Object Oriented Programming Training

Lab Book

Bootlin
http://bootlin.com
PAES-Institute Leipzig
http://www.paes.eit.htwk-leipzig.de

January 28, 2025

About this document

This document can be found on `http://www.paes.eit.htwk-leipzig.de/paes/oop.html`.

It was generated from LaTeX sources found on `http://git.bootlin.com/training-materials`.

More details about our training sessions can be found on `http://bootlin.com/training`.

Additional course material can be found on repository .

Copying this document

Training setup

Download files and directories used in practical labs

Install Workbench

Actual **Eclipse** (Version September 2020) is the last stable Eclipse Version. We are using it in our laboratory too. 5 Steps to Install Eclipse

1. Download the Eclipse Installer. Download Eclipse Installer from https://projects.eclipse.org/projects/tools.oomph.
2. Start the Eclipse Installer executable.
3. Select the package to install.
4. Select your installation folder.
5. Launch Eclipse

The download addresses can be different regarding to your developer host (32GB or 64GB). Please choose the appropriate version!

Now go into the new folder and execute the Eclipse binary:

```
./eclipse
```

If needed you are able to configure the system in `eclipse.ini`

```
-vm
/usr/lib64/jvm/java-11-openjdk-11/bin
-startup
plugins/org.eclipse.equinox.launcher_1.5.800.v20200727-1323.jar
--launcher.library
/home/admin/.p2/pool/plugins/org.eclipse.equinox.launcher.gtk.linux.x86_64_1.1.1300.v20200819-0
-product
org.eclipse.epp.package.cpp.product
-showsplash
/home/admin/.p2/pool/plugins/org.eclipse.epp.package.common_4.17.0.20200910-1200
--launcher.defaultAction
openFile
--launcher.appendVmargs
-vmargs
-Declipse.p2.max.threads=10
-Doomph.update.url=http://download.eclipse.org/oomph/updates/milestone/latest
-Doomph.redirection.index.redirection=index:/->http://git.eclipse.org/c/oomph/org.eclipse.oomph
-Dosgi.requiredJavaVersion=11
-Dosgi.instance.area.default=@user.home/eclipse-workspace
-XX:+UseG1GC
-XX:+UseStringDeduplication
--add-modules=ALL-SYSTEM
-Dosgi.requiredJavaVersion=11
```

```
-Dosgi.dataAreaRequiresExplicitInit=true
-Xms256m
-Xmx2048m
--add-modules=ALL-SYSTEM
```

When the Workbench is launched the first thing you see is a dialog that allows you to select where the workspace should be located. The workspace is the directory where your work will be stored. For now, just chose `\$HOME/workspace.cdt` to pick the default location. (You can also check the checkbox to prevent this question from being asked again.)

After the workspace location is chosen, a single Workbench window is displayed. A Workbench window offers one or more perspectives. A perspective contains editors and views, such as the Navigator. Multiple Workbench windows can be opened simultaneously. Initially, in the first Workbench window that is opened, the Resource perspective is displayed, with only the Welcome view visible. Click the arrow labeled **Workbench** in the Welcome view to cause the other views in the perspective to become visible.

(You can get the Welcome view back at any time by selecting **Help** → **Welcome**.)

A shortcut bar appears in the top right corner of the window. This allows the user to open new perspectives and switch between ones already open. The name of the active perspective is shown in the title of the window and its item in the shortcut bar is highlighted. The title bar of the Workbench window indicates which perspective is active. In this example, the Resource perspective is in use. The Navigator, Tasks, and Outline views are open along with an editor.

Lab data are now available in your `~/workspace.cdt` directory in your home directory. For each lab there is a directory containing various data. This directory will be used as starting point for your own working process.

You are now ready to start the real practical labs!

Install extra packages

Ubuntu comes with a very limited version of the `vi` editor. Install `vim`, a improved version of this editor.

```
sudo apt-get install vim
```

For openSUSE no action is required. IN the case no `gcc` Compiler is installed, install it with all necessary dependencies. This includes GNU-`make` too.

More guidelines

Can be useful throughout any of the labs

- Read instructions and tips carefully. Lots of people make mistakes or waste time because they missed an explanation or a guideline.
- Always read error messages carefully, in particular the first one which is issued. Some people stumble on very simple errors just because they specified a wrong file path and didn't pay enough attention to the corresponding error message.
- Never stay stuck with a strange problem more than 5 minutes. Show your problem to your colleagues or to the instructor.
- You should only use the root user for operations that require super-user privileges, such as: mounting a file system, loading a kernel module, changing file ownership, configuring the network. Most regular tasks (such as downloading, extracting sources, compiling...) can be done as a regular user.

- If you ran commands from a root shell by mistake, your regular user may no longer be able to handle the corresponding generated files. In this case, use the `chown -R` command to give the new files back to your regular user.
  Example: `chown -R myuser.myuser linux-3.4`

Working Environment

Here are coming soon some more information about the used virtualized (Virtualbox) OpenSuse 15.2 machine. This virtual machine is free available, please contact your instructor how to get access.

Plain C - Laboratory Training Files

Lab 1.1: Introduction into c

The following exercises and samples are taken Literature:
K&R: Brian W. Kernighan, Dennis M. Ritchie, The C Programming Language, based on Draft-Proposed ANSI C (2nd Edition, Prentice Hall, ISBN 0-13-110362-8).

These exercises include sample programs and easy exercises to learn the language C. They are not a complete script for the course. Detailed explanations are given in the lecture.

Source Code 1.1: Plain c - Hello World (Exercise: 1.1)

```
/*
 ============================================================================
 Name        : PlainCHelloWorld.c
 Author      : A. Pretschner
 Version     :
 Copyright   : Your copyright notice
 Description : Hello World in C, Ansi-style
 ============================================================================
 */

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

int main(void) {
	puts("!!!Hello World!!!"); /* prints !!!Hello World!!! */
	return EXIT_SUCCESS;
}
```

This program is compiled with the command `gcc PlainCHelloWorld.c`. The compiler then creates an executable file named `a.out`. This program can be executed by typing `./a.out`. The program causes the screen output of the text **Hello, World!**. Compile the program within the Eclipse-Environment! Watch out the Terminal in Eclipse!
Task 1.1: *Change the program so that the words "Hello", and "World!" are output in two consecutive lines. Experiment with other output forms.* Here comes another program in multiple versions:

Source Code 1.2: Fahrenheit-Celsius (Exercise: 1.2)

```
/*
============================================================================
 Name        : fci.c
 Author      : A. Pretschner
 Version     :
 Copyright   : Kernighan & Ritchie 2. Edition
 Description : print of an Fahrenheit-Celsius-Table
                             for fahr = 0, 20, ..., 300, int-Version
 ============================================================================
 */

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

```c
int main() {
        int fahr, celsius; /* Variable Declaration */
        int lower, upper, step; /* from int-Typ                         */

        lower = 0; /* lower limit...                 */
        upper = 300; /* upper table limit         */
        step = 20; /* step size                        */

        fahr = lower;
        while (fahr <= upper) { /* while-construct                */
                celsius = 5 * (fahr - 32) / 9;
                printf("%d\t %d\n", fahr, celsius);
                fahr = fahr + step;
        }
}
```

The program compiled and executed brings up:

```
  0 | -17
 20 |  -6
 40 |   4
 60 |  15
... | ...
300 | 148
```

Task 1.2: *What output do you get when replacing the **printf**-line with one of the following Orders:*

```c
printf("%5d %20dnn", fahr, celsius);
printf("%-5d %20dnn", fahr, celsius);
printf("%3d %6dn n", fahr, celsius);
```

Explain the results!

Source Code 1.3: continued (Exercise: 1.3)

```c
/*
 ============================================================================
 Name        : fcf.c
 Author      : A. Pretschner
 Version     : Kernighan & Ritchie 2. Edition
 Copyright   : Your copyright notice
 Description : outputs Fahrenheit-Celsius-Table for
 fahr = 0, 20, ..., 300; float-Version
 ============================================================================
 */

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

int main() {
        float fahr, celsius; /*  float-Variable                 */
        int lower, upper, step; /*     from int-Typ                 */

        lower = 0; /* lower limit...                 */
        upper = 300; /* upper limit of the table          */
        step = 20; /* step size                        */

        fahr = lower;
        while (fahr <= upper) {
                celsius = 5.0 / 9.0 * (fahr - 32.0);
                printf("%3.0f\t %6.1f\n", fahr, celsius);
                fahr = fahr + step;
        }
}
```

The output values are now given as ”-17.8”.
Another two versions below:

Source Code 1.4: continued (Exercise: 1.4)

```
/*
 ============================================================================
 Name        : fc2.c
 Author      : A. Pretschner
 Version     : Kernighan & Ritchie 2. Edition
 Copyright   : Your copyright notice
 Description : outputs Fahrenheit-Celsius-Table for fahr = 0, 20, ..., 300;
Version with for-construct
 ============================================================================
 */

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


int main() {
        int fahr;
        for ( fahr = 0; fahr <= 300; fahr = fahr + 20)
                printf("%3d\t %6.1f\n", fahr, (5.0/9.0)*(fahr-32));
}
```

and

Source Code 1.5: continued (Exercise: 1.5)

```
/*
 ============================================================================
 Name        : fc3.c
 Author      : A. Pretschner
 Version     : Kernighan & Ritchie 2. Edition
 Copyright   : Your copyright notice
 Description : outputs Fahrenheit-Celsius-Table for fahr = 0, 20, ..., 300;
Version with for-construct and symbolic constants
 ============================================================================
 */

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

#define LOWER 0
#define UPPER 300
#define STEP 20
int main() {
int fahr;
for (fahr = LOWER; fahr <= UPPER; fahr = fahr + STEP)
printf("%3d\t %6.1f\n", fahr, (5.0/9.0)*(fahr-32));
}
```

Task 1.3: *Create a program that converts Celsius degrees to Fahrenheit degrees. Write one version with the* ***while*** *construct, another with a* ***for*** *loop.*

Source Code 1.6: Standard Input/Output (Exercise: 1.6)

```
/*
 ============================================================================
 Name        : copy1.c
 Author      : A. Pretschner
 Version     : Kernighan & Ritchie 2. Edition
 Copyright   : Your copyright notice
 Description : copy  Standard-Input to Standard-Output
 ============================================================================
 */

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

int main() {
	int c;
	c = getchar();
	while ( c != EOF ) {
		putchar(c);
		c = getchar();
	}
}
```

Translate this program both with the command `gcc -o copy copy.c` and the Eclipse-Environment. The executable file is then called "copy". (This is done by the -o-option of `gcc`. To learn more about other compiler options, you can call `gcc`.) The copy command then causes the standard input to be copied to the standard output. For example, you can use `copy < datei1 > datei2` to copy the file.
A shorter version of the program is the following. It does exactly the same as version 1.

Source Code 1.7: continued (Exercise: 1.7)

```
/*
 ============================================================================
 Name        : copy1.c
 Author      : A. Pretschner
 Version     : Kernighan & Ritchie 2. Edition
 Copyright   : Your copyright notice
 Description : copy  Standard-Input to Standard-Output - Version 2
 ============================================================================
 */

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

int main() {
	int c;
	c = getchar();
	while ( c != EOF ) {
		putchar(c);
		c = getchar();
	}
}
```

Task 1.4: *What is the value of the expression `getchar() != EOF`? Write a program that returns this value. What is the value of EOF?* The following program counts the read characters:

Source Code 1.8: continued (Exercise: 1.8)

```
/*
 ============================================================================
 Name        : count1.c
 Author      : A. Pretschner
 Version     : Kernighan & Ritchie 2. Edition
 Copyright   : Your copyright notice
 Description : counts the characters read in the standard input
                           uses data type long - Version 1
 ============================================================================
*/

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

int main() {
        long nc;

        nc = 0;
        while ( getchar() != EOF ){
                printf("%ld\n",nc);
                ++nc;
        }
}
/*
 ============================================================================
 Name        : count2.c
 Author      : A. Pretschner
 Version     : Kernighan & Ritchie 2. Edition
 Copyright   : Your copyright notice
 Description : counts the characters read in the standard input
                           uses data type double - Version 1
 ============================================================================
*/

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

int main() {
        double nc;
        for ( nc = 0; getchar() != EOF; ++nc ){
                printf("%.0f\n",nc);
        }
}
```

In the second version the ”empty” for-loop has to be considered! All the work is already done in the test part and by increment the counter variable. The following program counts the lines of a file.

Source Code 1.9: Read lines (Exercise: 1.9)

```
/*
 ============================================================================
 Name        : count3.c
 Author      : A. Pretschner
 Version     : Kernighan & Ritchie 2. Edition
 Copyright   : Your copyright notice
 Description : counts the read lines in the standard input
 ============================================================================
*/
```

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


int main() {
        int c, nl;
        nl = 0;
        while ( (c = getchar()) != EOF ){
                if ( c == '\n')      /* check logical equality */
                        ++nl;
        printf("%d\n", nl);
        }
}
```

A character is marked in C with single quotation marks. So the character G is called 'G', the "Newline" character is called 'nn'.

Task 1.5: *Write a program that copies the standard input to the standard output, but replaces all spaces with newlines. (Note: A space is indicated by ' '.)*

$\boxed{\text{Submit your answers from Lab 1.1 to Moodle one week after the assignment}}$

Lab 1.2: Structure of simple c programs

A C program consists of

- preprocessor statements such as
```
    #include <stdio.h>
    #define UPPER 300
```
- (global) declarations and definitions of variables such as `int number; int digit = 42;`
- Function definitions like
```
     main() {
        ...
    }
```

A **function** or function definition consists of a *name* (e.g.`main, getchar`), followed by a pair of parentheses () within which "formal variables" may appear, followed by a compound statement (compound statement, a sequence of statements or statements bundled by a pair of curly parentheses ...).

A **statement** (instruction, command) can be of different structure. We will explore the possibilities for this purpose in this course. Examples are the above mentioned declarations and definitions. Assignments such as `a = 5; digit = 42;` are also statements, more generally each expression followed by a semicolon is a statement, a sequence of statements can be inserted in a pair of curly braces {...} and thus bundled into a compound statement.

An **expression** is "everything that has a (number) value", such as constants like numbers (7, 3.14159), characters ('a', 'Z', '7'), arithmetic combinations of such (3+4, 5*6, 7*(8-9)), but also function calls like

`getchar(), putchar(c), printf("% d", a);`

the latter consist of the Name of the function, followed by (possibly followed by a list of comma-separated values (arguments) for each formal variable, followed by **)**. The value of such a function call is the value that the associated function takes for the arguments.

Likewise, assignment expressions such as `digit = 42, c = getchar()` are expressions, the value is the value of the assigned expression. Therefore, for example, chains of assignment expressions such as `a = b = 17+4` make sense; they are to be evaluated "from right to left"; the value of the expression 17+4 is 21, this value is assigned to the variable b, the value of this assignment is of course 21 again, which is then assigned to the variable a is assigned.

You can easily output the value of an expression with a program of the following type.

Source Code 1.10: printf() (Exercise: 2.1)

```
#include <stdio.h>
main() {}
  int a, b;
  printf("Statement: %d\n", a = b = 17+4 );
}
```

More generally, in the `printf` function above, there could be any meaningful expression as a second argument. (”Sensible” here means that the program ”understands” this expression at this point; for example, the must be defined or declared beforehand for any variables that occur. An expression, for example, is also a ”String”, a variable specified in sequence of characters like ”abcxyz”, ”% d\n” enclosed by quotation marks, as well as a variable name like `fahr` is an expression. The value of a variable is always the number assigned to it, this value is altered in the program flow. The value of a string is explained later.

C - Operators:

1. unary operators:
   +, - (sign), ! (negation), (bit complement), ++ (increment), – (decrement, both as pre- and post-operators), * (redirection, indirection), & (address operator), `sizeof` (returns size of a data type in bytes). Operators applied to expressions produce expressions that have values. The in-/decrement operators differ in their prefix or suffix notation:
   ```
   int a, b;
   a = 5;
   b = a++; /* Value of a is copied to b, then(!) is incremented */
   ```
   returns as result: b has the value 5, a has the value 6. Opposite:
   ```
   int a, b;
   a = 5;
   b = ++a; /* a is incremented first(!), the new value of a is copied to b. */
   ```
   Result: b has the value 6 a has the value 6.
2. binary operators
   (a) Arithmetic operators:
   +, -, *, /, % (remainder for integer division)
   (b) Relational (logical) operators:
   < (smaller), <= (smaller-equal), > (larger), >= (larger-equal), == (equal), ! = (unequal), && (logical) ’AND’), || (logical ’OR’),
   (c) Bit arithmetic operators:<<, >> (bitshift operators)

   - $a << 3$ shifts the bit sequence from a by 3 positions to the left,
   - $a >> 5$ shifts the bit sequence from a by 5 positions to the right,
   - & (bitwise ’AND’), || (bitwise ’OR’), ^ (bitwise exclusive ’OR’).
3. conditional operator (ternary):
   Syntax: expression1 ? expression2 : expression3
   Returns as value codeexpression2, if `expression1` not equal to 0, and `expression3` else (compare if - else). Example:
   ```
   int a, b, c;
   a = 3; b = 5;
   c = ( a > b ) ? a : b;
   returns the value 5 for c.
   ```
   returns value 5 for c.
4. assignment operators:
   = , Example: a = 17;
   An assignment returns a value, namely that of the assigned size, so multiple assignments such as a = b = u = 17 are possible. Example: a += 2 is logically equivalent with a = a + 2, analogously for each (bit-)arithmetic binary Operator: a /= b is equivalent to a = a/b and so on.
5. comma operator: (groups expressions as separator).

Task 2.1: *Use a small program to test all of the above operators for their effectiveness by outputting values of expressions in which the operators appear.*

C - Keywords (forbidden as names for variables, functions, etc.):

(a) for the designation or definition of data types:
`char, int, unsigned, signed, short, enum, long, float, double, void, const, struct, union, typedef, sizeof;`

(b) for the qualification of variables:
`auto, static, external, register, volatile;`

(c) for program control:
`while, for, if, else, do, switch, case, default, break, continue,return, goto.`

(d) reserved by some implementations:
`fortran, asm`

C - data types: `char` - 1 byte, can contain 'one character' (e.g. 'a', '3', '&', ...).
`int` - integer, size depends on implementation.
`float` - Single precision floating-point number.
`double` - Double precision floating-point number.

`int` can be qualified by `unsigned, short, long` to `unsigned int, short int, long int`, or short to `unsigned, short, long`.
`char` can be qualified to `unsigned char`

Source Code 1.11: Precision format ranges.

	IBM-PC				**32-Bit-Workstations**			
char	-128	–	+127	(1 Byte)	-128	–	+127	(1 Byte)
unsigned char	0	–	255	(1 Byte)	0	–	255	(1 Byte)
int	-32768	–	+32767	(2 Bytes)	(wie long)			
unsigned	0	–	65535	(2 Bytes)	0	–	4294967295	(4 Byte
short	-32768	–	+32767	(2 Bytes)	-32768	–	+32767	(2 Byte
long	-2147483648	–	+2147483647	(4 Byte)				
unsigned long	0	–	4294967295	(4 Byte)				
float	+/- 1.17 E-38	–	3.4 E+38	(4 Byte)	(IEEE-Format)			
double	+/- 2.22 E-308	–	1.7 E+308	(8 Byte)	(IEEE-Format)			

Here is an example program with which you can output the calculation ranges of the standard data types for your computer:

Source Code 1.12: Ranges (Exercise: 2.2)

```
/*
 ============================================================================
 Name        : ranges.c
 Author      : A. Pretschner
 Version     : Kernighan & Ritchie 2. Edition
 Copyright   : Your copyright notice
 Description : shows the number ranges
 ============================================================================
*/

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


#include <limits.h>
```

```c
#include <float.h>
#include <stdio.h>
#include <stdbool.h> // For defintion of bool

int main() {
  printf("The sizes of the default data types for this machine are:\n");
  printf("char: \t%d Byte\n", sizeof(char));
  printf("\t Bits per char = %2d\n", CHAR_BIT);
  printf("\t min char = %4d\n", CHAR_MIN);
  printf("\t max char = %4d\n", CHAR_MAX);
  printf("\t max unsigned char = %4d\n", UCHAR_MAX);
  printf("short: \t%d Byte\n", sizeof(short) );
  printf("\t min short = %6d\n", SHRT_MIN);
  printf("\t max short = %6d\n", SHRT_MAX);
  printf("\t max unsigned short = %6d\n", USHRT_MAX);
  printf("int: \t%d Byte\n", sizeof(int) );
  printf("\t min int = %11d\n", INT_MIN);
  printf("\t max int = %11d\n", INT_MAX);
  printf("\t max unsigned int = %11u\n",UINT_MAX);
  printf("long: \t%d Byte\n", sizeof(long) );
  printf("\t min int = %21ld\n" ,LONG_MIN);
  printf("\t max int = %21ld\n", LONG_MAX);
  printf("\t max unsigned long = %21lu\n",ULONG_MAX);
  printf("float: \t%d Byte\n", sizeof(float) );
  printf("\t Praezision: = %2d Dezimalstellen\n", FLT_DIG);
  printf("\t min float: = %e\n", FLT_MIN);
  printf("\t max float: = %e\n", FLT_MAX);
  printf("double:\t%d Byte\n", sizeof(double) );
  printf("\t Precision: = %2d Dezimalstellen\n", DBL_DIG);
  printf("\t min double: = %e\n", DBL_MIN);
  printf("\t max double: = %e\n", DBL_MAX);
}
```

Output of the program for a 64-bit workstation:
The sizes of the default data types for this machine are:

```
char:    1 Byte
         Bits per char        =                        8
         min char             =                     -128
         max char             =                      127
         max unsigned char    =                      255
short:   2 Byte
         min short            =                   -32768
         max short            =                    32767
         max unsigned short   =                    65535
int:     4 Byte
         min int              =              -2147483648
         max int              =               2147483647
         max unsigned int     =               4294967295
long:    8 Byte
         min int              =     -9223372036854775808
         max int              =      9223372036854775807
         max unsigned long    =     18446744073709551615
```

(float and double like for 32 Bit)

Task 2.2: *Write a program to test* ***all*** *of your machine's default data types. Perhaps have a* *glance into `limits.h` ans codefloat.h!*

HINT: Pressing `<F3>` when the cursor is located above the include files, *Eclipse* try to open the referenced file if available.

