

# Introduction

Michael Hanke

School of Engineering Sciences

Program construction in C++ for Scientific Computing



# Outline

- 1 Introduction
- 2 A First C++ Program
- 3 Compiling and Debugging
- 4 A Simple Example: Newton's method
- 5 Functions, References, Pointers
- 6 Summary

# From Mathematical Formulae to Scientific Software

- Motivations
  - Computer simulation of physical processes
  - Physical process  $\longrightarrow$  mathematical model  $\longrightarrow$  algorithm  $\longrightarrow$  software program  $\longrightarrow$  simulation result
- Application of numerical algorithms (discrete approximations of analytical solutions)
- Widely used:
  - Simulation of natural phenomena
  - Applications in industry
  - Applications in medicine
  - Applications in finance

© Trygve Kastberg Nilssen and Xing Cai

# Scientific Software

- Desired properties
  - Correct
  - Efficient (speed, memory, storage)
  - Easily maintainable
  - Easily extendible
- Important skills
  - Understanding the mathematical problem
  - Understanding numerics
  - Designing algorithms and data structures
  - Selecting and using libraries and programming tools
  - Verify the correctness of the results
  - *Quick learning of new programming languages*

© Trygve Kastberg Nilssen and Xing Cai

# A Typical Scientific Computing Code

- Starting point
  - Computational problem
- Pre-processing
  - Data input and preparation
  - Build-up of internal data structure
- Main computation
- Post-processing
  - Result analysis
  - Display, output and visualization

© Trygve Kastberg Nilssen and Xing Cai

# A Two-Step Strategy

- Correct implementation of a complicated numerical problem is a challenging task
- Divide the task into two steps:
  - Express the numerical problem as a complete algorithm
  - Translate the algorithm into a computer code using a specific programming language

© Trygve Kastberg Nilssen and Xing Cai

# Choosing a Programming Language

- There exists hundreds (if not thousands) of different programming languages
- In Scientific Computing, only a few received wider attention: Fortran (77, 95, 2003, 2008), C, C++, Matlab (GNU Octave), Python, Maple/Mathematica (you may have other preferences)
- Issues that influence the choice of the programming language:
  - Computational efficiency
  - Costs of development cycle, maintainability
  - Built-in high performance utilities
  - Support for user-defined data types
- For different parts of the project, different programming languages may be suitable: Interoperability

# Properties

- Codes in compiled languages
  - run normally fast
  - longer development cycle
- Codes in interpreted languages
  - run normally slow
  - often fast development cycle (*very* high-level built-in numerical functionality)
- Different compiled languages may have different efficiency

# User-Defined Data Types

- Built-in primitive data types may not be enough for complicated numerical programming
- Need to “group” primitive variables into a new data type:
  - struct in C (only data, no function)
  - class in C++, Java & Python (and Fortran 2003)
  - Class hierarchies are a powerful tool: [Object-oriented programming \(OOP\)](#)
- OOP may lead to huge slow-down of your executables!

© Trygve Kastberg Nilssen and Xing Cai

# Course Contents

- Object-oriented programming, basic notions in, and syntax of, C++
- Objects, classes and its definition, constructors and destructors
- Operators, operator overloading, polymorphism
- Basics of abstract classes, inheritance, generic programming
- Selected components of the C++ standard library
- Structured and unstructured grids, data structures for their implementation
- Implementation of numerical methods for partial differential equations
- Efficient implementation of numerical algorithms

# Prerequisites

- Basic course in programming and computer science
- Basic course in numerical analysis
- Recommended: Advanced course in numerical analysis

# Outcomes

The aim of the course consists of providing knowledge how advanced numerical methods and complex algorithms in Scientific Computing can be implemented in C++.

After completion of the course the students can

- Construct simple classes for often used mathematical objects;
- Create abstract classes and define simple template classes;
- Implement data structures for manipulating realistic geometry and complex grids for numerically solving partial differential equations;
- Optimize data structures and algorithms in C++ with respect to efficient computations for large-scale problems.

