

COMPUTER SCIENCE I & II

Summary of the Lectures held by
Prof. Dr. F. E. Cellier and Prof. Dr. F. Mattern

Lukas Cavigelli, June 2010
lukasc[at]ee.ethz.ch

C++ QUICK REFERENCE

DATATYPES

DEFAULT DATATYPES

char	1 Byte	-128 to 127
short	2 Bytes	-32'768 to 32'767
int	2/4 Bytes	usually as below
long int	4 Bytes	-2'147'483'648 to ...
bool	1 Byte	0 or 1
float	4 Bytes	1.2e-38 to 3.4e38
double	8 Bytes	2.2e-308 to 1.8e308
long double	12 Bytes	
void	0 Bytes	-

Maximal values can differ depending on your platform.

unsigned: removes negative range and appends it at the end.

Alternative: std::numeric_limits<signed long>::max()

INLINE TYPE SPECIFICATION

```
2L, 2LL, 2U, 2UL, 2.0 //not case sensitive
2.0f //f has no effect, float
2f //error
2.0d //error
2.1l //long float = double
2.1ll //error
```

VARIABLE/FIELD DECLARATION

```
(vol)(binding)(const)(size)(sign) type name;
vol:      volatile
binding:  register, static, mutable, extern
const:    const or don't specify, if editable
size:     long, short, double, ...
sign:     signed, unsigned
//extreme example:
extern const short unsigned char myVar;
```

DECLARATION FLAGS

volatile: never cache in registers, for shared memory
register: try keeping it in a CPU register
static: value is kept despite leaving the scope
mutable: can be edited from a const function
extern: only declaration(setting the name) instead of definition (requesting memory)

VARIABLE/FIELD NAMES

Variable names cannot start with: any keyword mentioned below or a digit. They should not start with an underscore (_)

Variable names cannot contain: spaces, special characters (ä, ...). Variable names are case sensitive and <31 characters long.

• Keywords:

and, and_eq, asm, auto, bitand, bitor, bool, break, case, catch, char, class, compl, const, const_cast, continue, default, delete, do, double, dynamic_cast, else, enum, explicit, export, extern, false, float, for, friend, goto, if, inline, int, long, mutable, namespace, new, not, not_eq, nothrow, operator, or, or_eq, private, protected, public, register, reinterpret_cast, return, short, signed, sizeof, static, static_cast, struct, switch, template, this, throw, true, try, typedef, typeid, typename, union, unsigned, using, virtual, void, volatile, wchar_t, while, xor, xor_eq

INITIALIZATION

```
int a, b = 5; //a = pseudo-random, b = 5
int c(5);
int d = 1, e = 0, f;
```

NUMBERS

11: decimal (10+1), 011: octal (8+1=9), 0x11: hex (16+1=17)
1000 = 1E3 = 1e3 = 1E+3; 0.1 = 1e-1 = 1E-1

TYPE CASTING

```
new_type expr2 = (new_type) expr1;
new_type expr2 = new_type (expr1); //same
float a = 7/2; /*a=3*/a = 7/(float)2; /*a=3.5
a = float(7/2); /*a=3.5*/ int b = 7/8; //b=0
```

ARRAYS

```
int arr[4]; //arr[0] to arr[3] defined
int arr[][2] = {{1,2},{3,4},{5}};
//ok, 1st dimension is inferred:
//arr[3][2] has a pseudo-random value
int arr[][ ] = {{1,2},{3,4}};
//error: only 1st dimension can be inferred
Initialization of array: size has to be fix at compile-time.
This means: fixed numbers or const variables.
An array is a const pointer. For more → pointer-arithmetics.
```

C-STRINGS

```
char str[] = "bla"; //str[0]='b', str[1]='l', str[2]='a', str[3]='\0'
char str[] = {'b', 'l', 'a', '\0'};
char str[3] = "bla"; //error: array too small
```

BASIC PROGRAM STRUCTURE

```
#include <iostream.h> //argc: #items in argv
int main(int argc, char **argv) {
    cout << "Hello World!" << endl;
    return 0; } //argv: array of c-strings
```

SCOPES AND STATIC VARIABLES

The scope is marked by curly brackets: { ... }
Variables declared inside a scope are only accessible in there.
Their memory is released again and their value lost.

static variables are not accessible outside their scope, but the memory is not released and they keep their value.