Submit your answers from Lab 1.2 to Moodle one week after the assignment

Lab 1.3: Syntax of printf and scanf, while and for

```
printf("What is this?"); /*prints: What is this?*/
printf("What is this?\n"); /*prints: What is this? and adds a newline character*/
printf("% d", a); /*prints the contents of the (int-) variable a decimal,*/
printf("% o", b); /*prints the contents of the (int-) variable b octal,*/
printf("%x", c); /*does what?*/
printf("%4.6f", d); /*thinks d is a float variable and prints??*/
/*In the last example, replace f with e, g. What happens?*/
printf("%s", string); /*considers string a string,*/
printf("%c", charact); /*interprets charact as character.*/
/*What do the control variables "\%10s", "-20s", "-20.4s" etc. means?*/
```

Similar rules apply to the arguments of the `scanf` input function.
Example:

```
scanf("\%d", &a)
```

expects (e.g. from the keyboard) a decimal int-entry and assigns this as the value of the (int-) variables a to. (Not a itself appears as an argument, but &a, the ”address” or a pointer pointing to a).
Examples for the syntax of while and for:
The following examples 3.x are not all exemplary from a programming point of view. They should be only by means of a simple example the formal equivalence of the for-construct with the one given in the lecture in the extended while construct. Judge for yourself about the usefulness and Readability of the different variants.

Source Code 1.13: scanf() (Exercise: 3.1)

```
/*
 ============================================================================
 Name        : scan.c
 Author      : A. Pretschner
 Version     : Kernighan & Ritchie 2. Edition
 Copyright   : Your copyright notice
 Description : scanf, printf, while syntax
 ============================================================================
*/

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

int main () {
          int a, b, c;
          printf("Input a, b : ");
          scanf("%d %d", &a, &b);
          while (b != 0) {
                c = a % b; a = b; b = c;
          }
          printf("%s %d\n","ggt =", a);
}
```

Remark: The `while` construct in this program can also be used with the help of the comma operator in this way:

```
  while (b != 0)
      c = a % b, a = b, b = c;
```

Task 3.1: *What happens if the `scanf` line is replaced by the following? scanf ("%d %d ", a, b); Try to give an explanation based on the `scanf` manual entry.*

Source Code 1.14: continued (Exercise: 3.2)

```
/*
 ============================================================================
 Name        : scan2.c
 Author      : A. Pretschner
 Version     : Kernighan & Ritchie 2. Edition
 Copyright   : Your copyright notice
 Description : scanf, printf, while syntax
 ============================================================================
*/

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

int  main() {
  int a, b, c;
  printf("Input a, b : ");
  for( scanf ("%d %d", &a, &b); b != 0; ) {
    c = a % b; a = b; b = c;
  }
  printf("%s %d\n","ggt =", a);
}
```

Source Code 1.15: continued (Exercise: 3.3)

```
  #include <stdio.h>
  main() {
    int a, b, c;
    printf("Input a, b : ");
    for(scanf ("%d %d", &a, &b); b!=0; c = a % b, a = b, b = c)
      ; /* empty for-Statement */
      printf("%s %d\n","ggt =", a);
    }
```

Source Code 1.16: continued (Exercise: 3.4)

```
  #include <stdio.h>
  main() {
    int a, b, c;
    for( printf("Input a, b : "), scanf ("%d %d", &a, &b);\
      b!=0; c = a % b, a = b, b = c)
      ;
    printf("%s %d\n","ggt =", a);
  }
```

Source Code 1.17: continued (Exercise: 3.5)

```
  #include <stdio.h>
  main() {
    int a, b, c;
    printf("Input a, b : ");
    scanf ("%d %d", &a, &b);
    for( ;b != 0; ) {
      c = a % b; a = b; b = c;
    }
  printf("ggt = %d\n", a);
  }
```

Task 3.2: *Change the program using a* `while(1)` *loop so that it repeatedly interactively queries a pair of numbers and returns the* `ggt` *of the pair. How could you create an abort option for program such an "endless" loop?*

Source Code 1.18: continued (Exercise: 3.6)

```
/*
 ============================================================================
 Name        : scan3.c
 Author      : A. Pretschner
 Version     : Kernighan & Ritchie 2. Edition
 Copyright   : Your copyright notice
 Description : scanf, printf, while syntax
 ============================================================================
 */

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

main() { /* main program */
        int a, b;
        printf("Input a, b");
        scanf("%d %d", &a, &b);
        printf("GGT = %d", ggt(a, b));/* calls subroutine */
}

int ggt(int a, int b) { /* subroutine ggt with int-arguments a,b*/
        int c; /* declaration  of the "locale variables" */
        if (b != 0) /* subroutine statements ... */
                c = ggt(b, a % b); /* the  program calls it self,
                 means "recursive" */
        else
                c = a;
        return c; /* value of c returns to ... */
} /* ... the caller program */
```

Source Code 1.19: continued (Exercise: 3.7)

```
  #include <stdio.h>
  main() {
    int a,b;
    printf("Input a, b : ");
    scanf("%d %d", &a, &b);
    printf("GGT = %d", ggt( a, b));
  }

  int ggt(int a, int b) {} /* Similar like above, but with one exception ... */
    return ( b != 0 ) ? ggt( b, a % b) : a; /* ...C-typical short cuts */
  }
```

Source Code 1.20: continued (Exercise: 3.8)

```
/*
 ============================================================================
 Name        : scan4.c
 Author      : A. Pretschner
 Version     : Kernighan & Ritchie 2. Edition
 Copyright   : Your copyright notice
 Description : The second oldest computer program in the world-: from anno -250.
               Copyright: ERATOSTHENES, philologist, computer scientist and geographer
 ============================================================================
 */

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

#define TRUE  1                /* global constants deklaration */
#define FALSE 0
#define MAX  10000

int main() {
      int i,prime,k,count,size;
      char flags[MAX];        /*  array declaration of .. */
             /* MAX variable flags[0], ...flags[MAX - 1] */
             /* of type "character"; 1 byte */

      printf("Determination of all prime numbers up to "); scanf("%d", &size);

      size = (size+1)/2-2; count = 1; /* only odd numbers are filtered */

      for (i = 0; i <= size; i++)     /* initialize the  ... */
          flags[i]=TRUE;              /* ... whole array */

      for (i = 0; i <= size; i++) {
          if (flags[i]) {            /* if flag[i]=TRUE, then i+i+3 prim */
                prime = i+i+3; k=i+prime;
                while (k<=size) {               /* allthow scretch all  ... */
                    flags[k] = FALSE; k += prime;  /* nontrivial */
                }                              /* multiples */
                count++;                       /* and count. */
           }
       }
       printf("\n%d primes\n",count);
}
```

Task 3.3: *Change the program so that the individual prime numbers are output.*

Submit your answers from Lab 1.3 to Moodle one week after the assignment

Lab 1.4: Continuous Exercises

The examples in this section are intended to demonstrate the programming techniques discussed so far and provide an opportunity for repetition.

- 4.1, 4.2 are examples of the concept of function,
- 4.3 to 4.5 are Modifications of the idiom `while( (c=getchar()) != EOF) putchar(c)`,
- 4.6 is a combination of this all

Source Code 1.21: continued (Exercise: 4.1)

```
 /*
 ============================================================================
 Name        : scan5.c
 Author      : A. Pretschner
 Version     : Kernighan & Ritchie 2. Edition
 Copyright   : Your copyright notice
 Description : scanf, printf, while syntax
 ============================================================================
 */

#include <stdio.h>

 int main() {
    int i;
    for ( i = 0; i < 10; i++) {
            printf("%d %6d %6d\n", i, power(2,i), fact(i));
    }
 }

    int power(int base, int n) { /* calculate base power n */
      int i, p = 1;                /* during declaration initialization of p*/
      for (i = n; i > 0; i--)
      p = p * base;                /* shorter: p *= base; */
      return p;
    }

    int fact(int y){                 /* calculates 1*2* ... *(y-1)*y */
      int result = 1;
      while (y != 0)                 /* Attention! if y < 0 .... */
              result *= y--;                 /* shorter: result = result * y; y--; */
      return result;
    }
```

Task 4.1: *In both previous functions, no checks of the arguments are provided. Correct this.*

Source Code 1.22: Square Root (Exercise: 4.2)

```
 /*
 ============================================================================
 Name        : scquare1.c
 Author      : A. Pretschner
 Version     : Kernighan & Ritchie 2. Edition
 Copyright   : Your copyright notice
 Description : Square root without error output for negative radicands
 ============================================================================
 */

#define abs(A) ( (A) < 0 ? -(A) : (A) ) /* "function" per definition */
#define DELTA 1.0e-16

double squareroot(double n) {
        double x = n, alt_x = 0;
        if (n <= 0)
                return 0;
        while ( abs(x - alt_x) > DELTA ) {
                alt_x = x;
                x = (n/x + x)/2;
        }
        return x;
}

int main() {
        int i;
        double squareroot();
        for (i=0; i < 10; i++)
```

```
                printf( "%d %20.16f\n", i, squareroot( (double)i ) );
        }
```

Task 4.2: *Improve the `squareroot()` function so that in the case of a negative radicand a corresponding error message appears.*

Source Code 1.23: Crypto (Exercise: 4.3)

```
/*
 ============================================================================
 Name        : crypto.c
 Author      : A. Pretschner
 Version     : Kernighan & Ritchie 2. Edition
 Copyright   : Your copyright notice
 Description : scanf
 ============================================================================
*/

#include <stdio.h>
#include <ctype.h>

int main() { /* cryptograph */
        int c;
        while ( (c = getchar()) != EOF)
      if ((c > 63) & (c < 91))
             putchar(tolower(c));
}
```

Task 4.3: *Modify this program so that* ***any*** *cryptographic key that can be entered in response to a query can be used.*

Source Code 1.24: Line numbers (Exercise: 4.4)

```
/*
 ============================================================================
 Name        : linenumbers.c
 Author      : A. Pretschner
 Version     : Kernighan & Ritchie 2. Edition
 Copyright   : Your copyright notice
 Description : line numbering
 ============================================================================
*/

#include <stdio.h>
int main() { /* inserts line number*/
    int c, nl;
    nl = 1;
    printf("%4d %4c", nl, ' ');
    while ((c = getchar()) != EOF) {
      if (c == '\n') {
        ++nl; putchar(c);
        printf("%4d %4c", nl, ' ');
      }
      else putchar(c);
    }
}
```

Task 4.4: *Watch the output of the program and modify the program so that the line numbers appear as comments in a C-program, i.e. framed by /* ... */ .*

Source Code 1.25: Line words (Exercise: 4.5)

```
*
 ============================================================================
 Name        : linenwords.c
 Author      : A. Pretschner
 Version     : Kernighan & Ritchie 2. Edition
 Copyright   : Your copyright notice
 Description : word line numbering
 ============================================================================
*/

#include <stdio.h>
#define IN 1
#define OUT 0

int main() {         /* counts lines, words, char's in input */
        int c, nl, nw, nc, state;
        state = OUT;
    nl = nw = nc = 0;
    while ( (c = getchar()) != EOF) {
      ++nc;
      if (c == '\n')
              ++nl;
      if (c == ' ' || c == '\n' || c == '\t')
              state = OUT;
      else if (state == OUT) {
              state = IN;
              ++nw;
                    }
      printf("%d %d %d\n", nl, nw, nc);
    }
}
```

Task 4.5: *Write a program that outputs the words of its input line by line.*
Example 4.6 The following program reads a series of text lines and outputs the longest line at the end:

Source Code 1.26: Longest word (Exercise: 4.6)

```
/*
 ============================================================================
 Name        : longesword.c
 Author      : A. Pretschner
 Version     : Kernighan & Ritchie 2. Edition
 Copyright   : Your copyright notice
 Description : outputs longest line of text
 ============================================================================
*/
#include <stdio.h>
#define MAXLINE 4
int getLine (char line[], int maxline);
void copy(char to[], char from[]);


int  main() {
        int len;
        int max;
        char line[MAXLINE];     /* current input line */
        char longest[MAXLINE]; /* longest line read so far */
        max = 0;
        while ( (len = getLine(line, MAXLINE)) > 0)
                if (len > max) {
                        max = len;
                        copy(longest, line);
                }
        if (max > 0)
                printf("%s", longest);
        return 0;
}

/* getlines reads line by line to line[] and returns the lenght of it */
int getLine(char line[], int maxline) {
    int c, i;
    for ( i=0; i < maxline-1 && (c=getchar()) != EOF && c != '\n'; ++i)
      line[i] = c;
    if (c == '\n') {
      line[i] = c;
      ++i;
    }
    line[i] = '\0';
    return i;
}

/* copy copies "from" to "to" without checking the size of "to" */
void copy(char to[], char from[]) {
    int i = 0;
    while ( (to[i] = from[i]) != '\0')
      ++i;
}
```

Task 4.6: *Write a program that removes all blanks and tabs from a file in each line, as well as empty lines.*

Submit your answers from Lab 1.4 to Moodle one week after the assignment

Lab 1.5: Arrays and Pointer

Examples for initializing arrays:
`unsigned int prim[] = {- 2, 3, 5, 7, 11, 13, 17, 19 }`

defines an array `prim` of 8 variables `prim[0], ..., prim[7]` of the type `unsigned int`. The size operator `sizeof(prim)` returns the value `32 = 8 * sizeof(int)`.

The expression `char word[] = "Hallo\n";` is abbreviation for `char word[] = {'H', 'a', 'l', 'l', 'o', '\n', '\0'};` The notation ”...” always embeds a terminating `\0` character. Therefore `sizeof(word)` returns the value 7, while `strlen(word)` returns 6. For example, `word[1] == 'a'`.

Variables have addresses, which are integers that denote the location in (virtual) memory that is intended for the variable contents of the computer. The memory is divided into consecutive bytes (a byte usually consists of 8 bits), the bytes are numbered consecutively, the number assigned to a byte is the address of this byte and also any sequence of bytes beginning with this byte. The address of a variable is thus the number of the first byte of the byte sequence containing the contents of this variable. The **address** of the variable a is `&a`.

For each variable type there is a pointer or pointer type: With the declaration `double *pf` one declares a pointer variable `pf` of the type ”pointer to double”. Here you can assign addresses of double variables: If `double z` is declared, you get the following by `pf = &z` the address of `z` in pf. The dereferencing operator * allows you to access the value that a pointer variable shows: `x = *pf` assigns a double value to the double variable x, to which pf points, in this case, the value of z.

A short example of this:

Source Code 1.27: Pointer (Exercise: 5.1)

```
 /*
 ============================================================================
 Name        : pointer.c
 Author      : A. Pretschner
 Version     : Kernighan & Ritchie 2. Edition
 Copyright   : Your copyright notice
 Description : pointer and addresses
 ============================================================================
*/

 #include <stdio.h>
 int main() {
    double x, z;
    double *pf; /* pointer variable of type double */
    z = 3.14;
    x = 2.71;
    pf = &z; /* pf points to z */
    printf("x = %f, z = %f, pf = %d, *pf = %f\n", x, z, pf, *pf);
    z += 4.2;
    printf("x = %f, z = %f, pf = %d, *pf = %f\n", x, z, pf, *pf);
    x = *pf; /* Dereferencing of pf and assignment to x */
    printf("x = %f, z = %f, pf = %d, *pf = %f\n", x, z, pf, *pf);
    return 0;
  }
```

The program output is:

```
x = 2.710000, z = 3.140000, pf = 2147476076, *pf = 3.140000
x = 2.710000, z = 7.340000, pf = 2147476076, *pf = 7.340000
x = 7.340000, z = 7.340000, pf = 2147476076, *pf = 7.340000
```

In an array, for example `a[]` of a given `type` type, the array variables `a[i]` have consecutive addresses. The difference between two consecutive variables is equal to `sizeof(type)`, i.e. the number of bytes of this type used in memory.

Pointer variables follow an additive arithmetic: For the declaration `type *p` the `p = p + 3;` means the Increase of p by `3 * sizeof(type)`, so that within an array of type `type` 3 variables are incremented. The same applies to – ("minus"). In the above example, after `p = &a[1]; p = p + 3;` the pointer variable p points to `a[4]`, so it is then `p== &a[4]`.

A program example for this:

Source Code 1.28: continued (Exercise: 5.2)

```
 /*
 ============================================================================
 Name        : pointer1.c
 Author      : A. Pretschner
 Version     : Kernighan & Ritchie 2. Edition
 Copyright   : Your copyright notice
 Description : pointer and addresses
 ============================================================================
*/

 #include <stdio.h>
```

```
#define LEN 5
main() {
  double x[LEN];
  char s[LEN];
  int i;
  double *pd;
  char *pc;
  printf("sizeof(double) = %d, sizeof(char) = %d\n",sizeof(double), sizeof(char));
  for (i=0; i < LEN; i++)
    printf("&x[%d] = %d, &s[%d] = %d\n", i, &x[i], i, &s[i]);
  printf("x = %d, s = %d\n", x, s);
  printf("x+1 = %d, s+1 = %d\n", x+1, s+1);
  x[0] = 3.14; s[0] = 'H';
  x[1] = 2.14; s[1] = 'a';
  x[4] = 333.111, s[4] = 'o';
  printf("*x = %f, *s = %c\n", *x, *s );
  printf("*(x+1) = %f, *(s+1) = %c\n", *(x+1), *(s+1) );
  printf("x[1] = %f, s[1] = %c\n", x[1], s[1]);
  pd = &x[1]; pc = &s[1];
  printf("Zeigerarithmetik:\n");
  printf("pd = %d, pc = %d\n", pd, pc);
  pd += 3; pc += 3;
  printf("pd = %d, pc = %d\n", pd, pc);
  printf("*pd = %f, *pc = %c\n", *pd, *pc);
  return 0;
}
```

Task 5.1: *Interpret the output of the last program in all details in the sense of pointer arithmetic.*

Pointers *p, *q of the same type can be subtracted, p-q is the integer n, for which in the above pointer arithmetic `p = q + n` applies. The number n can be negative.

Task 5.1: *Study the pointer arithmetic, in particular the facts just described, on the basis of your own example programs; for example, output the difference for two pointers of the same type. What happens if you try to subtract two pointers of different types?*

Source Code 1.29: Switch (Exercise: 5.3)

```
/*
============================================================================
Name        : switch.c
Author      : A. Pretschner
Version     : Kernighan & Ritchie 2. Edition
Copyright   : Your copyright notice
Description : this program demonstrates the Switch statement and counts single digits,
              white space and other characters
============================================================================
*/

#include <stdio.h>
int main() {
  int c, i, nwhite, nother, ndigit[10];
  nwhite = nother = 0;
  for (i = 0; i < 10; i++ )
    ndigit[i] = 0;
  while ( ( c = getchar() ) != EOF ) {
    switch (c) {
      case '0': case '1': case '2': case '3': case '4':
      case '5': case '6': case '7': case '8': case '9':
          ndigit[c-'0']++;
          break;
      case ' ':
      case '\n':
      case '\t':
          nwhite++;
          break;
      default: nother++;
      break;
    }
  }
printf("Digits =");
for (i = 0; i < 10; i++)
  printf(" %d", ndigit[i]);
printf(", white space = %d, other = %d\n", nwhite, nother);
return 0;
}
```

Task 5.2: *Change this program so that single letters are also counted.*
A solution to Exercise 1.26 is the following program:

Source Code 1.30: Longest word (Exercise: 5.4)

```
  /*
 ============================================================================
 Name        : longestword3.c
 Author      : A. Pretschner
 Version     : Kernighan & Ritchie 2. Edition
 Copyright   : Your copyright notice
 Description : this program demonstrates the Switch statement and counts single digits,
               white space and other characters
 ============================================================================
 */

#include <stdio.h>
#include <string.h>

  #define MAXLINE 1000
  int getLine(char line[], int maxline);
  int trim( char s[]);

int main() {
    int len;
    int max;
    char line[MAXLINE];     /* current input line  */
    char longest[MAXLINE];  /* longest read line so far  */
    max = 0;
    while ( (len = getLine(line, MAXLINE)) > 0) {
      trim(line);
      if (strlen(line ) > 0)
        printf("%s\n",line);
    }
    return 0;
  }

  /* getLine reads one line to s[] and returns the lenght of it */
  int getLine(char line[], int maxline) {
    int c, i;
    for ( i=0; i < maxline-1 && (c=getchar()) != EOF && c != '\n'; ++i)
    line[i] = c;
    if (c == '\n') {
      line[i] = c;
      ++i;
    }
    line[i] = '\0';
    return i;
  }

  /* trim: removes blanks, tabs, newlines at the end of a line */
  int trim( char s[]) {
    int n;
    for ( n = strlen(s)-1; n >= 0; n--)
    if (s[n] != ' ' && s[n] != '\t' && s[n] != '\n')
      break;
    s[n+1] = '\0';
    return n;
  }
```

Task 5.3: *The `printf` statement in `main()` contains an `\n` . Change the `trim` so that it does not contain require it, but there is still a line break after each line.*

Submit your answers from Lab 1.5 to Moodle one week after the assignment

Lab 1.6: Some Gaming Programs

For relaxation some game programs: *"Eight queens problem"* (even for n queens), *"Towers from Hanoi"*, *"Horse Jump"*

6.1 The n queen problem:

n "Queens" are to be placed on a chess board of size n times n in such a way, that none is attacked by another in the sense of the rules of the queens' moves in chess.
Typical input and output:

```
Which number ( < 20) ? 8
1 5 8 6 3 7 2 4
```

This means: The Queen in the first column is in row 1, the Queen in the second column is in row 5 and so on. The function `int try()` implements a "Backtracking" - Algorithm.
Task 6.1.1: *Modify the program so that all possible solutions are output.*
Task 6.1.2: *Change the output so that a field of characters of size n is displayed, where the positions of the Queens are marked by a '*'.*

Source Code 1.31: Queens (Exercise: 6.1)

```
/*
 ===========================================================================
 Name        : queens.c
 Author      : A. Pretschner
 Version     : Kernighan & Ritchie 2. Edition
 Copyright   : Your copyright notice
 Description : queens and more
 ===========================================================================
 */

#define ZZ 20 /* number of rows and columns */
int max, QUEEN[ ZZ ], ROW[ ZZ ], DIA0[ 2*ZZ-1 ], DIA1[ 2*ZZ-1 ];
int main() {
  int i;
  printf("Which number ( < %d) ? ", ZZ);
  scanf("%d", &max);
  if ( try(0) )
    for (i=0; i < max; i++)
      printf("%d ", QUEEN[i]);
    printf("\n");
    return 0;
}

int try(int i) {
  int j, q;
  for (j = q = 0; q == 0 && j < max; j++) {
    if ( ROW[j] == 0 && DIA0[i+j] == 0 && DIA1[i-j+max-1] == 0) {
      QUEEN[i] = j+1;
      ROW[j] = DIA0[i+j] = DIA1[i-j+max-1] = 1;
      if (i == max - 1)
        q = 1;
      else
        if ( (q = try(i+1) ) == 0)
          ROW[j] = DIA0[i+j] = DIA1[i-j+max-1] = 0;
    }
  }
  return(q);
}
```