- PRO1: 3 homework projects (from simple to advanced)
- TEN1: Written examination (a 4th homework can give up to 3 bonus credits)
- *The forth homework is compulsory for doctoral students!*
- Literature:
  - *Primary*: Stanley B. Lippman, Josée Lajoie, Barbara E. Moo: *C++ Primer (5th ed.)*. Addison-Wesley, 2013
  - Jan Skansholm: *C++ direkt (3:e upplagan)*. Studentlitteratur, 2013
  - Suely Oliveira, David Steward: *Writing Scientific Software: A Guide to Good Style*. Cambridge University Press, 2006
  - Yair Shapira: *Solving PDEs in C++ (2nd ed.)*. SIAM, 2012
- Register yourself in MyPages at latest on 31st August!

# Fibonacci's Numbers

The sequence of Fibonacci's numbers  $f_i$  is defined by the recursive definition

$$f_0 = 0, \quad f_1 = 1,$$

$$f_i = f_{i-1} + f_{i-2}, \quad i = 2, 3, \dots$$

Compute Fibonacci's Numbers:  
matlab

```
% Computation of the first 10 numbers
clear
n = 10;
f = zeros(1,n);           % Not necessary, but useful
f(1) = 0;
f(2) = 1;
for i = 3:n
    f(i) = f(i-1)+f(i-2);
end
fprintf('%8d',f)
```

Compute Fibonacci's Numbers:  
Java

```
// Computation of the first 10 numbers
import java.lang.*;
import java.io.*;
class fibonacci {
    public static void main(String [] str) {
        int i, n = 10;
        int [] f = new int[n];
        f[0] = 0;
        f[1] = 1;
        for (i = 2; i < n; i++) {
            f[i] = f[i-1]+f[i-2];
        }
        for (i = 0; i < n; i++)
            System.out.print(f[i] + " ");
        System.out.println("");
    }
}
```

## General:

- C++ is case sensitive.
- All valid statements are terminated by a semi-colon, `i = i+1;`
- Several statements can be collected in a compound statement, `{i = i+1; j = j+2;}`
- All variables must have a specified type.
- All names must be declared (or defined) before use!

## Comments:

- everything between `/*` and `*/`
- from `//` to end of line

## Short-hand operators:

- The following two statements are equivalent: `a = a+b;` and `a += b;` (`--`, `*=`, `/=`).
- The following are equivalent: `i = i+1;` `i++;` and `++i;` (*as far as the effect on `i` is concerned!*)

## Fibonacci's Numbers: C++

```
// Implementation: C-style
#include <iostream>
#include <iomanip>
#define n 10 /* Convention: Use caps: N */
int main() {
    int i, f[n];
    f[0] = 0;
    f[1] = 1;
    for (i = 2; i < n; i++) {
        f[i] = f[i-1]+f[i-2];
    }
    for (i = 0; i < n; i++)
        std::cout << std::setw(8) << f[i] << ", ";
    std::cout << std::endl;
    return 0;
}
```

Note: *C++ and Java are really different programming languages!*

# Access to Library Functions

- Most of the functionality of C++ is included in libraries
- Two types of libraries: General functions vs Standard Template Library (STL)
- In order to get access to the libraries, their declarations must be included by issuing a preprocessor directive

```
#include <library>
```

- The **C-libraries** are compatible with C++. In order to get access to them, a modified version of the C header file should be included: For example, the mathematical library `math.h` can be used via

```
#include <cmath>
```

# Namespaces

- Every name belongs to a **namespace**.
- Namespaces are used to avoid collisions between identifiers in libraries and our own definitions.
- All names of the C++ standard libraries belong to the namespace `std`.
- Names in namespaces can be accessed via the double colon notation,

```
std::cout
```

- A namespace can be made “visible” by using the following construct:

```
using namespace std;
```

- *Warning: Be careful to ambiguities when opening many namespaces!*

## Scope of a Name

- At a particular point in a program, each name refers to a specific entity.
- However, a given name can be reused to refer to different entities at different points in a program.
- Names are visible from the point they are declared until the end of the scope in which the declaration appears.
- Example:

```
#include <iostream>
int main() {
    int sum = 0;
    for (int i = 1; i <= 10; i++) sum += i;
    std::cout << sum << std::endl;
    return 0;
}
```

`main` has global scope, `sum` is visible inside the `main` block, `i` is only visible in the `for` loop.

