**C++ QUICK Reference**

**DATATYPES**

**INTEGRAL TYPES**

<table>
<thead>
<tr>
<th>Type</th>
<th>Size</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>1 B</td>
<td>-128 to 127</td>
</tr>
<tr>
<td>short</td>
<td>2 B</td>
<td>-32768 to 32767</td>
</tr>
<tr>
<td>int</td>
<td>4 B</td>
<td>Usually as below</td>
</tr>
<tr>
<td>long int</td>
<td>8 B</td>
<td>-2'147'483'648 to ...</td>
</tr>
<tr>
<td>float</td>
<td>4 B</td>
<td>1.2e-38 to 3.4e38</td>
</tr>
<tr>
<td>double</td>
<td>8 B</td>
<td>2.2e-308 to 1.8e308</td>
</tr>
</tbody>
</table>

**INLINE TYPE SPECIFICATION**

2L, 2U, 2UL, 2.0 // not case sensitive
2.0f // the f makes it a float instead of double
2.0 // double
2f // error
2.0d // error
2.11 // long float = double

**VARIABLE/FIELD DECLARATION**

(storage-class-spec)(co-qualifier)(type) name;

(char STORAGE CLASS SPEC volatile)
volatile register:
registers, compiler, object is frequently used

**DECLARATION FLAGS**

volatile: usually prevents optimization
register: tells compiler, this object is frequently used

**SIMPLE PROGRAM STRUCTURE**

```cpp
int main(int argc, char **argv) {
    // code;
    return 0;
} // array: array of c-strings
```

**SCOPE AND STATIC VARIABLES**

The scope is marked by curly brackets: {} ...

**VARIABLE/FIELD NAMES**

Variable names cannot start with: any keyword mentioned below or a digit.

- Keywords:
  - and, and_eq, as, auto, bitand, bitor, bool,
  - break, case, class, const, continue, default, do,
  - double, dynamic_cast, else, enum, explicit,
  - export, extern, float, for, friend,
  - goto, if, inline, int, long, mutable,
  - namespace, new, not, not_eq, noexcept,
  - operator, or, or_eq, private, protected,
  - public, register, reinterpret_cast, return,
  - short, sizeof, static_cast, struct,
  - switch, template, this, throw, true,
  - typeid, union, using, virtual, void, volatile,
  - wchar_t, while, xor, xor_eq

**OPERAND PRECEDENCE & ASSOCIATIVITY**

int a, b = 3, c; // a = c = undeclared, b = 3
int d(5);

**INCREMENT & DECREMENT OPERATOR**

Increment & decrement can be omitted

new_type expr2 = (new_type)

**TYPE CASTING**

new_type expr2 = (new_type) expr1;

```cpp
//function-style cast or constructor & assignment
float a = 7/2; //a=3.5
```

**BRANCHING**

```cpp
if (...) else if (...) else...
```

**SWITCH**

```cpp
switch (i) {
    case 5: cout << "foo";
    case 6: cout << "bar"; break;
    case 7: cout << "pong"; break;
    default: cout << "wtf"; break;
}
```

**FOR**

```cpp
for(int i = 0; i < 12; i++) { foo();
```

**WHILE**

```cpp
while(condition) {
    // run the loop while condition is true
    foo(a);
    if(condition2) break; // leaves the loop early
    if(condition3) continue; // goes to the start
}
```