HINT: The N Queen is the problem of placing N chess queens on an N×N chessboard so that no two queens attack each other. For example, following is a solution for 4 Queen problem (https://www.geeksforgeeks.org/n-queen-problem-backtracking-3/). The N Queens Problem is a puzzle of placing N Queens on a N * N Chessboard in such a way that no two queens can attack each other i.e., no two queens should be placed horizontally, vertically or diagonally. In other words, any queen should not be in the same row, column or diagonal of any other queen. Another solution one can find here https://www.codingalpha.com/queens-problem-algorithm-c-program/

```
/*
 ============================================================================
 Name        : queens2.c
 Author      : A. Pretschner
 Version     :
 Copyright   : www.codingalpha.com
 Description : Queens and more
 ============================================================================
 */

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

int chess_board[20], count;

int display(int);
void queen_function(int, int);
int placeholder(int, int);

void queen_function(int row_value, int limit)
{
	int column_value;
	for(column_value = 1; column_value <= limit; column_value++)
	{
		if(placeholder(row_value, column_value))
		{
			chess_board[row_value] = column_value;
			if(row_value == limit)
			{
				display(limit);
			}
			else
			{
				queen_function(row_value + 1, limit);
			}
		}
	}
}

int placeholder(int row_value, int column_value)
{
	int count;
	for(count = 1; count <= row_value - 1; count++)
	{
		if(chess_board[count] == column_value)
		{
			return 0;
		}
		else
		{
			if(abs(chess_board[count] - column_value) == abs(count - row_value))
			{
				return 0;
			}
		}
	}
	return 1;
}

int display(int limit)
{
	int m, n;
	printf("\n\n\tPossible Solution %d:\n\n", ++count);
	for(m = 1; m <= limit; m++)
	{
		printf("\t[%d]", m);
	}
	for(m = 1; m <= limit; m++)
	{
		printf("\n\n[%d]", m);
		for(n = 1; n <= limit; n++)
		{
			if(chess_board[m] == n)
			{
				printf("\tQ");
			}
			else
```

```
                        {
                                printf("\t*");
                        }
                }
        }
}

void main()
{
        int limit;
        printf("\nEnter Number of Queens:\t");
        scanf("%d", &limit);
        if(limit <= 3)
        {
                printf("\nNumber should be greater than 3 to form a Matrix\n");
        }
        else
        {
                queen_function(1, limit);
        }
        printf("\n\n");
}
```

The Algorithm is based an the recursively call to the search function here `queen_function(int row_value, int limit)`. The program delivers:

```
 Enter Number of Queens:        5



          Possible Solution 1:

           [1]        [2]        [3]        [4]        [5]

[1]        Q        *        *        *        *

[2]        *        *        Q        *        *

[3]        *        *        *        *        Q

[4]        *        Q        *        *        *

[5]        *        *        *        Q        *

          Possible Solution 2:

           [1]        [2]        [3]        [4]        [5]

[1]        Q        *        *        *        *

[2]        *        *        *        Q        *

[3]        *        Q        *        *        *

[4]        *        *        *        *        Q

[5]        *        *        Q        *        *

          Possible Solution 3:
```

```

\dots
```

in case of $N05$ 10 different possible solutions.
The `queens.c` program taken from Kernighan & Ritchie defines the `try()` function, which also is called recursively. This programs delivers for the same amount of Queens $N = 5$ and brings back solution nbr.2 from above example!

```
Which number ( < 20) ? 5
1 3 5 2 4
```

Imortant : Have a glance at https://de.wikipedia.org/wiki/Damenproblem and see the animation of the recursive Back-Tracking-Algorithm. The queens problem can be solved efficiently by a recursive algorithm by understanding the problem with n queens in such a way that it is necessary to add another queen to each solution with n - 1 queens. Ultimately any queen problem can thus be traced back to a problem with 0 queens, which is nothing else than an empty chessboard.

6.2 The tower of Hanoi:

A tower consisting of a number of discs arranged in order of size (largest at the bottom) shall be moved from one position (”`left`”) to another (”`right`”). Only one auxiliary position (”`middle`”) may be used for intermediate storage of discs. Condition: Never place a disc on a smaller one in one of the positions. For the classic standard problem `n = 10`.

Source Code 1.32: Tower of Hanoi (Exercise: 6.2)

```c
/*
 ============================================================================
 Name        : tower.c
 Author      : A. Pretschner
 Version     : Kernighan & Ritchie 2. Edition
 Copyright   : Your copyright notice
 Description : tower of hanoi
 ============================================================================
 */

typedef enum {left, middle, right} tower;

int main() {
  int discnumber;
  void move();
  while (1) {
    printf("\ndiscnumber (Exit with 0): ");
    scanf("%d", &discnumber);
    if (discnumber <= 0)
      break;
    printf ("%d, %s\n",discnumber,"Discs need the following moves:");
    move (discnumber, left, middle, right);
  }
}

void move (int amount, tower from, tower withhelp, tower to) {
  void move_disc();
  if ( amount == 1 )
    move_disc (from, to);
  else {
    move ( amount - 1, from, to, withhelp);
    move_disc ( from, to );
    move ( amount - 1, withhelp, from, to);
  }
}
void move_disc ( tower from, tower to ) {
  void print_tower();
  printf ("Disc from ");
  print_tower (from);
  printf (" to ");
  print_tower (to);
  printf ("\n");
}

void print_tower ( tower which ) {
  switch (which) {
    case left : printf ("LEFT "); break;
    case middle : printf ("MIDDLE "); break;
    case right : printf ("RIGHT"); break;
  }
}
```

6.3 Jump on a horse:

The following program calculates a sequence of knight moves in the sense of chess, which covers an entire chessboard of size ZZ x ZZ in such a way that each square is reached exactly once.

Source Code 1.33: Horse Jump (Exercise: 6.3)

```
/*
 ============================================================================
 Name        : horsemove.c
 Author      : A. Pretschner
 Version     : Kernighan & Ritchie 2. Edition
 Copyright   : Your copyright notice
 Description : Horse Jump Program
 ============================================================================
 */
#include <stdio.h>
#include <stdlib.h>

#define ZZ 8 /* field array size */
int z;
int FIELD[ ZZ ][ ZZ ]; /* Field */
int a[] = {- 2, 1, -1, -2, -2, -1, 1, 2} ; /* coordinates of Jumper moves */
int b[] = {- 1, 2, 2, 1, -1, -2, -2, -1} ;
int x1, y1;
int try( int x, int y);
void Print();

int main() {
  z = 4;
  while ( 1 ) {
    printf("Rows: ");
    scanf("%d",&z);
    if ( z <= 1 || z > ZZ)
      exit(0);
  { int i,j;
    for (i=0;i<z;i++)
      for (j=0;j<z;j++)
        FIELD[i][j] = 0;
  }
  printf("Initial values x1 y1 = ");
  scanf("%d %d", &x1, &y1);
  FIELD[ x1-1 ][ y1-1 ] = 1;
  if ( try( x1-1, y1-1) )
    Print();
  else printf("\nNone Solutions\n");
  }
}

int try( int x, int y){
        int k, q, u, v;
        for (k = q = 0; q == 0 && k < 8; k++){
                u = x+a[k]; v = y+b[k];
                if ( 0 <= u && u < z && 0 <= v && v < z && FIELD[u][v] == 0 ){
                        FIELD[u][v] = FIELD[x][y] + 1;
                        if ( FIELD[u][v] == z*z )
                                q = 1;
                        else
                                if ( (q = try( u, v) ) == 0)
                                        FIELD[u][v] = 0;
                }
        }
        return(q);
}

void Print() {
  int i, j;
  printf("\n");
  for (i = 0; i < z; i++) {
          for (j = 0; j < z; j++)
                  printf("%3d", FIELD[i][j]);
          printf("\n");
  }
}
```

Typical In- output:

```
Rows: 6
Initial valuey x1 y1 =1 1

 1 16  7 26 11 14
34 25 12 15  6 27
17  2 33  8 13 10
32 35 24 21 28  5
23 18  3 30  9 20
36 31 22 19  4 29
```

Task 6.2.1: *Modify the program so that a ”cyclic” horse jump is found, then that the first field is reached by a jumper move from the last field.*

Submit your answers from Lab 1.6 to Moodle one week after the assignment

Lab 1.7: command line

Example 7.1: Command line arguments, file management

A program can process command line arguments. If the name of the executable program is `progname`, then you the call

`progname string1 string2 string3 ...`
which gives the program `progname` access to an array of strings
`*argv[]`
with the values
`argv[0] = progname, argv[1] = string1, argv[2] = string2, ...`
.

Source Code 1.34: command line (Exercise: 7.1)

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

Task 7.1: *Assume that the executable version of this program is called* `a.out`*. What does the call* `a.out this is only a test` *accomplish? Interpret the entire output!*
Since the number of arguments in the command line is not a priori fixed, you need the variable `argc`, which contains the current argument number after the call. This can be used, for example, to write a `copy` Program that with the command call `copy source target` copies the file source to the file target, without using the redirection technique of the operating system. For this you have to be able to open, close, read and write files. The `data type` is a ”file pointer” `FILE *`, which is available from the IO library by default as well as the functions `FILE *fopen(char *name, char * mode)`, `int fclose(FILE *fp)`, `int getc(FILE *fp)`, `int putc(int c, FILE *fp)` and others.

Source Code 1.35: continued (Exercise: 7.2)

```c
/*
 ============================================================================
 Name        : copyfile.c
 Author      : A. Pretschner
 Version     : Kernighan & Ritchie 2. Edition
 Copyright   : Your copyright notice
 Description : copies files
 ============================================================================
 */

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

int main( int argc, char *argv[] ) {
  int c;
  FILE *source, *target, *eopen();
  source = eopen(argv[1], "r");
  target = eopen(argv[2], "w");
  while ( (c = getc(source)) != EOF )
    putc(c, target);

  fclose(source);
  fclose(target);
}
  /* opening File  with  error handling */
  FILE *eopen(char *name, char *mode) {
          FILE *fp;
          if ( (fp = fopen(name,mode)) == NULL) {
                  perror(name); /* Library function delivers error diagnosis */
                  exit(1);
          }
  return fp;
  }
```

After translating with `gcc -o copy copy.c` you can use `copy copy.c ttt` to copy a copy of `copy.c` called `ttt`. If no file named qqq exists, the call `copy qqq ttt` results in `perror()` the error message `qqq: No such file or directory`.

Task 7.2: *Test the `perror()` error message assuming that the file to be opened is not readable or writeable. (Change with the Unix command `chmod`.)*

Source Code 1.36: continued (Exercise: 7.3)

```
/*
 ============================================================================
 Name        : copyfile.c
 Author      : A. Pretschner
 Version     : Kernighan & Ritchie 2. Edition
 Copyright   : Your copyright notice
 Description : concat files, output to stdout
 ============================================================================
 */

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

char *prog;
int main( int argc, char *argv[] ) {
  int i; FILE *quelle, *eopen(char *, char *);
  void filecopy(FILE *, FILE *);
  prog = argv[0];
  if (argc == 1)
    filecopy(stdin, stdout);
  else
    for(i = 1; i < argc; i++) {
      quelle = eopen(argv[i], "r");
      filecopy(quelle, stdout);
      fclose(quelle);
    }
  exit(0);
}

void filecopy(FILE *q, FILE *z) {
  int c;
  while ((c = getc(q)) != EOF)
  putc(c,z);
}

FILE *eopen(char *name, char *mode) {
  FILE *fp;
  if ( (fp = fopen(name,mode)) == NULL) {
    fprintf(stderr,"eopen in %s: File %s can't be opened in mode %s.\n", prog, name, mode);
    exit(1);
  }
  return fp;
}
```

Task 7.3: *Modify this program so that the output is written to a file when a command line* *argument is given in the call.*

Source Code 1.37: ASCII (Exercise: 7.4)

```c
/*
 ============================================================================
 Name        : ascii.c
 Author      : A. Pretschner
 Version     : Kernighan & Ritchie 2. Edition
 Copyright   : Your copyright notice
 Description : Ascii-Character output. Input octal, decimal, hex
               ASCII = American Standard Code for International Interchange
 ============================================================================
 */

#include <stdio.h>
#include <ctype.h>
#include <string.h>

char *name[] ={ "NUL","SOH","STX","ETX","EOT","ENQ","ACK","BEL",
"BS", "HT", "LF", "VT", "FF", "CR", "SO", "SI",
"DLE","DC1","DC2","DC3","DC4","NAK","SYN","ETB",
"CAN", "EM","SUB","ESC", "FS", "GS", "RS", "US",
"SP" };

char *input =
                " Input of single character provides its Ascii value \n\
                Input of Control-Char (^Z) provides its Ascii value\n\
                Input of multiple number provides its Ascii character\n\
                Input of Character with prefix\n\
                o, O or 0: octal read character\n\
                d or D: decimal read character\n\
                x or X: hexadecimal read character\n\
                tab or TAB: Ascii-Table output\n\
                Exit with Ctrl-C\n";

int main() {
  char s[10];
  int c,d;
  void table();
  printf( input );
  while (1) {
    printf(": ");
    scanf("%s",s);
    if ( strcmp(s, "tab") == 0 || strcmp(s, "TAB") == 0) {
      table();
      continue;
    }
  if ( strlen(s) == 1 ) c = s[0];
    else if ( s[0] == '^' )
            c = s[1] & 0x1f ;
         else if ( s[0] == 'o' || s[0] == 'O' || s[0] == '0')
                sscanf(&s[1],"%o",&c);
              else if ( s[0] == 'd' || s[0] == 'D')
                     sscanf(&s[1],"%d",&c);
                   else if ( isdigit( s[0] ) )
                          sscanf(s, "%d", &c);
```

```
                        else if ( s[0] == 'x' || s[0] == 'X' )
                                sscanf(&s[1],"%x",&c);

    c &= 0xff;
    if ( !iscntrl(c) && c != ' ')
      printf("ASCII(%c) = o%o d%d x%x\n", c,c,c,c);
    else if ( c == ' ' )
            printf("ASCII( ) = %s = o%o d%d x%x\n", name[c],c,c,c);
         else if ( c != 0x7f )
                printf("ASCII(^%c) = %s = o%o d%d x%x\n", c+0x40,name[c],c,c,c);
              else printf("ASCII(DEL) = o%o d%d x%x\n",c,c,c);
  }
}

void table() {
    int i;
    printf("\n");
    for (i=0; i < 31; i++)
            printf("\t%3d %2x %s\t%3d %2x %c\t%3d %2x %c\t%3d %2x %c\n",\
                            i,i,name[i],i+32,i+32,i+32,i+64,i+64,i+64,i+96,i+96,i+96);
    printf("\t%3d %2x %s\t%3d %2x %c\t%3d %2x %c\t%3d %2x DEL\n",\
                            i,i,name[i],i+32,i+32,i+32,i+64,i+64,i+64,i+96,i+96);
    printf("\n");
}
```

The program returns the output of single characters octal, decimal, hexadecimal and as character, example:

`ASCII(a) = o141 d97 x61`

for input of `a, 97, x61` or similar.

`ASCII(^ D) = EOT = o4 d4 x4`

for input `d ^` or similar, etc. Input of `tab` returns the Ascii-table in the following form:

```
 0  0 NUL        32 20            64 40 @        96 60 `
 1  1 SOH        33 21 !          65 41 A        97 61 a
 2  2 STX        34 22 "          66 42 B        98 62 b
 3  3 ETX        35 23 #          67 43 C        99 63 c
 4  4 EOT        36 24 $          68 44 D       100 64 d
 5  5 ENQ        37 25 %          69 45 E       101 65 e
 6  6 ACK        38 26 &          70 46 F       102 66 f
 7  7 BEL        39 27 '          71 47 G       103 67 g
 8  8 BS         40 28 (          72 48 H       104 68 h
 9  9 HT         41 29 )          73 49 I       105 69 i
10  a LF         42 2a *          74 4a J       106 6a j
11  b VT         43 2b +          75 4b K       107 6b k
12  c FF         44 2c,           76 4c L       108 6c l
13  d CR         45 2d -          77 4d M       109 6d m
14  e SO         46 2e .          78 4e N       110 6e n
15  f SI         47 2f /          79 4f O       111 6f o
16 10 DLE        48 30 0          80 50 P       112 70 p
17 11 DC1        49 31 1          81 51 Q       113 71 q
18 12 DC2        50 32 2          82 52 R       114 72 r
19 13 DC3        51 33 3          83 53 S       115 73 s
20 14 DC4        52 34 4          84 54 T       116 74 t
21 15 NAK        53 35 5          85 55 U       117 75 u
22 16 SYN        54 36 6          86 56 V       118 76 v
23 17 ETB        55 37 7          87 57 W       119 77 w
24 18 CAN        56 38 8          88 58 X       120 78 x
```

25	19	EM	57	39	9	89	59	Y	121	79	y
26	1a	SUB	58	3a	:	90	5a	Z	122	7a	z
27	1b	ESC	59	3b	;	91	5b	[	123	7b	-
28	1c	FS	60	3c	<	92	5c	\	124	7c	|
29	1d	GS	61	3d	=	93	5d	]	125	7d	"
30	1e	RS	62	3e	>	94	5e	^	126	7e	~
31	1f	US	63	3f	?	95	5f	_	127	7f	DEL

Task 7.4: *Expand the program so that the strings for the control characters are entered as well. (such as `LF`, `EOT`, `DEL`, etc.).*

Submit your answers from Lab 1.7 to Moodle one week after the assignment

Lab 1.8: Strcutures

Data types can be combined in a structured way to form more complex data ”structures”. For example, in a graphics package it can be useful to combine points on the screen - described by their pixel coordinates - to form a new data type. This is achieved by the following Declaration:

```
 struct point {   int x;   int y; };
```

This creates a new data type called `struct point`. Declarations are then possible such as

```
struct point a, b, z;
```

etc., syntactically analogous to `int a, b, c;`
The declaration

```
struct point p;
```

defines a variable `p` of type `struct point`. By

```
struct point p = {123, 456};
```

an initialization is made. Access on the components (= ”members”) of the structure is defined by the syntactic construction `structure-name.member`

Task 8.1: *What value does `sizeof(struct point)` return in this case ?*

In the above case, `p.x` is the x-component of p.
For example, you can use

```
printf("%d %d", p.x, p.y);
```

to print the coordinates of p, with

```
 double dist, sqrt(double); ... dist = sqrt((double)p.x * p.x + (double)p.y * p.y);
```

you can calculate the distance to the `point (0,0)`. Structures can be nested. With

```
 struct rect{ struct point ll; struct point tr; };
```

you can declare a data type `rect`, which defines a (horizontal) rectangle by fixing the lower left and top right corner point. After the declaration

```
struct rect screen;
```

`screen.ll.x` depicts the x-coordinate of the lower left corner etc.
Structures can be passed as variables to functions and returned as values by them. For example, the following is a function that turns two int into one p:

```
struct point mypoint(int x, int y) {
    struct point temp;
    temp.x = x;
    temp.y = y;
    return temp;
}
```

Two points can be added (as vectors), for example, :

```
struct point add(struct point p1, struct point p2) {
    p1.x += p2.x;
    p1.y += p2.y;
    return p1;
}
```

Through, `struct point *p;` a pointer p to objects of type `struct point` is explained. `(*p).x` then denotes the x-coordinate of a `struct point` object to which p points. Equivalent to this is the notation `p->x`.

Source Code 1.38: complex numbers (Exercise: 8.1)

```
/*
 ============================================================================
 Name        : complex.c
 Author      : A. Pretschner
 Version     : Kernighan & Ritchie 2. Edition
 Copyright   : Your copyright notice
 Description : complex numbers
 ============================================================================
*/

#include <stdio.h>
struct complex {
    double r;
    double i;
};

typedef struct complex cpx;
cpx makecpx(double, double);
cpx sum(cpx, cpx);
cpx product(cpx, cpx);
cpx power(cpx, int);
void compprint(cpx);

main() { /* calculates the power of complex numbers */
  int k;
  cpx basis, result;
  basis = makecpx(1,1);
  for (k=0; k < 10; ++k) {
    result = power( basis, k );
    printf("%2d ", k );
    compprint(result);
  }
}

cpx makecpx(double r, double i) { /* makes complex number */
  cpx tmp; /* Real Part r, Imaginary Part i */
  tmp.r = r; tmp.i = i;
  return tmp;
}

cpx sum(cpx a, cpx b) { /* Sum of two  complex numbers */
  a.r += b.r;
  a.i += b.i;
  return a;
}

cpx power( cpx basis, int expo) { /* basis power exponent, basis complex */
  cpx u = {- 1, 0};
  while (expo > 0) {
    if (expo % 2) {
      expo--;
      u = product(basis, u);
    }
    else {
      expo /= 2;
      basis = product(basis, basis);
    }
  }
return u;
}

cpx product(cpx x, cpx y) {/* Product of two complex numbers */
  cpx u;
  u.r = x.r * y.r - x.i * y.i;
  u.i = x.r * y.i + x.i * y.r;
  return u;
}

void compprint(cpx z) { /* prints  complex number */
  if (z.r != 0 && z.i != 0)
    printf("%5.2f + %5.2f * i\n", z.r, z.i);
  else if (z.r == 0 && z.i != 0)
        printf("%13.2f * i\n", z.i);
        else if (z.r != 0 && z.i == 0)
              printf("%5.2f\n", z.r);
              else printf("0\n");
}
```

The program outputs:

```
0          1.00
1          1.00 +  1.00 * i
2                  2.00 * i
3         -2.00 +  2.00 * i
4         -4.00
5         -4.00 + -4.00 * i
6                 -8.00 * i
7          8.00 + -8.00 * i
8         16.00
9         16.00 + 16.00 * i
```

Task 8.2: *Complete this program by adding subtraction, absolute value and applications of your choice etc.*

Task 8.3: *Define a data type `rat`, which can calculate rational numbers by pairs of integers and implement the basic arithmetic operations of the fraction, based on the int arithmetic.*

Submit your answers from Lab 1.8 to Moodle one week after the assignment

Lab 1.9: Using Functions Names

Function names are managed as pointers to the beginning of the code for the functions. Arrays of Functions can be constructed. The program uses the math library and must therefore be compiled with `gcc isqrt.c -lm`.

