Precourse to the PeC^3 Summer School on

Introduction on Numerical Modelling with Differential Equations

Ole Klein[1]

[1]Universität Heidelberg
Interdisziplinäres Zentrum für Wiss. Rechnen
Im Neuenheimer Feld 205, D-69120 Heidelberg
email: ole.klein@iwr.uni-heidelberg.de

UNIVERSIDAD NACIONAL AGRARIA
LA MOLINA

DAAD
Deutscher Akademischer Austausch Dienst
German Academic Exchange Service

BMZ
Federal Ministry
for Economic Cooperation
and Development

PeC^3
Peruvian Competence Center
of Scientific Computing

Universidad Nacional Agraria La Molina,
Lima, Perú, October 23–25, 2019

Contents I

What is C++?

C++ is a programming language that was developed for both system and application programming.

- Support for several programming paradigms: imperative, object oriented, generic (templates)
- Emphasizes efficiency, performance and flexibility
- Applications range from embedded controllers to high-performance super computers
- Allows direct management of hardware resources
- "Zero-cost abstractions", "pay only for what you use"
- Open standard with several implementations
- Most important compilers:
  - Open source: GCC and Clang (LLVM)
  - Proprietary: Microsoft and Intel

History

- 1979: Bjarne Stroustrup develops "C with Classes"
- 1985: First commercial C++ compiler
- 1989: C++ 2.0
- 1998: Standardization as ISO/IEC 14882:1998 (C++98)
- 2011: Next important version with new functionality (C++11)
- 2014: C++14 with many bug fixes and useful features
- 2017: C++17, current version
- 2020: upcoming standard

First C++ Program

```cpp
/* make parts of the standard library available */
#include <iostream>
#include <string>

// custom function that takes a string as an argument
void print(std::string msg)
{
  // write to stdout
  std::cout << msg << std::endl;
}

// main function is called at program start
int main(int argc, char** argv)
{
  // variable declaration and initialization
  std::string greeting = "Hello, world!";
  print(greeting);
  return 0;
}
```

Central Components of a C++ Program

A basic C++ program consists of two components:

Include directives to import software libraries:

- always the first lines of a program
- only one include directive per line

User-defined functions:

- like mathematical functions, with arguments and return value
- every program must implement the function

```
int main(int argc, char** argv)
{
  ...
}
```

This function is called by the operating system when the program is executed.

Comments

Rules for comments in C++ code:

- Comments can be placed anywhere in the code
- Comments starting with `//` (C-style comments) end at a line break:

```
int i = 42; // the answer
int x = 0;
```

- Multiline comments are started by `/*` and ended by `*/`:

```
/* This comment spans
   multiple lines */
int x = 0;
```

Functions

- During execution of a C++ program functions are called, starting with the special function `main(int argc, char** argv)`
- Functions can call other functions
- Function definitions consist of a function signature and a function body:

```
return-type functionName(arg-type argName, ...) // signature
{
  // function body
}
```

- The signature defines the name of the function and which arguments it needs
- In C++, a function **always** has a return type. The special type `void` is used if a function shouldn't return anything.
- The function body describes what the function does

Statements

```
int i = 0;
i = i + someFunction();
anotherFunction();
return i;
i = 2; // never executed
```

- A C++ function consists of a number of statements, executed one by one
- Statements are separated by semicolons
- The special statement `return val;` **immediately** leaves the current function and returns `val` as its return value
  - `void` functions can leave out the value or even the whole `return` statement

Variables

- Variables are used for storing intermediate values
- In C++, variables always have a fixed **type** (integer, floating-point, text, ...)
- Variables may contain upper and lower case letters, digits and underscores, but may not start with a digit
- Names are case sensitive!
- Variables have to be **declared** before they can be used
  - Normal variables are declared through a statement:

```
variable-type variableName = initial-value;
```

  - Function arguments are declared in the function signature:

```
void func(var-type1 arg1, var-type2 arg2)
```

Important Variable Types

C++ knows many variable types, here some important ones (ranges valid on 64-bit Linux):

```
// 32-bit integer, whole numbers in [−2^31, 2^31 − 1]
int i = 1;
// 64-bit integer, whole numbers in [−2^63, 2^63 − 1]
long l = 1;
// 8-bit integer, whole numbers in [−2^7, 2^7 − 1]
char c = 1;
// Boolean (truth value), true (=1) or false (=0)
bool b = true;
// Text (sequence of symbols), requires #include <string>
std::string msg = "Hello";
// Floating point with double accuracy
double d = 3.141;
// Floating point with single accuracy
float f = 3.141;
```