**DO-WHILE**

```cpp
do {
    foo();
} while (condition); // runs at least once
```

**POINTERS & REFERENCES**

```cpp
int a = 42; int *p = &a;
```

**DECLARATION**

```cpp
new_type expr2 = (new_type) expr1;
```
```cpp
int a = 1, b = 2; int* c = &a; // what you want const int* d = &a; //correct
d = 5; // error *d = &d; //correct
const int* e = &a; // like an array *e = 5; // correct *e = &b; // error
const int f = 7; const int* g = &f; // ok
int* h = &f; // error
int i = (int*)0x33; // points to mem x (33), cast necessary
int j, k, l; // only if i is int, k and l int!

DYNAMIC MEMORY (NEW, DELETE)
For c-type dynamic memory see <stdio.h>. C++ only:
new allocates memory, delete releases it again
int* a = new int; int* b = new int[5]; // both ok
int* c = new int[6]; //int* d = new int[3][2]; // *c, *d, error
int* e = new int[3][2]; //int* f = new int[3][2][3]; // both error

Try: new = delete and new ...
// = delete[] ... // Usually
Exception on lack of memory:
int* dp = new (nothrow) int[100]; // no exception, but may return null-pointer

REFERENCE OPERATOR (*)
andy = 25; fred = andy; ted = &andy; 

1755 1775 1777
| 25 |
| 25 |

DEREFERENCE OPERATOR (**)
andy = 25; ted = &andy; beth = **ted; 

1775 1777 | 25 |

FUNCTION POINTERS
int add(int a, int b) { return a + b; } int sub(int b, int c) { return a - b; }
int op(int x, int y, int (*func)(int, int)){
int g; g = (*func)(x, y); return g; }
void foo(){int m, n; int * (minus) (int) = sub; m = op(7, 5, add); n = op(20, n, minus); }

OVERLOADING
Overloading means declaring multiple functions with the same name. They need different signatures, so that the compiler (and you) know which one to take.
Overloading functions with optional args is especially risky.

PROTOTYPES/SIGNATURES
int foo(float); // or int foo(float a); //error
Protopotypes contain name and signature of a function.
Usually there is a header file defines with prototypes of all functions.
Prototypes are not necessary, if you define a function before it is used. There can never be multiple definitions for one prototype.

MEMBER FUNCTIONS
[virtual] [static] [inline] [const] return-type
name(arg-type arg-name, arg-type arg2)
[const] arg2-name[def_value]) [const] [volatile]
{ return return-value; } virtual:
[...] dynamic binding static:
Function can be called without instantiated
class and can only access static members.
A static function cannot be const.
inline:
Asks the compiler to copy the code, where the
function is called rather than calling it. This
is faster, but results in a larger binary. No
reassignment, the returned reference cannot be changed
const(*):
All optional params have to be at the end.
const(
): //correct
volatile:
Prevents optimization. Member functions of
volatile objects should be declared volatile

FUNCTIONS
int add(int a, int b) { return a + b; } int sub(int b, int c) { return a - b; } int op(int x, int y, int (*func)(int, int)){
int g; g = (*func)(x, y); return g; }
void foo(){int m, n; int * (minus) (int) = sub; m = op(7, 5, add); n = op(20, n, minus); }

ARGUMENTS
• call by value (copy the argument)
void foo(int arg) { // The value of arg is copied
//even classes and structs are copied!
int a = 5; foo(a); // a will certainly remain 5
• call by reference
void foo(int* arg) { // The address of arg is copied
int a = 5; foo(a); // a can change
• implicit call by reference (call by value of the pointer)
void foo(int* arg) { // The address of g is copied
int a = 5; foo(a); // a’s value can change
int b = 5; foo(&b); // same as above
The same applies to the return value: copied by default
Never return a reference to a local variable!

DATA TYPES
int 64; // error: refs require init
int b = k; // correct declaration
b = m; // error: refs cannot change
b = y; // correct value assignment
 operator(*.a) is equivalent to *a=b. Where b can be field, function,...
a=b, m = b; // correct assignment of b’s value

POINTER-ARITHMETICS
int b[100]; int* p; p = b; // implicit array-to-pointer conversion
p = &b[0]; // same thing as above

References
```
```
**ABSTRACT CLASSES**

A class is abstract, if at least one of its functions is purely virtual.

**Declaration:**

```
virtual double foo() = 0;
```

**ACCESS MODIFIERS**

Modifiers are only allowed inside classes.

- `public`: Accessible by everyone
- `protected`: Only accessible within the class
- `private`: Only accessible within and from inheriting class

**STATIC & CONST VARIABLES**

`static` variables are only destructed and freed on program termination. Even if they are not accessible anymore, they stay in memory, so you can access them again later and they still have the same value.

`const` variables on class level have the same value among all instances of a class. The cannot be initialized inside the class definition, only inside the `.cpp`-file like that:

```
int ClassName::staticVariableName = 5;  // cannot be further inherited.
```

**OVERRIDING OPERATORS & TYPE CASTS**

