Programming in C/C++

- Streams & File I/O -

Streams (C++)

- Object oriented abstraction for sequential I/O
- Formatting/Manipulators
- File streams, String streams

Streams

EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

- Streams are an **abstraction**, they define a common **interface for input and output** using different **devices** like: files, keyboard, console, network, …
- Streams know how to communicate with their device
- Standard types are already supported by existing stream operators
- We have already seen that the standard libary contains streams:

Stream	Description
`std::cin`	Standard input stream
`std::cout`	Standard output stream – buffered
`std::cerr`	Standard error stream – unbuffered, immediate output, might “mingle” with cout

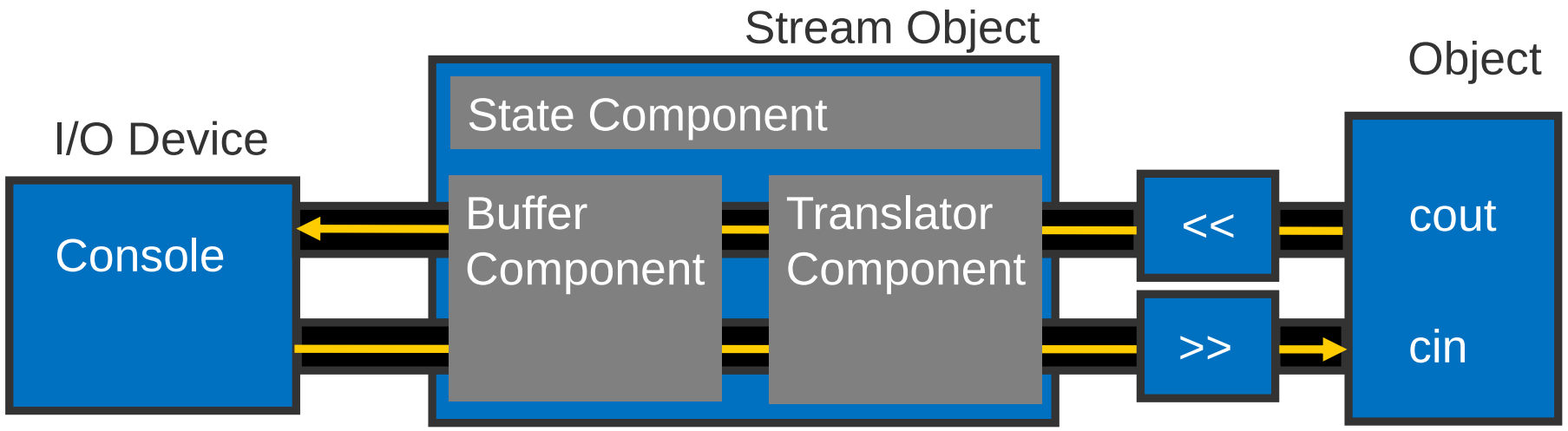
Default:

- Output to `cout/cerr` is printed on console
- Input from keyboard can be read with `cin`

Model for streams in C++

- Translator Component
  - **`basic_istream, basic_ostream:`** Implementation **`- operator<<, operator>>`**
- Buffer Component
  - **`basic_streambuf:`** Access **`to input/output buffer`**
- State Component
  - **`ios_base, basic_ios:`** formatting, (error) status

Stream classes are highly configurable.



Stream status

EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

- Every stream object stores its status (**`iostate`**)

- Status consists of three bit:
  - **badbit** set if stream is damaged (e.g., read/write error)
  - **failbit** set if read/write operation fails
  - **eofbit** set if EOF = end of file is read

- Query status using predicates:

`bool good();`	true, if no status bit set (valid stream)
`bool bad();`	true, if badbit set
`bool fail();`	true, if failbit set
`bool eof();`	true, if eofbit set

- `clear(iostate)` *resets the stream and unsets the passed bit*

Stream formatting

EBERHARD KARLS UNIVERSITÄT TÜBINGEN

- Streams have an internal state how input/output should be formated
- This state can be set with flags in the base class `ios_base` of our stream:

```
class ios_base
{
public:
    typedef <...> fmtflags;
    static const fmtflags
    …
    boolalpha,     // use symbolic names for bools
    dec, hex, oct, // decimal base for numbers
    …
};
```

- Changing **`fmtflags`** changes the format

Example:

```
    bool t = true, f = false;
    cout << t << "/" << f << endl; // outputs 1/0
    cout.setf(ios_base::boolalpha);// set flag
    cout << t << "/" << f << endl; // outputs true/false
```

Stream manipulators

EBERHARD KARLS UNIVERSITÄT TÜBINGEN

- Manipulators are an easier way to change status of a stream compared to **`setf`**
- Manipulators are objects that can be applied to a stream using the stream operators
- Example: `boolalpha`

```
cout << boolalpha << t << "/" << f << endl;
```

Same result as on last slide with setf().

- `boolalpha` changes state of stream object -> subsequent output will be affected
- To undo boolalpha use manipulator `noboolalpha`

Stream manipulators

- Similar to **boolalpha** we can change the base for number output using the manipulators **hex**, **oct** and **dec**
- Manipulators **showbase** und **noshowbase** turns output of base on and off (prefix 0 für octal, 0x für hexadecimal)
- Manipulators **showpos** und **noshowpos** show/hide the plus sign of positive numbers (only base 10)

```
int j = 123;
cout << hex << j << endl;
cout << oct << j << endl;
cout << dec << j << endl;
cout << showbase;
cout << hex << j << endl;
cout << oct << j << endl;
cout << dec << j << endl;
cout << showpos;
cout << j << endl;
```

```
7b
173
123
0x7b
0173
123
+123
```

Stream manipulators `#include <iomanip>`

EBERHARD KARLS UNIVERSITÄT TÜBINGEN

- Manipulators can configure:
  - **Precision**: Number of digits
  - **Style**: Fixed point or scientific notation

of floating point numbers.