Integer variables restricted to positive numbers by prepending **unsigned**, with range $[0, 2^{\text{bits}} - 1]$

Scopes and Variable Lifetime

- A **block scope** is a group of statements within braces (e.g., function body)
- Scopes can be nested arbitrarily
- Variables have a limited **lifetime**:
  - the lifetime of a variable starts with its declaration
  - it ends when leaving the scope where it was declared

```
int cube(int x)
{
  // x exists everywhere in the function
  {
    int y = x * x; // y exists from here on
    x = x * y;
  } // here y doesn't exist anymore
  return x;
} // here x doesn't exist anymore
```

Scopes and Name Collisions

It is impossible to create two variables with the same name within a scope, and names in an inner scope temporarily shadow names from an outer scope

```
{
  int x = 2;
  int x = 3; // compile error!
}

int abs(int x) { ... } // absolute value

{
  int x = -2;
  {
    double x = 3.3; int abs = -2;
    std::cout << x << std::endl; // prints 3.3
    x = abs(x); // compile error, here abs is a variable!
  }
  x = abs(x); // now: x == 2
}
```

Namespaces

Scopes can also have an name, and are then called *namespaces*. Namespaces are used to group parts of a program together that share functionality or form a larger unit. They are a tool for the organization of large code bases.

```
namespace MyScientificProgram {
  namespace LinearSolvers {
    // any user-defined functions, objects, etc.,
    // that deal with linear solvers
  }

  namespace NonlinearSolvers {
    // any functions, etc., that belong to nonlinear solvers,
    // e.g., Newton's method
  }
}
```

Tools provided by the C++ standard library are in namespace `std`.

Expressions

We use expressions to calculate things in C++

- Expressions are combinations of values, variables, function calls, and mathematical operators, that produce some value that can be assigned to a variable:

```
i = 2;
j = i * j;
d = std::sqrt(2.0) + j;
```

- Composite expressions like `(a * b + c) * d` use standard mathematical precedence rules, known as **operator precedence**

Rule Overview

`https://en.cppreference.com/w/cpp/language/operator_precedence`

Operators for Numbers

- The usual binary operators +,−,∗,/
- a % b calculates the remainder of integer division of a by b:

```
13 % 5 // result: 3
```

- Division of integers always rounds to 0
- Integer division by 0 crashes the program
- = assigns its righthand side to its lefthand side, and at the same time returns this value

```
a = b = 2 * 21; // both a and b have value 42
```

- Abbreviations for frequent combinations:

```
a += b; // shortcut for a = a + b (also for -,*,/,%)
x = i++; // post-increment, shortcut for x = i; i = i + 1;
x = ++i; // pre-increment,  shortcut for i = i + 1; x = i;
```

  - of course there's also pre- and post-decrement (−−)

Comparison Operators

- Comparison operators produce truth values (`bool`):

```
4 > 3; // true
```

- Available operators

```
a < b;  // a strictly less than b
a > b;  // a strictly greater than b
a <= b; // a less than or equal to b
a >= b; // a greater than or equal to b
a == b; // a equal to b (note the double =!)
a != b; // a not equal to b
```

Combination of Truth Values

Test results can be combined using symbolic or text-based operators:

- Combination of several tests with "and" or "or":

```
a == b ||  b == c; // a equal b or b equal c
a == b or  b == c; // a equal b or b equal c

a == b &&  b == c; // a equal b and b equal c
a == b and b == c; // a equal b and b equal c
```

- Inversion of a truth value:

```
!true == false;
not true == false;
```

Texts / Strings

- Texts (strings) are stored in variables of type `std::string`
- Fixed strings are enclosed in double quotes

```
std::string msg = "Hello world!";
```

- Strings can be concatenated with `+`

```
std::string hello = "Hello, ";
std::string world = "world";
std::string msg = hello + world;
```

- They can be compared with `==` and `!=`

```
std::string a = "a";
a == "b"; // false
```

Warning

When comparing or concatenating strings, the left argument must **always** be a variable!