OPERATOR PRECEDENCE & ASSOCIATIVITY

Prece- dence	Operator	Descri- ption	Associat- ivity, can override
1	::	scope resolution	~, no
2	() [] -> . ++ --	grouping op array member member of ptr member access op increment decrement	ltr, yes ltr, yes ltr, yes ltr, no ?, yes ?, yes
3	! ~ - + * & new delete type() sizeof	logical negation bitw. complement <u>unary</u> minus, e.g. x = -a; <u>unary</u> plus dereference address of alloc memory dealloc memory cast to <type>, also (type)val sizeof(x)	rtl, yes rtl, yes rtl, yes rtl, yes rtl, yes rtl, yes rtl, yes rtl, no
4	->* .*	member pointer mem. obj. selec.	ltr, yes ltr, no
5	* / %	multiplication division modulus	ltr, yes ltr, yes ltr, yes
6	+ -	addition subtraction	ltr, yes ltr, yes
7	<< >>	bit shift left bit shift right	ltr, yes ltr, yes
8	< <= > >= greater-or-equal	less-than less-or-equal greater-than greater-or-equal	ltr, yes ltr, yes ltr, yes ltr, yes
9	== !=	equal-to not-equal-to	ltr, yes ltr, yes
10	& bitand	bitwise and	ltr, yes ltr, yes
11	^ xor	bitwise xor	ltr, yes ltr, yes
12	 bitor	bitwise or	ltr, yes ltr, yes
13	&& and	logical and	ltr, yes ltr, yes
14	 or	logical or	ltr, yes ltr, yes
15	a?:b:c	if-then-else	rtl, no
16	= +=	assignment op add-and-assign	rtl, yes rtl, yes
17	throw	throw exception	~, no
18	,	sequential op	ltr, yes

ltr=left-to-right

There are more commands like +=:

+=, -=, *=, /=, %/, &=, ^=, |=, <<=, >>=

a = b = c; //b = c, a = c

INCREMENT & DECREMENT OPERATOR

```
int a, b = 5; a = b++; //a = 5, b = 6
int c, d = 5; c = --d; //c = 4, d = 4
```

BRANCHING

IF ... [ELSE IF ...] [ELSE ...]

```
if(a==b) { foo(); }
else if (b==c) { bar(); } //can be omitted
else if (b==c) { bar(); } //can be omitted
else { foobar(); } //can be omitted
```

SWITCH

```
switch(i){
    case 5: cout << "foo";
    case 6: cout << "bar"; break;
    case 7: cout << "omg"; break;
    default: cout << "wtf";
}
//i == 5 -> "foobar"
//i == 9 -> "wtf"
//you cannot declare variables inside a case label!
```

LOOPS

```
break; //leaves the containing loop
continue; //jumps to the start of the loop
goto lbl; //jumps to "lbl:"
lbl: //goto lbl;" continues here
return; //leaves the containing function
```

FOR

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

WHILE

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

DO-WHILE

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

POINTERS & REFERENCES

DECLARATION

```
int a=1,b=2; int* c = &a; //what you want
const int* d = &a;
*d = 5; /* error */ d = &b; //correct
int* const e = &a; //like an array
*e = 5; /* correct */ e = &b; //error
```

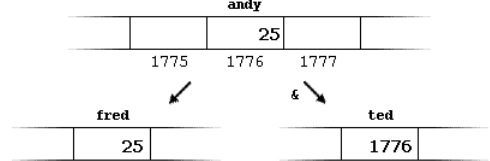
```
const int f = 8; const int* g = &f //ok
int* h = &f; //error
int* i = (int*)0x33;
//i points to mem-addr 0x33, cast necessary
```