```
operator int() const { return 4; }
```

**INHERITANCE**

In C++, `virtual`-relation a e.g. banana is a fruit.

- `public` ("is-a" relation): The child-class can be casted to its parent-class.
- `protected`: public -> protected, protected -> private (can not be inherited)
- `private` ("has-a" relation)
  - `public` -> private, protected -> private, inaccessible

The `protected` inherited functions can be further inherited. Not castable.

- `private` ("has-a" relation)
  - `public` -> private, protected -> private, inaccessible

The `inherited` function can NOT be further inherited. Not castable.

**NAMESPACE**

```
namespace foo {  // only known inside this namespace
  const int a = 2;
  namespace sub {  // only known inside this namespace
    int bar() { return 2; }
  }
}
```

```
using namespace foo;  // enables access to the namespace without having to always refer to it (foo::bar())
```

**POLYMORPHISM**

```
class A {  
  int ii;  
};
class B {  
  char* ii;  
};
class C : A, B {  
  int jj;  
};
```

```
// Some ambiguities (note C++4.2)  
C* pc; pc = &ii;  
// error: A::ii or B::ii;  
```

```
// C++11  
pc->A::ii;  
// C++14  
pc->B::ii;  
```

```
// the same is valid for functions.  
D* d;  
// D is a container  
// access through explicit qualification  
void C::f();  
int k = A::l1;:  
```

```
// D Dynamic Casting  
C* pc = new C;  
L* pl = pc;  
// error: ambiguous  
pl = (L*) pc;  
// error: ambiguous
```

**TEMPLATES**

**FUNCTION TEMPLATES**

```
void Swap(Typ e, Typ b) {  
  Typ temp = e;  
  e = b;  
  b = temp;  
}
```

```
int x, y;  
Swap(x, y);  
```

**LIBRARIES**

**FILE STREAMS**

```
ifstream file1, file2;  
file1 >> a;  
file2 >> b;  
```

**C++ STRING CLASS & FUNCTIONS**

```
string s;  
cout << s;  
```

```
string str = "Hello World!";  
cout << str;  
```

```
// print 5 digits after point of float/dbl  
cout.precision(5);  
```

**METADATA**

```
// get rid of an int from console into 1  

control characters:  
<table>
<thead>
<tr>
<th>\s</th>
<th>\f</th>
<th>\r</th>
<th>\t</th>
<th>\n</th>
<th>\v</th>
<th>\a</th>
<th>\b</th>
<th>\e</th>
<th>\m</th>
<th>\h</th>
</tr>
</thead>
<tbody>
<tr>
<td>\b</td>
<td>\f</td>
<td>\r</td>
<td>\t</td>
<td>\n</td>
<td>\v</td>
<td>\a</td>
<td>\b</td>
<td>\e</td>
<td>\m</td>
<td>\h</td>
</tr>
<tr>
<td>\s</td>
<td>\f</td>
<td>\r</td>
<td>\t</td>
<td>\n</td>
<td>\v</td>
<td>\a</td>
<td>\b</td>
<td>\e</td>
<td>\m</td>
<td>\h</td>
</tr>
<tr>
<td>\s</td>
<td>\f</td>
<td>\r</td>
<td>\t</td>
<td>\n</td>
<td>\v</td>
<td>\a</td>
<td>\b</td>
<td>\e</td>
<td>\m</td>
<td>\h</td>
</tr>
</tbody>
</table>
```

**EXCEPTIONS**

**CHARACTER FUNCTIONS**

```
char toupper(char);  
```

```
// the arg has to be an unsigned char or an EOF  
```

**DYNAMIC BINDING**

Static binding is just as the keyword virtual is used by default and a little bit more performant

- function cannot be overwritten (but still overloaded)
- dynamic binding is only specified by the keyword virtual
- can be overwritten from an inheriting class.

