Function pointers

- Declaration
- Everything is a pointer!

Function pointers

- Function calls can be realized with pointers
- Declaration of a function pointer:
  - <return type> (*pointer)(<input>); declares a pointer pointer for functions with parameters <input> and return value of type <return type>
- If the address of a function is assigned to a function pointer, they must have the same structure
  - Same return value
  - Same list of input parameters
- Call of a function via a pointer is a normal function call

Elementary example

```
1 #include <stdio.h>
2
3 void output1(char* string) {
4   printf("*%s*\n",string);
5 }
6
7 void output2(char* string) {
8   printf("#%s#\n",string);
9 }
10
11 main() {
12   char string[] = "Hello World";
13   void (*output)(char* string);
14
15   output = output1;
16   output(string);
17
18   output = output2;
19   output(string);
20 }
```

- Declaration of a function pointer in line 13
- Assignment to function pointer in lines 15 and 18
- Function calls via pointer in lines 16 and 19
- **Output:**

```
*Hello World*
#Hello World#
```


Bisection method

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

double bisection(double (*fct)(double x),
                 double a, double b, double tol) {
  double m = 0;
  double fa = 0;
  double fm = 0;

  assert(a < b);
  fa = fct(a);
  assert(fa*fct(b) <= 0);

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

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

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

  double x = bisection(f,a,b,tol);
  printf("Approximate zero x=%1.15e\n",x);
}
```

- Approximation of zeros of $f(x) = x^2 + e^x - 2$

Basic data types

- Arrays & Pointers
- sizeof

Basic data types

- Basic data types in C
  - Data type for characters (e.g., letters)
    - char
  - Data type for integers
    - int
  - Data types for floating point numbers
    - float
    - double
- Pointers area treated as basic data types
- Modifiers can be applied to basic types, e.g.,
  - short/long modifies the amount of storage allocated for the variable
    - e.g., long double is an extended precision floating-point number
    - Amount of storage depends on compiler
- Declaration and use as so far
- Arrays and pointers of basic types possible
- More details later!

The command sizeof

```
#include <stdio.h>

void printSizeOf(double vector[]) {
  printf("sizeof(vector) = %d\n",sizeof(vector));
}

main() {
  int var = 43;
  double array[12];
  double* ptr = array;

  printf("sizeof(var) = %d\n",sizeof(var));
  printf("sizeof(double) = %d\n",sizeof(double));
  printf("sizeof(array) = %d\n",sizeof(array));
  printf("sizeof(ptr) = %d\n",sizeof(ptr));
  printSizeOf(array);
}
```

- If var is a variable with basic data type, sizeof(var) returns the size of the variable in bytes
- If type is a data type, sizeof(type) returns the size of a variable of this type in bytes
- If array is a *local static array*, sizeof(array) returns the size of the array in bytes
- Internally ptr and array are two pointers to double and contain (= point to) the same storage location
- Output:

```
sizeof(var) = 4
sizeof(double) = 8
sizeof(array) = 96
sizeof(ptr) = 8
sizeof(vector) = 8
```


Functions

- Basic data types are passed to functions via call by value
- The return value of a function can be only empty (void) or a basic data type

Arrays do not exist in C!

- Strictly speaking, arrays do not exist in C!
- Declaration int array[N];
  - creates a pointer array of type int*
  - allocates storage of size N times the size of an int starting at the storage location with address stored in array
  - i.e., array contains the address of array[0]
- This explains why arrays are always passed with call by reference to functions
- What is really happening is that the function receives a pointer to the storage location in which the array is stored

Runtime error

