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

الماس فناوری ابری پاسارگاد- آلفا

مدرس: سعید امراللهی بیوکی

Agenda 13/24

Session 13. Namespaces and Exception Handling

- The C global namespace
- Scope and Lifetime
- C++ namespace as logical modularity
- Standard library namespace: std
- Errors and Error handling
- The types of errors: Compile-time, Link-time, Run-time and Logic errors
- Compile-time errors: Syntax errors, Type errors
- Compiler as the best friend against errors
- Standard library exception classes
- Preventing exception propagation: C++11 noexcept
- Q&A

150 min (incl. Q & A)

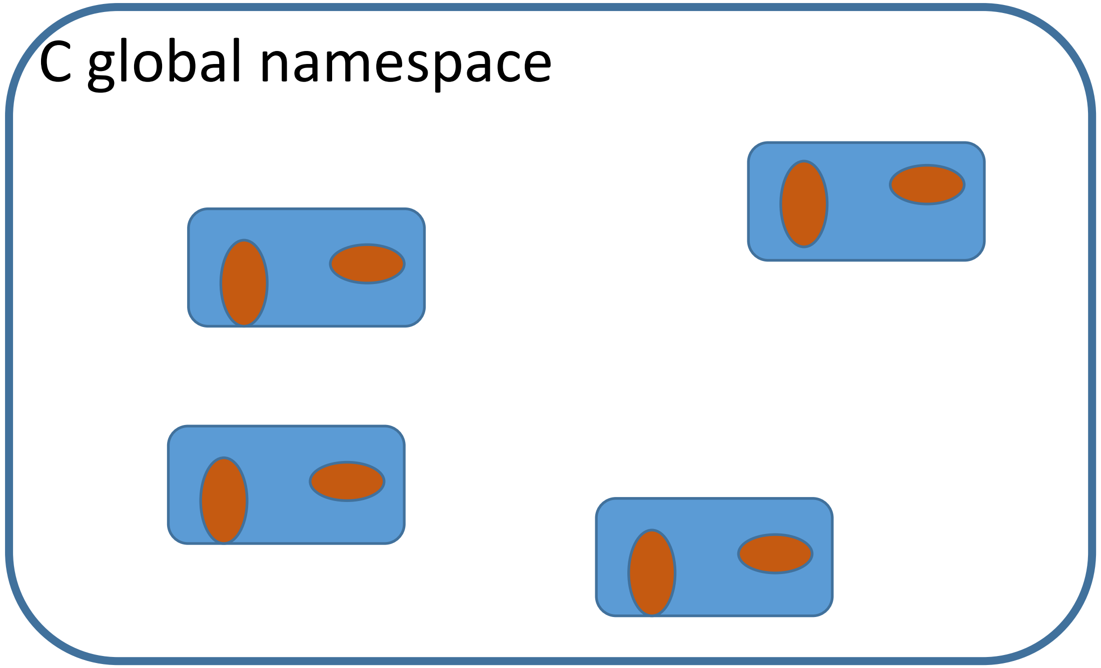
Global namespace

Global namespace

- C provides a single global namespace for all names that don’t conventionally fit into a single function, a single *struct*, or a single file.

Global namespace

- C provides a single global namespace for all names that don't conventionally fit into a single function, a single *struct*, or a single file.
- The Problem: Name clashes



Global namespace

- C provides a single global namespace for all names that don’t conventionally fit into a single function, a single *struct*, or a single file.
- The Problem: Name clashes

```
// 1.h
char f(char);
int f(int);
class String { /* … */ };
```

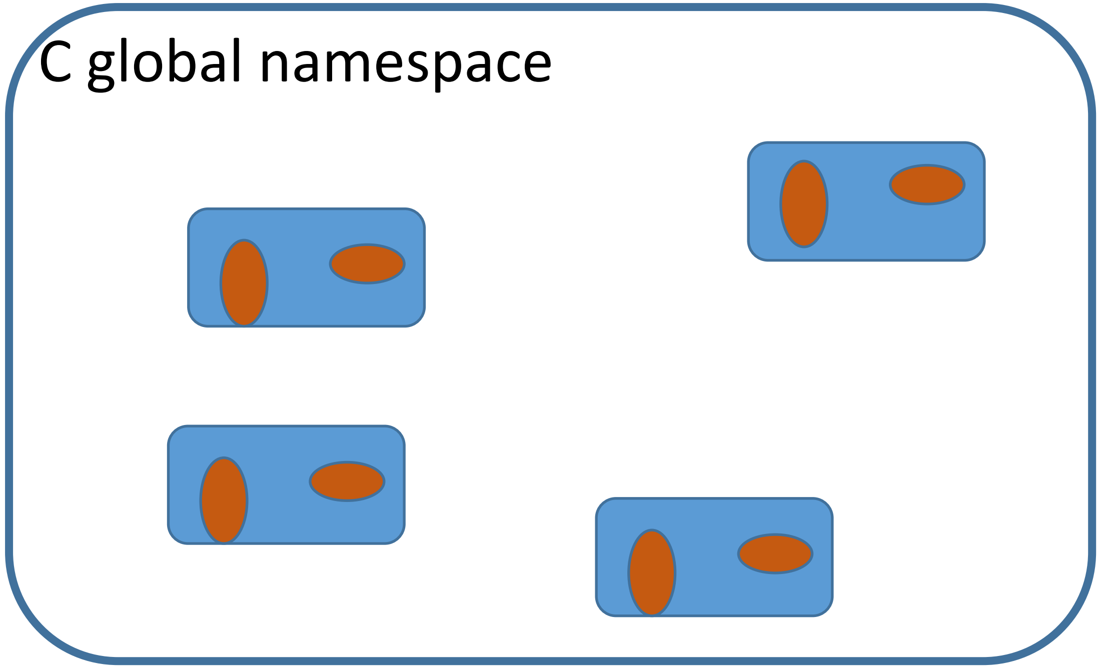
```
// 2.h
char f(char);
int f(int);
class String { /* … */ };
```

```
// third_party.c
#include “1.h”
#include “2.h”
int main()
{
  char c = f(‘a’);

}
```

'String' : 'class' type redefinition

function 'char f(char)' already has a body



Global namespace

- C provides a single global namespace for all names that don’t conventionally fit into a single function, a single *struct*, or a single file.
- The Problem: Name clashes

```
// 1.h
char f(char);
int f(int);
class String { /* … */ };
```

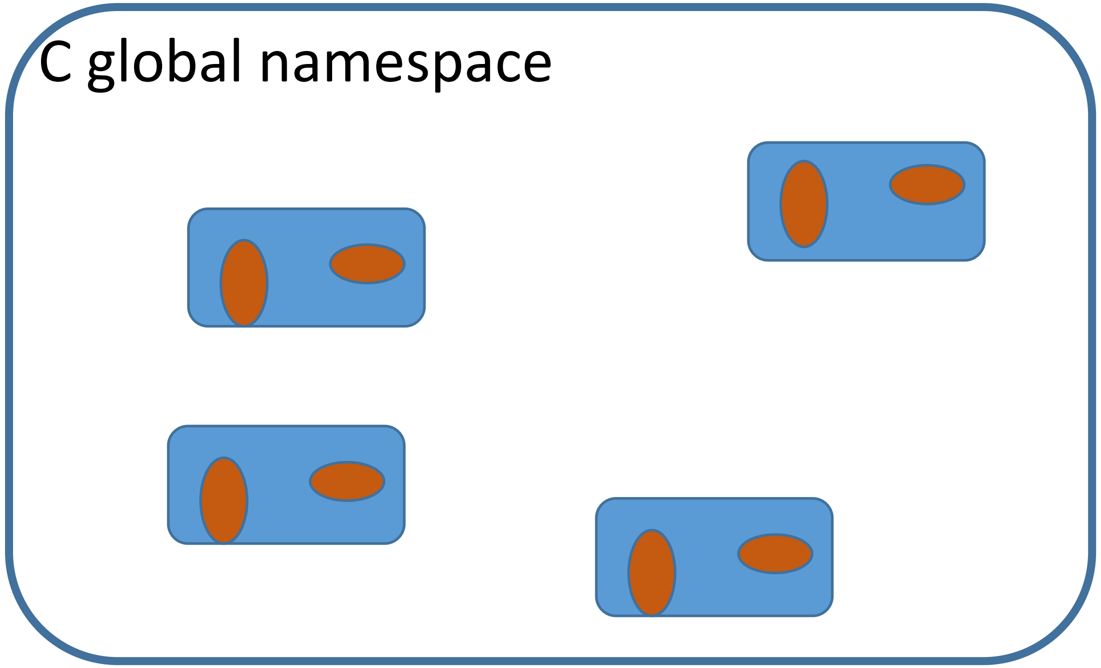
```
// 2.h
char f(char);
int f(int);
class String { /* … */ };
```

```
// third_party.c
#include “1.h”
#include “2.h”
int main()
{
  char c = f(‘a’);

}
```

'String' : 'class' type redefinition

function 'char f(char)' already has a body



- Solution: C++ namespace

Global namespace

- C provides a single global namespace for all names that don’t conventionally fit into a single function, a single *struct*, or a single file.
- The Problem: Name clashes

```
// 1.h
char f(char);
int f(int);
class String { /* … */ };
```

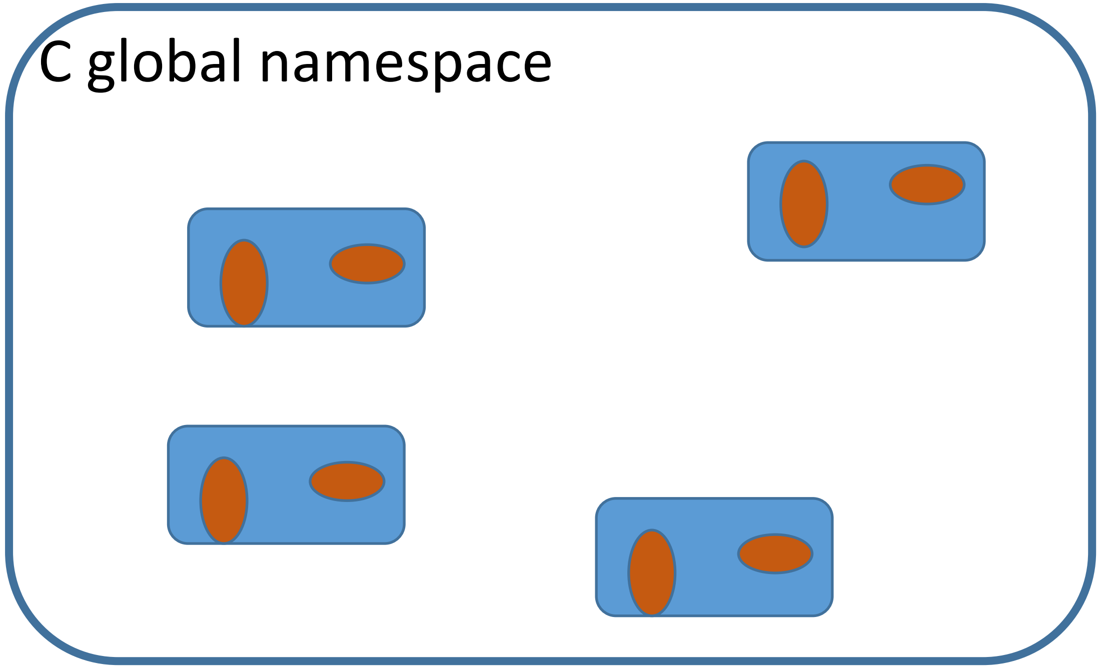
```
// 2.h
char f(char);
int f(int);
class String { /* … */ };
```

```
// third_party.c
#include “1.h”
#include “2.h”
int main()
{
  char c = f(‘a’);

}
```

