Plan for today

TECHNISCHE UNIVERSITÄT DARMSTADT

- strings in C
- logical and composite logical expressions
- local variables
- functions
- recursive functions
- bundling and encapsulating data, a very bit C++

- Discussion and outlook (Maple)

Programming language C

TECHNISCHE UNIVERSITÄT DARMSTADT

Strings in C are nothing else than zero-terminated arrays of char.

"Hello World"-String in memory:

ac =											
H 72	e 101	l 108	l 108	o 111	 32	W 87	o 111	r 114	l 108	d 100	'\0' 0

```
int strlen(char *s)
{
      int n;
      for (n = 0; *s != '\0';s++)
          // same as s != 0, not the same as s != '0'
          n++;
      return n;
}
```

let us replace the "H" bei "0":

```
ac[0] = '0';
printf(" %c %d %d\n",ac[0],ac[0],strlen("Hello World");
```

generates the following output:

0 48 11

Simple Logical Expressions

C uses logical expressions in while and for loops and in if statements.
A logical expression evaluates to either True or False!

Logical operators must compare terms of the same type!

The binary logical operators are:

x > y	x is greater than y
x >= y	x is greater or equal to y
x < y	x is less than y
x <= y	x is less or equal to y
x == y	x is equal to y (**attention!!! two =**)
x != y	x is not equal to y

if **le** is a logical expression then **!le** has the opposite true/false value

Composite Logical Expressions

Two or more simple logical expressions can be combined with the logical operators ‘!’, ‘and and ‘or’ into a single compound expression. Let l1 and l2 be logical expressions.

|| is the C operator for ‘or’
Example: l1 || l2 **is true if either** l1 is true **or** l2 is true **or both are true**

&& is the C operator for ‘and’
Example: l1 && l2 **is true if either** l1 is true **only if both l1 and l2 are true**

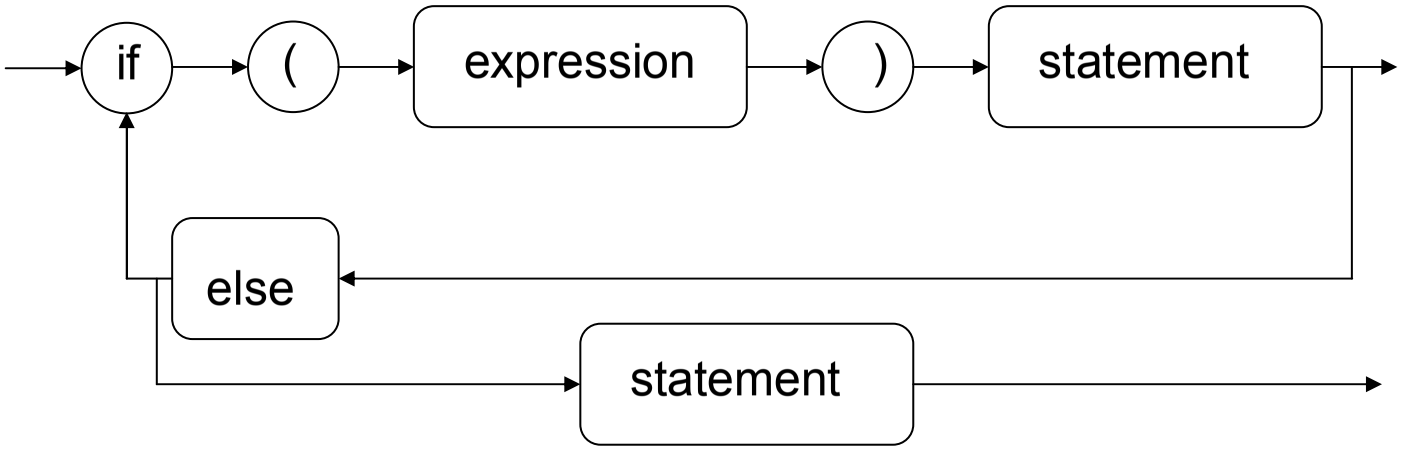
! is the C operator for ‘not’
Example: !l1 is true if and only if l1 is false

