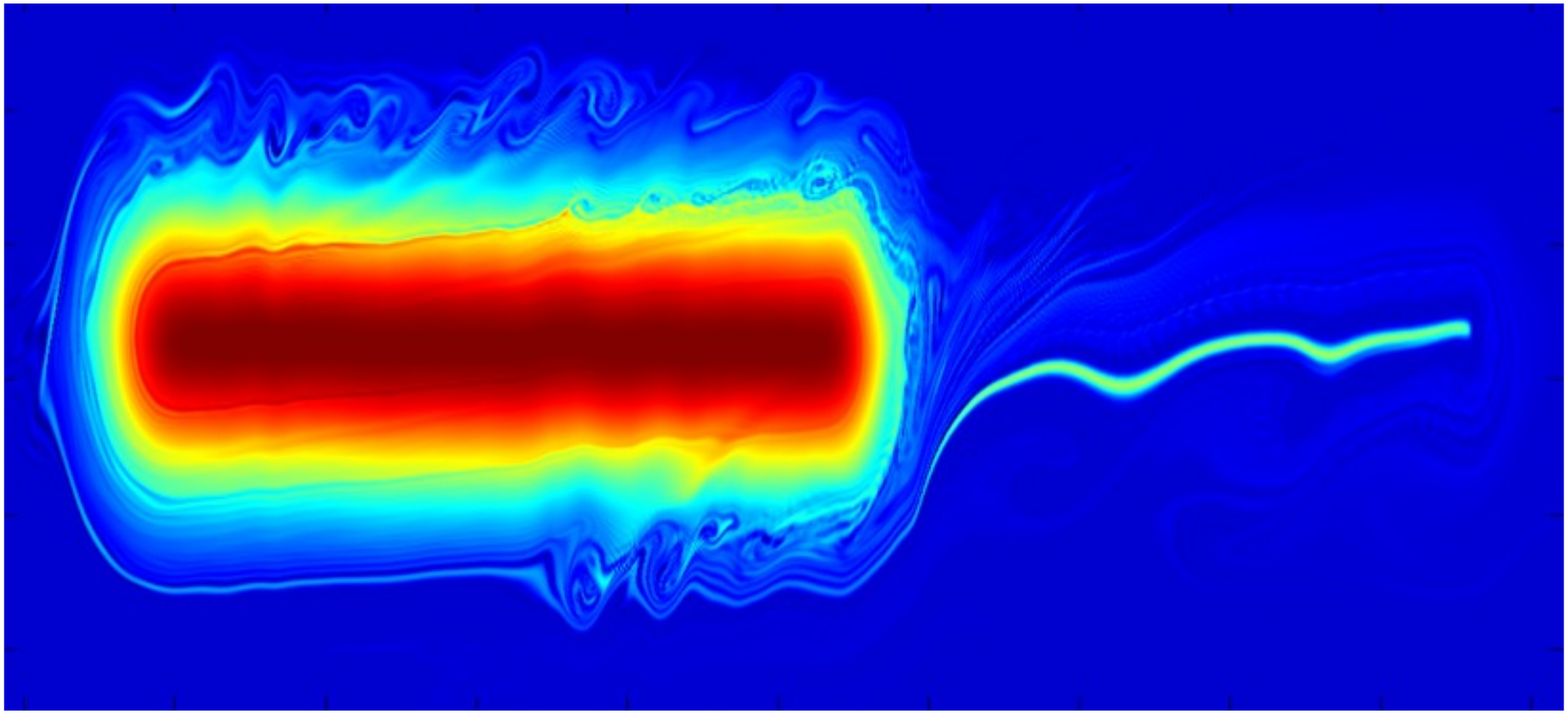
HEINRICH HEINE
UNIVERSITÄT DÜSSELDORF



Computational Physics 2015/2016
Chapter 1 - First steps in C++

Hello World

```
/*
 * hello.cxx
 *
 *  Created on: 09.04.2013
 *      Author: goetz
 */

// single line comment


#include <iostream>

int main(){

     std::cout << "Hello World" << std::endl;

     return 0;
}
```

```
/*
 * hello.cxx
 *
 *  Created on: 09.04.2013
 *      Author: goetz
 */

// single line comment


#include <iostream>

using namespace std;

int main(){

     cout << "Hello World" << endl;

     return 0;
}
```

Defining the *namespace (std* for standard*)*, we can leave away the *scope resolution* (std::) operator. Your typical program in the beginning will always have the line `using namespace std;`

C++ is a typed language

```
#include <iostream>

using namespace std;

int main(){

        a=5;           // Will not work, no type for variable a specified
    int b=4;           // This works, b will be of type int (=an integer number)
    float e = 2.714;   // Variable e is a floating point number with single precision and value 2.714
    double d = 2.714; // Variable d is a floating point number with double precision and value 2.714
    double g;          // Variable g is a double precision number, but has no value assigned yet.
                       // Warning: It will have a value, which is NOT necessarily zero!!

    cout << "b = " << b << endl;

    cout << "e = " << e << ", d = " << d << ", g = " << g << endl;

    return 0;
}
```

Each variable has to declared before using it.
A declaration has to specify the datatype. The declaration may also assign a value to the variable.