'String' : 'class' type redefinition

function 'char f(char)' already has a body

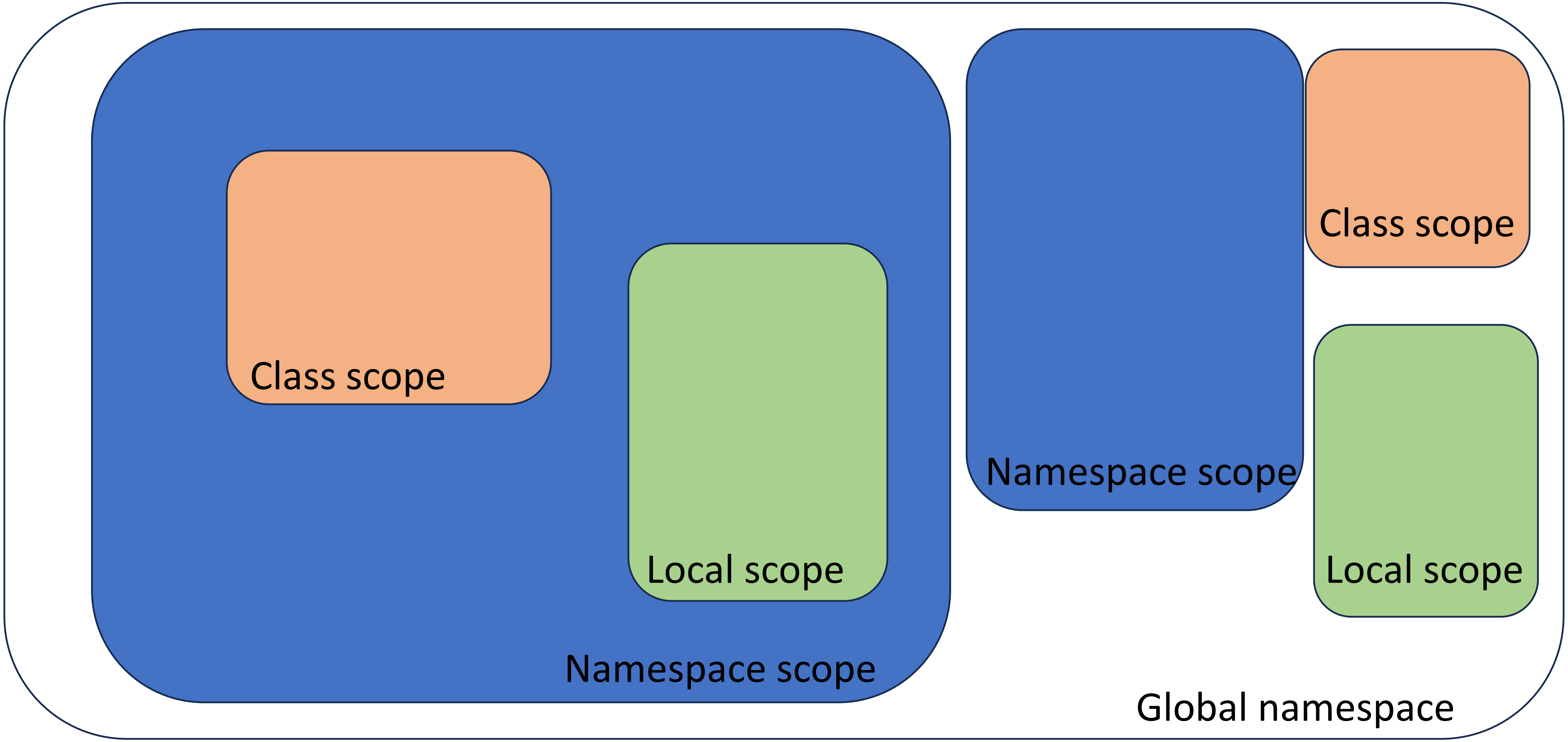

C global namespace

Language-technical rule

Locality is good.

- Solution: C++ namespace

Global namespace



Scope

- In C^{++}, braces tell the implementation to treat whatever appears between them as a unit.

```
// simple block
{---------------------→Beginning of block
  /* … */
}---------------------→ End of block
```

```
// user-defined type
class A {
  // …
};
```

```
// user-defined structure
struct B {
  // …
};
```

```
// namespace definition
namespace A {
  // …
}
```

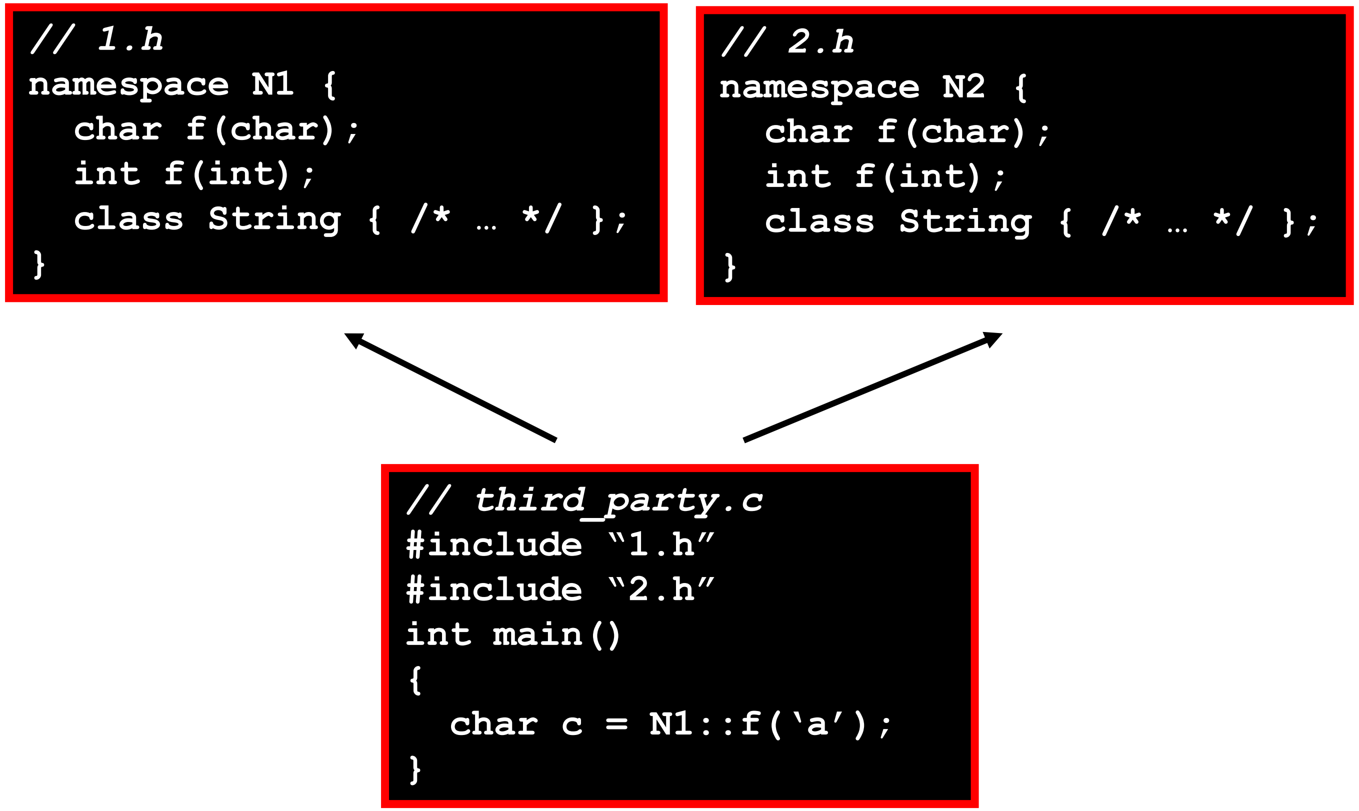
```
// function definition
void f() {
  // …
};
```

```
// if statement
if (a > b) {
  /* … */
}
```

```
// while statement
while (…) {
  // …
};
```

- Local scope
- Function Scope
- Class Scope
- Namespace Scope

Namespace



```
// 1.h
namespace N1 {
  char f(char);
  int f(int);
  class String { /* … */ };
}
```

```
// 2.h
namespace N2 {
  char f(char);
  int f(int);
  class String { /* … */ };
}
```

```
// third_party.c
#include “1.h”
#include “2.h”
int main()
{
  char c = N1::f(‘a’);
}
```

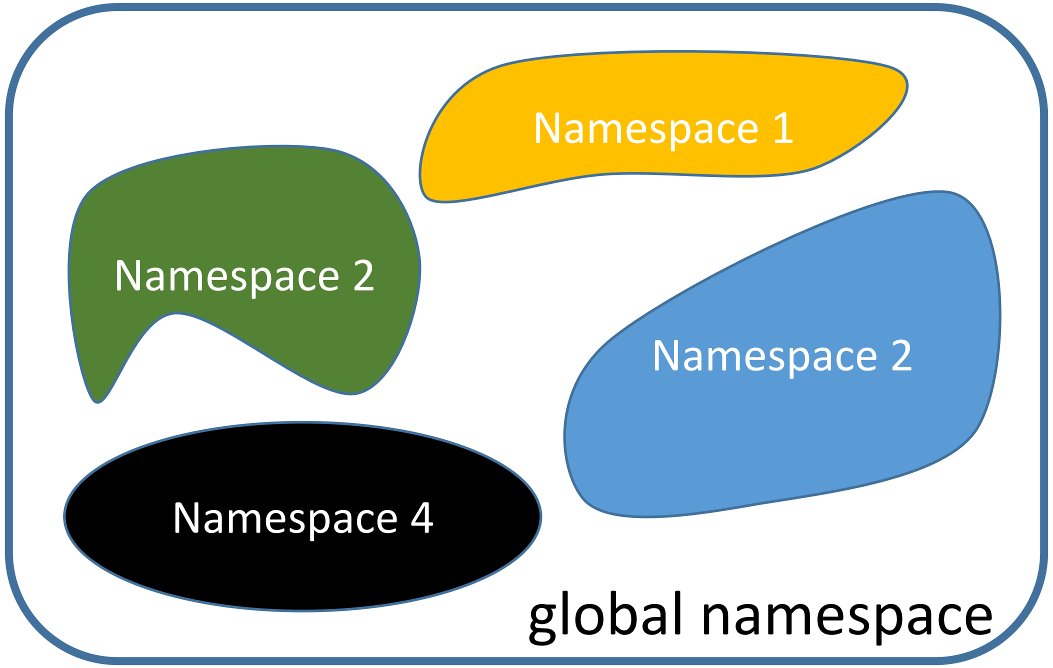
Namespace as a modularity mechanism

- A namespace is a mechanism for expressing logical grouping.
- The notion of a namespace is provided to directly represent the notion of a set of facilities that directly belong together.

Namespace as a modularity mechanism

- A namespace is a mechanism for expressing logical grouping.

- The notion of a namespace is provided to directly represent the notion of a set of facilities that directly belong together.
- A namespace should express some logical structure. They contents of a namespace should be seen as a logical unit.

