Object－Oriented Programming

Week 6

Functions

Weng Kai

const

Const

- declares a *variable* to have a constant value

```
const int x = 123;
x = 27; // illegal!
x++; // illegal!

int y = x; // Ok, copy const to non-const
y = x;     // Ok, same thing

const int z = y; // ok, const is safer
```

Constants

- Constants are variables
  - Observe scoping rules
  - Declared with “const” type modifier
- A const in C++ defaults to internal linkage
  - the compiler tries to avoid creating storage for a const -- holds the value in its symbol table.
  - extern forces storage to be allocated.

Compile time constants

```
const int bufsize = 1024;
```

- value must be initialized
- unless you make an explicit extern declaration:

```
extern const int bufsize;
```

- Compiler won't let you change it

- Compile time constants are entries in compiler symbol table, not really variables.

Run-time constants

- const value can be exploited

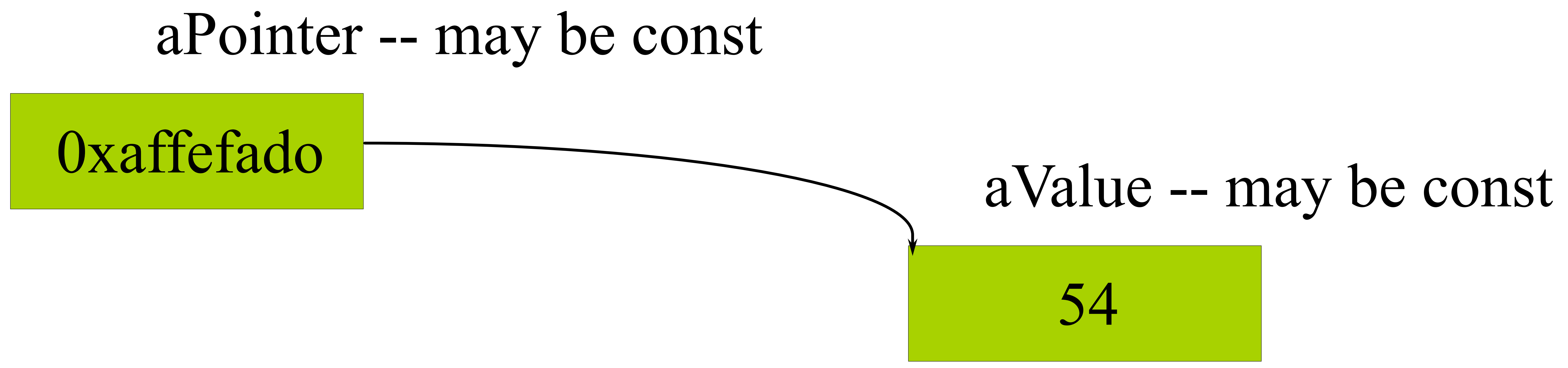
```
const int class_size = 12;
int finalGrade[class_size]; // ok

int x;
cin >> x;
const int size = x;
double classAverage[size]; // error!
```

Aggregates

- It’s possible to use **const** for aggregates, but storage will be allocated. In these situations, **const** means “a piece of storage that cannot be changed.” However, the value cannot be used at compile time because the compiler is not required to know the contents of the storage at compile time.

```
const int i[] = { 1, 2, 3, 4 };

float f[i[3]]; // Illegal

struct S { int i, j; };

const S s[] = { { 1, 2 }, { 3, 4 } };

double d[s[1].j]; // Illegal
```

Pointers and const



- `char * const q = "abc"; //  q is const`
  ```
  *q = 'c'; // OK
  q++;      // ERROR
  ```
- `const char *p = "ABCD";`
  ```
  // (*p) is a const char
  *p = 'b'; // ERROR! (*p) is the const
  ```

Quiz: What do these mean?

```
string p1( “Fred" );
const string* p = &p1;
string const* p = &p1;
string *const p = &p1;
```

Pointers and constants

	`int i;`	`const int ci = 3;`
`int *  ip;`	`ip = &i;`	`ip = &ci; //Error`
`const int *cip`	`cip = &i;`	`cip = &ci;`