- Manipulators **`scientific`** and **`fixed`** switch style
- Manipulator **`setprecision(int)`** sets the precision
- Get current precision with **`precision()`**

```
double x = 1234.56789;

cout << x << endl;

cout << cout.precision() << endl;

cout << setprecision(4);
cout << x << endl;

cout << scientific << x << endl;
```

```
1234.57
6
1235
1.2346e+03
```

Stream manipulators

`#include <iomanip>`

EBERHARD KARLS UNIVERSITÄT TÜBINGEN

- Printing tabular data also possible with streams
- Manipulators:
  - **`setw(int)`** defines the minimum with of the next output
  - **`left/right`**: align left/right
  - **`setfill(char)`**: set fill char (default: space)
- **`setw`** is special because it only affects the next output and does not change the state of the stream permanently.

```
cout << setw(10) << right;
cout << 1234.56 << endl;

cout << setfill('#');
cout << setw(10);
cout << 2345.67 << endl;
```

```
   1234.56
###2345.67
```

Stream manipulators - input <iomanip>

EBERHARD KARLS UNIVERSITÄT TÜBINGEN

- There are also manipulators for input streams
- **`skipws`** *(skip whitespace)* and **`noskipws`** defines if whitespaces (space/tab) will be ignored.
- A stream operator for single **`char`** also exist. Allows processing input character by character.
- Simple **`while`**-Loop reads all characters until end-of-file (CTRL-D) :

  `while (cin >> c) cout << c;`

```
char c;
cin >> skipws;
while (cin >> c)
{
  cout << c;
}
```

```
Input:
a b     c     d
```

```
Output:
abcd
```

```
Output with noskipws:
a b     c     d
```

Stream manipulators

EBERHARD KARLS UNIVERSITÄT TÜBINGEN

`<iomanip>`

boolalpha noboolalpha	switches between textual and numeric representation of booleans
showbase noshowbase	controls whether prefix is used to indicate numeric base
showpoint noshowpoint	controls whether decimal point is always included in floating-point representation
showpos noshowpos	controls whether the + sign used with non-negative numbers
skipws noskipws	controls whether leading whitespace is skipped on input
uppercase nouppercase	controls whether uppercase characters are used with some output formats
unitbuf nounitbuf	controls whether output is flushed after each operation
internal left right	sets the placement of fill characters
dec hex oct	changes the base used for integer I/O
fixed scientific hex float defaultfloat	changes formatting used for floating-point I/O
ws	consumes whitespace
ends	outputs '\0'
flush	flushes the output stream
endl	outputs '\n' and flushes the output stream
emit_on_flush noemit_on_flush	controls whether a stream's basic_syncbuf emits on flush
flush_emit	flushes a stream and emits the content if it is using a basic_syncbuf
resetiosflags	clears the specified ios_base flags
setiosflags	sets the specified ios_base flags
setbase	changes the base used for integer I/O
setfill	changes the fill character
setprecision	changes floating-point precision
setw	changes the width of the next input/output field
get_money	parses a monetary value
put_money	formats and outputs a monetary value
get_time	parses a date/time value of specified format
put_time	formats and outputs a date/time value according to the specified format
quoted	inserts and extracts quoted strings with embedded spaces

File I/O Streams <fstream>

EBERHARD KARLS UNIVERSITÄT TÜBINGEN

- C++ file streams allow permanent storing of data on disc and reading it back in later.

Stream	Description
std::ifstream	Stream to read form input file
std::ofstream	Stream to write to output file
std::fstream	Stream to read/write to file

std::ifstream – input file stream

Basic use:

- Create an instance by passing a filename to constructor:

```
std::ifstream ifs("input.txt");
```

- Read from file like from cin
- If file doesn‘t exist/can‘t be read, stream is in invalid state
- Detect invalid state:

```
if (!ifs) … // Error opening file?
```

File I/O Streams `<fstream>`

EBERHARD KARLS UNIVERSITÄT TÜBINGEN

- C++ file streams allow permanent storing of data on disc and reading it back in later.

Stream	Description
`std::ifstream`	Stream to read form input file
`std::ofstream`	Stream to write to output file
`std::fstream`	Stream to read/write to file

`std::ofstream` – output file stream

Basic use, similar to `ifstream`:

- Create an instance by passing a filename to constructor:

```
std::ofstream ofs("output.txt");
```

- Write to file like from **`cout`**:

```
std::ofstream os("output.txt");
os << "Hello World!" << endl;
os.close();
```

open()

- If we don't want to open a file immediately, we can call the default constructor of ifstream.

