Programming Techniques for Scientific Simulations

# IMPORTANT

* C++: const whenever possible!
* Python: don’t forget docstring and self in non-static member functions

# Table of contents

Week 1: Version control, Git

* Exercise: Epsilon, Simpson

Week 2: a) Preprocessing/Compiling/Linking/Libraries b) Make

* Exercise: Static/dynamic arrays, Simpson (pointer), format output

Week 3: a) CMake b) Templates c) Classes

* Exercise: Epsilon (template), Penna (concept)

Week 4: Classes (static, mutable), Op. overloading, friend, lambda

* Exercise: Op. overloading (Z2), Simpson (functor), Penna header

Week 5: a) ctor, dtor, = b) Templates, Traits, Concepts

* Exercise: 2D Simpson, Traits, Penna implementation

Week 6: Exceptions, random numbers, timing

* Exercise: Exception, Random/Timer libraries, Test Penna

Week 7: std data structures, generic algorithms

* Exercise: Iterators, Benchmarking std::container, Penna Population

Week 8: Inheritance, Polymorphism

* Exercise: Inheritance, Simpson (ABS) + benchmark, Penna fishing

Week 9: Hardware, cache

* Exercise: Cache effects, Penna simulation (vector)

Week 10: Optimization, numerical libraries

* Exercise: Matrix multiplication (BLAS/LAPACK)

Week 11: Optimization in C++, inline, lazy eval., expr. templates

* Exercise: LA with LAPACK, metaprog. factorial + derivation

Week 12: Introduction to Python, types, control flow

* Exercise: Python: Penna, Golf

Week 13: a) NumPy, Matplotlib b) I/O

* Exercise: NumPy, matplotlib, venv

# Git

**.gitignore**

#Ignore all  
\*  
#Unignore all with extensions  
!\*.\*  
#Unignore all dirs  
!\*/  
#Result: Ignore only binaries (no extension)

Setup:

git config –global user.name “Robin”

git config -global user.email “a@b.c”

Commands: init, fetch, status, log, help, add, commit, push, pull, branch, checkout, merge

# Unix Shell

* Move/copy: mv/cp source/file.c dest/file.c
* Copy directory: cp -r <src> <dest>
* Rename: mv ./asdf.txt ./qwer.txt
* Remove: rm (-r) <path>
* Redirect output: wc -l \*.txt > out.txt
* Sort & append: sort -n out.txt >> out.txt
* Input from file: ./main < input.txt

./main < <(echo 1 2 3)

echo 1 2 3 | ./main

Pipe: Combine multiple commands

wc -l \*.txt | sort -n | head -n 1

ls | grep “test” | tail -n 2

Wildcard: \* = zero or more chars; ? = 1 character

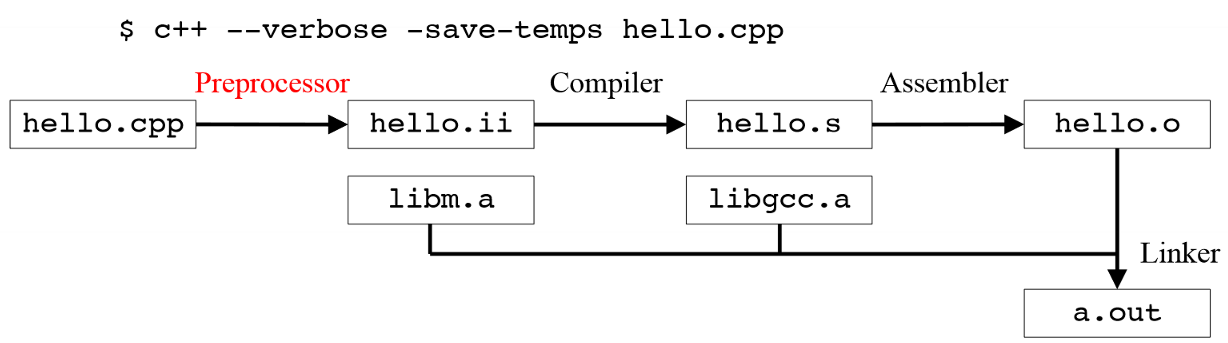
Tools:

* wc <file>: #lines #words #chars (-l only #lines)
* sort <file> (default alphanum., -n for numerical, -r reverse)
* cat / less: print content of file
* echo: print argument
* cut: cut out sections of each line (e.g. text before “,”)

cut -d , -f 1 test.txt

Robin, 19 Robin

# Compiling & Linking



Run until: Preprocessor (-E), Compiler (-S), Assembler (binary) (-c)

**Compile multiple files separately, link afterwards**

Ein Bild, das Text enthält.

Automatisch generierte Beschreibung

**Include guards (for header files)**

**#ifndef HEADER\_HPP**  
**#define HEADER\_HPP**  
*// declarations*   
**#endif */\* HEADER\_HPP \*/***

# Libraries

* Static libraries: lib\*.a
  + At link time, only the used functions from the archive are copied into the executable
  + Compile the sources into object files

g++ -c square.cpp

* + Pack the object files into a static library

ar -crs libsquare.a square.o

* + Name must be libsomething.a
* Shared libraries: lib\*.so
  + The functions from the library are not copied into the executable. Instead, the library is loaded only once into memory where it can be used by any executable.
  + Compile the sources into Position Independent Code (PIC) object files

g++ -fPIC -c square.cpp

* + Pack the object files into a shared library

g++ -shared -fPIC -o libsquare.so square.o

* + The name must be libsomething.so
* Use libraries
  + Compile the main (include path to library!)

g++ -c -I lib main.cpp

* + Link the object files

g++ -o main main.o -Llib -lsquare

* + -L + folder of library, -l + name of library
  + Order of libraries is important! If libA.a calls a function in libB.a, you need to link in the right order: -lA -lB
* cf. Demo Week 2 / Ex. 2-2

Document library with: Signature of functions, semantics, pre-/postconditions, dependencies, exception guarantees, references

# Make & CMake

**Make**

* make (runs Makefile) or run make -f make.mk (builds first target by default). make -n just prints commands (useful for debugging)
* Define targets with dependencies/prerequisites

**target: prerequisites**  
[TAB] commands

* Use this struct to build all

**-include config.mk**

**.PHONY: all**  
**all: square**

* config.mk defines compiler version, flags etc.; should not go under version control / be pushed if personal paths are used.
* Variables

|  |  |
| --- | --- |
| $@ | file name of the target of the rule |
| $< | name of the first prerequisite |
| $^ | names of all the prerequisites, separated by space |
| CC | program for compiling C progs; def. cc |
| CXX | program for compiling C++ progs; def. g++ |
| RM | command to remove a file; def “rm -f” |
| CFLAGS | extra flags for C compiler |
| CXXFLAGS | extra flags for C++ compiler |
| LDFLAGS | extra flags for compiler when linking |
| LDLIBS | library flags/names for compilers when linking |

**square.o: square.cpp square.hpp**  
 ${CXX} ${CXXFLAGS} $<  
**main.o: main.cpp square.hpp**  
 ${CXX} ${CXXFLAGS} $<  
**square: main.o square.o**  
 ${CXX} ${CXXFLAGS} -o $@ $^

* Cleaning

.PHONY clean  
**clean:**  
 ${RM} -v \*.o square

* cf. Demo Week 2 (square/square lib) / Ex. 2-2 / 2-3

**CMake**

* cross-platform build system generator
* Create CMakeLists.txt in source directory
* cd <build-dir>; cmake <src-dir>
* Use ccmake <src-dir> to specify settings, compiler flags, etc.
* cf. Ex 3-1 / Demo Week 3
* Useful functions
  + project() set project name, useful for debugging
  + add\_executable(<target name> <src files>) compile an executable
  + add\_library(<name> <type> <src files>) create lib (SHARED/STATIC)
  + target\_link\_libraries(<target> <libs>) makes lib accessible to the target, adds it in linking process
  + add\_subdirectory(<path>) run CMakeLists in a subdirectory, e.g. for library
  + include\_directories(<path>) tell preprocessor where to look for files to include
* cf. Ex 06 / Penna on how to use subdirectories.

