Floating-point numbers

- Analytical binary representation
- Floating-point number system $\mathbb{F}(2, M, e_{\text{min}}, e_{\text{max}})$
- Ill-posed problems
- Computing errors and equality

- `float`, `double`

Floating-point representation 1/2

- Theorem: For each $x \in \mathbb{R}$ there exist
  - Sign $\sigma \in \{\pm 1\}$
  - Digits $a_k \in \{0, 1\}$
  - Exponent $e \in \mathbb{Z}$

  such that $x = \sigma \left(\sum_{k=1}^{\infty} a_k 2^{-k} \right) 2^e$
- The representation is not unique, e.g., $1 = \sum_{k=1}^{\infty} 2^{-k}$

Remarks

- The result holds for any basis $b \in \mathbb{N}_{\geq 2}$
  - The digits then satisfy $a_j \in \{0, 1, \dots, b-1\}$
- Decimal system $b = 10$ is very common
  - $47.11 = (4 \cdot 10^{-1} + 7 \cdot 10^{-2} + 1 \cdot 10^{-3} + 1 \cdot 10^{-4}) \cdot 10^2$
    - $a_1 = 4,\ a_2 = 7,\ a_3 = 1,\ a_4 = 1,\ e = 2$
- For $b = 2$, fractions are representable as finite sums if and only if the denominator is a power of 2:
  - In $\sum_{k=1}^{M} 2^{-k}$, the denominator is a power of 2
  - Uniqueness of the integer factorization
- e.g., no exact representation for $1/10$ with $b = 2$

Floating-point representation 2/2

- Theorem: For each $x \in \mathbb{R}$ there exist
  - Sign $\sigma \in \{\pm 1\}$
  - Digits $a_k \in \{0, 1\}$
  - Exponent $e \in \mathbb{Z}$

  such that $x = \sigma \left(\sum_{k=1}^{\infty} a_k 2^{-k} \right) 2^e$
- The representation is not unique, e.g., $1 = \sum_{k=1}^{\infty} 2^{-k}$

Floating-point numbers

- Floating-point number system $\mathbb{F}(2, M, e_{\text{min}}, e_{\text{max}}) \subset \mathbb{Q}$
  - Mantissa length $M \in \mathbb{N}$
  - Exponential barriers $e_{\text{min}} < 0 < e_{\text{max}}$
- $x \in \mathbb{F}$ has representation $x = \sigma \left(\sum_{k=1}^{M} a_k 2^{-k} \right) 2^e$ with
  - Sign $\sigma \in \{\pm 1\}$
  - Digits $a_j \in \{0, 1\}$ mit $a_1 = 1$
    - So-called normalized floating-point number
  - Exponent $e \in \mathbb{Z}$ with $e_{\text{min}} \leq e \leq e_{\text{max}}$
- The representation of $x \in \mathbb{F}$ is unique
- Digit a_1 must not be stored
  - Implicit first bit

Proof of the theorem

- Without loss of generality, assume $x \geq 0$
  - If not, multiply by $\sigma = -1$
- Let $e \in \mathbb{N}_0$ with $0 \leq x < 2^e$
- Without loss of generality, assume $x < 1$
  - If not, divide by $\sigma = -1$
- Construction of the digits a_j via bisection
- Claim: There exist digits $a_j \in \{0, 1\}$ such that

  $x_n := \sum_{k=1}^{n} a_k 2^{-k}$ satisfies $x \in [x_n, x_n + 2^{-n})$
- Induction base case: It holds that $x \in [0, 1)$
  - If $x \in [0, 1/2)$, choose $a_1 = 0$, i.e., $x_1 = 0$
  - If $x \in [1/2, 1)$, choose $a_1 = 1$, i.e., $x_1 = 1/2$
    - $x_1 = a_1/2 \leq x$
    - $x < (a_1 + 1)/2 = x_1 + 2^{-1}$
- Induction step: It holds that $x \in [x_n, x_n + 2^{-n})$
  - If $x \in [x_n, x_n + 2^{-(n+1)})$, choose $a_{n+1} = 0$, i.e., $x_{n+1} = x_n$
  - If $x \in [x_n + 2^{-(n+1)}, x_n + 2^{-n})$, choose $a_{n+1} = 1$
    - $x_{n+1} = x_n + a_{n+1} 2^{-(n+1)} \leq x$
    - $x < x_n + (a_{n+1} + 1) 2^{-(n+1)} = x_{n+1} + 2^{-(n+1)}$
- It follows that $|x_n - x| \leq 2^{-n}$, hence $x = \sum_{k=1}^{\infty} a_k 2^{-k}$

Arithmetic for floating-point numbers

- Result `Inf`, `-Inf` if overflow (oder `1./0.`)
- Result `NaN`, if not defined (z.B. `0./0.`)
- Arithmetic is approximate, not exact

Ill-posed problem

- A problem is numerically ill-posed, if small changes in the data lead to large changes in the result
  - e.g., does a triangle with given side lengths have a right angle?
  - e.g., is a given point located on a circle?
- Implementation meaningless, as result random!

