Software Languages.Lab

Structuur van Computerprogramma's 2

dr. Dirk Deridder
Dirk.Deridder@vub.ac.be
http://soft.vub.ac.be/

Vrije Universiteit Brussel - Faculty of Science and Bio-Engineering Sciences - Computer Science Department - Pleinlaan 2 - 1050 Brussels - Belgium

Software
Languages.Lab

It is advisable to study book chapters 1-3 upfront

Chapter 3 - User Defined Types

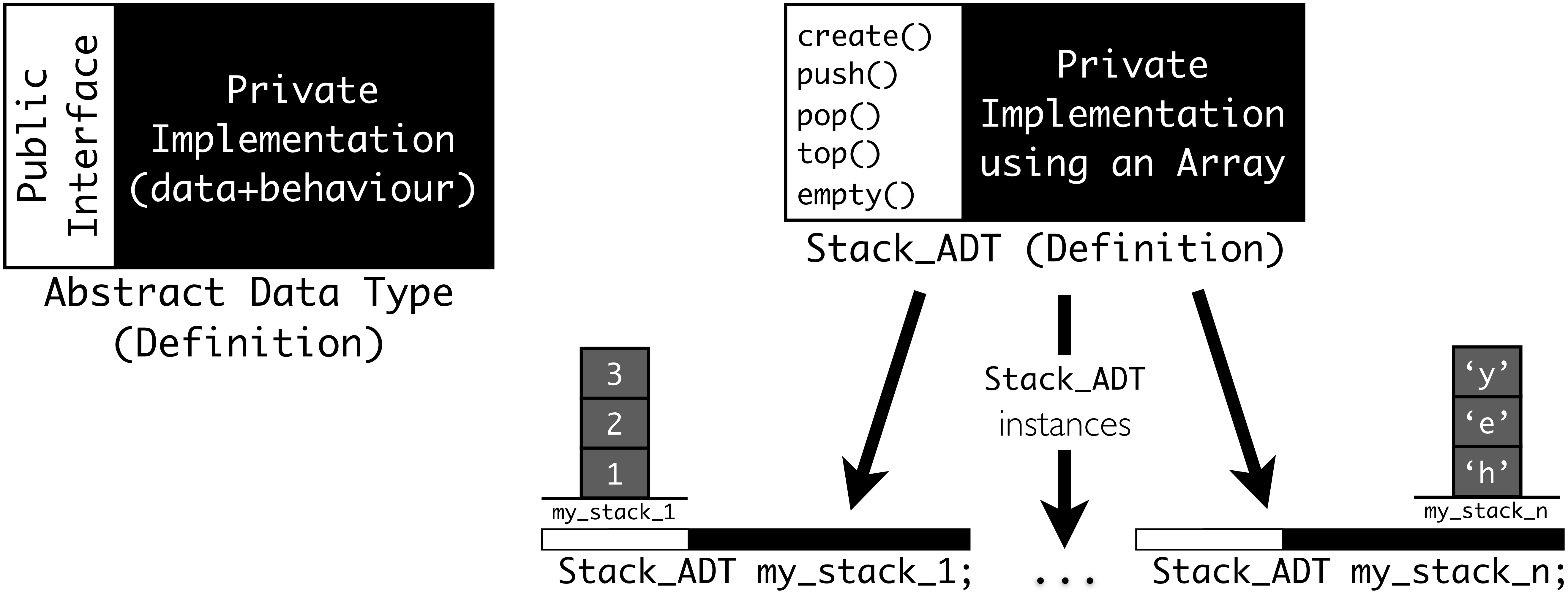
Structuur van Computerprogramma's 2

User Defined Types

Overview of Concepts

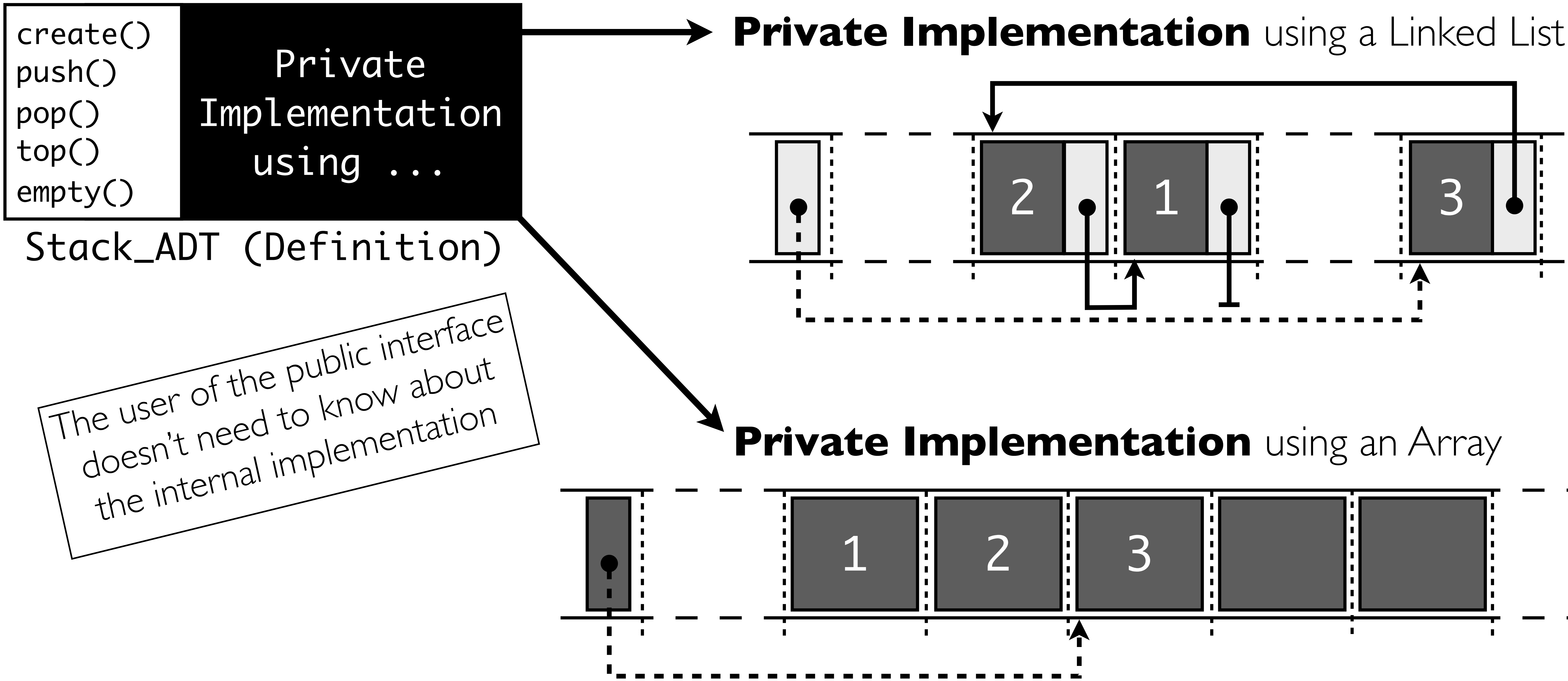
- Abstract Data Type (ADT)
- Public interface, private implementation, access specifiers
- Abstraction, Encapsulation
- Data member declarations, function member declarations
- Function member definitions
- Classes, class objects, Target class object, message sending, methods
- Function member overloading, operator overloading
- Constructors (ctor), Copy Constructors (cctor), Destructors, Object finalization, Operators
- Default ctor, default cctor, default assignment operator
- Memberwise initialization, member list initialization
- Inline member function definition, Member functions with default parameters, User-defined conversions
- Forbidding operations
- Member objects, Member references, Static members, Friends
- Class object life-cycle
- Nested classes
- Enumeration Types, Overriding enumerated type values, typedef,