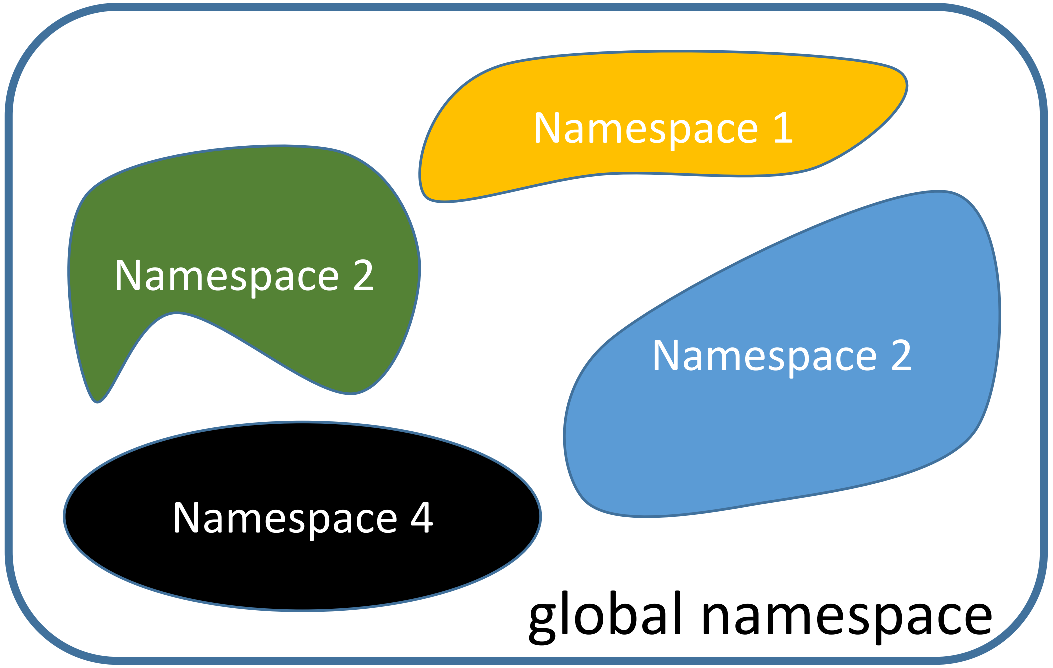


Namespace as a modularity mechanism

- A namespace is a mechanism for expressing logical grouping.

- The notion of a namespace is provided to directly represent the notion of a set of facilities that directly belong together.
- A namespace should express some logical structure. They contents of a namespace should be seen as a logical unit.

- In real programs, each “module” represented by a separate namespace.

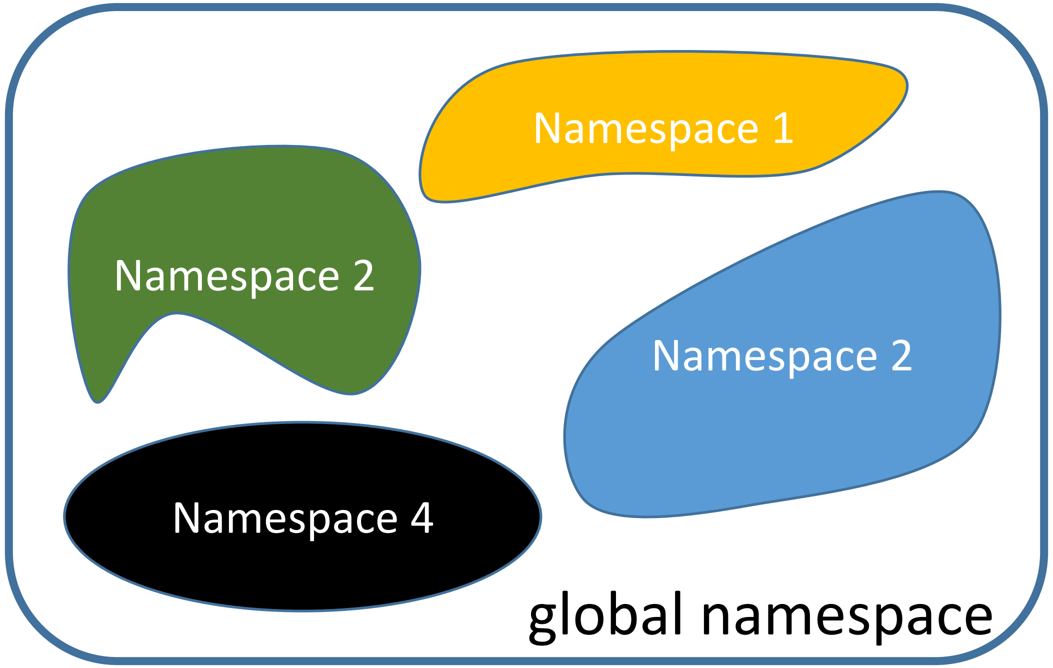


Namespace as a modularity mechanism

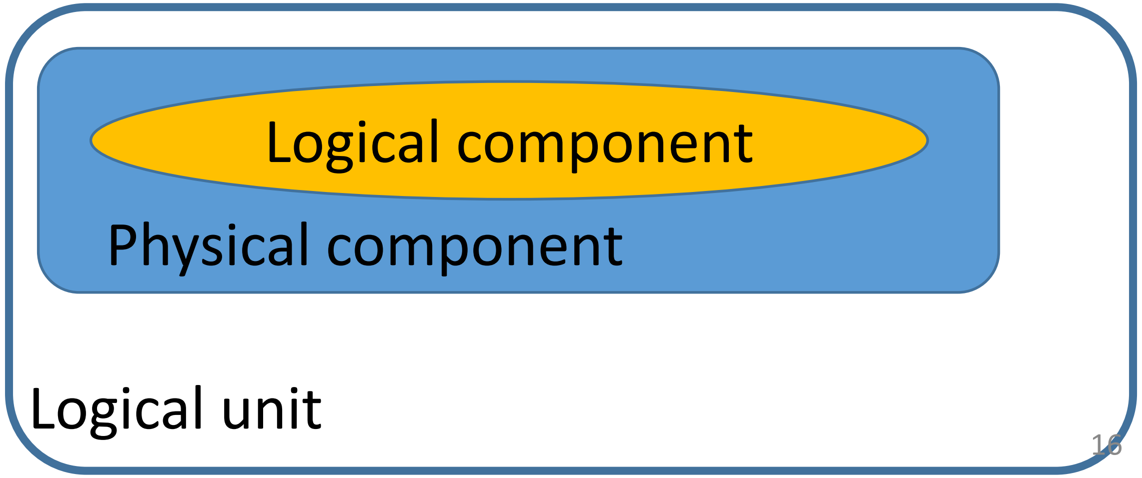
- A namespace is a mechanism for expressing logical grouping.

- The notion of a namespace is provided to directly represent the notion of a set of facilities that directly belong together.
- A namespace should express some logical structure. They contents of a namespace should be seen as a logical unit.

- In real programs, each “module” represented by a separate namespace.

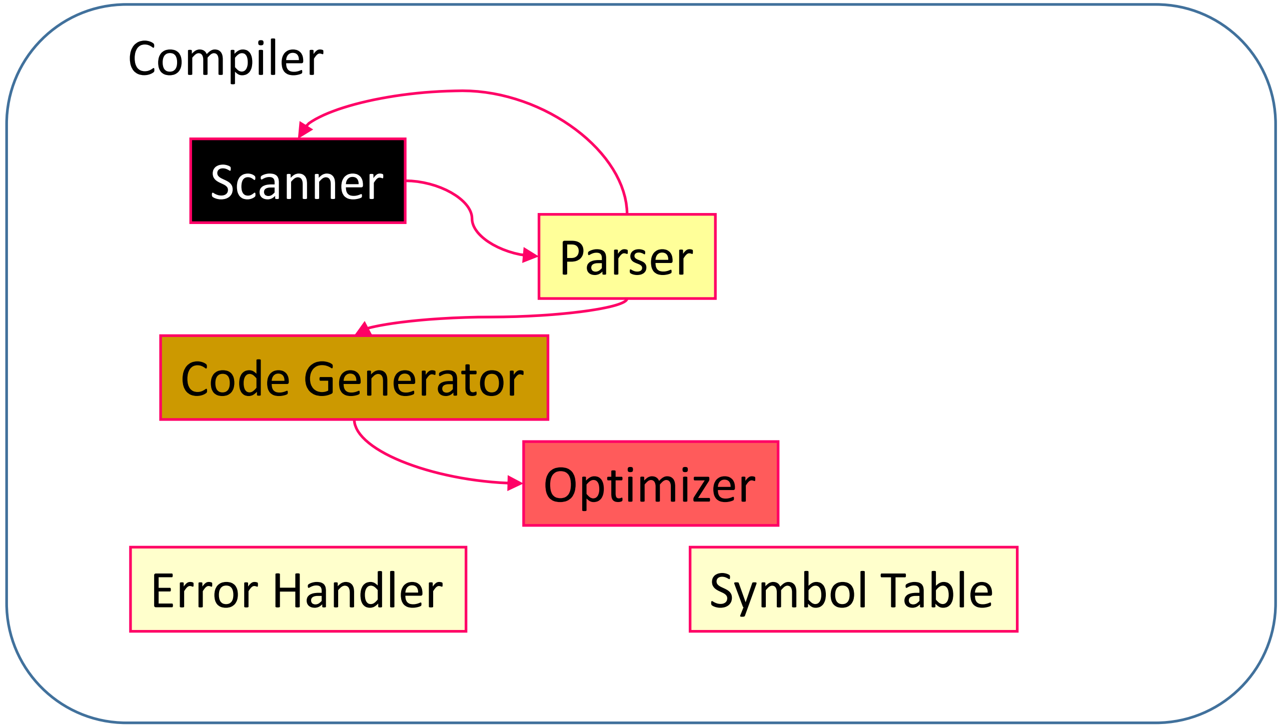


- Namespace → Logical unit or structure
- Files & Libraries → physical component
- Classes and Functions → Logical component

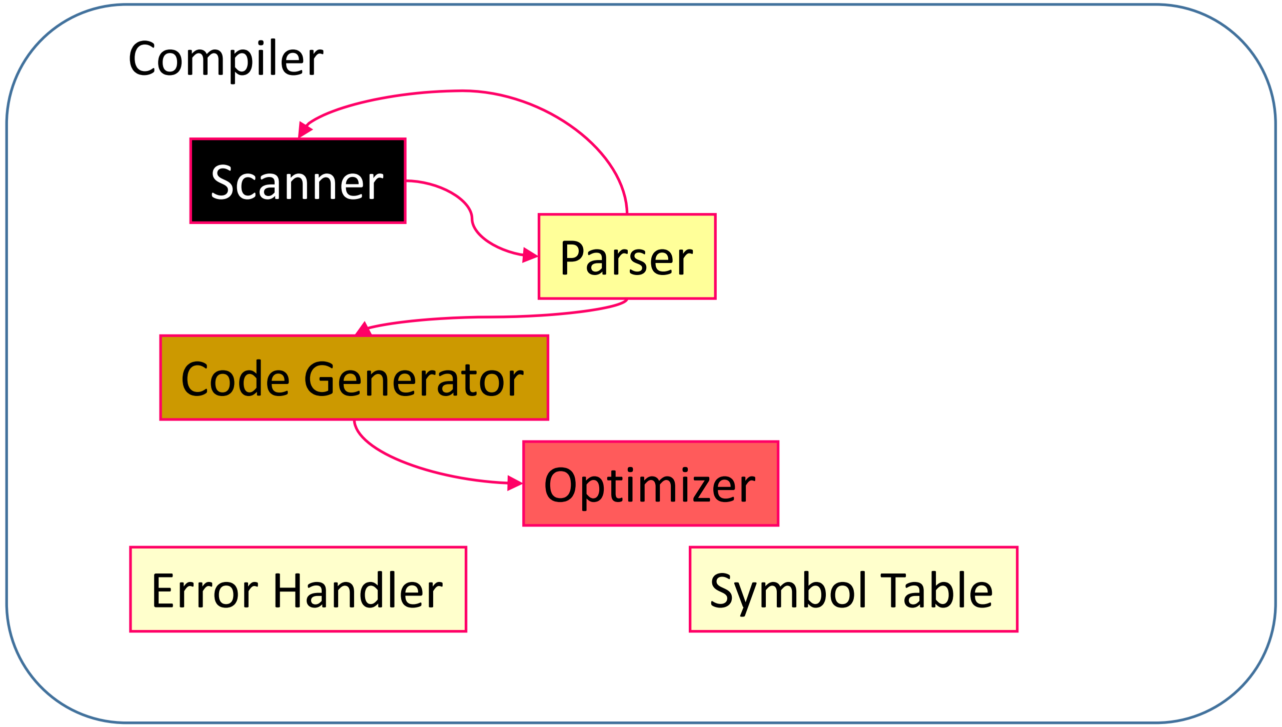


Namespace: an example

Namespace: an example



Namespace: an example



```
namespace Scanner {
  /* … */
} // no need for ;
```

```
namespace Parser {
  /* … */
}
```

