Datastructures / Programming in C

Andy D. Pimentel

System and Network Engineering Lab
a.d.pimentel@uva.nl
The course

- In first 3 weeks, 3 lectures/week. Hereafter, 2 lectures/week.
- We might not need the full period though
- Programming in C (Andy Pimentel) / theory of datastructures (Raphael Poss)
- Invited lecture on (tools for) debugging (Simon Polstra)
- Not an exhaustive course on the C programming language (the ’man’ command is your best friend!)
The course (cont’ed)

- 4 practical assignments (66.6% of grade) + written examination (33.3% of grade)
  - Both parts need to be ≥ 5
  - Weights of lab assignments: 1 : 10%, 2 : 30%, 3 : 30%, 4 : 30%
  - All assignments need a minimum score of 4.5!
  - Every assignment consists of a base part (max. grade: 7.5) and a bonus part

- Deadlines: Friday 5 Feb. (!!), Wed. 17 Feb. (code) + Friday 19 Feb. (report), Friday 4 March, Friday 18 March
Outline

- From Java to C
  - Some similarities and differences
- Overview of the C language
  - Assumes basic programming skills (Programmeren I)
  - Topics that will be covered
    - The pointer concept
    - Functions
      - Call by value
      - Call by reference
    - Non-primitive data types
      - Structs & arrays
      - Strings
Topics that will be covered (cont’d)
- Memory management
- I/O functions

Introduction to datastructures in C
- Single linked lists (stack, queue, etc.)
- Double linked lists
- Binary trees
Literature on C

Some suggestions

Online Literature on C

Some suggestions

- See website
- EssentialC pdf at http://cslibrary.stanford.edu/101/
  http://cslibrary.stanford.edu/102/ for linked list basics
- Online version of The C Book by Banahan, Brady and Doran: http://publications.gbdirect.co.uk/c_book/
From Java to C: some background

Some history

- 1960: ALGOL, the ancestor of all imperative languages
- 1963: CPL, Cambridge University
- 1967: BCPL, Martin Richard
- 1970: B, Ken Thompson
- 1978: C, Brian Kernighan and Dennis Ritchie (K&R) "The C programming Language" (Prentice Hall). This definition is now known as Traditional C.
Need for standardization arose during the 80’s

- 1989
  - ANSI C, The American National Standard
  - ISO C, The International Standard

Since then, we have had C95 (1995), C99 (1999) and now C11 (2011)

- All C compilers support C89 (ANSI C or ISO C), but C99 and C11 are only partly supported and/or only by some compilers
From Java to C: some background (cont’d)

In the 80’s, C was extended in C++ (Bjarne Stroustrup)
- Better type-checking
- Support for object-oriented programming

Java started as language for software for consumer electronics (embedded systems) : Oak by James Gosling (1990)
- Strong reliability!
- At that time, Java miserably failed for embedded systems
- With the emergence of the Web, Java "skyrocketed"
- Now Java has finally become big in the embedded realm (Android)
From Java to C: some background (cont’d)

Java builds on C, not on C++

- Java inherited much of C’s syntax
- Many important language features have the same semantics in both languages

So, what are the main differences between Java and C? Well, to be very crude:

- C is not OO
- C has a "lower abstraction level" (it’s simply closer to assembly)
Differences between Java and C

The C language does not or hardly have (the notion of)

- classes and interfaces, and their (parent-child) relationship
- explicit error and exception handling
- the "complicated" and explicit way of performing I/O
- threads (no support for concurrent programming in the language)...in C11 there is some support though
- support for "internationalization" (locale and unicode)
Similarities between Java and C

- Both Java and C feature **primitive datatypes** like:
  - bool (for C: as of C99)
  - char, short, int, and long
  - float and double

- A Java class might be compared with a **struct** in C (to be discussed)

- A Java interface might be compared with a **header file** (.h file) in C, and the class that implements the interface with the corresponding source file (.c file) in C (to be discussed)

- The set of operators is very similar for both languages

- Language constructs like **if, if then else, switch, while, do, for, break, return etc. follow a similar syntax**
Differences between Java and C (cont’d)

An important difference: in Java, the memory is completely hidden to the programmer

- Primitive data are stored in memory but can only be accessed using their variable-names

- Consider the declaration `int n1 = 13, n2 = 128;`
  - The variables `n1` and `n2" refer" to two different memory locations where the values 13 and 128 can be found
  - After `n1 = n2`, the variables `n1` and `n2` still refer to different locations, which now both store 128

- It is impossible to get information about locations of data!
- It is impossible to manipulate these locations explicitly!
In Java, all other data are contained in objects (instantiations of classes) that can be accessed through reference variables

- Consider the decl. `Point p1, p2, p3, p4;` and `p1 = new Point(1,2); p2 = p1; p3 = new Point(3,4); p4 = p2;`
- `Refs p1, p2 and p4 refer to a single Point object with value (1,2)`
- `Ref p3 refers to another Point object with value (3,4)`
- `After p3 = p4, all references point at the object with value (1,2) and the object with value (3,4) has become inaccessible forever`

It is often said that "Java has no pointers", but it has reference variables...which are pointers
References and pointers

The value of a reference variable (pointer) is a memory address. But, in Java

- pointer values are completely encapsulated
- pointer values cannot be manipulated
- pointer arithmetic (to be discussed) is not possible
- memory management is implemented by the `new` operator
- memory that is no longer in use is reclaimed automatically

In C, however, the memory of the computer is completely visible! (and memory management is a programmer’s job!)
References and pointers in C

Data in C are stored in memory and can be accessed by their names, **BUT**

- It is also possible to get information about their memory location:
  - if n is an integer variable, then &n is its location in memory!

- It is also possible to do calculation with these addresses:
  - if A[i] is the i-th element of array A, then A+i is its address,
    A+i-1 is the address of the previous element,
    and A+i+1 is the address of the next element
Non-primitive or structured data are either an array, struct, union, or a combination of those.

- **array**: a collection of homogeneous data, i.e. all elements of an array must be of the same type
- **struct**: a collection of heterogeneous data, i.e. components of a struct can be of different types, and of any complexity (structs can contain structs which can contain ...).
Non-primitive data in C (cont’d)

So

- typedef struct {
    
    int x, y;

} Point;

declares a new type called Point, and

- typedef struct {
    
    Point center;
    int radius;

} Circle;

declares a new type called Circle, and so on and so forth.
Memory management

For each data type (primitive or structured) a pointer type can be declared, either explicitly as in

- `typedef Point* PointRef`
- `typedef Circle* CircleRef`

or implicitly as in

- `Circle *theCircle`
  (which is the same as `Circle theCircle` in Java)

In C, the memory management needs to be programmed explicitly
Memory management (cont’d)

Circle *theCircle = malloc(sizeof(Circle));
theCircle->center = malloc(sizeof(Point));
theCircle->center->x = 1;
theCircle->center->y = 2;
theCircle->radius = 5;

corresponds to the following Java statement:

Circle theCircle = new Circle(
    new Point(1,2),5
);
Program structure: Java v.s. C

Echo command line arguments in Java:

```java
import java.lang.*; // is imported by default
public class Echo {
    public static void main(String[] args) {
        for (int i = 0; i < args.length; i++)
            System.out.print("args["+i+"] = "+args[i]+"\n");
        System.exit(0);
    }
}
```

Echo in C:

```c
#include <stdio.h> // needed for the function printf()
int main(int argc, char *argv[]) {
    int i;
    for (i = 0; i < argc; i++)
        printf("argv[%d] = %s\n", i, argv[i]);
    return 0;
}
```
Constants and variables

Libraries:

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

Constants:

```c
#define EOL '\n'
#define EOS '\0'
#define TAB '\t'
#define FALSE 0
#define TRUE 1
#define SECONDS_PER_DAY 86400
#define MONTHS_PER_YEAR 12
#define PI (4*atan(1)) // macro (needs math.h)
```

Variables:

- name, type and value
- scope and lifetime
**Primitive types**

- **int**: short int \(\leq\) int \(\leq\) long int \(\leq\) long long int
domains are given in `limits.h` (SHRT_MIN, LONG_MAX, etc.)

- truth values: false is represented by the integer value 0, any non-0 integer value represents true

- a boolean variable is often declared as an int (C99 has the type `bool`)

- **char**: char is an integer type with restricted domain
char variables are often declared as int

- real number values: float \(\leq\) double \(\leq\) long double
also see `float.h` and `double.h` for useful constants

- In C, the type byte does not exist

- In C, the type char consumes 1 byte of memory [-128 .. +127]

- Consequently, char is used when you need the type byte
Operators and expressions

Operators

- Arithmetic operators: -, +, *, / and %
- Relational operators: <, <=, > and >=
- Equality testing: == and !=
- Logic operators: !, && and ||

as in Java

Expressions

- Arithmetic expressions: similar to Java and mixed arithmetic requires the usual casts
Operators (cont’d)

- Relational arithmetic:
  usual tests with <, <=, >, >=, == and != as in Java, BUT also:
  - testing integer expression E for 0 or non-0
    ```
    if ((E != 0) == TRUE) ...
    ```
  - In Java you will write `if (E != 0) ...`
  - In C you can write `if (E) ...`
    E is interpreted as TRUE if E evaluates to anything not equal to 0, and as FALSE otherwise
Standard output (output to the screen) is implemented by the function `printf()` from the library `stdio.h` (see ref manual: "man printf")

Formatted standard input (input from the keyboard) is implemented by the function `scanf()` from the library `stdio.h`

For unformatted standard input, use `getchar()`, `gets()` or `fgets()`

I/O in C seems simpler than in Java but there are many pitfalls!

..and error and exception handling do not exist in C!
Type definition and macro’s

- Standard C as of ANSI (1989) is a (fairly) strongly typed language and contains the possibility to define your own types:
  - Readability of your code
  - Typechecking by the compiler

- A macro (does not exist in Java) is used to perform global textual substitution in the program text: macros are preprocessed before compilation
  - Readability of your code
  - Higher performance of execution.
C does not have methods but functions (similar to static Java methods)
\[ z = f(x, y), \text{i.e. returnType functionName(parameters)} \]

A function to calculate the factorial of \( n \) might read:
```
int fac(int n) { // required n >= 0
    int f = 1;
    for (; n; n--) f *= n;
    return f;
}
```
or as
```
int fac(int n) { // required n >= 0
    return (!n ? 1 : n * fac(n-1));
}
```
The declaration of a function requires a **prototype**. Again let

```c
int fac(int n) { // required n >= 0
    int f = 1;
    for (; n; n--) f *= n;
    return f;
}
```

be the definition of a function that calculates \( n! \) (factorial), then the prototype of this function is:

```c
int fac(int n); // argument n is optional in prototype
```

Now the compiler knows about

- the existence of a function with the name `fac`
- the return type of the function
- the ordered list of the types of the arguments of the function.
Let `eltType` denote a type and `elt` a variable of that type
Let `arrayOfElt` denote the name for an array of elements of `eltType`
On first sight arrays in C and Java are quite similar

- Setting the i-th element: `arrayOfElt[i] = elt;`
- Getting the i-th element: `elt = arrayOfElt[i];`

BUT

- Declaration
  - `eltType[] arrayOfElt;` is what you learned in Java
  - `eltType arrayOfElt[];` is what you have to do in C

- Creation
  - `eltType[] arrayOfElt = new eltType[1024];` in Java
  - `eltType arrayOfElt[1024];` in C
Arrays (cont’d)

Usage

- the length of a Java array is given by the length field
  
  ```java
  for (int i = 0; i < arrayOfElt.length; i++) {
    ...
  }
  ```

- in C you need a second item that represents the length
  
  ```c
  for (i = 0; i < ARRAY_LENGTH; i++) {
    ...
  }
  ```

Array parameter

- ... method(eltType[] A) { ... } in Java
- ... function(eltType A[]) { ... } in C

but often

- ... function(eltType *A) { ... } also since the name A of
  an array of elements of eltType is a constant pointer to eltType
  in C
Arrays (cont’d)

- Returning arrays

  - `eltType[] method(...) {...} 
  
    a Java-method can return an array of `eltType`

  but

  - a C-function cannot return an array of `eltType`

  - However, a C-function can return a pointer to `eltType`, and

    `eltType *function(...) {...} 

    is the way to return an array in C
Strings (*string.h* and *stdlib.h*)

In Java, String objects are instantiations of the class `String` and they behave like a Java array of char. Subtle differences:

- Strings are immutable (cannot be changed) and when you need modifiable Strings you have to use the class `StringBuffer`

- The length of a String object `s` is not given by a length field but by its length method: `s.length()`

In C, strings are `'\0'` terminated arrays of character

- This 0-character terminates the string (last character of the string).
- This character does not belong to the string!
- An empty string contains one character, namely `'\0'`
- The length of a string is equal to the number of characters excluding the terminating 0-character!
A brief string intermezzo

```c
#include <stdio.h>    // we use printf() and getchar()
#include <string.h>
#define MAX_CHARS 255

void print_string(char s[]);    // prototype
char str[MAX_CHARS];

void print_string(char s[]) {
    int i;
    for (i = 0; i < strlen(s); i++)
        printf("%c", s[i]);
}

int main(void) {
    int i = 0;
    /* BUG!: */
    while ((i < MAX_CHARS) && (str[i++] = getchar()) != '\n');
    str[i] = '\0';
    print_string(str);
}
```
A brief string intermezzo (cont’d)

```c
#include <stdio.h>
#include <string.h>
#include <ctype.h> // we use tolower()
#define MAX_CHARS 255

void to_lower(char s[]); // prototype
char str[MAX_CHARS];

void to_lower(char s[]) {
    int i;
    for (i = 0; i < strlen(s) - 1; i++)
        s[i] = tolower(s[i]);
}

int main(void) {
    int i = 0;
    while ((i < (MAX_CHARS-1)) && (str[i++] = getchar()) != ‘\n’);
    str[i] = ‘\0’; to_lower(str);
    puts(str); // note the output! (1 extra newline)
}

Later more on strings!
```
Some miscellaneous differences

- The type `char` consumes 2 bytes and the alphabet is unicode in Java, while C’s `char` consumes 1 byte and the alphabet is ASCII.