Programming Tasks

Task 1

Consider the "hello, world!" program. Modify the program so that it uses a function `mark` that

- reads a `std::string` from `std::cin`
- adds an exclamation mark "!" to its end
- prints the resulting string using `std::cout`

Task 1

Add a function `calculate` that

- reads in two numbers, an `int` and a `double`
- prints their sum and product on one line, separated by a space
- returns their difference as a value

I/O Streams under UNIX

- UNIX (and Linux) programs communicate with the operating system using so-called I/O (Input/Output) **streams**
- Streams are one-way streets — you can either read from them or write to them
- Every program starts with three open streams, namely

stdin Standard input reads user input from the terminal, is connected to **file descriptor** 0

stdout Standard output receives results printed by the program, is connected to **file descriptor** 1

stderr Standard error output receives diagnostic messages like errors, is connected to **file descriptor** 2

Redirecting I/O Streams I

- Normally all standard streams are connected to the terminal
- Sometimes it is useful to redirect these streams to files
- **stdout** is redirected by writing `"> fileName"` after the program name

```
[user@host ~] ls > files
[user@host ~] cat files
file1
file2
files
```

  The output file is created before the command is executed
- Error messages are still printed to the terminal

```
[user@host ~] ls missingdir > files
ls: missingdir: No such file or directory
[user@host ~] cat files
[user@host ~]
```

Redirecting I/O Streams II

- These three operators may be combined, of course
- **stdin** is read from a file by appending "< fileName"

```
[user@host ~] cat # no argument means copying stdin to stdout
terminal input^D # (CTRL+D) terminates input
terminal input
[user@host ~] cat < files
file1
file2
files
```

- **stderr** is saved to file by using "2> fileName"

```
[user@host ~] ls missingdir 2> error
[user@host ~] cat error
ls: missingdir: No such file or directory
```

- These three operators may be combined, of course

Printing to the Terminal

- A C++ program can use the three streams **stdin**, **stdout** and **stderr** to communicate with a user on the terminal (shell)
- Output uses `std::cout`. Everything we want to print is "pushed" into the standard output using `<<`

```
#include <iostream> // required for input / output
...
std::string user = "Joe";
std::cout << "Hello, " << user << std::endl;
```

- A line break is created by printing `std::endl` (**end line**)

Reading from the Terminal

- Reading user input uses `std::cin`
- The corresponding variable has to be created first
- Input is "pulled" out of standard input with `>>`

```
#include <iostream>
...
std::string user = "";
int answer = 0;
std::cout << "Enter your name: " << std::endl;
std::cin >> user;
std::cout << "Enter your answer: " << std::endl;
std::cin >> answer;
std::cout << "Hi " << user << "! Your answer was: "
          << answer << std::endl;
```

- Input on the terminal has to be committed using the return key

Control Flow

Most programs are impossible or very difficult to write as a simple sequence of statements in fixed order.

Examples:

- a function returning the absolute value of a number
- a function catching division by zero and printing an error message
- a function summing all numbers from 1 to $N \in \mathbb{N}$
- ...

Programming languages contain special statements that execute different code paths based on the value of an expression

Branches

- The **if** statement executes different code depending on whether an expression is true or false

```
int abs(int x)
{
  if (x > 0)
  {
    return x;
  }
  else
  {
    return -x;
  }
}
```

- The **else** clause is optional:

```
if (weekday == "Wednesday")
{
  cpp_lecture();
}
```

Repetition

Often, a program has to execute the same code several times, e.g., when calculating the sum $\sum_{i=1}^{n} i$

Two different approaches:

- **Recursion**: the function calls itself with different arguments
- **Iteration**: a special statement executes a list of statements several times