DYNAMIC MEMORY (NEW, DELETE)

For c-type dynamic memory see <cstdlib>. 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)[3];` //c=6, d:error
`int* e = new int[3][2];` `int**f = new int[3][2]` //both error
Trick: `#new = #delete` and `#new ... [] = #delete[]` ... //usually
Exception on lack of memory:
`int* dp = new (nothrow) int[100];`
//no exception, but may return null-pointer

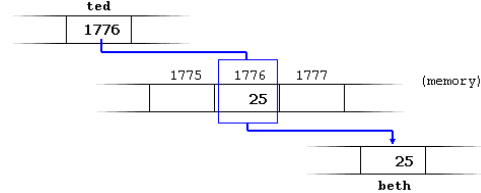
REFERENCE OPERATOR (&)

```
andy = 25; fred = andy; ted = &andy;
```



DEREFERENCE OPERATOR (*)

```
andy = 25; ted = &andy; beth = *ted;
```



REFERENCES

```
int &a; //error: refs are never null
int &b = k; //correct declaration
&b = m; //error: refs cannot change
b = v; //correct value assignment
```

-> OPERATOR

```
(*a) . b is equivalent to a->b Where b can be field, function, ...
a->b = 25; //correct assignment of b's value
```

POINTER-ARITHMETICS

```
int b[100]; int* p;
p = b; //assigns b's address to p
p = &b[0]; //same thing as above
b = p; //error: b is const
p = p+1; //adds (4=1*sizeof(int)) to p(= &b[1])
//-----
b[5] = 0; // b [offset of 5] = 0
*(b+5) = 0; // pointed by (b+5) = 0
//1st == 2nd expression. a can be pointer or array.
```

FUNCTIONS / METHODS

```
[virtual] [static] [const] return-type
name(arg1-type arg1-name, arg2-type [const]
arg2-name[=def_value]) [const] { 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**.
the returned reference cannot be changed
const (1st): arg is **const** within the function
const (arg): overloads the function with a default value
def_value: All optional params have to be at the end.
const (2nd): All the class' fields are treated **const**

ARGUMENTS

- call by value (copy the argument)**
`void foo(int arg1);` // The value of arg1 is copied
//even classes and structs are copied!
`int a = 5; foo(a);` //a will certainly remain 5
- call by reference**
`void foo(int& arg1);` // The value of arg1 is copied
`int a = 5; foo(a);` //a can change
- call by reference (pointer)**
`void foo(int* arg1);` // The value of arg1 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!

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,int) = sub;
    m = op(7,5,add); n = op(20,m,minus); }
```

OVERLOADING

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

INLINE

inline functions are not called like ordinary functions, but the code is copied there. This is faster, but results in a larger binary. Not possible for recursion.

PROTOTYPES/SIGNATURES

```
int foo(float); //or int foo(float a);
Prototypes contain name and signature of a function.
```

Usually there is a header file 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.

ADVANCED DATATYPES

STRUCTS

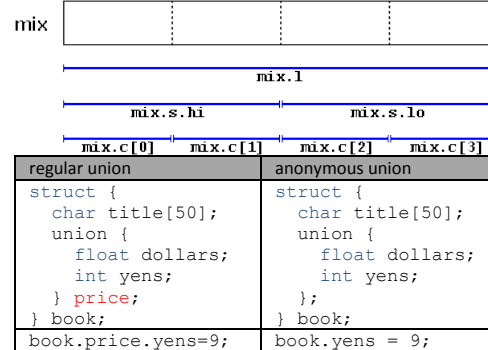
Structs are copied on assignment.
`struct foo_t {`
 type1 a;
 type2 b;
`}` `bar;` // 'bar' is optional, to instantiate

BITFIELDS

```
struct foo_t {
    type1 a:2; // a is 2 bits
    unsigned int b:3; // b is 3 bits
} bar; // 'bar' is optional, to instantiate
```