- Declarations of variables can occur everywhere in Java but in C declarations must occur at the beginning of a block (not true anymore in C99).

- Local loop variables are allowed in Java but not in C. So `for (int f = 1; n; n--) f *= n;` generates a compile error in ANSI C (OK for C99).

- Method (function) overloading is not possible in C.

- Methods that have no parameters are declared in Java by `... method() but in C you have to write ... function(void)`.
Some miscellaneous differences (cont’d)

- Initialization of variables in Java:
  - Class variables are initialized at class loading time
  - Instance variables are initialized at object creation time

- Initialization of variables in C:
  - Never ever presume that variables are initialized at all!!
  - Play it safe and initialize every variable at the moment of its declaration
Some miscellaneous differences (cont’d)

The method `main` in Java:

- `public static void main (String[] args) {}`
  - the returntype is `void`
  - and `args` is an array of `String`

The function `main` in C occurs in two flavours:

- `int main(void) {}`
  - the returntype is always `int`
  - the parameterlist is empty

- `int main (int argc, char *argv[]) {}`
  - the returntype is always `int`
  - `argv` is een array of pointers to `char`
  - and `argc` should be equal to the length of this array
Variable addresses and values

In a high-level language, memory locations are not known as addresses but they are known by name: the name of a constant, a variable, an array or a function.

Let us concentrate on variables:
- The notion of a variable is an abstraction of a memory location or address.
- When declaring a variable, say,
  ```c
  int n;
  ```
  the compiler binds `n` to an address in memory storing the value of `n` (e.g., `n = 81`)
- The value can be accessed, retrieved and changed using its name
  ```c
  n = n * 2;
  ```
LHS value and RHS value

Variables have a type to be able to interpret the stored value. Thus:

- if \( n \) is of type \texttt{int} and its address contains bitpattern 0..001010001 then its value will be 81
- if \( c \) is a \texttt{char} and its address holds 01010001 then its value will be ’Q’.

Variables are essentially entities with two attributes:

- address or lhsvalue → where does the entity exist?
- content or rhsvalue or value → what does the entity contain?

Consider \( n = n \times 2; \)

- on the righthand side (rhs) the name \( n \) stands for its value and the expression \( n \times 2 \) evaluates to 162
- on the lefthand side (lhs) the name \( n \) stands for its address and \( n = 162; \) means that 162 will be stored at this address
Declarations, assignments and compiler actions

The essential difference between address (lhsvalue) and value (rhsvalue) can be made even more clear even by putting the following question:
Did you ever write \( 13 = n; \) ?

- **Declaration**: \( \text{int anInt;} \)
- **Assignment**: \( \text{anInt = 13;} \)
- **Compiler actions**:
  - allocate \( i \) bytes of memory (e.g. addresses 1004 .. 1007), with \( i \) the number of bytes required to store an integer
  - add name, type, address, .. to internal symboltable
  - store value 13 at address of \( \text{anInt} \)
The pointer concept

Let us take a bold step!
- an lvalue is an address
- declare variables that can hold an address: why not?

A variable that can hold an address is called a pointer variable or simply a pointer or reference

A pointer is said to "point at" or "refer to" the address it holds
In C, a pointer variable is declared by preceding its name by `*`:

```c
int *anIntPtr;
```

declares a pointer variable with the name `anIntPtr`

Reading from right to left:

- `anIntPtr` is a pointer to an integer
- `*anIntPtr` is an expression that evaluates to an integer
Declaration of a pointer

Declaration: int *nIntPtr;

- Allocates \( i \) bytes of memory (e.g. at addresses 1004 .. 1007) with \( i \) the number of bytes required to store an address

- Adds name, type, address to internal symboltable

The variable \( \text{anIntPtr} \) has no value yet: the pointer is not initialized

Our intention is to initialize this pointer by assigning an address where you can find an integer
Assignment to a pointer

Let us assign the address value 1004 of our previous variable `anInt`:

```c
int anInt;
int *anIntPtr;
anIntPtr = &anInt;
```

`&anInt` is the C expression for ’address of the variable `anInt’’

- The address operator `&` determines the address of variable `anInt`
  and this address is assigned to the pointer variable `anIntPtr`
- We say that `anIntPtr` points at or refers to `anInt`, and
- `anIntPtr` holds the address of `anInt`
An important question now is how to get the value of the variable anInt using anIntPtr, which is the pointer that points at it!

```c
int n2, *nPtr;
int n1 = 13;
nPtr = &n1;
```

The value of n1 can be retrieved (and stored into n2) in an indirect way:

```c
n2 = *nPtr;
```

and now n2 holds the value 13 as well!
The dereferencing operator * (cont’d)

After \( n2 = *nPtr; \) the variable \( n2 \) is equal to 13.

- The operator * is called the dereferencing operator
- The evaluation of expression \( *nPtr \) is called dereferencing the pointer
  - \( nPtr \) holds the address of \( n1 \) (\&\( n1 \), which is 1004 here)
  - \( *nPtr \) is an expression of type int, evaluation yields the value stored at \&\( n1 \), that is \( *\&n1 \) or simply \( n1 \)
- Summary: int \( n \), *\( pn \); \( pn = \&n; \)
  - \( n \) is a variable of type int, \( pn \) is a variable of type pointer to int, \&\( n \) is an expression that evaluates to the address of \( n \), and *\( pn \) is an expression that evaluates to the value of \( n \)
A sample program

A small program to display a "memory map":

```
#include <stdio.h> // needed for printf()

int main(void) {
    int j = 13,
        k = 47,
        *p = &k; // declare pointer p and initialize it

    printf("j is %d and its address is %p\n", j, &j);
    printf("k is %d and its address is %p\n", k, &k);
    printf("p is %p and its address is %p\n", p, &p);
    return 0;
}
```

- **format specifier\%p** is used to print the value of a pointer variable (an address)
- **format specifier\%d** is used to print the value of an integer variable (you can also use \%i)
A sample run

\[ \text{j is 13 and its address is 00bc2610 i.e. } &j \]
\[ \text{k is 47 and its address is 00bc260c i.e. } &k \]
\[ \text{p is 00bc260c and its address is 00bc2608 i.e. } &p \]
include <stdio.h>
int main(void) {

    int j = 13,
        k = 47,
        *p = &k;

    printf("%p : address of pointer variable p\n", &p);
    printf("%p : the value of p is the address of k\n", p);
    printf("%p : the address of k (checked)\n", &k);
    printf("%d : the value *p is the value of k\n", *p);
    printf("%d : the value of k (checked)\n", k);
    *p = 87; // change whatever p points at into 87
    printf("%d : the new value *p\n", *p);
    printf("%d : the value of k (indeed k has changed)\n", k);
    printf("%d : the value of k again (i like playing)\n", **&p);
    return 0;
}
Sample run of program 2

00bc2388 : address of pointer variable p
00bc238c : the value of p is the address of k
00bc238c : the address of k (checked)
47 : the value *p is the value of k
47 : the value of k (checked)
87 : the new value *p
87 : the value of k (indeed k has changed)
87 : the value of k again (i like playing)
Pointer syntax and pitfalls

The declarations

- `int* p;`
- `int *p;`
- `int* p;`
- `int * p;`
- `int * p;`

are essentially identical from the compiler’s point of view.
Pointer initializations

Initializations

```c
int k, *p; // declare k as int and p as pointer to int
p = &k;   // initialize p
*p = 87;  // initialize k
```

can also be written as

```c
int k, // declare k as int
    *p = &k; // init pointer p to int at address of k
    *p = 87; // initialize k
```

Do not confuse the expressions

```c
int *p = &k;
```

which declares pointer `p` and assigns the address value `&k`, and

```c
*p = 87;
```

which assigns value 87 to `k` (which is pointed at by `p`).
Pointer initializations: a warning

Consider the following fragment

```c
int k = 47, *p; // declare and initialize int k
    // and declare pointer p to int
*p = 87; // ERROR!!!
```

- Pointer `p` has been declared but not initialized!
- `p` points nowhere and the assignment is an error that will not be detected....until you execute!

Always initialize your pointers!
Pointer initializations (cont’d)

One way to initialize is by \( p = \text{NULL}; \)

- Now the error will be detected (in most cases) when dereferencing a null-pointer

The other way has been shown above: \( p = \& k; \)

- a valid address is assigned to pointer \( p \), referring to integer \( k \), which can be changed using \( *p \)

Notes:

- NULL is the universal pointer constant that can be assigned to any pointer

\( \text{printf("\%p", \text{NULL}); yields 00000000} \)
Functions: call by value and call by reference

returnType functionName(formalParameterTypeList) {
    // definition or implementation
}

- returnType
- void
- any value type or any pointer type
- formalParameterTypeList
  - void
  - comma-separated sequence of formalParameterType
- formalParameterType
  - typeIdentifier (optionally followed by variableIdentifier)

Function prototype (required in ANSI C):
returnType functionName(formalParameterTypeList);
Functions: call by value and call by reference

Formal parameters can always be classified:

- **Value parameters**
  - Used to import the necessary data for the function to do its job
  - This import mechanism is known as **call by value**

- **Reference parameters**
  - Used to export the results after the function has done its job
  - Can also be used for import: the function modifies the imported data and exports the modified data using the same parameters
  - This export mechanism is known as **call by reference**
Call by value and call by reference (cont’d)

```c
int min(int n1, int n2) {
    // determine minimum of n1 and n2 and return result
}
```

usage: `theMin = min(a, b);`

or

```c
void min(int n1, int n2, int *min) {
    // determine min of n1 and n2 and
    // return result by ref
}
```

usage: `min(a, b, &theMin);`

**Import a, b and &theMin and export minimum of a and b, i.e.**

`theMin`
void minmax(int n1, int n2, int *min, int *max) {
    // determine min and max of n1 and n2 and
    // return by ref
}
usage: min(a, b, &theMin, &theMax);

or

void sort3(int *n1, int *n2, int *n3) {
    // sort n1, n2 and n3 and return results by ref
}
usage: sort3(&a, &b, &c);

Export a, b and c such that a <= b <= c
Call by value example

```c
#include <stdio.h> // we use printf()

void test(int); // prototype

int main(void) {
    int k = 5;
    printf("before calling test() k is %d.\n", k); // 5
    test(k);
    printf("after calling test() k is %d.\n", k); // 5
}

void test(int n) {
    printf("inside test() n is %d.\n", n); // 5
    printf("now we change n.\n");
    n++;
    printf("now we have changed n.\n");
    printf("inside test() n is %d.\n", n); // 6
}
```
Call by value example (cont’d)

- At the moment that the function `test` is called
  - memory is created for formal parameter `n` (on the stackframe)
  - value 5 of actual argument `k` is copied into formal parameter `n`

- During execution of the function `test`
  - the value of the local variable `n` is used and even changed
  - but whatever happens with `n` will have no effect on `k` at all!! (only on its copy on the stackframe)

- After termination of the function `test`
  - the local variables that correspond with the formal parameters do not exist any longer (stackframe is destroyed)
  - the value of the variable `k` that lives outside the function is still 5
Assume function \( f: \text{void} \ f(\text{int} \ *n) \ \{ \ldots \} ; \)

Usage: \( f(&a); \), where \( a \) is a variable of type \( \text{int} \)

- address \( &a \) is copied into formal parameter \( n \) (on the stackframe)
  and now we have two memory locations, named \( n \) and \( a \), such that
  - \( n \) contains the address of \( a \)
  - \( a \) contains an integer value
  - \( *n \) is an expression that evaluates to the value of \( a \)
  - we say that \( a \) and \( *n \) are synonymous: what happens with \( *n \) during execution of the function \( f \) also happens with the actual variable \( a \) since \( n \) equals \( &a \)
Call by reference: an example

```c
#include <stdio.h> // we use printf()

void test(int, int *); // prototype

int main(void) {
    int K = 5;
    printf("before calling test() K is %d.\n", K); // 5
    test(K, &K);
    printf("after calling test() K is %d.\n", K); // 6
}

void test(int p, int *q) {
    printf("inside test(): p = %d and *q = %d.\n", p, *q);
    // 5 5
    printf("now we increase the value of *q.\n"); (*q)++;
    printf("inside test(): p = %d and *q = %d.\n", p, *q);
    // 5 6
}
```

Call by reference: `int getInput(void)`

Get structured input via `scanf()`

```c
int main(void) {
    int c;
    printf("an int please.\n>>\t"); // get input
    scanf("%d", &c); // store at address of c
    printf("you entered %d.\n", c); // print to stdout
}

Now using a function `int getInput(void)`;

```c
int getInput(void) {
    int k; // local variable k
    printf("an int please.\n>>\t"); // user enters 123
    scanf("%d", &k); // store 123 at &k
    return k; // and return value of k
}

int main(void) {
    int c;
    c = getInput(); // ask user for value and assign to c
    printf("you entered %d.\n", c); // print value of c
}
```
Call by reference: \texttt{void getInput(int \*n);} 

Now using a call-by-ref function \texttt{void getInput(int \*n);} 

\begin{verbatim}
void getInput(int \*n) {  // n is address of int
    printf("an int please.\n\n>>\t");  // user enters 123
    scanf("%d", n);  // store value at n
}
\end{verbatim}

\textbf{Note!} \texttt{scanf("%d", n);} \textbf{and NOT} \texttt{scanf("%d", \&n);} 

\begin{verbatim}
int main(void) {
    int c;
    getInput(\&c);  // ask user for value and assign to c
    printf("you entered %d.\n", c);  // print value of c
}
\end{verbatim}

\texttt{printf("%d", \ldots)} needs the value of an integer! 
\texttt{scanf("%d", \ldots)} needs a pointer to an integer!
Call by reference: swapping