```
class A {  
  virtual void f() {  
  }
};  
```

```
class C : A {  
  void f() {  
  }
};  
```

```
A* pa = new C;  
```

```
// invokes C::f();  
```

```
//print 5 digits after point of float/dbl cout.precision(5);  
```

```
// read an int from console into 1  
```

```
ifstream file1, file2;  
file1 >> a;  
file2 >> b;  
```

```
//print 5 digits after point of float/dbl cout.precision(5);  
```

```
int f;  
```

```
// control characters:  

control characters:  
<table>
<thead>
<tr>
<th>\s</th>
<th>\f</th>
<th>\r</th>
<th>\t</th>
<th>\n</th>
<th>\v</th>
<th>\a</th>
<th>\b</th>
<th>\e</th>
<th>\m</th>
<th>\h</th>
</tr>
</thead>
<tbody>
<tr>
<td>\b</td>
<td>\f</td>
<td>\r</td>
<td>\t</td>
<td>\n</td>
<td>\v</td>
<td>\a</td>
<td>\b</td>
<td>\e</td>
<td>\m</td>
<td>\h</td>
</tr>
<tr>
<td>\s</td>
<td>\f</td>
<td>\r</td>
<td>\t</td>
<td>\n</td>
<td>\v</td>
<td>\a</td>
<td>\b</td>
<td>\e</td>
<td>\m</td>
<td>\h</td>
</tr>
<tr>
<td>\s</td>
<td>\f</td>
<td>\r</td>
<td>\t</td>
<td>\n</td>
<td>\v</td>
<td>\a</td>
<td>\b</td>
<td>\e</td>
<td>\m</td>
<td>\h</td>
</tr>
</tbody>
</table>
```

**FILE STREAMS**

```
#include <fstream>  
```

```
ifstream file1("foo.txt");  
```

```
// fileloc (\c)  
if (file1.is_open()) {  
  file1 << a;  
}  
```

**C++ STRING CLASS & FUNCTIONS**

```
std::string str = "Hello World!";  
cout << str;  
```

```
// print 5 digits after point of float/dbl  
cout.precision(5);  
```

```
int f;  
```

```
// read an int from console into 1  
```

```
ifstream file1, file2;  
file1 >> a;  
file2 >> b;  
```

```
//print 5 digits after point of float/dbl cout.precision(5);  
```

```
int f;  
```

```
// control characters:  

control characters:  
<table>
<thead>
<tr>
<th>\s</th>
<th>\f</th>
<th>\r</th>
<th>\t</th>
<th>\n</th>
<th>\v</th>
<th>\a</th>
<th>\b</th>
<th>\e</th>
<th>\m</th>
<th>\h</th>
</tr>
</thead>
<tbody>
<tr>
<td>\b</td>
<td>\f</td>
<td>\r</td>
<td>\t</td>
<td>\n</td>
<td>\v</td>
<td>\a</td>
<td>\b</td>
<td>\e</td>
<td>\m</td>
<td>\h</td>
</tr>
<tr>
<td>\s</td>
<td>\f</td>
<td>\r</td>
<td>\t</td>
<td>\n</td>
<td>\v</td>
<td>\a</td>
<td>\b</td>
<td>\e</td>
<td>\m</td>
<td>\h</td>
</tr>
<tr>
<td>\s</td>
<td>\f</td>
<td>\r</td>
<td>\t</td>
<td>\n</td>
<td>\v</td>
<td>\a</td>
<td>\b</td>
<td>\e</td>
<td>\m</td>
<td>\h</td>
</tr>
</tbody>
</table>
```

**C++ STRING CLASS & FUNCTIONS**