Abstract Data Types (ADT's) (I)



- enables the definition of new types of data objects with an associated set of special-purpose operations
- have a **public interface** specifying the available operations on the type
- have a **private implementation** that describes
  - **data** : how is information for an object of the ADT represented
  - **behaviour** : how are operations provided by the ADT implemented

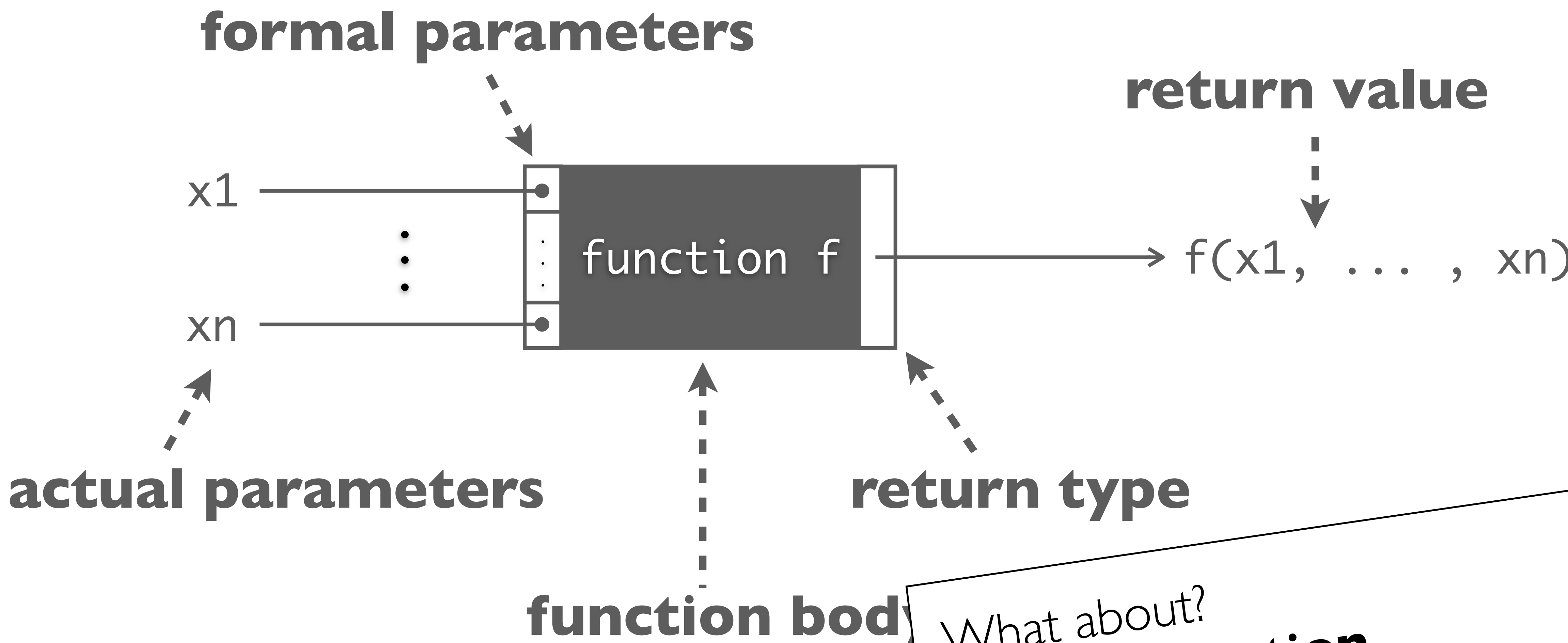
Abstract Data Types (ADT's) (2)



Advantages (similar to functional abstraction):

- **Abstraction**: ADT's can be used as if they were built-in types, ignoring the possibly complex implementation by using a simpler interface
- **Encapsulation**: ADT's shield users from internal implementation changes as long as their interface remains the same

ADT's versus Functions? (remember ...)