# Generic Programming

* If functions are **overloaded**, the compiler chooses the best fit.
* No type safety when using **macros** + unexpected side effects

**#define MIN(x, y) (x < y ? x : y)**  
MIN(x++, y++); *// smaller number incremented twice!*  
 (x++ < y++ ? x++ : y++)

* **Templated** version: Usage causes instatiation. Type safe, compile error if misused.

**template** <**typename** T>  
T **min**(T x, T y) {  
 **return** (x < y ? x : y); }  
**int** **min**(**int** x, **int** y); *// if called with int args*  
**double** **min**(**double** x, **double** y); *// with double args*

Requirements on T? operator< with result convertible to bool; Copyable (pass by value), not needed if changed to const T&

* **Definition Polymorphism**: Using many different types through the same interface
* Documenting a function template: Pre-/Postconditions, semantics, exception guarantees. New: concept requirements on types.
* **Complete source code of the template function must be in a header file**
* Specialize a templated function/struct etc.

**template**<**typename** T>  
**struct** **helper** { **typedef** T type; };  
**template**<>  
**struct** **helper**<int> {**typedef** **double** type; };

**Type Traits**

* “Calculate” which type should be used in the template e.g. when adding two vectors of different types
* Examples slide 05b.6f.
* **Concept**: Set of requirements on types
  + The operations the type must provide
  + Their semantics (meaning of the operations)
  + Their time/space complexity
* Concept defined by C standard. E.g. *regular type*
  + CopyConstructible
  + Assignable
  + EqualityComparable
  + Destructible
* cf. Demo Week 5 (Traits)

# Data structures

**Classes**

* public: only representation-independent interface, accessible to all
* private: representation-dependent functions and data members
* friend: declarators allow related function/classes access to representation
* Default constructor = constructor without arguments
* Constructor: Order of member initialization: same as declaration in class. Sometimes initializer list is necessary: No other way of setting members that are const, reference or of a class type without default ctor. **No const members if class should be copy-assignable!**
* Use typedef to define recurring types (typedef double coord\_t;) or using (using coord\_t = double;)
* mutable keyword allows member variable to change value through a const member function

Ein Bild, das Text enthält.

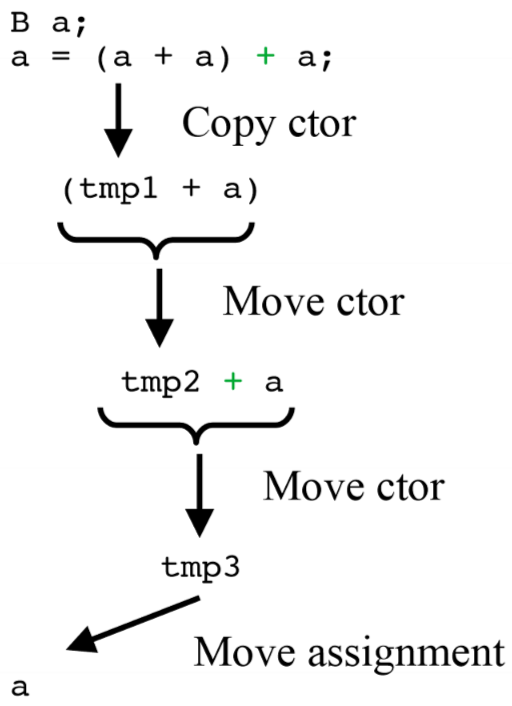
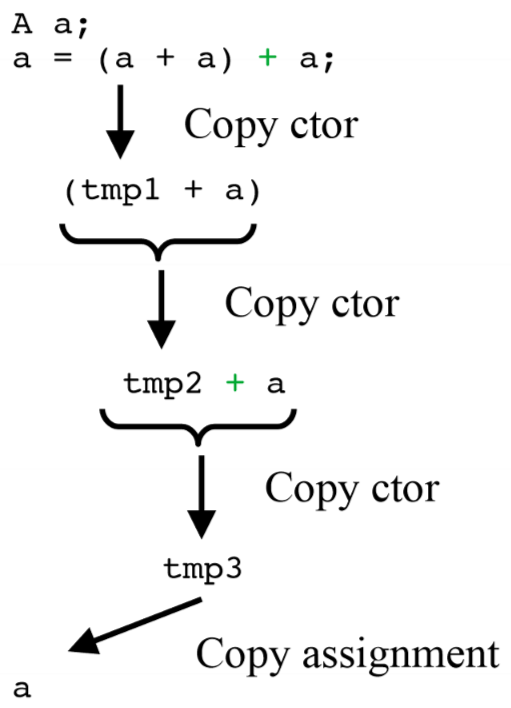
Automatisch generierte Beschreibung