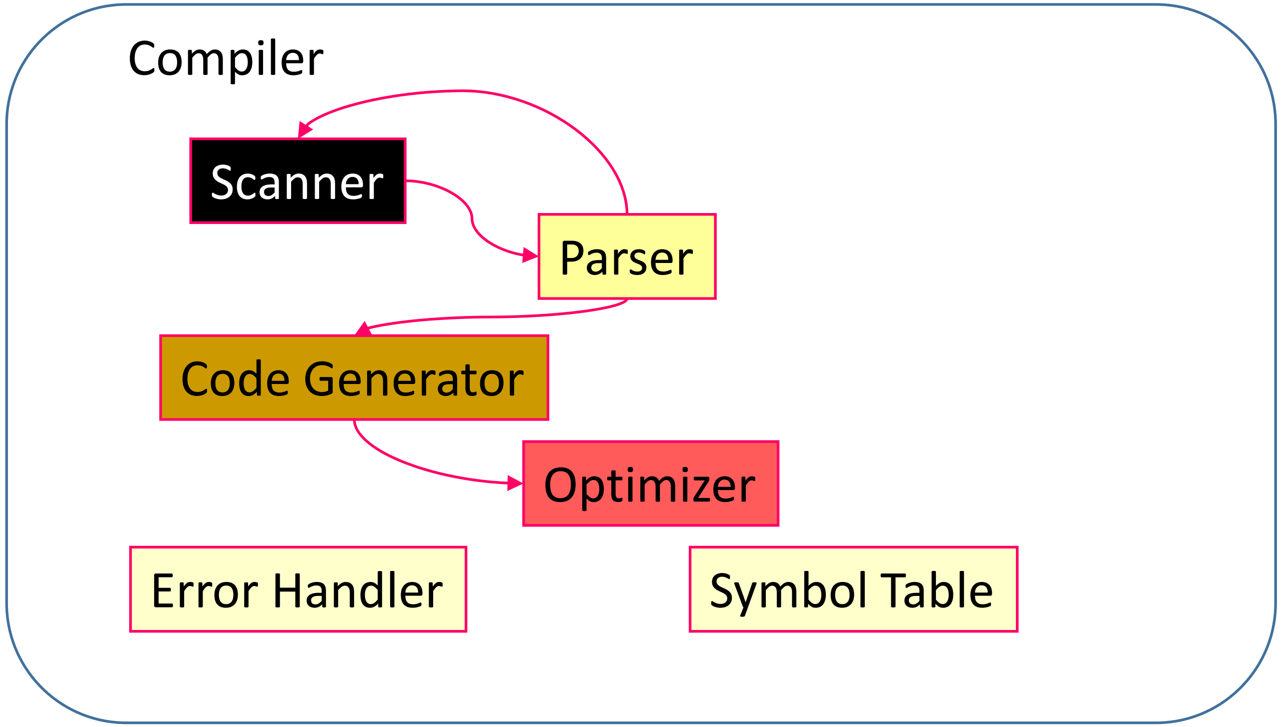
```
namespace CodeGenerator {
  /* … */
}
```

```
namespace Optimizer {
  /* … */
}
```

```
namespace Parser {
  using namespace Scanner;
  // …
}
```

```
namespace CodeGenerator {
  using namespace Parser;
  // …
}
```

Namespace: an example



Compiler

Scanner

Parser

Code Generator

Optimizer

Error Handler

Symbol Table

```
namespace Scanner {
  /* … */
} // no need for ;
```

```
namespace Parser {
  /* … */
}
```

```
namespace CodeGenerator {
  /* … */
}
```

```
namespace Optimizer {
  /* … */
}
```

```
namespace Parser {
  using namespace Scanner;
  // …
}
```

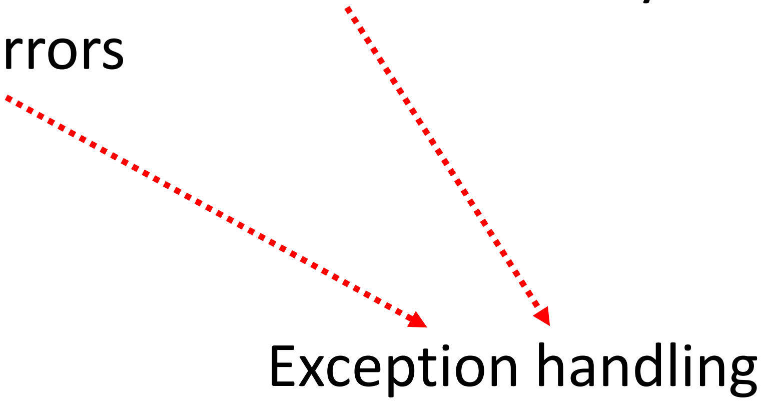
```
namespace CodeGenerator {
  using namespace Parser;
  // …
}
```

- Use namespace to express logical structure.
- Put every non-local name, except main() in some namespace.
- Use namespaces for your company/organization codebase.

Errors

- Errors

  - Compile-time errors: errors found by the compiler
    - Syntax errors
    - Type errors
  - Link-time errors: errors found by the linkers
  - Run-time errors: errors found by checking at programs
  - Logic errors