Access mode	
`in`	open for input
`out`	open for output
`app`	append: seek to end before every write
`ate`	seek to end only once after open
`trunc`	truncate stream at open
`binary`	binary mode, otherwise text file

```
string inFileName;
ifstream inFile;            // default construct ifstream
infile.open(inFileName.c_str(), ios::in); // now: open and bind ifstream to file
if (!infile) {
    cerr << “Error : unable to open : “ << inFileName << endl;
    return -1;
}
```

Combine Access Modes

EBERHARD KARLS UNIVERSITÄT TÜBINGEN

- Access modes can be combined

```
ios::out                 // default for ofstream
ios::out | ios::app
ios::out | ios::trunc    // same as ios::out
ios::in                  // default for ifstream
ios::in  | ios::out      // default for fstream (no truncation)
ios::in  | ios::out | ios::trunc
```

- **`ate`** can always be added (jump to the end once when file is opened)

Example: Character-at-a-time Copy

EBERHARD KARLS UNIVERSITÄT TÜBINGEN

- istream::get(): read single byte

```
bool charCopy(istream & _inS, ostream & _outS) {
   if (!_inS || !_outS) { return false; }  // check status

   _inS.clear();                           // clear any errors
   _outS.clear();
   char curC;
   while (!_inS.eof()) {
     _inS.get(curC); // use get instead of >> to extract white spaces
     if (_inS.fail()) { return false; }
     _outS << curC;
   }
   return true;
 }
 ifstream inFile(inFileName.c_str());
 ofstream outFile(outFileName.c_str());
 charCopy(inFile, outFile);
 inFile.close();
 outFile.close();
```

Example: Buffered Binary Copy

EBERHARD KARLS UNIVERSITÄT TÜBINGEN

- faster: read()/write() multiple bytes at once

```
const static int BUF_SIZE = 4096;
  bool bufferedCopy(istream &_inS, ostream &_outS) {
  char buf[BUF_SIZE];
    if (!_inS || !_outS)
      return false; // check status
  _inS.clear();
  _outS.clear(); // clear any errors
  do {
    _inS.read(&buf[0], BUF_SIZE);        // Read at most n bytes into buf
    _outS.write(&buf[0], _inS.gcount()); // then write the buf to output
  } while (_inS.gcount() > 0);
  return true;
}
ifstream inFile(inFileName.c_str(), ios_base::in | ios_base::binary);
ofstream outFile(outFileName.c_str(), ios_base::out | ios_base::binary);
bufferedCopy(inFile, outFile);
inFile.close();
outFile.close();
```

stringstream <sstream>

`stringstream` uses a string for I/O

- Write and read to a stream in memory instead of file or console
- Used to parse text

```
// Get a line
while (getline(cin, line)) {
  istringstream streamLine(line);
  // Get individual white space separated tokens
  while (streamLine >> token) {
    // Process word
  }
}
```

- Or to convert numbers to **string**

```
stringstream s("123 578.78");
int n;
s >> n;
if (s.fail()) { cerr << "no number!" << endl; }
float x;
s >> x;
```

Extra: stream redirects

- Useful for debugging e.g., a lot of console output

Console Output – Redirection

EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

- redirect `stdout` to file

```
program > file.txt
```

- redirect `stderr` to file

```
program 2> file.txt
```

- redirect `stdout` & `stderr` to file

```
program &> file.txt
```

- redirect `stdout` and `stderr` to two separate files

```
program > file_stdout.txt 2> file_stderr.txt
```

- redirect `stdout` to `stderr`

```
program 1>&2
```

- redirect `stderr` to `stdout`

```
program 2>&1
```

Console Output – Pipe and Tee

EBERHARD KARLS UNIVERSITÄT TÜBINGEN

- Forward `stdout` to `stdin` of another program

```
program1 | program2
```

- Pipe to multiple files while keeping `stdout`

```
program 1 | tee file1.txt file2.txt
```

Serialization

- Making our class compatible with streams, own stream operator
- How to store and restore an object
- Binary representation
- Serialization of PODs

Stream I/O for User-defined Types

EBERHARD KARLS UNIVERSITÄT TÜBINGEN

- **Goal**: Make our user defined type compatible with streams

```
std::cout << myObj;
```

- How can we make this happen?

- **Part 1 of Solution**: Function overloading of the free functions:

```
ostream &operator<<(std::ostream &os, const T &_obj);
```

```
istream &operator>>(istream &is, T &obj);
```

- Syntax works but can we e.g., print protected or private member variables?

Stream I/O for User-defined Types – Example

EBERHARD KARLS UNIVERSITÄT TÜBINGEN

```
class Person {
  string lastName;
  string firstName;
  int age;
public:
  Person() = default;
  Person(const string &_ln, const string &_fn, int _age);
  // friend has access to private members
  friend ostream &operator<<(ostream &, const Person &); // out
  friend istream &operator>>(istream &_is, Person &_p);  // in
};

ostream &operator<<(ostream &_os, const Person &_p) {
  _os << _p.firstName << " " << _p.lastName << " " << _p.age;
  return _os;
}
…
 Person john{"John", "Dow", 13};
 cout << "Person: " << john;   // print to output stream
```

- **Part 2 of solution**: stream operator needs to be a `friend`

Stream I/O for User-defined Types – Example

EBERHARD KARLS UNIVERSITÄT TÜBINGEN

```
class Person {
 …
  // friend has access to private members
  friend ostream &operator<<(ostream &, const Person &);
  friend istream &operator>>(istream &_is, Person &_p);
};

istream &operator>>(istream &_is, Person &_p) {
  _is >> _p.firstName >> _p.lastName >> _p.age;
  if (!_is) // check for failure
    _p = Person{};
  return _is;
}

…

Person p1;
cin >> p1;   // read from input stream
```

Binary Serialization – POD (Plain Old Data)