```
 1 #include <stdio.h>
 2 #include <assert.h>
 3
 4 double* scanVector(int length) {
 5   assert(length > 0);
 6   double vector[length];
 7   int j = 0;
 8   for (j=0; j<length; ++j) {
 9     vector[j] = 0;
10     printf("vector[%d] = ",j);
11     scanf("%lf",&vector[j]);
12   }
13   return vector;
14 }
15
16 main() {
17   double* x;
18   int j = 0;
19   int dim = 0;
20
21   printf("dim = ");
22   scanf("%d",&dim);
23
24   x = scanVector(dim);
25
26   for (j=0; j<dim; ++j) {
27     printf("x[%d] = %f\n",j,x[j]);
28   }
29 }
```

- Syntax of this program is fine
- Problem: The storage for `x` is declared inside the function and is therefore lost after the function call in line 24, i.e., the pointer in lines 6 and 13 points to some random unknown location
- Workaround: call by reference (as before) or manual memory allocation (discussed now)

Dynamic vectors

- Static & dynamic vectors
- Vectors & Pointers
- Dynamic memory allocation

- `stdlib.h`
- `NULL`
- `malloc`, `realloc`, `free`

- `#ifndef ... #endif`

Static vectors

- `double array[N];` declares a static vector `array` of length `N` with `double` coefficients
  - Indexing `array[j]` with $0 \leq j \leq N-1$
  - `array` is internally of type `double*`
    - It contains the address of `array[0]`
    - It is a so-called base pointer
  - The length `N` cannot be changed during the program execution

- Functions cannot determine the length `N`
  - It must be passed as an input parameter

Memory allocation

- Manual memory allocation allows for vector with dynamic length

- Inclusion of the standard library `stdlib.h`
  - `#include <stdlib.h>`
  - Commands `malloc`, `free`, `realloc`

- `pointer = malloc(N*sizeof(type));`
  - Memory allocation for a vector of length `N` with coefficients of type `type`
    - `malloc` wants input in bytes → `sizeof`
  - `pointer` must be of type `type*`
    - `pointer` receives the address of the first coefficient `pointer[0]` (base pointer)

- **Common (runtime) error:** Forgetting `sizeof`!
- **Watch out!** Allocated memory is not initialized!

- **Convention:** Pointers without memory are initialized with the value `NULL`
  - This leads immediately to an error if one tries to access its (non-existing) storage location

- `malloc` returns `NULL`, if the allocation is unsuccessful
  - i.e., the memory could not be allocated

Memory deallocation

- free(pointer) deallocates a previously allocated memory
  - pointer must have been output of malloc
- **Watch out!**
  - The memory is deallocated, but the pointer still exists
  - Successive access leads to runtime error
- **Watch out!**
  - Do not forget to deallocate the memory and set the corresponding pointer to NULL

Example

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

double* scanVector(int length) {
  int j = 0;
  double* vector = NULL;
  assert(length > 0);

  vector = malloc(length*sizeof(double));
  assert(vector != NULL);
  for (j=0; j<length; ++j) {
    vector[j] = 0;
    printf("vector[%d] = ",j);
    scanf("%lf",&vector[j]);
  }
  return vector;
}

void printVector(double* vector, int length) {
  int j = 0;
  assert(vector != NULL);
  assert(length > 0);

  for (j=0; j<length; ++j) {
    printf("vector[%d] = %f\n",j,vector[j]);
  }
}

main() {
  double* x = NULL;
  int dim = 0;

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

  x = scanVector(dim);
  printVector(x,dim);

  free(x);
  x = NULL;
}
```


Dynamic vectors

- pointer = realloc(pointer,Nnew*sizeof(type))
  - Change of memory allocation
    - Additional allocation if $\mathtt{Nnew} > \mathtt{N}$
    - Reduction of allocated memory if $\mathtt{Nnew} < \mathtt{N}$
  - Former content remains (if possible)
  - Return value NULL if reallocation unsuccessful

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

main() {
  int N = 5;
  int Nnew = 10;
  int j = 0;

  int* array = malloc(N*sizeof(int));
  assert(array != NULL);
  for (j=0; j<N; ++j){
    array[j] = j;
  }

  array = realloc(array,Nnew*sizeof(int));
  assert(array != NULL);
  for (j=N; j<Nnew; ++j){
    array[j] = 10*j;
  }

  for (j=0; j<Nnew; ++j){
    printf("%d ",array[j]);
  }
  printf("\n");
  free(array);
  array = NULL;
}
```

- Output:

```
0 1 2 3 4 50 60 70 80 90
```


Remarks

- Memorize and do not change base pointer (= output of malloc or realloc)
  - Necessary for error-free free and realloc
- Do not forget sizeof with malloc and realloc
  - Type of base pointer must agree with sizeof
- Check that the output of malloc and realloc is not NULL, e.g., with assert
  - To ensure that the allocation was successful
- Allocated memory is not initialized
  - Initialization right after allocation
- Memorize the length of a dynamic array
  - It cannot be determined by the program
- Deallocate unneeded memory
  - Deallocate before the end of the block }, otherwise the base pointer is gone
- Set pointers without memory to NULL
  - Error message, if the program tries to access
- Never use realloc and free on static arrays
  - Runtime error, as deallocation is automatically done by the compiler

Vector library

- Subdivision of the source code into more files
- Precompiler, Compiler, Linker
- Object code

- `gcc -c`
- `make`

Source code subdivision

- Subdivide long source codes into more files
- Advantage:
  - More clear structure
  - Creation of libraries
    - Re-use of old code
    - Help to avoid mistakes
- `gcc name1.c name2.c ...`
  - Creation of *one* executable
  - Ordering of the code not important
  - Similar to `gcc all.c`
    - If `all.c` contains the entire source code
  - Function names must be unambiguous
  - `main()` can appear only once

Precompiler, Compiler & Linker

- The compilation process consists of four steps
  - 1. Preprocessing
  - 2. Compiling
  - 3. Assembling → Object code
  - 4. Linking → Executable
- Library = Precompiled object code with corresponding header file
- Standard linker in Unix is `ld`
- `gcc -c name.c` performs the compilation process until Step 3
  - Creation of object code `name.o`
  - Also good for debugging of syntax error
- To compile a file with additional object code:
  - `gcc name.c bib1.o bib2.o ...`
  - `gcc name.o bib1.o bib2.o ...`
  - Ordering and number of files do not count
- Aim: Creation library
  - Useful to reduce compilation time and mistakes

A first library

```
#ifndef _DYNAMICVECTORS_
#define _DYNAMICVECTORS_

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

// allocate + initialize dynamic double vector of length n
double* mallocVector(int n);

// free a dynamic vector and set the pointer to NULL
double* freeVector(double* vector);

// extend dynamic double vector and initialize new entries
double* reallocVector(double* vector, int n, int nnew);

// allocate dynamic double vector of length n and read
// entries from keyboard
double* scanVector(int n);

// print dynamic double vector of length n to shell
void printVector(double* vector, int n);

#endif
```

- Header file `dynamicvectors.h` of the library
  - All function signatures
  - Comments about function functionalities
- The header file starts with
  `#ifndef NAME`
  `#define NAME`
- The header file ends with
  `#endif`
- It allows for multiple inclusions and avoids multiple declarations

Source code (1/2)

```
1  #include "dynamicvectors.h"
2
3  double* mallocVector(int n) {
4    int j = 0;
5    double* vector = NULL;
6    assert(n > 0);
7
8    vector = malloc(n*sizeof(double));
9    assert(vector != NULL);
10
11   for (j=0; j<n; ++j) {
12     vector[j] = 0;
13   }
14   return vector;
15 }
16
17 double* freeVector(double* vector) {
18   free(vector);
19   return NULL;
20 }
21
22 double* reallocVector(double* vector, int n, int nnew) {
23   int j = 0;
24   assert(vector != NULL);
25   assert(n > 0);
26   assert(nnew > 0);
27
28   vector = realloc(vector,nnew*sizeof(double));
29   assert(vector != NULL);
30   for (j=n; j<nnew; ++j) {
31     vector[j] = 0;
32   }
33   return vector;
34 }
```

- Inclusion of header files (line 1)
  - `#include "..."` including the full path
  - `#include <...>` for the standard directory

Source code (2/2)