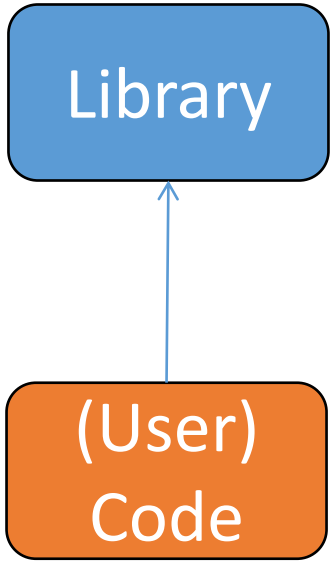
Exception handling

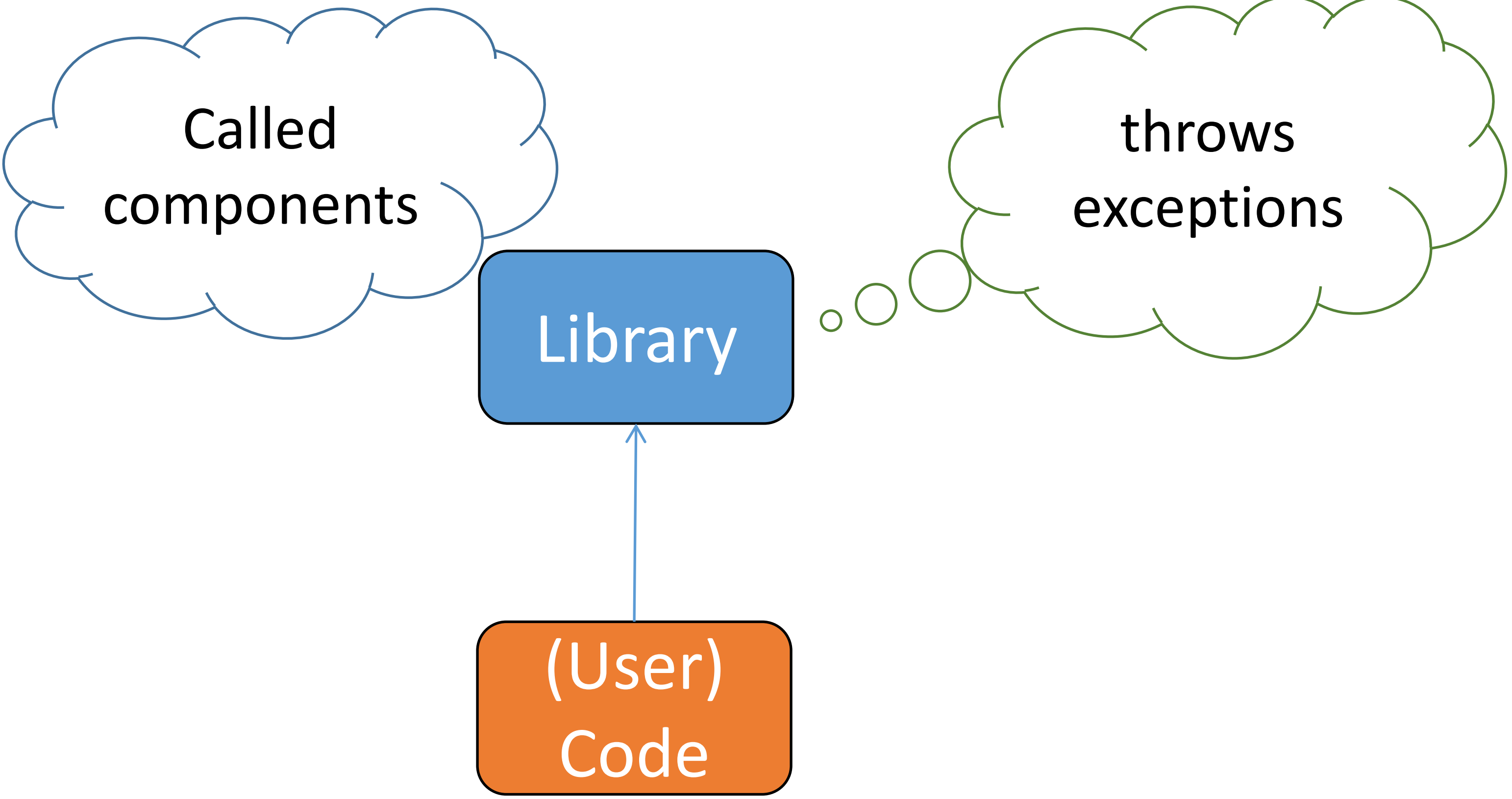
Exception handling

Exception handling

Library

Exception handling


Library
(User) Code

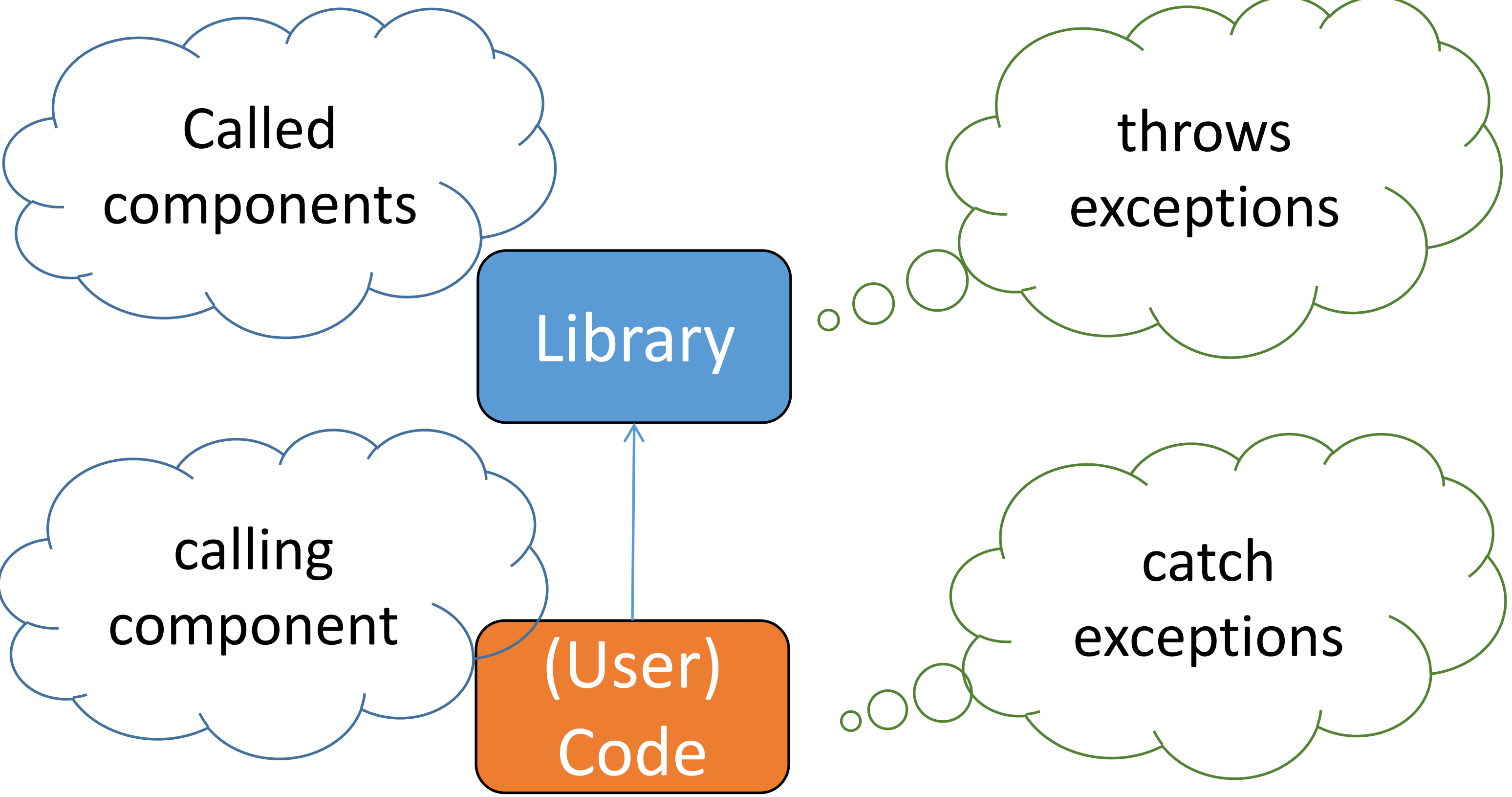


Called components

Library

throws exceptions

(User) Code



Called components

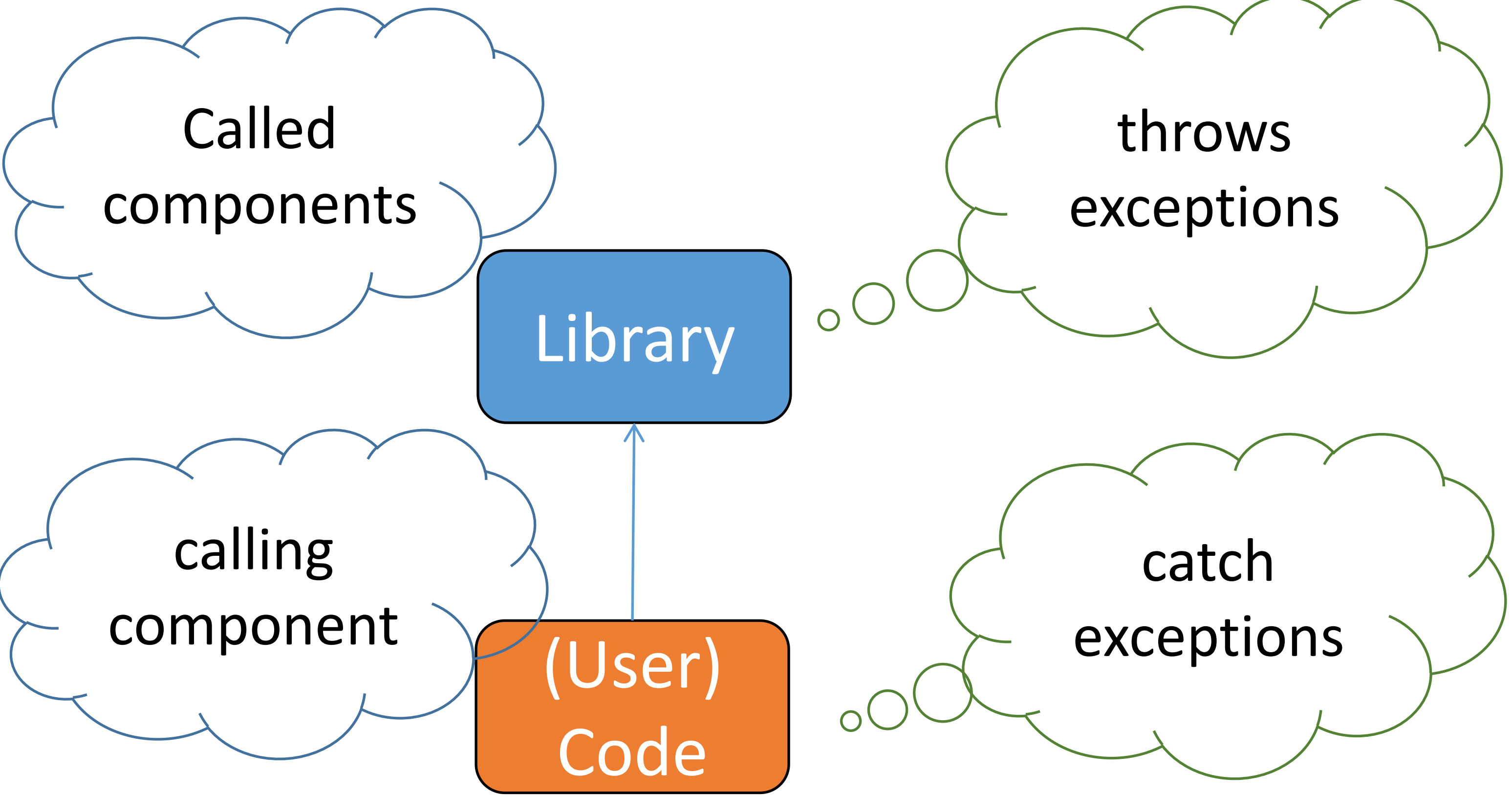
Library

throws exceptions

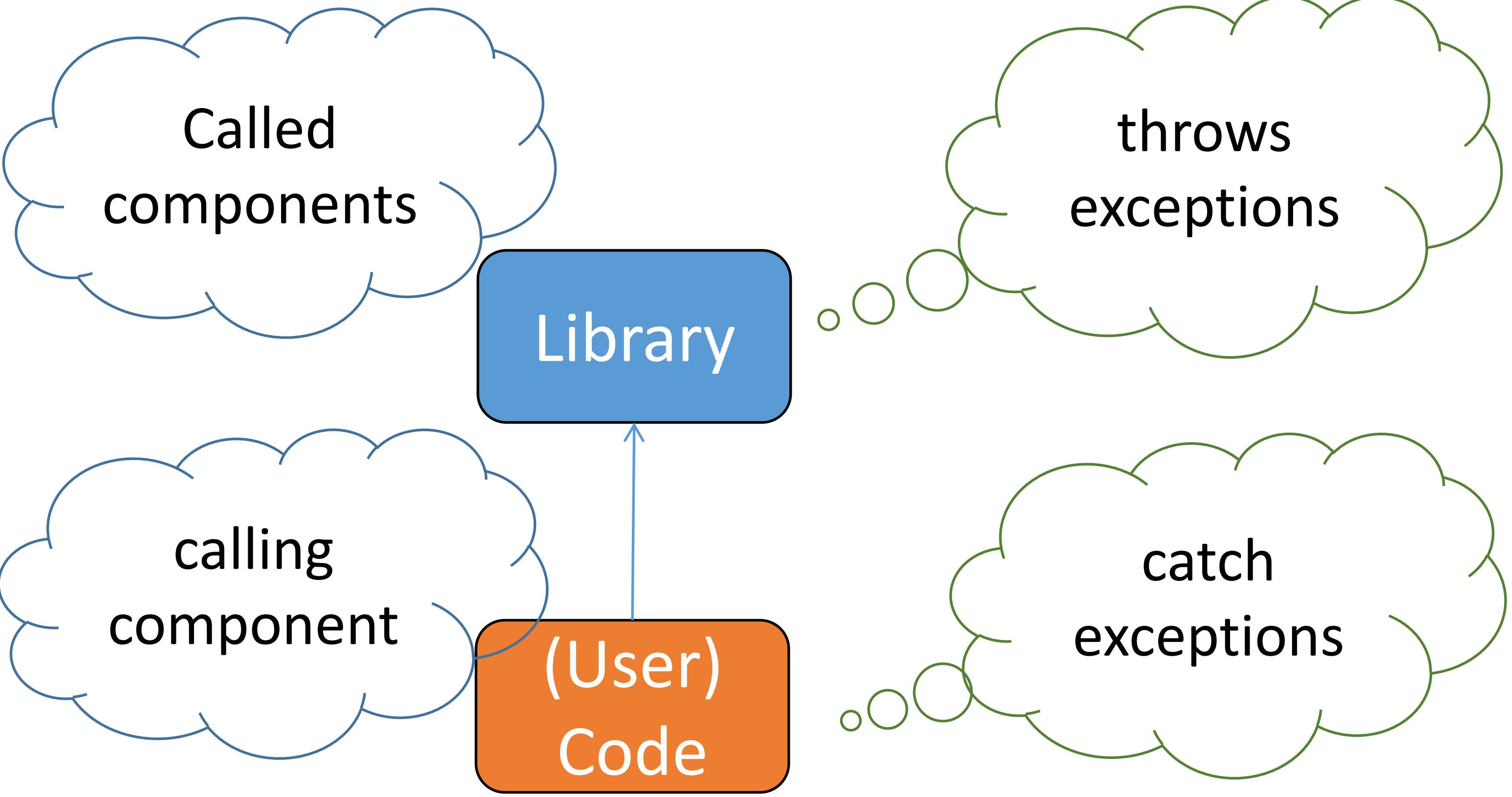
calling component

(User) Code

catch exceptions



- The author of a library can detect a run-time error but does not in general have any idea what to do about it.



Called components

throws exceptions

Library

calling component

(User) Code

catch exceptions

- The author of a library can detect a run-time error but does not in general have any idea what to do about it.
- The user of a library may know to cope with run-time error but cannot easily detect it.

