Contemporary C++: Learning Modern C++ in a Modern Way

الماس فناوری ابری پاسارگاد- آلفا

مدرس: سعید امراللهی بیوکی

Agenda 9/24

Session 9. More on Classes: destructors, static members, self-reference and more (part II)

- Class members initialization vs. assignment
- C++11 In-class member initialization
- Special member functions: Destructors
- Static members: data members and member functions
- Self-reference: the this pointer
- Explicit constructor
- Physical and Logical constness: the mutable keyword
- Friends
- Q&A

150 min (incl. Q & A)

The Date Class so far …

```
class Date {
    Date(int = 0, int = 0, int = 0); // day, month, year
    Date(const char*); // Date in string rep.
    int day() const { return d_; }
    int month() const { return m_; }
    int year() const { return y_; }
    void add_year(int y) { y_ += y; }
    void add_month(int m) { /* … */ }
    void add_day(int d) { /* … */ }
private:
    int d_, m_, y_;
};
```

The Vector Class so far …

```
class Vector {
    int size_;
    int* elem_; // pointer to the 1st element (of type int)
public:
    Vector(int s); // constructor: allocate s ints,
                   // let elem point to them
                   // store s in sz
    int size() const { return size_; }
    int get(int i) const { return elem_[i]; }
    void set(int i, int val) { elem_[i] = val; }
};
```

class member Initialization

- Initialization vs. Assignment
- In C++, an initialization occurs when a new object is created; an assignment changes the value of an existing object (no new object is created.

```
Thing t = x;      // Initialization (new Thing created)
t = x;            // Assignment (value of existing Thing changed)
```

Initialization

```
class Date {
    Date(int d, int m, int y)
    {
        d_ = d; m_ = m; y_ = y; // assignment
    }
    // …
private:
    int d_, m_, y_;
};
```

Assignment

Vs.

```
class Date {
    Date(int d, int m, int y) :
        d_(d), m_(m), y_(y) // initialization
    {
    }
    // …
private:
    int d_, m_, y_;
};
```

- Order of initialization
- Uniform initialization:

Uniform initialization

C++98

```
int i = 1; // assignment style
int j = i; // initialization
int* ip = &i;
string a[] = { "foo", "bar" };
vector<int> v{1, 2, 3}; // error
Point p(1, 2); // functional initialization
Point* pp = new Point{1, 2};
```

Vs.

C++11

```
int i{1};
int j{i};
int* ip{&i};
string a[] = { "foo", "bar" };
vector<int> v{1, 2, 3}; // OK
Point p = {1, 2}; // = is optional
Point p2{1, 2};
Point* pp{new Point{1,2}};
```

- The class Date

```
class Date {
    Date(int d, int m, int y) :
        d_{d}, m_{m}, y_{y} // initialization
    {
    }
    // …
private:
    int d_, m_, y_;
};
```

In-class member initializers

- C++98 → Only *static const members of integral types* can be initialized in-class, and the initializer has to be a constant expression.

```
class Values {
   static const long c = 3e8; // the light speed
    static const float g = 9.81; // error: not integral type
   const int i = 10; // error: not static
    static int j = 11; // error: not const
};
```

- C++11 → You can initialize non-static data members. Often, all constructors use a common initializer for a member:
- Example:

```
class FinInst { // financial instrument
    string market = "Dow";
   string symbol;
public:
   FinInst() {}
   FinInst(string symb) : symbol(symb) {}
};
FinInst Microsoft("MSFT"), Oracle("ORCL");
```

In-class member initializers- details

- An in-class member initializer is effectively *syntactic sugar* for an ordinary mem-initializer.

```
class A {
   int a = 0;
};
```