Recursion

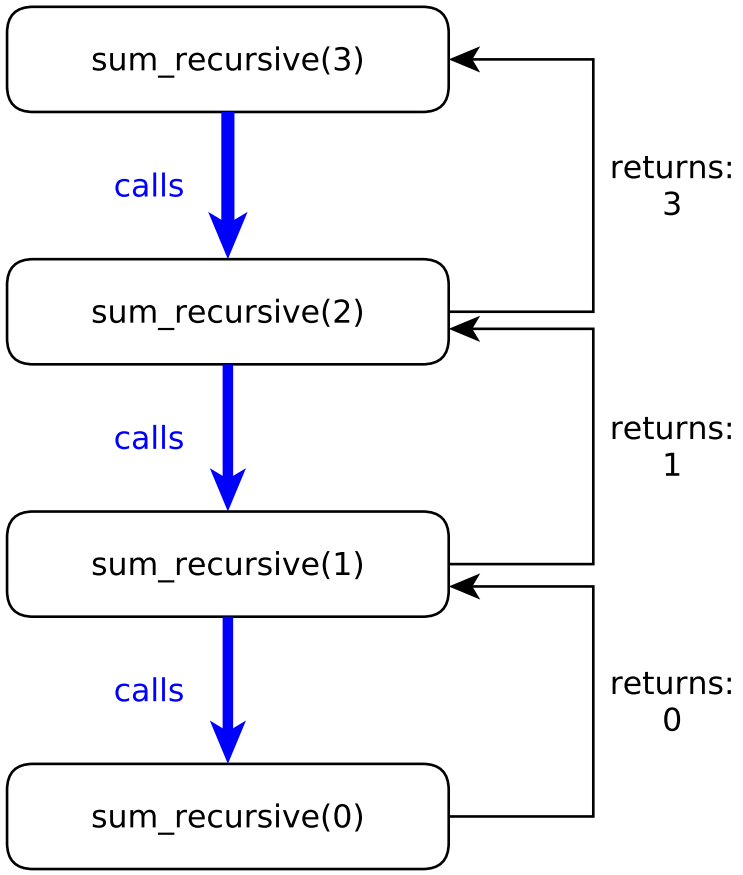
- Idea: a function calls itself again and again with changed input arguments, until some termination criterion is fulfilled:

```
int sum_recursive(int n)
{
  if (n > 0)
  {
    return sum_recursive(n - 1) + n;
  }
  else
  {
    return 0;
  }
}
```

- Requires at least one **if** statement, with exactly one of the branches calling the function again!
- Not suitable for functions that don't return anything and only have side effects (e.g., printing the first N numbers on the terminal)

Recursion: Example

- `sum_recursive(3)` calls `sum_recursive(2)` ...
- ...which calls `sum_recursive(1)` ...
- ...which calls `sum_recursive(0)` ...
- ...which ends the recursion due to the special case.
- Values on arrows are return values of corresponding function call.



Iteration Using While Loop

- A `while` loop executes the following block repeatedly, as long as its expression evaluates as true

```
int sum_iterative(int n)
{
  int result = 0;
  int i = 0;
  while (i <= n)
  {
    result += i;
    ++i;
  }
  return result;
}
```

- Often easier to understand
- Often more explicit and requires more variables

Iteration with For Loop I

- Many loops are executed several times for different values of some counting variable (index)
- C++ has a special **for** loop for these cases:

```
int sum_for(int n)
{
  int result = 0;
  for (int i = 0 ; i <= n ; ++i)
  {
    result += i;
  }
  return result;
}
```

- communicates to the reader that we sum over some index
- restricts lifetime of `i` to the loop itself
- somewhat more complicated than a **while** loop

Iteration with For Loop II

Every `for` loop can be converted into an equivalent `while` loop:

```
for (int i = 0 ; i <= n ; ++i)
{
  ...
}
```

becomes

```
{
  int i = 0;
  while (i <= n)
  {
    ... ;
    ++i;
  }
}
```

Integer Powers

Consider $q \in \mathbb{N}$ raised to the power of $n \in \mathbb{N}$.

Recursive definition:

$$q^n := \begin{cases} q^{n-1} \cdot q & \text{if } n > 0 \\ 1 & \text{if } n = 0 \end{cases}$$

"Iterative definition":

$$q^n := \underbrace{q \cdot \dots \cdot q}_{n \text{ times}}$$

Task

How can this function be implemented in C++?

Recursive Power Function

```
int pow_recursive(int q, int n)
{
  if (n == 0)
    return 1;
  else
    return q * pow_recursive(q,n-1);
}
```

- if $n = 0$ nothing needs to be computed
- else compute q^{n-1} and multiply by q (compare definition)

Iterative Power Function

```
int pow_iterative(int q, int n)
{
  int out = 1;
  for (int i = 0; i < n; i++)
    out *= q;
  return out;
}
```

- start out with value 1 (case $n = 0$)
- multiply with q n times

Recursive Power Function II

```
int pow_rec_fast(int q, int n)
{
  if (n == 0)
    return 1;
  else
  {
    int t = pow_rec_fast(q,n/2);
    if (n % 2 == 0)
      return t*t;
    else
      return q * t*t;
  }
}
```

- $q^n = q^{2k} = \left(q^k\right)^2$ for n even
- $q^n = q^{2k+1} = \left(q^k\right)^2 \cdot q$ for n odd

Iterative Power Function II

```
int pow_iter_fast(int q, int n)
{
  int out = 1;
  while (n > 0)
  {
    if (n % 2 != 0)
    {
      out *= q;
      n--;
    }

    q = q*q;
    n /= 2;
  }

  return out;
}
```

Which Version is best?

