C++ PROGRAMMING

Lecture 9

Secure Software Engineering Group
Philipp Dominik Schubert

HEINZ NIXDORF INSTITUT
UNIVERSITÄT PADERBORN

CONTENTS

1. Object oriented programming II
2. Type conversion / type casting
3. File IO

HEINZ NIXDORF INSTITUT
UNIVERSITÄT PADERBORN
Fraunhofer
IEM

© Heinz Nixdorf Institut / Fraunhofer IEM

Call the parent's (virtual) function

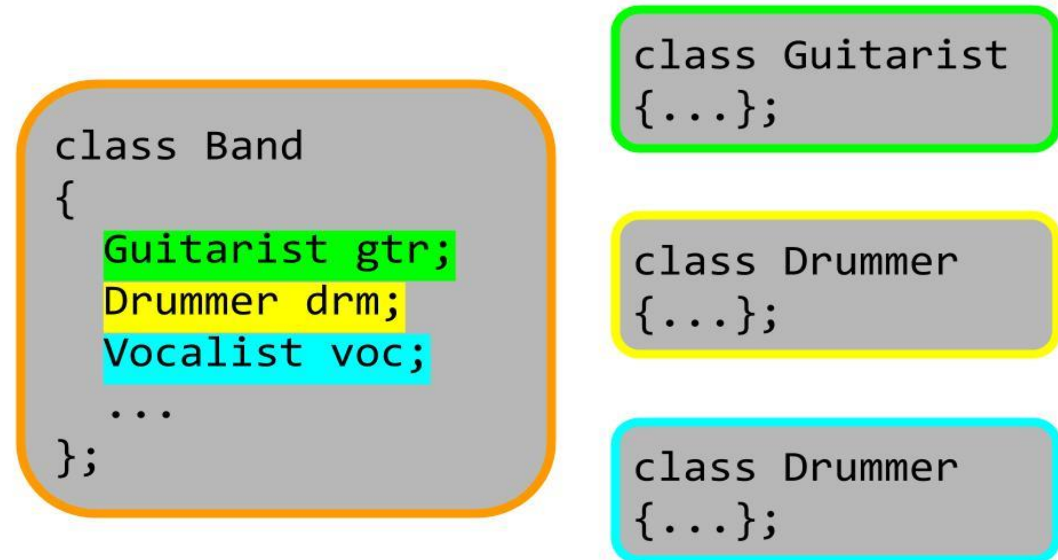
```
#include <iostream>
struct A {
  virtual ~A() = default;
  virtual void function() {
    std::cout << "A\n"; }
};
struct B : A {
  virtual ~B() override = default;
  void function() override {
    std::cout << "B\n"; }
};
struct C : B {
  virtual ~C() override = default;
  void function() override {
    std::cout << "C\n"; }
};
```

```
int main() {
  C c;
  // call the specialized version of C
  c.function();
  // call parent's specializations
  c.B::function();
  c.A::function();
  return 0;
}
```


© Heinz Nixdorf Institut / Fraunhofer IEM

Composition vs Inheritance

- Sometimes composition is more useful than inheritance
- Depends on the nature of the problem you wish to solve



- The GO programming language does not support inheritance

© Heinz Nixdorf Institut / Fraunhofer IEM
[Figure taken from http://pumpkinprogrammer.com/2014/07/14/c-tutorial-inheritance-and-composition/]