Computing error

- In view of computing errors, one should *never* test the equality of two floating-point numbers
  - Check whether $|x-y|$ is small, rather than $x=y$
  - e.g., $|x-y| \leq \varepsilon \cdot \max\{|x|,|y|\}$ with $\varepsilon = 10^{-13}$

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

main() {
  double x = (116./100.)*100.;

  printf("x=%f\n",x);
  printf("floor(x)=%f\n",floor(x));

  if (x==116.) {
    printf("There holds x==116\n");
  }
  else {
    printf("Surprise, surprise!\n");
  }
}
```

- Output:

```
x=116.000000
floor(x)=115.000000
Surprise, surprise!
```


Variable types `float`, `double`

```
#include <stdio.h>
main() {
  double x = 2./3.;
  float y = 2./3.;
  printf("%f, %1.16e\n", x, x);
  printf("%f, %1.7e\n",y, y);
}
```

- Floating-point numbers constitute a finite subset of $\mathbb{Q}$
- `float` is usually *single precision* according to the IEEE-754 standard
  - $\mathbb{F}(2,24,-126,127) \to$ 4 bytes
  - ca. 7 significant decimal digits
- `double` is usually *double precision* according to the IEEE-754 standard
  - $\mathbb{F}(2,53,-1022,1023) \to$ 8 bytes
  - ca. 16 significant decimal digits
- Placeholders in `printf` and `scanf`

Data type	`printf`	`scanf`
`float`	`%f`	`%f`
`double`	`%f`	`%ℓf`

  - Placeholder `%1.16e` for floating-point representation
  - See `man 3 printf`
- Output:

```
0.666667, 6.6666666666666663e-01
0.666667, 6.6666669e-01
```


Structures

- Why structures?
- Members
- Point operator `.`
- Arrow operator `->`
- Shallow copy vs. deep copy

- `struct`
- `typedef`

Declaration of structures

- **Functions**
  - Callable group of statements that together perform a task
  - Abstraction (structured programming)
- **Structures**
  - Combination of variables of different types in a new data type
  - Abstraction with data
- **Example:** Management of SciProgMath students
  - The same data for each student

```
// Declaration of structure
struct _Student_ {
  char* firstname; // First name
  char* lastname;  // Last name
  int studentID;   // Student ID
  int studiesID;   // Studies ID
  double test;     // Points in final test
  double exercise; // Points in exercises
};

// Declaration of corresponding data type
typedef struct _Student_ Student;
```

- Semicolon after structure declaration block
- Creation of new variable type `Student`

Structures & Members

- Data types of a structure are called members
- Access to members with point operator
  - `var` variable of type `Student`
  - e.g., member `var.firstname`

```
// Declaration of structure
struct _Student_ {
  char* firstname; // First name
  char* lastname;  // Last name
  int studentID;   // Student ID
  int studiesID;   // Studies ID
  double test;     // Points in final test
  double exercise; // Points in exercises
};

// Declaration of corresponding data type
typedef struct _Student_ Student;

main() {
  Student var;
  var.firstname = "Max";
  var.lastname = "Mustermann";
  var.studentID = 0;
  var.studiesID = 680;
  var.test = 25.;
  var.exercise = 35.;
}
```


Remarks on structures

- Originally, the C standard did not allow to use
  - Structures as input parameter of a function
  - Structures as output parameter of a function
  - Assignment operator (=) for an entire structure
- In the meantime, it is allowed, however:
  - Pass structures to functions dynamically (via pointers)
  - Write assignment (= copy) by yourself
  - Assignment (=) creates a so-called *shallow copy*
- Shallow copy:
  - Only the first level is copied
  - i.e., values for basic variables
  - i.e., addresses for pointers
  - Moreover: Copy has the same dynamic data
- Deep copy:
  - All levels of a structure are copied
  - Plus: Copy of the dynamic data

Structures: Memory allocation

- Create also the functions
  - `newStudent`: Memory allocation and initialization
  - `freeStudent`: Memory deallocation
  - `cloneStudent`: Complete copy of the full structure including the dynamic data e.g., member `firstname` (so-called *deep copy*)
  - `copyStudent`: Copy of the first level excluding the dynamic data (so-called *shallow copy*)

