"; }
void MyLib::Cat::Meow() { cout << "meow\n"; }
```

Using names from a namespace

- Use scope resolution to qualify names from a namespace.
    - Can be tedious and distracting.

```
#include "MyLib.h"
void main() {
   MyLib::foo();
   MyLib::Cat c;
   c.Meow();
}
```

Using-Declarations

- Introduces a local synonym for name
- States in one place where a name comes from.
- Eliminates redundant scope qualification:

```
void main() {
   using MyLib::foo;
   using MyLib::Cat;
   foo();
   Cat c;
   c.Meow();
}
```

Using-Directives

- Makes all names from a namespace available.
- Can be used as a notational convenience.

```
void main() {
   using namespace std;
   using namespace MyLib;
   foo();
   Cat c;
   c.Meow();
   cout << "hello" << endl;
}
```

Ambiguities

- Using-directives may create potential ambiguities.
- Consider:

```
// Mylib.h
namespace XLib {
   void x();
   void y();
}
namespace YLib {
   void y();
   void z();
}
```

Ambiguities (cont)

- Using-directives only make the names available.
- Ambiguities arise only when you make calls.
- Use scope resolution to resolve.

```
void main() {
   using namespace XLib;
   using namespace YLib;
   x(); // OK
   y(); // Error: ambiguous
   XLib::y(); // OK, resolves to XLib
   z(); // OK
}
```

Namespace aliases

- Namespace names that are too short may clash
- names that are too long are hard to work with
- Use aliasing to create workable names
- Aliasing can be used to version libraries.

```
namespace supercalifragilistic {
   void f();
}
namespace short = supercalifragilistic;
short::f();
```

Namespace composition

- Compose new namespaces using names from other ones.
- Using-declarations can resolve potential clashes.
- Explicitly defined functions take precedence.

```
namespace first {
   void x();
   void y();
}
namespace second {
   void y();
   void z();
}
```

Namespace composition (cont)

```
namespace mine {
   using namespace first;
   using namespace second;
   using first::y(); // resolve clashes to first::x()
   void mystuff();
   // ...
}
```

Namespace selection

- Compose namespaces by selecting a few features from other namespaces.
- Choose only the names you want rather than all.
- Changes to “orig” declaration become reflected in “mine”.

```
namespace mine {
   using orig::Cat; // use Cat class from orig
   void x();
   void y();
}
```

Namespaces are open

- Multiple namespace declarations add to the same namespace.
    - Namespace can be distributed across multiple files.

```
//header1.h
namespace X {
   void f();
}
```

```
// header2.h
namespace X {
   void g(); // X how has f() and g();
}
```

What we've learned today?

- compsition
- initialization list
- namespace