EBERHARD KARLS UNIVERSITÄT TÜBINGEN

```
template <typename POD>
std::ostream &serialize(std::ostream &os, const POD &pod) {
  // this only works on built in data types (PODs)
  static_assert(std::is_trivial<POD>::value &&
                std::is_standard_layout<POD>::value,
                "Can only serialize POD types with this function");
  os.write(reinterpret_cast<char const *>(&pod), sizeof(POD));
  return os;
}

template <typename POD> std::istream &deserialize(std::istream &is, POD &pod) {
  static_assert(std::is_trivial<POD>::value &&
                std::is_standard_layout<POD>::value,
                "Can only deserialize POD types with this function");
  is.read(reinterpret_cast<char *>(&pod), sizeof(POD));
  return is;
}
```

- Simply use **`read`** and **write**
- Only works for trivial structs / classes (no string, pointers to member, …)

Binary Serialization – POD (Plain Old Data)

EBERHARD KARLS UNIVERSITÄT TÜBINGEN

```
template <typename POD>
std::ostream &serialize(std::ostream &os, const POD &pod) {
  // this only works on built in data types (PODs)
  static_assert(std::is_trivial<POD>::value &&
                std::is_standard_layout<POD>::value,
                "Can only serialize POD types with this function");
  os.write(reinterpret_cast<char const *>(&pod), sizeof(POD));
  return os;
}

template <typename POD> std::istream &deserialize(std::istream &is, POD &pod) {
  static_assert(std::is_trivial<POD>::value &&
                std::is_standard_layout<POD>::value,
                "Can only deserialize POD types with this function");
  is.read(reinterpret_cast<char *>(&pod), sizeof(POD));
  return is;
}
```

```
class Point3D {
public:
  float x, y, z;
  int idx;
};
…
  Point3D p1{1, 2, 3, 1};
  Point3D p2{4, 5, 6, 2};
  serialize(cout, p1);
  serialize(cout, p2);

  Point3D p3;
  Point3D p4;
  deserialize(cin, p3);
  deserialize(cin, p4);

// output will be non-human
readable
```

- Simply use **`read`** and **`write`**
- Only works for trivial structs / classes (no string, pointers to member, …)

Binary Serialization – Vector of POD


```cpp
template <typename POD>
std::ostream &serialize(std::ostream &os, std::vector<POD> const &v) {
  // this only works on built in data types (PODs)
  static_assert(std::is_trivial<POD>::value &&
                std::is_standard_layout<POD>::value,
                "Can only serialize POD types with this function");
  auto size = v.size();
  os.write(reinterpret_cast<char const *>(&size), sizeof(size));  // write out #elements (=size) as first datum
  os.write(reinterpret_cast<char const *>(v.data()), v.size() * sizeof(POD));
  return os;
}
template <typename POD>
std::istream &deserialize(std::istream &is, std::vector<POD> &v) {
  static_assert(std::is_trivial<POD>::value &&
                std::is_standard_layout<POD>::value,
                "Can only deserialize POD types with this function");
  decltype(v.size()) size;
  is.read(reinterpret_cast<char *>(&size), sizeof(size));       // read in number of elements we expect first
  v.resize(size);
  is.read(reinterpret_cast<char *>(v.data()), v.size() * sizeof(POD));
  return is;
}
```

```cpp
std::vector<Point3D> vec = {p1, p2, p1, p2};
serialize(cout, vec);

std::vector<Point3D> vec;
deserialize(cin, vec);
```

- Simply use **`read`** and **`write`**
- Only works for trivial structs / classes (no string, pointers to member, …)

Serialization – Issues

EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

- Classes with user-defined constructor are not POD
- e.g., **`strings`** need specific treatment

- Class hierarchies might need to store what type is beeing serialized

- Binary storage
  - Pay attention to machine depenendent representations
  - Little or big endian
  - Byte-code ;-)

- Further reading:
  - https://isocpp.org/wiki/faq/serialization

- C++ doesn't have reflections -> manual code necessary

Recommendation: Consider using a serialization library

Detail: C-Style File-I/O

- A low-level API for file access
- C-Style streams
- The file descriptor FILE
- Reading, writing, and navigating (seek) through files

`FILE` – C-Style

EBERHARD KARLS UNIVERSITÄT TÜBINGEN

- Access to a file is provided in C with a so-called **file descriptor**.

```
#include <stdio.h>
FILE * fDescr;
```

- **Encapsulates** all the information necessary for processing (depending on the OS)
- Automatic allocation and release (no **`new / delete`**!)
- Remembers **current position** in the file
- Think of it: C-style stream abstraction (no classes, only functions)

- Standard operations:
  - open
  - write
  - read
  - seek (find position)
  - close

FILE - fopen


```
FILE * fopen ( const char * filename, const char * mode ); // opens, creates, or extends file
```

- Returns **`NULL`** if not successful
- Buffered (if possible)

- Different modes of operation `mode`:
  - `"r"` - read access
  - `"w"` - write access
  - `"a"` - append to the end
  - `"b"` - binary format - otherwise always as text file (readable)
  - `"w+"` - read and write access. If file exists discard content.

  See https://cplusplus.com/reference/cstdio/fopen/ for other options `...`

Do not open many files at once. Operating system might run out of handles.