```
typedef basic_string<char> string;  
```

```
typedef basic_string<char> tstring;  
```

```
// below 'string' means any of the above  
```
istream std::getline ( istream &str, string &str, char delim='\n' );
const char *c_str() { return c_string->c_string; }
size_t size() = s.length();
char c = s[2];
string s, s.substr( size_t pos, size_t len);
size_t idx = s.find( string str );
size_t s = s.insert( size_t pos, string str, int x = s.compare( string str ));
// If equal

COMMON CONCEPTS

RANDOM NUMBER GENERATION

#include <ctime>
#include <cstdlib>
#include <iostream>

US-ASCII TABLE

LEFT TO RIGHT:  C  D  E  F  G  H  I  J

<table>
<thead>
<tr>
<th>Case</th>
<th>Average</th>
<th>Worst-case</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

NAME: best-case  avg-case  worst-case

COMPLEXITY OF BASIC ELEMENTS

simple instruction: \( O(1) \)
sequence of \( O(n) \): \( O(n) \)
log(n) fun.: \( O(\log(n)) \)

NOTATIONS OVERVIEW

\( O(g(n)) \)  \( o(g(n)) \leq g(n) \), for some k
\( \Omega(g(n)) \)  \( \omega(g(n)) \geq g(n) \), for every c

BUILD-PROCESS & COMPILER

G++ COMPILER FLAGS

- \(--ansi --pedantic \)--Wall -O mo myprog myprog.cpp
- \(--std=c++11\)
- \(--no-exceptions\)
- \(--pedantic\)

MAKEFILES

- \$ Simple Makefile for a small project
- \$COMPLEXSTACK: clean
- \$CPFLAGS = -Wall --pedantic --ansi -LFLAGS = -Wall
- \$OBJ = -main.o -Stack.o Complex.o
- \$OBJ = -main.o -Stack.o Complex.o
- \$OBJ = -main.o -Stack.o Complex.o

IMPROVEMENTS, CHECKLIST

COMPLETE "DON'TS"

- Assigning values to a function, especially when returning a reference:
- int foo(int arg) { return return arg; }
- Don't use a global function:
- \#define \#include
- Don't include libraries:
- \#include libraries
- Semicolons:
- Case sensitivity:
- Are functions, variables and classes really defined on their first usage?
- Headers included?
- Is the size of arrays everywhere defined with const variables?
- In C++ it's NULL and not null:
- Use a macro with an inline function:
- All purely virtual functions defined?
- Semicolon after class, struct, enum... definition:
- Circular dependency without pointers:
- "*" of const:
- Provide only const pointer to only const fields:
- Is an array of const*:
- Pass to a const field, the function's arg has to be:
- Are all modifiers correct? (e.g. const, destructor): public:
- Destructor specified dealloc memory:
- Constructors all correct? (virtual, if class inherited)
- No modifiers outside of classes:
- No in-class initialization:
- Constructor (e.g. constructor):
- "error"
- in C++ modifiers need a const:
- A kflf/*error*/ A kflf/*ok*/ A k = A();
- Just for the beauty: catch eventual bad_alloc exceptions:
- Watch out for ambiguous function calls (unsigned vs. char):
- Watch out for ambigu function calls (float vs. double, float10):
- String length: space for \$reserved:
- unsigned char used for characters (not signed?)
- Every pointer correctly used? \(.\). vs. \(->\)
- Never returned a reference to a local variable?

IDEAS

- class diagram
- program flow diagram
- memory diagram
- Recursion useful?

POSSIBLE MISTAKES / CHECKLIST

- vs.-
- using namespace
- \#include libraries
- Semicolons:
- Case sensitivity:
- Are functions, variables and classes really defined on their first usage?
- Headers included?
- Is the size of arrays everywhere defined with const variables?
- In C++ it's NULL and not null:
- Use a macro with an inline function:
- All purely virtual functions defined?
- Semicolon after class, struct, enum... definition:
- Circular dependency without pointers:
- "*" of const:
- Provide only const pointer to only const fields:
- Is an array of const*:
- Pass to a const field, the function's arg has to be:
TODO
Stack
Queue
Sorting: Bubble sort
Filter folding
File Paging
Serialization
Binary tree
long numbers
Initializing a multidimensional array

JAVA
BASIC DIFFERENCES TO C++

COMPILER
javac myProgramm.java //to compile
java myprogramm //execute
javap –c xx.java //generates bytecode

DATATYPES
Integer
byte(8 bits)
short (16 bits)
int (32 bits)

NAMES
variables, methods: small
classes: capital
constants: all capitals

INSTANCEOF
Hund h;
Tier fiffy;
if (fiifi instanceof Hund) { f = (Hund)fiify; }