C++ - First steps

HEINRICH HEINE
UNIVERSITÄT DÜSSELDORF

C++ itself is a small language

alignas (since C++11)	enum	return
alignof (since C++11)	explicit	short
and	export	signed
and_eq	extern	sizeof
asm	false	static
auto(1)	float	static_assert(since C++11)
bitand	for	static_cast
bitor	friend	struct
bool	goto	switch
break	if	template
case	inline	this
catch	int	thread_local(since C++11)
char	long	throw
char16_t(since C++11)	mutable	true
char32_t(since C++11)	namespace	try
class	new	typedef
compl	noexcept(since C++11)	typeid
const	not	typename
constexpr(since C++11)	not_eq	union
const_cast	nullptr (since C++11)	unsigned
continue	operator	using(1)
decltype(since C++11)	or	virtual
default(1)	or_eq	void
delete(1)	private	volatile
do	protected	wchar_t
double	public	while
dynamic_cast	register	xor
else	reinterpret_cast	xor_eq

cppreference.com

What types are there in C++?

- C++ brings along several built in datatypes:
  - Datatypes for integers: bool, char, short, int, longt int
  - Datatypes for floating point numbers: float, double, long double
  - Complex numbers: complex<float>, complex<double>, complex<int> via the complex class of the Standard Template Library (STL)
  - Strings: Datatype string available via the string class of the STL
- There are many other C++ datatypes, mostly from the STL
- We can (and will) define our own datatypes (structs and classes)

Why does C++ require typing?

- The idea behind typing:
  - Knowing the datatype of a variable, the compiler knows which operations are permitted and which are not.
    Examples:
    - `int a=2; int b=3; int c; c=a+b;` This is fine.
    - `int a=2; double b=3.3; int c; c = a + b ;`
      This will be rejected by the compiler. We try to assign a double value to an integer variable. Obviously the result will be inaccurate.
      We can do this, but we will have to tell the compiler that we really want this!

- C++ performs tight type checking at compile time (and only then, contrary to e.g. Java).
- Only if all operations are defined and all datatypes are correct, the program will compile successfully. In this way the compiler makes sure, that the program will run.
- This does not mean the program is going to be correct... and there is a way to manipulate type-checking (using void-pointers, more on them later)

From source code to binary

- To obtain a running binary file, we have to compile our source code
- C++ sources usually have file extension: .cc , .C, .CC, .cxx, .cpp
- We will use the C++ compiler from GCC (GNU Compiler Collection) called g++
- Most simple compiler command:
  - g++ sourcefile.cxx -o prog1
  - This will compile sourcefile.cxx into a binary with the name prog1 (note, different from Windows, there is no file extension like .exe indicating that this is an executable file)
- Additional options might be:
  - g++ sourcefile.cxx -o prog1 -Wall -ansi -g
  - -Wall : Show all warnings, -ansi: reject usage of g++ extensions not matching ISO standard, -g: include debug information

A hand full of operators

- "*C++ Operators can manipulate, process and create objects*" is the most general statement you can make about operators. The full implications of this will become clear later.
- We keep it simple first: Operators for numerical datatypes.
  - For variables of types short, int, float, double, ... there are the obvious operators +, -, /, * and = , e.g. :
    double a=1; double b,c;
    b=a; c = a + 2*b;
  - Furthermore there are operators +=, -=, *=, /=, which manipulate a variable:
    d += 2; // i.e. d -> d + 2, equivalent result is obtained from d = d+2;
    c *=d; // i.e. c -> c*d, equivalent result would come from c = c*d;
    These operators are shortcuts and should be used wherever possible!
  - The comparison operator to check whether two numbers are equal is == (and not = !!)
  - For integer datatypes (short, int,...) there are additional operators ++ and --:
    int i=1; i++; // equiv. to i +=1; and i = i + 1; , but quicker than both