FILE - fclose

EBERHARD KARLS UNIVERSITÄT TÜBINGEN

```
int fclose ( FILE * stream ); // close file and release internal data
```

```
int fputs ( const char * str, FILE * stream ); // write string and advance position
```

```
char * fgets ( char * str, int num, FILE * stream ); // read (max.) num characters from
file and advances position.
```

stops on: **line break** or **end of file** (EOF) -> used to read lines

```
int feof ( FILE * stream ); // 0 if end position in the file has been reached
```

FILE – C-Style


```c
#include <stdio.h>

int main()
{
   FILE * pFile;
   char buffer [100];
   pFile = fopen ("myfile.txt" , "r");
   if (pFile == NULL) perror ("Error opening file");
   else
   {
     while ( ! feof (pFile) )
     {
       if ( fgets (buffer , 100 , pFile) == NULL ) break;
       fputs (buffer , stdout);
     }
     fclose (pFile);
   }
   return 0;
}
```

File descriptor for standard output, automatically opened

FILE - fprintf


```
int fprintf ( FILE *stream, const char *format, ... );
```

- Formatted output to a file (compare to `printf`)

Format string:

- Type specification
  - `%d` or `%i` - output of integer values
  - `%u` - Output of unsigned values
  - `%f` - Output of floating point numbers

- Formatting
  - `%`<number>[`d`|`u`|`f`] - right justified with <number> digits
  - `%.`<number>`f` - number of decimal places
  - `%.`<number>`d` - number of output digits
  - `\n` - new line

FILE - fscanf


```
int fscanf ( FILE *stream, const char *format, ... );
```

- Formatted reading from a file (remember: **printf**)
- Behind the format string, addresses of variables
- Equal operations on strings

```
int sscanf ( FILE *stream, const char *format, ... );
```

FILE - Example: fscanf

EBERHARD KARLS UNIVERSITÄT TÜBINGEN

[cplusplus.com]

```c
#include <stdio.h>

int main ()
{
  char str [80];
  float f;
  FILE * pFile;

  pFile = fopen ("myfile.txt","w+");
  fprintf(pFile, "%f %s", 3.1416, "PI"); // write '3.1415 PI'
  rewind (pFile); // rewind to start of file
  fscanf (pFile, "%f", &f);
  fscanf (pFile, "%s", str);
  fclose (pFile);
  printf ("I have read: %f and %s \n",f,str);
  return 0;
}
```

FILE - fseek

EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

```
int fseek ( FILE *stream, long int offset, int origin );
```

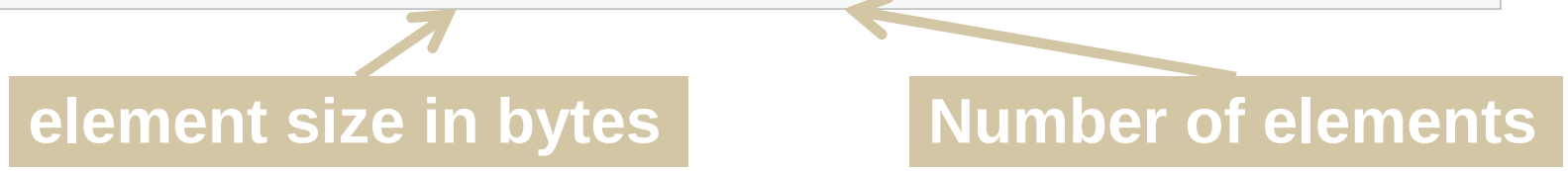
- Sets a new position in the file for the next access (reading or writing)

- Positioning relative to the position
  - **SEEK_SET** Start
  - **SEEK_CUR** Current position
  - **SEEK_END** End of file *

FILE - fwrite, fread

EBERHARD KARLS UNIVERSITÄT TÜBINGEN

```
size_t fwrite(const void *ptr, size_t size, size_t count, FILE *stream);
```



- Writes (`size * count`) bytes from `*ptr` to the file binary format
- Returns the number of successfully written elements

```
size_t fread(void *ptr, size_t size, size_t count, FILE *stream);
```

- Reads a number of bytes into the file in binary format
- Returns the number of successfully read elements

Errors and Exceptions

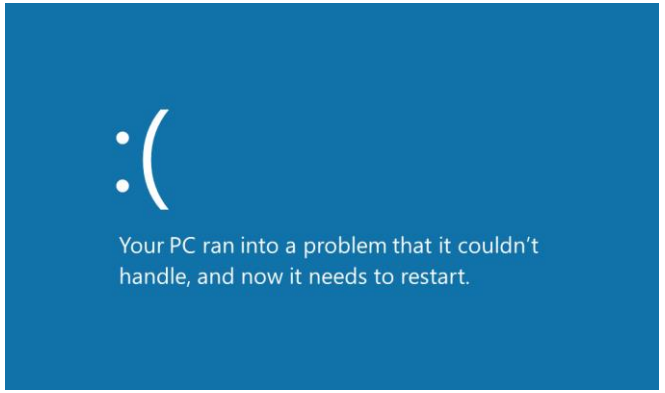
- What could possibly go wrong?

[from "An introduction to C++“, Hauswedell]

Error Handling

EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

- How to deal with runtime errors (exceptional events) like e.g.
  - cannot write to file (disk full, permission error, …)
  - out of bounds errors
  - memory limit exceeded
  - unexpected input
  - ...
- Different strategies:
  - exit program (probably with some error message)
  - try to handle the error and (if possible) continue the program



`exit` Program

EBERHARD KARLS UNIVERSITÄT TÜBINGEN

```
#include <fstream>
#include <iostream>

void writefile(std::ofstream &os, std::string const &text) {
  if (os) {
    os << text;
  }

  if (!os) // did writing of text succeed?
  {
    std::cerr << "some error writing to file" << std::endl;
    exit(1);
  }
}
```

- **`exit(1)`** terminates the program returning a non-zero value to indicate a runtime error
- (never do that in a library!)