```c
void swap(int *jPtr, int *kPtr) {
    int temp = *jPtr; *jPtr = *kPtr; *kPtr = temp;
}
int main(void) {
    int n1 = 13, n2 = 47;
    printf("n1 = %d and n2 = %d \n", n1, n2);
    swap(&n1, &n2);
    printf("n1 = %d and n2 = %d \n", n1, n2);
    return 0;
}

indeed prints

n1 = 13 and n2 = 47
n1 = 47 and n2 = 13
```
Call by reference: some more examples

Sorting two integers

```c
void sort2(int *min, int *max) {
    if (*min > *max)
        swap(min, max);
}
```

Determine minimum and maximum of three integers

```c
void minmax(int n1, int n2, int n3, int *min, int *max) {
    int min23 = ((n2 < n3) ? n2 : n3),
    max23 = ((n2 > n3) ? n2 : n3);
    *min = ((n1 < min23) ? n1 : min23);
    *max = ((n1 > max23) ? n1 : max23);
}
```
Call by reference: some more examples (cont’d)

Sorting three integers

```c
void sort3(int *min, int *mid, int *max) {
    sort2(min, mid); // now *min <= *mid
    sort2(min, max); // now *min <= *max
    sort2(mid, max); // now *mid <= *max
    // hence *min <= *mid <= *max
}
```

Determine remainder and quotient

```c
void remquo(int *n1, int *n2) {
    int rem = *n1 % *n2,
    quo = *n1 / *n2; // take care!!
    *n1 = rem; *n2 = quo;
}
...
int a = 105, b = 17;
remquo(&a, &b); // original values of a and b are lost!
```
**I/O functions: `printf()`**

```c
int printf("format string", list of expressions);
```

- format string consists of a sequence of characters delimited by " and when you need to print " you must use \\
- format specifiers control the printing of their arguments
- a format specifier starts with % and ends with a conversion character
- when you need to print % you must use \%

`printf` evaluates expressions in the list and prints the resulting values according to the specifications of the format string

`printf` returns the number of characters it writes to standard output if successful, otherwise it returns -1.
I/O functions: `printf()` (cont’d)

```c
int c = 299997;
printf("c = %d km/sec and pi = %5.3f.\n", c, 4*atan(1));
```

- **format string** → `c = %d km/sec and pi = %5.3f.\n`
- **regular characters** → `c = km/sec and pi = .\n`
- **format specifiers** → `%d %5.3f`
- **expressions** → `c` and `4*atan(1)`

- `%d` tells `printf` to print the value of `c` as an integer
- `%5.3f` tells `printf` to print the result of the evaluation of the expression as a real number in a field of width 5 with a precision of 3 digits after the decimal point
- For more details: see reference manual!
I/O functions: scanf()

The function `scanf()` parses formatted input, reads characters from an inputstream and converts sequences of characters according to its control string format.

```c
char c;
int n;
double x;

printf("enter a character please.\n");
scanf("%c", &c);
printf("enter a whole number please.\n");
scanf("%d", &n);
printf("enter any real number please.\n");
scanf("%f", &x);
```
I/O functions: `scanf()`

The function `scanf()` can be tricky in its use! An example:

```c
void Test2(void) {
    char c1, c2, c3; int n1, n2; double x;
    printf("Enter three chars, two ints an one real.\n");
    printf("Separate input elements by a single space.\n>>\t");
    scanf("%c%c%c%d%d%lf", &c1, &c2, &c3, &n1, &n2, &x);
    printf("|%c||%c||%c||%d||%d||%f|\n", c1, c2, c3, n1, n2, x);
}
```

Sample run:

Enter three chars, two ints an one real.
Separate input elements by a single space.
>>a b c 123 1234 1.0
|a|| ||b||44615512||44240382||-1096337729661848233.000000|

This is not what I had in mind...
Later more on I/O functions!
Defining your own datatypes

C offers the possibility to introduce new types using the `typedef` mechanism:

```c
typedef oldType newType;
```

This mechanism can be used to introduce types that do not exist in C. Some simple examples:

```c
typedef int boolean;
const boolean FALSE = 0, TRUE = 1;
```

From now on you can use your own truth-value type `boolean` whenever you mean boolean:

```c
typedef int boolean;
boolean isPrime(int k); // NB: C99 defines bool!!
```

Maybe you do not like the type `double`

```c
typedef double real;
real circleArea(real radius);
```
Enumerated types

Defining enumerated types is also done with `typedef`

```c
typedef enum {comma-separated list of identifiers} newType;
```

The internal representation of the identifiers uses integer values starting from 0 (unless otherwise specified). Some examples:

```c
typedef enum {FALSE, TRUE} bool;
typedef enum {RED, ORANGE, GREEN} color;
```

```c
color trafficLight;
```

and now `trafficLight` is a variable of type `color` that can take only 3 values

```c
switch (trafficLight) {
    case RED : /* stop */ break;
    case ORANGE : /* accelerate */ break;
    case GREEN : /* go */ break;
    default : /* weird traffic light */ break;
}
```
Enumerated types (cont’d)

Some more examples

typedef enum {UNDEF, MON, TUE, WED, THU, FRI, SAT, SUN} day1;
internal rep 0 1 2 3 4 5 6 7

typedef enum {MON = 1, TUE, WED, THU, FRI, SAT, SUN} day2;
internal rep 1 2 3 4 5 6 7

typedef enum {SUN, MON, TUE, WED, THU, FRI, SAT} day;
internal rep 0 1 2 3 4 5 6

This last type is most convenient:

day aDay, tomorrow, aWorkDay; // three variables of type day
day theNextDay(day d) { // function that returns tomorrow
  return (d+1) % 7;
}
day d;
for (d = MON; d < SAT; d++) { // a for loop processing work days
  // work!!
}
A **struct** is a datatype where variables of possibly different types can be grouped together, e.g.

- your name: one or more initials and a last name
- your home address: street, streetnum, postcode, city
- a circle: center and radius
- a point: x- and y-coordinate, etc. etc.

A point is characterized by x- and y-coordinate:

```c
struct point {    // defines the type struct point
double x;        // field x of type double
double y;        // field y of type double
};                // do not forget the semicolon!
struct point p, q; // variables p and q of type struct point
```

Now define points p and q:

```c
p.x = 0.0; p.y = 0.0; q.x = 3.0; q.y = 4.0;
```
Struct(ures): points and circles

```c
struct circle1 { // defines the type struct circle1
double x; // x-coordinate of type double
double y; // y-coordinate of type double
double radius; // radius of type double
}; // do not forget the semicolon!

struct circle1 cl; // variable cl of type struct circle1

cl.x = cl.y = 0.0; // cl has the origin as its center
cl.radius = 5.0; // and its radius is equal to 5.0

double circleArea(struct circle1 anyCircle) {
    return(PI * anyCircle.radius * anyCircle.radius);
}
```
Struct(ures): points and circles (cont’d)

Composition of structs

```c
struct circle2 {
    struct point center; // center of type struct point
double radius; // radius of type double
}; // do not forget the semicolon!
```

```c
struct circle2 c2; // variable c2 of type struct circle2

c2.center.x = 4.0;
c2.center.y = 5.0; // c2 has center [4.0, 5.0]
c2.radius = 10.0; // and the radius is equal to 10.0
```

The new types are `struct point`, `struct circle1` and `struct circle2`. Here `point`, `circle1` and `circle2` are called **structure tags**
Struct(ures): points and circles (cont’d)

It is far more convenient to have a type as short as the tag which can be achieved using the `typedef` mechanism:

```c
typedef struct point {   // tag is optional
    double x;
    double y;
} point;              // defines the new type point
point p, q;           // variables of type point

typedef struct circle {  // tag is optional
    point center;
    double radius;
} circle;             // defines the new type circle
circle c;             // variable of type circle
```
Struct(ures): a larger example

A library of books

- Capacity of 1024 books \((\text{libCapacity} = 1024)\)
- The number of books currently present is given by \(\text{numBooks}\)
- Each book is characterized by title, author, isbn-code and purchase date

Design of the library

- Since the library has a fixed capacity, books will be stored in an array that can contain 1024 books at most
- The current number of books will be kept in \(\text{int numBooks and books are arranged} \ 0..\text{numBooks}-1 \ \text{with} \ \text{numBooks} <= \ \text{libCapacity}\).
- The array of books will be called \(\text{bookStore}\) and contains elements of type \(\text{book}\)
- the library itself is of type \(\text{library}\)
#define LIB_CAPACITY 1024

typedef struct library {
    int numBooks; // current number of books
} library; // type library

library theLibrary; // variable theLibrary

typedef struct book {
    string title, author, isbncode;
    date purchase;
} book;

book aBook;
Struct(ures): a larger example (cont’d)

```c
#define MAX_STR_SIZE 32
typedef char string[MAX_STR_SIZE];
string aString;

typedef struct date {
    int day,
    month,
    year;
} date;
date aDate;

Initialization of a date variable:

date aDate = {
    7, 4, 2004
};
```
Struct(ures): a larger example (cont’d)

Initialization of a book variable

```c
book aBook = {
    "Programming in ANSI C", // title
    "Kochan SG",              // author
    "0-672-3039-6",           // isbn code
    {15, 8, 1994}             // date
};
```

In the same spirit theLibrary can be initialized

```c
library theLibrary = {
    {title0, author0, isbn0, {day0, month0, year0}},
    {title1, author1, isbn1, {day1, month1, year1}},
    {title2, author2, isbn2, {day2, month2, year2}},
    {title3, author3, isbn3, {day3, month3, year3}},
    {title4, author4, isbn4, {day4, month4, year4}}},
5
};
```
Struct(ures): a larger example (cont’d)

Retrieving data

book lastBook = theLibrary.bookStore[theLibrary.numBooks - 1];
date theRequiredDate = lastBook.purchase;

Inserting and updating data

book newBook;
date yesterday = {6, 4, 2004};
strcpy(newBook.title, "Il nome della rosa");
strcpy(newBook.author, "Eco U");
strcpy(newBook.isbncode, "90-351-1413-2");
newBook.purchase = yesterday;
theLibrary.bookStore[theLibrary.numBooks++] = newBook;
Struct(ures): assignments

book copyOfBook123 = theLibrary.bookStore[123];
library copyOfTheCompleteLibrary = theLibrary;

follows the same pattern as any value-type assignment like
int n2 = n1;

So, a struct can, e.g., be used as a formal parameter in a function

double distanceToOrigin(point p) {
    return(sqrt(p.x * p.x + p.y * p.y));
}

usage:

point somePoint = {1.0, 5.0};
double d = distanceToOrigin(somePoint);
Functions returning a struct

A function can return a struct

```c
book newBook(string title, string author, string isbn, date d) {
    book b;
    strcpy(b.title, title);
    strcpy(b.author, author);
    strcpy(b.isbn, isbn);
    b.purchase = d;
    return b;
}
```

Usage:

```c
string title = "Il nome della rosa",
    author = "Eco U",
    isbn = "90-351-1413-2";
date d = {6, 4, 2004};
book b = newBook(title, author, isbn, d);
```
Arrays in C

First a few simple examples of how to declare and initialize arrays

- `int A[10];` declares array `A` containing 10 un-initialized integers
- `int A[10] = {0};` declares the same array `A` but initializes the elements to 0
- `int A[10] = {0, 10, 20, 30, 40};` initializes the first 5 elements as given, while the remaining elements are set to 0
- `int A[10] = {0, 10, 20, 30, 40, 50, 60, 70, 80, 90};` initializes all elements as given

In C, array elements are counted starting at index value 0! So: `A[0] ... A[9]`
Arrays in C (cont’d)

An example: determine the sum of all array elements

```c
int A[10] = {0, 10, 20, 30, 40, 50, 60, 70, 80, 90};
int temp = A[7]; // assigns value 70 to temp
int i = 10,
    sum = 0;

do
    sum += A[--i];
while (i);
```

Datastructures / Programming in C – A.D. Pimentel – p. 90/226
Arrays in C (cont’d)

Some common declarations

- #define MAX_SIZE ...
  elt_type A[MAX_SIZE];
- typedef elt_type elt_array_type[MAX_SIZE];
  Now, you can use e.g.: elt_array_type A, B;

Example:

typedef int int_vec32[32];
int_vec32 v1, v2;

defines two vector-type variables, where a vector type is an array of 32 integers
A brief intermezzo on `sizeof()`

`sizeof()` is an operator that takes one argument (a type, a variable or an expression) and returns the size of its argument in bytes:

\[
\begin{align*}
nc &= \text{sizeof}(\text{char}); \quad // \ nc == 1 \\
ns &= \text{sizeof}(\text{short}); \quad // \ ns == 2 \\
ni &= \text{sizeof}(\text{int}); \quad // \ ni == 4 \\
nf &= \text{sizeof}(\text{float}); \quad // \ nf == 4 \\
nd &= \text{sizeof}(\text{double}); \quad // \ nd == 12
\end{align*}
\]

The type argument can also be user-defined:

\[
\begin{align*}
typedef \text{double} & \quad \text{elt}; \quad // \text{type elt is double} \\
typedef \text{elt}* & \quad \text{pelt}; \quad // \text{type pelt is pointer to elt} \\
\text{elt} & \quad \text{e, *ep}; \quad // \text{e and *ep are type elt} \\
\text{elt} & \quad \text{e1[1024]}; \quad // \text{e2 is type array of elt}
\end{align*}
\]
A brief intermezzo on `sizeof()` (cont’d)

Now

- `sizeof(elt) == sizeof(e) == sizeof(*ep) == 12`
  All double types

- `sizeof(elt *) == sizeof(pelt) == sizeof(ep) == 4`
  All pointer types

- `sizeof(e1) == 12288`  // Size of array

BUT

```c
void foo(elt h[]) {
    printf("%d\n", sizeof(h)); // WILL PRINT 4!!!
}
int main(void) {
    foo(e1);
}
```
Arrays in C (cont’d)

- Arrays are not value types but pointer types
- Arrays identifiers are constant pointers (cannot be assigned to)
- For `int theArray[10];` the compiler
  - selects a free block of memory to store 10 integers contiguously!
  - base address, `&theArray[0]`, is assigned to the name of the array
- `theArray == &theArray[0]` is a constant that cannot be changed.
Arrays in C (cont’d)

With `theArray` equal to `&theArray[0]`, we also have

- `theArray + i == &theArray[i]`
- `theArray[i] == *(theArray + i)`

If we have `int a[10], b[10]`; then

`a = b;

b = a;

are forbidden assignments! (would you ever write `13 = 18`...)
Arrays in C (cont’d)

Let int a[10], *b;
then
a = b;
is illegal too (same reason)

But
b = a;
is OK and can be very useful!

Note: you do not get a copy of the array, only a pointer to it
(b == &a[0])!
Assume

```
int a[] = {0, 10, 20, 30, 40, 50, 60, 70, 80, 90};
int *p;  // a pointer to int not yet initialized
p = a;   // now p points to element a[0] of array a
        // i.e. p == &a[0] and *p == a[0]
        // and  p+i == &a[i] and *(p+i) == a[i]
        // but *(p+i) and *p+i are different things!!
```

- **++a or a++** will not work to reach consecutive elements of array a
- **But ++p or p++** does the trick!
Arrays in C (cont’d)

Printing array elements:

```c
for (i = 0; i < n; i++) printf("%4d", a[i]);
for (i = 0; i < n; i++) printf("%4d", p[i]);
for (i = 0; i < n; i++) printf("%4d", *(a+i));
for (i = 0; i < n; i++) printf("%4d", *(p+i));
```

Or

```c
for (i = 0; i < n; i++) {printf("%4d", *p); p = p+1;}
for (i = 0; i < n; i++) {printf("%4d", *p); p++;
for (i = 0; i < n; i++) {printf("%4d", *p++);}
Arrays in C (cont’d)

- $n = *p++;$ is equivalent to $n = *p; \ p = p+1;$
- $n = *(p++);$ is equivalent to $n = *p; \ p = p+1;$
- $n = *(++p);$ is equivalent to $p = p+1; \ n = *p$
- $n = (*p)++;$ is equivalent to $n = *p; \ *p = *p+1;$
- $n = ++(*p);$ is equivalent to $*p = *p+1; \ n = *p;$

Be careful though!