C++ - First steps

HEINRICH HEINE
UNIVERSITÄT DÜSSELDORF

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

int main(){
    int i,j=1; // i created, j created and set to 1 (only j, not i!)
    double ratio;

    j++; // after this j = 2
    i = 11;
    i-=j; // after this j=2, i=9

    int r = i/j; // Careful!
                 // Integer division, 9/2 -> 4, i.e. r = 4
    ratio = i/j; // Again integer division!! ration will have value 4,
                 // not 4.5, even though ratio is of type double

    ratio = 9/2; // Yields 4, since 9 and 2 are integer numbers
    ratio = 9/2.0; // This gives ratio = 4.5, since 2.0 is interpreted as
                   // floating point number

    ratio *= 2*(i+j); // i=2, j=9 -> ratio becomes 99

    cout << "ratio = " << ratio << ", r = " << r << endl;

    return 0;
}
```

HEINRICH HEINE
UNIVERSITÄT DÜSSELDORF

Expressions and conditional statements

- Expressions are instructions that compare two objects and return either true or false
- Typically we will mostly need: >, >= , <, <=, == (equal) , != (unequal), && (and), ! (not) and || (or)
- Conditional statements are `if-else`, `while`, `do-while`, for and `switch`

```
if (expression)
    statement
```

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

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

statement may either be a single line of code or a block of code enclosed by brackets { ... }:

```
if (a==b)
     cout << "A = B" << endl;
else
     cout << "A is not equal to B" << endl;
```

```
if (a>b){
     a -=b;
     a /=b;
}
else{
     a +=b;
     a *=b;
}
```

for - loops

```
int i;
for(i=1; i<10; i++){
    cout << i << endl;
}
```

A for-loop always has the structure

```
for(initialization; condition; step)
    statement
```

- Usually we define the counter-variable within the initialization of the loop.
  If the loop body just contains a single statement, we may leave away the brackets

```
for(int i=1; i<10; i++)
    cout << i << endl;
```

- Usually counters have the names i,j,k. Their values can be changed within the loop body, but great care needs to be taken!

```
for(int i=1; i<10; i++){
    i += 1;
    cout << i << endl;
}
```

- Counters don't have to be integers

```
double x=0, dx=0.1;
for(double x=0; x<10; x+=dx)
    cout << x << "\t " << sin(x) << endl;
```

for - loops

- We can also go backwards in a for-loop

```
for(int i=10; i>0; i--){
    cout << i << endl;
}
```

- With the break command, we can exit the for loop prematurely

```
int i,j= 10;
for(i=0; i<100; i++){
    if (i==j) break;
    cout << i << endl;
}
cout << "j is " << i << endl;
```

- Using the continue command, we can skip a part of the for loop

```
for(int i=0; i<10; i++){
    (something to be done for all i)
    if (i==j) continue;
    (something to be done just for i unequal to j )
}
```

Reading data from the console

```
// Example: Sum of first N integers,
// where N is read from console

#include <iostream>

using namespace std;

int main(){
    int N,sum=0;

    cout << "N = ";
    cin >> N; // Value of N is read from console

    cout << "N = " << N << endl;

    for(int i=1; i<=N; i++) sum+= i;

    cout << "sum = " << sum << endl;