Two different measures of "best":

Readability and conciseness: definitely the first two versions

Speed: measure time for 100 billion (10^8) evaluations of 2^{30}

Version	Time	Version	Time
recursive	11.113s	recursive 2	1.988s
iterative	7.663s	iterative 2	1.397s

`std::pow()`: 3.378s

Note: This is the maximum range of the user-defined versions. The built-in `std::pow()` works for a much wider range, and also works for floating-point arguments. This explains why it is more expensive.

Which Version is best? II

These results were obtained without optimization. Compilers can optimize code and produce equivalent programs that are significantly faster (e.g., compiler option -O2).

What changes when we turn on optimization?

Version	Time	Version	Time
recursive	0.001s	recursive 2	1.317s
iterative	0.001s	iterative 2	0.257s

`std::pow()`: 0.001s

The simplest versions are now the fastest! Why? Because they are simple enough to be optimized away (evaluated **at compile time**)

Therefore: don't think too much about optimal code, most of the time it is irrelevant, and if not, **measure**

Classes and Objects

Objects are representations of components of a program, i.e., a self-contained collection of data with associated functions (called methods).

Classes are blueprints for objects, i.e., they define how the objects of a certain data type are structured.

Classes provide two special types of functions, *constructors* and *destructors*, which are used to create resp. destroy objects.

Example: Matrix Class

```
  class Matrix
{
  private: // can't be accessed by other parts of program

    std::vector<std::vector<double> > entries; // data
    int numRows;                               // number of rows
    int numCols;                               // number of columns

  public: // defines parts of object that are visible / usable

    Matrix(int numRows_, int numCols_); // constructor
    double& elem(int i, int j);         // access entry
    void print();                       // print to screen
    int rows();                         // number of rows
    int cols();                         // number of columns
};
```

Encapsulation

```
class Matrix
{
  public:
    // a list of public methods
  private:
    // a list of private methods and attributes
};
```

The keyword **public**: marks the description of the interface, i.e., those methods of the class which can be accessed from the outside.

The keyword **private**: accompanies the definition of attributes and methods that are *only available to objects of the same class*. This includes the data and implementation-specific methods reqired by the class. To ensure data integrity it should *not* be possible to access stored data from outside the class.

Definition of Methods

```
class Matrix
{
  public:
    // ...
    double& elem(int i, int j)
    {
      return entries[i][j];
    }
};
```

The method definition (i.e., listing of the actual function code) can be placed directly in the class (so-called inline functions). In the case of inline functions the compiler can omit the function call and use the code directly.

Definition of Methods II

```
void Matrix::Matrix(int numRows_, int numCols_)
{
  entries.resize(numRows_);
  for (int i = 0; i < entries.size(); i++)
      entries[i].resize(numCols_);
  numRows = numRows_;
  numCols = numCols_;
}
```

If methods are defined outside the definition of a class, then the name of the method must be prefixed with the name of the class followed by two colons.

Generic Programming

Often the same algorithms are required for different types of data. Writing them again and again is tedious and error-prone.

```
int square(int x)
{
    return(x*x);
}
long square(long x)
{
    return(x*x);
}
```

```
float square(float x)
{
    return(x*x);
}
double square(double x)
{
    return(x*x);
}
```

Generic programming makes it possible to write an algorithm once and parameterize it with the data type. The language device for this is called **template** in C++ and can be used for functions, classes, and variables.

Function Templates

A function template starts with the keyword **template** and a list of template arguments, separated by commas and enclosed by angle brackets:

```
template<typename T>
T square(T x)
{
    return(x*x);
}
```

This way, the function basically has two types of arguments:

- Types, specified in angle brackets
- Variables, specified in parentheses

This becomes clearer when actually calling the function (see below).

Template Instantiation

At the first use of the function with a specific combination of data types the compiler automatically generates code for these types. This is called *template instantiation*, and has to be unambiguous.

Ambiguities can be avoided through:

- Explicit type conversion of arguments
- Explicit specification of template arguments in angle brackets:

```
std::cout << square<int>(4) << std::endl;
```

The argument types must match the declaration and the types have to provide all the necessary operations (e.g. the `operator*()`).