- It is completely valid to have nested scopes!

# Understanding the Compilation Process: Preprocessor

- The compilation of a source file into an object code proceeds in two steps:
  - Execution of a preprocessor: Generation of the “pure” C++ code
  - Invokation of the compiler
- The preprocessor interpretes Preprocessor directives:  
`#<directive> [<parameters>]`
- Most important directives:
  - `#include` (inclusion of a header file)
  - `#define` (definition of macros)
  - `#ifdef ... #else ... #endif` (conditional compilation)
  - `#ifndef` (analogous)

- An important usage example:

```
#define DEBUG 1
...
#ifdef DEBUG
... (code for debugging program version)
#endif
```

- Macros can be defined on the compiler command line.
- Note: **By convention, the macro NDEBUG is used for switching debugging mode!**

- I/O is organised by using `streams`.
- The I/O functionality is provided in the `iostream` (and `omanip`) libraries.
- The standard output stream is `cout`, the standard input stream `cin`.
- `cout` and `cin` are streams of characters.
- For each item to transfer, automatic conversion to/from character streams will take place.
- `std::setw(int)` sets the length of the output field
- `std::endl` is the end-of-line marker.
- `cin` works as expected for terminal input.
- `cerr` is the standard output for error and debug messages.

```
#include <iostream>
int main() {
    int i;
    std::cout << "Enter an integer: ";
    std::cin >> i;
    std::cout << "You entered " << i << std::endl;
}
```

The C I/O routines can also be used. However, **one should avoid to use both for the same streams!**

## Flow Control: for Statement

Loops iterate over statements

for statement

```
for (expr1; expr2; expr3)  
    statement
```

The expressions should be interpreted as follows:

*expr1* Executed before the first iteration

*expr2* Iterate while expression is true

*expr3* Executed at the end of each iteration

*expr1* and *expr3* may contain several comma separated statements  
(not terminated by ;)

# Statements

- A statement is either a simple statement or a compound statement.
- A simple statement is an expression followed by a semicolon (;).
- A compound statement has the syntax (note the missing semicolon at the end!)

```
{ statement; statement; ... }
```

# Flow Control: `if` Statement

An `if` statement allows the program to select different execution paths depending on the data.

## `if` statement

```
if (expression)
    statement
else
    statement
```

- Any valid statements are allowed including compound statements and new `if` statements.
- The `else` clause is optional.

## if Statement (cont)

Example for computing  $\max(a, b)$ :

```
if (a > b)
    max = a;
else
    max = b;
```

Useful operators:

- Equality: `==`, `!=`
- Relational: `<`, `<=`, `>=`, `>`
- Logical: `&&` (and), `||` (or), `!` (not)

# Understanding

What is the difference between the following code snippets?

- Example 1:

```
#define MAX(a,b) ((a) > (b) ? (a) : (b))
```

- Example 2:

```
max = a > b ? a : b;
```

- Example 3:

```
if (a > b) max = a;  
else max = b;
```

## Example: if Statements

Which of the following code snippets is syntactically correct? Why or why not?

- Example 1:

```
if (a > b) {  
    max = a;  
};  
else {  
    max = b;  
}  
c = max;
```

- Example 2:

```
if (a > b) {  
    max = a;  
};  
c = max;
```

Flow Control: `while` Statement`while` and `do/while` statements

```
while (expr1)  
    statement  
do statement  
while (expr2);
```

The statements should be interpreted as follows:

*while* The statement is executed as long as *expr1* is true (possibly never).

*do* The statement is executed at least once until *expr2* becomes false.

*expr1* and *expr2* may contain several comma separated statements (not terminated by `;`).