Traditional error handling

Traditional error handling

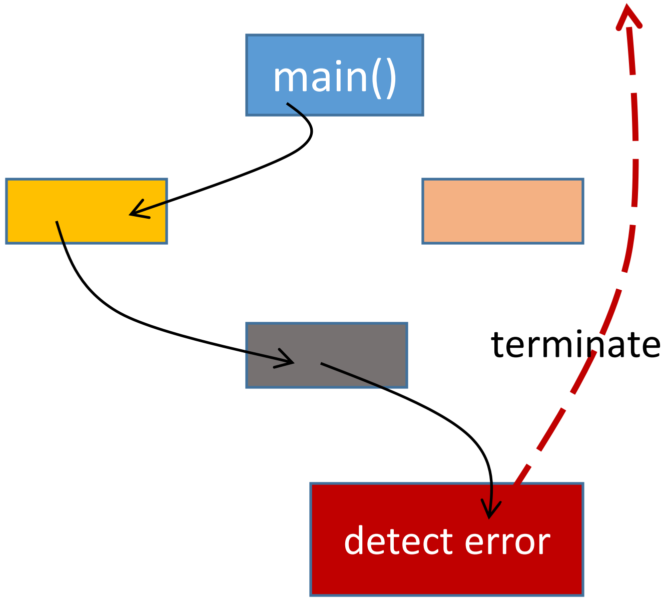
1. terminate the program

   - exit(), abort()

Traditional error handling

1. terminate the program

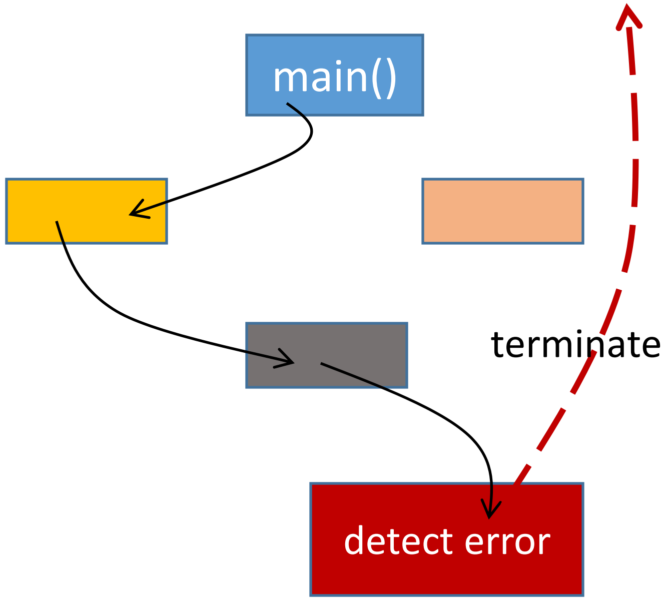
   - exit(), abort()



Traditional error handling

1. terminate the program

   - exit(), abort()

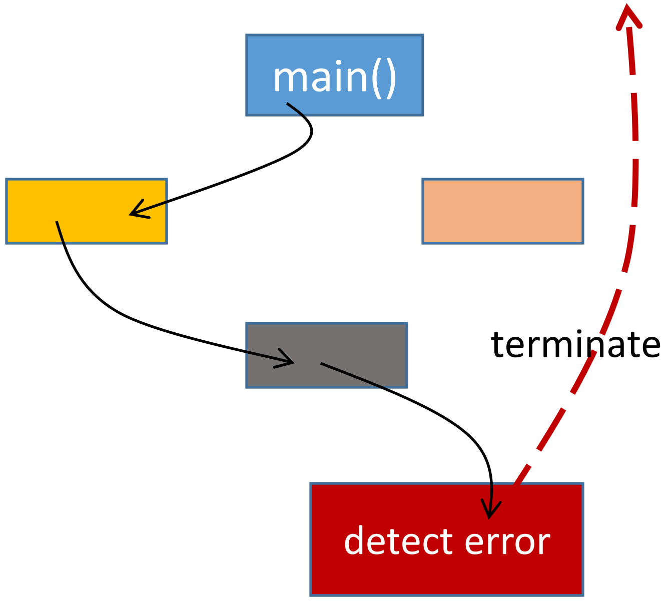


2. return a value representing ‘‘error’’

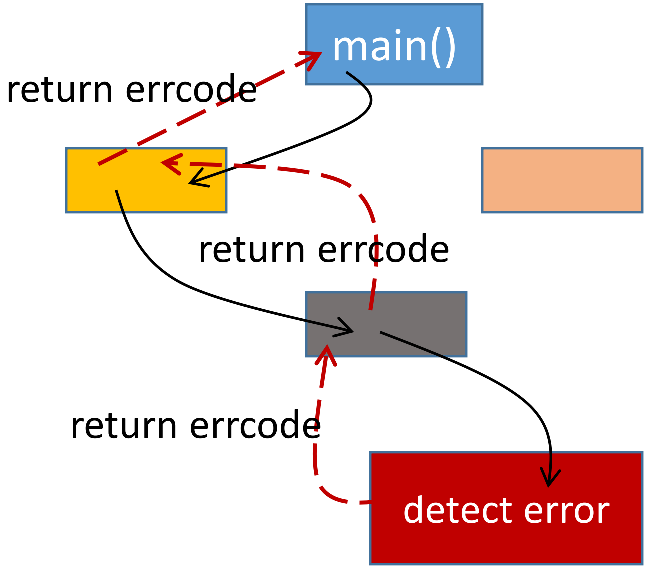
Traditional error handling

1. terminate the program

   - exit(), abort()



2. return a value representing ''error''



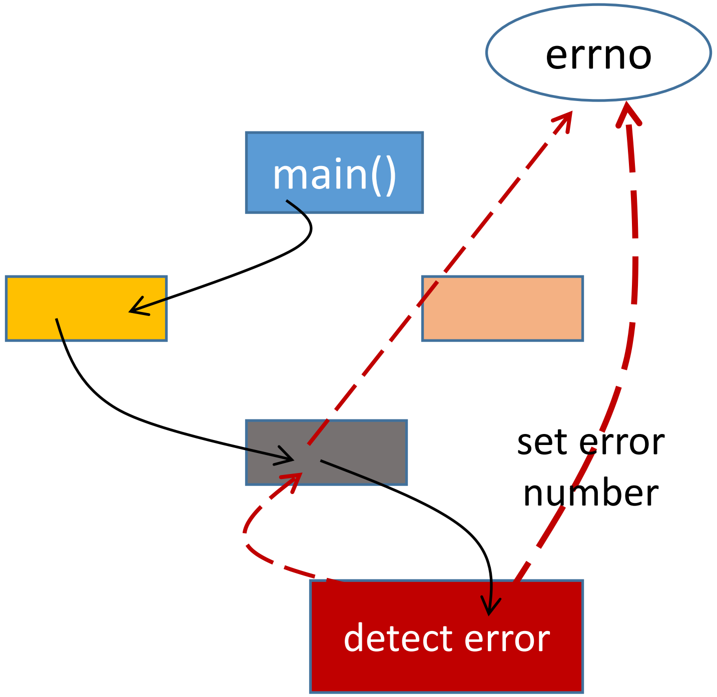
Traditional error handling cont.

Traditional error handling cont.

3. Return a legal value and leave the program in an illegal state.

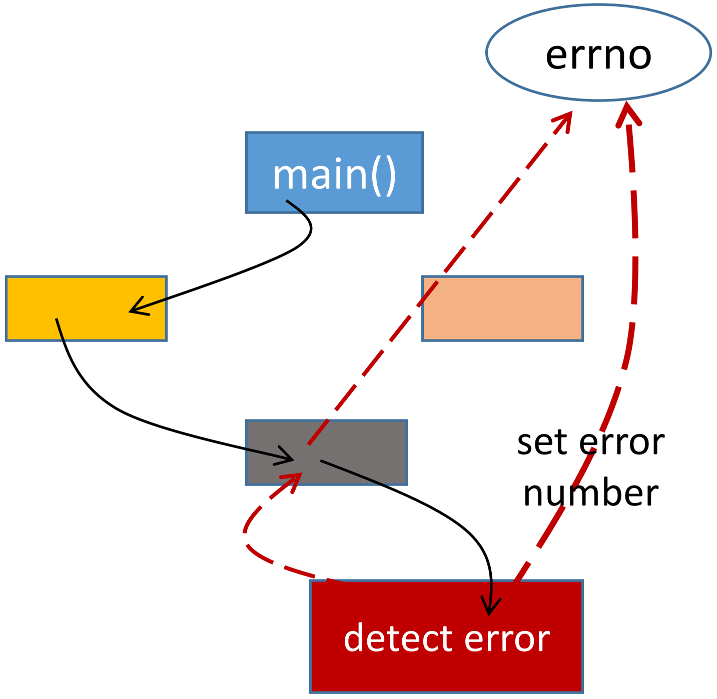
Traditional error handling cont.

3. Return a legal value and leave the program in an illegal state.



Traditional error handling cont.

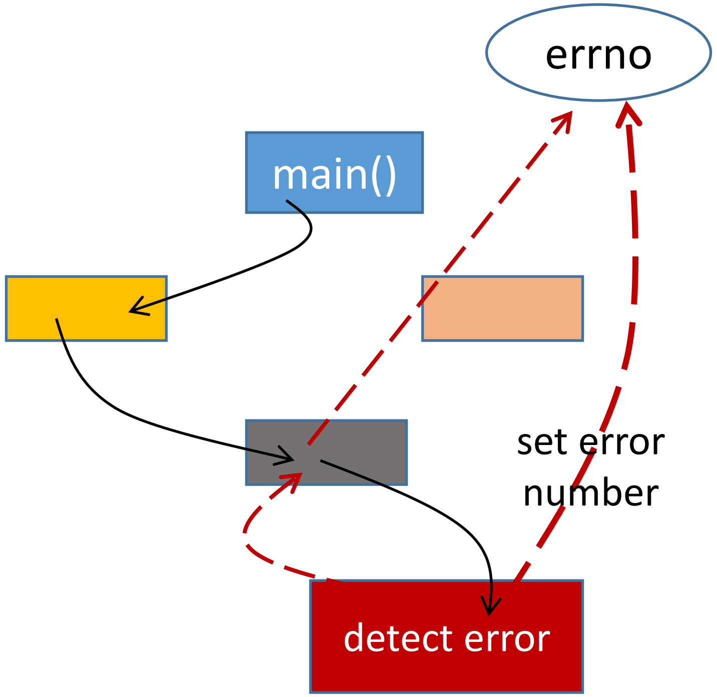
3. Return a legal value and leave the program in an illegal state.



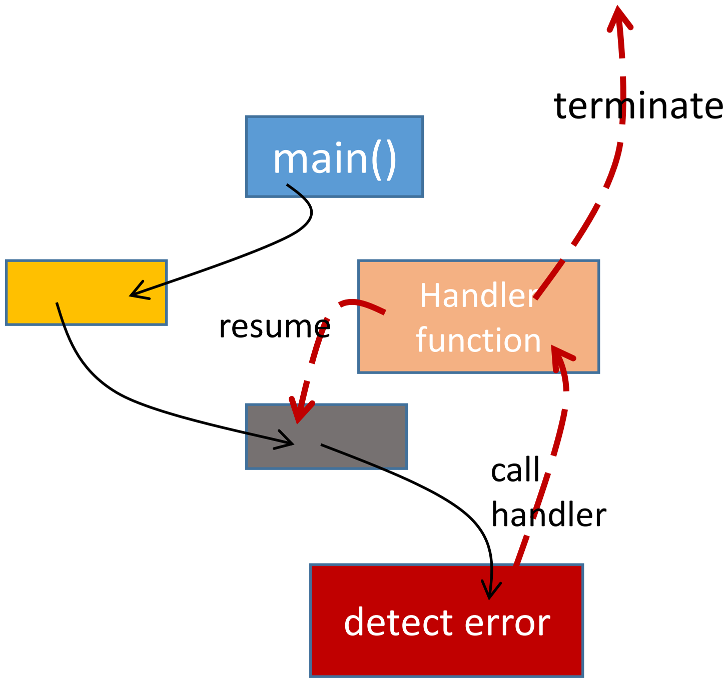
4. call an error-handler function

Traditional error handling cont.