It’s equivalent to

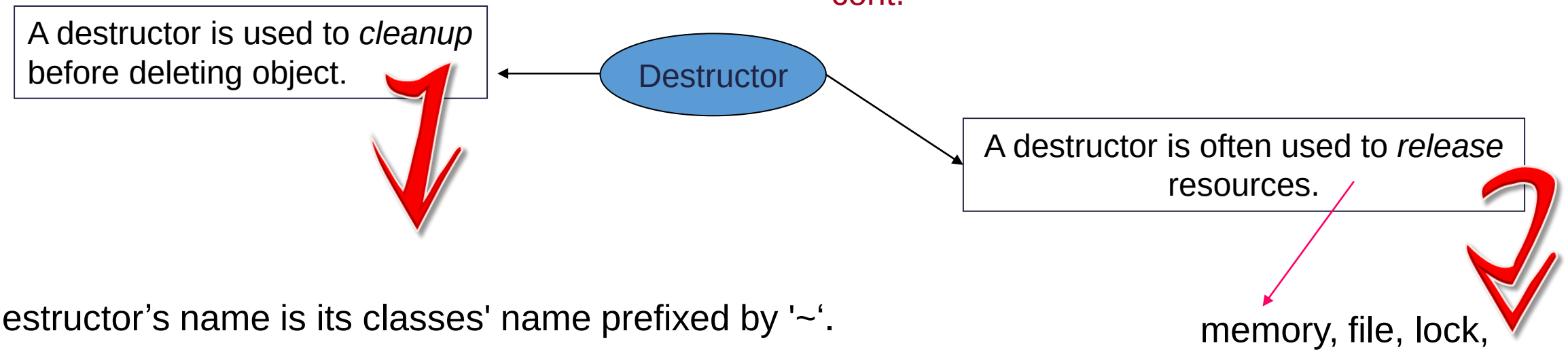
```
class A {
   int a;
public:
   A() : a(0) {}
};
```

- If a member is initialized by both an in-class initializer and a constructor, only the constructor's initialization is done (it "overrides" the default).

- Another example:

```
struct Univ {
    string name;
    int rank;
    string city = "unknown";
};
```

Destructor cont.



- The destructor’s name is its classes' name prefixed by '~‘.
- A destructor is implicitly called whenever an object goes out of scope or is deleted.
- Destructors do not return value and don’t have arguments.
- Destructors can’t be overloaded.
- If a user has declared a destructor, that one will be used.
- `Date` and `Point` don’t need to define destructor.

Constructors & destructors: another example

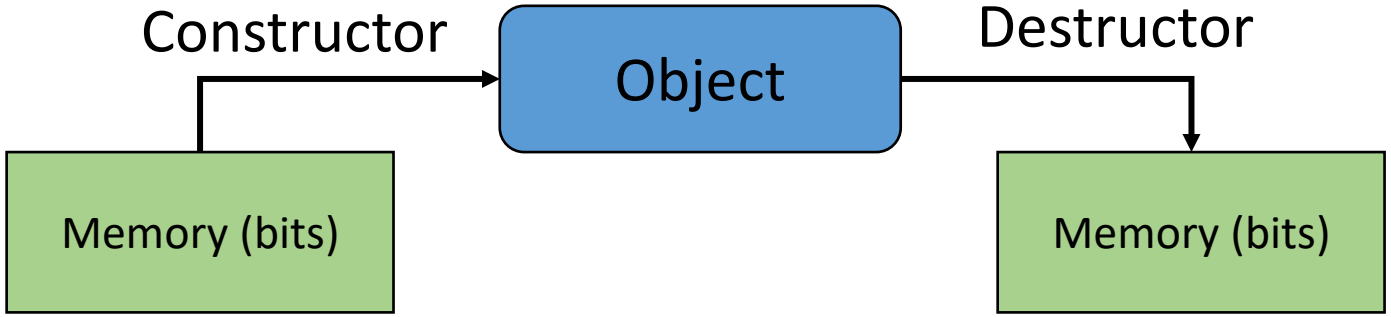
- The class String

```
#include <cstring>

class String {
    char* s;
    int sz;
public:
    String() : s{ new char[sz = 1] } // "", 1 more space for terminator
    {
        s[0] = '\0';
    }

    String(const char* p) : s{ new char[sz = std::strlen(p) + 1] }
    {
        strcpy(s, p);
    }
    ~String()
    {
        delete [] s;
    }
    // other member functions
};
```

Constructors & destructors



Marshal P. Cline:

- Constructors build objects from dust.
- Constructors turn a pile of arbitrary bits into a living object.
- A destructor gives an object its last rites. Destructors are a "prepare to die" member function.

- Constructor: resource Acquisition
- Constructor: resource Release

Concrete classes: destructors

Object initialization

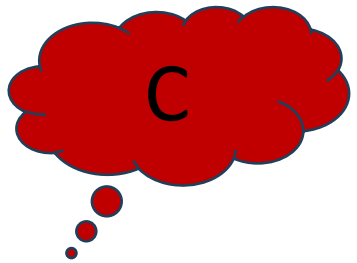
Object initialization

vs.