Programming language C

TECHNISCHE UNIVERSITÄT DARMSTADT

Flow control:

Alternative program flow



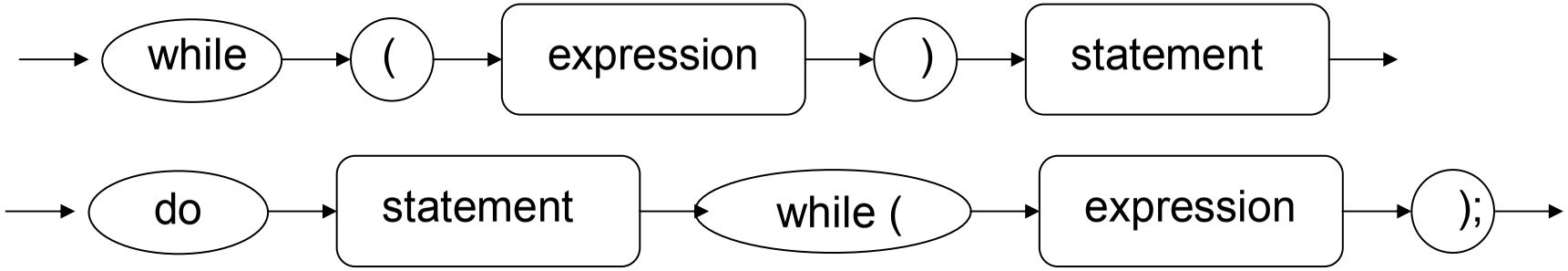

```
if (i < 3) printf("i is smaller than 3");
else if (i > 3) printf("i is larger than 3.");
else printf("i is equal to 3.");
```

Programming language C

TECHNISCHE UNIVERSITÄT DARMSTADT

Flow control:

while loops




```
while (i < 3) {
   printf("i is %d",i);
   i = i + 1;
   i++;
}
```

```
do {
   printf("i is %d",i);
   i = i + 1;
   i++;
} while (i < 3);
```

Remark:

```
statement1; while (expr) { statements; last statement; }
```

is equivalent to

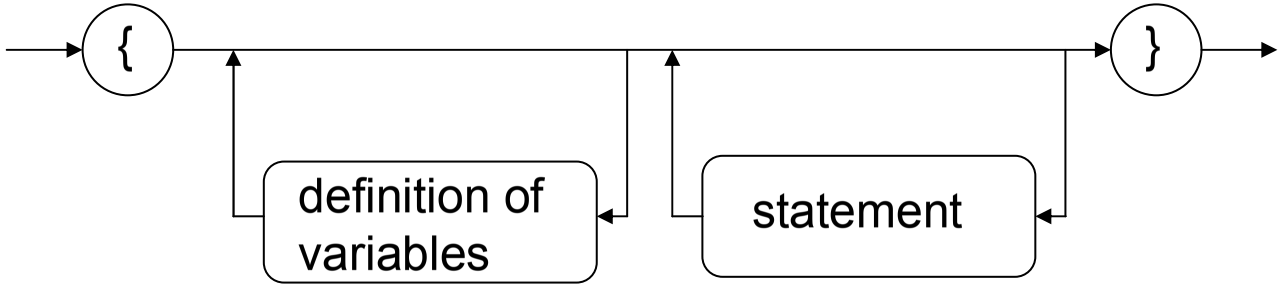
```
for (statement1; expr; last statement) { statements; }
```

Programming language C

TECHNISCHE UNIVERSITÄT DARMSTADT

Flow control II:

Blocks: a block is a sequence of statements, grouped by brackets



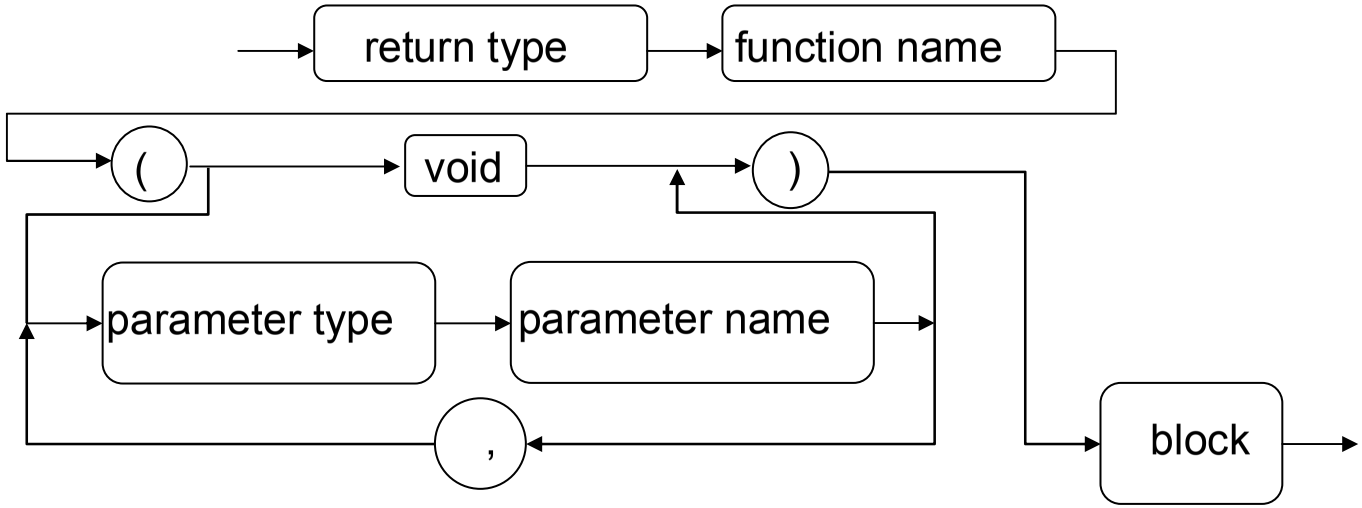

```
#include <stdio.h>
 int main(void) {
    printf("Hello World\n“);
    return 0;
 }
```

block

Programming language C

TECHNISCHE UNIVERSITÄT DARMSTADT

Functions (procedures, methods), --> named blocks




```
int square(int x)
{
    int result = x*x;
    return result;
}
```

```
usage in main:
....
  int a;
  a = square(x);
...
```

Functions

- There are only very few functions pre-built in C. We have to write our functions ourselves.
- There are some rules for functions in C:
  - Functions may return a value, i.e. they may be declared to be of a specific type as float or int. Functions declared to be void return nothing.
  - All arguments to a function are passed „by value“ and are uneffected by the function call. A copy of the original data is passed.

Example: Real Roots

We want to write a function which calculates the roots of a real quadratic equation.

Idea:
Input: A, B, C of the equation $A*x^2 + B*x + C = 0$
Returned value:
-1 if A = 0
0 if there are two distinct real roots
1 if a pair of complex roots exists
2 if the two roots are identical
and: the roots for the 4 cases in two variables r1 and r2

$$r_{1,2} = \frac{-B \pm \sqrt{B^2 - 4AC}}{2A}$$

Programming language C

TECHNISCHE UNIVERSITÄT DARMSTADT

Example: Real Roots of $A*x^2 + B*x + C = 0$

```
int roots(double A, double B, double C, double *r1, double * r2) {
  double d, dr;
  if (A == (double)0.0) { *r1 = -C/B; return -1; }
  d = B*B - (double)4.0*A*C;
  if ( d > (double)0.0 ) {
      dr = sqrt(d);
      *r1 = (-B-dr) / ((double)2.0 * A);
      *r2 = (-B+dr) / ((double)2.0 * A);
      return 0;
  } else  if (d == (double)0.0) {
      *r1 = *r2 = (-B-dr) / ((double)2.0 * A);
      return 2;
  } else {
      /* no implementation */
      return 1;
  }
}
```