3. Return a legal value and leave the program in an illegal state.



4. call an error-handler function



Try, throw and catch: an example

Try, throw and catch: an example

1

```
void task_master() // calling component
{
     try {
          auto result = do_task();
          // use result
     }
     catch (some_error) {
          // failure to do_task: handle problem
     }
}
```

Try, throw and catch: an example

1

```
void task_master() // calling component
{
     try {
          auto result = do_task();
          // use result
     }
     catch (some_error) {
          // failure to do_task: handle problem
     }
}
```

2

```
int do_task() // called component
{
     if (/* could perform the task */)
          return result;
     else
          throw some_error{};
}
```

Try, throw and catch: an example

1

```
void task_master() // calling component
{
     try {
          auto result = do_task();
          // use result
     }
     catch (some_error) {
          // failure to do_task: handle problem
     }
}
```

2

```
int do_task() // called component
{
     if (/* could perform the task */)
          return result;
     else
          throw some_error{};
}
```

- An exception is an object thrown to represent the occurrence of an error.

Try, throw and catch: an example

- An exception is an object thrown to represent the occurrence of an error.

```
struct range_error {
    // …
};
void f(int i)
{
    if (n < 0 || max < n) throw range_error{};
}
```

1

```
void task_master() // calling component
{
    try {
        auto result = do_task();
        // use result
    }
    catch (some_error) {
        // failure to do_task: handle problem
    }
}
```

2

```
int do_task() // called component
{
    if (/* could perform the task */)
        return result;
    else
        throw some_error{};
}
```

exception handling: The Benefits

exception handling: The Benefits

- It is an alternative to the traditional techniques when they are insufficient, inelegant, or error-prone.

exception handling: The Benefits

- It is an alternative to the traditional techniques when they are insufficient, inelegant, or error-prone.
- It is complete; it can be used to handle all errors detected by ordinary code.

exception handling: The Benefits

- It is an alternative to the traditional techniques when they are insufficient, inelegant, or error-prone.
- It is complete; it can be used to handle all errors detected by ordinary code.
- Allows the programmer to explicitly separate error-handling code from “ordinary code.”

exception handling: The Benefits

- It is an alternative to the traditional techniques when they are insufficient, inelegant, or error-prone.
- It is complete; it can be used to handle all errors detected by ordinary code.
- Allows the programmer to explicitly separate error-handling code from “ordinary code.”
- Supports a more regular style of error handling, thus simplifying cooperation between separately written program fragments.

exception handling: The Benefits

- It is an alternative to the traditional techniques when they are insufficient, inelegant, or error-prone.
- It is complete; it can be used to handle all errors detected by ordinary code.
- Allows the programmer to explicitly separate error-handling code from “ordinary code.”
- Supports a more regular style of error handling, thus simplifying cooperation between separately written program fragments.
- without exceptions, a lot of powerful features like constructors, overloaded operators, and templates are either not as robust as we need them to be, or simply can not be used in situations where good error handling is required.
- Constructors don’t have return value (not even void).

Exception handling: some thoughts

- Error handling and handling exceptional conditions
- Synchronous exceptions: array bound checks, I/O errors, stack overflow, underflow, … asynchronous exceptions: division by zero
- It’s non-local by nature based on Stack unwinding.

OS: Signal

- “Exceptional” does not mean ‘‘almost never happens’’ or “disastrous.”
- “Exceptional mean: some part of the system couldn’t do what it was asked to do.’’

Exception class hierarchies

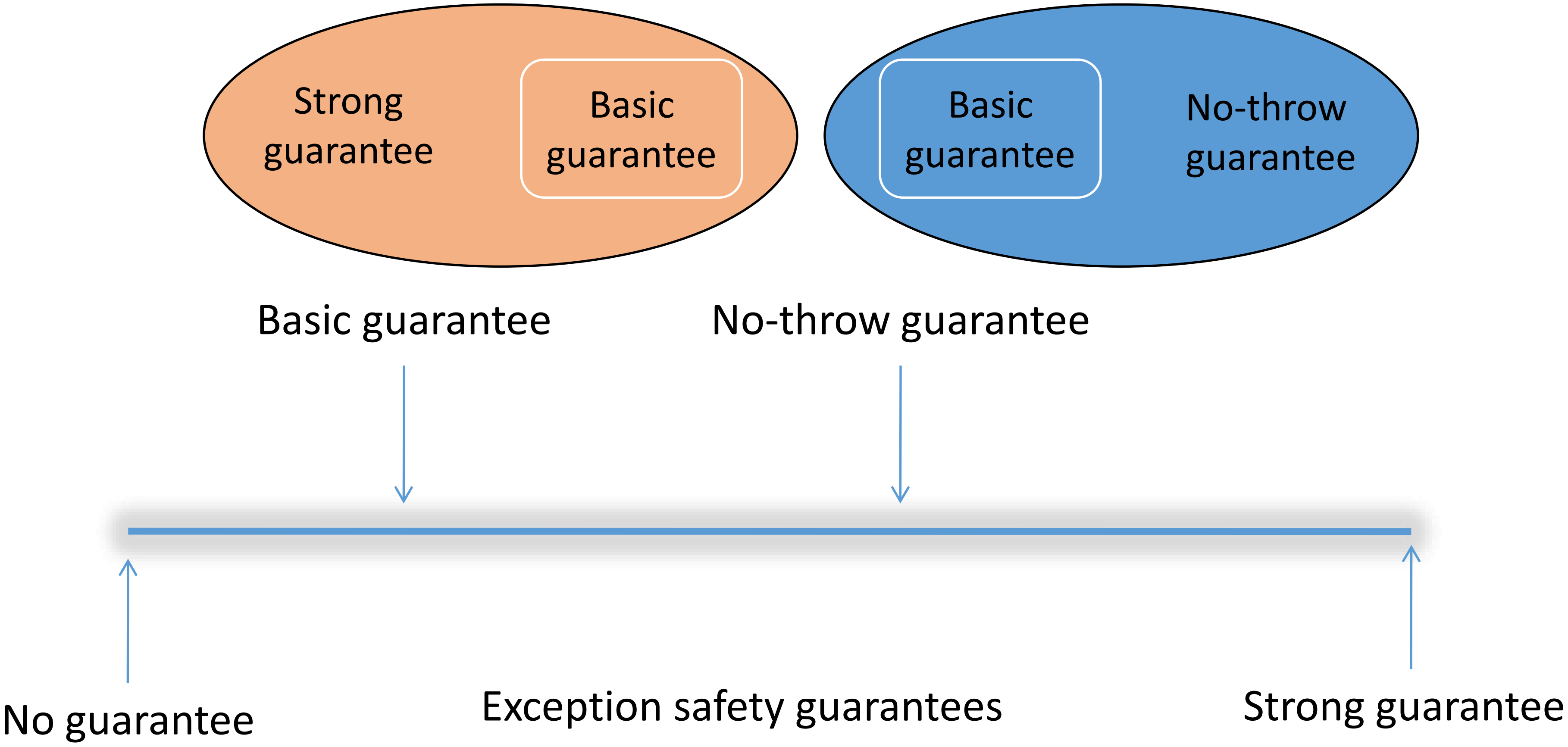
- An exception is an object of some class representing an exceptional occurrence.

```
class Matherr { };
class Overflow :public Matherr { };
class Underflow :public Matherr { };
class Zerodivide :public Matherr { };
// ...
```

Exception safety

- An operation on an object is said to be ***exception safe*** *if that operation leaves the object in a valid* state when the operation is terminated by throwing an exception.

  - Class invariant
    - Dollar
    - Range
    - Date
    - string
    - vector



Strong guarantee

Basic guarantee

Basic guarantee

No-throw guarantee

Basic guarantee

No-throw guarantee

No guarantee

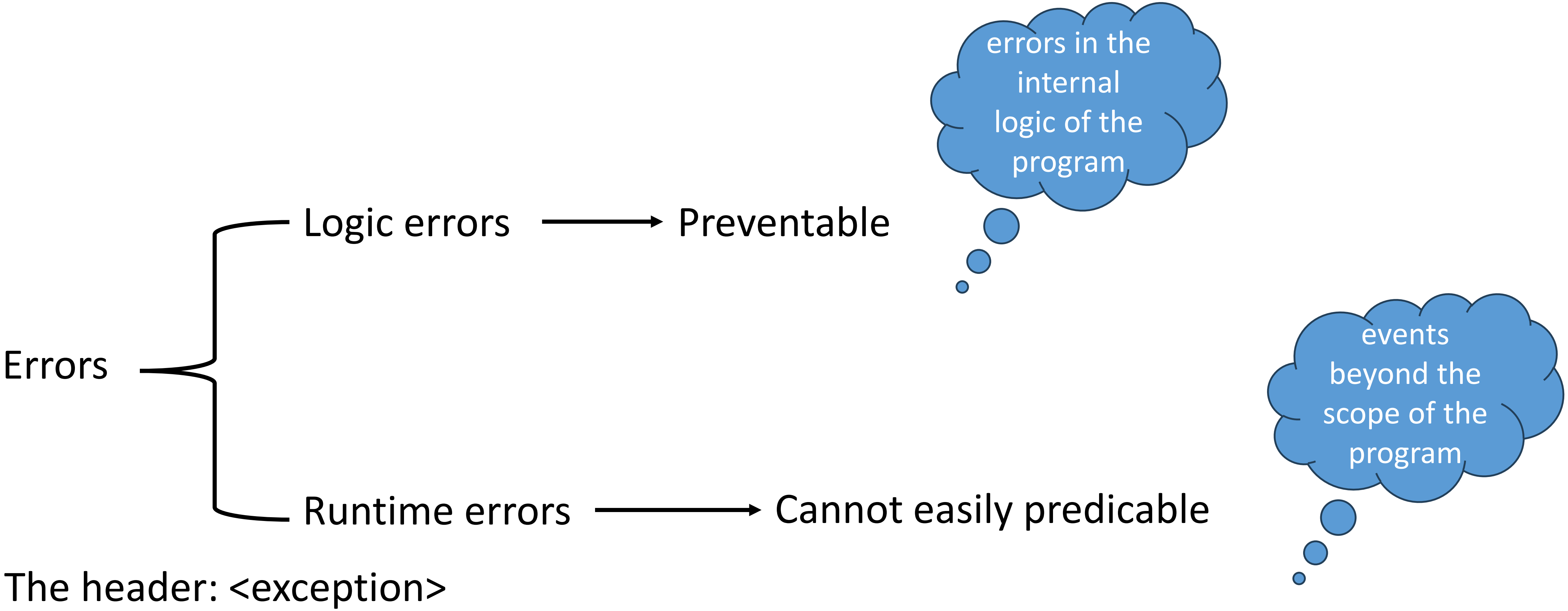
Exception safety guarantees

Strong guarantee

Standard library exception safety guarantees

- Standard library implementer vs. standard library user
- *Basic guarantee* for all operations.
- *Strong guarantee* for key operations.
- *No-throw guarantee* for some operations.