* static member: one variable for all objects of the same class, e.g. a count/id. Must be initialized. Exist even without an object, thus access via scope operator (::)
* friend grants a function or another class access to the private and protected members of the class
* Special member functions

T(); *// default ctor*  
~T(); *// dtor*  
T(**const** T&); *// copy ctor*  
T& **operator**=(**const** T&); *// copy assignment*  
T(T&&); *// move ctor*  
T& **operator**=(T&&); *// move assignment*

Move ctor/assignment moves a temporary object into the new/another object. Acts on rvalue references/xvalues.

* Copying vs. moving



* Rule of three: dtor, copy ctor + assignment
* Rule of five: dtor, copy ctor + assignment, move ctor + assignment
* std::move produces an xvalue expression.

std::move(first, last, first\_new); *// moves [first, last[ to first\_new*  
T a = T(std::move(b)); *// move ctor*

* cf. Demos Week 5 (SArray) for implementation details (SArrayT for templated version)
* cf. Ex. 7-1 Iterators
* Some operators might need to be written twice, with and without const

**Static Variables**

**Variables persist through the whole program and are not deleted at the end of scope**

*// function example*  
**void** **foo**() {  
 *//only executed the first time*  
 **static** **int** count = 0;   
 *//executed every time, value persists*   
 count++;  
}  
*// loop example*  
**for**(**int** i = 0; i < 5; ++i) {  
 **static** **int** count = 0;  
 count++;  
}

**Funktoren**

* Struct/Class, deren operator() überladen ist.
* Funktionen, die aber auch einen Zustand annehmen können.
* **Lambda-Expressions** sind Inline-Funktoren/anonyme Fkt.

Ein Bild, das Text enthält.

Automatisch generierte Beschreibung

* + Return type und Parameter können weggelassen werden
  + Minimale Lambda-Expression: []{} (Aufruf: []{}();)
  + Verschiedene Captures
    - [x]: Zugriff auf kopierten Wert von x (nur lesend)
    - [&x]: Zugriff auf Referenz von x
    - [&x, y]: Zugriff auf Ref. x und Wert von y
    - [&]: Default-Referenz-Zugriff auf alle Objekte im Kontext der Lambda-Expression
    - [=]: Default-Werte-Zugriff auf alle Objekte im Kontext der Lambda-Expression.
  + Achtung bei Funktoren innerhalb von Klassen

**struct** **mutant** {  
 **int** i = 0;  
 **void** **do**(){ [=]{i = 1;}(); } };  
mutant m; m.**do**();  
cout << m.i; *// 1, weil bei der 𝜆-Expr. der this-Pointer (this->i) implizit kopiert wird, nicht die Variable selbst.*

