Programming techniques for scientific simulations

Autumn semester 2016

Preparing for the course

- D-PHYS account: https://admin.phys.ethz.ch/newaccount
- Software to install on your computer
  - All operating systems:
    - C++ compiler
    - git
    - CMake
  - Additionally for Linux:
    - make
  - Additionally for MacOS X:
    - Xcode with command line tools
- The assistants will help you in the exercise classes

Lecture homepage

- http://tinyurl.com/ethz-pt16
- Sign up for an exercise group
- Updated regularly with lecture contents:
  - News about the class
  - Lecture notes
  - Exercise sheets
- Discussion forum: ask your classmates!

About the course

- RW (CSE) students
  - Mandatory lecture in the 3rd semester in the bachelor curriculum
- Physics students
  - Recommended course as preparation for:

    Computational Physics Courses:

    Introduction to Computational Physics (AS)
    Computational Statistical Physics (SS)
    Computational Quantum Physics (SS)

    Semester thesis in Computational Physics
    Masters thesis in Computational Physics
    PhD thesis in Computational Physics

Contents of the lecture

- Important skills for scientific software development
  - Version control
  - Build systems
  - Debugging
  - Profiling and optimization
- Advanced C++ programming
  - Object oriented programming
  - Generic programming and templates
  - Runtime and compile time polymorphism
- Libraries
  - High performance libraries: BLAS, LAPACK
  - C++ libraries: Standard library, Boost
  - Library design

Why C++?

- Generic high level programming
  - Shorter development times
  - Smaller error rate
  - Easier debugging
  - Better software reuse
- Efficiency
  - As fast or faster then FORTRAN
  - Faster than C, Pascal, …
- Job skills
  - We all need to find a job some day...

Generic programming

- Print a sorted list of all words used by Shakespeare

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <string>
#include <iterator>

using namespace  std;

int main()
 {
   vector<string> data;
   copy(istream_iterator<string>(cin), istream_iterator<string>(), back_inserter(data));
   sort(data.begin(), data.end());
   unique_copy(data.begin(), data.end(), ostream_iterator<string>(cout,"\n"));
}
```

Why C++?

	C++	C	Java	FORTRAN	FORTRAN 95
Efficiency	√√	√	×	√√	√
Modular Programming	√	√	√	×	√
Object Oriented Programming	√	×	√	×	√
Generic Programming	√	×	×	×	×

A first C++ program

```
/* A first program */

#include <iostream>

using namespace std;

int main()
{
  cout << "Hello students!\n" ;
 // std::cout without the using declaration
 return 0;
}
```

- /* and */ are the delimiters for comments
- includes declarations of I/O streams
- declares that we want to use the standard library ("std")
- the main program is always called "main"
- "cout" is the standard output stream.
- "<< " is the operator to write to a stream
- statements end with a ;
- // starts one-line comments
- A return value of 0 means that everything went OK

More about the std namespace

```
#include <iostream>
using namespace std;
int main()
{
  cout << "Hello\n" ;
}
```

```
#include <iostream>
using std::cout;
int main()
{
  cout << "Hello\n" ;
}
```

```
#include <iostream>
int main()
{
  std::cout << "Hello\n" ;
}
```

- All these versions are equivalent
- Feel free to use any style in your program
- Never use using statements globally in libraries!

A first calculation

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

using namespace std;

int main()
{
 cout << "The square root of 5 is"
 << sqrt(5.) << "\n" ;
 return 0;
}
```

- `<cmath>` is the header for mathematical functions
- Output can be connected by `<<`
- Expressions can be used in output statements
- What are these constants?
  - `5.`
  - `0`
  - `"\n"`

Integral data types

- Signed data types
  - `short, int, long, long long`
  - Not yet standard: `int8_t, int16_t, int32_t, int64_t`
- Unsigned data types
  - `unsigned short`, `unsigned int, unsigned long, unsigned long long`
  - Not yet standard: `uint8_t, uint16_t, uint32_t, uint64_t`
- Are stored as binary numbers
  - `short`: usually 16 bit
  - `int`: usually 32 bit
  - `long`: usually 32 bit on 32-bit CPUs and 64 bit on 64-bit CPUs
  - `long long`: usually 64 bits