UNIONS

```
//all vars occupy the same memory space
union mix_t {
    // 'mix_t' optional, type to instantiate later
    long l;
    struct {short hi; short lo;} s;
    char c[4];
} mix; // 'mix' is optional, to instantiate
```



ENUMS

```
enum CountryName_t { /* 'CountryName_t'
optional, to instantiate later*/
US=1, FR=4, CH, GB=9} ia, ib;
// 'ia, ib' optional, to instantiate
// values are generated if not assigned
// only numbers can be assigned
// usage:
CountryName_t foo = CH;
cout << foo; //writes 5
```

EXCEPTIONS

Usually thrown objects inherit from `exception`, but there is no need to do so. You can i.e. throw error-codes as int.

- throw-declaration**
`void foo() throw();` //cannot throw
`int bar() throw(int);` //can throw int only
`int bar();` //can throw anything
- throw-invocation**
`throw 4;`
- try { ... } catch { ... }** //one catch is necessary
`class A : B { ... };`
`try { ... }` //mandatory
`catch(int errNo){...}` //catches ints only
`catch(B& foo){...}` //catches Bs only
`catch(A& bar){...}` //never executes
`catch(...){...}` //catches everything
"..." is meant literally!! Uncaught exceptions terminate the program!

TYPEDF

```
typedef type newSynonymName;
typedef long bigint;
//now you can use bigint instead of int
```

OBJECT-ORIENTED PROGRAMMING (OOP)

CLASSES

```
class A : B, C { ... };
```

In-class init is not permitted for non-static and non-const:
`class D { int a = 5; } //error`

THIS-POINTER

The this-Pointer always points to the underlying object (class).
`class A { int b; void foo() { this->b = 5; } };`

CONSTRUCTOR

```
class C:A,B{ public:C(int x) : B(),A() {} }
//construct b first, then a
There is a default constructor doing nothing, if not specified.
Implicit type cast: C foo = 4;
Implicit init: C foo(4);
Explicit init: C foo = C(4);
The keyword explicit disables implicit type cast.
```

Implicit init not allowed for constructor without args:
`A foo(); /*error*/ A foo; A bar = A(); //ok`

for more → **Inheritance, Polymorphism**

COPY-CONSTRUCTOR

If not specified: bit by bit copy. To specify:
`class A { A(A & copyFromMe) { ... } ... };`
`A a1; A a2 = a1; //calls the copy constructor`
`A a3 = A(a1); //same as above, explicit`

DESTRUCTOR

The destructor of an object is called if it's "delete"-ed.
`class A { public: [virtual] ~A () { ... } ... };`
Attention! `A* b1 = new B; delete b1;`
only calls A's destructor! -> `virtual-ize` and call B's too!!

STATIC & CONST VARIABLES

`static` variables' memory is not freed, once it's not accessible anymore, but stays in memory, so once you can access it again it still has the same value.
`static` variables on class-level have the same value among all instances of a class.

`const` variables cannot be changed. On a class level they always have to be static: `static const int a = 3;`

INHERITANCE

Is by default `private`. `friend`-methods can never be inherited.

- **public ("is-a" relation; e.g. banana is a fruit)**
public -> public, protected -> protected, private -> --

- **protected**
public -> protected, protected -> protected, private -> --
The inherited methods can be further inherited

- **private ("has-a" relation)**
public -> private, protected -> private, private -> --
The inherited methods CANNOT be further inherited.
Alternative: create the object as a field

virtual flag by example:

```
class L { public: virtual void eat(); };
class A : public virtual L { ... };
class B : public virtual L { ... };
class C : public A, public B { ... };
class D : public C { ... };
//because A and B inherit L 'virtual' it can be
bound to the same L. Else there would be 2 Ls in C
The most derived class (here: D) has to
construct all virtual-inherited sub-
objects(here just L). This cannot be done
by C for D!!
```

DYNAMIC BINDING

`static` binding is
- used by default and far more performant
- method cannot be overwritten (but still overloaded)
`dynamic` binding is
- specified by the keyword `virtual`
- can be overwritten from a inheriting class.