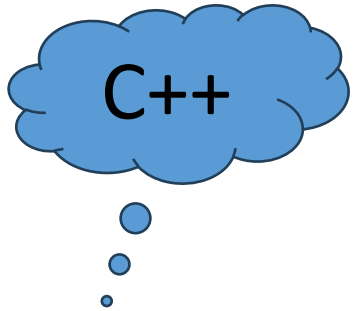
Object initialization

Object initialization

```
struct X {
    // data
};
void init(struct X*, /* init. parameter */);
void clean_up(struct X*);
```

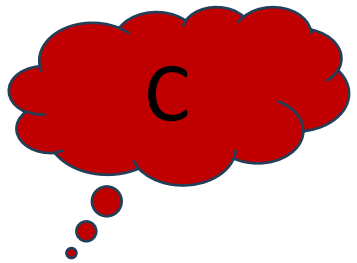


Vs.



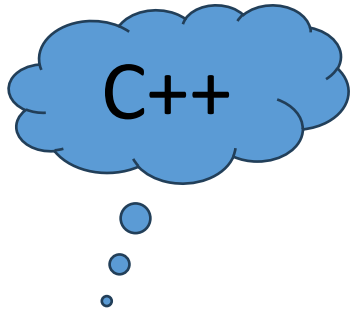
Object initialization

```
struct X {
    // data
};
void init(struct X*, /* init. parameter */);
void clean_up(struct X*);
```



Vs.

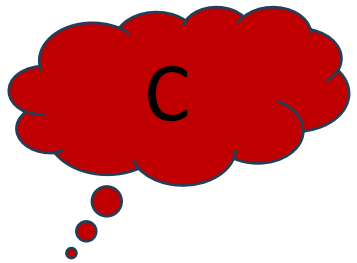
```
class X {
    // data member
public:
    X() { /* … */ }
    ~X() { /* … */ }
    // other member functions
};
```



Object initialization

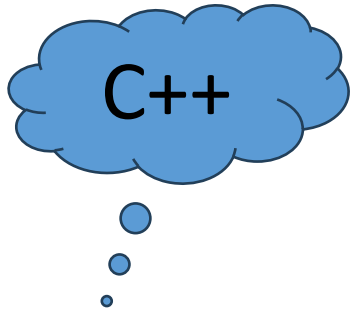
```
struct X {
    // data
};
void init(struct X*, /* init. parameter */);
void clean_up(struct X*);
```

```
{
    struct X x1;
    init(&x1);
    clean_up(&x1);
}
```



Vs.

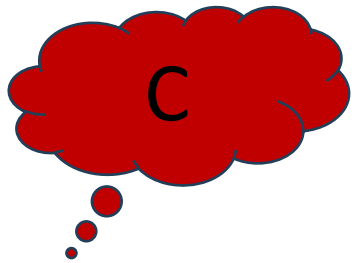
```
class X {
    // data member
public:
    X() { /* … */ }
    ~X() { /* … */ }
    // other member functions
};
```



Object initialization

```
struct X {
    // data
};
void init(struct X*, /* init. parameter */);
void clean_up(struct X*);
```

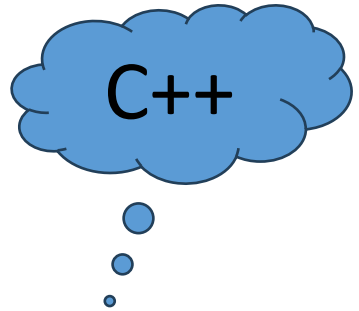
```
{
    struct X x1;
    init(&x1);
    clean_up(&x1);
}
```



vs.

```
class X {
    // data member
public:
    X() { /* … */ }
    ~X() { /* … */ }
    // other member functions
};
```

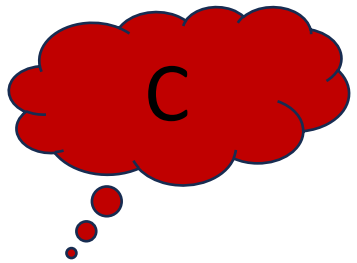
```
{
    X x1;
}
```



Object initialization

```
struct X {
    // data
};
void init(struct X*, /* init. parameter */);
void clean_up(struct X*);
```