Example: Unary Function Template

```cpp
#include <cmath>
#include <iostream>

template<typename T>
T square(T x)
{
  return(x*x);
}

int main()
{
  std::cout << square<int>    (4)    << std::endl;
  std::cout << square<double>(M_PI) << std::endl;
  std::cout << square        (3.14) << std::endl;
}
```

Example: Binary Function Template

```cpp
#include <iostream>

template<class U>
const U& maximum(const U& a, const U& b)
{
  if (a > b)
    return a;
  else
    return b;
}

int main()
{
  std::cout << maximum(1,4)           << std::endl;
  std::cout << maximum(3.14,7.)       << std::endl;
  std::cout << maximum(6.1,4)         << std::endl; // comp. error
  std::cout << maximum<double>(6.1,4) << std::endl; // unambiguous
  std::cout << maximum(6.1,double(4)) << std::endl; // unambiguous
  std::cout << maximum<int>(6.1,4)    << std::endl; // warning
}
```

Specialization of Function Templates

It is possible to define special template functions for certain parameter values. This is called *template specialization*. It can, e.g., be used for speed optimizations:

```
template <typename V, size_t N>
double scalarProduct(const V& a, const V& b)
{
    double result = 0;
    for (size_t i = 0; i < N; ++i)
        result += a[i] * b[i];
    return result;
};

template<typename V>
double scalarProduct<V,2>(const V& a, const V& b)
{
    return a[0] * b[0] + a[1] * b[1];
};
```

Class Templates

In addition to function templates, it is often also useful to parameterize classes. Here is a simple stack as an example:

```
template<typename T>
class Stack
{
  private:
    std::vector<T> elems; // storage for elements of type T

  public:
    void push(const T&);  // put new element on top of storage
    void pop();           // retrieve uppermost element
    T top() const;        // look at uppermost element
    bool empty() const    // check if stack is empty
    {
      return elems.empty();
    }
};

// + implementations of push, pop, and top
```

Further Reading

Online Tutorials

`http://www.cplusplus.com/doc/tutorial/`

Quick Reference

`https://en.cppreference.com/w/`

Other Resources

`https://isocpp.org/get-started/`

Best Practices for Scientific Computing

G. Wilson, D.A. Aruliah, C.T. Brown, N.P.C. Hong, M. Davis, R.T. Guy, S.H.D. Haddock, K.D. Huff, I.M. Mitchell, M.D. Plumbley, B. Waugh, E.P. White, P. Wilson: *Best Practices for Scientific Computing*, PLOS Biology

1. Write programs for people, not computers
2. Let the computer do the work
3. Make incremental changes
4. Don't repeat yourself (or others)
5. Plan for mistakes
6. Optimize software only after it works correctly
7. Document design and purpose, not mechanics
8. Collaborate

Write Programs for People, not Computers

Software must be easy to read and understand by other programmers (especially your future self!)

- Programs should not require readers to hold more than a handful of facts in memory at once
  - Short-term memory: The Magical Number Seven, Plus or Minus Two
  - Chunking (psychology): binding individual pieces of information into meaningful whole
  - Good reason for encapsulation and modularity
- Make names and identifiers consistent, distinctive, and meaningful
- Make coding style and formatting consistent

Minimum Style Requirements

Programming style conventions can be inconvenient, but there is a bare minimum that should be followed in any case:

- Properly indent your code, use meaningful names for variables and functions
- Comment your code, but *don't* comment every line: main idea/purpose of class or function, hidden assumptions, critical details
- Don't just fix compilation errors, also make sure that the compiler doesn't issue warnings

This makes it easier for other people to understand your work, including:

- Colleagues you may ask for help or input
- Other researchers extending or using your work
- Yourself in a few weeks or months (!)

Let the Computer do the Work[1]

Typing the same commands over and over again is both time-consuming and error-prone

- Make the computer repeat tasks (*i.e., write shell scripts or python scripts*)
- Save recent commands in a file for re-use (*actually, that's already done for you, search for "reverse-i-search" online*)
- Use a build tool to automate workflows (*e.g., makefiles, cmake*)

But make sure you don't waste time building unnecessarily intricate automation structures!

