# while-based loops

- Condition-controlled loops
- Pre-test vs. post-test loops
- Operators ++ and --

- while
- do - while



## The while loop

- Structure: while(condition) statement
  - condition is evaluated
  - If condition is true, statement is executed
  - This repeats until condition becomes false
- condition is checked *before* the block is executed
  - So-called pre-test loop
  - No execution of statement is possible
- statement can be a block of code
- Compare with for( ; condition ; ) in binSearch

```
#include <stdio.h>

main() {
  int counter = 5;

  while (counter > 0) {
    printf("%d ",counter);
    counter = counter-1;
  }
  printf("\n");
}
```

- Output:
  5 4 3 2 1



## Operators ++

- ++a and a++ are arithmetically equivalent to a=a+1
- Additional evaluation of the variable a
- Pre-increment ++a
  - First increase, then evaluate
- Post-increment a++
  - First evaluate, then increase

```
#include <stdio.h>

main() {
  int a = 0;
  int b = 43;

  printf("1) a=%d, b=%d\n",a,b);

  b = a++;
  printf("2) a=%d, b=%d\n",a,b);

  b = ++a;
  printf("3) a=%d, b=%d\n",a,b);
}
```

- Output:
  1) a=0, b=43
  2) a=1, b=0
  3) a=2, b=2



## Operators ++ and --

- Operators similar to a++ and ++a
  - Pre-decrement --
    - First decrease, then evaluate
  - Post-decrement --
    - First evaluate, then decrease
- Note the difference in condition-controlled loops!

```
 1 #include <stdio.h>
 2
 3 main() {
 4   int counter = 5;
 5
 6   while (--counter>0) {
 7     printf("%d ",counter);
 8   }
 9   printf("\n");
10 }
```

- Output: 4 3 2 1 (for --counter in line 6)
- Output: 4 3 2 1 0 (for counter-- in line 6)

## Bisection method (revisited)

- Input
  - Continuous function $f : [a, b] \to \mathbb{R}$ satisfying $f(a)f(b) \le 0$
  - Tolerance $\tau > 0$
- Intermediate value theorem
  - There exists $x \in [a, b]$ with $f(x) = 0$
- Task
  - Find approximation of a zero of $f$
  - i.e., find $x_0 \in [a, b]$ with the following property: $\exists x \in [a, b]$ such that $f(x) = 0$ and $|x - x_0| \le \tau$
- Bisection method = Interval halving method
  - As long as $|b - a| > 2\tau$
    - Compute midpoint $m$ of $[a, b]$ and $f(m)$
    - If $f(a)f(m) \le 0$, consider $[a, m]$
    - Otherwise consider $[m, b]$
  - $x_0 := m$ is the desired approximation
- The method terminates after a finite number of steps
- Convergence towards $x \in [a, b]$ with $f(x) = 0$ as $\tau \to 0$



## Bisection method (revisited)

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

double f(double x) {
  return x*x + exp(x) -2;
}

double bisection(double a, double b, double tol){
  double fa = f(a);
  double m = 0.5*(a+b);
  double fm = 0;

  while ( b - a > 2*tol ) {
    m = 0.5*(a+b);
    fm = f(m);
    if ( fa*fm <= 0 ) {
      b = m;
    }
    else {
      a = m;
      fa = fm;
    }
  }
  return m;
}

main() {
  double a = 0;
  double b = 10;
  double tol = 1e-12;
  double x = bisection(a,b,tol);