Templates vs Inheritance

- When to use what?

```
17  #ifndef PHASAR_PHASARLLVM_IFDSIDE_IDETABULATIONPROBLEM_H_
18  #define PHASAR_PHASARLLVM_IFDSIDE_IDETABULATIONPROBLEM_H_
19
20  #include <iostream>
21  #include <memory>
22  #include <set>
23  #include <string>
24
25  #include "phasar/PhasarLLVM/ControlFlow/ICFG.h"
26  #include "phasar/PhasarLLVM/DataFlowSolver/IfdsIde/EdgeFunctions.h"
27  #include "phasar/PhasarLLVM/DataFlowSolver/IfdsIde/IFDSTabulationProblem.h"
28  #include "phasar/PhasarLLVM/DataFlowSolver/IfdsIde/JoinLattice.h"
29
    You, 5 months ago | 4 authors (You and others)
30  namespace psr {      You, 3 years ago • updated example plug-ins
31
32  class ProjectIRDB;
33  template <typename T, typename F> class TypeHierarchy;
34  template <typename V, typename N> class PointsToInfo;
35
36  template <typename AnalysisDomainTy,
37            typename Container = std::set<typename AnalysisDomainTy::d_t>>
    You, 5 months ago | 3 authors (You and others)
38  class IDETabulationProblem
39      : public IFDSTabulationProblem<AnalysisDomainTy, Container>,
40        public virtual EdgeFunctions<AnalysisDomainTy>,
41        public virtual JoinLattice<AnalysisDomainTy>,
42        public virtual EdgeFactPrinter<AnalysisDomainTy> {
43  public:
44    using d_t = typename AnalysisDomainTy::d_t;
45    using n_t = typename AnalysisDomainTy::n_t;
46    using f_t = typename AnalysisDomainTy::f_t;
47    using t_t = typename AnalysisDomainTy::t_t;
48    using v_t = typename AnalysisDomainTy::v_t;
49    using l_t = typename AnalysisDomainTy::l_t;
50    using i_t = typename AnalysisDomainTy::i_t;
51
52    static_assert(std::is_base_of_v<ICFG<n_t, f_t>, i_t>,
53                  "Type parameter i_t must implement the ICFG interface!");
```


© Heinz Nixdorf Institut / Fraunhofer IEM

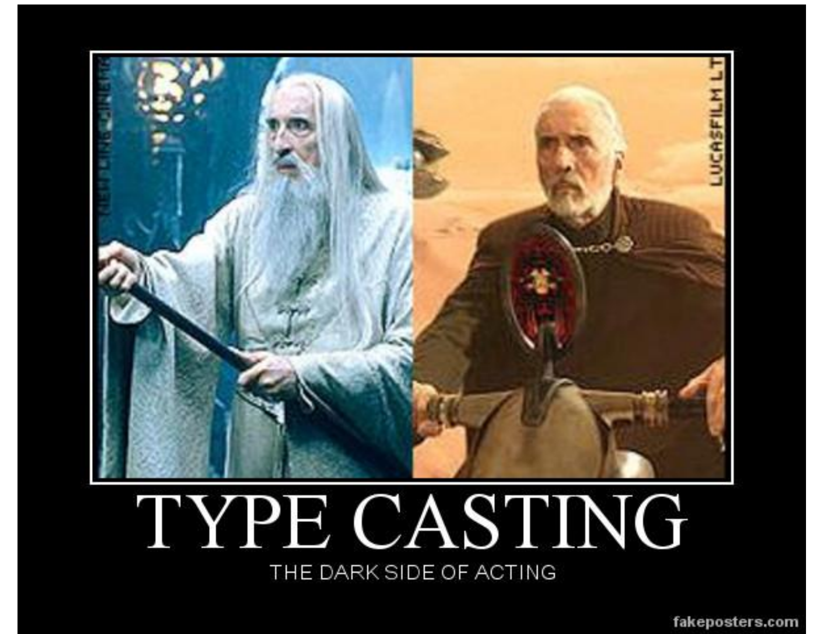
Templates vs Inheritance

- Inheritance
  - Is vertical and goes from abstract to more concrete
    - Example: Shape, triangle, right angle triangle
    - Runtime abstraction
    - Dynamic polymorphism
- Templates
  - Is horizontally, defines parallel instances of code
    - Example sorting: integers, doubles, …, can all be sorted
    - Code generation tool
    - Static polymorphism
- Both are orthogonal: can be used in combination

```
17  #ifndef PHASAR_PHASARLLVM_IFDSIDE_IDETABULATIONPROBLEM_H_
18  #define PHASAR_PHASARLLVM_IFDSIDE_IDETABULATIONPROBLEM_H_
19
20  #include <iostream>
21  #include <memory>
22  #include <set>
23  #include <string>
24
25  #include "phasar/PhasarLLVM/ControlFlow/ICFG.h"
26  #include "phasar/PhasarLLVM/DataFlowSolver/IfdsIde/EdgeFunctions.h"
27  #include "phasar/PhasarLLVM/DataFlowSolver/IfdsIde/IFDSTabulationProblem.h"
28  #include "phasar/PhasarLLVM/DataFlowSolver/IfdsIde/JoinLattice.h"
29
    You, 5 months ago | 4 authors (You and others)
30  namespace psr {      You, 3 years ago • updated example plug-ins
31
32  class ProjectIRDB;
33  template <typename T, typename F> class TypeHierarchy;
34  template <typename V, typename N> class PointsToInfo;
35
36  template <typename AnalysisDomainTy,
37            typename Container = std::set<typename AnalysisDomainTy::d_t>>
    You, 5 months ago | 3 authors (You and others)
38  class IDETabulationProblem
39      : public IFDSTabulationProblem<AnalysisDomainTy, Container>,
40        public virtual EdgeFunctions<AnalysisDomainTy>,
41        public virtual JoinLattice<AnalysisDomainTy>,
42        public virtual EdgeFactPrinter<AnalysisDomainTy> {
43  public:
44    using d_t = typename AnalysisDomainTy::d_t;
45    using n_t = typename AnalysisDomainTy::n_t;
46    using f_t = typename AnalysisDomainTy::f_t;
47    using t_t = typename AnalysisDomainTy::t_t;
48    using v_t = typename AnalysisDomainTy::v_t;
49    using l_t = typename AnalysisDomainTy::l_t;
50    using i_t = typename AnalysisDomainTy::i_t;
51
52    static_assert(std::is_base_of_v<ICFG<n_t, f_t>, i_t>,
```