What about?

- **Encapsulation**
- **Public interface**
- **Private implementation**
- **Abstraction**

- Basic means for modularising an implementation
  - Structured programming
- Reuse instead of code duplication
  - Decrease code size and eliminate potential duplication of errors: easier to maintain!

Which mechanism(s) do you already know that help in realising these characteristics of ADT's?

Classes are C++ ADT's

- The **interface** is provided by public **function member declarations**
- The **implementation** is provided by
  - **private data member declarations**
  - **function member definitions**

`class` keyword

access specifiers, by default everything is `private` (+)

```
class NameOfClass {
 public:
   MemberDeclarations
 private:
   MemberDeclarations
};
```

start of scope

the definitions are defined outside the class (x)

what happens if you forget this semicolon?

```
../src/demo8.cpp:240: error:
  new types may not be defined in a return type
../src/demo8.cpp:240: note:
  (perhaps a semicolon is missing after the definition of 'Stack')
../src/demo8.cpp:240: error:
  two or more data types in declaration of 'main'
```

Software Languages Lab

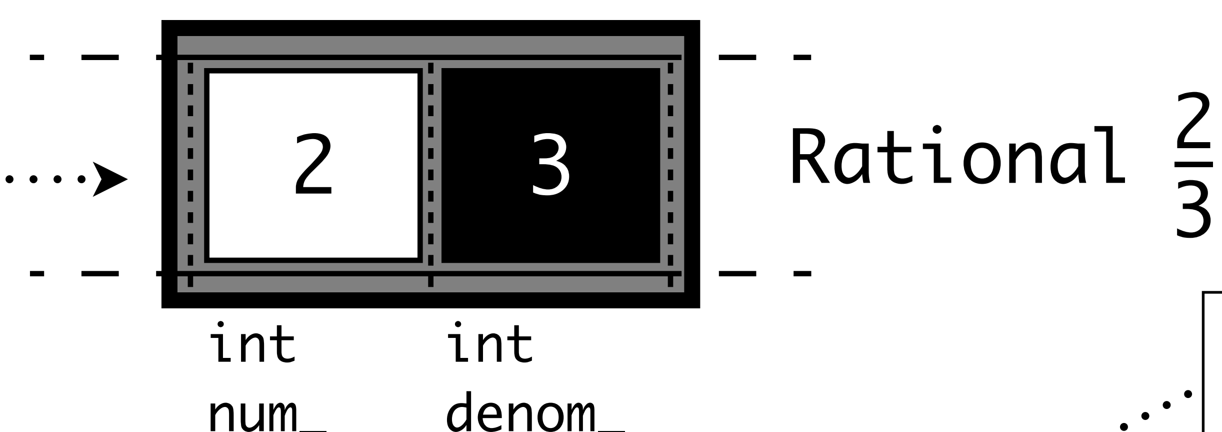
Class Declaration, Class Objects, Data/Function Members

```
#ifndef RATIONAL_H
#define RATIONAL_H

class Rational { // defines a scope called Rational
public:
   // public interface to be supplied
private:
   // implementation part
   int num_;
   int denom_;    // must not be 0!
};

#endif
```

This declaration of the class goes into a header file

private implementation part (data member declarations, member function declarations, ...)

a **class object** (a.k.a. instance)



Similar to using built-in types

```
Rational r; // just another object definition
```

Member Function Declarations

```
#ifndef RATIONAL_H
#define RATIONAL_H

class Rational {
public:
   Rational multiply(Rational r);
   Rational add(Rational r);
private:
   int num_;
   int denom_; // denom_ != 0
};

#endif
```

The actual definition of the member functions goes into a separate .cpp file or inline

public interface (member function declarations)

Calling a member function: specify the **target class object**

```
#include "rational.h"

int main() {
   Rational r, r1, r2;
   r = r1.multiply(r2);
}
```

similar to using built-in types

the '.' calls the multiply function in the context of the `r1` object **(message send)**

Member Functions may be overloaded

```
class Rational {
public:
   Rational multiply(Rational r);
   Rational add(Rational r);
   Rational add(double d);

private:
   int num_;
   int denom_; // denom_ != 0
};
```

Both will have a different implementation (shielded from users)

If you want your newly defined data type to interact with other datatypes then you should specify different function versions

```
Rational r1;
Rational r2;

r1.add(r2);
r1.add(3.3);
```

Koenig Lookup will be used to find best match (no magic !)

```
...
r1.add(5);
r1.add("8/3");
```

The list of overloaded functions can become large (cf. iostream)

Initializing a Class Object with Constructors (ctor)

You are building your own type so you should specify how new objects (values) of this type are created: **Constructors** (a.k.a. **ctor**)

```
class Rational {
public:
   Rational(int num, int denom);
   Rational multiply(Rational r);
   Rational add(Rational r);
private:
   int num_;
   int denom_; // denom_ != 0
};
```

a constructor has the same name as the class(type), no return type is specified!

Constructors are more flexible than "just" giving an initial value

```
Rational r(2, 3); // initialize r using Rational::Rational(2,3)
```

A class acts like a namespace, so we use the scope resolution operator to uniquely identify the functions (name can be reused in a different namespace)

Overloading Constructors

```
class Rational {
public:
    Rational(int num, int denom);
    Rational(int num);
    Rational();
    Rational(const Rational& r);

    Rational multiply(Rational r);
    Rational add(Rational r);
private:
    int num_;
    int denom_; // denom_ != 0
};
```

initialize to **num/denom**

initialize to **num/1**

default ctor; initialize to **0/**

initialize to a copy of **r**
a.k.a **copy constructor**

Calling the constructor is like calling a member function, so can we write:
`Rational r1();`
?

```
Rational r1;      // calls Rational::Rational();
Rational r2(r1); // calls Rational::Rational(const Rational&);
Rational r3(5);  // calls Rational::Rational(int);
Rational r4(5,4);// calls Rational::Rational(int, int);
```

The Default Constructor

```
class Rational {
private:
   int num_;
   int denom_;
};
```

What happens if we don't provide a constructor?