  printf("Approximate zero x=%g\n",x);
  printf("Function value f(x)=%g\n",f(x));
}
```

- Using the variables `fa` and `fm` avoids a double evaluation of `f`



## Euclidean algorithm

- Input: Two integers $a, b \in \mathbb{N}$
- Task: Compute greatest common divisor $gcd(a, b) \in \mathbb{N}$
- Euclidean algorithm:
  - If $a = b$, then $gcd(a, b) = a$
  - If $a < b$, swap $a$ and $b$
  - It holds that $gcd(a, b) = gcd(a - b, b)$, because:
    - Let $g$ a divisor of $a, b$
    - i.e., $ga_0 = a$ and $gb_0 = b$ with $a_0, b_0 \in \mathbb{N}$, $g \in \mathbb{N}$
    - Hence, $g(a_0 - b_0) = a - b$ and $a_0 - b_0 \in \mathbb{N}$
    - i.e., $g$ divides both $b$ and $a - b$
    - Hence, $gcd(a, b) \le gcd(a - b, b)$
    - Similarly, $gcd(a - b, b) \le gcd(a, b)$
  - Replace $a$ with $a - b$ and repeat the above steps
- Algorithm yields $gcd(a, b)$ in finitely many steps:
  - If $a \ne b$, $n := \max\{a, b\} \in \mathbb{N}$ becomes smaller at each step
  - After finitely many steps, $a \ne b$ does not hold



## Euclidean algorithm

```
#include <stdio.h>

main() {
  int a = 200;
  int b = 110;
  int tmp = 0;

  printf("gcd(%d,%d)=",a,b);

  while (a != b) {
    if ( a < b) {
      tmp = a;
      a = b;
      b = tmp;
    }
    a = a-b;
  }

  printf("%d\n",a);
}
```

- Computation of gcd of $a, b \in \mathbb{N}$
- Basic idea: $gcd(a, b) = gcd(a - b, b)$ for $a > b$
- For $a = b$, it holds that $gcd(a, b) = a = b$
- Output:

```
gcd(200,110)=10
```

## Euclidean algorithm (improved)

- Key part of the previous implementation

```
while (a != b) {
  if ( a < b) {
    tmp = a;
    a = b;
    b = tmp;
  }
  a = a-b;
}
```

- Recall: $a\%b$ is the rest of the integer division $a/b$
- Euclidean algorithm iterates $a := a - b$ until $a \leq b$
  - i.e., until $a = a\%b$
  - Finally, it holds that $a = 0$ and $b = gcd(a, b)$

```
while (a != 0) {
  if ( a < b) {
    tmp = a;
    a = b;
    b = tmp;
  }
  a = a%b;
}
```

- The rest always satisfies $a\%b < b$
  - i.e., always variable swap after the computation
  - Finally, it holds that $b = 0$ and $a = gcd(a, b)$

```
while (b != 0) {
  tmp = a%b;
  a = b;
  b = tmp;
}
```



## The do-while loop

- Structure: do statement while(condition)
  - statement is executed and condition is evaluated
  - If condition is false, break
  - This repeats until condition becomes false
- condition is checked *after* the block is executed
  - So-called post-test loop
  - At least *one* execution of statement
- statement can be a block of code

```
1  #include <stdio.h>
2
3  main() {
4    int counter = 5;
5
6    do {
7      printf("%d ",counter);
8    }
9    while (--counter>0);
10   printf("\n");
11 }
```

- Output:

  5 4 3 2 1

- counter-- in line 9 yields the output: 5 4 3 2 1 0



## Another example

```
#include <stdio.h>

main() {
  int x[2] = {0,1};
  int tmp = 0;
  int c = 0;

  printf("c=");
  scanf("%d",&c);

  printf("%d %d ",x[0],x[1]);

  do {
    tmp = x[0]+x[1];
    x[0] = x[1];
    x[1] = tmp;
    printf("%d ",tmp);
  }
  while(tmp<c);

  printf("\n");
}
```

- The Fibonacci sequence diverges towards infinity
  - $x_0 := 0$, $x_1 := 1$ and $x_{n+1} := x_{n-1} + x_n$ for $n \in \mathbb{N}$
- Task: Given a threshold $c \in \mathbb{N}$, compute the first sequence member satisfying $x_n > c$
- Input $c = 1000$ yields output:

  c=1000

  0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597



## break and continue

