

3. Modularity

Sven Gestegård Robertz  
Computer Science, LTH

2019



## Outline

- 1 Source code organization
- 2 Stack allocation
- 3 Error handling
  - Exceptions
  - Catching exceptions
  - Throwing exceptions
  - Exceptions and resource management
  - Specification of exceptions
  - Static assert
- 4 Input and output
- 5 namespace

## Program organization

- ▶ A program consists of many separately developed parts
  - ▶ user-defined types
  - ▶ functions
  - ▶ templates
- ▶ managed by clearly defined interactions.
- ▶ Separate
  - ▶ interfaces
  - ▶ implementations

## Program organization Free functions vs. member functions

- ▶ Operations on a class can be expressed as
  - ▶ member functions
    - ▶ use for operations that need access to the representation
  - ▶ free functions
    - ▶ can be added without affecting other users of the class

## General program structure Organizing a program in several files

- ▶ Handle declarations and definitions
  - ▶ An entity must be defined at most once
  - ▶ The declaration is needed in all places it is used
  - ▶ `#include`

## The One Definition Rule

- ▶ For a *translation unit*, there must be no more than one definition for
  - ▶ a template
  - ▶ a type
  - ▶ a function, or
  - ▶ an object
- ▶ In the entire *program*, an object or *non-inline* function cannot have more than one definition.
- ▶ If an object or function is used, it must have a definition.
- ▶ Types and templates can be defined in multiple translation units, but must be the same.

## General program structure

### Example: Header file

Minimal example:

*mean* – a mean value library

- ▶ mean.h
- ▶ mean.cc

Use:

- ▶ main.cc

#### mean.h: declarations

```
double mean(double x1, double x2);
double mean(int x1, int x2);
```

## General program structure

### Example: Source code file

#### mean.cc: definitions

```
#include "mean.h" // make declarations visible so that the
                  // compiler can check that they agree
double mean(double x1, double x2)
{
    return (x1+x2)/2;
}

double mean(int x1, int x2)
{
    return static_cast<double>(x1 + x2) / 2;
}
```

## General program structure

### Example: main program

#### main.cc: use

```
#include <iostream>
using std::cout;
using std::endl;

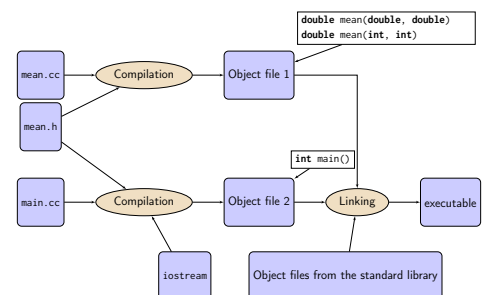
#include "mean.h" // make declarations visible
                  // to be able to use them

int main()
{
    double a=2.3;
    double b=3.9;
    int m=3;
    int n=4;
    cout << mean(a, b) << endl;
    cout << mean(m, n) << endl;
}
```

## General program structure

### Separate compilation

- ▶ Function declarations are placed in *header files* (.h)
- ▶ The implementation is split into multiple source files (.cc)
- ▶ Separate compilation of each source file(.cc)
- ▶ Linking the program



## File structure for classes

### include guards

Class definitions are placed in header files (.h or .hpp)

- ▶ All users of a type need the definition
- ▶ A header file can be included more than once (e.g., via other header files)
- ▶ To avoid defining a type multiple times use *include guards*:

```
#ifndef F00_H
#define F00_H
//...
class Foo {
//...
};
#endif
```

Member functions are placed in a source file (.cc or .cpp)

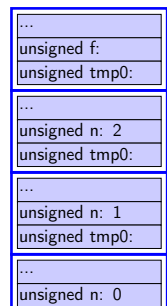
## Memory allocation

### stack allocation

```
unsigned fac(unsigned n)
{
    if(n == 0)
        return 1;
    else return n * fac(n-1);
}

int main()
{
    unsigned f = fac(2);
    cout << f;
    return 0;
}
```

main()  
fac()  
fac()  
fac()



## Error handling

### Three levels of error handling

- 1 Directly handle the error locally and continue execution
- 2 Categorize and pass error to another module that is expected to handle it
- 3 Identify the error, give an error message, and crash the program (*"fail-fast"*, e.g., `assert`)

Level 2: exceptions (or return values)