The compiler will not initialize data members of built-in types, but will call constructors for data members of class types

```
class Rational {
public:
   Rational(int num, int denom);
   Rational(const Rational& r);
private:
   int num_;
   int denom_;
};
```

What happens if we provide constructors except a default one?

```
Rational r;
```

Once you declare constructors, the compiler will assume that these are the only valid initialization options (error)

```
../main.cpp: In function 'int main()':
../main.cpp:16: error: no matching function for call to 'Rational::Rational()'
../rationaltest.h:14: note: candidates are: Rational::Rational(int, int)
../rationaltest.h:11: note:                 Rational::Rational(const Rational&)
make: *** [main.o] Error 1
```

The Copy Constructor (cctor) Revisited

A **cctor** is used for passing class objects by value

```
class Rational {
public:
   Rational(int num, int denom);
private:
   int num_;
   int denom_;
};

int f(Rational r) {
...}
```

Calling this function results in passing **r** by value, so its value will be copied to the function call frame. This copying is done by a **copy constructor**

```
Rational x;
f(x);
```

What happens if you call function f?

A **default cctor** is provided by the compiler if you do not explicitly define one. This does a **memberwise initialization** of each data member from the source to the target. If a data member is of a user-defined type, its cctor will be called.

Performance matters! cctor's get called a lot!

Software Languages Lab

Member Function Definitions

```
#include "rational.h"

// a/b * c/d = (a*c)/(b*d)
Rational
Rational::multiply(Rational r) {
   int num(num_ * r.num_);
   int denom(denom_ * r.denom_);

   return Rational(num, denom);
}

// a/b + c/d = (a*d + c*b)/(b*d)
Rational
Rational::add(Rational r) {
   int num = num_ * r.denom_ + r.num_ * denom_;
   int denom = denom_ * r.denom_;

   return Rational(num, denom);
}
```

Define **multiply** in the scope of **Rational** with the scope resolution operator (avoids ambiguous names)

num_ and **denum_** are private to the outside world, but accessible from within the class implementation

value constructor !

These definitions go into a .cpp file

Constructor Definition

Use a **member initialization list** in your ctor definition to initialize data members directly

remember: no return type !

```
Rational::Rational(int num, int denom) : num_(num), denom_(denom) {

   if (denom == 0)
      abort();
}
```

member initialization list after ‘:’

constrain initialization of denominator to non-zero values, simply terminate the program if not satisfied (not clean !)

Alternative definition not using the member initialization list:

```
Rational::Rational(int num, int denom) {
   num_ = num;
   denom_ = denom;
   if (denom == 0)
      abort();
}
```

What about efficiency?

Member Initialization and Constructors

Data members are initialized **before** the ctor body is executed:

1. If no member initialization is specified in the ctor definition then initialize the members using the member type's default ctor.
   Generate a compile error if the member type is a user-defined type (not primitive) and has no default ctor defined.
   (note that primitive types do not have a default ctor)

2. Execute the body of the constructor

To initialize the "whole", first initialize the parts, then do additional work, if any.

`abort()` versus `exit()` (FYI)?

Check the ANSI/ISO/IEC 14882 standard p. 342

Table 18—Header <cstdlib> synopsis

Type	Name(s)		
Macros:	`EXIT_FAILURE`		`EXIT_SUCCESS`
Functions:	`abort`	`atexit`	`exit`

The contents are the same as the Standard C library header `<stdlib.h>`, with the following changes:

```
abort(void)
```

3 The function `abort()` has additional behavior in this International Standard:

— The program is terminated without executing destructors for objects of automatic or static storage duration and without calling the functions passed to `atexit()` (3.6.3).

```
extern "C" int atexit(void (*f)(void))
extern "C++" int atexit(void (*f)(void))
```

4 **Effects:** The `atexit()` functions register the function pointed to by *f*, to be called without arguments at normal program termination.

5 For the execution of a function registered with `atexit()`, if control leaves the function because it provides no handler for a thrown exception, `terminate()` is called (18.6.3.3).

6 **Implementation Limits:** The implementation shall support the registration of at least 32 functions.

7 **Returns:** The `atexit()` function returns zero if the registration succeeds, nozero if it fails.

```
exit(int status)
```

8 The function `exit()` has additional behavior in this International Standard:

— First, objects with static storage duration are destroyed and functions registered by calling `atexit` are called. Non-local objects with static storage duration are destroyed in the reverse order of the completion of their constructor. (Automatic objects are not destroyed as a result of calling `exit()`.)[207] Functions registered with `atexit` are called in the reverse order of their registration, except that a function is called after any previously registered functions that had already been called at the time it was registered.[208] A function registered with `atexit` before a non-local object `obj1` of static storage duration is initialized will not be called until `obj1`'s destruction has completed. A function registered with `atexit` after a non-local object `obj2` of static storage duration is initialized will be called before `obj2`'s destruction starts. A local static object `obj3` is destroyed at the same time it would be if a function calling the `obj3` destructor were registered with `atexit` at the completion of the `obj3` constructor.