Error Handling

- **Problem:** Often handling of the error must be performed
  at a position that is different from where the error occurred.

- How to communicate an error in some function back to the caller?

- Two approaches:
  - Returning error codes
  - Exceptions (only C++)

Error Code

EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

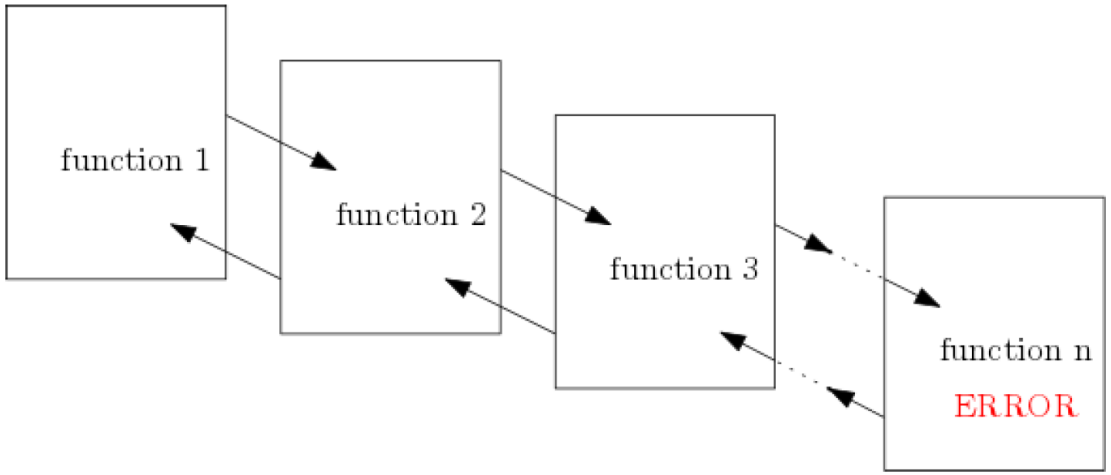
- Let a function return a code (or object) to signal that some error occurred
- Code 0 usually means 'ok' while every other value encodes a certain problem

```
int writefile(std::ofstream &os, std::string const &text) {
  if (os) {
    os << text;
  }
  if (!os) {
    std::cerr << "some error writing to file" << std::endl;
    return 1;
  }
  return 0;
}
```

Error Codes

- **Pro**:
  - Easy,
  - Now the error can be handled in the code that calls **`writefile`**

- **Contra**:
  - What if we want to handle the error in the function that calls the function that calls?
  - the function that calls the function that calls **`writefile`**?



Error Code Propagation

EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

```
int function1() {
  // …
  int rc = function2();
  if (rc != 0)
    // … error handling
    // …
    return 0;
}


int function2() {
  // …
  int rc = function3();
  if (rc != 0)
    return rc;
  // …
  return 0;
}
```

```
int function3() {
  // …
  int rc = function4();
  if (rc != 0)
    return rc;
  // …
  return 0;
}


int function4() {
  // …
  if (... some error condition...)
    return error_code;
  // …
  return 0;
}
```