Source Code 1.39: root functions (Exercise: 9.1)

```
/*
 ============================================================================
 Name        : rootfunctions.c
 Author      : A. Pretschner
 Version     : Kernighan & Ritchie 2. Edition
 Copyright   : Your copyright notice
 Description : integer root functions
 ============================================================================
*/

#include <stdio.h>
#include <time.h>  /* typedef int time_t ; replacement, if time.h does not exist */
#include <math.h> /* For comparison: With the floating point library */
                                 /* Setting Compiler flag -lm is necessary*/


/* Four forward declarations of functions: */
int isqrt(); int isqrt_0(); int isqrt_1(); int isqrt_2(); int isqrt_3();

/* Entry of these function names in an array: */
int (*fct[])() = {isqrt, isqrt_0, isqrt_1, isqrt_2, isqrt_3, NULL};

/* Following program measures the runtimes of these functions: */
int main() {
  int root, i, iter = 1;
  long n;
  time_t tta, tte;
  int (**f)();
  while( iter ) {
    printf("\nIteration (Exit with 0): " );
    scanf("%d",&iter);
    if ( iter == 0) break;
    printf("Root of: ");
    scanf("%ld",&n);
    for( f = fct; *f; f++) {
      time( &tta );
      for ( i = 0; i < iter; i++)
              root = (**f)(n);
      time( &tte );
      printf("result = %u, Time: %ld sec.\n",root,tte-tta);
    }
  }
}

/* The following functions determine the whole number amount of the Square root of n. */
int isqrt(long n) { /* only with  Bitshifts, Additions und Subtractions */
  register int i;
  register long root=0, left=0;
  for ( i = (sizeof(long)<<3) - 2; i >= 0; i -= 2 ) {
          left = (left<<2) | (n>>i & 3) ;
          root <<= 1;
          if ( left > root )
                  left -= ++root, ++root;
            }
  return (int)(root>>1) ;
}

int isqrt_0(long n) { /* Variation of isqrt */
  register unsigned long m, root = 0, left = n;
  for ( m=(long)1; m < n>>2; m <<= 2) ;
  for ( ; m; m >>= 2 ) {
    if ( ( left & -m ) > root )
    left -= ( root += m ), root += m;
    root >>= 1;
  }
  return (int)root;
}

int isqrt_1(long n) { /* Variation of isqrt */
  register unsigned long m, root = 0, left = n;
  for ( m = (long)1<<( (sizeof(long)<<3) - 2); m; m>>= 2 ) {
    if ( ( left & -m ) > root )
      left -= ( root += m ), root += m;
    root >>= 1;
  }
  return (int)root;
}

int isqrt_2(long n) { /* Integer variant of Newton's Method */
  register unsigned int a = 0xffffffff, b = a-1;
  while ( b < a ) {
    a = b;
    b = (a+n/a) >> 1;
  }
  return a;
}

int isqrt_3(long n) {
  return (int) sqrt( (double)n );
}
```

Task 9.1: *Add another function to the array `fct`. Confirm the pointer character of the Function names by outputting the pointer values with `printf`.*

Example: Program for resolving difficult declarations

The following program `dcl` converts a formal C declaration into an English description.

Task 9.2: *Use program dcl to analyze all following declarations.*

```
FILE *fopen()
fopen: function returning pointer to FILE
char *argv[]
argv: array[] of pointer to char
char **argv
argv: pointer to pointer to char
char *name[]
name: array[] of pointer to char
double *fct()
fct: function returning pointer to double
double (*fct)()
fct: pointer to function returning double
int *fct[]()
fct: array[] of function returning pointer to int
int *fct()[]
fct: function returning array[] of pointer to int
int (*fct)[]()
fct: pointer to array[] of function returning int
int (*fct)()[]
fct: pointer to function returning array[] of int
```

Source Code 1.40: Parser (Exercise: 9.2)

```
/*
```

```c
 ============================================================================
 Name        : dcl.c
 Author      : A. Pretschner
 Version     : Kernighan & Ritchie 2. Edition
 Copyright   : Your copyright notice
 Description : Parser for C-Declarations K&R 2nd, p. 123
 ============================================================================
*/


#include <stdio.h>
#include <string.h>
#include <ctype.h>
#define MAXTOKEN 100
enum {NAME, PARENS, BRACKETS };
void dcl(void);
void dirdcl(void);
int gettoken(void);
int tokentype;
char token[MAXTOKEN];
char name[MAXTOKEN];
char datatype[MAXTOKEN];
char out[1000];

int main() { /* converts declarations into word descriptions */
  while ( gettoken() != EOF) {
    strcpy(datatype, token);
    out[0] = '\0';
    dcl(); /* parse rest of line */
    if ( tokentype != '\n')
      printf("syntax error\n");
    printf("%s: %s %s\n", name, out, datatype);
  }
  return 0;
}

void dcl(void) { /* dcl: Parser for declarator */
  int ns;
  for (ns=0; gettoken() == '*';) /* count *'s */
    ns++;
  dirdcl();
  while ( ns-- > 0 )
    strcat(out," pointer to");
}

void dirdcl(void) { /* dirdcl: Parser for direct declarator */
  int type;
  if( tokentype == '(') { /* ( dcl ) */
    dcl();
    if (tokentype != ')')
      printf("error: missing )\n");
  }
  else        if (tokentype == NAME)
        strcpy(name, token);
      else printf("error: expected name or (dcl)\n");
  while ((type=gettoken()) == PARENS || type == BRACKETS )
  if (type == PARENS)
    strcat( out, " function returning");
  else {
    strcat( out, " array");
    strcat( out, token);
    strcat( out," of");
  }
}

int gettoken(void) { /* delivers next token */
  int c, getch(void);
  void ungetch(int);
  char *p = token;
  while (( c = getch()) == ' ' || c == '\t')
    ;
  if ( c == '(') {
    if (( c = getch()) == ')') {
      strcpy(token, "()");
      return tokentype = PARENS;
    }
    else {
      ungetch(c);
      return tokentype = '(';
    }
  }
  else        if ( c == '[') {
          for ( *p++ = c; (*p++ = getch()) != ']';) ;
          *p = '\0';
          return tokentype = BRACKETS;
        }
        else if ( isalpha(c) ) {
                for ( *p++ = c; isalnum( c = getch()); )
                *p++ = c;
                *p = '\0';
                ungetch(c);
                return tokentype = NAME;
             }
             else return tokentype = c;
}

/* The following program pair allows the "writing back" of characters
into an input stream, see also man ungetc */
#define BUFSIZE 100 /* K&R 2nd, p. 79 */
char buf[BUFSIZE];
int bufp = 0;
int getch( void ) {
  return ( bufp > 0) ? buf[--bufp] : getchar();
}

void ungetch( int c ) {
  if ( bufp >= BUFSIZE )
    printf("ungetch: too many characters\n");
  else
    buf[bufp++] = c;
}
```

Submit your answers from Lab 1.9 to Moodle one week after the assignment

Lab 1.10: Operator precedence in C

Operatoren	Assoziativitaet
`() [] -> .`	left to right
`! ~ ++ -- + - * & (type) sizeof`	right to left
`* / %`	left to right
`+ -`	left to right
`<< >>`	left to right
`< <= > >=`	left to right
`== !=`	left to right
`&`	left to right
`^`	left to right
`|`	left to right
`&&`	left to right
`||`	left to right
`?:`	right to left
`= += -= *= /= %= &= ^= |= <<= >>=`	right to left
`,`	left to right

The unary $+, -, *,$ & have higher precedence than the binary forms.
The associativity controls whether an operand is to be placed between two operators of the same priority. The left or right operator of these operators is taken first, if both possibilities are syntactically meaningful:

```
a - b + c
```
therefore means
```
(a - b) + c
```
, not
```
a - (b + c)
```
, while
```
a = b += 3
```
means
```
a =(b += 3)
```
or more detailed:
```
b = b+3, a = b
```
.

Task 10.1: *What does* `-a++` *mean? Is there a difference to* `-++a` *? Does* `(-a)++` *make sense?*

Example for a "self-referential" structure and for dynamic memory management

Source Code 1.41: binary tree (Exercise: 10.1)

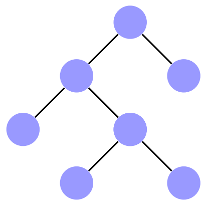
```
/*
 ===========================================================================
 Name        : tree.c
 Author      : A. Pretschner
 Version     : Kernighan & Ritchie 2. Edition
 Copyright   : Your copyright notice
 Description : binary tree as example for dynamic storage management
 ===========================================================================
 */
#include<stdio.h>
#include<ctype.h>
#include<string.h>
#include<stdlib.h>

#define MAX_WORD 40
```

```c
char *progname;
struct node {
  char *word;
  int number;
  struct node *left;
  struct node *right;
  };

typedef struct node pnod;
FILE *source, *target;
/* main reads the words from the source file and returns them
        lexicographically sorted by frequency
        to the target file.
*/

/*forward declarations*/
int readword(char *w, int lim);
void inorder(pnod *p);

int main(int argc, char *argv[]) {
  pnod *subtree, *search();
  char word [ MAX_WORD ] ;
  int c;
  FILE *eopen();
  progname = argv[0];
  if ( argc != 3 ) {
    fprintf(stderr,"Syntax: %s sourcefile targetfile\n",progname);
    exit(1);
  }
  source = eopen(argv[1],"r");
  subtree = NULL;
  while ( ( c = readword( word, MAX_WORD) ) != EOF )
          if (c == 'a')
                  subtree = search( subtree, word );
  fclose(source);
  target = eopen(argv[2],"w");
  inorder( subtree );
  fclose(target);
}

/* readword reads a word of length < lim from a file in to
            memory to which w points and returns 'a' if it
            has really read alpha-characters, but EOF in the case of to
            large word length. If non-alpha characters are read,
            these are returned individually.
*/
int readword(char *w, int lim) {
  int c;
  while ( !isalpha( c = getc(source) ) )
          if ( c == EOF )
                  return c;
  *w++ = c;
  lim--;
  while ( isalpha( c = getc(source) ) && lim > 0 ) {
          *w++ = c; lim--;
  }
  if (lim == 0) {
          printf("%s: Error: Word too long ", progname);
      return EOF;
  } else
          *w = 0;
  if (lim < MAX_WORD)
          return 'a';
  else
          return c;
}

FILE *eopen(char *file,char *mode) {
  FILE *fp, *fopen();

  if ( (fp = fopen(file, mode)) != NULL )
    return fp;
  fprintf(stderr, "\
    %s: File %s can not in modi  %s not be opened\n",\
    progname, file, mode );
  exit(1);
}

/* search searches the tree for the occurrence of the word w,
          if necessary, enters it lexicographically correctly or increases it
          and returns the pointer to its node.
          The c-function strcmp compares two character strings lexicographically
          with return of a negative number, 0 or a positive number.
*/
pnod *search(pnod *p, char *w) {
  pnod *p_node();
  char *remember_word();
  int cond;
  if (p == NULL) {
    p = p_node();
    p->word = remember_word(w);
    p->number = 1;
    p->left = p->right = NULL;
  }
  else if ((cond = strcmp(w, p->word)) == 0)
          p->number++;
       else if (cond < 0)
                          p->left = search(p->left, w);
                    else p->right = search(p->right, w);
  return p ;
}

/* p_node returns a pointer to free disk space for
          a node variable back.  */
pnod *p_node(void) {
   return (pnod *)malloc( sizeof( pnod ) ) ;
}

/* inorder outputs the subtree, to which p points, in in order.
           (recursive version!)
*/
```

```
void inorder(pnod *p) {
  if (p != NULL) {
    inorder(p->left);
    fprintf( target, "%4d %s\n", p->number, p->word);
    inorder(p->right);
  }
}

/* remember_word  stores the word whose first character s points to,
                  and specifies a pointer to its new location.
                  The c-functions strlen, strcpy give the length of a
                  Return or copy character string.
*/
char *remember_word(char *s) {
  char *p;
  if ( ( p = (char *)malloc( strlen(s)+1 ) ) != NULL )
          strcpy(p, s);
  return p ;
}
```

Note: The above example demonstrates the usage of *binary trees*. A **binary tree** is a *tree data* structure in which each node has at most **two** children, which are referred to as the **left child** and the **right child** (https://en.wikipedia.org/wiki/Binary_tree). In the example the simple binary-tree-definition is use:

- A **rooted binary tree** has a root node and every node has at most two children.
- A **full binary tree** (sometimes referred to as a *proper* or *plane* binary tree) is a tree in which every node has either 0 or 2 children. Another way of defining a full binary tree is a recursive definition. A full binary tree is either a single vertex or a tree whose root node has two subtrees, both of which are full binary trees.



A full binary tree

```
int main(int argc, char *argv[]) {
 pnod *subtree, *search();
 char word [ MAX_WORD ] ;
 int c;
 FILE *eopen();
 progname = argv[0];
 if ( argc != 3 ) {
   fprintf(stderr,"Syntax: %s sourcefile targetfile\n",progname);
   exit(1);
 }
 source = eopen(argv[1],"r");
 subtree = NULL;
 while ( ( c = readword( word, MAX_WORD) ) != EOF )
         if (c == 'a')
                 subtree = search( subtree, word );
 fclose(source);
 target = eopen(argv[2],"w");
 inorder( subtree );
 fclose(target);
}
```

One can see, that in `main()` the program must be called with two arguments: `sourcefile` and targetfile. The `sourcefile` contains the inspected text and the result (sorting, counting ...) will be stored in the `targetfile`. The `while ()` loop reads recursively the words of the given text and builds up the subtrees for the found different words and count them up too. The **binary tree** Algorithm was implemented in function `search()`. Make some Debug-Sessions to understand the program behavior.

Example `sourcefile`:

```
This short text example shows the application of binary trees.
Binary trees are the most commonly used subspecies of trees in computer science.
In contrast to other types of trees, the nodes of a binary tree can have only two direct descendants.
In most cases it is required that the child nodes can be clearly divided into a left and a right child.
A good example for such a binary tree is the pedigree, where the parents have to be modeled by the child nodes.
A binary tree is either empty or it consists of a root with a left and right subtree,
which in turn are binary trees. If a subtree is empty, the corresponding child node is called missing.
In graphical representations the root is usually placed at the top and the leaves at the bottom.
Accordingly, a path from the root to the leaf is one from top to bottom.
```

`targetfile`:

```
   2 A
   1 Accordingly
   1 Binary
   1 If
   3 In
   1 This
   8 a
   3 and
   1 application
   2 are
   2 at
   2 be
   5 binary
   2 bottom
   1 by
   1 called
   2 can
   1 cases
   4 child
   1 clearly
   1 commonly
   1 computer
   1 consists
   1 contrast
   1 corresponding
   1 descendants
   1 direct
   1 divided
   1 either
   2 empty
   2 example
   1 for
   2 from
   1 good
   1 graphical
   2 have
   2 in
   1 into
   7 is
   2 it
   1 leaf
   1 leaves
   2 left
   1 missing
   1 modeled
   2 most
   1 node
```

```
      3 nodes
      5 of
      1 one
      1 only
      1 or
      1 other
      1 parents
      1 path
      1 pedigree
      1 placed
      1 representations
      1 required
      2 right
      3 root
      1 science
      1 short
      1 shows
      1 subspecies
      2 subtree
      1 such
      1 text
      1 that
     14 the
      4 to
      2 top
      3 tree
      5 trees
      1 turn
      1 two
      1 types
      1 used
      1 usually
      1 where
      1 which
      1 with
```

Explain why currently the program does not work for *german Umlaute*?
Task 10.2 *Modify the program so that words independently case-insensitive are sorted alphabetically.*

Submit your answers from Lab 1.10 to Moodle one week after the assignment

C++ - Laboratory Training Files

Lab 2.1: Introduction into c++

The following exercises and samples are taken Literature:
Kirch, Ulla; Prinz, Peter: C++ Learn and apply professionally (7nd Edition 2015, mitp Publishing, ISBN 978-3-95845-028-8).
These exercises include sample programs and easy exercises to learn the language C++. They are not a complete script for the course. Detailed explanations are given in the lecture.

Source Code 2.42: To Upper (Exercise: 11.1)

```
/*
 ============================================================================
 Name        : toupper.cpp
 Author      : A. Pretschner
 Version     : Kirch & Prinz, 7. Edition
 Copyright   : Your copyright notice
 Description : Filter to convert to capital letters
               Umlaute are used in the extended ASCII character set
               since this is the character set at execution time.
               Be careful and set iso-8859-1 encoding as project's resource text file encoding!
                                The input from the keyboard with Ctrl+Z (DOS/Windows) or Ctrl+D (Unix).
 ============================================================================
 */


#include <iostream>
#include <cctype>
using namespace std;

int main()
{
   char c;
   while ( cin.get(c) )       // read character as long it is possible
   {
      switch( c )
      {
        case 'ä'   :  c = 'Ä';         // German Umlaute
//          case '\x84':  c = '\x8E';       // literal ä
                       break;
        case 'ü'   :  c = 'Ü';         // literal ü
//          case '\x94':  c = '\x99';
                       break;
        case 'ö'   :  c = 'Ö';         // literal ö
//          case '\x81':  c = '\x9A';
                       break;
        case 'ß'   :   cout.put('S'); c = 'S';    // literal ß
//         case '\xE1':
                       break;

         default:   c = toupper(c);    // remaining literals
       }
      cout.put(c);     // output converted literal
   }
   return 0;
}
```

The above example demonstrates the usage of "filter programs" which are used to convert one literal into other literal. `toupper()` and `tolower()` fulfill appropriate literal transformations. In `c++` exist different Macros, which names start with `is...` `islower(c)` checks if `c` contains lower literals and if so returns the value `true` otherwise `false`.

Source Code 2.43: isdigit() (Exercise: 11.2)

```
char c; cin >>c;  //read input character
if( !isdigit(c))  //checks
  cout << "Character is not a number\n";
```

Next Example shows one possible implementation:

Source Code 2.44: islower() (Exercise: 11.3)

```
#define toupper(c) \
  (islower(c) ? ((c) - 'a' + 'A') : (c))
```

This code make use of the constant difference between lower and upper literal positions in the ASCII or EBCDIC coding standard. With help of `kbhit()` one is able to ask if the keyboard was hit by user. If true the pressed down key can be read with `getchar()`. Both functions are not part of the ANSI-Standard but are available on most systems and call the operating system routines defined in `conio.h`

HINT : The functions `getchar()` and `kbhit()` are special functions defined by the program itself! Important to note is also the get introduced with the includes `termios.h` and `mymacros.h`. Unfortunately the Eclpise Editor does not resolve C/ C++ Macros correctly. If necessary one can suppress the Error Warnings.

Source Code 2.45: kbhit() (Exercise: 11.4)

```
int c;
if (kbhit() != 0){ //space key pressed?
  c = getchar();   //reading character
  if ( c==27)      //character eq ESC?
    //....
}
```

Unfortunately those functions are not available under Linux. The next code demonstrates how these function can be implemented by your self:

Source Code 2.46: kbhit() in Linux (Exercise: 11.5)

```
#include         <string>
#include        <stdio.h>
#include         <iostream>
#include         <cstdint>
#include         <cstring>
#include         <sys/ioctl.h>
#include         <termios.h>

int getch();

int getch(){
        static int ch = -1, fd = 0;
        struct termios tnew, told;
        fd = fileno(stdin);
        tcgetattr(fd, &told);
        tnew = told;
        tnew.c_lflag &= ~(ICANON|ECHO);
        tcsetattr(fd, TCSANOW, &tnew);
        ch = getchar();
        tcsetattr(fd, TCSANOW, &told);
   return ch;
}

int kbhit(void);

int kbhit(void) {
        struct termios term, oterm;
        int fd = 0;
        int c = 0;
        tcgetattr(fd, &oterm);
        memcpy(&term, &oterm, sizeof(term));
        term.c_lflag = term.c_lflag & (!ICANON);
        term.c_cc[VMIN] = 0;
        term.c_cc[VTIME] = 1;
        tcsetattr(fd, TCSANOW, &term);
        c = getchar();
        tcsetattr(fd, TCSANOW, &oterm);
        if (c != -1)
                ungetc(c, stdin);
        return ((c != -1) ? 1 : 0);
}
```

In case of special keys like `F1, F2 ... Insert, Delete ...` `getchar()` returns first `0` or `0xE0` (=224), afterwards the second call delivers the pressed key character.

Source Code 2.47: Macros (Exercise: 11.6)

```
/*
 ============================================================================
 Name        : mymacros.h
 Author      : A. Pretschner
 Version     : Kirch & Prinz, 7. Edition
 Copyright   : Your copyright notice
 Description : Header-file with Macros
               ABS, MIN, MAX, CLS, LOCATE, COLOR, NORMAL, INVERS
               and symbolic constants for the colors.
 ============================================================================
 */

#ifndef _MYMAKROS_
#define _MYMAKROS_

#include <iostream>
using namespace std;

// ------------------------------------------------------------
// Macro MIN
// call:  MIN(x,y)
// returns minimum of x and y

#define MIN(a,b) ( (a) <= (b) ? (a) : (b))

//------------------------------------------------------------
```

```
// Macros for the display controller
// --------------------------------------------------------------
// Macro CLS
// call:  CLS;
// clears display

#define CLS   (cout << "\033[2J")  // clear display

// --------------------------------------------------------------
// Macro LOCATE
// call:  LOCATE(row, column);
// setting cursor at given position (row, column).
// (1,1) is left upper Edge.

#define LOCATE(z,s) (cout <<"\033["<< (z) <<';'<< (s) <<'H')

// --------------------------------------------------------------
// Macro COLOR
// call:  COLOR(foreground, background);
// setting foreground and background color for next output.

#define COLOR( v, h) (cout << "\033[1;3"<< (v) \
                           <<";4"<< (h) <<'m' << flush)
//  1: foreground light
// 3x: foreground color x
// 4x: background color  x

// color values for Macro COLOR
// sample call: COLOR( WHITE,BLUE);

#define BLACK         0
#define RED     1
#define GREEN   2
#define YELLOW  3
#define BLUE    4
#define MAGENTA 5
#define CYAN    6
#define WHITE   7

// --------------------------------------------------------------
// Macro NORMAL
// call:  NORMAL;
// resetting display attributes to default.

#define NORMAL  (cout << "\033[0m")

// --------------------------------------------------------------
// Macro INVERS
// call:  INVERS;
// Following output will be inverse.

#define INVERS  (cout << "\033[7m")

#endif   //  _MYMAKROS_
}
```

Task 11.1: *Write a) the Macro ABS which returns the absolute value of a number and b)*

the Macro MAX, which determines the greater value of two numbers. In each case use the operator `?:` *Write these Macros into the header file* `mymacros.h`*. Write the* `mymacros_t.cpp` *Application to test them by executing it directly on the console (because of display drivers not available inside the Eclipse Console!)*
Next example depicts an old computer game - the bouncing ball:

Source Code 2.48: Ball Example (Exercise: 11.7)

```
/*
 ============================================================================
 Name        : ball1.cpp
 Author      : A. Pretschner
 Version     : Kirch & Prinz, 7. Edition
 Copyright   : Your copyright notice
 Description : simulates bouncing ball
 ============================================================================
 */


#include <iostream>
#include <string>
using namespace std;

#define DELAY  10000000L             // delay
#define CLS   (cout << "\033[2J")    // clear display
#define LOCATE(z,s) (cout <<"\033["<< z <<';'<< s <<'H')
        // positioning Cursor in row z and column s

int main() {
        int x = 2, y = 3, dx = 1, velo = 0;
        string floor(79, '-'),header = "****  BOUNCING BALL  ****";

        CLS;
        LOCATE(1,25);  cout << header;
        LOCATE(25,1);  cout << floor;

        while(true) {      // Ball shall bounce for ever
                LOCATE(y,x);
                cout << 'o' << endl;                // show Ball
                for( long warten = 0; warten < DELAY; ++warten)
                        ;
                if(x == 1 || x == 79)
                        dx = -dx;   // on wall?
                if( y == 24 ) {                      // on floor?
                        velo = - velo;
                        if( velo == 0 )
                                velo = -7;   // new trigger
                }
                velo += 1;                   // speed = 1

                LOCATE(y,x);
                cout <<  ' ';    // clear display
                y += velo;  x += dx;        // new position
   }
   return 0;
}
```

Task 11.2: *Complement the above example (*`ball1.cpp`*) so that a) the ball is shown light*

white on blue background, b) the program is finished with the key stroke `ESC` and finally c) the ball speed can be speed up with `key +` and speed down with `key -`. For the solution you need to use the mentioned functions `kbhit()` and `getchar()`
Task 11.3: *Display the content of an arbitrary text file. Write the filter program to suppress all control sequences with exception `\n` and `\t` . Usually control sequences are in the range of 0 to 31 (ASCII-Standard). Concatenating control sequences shall be substituted with one single space, one single character in between of two subsequent control sequences shall be suppressed.*
HINT: Because of the last demand, you need to memorize the predecessor so that you are able to suppress it in case of an subsequent control sequence!