## Throwing exceptions

### Example: checking arguments in the `Vector` class

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

int& Vector::at (int i) {
    if (i<0 || i>=sz) throw out_of_range("Vector::at");
    return elem[i];
}
```

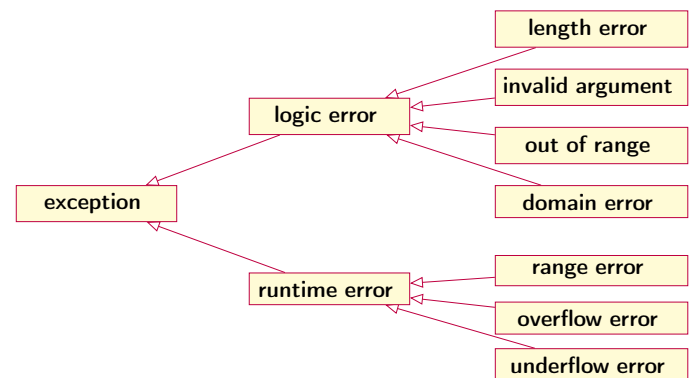
- ▶ *NB: to allow checking arguments, we use a signed integer type for values that should always be positive*
- ▶ `Vector` cannot reasonably handle the error locally, only the caller can know why it passed a certain argument

## Exceptions

- ▶ Error handling is done with `throw` and `catch`. Like Java.
- ▶ *"stack unwinding"* until a matching `catch` is found.
- ▶ When an exception is thrown, activation records are popped off the stack until a function containing a matching `catch` is found. (*"stack unwinding"*)
- ▶ If an exception is not caught, the program crashes. (by calling `std::terminate()`.)
- ▶ Standard classes for exceptions: `#include <stdexcept>`

## The exception classes of the standard library

Class hierarchy for classes in `<stdexcept>`



## Error handling

### Catching exceptions

```
try {
    // Code that may throw
}
catch (some_exception&) {
    // Code handling some_exception
}
catch (another_exception&) {
    // Code handling another_exception
}
catch (...) {
    // default/generic exception handling
}
```

The first `catch` clause with a matching type is selected.  
⇒ Catch derived classes before the base class.

... is valid C++, matches anything

## Catching exceptions

### Example:

```
try {
    cout << "Enter a number: ";
    int i;
    if (cin >> i) {
        int r = f(i);
        cout << "Result: " << r << endl;
    }
}
catch(std::overflow_error&) {
    cout << "Overflow error\n";
}
catch(std::exception& e) {
    cout << typeid(e).name() << ": " << e.what() << endl;
}
```

## Catching exceptions

### Exempel:

```
try {
    cout << "Enter a number: ";
    int i;
    if (cin >> i) {
        int r = f(i);
        cout << "Result: " << r << endl;
    }
}
catch(std::overflow_error&) {
    cout << "Overflow error\n";
}
catch(std::exception& e) {
    cout << typeid(e).name() << ": " << e.what() << endl;
}

```

predefined function in the class exception

## Catching exceptions ... and rethrowing

```
try{
    do_something();
}
catch {std::length_error& le) {
    // handle length error
}
catch {std::out_of_range&) {
    // handle out_of_range
}
catch (...) {
    throw; // default: pass on
}

```

## Throwing exceptions

### Creating own exceptions as subclasses

```
#include<stdexcept>

class communication_error : public runtime_error {
public:
    communication_error(const string& mess = "")
        : runtime_error(mess) {}
};

```

### Throwing

```
throw communication_error("Checksum error");

```

## Throwing exceptions

### Creating custom exceptions

```
struct MyOwnException{
    MyOwnException(const std::string& msg, int val)
        : m{msg},v{val} {}
    std::string m;
    int v;
};

```

### Using custom exceptions

```
void f() {
    throw MyOwnException("An error occurred", 17);
}

void test1() {
    try{
        f();
    } catch(MyOwnException &e) {
        cout << "Exception: " << e.m << " - " << e.v << endl;
    }
}

```

## Catching exceptions

Resource mangement: destructors and "stack unwinding"

```
struct Foo {
    int x;
    Foo(int ix) :x{ix} {
        cout << "Foo("<<x<<")\n";
    }
    ~Foo() {
        cout << "~Foo("<<x<<")\n";
    }
};

void test(int i)
{
    Foo f(i);
    if(i == 0) {
        throw std::out_of_range("noll?");
    } else {
        Foo g(100+i);
        test(i-1);
        cout << "after call to test("
            << i-1 << ")\n";
    }
}

```