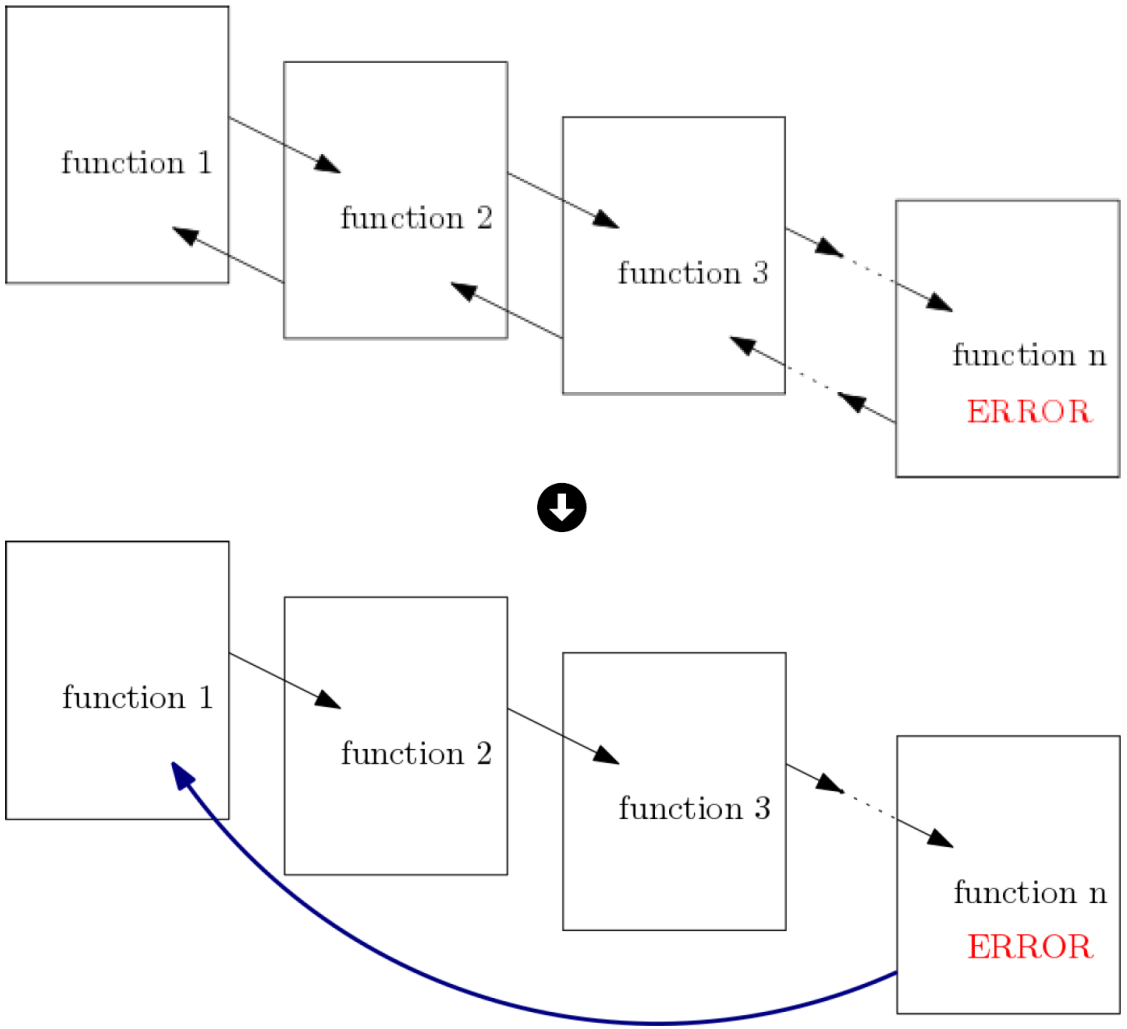
Exceptions

- Code where error occurs throws an exception
- Code in potentially totally different part of the program might handle the error and catches the exception
- **`try`**-block is put around code that potentially throws an exception
- **`catch`**-block is only active if matching exception occurs

```
void Vector3::normalize()
{
  float l = sqrt(x *x + y*y + z * z);
  if (l == 0.0)
  {
    throw string("zero!");
  }
  x /= l;
  …
}

...
Vector3 r = ….;
try
{
  r.normalize();
}
catch (string& error)
{
  cerr << error << endl;
  …
}
```

Exceptions Allow Long Jumps



Error Codes → Exceptions

EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

```
int function1() {
  // …
  try {
    function2();
  } catch (SomeException const &e)  {
    // … error handling
  }
  // …
  return 0;
}


int function2() {
  // …
  function3();
  // …
  return 0;
}
```

```
int function3() {
  // …
  function4();
  // …
  return 0;
}
int function4() {
  // …
  if (... some error condition...)
    throw SomeException();
  // …
  return 0;
}
```

- **`function4()`** throws exception
- **`function1()`** handles error in **`catch`** block

Exceptions – Details

EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

- Details:
  - In case of a runtime error an exception is thrown by **`throw expression`**
    - The **exception object** is copy initialized from **`expression`**
  - **Destructors of all objects in the functions on the jump to the matching catch block are called** -> **stack unwinding**
  - The error is handled in the matching catch block
  - The program continues **after** the catch block

- Questions:
  - What types can be thrown?
  - Which one is the matching catch block?
  - What happens if during stack unwinding another exception is thrown?

Exception Object

- Exceptions can be (nearly) arbitrary objects that usually carry useful information about the error like an error message, source code line or internal states …
- Temporary object copy initialized by the throw *expression*
  - if a class is thrown, copy/move constructor and destructor must be accessible
- array/functions ➜ pointer/pointer to function
- can be caught by **`lvalue`**-reference and modified
- can be **re-thrown**
- destroyed after last catch clause exits without *rethrowing*

```
class MyExc {
public:
  MyExc(MyExc const &) = delete;
};

void f() {
  // …
  throw MyExc(); // compile error
  // …
  throw "Error"; // char const *
  // …
  throw 21; // int
}
```

Error Handler Selection

EBERHARD KARLS UNIVERSITÄT TÜBINGEN

- General rule for matching catch blocks: the **first** handler that matches is selected

- Conversions:
  - array ⇒ pointer
  - function ⇒ pointer
  - qualification conversion
  - pointer conversion
  - **No** user-defined conversion!

```
void g() {
  try {
    evilStuff();
  } catch (unsigned i) {
    cout << "unsigned int" << endl;
  } catch (int i) {
    cout << "int" << endl;
  } catch (...) { // catches all
    cout << "general" << endl;
  }
}
```

```
void f(){ throw 10; }   // int    --> "int"
void f(){ throw 10.0; } // double --> "general"
```

Error Handler Selection

- General rule for matching catch blocks:
  the **first** handler that matches is selected

- Conversions:
  - array ⇒ pointer
  - function ⇒ pointer
  - qualification conversion
  - pointer conversion
  - **No** user-defined conversion!

```
struct Base { /* … */ };
struct Derived : Base { /* … */ };

void g() {
  try {
    evilStuff();
  } catch (Derived const &d) {
    cout << "Derived" << endl;
  } catch (Base const &b) {
    cout << "Base" << endl;
  } catch (...) {
    cout << "general" << endl;
  }
}
```

Handlers for derived exception classes need to be listed before their base classes. Otherwise, the base handler will be called!

```
void f(){throw Base();}    //  --> "Base"
void f(){throw Derived();} //  --> "Derived"
```

Error Handler Selection

EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

- If no matching error handler is found **`terminate()`** terminates the program.

```
/home/user/C++-Course/Workspace/VL1/build/Exceptions
terminate called after throwing an instance of 'Base'
Aborted
```

std::exception

EBERHARD KARLS UNIVERSITÄT TÜBINGEN

- **std::exception** is the base class of all exceptions thrown by the standard library