# Example: while Statement

What is the result of the following code snippet?

```
int i = 0, j = 0;
while (i < 2, j < 10) { i++; j++; }
cout << i << endl << j << endl;
```

- The extensions for C++ source files are .cpp, .cc, .C
- g++
  - g++ `-Wall -o prog prog.cpp`
  - Add `-g` to enable debugging
  - Add `-O, -O2, -O3` for optimization (until `-O6`)
  - The program is automatically linked against standard libraries.
  - *Always check correctness of output when using optimization!*
- Optimizing compilers
  - Good for performance, usually not as good as g++ with respect to error messages and warnings
  - Examples: Intel's `icpc`, Portland's `pgc++`, Sun's `CC`
  - I am using g++ 4.8.5 in all my examples.
  - Check C++11 compatibility!

# Debugging

## Fact

*The best and most efficient way of debugging is writing a clear and well structured code.*

- Together with your code, develop a debugging strategy
- Instrument your code with debugging instructions
  - ```
#include <cassert>
assert(...);
```
  - ```
#ifndef NDEBUG
cerr << "I am alive";
cerr << __FILE__ << ", " << __LINE__ << endl;
#endif
```
- In C++, there are very elegant ways to implement debugging routines. (operator overloading)

- gdb: Very basic, tedious to use
  - `g++ -g -Wall -o prog prog.cpp`
  - `gdb prog`
    - In gdb: type `run` to start your program
    - `bt`: prints current call stack (list of nested functions)
    - `p x`: prints value of variable `x`
    - `break file.c:123`: sets break point
    - `continue`: continues execution
    - `clear 1`: removes break point 1
    - `l`: lists program code
- Valgrind: memory debugger; uses emulation

```

DDD: /home/hanke/vorlesun/parpro1/FC/fibonacci.c
File Edit View Program Commands Status Source Data Help
(): cci.c:16
Lookup Find Clear Watch Print Display Plot Show Rotate Set Undisp

/* Simple test program for showing C */
#include <stdio.h>

#define n 10

int main(int argc, char *argv[]) {
    int i, f[n];

    f[0] = 1;
    f[1] = 1;

    for (i = 2; i < n; i++)
        f[i] = f[i-1]+f[i-2];

    for (i = 0; i < n; i++)
        printf("%8d", f[i]);
        printf("\n");
}

argv=<value optimized> at fibonacci.c:16
(gdb) ]

Breakpoint 1, main (argc=<value optimized>, argv=<value optimi

```

# Performance Tools

- A profiler gives time spent in various functions (subroutines)
- `gprof` (Read the compiler manual!)
  - compile with `-pg`
  - run `prog`
  - run `gprof ./prog >prog.prof`
  - look at statistics in `prog.prof`



# The Task

Consider the nonlinear equation

$$x = \cos x, \quad x \in \mathbb{R}.$$

Find a solution!

- It is easy to see that this equation has exactly one solution.
- We estimate that the solution is close to 0.7.
- Newton's method is an appropriate method for solving the equation.
- It's convergence depends on a good initial guess.

# The Algorithm

- Let  $f(x) = x - \cos x$ .
- Iteration:

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}$$

- In order to find an appropriate initial guess: Let us experiment with  $x_0$ .

```
#include <iostream>
#include <cmath>
using namespace std;
int main() {
    cout << "Give initial guess " << endl;
    double x;
    cin >> x;
    double err, tol=1e-12, x1;
    int it, maxit=100;
    it = 0;
    err = tol + 1;
    while( err > tol && it < maxit ) {
        x1 = x - (x-cos(x))/(1+sin(x));
        err = fabs( x1 - x ); x = x1; it++;
    }
    if( err <= tol ) cout << "The root is " << x << endl;
    else cout << "Error, no convergence \n";
}
```

# C++ Standard Types

- Integer types** char, short int, int, long int, long long int, bool  
Integer types can have the attribute unsigned (like in unsigned char).
- Float types** float, double, long double
- Pointers** Contain addresses of objects
- References** Explained later
- Void** Describe “nothing”

## Function definition

```
return-type function-name(parameters) {  
    // statements  
    return value;  
}
```

**return-type** type of value returned or void if none

**function-name** name

**parameters** comma separated list of types and names of parameters

**value** value returned upon termination (not needed if *return-type* is void)

- The function name and the sequence of parameter types are called the *signature* of the function.
- Several functions can have the same name if only their signature differs (function overloading): Extremely important for object-oriented programming.