* **cf. Ex. 5-2 Simpson2D (nested Lambdas)**

**Smart Pointers**

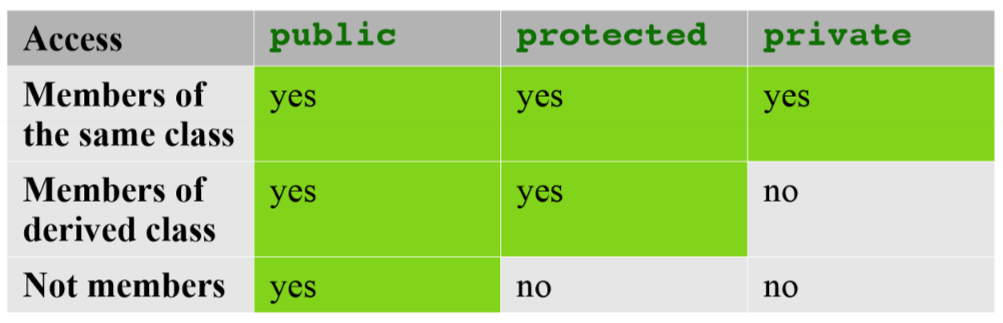
std::shared\_ptr<**int**> p = std::make\_shared<**int**>(3);

**Operator overloading**

Ein Bild, das Tisch enthält.

Automatisch generierte Beschreibung

**Polymorphism**



* The derived class inherits all members of the base class except: ctors, dtor, assignment operators, friends, private members.
* **Virtual functions**: Can be redefined in a derived class while preserving its interface and get correctly redirected to the derived class when called through a base class pointer/reference.
* Classes that declare or inherit virtual functions are called polymorphic classes.
* **Abstract Base Class (ABC)** is a class that contains one (or more) pure virtual functions. Pure virtual functions are functions without a definition in the base class. We are declaring an interface that all derived classes must adhere! ABC’s cannot be used to instantiate objects.
* Use override keyword to make sure that the function is virtual and is overriding a virtual function from a base class (inheritance05.cpp).
* Use final keyword to make sure that the function cannot be overridden by derived classes

|  |  |
| --- | --- |
| **Runtime polymorphism** | **Compile time polymorphism** |
| Use virtual functions | Use templates |
| Decision at runtime | Decision at compile time |
| Works for objects derived from the common base | Works for objects having the right members (concepts) |
| One function created for the base class saves space | A new function created for each class used more space |
| Virtual function call needs lookup in type table slower | No virtual function call, can be inlined faster |
| Extension possible using only definition of base class | Extension needs definition and implementations of all functions |
| Most useful for application frameworks, user interfaces, big functions | Useful for small, low-level constructs, small fast functions, generic algorithms |

# Exceptions

* Exception is an object of any type
* Thrown using the throw keyword

**if**(n <= 0) **throw** “n too small”; *// throw char[]*  
**if**(index >= size) **throw** std::range\_error(“index”);

* Std. exceptions from <stdexcept>, derived from std::exception
* std::logic\_error
  + domain\_error: value outside domain of variable
  + invalid\_argument: argument is invalid
  + length\_error: size too big
  + out\_of\_range: argument has invalid value
* std::runtime\_error
  + range\_error: invalid value occurred as part of calculation
  + overflow\_error: value got too large
  + underflow\_error: value got too small
* Specifier noexcept: compiler know that the function will never throw an exception (e.g. destructors should never fail).
* cf. Demo Week 6

# Optimizing, Timing & Profiling

**Optimizations**

* Most important: Use compiler optimizations. (-O0, …, O5, Os (size)), use -fopt-info to see optimization reports
* Use different algorithm/data structure for lower asymptotic runtime
* Use Assembly instructions. Code becomes non-portable!
* Between asm and C++: Compiler intrinsics
  + Many compilers support a long list of intrinsic functions that look like functions but get mapped directly to assembly statements.
  + Needs special compiler flag
  + cf. Demo Week 10
* Change associativity (e.g. of matrices), can be done by compiler in simple cases or precompute certain things.
* Minimize work done in loops.
* Dead code removal: Compiler detects if a statement is never executed.
* Keep storage order in mind (row-/col-major, strides)