© Heinz Nixdorf Institut / Fraunhofer IEM
[See http://stackoverflow.com/questions/7264402/when-to-use-template-vs-inheritance/]
HEINZ NIXDORF INSTITUT
UNIVERSITÄT PADERBORN
Fraunhofer
IEM

Type Casting



- Casting is the mechanism of type conversion
- Happens implicitly for most primitive types
- Be careful using explicit casting
  - The type system is your friend
- C-style: `(new type) expression`
  - Please do not use the C-style cast, use C++'s version!
  - C's rules for casting are a nightmare, C++ is more explicit about casting

```
int main() {
  int i = (int) 12.1234; // cut off after decimal point (use floor/ceil from <cmath>)
  short c = (short) 123123123123; // bad: overflow
  double d = (double) 14; // okay
  // computation is casted to the "most precise" representation involved (double)
  double p = 12.5 * i;
  return 0;
}
```


© Heinz Nixdorf Institut / Fraunhofer IEM

Type Casting and the `explicit` keyword

```cpp
#include <iostream>
class MyType {
public:
  MyType(int i) : i(i) {}
  friend std::ostream &operator<<(
      std::ostream &os,
      const MyType &t) {
    return os << t.i;
  }

private:
  int i;
};

MyType makeType() { return 42; }

int main() {
  MyType t = makeType();
  std::cout << t << '\n';
  return 0;
}
```

```cpp
#include <iostream>
class MyType {
public:
  explicit MyType(int i) : i(i) {}
  friend std::ostream &operator<<(
      std::ostream &os,
      const MyType &t) {
    return os << t.i;
  }

private:
  int i;
};

MyType makeType() { return MyType(42); }

int main() {
  MyType t = makeType();
  std::cout << t << '\n';
  return 0;
}
```


© Heinz Nixdorf Institut / Fraunhofer IEM

Dynamic cast – reference version

- Goes hand in hand with OOP

```
#include <iostream>
struct A {
  // must be polymorphic otherwise we cannot use a dynamic_cast
  virtual ~A() = default;
  virtual void f() {}
};
struct B : A {};
struct C : A {};
struct D : B, C {};
```

```
int main() {
  // the most derived object
  D d;
  // upcast, dynamic_cast may be used but unnecessary
  A &a = d;
  // downcast
  B &b = dynamic_cast<B&>(a);
  // sidecast
  C &c = dynamic_cast<C&>(b);
  return 0;
}
```


© Heinz Nixdorf Institut / Fraunhofer IEM

Dynamic cast – pointer version

```
#include <iostream>

struct Base {
// must be polymorphic, otherwise
// we cannot use dynamic_cast
  virtual ~Base() = default;
  virtual void f() {}
};

struct Derived: Base {};
```

```
int main() {
  Base *b1 = new Base;
  if(Derived *d = dynamic_cast<Derived*>(b1)) {
    std::cout << "downcast from b1 to d
                successful\n";
  }
  Base *b2 = new Derived;
  if(Derived *d = dynamic_cast<Derived*>(b2)) {
    std::cout << "downcast from b2 to d
                successful\n";
  }
  delete b1;
  delete b2;
  return 0;
}
```