Remember:

```
*ip  = 54;  // always legal since ip points to int
*cip = 54;  // never legal since cip points to const int
```

String Literals

```
char* s = "Hello, world!";
```

- s is a pointer initialized to point to a string constant
- This is actually a `const char* s` but compiler accepts it without the const
- Don't try and change the character values (it is undefined behavior)

- If you want to change the string, put it in an array:

```
char s[] = "Hello, world!";
```

Conversions

- Can always treat a non-const value as const

```
void f(const int* x);

int a = 15;

f(&a); // ok

const int b = a;


f(&b); // ok

b = a + 1; // Error!
```

You cannot treat a constant object as non-constant without an explicit cast (const_cast)

Passing by const value?

```
void f1(const int i) {

  i++; // Illegal -- compile-time error

}
```

Returning by const value?

```
int f3() { return 1; }

const int f4() { return 1; }

int main() {

  const int j = f3(); // Works fine

  int k = f4(); // But this works fine too!

}
```

Passing and returning addresses

- Passing a whole object may cost you a lot. It is better to pass by a pointer. But it’s possible for the programmer to take it and modify the original value.
- In fact, whenever you’re passing an address into a function, you should make it a **const** if at all possible.
- Example: ConstPointer.cpp, ConstReturning.cpp

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?
- 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 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*

```
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!

    ...

};
```

static

Static in C++

Two basic meanings

- Static storage
  - allocated once at a fixed address
- Visibility of a name
  - internal linkage
- Don't use static except inside functions and classes.

Uses of "static" in C++

Static free functions	~~Internal linkage~~ *(deprecated)*
Static global variables	~~Internal linkage~~ *(deprecated)*
Static local variables	Persistent storage
Static member variables	Shared by all instances
Static member function	Shared by all instances, can only access static member variables

Global static hidden in file

File1

```
int g_global;
static int s_local;

void
func() {
  ...
}

static
void
hidden() { ...}
```

?

File2

```
extern int g_global;
void func();

extern int s_local;
int
myfunc() {
  g_global += 2;
  s_local *= g_global;
  func();
}
```

Static inside functions

- Value is remembered for entire program
- Initialization occurs only once
- Example:
  - count the number of times the function has been called

```
void f() {
    static int num_calls = 0;
     ...
    num_calls++;
}
```

Static applied to objects

- Suppose you have a class

```
class X {
  X(int, int);
   ~X();
...
};
```

- And a function with a static X object

```
void f() {
  static X my_X(10, 20);
   ...
}
```

Static applied to objects ...

- Construction occurs when definition is encountered
  - –Constructor called at-most once
  - –The constructor arguments must be satisfied
- Destruction takes place on exit from *program*
  - –Compiler assures LIFO order of destructors

Conditional construction

- Example: conditional construction

```
void f(int x) {
 if (x > 10) {
     static X my_X(x, x * 21);
     ...
 }
```

- `my_X`
  - is constructed once, if f() is ever called with x > 10
  - retains its value
  - destroyed only if constructed

Global objects

- Consider

```
#include "X.h"
X global_x(12, 34);
X global_x2(8, 16);
```

- Constructors are called before main() is entered
  - Order controlled by appearance in file
  - In this case, `global_x` before `global_x2`
  - main() is no longer the *first* function called
- Destructors called when
  - main() exits
  - exit() is called

Static Initialization Dependency

- Order of construction within a file is known
- Order between files is *unspecified*!
- Problem when non-local static objects in different files have dependencies.
- A non-local static object is:
  - defined at global or namespace scope
  - declared static in a class
  - defined static at file scope

Static Initialization Solutions

- Just say no -- avoid non-local static dependencies.
- Put static object definitions in a single file in correct order.

Can we apply static to members?