Dynamic binding sample:

```
class A { virtual void f(); };
class C : A { void f(); };
A* pa = new C;
pa->f(); //invokes C::f();
```

ABSTRACT CLASSES

A class is abstract, if at least one of its methods is purely virtual.
Declaration: `virtual double foo() = 0;`

ACCESS MODIFIERS

Modifiers are only allowed inside classes.

public: Accessable by everyone
private: Only accessible within the class
protected: Only accessible within and from inheriting class
friend: Accessable to that class for which it is friend

OVERRIDING OPERATORS & TYPE CASTS

```
operator int(){ return 4; } //typecast
double& Matrix::operator()(int c, int r);
Matrix a; double val = a(5,4);
void Matrix::operator() () { this->delete() }
```

```
void operator=(const A bla){}; //assign-op
A a1; A a2; a2 = a1; //assignment
//do not confuse with copy-constructor
```

```
int b1; A b2; b1+b2; //calls funct below
friend float operator+(int i,A j) const;
```

```
//creating new operators is not allowed(i.e. $)
int main(){int i = 2; return i.operator+(1);} //ok
```

NAMESPACES

```
const int a = 2;
namespace foo {
    const int a = 1;
    int bar() { return a; } //returns 1;
    int foo() { return ::a; } //returns 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 { };
//=====Ambiguities=====
C* pc; pc->ii;
//error: A::ii or B::ii?
pc->A::ii; //C's A's ii
//the same is valid for methods.
//=====Multiple Sub-objects=====
class L { int m; };
class A : L { ... };
class B : L { ... };
class C : A, B { ... };
//-> two Ls in C, they can be different
//access through explicit qualification:
void C::f() { int k = A::L::m; }
//=====Casting=====
C* pc = new C;
L* pl = pc; //error: ambiguous
pl = (L*)pc; //error: ambiguous
pl = (L*)(A*)pc; //ok.
//=====Casting in method calls=====
extern f(L*); //some standard method
A aa; C cc;
F(&aa); // ok.
F(&cc); // error: ambiguous
F((A*)&cc); // ok.
```

TEMPLATES

FUNCTION TEMPLATES

```
//declaration/definition
template <class Any>
void Swap(Any &a, Any &b) {
```

```
Any temp; temp = a; a = b; b = temp; }
//usage
int x,y; Swap(x,y); //correct
```

CLASS TEMPLATES

still to do

LIBRARIES

• Location

```
#include "blabla.h" //search locally
#include <blabla.h> //search standard lib
```

• Language

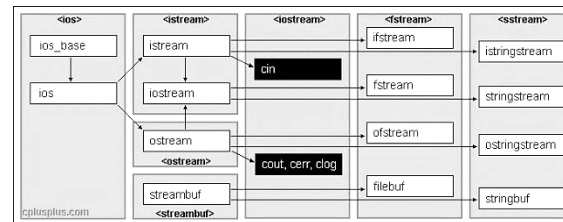
```
#include <blabla> //c++
#include <cblabla> //c
#include <blabla.h> //c, old style
```

HEADER FILES

Header files (.h) are just like regular .cpp files. They are usually used to declare classes and prototypes of functions and to define constants and small functions. They are often shipped with a library for the programmer to know what's inside. The have to be included when compiling!