```
#include <stdio.h>

main() {
  int j = 0;
  int k = 0;

  for (j=0; j<4; ++j) {
    if (j%2 == 0) {
      continue;
    }
    for (k=0; k < 10; ++k) {
      printf("j=%d, k=%d\n",j,k);
      if (k > 1) {
        break;
      }
    }
  }
  printf("End: j=%d, k=%d\n",j,k);

}
```

- continue and break in statement of the loop
  - continue terminates the current iteration of the loop and continues with the next iteration
  - break terminates the loop and continues executing the code after the loop (if any)
- Output:

  j=1, k=0

  j=1, k=1

  j=1, k=2

  j=3, k=0

  j=3, k=1

  j=3, k=2

  End: j=4, k=2

## ‘as long as’ vs. ‘until’

```
1  #include <stdio.h>
2
3 main() {
4   int a = 200;
5   int b = 110;
6   int tmp = 0;
7
8   printf("gcd(%d,%d)=",a,b);
9
10  while (1) {
11    if (a == b) {
12      break;
13    }
14    else if ( a < b) {
15      tmp = a;
16      a = b;
17      b = tmp;
18    }
19    a = a-b;
20  }
21
22  printf("%d\n",b);
23 }
```

- `for` and `while` loops runs depending on `condition`
  - i.e., the loop runs as long as `condition` is true
- Algorithm implementation can be only based on a termination condition `done`
  - i.e., it is interrupted if `done` is true
  - i.e., `condition` = logical complement of `done`
- Easy realization via infinite loop and `break`
  - Condition in line 10 is always true
  - Termination only via `break` in line 12



# Simple error control

- Avoid runtime error
- Intentional error-caused termination

- `gcc -c`
- `gcc -c -Wall`
- `assert`
- `#include <assert.h>`



## Motivation

- Fact: All programmers make mistakes
  - A program is usually not correct the first time
- Most of the programming time is usually spent in finding and correcting own mistakes
- Efficiency in error search is a big distinction between *professionals* and *beginners*
- Syntax errors are easy to identify
  - The compiler reports even the line number
  - Check regularly the syntax while programming
    - `gcc -c name.c` generates only object code
    - `gcc -Wall name.c` enables compiler warnings
- Runtime error are more difficult to identify
  - The program works, but does not what it should
  - Sometimes the error is noticed after a long time

    ⇒ This can have undesired effects (see later)



## Good habits that help programmers to avoid mistakes

- Follow programming convention
  - Use meaningful and consistent names
    - e.g., for variables, functions, etc.
  - Use a consistent layout of code
    - Indentation
- Add explanations with comments in all relevant part of the code
  - e.g., non-obvious conditional statements
  - e.g., functions (specify aim, input, output)
- Break code into small functions
  - Each function expresses a logical action
    - Make testing and debugging easier
  - Check input for admissibility
    - Abortion if non-admissible
  - Ensure that the output is admissible
- Do not program all code at the same time
  - This is a usual mistake of beginners

## Library assert.h

```
1 #include <stdio.h>
2 #include <assert.h>
3
4 void test(int x, int y) {
5   assert(x<y);
6   printf("It holds x < y\n");
7 }
8
9 main() {
10   int x = 0;
11   int y = 0;
12
13   printf("x = ");
14   scanf("%d",&x);
15
16   printf("y = ");
17   scanf("%d",&y);
18
19   test(x,y);
20 }
```

- **Aim:** Termination with error message, whenever the function notices that input/output is not admissible
- `#include <assert.h>`
  - `assert(condition);` yields erroneous termination if `condition` does not hold
- **Input:**
  `x = 2`
  `y = 1`
- **Output:**
  `Assertion failed: (x<y), function test, file assert.c, line 5.`



## Example: Euclidean algorithm

```
// author: Dirk Praetorius
// last modified: 30.03.2017

// Euclidean algorithm to compute the gcd
// based on gcd(a,b) = gcd(a-b,b) for a>b
// and gcd(a,b) = gcd(b,a)

int euklid(int a, int b) {
  assert(a>0);
  assert(b>0);
  int tmp = 0;