Example: Real Roots

```
#include <math.h>
#include <stdio.h>

int roots(double A, double B, double C, double *r1, double * r2);

int main(void) {
   double A, B, C, result1, result2;
   switch ( roots(A, B, C, &result1, & result2)) {
      case -1:  printf("Linear case, one root. r=%lf\n",result1);
                break;
      case  2:  printf("One root. r=%lf\n",result1);
                break;
      case  0:  printf("Two real roots. r=%lf and r=%lf\n",
                                              result1,result2);
                break;
     default:   printf("Complex roots not implemented.\n");
   }
   return 0;
}
```

Mathematical Functions

- C knows only basic arithmetics. However, there is an ANSI Standard C library of mathematical functions. The library has its own header file: <math.h> Do not forget to link –lm (gcc myprog.c –o myprog -lm)

- The standard C mathematical functions work only with the type *double*.
  - *double* in and *double* out

- Some of the standard functions are:

sin(x)	asin(x)	sinh(x)	exp(x)
cos(x)	acos(x)	cosh(x)	log(x)
tan(x)	atan(x)	tanh(x)	log10(x)
sqrt(x)	atan2(x,y)	pow(x,y)	fabs(x)

Programming language C

TECHNISCHE UNIVERSITÄT DARMSTADT

Variables, visibility:

- global variables are accessible everywhere; are defined outside any block
- local variables have a “scope”. They are only valid in those blocks, where they have been defined.
  Examples:

```
{
 int i;
 i = 2;
 {
   int j,k;
   j = 3;
   k = i + j;
 }
 i = k; // Error!
}
```

```
{
 int i, j;
 i = 0;
 {
   int i;
   i = 1; //new definition of i
          //shadows old one
 }
 j = i; // j = 0
}
```

Programming language C

TECHNISCHE UNIVERSITÄT DARMSTADT

Memory allocation

static global variable allocation: outside of any procedure

- global: accessible from everywhere
- static: is fixed with the start of the program

local variable allocation: inside blocks or functions

- only accessible in the currently alive blocks

dynamic allocation: with the help of malloc(size_t s) and free()

e.g.

```
int *a = (int*)malloc(sizeof(int) * 1000);
//use a in the same way as an array
free(a);
```

Recursion

Observation: gcd(a, b) = gcd(b, a mod b)

Claim: The set of divisors of a and b is equal to the set of divisors of b and (a mod b). (Therefore, especially the greatest common divisor is the same.)

"=>" Let t be a divisor of a and b, i.e. there exist positive integers x and y with a = xt and b = yt

Now, let r := a mod b. Then there is a positive integer z with

a = zb + r = zyt + r

=> (a =) xt = zyt + r => t(zy-x) + r = 0 => r = t(x-zy)

thus: r is a multiple of t, and: **t is a divisor of r**

Recursion

Observation: gcd(a, b) = gcd(b, a mod b)

Claim: The set of divisors of a and b is equal to the set of divisors of b and (a mod b). (Therefore, especially the greatest common divisor is the same.)

"<=" Let t be a divisor of b and of r := a mod b. Then there exist positive integers x and y with b=xt and r=yt.

Because r = a mod b, there is a positive integer z with a=zb+r

=b =r

=> a = z xt + yt = (zx+y)t

thus: a is a multiple of t, and: **t is a divisor of a**

Programming language C

TECHNISCHE UNIVERSITÄT DARMSTADT

Recursion

Observation: gcd(a, b) = gcd(b, a mod b)

C function, Euklid's algorithm:

```
unsigned int gcd(unsigned int a, unsigned int b)
{
   if (b == 0) return a;
   else return gcd(b, a % b);
}
```

Euklid's algorithm, number of function calls

We inspect the sequence of parameters to the function gcd:
$(a,b) = (a_0,b_0) \rightarrow (a_1,b_1) \rightarrow \ldots (a_{k-1},b_{k-1}) \rightarrow (a_k,b_k)$

Observation: for all $i < k-1$ is valid: $b_i > 2b_{i+2}$
Proof: It is $a_{i+1}=b_i$ and $b_{i+2}=a_{i+1} \bmod b_{i+1}$
from $b_{i+2}=a_{i+1} \bmod b_{i+1}$ follows
there is a positive integer x with $a_{i+1} = xb_{i+1} + b_{i+2}$
$\Rightarrow b_i = a_{i+1} = xb_{i+1} + b_{i+2}$
Because $x \geq 1$ and $b_{i+1} > b_{i+2}$ it follows that $b_i > 2b_{i+2}$

