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

آلفا ALPHA

الماس فناوری ابری پاسارگاد- آلفا

مدرس: سعید امراللهی بیوکی

Agenda 12/24

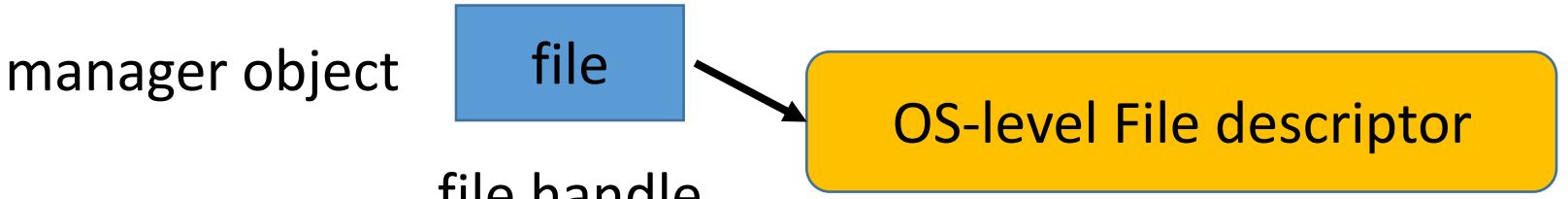
Session 12. More on RAII, Object construction and destruction, Concrete objects, and Three-way comparison operators

- Constructors and class invariants
- C++98 Classes essential operations: Construction, Copy operations, and Destruction
- Law of big three
- The C++11 R-value references and move semantics
- Move operations: Move constructor and Move assignment operators
- Extending Classes essential operations: Construction, Copy and Move operations, and Destruction
- Law of big five
- Construction, Destruction, and Resource Acquisition Is Initialization
- Q&A

150 min (incl. Q & A)

Handle

- Handle-to-data model
- a.k.a Handle-Body Idiom
- Manager object: Well-behaved abstraction



2

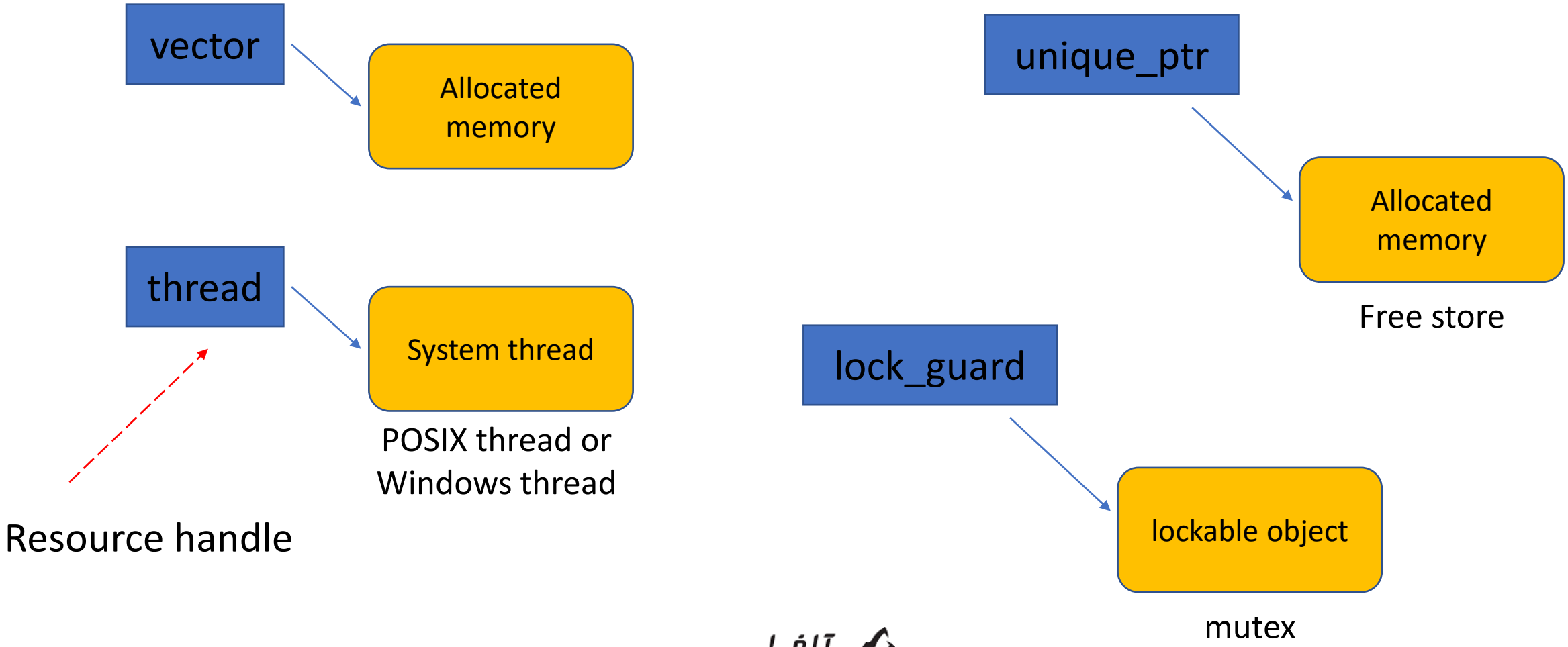
```
void example_usage() {
    // open file (acquire resource)
    file logfile("logfile.txt");
    logfile.write("hello logfile!");
    // continue using logfile ...
    // throw exceptions or return without
    //  worrying about closing the log;
    // it is closed automatically when
    // logfile goes out of scope
}
```