Standard library Exception classes



Errors

Logic errors → Preventable

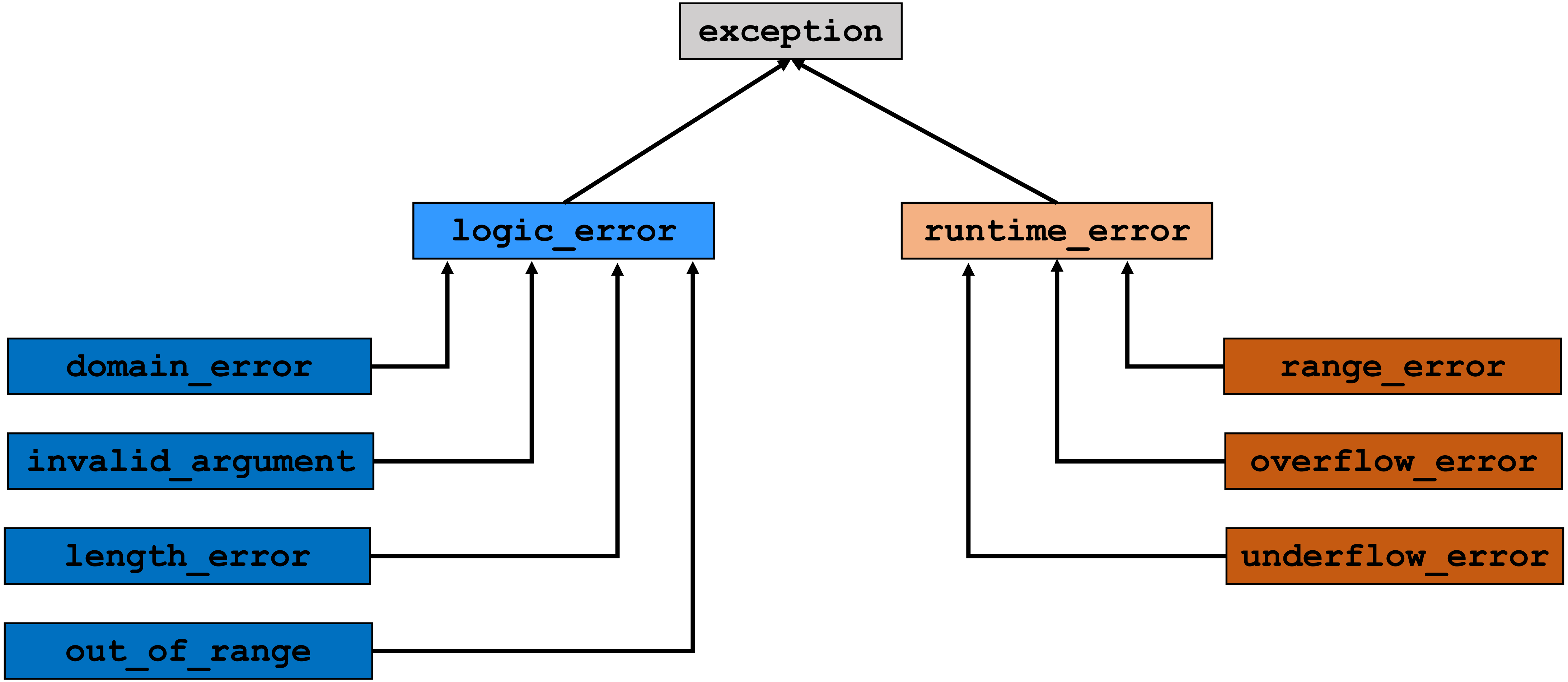
errors in the internal logic of the program

Runtime errors → Cannot easily predicable

events beyond the scope of the program

- The header: <exception>
- The header: <stdexcept>

standard library Exception classes



Example: length_error

1

```
class Vector {
    int* elem_;
    int size_;
public:
    Vector(int sz)
    {
        elem_ = new int[size_ = sz];
    }
};
```

2

```
void f()
{
    Vector v(42);
}
```

3

```
Vector::Vector(int sz)
{
    if (sz < 0)
        throw length_error
                {"Vector constructor: negative size"};
    elem_ = new double[sz];
    size_ = ss;
}
```

4

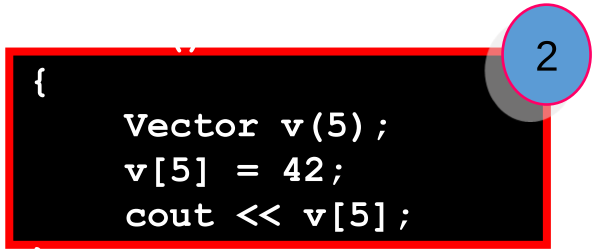
```
void test(int n)
{
  using std::length_error;
  using std::bad_alloc;
  try {
    Vector v(n);
  }
  catch (length_error& err) {
    // ... handle negative size ...
  }
  catch (bad_alloc& err) {
    // ... handle memory exhaustion ...
  }
}
void run()
{
  test(-27);
  // throws length_error (-27 is too small)
  test(1'000'000'000);
  // may throw bad_alloc
  test(10); // likely OK
}
```

Example: out_of_range

1

```
class Vector {
    int* elem_;
    int size_;
public:
    int operator[](int i) const; // getter
    int& operator[](int i); // setter
    void push_back(int val); // add a new element
};
```

2

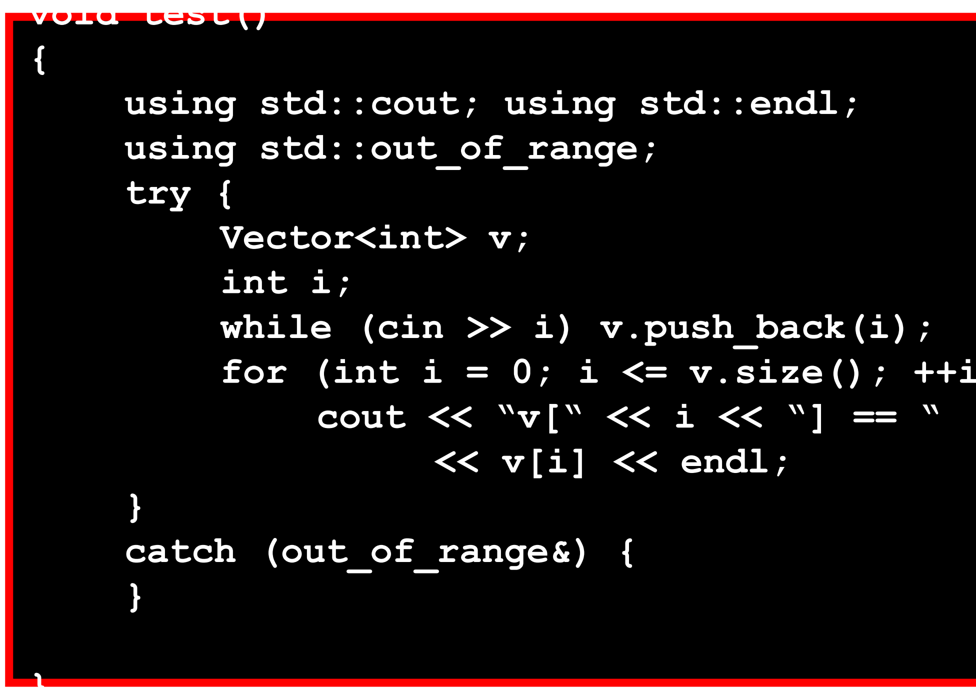


```
{
      Vector v(5);
      v[5] = 42;
      cout << v[5];
```

3

```
int Vector::operator(int i) const
{
    if (i < 0 || i >= size_)
        throw out_of_range{“Element access: out of range error"};
    return elem_[i];
}

// …
```



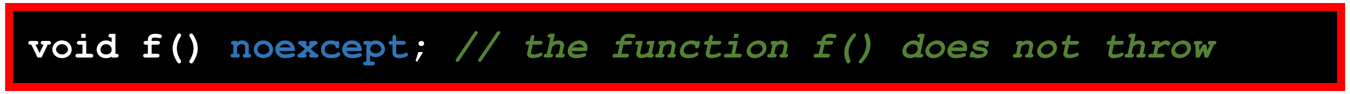
```
void test()
{
    using std::cout; using std::endl;
    using std::out_of_range;
    try {
        Vector<int> v;
        int i;
        while (cin >> i) v.push_back(i);
        for (int i = 0; i <= v.size(); ++i
            cout << “v[“ << i << “] == “
                 << v[i] << endl;
    }
    catch (out_of_range&) {
    }
```

Preventing exception propagation: noexcept

- C++ function
  - non-throwing exceptions
  - potentially throwing exceptions

- noexcept specifier

- If a function cannot throw an exception or if the program isn't written to handle exceptions thrown by a function, that function can be declared noexcept.

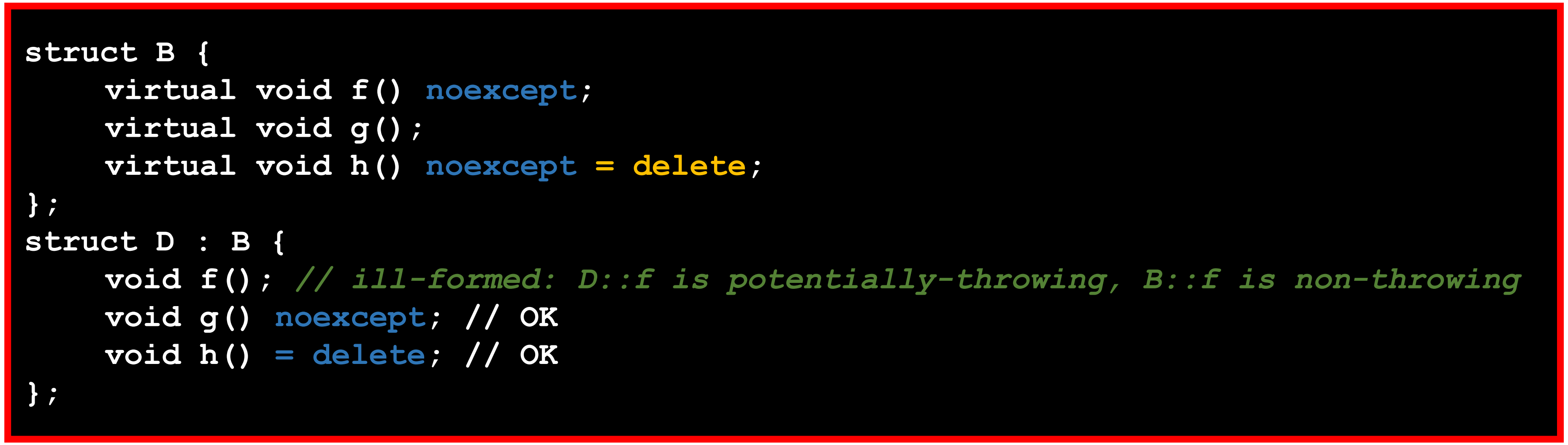


```
void f() noexcept; // the function f() does not throw
```

- Declaring a function noexcept can be most valuable for a programmer reasoning about a program and for a compiler optimizing a program.
- An optimizer need not worry about control paths from exception handling.

noexcept cont.

- If a virtual function is non-throwing, all declarations, including the definition, of every overridden must be non-throwing as well, unless the overridden is defined as deleted:



```
struct B {
    virtual void f() noexcept;
    virtual void g();
    virtual void h() noexcept = delete;
};
struct D : B {
    void f(); // ill-formed: D::f is potentially-throwing, B::f is non-throwing
    void g() noexcept; // OK
    void h() = delete; // OK
};
```

noexcept in the standard library

- noexcept is widely and systematically used in the standard library to improve performance and clarify requirements.

exception handling: General guidelines

- For effective error handling, the language mechanisms must be used based on a strategy.

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

Q & A