Therefore, $b_0 > 2^{(k-1)/2} \cdot b_{k-1} \geq 2^{(k-1)/2}$ and the number of function calls to gcd is less than $k < 1 + 2 \log(b_0)$

Programming language C

TECHNISCHE UNIVERSITÄT DARMSTADT

Keywords:

auto	break	case	char	const	continue
default	do	double	else	enum	extern
float	for	goto	if	int	long
register	return	short	signed	sizeof	static
struct	switch	typedef	union	unsigned	void
volatile	while				

C++ classes

TECHNISCHE UNIVERSITÄT DARMSTADT

- we switch to another compiler tool: g++ and another source-file suffix .cc
- a class is a bundle of variables (attributes) and functions (methods)
- Syntax

```
class <classname> {
 [public, protected, private:]
   variable declaration1;
   variable declaration2;
     ...
   method1
   method2 ...
};
```

Example:

```
class A {
   public:
        int a;
        int b;
        int add(int a, int b) {
              return a+b;
        }
};
```

- access over „.“ or „->“, instantiation via „new“

```
...
A* p_o = new A;
A o;
x = o.a;
y = p_o->a;
delete p_o;
```

C++ classes

TECHNISCHE UNIVERSITÄT DARMSTADT

- C-integers have 32 or 64 bits. We want to build our own arithmetics in order to deal with bigger integers.
- Idea: Take two integers and interprete them as one number, and use the system arithmetics as far as possible.

```
class BiggerInt {
  private:
    long high_bits;
    unsigned long low_bits;
  public:
    BiggerInt *add(BiggerInt *a, BiggerInt *b) ; // add two numbers
    BiggerInt *sub(BiggerInt *a, BiggerInt *b) ; // subtract number b from a
    BiggerInt *mul(BiggerInt *a, BiggerInt *b) ; // multiply two numbers
    BiggerInt *div(BiggerInt *a, BiggerInt *b) ; divide a by b
    BiggerInt(long a); // “constructor”; aim: create a long valiable and initialize it
    BiggerInt(long h, unsigned long l); // “constructor”; aim: create a new BiggerInt
    void PrintBits(void) ;
    ~BiggerInt();
};
```

C++ classes

TECHNISCHE UNIVERSITÄT DARMSTADT

```
BiggerInt *add(BiggerInt *a, BiggerInt *b) {
   int carry; unsigned long low_sum; long high_sum;

   low_sum = a->low_bits + b->low_bits;
   if (low_sum < a->low_bits || low_sum < b->low_bits) carry = 1;
   else carry = 0;
   high_sum = a->high_bits + b->high_bits + carry;
   return new BiggerInt(high_sum, low_sum);
}
BiggerInt(long a) {
   if (a < 0) high_bits = -1;
   else high_bits = 0;
   low_bits = (unsigned long) a;
 }
 BiggerInt(long h, unsigned long l) {
   high_bits = h;
   low_bits = l;
 }
```

C++ classes

TECHNISCHE UNIVERSITÄT DARMSTADT

- how objects are accessed

```
int main(void)
{
  BiggerInt *bi1 = new BiggerInt(73); // get memory for a BiggerInt and call
                                      // constructor
  BiggerInt *bi2 = new BiggerInt(73);
  BiggerInt *bi3 = bi1->add(bi1,bi2); // (*b1).add(bi1,bi2);

  bi1->PrintBits();
  bi3->PrintBits();

  delete b1;
  delete b2;
  delete b3;
}
```

C++ further reading

TECHNISCHE UNIVERSITÄT DARMSTADT

- Good Object Orientation allows „inheritance“.
- Example: Let A be a class with some attributes and some methods.

  If we now build a class B by the definition

```
class B : public A {
   ...
}
```

  B contains all attributes and methods of A, execpt constructors and destructors.
  We can add additional attributes and methods to B.
- **In fact, C++ starts here, but we end up here, because we want to work with the software tool „Maple“ next time.**