- $(*(p++))++, \ ++(*(++p), \ ++((*p)++), ...$
int main() {
    int a[] = {99, 10, 20, 30, 40, 50, 60, 70, 80, 90};
    int i, *p = a;

    printf("\n i ");
    for (i=0; i<10; i++) printf("%4d",i);
    printf(" a[0]\n");
    printf("=====================================");
    printf("=======================================\n");
    printf(" a[i] ");
    for (i=0; i<10; i++)
        printf("%4d", a[i]);
    printf("%8d",a[0]);
    printf(" *(p+i) ");
    for (i=0; i<10; i++)
        printf("%4d", *(p+i));
    printf("%8d",a[0]);
    printf("-------------");
Arrays in C (cont’d)

```c
printf("\nExp1:  *p++       "); p = a;
for (i=0; i<10; i++) printf("%4d",*p++);  printf("%8d",a[0]);
printf("\nExp2:  (*p)++     "); p = a;
for (i=0; i<10; i++) printf("%4d",(*p)++); printf("%8d",a[0]);
printf("\nExp3:  *(p++)      "); p = a;
for (i=0; i<10; i++) printf("%4d",*(p++)); printf("%8d",a[0]);
printf("\n---------------");
printf("\nExp4:  ++*p        "); p = a;
for (i=0; i<10; i++) printf("%4d",++*p);  printf("%8d",a[0]);
printf("\nExp5:  ++(*p)      "); p = a;
for (i=0; i<10; i++) printf("%4d",++(*p)); printf("%8d",a[0]);
printf("\n---------------");
printf("\nExp6:  *++p        "); p = a;
for (i=0; i<10; i++) printf("%4d",*++p);  printf("%8d",a[0]);
printf("\nExp7:  *(++p)      "); p = a;
for (i=0; i<10; i++) printf("%4d",*(++p)); printf("%8d",a[0]);
printf("\n\n");
printf("="*27"\n");
printf("="*37"\n");
```

### Arrays in C (cont’d)

<table>
<thead>
<tr>
<th>i</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>a[0]</th>
</tr>
</thead>
<tbody>
<tr>
<td>a[i]</td>
<td>99</td>
<td>10</td>
<td>20</td>
<td>30</td>
<td>40</td>
<td>50</td>
<td>60</td>
<td>70</td>
<td>80</td>
<td>90</td>
<td>99</td>
</tr>
<tr>
<td>*(p+i)</td>
<td>99</td>
<td>10</td>
<td>20</td>
<td>30</td>
<td>40</td>
<td>50</td>
<td>60</td>
<td>70</td>
<td>80</td>
<td>90</td>
<td>99</td>
</tr>
</tbody>
</table>

**Exp1:** \*p++
```
99 10 20 30 40 50 60 70 80 90 99
```

**Exp2:** (*p)++
```
99 100 101 102 103 104 105 106 107 108 109
```

**Exp3:** *(p++)
```
109 10 20 30 40 50 60 70 80 90 109
```

**Exp4:** ++\*p
```
110 111 112 113 114 115 116 117 118 119 119
```

**Exp5:** ++(*p)
```
120 121 122 123 124 125 126 127 128 129 129
```

**Exp6:** *++p
```
10 20 30 40 50 60 70 80 90 ? 129
```

**Exp7:** *(++p)
```
10 20 30 40 50 60 70 80 90 ? 129
```

---
Arrays in C (cont’d)

Arrays are passed by referenced (because they are pointer types)

```c
void print (int a[], int n); // print elts a[0..n-1]
void process (int a[], int n); // square elts a[0..n-1]

int main(void) {
    int A[] = {1, 2, 3, 4, 5, 6, 7, 8, 9},
    N = 9;
    printf("Array before processing:"); print(A, N);
    process(A, N);
    printf("Array after processing:"); print(A, N);
    return 0;
}

void print(int a[], int n) {
    int i;
    for (i = 0; i < n; i++) printf("%3d", a[i]);
    printf("\n");
}

void process(int a[], int n) {
    int i;
    for (i = 0; i < n; i++) a[i] = a[i] * a[i];
}
```

Arrays in C (cont’d)

Output:

Array before processing: 1 2 3 4 5 6 7 8 9
Array after processing: 1 4 9 16 25 36 49 64 81

Swapping integers

```c
void swap(int n1, int n2) { // incorrect swap
    int temp = n1; n1 = n2; n2 = temp;
}
void swap(int *n1, int *n2) { // correct swap
    int temp = *n1; *n1 = *n2; *n2 = temp;
}
```

Swapping array elements:

```c
swap(&a[j], &a[k]); // addresses of elements j and k
swap(a+j, a+k); // addresses of elements j and k
```
Arrays in C (cont’d)

In a function `return_type fun(elt_type x[], int n);`

- when an actual array `elts` is passed as an argument, then the formal parameter `x` receives a copy of `&elts[0] == elts`
- `elts` is a constant pointer
- `x` is a common pointer that is initialized to `&elts[0]`
- The value of `elts` cannot be changed but `x` can!

```c
int max (int a[], int n) { // determine maximum
    int maxsofar = 0; // init maxsofar on 0
    while (n) {
        if (*a > maxsofar) maxsofar = *a;
        n--; a++;
    }
    return maxsofar;
}
```
Arrays in C (cont’d)

The function declarations

- `ret_type fun(elt_type x[32]); // compiler reads elt_type*`
- `ret_type fun(elt_type x[]); // compiler reads elt_type*`
- `ret_type fun(elt_type *x); // compiler reads elt_type*`

are all equivalent (the compiler ignores the size)

You always have to inform the function about how many elements must be processed:

- Process the first n elements x[0..n):
  ```
  return_type fun(elt_type *x, int n);
  ```

- Process all elements from `lbnd` to `ubnd` inclusive
  ```
  return_type fun(elt_type *x, int lbnd, int ubnd);
  ```
Pointer arithmetic

Since the compiler now knows about the type of elements (\texttt{elt\_type}) stored in array \texttt{x}, it supports the following address arithmetic:

\begin{align*}
\&x[j] - \&x[0] &= x+j - x = j
\end{align*}

That is \( j \) elements, or \( j \times \text{sizeof(elt\_type)} \) bytes

Possible types of pointer arithmetic:

- \texttt{ptr + int}: \( x + j \rightarrow j \) elements to the right of \( x \)
- \texttt{ptr - int}: \( y - k \rightarrow k \) elements to the left of \( y \)
- \texttt{ptr - ptr}: \( x - y \rightarrow \) number of elements from \( x \) to \( y \)
- \texttt{ptr < ptr}: \( x < y \rightarrow \) and all other comparison operators and equality and inequality testing

Forbidden: \texttt{ptr * int, ptr / int, ptr \& int, ptr + ptr, etc.}
Arrays as function return types

C functions cannot return arrays, BUT
C functions can return a pointer to the first element of an array!

```c
elt_type[] fun( .. ) // is forbidden in C
elt_type *fun( .. ) // is OK!
```

Be careful though!

```c
elt_type *fun( .. ) {
    elt_type elt_array[32]; // array on stackframe!
    // do whatever you want to do with array elt_array
    return elt_array;
}
```

After execution of `fun()`, `elt_array` does not exist anymore!

Here you have to allocate global memory (to be discussed later):
```c
elt_type *elt_array = malloc(32*sizeof(elt_type));
```
Assigning and copying arrays

In

\[
\text{int } n1 = 13, \\
n2; \\
n2 = n1;
\]

the value of \( n1 \) is assigned to \( n2 \)

But

\[
\text{int } n1[5] = \{0, 1, 2, 3, 4\}, \\
n2[5]; \\
n2 = n1;
\]

does not work...remember \( n1 \) and \( n2 \) are constant pointers!

So, copying needs to be done element-wise!
Copying arrays (cont’d)

int i; for (i = 0; i < 5; i++) n2[i] = n1[i];

Or as a function:

```c
void make_copy(elt_type *dst, elt_type *src, int n) {
    // We assume that the dst array already exists
    while (--n >=0)
        dst[n] = src[n];
}
```
A sequence of characters delimited by double quotes, like
"Programming in C is fun!"
is called a string (the double quotes do not belong to the string)

- Technically, a string is an array of characters
- However, there is extra requirement: the character array needs to end with a ‘\0’ (null) character!
- The null-character does not belong to the string itself

```c
char message[] = {'H','e','l','l','o','!'};
```
is an array of char of length 6,

```c
char message[] = {'H','e','l','l','o','!'','\0'};
```
is a string of length 6, and

```c
char empty_message[] = {'\0'};
```
is a character array of length 1 and a string of length 0!
Initializing strings

Strings allow easier initializations

```c
char message[] = {"Hello!"};
char message[] = "Hello!";
```

and the preferred size can be given as well

```c
char message[13] = "Hello there!";
char message[64] = "Hello there!";
```

but

```c
char message[12] = "Hello there!"; // is not a string!!
```

Usually however we will not use char array but we prefer `char*`. Thus:

```c
char *s = "The string to be copied."; // size 25 = 24 + 1
char *t = "The string to receive a copy."; // size 30 = 29 + 1
```

In both cases the internal representation is an array of char where the size is given by the number of characters plus 1 (to accommodate ’\0’).
The length of a string

The length of a string: i.e. the actual number of characters excluding ‘\0’

```c
#define EOS '\0'
int length(char *s) {
    int i = 0;
    while(s[i] != EOS)
        ++i;
    return i;
}
```

- the test `(s[i] != EOS)` can be abbreviated to `(s[i])` since EOS == FALSE
- `s[i]` could be written `*(s+i)`, and
- what about a further abbreviation of the loop to

```c
int length(char *s) {
    int i = 0;
    while(s[i++]) ; // ERROR!
    return i;
}
```
void stringcopy(char *dst, char *src) {
    int i = 0;
    while(src[i] != EOS) { // copy chars
        dst[i] = src[i]; // assume that dst has room!
        i++;
    }
    dst[i] = src[i]; // copy EOS
}

Or:

void stringcopy(char *dst, char *src) {
    int i = 0;
    while((dst[i] = src[i]) != EOS) { // copy all!!
        i++;
    }
}

Copying (including EOS!) takes place before the test
Actually, the explicit testing against EOS is superfluous as well!

```c
void stringcopy(char *dst, char *src) {
    while(*(dst++) = *(src++)) ; // copy, update, test
}
```

- `*dst = *src;` copy current character `*src` into `dst`
- `src++;` continue with next char from `src`
- `dst++;` to be copied into next position in `dst`
- any expression has a value and the value of
  `*(dst++) = *(src++)`
  is the value of its lefthand side i.e. the character that has just been copied and if this character is EOS the loop exits on FALSE!
Here is a surprise...assume the following code

```c
char *s = "The string to be copied.";
char *t = "The string to receive a copy.";
int s_length = length(s);       // yields 24 I believe
int t_length = length(t);       // yields 29 I believe
printf("string s reads: %s\n", s);
printf("string t reads: %s\n", t);
stringcopy(t, s);
printf("string s reads: %s\n", s);
printf("string t reads: %s\n", t);
```

Then what do you expect for output from string t?
int stringcompare(char* s, char* t) {
    while (*s == *t) {
        if (*s == EOS) // equal and eos
            return 0;
        s++; t++;
    }
    return ((int)(*s - *t)); // nog effe sjekke
}
Suppose string $s$ has enough memory to append the characters of string $t$

```c
void stringconcat(char s[], char t[]) {
    int i = 0, j = 0;
    for (; s[i]; i++) ; // or i = strlen(s)
    for (; s[i] = t[j]; i++, j++) ;
}
```

Or:

```c
void stringconcat(char s[], char t[]) {
    while (*s) s++;
    while (*s++ = *t++) ;
}
```

What about: ?

```c
void stringconcat(char s[], char t[]) {
    while (*s++) ;
    while (*s++ = *t++) ;
}
```
Translating some well-known problems

It is an useful exercise to translate some well-known programming problems into the language of pointers and pointer arithmetic

- selection sort
- insertion sort
- binary search, recursive version
- binary search, iterative version
Selection sort