Integer representations

- An n-bit integer is stored in $n/8$ bytes
  - Little-endian: least significant byte first
  - Big-endian: most significant byte first
  - Exercise: write a program to check the format of your CPU
- Unsigned | | | | `n bits mantissa x` | | | | | |
  - x just stored as n bits, values from 0 … 2^n-1
- Signed | S | | | `n-1 bits mantissa x` | | | |
  - Stored as 2's complement, values from -2^{n-1} … $2^{n-1}-1$
  - Highest bit is sign **S**
  - $x \geq 0$: **S**=0, rest is x
  - $x < 0$: **S**=1, rest is `~(-x -1)`
  - Advantage of this format: signed numbers can be added like unsigned

Integer constants

- Integer literals can be entered in a natural way
- Suffixes specify type (if needed)
  - int: `0`, `-3`, ….
  - unsigned int: `3u`, `7U` ,...
  - short: `0S`, `-5s`, ...
  - unsigned short: `1us`, `9su`, `6US`, ...
  - long: `0L`, `-5l`, ...
  - unsigned long: `1ul`, `9Lu`, `6Ul`, ...
  - long long: `0LL`, `-5ll`, ...
  - unsigned long long: `1ull`, `9LLu`, `6Ull`, ...

Characters

- Character types
  - Single byte: `char, unsigned char, signed char`
    - Uses ASCII standard
  - Multi-byte (e.g. for Japanese: 大): `wchar_t`
    - Unfortunately is not required to use Unicode standard
- Character literals
  - `'a', 'b', 'c', '1', '2', …`
  - `'\t'` … tabulator
  - `'\n'` … new line
  - `'\r'` … line feed
  - `'\0'` … byte value 0

Strings

- String type
  - C-style character arrays `char s[100]` should be avoided
  - C++ class `std::string` for single-byte character strings
  - C++ class `std::wstring` for multi-byte character strings
- String literals
  - `"Hello"`
  - Contain a trailing '\0', thus `sizeof("Hello")==6`

Boolean (logical) type

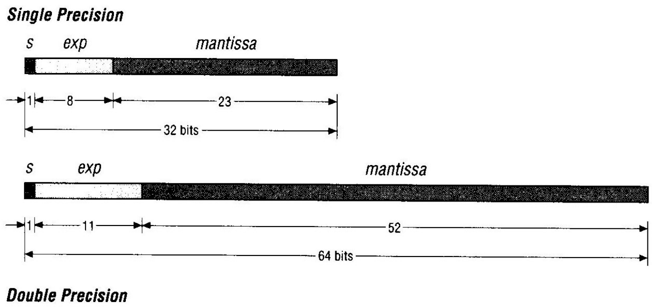
- Type
  - `bool`
- Literal
  - `true`
  - `false`

Floating point numbers

- Floating point types
  - single precision: `float`
    - usually 32 bit
  - double precision: `double`
    - Usually 64 bit
  - extended precision: `long double`
    - Often 64 bit (PowePC), 80 bit (Pentium) or 128 bit (Cray)
- Literals
  - single precision: `4.562f`, `3.0F`
  - double precision: `3.1415927`, `0.`
  - extended precision: `6.54498467494849849489L`

IEEE floating point representation

- The 32 (64) bits are divided into sign, exponent and mantissa



Type	Exponent	Mantissa	Smallest	Largest	Base 10 accuracy
`float`	8	23	1.2E-38	3.4E+38	6-9
`double`	11	52	2.2E-308	1.8E+308	15-17

Converting to/from IEEE representation

- Sign
  - Positive: 0,   Negative: 1
- Mantissa
  - Left shifted until leftmost digit is 1, other digits are stored
- Exponent
  - Half of the range (127 for float, 1023 for double) is added

```
172.625                        Base 10
10101100.101 X 2 ** 0          Base 2
1.0101100101 X 2 ** 7          Base 2 Normalized
```

Add 127 for bias=134

```
0 10000110 01011001010000000000000
             1. Assumed bit and binary point
```

Floating point arithmetic

- Truncation can happen because of finite precision

$$\begin{array}{l} 1.00000 \\ 0.0000123 \\ \hline 1.00001 \end{array}$$

- Machine precision ε is smallest number such that $1+\varepsilon \neq 1$
  - Exercise: calculate ε for `float`, `double` and `long double` on your machine
- Be very careful about roundoff
  - For example: sum numbers starting from smallest to largest
  - See examples provided