Submit your answers from Lab 2.1 to Moodle one week after the assignment

Lab 2.2: Reference and Pointer

This Laboratory continues section Lab 2.2: Arrays and Pointer of the last chapter Lab 2.2: Introduction into c. Here you will learn the usage, definition and implementation of pointers and their return values. Access to pointer variables may be through `Call by Reference` or `Call by Value`.
A **Reference** is another name (alias) for an already existing object. Per definition no new memory allocation is performed for that reference.

Source Code 2.49: References (Exercise: 12.1)

```
/*
 ============================================================================
 Name        : ref1.cpp
 Author      : A. Pretschner
 Version     : Kirch & Prinz, 7. Edition
 Copyright   : Your copyright notice
 Description : demonstrates reference definition and usage
 ============================================================================
 */


#include <iostream>
#include <string>
using namespace std;

float x = 10.7F;                          // global

int main()
{
   float  &rx = x;           // rx : local Reference to x
// double &ref = x;          // Error: wrong data type !

   rx *= 2;

   cout << "   x = " <<  x << endl
        << "  rx = " << rx << endl;

   const float& cref = x;    // cref : read-only-Reference

   cout << "cref = " <<  cref << endl;   // ok!
// ++cref;                   // Error: read-only!

   const string str = "I'm an constant string!";
// str = "Das geht nicht!";  // Error: str constant!
// string& text = str;       // Error: str constant!
   const string& text = str; // ok!

   cout << text << endl;     // ok! reading text .

   return 0;
}
```

The **Definition** of an `Reference` is done through `&` (Ampersand). If `T` is a type, than `T&` means ”Reference to T”

Source Code 2.50: References (Exercise: 12.2)

```
float x = 10.7;
float& rx = x;
// or alternatively
float &rx = x;
```

`rx` is now another name for variable `x` of type ”Reference to float”
References are strictly bind to variables (here `x`). The character `&` for reference occurs only in the declaration part of that variable and has nothing to do with the address operator `&`. The address operator delivers the address of the referenced object. `&rx`. A Reference must be initialized through definition and can not altered later.
Call by Reference
this can be done ether with References and/or Pointers. One parameter with Reference type is an alias name for the argument. During the function call the reference parameter is initialized directly to the object. The function works therefor directly with those objects.
`void test (int& a) {++a;}`
After the function call the referenced variable `var` is incremented accordingly `test (var);`
In difference to **Call by Value** for the function argument no complex statement like `a+b` is allowed, because that argument needs to be an object with according type.

Source Code 2.51: References (Exercise: 12.3)

```
/*
 ============================================================================
 Name        : ref3.cpp
 Author      : A. Pretschner
 Version     : Kirch & Prinz, 7. Edition
 Copyright   : Your copyright notice
 Description : return reference type
 ============================================================================
 */


#include <iostream>
#include <string>
using namespace std;

double& refMin( double&, double&);  // return Reference to Minimum

int main(){
   double x1 = 1.1,
          x2 = x1 + 0.5,
          y;

   y = refMin( x1, x2);             // assign Minimum to y.
   cout << "x1 = " << x1 << "      "
```

```
         << "x2 = " << x2 << endl;
    cout << "Minimum: " << y  << endl;

    ++refMin( x1, x2);                // ++x1, because x1 is Minimum
    cout << "x1 = " << x1 << "     "      // x1 = 2.1
         << "x2 = " << x2 << endl;        // x2 = 1.6

    ++refMin( x1, x2);                // ++x2, because now  x2 is Minimum

    cout << "x1 = " << x1 << "     "      // x1 = 2.1
         << "x2 = " << x2 << endl;        // x2 = 2.6

    refMin( x1, x2) = 10.1;           // x1 = 10.1, because now x1 is Minimum

    cout << "x1 = " << x1 << "     "      // x1 = 10.1
         << "x2 = " << x2 << endl;        // x2 = 2.6

    refMin( x1, x2) += 5.0;           // x2 += 5.0, because now x2 is Minimum

    cout << "x1 = " << x1 << "     "      // x1 = 10.1
         << "x2 = " << x2 << endl;        // x2 = 7.6
    return 0;
}

double& refMin( double& a, double& b){  // Reference to Minimum
    return a <= b ? a : b;
}
```

Pointer

A pointer represents the address and type of another object. The address operator `&` delivers already a pointer to the object. `&var // object address of variable var`
The statement `&var` describes one constant pointer. In **C++** pointer variables can be used. That are variables, which can store the address to another object. `int *ptr;// or: int* ptr;`. This statement defines the variable `ptr` with type `int*`. During declaration the character `*` means always "pointer to". Types of pointers are derived types `T* int a, *p, &r=a;` After the pointer declaration it's necessary to point to the memory area like: `ptr = &var;`
References and Pointers are similar - both point to another object in the memory, a pointer thus is not an alias name than an own object in difference to the only addressed object by the Reference.
Some more Pointer examples follow::

Source Code 2.52: Pointer (Exercise: 12.4)

```
/*
 ============================================================================
 Name        : pointer1.cpp
 Author      : A. Pretschner
 Version     : Kirch & Prinz, 7. Edition
 Copyright   : Your copyright notice
 Description : demonstrates pointer values and variables
 ============================================================================
 */


#include <iostream>
using namespace std;
```

```
int var, *ptr;  // variable definitions var and ptr

int main(){          // variable values and addresses
                   // output var and ptr.
   var = 100;
   ptr = &var;

   cout << " Value of var:      " <<  var
        << "   Address of var: " <<  &var
        << endl;
   cout << " Value of ptr: "       <<  ptr
        << "   Address of ptr: " <<  &ptr
        << endl;

   return 0;
}
```

Now the address of the variables `x` and `y` are used as arguments of `swap()`. Thus, the arguments of `swap()` are declared as float pointers:

Source Code 2.53: Pointer (Exercise: 12.5)

```
/*
 ============================================================================
 Name        : pointer2.cpp
 Author      : A. Pretschner
 Version     : Kirch & Prinz, 7. Edition
 Copyright   : Your copyright notice
 Description : Function swap() demonstrates pointer usage as input parameter (argument)
 ============================================================================
 */


#include <iostream>
using namespace std;

void swap( float *, float *);   // Prototyp of swap()

int main()
{
  float x = 11.1F;
  float y = 22.2F;

  cout << "x and y before swap:  "
       << x << "   " << y << endl;

  swap( &x, &y );

  cout << "x and y after swap: "
       << x << "   " << y << endl;

  return 0;
}

void swap(float *p1, float *p2)
{
  float temp;          // help variable
  temp = *p1;           // through call p1 points to x
  *p1  = *p2;           // and p2 to y
  *p2  = temp;
}
```

Source Code 2.54: Reference (Exercise: 12.6)

```
/*
 ============================================================================
 Name        : ref4.cpp
 Author      : A. Pretschner
 Version     : Kirch & Prinz, 7. Edition
 Copyright   : Your copyright notice
 Description : Reference type statements for string operations
 ============================================================================
 */


#include <iostream>
#include <string>
#include <cctype>                 // includes toupper()
using namespace std;

void strToUpper( string& );      // Prototype

int main()
{
   string text("Test with Assignments\n");

   strToUpper(text);
   cout << text << endl;

   strToUpper( text = "sour");
   cout << text << endl;

   strToUpper( text += " makes funny!\n");
   cout << text << endl;

   return 0;
}

void strToUpper( string& str){    // converts str to upper case
    int len = str.length();

    for( int i=0; i < len; ++i)
      str[i] = toupper( str[i]);
}
```

Task 12.1: *What does change in above code if the argument of function* `strToUpper()` *is declared as string than string& ?*

Task 12.2: *Write the function* `circle()` *of type* `void` *for calculating the circle's circumference and area. The three function arguments are* `radius` *(read-only reference to* `double`*) and two result variables (references to* `double`*). Test the program with radius* 0.5, 1.0, 1.5, ..., 10.0

Source Code 2.55: swap() (Exercise: 12.7)

```
/*
// logically erroneous swap() version.
// What's wrong?
```

```
void swap(float *p1, float *p2)
{
  float *temp;         // helper variable

  temp = p1;
  p1   = p2;
  p2   = temp;
}
*/
```

Calling `swap(&x,&y)` does not swap the values of `x` ans `y`.

Task 12.3: *a) The above shown version of `swap()` can be compiled easily, but does not work correctly. What's going wrong? b) Test the corrected `swap()` function. Write and test a new version of `swap()` which is working with references instead pointers.*

The square equation

$$a * x^2 + b * x + c = 0 \tag{2.1}$$

provides the real solution

$$x_{1/2} = (-b \pm \sqrt{(b^2 - 4ac)})/2a \tag{2.2}$$

if $b^2 - 4ac >= 0$, if $b^2 - 4ac < 0$ then no real solution exists.

Task 12.4: *a) Construct a function `squareEQ()` for calculation the square equation. Arguments of the `squareEQ()` are parameters a,b,c and two pointers to the solutions. The function's return value is `false` if no real solution exists, otherwise `true`. Test the function for following equations and output the equation and it's solution to the console.*

$$\begin{aligned} 2x^2 - 2x - 1.5 &= 0 \\ x^2 - 6x + 9 &= 0 \\ 2x^2 + 2 &= 0 \end{aligned}$$

Submit your answers from Lab 2.2 to Moodle one week after the assignment

Lab 2.3: Basic Concepts OOP using c++

Object oriented programming – As the name suggests uses **objects** in programming. Object oriented programming aims to implement real world entities like *inheritance, hiding, polymorphism* etc in programming. The main aim of OOP is to bind together the data and the functions that operates on them so that no other part of code can access this data except that function.
Let us learn about different characteristics of an Object Oriented Programming language:

Object: Objects are basic run-time entities in an object oriented system, objects are instances of a class these are defined user defined data types.

Class: Class is a blueprint of data and functions or methods. Class does not take any space.

Encapsulation: Wrapping up(combing) of data and functions into a single unit is known as encapsulation. The data is not accessible to the outside world and only those functions which are wrapping in the class can access it. This insulation of the data from direct access by the program is called data hiding or information hiding.

Inheritance: inheritance is the process by which objects of one class acquire the properties of objects of another class. It supports the concept of hierarchical classification. Inheritance provides re usability. This means that we can add additional features to an existing class without modifying it.

Polymorphism: polymorphism means ability to take more than one form. An operation may exhibit different behaviors in different instances. The behavior depends upon the types of data used in the operation. C++ supports operator overloading and function overloading.

Dynamic Binding: In dynamic binding, the code to be executed in response to function call is decided at run time. C++ has virtual functions to support this.

Message Passing: Objects communicate with one another by sending and receiving information to each other. A message for an object is a request for execution of a procedure and therefore will invoke a function in the receiving object that generates the desired results. Message passing involves specifying the name of the object, the name of the function and the information to be sent.

Source Code 2.56: Class Person (Exercise: 13.1)

```
class person
{
        char name[20];
        int id;
public:
        void getdetails(){}
};

int main()
{
person p1; //p1 is a object
}
```

Source Code 2.57: Class Name (Exercise: 13.2)

```
class class_name
{
private:
        //data members and member functions declarations
public:
        //data members and member functions declarations
protected:
        //data members and member functions declarations
};
```

The header and source file of a more complex example is shown below:

Source Code 2.58: account.h (Exercise: 13.3)

```
/*
 ============================================================================
 Name        : account.h
 Author      : A. Pretschner
 Version     : Kirch & Prinz, 7. Edition
 Copyright   : Your copyright notice
 Description : Definition of class BankAccount
 ============================================================================
 */


#ifndef _ACCOUNT_   // prevents multiple inclusion.
#define _ACCOUNT_

#include <iostream>
#include <string>
using namespace std;
```

```
class BankAccount
{
  private:               // private Fields:
    string name;             // account owner
    unsigned long nr;        // number
    double balance;            // balance

  public:                // public Interface:
    bool init( const string&, unsigned long, double);
    void display();
};

#endif   //  _ACCOUNT_
```

Source Code 2.59: account.cpp (Exercise: 13.4)

```
/*
 ============================================================================
 Name        : account.cpp
 Author      : A. Pretschner
 Version     : Kirch & Prinz, 7. Edition
 Copyright   : Your copyright notice
 Description : method definitions of BankAccount init() and display()
 ============================================================================
 */

#include "account.h"         // class definition
#include <iomanip>
using namespace std;

// method  init() copies given arguments
// into private class fields
bool BankAccount::init(const string& i_name,
                 unsigned long i_nr,
                 double        i_balance)
{
   if( i_name.size() < 1)     // no empty name
     return false;

   name  = i_name;
   nr    = i_nr;
   balance = i_balance;

   return true;
}

// method display() shows private fields on screen
void BankAccount::display()
{
   cout << fixed << setprecision(2)
        << "--------------------------------------\n"
        << "Bank account owner  :" << name  << '\n'
        << "Bank account number :" << nr    << '\n'
        << "Bank Account balance:" << balance << '\n'
        << "--------------------------------------\n"
        << endl;
}
```

Pointer to objects

Each object of an class posses - as any other object - an address in memory. Those address can be linked to an pointer

Source Code 2.60: Pointer to Objects (Exercise: 13.5)
```
BankAccount debit("Bill, Claudia", 654321, 123.5);
BankAccount *ptrAccount = &debit;
```
The new pointer variable `ptrAccount` points to the object `debit`, thus `*ptrAccount` is the object by itself:

Source Code 2.61: Pointer to Objects (Exercise: 13.6)
```
(*ptrAccount).display();
```
Above source shows the access to the class method `display()`. Please note the brackets for `(*.ptrAccount)`, they are necessary because of higher priority of the point-operator than the *-operator.

Source Code 2.62: Pointer to Objects (Exercise: 13.7)
```
/*
 ============================================================================
 Name        : ptrObj.cpp
 Author      : A. Pretschner
 Version     : Kirch & Prinz, 7. Edition
 Copyright   : Your copyright notice
 Description : working with pointer of BankAccount objects
 ============================================================================
 */

#include "account.h"   // includes <iostream>, <string>

bool getAccount( BankAccount *pBankAccount);    // Prototype

int main(){
   BankAccount debit1, debit2, *ptr = &debit1;

   ptr->init("Flemming, Joy",        // debit1.init(...)
              3512345, 99.40);
   ptr->display();                  // debit1.display()

   ptr = &debit2;         // ptr points to debit2.

  if( getAccount( ptr))    // read new account data
      ptr->display();   //   show its data
   else
      cout << "Wrong Input!" << endl;

  return 0;
}

// --------------------------------------------------
// getAccount() reads new data via pointer from console into new account

bool getAccount( BankAccount *pBankAccount)
{
   string name, line(50,'-');
   unsigned long nr;
   double startBalance;

   cout << line << '\n'
        << "Input data for new Bank Account: \n"
        << "Owner: ";
   if( !getline(cin,name) || name.size() == 0)
     return false;

   cout << "Number:  ";
   if( !(cin >> nr))           return false;

   cout << "Start Balance: ";
   if( !(cin >> startBalance)) return false;

   // Input ok
   pBankAccount->init( name, nr, startBalance);

   return true;
}
```

Structs and Unions

In classical programming languages different data are linked together by wrapping them into `records`. In OOP a record is a class with only public data fields without having methods. Thus structures can be build with the keyword `class` too.

Source Code 2.63: Class and Struct (Exercise: 13.8)

```
class Date
  {public: short day, month, year;};
  ...
  ...
struct Date {short day, month, year;};
```

Each `class` can be defined with the key word `struct` too:

Source Code 2.64: Struct (Exercise: 13.9)

```
struct BankAccount {
  private: // ...
  public: // ...
  };
```

The difference between `class` and `struct` is their data encapsulation. For `struct` constructs the data access is defined as `public`.

Task 13.1: *A program needs for showing the date a class). a) Define for that reason the `class Date` with three class fields year, day and month and additionally construct the following functions: `void init (int day, int month, int year`, `void init (void)` and `void print (void)`. Construct the header file and the source code file accordingly. Method `print()` outputs the date in form `Day.Month.Year`, method `init()` with three arguments copies their values into the class fields, method `init()` without arguments writes the current data into the class fields. Test the result with one test application, in which you design two objects for showing the current date.*

Hint:

- Use declared functions from `ctime` like:
  ```
  time_t time(time_t *ptrSec);
  ```

```
    struct tm *localtime(const time_t *ptrSec);
```

- The through `ptrSec` addressed structure is defined in http://www.cplusplus.com/reference/ctime/tm/

Submit your answers from Lab 2.3 to Moodle one week after the assignment

Lab 2.4: Class Methods, this-Pointer and return values

Special member functions are member functions that are implicitly defined as member of classes under certain circumstances. There are six:

Member Function	**typical form for class c**
Default constructor	C::C();
Destructor	C:: C();
Copy constructor	C::C(const C&);
Copy assignment	C& operator= (const C&);
Move constructor	C::C(C&&);
Move assignment	C& operator= (C&&);

Default Constructor

The `default` constructor is the constructor called when objects of a class are declared, but are not initialized with any arguments. If a class definition has no constructors, the compiler assumes the class to have an implicitly defined default constructor. Therefore, after declaring a class like this:

```
class Example {
  public:
    int total;
    void accumulate (int x) { total += x; }
};
```

The compiler assumes that Example has a default constructor. Therefore, objects of this class can be constructed by simply declaring them without any arguments: `Example ex;`
But as soon as a class has some constructor taking any number of parameters explicitly declared, the compiler no longer provides an implicit default constructor, and no longer allows the declaration of new objects of that class without arguments. For example, the following class:

```
class Example2 {
  public:
    int total;
    Example2 (int initial_value) : total(initial_value) { };
    void accumulate (int x) { total += x; };
};
```

Here, we have declared a constructor with a parameter of type int. Therefore the following object declaration would be correct:

```
Example2 ex (100);    // ok: calls constructor
```

But the following:

```
Example2 ex;          // not valid: no default constructor
```

Would not be valid, since the class has been declared with an explicit constructor taking one argument and that replaces the implicit default constructor taking none. Therefore, if objects of this class need to be constructed without arguments, the proper default constructor shall also be declared in the class. For example:

```
// classes and default constructors
#include <iostream>
#include <string>
using namespace std;
```

```
class Example3 {
    string data;
  public:
    Example3 (const string& str) : data(str) {}
    Example3() {}
    const string& content() const {return data;}
};

int main () {
  Example3 foo;
  Example3 bar ("Example");

  cout << "bar's content: " << bar.content() << '\n';
  return 0;
}
```

Here, Example3 has a default constructor (i.e., a constructor without parameters) defined as an empty block:

```
Example3() {}
```

This allows objects of class Example3 to be constructed without arguments (like foo was declared in this example). Normally, a default constructor like this is implicitly defined for all classes that have no other constructors and thus no explicit definition is required. But in this case, Example3 has another constructor:

```
Example3 (const string& str);
```

And when any constructor is explicitly declared in a class, no implicit default constructors is automatically provided.

Destructor

Destructors fulfill the opposite functionality of constructors: They are responsible for the necessary cleanup needed by a class when its lifetime ends. The classes we have defined in previous chapters did not allocate any resource and thus did not really require any clean up.
But now, let's imagine that the class in the last example allocates dynamic memory to store the string it had as data member; in this case, it would be very useful to have a function called automatically at the end of the object's life in charge of releasing this memory. To do this, we use a *destructor*. A destructor is a member function very similar to a default constructor: it takes no arguments and returns nothing, not even void. It also uses the class name as its own name, but preceded with a tilde sign ( ):

```
// destructors
#include <iostream>
#include <string>
using namespace std;

class Example4 {
    string* ptr;
  public:
    // constructors:
    Example4() : ptr(new string) {}
    Example4 (const string& str) : ptr(new string(str)) {}
    // destructor:
    ~Example4 () {delete ptr;}
    // access content:
    const string& content() const {return *ptr;}
};
```

```
int main () {
  Example4 foo;
  Example4 bar ("Example");

  cout << "bar's content: " << bar.content() << '\n';
  return 0;
}
```

On construction, Example4 allocates storage for a string. Storage that is later released by the destructor. The destructor for an object is called at the end of its lifetime; in the case of foo and bar this happens at the end of function main.