```
Student* newStudent() {
  Student* pointer = malloc(sizeof(Student));
  assert( pointer != NULL);

  (*pointer).firstname = NULL;
  (*pointer).lastname = NULL;
  (*pointer).studentID = 0;
  (*pointer).studiesID = 0;
  (*pointer).test = 0.;
  (*pointer).exercise = 0.;

  return pointer;
}
```

Structures & Arrow operator

- In the program, pointer is of type Student*
- Access to members, e.g., (*pointer).firstname
  - Better syntax pointer->firstname
- Structures never static, always dynamic
  - Use directly student for type Student*
- Better implementation of the function newStudent below

```
// Declaration of structure
struct _Student_ {
  char* firstname; // First name
  char* lastname;  // Last name
  int studentID;   // Student ID
  int studiesID;   // Studies ID
  double test;     // Points in final test
  double exercise; // Points in exercises
};

// Declaration of corresponding data type
typedef struct _Student_ Student;

// allocate and initialize new student
Student* newStudent() {
  Student* student = malloc(sizeof(Student));
  assert(student != NULL);

  student->firstname = NULL;
  student->lastname = NULL;
  student->studentID = 0;
  student->studiesID = 0;
  student->test = 0.;
  student->exercise = 0.;

  return student;
}
```


Structures: Memory deallocation

- Deallocation of a dynamic variable of type Student
- Be careful: Deallocate the dynamically allocated memory before deallocating the pointer to the structure

```
// free memory allocation
Student* delStudent(Student* student) {
  assert(student != NULL);

  if (student->firstname != NULL) {
    free(student->firstname);
  }

  if (student->lastname != NULL) {
    free(student->lastname);
  }

  free(student);
  return NULL;
}
```


Shallow Copy

- Copy of a dynamic variable of type Student
  - Copy of only the first level of the structure excluding the dynamically allocated memory

```
// shallow copy of student
Student* copyStudent(Student* student) {
  Student* copy = newStudent();
  assert(student != NULL);

  // Watch out! Pointer!
  copy->firstname = student->firstname;
  copy->lastname = student->lastname;

  // Copy of the simple data
  copy->studentID = student->studentID;
  copy->studiesID = student->studiesID;
  copy->test = student->test;
  copy->exercise = student->exercise;

  return copy;
}
```


Deep Copy

- Copy of a dynamic variable of type Student
  - Copy of all levels of the structure including the dynamically allocated memory
- Be careful: Copy also the member with dynamically allocated memory

```
// deep copy of student
Student* cloneStudent(Student* student) {
  Student* copy = newStudent();
  int length = 0;
  assert( student != NULL);

  if (student->firstname != NULL) {
    length = strlen(student->firstname)+1;
    copy->firstname = malloc(length*sizeof(char));
    assert(copy->firstname != NULL);
    strcpy(copy->firstname, student->firstname);
  }


  if (student->lastname != NULL) {
    length = strlen(student->lastname)+1;
    copy->lastname = malloc(length*sizeof(char));
    assert(copy->lastname != NULL);
    strcpy(copy->lastname, student->lastname);
  }

  copy->studentID = student->studentID;
  copy->studiesID = student->studiesID;
  copy->test = student->test;
  copy->exercise = student->exercise;

  return copy;
}
```

Arrays of structures

- Aim: Generate array with SciProgMath students
- No static arrays, dynamic arrays
  - Student data are of type `Student`
  - Hence, they are managed with type `Student*`
  - Hence, an array of type `Student**` is needed

```c
// Declare array
Student** participant=malloc(N*sizeof(Student*));

// Allocate memory for participants
for (j=0; j<N; ++j){
  participant[j] = newStudent();
}
```

- Access to members as before
  - `participant[j]` has type `Student*`
  - Hence, e.g., `participant[j]->firstname`

Nesting of structures

```c
struct _Address_ {
  char* street;
  char* number;
  char* city;
  char* zip;
};
typedef struct _Address_ Address;

struct _Employee_ {
  char* firstname;
  char* lastname;
  char* title;
  Address* home;
  Address* office;
};
typedef struct _Employee_ Employee;
```

- Organize data of employees
  - Name, private address, work address
- For `employee` of type `Employee*`
  - `employee->home` is a pointer to `Address`
  - Hence, e.g., `employee->home->city`
- Be careful with allocating, deallocating, copying

Structures & mathematics

- Structures for mathematical objects:
  - Vectors in $\mathbb{R}^n$

Structures and vectors