## Function Definition (cont)

- Example:

```
double average(double x, double y) {  
    return 0.5*(x+y);  
}
```

# The main Function

- Every program must have exactly one (nonmember) `main` function.
- Program execution starts at the `main` function.
- The signatures of `main` may be:

```
int main()  
int main(int argc, char *argv[])
```
- `argc` is the number of arguments given to the program, while `argv` contains `argc` of (C-style) strings (the actual arguments).
- A return of 0 means generally success.
- However, it is safer to use predefined variables: `EXIT_SUCCESS`, `EXIT_FAILURE` etc.

- A function definition includes a complete description of the internals. It will be compiled if available.
- What about incremental compilation?
- Assume that we have written a function becoming part of a library. Later on, it shall be used in a main program.
- Since the internals are unimportant for the main program, it is sufficient to know the interface to that function. Such functionality is provided by a **function declaration**:

## Function declaration

```
return-type function-name(parameters);
```

# Function Call

- A function is called by giving its name and the parameters in sequence. The parameters must have a type corresponding to the functions definition:

```
function-name(parameters)
```

- By default, all parameters are copied into local variables in the function body (call by value).
- Hence, changes made to the parameters have no effect outside the function.
- If changes of parameters should have effect outside the function, the argument must be *passed by reference*.
- Passing by reference is indicated by the &-operator:

```
type function-name(atype& byref, ...);
```

- Note: In order to avoid excessive memory copying for huge objects, call by reference should be used.

## Example: Call by Reference

```
#include <iostream>
using namespace std;
void change(int val, int& ref) {
    val = 1;
    ref = 1;
}

int main() {
    int i = 0, j = 0;
    cout << i << j << " --> ";
    change(i,j);
    cout << i << j << endl;
}
```

Output: 00 --> 01

# Reference Variables

- In the same way as pointer variables, reference variables can be defined:

```
int n;  
int &ri = n;
```

Semantics: `ri` is a (constant!) pointer to `n`. Logically, it is simply another name for `n`.

- Since a reference variable cannot change its value, it must always be initialized when defined!
- However, an expression `ri = 5;` is well-defined and leads, in our example, to setting `n` to 5.

# Parameters With Default Values

- Sometimes, certain parameters for a function may include reasonable defaults.
- Example: A standard value for the second argument of our average function ( $y$ ) could be 1.
- There are two possibilities to resolve this situation:
  - Define two versions of the average function with different signature:

```
double average(double x, double y);  
double average(double x);
```
  - Use default values:

```
double average(double x, double y = 1.0);
```
- Note: If one parameter in the parameter list has a default value, all subsequent must have it, too.

# C-Style Arrays

All basic and derived types (including classes) can be extended to be vector-valued,

## C-style array

```
type name[N];
```

- Memory for N type-objects is allocated (statically) when the variable enters scope.
- *Note:* The size N must be known at compile time!
- Elements are accessed by name[i] where  $0 \leq i < N - 1$ .
- Example:

```
double gridpoint[5];  
for (int i = 0; i < 5; i++)  
    gridpoint[i] = 0.25*i;
```

- *Note:* Indexing errors are not caught by the compiler and may cause strange behavior at run time.