- Static means
  - Hidden
  - Persistent
- Hidden: *A static member is a member*
  - Obeys usual access rules
- Persistent: *Independent of instances*
- Static members are class-wide
  - variables or
  - functions

Static members

- Static member variables
  - Global to all class member functions
  - *Initialized once, at file scope*
  - provide a place for this variable and init it in .cpp
  - No ‘static’ in .cpp

- Example: StatMem.h, StatMem.cpp

Static members

- Static member functions
  - Have no implicit receiver ("this")
    - (why?)
  - *Can access only static member variables*
    - (or other globals)
  - No ‘static’ in .cpp
  - Can’t be dynamically overridden

- Example: StatFun.h, StatFun.cpp

To use static members

- <class name>::<static member>
- <object variable>.<static member>

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

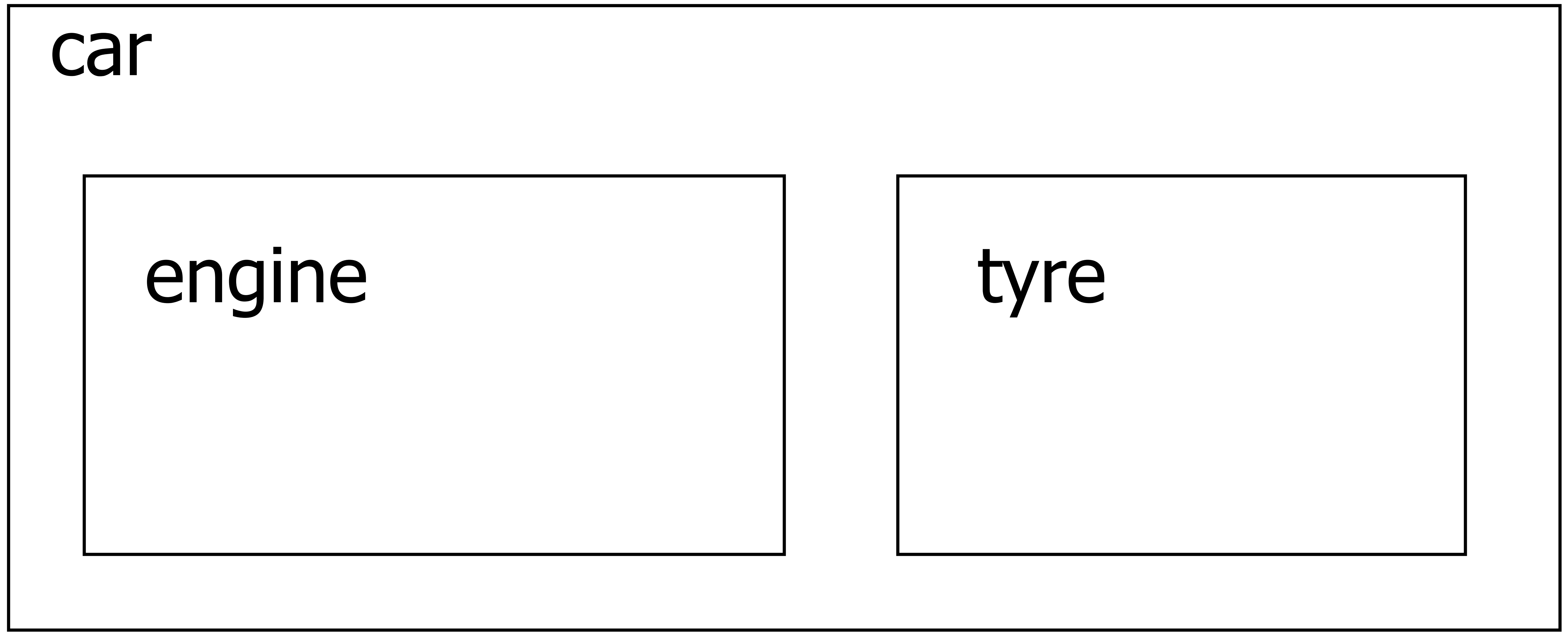
- Multiple namespace declarations add to the same namespace.
  - Namespace can be distributed across multiple files.

```
//header1.h
namespace X {
  void f();
}
// header2.h
namespace X {
  void g(); // X how has f() and g();
}
```

Inheritance

Reusing the implementation

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



Composition

- Objects can be used to build up other objects
- Ways of inclusion
  - Fully
  - By reference
- Inclusion by reference allows sharing