1

```
#include <cstdio>
#include <stdexcept> // std::runtime_error
class file { // a file descriptor wrapper
public:
    file(const char* filename):
        file_(std::fopen(filename, "w+")) {
        if (!file_) {
                throw std::runtime_error("file open failure");
        }
    }
    ~file() {
        if (std::fclose(file_)) {
            // failed to flush latest changes.
        // handle it
        }
    }
    void write(const char* str) {
        if (EOF == std::fputs(str, file_)) {
                throw std::runtime_error("file write failure");
        }
    }
private:
    std::FILE* file_;
};
```

Handle

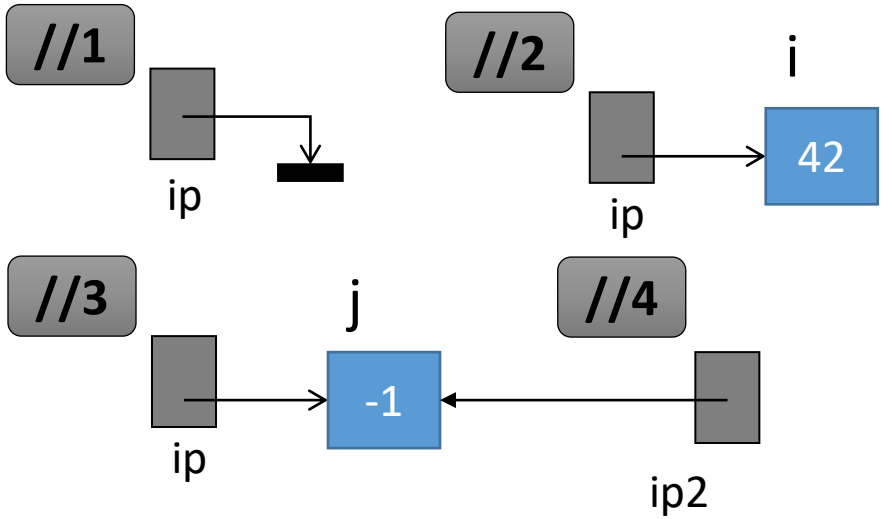
- Resource handle
- Handle-to-data model
- Resource: memory, file, thread, …



C-style pointers- basic concepts

- A pointer points to an object (or not).

```
int* f()
{
    int i = 42;
    int* ip = nullptr; // 1: p points to nothing
    ip = &i; // 2: p holds the address of i
    int j = -1;
    ip = &j; // 3: ip points to j
    int ip2 = ip; // 4: ip2 points to j
    /// long long* lp = ip; // error: type mismatch
    return ip; // return the copy of ip
}
```



- & is address-of operator.
- * is dereference operator.

- These pointers have *value semantics*. An object with value semantics is an object that you can *copy* and *assign to.*
- int and std::string are perfect examples of types with value semantics.
- Pointer “knows” the type of object that points to!
- Pointer does not “know” how many object points to!

```
int i = 0;
int j = i; // copy
j = i; // assignment
string s = "A string …";
string t = s; // another string
string u;
u = t; // copy assignment op.
```