```
int main() {
    Foo f(42);
    try{
        Foo g(17);
        test(2);
    } catch(std::exception& e) {
        cout<<e.what()<< endl; }
}
    Foo(42)
    Foo(17)
    Foo(2)
    Foo(102)
    Foo(1)
    Foo(101)
    Foo(0)
    ~Foo(0)
    ~Foo(101)
    ~Foo(1)
    ~Foo(102)
    ~Foo(2)
    ~Foo(17)
    noll?
    ~Foo(42)

```

## Catching exceptions

Resource mangement: destructors and "stack unwinding"

```
struct Foo {
    int x;
    Foo(int ix) :x{ix} {
        cout << "Foo("<<x<<")\n";
    }
    ~Foo() {
        cout << "~Foo("<<x<<")\n";
    }
};

void test(int i)
{
    Foo f(i);
    if(i == 0) {
        throw std::out_of_range("noll?");
    } else {
        Foo g(100+i);
        test(i-1);
        cout << "after call to test("
            << i-1 << ")\n";
    }
}

```

```
int main() {
    Foo f(42);
    try{
        Foo g(17);
        test(2);
    } catch(std::exception& e) {
        cout<<e.what()<< endl; }
}
    Foo(42)
    Foo(17)
    Foo(2)
    Foo(102)
    Foo(1)
    Foo(101)
    Foo(0)
    ~Foo(0)
    ~Foo(101)
    ~Foo(1)
    ~Foo(102)
    ~Foo(2)
    ~Foo(17)
    noll?
    ~Foo(42)

```

## Catching exceptions

Resource mangement: destructors and "stack unwinding"

```
struct Foo {
    int x;
    Foo(int ix) :x{ix} {
        cout << "Foo("<<x<<")\n";
    }
    ~Foo() {
        cout << "~Foo("<<x<<")\n";
    }
};

void test(int i)
{
    Foo f(i);
    if(i == 0) {
        throw std::out_of_range("noll?");
    } else {
        Foo g(100+i);
        test(i-1);
        cout << "after call to test("
            << i-1 << ")\n";
    }
}
```

```
int main() {
    Foo f(42);
    try{
        Foo g(17);
        test(2);
    } catch(std::exception& e) {
        cout<<e.what()<< endl; }
}

    Foo(42)
    Foo(17)
    Foo(2)
    Foo(102)
    Foo(1)
    Foo(101)
    Foo(0)
    ~Foo(0)
    ~Foo(101)
    ~Foo(1)
    ~Foo(102)
    ~Foo(2)
    ~Foo(17)
    noll?
    ~Foo(42)
```

## Catching exceptions

Resource mangement: destructors and "stack unwinding"

```
struct Foo {
    int x;
    Foo(int ix) :x{ix} {
        cout << "Foo("<<x<<")\n";
    }
    ~Foo() {
        cout << "~Foo("<<x<<")\n";
    }
};

void test(int i)
{
    Foo f(i);
    if(i == 0) {
        throw std::out_of_range("noll?");
    } else {
        Foo g(100+i);
        test(i-1);
        cout << "after call to test("
            << i-1 << ")\n";
    }
}
```

```
int main() {
    Foo f(42);
    try{
        Foo g(17);
        test(2);
    } catch(std::exception& e) {
        cout<<e.what()<< endl; }
}

    Foo(42)
    Foo(17)
    Foo(2)
    Foo(102)
    Foo(1)
    Foo(101)
    Foo(0)
    ~Foo(0)
    ~Foo(101)
    ~Foo(1)
    ~Foo(102)
    ~Foo(2)
    ~Foo(17)
    noll?
    ~Foo(42)
```

## Catching exceptions

Resource mangement: destructors and "stack unwinding"

```
struct Foo {
    int x;
    Foo(int ix) :x{ix} {
        cout << "Foo("<<x<<")\n";
    }
    ~Foo() {
        cout << "~Foo("<<x<<")\n";
    }
};

void test(int i)
{
    Foo f(i);
    if(i == 0) {
        throw std::out_of_range("noll?");
    } else {
        Foo g(100+i);
        test(i-1);
        cout << "after call to test("
            << i-1 << ")\n";
    }
}
```

```
int main() {
    Foo f(42);
    try{
        Foo g(17);
        test(2);
    } catch(std::exception& e) {
        cout<<e.what()<< endl; }
}

    Foo(42)
    Foo(17)
    Foo(2)
    Foo(102)
    Foo(1)
    Foo(101)
    Foo(0)
    ~Foo(0)
    ~Foo(101)
    ~Foo(1)
    ~Foo(102)
    ~Foo(2)
    ~Foo(17)
    noll?
    ~Foo(42)
```

## Catching exceptions

Resource mangement: destructors and "stack unwinding"