Copy Constructor

When an object is passed a named object of its own type as argument, its *copy* constructor is invoked in order to construct a copy. A copy constructor is a constructor whose first parameter is of type reference to the class itself (possibly `const` qualified) and which can be invoked with a single argument of this type. For example, for a `class MyClass`, the copy constructor may have the following signature: `MyClass::MyClass (const MyClass&);`
If a class has no custom copy nor move constructors (or assignments) defined, an implicit copy constructor is provided. This copy constructor simply performs a copy of its own members. For example, for a class such as:

```
class MyClass {
  public:
    int a, b; string c;
};
```

An implicit copy constructor is automatically defined. The definition assumed for this function performs a shallow copy, roughly equivalent to:
`MyClass::MyClass(const MyClass& x) : a(x.a), b(x.b), c(x.c) {}`
This default copy constructor may suit the needs of many classes. But *shallow copies* only copy the members of the class themselves, and this is probably not what we expect for classes like class Example4 we defined above, because it contains pointers of which it handles its storage. For that class, performing a shallow copy means that the pointer **value is copied**, but **not the content** itself; This means that both objects (the copy and the original) would be sharing a single string object (they would both be pointing to the same object), and at some point (on destruction) both objects would try to delete the same block of memory, probably causing the program to crash on runtime. This can be solved by defining the following custom copy constructor that performs a deep copy:

```
// copy constructor: deep copy
#include <iostream>
#include <string>
using namespace std;

class Example5 {
    string* ptr;
  public:
    Example5 (const string& str) : ptr(new string(str)) {}
    ~Example5 () {delete ptr;}
    // copy constructor:
    Example5 (const Example5& x) : ptr(new string(x.content())) {}
    // access content:
    const string& content() const {return *ptr;}
};

int main () {
  Example5 foo ("Example");
  Example5 bar = foo;

  cout << "bar's content: " << bar.content() << '\n';
  return 0;
}
```

The *deep copy* performed by this copy constructor allocates storage for a new string, which is initialized to contain a copy of the original object. In this way, both objects (copy and original) have distinct copies of the content stored in different locations.

Copy assignment

Objects are not only copied on construction, when they are initialized: They can also be copied on any assignment operation. See the difference:

```
MyClass foo;
MyClass bar (foo);       // object initialization: copy constructor called
MyClass baz = foo;       // object initialization: copy constructor called
foo = bar;               // object already initialized: copy assignment called
```

Note that baz is initialized on construction using an *equal sign*, but this is not an assignment operation! (although it may look like one): The **declaration** of an object is **not an assignment** operation, it is just another of the syntaxes to call single-argument constructors.
The assignment on foo is an assignment operation. No object is being declared here, but an operation is being performed on an existing object; foo.
The copy assignment operator is an overload of operator= which takes a value or reference of the class itself as parameter. The return value is generally a reference to *this (although this is not required). For example, for a class MyClass, the copy assignment may have the following signature:

```
MyClass& operator= (const MyClass&);
```

The copy assignment operator is also a special function and is also defined implicitly if a class has no custom copy nor move assignments (nor move constructor) defined.
But again, the implicit version performs a shallow copy which is suitable for many classes, but not for classes with pointers to objects they handle its storage, as is the case in Example5. In this case, not only the class incurs the risk of deleting the pointed object twice, but the assignment creates memory leaks by not deleting the object pointed by the object before the assignment. These issues could be solved with a copy assignment that deletes the previous object and performs a deep copy:

```
Example5& operator= (const Example5& x) {
  delete ptr;                      // delete currently pointed string
  ptr = new string (x.content());  // allocate space for new string, and copy
  return *this;
}
```

Or even better, since its string member is not constant, it could re-utilize the same string object:

```
Example5& operator= (const Example5& x) {
  *ptr = x.content();
  return *this;
}
```

Move constructor and assignment

Similar to copying, moving also uses the value of an object to set the value to another object. But, unlike copying, the content is actually transferred from one object (the source) to the other (the destination): the source loses that content, which is taken over by the destination. This moving only happens when the source of the value is an *unnamed object*.

Unnamed objects are objects that are temporary in nature, and thus haven't even been given a name. Typical examples of unnamed objects are return values of functions or type-casts.
Using the value of a temporary object such as these to initialize another object or to assign its value, does not really require a copy: the object is never going to be used for anything else, and thus, its value can be moved into the destination object. These cases trigger the move constructor and move assignments:
The move constructor is called when an object is initialized on construction using an unnamed temporary. Likewise, the move assignment is called when an object is assigned the value of an unnamed temporary:

```
MyClass fn();            // function returning a MyClass object
MyClass foo;             // default constructor
MyClass bar = foo;       // copy constructor
MyClass baz = fn();      // move constructor
foo = bar;               // copy assignment
baz = MyClass();         // move assignment
```

Both the value returned by `fn` and the value constructed with MyClass are unnamed temporaries. In these cases, there is no need to make a copy, because the unnamed object is very short-lived and can be acquired by the other object when this is a more efficient operation.
The move constructor and move assignment are members that take a parameter of type *rvalue reference* to the class itself:

```
MyClass (MyClass&&);             // move-constructor
MyClass& operator= (MyClass&&);  // move-assignment
```

An *rvalue reference* is specified by following the type with two ampersands (&&). As a parameter, an *rvalue reference* matches arguments of temporaries of this type.
The concept of moving is most useful for objects that manage the storage they use, such as objects that allocate storage with new and delete. In such objects, copying and moving are really different operations:

- Copying from A to B means that new memory is allocated to B and then the entire content of A is copied to this new memory allocated for B.
- Moving from A to B means that the memory already allocated to A is transferred to B without allocating any new storage. It involves simply copying the pointer.

Source Code 2.65: Struct (Exercise: 14.1)

```
/*
============================================================================
 Name        : example6.cpp
 Author      : A. Pretschner
 Version     :
 Copyright   :
 Description : Move Constructor / Assignment
 ============================================================================
 */

#include <iostream>
#include <string>
using namespace std;

class Example6 {
    string* ptr;
  public:
    Example6 (const string& str) : ptr(new string(str)) {}
    ~Example6 () {delete ptr;}

    // move constructor
    Example6 (Example6&& x) : ptr(x.ptr) {x.ptr=nullptr;}

    // move assignment
    Example6& operator= (Example6&& x) {
      delete ptr;
      ptr = x.ptr;
      x.ptr=nullptr;
      return *this;
    }

    // access content:
    const string& content() const {return *ptr;}

    // addition:
    Example6 operator+(const Example6& rhs) {
      return Example6(content()+rhs.content());
    }
};


int main () {
  Example6 foo ("Exam");
  Example6 bar = Example6("ple");   // move-construction

  foo = foo + bar;                  // move-assignment

  cout << "foo's content: " << foo.content() << '\n';
  return 0;
}
```

Compilers already optimize many cases that formally require a move-construction call in what is known as Return Value Optimization. Most notably, when the value returned by a function is used to initialize an object. In these cases, the move constructor may actually never get called. Note that even though *rvalue references* can be used for the type of any function parameter, it is seldom useful for uses other than the move constructor. *Rvalue references* are tricky, and unnecessary uses may be the source of errors quite difficult to track.

Task 14.1: *A warehouse management program needs the `class Article`. a) Define for that reason the `class Article` with the below given fields and methods. The constructor is called with the default values of the data fields. Thus, the default constructor is declared too. Only the access methods for the class fields are defined inline, a negative price is not allowed instead 0.0 should be stored. b) Implement the constructor and destructor in a separate source file, define additionally a global variable for the article count, which will be incremented by the constructor and decremented by the destructor. `print()` outputs the formatted objects. Test the program by defining of 4 articles of your choice. Create a `test()` function for showing the objects. c) Extend the program in such way that the `test()` function will be called with one argument of type `Article`. d) Explain, why the object counter after the program exit is negative?*

Hint:

- The `class Article` consists of:

```
private fields:
  article number of type long
  article name of type string
  selling price of type double
```

```
      public methods:
        Article (long, const string&, double);
        ~Article();
        void print();
        set- and get- methods for each field
```

- The output by call to the constructor:

```
A object for article ... was created.
It's the ....th article.
```

- The output by call to the destructor:

```
The object for article ... was destroyed.
In stock are still ... article
```

Task 14.2: *In Structs and Unions the `class Date` with the fields `day, month, year` was created. Extend that example with following methods:*

- *The constructor and the `setDate()` method replace the former `init()` method. The default constructor initializes the new object among others with the default value* 1.1.1. *The `setDate()` without arguments initializes the object with the current date.*
- *The constructor and `setDate()` method with three arguments initializes a new object with those parameters.*
- *Methods `isEqual()` and `isLess()` allow the comparison with the given date.*
- *Method `asString()` returns a reference to a formatted string (`tt.mm.yyyy`). For that purpose the numerical values must be transformed into decimal numbers, usually during the output operator << usage. In analogy to `cin` and `cout` exist so called `String-streams` with similar functionality.*
- *Test the application in which all methods are used*

Hint: *Stream class* to operate on strings. Objects of this class use a *string buffer* that contains a sequence of characters. This sequence of characters can be accessed directly as a string object, using member `str` (`http://www.cplusplus.com/reference/sstream/stringstream/`).

Task 14.3: *The `class Date` does not yet check the date validity. Extend the above program so that the constructor and the `setDate()` method do range checking.*

- *First, write a function `isLeapYear()` of type `bool` checking if the given date is a leap year. This function shall be defined as a global inline function in the header file.*
- *Extend `setDate()` to carry out range validation for the given date. The constructor can call `setDate()`.*
- *Test the new Version on the console.*

Lab 2.5: Polymorphism

Pointers to base class

One of the key features of class inheritance is that a pointer to a derived class is type-compatible with a pointer to its base class. *Polymorphism* is the art of taking advantage of this simple but powerful and versatile feature.
The example about the rectangle and triangle classes can be rewritten using pointers taking this feature into account:

```
// pointers to base class
#include <iostream>
```

```
using namespace std;

class Polygon {
  protected:
    int width, height;
  public:
    void set_values (int a, int b)
      { width=a; height=b; }
};

class Rectangle: public Polygon {
  public:
    int area()
      { return width*height; }
};

class Triangle: public Polygon {
  public:
    int area()
      { return width*height/2; }
};

int main () {
  Rectangle rect;
  Triangle trgl;
  Polygon * ppoly1 = &rect;
  Polygon * ppoly2 = &trgl;
  ppoly1->set_values (4,5);
  ppoly2->set_values (4,5);
  cout << rect.area() << '\n';
  cout << trgl.area() << '\n';
  return 0;
}
```

Function main declares two pointers to Polygon (named `ppoly1` and `ppoly2`). These are assigned the addresses of `rect` and `trgl`, respectively, which are objects of type Rectangle and Triangle. Such assignments are valid, since both Rectangle and Triangle are classes derived from Polygon.

Dereferencing `ppoly1` and `ppoly2` (with `*ppoly1` and `*ppoly2`) is valid and allows us to access the members of their pointed objects. For example, the following two statements would be equivalent in the previous example:

```
ppoly1->set_values (4,5);
rect.set_values (4,5);
```

But because the type of `ppoly1` and `ppoly2` is pointer to Polygon (and not pointer to Rectangle nor pointer to Triangle), only the members inherited from Polygon can be accessed, and not those of the derived classes Rectangle and Triangle. That is why the program above accesses the area members of both objects using `rect` and `trgl` directly, instead of the pointers; the pointers to the base class cannot access the area members.

Member area could have been accessed with the pointers to Polygon if area were a member of Polygon instead of a member of its derived classes, but the problem is that Rectangle and Triangle implement different versions of area, therefore there is not a single common version that could be implemented in the base class.

virtual members

A virtual member is a member function that can be redefined in a derived class, while preserving its calling properties through references. The syntax for a function to become virtual is to precede its declaration with the virtual keyword:

```
// virtual members
#include <iostream>
using namespace std;

class Polygon {
  protected:
    int width, height;
  public:
    void set_values (int a, int b)
      { width=a; height=b; }
    virtual int area ()
      { return 0; }
};

class Rectangle: public Polygon {
  public:
    int area ()
      { return width * height; }
};

class Triangle: public Polygon {
  public:
    int area ()
      { return (width * height / 2); }
};

int main () {
  Rectangle rect;
  Triangle trgl;
  Polygon poly;
  Polygon * ppoly1 = &rect;
  Polygon * ppoly2 = &trgl;
  Polygon * ppoly3 = &poly;
  ppoly1->set_values (4,5);
  ppoly2->set_values (4,5);
  ppoly3->set_values (4,5);
  cout << ppoly1->area() << '\n';
  cout << ppoly2->area() << '\n';
  cout << ppoly3->area() << '\n';
  return 0;
}
```

In this example, all three classes (Polygon, Rectangle and Triangle) have the same members: width, height, and functions `set_values` and `area`.
The member function area has been declared as **virtual** in the base class because it is later redefined in each of the derived classes. Non-virtual members can also be redefined in derived classes, but *non-virtual members of derived classes cannot be accessed through a reference of the base class*: i.e., if virtual is removed from the declaration of area in the example above, all three calls to area would return zero, because in all cases, the version of the base class would have been called instead.
Therefore, essentially, what the virtual keyword does is to allow a member of a derived class with the same name as one in the base class to be **appropriately called from a pointer**, and

more precisely when the type of the pointer is a **pointer to the base class** that is pointing to an object of the derived class, as in the above example.
A class that declares or inherits a virtual function is called a **polymorphic class**.
Note that despite of the virtuality of one of its members, Polygon was a regular class, of which even an object was instantiated (poly), with its own definition of member area that always returns 0.

abstract base classes

Abstract base classes are something very similar to the Polygon class in the previous example. They are classes that can only be used as base classes, and thus are allowed to have virtual member functions without definition (known as pure virtual functions). The syntax is to replace their definition by =0 (an equal sign and a zero):
An abstract base Polygon class could look like this:

```
// abstract class CPolygon
class Polygon {
  protected:
    int width, height;
  public:
    void set_values (int a, int b)
      { width=a; height=b; }
    virtual int area () =0;
};
```

Look at the following more complex Example "Vehicle Management": the `vehicle.h`, `vehicle.cpp` and `vehicle_t.cpp` with the produced output:

```
After pointer Down-Cast:

----------------------------
Vehicle-Number:   3265
Producer:   BMW
Type:          520i
Sun Roof:  yes

After reference Down-Cast:

----------------------------
Vehicle-Number:   9154
Producer:   Mercedes Benz
Type:          190 SL
Sun Roof:  yes
```

Source Code 2.66: Struct (Exercise: 15.1)

```
/*
============================================================================
 Name        : vehicle.h
 Author      : A. Pretschner
 Version     :
 Copyright   :
 Description : defines base class  Vehicle and derivated classes Car and Truck
 ============================================================================
 */

#ifndef _VEHICLE_H_
#define _VEHICLE_H_
```

```
#include <iostream>
#include <string>
using namespace std;

class Vehicle{
  private:
     long   nbr;
     string producer;

  public:
     Vehicle( long n = 0L, const string& prod = "");
     virtual ~Vehicle() {}

     // access methods:
     long  getNbr(void) const { return nbr; }
     void  setNbr( long n ) { nbr = n; }

     const string& getPeoducer() const { return producer; }
     void  setProducer(const string& h){ producer = h; }

     virtual void display() const;   // show vehicle
};

class Car : public Vehicle {
   private:
     string carType;
     bool   sunroof;

   public:
     Car(const string& tp, bool sd,
         int n = 0 , const string& h = "");

     // access methods:
     const string& getTyp( void ) const { return carType; }
     void  setTyp( const string s ) { carType = s; }

     bool  getSunRoof( void ) const { return sunroof; }
     void  setSunRoof( bool d ) { sunroof = d; }

     void  display( void ) const;
};

class Truck : public Vehicle {
  private:
     int    axles;
     double load;

  public:
     Truck( int a, double t, int n, const string& hs);

     void   setAxles( int l){ axles = l;}
     int    getAxles() const   { return axles; }
     void   setCapacity( double t) { load = t;}
     double getCapacity() const    { return load; }

     void display() const;
};
#endif
```

Source Code 2.67: Struct (Exercise: 15.2)

```
/*
============================================================================
 Name        : vehicle.cpp
 Author      : A. Pretschner
 Version     :
 Copyright   :
 Description : main() function for Vehicle application
 ============================================================================
 */

#include "vehicle.h"

// Methods of the Vehicle base class :
// ------------------------------------------------------
Vehicle::Vehicle( long n, const string& prod)
    : nbr(n), producer(prod){}

void Vehicle::display() const {
    cout << "\n---------------------------- "
         << "\nVehicle-Number:   " << nbr
         << "\nProducer:   " << producer
         << endl;
}

// Methods of derived Car class:
// -------------------------------------------------------
Car::Car(const string& tp, bool sd, int n, const string& hs)
    : Vehicle( n, hs), carType( tp ), sunroof( sd ){}

void Car::display( void) const {
   Vehicle::display();            // base class method

   cout <<   "Type:          " << carType
        << "\nSun Roof:  ";
   if(sunroof)
       cout << "yes "<< endl;
   else
       cout << "no " << endl;
}

// Methods of derived Truck class:
// ---------------------------------------------------
Truck::Truck( int a, double t, int n, const string& hs)
        : Vehicle( n, hs), axles( a ), load( t ){}

void Truck::display() const {
    Vehicle::display();

    cout <<   "Axles:       " << axles
         << "\nCapacity:   " << load << " tons"
         << endl;
}
```

Source Code 2.68: Struct (Exercise: 15.3)

```
/*
============================================================================
 Name        : vehicle_t.cpp
 Author      : A. Pretschner
 Version     :
 Copyright   :
 Description : main() tests for dynamic casts
 ============================================================================
 */


#include "vehicle.h"

bool inspect( Car* ),           // Inspection of different
     inspect(Truck* );            // vehicle types.

bool distribute(Vehicle* vehiclePtr);   // distribute vehicle for inspection

int main() {
    Vehicle* vehiclePtr = new Car("520i", true, 3265, "BMW");

    Car* carPtr = dynamic_cast<Car*>(vehiclePtr);
    if( carPtr != NULL) {                                // ok?
            cout << "\nAfter pointer Down-Cast: " << endl;
            carPtr->display();
    }

    Car cabrio("190 SL", true, 9154, "Mercedes Benz");
    Vehicle& r_vehicle = cabrio;

    Car& r_car = dynamic_cast<Car&>(r_vehicle);
    // ok? throws exception if not ok.

    cout << "\nAfter reference Down-Cast: " << endl;
    r_vehicle.display();
    cin.get();

    Truck* truckPtr = new Truck(8, 7.5, 5437, "Volvo");
    distribute(vehiclePtr);
    distribute(truckPtr);

    return 0;
}

bool distribute( Vehicle* vehiclePtr) {
   Car* carPtr = dynamic_cast<Car*>(vehiclePtr);
   if( carPtr != NULL)
      return inspect( carPtr);

   Truck* truckPtr = dynamic_cast<Truck*>(vehiclePtr);
   if( truckPtr != NULL)
      return inspect( truckPtr);

   return false;
}
```

```
bool inspect(Car* carPtr){
    cout << "\nChecking car!" << endl;
    cout << "\nHere it comes:";
    carPtr->display();
    return true;
}

bool inspect(Truck* truckPtr){
    cout << "\nChecking truck!" << endl;
    cout << "\nHere it comes:";
    truckPtr->display();
    return true;
}
```

Hints:

- *Destructors* - Deleting a derived class object using a pointer to a base class that has a non-virtual destructor results in undefined behavior. To correct this situation, the base class should be defined with a virtual destructor (`~Vecicle`). Making base class destructor virtual guarantees that the object of derived class is destructed properly, i.e., both base class and derived class destructors are called.
- *Early binding and Late binding* - Binding refers to the process of converting identifiers (such as variable and performance names) into addresses. Binding is done for each variable and functions. For functions, it means that matching the call with the right function definition by the compiler. It takes place either at compile time or at runtime. *Early Binding* (**compile-time time polymorphism**) As the name indicates, compiler (or linker) directly associate an address to the function call. It replaces the call with a machine language instruction that tells the mainframe to leap to the address of the function. In *Late Binding* (**run time polymorphism**) the compiler adds code that identifies the kind of object at runtime then matches the call with the right function definition. This can be achieved by declaring a **virtual function**.
- **VMT Virtual Method Table** realizes the late binding process. For each class with at least one virtual method a VMT (vector with the virtual method addresses for each class) will be created. Each object of an polymorphic class owns a pointer to the VMT. The call to an virtual method happens in two steps: a) read the VMT pointer, b) find the appropriate address of the virtual method.
- *Down-Casts* - of objects in class hierarchies are not safe (`static_cast<>();`), if the referenced object does not have the type of the derived class (runtime errors). In polymorphic classes safe Down-Casts are possible with `dynamic_cast<type>(statment)` because of the conversation check at runtime.

Task 15.1: *Extend the above "Vehicle management" program in such way, that the car park of an rental company can be managed. First `class CityCar` shall be defined.*

- *Define `class CityCar` as vector of pointer to the `class Vehicle` and one `int` Variable for the vector length. The constructor initializes the vector with length 0. The destructor shall be able to free all dynamically reserved memory (virtual destructor!)*
- *The method `insert()` has to be implemented in two versions (see programming hints below!) The method returns false, when no new object can be submitted (vector is full).*
- *Create new source file for `menu()` (see below!)*
- *Write getter functions for cars and trucks.*
- *Create object `CityCar`in the `main()` application, afterwards insert each one of a car and a truck.*

Hints:

```
 //insert new car
  bool insert (const strin& tp, bool sd, long n, const string& prod);
 // insert new truck
  bool insert (int a, double t, long n, const string& prod)
****Car-Park-Management-System**********
C = add new car
T = add new truck
S = show car park
E = exit program

Your choice:
```

Task 15.2: *Program an automated register cash system.*

- *Declare the the methods `scanner()` and `printer()` in the base class `class Product` as virtual. Define an virtual destructor too.*
- *Write the function `register()`, which in loop allows the registering and listing of shopping products in an Super Market. In that function a vector with 100 pointers to base class Product shall be initialized. The cashier answers in dialog if an wrapped or unwrapped product will be scanned. The memory of each scanned product shall be assigned dynamically and to be linked to the next pointer in vector. After the scanning process all products are shown sequentially, the prices are added and the total price shows up.*
- *Program the `main()` application for that register cash system. In a loop the cashier shall be asked if more customers are available, otherwise the program is finished.*

Hints: inside `register()`

```
What is the next Article?
  0 = none
  1 = un-wrapped Article
  2 = wrapped Article
  ?
```

inside `main()`

```
Next customer (y/n) ?
If yes -> register
```

Submit your answers from Lab 2.5 to Moodle one week after the assignment

Lab 2.6: Abstract Classes

Defines an abstract type which cannot be instantiated, but can be used as a base class. An interface describes the behavior or capabilities of a C++ class without committing to a particular implementation of that class. The C++ interfaces are implemented using abstract classes and these abstract classes should not be confused with data abstraction which is a concept of keeping implementation details separate from associated data.