**C++ specific optimizations**

* C++ compiler with templates is a Turing machine
* **Template Metaprogramming (TMP)**: Perform loops (recursion) and do branches at compile time
* cf. Demo Week 11 for examples (dot product & Unruh (primes))
* **Expression templates** (ET). Example:

a = b + c + d;

**for**(**int** i = 0; i < a.size(); ++i)  
 a[i] = b[i] + c[i] + d[i];

* Lazy evaluation: Postpone evaluation of an expression until assignment. Don’t evaluate temporary terms of a computation.
* cf. Demo Week 11 (etvector)

**Timing**

* C++: <chrono> but use simple timer library
  + cf. Ex. 6 timer
* Time executables with: time ./main

# Python

* Get help using help(), help(int), help(object)

**List**

x = [0, 1, 2, 3, 3] *# list*  
x[2] == 2  
x.insert(0, 5) *# insert(ind, val) [5, 0, 1, 2, 3, 3]*  
x.pop(0) *# [0, 1, 2, 3, 3]*  
x = [0, 1, 2, ‘three’] *# any types*  
x[-2] == 2 *# negative index -> access from back*  
x += [4, 5, 6] *# [0, 1, 2, ‘three’, 4, 5, 6]*  
x[1:4] *# [start:end+1] [1, 2, ‘three’]*  
x[1:] *# [start:end] [1, 2, 3, ,’three’, 4, 5, 6]*  
x[0:-1] *# [start:end-1] [0, 1, 2, ‘three’, 4, 5]*   
x[0:7:4] == x[::4] *# [start;end+1:step] [0, 4]*  
x[-1:0:-2] *# reverse sliciing [6, 4, 2]*  
x[::-2] *# [6, 4, 2, 0]*

**Tuples (immutable)**

x = (1, 2, 3)  
x[1] = 3 *# type error, not possible*

**Dictionary**

x = dict(a=1, b=2, c=’three’)  
x = {‘a’:1, ‘b’:2, ‘c’:’three’}  
x[‘a’] == 1  
x[1] = 4 *# adding new entry, any type can be key*  
x.keys() *# [‘a’, ‘c’, 1, ‘b’] # order not preserved*  
x.values() *# [1, ‘three’, 4, 2]*  
x.items() *# [(‘a’, 1), (‘c’, ‘three’),*

*(1, 4), (‘b’, 2)]*

**for loop**

**for** <item> **in** <collection>: <statements>  
**for** item **in** [0, ‘a’, 7, 1j]: print(item)  
**for** i **in** "StRiNg": print(i)   
b = [i+1 **for** i **in** range(3) **if** i!=2] *# [1, 2]*

**Functions**

**def** **func**(a, b=2, c="default"):  
 ...  
 **return** ...  
func(4); func(4, 5); func(4, 5, "yo"); func(4, 5, 6)  
func(b=4,c=5,a=6) *# out-of-order call*  
*#variadic args*  
**def** **fun**(a, b=2, \*args, c, d=4, \*\*kwargs): ...  
fun(positional\_args, keyword\_args) *# call fun*

a positional argument, b pos. arg. with default, args variadic positional args, c keyword-only arg, d keyword-only with default, kwargs variadic keyword-only argument

**Lambda**

**def** **F**(a, b): **return** a-b  
F = **lambda** a, b: a - b

**Classes**

* ctor: \_\_init\_\_; dtor: \_\_del\_\_; op+: \_\_add\_\_; op\*: \_\_mul\_\_; op/: \_\_truediv\_\_; //: \_\_floordiv\_\_
* don’t forget self as first argument!
* cf. slide 12.56 Inheritance, Decorators (@expr) etc.

NumPy / Matplotlib: see cheatsheets & Demos (Week 13) & Slides 13a

Efficient access instead of loop

**import** numpy **as** np  
a = np.linspace(1, 10, 10, dtype=float)  
b = np.random.random(10) \* 10  
b[:] += a[::-1] *# b[i] += a[n-i-1]*