C-style pointers problems

1. Pointers that allocate memory using new operator don't have the *value semantic.*

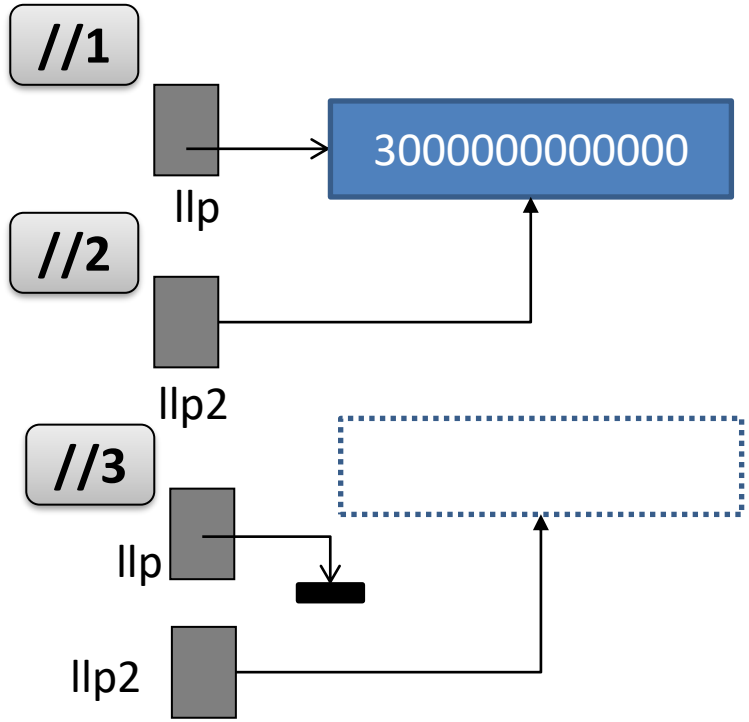
```
char* p = "No value semantics";
char* q = p; // p and q point to single data
long long* llp = new long long(3'000'000'000'000); // 1
long long* llp2 = llp; // 2
delete llp; // 3
delete llp2; // double deletion catastrophic
```

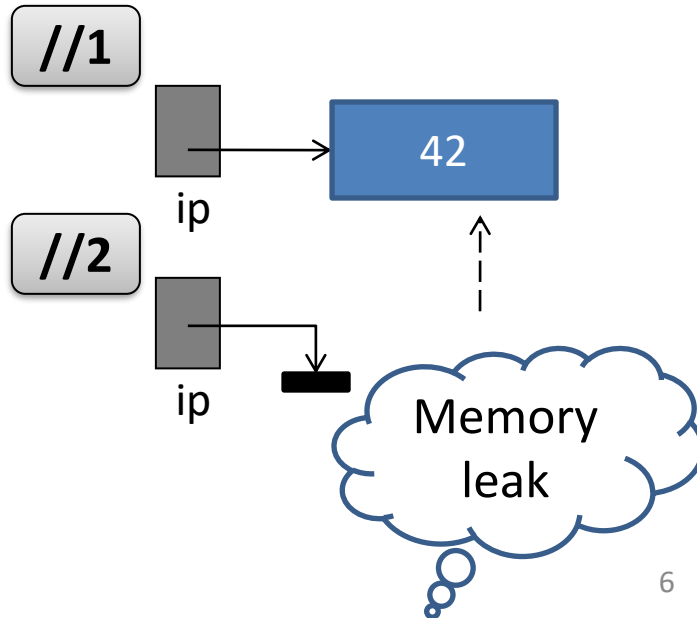
- Shallow copy vs. Deep copy

2. Pointers don't have *ownership management.*

```
int* ip = new int(42); // 1: ip owns the allocated memory
// …
ip = nullptr; // 2: assign something else to p: lose ownership
```

- the variable ip not only points to, but also *owns, the memory allocated for the integer object.*





C-style pointers problems cont.

3. The problem about “automatic” *memory management.*

- new and delete
- new [] and delete []

```
class MyClass {
    // …
};

void f()
{
    MyClass* p = new MyClass(); // memory allocation
    MyClass* pa = new MyClass[10];
    // …
    delete p; // memory release
    delete [] pa;
}
```

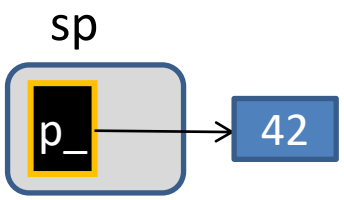
4. No exception safety

- Exception-safety: the notion that a program is structured so that throwing an exception doesn't cause unintended side effects.
- typical side-effect: Memory leak
- RAII: Resource Acquisition is Initialization

```
X* f()
{
  X* p = new X;
  // do something
  // maybe throw an exception: memory leak
  return p;
}
```

C-style pointers problems- summary

- Problems:
  - No value semantics
  - No ownership management
  - Resource leak
  - No exception safety
- Solution: Put a raw pointers in a class and try to mimic their behaviors.