Syntax

A class is made abstract by declaring at least one of its functions as *pure virtual function*. A pure virtual function is specified by placing **"= 0"** in its declaration as follows

```
class Box {
   public:
      // pure virtual function
      virtual double getVolume() = 0;

   private:
      double length;      // Length of a box
      double breadth;     // Breadth of a box
      double height;      // Height of a box
};
```

The purpose of an **abstract class** (often referred to as an *ABC*) is to provide an appropriate base class from which other classes can inherit. Abstract classes cannot be used to instantiate objects and serves only as an interface. Attempting to instantiate an object of an abstract class causes a compilation error.
Thus, if a subclass of an ABC needs to be instantiated, it has to implement each of the virtual functions, which means that it supports the interface declared by the ABC. Failure to override a pure virtual function in a derived class, then attempting to instantiate objects of that class, is a compilation error.
Classes that can be used to instantiate objects are called **concrete classes**.
An object-oriented system might use an abstract base class to provide a common and standardized interface appropriate for all the external applications. Then, through inheritance from that abstract base class, derived classes are formed that operate similarly.
The capabilities (i.e., the public functions) offered by the external applications are provided as pure virtual functions in the abstract base class. The implementations of these pure virtual functions are provided in the derived classes that correspond to the specific types of the application.
This architecture also allows new applications to be added to a system easily, even after the system has been defined.

Abstract Class Examples

Consider the following example where parent class provides an interface to the base class to implement a function called `getArea()`:

Source Code 2.69: Abstract Class (Exercise: 16.1)

```
/*
============================================================================
 Name        : shape_t.cpp
 Author      : A. Pretschner
 Version     :
 Copyright   :
 Description : main() shows abstract class usage
 ============================================================================
 */


#include <iostream>

using namespace std;

// Base class
class Shape {
   public:
      virtual ~Shape() {}          //virtual destructor
      virtual int getArea() = 0;   // pure virtual function providing interface framework.
      void setWidth(int w) {
         width = w;
      }

      void setHeight(int h) {
         height = h;
      }

   protected:
      int width;
      int height;
};

// Derived classes
class Rectangle: public Shape {
   public:
      int getArea() {
         return (width * height);
      }
};

class Triangle: public Shape {
   public:
      int getArea() {
         return (width * height)/2;
      }
};

int main(void) {
   Rectangle Rect;
   Triangle  Tri;

   Rect.setWidth(5);
   Rect.setHeight(7);

   // Print the area of the object.
   cout << "Total Rectangle area: " << Rect.getArea() << endl;

   Tri.setWidth(5);
   Tri.setHeight(7);

   // Print the area of the object.
   cout << "Total Triangle area: " << Tri.getArea() << endl;

   return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
Total Rectangle area: 35
Total Triangle area: 17
```

You can see how an abstract class defined an interface in terms of `getArea()` and two other classes implemented same function but with different algorithm to calculate the area specific to the shape.

Below, a more complex example dealing with the `class Employee`. This abstract class does not implement the income implementation for different kind of employees, this is provided in each derived class separately. Consider the derived `class Worker : public Employee` with defined function `double income() static;`

Source Code 2.70: More Complex Abstract Class (Exercise: 16.2)

```
/*
============================================================================
 Name        : employee.h
 Author      : A. Pretschner
 Version     :
 Copyright   :
 Description : defines base class  Employee and their descendants
 ============================================================================
 */

#ifndef _EMPLOYEE_H
#define _EMPLOYEE_H

#include <string>
#include <iostream>
#include <iomanip>
using namespace std;

class Employee{
   private:
     string name;
     // more details
   public:
     Employee( const string& s = ""){ name = s; }
     virtual ~Employee() {};         // Destructor

     const string&  getName() const{ return name; }
     void  setName( const string& n){ name = n; }

     virtual void display() const;

     virtual double income() const = 0;

     virtual Employee& operator=(const Employee&);
};

class Worker : public Employee{
   private:
     double wage;
     int    hours;

   public:
    Worker(const string& s="", double w = 0.0, int h = 0)
             : Employee(s), wage(w), hours(h){ }

    double getWage() const { return wage; }
    void   setWage( double w ){ wage = w; }

    int    getHours() const { return hours; }
    void   setHours(int h ) { hours = h; }

    void   display() const;
    double income() const { return wage * hours; }

    Employee& operator=(const Employee&);
    Worker& operator=(const Worker&);
};

class Staff : public Employee{
  private:
    double salary;       // monthly salary

  public:
    Staff( const string& s="", double val = 0.0)
                : Employee(s), salary(val){ }

    double getSalary() const { return salary; }
    void   setSalary( double val){ salary = val; }

    void   display() const;

    double income()const{ return salary; }

    Employee& operator=( const Employee& );
    Staff& operator=( const Staff& );
};

#endif
```

Source Code 2.71: More Complex Abstract Class (Exercise: 16.3)

```
/*
============================================================================
 Name        : employee.cpp
 Author      : A. Pretschner
 Version     :
 Copyright   :
 Description : defines methods of base class  Employee and their descendants
 ============================================================================
 */

#include "employee.h"

Employee& Employee::operator=(const Employee & e){
   if( this != &e )         // no self assignment
           name = e.name;

   return *this;
}

void Employee::display() const{
    cout << "\n----------------------------"
         << "\nName:           " << name;
}

// Redefinition, virtual
Employee& Staff::operator=(const Employee& e)
{
   if( this != &e ){         // no self assignment
     Employee::operator=( e );
     salary = 0.0;
   }

   return *this;
}

// Standard-Zuweisung, nicht virtuell
```

```
Staff& Staff::operator=(const Staff& s)
{
  if( this != &s )   {
      Employee::operator=( s );
      salary = s.salary;
  }
  return *this;
}

void Staff::display() const
{
   Employee::display();
   cout << "\nSalary:          "
        << fixed << setprecision(2) << salary << endl;
}

// Redefinition, virtual
Employee& Worker::operator=(const Employee& e){
   if( this != &e ) {          // no self assignment
     Employee::operator=( e );
     wage = 0.0;
     hours  = 0;
   }

   return *this;
}

// Standard-Assignment, none virtual
Worker& Worker::operator=(const Worker& w) {
  if( this != &w ) {
      Employee::operator=( w );
      wage = w.wage;
      hours  = w.hours;
  }
  return *this;
}

void Worker::display() const
{
   Employee::display();
   cout << fixed << setprecision(2)
        << "\nHourly wage:    " << wage
        << "\nHours:  " << hours << endl;
}
```

Hint:

- Operator function methods can be defined virtually (see `class Employee`). In consequence always the correct method will be executed if an object pointer to a derived class or a reference to the base class is used.
- Redefinition for the standard assignment means the standard assignment uses the same signature in each derived class (see `Employee&  operator=(const Employee&)`).
- Additionally the standard assignment for each descendant must be declared.

To test the example you can use following test application:

Source Code 2.72: More Complex Abstract Class (Exercise: 16.4)

```
/*
============================================================================
 Name        : employee_t.cpp
 Author      : A. Pretschner
 Version     :
 Copyright   :
 Description : main() tests Employee and their descendant classes
 ============================================================================
 */

#include "employee.h"

// Demo-Function:
void copy( Employee& a, const Employee& b );

int main(){

    Staff doorman("Bright, Paul", 2350.0);
    doorman.display();
    cout << "\nStaff income: "
         << doorman.income() << endl;

    Staff substitude;
    copy(substitude, doorman);
    substitude.display();

    substitude = doorman;      // none virtual assignment
    cout << "\nAfter assignment: ";
    substitude.display();
    cin.get();

    Worker mason("Miller, Lewis", 53.50, 40);
    mason.display();
    cout << "\nWorker income:   "
         << mason.income() << endl;

    Worker apprentice;
    copy(apprentice, mason);
    cout << "\nApprentice income after copy: "
         <<  apprentice.income()<< endl;

    apprentice.setWage(12.0);
    apprentice.setHours(20);
    mason = apprentice;         // none virtual assignment

    cout << "\nMason income after assignment: "
         << mason.income();
    mason.display();

    cout << "\nNow with dynamically allocated memory: ";
    Employee* ptrE1;
        ptrE1 = new Worker("Nell, Rudi",45., 40);
    ptrE1->display();
        cout << "\nIncome:   " << ptrE1->income() << endl;

    Employee* ptrE[2];
    ptrE[0] = new Worker("Nell, Rudi",45., 40);
    ptrE[1] = new Staff("Summer, Eve", 3850.0);
    for( int i = 0; i < 2; ++i)  {
            ptrE[i]->display();
      cout << "\nIncome of " << ptrE[i]->getName() << " :  "
           << ptrE[i]->income() << endl;
    }
    return 0;
}

void copy(Employee& a,const Employee& b) {
    a = b;         // call virtual assignment
}
```

Hint:

- Method `copy()` introduces the virtual object assignment. It requires the standard assignment definition of each derived class.

Task 16.1: *Extend the following none homogeneous List and implement all missing functions.*

- *Define the destructor of `class NonehomogeneousList` which releases the memory space of each list item.*
- *Implement the missing methods `getPrev()`, `insert()` and `insertAfter()` (see hints below)*
- *Implement the methods `displayAll()`, which runs sequentially through the list and outputs each object.*
- *Test the function of the methods, check if object's comments (if existent) are shown too.*
- *Create method `getPos()` which determines the position of an list item that shall be deleted. If the item exists address is the return value of the object otherwise NULL is returned.*
- *Write the method `erasePos()` which deletes one item at a specific position. Make a distinction whether the item is the first list element or not. Because of the virtual `class cell` destructor, only one version of `erasePos()` is necessary.*
- *Create method `erase()` which deletes for a given name the appropriate list item.*
- *Test the different erase functions and output always the remaining list elements.*
- *Implement a copy constructor for the standard assignment. Call `insert()` for building the list in their properly version. The operator `typeid()` delivers the list item's object type, which is defined in the header file `typeinfo`. `if (typeid(*ptr) == typeid(DerivedItem)) ....` It's true, if `*ptr` is of type `class DerivedItem`.*
- *Test the copy constructor and the assignment.*

Source Code 2.73: Cell Class for the list example (Exercise: 16.5)

```
/*
==============================================================================
 Name        : cell.h
 Author      : A. Pretschner
 Version     :
 Copyright   :
 Description : defines base class  Cell and derived BaseEl, DerivedEl
 ==============================================================================
 */

#ifndef _CELL_
#define _CELL_

#include <cstdio>              // Definition of NULL
```

```
#include <string>
#include <iostream>
using namespace std;

class Cell{
  private:
   Cell* next;                          //pointer to the next list element

  protected:
   Cell(Cell* suc = NULL ){ next = suc; }

  public:
   virtual ~Cell(){ }
   Cell* getNext() const { return next; }
   void  setNext(Cell* suc) { next = suc; }

   virtual void display() const = 0;
};

class BaseEl : public Cell{
  private:
   string name;

  public:
   BaseItem( Cell* suc = NULL, const string& s = "")
       :  Cell(suc), name(s){}

   // access to cell data:
   void    setName(const string& s){ name = s; }
   const string& getName() const { return name; }


   void display() const {
       cout << "\n----------------------------"
            << "\nName:      " << name << endl;
   }
};

class DerivedItem : public BaseItem{
   private:
     string comment;

   public:
     DerivedItem(Cell* suc = NULL,  const string& s="", const string& b="")
         : BaseItem(suc, s), comment(b){ }
     // access to cell notes:
     void    setComment(const string& b){ comment = b; }
     const string& getComment() const { return comment; }

     void display() const   {
         BaseItem::display();
         cout << "Comments: " << comment << endl;
     }
};
#endif
```

Hints:

- The List consists of various elements (cells), which are linked by the `next` pointer to the successor cell. `class BaseItem` and `DerivedItem` are derived from `class Cell` and form the data container (here a simple string). `class DerivedItem` contains additionally a comment field for the list element.
- Because no `cell` instances shall be instantiated directly, the constructor of this class is declared `protected`.
- You can build list objects with `class BaseItem` and/ or `DerivedItem` elements.

Source Code 2.74: Class NonehomogeneousList (Exercise: 16.6)

```
/*
============================================================================
 Name        : list.h
 Author      : A. Pretschner
 Version     :
 Copyright   :
 Description : defines base class  NonehomogeneousList
 ============================================================================
 */

#ifndef _LIST_H
#define _LIST_H

#include "cell.h"

class NonehomogeneousList
{
  private:
    Cell* first;

  protected:
    Cell* getPrev(const string& s);
    void  insertAfter(const string& s, Cell* prev);
    void  insertAfter(const string& s,const string& b, Cell* prev);

  public:
    NonehomogeneousList(){ first = NULL; }
    NonehomogeneousList(const NonehomogeneousList& src);
    ~NonehomogeneousList();

     NonehomogeneousList& operator=( const NonehomogeneousList& src);

     void  insert(const string& n);
     void  insert(const string& n, const string& b);

     void displayAll() const;
};

#endif
}
```

Source Code 2.75: Test Application for the NonehomogeneousList Class (Exercise: 16.7)

```
/*
============================================================================
 Name        : list_t.cpp
 Author      : A. Pretschner
 Version      :
 Copyright    :
 Description : main() tests Cell and their descendant classes
 ============================================================================
 */


#include "list.h"

int main(){
        NonehomogeneousList list1;

   list1.insert("Rambo, Max");
   list1.insert("Trump, Donald", "alwyas lying");
   list1.insert("Obama, Barack", "topfit");
   list1.insert("Banderas, Antonio");
   cout << "\n1. List:  " << endl;
   list1.displayAll();
   cin.get();

   NonehomogeneousList list2(list1),               // Copy-Constructor
             list3;                               // empty list.

   cout << "\n------------------------------"
        << "\nItem copied!" << endl;

   list2.insert("Funny, Peter", "in bad mood");

   list3 = list2;              // Assignment

   cout << "\nAfter Assignment:\n"
           "\n3. List:  " << endl;
   list3.displayAll();

   return 0;
}
```

Hint:

- Above source code is because of the missing method implementations not running yet!
- Create `list.cpp` source file and implement the missing functions defined in the header file!

Submit your answers from Lab 2.6 to Moodle one week after the assignment

Lab 2.7: Exception Handling

An exception is a problem that arises during the execution of a program. A C++ exception is a response to an exceptional circumstance that arises while a program is running, such as an attempt to divide by zero. Exceptions provide a way to transfer control from one part of a program to another. C++ exception handling is built upon three keywords: `try`, `catch`, and `throw`.

```
try {
   // protected code
} catch( ExceptionName e1 ) {
   // catch block
} catch( ExceptionName e2 ) {
   // catch block
} catch( ExceptionName eN ) {
   // catch block
}
```

Exception Syntax

throw A program throws an exception when a problem shows up. This is done using a throw keyword. Exceptions can be thrown anywhere within a code block using throw statement. The operand of the throw statement determines a type for the exception and can be any expression and the type of the result of the expression determines the type of exception thrown. Following is an example of throwing an exception when dividing by zero condition occurs

```
double division(int a, int b) {
   if( b == 0 ) {
      throw "Division by zero condition!";
   }
   return (a/b);
```

catch A program catches an exception with an exception handler at the place in a program where you want to handle the problem. The catch keyword indicates the catching of an exception.

try A try block identifies a block of code for which particular exceptions will be activated. It's followed by one or more catch blocks. The catch block following the try block catches any exception. You can specify what type of exception you want to catch and this is determined by the exception declaration that appears in parentheses following the keyword catch.

```
try {
   // protected code
} catch( ExceptionName e ) {
  // code to handle ExceptionName exception
```

Above code will catch an exception of `ExceptionName` type. If you want to specify that a catch block should handle any type of exception that is thrown in a try block, you must put an ellipsis, ..., between the parentheses enclosing the exception declaration as follows

```
try {
   // protected code
} catch(...) {
  // code to handle any exception
```

C++ provides a list of standard exceptions defined in <exception> which we can use in our programs. These are arranged in a parent-child class hierarchy see also `https://en.cppreference.com/w/cpp/error/exception`.

Assuming a block will raise an exception, a method catches an exception using a combination of the try and catch keywords. A try/catch block is placed around the code that might generate an exception. Code within a try/catch block is referred to as **protected code**, and the syntax for using try/catch as follows. You can list down multiple catch statements to catch different type of exceptions in case your try block raises more than one exception in different situations.

```
#include <iostream>
using namespace std;

double division(int a, int b) {
   if( b == 0 ) {
      throw "Division by zero condition!";
   }
   return (a/b);
}

int main () {
   int x = 50;
   int y = 0;
   double z = 0;

   try {
      z = division(x, y);
      cout << z << endl;
   } catch (const char* msg) {
     cerr << msg << endl;
   }

   return 0;
}
```

Because we are raising an exception of type `const char*`, so while catching this exception, we have to use `const char*` in catch block. If we compile and run above code, this would produce the following result:

```
Division by zero condition!
```

A more complex Example

Source Code 2.76: Test Application Exception throwing (Exercise: 17.1)

```
/*
============================================================================
 Name        : call_error.cpp
 Author      : A. Pretschner
 Version     :
 Copyright   :
 Description : definition and test of calc() with exception
 ============================================================================
 */

#include <iostream>
#include <string>
using namespace std;

double calc( int a, int b );

class Error{
    // specific error message
};

int main(){
    int x, y;
    double erg;
    bool flag = false;

    do {
       try {                               // try-Block
         cout << "input two positive Integer numbers: ";
         cin >> x >> y;
         erg = calc( x, y);
         cout << x << "/" << y << " = " << erg << endl;
         flag = true;        // leave loop.
       }
       catch( string& s){              // 1. catch-Block
```

```
         cerr << s << endl;
       }
       catch( Error& z){               // 2. catch-Block
         cerr << "Division by 0! \n" << endl;
       }
       catch(...){                     // 3. catch-Block
         cerr << "Unexpected Exception! \n";
         exit(1);
       }
    }while(!flag);

    // and running  ...
    return 0;
}

double calc( int a, int b ){
   if ( b < 0 )
      throw (string)"Denominator is negative!";

   if( b == 0 )   {
      Error errorObj;
      throw errorObj;
   }

   return ((double)a/b);
}
```

You can define your own exceptions by inheriting and overriding exception class functionality. Following is the example, which shows how you can use `std::exception` class to implement your own exception in standard way:

Source Code 2.77: Test Application throwing own exception (Exercise: 17.2)

```
/*
============================================================================
 Name        : call_error_extended.cpp
 Author      : A. Pretschner
 Version     :
 Copyright   :
 Description : definition and test of calc() with own created exception
                              MathError
 ============================================================================
 */

#include <string>
#include <iostream>
using namespace std;

class MathError: public exception {
   private:
     string message;
   public:
     MathError( const string& s) : message(s) {}
     virtual const string& getMessage()const {return message;}
};


 double calc( int a, int b );
```

```
int main() {
   int x, y;  bool flag = false;

   do {
     try                                  // try-Block
     {
       cout << "input two positive Integer Numbers: ";
       cin >> x >> y;
       cout << x <<"/"<< y <<" = "<< calc( x, y) << '\n';
       flag = true;         // leave loop.
     }
     catch( MathError& err)               // catch-Block
      {
        cerr << err.getMessage() << endl;
      }
      }while(!flag); // and running...
    return 0;
}

double calc( int a, int b ){
   if ( b < 0 )
      throw MathError("Denominator is negative!");
   if( b == 0 )
      throw MathError("Division by 0!");
   return ((double)a/b);
}
```

Task 17.1: *`class FloatVec` (see source code below) shall implement exception handling in case of wrong vector index usage.*

- *Define in the header file `floadVec.h` the error `class BadIndex` with one data field for securing the illegal index. The constructor copies one integer argument into that data field, method `getBadIndex()` returns the value of that field. Both index operations shall throw `BadIndex` type exceptions. This behavior is assumed for all other methods like `insert()` and `remove()`.*
- *The return type of `insert()` and `remove()` shall be `void`.*
- *Change the operator definitions so, that `BadIndex` exceptions are thrown if the the given index is outside the allowed range.*
- *Define and initialize a static vector in a `main()` test application. Let input a index value in dialog and show the vector element as long as the values are inside the allowed range. In case of range violation throw the exception handling process through the `catch()` handler (see hints below).*
- *Construct then a none static vector. A further exception handling shall now be in place. Input additional vector elements through the `insert()` method inside `try catch()` statement. If the write process fails the exception shall be thrown otherwise output all vector elements.*

Hints:

- Message thrown by 1. exception: Error during read access (illegal index) !
- Message thrown by 2. exception: Error during write access (illegal index) !

```
double& FloadVec::operator[] (int i) throw(out_of_range){
  if( i<0 || i>=cnt )
    throw out_of_range("Index out of Range");
  else return vecPtr[i];
}
```

```
int main(){
  try {
    //working with float vector values
  }
  catch (out_of_range& err){
    cout << err.what() << endl;
  }
  //run programm ....
}
```

Source Code 2.78: Class FloatVec throwing own exception (Exercise: 17.3)

```
/*
============================================================================
 Name        : floatvec.h
 Author      : A. Pretschner
 Version     :
 Copyright   :
 Description : defines FloatVec class with Exception Handling
 ============================================================================
 */
#ifndef _FLOATVEC_
#define _FLOATVEC_

#include <iostream>
using namespace std;

class BadIndex{
  private:
   int index;
  public:
    BadIndex(int i){index = i;}
    int getBadIndex() const {return index;}
};

class FloatVec
{
   private:
     float* vecPtr;    // dynamic  Element
     int max;          // Max count, without new memory allocation.
     int cnt;          // Element count

     void expand( int newRange);  // helper function for vector enlargement
   public:
     FloatVec( int n = 256 );
     FloatVec( int n, float value);

     FloatVec(const FloatVec& src);            // copy constructor
     FloatVec(FloatVec&& src);                 // Move-constructor

     ~FloatVec();

     FloatVec& operator=( const FloatVec&);   // assignment
     FloatVec& operator=(FloatVec&& src);     // Move-assignment

     int  length() const { return cnt; }
     // Index-Operators:
    float& operator[](int i) throw(BadIndex);
    float operator[](int i) const  throw(BadIndex);

    // Methods for attaching of float-value
    // or  float-vector:
    void append( float value);
    void append( const FloatVec& v);

    // Methods for inserting of float-value
    // or of float-vector:
    void insert( float value, int pos) throw(BadIndex);
    void insert( const FloatVec& v, int pos ) throw(BadIndex);

    void remove(int pos) throw(BadIndex);  // delete Position pos.

   // adding second Vector element wise :
   FloatVec& operator+=( const FloatVec& v2);

   void swap( FloatVec& v2) {      // swap *this with  v2
      if( this != &v2 ){
        // swap data fields:
         float* p = vecPtr;  vecPtr = v2.vecPtr;  v2.vecPtr = p;
         int i;
         i = max;   max = v2.max;  v2.max = i;
         i = cnt;   cnt = v2.cnt;  v2.cnt = i;
      }
   }

    // vector output
   friend ostream& operator<<( ostream& os, const FloatVec& v);
};

// Global swap Function:
inline void swap( FloatVec& v1, FloatVec& v2){     // swap v1 with v2.
   v1.swap(v2);
}

// element wise addition of two vectors:
FloatVec operator+( const FloatVec& v1, const FloatVec& v2);

#endif   // _FLOATVEC_
```