```
double* scanVector(int n) {
  int j = 0;
  double* vector = NULL;
  assert(n > 0);

  vector = mallocVector(n);
  assert(vector != NULL);

  for (j=0; j<n; ++j) {
    printf("vector[%d] = ",j);
    scanf("%lf",&vector[j]);
  }
  return vector;
}

void printVector(double* vector, int n) {
  int j = 0;
  assert(vector != NULL);
  assert(n > 0);

  for (j=0; j<n; ++j) {
    printf("%d: %f\n",j,vector[j]);
  }
}
```


Main program

```
#include "dynamicvectors.h"

main() {
  double* x = NULL;
  int n = 10;
  int j = 0;
  x = mallocVector(n);
  for (j=0; j<n; ++j) {
    x[j] = j;
  }
  x = reallocVector(x,n,2*n);
  for (j=n; j<2*n; ++j) {
    x[j] = 10*j;
  }
  printVector(x,2*n);
  x = freeVector(x);
}
```

- Main program includes the header of the library
- Compilation via
  - `gcc -c dynamicvectors.c`
    - Creation of object code `dynamicvectors.o`
  - `gcc dynamicvectors_main.c dynamicvectors.o`
    - Creation of executable `a.out`

Static libraries and make

```
1 exe : main.o dynamicvectors.o
2         gcc -o exe main.o dynamicvectors.o
3
4 main.o : dynamicvectors_main.c dynamicvectors.h
5         gcc -c dynamicvectors_main.c -o main.o
6
7 dynamicvectors.o : dynamicvectors.c dynamicvectors.h
8         gcc -c dynamicvectors.c
```

- UNIX command `make` allows to treat code dependencies automatically
  - Automatization saves time in compiling
  - New object code is created only for changed source code
- Calling `make` builds targets specified in `Makefile`
- Calling `make -f filename` uses `filename` instead
- The file includes dependencies and commands, e.g.,
  - Line 1 = Dependencies (without indentation)
    - The file `exe` depends on...
  - Line 2 = Commands (one tab-indentation)
    - If `exe` is older than its dependencies, the command is executed (and only in that case!)

Strings

- Static & dynamic strings
- "..." vs. '...'
- string.h

Strings

- Strings = Arrays of char
  - Static: char array[N];
    - N = Static length
    - Simultaneous declaration and initialization
      char array[] = "text";
  - Dynamic: Type char*
    - Same approach as before for vectors
- Define static strings with double quotation marks "..."
- Access to single characters with single quotation marks '...'
- Access to sub-strings not possible
- Strings end with the null character \0
  - Take this into account when determining the length of dynamic strings
    - Allocate 1 byte more
    - Do not forget \0 at the end
  - For static strings this is automatic
    i.e., effective length N+1 and array[N]='\0'
- Static string can also be passed to functions (in double quotation marks)
  - e.g., printf("Hello World!\n");

Functions for string manipulation

- Important functions in stdio.h
  - sprintf: Conversion variable → string
  - sscanf: Conversion string → variable
- Several functions in stdlib.h, e.g.,
  - atof: Conversion string → double
  - atoi: Conversion string → int
- Many functions in string.h, e.g.,
  - strchr, memchr: Search for char in a string
  - strcmp, memcmp: Compare two strings
  - strcpy, memcpy: Copy strings
  - strlen: Determine string length (without \0)
- Include header files with #include <name>!
- Be careful when working with strings: Functions cannot know whether the storage allocated for the output string is sufficient (→ Runtime error!)

Example

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3 #include <string.h>
 4 #include <assert.h>
 5
 6 char* stringCopy(char* source) {
 7   int length = 0;
 8   char* result = NULL;
 9   assert(source != NULL);
10
11   length = strlen(source);
12   result = malloc((length+1)*sizeof(char));
13   strcpy(result,source);
14   return result;
15 }
16
17 main() {
18   char* string1 = "Hello World?";
19   char* string2 = stringCopy(string1);
20   string2[11] = '!';
21   printf("%s %s\n",string1,string2);
22 }
```