Implementation-specific properties of numeric types

- defined in header `<limits>`
- `numeric_limits<T>::is_specialized` // is true if information available
- most important values for integral types
  - `numeric_limits<T>::min()` // minimum (largest negative)
  - `numeric_limits<T>::max()` // maximum
  - `numeric_limits<T>::digits` // number of bits (digits base 2)
  - `numeric_limits<T>::digits10` // number of decimal digits
  - and more: `is_signed`, `is_integer`, `is_exact`, ...
- most important values for floating point types
  - `numeric_limits<T>::min()` // minimum (smallest nonzero positive)
  - `numeric_limits<T>::max()` // maximum
  - `numeric_limits<T>::digits` // number of bits (digits base 2)
  - `numeric_limits<T>::digits10` // number of decimal digits
  - `numeric_limits<T>::epsilon()` // the floating point precision
  - and more: `min_exponent`, `max_exponent`, `min_exponent10`, `max_exponent10`, `is_integer`, `is_exact`
- first example of templates, use by replacing T above by the desired type: `std::numeric_limits<double>::epsilon()`

A more useful program

```
#include <iostream>
#include <cmath>
using namespace std;
int main()
{
  cout << "Enter a number:\n" ;
  double x;
  cin >> x;
  cout << "The square root of " << x << " is "
  << sqrt(x) << "\n" ;
  return 0;
}
```

- a variable named 'x' of type 'double' is declared
- a double value is read and assigned to x
- The square root is printed

Variable declarations

- have the syntax: `type variablelist;`
  - `double x;`
  - `int i,j,k;` // multiple variables possible
  - `bool flag;`
- can appear anywhere in the program

```
int main() {
…
double x;
}
```

- can have initializers, can be constants
  - `int i=0;` // C-style initializer
  - `double r(2.5);` // C++-style constructor
  - `const double pi=3.1415927;`

Advanced types

- **Enumerators** are integer which take values only from a certain set

```
enum trafficlight {red, orange, green};
enum occupation {empty=0, up=1, down=2, updown=3};
trafficlight light=green;
```

- **Arrays** of size n

```
int i[10]; double vec[100]; float matrix[10][10];
```

  - indices run from 0 … n-1! (FORTRAN: 1…n)
  - last index changes fastest (opposite to FORTRAN)
  - Should not be used in C++ anymore!!!
- Complex types can be given a new name

```
typedef double[10] vector10;
vector10 v={0,1,4,9,16,25,36,49,64,81};
vector10 mat[10]; // actually a matrix!
```

Expressions and operators

- Arithmetic
  - multiplication: `a * b`
  - division: `a / b`
  - remainder: `a % b`
  - addition: `a + b`
  - subtraction: `a - b`
  - negation: `-a`
- Increment and decrement
  - pre-increment: `++a`
  - post-increment: `a++`
  - pre-decrement: `--a`
  - post-decrement: `a--`
- Logical (result bool)
  - logical not: `!a`
  - less than: `a < b`
  - less than or equal: `a <= b`
  - greater than: `a > b`
  - greater than or equal: `a >= b`
  - equality: `a == b`
  - inequality: `a != b`
  - logical and: `a && b`
  - logical or: `a || b`
- Conditional: `a ? b : c`
- Assignment: `a = b`

Bitwise operations

- Bit operations
  - bitwise not: `~a`
    - inverts all bits
  - left shift: `a << n`
    - shifts all bits to higher positions, fills with zeros, discards highest
  - right shift: `a >> n`
    - shifts to lower positions
  - bitwise and: `a & b`
  - bitwise xor: `a ^ b`
  - bitwise or: `a | b`
- The **bitset** class implements more functions. We will use it later in one of the exercises.
- Interested students should refer to the recommended C++ books
- The shift operators have been redefined for I/O streams:
  - `cin >> x;`
  - `cout << "Hello\n" ;`
- The same can be done for all new types: "operator overloading"
- Example: **matrix operations**
  - `A+B`
  - `A-B`
  - `A*B`

Compound assignments

- `a *= b`
- `a /= b`
- `a %= b`
- `a += b`
- `a -= b`
- `a <<= b`
- `a >>= b`
- `a &= b`
- `a ^= b`
- `a |= b`
- `a += b` equivalent to `a=a+b`
- allow for simpler codes and better optimizations

Special operators

- scope operators: `::`
- member selectors
  - `.`
  - `->`
- subscript `[]`
- function call `()`
- construction `()`
- `typeid`
- casts
  - `const_cast`
  - `dynamic_cast`
  - `reinterpret_cast`
  - `static_cast`
- `sizeof`
- `new`
- `delete`
- `delete[]`
- pointer to member select
  - `.*`
  - `->*`
- `throw`
- comma `,`
- all these will be discussed later