-----usual structure of a header file-----

```
#ifndef AA_H
#define AA_H
class Aa { int foo(int k) const;};
```



```
#endif /* AA_H */
```

<IOSTREAM> - INPUT & OUTPUT STREAMS

```
cout << "foo"; //console out
cerr << "foo"; //console error
clog << "foo"; //console log
cout << "foo" << endl << "bar";
```

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

```
int i;
cin >> i; //reads an int from console into i
```

control characters:

```
\b backspace \f form feed \' apostrophe \t tab
\r return \n newline \" quote
\\ character #n (octal) \\nn character #nn (hex)
```

<CSTDIO> - C INPUT & OUTPUT

<FSTREAM> - FILE STREAMS

```
#include <fstream>
#include <iostream>
int main() { int a, b;
ofstream file1("foo.txt"); //c-string!
if (file1.is_open()) { file1 << a; }
ifstream file2("bar.txt"); file2 >> b;
return 0; }
```

<CMATH> - MATHEMATICAL FUNCTIONS

trig. functions	sin(x), cos(x), tan(x)
inv. trig. func.	asin(x), acos(x), atan(x)
arctan(y/x)	atan2(y,x)
hyperbolic trig	sinh(x), cosh(x), tanh(x)
exponential, log	exp(x), log(x), log10(x)
exp, log (2 pow)	ldexp(x,n), frexp(x,*e)
division & remain	modf(x, *ip), fmod(x,y)
powers	pow(x,y), sqrt(x)
rounding	ceil(x), floor(x), abs(x)

All args and `return` values are `double`.

<TYPEINFO>

<EXCEPTION> - DEFAULT EXCEPTION CLASS

<CTYPE> - CHARACTER FUNCTIONS

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

<CSTDLIB>

memory: `malloc`, `calloc`, `realloc`, `free`
files: `remove("foo.txt");`
`exit(int);`

<VECTOR> - DYNAMIC SIZE ARRAY

<STRING> - C++ STRING CLASS & FUNCTIONS

```
typedef basic_string<char> string;
typedef basic_string<wchar_t> wstring;
//below 'string' means any of the above
istream& std::getline ( istream& is,
string& str, char delim = '\n' );
const char* s.c_str(); //string to c-string
size_type s.length();
much more to come
```

COMMON CONCEPTS

RANDOM NUMBER GENERATION

```
#include <cstdlib>
#include <ctime>
#include <iostream>
using namespace std;
inline int randomNumber(int min, int max) {
return rand()%(max-min+1) + min;
}
```

```
int main() {
    srand(time(NULL)); // init with seed
    cout << randomNumber(1,100) << endl;
    return 0;
}
```

BITMASKS

Bitmaks can be used after applying encryption algorithms to make sure the characters are actual letters and not control commands.

COPYING AN ARRAY

If it's a char-array, use strcpy; else you can embed the array into a struct or class and copy that.

TIME COMPLEXITY: O(...)

Slowest to fastest. K = constant, N = # items:

hyper-exponential	$N^{K_1^{K_2^N}}$
super-exponential	N^N
factorial time	$N!$
exponential time	K^N
polynomial time	N^K
linearithmic time	$N \cdot \log(N)$
linear time	$K \cdot N$
logarithmic time	$K \cdot \log(N)$
constant time	K

1 < log n < √n < n < n(log n)² < n² < n^a < aⁿ; log_cb = log_ab / log_ac
Cases: best-case, average-case, worst-case, amortized worst-case (average worst-case for infinite #inputs)

NO FALSE TRUST

1.
- O-Notation describes the upper limit
2.
- $g(n) = C \cdot O(N) \rightarrow C$ can be very large!

RULES

- $C = O(1)$
- $C \cdot f(n) = O(f(n))$
- $O(f(n)) + O(f(n)) = O(f(n))$
- $O\left(O(f(n))\right) = O(f(n))$
- $g(n) = a_k \cdot n^k + a_{k-1} \cdot n^{k-1} + \dots + a_0 = O(n^k)$
- $O(f(n)) \cdot O(g(n)) = O(f(n) \cdot g(n))$
- $O(f(n)) + O(g(n)) = O(\max\{f(n), g(n)\})$

Name	best-case	avg.-case	worst-case
Prim	$O(E)$	---	$O(V^2)$
Quicksort	$O(n \log(n))$	$O(n \log(n))$	$O(n^2)$
Bubble, Insertion	$O(n^2)$	$O(n^2)$	$O(n^2)$
Mergesort	$O(n \log(n))$	$O(n \log(n))$	$O(n \log(n))$