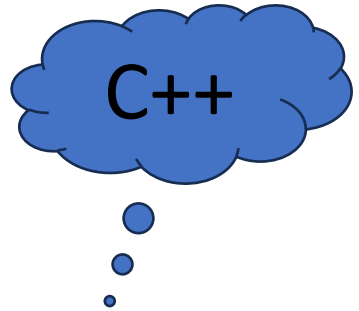
```
{
    struct X x1;
    init(&x1);
    clean_up(&x1);
}
```



vs.

```
class X {
    // data member
public:
    X() { /* … */ }
    ~X() { /* … */ }
    // other member functions
};
```

```
{
    X x1;
}
```




```
void f(int n)
{
    int* p = new int[n];
    vector v(n);
    // … use p and v …
    delete [] p;
}
```

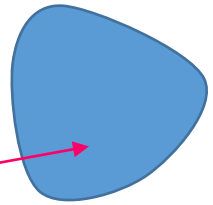
- Don’t put objects on the free store if you don’t have to; prefer scoped variables.
- Avoid “naked new” and “naked delete”.
- Use RAII.
- Use standard-library facilities as a model for flexible, widely usable software.

The this pointer

• In the body of a non-static member function, the keyword this is a prvalue (pure r-value) expression whose value is the address of the object for which the function is called.

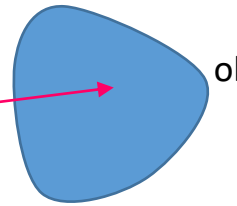
- Constant objects

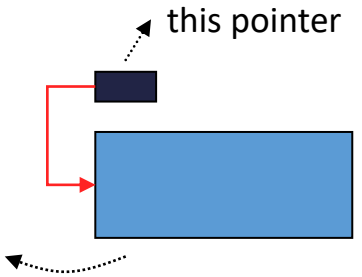
const member functions



- Non-constant objects

Const/Non-const member functions





• A class member function always ‘‘knows’’ for which object it was invoked.

• non-static non-const member function

```
X* const this; // const pointer to
               // non-const data
```

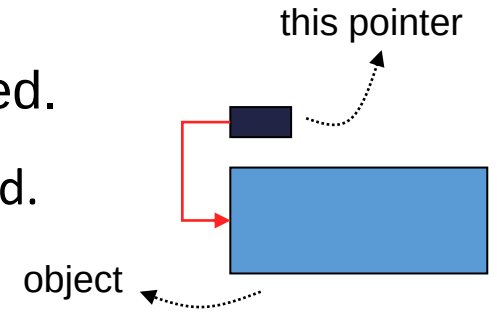
```
int i = 1; int j = 2;
int* const cp = &i;
(*cp)++; // ok
cp = &j; // error
```

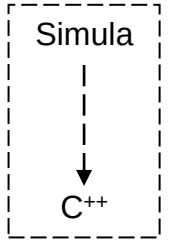
• non-static const member function:

```
const X* const this; // const pointer
                     // to const data
```

```
int i = 1; int j = 2;
const int* p = &i;
(*p)++; // error
p = &j; // ok
```

Self Reference: this pointer





- A class member function always "knows" for which object it was invoked.
- this is a pointer to the object for which a non-static member function is called.
- Chaining of Operations or *Method Chaining*

```
// Date.h
class Date {
public:
  Date(int, int, int);
  Date& AddYear(int); // return itself
  Date& AddMonth(int);
  Date& AddDay(int);
private:
  int year, month, day;
};
```

```
// user.cpp
void g()
{
  Someday d(1385, 7, 30); // Iranian date
  d.AddYear(1).AddMonth(2).AddDay(3);
}
```

d

```
// Date.cpp
Date::Date(int y, int m, int d) :
   year(y), month(m), day(d) {}
Date& AddYear(int y)
{
  year += y;
  return *this;}
Date& Date::AddMonth(int m)
{
  if (month + m > 12) {
    // ...
  }
  else {
    month += m;
  }
  return *this;
}
Date& Date::AddDay(int d)
{
  // do the hard part
  return *this;
}
```

Static members

Static member
- Static data member
- Static member function

- A variable that is part of a class, yet is not part of an object of that class, is called a static data member.
- There is exactly one copy of a static member instead of one copy per object, as for ordinary *non-static* members.

```
class Date {
  int Day, Month, Year;
  static Date default_date;
public :
  Date(int dd =0, int mm =0, int yy =0);
  // ...
  static void set_default(int, int , int);
};
Date::Date(int dd, int mm, int yy)
{
  Day = dd ? dd : default_date.d;
  Month = mm ? mm : default_date.m;
  Year = yy ? yy : default_date.y;
  // check that the Date is valid
}
```

Static members cont.