[1] *italics: my personal remarks*

Make Incremental Changes

Requirements of scientific software typically aren't known in advance

- Work in small steps with frequent feedback and course correction (e.g., agile development)
- Use a version control system (e.g., Git, GitLab)
- Put everything that has been created manually under version control
  - The source code, of course
  - Source files for papers / documents
  - Raw data (from field experiments or benchmarks)

Large chunks of data (data from experiments, important program results, figures) can be stored efficiently using Git-LFS (large file storage) extension

Don't Repeat Yourself (or Others)

Anything that exists in two or more places is harder to maintain and may introduce inconsistencies

- Every piece of data must have a single authoritative representation in the system
- Modularize code rather than copying and pasting
- Re-use code instead of rewriting it
- Make use of external libraries (*as long as it is not too cumbersome*)

First point is actually known as DRY principle in general software design and engineering

Plan for Mistakes

Mistakes are human and happen on all levels of software development (e.g. bug, misconception, problematic design choice), even in experienced teams

- Add assertions to programs to check their operation
  - Simple form of error detection
  - Can serve as documentation (that is auto-updated)
- Use an off-the-shelf unit testing library
  - Can assist in finding unexpected side effects of code changes
  - In case of GitLab: GitLab CI (continuous integration)
- Turn bugs into test cases
- Use a symbolic debugger (e.g. GDB, DDD, Gnome Nemiver, LLDB)

Assertions should only be used to catch misconceptions and omissions by programmers, not expected runtime errors (file not found, out of memory, solver didn't converge)!

The Importance of Backups

Losing your program code means losing months or even years of hard work. There are three standard ways to guard against this possibility:

- Create automatic backups (*but know how the restore process works!*)
- Create a manual copy on an external drive or another computer (*messy!*)
- Use a source code repository to manage the files

It's a good idea to use two approaches as a precaution

Of the above options, using a repository has the largest number of benefits

The Importance of Backups

Advantages of using a repository for code management:

- The repository can be on a different computer, i.e., the repository is automatically also an old-fashioned backup
- Repositories preserve project history, making it easy to retrieve older code or compare versions
- Repositories can be shared with others for collaboration

One of the tools most often used for this is Git, which can be used in conjunction with websites like GitHub or Bitbucket. An open source alternative is GitLab, which can also be installed locally to provide repositories for the research group.

Optimize Software Only After it Works Correctly

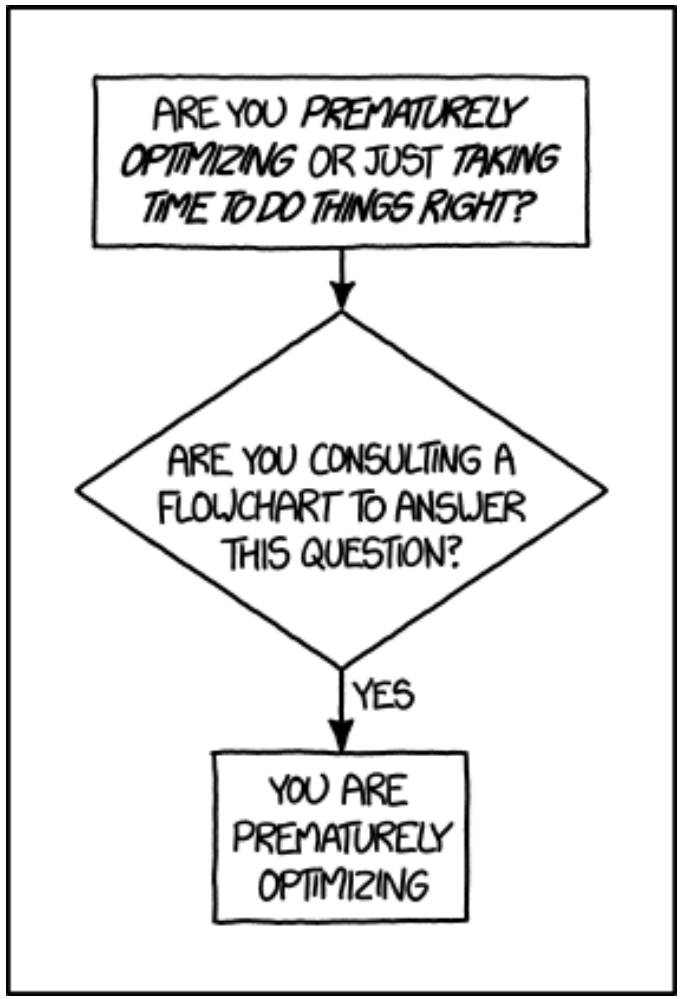
Identifying potential bottlenecks is hard, and optimization of other parts of the program is a waste of time