HEINZ NIXDORF INSTITUT
UNIVERSITÄT PADERBORN
Fraunhofer
IEM

Check ISA relation with dynamic cast

```
#include <iostream>
#include <typeinfo>
struct A {
        virtual ~A() = default;
        virtual void f() {};
};
struct B : A {};
struct C : A {};
struct D : B, C {};
```

- Use exceptions in reference-version
- Check for `nullptr` in pointer-version

```
int main() {
  D d;
  try {
    A &a = dynamic_cast<A&>(d);
    std::cout << "d is an A\n";
  } catch (const std::bad_cast &e) {
    std::cout << "cast failed\n";
  }
  D *d_ptr = new D();
  if (A *a_ptr = dynamic_cast<A*>(d_ptr)) {
    std::cout << "d_ptr is an A\n";
  } else {
    std::cout << "cast failed\n";
  }
  delete d_ptr;
  return 0;
}
```


© Heinz Nixdorf Institut / Fraunhofer IEM

Const cast

- Cast constness away
  - You can remove constness when underlying type is not const!
- This is bad style
- You are working against the type system
- Type system shall prevent you from errors
- Only use a const cast in rare / extreme situtations

```
#include <iostream>
void modify(const int& i) {
  const_cast<int&>(i) = 100;
}
int main()  {
  // note i is not declared const
  int i = 3;
  const int& cref_i = i;
  // cref_i = 5; would fail!
  const_cast<int&>(cref_i) = 4;
  std::cout << "i = " << i << '\n';
  int other = 1;
  modify(other);
  std::cout << other << '\n';
  return 0;
}
```


© Heinz Nixdorf Institut / Fraunhofer IEM
[Figure taken from http://www.drodd.com/images14/forrest-gump17.jpg]

Reinterpret cast

- Converts type by reinterpreting underlying bit pattern
- Returns a different interpretation of bits

```
#include <iostream>
int main() {
  int bits = 0b110001010; // is 394 in decimal
  double& d = reinterpret_cast<double&>(bits); // is 6.99566e-249
  std::cout << bits << '\n';
  std::cout << d << '\n';
  return 0;
}
```

- Very rarely needed
  - Examples
    - If you have to implement some device drivers
    - Or fiddle around with shared object libraries

© Heinz Nixdorf Institut / Fraunhofer IEM

Static cast

- Performing an explicit cast
- Casting in hierarchies with non-virtual member functions / or on primitive types
- Old C-style
  - `(new type) expression`

```
int i = 2;
double d = (double) i * i;
```

- C++ style

```
int i = 42;
double d  = static_cast<double>(i);
```

- Implicit conversion

```
int j = 3.456;
double e = j;
bool mybool = 1;
```


© Heinz Nixdorf Institut / Fraunhofer IEM

Just a note on GOTO

- An evil mechanism
- GOTO performs an unconditional jump
- When to use GOTO?
  - If you'd like to introduce bugs
  - If you'd like to make maintenance impossible
  - If you'd like to make a program unreadable
- Every program with GOTO can be expressed in a program without GOTO
- Limited useful applications
  - Some error handling
- Restriction:
  - You can not jump across functions
    - (Use `setjmp()`/`longjmp()` for that)

```
#include <iostream>
int main() {
  int counter = 0;
label: // loop using goto
  std::cout << counter << '\n';
  if (counter < 10) {
    counter++;
    goto label;  // jump
  }
  for (int x = 0; x < 3; x++) {
    for (int y = 0; y < 3; y++) {
      std::cout << "x + y smaller than 3\n";
      if (x + y >= 3)
        goto endloop;
    }
  }
endloop:
  std::cout << '\n';
  return 0;
}
```


© Heinz Nixdorf Institut / Fraunhofer IEM

Just a note on GOTO

```c
static OSStatus
SSLVerifySignedServerKeyExchange(SSLContext *ctx, bool isRsa, SSLBuffer signedParams,
                                 uint8_t *signature, UInt16 signatureLen)
{
    OSStatus        err;
    ...

    if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
        goto fail;
    if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
        goto fail;
        goto fail;
    if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
        goto fail;

    // code ommitted for brevity...

    err = sslRawVerify(ctx,
                       ctx->peerPubKey,
                       dataToSign,                /* plaintext */
                       dataToSignLen,             /* plaintext length */
                       signature,
                       signatureLen);
    if(err) {
        sslErrorLog("SSLDecodeSignedServerKeyExchange: sslRawVerify "
                    "returned %d\n", (int)err);
        goto fail;
    }

fail:
    SSLFreeBuffer(&signedHashes);
    SSLFreeBuffer(&hashCtx);
    return err;

}
```

Oops...