- swap a[0] with smallest elt in a[0..n)
- sort the remaining part a[1..n)

```c
void selection_sort(int *a, int n) {
    if (n) {
        swap(a, min(a, a + n - 1));
        selection_sort(++a, --n); // continue recursively
    }
}
int *min(int *a, int *b) {
    return((a == b) ? b : min2(a, min(a + 1, b)));
}
int *min2(int *a, int *b) {
    return((*a < *b) ? a : b);
}
void swap(int *a, int *b) {
    int c = *a; *a = *b; *b = c;
}
```
**Insertion sort**

- if a[0..m) is sorted shift/rotate a[m] to the left until it arrives at its proper location
- apply the shift/rotation for m = 2 .. n-1

```c
void insertion_sort(int *a, int n) {
    int *m = a, // ptr to first element
        *z = a + n; // ptr beyond the last elt
    while (++m < z) { // process [m..z)
        int *i = a;
        while (*i < *m) i++; // get lbnd for shift
        rotate(i, m); // perform rotation
    }
}

void rotate(int *lbnd, int *ubnd) {
    int cur = *ubnd; // value to investigate
    while (ubnd-- > lbnd)
        *(ubnd + 1) = *ubnd; // shift elements
    *(ubnd + 1) = cur; // put cur in empty loc
}
```
Binary search (recursive)

- divide search domain into equal parts
- decide where the search must continue

```c
int *bins_rec(int to_be_found, int *lbnd, int *ubnd) {
    int *mid = NULL;
    if (lbnd > ubnd)
        return NULL;
    mid = lbnd + (ubnd - lbnd) / 2;
    switch (compare(to_be_found, *mid)) {
        case -1: return (bins_rec(to_be_found, lbnd, mid - 1));
        case 0: return (mid);
        case +1: return (bins_rec(to_be_found, mid + 1, ubnd));
    }
}

int compare(int a, int b) {
    if (a < b) return -1;
    if (a > b) return +1;
    return 0;
}
```
Binary search (iterative)

```c
int *bins_iter(int to_be_found, int *lbnd, int *ubnd) {
    int *mid = NULL;
    while (lbnd <= ubnd) {
        mid = lbnd + (ubnd - lbnd) / 2;
        switch (compare(to_be_found, *mid)) {
            case -1: ubnd = mid - 1; break;
            case  0: return (mid);
            case +1: lbnd = mid + 1; break;
        }
    }
    return NULL;
}

int compare(int a, int b) {
    if (a < b) return(-1);
    if (a > b) return(+1);
    return 0;
}
```
And now for something completely different...

Memory management!

- When you declare a simple variable, \texttt{int n;}
- or a struct with several members,
  \begin{verbatim}
  typedef struct multi {
      int digits[4096];
      int msd;
  } multi m1, m2;
  \end{verbatim}
- or some big arrays,
  \begin{verbatim}
  double sin_values[32768],
  cos_values[32768];
  \end{verbatim}

the compiler automatically allocates the correct amount of memory. This is called \texttt{static-} or \texttt{compile-time memory allocation}
Memory management (cont’d)

Run-time memory allocation: It is frequently desirable, even necessary, to allocate memory while the program is running

typedef ... key_type;
typedef ... data_type;
typedef struct {
    key_type key;
    data_type data;
} elt_type;

elt_type e; // var e of type elt_type
elt_type *p; // var p of type ptr to elt_type
    // p is not initialized, it is an invalid pointer

Previously we have seen

- p = NULL; // initializes p to NULL
- p = &e; // and now p refers to element e
Memory management (cont’d)

Dynamic memory allocation

- determine the size of memory to which the pointer should point, e.g. `1 * sizeof(elt_type)`
- request that amount of memory
  
  `malloc(1 * sizeof(elt_type))`

  ```c
  p = malloc(1 * sizeof(elt_type));
  p = calloc(1, sizeof(elt_type)); // alternative
  ```

- `malloc()` returns a pointer to dirty memory
- `calloc()` returns a pointer to clean memory (all bits of all bytes are 0)
- the type of pointer returned by the functions `malloc()` and `calloc()` is called the generic pointer type i.e. `void *` or `pointer to void`
- A memory request can fail (the NULL pointer is returned): **Always validate the result of your memory requests!**
Memory is a finite resource: deallocating memory

- `free(p)` deallocates the memory pointed to by `p`
- Pointer `p` is invalid now, but the run-time system may not reclaim the memory immediately (you might still be able to use `p` for some time!)
- Use `free(p); p = NULL;` to avoid these errors: now you will get an "cannot dereference a null pointer" error
Memory management (cont’d)

An dynamic memory allocation example

```c
elt_type *copy_elts(elt_type *p, int lbnd, int ubnd) {
    elt_type *cp = malloc((ubnd - lbnd + 1) * sizeof(elt_type));
    if (cp) { // got the memory
        int i;
        for (i = 0 ; i <= ubnd - lbnd; i++)
            cp[i] = *(p + lbnd + i); // or p[lbnd + i];
    }
    return cp;
}
...
elt_type *elts, *copy;
// elts is filled, used, processed ... and
// now we need a copy of the segment elts[lbn..ubnd]
copy = copy_elts(elts, 18, 54);
if (!copy) {
    printf("Copy failed! Think hard!\n");
    // see also perror() ("man perror")
    exit(1);
}
```

Memory management (cont’d)

The function `realloc()`

```c
int cur_size = 32;
extype *elts = malloc (cur_size * sizeof(elt_type));  
// elts is filled, used, processed, ...
// and now we need to enlarge it
// increase the memory by an additional 24 elements
cur_size += 24;
elts = realloc(elts, cur_size * sizeof(elt_type));
...
// increase the memory by a factor of 2
elts = realloc(elts, (cur_size *= 2) * sizeof(elt_type));
```

- Enlarge or shorten a dynamically allocated array
- The function imports a pointer to "an old block" of memory, and an integer that determines the size of "a new block" of memory
- It Allocate the new block and copies the contents of the old block into the new block
Revisiting call-by-reference

```c
void f1(res_type *r1); // prototype for function f1
    // parameter r1 is ptr to res_type

int main(void) {
    res_type theResult1; // actual theResult1 is res_type
    ...
    f1(&theResult1); // invocation of the function f1
        // actual expression is ptr to res_type

Now, if the result that must be passed back is itself a pointer variable

void f2(res_type **r2); // prototype for function f2
    // r2 is ptr to ptr to res_type

int main(void) {
    res_type* theResult2; // theResult2 is ptr to res_type
    ...
    f2(&theResult2); // invocation of the function f2
        // expression is ptr to ptr to res_type
    ...
}

Now we have a **double indirection or pointer to pointer**
Pointers to pointers are common in C. For example, create a list of words and sort the list alphabetically.

- **word** can be represented as an array of characters: type `char *`
- **wordlist** can be represented as an array of words: type `char **`
- sorting this list passing it by reference to a function requires a formal parameter of type `char ***`

Some function prototype examples:

```c
void fill_wordlist(char ***wlist, int *length);
void sort_wordlist(char ***wlist, int length);
void show_wordlist(char **wlist, int length);
```
Example: a word list

Some design decisions

- A word consists of letters (’a’ .. ’z’) and the length of a word is always less than 64
- A word is of type pointer to char and is represented as a string (terminated by ’\0’)
- Words are stored in the minimal necessary amount of memory
- A wordlist consists of words and the size of the list is always less than or equal to its capacity
- A wordlist is created with some small initial size, then if more words are coming in the capacity is enlarged, and finally the list is trimmed down to the minimal amount of memory
- The user indicates end of input by typing ’!’ (called EOI), (s)he enters zero or more words terminated by return (called EOL) and after each EOL (s)he is asked to continue or to terminate input
# Example: a word list (cont’d)

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

#define EOL   '\n'
#define EOS   '\0'
#define EOI   '!'
#define initial_capacity  5
#define buffer_size       64

typedef char*  word;
typedef word* wordlist;

void fill_wordlist (wordlist *wl, int *n);  // 1st par. is char ***!
void prompt_for_input(void) ;
void eol_message(void) ;
void get_word(int c, word w) ;
void store_word(word w, wordlist wl, int i) ;
void show_wordlist (wordlist wl, int n);
void sort_wordlist (wordlist wl, int n);
void free_wordlist (wordlist *wl, int *n);
```
Example: a word list (cont’d)

```c
int main(void) {
    wordlist the_wordlist = malloc(initial_capacity * sizeof(word));
    int wordlist_size = initial_capacity;

    printf("FILL.\n");
    fill_wordlist(&the_wordlist, &wordlist_size);
    show_wordlist( the_wordlist, wordlist_size);
    printf("SORT.\n");
    sort_wordlist( the_wordlist, wordlist_size);
    show_wordlist( the_wordlist, wordlist_size);
    printf("FREE.\n");
    free_wordlist(&the_wordlist, &wordlist_size);
    show_wordlist( the_wordlist, wordlist_size);
    return 0;
}
```
Example: a word list (cont’d)

```c
void prompt_for_input(void) {} // instructions for the user
void eol_message(void) {} // continue or terminate input?
void get_word(int c, word w) {}

void store_word(word w, wordlist wl, int i) {
    wl[i] = malloc((1+strlen(w)) * sizeof(char));
    strcpy(wl[i], w);
}

void show_wordlist(wordlist wl, int n) {
    int i;
    if (n == 0) {
        printf("Empty wordlist!\n");
        return;
    }
    for (i = 0; i < n; i++)
        printf("%s\n", wl[i]);
}
```
Example: a word list (cont’d)

```c
void fill_wordlist(wordlist *wl, int *n) {
    int cur_size = *n, word_count = 0, cur_char;
    word buffer = malloc(buffer_size * sizeof(char));
    prompt_for_input();
    while((cur_char = getchar()) != EOI) {
        if (cur_char == EOL)
            eol_message();
        else {
            get_word(cur_char, buffer);
            if (word_count == cur_size)
                *wl = realloc(*wl, (cur_size *= 2) * sizeof(word));
            store_word(buffer, *wl, word_count++);
        }
    }
    free(buffer);
    if (cur_size > word_count)
        *wl = realloc(*wl, word_count * sizeof(word));
    *n = word_count;
}
```
Example: a word list (cont’d)

```c
void free_wordlist(wordlist *wl, int *n) {
    word w;
    while (*n) { // first release all words
        w = *(*wl + --(*n)); // get word (from z to a)
        free(w); // free word
    }
    free(*wl); *wl = NULL; // then release the list
}

void sort_wordlist(wordlist wl, int n) { // selection sort!!
    word *a = wl, *z = wl + n - 1; // first and last word
    for (; a < z; a++) {
        word *min = a, // init min temporarily
            *k = a+1, // start search for min
            w; // used for the swap
        for (; k <= z; k++) // get actual min
            if (strcmp(*min, *k) > 0) // libfunction strcmp
                min = k;
        w = *min; *min = *a; *a = w; // swap a and min(a+1..z)
    }
}
```
Dangling pointers

- A pointer that has never been initialized (elt_type *p; or a pointer that refers to memory that has been free’d (free(p);) is called a **dangling pointer**

- Dangling pointers point somewhere, but the memory is not yours!

- They often cause the infamous "Segmentation fault" errors

Some causes for dangling pointer problems

- Letting a global pointer p refer to a local function variable

```c
int *p;
void fun f(...) {
    int i = 13;
    int p = &i;
    ...
}
```
Dangling pointers (cont’d)

- Letting a function return a pointer to local data

```c
int *fun(...) {  
    int a[32];  
    ...  
    return a;  
}
int *res = fun(...);
```

- Letting a function return the address of local data

```c
elt_type *fun(...) {  
    elt_type e;  
    ...  
    return(&e);  
}
elt_type *pe = fun(...);
```
A memory leak occurs when memory has been dynamically allocated, is no longer in use, but has not been free’d.

Memory leaks often occur when nested structures must be free’d, e.g.

- top level: a dynamic array of pointers to elements,
- sub level: elements are dynamically allocated blocks of memory where actual data are stored

Memory must be released bottom up!
void free_wordlist(wordlist *wl, int *n) {
    free(*wl);
    *wl = NULL;
    *n = 0;
}

produces a huge memory leak!

void free_wordlist(wordlist *wl, int *n) {
    word w;
    while (*n) {
        w = *(*wl + --(*n));
        free(w);
    }
    free(*wl); *wl = NULL;
    *n = 0;
}

is correct!
Arrays and structs revisited

- Arrays are reference types...why? Performance!
  - Copying a large array (e.g. function argument) takes time
  - Copying a pointer is basically for free
- The same holds for structs, which can also be BIG (e.g. structs containing arrays!)
  - But structs are value types...
  - You can copy structs by direct assignment
Arrays and structs revisited (cont’d)

Consider the some code for the support for extremely large numbers (multi’s)

```c
#define MAX_DIGITS 16384
typedef struct multi {
    int J[MAX_DIGITS];
    int msd;
} multi; // sizeof(multi) == 65540 bytes!!!

multi m_init (int n); // translate n to multi
void m_show (multi *m); // by reference?? YES!!!

int main(void) {
    multi f, cf;
    f = m_init(123456789); m_show(&f); // f: 1 2345 6789
    cf = f;
    m_show(&cf); // cf: 1 2345 6789
    cf.J[0] = 0;
    m_show(&cf); // cf: 1 2345 0000
    m_show(&f); // f: 1 2345 6789
    return 0;
}
```
Arrays and structs revisited (cont’d)

```c
multi m_init (int n) {
    multi m = {{0}, -1};
    do {
        m.J[++(m.msd)] = n % RADIX;    // RADIX == 10000
        n /= RADIX;
    } while (n);
    return m;
}

void m_show (const multi *m) {
    int j = 0, field_width = 5;
    while (j <= (*m).msd) {
        print_digit((*m).J[(*m).msd - j], field_width);
        if (++j % N_COLUMNS == 0)
            new_line;
    }
}
```

- Explain why call by reference has been used!
- Explain the use of const pointer to const data!
- Together: efficiency using call by ref and protection using const