```
struct Foo {
    int x;
    Foo(int ix) :x{ix} {
        cout << "Foo("<<x<<")\n";
    }
    ~Foo() {
        cout << "~Foo("<<x<<")\n";
    }
};

void test(int i)
{
    Foo f(i);
    if(i == 0) {
        throw std::out_of_range("noll?");
    } else {
        Foo g(100+i);
        test(i-1);
        cout << "after call to test("
            << i-1 << ")\n";
    }
}
```

```
int main() {
    Foo f(42);
    try{
        Foo g(17);
        test(2);
    } catch(std::exception& e) {
        cout<<e.what()<< endl; }
}

    Foo(42)
    Foo(17)
    Foo(2)
    Foo(102)
    Foo(1)
    Foo(101)
    Foo(0)
    ~Foo(0)
    ~Foo(101)
    ~Foo(1)
    ~Foo(102)
    ~Foo(2)
    ~Foo(17)
    noll?
    ~Foo(42)
```

## Catching exceptions

Resource mangement: destructors and "stack unwinding"

```
struct Foo {
    int x;
    Foo(int ix) :x{ix} {
        cout << "Foo("<<x<<")\n";
    }
    ~Foo() {
        cout << "~Foo("<<x<<")\n";
    }
};

void test(int i)
{
    Foo f(i);
    if(i == 0) {
        throw std::out_of_range("noll?");
    } else {
        Foo g(100+i);
        test(i-1);
        cout << "after call to test("
            << i-1 << ")\n";
    }
}
```

```
int main() {
    Foo f(42);
    try{
        Foo g(17);
        test(2);
    } catch(std::exception& e) {
        cout<<e.what()<< endl; }
}

    Foo(42)
    Foo(17)
    Foo(2)
    Foo(102)
    Foo(1)
    Foo(101)
    Foo(0)
    ~Foo(0)
    ~Foo(101)
    ~Foo(1)
    ~Foo(102)
    ~Foo(2)
    ~Foo(17)
    noll?
    ~Foo(42)
```

## Specifying exceptions in C++11

The keyword **noexcept** specifies if a function may throw or not. No specification is equal to **noexcept(false)**.

In the function declaration

```
struct Foo {
    void f();
    void g() noexcept;
};
```

and in the function definition

```
#include <stdexcept>
void Foo::f() {
    throw std::runtime_error("f failed");
}
void Foo::g() noexcept{
    throw std::runtime_error("g lied and failed");
}
```

## Exception specification

### Example usage

```
#include<typeinfo> // for typeid

void test_noexcept()
{
    Foo f;

    try {
        f.f();
    }catch(std::exception &e) {
        cout << typeid(e).name() << ": " << e.what() << endl;
    }
    try {
        f.g();
    }catch(std::exception &e) {
        cout << typeid(e).name() << ": " << e.what() << endl;
    }
    cout << "done\n";
}

St13runtime_error: f failed
terminate called after throwing an instance of 'std::runtime_error'
what(): g lied and failed
```

## Exception specification

### older C++, do not use

Older C++ had "exception lists" for a function: the types of exceptions that may be generated by the function are specified with the keyword **throw**.

### Example of exception list:

```
int f(int) throw(typ1, typ2, typ3) {
    //...
    throw typ1("Error of type 1 occurred");
    //...
    throw typ2("Error of type 2 occurred");
    //...
    throw typ3("Error of type 3 occurred");
}
```

No list  $\Rightarrow$  Any type of exception may be thrown  
Empty list (throw())  $\Rightarrow$  No exceptions may be thrown

## Rules of thumb for exceptions

- ▶ Consider error handling early in the design process
- ▶ Use specific exception types, not built-in types. (do not `throw 17;`, `throw false;`, etc.)
- ▶ "Throw by value, catch by reference"
- ▶ If a function should not throw, declare `noexcept`.
- ▶ Specify *invariants* for your types
  - ▶ The constructor establishes the invariant, or throws.
  - ▶ Member functions can rely on the invariant.
  - ▶ Member functions must not break the invariant.
  - ▶ Example: `Vector`
    - ▶ the size `sz` is a positive number
    - ▶ the array `elem` points to has size `sz`
    - ▶ if the allocation of the array fails `std::bad_alloc` is thrown

If something can be checked at compile-time, use `static_assert`.

## Static assert

If something can be checked at compile-time, use `static_assert`.