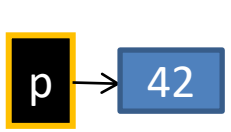


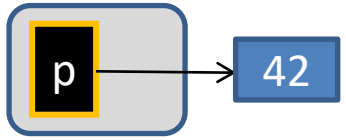

```
template <class T>
class SmartPtr {
public:
     explicit SmartPtr(T* pointee) : pointee_{pointee} {}
     SmartPtr& operator=(const SmartPtr& other);
     ~SmartPtr();
T& operator*() const
{
     return *p_;
}
T* operator->() const
{
     return p_;
}
private:
     T* p_;
};

int main()
{
     SmartPtr sp{new int{42}};
}
```

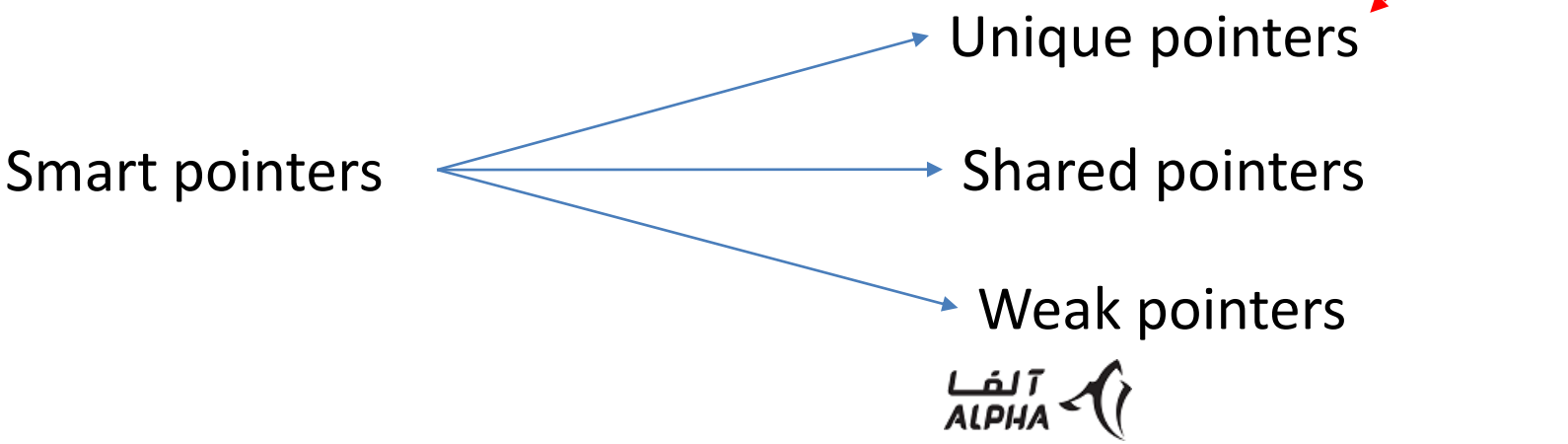
Smart pointers: definitions

Andrei Alexandrescu

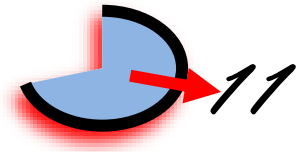
- A smart pointer is a C++ class that mimics a regular pointer in syntax and some semantics, but it does more.
  - A smart pointer is a container for a (raw) pointer.
- raw pointer vs. smart pointer





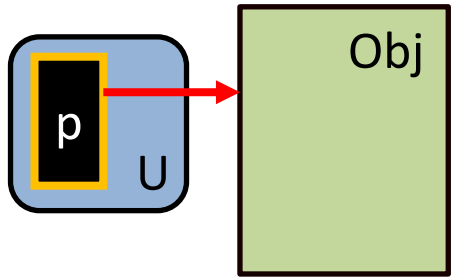


Unique_ptr



- A *unique pointer* is an object that owns another object and manages that other object through a pointer. More precisely, a unique pointer is an object *u* that stores a pointer p to a second object *obj* and will dispose of *p* when *u* is itself destroyed (e.g., when leaving block scope). In this context, *u* is said to *own* obj. **from *Committee Draft***

- A unique_ptr represent *unique ownership*.
- unique ownership a.k.a strict ownership or exclusive ownership
- unique_ptr is the C++11 replacement for auto_ptr.



Unique_ptr- details

- A unique_ptr owns the object to which it holds a pointer.
- A unique_ptr can not be copied, but they can be moved.

```
template<class T> class unique_ptr {
public:
    using element_type = T;
    // constructors
    constexpr unique_ptr(); // default ctor
    explicit unique_ptr(pointer p);
    unique_ptr(unique_ptr&& u); // move ctor
    constexpr unique_ptr(nullptr_t);
    template<class U, class E> unique_ptr(unique_ptr<U, E>&& u); // template ctor
    // destructor
    ~unique_ptr();
    // assignment
    unique_ptr& operator=(unique_ptr&& u); // move assignment
    template<class U, class E>
    unique_ptr& operator=(unique_ptr<U, E>&& u); // conversion move assignment!
    unique_ptr& operator=(nullptr_t); // nullptr assignment
    // disable copy operations
    unique_ptr(const unique_ptr&) = delete;
    unique_ptr& operator=(const unique_ptr&) = delete;
};
```

Unique_ptr vs. C-Style pointers: Performance comparison

```
{
  int* ip = new int{0};
  // …
  delete ip;
}
```

```
#include <memory>
{
  int* up = std::unique_ptr<int>{new int{0}};
  // …
}
```

- The unique_ptr provides a semantics of *exclusive ownership*.