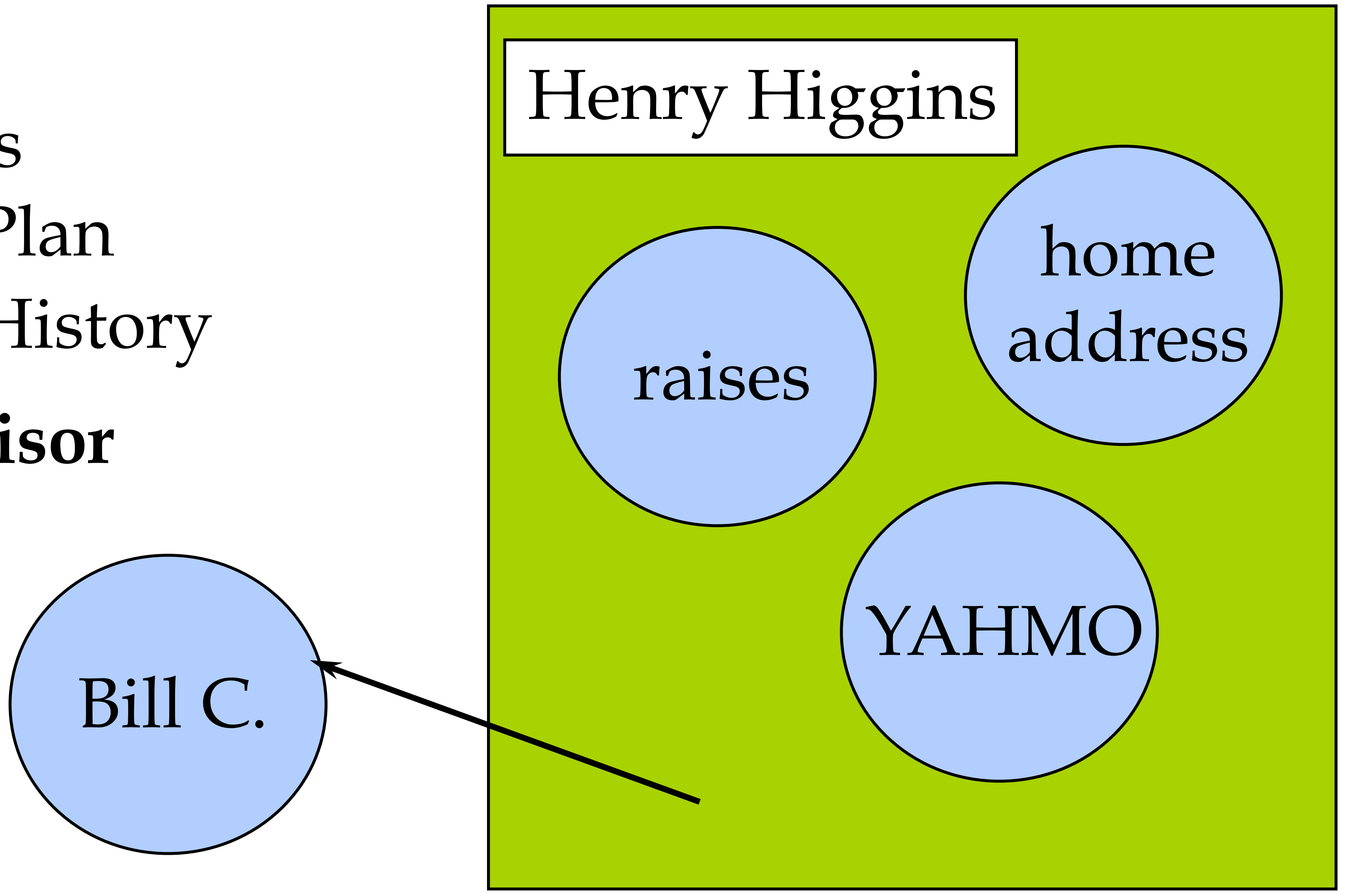
- For example, an Employee has a
  - Name
  - Address
  - Health Plan
  - Salary History
    - Collection of Raise objects
  - Supervisor
    - Another Employee object!

Composition in action

Classes

Employe
e
- 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
- Syntax:

```
name( args ) [‘:’ init-list] ‘{‘
```

Question

- 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" );
```