— Next, all open C streams (as mediated by the function signatures declared in `<cstdio>`) with unwrit-

Short (incomplete) Explanation:

`abort()` *immediately terminates the program (e.g. no destructors called, no additional housekeeping functions called).*

`exit()` *terminates the program but allows you to specify a 'clean' way to terminate the program using the* `atexit()` *function (also destructors are called)*

Use **exception handling** instead (later)

Inline Member Function Definition

```
#ifndef RATIONAL_H
#define RATIONAL_H
class Rational {
public:
    // interface
    Rational(int num, int denom) :
        num_(num), denom_(denom) {
        if (denom == 0)
            abort();
    }

    Rational multiply(Rational r) {
        return Rational(num_ * r.num_, denom_ * r.denom_);
    }

    Rational add(Rational r) {
        return Rational(num_ * r.denom_ + r.num_ * denom_, denom_ * r.denom_);
    }

private:
    // implementation part
    int num_;
    int denom_; // denom_ != 0
};
#endif
```

No **`inline`** keyword needed, simply define it within the class scope

Other option is to define the function outside the class scope (similar to standard definitions) using the **`inline`** keyword (in the header file !)

Member Functions with Default Parameters

```cpp
#ifndef RATIONAL_H
#define RATIONAL_H

class Rational {
public:
   Rational(int num = 0, int denom = 1);

   Rational multiply(Rational r);
   Rational add(Rational r);

private:
   int num_;
   int denom_; // denom_ != 0
};

#endif
```

What's a free benefit of using default parameters here?

Using default parameters saves 2 overloaded ctor functions namely:
`Rational(int num);`
and
`Rational();`

User-Defined Conversions

C++ provides automatic conversion of built-in types (not for classes)

a) Use ctor's as conversion functions:

```
Rational r;
r.multiply(2); // Rational tmp(2,1); r.multiply(tmp);
```

To prevent the compiler using a ctor for this purpose, prefix the ctor with the **`explicit`** keyword

b) Define specific conversion member functions

```
class NameOfClass { ...
    operator TypeName();
};
```

```
class Rational {
public:
...
operator double() { return double(num_)/denom_; }
..
};
Rational r(1, 3);

std::cout << r;
```

double

double

int int

Why are specific conversion functions needed?

`operator<<(ostream&,Rational)` is not defined, but it will print `0.333` because of the conversion function:

```
    double tmp(r.operator double());
    operator<<(cout,tmp);
```

Operator Overloading

```
#ifndef RATIONAL_H
#define RATIONAL_H
class Rational {
public:
   Rational(int num = 0, int denom = 1);
   Rational operator+(Rational r) { return add(r); }
   Rational operator*(Rational r) { return multiply(r); }
   Rational multiply(Rational r);
   Rational add(Rational r);

private:
   int num_;
   int denom_;
};
#endif
```

Take care of the arity of the operators !
(one parameter less than with normal operator function since the target object is known)

```
Rational r1(1, 2);
Rational r2(3, 4);
Rational r3(5, 6);

r1+r2*r3;
```

```
r1.add(r2.multiply(r3));
```

Operator precedence still holds !

Overloading by Non-member Functions

Problem:

```
Rational r;
r+2;
2+r;
```

First operand is class object so operator member functions are considered for resolving the call:
`Rational tmp(2); r.operator+(tmp);`

Compile error ! First operand is no class object so set of candidate functions is not extended:
`no function operator+(int, Rational)`

Solution: Overload the operator as an ordinary function (see earlier)

```
inline Rational operator+(Rational r1, Rational r2) {
   return r1.add(r2);
}
```

Note: defined outside the class !
Otherwise it is considered as a member function.

As a result the normal conversion strategy works:

```
2+r; // Rational tmp(2); operator+(tmp, r);
```

`operator+(int, int)` and `operator+(Rational, Rational)` are now considered for resolving the call

Software Languages Lab

Operators that can be overloaded

`[]`	`()`	`++`	`--`	`~`	`!`	`-`	`+`
`*`	`new`	`delete`	`delete []`	`new []`	`/`	`%`	`,`
`->*`	`<<`	`>>`	`<`	`<=`	`>`	`>=`	`==`
`!=`	`&`	`^`	`|`	`&&`	`||`	`=`	`*=`
`/=`	`%=`	`+=`	`-=`	`<<=`	`>>=`	`&=`	`|=`
`->`	`^=`						

Remarks:

- `=`, `[]`, `()` and `->` (member selection) must be defined as non-static member functions to ensure that the first argument is an lvalue.
- can only be overloaded in the case that there is at least one operand of a user-defined type
- there are few restrictions on the semantics of the overloaded definition
  - e.g., normally `++a` is the same as `a += 1` but this need not be true for a user-defined operator

Overloading the Assignment Operator =

```
#include <math.h>

class Rational {
public:
  Rational(int num = 0, int denom = 1);
...
  Rational& operator=(double d) {
      int units(rint(d)); // rint(double) rounds to nearest int
      int hundreds(rint((d - units) * 100));
      num_ = units * 100 + hundreds;
      denom_ = 100;
      return *this;
  }

private:
...
};

Rational r;
r = 1.3; // sets r to 130/100
```

`*this` is a reference to the target object. It is returned here to conform to the standard semantics of assignments (see earlier). More about `*this` later...

The Default Assignment Operator

- For a class `C`, a **default assignment operator**
  `C& operator=(const C&)`
  is **always available**, even if it was not defined.
- The default assignment operator performs a **member-wise assignment** of the second operand to the first operand (bitwise copy).
- During the memberwise assignment: if a data member is of a type that has a user-defined assignment operator, then that one will be used instead of the bitwise copy

```
Rational r1(1, 2);
Rational r2;
r2 = r1 * 3; // r2 = 3/2
```

What's the difference with a copy constructor?