Delegating constructors

- C++98: the common code in several constructors: init member functions

```
#include <stdexcept>
#include <string>
class X {
    int a;
    validate(int x) { if (0 < x && x <= max) a = x; else throw out_of_range("Range error!"); }
public:
    X(int x) { validate(x); }
    X() { validate(42); }
    X(std::string s) { auto temp = std::stoi(s); validate(temp); }
};
```

- C++11: delegating constructors, a.k.a *Forwarding constructor*

```
class X {
    int a;
    validate(int x) { if (0 < x && x <= max) a = x; else throw out_of_range("Range error!"); }
public:
    X(int x) { validate(x); }
    X() { X(42); } // delegating ctor
    X(std::string s) : X{std::stoi(s)} {} // delegating ctor
};
```

Delegating constructors- An example

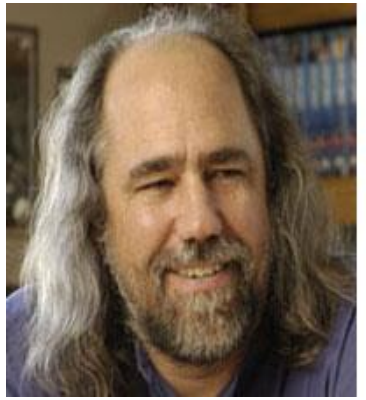
- To avoid code duplicate

```
class FullName {
    string firstName_;
    string middleName_;
    string lastName_;
public:
    FullName(string firstName, string middleName, string lastName);
    FullName(string firstName, string lastName);
    FullName(const FullName& name);
};
FullName::FullName(string firstName, string middleName, string lastName)
    : firstName_(firstName), middleName_(middleName), lastName_(lastName)
    {
    }
FullName::FullName(const FullName& name) // delegating copy constructor
    : FullName(name.firstName_, name.middleName_, name.lastName_)
    {
    }
FullName::FullName(string firstName, string lastName) // delegating constructor
    : FullName(firstName, "", lastName)
    {
    }
```

Delegating constructors and exception safety

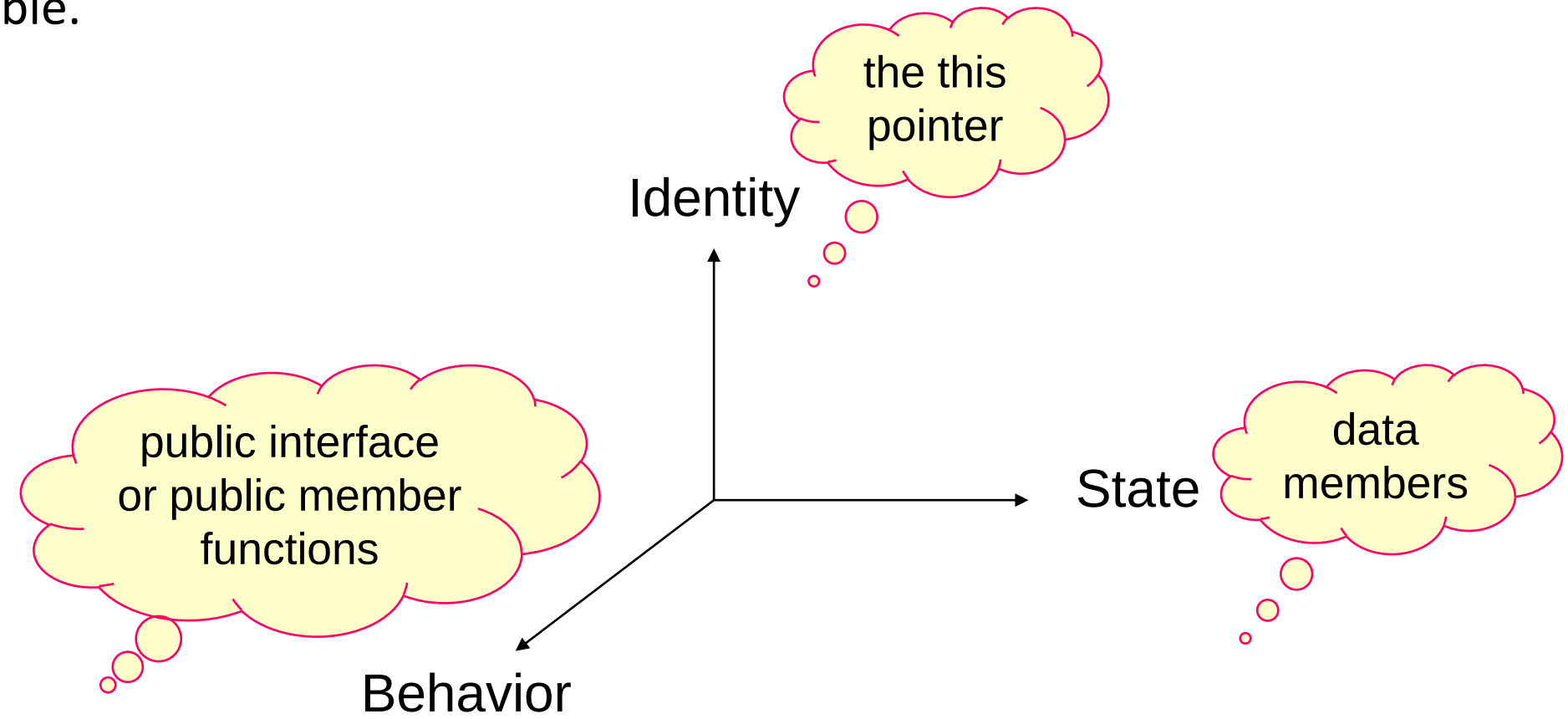
- An object is not considered constructed until its constructor completes.