- Use a profiler to identify bottlenecks
- Write code in the highest-level language possible

This is an optimization with regard to development time

High-level prototypes (e.g. in Python, Matlab) can serve as oracles for low-level high-performance implementations (e.g. in C++)

Code and Coding Efficiency



Source: Randall Munroe, xkcd.com

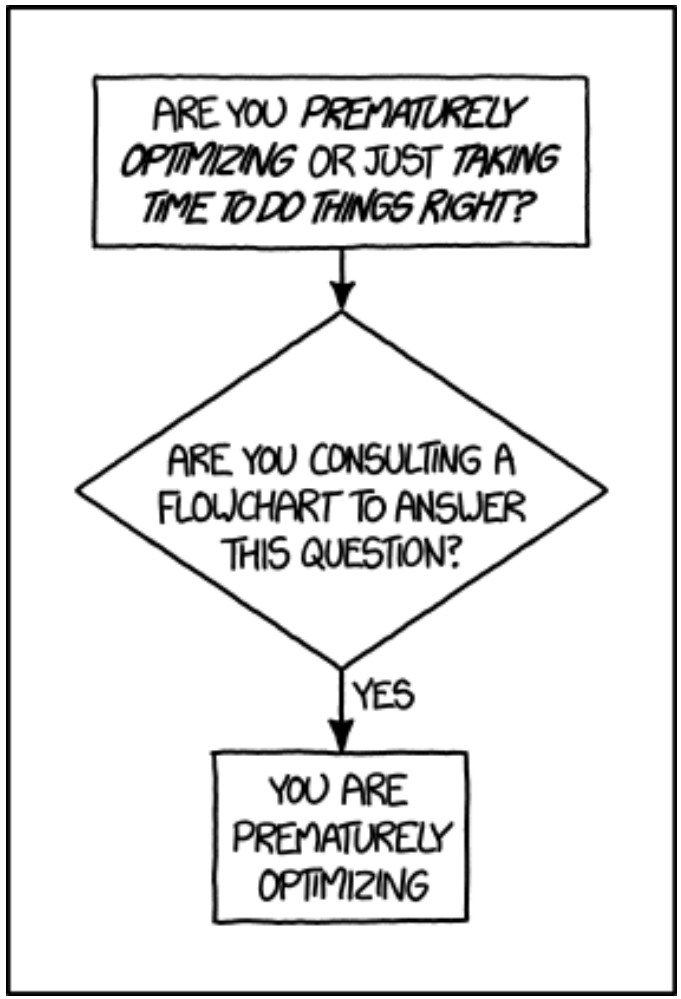
The first 90 percent of the code accounts for the first 90 percent of the development time. The remaining 10 percent of the code accounts for the other 90 percent of the development time.
— Tom Cargill, Bell Labs

Even experts are often unable to accurately predict the time requirements for developing a piece of software

This means:

- Project plans should be conservative
- Don't underestimate the work needed for the hard parts of coding

Code and Coding Efficiency



Source: Randall Munroe, xkcd.com

Hofstadter's Law: It always takes longer than you expect, even when you take into account Hofstadter's Law.
— Douglas Hofstadter, Gödel, Escher, Bach: An Eternal Golden Braid

As a result, there are two measures of efficiency, the time your program needs to run, and the time you need to write it

- Don't optimize too soon, get a working prototype first (you will likely need to change optimized parts along the way)
- Look for bottlenecks and only optimize those (your program will spend most of its time in roughly 5% of your code)

Document Design and Purpose, not Mechanics

The main purpose of documentation is assistance of readers who are trying to use an implementation, choosing between different implementations, or planning to extend one

- Document interfaces and reasons, *not* implementations
  - Function and method signatures
  - Public members of classes
- Refactor code in preference to explaining how it works
- Embed documentation for a piece of software in that software

Refactored code that needs less documentation will often save time in the long run, and documentation that is bundled with code has much higher chance to stay relevant

Collaborate

Scientific programs are often produced by research teams, and this provides both unique opportunities and potential sources of conflict

- Use pre-merge code reviews
- Use pair programming when bringing someone new up to speed and when tackling particularly tricky problems
- Use an issue tracking tool

Pre-merge reviews are the only way to guarantee that code has been checked by another person

Pair programming is very efficient but intrusive, and should be used with care