- Static members – both function and data members – must be defined.

```
Date Date::default_date(1 ,1 ,1970); // UNIX epoch time
void Date::set_default(int d, int m, int y)
{
  Date::default_date = Date(d ,m ,y);
}
```

- All access control rules are applied to static members.
- Counting objects of a class: another usage of static members

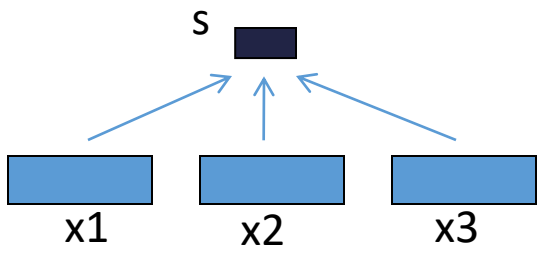
```
// X.h
class X {
public:
  X();
  static int get_count();
private:
  static int obj_count;
};
```

```
// X.cpp
int X::obj_count = 0;
X::X() { obj_count++; }
int X::get_count() { return obj_count;
}
```

```
// Main.cpp
void f()
{
  X x, y, z; // X::ObjCount == 3;
}
```

Static data member representation

- A static data member is not part of the subobjects of a class.




```
struct X {
  static int s;
  void f();
} x1, x2, x3;
```

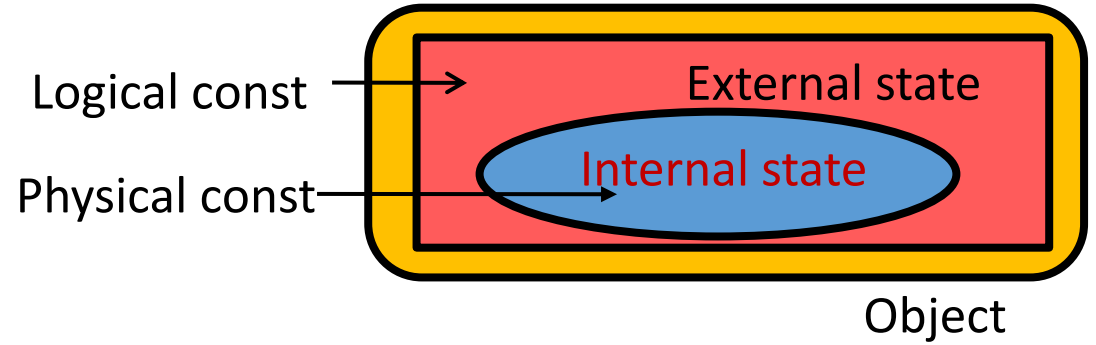
- A static data member in a class is almost equivalent to a global variable. Reference to a static data member is translated into a reference to a global variable (usually maintained in the data segment of the process) with decorated name.
- Static data members are initialized and destroyed exactly like non-local variables.
- Once the static data member has been defined, it exists even if no objects of its class have been created.
- A static data member shall not be *mutable*.

Static member function representation

- Call to static member function is translated as a call to global function. with decorated name.
- A static member function does not have a this pointer.
- A static member function can’t be const.

Fibonacci number: static data members

Physical and Logical constness: mutable



- Logical vs. Physical constness
- Sometimes, a member function is logically *const,* but it still needs to change the value of a member.
- Internal caching, Lazy evaluation, Efficient computation, mutexes, memo caches, lazy evaluation, and access instrumentation…
- Internal caching/Lazy evaluation

```
class C {
    int a, b;
    mutable int s;
    mutable bool cache_valid; // is sum valid?
public:
    C(int a_, int b_) : a(a_), b(b_), cache_valid(false) {}
    int get_a() const { return a; }
    int get_b() const { return b; }
    int sum() const
    {
        if (!cache_valid) s = a + b; cache_valid = true;
    }

    C& set_a(int a_) { a = a_; cache_valid = false; return *this; }
    C& set_b(int b_) { b = b_; cache_valid = false; return *this; }
};
```

- The mutable specifier on a class data member nullifies a const specifier applied to the containing class object and permits modification of the mutable class member even though the rest of the object is const.