Overloading the ++ and -- Operators

Remember: the parameter list is different from ordinary functions (one less argument, target object is known)

```
class Rational {
public:
   ...
   Rational operator++() {                 // prefix version, e.g. ++r
      Rational r(num_+denom_, denom_);     // ++(3/5) = 3/5 + 5/5
      num_ += denom_;                      // update internal state
      return r;                            // return the incremented r
   }

   Rational operator++(int) {              // postfix version, e.g. r++
      Rational r(num_, denom_);            // remember the original r
      num_ += denom_;                      // update internal state
      return r;                            // return the original r
   }
private:
...
};

Rational r(1, 2);
r1 = ++r;                                   // r1 = r = 3/2
r2 = r++;                                   // r2 = 3/2, r = 4/2
```

Forbidding Operators

```
class Server {
public:
   Server(std::ostream& log, int port);
...

private:
   std::ostream& log;
   // we forbid making copies of a Server object by
   // declaring the copy constructor and assignment
   // operator to be private (no definition is
   // needed, nobody can call them)
   Server(const Server&);
   Server& operator=(const Server&);
...
};
...
void start_protocol_bad(Server s); // calling it gives error: why?
void start_protocol_ok(Server& s); // ok
```

Sometimes you want to disable certain operators so that no one can use them on an object (e.g. cloning a server object)

copy constructor

assignment operator

Finalizing Objects using Destructors

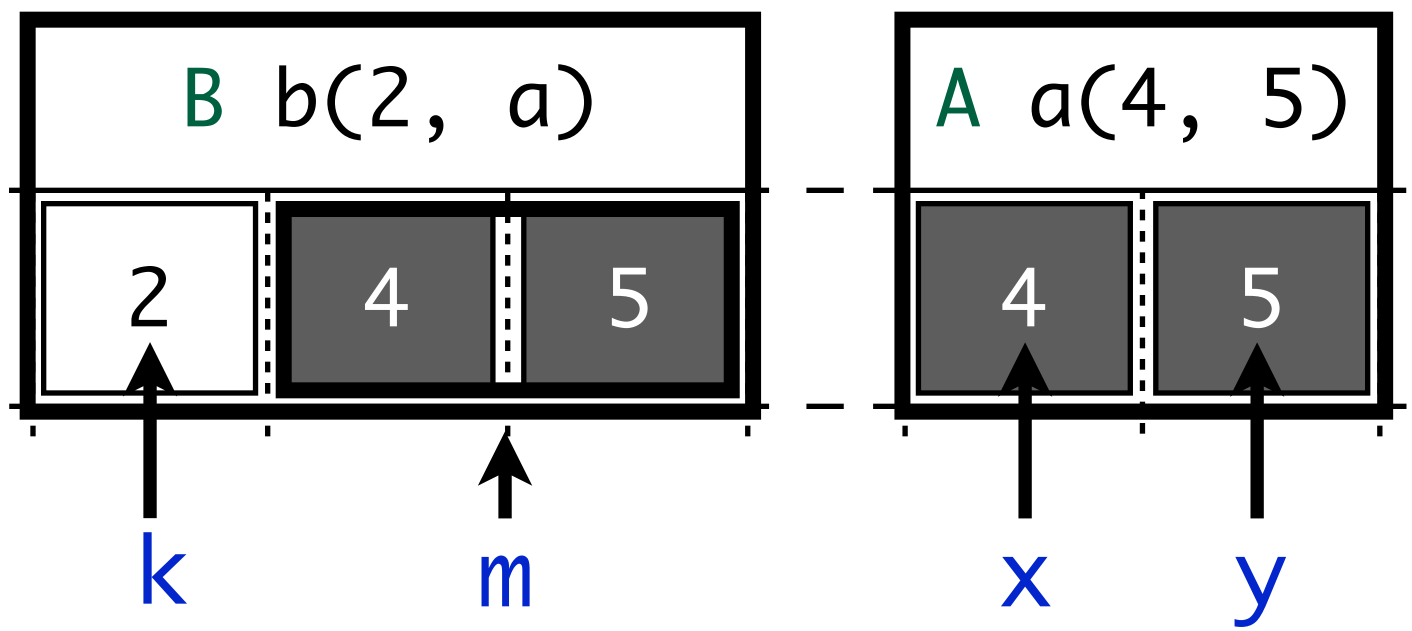
```
class NameOfClass {
  ...
    ~ClassName();
  ...
};
```

Destructors are automatically called by the system before the object is destroyed.
(useful for housekeeping tasks)

```
#include <unistd.h>             // for close(int)
class File {
public:
   File(const char* filename);
   ~File() {                      // destructor; why no parameters?
      close(fd_);                 // close file descriptor
   }
...
private:
   int fd_;                       // file descriptor corresponding
                                  // to opened file
...
};

void process_file(const char* name) {
   File f(name);
... // on return, f is automatically (correctly) destroyed
}
```

Member Objects

```
class A { // ...
public:
   A(int i, int j) : x(i), y(j) { }
private:
   int x; // data member
   int y; // data member
};

class B { // ...
public:
   B(int i, A& a) :k(i), m(a) { }
private:
   int k; // data member
   A    m; // a member object
};
...
A a(4, 5);
B b(2, a);
```