COMPLEXITY OF BASIC ELEMENTS

simple instruction	$O(1)$
sequence(f1;f2;f3;)	$O(\max\{O(f_1), O(f_2), O(f_3)\})$
loop(n runs)	$O(n \cdot f_{inner}(n))$
if-else (if(foo){f1;}else{f2;})	$O(\max\{f_1, f_2\})$

NOTATIONS OVERVIEW

$O(g(n))$	$O(g(n)) \leq g(n) \cdot k$, for some k
$\Omega(g(n))$	$\Omega(g(n)) \geq g(n) \cdot k$, for some k
$\Theta(g(n))$	$g(n) \cdot k_1 \leq \Theta(g(n)) \leq g(n) \cdot k_2$, for some k_1, k_2
$o(g(n))$	$o(g(n)) \leq g(n) \cdot \varepsilon$, for every ε
$\omega(g(n))$	$\omega(g(n)) \geq g(n) \cdot k$, for every k

US-ASCII TABLE

Dec	Char	Dec	Char	Dec	Char	Dec	Char
0	NUL	64	@	128	Ç	192	Ł
1	SOH	65	A	129	Ù	193	Ű
2	STX	66	B	130	É	194	Ź
3	ETX	67	C	131	À	195	Ț
4	EOT	68	D	132	Ā	196	—
5	ENQ	69	E	133	À	197	+
6	ACK	70	F	134	Ā	198	ƒ
7	BEL	71	G	135	Ç	199	ƒ
8	BS	72	H	136	È	200	Ł
9	TAB	73	I	137	È	201	ƒ
10	LF	74	J	138	È	202	ƒ
11	VTB	75	K	139	İ	203	ƒ
12	FF	76	L	140	İ	204	ƒ
13	CR	77	M	141	İ	205	=
14	SO	78	N	142	À	206	ƒ
15	SI	79	O	143	À	207	±
16	DLE	80	P	144	É	208	ƒ
17	DC1	81	Q	145	æ	209	ƒ
18	DC2	82	R	146	Æ	210	ƒ
19	DC3	83	S	147	Ö	211	Ł
20	DC4	84	T	148	Ö	212	Ł
21	NAK	85	U	149	Ö	213	ƒ
22	SYN	86	V	150	Ü	214	ƒ
23	ETB	87	W	151	Ü	215	ƒ
24	CAN	88	X	152	Ÿ	216	=
25	EM	89	Y	153	Ö	217	ƒ
26	SUB	90	Z	154	Ü	218	ƒ
27	ESC	91	[	155	€	219	■
28	FS	92	\	156	€	220	■
29	GS	93	]	157	¥	221	■
30	RS	94	^	158	?	222	?
31	US	95	_	159	?	223	?
32	space	96	`	160	á	224	α
33	!	97	a	161	í	225	ß
34	"	98	b	162	ó	226	Γ
35	#	99	c	163	ű	227	π
36	\$	100	d	164	ñ	228	Σ
37	%	101	e	165	Ŋ	229	σ
38	&	102	f	166	ª	230	μ
39	'	103	g	167	º	231	ı
40	(	104	h	168	¿	232	Φ
41	)	105	i	169	?	233	Θ
42	*	106	j	170	¬	234	Ω
43	+	107	k	171	½	235	δ
44	,	108	l	172	¼	236	∞
45	-	109	m	173	j	237	Φ
46	.	110	n	174	«	238	€
47	/	111	o	175	»	239	∩
48	0	112	p	176	?	240	≡
49	1	113	q	177	■	241	±
50	2	114	r	178	■	242	≥
51	3	115	s	179	—	243	≤
52	4	116	t	180	—	244	?
53	5	117	u	181	=	245	?