Object



Grady Booch:

- An object has state, behavior, and identity; the structure and behavior of similar objects are defined in their common class, the term *instance* and object are interchangeable.



Object construction and destruction

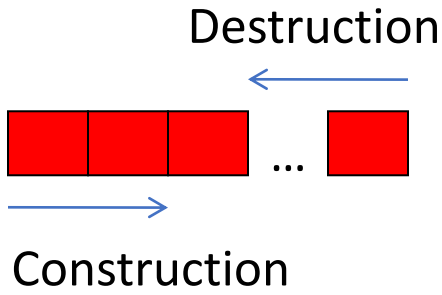
- Local objects

Destructors for local variables are executed in reverse order of their construction. *First constructed, last destructed*.

```
void userCode()
{
    Fred a;
    Fred b;
    // ...
}
```

- Array of objects

Default constructor is called.




```
class X {
  X() = delete;
};

X a[100]; // error
```

- Class objects as members

Recursive objects

Construction:
- The members' constructors (in the order of declaration)
- The class' own constructor

Destruction:
- The class' own destructor
- The members' destructors (in the reverse order)

```
class C {
  C1 m1;
  C2 m2;
  // …
};
```

Object construction and destruction

Given this program:

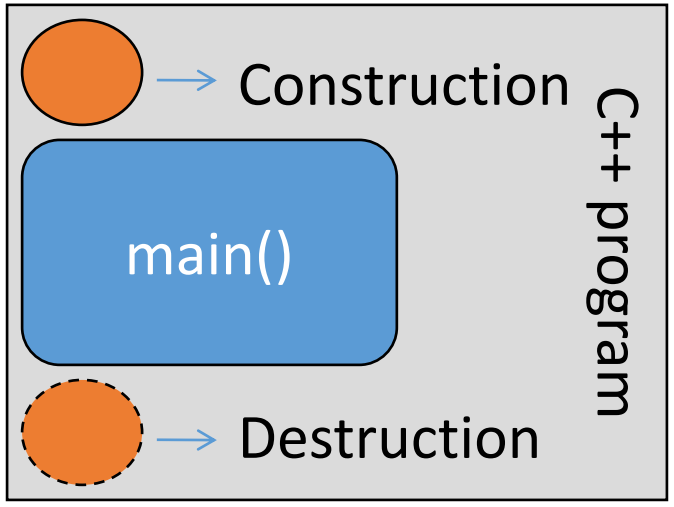
```
#include <iostream>
int main()
{
  std::cout << "Hello, world!\n";
}
```

modify it to produce this output:

Initialize
Hello, world!
Cleanup

Do not change *main() in any way.*

Adapted from Bjarne Stroustrup. The C++ Programming Language. Addison-Wesley, Reading, MA, 2000. 3rd edition. section 10.6 exercise 15.



C++ object construction rules:

- A global, namespace, or class static object, is created once "at the start of the program" and destroyed once at the termination of the program.
- Non-local objects (that is defined outside any function like global, namespace, and class static variables) is initialized (constructed) before main() is invoked, and any such variable that has been constructed will have its destructor invoked after exit from main().

Object construction and destruction

```
#include <iostream>
struct Wrapper { // simple wrapper
  Wrapper() { std::cout << "Initialize\n"; } // constructor
  ~Wrapper() { std::cout << "Cleanup\n"; } // destructor
} main_wrapper; // main function wrapper
int main()
{
  std::cout << "Hello, world!\n";
}
```

- Cool -10 liners code.