Mutable: more examples

- mutexes

```
class Y { // very simple
    int data;
    mutable std::mutex m;
public:
    int get() const
    {
        std::lock_guard<std::mutex> lock_a(m);
        return data;
    }
    void incr()
    {
        std::lock_guard<std::mutex> lock_a(m);
        ++data;
    }
public:
    Y(int d): data{d} {}
};
```

Constructors and Conversion

- Constructor as implicit conversion operator: By default, a constructor invoked by a single argument acts as an implicit conversion from its argument type to its type.

1

```
class Complex {
public: // ctors
  Complex(double r, double i) { re = r; im = i; } // real and imaginary
  Complex(double r) { re = r; im = 0.0; } // real numbers are complex
  Complex() { re = im = 0.0; } // default ctor
private: // representation
  double re, im;
};
```

2

```
void use_cmplx(double r, double i)
{
    Complex c1{r}; // real number: explicit calling
    Complex c2 = r; // OK: real number: implicit conversion
}
```

3

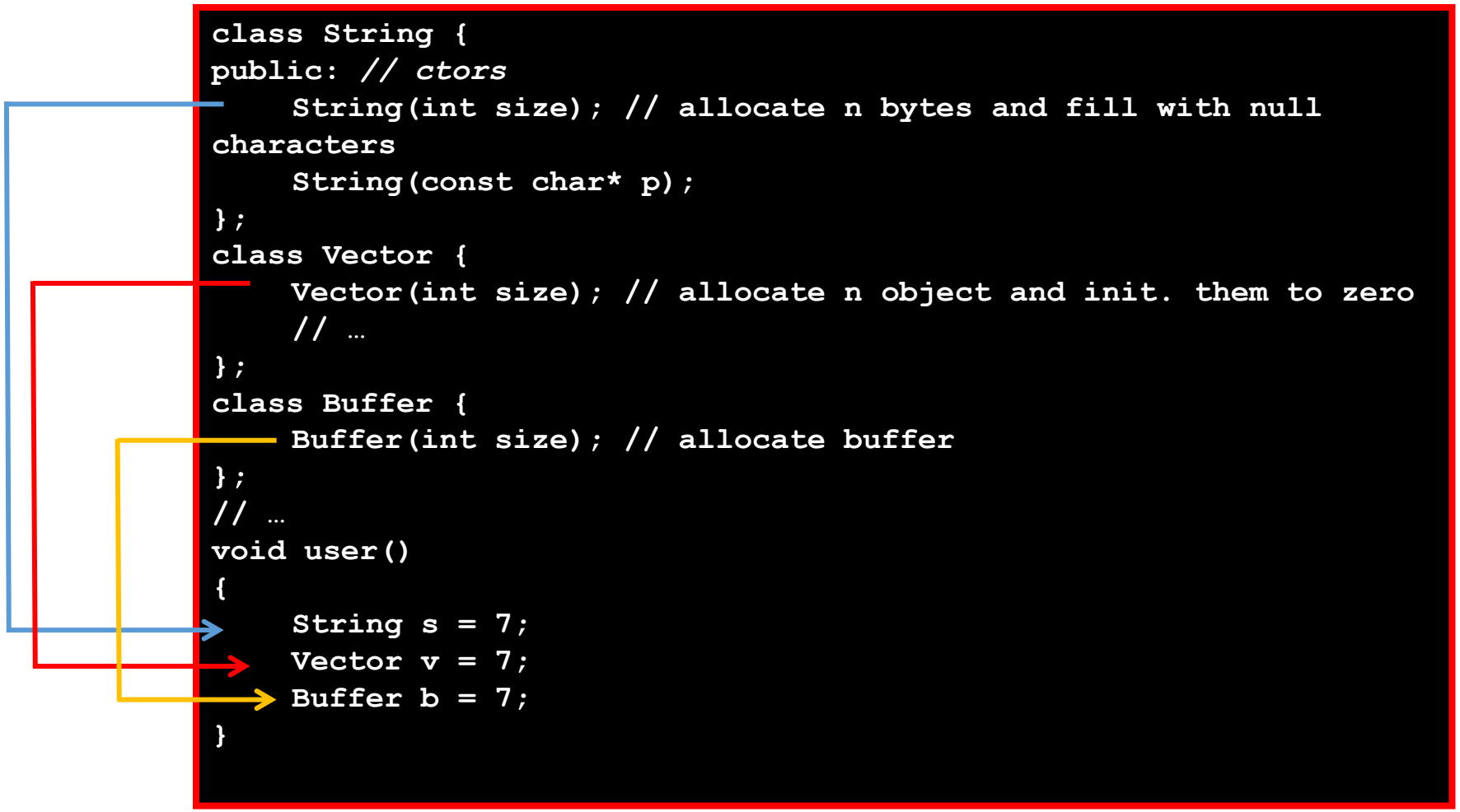
```
class Date {
public: // ctors
  Date(int = 0, int = 0, int = 0);
private: // representation
  int d, m, y;
};
```