Never gets called (but needed to be)...

Despite the name, always returns "it's OK!!!"

HEINZ NIXDORF INSTITUT UNIVERSITÄT PADERBORN
Fraunhofer IEM

[Image taken from http://zdnet1.cbsistatic.com/hub/i/r/2014/10/04/adf68e5f-4bdc-11e4-b6a0-d4ae52e95e57/resize/770xauto/602ba37ff5be136930d7e9ab78e65bc2/ssl-validation-setup-failure.png]

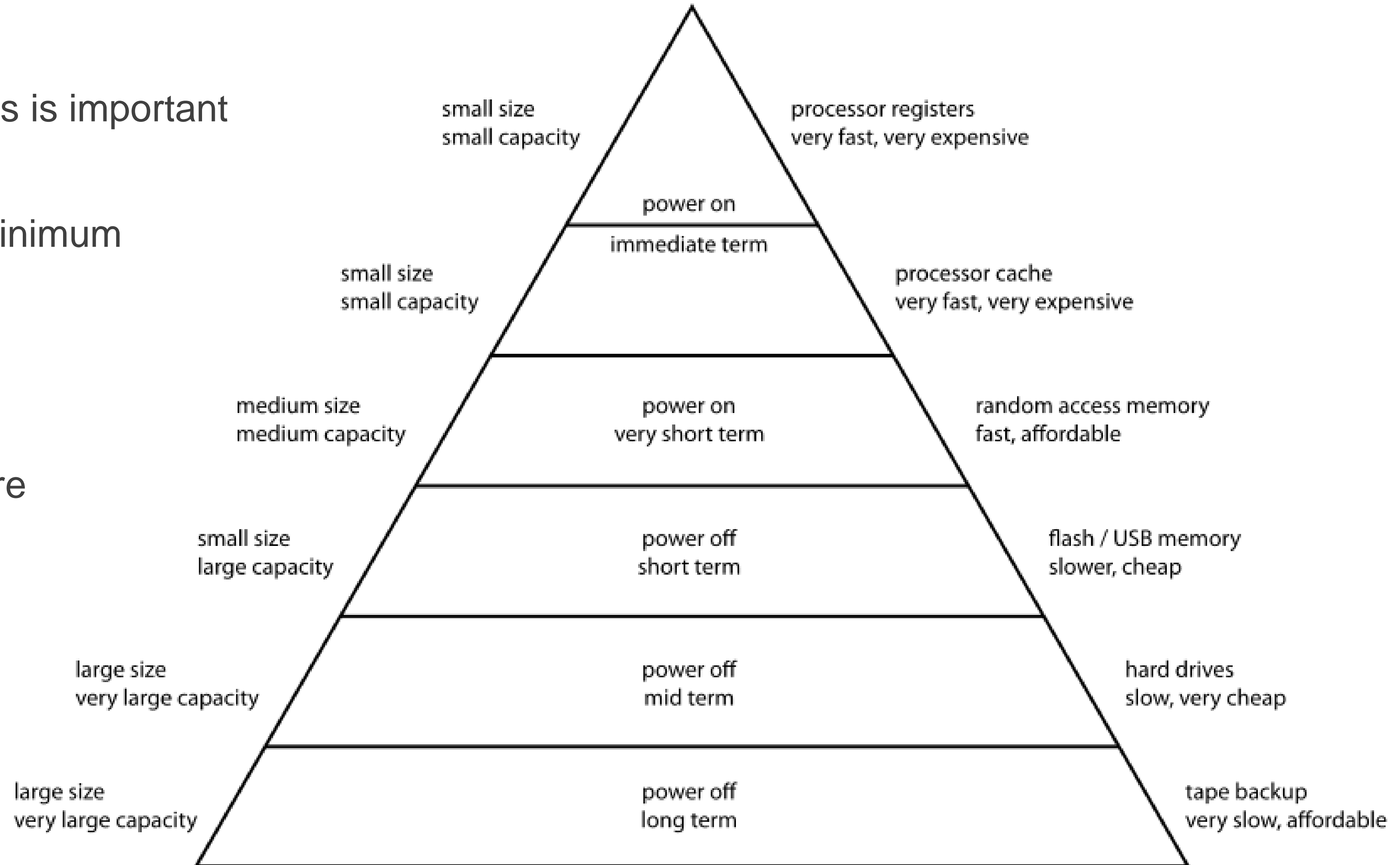
File IO

- File input and output operations
- How to get huge data into and out of your program?
  - Use files
  - Data to read / write from files may be
    - Small
    - Medium
    - Huge
- Several ways of doing this
  - Most convenient ways are presented
    - Text files
    - Binary files
    - Serialization
      - Text vs. binary