Datastructures / Programming in C – A.D. Pimentel – p. 144/226
Arrays and structs revisited (cont’d)

Now suppose however that our type multi was defined as

```c
typedef struct multi {
    int *J;
    int msd;
} multi;    // sizeof(multi) == 8 bytes
```

and that `m_init()` has been rewritten accordingly

```c
int main(void) {
    multi f, cf;
    f = m_init(123456789); m_show(&f); // f: 1 2345 6789
    cf = f;
    m_show(&cf);    // cf: 1 2345 6789
    cf.J[0] = 0;
    m_show(&cf);    // cf: 1 2345 0000
    m_show(&f);     // f: 1 2345 0000

    return 0;
}
```

The memory to which `J` points is shared by `f.J` and `cf.J`!
Let's assume

typedef struct {
    key_type key;
    data_type data;
} elt_type e, *p;

- \( e.key \) and \( e.data \) works as usual

- \( (*p).key \) and \( (*p).data \)

  the parentheses are important for the priority properties of
  the dereferencing- and member-access operator

- To avoid these ugly expressions an abbreviated notation has
  been introduced: \( p->\text{member} \) has exactly the same
  meaning as \( (*p).\text{member} \)
Arrays of pointer to struct

- **Static array of pointer to struct**

```c
elt_type *static_aps[MAX_SIZE];
```

declares an array `static_aps` that contains `MAX_SIZE` pointers to `elt_type`

- **Dynamic array of pointer to struct**

```c
elt_type **dynamic_aps = calloc(INITIAL_SIZE, sizeof(elt_type *));
```

declares an array `dynamic_aps` that contains `INITIAL_SIZE` pointers to `elt_type` and that can be resized run-time

- `dynamic_aps[0]->_key = ...`
An example

```c
#define STUDENTS 128
typedef struct data {
    int coll_krtnr;
    ...
} data;
typedef struct student {
    key student_registration;
    data *student_information;
} student;
typedef student** course;
...
course ProgI, ProgII;
int i;
ProgII = malloc(STUDENTS * sizeof(student *));
for (i = 0; i < STUDENTS; i++) {
    ProgII[i] = malloc(sizeof(student));
    ProgII[i]->student_registration = 0;
    ProgII[i]->student_information = calloc(1, sizeof(data));
}
ProgII[0]->student_information->coll_krtnr = ...;
```
Validate memory requests!

- Always validate whether or not your memory request (e.g., malloc(), calloc(), etc.) was succesful!
- In case of an error, the perror() function can be used: it will specify the encountered error in the last system-call made

```c
elt_type *dynamic_aps;

dynamic_aps = calloc(SIZE, sizeof(elt_type));
if (dynamic_aps == NULL) {
    perror("Malloc in function XXX");
    exit(0);
}
```
I/O functions revisited

Some primitive I/O functions in C are:

- `fopen()` and `fclose()` // Opening/closing files
- `tmpfile()`, `rename()` and `remove()` // File removal/renaming
- `getc() and putc()`, `fgetc()` and `fputc()` // Read/write single character from IO stream
- `fgets()` and `fputs()` // Read/write string from IO stream
- `printf()`, `scanf()`, `fprintf()` and `fscanf()` // Formatted input/output
- `fseek()` and `ftell()` // file positioning
- `fread()` and `fwrite()` // binary reading/writing
I/O functions revisited (cont’d)

- `fopen()` and `tmpfile()` return a file pointer
  - when the operation fails the pointer has value NULL otherwise the pointer is well-defined (and refers to the file)
- Functions such as `fclose()`, `rename()`, `remove()`, `fflush()`, `fseek()`, return an integer value that indicates success/failure
- Functions such as `getc()`, `fgetc()`, `putc()` and `fputc()` return an integer that represents the character or EOF (a negative integer)
- Functions from the `scanf()` family return an integer that represents the number of arguments that are stored otherwise EOF
- Functions from the `printf()` family return an integer that represents the number of characters that are written otherwise -1
- Functions `fread()` / `fwrite()` return an integer that represents the number of elements read / written
A file I/O example

- Always validate the return value of an I/O operation
- You will see that 80% of the I/O code consists of validation
- Transfer of failure to the calling environment (simulate something like the error traces in Java): can be of great help when debugging your program

```c
int RET2EOL(char *org_name, char *dst_name) {
    FILE *org_ptr, *dst_ptr;
    int c;
    if ((org_ptr = fopen(org_name, "r")) == NULL) {
        perror("fopen");
        return 1;
    }
    if ((dst_ptr = fopen(dst_name, "w")) == NULL) {
        perror("fopen");
        return 1;
    }
}
```
A file I/O example (cont’d)

```c
while ((c = getc(org_ptr)) != EOF) {
    if (c == RET)
        c = EOL;
    if (putc(c, dst_ptr) == EOF) {
        perror("putc"); return 1;
    }
}
if (fclose(dst_ptr)) {
    perror ("fclose"); return 1;
}
if (fclose(org_ptr)) {
    perror("fclose"); return 1;
    return 0;
}
...
int failed = RET2EOL("opg6BILLY.c", "opgave6.c");
if (!failed)
    submit("opgave6.c");
```
We recall functions that return pointers, e.g.

```
elt *lin_search(elt to_be_found, elt *es, int lbnd, int ubnd) {
    int i;
    for (i = lbnd; i <= ubnd; i++)
        if (equal_elts(*(es + i), to_be_found))
            return (es + i);
    return NULL;
}
```

typedef char* string;

```
string str_copy(string dst, const string src) {
    char *p = dst;
    while(*p++ = *src++);
    return dst;
}
```
Functions and pointers (cont’d)

Pointers to functions:

- There is a pointer associated with each function
- Logically this pointer points to the base address of the code of the function
- The name of a function is a constant pointer in exactly the same spirit as the name of an array

```c
double fun(double x); // prototype of fun
double fun(double x) {} // definition of fun
double (*fun_ptr) (double x); // ptr to function
```

declares a pointer (fun_ptr) that might refer to the function fun, the type of this pointer is double (*) (double) and its name is fun_ptr
Pointers to functions (cont’d)

Important note on syntax:

```c
double *fun_ptr (double x);
```
declares a function prototype! But

```c
double (*fun_ptr) (double x);
```
declares a pointer `fun_ptr` that can refer to any function `double f(double);

In general, if function `f` has the following signature

```c
return_type f(parameters_list)
```
then the corresponding pointer type `pf` is given by

```c
return_type (*)(parameters_list)
```
and this type `pf` can be defined explicitly by

```c
typedef return_type (*)(parameters_list)
```
Pointers to functions (cont’d)

After the assignment
fun_ptr = fun;
the pointer will indeed point to (the code of) the function fun

- Pointers to functions allow functions to have parameters that are functions again
- Such functions are called higher-order functions
- They can be implemented without knowledge about the details of the function(s) that they are supposed to execute!
Higher-order functions

Example: the function `tabulate`

```c
void tabulate(double (*f)(double),
    double lbnd, double ubnd, double delta) {
    double x;
    for (x = lbnd; x <= ubnd; x += delta)
        printf("%12.8f\t%12.8f\n", x, (*f)(x));
}
```

prints a table of values $x$ against $f(x)$, for $x$ from $lbnd$ to $ubnd$ in steps $delta$, for any function of type `double` $f$(`double`) 

How?
Higher-order functions (cont’d)

Let

```c
double test(double x) {
    return(sqr(sin(x)) / log(x));
}
```

denote the function that should be tabulated; then call

```c
tabulate(test, 2.0, 4.0, 0.1);
```

Syntax: ANSI C allows us to write

- `double f(double)` for the function parameter `double`
- `(*f)(double)`
- and in the code we can use `f(x)` instead of `(*f)(x)`
Higher-order functions (cont’d)

E.g. the square root for any real positive x using Newton’s method: if y is an approximation to sqrt(x) then \( (y + x/y)/2 \) is a better one

```c
double newton(double x, double y) {
    return((y + x/y) / 2);
}

double sqrt_of(double x, double eps,
               double f(double, double)) {
    double approx = x; // first approx to sqrt(x) is x itself
    while(fabs(x - approx * approx) > eps)
        approx = f(x, approx);
    return approx;
}

Then use function `newton` in an actual call to `sqrt_of`:

```c
int main(void) {
    double pi = 4 * atan(1), sqrt_pi;
    sqrt_pi = sqrt_of(pi, 1E-12, newton);
    printf("sqrt(pi) = %12.8f\n", sqrt_pi);
}
```
Higher-order functions (cont’d)

Function parameters and the typedef mechanism (as discussed above):

```c
typedef double (*pf) (double, double);
```

declares the type `pf` as pointer to function that returns a double and imports two doubles. Using this new type `pf` we can now write

```c
double sqrt_of(double x, double eps, pf f);
// determine sqrt(x) within precision eps using function f
```

There is a lot more to say about higher-order functions:

- The concept is powerful and leads to clear, clean and concise code
- Read any textbook on functional programming
- Java did not inherit the concept of formal function parameters from C
- In Java anonymous inner classes are designed for this purpose
- In the discussion on lists and trees we will apply higher-order functions
Some notes about performance

- The C language is a reasonably efficient language
- Using the "-O" compiler options, you can play with compiler optimizations
- Keep the "90/10" rule in mind!
  - 90% of the time is spent in 10% of the code
- Algorithmic optimizations pay off much more than compiler optimizations
- Let’s give an example...
A prime is a number \( p \) that has no divisors other than 1 and itself.

A twin prime is a pair of numbers of the form \( tpn = (n-1, n+1) \) such that both \( n-1 \) and \( n+1 \) are prime.

Problem: determine all twin primes in a given interval \([lbnd .. ubnd]\)
Twin primes: top down design

Main

declare interval bounds lbnd and ubnd
and require that lbnd < ubnd-1

1.1 get values for lbnd and ubnd from keyboard
1.2 determine twinprimes in interval from lbnd to ubnd

Subproblem 1.1: getInterval(&lbnd, &ubnd);

the user of this program enters the lower- and upperbound of the interval from the keyboard
1.1.1 message "enter lowerbound please"
1.1.2 get lbnd

do

message "enter upperbound please"
get ubnd

while (ubnd < lbnd+2)
Twin primes: top down design (cont’d)

Thus:

```c
void getInterval(int *lbnd, int *ubnd) {
    printf("enter lowerbound please\n");
    scanf(lbnd);
    do {
        printf("enter upperbound please\n");
        scanf(ubnd);
    } while(*ubnd < *lbnd+2);
}
```

Subproblem 1.2

```c
void getTwinPrimes(int lbnd, int ubnd) {
    int n;
    for (n = lbnd + 1; n < ubnd; n++) // performance??
        if (isTwinPrime(n))
            print result [n-1, n+1]
}
```
Twin primes: top down design (cont’d)

Subproblem 1.2.1

```c
int isTwinPrime(int n) {
    return (isPrime(n-1) && isPrime(n+1)); // subproblem 1.2.1.1
}
```

Subproblem 1.2.1.1

```c
int isPrime(int k) {
    d = 2;
    while (d < k) { // performance??
        if (isDivisor(d, k)) // subproblem 1.2.1.1.1
            return 0;
        ++d;
    }
    return 1;
}
```

Subproblem 1.2.1.1.1

```c
int isDivisor(int d, int k) { return (k % d == 0); }
```

OK: now you can finish the implementation and you can test the program. However, what about the performance of your program?!
In while loop in isPrime(k) divisibility is checked for all d = 2, 3, 4,...k-1. After some thought it becomes clear that

- d can be restricted by d*d <= k: if d divides k then e = k/d divides k too
- if you first perform a check on k being even you can improve performance by an additional factor of 2

```c
int isPrime(int k) {
    if (isDivisor(2, k)) // subproblem 1.2.1.1.1
        return 0;
    else {
        int d = 3;
        while (d*d <= k) {
            if (isDivisor(d, k)) // subproblem 1.2.1.1.1
                return 0;
            d += 2;
        }
    }
    return 1;
}
```
Twin primes: performance

In `getTwinPrimes(lbnd, ubnd)`, for each `n` from `lbnd+1` to `ubnd-1` we perform the test `isPrime(n-1) && isPrime(n+1)`.

We can make this loop 6 times as fast!

- Consider numbers `6*k-2`, `6*k-1`, `6*k`, `6*k+1`, `6*k+2` and `6*k+3`.
- Only `6*k-1` and `6*k+1` can possibly be prime.
- Twinprimes must be of the form `(6*k-1, 6*k+1)`.
- Start loop with the smallest 6-fold larger than `lbnd`, and end it with the largest 6-fold smaller than `ubnd`, and use stepsize 6.

```c
void getTwinPrimes(int lbnd, int ubnd) {
    int n, lbnd6 = lbnd + 6 - lbnd%6,
        ubnd6 = ubnd - 6 + ubnd%6;
    for (n = lbnd6; n <= ubnd6; n += 6)
        if (isTwinPrime(n))
            printf("[%d, %d]\n", n-1, n+1);
}
```
Towards datastructures

- A struct can be defined to contain members that are of type pointer to any type
- If this type is again the same structure template the structure is called a selfreferential structure

First define standard elements as structs with a unique key and data:

typedef ... key_type; // user defined and comparable!
typedef ... data_type; // user defined
typedef data_type* data_ptr;
struct elt {
    key_type key;
    data_type data;
};
typedef struct elt elt_type;
typedef elt_type* elt_ptr;
Towards datastructures (cont’d)

An alternative element is a struct in which the second member is a pointer to the data instead of the data itself:

```c
struct elt {
    key_type   key;
    data_ptr   data;
};
typedef struct elt    elt_type;
typedef elt_type*     elt_ptr;
```

\[
\begin{array}{c}
K \rightarrow D \\
\text{dataPtr pD = &D;} \\
\text{eltType E;} \\
E.\text{key} = K; \ E.\text{dataP} = pD; \\
\text{eltPtr pE = &E;}
\end{array}
\]
Towards datastructures (cont’d)

We can define nodes that consist of a combination of an element and one or more pointers to other nodes. The types `x_node` and `x_ptr` are introduced (and for the moment $x$ stands for list or stack or queue, etc.)