54	6	118	v	182	—	246	+
55	7	119	w	183	—	247	≈
56	8	120	x	184	—	248	°
57	9	121	y	185	—	249	?
58	:	122	z	186	—	250	.
59	;	123	{	187	—	251	√
60	<	124		188	—	252	ˆ
61	=	125	}	189	—	253	ˆ
62	>	126	~	190	—	254	■
63	?	127	?	191	—	255	DEL

BUILD-PROCESS & COMPILER

PREPROCESSOR DIRECTIVES

```
#define NAME VALUE
#define NAME
#undef NAME
#ifdef NAME
#ifndef NAME
#if EXPR
#else
#elif EXPR
#endif
#error MSG
#include
#pragma
```

- Default variables:
__FILE__
__LINE__
__TIME__
__DATE__

G++ COMPILER FLAGS

```
g++ -ansi -pedantic -Wall -O -o myProg
myProg.h myProg.cpp

-----
-c : compile, assemble, do not link
-E : only run preprocessor, source -> stdout
-fall-virtual : dynamic binding by default
-fenum-int-equiv : allows conversion int->enum
-fno-default-inline : never optimize to inline
-g : produce debugging information
-Idir : search dir for include files
-O : optimize. Needs a lot of memory
-o file : place output in "file"
-S : Generates assembler files.
-Wall : output all warning to stdout
-ansi : enables compilation all ansi code
-pedantic : allows only ansi-conform code
There are more. (use 'man g++')
```

G++ AS COMPILER ONLY

```
g++ -Wall -pedantic -ansi -c main.cpp
Creates a files called main.o
```

G++ AS LINKER ONLY

```
g++ -Wall -o upncalc Complex.o Stack.o main.o
```

MAKEFILES

```
# Simple Makefile for a small project
.COMPLEXSTACK: clean
CC = g++
```

```
CFLAGS = -Wall -pedantic -ansi -c
LFLAGS = -Wall
OBSJ = main.o Stack.o Complex.o
upncalc: $(OBSJ)
    $(CC) $(LFLAGS) -o upncalc $^
$(OBSJ): %.o: %.cpp
    $(CC) $(CFLAGS) $<
main.o: Stack.h Complex.h
Stack.o: Stack.h Complex.h
Complex.o: Complex.h
clean:
    rm -rf *.o upncalc
# ^ has to be a tab!!!!
```

IMPROVEMENTS, CHECKLIST

COMPLETE “DON’T”S

- Assigning values to a function, especially when returning a reference
int& foo(int& arg1) { return arg1; }
void bar() {
 int a = 4; foo(a) = 6; } //a is now 6

IDEAS

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

POSSIBLE MISTAKES / CHECKLIST

- = vs. ==
- using namespace
- #include libraries
- Semicolons ;
- Case sensitivity
- Sind Funktionen, Variablen und Klassen wirklich deklariert bei erster Verwendung? (evtl. Header-File)
- Header-File included?
- Arrays with non-const size
- There is no 'null' in C/C++
- Recursion with an inline function
- All purely virtual functions defined?
- Semicolon after class, struct, enum, ... definition
- Circular dependency without pointers (struct a_t {a_t bla;});
- "chain of const": const pointer only to const fields
- "chain of const": const functions only to other const func.
- "chain of const": to pass a const field, the function's arg has to be const
- Are all modifiers correct? (e.g. con-/destructor public:)
- Destructor specified dealloc memory?
- Destructors all correct? (virtual, if class inherited)
- No modifiers outside of classes
- No in-class initialization: class A { int a=2; } //error
- In C++ modifiers need a colon :
- A k();/*err*/ A k;/*ok*/ A k = A(); //ok
- Just for the beauty: catch eventual bad_alloc-exceptions
- Watch out for ambiguous function calls (unsigned vs. char)
- Watch out for ambig. func. calls (float vs. double): foo(10);
- string length: space for \0 reserved?
- Every pointer correctly used? . vs. ->
- Never retured a reference to a local variable?