© Heinz Nixdorf Institut / Fraunhofer IEM

File IO

- Reading and writing from files is important
- But
  - Should be reduced to a minimum
  - Most expensive operation
    - Slooo…owly
  - Memory hierarchy
  - More details in HPC lecture



© Heinz Nixdorf Institut / Fraunhofer IEM
[Figure taken from https://upload.wikimedia.org/wikipedia/commons/thumb/0/0c/ComputerMemoryHierarchy.svg/300px-ComputerMemoryHierarchy.svg.png]

Writing text files using `std::ofstream`

```
#include <iostream>
#include <fstream>
#include <string>
int main() {
  std::string str = "Data is: ";
  int integer = 42;
  std::ofstream ofs("myfile.txt");
  std::cout << "is open? "
            << ofs.is_open() << '\n';
  ofs << "Hello World!\n";
  ofs << str << integer << '\n';
  ofs.close();
  std::cout << "is open? "
            << ofs.is_open() << '\n';
  return 0;
}
```

- Opens an output file stream
- Write formatted data to file stream
- Closes the output file stream
- `open()` might be used instead of constructor

  ```
  std::ofstream ofs;
  ofs.open("myfile.txt");
  ```
- `close()` is not really necessary
  - STL objects are RAII objects
  - File streams are closed when destructor is called
    - That is when they go out-of-scope
    - You cannot forget to close
    - But use close if you have a very “long” scope

© Heinz Nixdorf Institut / Fraunhofer IEM

Writing text files using `std::ofstream`

```
#include <iostream>
#include <fstream>
#include <string>
int main() {
  std::string str = "Data is: ";
  int integer = 42;
  std::ofstream ofs("myfile.txt", ofstream::app);
  std::cout << "is open? " << ofs.is_open() << '\n';
  ofs << "Hello World!\n";
  ofs << str << integer << "\n";
  ofs.close();
  std::cout << "is open? " << ofs.is_open() << '\n';
  return 0;
}
```

- A new text file is created
- Usually existing files are overwritten
- But different modes are possible
  - Provided as optional second argument

ate	**at** **e**nd	The *output position* starts at the end of the file.
app	**app**end	All output operations happen at the end of the file, appending to its existing contents.
trunc	**trunc**ate	Any contents that existed in the file before it is open are discarded.

© Heinz Nixdorf Institut / Fraunhofer IEM

Reading text files using `std::ifstream`

- Opens an input file stream
- Reads a text file line by line
  - Until the end of the file is reached
  - `std::getline()` might be the most used function for reading text
  - `std::getline()` reads from a stream until it hits a break line `'\n'`
- Closes input file stream

```
#include <iostream>
#include <fstream>
#include <string>
int main () {
  std::ifstream myfile ("myfile.txt");
  if (myfile.is_open()) {
    for (std::string line; std::getline(myfile, line);) {
      std::cout << line << '\n';
    }
  } else {
    std::cout << "Unable to open file\n";
  }
  return 0;
}
```


© Heinz Nixdorf Institut / Fraunhofer IEM

Reading formatted text files

```cpp
#include <fstream>
#include <iostream>
#include <sstream>
#include <string>
int main() {
  std::ifstream ifs("mynumbers.txt");
  if (ifs.is_open()) {
    for (std::string line; std::getline(ifs, line);) {
      std::istringstream iss(line);
      int a, b, c;
      if (!(iss >> a >> b >> c)) { break; }
        std::cout << a + b + c << '\n';
      }
  } else { std::cout << "could not open file\n"; }
  return 0;
}
```

- Reads a file line by line
- For each line an input string stream is created
- `std::istringstream` can be read formatted data
- Nice way of reading formatted data
- But caution
  - Code is a bit naive
  - Very fragile if file is ill-formatted
  - Additional error handling is needed