- logic_error
  - invalid_argument
  - domain_error
  - length_error
  - out_of_range
  - future_error(C++11)
- bad_optional_access(C++17)
- runtime_error
- range_error
  - overflow_error
  - underflow_error
  - regex_error(C++11)
  - nonexistent_local_time(C++20)
  - ambiguous_local_time(C++20)
  - tx_exception(TM TS)
  - system_error(C++11)
  - ios_base::failure(C++11)
  - filesystem::filesystem_error(C++17)

- bad_typeid
- bad_cast
- bad_any_cast(C++17)
- bad_weak_ptr(C++11)
- bad_function_call(C++11)
- bad_alloc
- bad_array_new_length(C++11)
- bad_exception
- ios_base::failure(until C++11)
- bad_variant_access(C++17)

Check cppreference.com for details

`std::exception`

EBERHARD KARLS UNIVERSITÄT TÜBINGEN

- **`std::exception`** and the derived classes provide a consistent interface

```
#include <exception> // std::exception
#include <iostream>  // std::cerr

// …
try {
  // …
} catch (const std::range_error &e) {
  // …
  std::cerr << e.what() << std::endl;
} catch (const std::invalid_argument &e) {
  // …
  std::cerr << e.what() << std::endl;
} catch (const std::exception &e) {
  // …
  std::cerr << e.what() << std::endl;
}
```

- **`std::exception`** has a single member function: **`what()`**.
- Disadvantage? No File or Line number!

Recommendation:

- use standard exceptions where appropriate
- define your own exceptions as derived from **`std::exception`** where necessary

Rethrowing

EBERHARD KARLS UNIVERSITÄT TÜBINGEN

- An exception handler can also **re-throw** the exception so it can be handled (again) in a catch for some **enclosing try-block**
- Error can be analyzed in different contexts: For example, to write the error in log file.

```
void f() {
  // …
  try {
    // …
  } catch (MyExc const &e) {
    // …
    throw; // re-throw exception
  } catch (...) {
    //<- not called by re-thrown
  }
}
```

```
void g() {
  // …
  try {
    f();
  } catch (MyExc const &e) {
    // … //handle exception again
  }
}
```

- **Important:** **`throw;`** and **`throw e;`** are not equivalent
  - **`throw;`** re-throws the original exception, **`throw e;`** would throw a copy

Exceptions – Pitfalls I

- What is the problem with this code? Can you spot it?

```
struct MyExc {
  std::string msg;
};

void g() {
  MyExc exc;
  // …
  throw &exc;
}
```

```
void f() {
  try {
    g();
  } catch (MyExc const *e) {
    cerr << e->msg;
  }
}
```

- Throwing a pointer to local object **`exc`** which is destructed during stack unwinding!

Exceptions – Pitfalls II

EBERHARD KARLS UNIVERSITÄT TÜBINGEN

- What is the problem with this code?

```
void g() {
  // …
  // …
  throw MyExc();
}
```

```
void f() {
  int *tmp = new int[1000];
  g();
  delete[] tmp;
}
```

- After **`g()`** throwing an exception the **`delete [] tmp`** is never executed
  → memory leak

- Can be avoided using RAII, **`std::array`** or **`std::vector`** or **`std::unique_ptr`**

Exceptions – Pitfalls III

- Exceptions in **constructors:**
- Destructor is called only for complete objects
- Which destructors will be called?

```
struct A {};
struct B {};

struct C : A {
  C() {
    throw 1;
  }
};

struct D : B, C {};
D d;
```

- Answer: **`A`** and **`B`**
  - **`B`** was constructed before **`C`**
  - **`A`** was constructed before **`C`** as its base class just before the exception was thrown

Exceptions – Pitfalls III

- Exceptions in **constructors:**
- Destructor is called only for complete objects
- What is the problem with this code?

```
class C {
  // …
  int *arr1, *arr2;
  C(size_t s) {
    arr1 = new int[s];
    arr2 = new int[s];
  }
  ~C() {
    delete[] arr1;
    delete[] arr2;
  }
};
```

- If exception is thrown during memory allocation of **`arr2`**:
  - Constructor not executed completely
  - → Object not fully created
  - → Destructor not called
  - → Memory of **`arr1`** not released ⇒ **`memory leak`**
- Solutions:
  - Catch exception in constructor and **`delete [] arr1`** in catch block
  - **Better:** use **`std::vector`**

Exceptions – Pitfalls IV

EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

- Exceptions in **destructors:**
- Destructors should not throw exceptions!

Why?

- **Remember**: During stack unwinding destructors of all local objects are called
- If a second exception is thrown during stack unwinding → **`terminate()`**

noexcept

- **`noexcept`** specifies whether a function could potentially throw an exception

```
void func1() noexcept;          // non-throwing
void func2();                   // potentially throwing
void func3() noexcept(true);    // same as noexcept
void func4() noexcept(false); // same as without noexcep
```

Facts:

- declaring a function as *non-throwing* enables some compiler optimization
- non-throwing functions can throw exceptions
  (directly or in subsequently called functions) → **`terminate()`**
- destructors are non-throwing per default
- for `=default` default constructor, copy/move constructor, copy/move assignment operator
  more complex rules exist
- **Important**: make your move constructor/assignment operator non-throwing to be used
  e.g. by STL containers like **`std::vector`**. Otherwise, slow copy.

Summary

- Streams
- C-Style File I/O
- Serialization
- Error handling, error codes, Exceptions