```
unsigned char fourth_byte(int x)
{
    static_assert(sizeof(int) >= 4, "int too small");
    return (x >> 24) & 0xff;
}
```

On a platform where `int` is 16 bit, this would not compile:

```
// error: static assertion failed: int too small
```

## Stream I/O

- ▶ The C++ standard library contains facilities for
  - ▶ Structured I/O ("formatted I/O")
    - ▶ reading values of a certain type, `T`
    - ▶ overload `operator>>`(`istream&`, `T&`) and
    - ▶ `operator<<`(`ostream&`, `const T&`)
  - ▶ Character I/O ("raw I/O")
    - ▶ `istream& getline(istream&, string&)`
    - ▶ `istream& istream::getline(char*, streamsize)`
    - ▶ `int istream::get()`
    - ▶ `istream& istream::ignore()`
    - ▶ ...
- ▶ NB! `getline()` as free function and member of `istream`.
- ▶ `std::getline()` has an overload for using another delimiter than `newline`.
- ▶ Choose raw or formatted I/O based on your application

## namespace

- ▶ Limit visibility of names
  - ▶ expressing which functions/classes/objects belong together
  - ▶ reduce risk of name clashes
  - ▶ cf. package in Java
- ▶ Accessing names in namespaces:
  - ▶ qualified name (with scope operator): `std::cout`
  - ▶ **using** declaration: `using std::cout;`
    - ▶ brings in a single name into the current scope
  - ▶ **using** directive: `using namespace std;`
    - ▶ brings in all names in namespace `std` into the current scope
    - ▶ avoid in general, or use in limited scope
  - ▶ never write **using**-directives in header files
    - ▶ introduces names in user code
- ▶ Namespaces *can be extended*
  - ▶ Except (with some exceptions) `std` ( $\Rightarrow$  undefined behaviour)

## namespace Example

### declarations (.h)

```
namespace foo {  
    void test();  
}  
  
namespace bar {  
    void test();  
}
```

```
int main()  
{  
    foo::test();  
    bar::test();  
    using namespace foo;  
    test();  
}
```

### definitions (.cc)

```
using std::cout;  
using std::endl;  
  
namespace foo {  
    void test()  
    { cout << "foo::test()\n";  
    }  
}  
  
void bar::test()  
{ cout << "bar::test()\n";  
}
```

```
foo::test()  
bar::test()  
foo::test()
```

## namespace

- ▶ Unnamed namespaces
  - ▶ local to a particular file (also if **#included**)
  - ▶ is used to hide names (cf. **static** in C)
  - ▶ names clash with global names

```
namespace foo {  
    void test()  
    { cout << "foo::test()\n";  
    }  
}  
  
int main()  
{  
    test();  
    foo::test();  
    ::test();  
}  
  
namespace {  
    void test()  
    { cout << "::test()\n";  
    }  
}
```

- ▶ Alternative names for namespaces (*namespace aliases*):  
`namespace my_name=a_really_long_and_weird_namespace_name;`

## Argument Dependent Lookup (ADL)

Name lookup is done in *enclosing scopes*, but...

```
namespace test{  
    struct Foo{  
        Foo(int v) :x{v} {}  
        int x;  
    };  
    std::ostream& operator<<(std::ostream& o, const Foo& f) {  
        return o << "Foo(" << f.x << ")";  
    }  
} // namespace test
```

```
int main()  
{  
    test::Foo f(17);  
    cout << f << endl;  
}
```

- ▶ The function `operator<<(ostream&, const Foo&)` is not visible in `main()`.
- ▶ Through ADL it is found in the namespace of its argument (`test`).

## Argument Dependent Lookup (ADL)

```
namespace test{  
    struct Foo;  
    std::ostream& operator<<(std::ostream& o, const Foo& f);  
  
    void print(const Foo& f)  
    { cout << f << endl;  
    }  
    void print(int i)  
    { cout << i << endl;  
    }  
} // namespace test
```

- ▶ The functions `test::operator<<()` and `test::print(const Foo&)` are found through ADL.
- ▶ The function `test::print(int)` is not found.
- ▶ unless `using test::print`.

```
int main()  
{  
    test::Foo f(17);  
    print(f);  
    print(17);  
    test::print(17);  
}
```

## Suggested reading

References to sections in Lippman

Exceptions 5.6, 18.1.1

Namespaces 18.2

I/O 1.2, 8.1–8.2, 17.5.2

## Next lecture

References to sections in Lippman

Classes 2.6, 7.1.4, 7.1.5

Constructors 7.5–7.5.4

(Aggregate classes) ("C structs" without constructors) 7.5.5

Destructors 13.1.3

this and const p 257–258

inline 6.5.2, p 273

friend 7.2.1

static members 7.6

Copying 13.1.1

Assignment 13.1.2

Operator overloading 14.1 – 14.3