  // reduction gcd(a,b) = gcd(a-b,b)
  // realized via integer division, until
  // b = 0. Then a==b, thus gcd = a

  while (b != 0) {
    tmp = a%b;
    a = b;
    b = tmp;
  }

  return a;
}
```

- `assert` ensures admissibility of input
  - i.e., it must be $a, b \in \mathbb{N}$



# Testing

- Motivation
- Quality assurance
- Types of test



## Mistakes can be costly

- Patriot Missile failure (February 1991)
  - Wrong treatment of rounding error
  - 28 deads, 100 injured
- Sinking of *Sleipner A platform* (August 1991)
  - Wrong finite element analysis
  - Damage 700 million Dollar
- Explosion of Ariane 5 rocket (June 1996)
  - Conversion `double` → `int`
  - Damage 500 million Dollar

## Quality assurance

- Simple, but true:
  - Fact 1: Software is made by humans
  - Fact 2: Making mistakes is human
  - Consequence: Software can include mistakes
- Desirable: Find errors before it is too late
- The later an error is identified, the more difficult becomes its correction
- Ideal work organization:
  - 1/3 of time for programming
  - 1/3 of time for testing
  - 1/3 of time for documenting
- In practice:
  - Most of the time for programming
  - Much less time for testing
  - Even less time for documenting ;-)



## Testing

- Testing is the process of executing a program with the intent of finding errors
  - G. Myers: *The art of sofware testing* (1979)
- A test is the comparison of the behavior of a program (what it does) with its desired behavior (what it should do)
- In practice, it is impossible to test all functions in a program with all possible combinations of input data
  - i.e., testing is by definition incomplete
- Problems with incomplete testing
  - Tests allows only to find mistakes
  - Tests do not provide a rigorous proof that the code has no errors
  - Test cases can theirselves be incorrect
- Important aspects in testing
  - Tests should consider 'realistic input'
  - Tests must be reproducible



## Types of test

- Structural tests (for each function)
  - Are all instructions executed or are there 'dead' parts of the code?
  - Analyze conditional statements!
    - e.g., check both occurrences (true/false) in `if ... else`
- Functional tests (for each function & program)
  - Does the function behave correctly for admissible input parameters? (i.e., is the result correct?)
  - Does the program (or a part of it) behave correctly? (i.e., is the result correct?)
  - Are non-admissible input parameters recognized?
  - Are limit cases and exceptions recognized and treated correctly?
  - What happens in the case of wrong input? (e.g., if the user makes a mistake)



## How to test?

- **Aim:** Is the function / program correct?
- Functional tests need test cases
  - (where the result is known)
- Tests should address conditional statements
  - Use parameters which lead to different outcomes in conditional statements
  - Try to test all outcome combinations
- Which cases are critical?
  - Inspect delicate parts of code
  - Be careful with type casting!
- Start early with testing
  - Test a function right after its implementation
  - Do not wait that the entire code is finished...
- Repeat all (!) tests after a change
  - i.e., test documenting is also important
- From now on, in the exercises you might find the following question:
  - How did you test your program?
  - Provide concrete examples to convince your tutor that you accurately tested your code

# Pointers

- Variables vs. Pointers
- Dereferencing
- Address-of operator &
- Dereference operator *
- Call by reference



## Variables vs. Pointers

- Variable = Symbolic name (identifier) of a storage location (memory address) containing some quantity of information (value) of a specific type (data type)
- Pointer = Variable containing the address of a storage location
- Dereferencing = Accessing the content of a storage location using the corresponding pointer



## Pointers in C

- Pointers and variables are closely related in C:
  - Variable `var` ⇒ `&var` corresponding pointer
  - Pointer `ptr` ⇒ `*ptr` corresponding variable
  - In particular, `*&var` = `var` and `&*ptr` = `ptr`
- Like any other variable, a pointer must be declared before use
- The declaration must include the type of the pointer, since `*ptr` must be a variable
  - `int* ptr;` declares `ptr` as pointer to an `int`
- As usual, simultaneous declaration and initialization are possible
  - `int var;` declares the variable `var` of type `int`
  - `int* ptr = &var;` declares `ptr` and assigns the address location of the variable `var` to it
    - In such assignments the type of the pointer and the variable must coincide
      - Usually the compiler gives a warning, e.g., `incompatible pointer type`
- The same holds for all other data types
- Some tasks are done more easily with pointers
- Other tasks cannot be done without pointers
  - e.g., dynamic memory allocation (see later)



## An elementary example