ARRAYS
int[] x = new int[7];
x[3] = 5;
int[] y = x;
y[3] = 9; //x[3] is now 9

I/O
string System.in.readline();
char System.in.read();
void System.out.println(…);
void System.exit(int);

CONSTANTS
static final type NAME = value;

INHERITANCE
class A extends B { ...
Multiple inheritance is not possible!

INTERFACES
interface Name1 { method_declarations; }
usage:
class Name2 implements Name1 {
method_implementation; }

GENERICS
public class Main<T extends java.lang.Exception> {
...

APPLETS
MULTITHREADING

EXCEPTIONS

BITSHIFT
a >>> b
signed rightshift, shifts the sign in
a = floor(a/2^b)
a >>>> b
unsigned rightshift, shifts a 0 in

STANDARD LIBRARY

STRINGS

PACKAGES
Define:
package test;

IMPORT:
import java.util.Random; //single class
import java.util.*; //whole package

FUNCTIONS
Arguments are all passed by reference.
Immutable objects can’t be edited of course

REFERENCES
Object a = new Object();
Always references except for:
byte, short, int,
long, float, double, char, boolean.
== just compares the addresses!

INHERITANCE
class A extends B {
...
}

INTERFACE
interface Name1 { method_declarations; }

CLASS
public class Main {
...

IMPORT
package test;
import java.util.Random;
import java.util.*;
ALGORITHMS & DATA STRUCT.

EUGYPTIAN MULTIPLICATION

Conceptually, a merge sort works as follows:
1. If the list is of length 0 or 1, then it is already sorted.
2. Divide the unsorted list into two sublists of about half the size.
3. Sort each sublist recursively by re-applying merge sort.
4. Merge the two sublists back into one sorted list.

Merge sort incorporates two main ideas to improve its runtime:
1. A small list will take fewer steps to sort than a large list.
2. Fewer steps are required to construct a sorted list from two sorted lists than two unsorted lists. For example, you only have to traverse each list once if they’re already sorted (see the merge function below for an example implementation).

HEAPIFICATION - Insertion sort is not primitive recursive (loop-calculable) but calculable in finite time and while calculable. The time required cannot be estimated.

ACKERMANN FUNCTION

Not primitive recursive (loop-calculable) but calculable in finite time and while calculable. The time required cannot be estimated. Application: benchmarking

FUNCTIONAL PROGRAMMING

PROOFING IDEAS

INVARIANTS: Invarianten definieren, z.B. dass immer c = a*b+z ausser ganz kurzfristig. Bsp. a verdoppeln, b halbieren, so dass (das evtl. nicht als Variable existierende) c konstant bleibt.

DATA STRUCTURES

WURZELBÄUME

Ebener Wurzelbaum: Linear geordnete Wurzelbäume E1 und E2 sind isomorph als Bäume und Wurzelbäume, nicht aber als ebene Wurzelbäume: 2 liegt links von 3, aber c liegt rechts von b.

OTHERS TREES

OTHER BINARY TREES

OPERATORBÄUME

Mögliche Lösung: Stack verwenden

ANDERE IDEE: Klammern einfügen, bei schliessender Klammer Operation ausführen. Auch hier gibt’s infix, prefix, postfix.

SYNTAXDIAGRAMS

SYNTAX

PROG RAMMING

GAME THEORY

MINIMAX

NEGAMAX

GAME TREES

MINIMAX

NEGAMAX

CORRECTNESS, ROBUST PROGRAMMING

INDUCTION

p. 16, 17

OF THE PROGRAM
IDEAS & TROUBLESHOOTING

IDEAS
- even or odd numbers: bool even = (a % 2 == 0 ? true : false);
- left resp. right-shift to multiply resp. divide by 2^n?
- divide and conquer
- recursion

CONCEPTS (IF YOU’RE STUCK)

POSSIBLE MISTAKES
Made sure it cannot come an infinite loop (break-condition)?

CONVERTING RECURSIVE TO FLAT AND BACK

TODO
- touring maschine
- sizeof zur Best. der Arraygrösse in c++?
- bitweiser Vergleich zweier Klassen-Instanzen
- typeof
```c++
//find the smallest and swap the first and smallest element
void selectionSort(int array[], int SIZE) {
    for (int pass = 0; pass < SIZE; pass++) {
        int potentialSmallest = pass; // assume this is smallest
        for (int i = pass + 1; i < SIZE; i++) { // find if a smaller one exists
            if (array[i] < array[potentialSmallest]) {
                potentialSmallest = i;
            }
        }
        if (potentialSmallest != pass) {
            int temp = array[pass]; //swap
            array[pass] = array[potentialSmallest];
            array[potentialSmallest] = temp;
        }
    }
}