    return 0;
}
```

The commands (actually they are objects) `cin` and `cout` come from the C++ extension `iostream`, hence we have to include this here

Includes

- The pre-processor command `#include <...>` allows access to all extensions of C++
- A few of them are:
  - iostream , for input and output from and to console
  - fstream , for file-IO
  - cmath , math functions (trigonometric functions, sqrt, pow, exp, ...)
  - complex , complex numbers
  - vector, vectors
  - string, strings
  - sstream, string streams, good for manipulating strings
- `#include <...>` commands tell the pre-processor to look for a *header-file* (common extension are .h, .hxx, .hpp, .H) in a specific directory (typically /usr/include/c++/...) and copy it's contents into the source file where the #include command stands. Only after this insertion the compiler will start its work.

Header file example - iostream

- The header-file contains information about the commands which are available and how they can be used (details later, when we talk about functions)

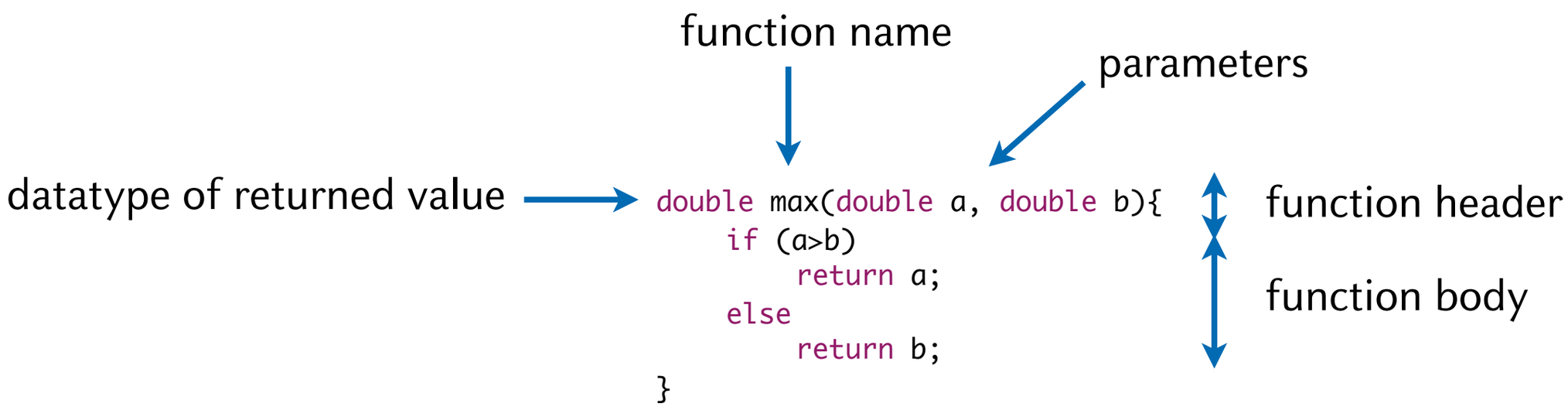
```
/** @file iostream
 *  This is a Standard C++ Library header.
 */

//
// ISO C++ 14882: 27.3  Standard iostream objects
//

#ifndef _GLIBCXX_IOSTREAM
#define _GLIBCXX_IOSTREAM 1

#pragma GCC system_header

#include <bits/c++config.h>
#include <ostream>
#include <istream>

_GLIBCXX_BEGIN_NAMESPACE(std)

  //@{
   extern istream cin;             ///< Linked to standard input
   extern ostream cout;            ///< Linked to standard output
   extern ostream cerr;            ///< Linked to standard error (unbuffered)
   extern ostream clog;            ///< Linked to standard error (buffered)

#ifdef _GLIBCXX_USE_WCHAR_T
   extern wistream wcin;           ///< Linked to standard input
   extern wostream wcout;          ///< Linked to standard output
   extern wostream wcerr;          ///< Linked to standard error (unbuffered)
   extern wostream wclog;          ///< Linked to standard error (buffered)
#endif
   //@}

   // For construction of filebuffers for cout, cin, cerr, clog et. al.
   static ios_base::Init __ioinit;

_GLIBCXX_END_NAMESPACE

#endif /* _GLIBCXX_IOSTREAM */
```

- You might see code with includes like this: `#include <math.h>`.
  These header files are for C, not C++. In most cases they will work, but it is in general safer to use the C++ version (i.e. `#include <cmath>`)

HEINRICH HEINE
UNIVERSITÄT DÜSSELDORF

Functions

- In C++ can structure our programs by putting some of our code into functions, which will then be called from the *main*-function or other functions
- In C++ a function is specified by its
  - name
  - parameter(s)
  - return value
- Each C++ function can only have one return value



Functions

```
#include <iostream>

using namespace std;

double max(double a, double b){
     if (a>b)
          return a;
     else
          return b;
}

int main(){
     double a=3;
     double b=5;
     double c=7;
     double m;

     m = max(a,b);
     cout << "Max{a,b} = " << m << endl;
     m = max(c,m);
     cout << "Max{a,b,c} = " << m << endl;


     return 0;
}
```

- Code for functions may be inserted above the main-function
- Parameters are identified by position and not by name
  - Variables for parameters may have the same name inside the function, but there is no need for this (although it can make things easier to read)
- A function is only uniquely identified by the name and the parameters
  - If there were a second function with the name `max` but with other parameters it would be a different function!
- Having several functions with the same name, but different parameter types is known as *function overloading*

HEINRICH HEINE
UNIVERSITÄT DÜSSELDORF

Functions

- Running this codes results in the output:

```
int version...
double version...
int version...
```

- Compiling with the line `md = max(h,a);` included the compiler tells us:

```
../BasicFunctions.cxx: In function 'int main()':
../BasicFunctions.cxx:30: error: call of overloaded 'max(int&, double&)' is ambiguous
../BasicFunctions.cxx:5: note: candidates are: double max(double, double)
../BasicFunctions.cxx:13: note:                 int max(int, int)
```

- Explicit casting of the integer h into a double works:
  `md = max( double(h) ,a);`
  - The double version will be used now

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

double max(double m, double n){
     cout << "double version..." << endl;
     if (m>n)
          return m;
     else
          return n;
}

int max(int m,int n){
     cout << "int version..." << endl;
     if (m>n)
          return m;
     else
          return n;
}

int main(){
     double a=1, b=5;
     int h=1, k=2;
     double md;
     int mi;

     mi = max(h,k);
     md = max(a,b);
     md = max(h,k);
//   md = max(h,a);

     return 0;
}
```

Functions

- We distinguish between the *declaration* and the *definition* of the function
- The *declaration* (also known as *prototype*) contains all information that somebody who wants to use the function needs
  - Name, parameters, return value
- The *definition* of a function contains all the code that the compiler needs to know if he finds that somebody is calling this function
- Declaration and definition may be the same piece of code, but often are not
- Separation between declaration and definition allows to structure codes into multiple files or even hide definitions completely from users (libraries)

Functions

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

// Declaration and Definition
double max(double m, double n){
    if (m>n) return m;
    else return n;
}

int main(){
    double a=1, b=5;
    double m = max(a,b);
    return 0;
}
```

splitting declaration and definition →

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

// Declaration
double max(double m, double n);

int main(){
    double a=1, b=5;
    double m = max(a,b);
    return 0;
}

//Definition
double max(double m, double n){
    if (m>n)
        return m;
    else
        return n;
}
```

- Why split declaration from definition?
  - Definitions may go into totally different files
  - Declaration (Prototypes) can also be in a different file (a header file)
  - Source code becomes more structured and readable

much more on functions later...

HEINRICH HEINE
UNIVERSITÄT DÜSSELDORF

Statically allocated arrays

- Arrays are structures which allow us to store multiple entries of the *same* datatype (think of it as a vector or a list of data)
- We distinguish between two types of arrays:
  - statically allocated (the length of the array is known at compile time)
  - dynamically allocated (the length will only be known at runtime)
- Statically allocated arrays are defined in the following way:

```
double p[5];
int k[10];
```

- To access the entries of the array, we also use the bracket operator []:

```
int j = k[0];   // First entry has index 0!
double d = p[4];
p[3] = 2.5;
```

- C++ will ***not*** check for out-of-bounds errors! Accessing memory which does not belong to your program is possible! In particular you can write into memory of other variables or programs!

```
p[10] = 2; // Will produce no error and no warning!
```

Statically allocated arrays

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

using namespace std;

int main(){
    double p[5];

    p[1] = 2.3;

    for(int i=0; i<5; i++)
        cout << "p[" << i << "] =" << p[i] << endl;

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

    for(int i=0; i<5; i++)
            cout << "p[" << i << "] =" << p[i] << endl;


    p[10] = 2.2;

    return 0;
}
```

much more on arrays later...

- In the program:
  - we reserve memory for 5 doubles (i.e. 5 x 64 bit = 320 bit)
  - write 2.3 into the second entry of the array
  - print all double values contained in the array (all except 2nd entry are not initialized ➠ they may be 0, but don't have to)
  - write sqrt(i) into the i-th position of the array
  - print everything again
  - and then write to memory which is not ours!! (potentially we are crashing the program here!)

Output:

```
p[0] =0
p[1] =2.3
p[2] =0
p[3] =0
p[4] =0
p[0] =0
p[1] =1
p[2] =1.41421
p[3] =1.73205
p[4] =2
```

Complex Numbers

- The template class complex allows us to represent complex numbers and to do some math

```
#include <iostream>
#include <complex>

using namespace std;

int main(){

    complex<double> c,d,expc;

    // cf = 1.0 + 0.0i
    complex<float> cf = complex<float>(1.0, 0.0);

    // c = 0 + i
    c = complex<double>(0.0, 1.0);
    // d = 1.2 + i 0.5
    d = complex<double>(1.2, 0.5);

    expc = exp(c);

    cout << "c = " << c << ",\t d = " << d << endl;
    cout << "exp(x) = " << expc << endl;
    cout << "c*d =" << c*d << endl;
    cout << "|c*d| = " << norm(c*d) << endl;
    cout << "Re(c*d) = " << real(c*d) << ", \t Im(c*d) = " << imag(c*d) << endl;

    return 0;
}
```

Output:

```
c = (0,1),          d = (1.2,0.5)
exp(x) = (0.540302,0.841471)
c*d =(-0.5,1.2)
|c*d| = 1.69
Re(c*d) = -0.5,             Im(c*d) = 1.2
```