```
#include <stdio.h>

main() {
  int var = 1;
  int* ptr = &var;

  printf("a) var = %d, *ptr = %d\n",var,*ptr);

  var = 2;
  printf("b) var = %d, *ptr = %d\n",var,*ptr);

  *ptr = 3;
  printf("c) var = %d, *ptr = %d\n",var,*ptr);

  var = 47;
  printf("d) *(&var) = %d,",*(&var));
  printf("*&var = %d\n",*&var);

  printf("e) &var = %p\n", &var);
}
```

- `%p` placeholder for `printf` for addresses
- Output:

```
a) var = 1, *ptr = 1
b) var = 2, *ptr = 2
c) var = 3, *ptr = 3
d) *(&var) = 47,*&var = 47
e) &var = 0x7fff518baba8
```

## Call by reference in C

- In C, basic data types are passed to functions via *call by value*
  - e.g., `int`, `double`, pointers
- *Call by reference* can be realized with pointers

```
#include <stdio.h>

void test(int* y) {
  printf("a) *y=%d\n", *y);
  *y = 43;
  printf("b) *y=%d\n", *y);
}

main() {
  int x = 12;
  printf("c) x=%d\n", x);
  test(&x);
  printf("d) x=%d\n", x);
}
```

- Output:

  c) x=12

  a) *y=12

  b) *y=43

  d) x=43



## Summary

- Call by value
  - Functions receive values of variables as input parameters and copy them in local variables
  - Changes to the input parameters are not effective outside of the function
- Call by reference
  - Functions receive addresses of variables as input parameters
  - Changes to the input parameters are effective also outside of the function
- In C, for basic data types, the standard approach is call by value
- Call by reference can be realized with pointers
- Arrays are always passed to functions via call by reference

## Why call by reference?

- Functions in C can have at most 1 return value
- If a functions should have more return values...



## Example

```
#include <stdio.h>
#include <assert.h>
#define DIM 5

void scanVector(double input[], int dim) {
  assert(dim > 0);
  int j = 0;
  for (j=0; j<dim; ++j) {
    input[j] = 0;
    printf("%d: ",j);
    scanf("%lf",&input[j]);
  }
}

void determineMinMax(double vector[],int dim,
                     double* min, double* max) {
  int j = 0;
  assert(dim > 0);

  *max = vector[0];
  *min = vector[0];
  for (j=1; j<dim; ++j) {
    if (vector[j] < *min) {
      *min = vector[j];
    }
    else if (vector[j] > *max) {
      *max = vector[j];
    }
  }
}

main() {
  double x[DIM];
  double max = 0;
  double min = 0;
  scanVector(x,DIM);
  determineMinMax(x,DIM, &min, &max);
  printf("min(x) = %f\n",min);
  printf("max(x) = %f\n",max);
}
```

- `determineMinMax` *returns* maximum and minimum of a vector via call by reference



## Remarks about pointer declarations

- Spaces are ignored by the compilers
- `*` is applied only to the successive name
- In particular,
  - `int* pointer;`, `int *pointer;`, and `int*pointer;` are fully equivalent
  - `int* pointer, var;` declares a pointer to `int` and a variable of type `int`
  - `int *pointer1, *pointer2;` declares two pointers to `int`
- For ease of readability, try to avoid declarations of lists including variables and pointers

## Pointer & Arrays

- Declaration `int array[N];` automatically creates a pointer `array` of type `int*`
- `int array[];` and `int* array;` are equivalent declarations