```c
struct node {
    elt_type elt;
    struct node *p1;
    struct node *p2;
    ...
    struct node *pn;
};
typedef struct node x_node;
typedef x_node* x_ptr;
```

```c
eltPtr pE = &E;
xNode xN;
xNode.eltP = pE;
/* pointer fields p1..pn */
xPtr xP = &xN;
```
Towards datastructures (cont’d)

Alternatively:

```c
struct node {
    elt_ptr elt;
    struct node *p1;
    struct node *p2;
    ...
    struct node *pn;
};

typedef struct node x_node;
typedef x_node* x_ptr;
```

```
eltPtr pE = &E;
xNode xN;
xNode.eltP = pE;
/* pointer fields p1..pn */
xPtr xP = &xN;
```
Towards datastructures (cont’d)

Or:

```c
struct node {
    key_type key;
    data_ptr data;
    struct node *p1;
    struct node *p2;
    ...
    struct node *pn;
};
typedef struct node x_node;
typedef x_node* x_ptr;
```

```c
dataPtr pD = &D;
xNode xN;
xN.key = K;
xN.dataP = pD;
/* pointer fields p1..pn */
xPtr xP = &xN;
```
Towards datastructures (cont’d)

Finally we define headernodes that consist of a combination of pointers that give access to the datastructure and provide global information about the current status. For example:

```c
struct x_header {
    x_ptr first;
    x_ptr last;
    int size;
};
```

represents a header node that contains two access pointers and an integer to store the number of nodes currently present in the datastructure.

Declaration without header node:

```c
typedef x_ptr x; // x stands for list stack or tree etc
x myX, yourX; // variables of type x
```

Declaration with header node:

```c
typedef struct x_header* x; // x stands for list stack tree
x myX, yourX; // variables of type x
```
"Constructors"

It is convenient to have "constructors" for elements and for nodes. Constructor that returns an element:

```c
elt_type new_elt(key_type k, data_type d) {
    elt_type e;
    e.key = k; e.data = d;
    return e;
}
```

Constructor that returns a pointer to an element element:

```c
elt_ptr new_elt(key_type k, data_type d) {
    elt_ptr pe = malloc(sizeof(elt_type)); // !!val
    pe->key = k; pe->data = d;
    return pe;
}
```
"Constructors" (cont’d)

Constructor that returns a pointer to a node that contains element e:

```c
x_ptr new_node(elt_type e) {
    x_ptr px = malloc(sizeof(x_node)); // !!validate
    px->elt = e;
    return px;
}
```

Constructor that returns a pointer to a node that contains pointer to elt:

```c
x_ptr new_node(elt_ptr pE) {
    x_ptr px = malloc(sizeof(x_node)); // !!validate
    px->eltP = pE;
    return px;
}
```

With the building blocks element and node we will construct several **dynamic datastructures** where nodes are linked together using pointers.
Lists: a definition

The most convenient way to define the datatype LIST is inductively:

- The list is empty, \( L = [ ] \), or
- The list is not empty, \( L = [ H : T ] \)

  - \( H \) denotes the first node (often called the head)
  - \( T \) denotes the remainder (often called the tail) of the list

- An empty list has no head and no tail
- The tail of a non-empty list is a list. Think about this definition!

Some basic LIST operations together with the axioms that must be satisfied:

- Create a new and empty list: \( create() = [ ] \)
- Add element \( E \) at the head of an existing list: \( insert(E, L) = [E:L] \)
- Remove the head from an existing non-empty list: \( delete(L) = T \) if \( L = [H:T] \)
Lists: axioms

- \text{is\_empty(create())} = \text{TRUE}
- \text{is\_empty(insert(E,L))} = \text{FALSE}
- \text{head(create())} = \text{ERROR} \ (\text{error in head: empty list})
- \text{tail(create())} = \text{ERROR} \ (\text{error in tail: empty list})
- \text{delete(create())} = \text{ERROR} \ (\text{error in delete: empty list})
- \text{head(insert(E,L))} = E
- \text{tail(insert(E,L))} = L
- \text{delete(insert(E, L))} = L
A possible (incomplete) list interface in C:

```c
list create_list();
list destroy_list(list L);   // YES!
bool is_empty_list(list L);
bool is_full_list(list L);   // YES!
int  size_of_list(list L);
list insert(elt_ptr pe, list L); // OR ...
list delete(list L);         // OR ...

data_ptr get_data(key_type k, list L);
list  set_data(key_type k, data_ptr pd, list L);
list  copy_list(list L);
list  revert_list(list L);
list  sort_list(list L);
void  traverse_list(list L);
list  process_list(list L, elt_ptr f(elt_ptr));
...
append, concat, merge, etc., etc.
```
Lists: implementation

We can now declare a list in terms of listnodes which are linked to each other. Each listnode stores an element and at least one pointer. This pointer points to the next node. This basic design is called a Singly Linked List (SLL).

```c
struct node {
    elt_ptr elt; // field to store pointer to element
    struct node* next // pointer to the next node
};
typedef struct node list_node;
typedef list_node* list_ptr;
typedef list_ptr list;
list L; // L is a variable of type list
    // L is a pointer to the first list node (if any)
```
Often it is convenient to introduce a header node that contains global information (size and one or more access pointers) about the list.

```c
struct node {
    elt_ptr elt; // field to store an element
    struct node* next // pointer to the next node
};
typedef struct node list_node;
typedef list_node* list_ptr;
struct header {
    list_ptr first, cur, last;
    int size;
};
typedef struct header list_header;
typedef list_header* list;
list L; // L is a pointer to the header node
```
SLL: implementation (cont’d)

If L is a variable of type list then

● If this is a list without header node, then L == NULL represents the empty list

● If this is a list with a header node, then L->first == L->cur == L->last == NULL and L->size == 0 represent the empty list!

Some SLL operations:

```c
list_ptr new_node(elt_ptr pe) {
    list_ptr temp = malloc(sizeof(list_node)); // !!val
    temp->elt = pe;
    temp->next = NULL; // set the next field to NULL!!
    return temp;
}
...
```

```
temp = new_node(pe);
```
A basic insert operation in which an existing list gets a new head. If the SLL is defined with a header node, the size field is updated in the epilogue.

```c
1  temp = newNode(E);
2  temp->next = first;
3  first = temp;
epilogue;
```
The second insert operation inserts a new node as the successor of the current node. If the SLL is defined with a header node the size field is updated in the epilogue. The case where the current node happens to be the last node is treated as a special case.
SLL: implementation (cont’d)

Insertion of a new last node. The case `last == NULL` is again a special case where the list is empty, prior to the insert operation.

```c
if (last)
1 temp = newNode(E);
2 last->next = temp;
3 last = temp;
    epilogue;
else SpecialCase;
```
The basic delete operation is the deletion of the first node of the list and in the second delete operation we remove the successor of cur.
SLL: implementation (cont’d)

SLL: cheap and expensive operations ...
insert predecessor of cur, delete cur, delete predecessor of cur and in particular
delete last node are expensive operations!

Variations on SLL:

- Cyclic Singly Linked Lists (CSLL)

/* CSLL = Cyclic Singly Linked List */
/* with header node */
Variations on SLL (cont’d)

- Doubly Linked Lists (DLL)

- Cyclic Doubly Linked Lists (CDLL)
A list interface

- Have a clear list interface
  - Applying datastructure hiding and encapsulation
  - The list interface is a collection of functions (prototypes) that act on lists including the documentation for the programmer
  - Be consistent with parameters and return values
- Never let your function perform a more complex task than its name suggests!

An example list interface:
list new_list();
/* pre true
// action create a new empty list and return the new list
// res a new empty list exists
// usage list L = new_list(); */

list release_list(list L);
/* pre list L exists
// action returns a new empty list
// memory for all its elements is released as well!!
// res a new empty list
// usage list L = release_list(); */

bool is_empty_list(list L);
/* pre list L exists
// action determine whether list L is empty or not
// res true if the list is empty and false otherwise
// usage if (is_empty_list(L)) {...}
*/
bool is_full_list(list L); /* pre list L exists
// action determine whether list L is full or not
// res true if the list is full and false otherwise
// usage if (is_full_list(L)) {...}
*/

list insert_first(elt_ptr pe, list L); /* pre list L exists and pe refers to an element e
// action insert element e as the new first element of list L
// res a new list with first element e
*/

list insert_last(elt_ptr pe, list L); /* pre list L exists and pe refers to an element e
// action insert element e as the new last element of list L
// res a new list with last element e
*/
list delete_first(list L);
/* pre list L exists and is not empty
// action delete the first element of list L
// res the first element has been removed */
list delete_last(list L);
/* pre list L exists and is not empty
// action delete the last element of list L
// res the last element has been removed */
data_ptr get_data(key_type k, list L);
/* pre list L exists and is not empty
// action return the data of the element with key k
// if no such element exists then return NULL
// res pointer to the data characterized by key k
// if no such key exists then NULL */
A list interface (cont’d)

```c
list set_data(key_type k, data_ptr pd, list L);
/* pre list L exists and is not empty
// action replace the data of the element with key k
// by data d pointed at by pd
// if no element with key k exists then no action
// res list L after updating element with key k
*/

int size_of_list(list L);
/* pre list L exists
// action determine the number of elements of list L
// res the number of elements
*/

list copy_list(list L);
/* pre list L exists
// action construct a copy of list L and return the copy
// and this implies a copy of all elements as well!!
// res the original list still exists!!
// and the copy is returned by the function
*/
```
list sort_list(list L);
/* pre  list L exists
   // action rearrange the elements according to keyvalue
   // and return the sorted list
   // res  the sorted list is returned by the function */
void traverse_list(list L);
/* pre  list L exists and is not empty
   // action visit all nodes from first to last (and print keys)
   // res  true */
typedef elt_ptr (*process_elt) (elt_ptr); // new type!!
//process_elt is pointer to function from elt_ptr to elt_ptr!!
list process_list(list L, process_elt f);
/* pre  list L exists and is not empty
   // action visit all nodes from first to last
   // and process elements by application of function f
   // res  the processed list */
A list interface (cont’d)

- The function `process_list` is a higher-order function: All elements of the list are processed by the user-defined function \( f \).

- Now what remains is the actual choice for the list structure and the implementation of the functions that are listed in the above interface:
  - array, sll, csll, dll, cdll, ...?

- Design decisions for the datastructure:
  - A `listnode` is a struct that contains a pointer to an element and pointers to the previous and to the next node.
  - A `listpointer` is a pointer to a listnode.
  - A `listheader` is a struct that contains pointers to first and last node and contains the size of the list.
  - A `list` is a pointer to a listheader.

In C, the interface is written in the headerfile `list.h` while the implementation is written in the sourcefile `list.c`.
You can create archives or libraries from object files

Assume e.g. the list functionality is implemented in `listcreate.c`, `listprint.c`, `listsort.c` and `list.h`

```bash
gcc -std=c99 -c listcreate.c listprint.c listsort.c
```

```bash
ar rc list.a listcreate.o listprint.o listsort.o
```

```bash
ranlib list.a  # (on some systems `ar` can do this as well)
```

Now `list.a` is a static library (archive): `gcc -std=c99 -o foobar foo.o bar.o list.a`

Or rename `list.a` into `liblist.a`, and `gcc -std=c99 -o foobar foo.o bar.o -L. -llist`
A library intermezzo (cont’d)

Alternatively, one could build dynamic (or shared) libraries

```sh
gcc -fPIC -std=c99 -c listcreate.c listprint.c listsort.c
```

```sh
gcc -shared liblist.so listcreate.o listprint.o listsort.o
```

```sh
gcc -std=c99 -o foobar foo.o bar.o -L. -llist
```

Then, update the `LD_LIBRARY_PATH` environment variable to include the directory where the library resides
Stack: last in, first out!

The stack is a time-ordered list with a restricted set of operations. The time-order of the elements on the stack is fixed by their time of arrival and the only more recent element can be accessed: this is the top of the stack.

- **create** creates a new and empty stack: \( \text{create()} = [ ] \)
- **push** adds a new top (with element \( E \)) to the stack: \( \text{push}(E, S) = [E:S] \)
- **pop** removes the top from the stack: \( \text{pop}(S) = T \iff S = [E:T] \)
- **top** inspects the top of the stack: \( \text{top}(S) = E \iff S = [E:T] \)
Stack: last in, first out (cont’d)

Some axioms for the basic operations:

- \( \text{is\_empty(create())} = \text{true} \)
- \( \text{is\_empty(push(E, S))} = \text{false} \)
- \( \text{top(create())} = \text{error in top: empty stack} \)
- \( \text{pop(create())} = \text{error in pop: empty stack} \)
- \( \text{top(push(E, S))} = E \)
- \( \text{pop(push(E, S))} = S \)
A stack interface

A possible interface:

```c
stack create_stack(); // create a new empty stack
bool is_empty_stack(stack s); // check empty or not
bool is_full_stack(stack s); // check full or not
int size_of_stack(stack s);

stack push(elt_ptr e, stack s); // add e on top if not full
stack pop(stack s); // remove top if not empty
elt_ptr top(stack s); // get top if not empty
```

Or ...