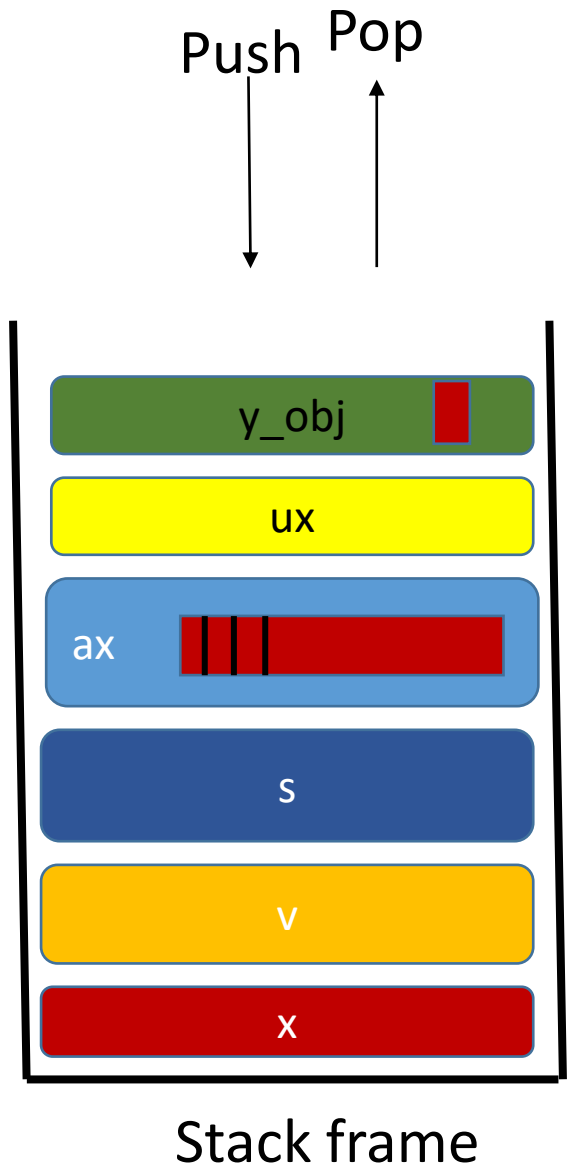
Resource Acquisition and resource release

```
void acquire()
{
     // acquire resource 1
     // acquire resource 2
     // …
     // acquire resource n

     // use resources …

     // release resource n
     // release resource n – 1
     // …
     // release resource 1
}
```

- Resource release in reverse order.

Scope-based resource management

```
class X {
  X() { /* … */ }
  ~X() { /* … */ }
  // …
};

void f()
{
  X x; // constructor
  vector<int> v { 0, 1, 2 };
  string s("Hello, world");
} // calling destructor(s) at the end of scope (in reverse order)

struct Y { // aggregate
  X x;
}
int main()
{
  f();
  {
     array<X, 10> ax;
     unique_ptr<X> ux(new X);
     Y y_obj;
  }
} // calling destructor(s) at the end of scope (in reverse order)
```



Stack frame


RAITest
Prog.