Operator precedences

- Are listed in detail in all reference books or look at http://www.cppreference.com/operator_precedence.html
- Arithmetic operators follow usual rules
  - `a+b*c` is the same as `a+(b*c)`
- Otherwise, *when in doubt use parentheses*

Statements

- simple statements
  - `;` // null statement
  - `int x;` // declaration statement
  - `typedef int index_type;` // type definition
  - `cout << "Hello world" ;` // all simple statements end with ;
- compound statements
  - more than one statement, enclosed in curly braces

```
{
 int x;
 cin >> x;
 cout << x*x;
}
```

The if statement

- Has the form

```
if (condition)
   statement
```

- or

```
if (condition)
   statement
else
   statement
```

- can be chained

```
if (condition)
   statement
else if(condition)
   statement
else
   statement
```

- Example:

```
if (light == red)
   cout << "STOP!" ;
else if (light == orange)
   cout << "Attention" ;
else {
   cout << "Go!" ;
}
```

The switch statement

- can be used instead of deeply nested if statements:

```
switch (light) {
    case red:
      cout << "STOP!" ;
      break;
    case orange:
      cout << "Attention" ;
      break;
    case green:
      cout << "Go!" ;
      go();
      break;
    default:
      cerr << "illegal color" ;
      abort();
}
```

- do not forget the `break`!
- always include a default!
  - the telephone system of the US east coast was once disrupted completely for several hours because of a missing default!
- also multiple labels possible:

```
switch(ch) {
    case 'a' :
    case 'e' :
    case 'i' :
    case 'o' :
    case 'u' :
      cout << "vowel" ;
    break;

  default:
      cout << "other
  character" ;
  }
```

The for loop statement

- has the form

```
for (init-statement ; condition ; expression)
    statement
```

- example:
  - ```
    for (int i=0;i<10;++i)
     cout << i << "\n" ;
    ```

- can contain more than one statement in for(;;), but this is very bad style!
  - ```
    double f;
    int k;
    for (k=1,f=1 ; k<50 ; ++k, f*=k)
      cout << k << "! = " << f<< "\n" ;
    ```

The while statement

- is a simpler form of a loop:

```
while (condition)
   statement
```

- example:

```
while (trafficlight()==red){
     cout << "Still waiting\n" ;
     sleep(1);
  }
```

The do-while statement

- is similar to the while statement

```
do
   statement
while (condition);
```

- Example

```
do {
   cout << "Working\n" ;
   work();
} while (work_to_do());
```

The break and continue statements

- `break` ends the loop immediately and jumps to the next statement following the loop
- `continue` starts the next iteration immediately
- An example:

```
while (true) {
  if (light()==red)
    continue;
  start_engine();
  if(light()==orange)
    continue;
  drive_off();
  break;
}
```

A loop example: what is wrong?

```
#include <iostream>
using namespace std;
int main()
{
  cout << "Enter a number: ";
  unsigned int n;
  cin >> n;

  for (int i=1;i<=n;++i)
    cout << i << "\n";

  int i=0;
  while (i<n)
    cout << ++i << "\n";

  i=1;
  do
    cout << i++ << "\n";
  while (i<=n);

  i=1;
  while (true) {
      if(i>n)
     break;
    cout << i++ << "\n";
  }
}
```

The goto statement

- will not be discussed as it should not be used
- included only for machine produced codes, e.g. FORTRAN -> C translators
- can always be replaced by one of the other control structures
- **we will not allow any `goto` in the exercises!**

Static memory allocation

- Declared variables are assigned to memory locations

```
int x=3;
int y=0;
```

- The variable name is a symbolic reference to the contents of some real memory location
  - It only exists for the compiler
  - No real existence in the computer

address	contents	name
0	3	x
4	0	y
8		
12		
16		
20		
24		
28		

Pointers

- Pointers store the address of a memory location
  - are denoted by a * in front of the name

    `int *p;` // pointer to an integer

  - Are initialized using the & operator

    `int i=3;`

    `p =&i;` // & takes the address of a variable

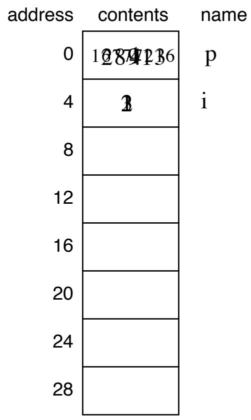
  - Are dereferenced with the * operator

    `*p = 1;` // sets i=1

  - Can be dangerous to use

    `p = 1;` // sets p=1: danger!