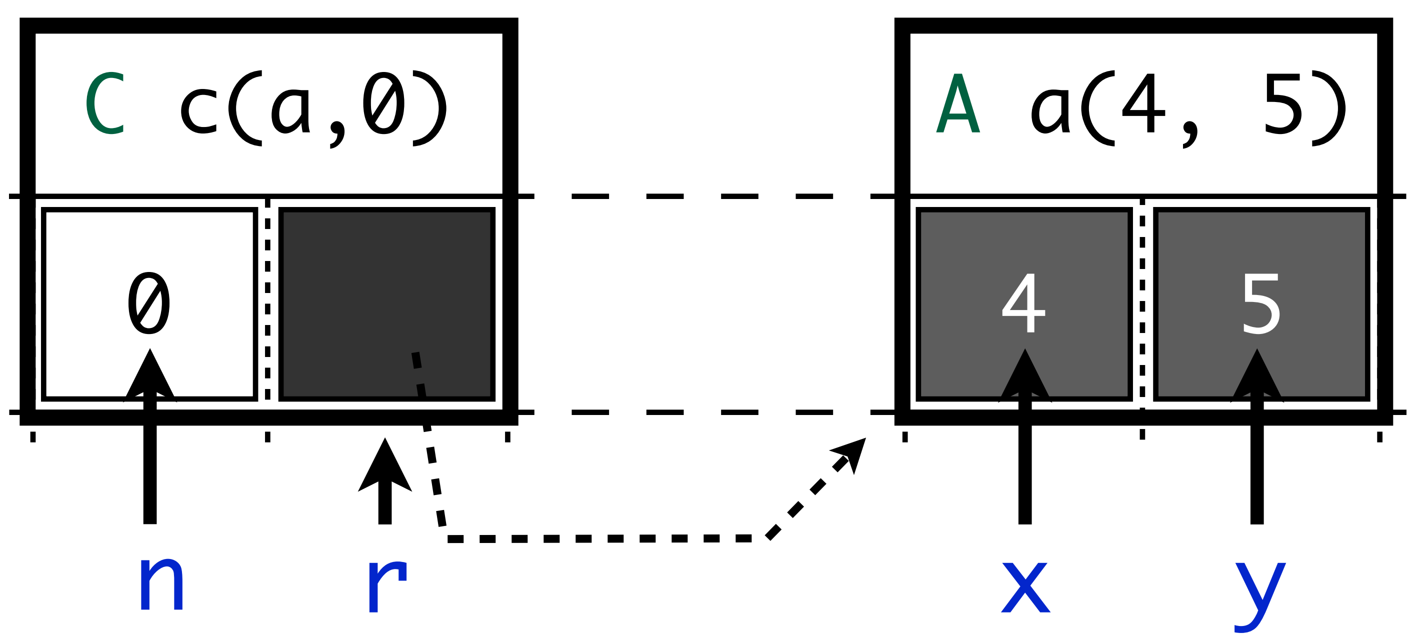


a **copy** is stored in `m` !

What is happening?

Member References

```
class A { // ...
public:
   A(int i, int j) : x(i), y(j) { }
private:
   int x; // data member
   int y; // data member
};

class C { // ...
public:
   C(A& a, int i) :r(a), n(i) { }
private:
   int n; // data member
   A& r; // not a member object
         // *must* be initialized!
};
...
A a(4, 5);
C c(a, 0);
```



What is happening?

Example: The Life-cycle of a Server Class Object

```
#include <fstream>
class Server {
public:
   Server(const char* logfilename, int port) : log_(logfilename), port_(port) {
      // set up server
      log_ << "server started\n";            // why does this work?
   }
   ~Server() { // close down server
      log_ << "server quitting\n";           // why does this work?
   }
   void serve() { // handle requests
   }
   // ...
private:
   Server(const Server&);
   Server& operator=(const Server&);
   std::ofstream log_;
   int port_;
};
```

The Life-cycle of a Class Object

1. (allocate memory)
2. Construction using a (possibly default) constructor function:
   i) Construct member objects in the order of their declaration (which should match the order in the initialization list of the constructor)
   ii) Execute the body of the constructor
3. Provide services via member function calls, or as parameter to ordinary functions
4. Destruction:
   i) Execute code of the destructor body, if there is a destructor
   ii) Destroy member objects
5. (deallocate memory)

Friends: Example

Giving other classes access to your private members

```
class Rational {
public:
   Rational(int num = 0, int denom = 1);
   Rational multiply(Rational r);
   Rational add(Rational r);

   // non-member function operator<< has
   // access to private members of Rational
   friend std::ostream& operator<<(std::ostream&, Rational);
private:
   int num_;
   int denom_;
};


// definition of operator<<
inline std::ostream& operator<<(std::ostream& os, Rational r) {
   return os << r.num_ << "/" << r.denom_;
}
...

Rational r(2, 3);
std::cout << r;
```

What happens?

Friends: Another Example

```
class Node {
   friend class IntStack; // everything is private
                          // but IntStack is a friend so only IntStack
                          // can use Node objects
private:                  // private is default, this line could be dropped
   Node(int, Node* next = 0);
   ~Node();
   Node* next() { return next_; }
   int item;
   Node* next_;
};

class IntStack {           // stack of int
public:
   IntStack();
   ~IntStack();
   IntStack& push(int);
   int top();
   bool empty();
private:
   Node* top_;             // pointer to topmost node
};
```

Software Languages Lab

Nested Classes

```
class IntStack {                    // stack of integers
public:
   IntStack();
   ~IntStack();
   IntStack& push(int);
   // ..
private:
   class Node {                    // why is this solution better
   public:                         // than a non-nested Node class?
      Node(int, Node* next = 0);
      ~Node();
      Node* next();
   private:
      int item;
      Node* next_;
   };

   Node* top_;                      // pointer to topmost node
};

inline Node*                        // also illustrates scope (::) operator
IntStack::Node::next() { return next_; }
```

Static Members: Declaration (point.h)

Persistent class-side state

```
#ifndef POINT_H
#define POINT_H
class Point {
public:
   Point(int X, int Y) : x(X), y(Y) { ++count; }
   Point(const Point&p) : x(p.x), y(p.y) { ++count; }
   ~Point() { --count; }

   // declaration (and definition) of a static member function
   static int get_count() { return count; }
...
private:
   static int count;     // declaration of a static data member
   int x;                // x-coordinate of point
   int y;                // y-coordinate of point
};
#endif
```

Static Members: Definition (`point.cpp`)

`point.cpp` should contain the definition of the static member:

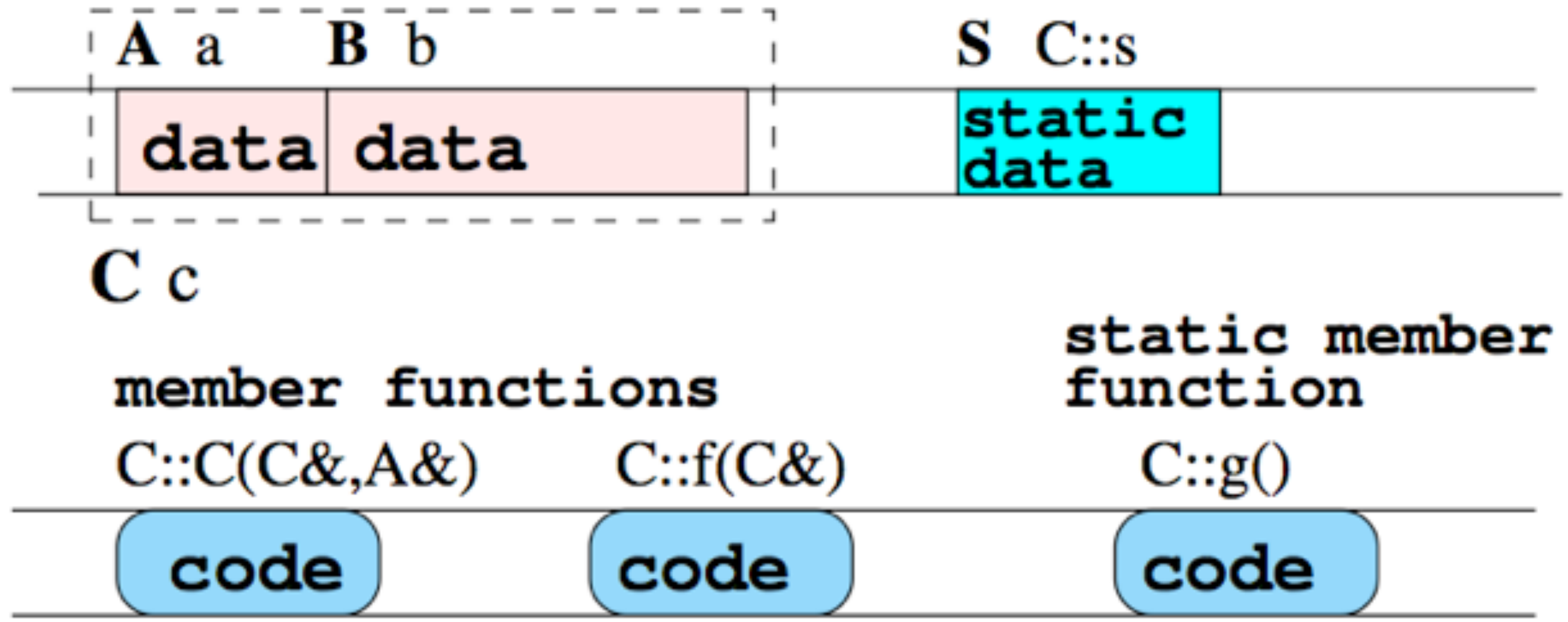
```
#include "point.h"
// definition of static data member
int Point::count(0);
```

Example:

```
#include "point.h"
Point p1(1, 2);
{
   Point p1(p);
   p.get_count(); // print 2
}

// no target needed:
Point::get_count(); // print 1
```

Implementing Classes

```
class C {
public:
   C(A& a);
   A f();
   static S g();

private:
   A a;
   B b;
   static S s;
};
```



- class objects:
  - have separate data area (**data members**)
  - share code (**function members**)
- **static data members** are shared and global
- **non-static member functions** have extra target object (lvalue) parameter

Software Languages Lab

Enumeration Types

```
enum NameOfType { EnumeratorList } ;
```

finite integral types

```
class File {
public:
   // defines 4 names in scope File
   enum Mode { READ, WRITE, APPEND };
   File(const char* filename, Mode mode = READ);
   ~File();
   Mode mode() { return mode_;    }
private:
   Mode mode_;
   // ...
};
```

```
File f("book.tex");
if (f.mode()==File::WRITE) {  // ... }
```

Overriding Enumerated Type Values (1)

```
class Http {
public:
   enum Operation { GET, HEAD, PUT};
   enum Status {
      OK = 200,
      CREATED = 201,
      ACCEPTED = 202,
      PARTIAL = 203,
      MOVED = 301,
      FOUND = 302,
      METHOD = 303,
      NO_CHANGE = 304,
      BAD_REQUEST = 400,
      UNAUTHORIZED = 401,
      PAYMENT_REQUIRED = 402,
      FORBIDDEN = 403,
      NOT_FOUND = 404,
      INTERNAL_ERROR = 500,
      NOT_IMPLEMENTED = 501
   };
\\...
};
```

Overriding Enumerated Type Values (2)

```
class Http {
public:
   enum Operation { GET, HEAD, PUT };
   enum Status {
   //...
   };
   //...
};

std::ostream& operator<<(std::ostream& os, Http::Status status) {
   switch (status) {
   case Http::OK:
      os << "OK";
      break;
   case Http::CREATED:
      os << "CREATED";
      break;
   case Http::ACCEPTED:
      os << "ACCEPTED";
      break;
      //...
   }
   return os;
}
```

Typedef

```
typedef Declaration;
```

Defines a short name for a (complex) type expression

```
typedef unsigned int uint;
uint x; // equivalent with unsigned int x;

typedef Sql::Command::iterator IT;

int square(int x) {
   return x * x;
}

// also for function types:
typedef int UnaryFunction(int);
// UnaryFunction is type int -> int
// f is pointer to function (see later), it is
// initialized to (point to) the 'square' function
UnaryFunction* f(square);
f(2); // same as square(2)
```