void bubbleSort(int array[], int SIZE) {
    for (int pass = 0; pass < SIZE; pass++) {
        for (int i = pass + 1; i < SIZE; i++) { //else place them below in the list
            if (array[i] < array[i - 1]) {
                temp = array[i];
                array[i] = array[i - 1];
                array[i - 1] = temp;
            }
        }
    }
}

int partition(int array[], int low, int high) { //Takes an element and compares it to all other elements in a partition
    int pivot = array[high];
    int i = low - 1;
    for (int j = low; j <= high - 1; j++) { //shift the bigger elements upwards
        if (array[j] <= pivot) {
            i++;
            int temp = array[i];
            array[i] = array[j];
            array[j] = temp;
        }
    }
    int temp = array[i + 1];
    array[i + 1] = array[high];
    array[high] = temp;
    return i;
}

void merge(int low, int mid, int high) { //merges two sorted sections
    int h1, i, j; //index about which the array is partitioned
    int hi = mid;
    int lo = low;
    int mid1;
    if (mid < high) { //if the other elements are larger, then place them above in the list
        temp = array[hi - 1];
    }
    else { //else the end of the high array has been reached
        temp = array[mid];
    }
    for (int i = low, j = mid + 1; i <= mid; i++) { //swap
        if (array[i] > temp) {
            array[i] = array[j];
            array[j] = temp;
        }
    }
    for (int i = mid, j = high; i >= mid; i--) { //pushes the larger forwards
        if (array[i] > temp) {
            array[i] = array[j];
            array[j] = temp;
        }
    }
}

int quicksort(int num, int top, int bottom) { //recursively sorts the list
    if (bottom - top < 1) return top;
    int middle = partition(num, top, bottom); //sort first section
    quicksort(num, top, middle); //sort second section
    return middle;
}

int main() { //if the end of the low array has been reached copy the rest over
    int k;
    for (k = 0; k <= high; k++)
        b[k] = array[k];
    for (k = 0; k <= mid; k++)
        b[k] = array[k];
    while (k < low)
        b[k] = array[k];
    for (j = 0; j < num; j++)
        b[j] = num[j];
    return 0;
}
```

# Bestimmt die Länge des Filters

```cpp
def get_filter_size(const char* filter):
    ifstream fin(filter);
    if(!fin.is_open()) {
        cerr << "Failed to open " << filter; return 0;
    }
    int len = 0;
    double d;
    while(fin >> d) len++;
    fin.close();
    return len;
```

## Funktion `discretefolding()`

```cpp
def discretefolding(double d[], int n, double f[], int filtersize):
    double dfolded = 0;
    for(int k = 0; k < filtersize; k++) {
        if((n-k+filtersize/2 < 0) || (n-k+filtersize/2 > filtersize))
            dfolded += f[k]*d[n-k+filtersize/2];
    }
    return dfolded;
```

## Funktion `filter()`

```cpp
def filter(char datafile[], char filterfile[], int filtersize, char outputfile[]):
    double* filterdata = new double[filtersize];
    double* bufferdata = new double[filtersize];
    ofstream fout(outputfile);
    if(!fout.is_open()) {
        cerr << "Failed to open output-stream!"; return false;
    }
    //Eingabestream für Daten erstellen
    ifstream findata(datafile);
    if(!findata.is_open()) {
        cerr << "Failed to open Data-Stream!"; return false;
    }
    //Garbage collector
    char c[200];
    findata.getline(c,200);
    //Filterdaten einlesen bis Buffer voll
    double value = 0;
    for(int i = 0; i < filtersize; i++)
        findata >> value;
    //Weiter Einlesen und fortlaufend verarbeiten
    while(findata >> value) {
        //Buffer shiften
        for(int i = 1; i < filtersize; i++)
            bufferdata[i] = bufferdata[i-1];
        bufferdata[filtersize-1] = value;
        fout << discretefolding(bufferdata, filtersize/2, filterdata, filtersize) << endl;
    }
```

## Funktion `main()`

```cpp
def main(int argc, char* argv[])
    if(argc != 4) {
        cerr << "Usage: " << argv[0] << " filter " << endl;
        return 0;
    }
    //Länge des Filters bestimmen und ausgeben
    int lf = get_filter_size(argv[1]);
    cout << "The filter has " << lf << " entries." << endl;
    filter(argv[2],argv[1],lf,argv[3]);
    return 0;
```