4

```
enum WEEKDAY { Monday, Tuesday, … Sunday };
WEEKDAY week_day(const Date&);
// …

WEEKDAY wd = week_day(15); // obscure and isn't cute
```

Constructors and Conversion: more examples



```
class String {
public: // ctors
    String(int size); // allocate n bytes and fill with null
characters
    String(const char* p);
};
class Vector {
    Vector(int size); // allocate n object and init. them to zero
    // …
};
class Buffer {
    Buffer(int size); // allocate buffer
};
// …
void user()
{
    String s = 7;
    Vector v = 7;
    Buffer b = 7;
}
```

The Explicit constructor cont.

- Implicit conversion vs. Explicit construction
- explicit keyword (proposed by Nathan Myers- 1995)

5

```
class Date {
public: // ctors
 explicit Date(int = 0, int = 0, int = 0);
private: // representation
  int d, m, y;
};
```

6

```
Date d1{15}; // OK: considered explicit
Date d2 = Date{15}; // OK: explicit
Date d3 = {15}; // error: implicit conversion is not allowed
Date d4 = 15; // error: implicit conversion is not allowed
WEEKDAY wd = week_day(15); // error: implicit conversion is not allowed
WEEKDAY wd2 = week_day({15}); // error: implicit conversion is not allowed
WEEKDAY wd3 = week_day(Date{15}); // error: implicit conversion is not allowed
```

initialization with =

- Copy initialization vs. Direct initialization

initialization without =

- {} initialization

```
Date d1 = 15; // copy init.
Date d2 = {15}; // copy init.
Date d3 {15}; // direct init.
Date d4 15; // nonsense: syntax error
```

- By default, declare a constructor that can be called with single argument explicit. You need a good reason not to do so.

Friends

- A friend of a class is a function or class that is given permission to name the private and protected members of the class. A class specifies its friends, if any, by way of friend declarations.

  -- C++ standardization committee draft, 11.8.4 **[class.friend]**

- Friends can be functions, other classes, or individual member functions of other classes.

- Friend classes are used when two or more classes are designed to work together and need access to each other's implementation in ways that the rest of the world shouldn't be allowed to have.

```
class X {
    int a;
    friend void friend_set(X*, int);
public:
    void member_set(int);
};
void friend_set(X* p, int i) { p->a = i; }
void X::member_set(int i) { a = i; }
```

- Examples:

- Database implementation: class `Database` and class `DatabaseCursor`
- Download manager: `Downloader` and `DownloadManager`
- Linear algebra library: `Vector` and `Matrix`

Friends cont.

```
class Fraction {
public:
    Fraction(int num = 0, int denom = 1);
    friend Fraction square(const Fraction& n);
private:
    int num_, denom_;
};
```

```
Fraction::Fraction(int num, int denom):
    num_{num},
    denom_{denom}
{}
Fraction square(const Fraction& n)
{
    return Fraction(n.num_* n.num_, n.denom_ * n.denom_);
}
```

Q Do friends violate encapsulation?

A No! If they're used properly, they actually *enhance* encapsulation. … If you use friends like just described, you'll keep private things private.

- Public data, the get/set interface
- Friends and I/O stream overloaded operators
- Friendship isn't inherited, transitive, reciprocal, virtual.

Thanks for your patience …

A man who asks a question is a fool for minute,
The man who does not ask, is a fool for a life.
- Confucius

Learning to ask the right (often hard) questions is an essential part of learning to think as a programmer.
- Bjarne Stroustrup *programming Principles and Practice Using C++, page 4.*

There is no stupid question, but there is stupid answer.
- Howard Hinnant