- Output:
  ```
  Hello World?  Hello World!
  ```
- Declaration and initialization of strings with double quotation marks "..." creates a static string with null character at the end (line 18)
- Access to single characters of a string with single quotation marks '...' (line 20)
- Placeholder for strings in printf is %s (line 21)

Integers

- Bits, bytes etc.
- `short`, `int`, `long`
- `unsigned`

Storage units

- 1 bit = 1 b = Smallest unit, storage of 0 or 1
- 1 byte = 1 B = Collection of 8 bits
- 1 kilobyte = 1 KB = 1024 bytes
- 1 megabyte = 1 MB = 1024 KB
- 1 gigabyte = 1 GB = 1024 MB
- 1 terabyte = 1 TB = 1024 GB

Storage of numbers

- The amount of bytes used to store numbers depends on the data type
- Main consequence:
  - For each data type, the number of representable numbers is finite
  - For each data type, there always exist the smallest and the largest representable numbers

Integers

- With n bits, it is possible to represent 2^n integers
- One usually considers
  - Either all integers in $[0, 2^n - 1]$
  - Or all integers in $[-2^{n-1}, 2^{n-1} - 1]$

Integer arithmetic

- Exact arithmetic in $[\texttt{intmin}, \texttt{intmax}]$
- Overflow: Result of a computation $>$ `intmax`
- Underflow: Result of a computation $<$ `intmin`
- Integer arithmetic is usually modular arithmetic
  - i.e., numbers "wrap around" upon reaching the extrema
    - `intmax` $+ 1$ returns `intmin`
    - `intmin` $- 1$ returns `intmax`
  - Not defined in C standard

```
#include <stdio.h>

main() {
  int j = 0;
  int n = 8*sizeof(int); // number bits per int
  int min = 1;

  // compute 2^(n-1)
  for (j=1; j<n; ++j) {
    min = 2*min;
  }
  printf("n=%d, min=%d, max=%d\n",n,min,min-1);
}
```

- Determine $[-2^{n-1}, 2^{n-1} - 1]$ for $n = 32$
- Output:

```
n=32, min=-2147483648, max=2147483647
```


2 billions are not so much!

```
#include <stdio.h>

main() {
  int n = 1;
  int factorial = 1;

  do {
    ++n;
    factorial = n*factorial;
    printf("n=%d, n!=%d\n",n,factorial);
  } while (factorial < n*factorial);

  printf("n=%d, n!>%d\n",n+1,n*factorial);
}
```

- Output:

```
n=2, n!=2
n=3, n!=6
n=4, n!=24
n=5, n!=120
n=6, n!=720
n=7, n!=5040
n=8, n!=40320
n=9, n!=362880
n=10, n!=3628800
n=11, n!=39916800
n=12, n!=479001600
n=13, n!=1932053504
n=14, n!>-653108224
```

Variable types short, int, long

- n bits $\Rightarrow 2^n$ integers
- In C, short, int, long have sign
  - i.e., integers in $[-2^{n-1}, 2^{n-1}-1]$
- Integers ≥ 0 with additional modifier unsigned
  - i.e., integers in $[0, 2^n - 1]$
  - e.g., unsigned int var1 = 0;
- It always holds short $\leq$ int $\leq$ long
  - Standard sizes: 2 bytes (short), 4 bytes (int)
  - It often holds int $=$ long
- Placeholders in printf and scanf

Data type	printf	scanf
short	%d	
int	%d	%d
unsigned short	%u	
unsigned int	%u	%u

Variable type char

- The size of a char is usually 1 byte
- Characters are interally coded as integers
  - Usually ASCII code
  - See, e.g., https://www.asciitable.com/
- ASCII code of a character can be obtained via single quotation marks
  - char var = 'A'; assigns the ASCII code of A to var
- Placeholder of a character for printf and scanf
  - %c as character
  - %d as integer

```
#include <stdio.h>

main() {
  char var = 'A';

  printf("sizeof(var) = %d\n", sizeof(var));
  printf("%c %d\n",var,var);
}
```

- Output:

```
sizeof(var) = 1
A 65
```