Closing brace

Closing brace

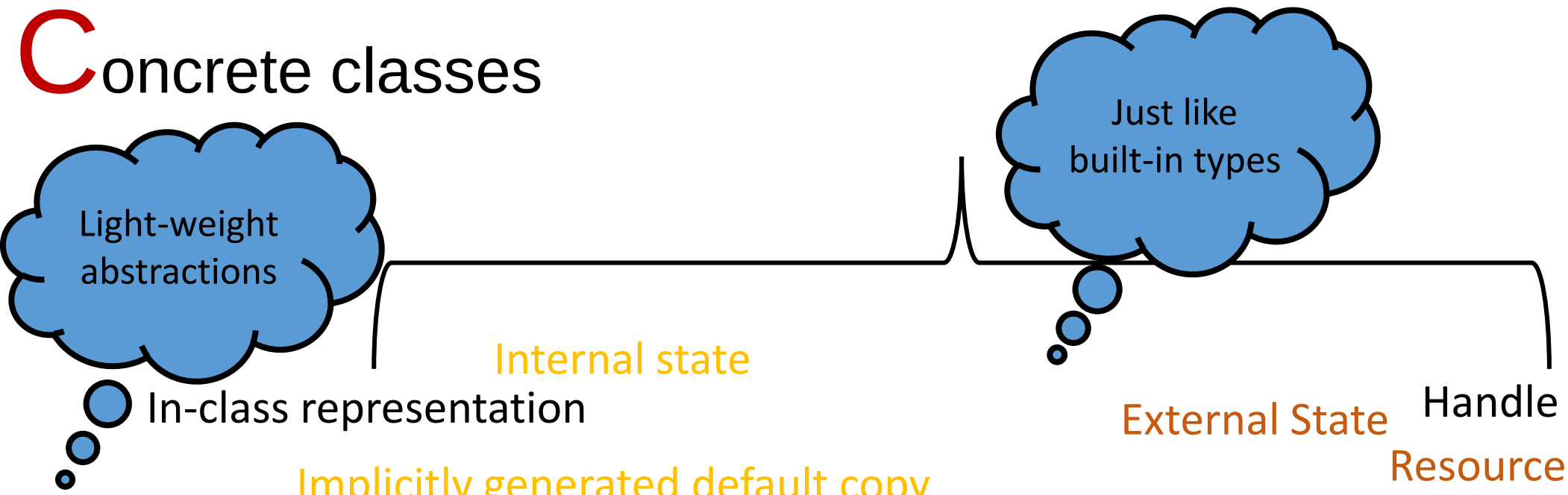
Roger Orr

Closing brace

}

Roger Orr

Concrete classes



- Concrete classes = Value-type classes
- Value-oriented programming

Two-way comparison operator

Operator name	Syntax	Prototype example (for class T)	
		Member function	**Non-member function**
Equal to	a == b	bool operator==(const T&)	bool operator==(const T&, const T&)
Not equal to	a != b	bool operator!=(const T&)	bool operator!=(const T&, const T&)
Less than	a < b	bool operator<(const T&)	bool operator<(const T&, const T&)
Greater than	a > b	bool operator>(const T&)	bool operator>(const T&, const T&)
Less than or equal to	a <= b	bool operator<=(const T&)	bool operator<=(const T&, const T&)
Greater than or equal	a >= b	bool operator>=(const T&)	bool operator>=(const T&, const T&)

MyInt comparison operators

1

```
// my_int.h
struct MyInt {
    int value;

    explicit MyInt(int val) : value{ val } {}

    bool operator<(const MyInt& rhs) const {
        return value < rhs.value;
    }
    bool operator==(const MyInt& rhs) const {
        return value == rhs.value;
    }
    bool operator!=(const MyInt& rhs) const {
        return !(*this == rhs);
    }
    bool operator<=(const MyInt& rhs) const {
        return !(rhs < *this);
    }
    bool operator>(const MyInt& rhs)  const {
        return rhs < *this;
    }
    bool operator>=(const MyInt& rhs) const {
        return !(*this < rhs);
    }
};
```

2

```
#include "my_int.h"
#include <iostream>

using std::cout;
int main()
{
    MyInt int2022{ 2022 };
    MyInt int2023{ 2023 };

    if (int2022 < int2023)
        cout << "Less than ...\n"; // OK
    if (int2022 < 2023) // error: operator mismatch
        cout << "Less than ...\n";

    return 0;
}
```

MyInt Comparison Operators

Prog.

- Comparison operators implementation based on each other.

آلفا ALPHA

the C Strcmp function

- Function prototype

```
int strcmp( const char *lhs, const char *rhs );
```

- Compares two null-terminated byte strings lexicographically. The sign of the result is the sign of the difference between the values of the first pair of characters (both interpreted as unsigned char) that differ in the strings being compared. The behavior is undefined if lhs or rhs are not pointers to null-terminated byte strings.
- Parameters
  - lhs, rhs - pointers to the null-terminated byte strings to compare
- Return value
  - Negative value if lhs appears before rhs in lexicographical order.
  - Zero if lhs and rhs compare equal.
  - Positive value if lhs appears after rhs in lexicographical order.

https://en.cppreference.com/w/c/string/byte/strcmp

StrcmpTest

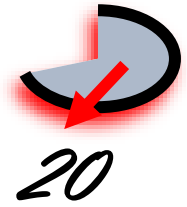
Prog.

- Readability

```
char* p = …
char* q = …
if (!strcmp(p, q)) // it means equality
```

آلفا ALPHA

Three-way comparison operator



- <compare> header
- A.k.a spaceship operator: the Perl programming language

std::strong_reordering

```
struct R {
    // ...
    auto operator<=>(const R& a) const = default;
};
void user(R r1, R r2)
{
    bool b1 = (r1<=>r2) == 0; // r1==r2
    bool b2 = (r1<=>r2) < 0; // r1<r2
    bool b3 = (r1<=>r2) > 0; // r1>r2
    bool b4 = (r1==r2);
    bool b5 = (r1<r2);
}
```

```
struct R2 {
int m;
auto operator<=>(const R2& a) const { return a.m == m ? 0 : a.m < m ? -1 : 1; }
};
```

SpaceshipOperatorForDoubleTest

Prog.

SpaceshipOperator

Prog.


Date5
Prog.

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