© Heinz Nixdorf Institut / Fraunhofer IEM

Reading / writing files in blocks

- Reading and writing line by line is slow
  - Not suited for large files (MB, GB, …)
- Files can be read and written in one or more blocks (raw bytes)
- Reading fewer blocks is better
- Idea
  - Read complete file in one go

```
#include <fstream>
#include <sstream>
#include <string>
std::string readTextFile(
      const std::string &Path){
  std::ifstream Ifs(Path);
  std::stringstream StrStream;
  std::string Contents;
  if (Ifs) {
    StrStream << Ifs.rdbuf();
    Contents = StrStream.str();
  }
  return Contents;
}
```


© Heinz Nixdorf Institut / Fraunhofer IEM

Memory mapped files

- Reading and writing files with `std::fstream` is fine
- Still, IO operations on disk are slow
- Reading huge files is expensive and slow
- There is another approach
- Memory mapped files
  - Idea
    - Map a file or portions of a file into much faster main memory (RAM)
    - In C use `mmap()` from `<sys/mman.h>`
    - In C++ use BOOST mapped file
      - BOOST is a large C++ library
    - Useful when file size > ~100 x X MB / GB / TB

© Heinz Nixdorf Institut / Fraunhofer IEM

Memory mapped files

- Read with BOOST mapped file

```
#include <iostream>
#include <boost/iostreams/device/mapped_file.hpp>
int main() {
  boost::iostreams::mapped_file_params params;
  params.path = "myfile.txt";
  //params.length = 512; // default: complete file
  params.new_file_size = pow(1024, 2); // 1 MB
  params.flags =
    boost::iostreams::mapped_file::mapmode::readonly;
  boost::iostreams::mapped_file  mf;
  mf.open(params);
  char* bytes = static_cast<char*>(mf.const_data());
  std::cout << bytes << '\n';
  mf.close();
  return 0;
}
```

- Configure memory mapping using params
- E.g.
  - Filename
  - Mode
  - Size of file
  - Size of mapping
- Open memory mapped file
- Obtain pointer to data
- Manipulate data
- Close memory mapped file

© Heinz Nixdorf Institut / Fraunhofer IEM

Memory mapped files

- Write with BOOST mapped file

```
#include <iostream>
#include <cmath>
#include <boost/iostreams/device/mapped_file.hpp>
int main() {
  boost::iostreams::mapped_file_params params;
  params.path = "data.txt";
  params.new_file_size = pow(1024, 2); // 1 MB
  params.flags =
    boost::iostreams::mapped_file::mapmode::readwrite;
  boost::iostreams::mapped_file  mf;
  mf.open(params);
  char* bytes = static_cast<char*>(mf.data());
  for (size_t i = 0; i < 10; ++i)
    bytes[i] = 'A';
  std::cout << bytes << '\n';
  mf.close();
  return 0;
}
```

- Configure memory mapping using params
- E.g.
  - Filename
  - Mode
  - Size of file
  - Size of mapping
- Open memory mapped file
- Obtain pointer to data
- Manipulate data
- Close memory mapped file

© Heinz Nixdorf Institut / Fraunhofer IEM

How to read files?

- Depends on your problem / file size
- Files up to several MB can be read / written …
  - Line based
  - Use `operator<<`, `operator>>`, `std::getline()`, …
- Files up to several hundred MB / GB can be read / written …
  - Block based
  - Allocate buffers and read / write whole blocks
- Files with sizes of several GB or TB can be read / written …
  - Memory mapped
- And if the state of objects have to be (re)stored?
  - Use serialization

© Heinz Nixdorf Institut / Fraunhofer IEM

Serialization

- Translate data structures or object state in a format that can be persisted
- Transform objects / data into one dimensional bit-string
- Usually stored …
  - as file on disk
  - in memory buffer
  - transmitted across network
- State can be reconstructed later when needed ➔ Deserialization
  - On same computer
  - On other computer
  - Semantically clone is created from stored state
- Tremendously useful (image you have to store intermediate results / caching)
- Simple for POD data, arbitrary complex when dealing with pointers / references