```c
void push(elt_ptr e, stack s); // add e on top if not full
elt_ptr pop(stack s); // remove and retrieve top if not empty
elt_ptr top(stack s); // retrieve top (not remove) if not empty
```
Stack structure

```c
struct node {
    elt_ptr pe; // pointer to an element
    struct node *link; // link to previous node on stack
};
typedef struct node stack_node;
typedef stack_node* stack_ptr;

struct header {
    stack_ptr top; // pointer to the top of the stack
    int size; // number of nodes currently present
};
typedef struct header stack_header;
typedef stack_header* stack;
```

/* Stack as SLL*/

![Diagram of Stack as SLL]
Stack: an SLL implementation

```c
stack create_stack(void) {
    stack s;
    if ((s = malloc(sizeof(stack_header))) == NULL) {
        // error handling
    }
    s->top = NULL; s->size = 0; return s;
}
void push(elt_ptr e, stack s) {
    stack_ptr t;
    if (is_full_stack(s)) return;
    if ((t = malloc(sizeof(stack_node))) == NULL) {
        // error handling
    }
    t->pe = e; t->link = s->top; s->top = t; s->size++;
}
elt_ptr pop(stack s) {
    elt_ptr ep;
    stack_ptr t;
    if (is_empty_stack(s)) return NULL;
    t = s->top; s->top = t->link;
    ep = t->pe; free(t); s->size--;
    return ep;
}
```
We’ve seen queues already: elements arrive at the rear end of the queue and are served at the front (FIFO order)

- **create** creates a new and empty queue: \( \text{create}() = [ ] \)
- **enqueue** adds a new node at the rear end: \( \text{enqueue}(E, Q) = [Q:E] \)
- **dequeue** removes the node at the front: \( \text{dequeue}(Q) = T \iff Q = [E:T] \)
- **front** inspects the front end of a queue: \( \text{front}(Q) = E \iff Q = [E:T] \)
- **rear** inspects the rear end of a queue: \( \text{rear}(Q) = E \) if and only if \( Q = [T:E] \)
Queue axioms

- is_empty(create()) = true
- is_empty(enqueue(E,S)) = false
- front(create()) = error in front: empty queue
- front(enqueue(E,Q)) = if is_empty(Q) then E else front(Q)
- rear(create()) = error in rear: empty queue
- rear(enqueue(E,Q)) = E
- dequeue(create()) = error in dequeue: empty queue
- dequeue(enqueue(E,Q)) = if is_empty(Q) then create() else enqueue(E, dequeue(Q))
Queue: a possible interface

```c
queue create_queue(); // create a new empty queue

bool is_empty_queue(queue q); // check empty or not
bool is_full_queue(queue q); // check full or not

void insert(elt_ptr e, queue q); // add e to rear if not full
elt_ptr serve(queue q); // serve front if not empty
elt_ptr front(queue q); // get front if not empty
elt_ptr rear(queue q); // get rear if not empty
```
Queue implemented as SLL or CSLL with header. First, the SLL declaration:

```c
struct node {
    elt_ptr pe;
    struct node *link;
};
typedef struct node queue_node;
typedef queue_node* queue_node_ptr;

struct queue_header {
    queue_node_ptr front; // pointer to the front of the queue
    queue_node_ptr rear;  // pointer to the rear of the queue
    int size;            // number of nodes currently present
};
typedef struct queue_header* queue;

queue Q; // Q is a variable of type queue
          // Q is a pointer to the header node
```
Queue CSLL structure with rear pointer only

```c
struct node {
    elt_ptr pe;
    struct node *link;
};
typedef struct node queue_node;
typedef queue_node* queue_node_ptr;
struct queue_header {
    queue_node_ptr rear; // pointer to the rear of the queue
    int size; // number of nodes currently present
};
typedef struct queue_header* queue;
```
Interfaces: design decisions

list delete_first(list L); // return updated list
// OR
bool delete_first(list *L); // return success/failure
// OR
// return pointer to "unhooked" element
elt_ptr delete_first(list *L, bool *deleted);

These design decisions can have a great impact on implementation details...
Consider the simplest SLL declaration you can imagine:

```
typedef int elt_type; // elements are integer
typedef struct list_node* list_node_ptr;
typedef struct list_node {
    elt_type elt;
    list_node_ptr next;
} list_node;
typedef list_node_ptr list; // renaming
```

How to declare a method that inserts a new element as the last element of an existing list?

- **Declare a function**: `list insert_func(elt_type E, list L);`
- **Declare a procedure**: `void insert_proc(elt_type E, list *L);`
The second question is how to define a method that inserts a new element as the last element of an SLL? We do not have a pointer to the last node, so this node $p$ must be found. $p->next$ is equal to NULL and must be replaced by a pointer to the new node.

- Recursive implementation: the function calls itself in order to find the last node: see `insert_func_rec` and `insert_proc_rec`

- Iterative implementation: a for- or while-loop is used to find the last node: see `insert_func_iter` and `insert_proc_iter`
Function v.s. procedure

```c
list insert_func_rec(elt_type E, list L) {
    if (!L)
        L = new_list(E, NULL);
    else
        L->next = insert_func_rec(E, L->next); // recursion
    return L;
}
void insert_proc_rec(elt_type E, list *L) {
    if (!*L)
        *L = new_list(E, NULL);
    else
        insert_proc_rec (E, &(*L)->next); // recursion
}
```

where `new_list` imports element `E` and returns a pointer to a listnode with `next` equal to `NULL`. 
list insert_func_iter1(elt_type E, list L) {
    if (!L)
        L = new_list(E, NULL);
    else {
        list_node_ptr p = L; // p is ptr to list_node
        while (p->next)
            p = p->next; // loop exit: p refers to the last node
        p->next = new_list(E, NULL); // set p->next
    }
    return L;
}

list insert_func_iter2(elt_type E, list L) {
    listNodePtr *p = &L; // p is ptr to ptr to list_node!!
    while (*p)
        p = &(*p)->next; // loop exit: p is equal to the
        // address of next of last node
        // value of p ie *p is equal to NULL
    *p = new_list(E, NULL); // and now p gets its proper value
    return L;
}
Implementation variants (cont’d)

```c
void insert_proc_iter1(elt_type E, list *L) {
    if (!*L)
        *L = new_list(E, NULL);
    else {
        list_node_ptr p = *L; // p is ptr to list_node
        while (p->next)
            p = p->next;
        p->next = new_list(E, NULL);
    }
}

void insert_proc_iter2(elt_type E, list *L) {
    list_node_ptr *p = L; // p is ptr to ptr to list_node
    while (*p)
        p = &(*p)->next;
    *p = new_list(E, NULL);
}
```
Another part of the "toolkit" deals with hierarchical structures known as trees. The simplest tree is a **Binary Tree** (BT). A binary treenode has the same structure as a doubly linked list (DLL) node.

- Both nodes contain fields for an element and two pointers.
  - In a DLL node these pointers point to predecessor and successor nodes. This gives the datatype its linear structure.
  - The pointers in a BT node point to the descendants (or children) of the node (the parent). This gives a hierarchical structure.

We will discuss three types of trees:

- Binary Tree (BT): each node is allowed to have at most two children
- Binary Search Tree (BST): This type of tree is again a BT but now the nodes are ordered using their key values
- General Tree: each parent can have as many children as (s)he likes
Binary Trees (BT)

- A binary tree is either empty, $T = []$, or
- the tree is represented by $T = [\text{lhs} : \text{root} : \text{rhs}]$

Here, the $\text{root}$ (node) denotes the very first node of the tree, and $\text{lhs}$ and $\text{rhs}$ refer to the subtrees on the left-hand and right-hand sides, respectively.

This definition clearly leads to recursive implementations, e.g.:

```
size_of(tree t) =
    is_empty(t) => 0
    otherwise => 1 + size_of(lhs) + size_of(rhs)
```
BT axioms

Assume

\[
\text{create()} = []
\]

\[
\text{construct}(L,E,R) = [L:E:R]
\]

then

\[
\text{is\_empty}(\text{create}()) = \text{true}
\]

\[
\text{is\_empty}(\text{construct}(L, E, R)) = \text{false}
\]

\[
\text{lhs\_child}(\text{create}()) = \text{error: empty tree}
\]

\[
\text{root}(\text{create}()) = \text{error: empty tree}
\]

\[
\text{rhs\_child}(\text{create}()) = \text{error: empty tree}
\]

\[
\text{lhs\_child}(\text{construct}(L, E, R)) = L
\]

\[
\text{root}(\text{construct}(L, E, R)) = E
\]

\[
\text{rhs\_child}(\text{construct}(L, E, R)) = R
\]

\[
\text{remove\_lhs}(\text{create}()) = \text{error: empty tree}
\]

\[
\text{remove\_rhs}(\text{create}()) = \text{error: empty tree}
\]

\[
\text{remove\_lhs}(\text{construct}(L, E, R)) = [L][]:E:R]
\]

\[
\text{remove\_rhs}(\text{construct}(L, E, R)) = [L:E:[]]
\]
A possible BT declaration

This declaration defines a BT with a header node that gives access to the root of the tree, and pointers like the leftmost and rightmost nodes of the tree.

typedef struct tree_header {
  tree_node_ptr root, cur;
  tree_node_ptr lmost, rmost;
  int size;
} tree_header;
typedef tree_header* tree

tree my_tree, your_tree, our_tree, T;
Binary Search Trees (BST)

- Elements are characterized by key values
- Here, we assume that keys are unique
- BST nodes, say \( T = [L:E:R] \), are ordered such that
  - all nodes in \( L \) have keys smaller than the key of root element \( E \)
  - all nodes in \( R \) have keys larger than the key of root element \( E \)

An operation like \( \text{get\_data}(k, T) \) then satisfies

\[
\text{get\_data}(k, []) = \text{error: empty tree} \\
\text{get\_data}(k, [L:E:R]) = \\
  \begin{align*}
    & \text{if } k < \text{key}(E) \text{ then } \text{get\_data}(k, L) \\
    & \text{if } k = \text{key}(E) \text{ return } (\text{data}(E)) \\
    & \text{if } k > \text{key}(E) \text{ then } \text{get\_data}(k, R)
  \end{align*}
\]
Insert element $F$ with given key $K$ at the right position in tree $T$. Pseudo code:

\[
\begin{align*}
\text{insert}(F, []) &= [[]:F:[]] \\
\text{insert}(F, [L:E:R]) &= \\
&\quad \text{if } \text{key}(F) < \text{key}(E) \text{ then } [\text{insert}(F,L):E:R] \\
&\quad \text{if } \text{key}(F) = \text{key}(E) \text{ then } \text{no action or increase freq} \\
&\quad \text{if } \text{key}(F) > \text{key}(E) \text{ then } [L:E:\text{insert}(F,R)]
\end{align*}
\]
tree insert(elt_ptr pe, tree t) {
    tree_node_ptr c = t;
    if (c == NULL)
        return new_node(pe);
    while (1) {
        if (pe->key < c->elt->key) {
            if (c->lhs) c = c->lhs;
            else {
                c->lhs = new_node(pe); break;
            }
        } else if (pe->key > c->elt->key) {
            if (c->rhs) c = c->rhs;
            else {
                c->rhs = new_node(pe); break;
            }
        } else {
            c->freq++; break;
        }
    }
    return t;
}
BST insert (cont’d)

```
tree insert(elt_ptr pe, tree t) {
    tree_node_ptr *c = &t;
    while ((*c) && (pe->key != (*c)->elt->key))
        c = ((pe->key < (*c)->elt->key) ? &(*c)->lhs : &(*c)->rhs);
    if (*c)
        (*c)->freq++; // multiple insert
    else
        *c = new_node(pe);
    return t;
}
```
Delete node with a given key $K$ from tree $T$, while preserving BST properties.

\[
delete(K, []) = \text{no action (element is not present)}
\]
\[
delete(K, [L:E:R]) =
\]
\[
\quad \text{if } K < \text{key}(E) \ [\text{delete}(K, L):E:R]
\]
\[
\quad \text{if } K = \text{key}(E) \ \text{element found: actual delete follows}
\]
\[
\quad \text{if } K > \text{key}(E) \ [L:E:\text{delete}(K, R)]
\]

The actual deletion of the node is somewhat complicated. We have 4 cases:

\[
delete([[][[]]) = []
\]
\[
delete([L:[]]) = L
\]
\[
delete([[][R]) = R
\]
\[
delete([L:R]) = ?
\]
The 4th case needs some thinking: to preserve the order of the key values, we can replace the element that must be deleted by its successor or its predecessor:

- Its successor is the leftmost node in its right subtree!
- Its predecessor is the rightmost node in its left subtree!

So, we have two equivalent strategies. Now, one of the following transformations needs to be performed (E is element to be deleted):

\[
[L:E:R] \Rightarrow [L:\text{min}(R):\text{delete}(\text{min}(R),R)]
\]
\[
[L:E:R] \Rightarrow [\text{delete}(\text{max}(L),L):\text{max}(L):R]
\]
Some help functions

```c
int number_of_children (tree_node_ptr t) {
    return(((t->lhs) ? 1 : 0) + ((t->rhs) ? 1 : 0));
}

tree_node_ptr* lmn(tree_node_ptr* t) {
    tree_node_ptr *c = t;
    while (*((c)->lhs) c = &((c)->lhs;
    return(c);
}

tree_node_ptr* rmn(tree_node_ptr* t) {
    tree_node_ptr *c = t;
    while (*((c)->rhs) c = &((c)->rhs;
    return(c);
}
```

BST delete (cont’d)
BST delete (cont’d)

tree delete(key_type k, tree t) {
    tree_node_ptr *c = &t;
    while ((*c) and (k != (*c)->elt->key))
        c = ((k < (*c)->elt->key) ? &(*c)->lhs : &(*c)->rhs); // 1
    if (*c) { // 2
        if (has_two_children(*c)) // 3
            c = transform(c);
        *c = del(*c); // 4
    }
    return t;
}

//1: now c is the address of the pointer to the node to be deleted if any
//2: yes! a node that contains an element with key k exists
//3: transform from two to less than two children
//4: now the node has less than two children and is deleted
tree_node_ptr *transform(tree_node_ptr *t) {
    int sL = size_of((*t)->lhs),
    sR = size_of((*t)->rhs);
    elt_ptr pe;
    tree_node_ptr *p = (sL < sR)? lmn(&(*t)->rhs) : rmn(&(*t)->lhs);
    pe = (*p)->elt; (*p)->elt = (*t)->elt; (*t)->elt = pe; // swap
    return p;
}

tree_node_ptr del(tree_node_ptr t) {
    tree_node_ptr p = t;
    t = (p->lhs) ? p->lhs : p->rhs; // link t to child (if any)
    free(p->elt); // release the element
    free(p); // release the node
    return t;
}