    `*p = 258;` // now messes up everything, can crash
- Take care: `int *p;` does not allocate memory!



Dynamic allocation

- Automatic allocation
  - `float x[10];` // allocates memory for 10 numbers
- Allocation of flexible size
  - `unsigned int n; cin >> n; float x[n];` // will not work
  - The compiler has to know the number!
- Solution: dynamic allocation
  - `float *x=new float[n];` // allocate some memory for an array
  - `x[0]=…;…` // do some work with the array x
  - `delete[] x;` // delete the memory for the array. x[i], *x now undefined!
- Don't confuse
  - `delete`, used for simple variables
  - `delete[]`, used for arrays

Pointer arithmetic

- for any pointer `T *p;` the following holds:
  - `p[n]` is the same as `*(p+n);`
- Adding and integer *n* to a pointer increments it by the *n* times the size of the type – and not by *n* bytes
- Increment ++ and decrement -- increase/decrease by one element
- Be sure to only use valid pointers
  - initialize them
  - do not use them after the object has been deleted!
  - catastrophic errors otherwise

Arrays and pointers

- are very similar, but subtly different!

```
int array[5];

for (int i=0;i < 5; ++i)
  array[i]=i;

int* p = array; // same as &array[0]
for (int i=0;i < 5; ++i)
  cout << *p++;

delete[] p; // will crash
array=0; // will not compile
p=0; // is OK
```

- see these examples!

```
int* pointer=new int[5];

for (int i=0;i < 5; ++i)
  pointer[i]=i;

int* p = pointer;
for (int i=0;i < 5; ++i)
  cout << *p++;
```

- ```
  p=pointer;
  delete[] p; // is OK
  delete[] pointer; // crash
  delete[] p; // will crash
  p=0; // is OK
  pointer=0; // is OK
  ```
  ```

A look at memory: array example

- Array example

```
int array[5];

for (int i=0;i < 5; ++i)
  array[i]=i;

int* p = array; // same as &array[0]
for (int i=0;i < 5; ++i)
  cout << *p++;


delete[] p;  // will crash
array=0;  // will not compile
p=0;  // is OK
```

address	contents	name
0	0	a[0]
4	1	a[1]
8	2	a[2]
12	3	a[3]
16	4	a[4]
20	0	p
24		
28		

A look at memory: pointer example

- Array example

```
int* pointer=new int[5];

for (int i=0;i < 5; ++i)
  pointer[i]=i;

int* p = pointer;
for (int i=0;i < 5; ++i)
  cout << *p++;

delete[] pointer;  // is OK
delete[] pointer;  // crash
delete[] p;  // will crash
p=0;  // is OK
pointer=0;  // is OK
```

address	contents	name
0	12	pointer
4	12	p
8		
12	0	
16	1	
20	2	
24	3	
28	4	

References

- are aliases for other variables:

```
float very_long_variabe_name_for_number=0;

float& x=very_long_variabe_name_for_number;
  // x refers to the same memory location

x=5; // sets very_long_variabe_name_for_number to 5;

float y=2;
x=y; // sets very_long_variabe_name_for_number to 2;
     // does not set x to refer to y!
```

A more flexible program: function calls

```
#include <iostream>
using namespace std;

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

int main() {
  cout << "Enter a
  number:\n";
  float x;
  cin >> x;
  cout << x << " " <<
  square(x) << "\n";
  return 0;
}
```

- a function "square" is defined
  - return value is float
  - parameter x is float
- and used in the program

Function call syntax

- syntax:

returntype functionname
    (*parameters*)
{
  functionbody
}

- *returntype* is "void" if there is no return value:

```
void error(char[] msg) {
  cerr << msg << "\n" ;
}
```

- There are several kinds of parameters:
  - pass by value
  - pass by reference
  - pass by const reference
  - pass by pointer

- Advanced topics to be discussed later:
  - inline functions
  - default arguments
  - function overloading
  - template functions

Pass by value

- The variable in the function is a copy of the variable in the calling program:

```
void f(int x) {
  x++; // increments x but not the variable of the calling program
  cout << x;
}

int main() {
  int a=1;
  f(a);
  cout << a; // is still 1
}
```

- Copying of variables time consuming for large objects like matrices

Pass by reference

- The function parameter is an alias for the original variable:

```
void increment(int& n) {
  n++;
}

int main() {
  int x=1; increment(x); // x now 2
  increment(5); // will not compile since 5 is literal constant!
}
```

- avoids copying of large objects:
  - `vector eigenvalues(Matrix &A);`
- but allows unwanted modifications!
  - the matrix A might be changed by the call to eigenvalues!

Pass by const reference

- Problem:
  - `vector eigenvalues(Matrix& A); // allows modification of A`
  - `vector eigenvalues(Matrix A); // involves copying of A`
- how do we avoid copying and prohibit modification?
  - `vector eigenvalues (Matrix const &A);`
  - now a reference is passed -> no copying
  - the parameter is const -> cannot be modified

Pass by pointer

- Another method to pass an object without copying is to pass its address
- Used mostly in C
- `vector eigenvalues(Matrix *m);`
- disadvantages:
  - The parameter must always be dereferenced: `*m;`
  - In the function call the address has to be taken:

```
Matrix A;
cout << eigenvalues(&A);
```

- rarely needed in C++

A swap example

- Five examples for swapping number
  - `void swap1 (int a, int b) { int t=a; a=b; b=t; }`
  - `void swap2 (int& a, int& b) { int t=a; a=b; b=t;}`
  - `void swap3 (int const & a, int const & b)`
    `{ int t=a; a=b; b=t;}`
  - `void swap4 (int *a, int *b) { int *t=a; a=b; b=t;}`
  - `void swap5 (int* a, int* b) {int t=*a; *a=*b; *b=t;}`
- Which will compile?
- What is the effect of:
  - `int a=1; int b=2; swap1(a,b); cout << a << " " << b << "\n";`
  - `int a=1; int b=2; swap2(a,b); cout << a << " " << b << "\n";`
  - `int a=1; int b=2; swap3(a,b); cout << a << " " << b << "\n";`
  - `int a=1; int b=2; swap4(&a,&b); cout << a << " " << b << "\n";`
  - `int a=1; int b=2; swap5(&a,&b); cout << a << " " << b << "\n";`

Type casts: static_cast

- Variables can be converted (cast) from one type to another
- **static_cast** converts one type to another, using the best defined conversion, e.g.
  - `float y=3.f;`
  - `int x = static_cast<int>(y);`
  - replaces the C construct `int x= (int) y;`
- Can also be used converts one pointer type to another, useful for low-level programming, for example to look at representations of floating point numbers or check for endianness
  - `float y=3.f;`
  - `float *fp = &y;`
  - `int *ip = static_cast<int*>(fp)`
  - `std::cout << *ip;`

Type casts: const_cast

- const_cast can be used to remove const-ness from a variable
  - Example: need to pass a `double*` to a C-style function which does not change the value, but I only have a `const double*`

```
void legacy_c_function (double* d);

void foo(const double* d) {
   // remove the const
   double* nonconst_d = const_cast<double*>(d);
   // now call the function
   legacy_c_function(nonconst_d);
 }
```

  - Use it very sparingly. Usually the need for `const_cast` is a sign of bad software design
- Other casts to be discussed later:
  - `dynamic_cast`
  - `boost::lexical_cast`
  - `boost::numeric_cast`

Namespaces

- What if a `square` function is already defined elsewhere?
- C-style solution: give it a unique name; ugly and hard to type

```
float ETH_square(float);
```

- Elegant C++ solution: namespaces
  - Encapsulates all declarations in a modul, called "namespace", identified by a prefix
  - Example:

```
namespace ETH
{
  float square(float);
}
```

- Namespaces can be nested
- Can be accessed from outside as:
  - `ETH::square(5);`
  - `using ETH::square;` `square(5);`
  - `using namespace ETH;` `square(5);`
- Standard namespace is `std`
- For backward compatibility the standard headers ending in .h import `std` into the global namespace. E.g. the file "iostream.h" is:

```
#include <iostream>
using namespace std;
```

Default function arguments

- are sometimes useful

```
float root(float x, unsigned int n=2); // n-th root of x

int main()
{
  root(5,3); // cubic root of 5
  root(3,2); // square root of 3
  root(3); // also square root of 3
}
```

- the default value must be a constant!

```
unsigned int d=2;
float root(float x, unsigned int n=d); // not allowed!
```