© Heinz Nixdorf Institut / Fraunhofer IEM

Text serialization using BOOST

```cpp
#include <iostream>
#include <fstream>
#include <boost/archive/text_oarchive.hpp>
#include <boost/archive/text_iarchive.hpp>
class Vec3 {
private:
  double x, y, z;
  friend class boost::serialization::access;
  template<class Archive>
  void serialize(Archive &ar, const unsigned int version){
    ar & x;
    ar & y;
    ar & z;
  }
public:
  Vec3() = default;
  Vec3(double a, double b, double c) : x(a), y(b), z(c) {}
  friend std::ostream& operator<<(std::ostream& os, const Vec3 v){
    return os << v.x << " " << v.y << " " << v.z; }
};
```

```cpp
int main() {
{
  Vec3 v1(1, 2, 3);
  std::cout << v1 << '\n';
  std::ofstream ofs("serVec3.txt");
  boost::archive::text_oarchive oa(ofs);
  oa << v1;
}
// other scope
{
  Vec3 new_v1;
  std::ifstream ifs("serVec3.txt");
  boost::archive::text_iarchive ia(ifs);
  ia >> new_v1;
  std::cout << new_v1 << '\n';
}
return 0;
}
```


© Heinz Nixdorf Institut / Fraunhofer IEM

Binary serialization using BOOST

```cpp
#include <iostream>
#include <fstream>
#include <boost/archive/binary_oarchive.hpp>
#include <boost/archive/binary_iarchive.hpp>
class Vec3 {
private:
 double x, y, z;
  friend class boost::serialization::access;
  template<class Archive>
  void serialize(Archive &ar, const unsigned int version){
    ar & x;
    ar & y;
    ar & z;
  }
public:
  Vec3() = default;
  Vec3(double a, double b, double c) : x(a), y(b), z(c) {}
  friend ostream& operator<< (ostream& os, const Vec3 v){
    return os << v.x << " " << v.y << " " << v.z; }
};
```

```cpp
int main() {
{
  Vec3 v1(1, 2, 3);
  std::cout << v1 << '\n';
  std::ofstream ofs("serVec3.bin");
  boost::archive::binary_oarchive oa(ofs);
  oa << v1;
}
// other scope
{
  Vec3 new_v1;
  std::ifstream ifs("serVec3.bin");
  boost::archive::binary_iarchive ia(ifs);
  ia >> new_v1;
  std::cout << new_v1 << '\n';
}
return 0;
}
```


© Heinz Nixdorf Institut / Fraunhofer IEM

Notes on serialization

- Other (structured) text serialization formats are possible
  - XML
  - JSON
  - …
- Have a look at:
  - https://www.boost.org/doc/libs/1_76_0/libs/serialization/doc/archives.html
- Do not forget to link with external libraries
  - Usually done by using `-l` compiler option (assuming that you have a system-wide Boost installation)
  - E.g.: `g++ -std=c++17 -Wall myprogram.cpp -o myprogram -lboost_serialization`

© Heinz Nixdorf Institut / Fraunhofer IEM

Text vs binary serialization

- Text (plain, XML, JSON, …)
  - Human readable
    - No extra debugging tools needed
  - Independent of computer architecture
    - No `sizeof` issues
    - No little-endian vs. big-endian issues
  - Can produce smaller results when most numbers are small

- Binary
  - Typically uses fewer CPU cycles
    - Important when IO is bottle-neck
  - Lets you ignore serration between adjacent values (since most values have fixed length)
  - Can produce smaller results when most numbers are large
  - Dependet of computer architecture!

- In general: no right answer

© Heinz Nixdorf Institut / Fraunhofer IEM

Notes on serialization

- Serialization is easy when using simple data types
  - See our `Vec3` class
- Serialization gets more complex when dealing with more complex types
  - Serializing derived classes
  - Serializing classes containing pointer/references data members
    - E.g. lists, graphs, …
    - But it is possible
  - Arrays
    - BOOST knows how to serialize built-in arrays
  - STL containers
    - BOOST knows how to serialize STL containers
  - All perfectly explained: http://www.boost.org/doc/libs/1_76_0/libs/serialization/doc/tutorial.html
- Later on: Google Protobuf (language-neutral, platform-neutral serialization)

© Heinz Nixdorf Institut / Fraunhofer IEM

Recap

- Object oriented programming II
- Type casting
- File IO
  - Reading and writing files
  - Dealing with files in blocks
  - Memory mapped files
  - Serialization

© Heinz Nixdorf Institut / Fraunhofer IEM

Thank you for your attention
Questions?