# Multi-Dimensional Arrays

Arrays can be extended to several dimensions,

## Multi-dimensional arrays

```
type name[N1] ... [Nk];
```

- Elements are accessed by `name[i1] ... [ik]`, where  $0 \leq i_j < N_j$ .
- If possible, multi-dimensional arrays should be avoided for efficiency.
- Multi-dimensional arrays should be mapped directly to one-dimensional ones by using an appropriate index mapping. Example: Fortran-style mapping of an  $M \times N$ -matrix:

$$a(i, j) \mapsto a[i + j * N]$$

# Pointers

- A pointer is an object containing an address of main memory.
- A pointer is allowed “to point” to objects of a certain type.
- **Pointer variables can be used as any other objects.**
- A pointer may be uninitialized or pointing to a non-existing object (for example if a variables leaves scope). This is called a **hanging pointer**.
- *Using a hanging pointer is forbidden!*
- **Note: Using a hanging pointer is one of the most common programming mistakes and extremely hard to debug!**

## Definition of pointers

*type \*p;*

Note: In the definition *type \*p, q;* *q* is *not* a pointer but a variable of type *type*!

# Operations With Pointers

- Making a pointer to an object:

```
type a;  
type *b = &a;
```

- Dereferencing: Finding the value of an object a pointer is pointing to.

```
int *p, a = 1;  
p = &a;  
cout << *p << std::endl;
```

## Examples: Pointers 1

```
#include <iostream>
using namespace std;
void change(int val, int& ref, int *ptr) {
    val = 1;
    ref = 1;
    *ptr = 1;
}

int main() {
    int i = 0, j = 0, k = 0;
    cout << i << j << k << " --> ";
    change(i,j,&k);
    cout << i << j << k << endl;
}
```

Output: 000 --> 011

Using pointers in function calls is not recommended! The only exception are C-type arrays.

```
#include <iostream>
using namespace std;
int main() {
    int a = 1, *p, *q;
    p = &a;
    q = p;
    *q = 2;
    cout << *p << *q << endl;
}
```

What is the output of this program?

- In a definition of an array *type*  $a[n]$ , the variable  $a$  is of type *type*  $*$ !

- Example:

```
double a[10];  
double *p1, *p2;  
p1 = &a[0];  
p2 = a;
```

The expression  $p1 == p2$  evaluates to true.

- Pointer arithmetic: For any nonnegative integer  $i$ ,  $*(a+i)$  and  $*(i+a)$  are identical to  $a[i]$ .
- Recommendation: In function declarations use *type*  $a[]$  instead of *type*  $*a$  (even if they are equivalent).

# Application of Pointer Arithmetic

When traversing an array, the following two code snippets are identical:

- Example 1

```
double a[10];  
for (int i = 0; i < 10; i++) a[i] = 0.0;
```

- Example 2

```
double a[10];  
for (double *p = a; p < a+10; p++) *p = 0.0;
```

# Dynamic Arrays

## Observations:

- The definition `double a[10];` allocates memory for 10 double objects at compile time, and stores a pointer to the memory block in `a`.
- The definition `double *p;` allocates memory for an address, only.

*Dynamic arrays must be allocated at run time. So a different mechanism is needed.*

## Dynamic Arrays (cont)

- Dynamic arrays can be allocated by  
$$\textit{pointer-var} = \textit{new type[size]};$$
- `new` allocates memory for size objects and returns the address of this block.

Example:

```
double *x;  
int n;  
cin >> n;  
x = new double[n];  
for (int i = 0; i < n; i++) x[i] = 0.1*i;
```

- Memory no longer needed should be deallocated such that it can be used for other purposes:

```
delete [] pointer-var;
```

# Some Tips And Pitfalls

- Allocating and deallocating is associated with an overhead. Try to “reuse” memory if possible.
- Memory should be deallocated before a pointer exits scope (Danger of **memory leak!**).
- Accessing a deallocated object or using an uninitialized pointer is forbidden (unpredictable program behavior!).
- In particular, **if two pointer point to the same memory region, deallocating one of them invalidates the other, too!**
- In order to allow for garbage collection it is always a good idea to deallocate memory in the opposite order of allocation.
- Deallocating memory is often the main purpose of class destructors.
- Not recommended: Explicit usage of malloc/free.

# Summary

- Basic C++ syntax has been introduced.
  - Pointers and references have been discussed.
  - C-style arrays are introduced.
  - We wrote our first C++ program.
  - We know how to compile and run a C++ program.
- 
- What will come next?
    - A more advanced example.