```c
#ifndef _STRUCT_VECTOR_
#define _STRUCT_VECTOR_

#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <math.h>

// declaration of new data type Vector
typedef struct _Vector_ {
  int n;          // Dimension
  double* entry;  // Vector coefficients
} Vector;

// Allocate and initialize new vector of length n
Vector* newVector(int n);

// free storage of allocated vector and return NULL
Vector* delVector(Vector* X);

// return length of a vector
int getVectorN(Vector* X);

// return coefficient Xi of vector X
double getVectorEntry(Vector* X, int i);

// assign new value to coefficient Xi of vector X
void setVectorEntry(Vector* X, int i, double Xi);

// some example functions...
Vector* inputVector();
double normVector(Vector* X);
double productVector(Vector* X, Vector* Y);

#endif
```

- Data type to store $x \in \mathbb{R}^n$
  - Dimension n of type `int`
  - Array coefficients x_j to store double

Allocate a vector

- The vector length $n \in \mathbb{N}$ is passed to the function
- Structure allocation, assignment of dimension n
- Allocation and initialization of the vector array

```
Vector* newVector(int n) {
  int i = 0;
  Vector* X = NULL;

  assert(n > 0);

  X = malloc(sizeof(Vector));
  assert(X != NULL);

  X->n = n;
  X->entry = malloc(n*sizeof(double));
  assert(X->entry != NULL);

  for (i=0; i<n; ++i) {
    X->entry[i] = 0;
  }
  return X;
}
```

Deallocate a vector

- Deallocate array
- Deallocate structure
- Return value NULL

```
Vector* delVector(Vector* X) {
  assert(X != NULL);
  free(X->entry);
  free(X);

  return NULL;
}
```


Access to structures

- Good programming style (unfortunately not very common): No direct access to structure members
- Better approach:
  - Write set and get functions for each member
  - So-called *mutator functions*

```
int getVectorN(Vector* X) {
  assert(X != NULL);
  return X->n;
}

double getVectorEntry(Vector* X, int i) {
  assert(X != NULL);
  assert((i >= 0) && (i < X->n));
  return X->entry[i];
}

void setVectorEntry(Vector* X, int i, double Xi){
  assert(X != NULL);
  assert((i >= 0) && (i < X->n));
  X->entry[i] = Xi;
}
```

- Writing data is not allowed without set!
- Reading data is not allowed without get!
- This approach is more compatible with later changes of the data structure
- This approach helps to avoid data inconsistencies (and very often also runtime errors)

Example: Read vector

```
Vector* inputVector() {

  Vector* X = NULL;
  int i = 0;
  int n = 0;
  double input = 0;

  printf("Dimension des Vektors n=");
  scanf("%d",&n);
  assert(n > 0);

  X = newVector(n);
  assert(X != NULL);

  for (i=0; i<n; ++i) {
    input = 0;
    printf("x[%d]=",i);
    scanf("%lf",&input);
    setVectorEntry(X,i,input);
  }

  return X;
}
```

- Read $n \in \mathbb{N}$ and a vector $x \in \mathbb{R}^n$ from the keyboard

Example: Euclidean norm

```
double normVector(Vector* X) {

  double Xi = 0;
  double norm = 0;
  int n = 0;
  int i = 0;

  assert(X != NULL);

  n = getVectorN(X);

  for (i=0; i<n; ++i) {
    Xi = getVectorEntry(X,i);
    norm = norm + Xi*Xi;
  }
  norm = sqrt(norm);

  return norm;
}
```

- Compute $\|x\| := \left(\sum_{j=1}^{n} x_j^2\right)^{1/2}$ for $x \in \mathbb{R}^n$

Example: Scalar product

```
double productVector(Vector* X, Vector* Y) {

  double Xi = 0;
  double Yi = 0;
  double product = 0;
  int n = 0;
  int i = 0;

  assert(X != NULL);
  assert(Y != NULL);

  n = getVectorN(X);
  assert(n == getVectorN(Y));

  for (i=0; i<n; ++i) {
    Xi = getVectorEntry(X,i);
    Yi = getVectorEntry(Y,i);
    product = product + Xi*Yi;
  }

  return product;
}
```

- Compute $x \cdot y := \sum_{j=1}^{n} x_j y_j$ for $x, y \in \mathbb{R}^n$