Task 17.2: *`class Fraction` (see source code below) shall implement exception handling in case of Division by 0. The constructor and the operator functions / and >> are influenced.*

- *Define into the `class Fraction` the error `class DivisionByZero` without data field. The Error class type then is `Fraction::DivisionbyZero`. Complete the constructor declaration and the operator functions / and >> with the exception handling specification,*
- *Change the operator definitions so, that `DivisionbyZero` exceptions are thrown if the denominator equals 0.*
- *Program three test cases inside `main()`:*
  1. *`try/ catch` Block testing the constructor during the initializing phase, throwing message "Error during initialization, denominator is 0!"*
  2. *`try/ catch` Block testing the division algorithm, throwing message "Error during division, division by 0 is not allowed!"*

3. *`try/ catch` Block testing the numerator/ denominator input dialog via console, throwing message "Error: Denominator is 0, input new denominator !=0:"*

Hints:

Source Code 2.79: Numerical Fraction class (Exercise: 17.4)

```
/*
============================================================================
 Name        : fraction.h
 Author      : A. Pretschner
 Version     :
 Copyright   :
 Description : numerical class for mathematical fractions
 ============================================================================
 */
#ifndef _FRACTION_
#define _FRACTION_

#include <iostream>
#include <cstdlib>
using namespace std;

class Fraction{
  private:
   long numerator, denominator;

   Fraction(long num = 0, long den = 1);

   Fraction operator-() const {
      return Fraction(-numerator, denominator);
   }

   Fraction& operator+=(const Fraction& a)  {
           numerator = a.numerator * denominator + numerator * a.denominator;
      denominator *= a.denominator;
      return *this;
   }

   Fraction& operator-=(const Fraction& a) {
      *this += (-a);
      return *this;
   }

   Fraction& operator++() {
           numerator += denominator;
      return *this;
   }

   Fraction& operator--(){
           numerator -= denominator;
      return *this;
   }

  friend Fraction operator+(const Fraction&, const Fraction&);
  friend Fraction operator-(const Fraction&, const Fraction&);
  friend Fraction operator*(const Fraction&, const Fraction&);
  friend Fraction operator/(const Fraction&, const Fraction&);
  friend ostream& operator<< (ostream& os, const Fraction& a);
  friend istream& operator>> (istream& is, Fraction& a);
};
#endif
```

Source Code 2.80: Numerical Fraction class Methods (Exercise: 17.5)

```
/*
============================================================================
 Name        : fraction.cpp
 Author      : A. Pretschner
 Version     :
 Copyright   :
 Description : Class Fraction methods implementation, friend functions
 ============================================================================
 */

#include "fraction.h"

// constructor:
Fraction::Fraction(long num, long den){
  if( num < 0 ) num = -num, den = -den;
  numerator = num;
  denominator  = den;
}

Fraction operator+(const Fraction& a, const Fraction& b){
   Fraction temp;
   temp.denominator = a.denominator * b.denominator;
   temp.numerator = a.numerator*b.denominator + b.numerator * a.denominator;
   return temp;
}

Fraction operator-(const Fraction& a, const Fraction& b ){
   Fraction temp = a;
   temp += (-b);
   return temp;
}

Fraction operator*(const Fraction& a, const Fraction& b ){
   Fraction temp;
   temp.numerator = a.numerator * b.numerator;
   temp.denominator  = a.denominator  * b.denominator;
   return temp;
}


Fraction operator/(const Fraction& a, const Fraction& b ){
   // a multiply with reciprocal of  b:
   Fraction temp;
   temp.numerator = a.numerator * b.denominator;
   temp.denominator  = a.denominator  * b.numerator;

   if( temp.denominator < 0 ){
     temp.numerator = -temp.numerator,
     temp.denominator  = -temp.denominator;
   }
   return temp;
}

ostream& operator<<(ostream& os, const Fraction& a){
  os << a.numerator << "/" << a.denominator;
  return os;
}

istream& operator>>(istream& is, Fraction& a){

  cin.sync(); cin.clear();
  cout << "Input fraction:\n"
          "  ->numerator:      ";
  is >> a.numerator;

  cin.sync(); cin.clear();
  cout << "  ->denominator != 0:  ";
  is >> a.denominator;

  if( !is) return is;

  if( a.denominator == 0) {
      cout << "\nError: Denominator is 0\n"
           << "  New Denominator != 0: ";   is >> a.denominator;
  }

  if( a.denominator < 0 ){
     a.numerator = -a.numerator,
     a.denominator  = -a.denominator;
  }
  return is;
}
```

Some explanations about **friend functions**:

In principle, private and protected members of a class cannot be accessed from outside the same class in which they are declared. However, this rule does not apply to *friends*. Friends are functions or classes declared with the **friend keyword**.
A non-member function can access the private and protected members of a class if it is declared a friend of that class. That is done by including a declaration of this external function within the class, and preceding it with the keyword friend:

```
// friend functions
#include <iostream>
using namespace std;

class Rectangle {
    int width, height;
  public:
    Rectangle() {}
    Rectangle (int x, int y) : width(x), height(y) {}
    int area() {return width * height;}
    friend Rectangle duplicate (const Rectangle&);
};

Rectangle duplicate (const Rectangle& param){
  Rectangle res;
  res.width = param.width*2;
  res.height = param.height*2;
  return res;
}

int main () {
  Rectangle foo;
  Rectangle bar (2,3);
  foo = duplicate (bar);
  cout << foo.area() << '\n';
  return 0;
}
```

The `duplicate` function is a friend of class Rectangle. Therefore, *function duplicate* is able to access the members width and height (which are private) of different objects of type Rectangle. Notice though that neither in the declaration of duplicate nor in its later use in main, function duplicate is **considered a member of class Rectangle**. **It isn't!** It simply has access to its private and protected members without being a member.

Source Code 2.81: Numerical Fraction test application (Exercise: 17.6)

```
/*
============================================================================
 Name        : fraction_t.cpp
 Author      : A. Pretschner
 Version     :
 Copyright   :
 Description : test application for  Fraction
 ============================================================================
 */

#include <iostream>
#include "fraction.h"
using namespace std;

int main(){
      Fraction c(5,0);
      Fraction a(1,3), b(3);

     cout << "\nSome test outputs:\n\n";

     cout << " a = " << a << endl;
     cout << " b = " << b << endl;

     cout << " a + b = " << (a + b) << endl;
     cout << " a - b = " << (a - b) << endl;
     cout << " a * b = " << (a * b) << endl;
     b = 0;
     cout << " a / b = " << (a / b) << endl;


   cout << "  --a =  " <<  --a << endl;
   cout << "  ++a  = " <<  ++a << endl;

   a += Fraction(1,2);
   cout << " a+= 1/2;  a = " << a << endl;

   a -= Fraction(1,2);
   cout << " a-= 1/2;  a = " << a << endl;
   cout << "-b = " << -b << endl;

   cout << "\nNow with input\n";
     cin.sync(); cin.clear();
           cin  >> a;

   cout << "\nYour input: " << a  << endl;

   return 0;
}
```

Lab 2.8: Templates

Overloaded functions

In C++, two different functions can have the same name if their parameters are different; either because they have a different number of parameters, or because any of their parameters are of a different type. For example:

```
// overloading functions
#include <iostream>
using namespace std;

int operate (int a, int b){
  return (a*b);
}

double operate (double a, double b){
  return (a/b);
}

int main (){
  int x=5,y=2;
  double n=5.0,m=2.0;
  cout << operate (x,y) << '\n';
  cout << operate (n,m) << '\n';
  return 0;
}
```

In this example, there are two functions called `operate`, but one of them has two parameters of type `int`, while the other has them of type `double`. The compiler knows which one to call in each case by examining the types passed as arguments when the function is called. If it is called with two int arguments, it calls to the function that has two int parameters, and if it is called with two doubles, it calls the one with two doubles.
Note that a function cannot be overloaded only by its return type. At least one of its parameters must have a different type.

Function templates

Overloaded functions may have the same definition. For example:

```
// overloaded functions
#include <iostream>
using namespace std;

int sum (int a, int b){
  return a+b;
}
```

```
double sum (double a, double b){
  return a+b;
}

int main (){
  cout << sum (10,20) << '\n';
  cout << sum (1.0,1.5) << '\n';
  return 0;
}
```

Here, sum is overloaded with different parameter types, but with the exact same body.
The function `sum` could be overloaded for a lot of types, and it could make sense for all of them to have the same body. For cases such as this, C++ has the ability to define functions with **generic types**, known as **function templates**. Defining a function template follows the same syntax as a regular function, except that it is preceded by the `template` keyword and a series of template parameters enclosed in angle-brackets <>:
`template <template-parameters> function-declaration`
The template parameters are a series of parameters separated by commas. These parameters can be generic template types by specifying either the `class` or `typename` keyword followed by an identifier. This identifier can then be used in the function declaration as if it was a regular type. For example, a generic sum function could be defined as:

```
template <class SomeType>
SomeType sum (SomeType a, SomeType b){
  return a+b;
}
```

It makes no difference whether the generic type is specified with keyword `class` or keyword `typename` in the template argument list (they are 100% synonyms in template declarations).
In the code above, declaring `SomeType` (a generic type within the template parameters enclosed in angle-brackets) allows `SomeType` to be used anywhere in the function definition, just as any other type; it can be used as the type for parameters, as return type, or to declare new variables of this type. In all cases, it represents a generic type that will be determined on the moment the template is instantiated.
Instantiating a template is applying the template to create a function using particular types or values for its template parameters. This is done by calling the function template, with the same syntax as calling a regular function, but specifying the template arguments enclosed in angle brackets:
`name <template-arguments> (function-arguments)`
For example, the sum function template defined above can be called with: `x = sum<int>(10,20);`

```
// function template
#include <iostream>
using namespace std;

template <class T>
T sum (T a, T b) {
  T result;
  result = a + b;
  return result;
}

int main () {
  int i=5, j=6, k;
  double f=2.0, g=0.5, h;
  k=sum<int>(i,j);
  h=sum<double>(f,g);
  cout << k << '\n';
  cout << h << '\n';
  return 0;
}
```

In this case, we have used `T` as the template parameter name, instead of `SomeType`. It makes no difference, and `T` is actually a quite common template parameter name for generic types.
In the example above, we used the function template `sum` twice. The first time with arguments of type `int`, and the second one with arguments of type `double`. The compiler has instantiated and then called each time the appropriate version of the function.

A more complex example

Source Code 2.82: Numerical Fraction test application (Exercise: 18.1)

```
/*
============================================================================
 Name        : stack.h
 Author      : A. Pretschner
 Version     :
 Copyright   :
 Description : template class Stack with push() pop() methods
 ============================================================================
 */
#ifndef STACK_H_
#define STACK_H_

template<class T>
class Stack{
  private:
    T* basePtr;      // pointer to the stack itself
    int top;         // stack top
    int max;         // maximal count
  public:
    Stack( int n ){ basePtr = new T[n]; max = n; top = 0; }
    Stack( const Stack<T>&);
    ~Stack(){ delete[] basePtr; }

    Stack<T>& operator=( const Stack<T>& );

    bool empty(){ return (top == 0); }
    bool push( const T& x);
    bool pop(T& x);
};

// Definition of  push() and pop()

template<class T>
bool Stack<T>::push( const T& x){
   if(top <= max - 1){              // if not full
       basePtr[top++] = x;
       return true;
   }
   else return false;
}
```

```
template<class T>
bool Stack<T>::pop( T& x){
   if(top > 0) {                  // if not empty
       x = basePtr[--top];
       return true;
   }
   else return false;
}

template<class T>
Stack<T>::Stack( const Stack<T>& stk){
   top = stk.top;
   max = stk.max;

   basePtr = new T[max];

   for(int i = 0; i < top; i++)
       basePtr[i] = stk.basePtr[i];
}

template<class T>
Stack<T>&  Stack<T>::operator=( const Stack<T>& stk){
   top = stk.top;
   max = stk.max;

   delete[] basePtr;
   basePtr = new T[max];

   for(int i = 0; i < top; i++)
       basePtr[i++] = stk.basePtr[i];
}

#endif   // STACK_H_
```

Source Code 2.83: Class DayTime (Exercise: 18.2)

```
/*
============================================================================
 Name        : daytime.h
 Author      : A. Pretschner
 Version     :
 Copyright   :
 Description : class Daytime with comparison operators ++ and -- and for in/output << >>
 ============================================================================
 */
#ifndef _DAYTIME_
#define _DAYTIME_

#include <iostream>
#include <iomanip>
using namespace std;

class DayTime{
  private:
    short hour, minute, second;
    bool overflow, underflow;
    void inc() {             // private helper function for ++
      ++second;
      if( second >= 60)     // handle overflow.
          second = 0,  ++minute;
      if( minute >= 60)
          minute = 0,  ++hour;
      if( hour >= 24)
          hour = 0,  overflow = true;
    }
    void dec() {             // private helper function for  --
      --second;
      if( second < 0)       // handle underflow.
          second = 59,  --minute;
      if( minute < 0)
          minute = 59,  --hour;
      if( hour < 0)
          hour = 0,  underflow = true;
    }

  public:
    DayTime( int h = 0, int m = 0, int s = 0)  {
      overflow = underflow = false;
      if( !setTime( h, m, s))
        hour = minute = second = 0;
    }

    bool setTime(int hour, int minute, int second = 0)    {
      if(    hour   >= 0  &&  hour < 24
          && minute >= 0  &&  minute < 60
          && second >= 0  &&  second < 60 )   {
        this->hour   = (short)hour;
        this->minute = (short)minute;
        this->second = (short)second;
        return true;
      }
      else
        return false;
    }

    int getHour()   const { return hour;   }
    int getMinute() const { return minute; };
    int getSecond() const { return second; };

    int asSeconds() const {   // daytime in seconds
      return (60*60*hour + 60*minute + second);
    }

    DayTime& operator++() {         // ++seconds
      inc();
      return  *this;
    }
    DayTime operator++(int) {       // seconds++
      DayTime temp(*this);
      inc();
      return temp;
    }

    DayTime& operator--() {          // --seconds
      dec();
      return  *this;
    }

    DayTime operator--(int) {       // seconds--
      DayTime temp(*this);
      dec();
      return temp;
    }
};

// ---  comparison operators  ---
//  t1 < t2
inline bool operator<( const DayTime& t1,
                       const DayTime& t2){
   return  t1.asSeconds() < t2.asSeconds();
}


//  t1 <= t2
inline bool operator<=( const DayTime& t1,
                        const DayTime& t2){
        return  t1.asSeconds() <= t2.asSeconds();
}

//  t1 == t2
inline bool operator==( const DayTime& t1,
                        const DayTime& t2){
        return  t1.asSeconds() == t2.asSeconds();
}

//  t1 != t2
inline bool operator!=( const DayTime& t1,
                        const DayTime& t2){
        return !(t1 == t2);
}

//  t1 > t2
inline bool operator>( const DayTime& t1,
                       const DayTime& t2){
        return (t2 < t1);
}

//  t1 >= t2
inline bool operator>=( const DayTime& t1,
                        const DayTime& t2){
        return !(t1 < t2);
}

//  ---  In- and Output  ---
inline ostream& operator<<( ostream& os, const DayTime& t){
   int w = os.width();        // get field width.
   if( w > 12)
```

```
     os << setw(w-12) << "";

   os << setfill('0')
      << setw(2) << t.getHour()    << ':'
      << setw(2) << t.getMinute()  << ':'
      << setw(2) << t.getSecond()  << " Hour";
   os << setfill(' ');
   return os;
}

inline istream& operator>>( istream& is, DayTime& t){
   cout << "Input Daytime (Format hh:mm:ss) : ";
   int hr = 0, min = 0, sec = 0;
   char c1 = 0, c2 = 0;
   if( !(is >> hr >> c1 >> min >> c2 >> sec))
     return is;
   if( c1 != ':' || c2 != ':' || ! t.setTime(hr,min,sec))
     is.setstate( ios::failbit);   // Failure!
                                   //  => set fail-Bit.
   return is;
}

#endif   // _DAYTIME_
```

Task 18.1: *Define function template `interpolSearch()` which searches elements in an numerical ordered vector. The vector elements are of type `T`, the search algorithm is descirbed below.*

- *The function templates owns three parameters: the search element of type `T`, a pointer to the first vector element, number of the vector elements. The found element index or otherwise -1 is returned.*
- *The search algorithm is defined in `search.h`*
- *Define `insertionSort()` which sorts the vector in ascending order. The function template owns two parameters: pointer to the first vector element, number of elements and now return value.*
- *Define the function `display()` to output of one vector element.*
- *The functions `interpolSearch()`, `insertionSort()` and `display()` shall be template functions of type `double` and `short`. For testing reasons create one vector of each kind. Write the `main()` output the ordered vector (see below one possible output).*
- *Extend the the `main()` function so, that each search function is been called with one existing and not existing vector element.*

Source Code 2.84: Possible interpolation test output (Exercise: 18.3)

```
Instantiation for short-type:
The Vector:
7  9  2  4  1

After sorting:
The Vector:
1  2  4  7  9

Vectorelement? 7

Found!

Instantiation  for double-type:
```

```
The Vector:
5.7  3.5  2.1  9.4  4.3

After Sorting:
The Vector:
2.1  3.5  4.3  5.7  9.4

Vectorelement? 9.9
Not found!
```

Hints:

- It is required, that the vector is numerical ordered in ascending manner.
- The interpolation search expects the search value to be at the compared vector position. If the searched value is less than the vector element value at this position, than the search algorithm is continued at the left part of the vector (like binary search algorithms) otherwise at the right part of the vector.
  ```
  double temp = (double) (key-vp[begin]);
  temp /= (vp[end] - vp[begin]);
  temp = temp * (end -begin) + 0.5
  expect = begin + (int)temp;
  ```
- `key` is the search value and `expect` the index position of the expected vector element
- The insertion-Sort-algorithm considers a left already ordered vector part and a right not ordered vector part. Each next element in the not sorted part is taken out of the vector. As long as a greater element in the sorted part of the vector, beginning with the top element, will be found, the position of the out taken element will be increased. If one smaller is found, this element will be inserted. At the beginning of the algorithm the left already sorted part consists only of the first vector element.

Task 18.2: *Define class template `Vector<T,n>` which enables up to `n` vector elements of type `T`. Accessing vector elements outside the vector shall lead into exception `class BadIndex`, if no free space is available the exception `class OutOfRange` shall be thrown.*

- *Define first the exception classes and reuse the example given in exercise 2.78 for the `BadIndex`.*
- *Change the already defined `FloatVec` into `class Vector<T, n>`. Default value for `n` is 255. The vector memory is assigned now statically, define the default constructor, and special constructor with for initialized vector elements. The copy constructor and destructor as also the assignment constructor are not to be self-created necessarily.*
- *Access methods are `size()` returning the maximum number of elements and `lenght()` returning the current number of elements in the vector. Additionally the same methods as in 2.78 are to be designed, but now as `void` methods. These methods shall throw exceptions of classes `BadIndex` and `OutOfRange`. Overload the index operator (throwing exception `BadIndex`) and the shift operator <<.*
- *Test the program with different objects. You can use for this matter the above designed `class DayTime` 2.86. Define a vector with 5 `class Time` elements and output the vector on screen.*

Source Code 2.85: Possible vector test output (Exercise: 18.4)

```
Here comes the constant double-Vector:
     9.9     9.9     9.9     9.9     9.9     9.9     9.9     9.9     9.9     9.9

Here comes the new double-Vector:
     9.9     9.9     9.9     9.9     9.9     9.9     9.9     9.9     9.9     9.9

Here the changed double-Vector:
     9.9     9.9     9.9     9.9     9.9     9.9     9.9     9.9     9.9      10      20
```

```
Here the int-Vector:
   83   86   77   15   93   35   86   92   49   21

And duplicated:
   83   86   77   15   93   35   86   92   49   21   83   86   77   15   93   35   86   92   49   21

Here the Copy of int-Vector:
   83   86   77   15   93   35   86   92   49   21   83   86   77   15   93   35   86   92   49   21
Input Daytime (Format hh:mm:ss) : 13:56:00
Input Daytime (Format hh:mm:ss) : 01:34:55
Input Daytime (Format hh:mm:ss) : 23:00:15

Here the  Vector with Daytime type objects:
        13:56:00 Hour        01:34:55 Hour        23:00:15 Hour
```

Lab 2.9: Container

The C++ STL (Standard Template Library) is a powerful set of C++ template classes to provide general-purpose classes and functions with templates that implement many popular and commonly used algorithms and data structures like vectors, lists, queues, and stacks.
At the core of the C++ Standard Template Library are following three well-structured components:

Containers are used to manage collections of objects of a certain kind. There are several different types of containers like dequeues, lists, vectors, maps etc.

Algorithms act on containers. They provide the means by which you will perform initialization, sorting, searching, and transforming of the contents of containers.

Iterators are used to step through the elements of collections of objects. These collections may be containers or subsets of containers.

Let take the following program that demonstrates the vector container (a C++ Standard Template) which is similar to an array with an exception that it automatically handles its own storage requirements in case it grows:

Source Code 2.86: Test application of STL classes (Exercise: 19.1)

```
#include <iostream>
#include <vector>
using namespace std;

int main() {

   // create a vector to store int
   vector<int> vec;
   int i;

   // display the original size of vec
   cout << "vector size = " << vec.size() << endl;

   // push 5 values into the vector
   for(i = 0; i < 5; i++) {
      vec.push_back(i);
   }

   // display extended size of vec
   cout << "extended vector size = " << vec.size() << endl;

   // access 5 values from the vector
   for(i = 0; i < 5; i++) {
      cout << "value of vec [" << i << "] = " << vec[i] << endl;
   }

   // use iterator to access the values
   vector<int>::iterator v = vec.begin();
   while( v != vec.end()) {
      cout << "value of v = " << *v << endl;
      v++;
   }

   return 0;
}
```

Here are following points to be noted related to various functions we used in the above example:

- `push_back()` member function inserts value at the end of the vector, expanding its size as needed.
- `size()` function displays the size of the vector.
- `begin()` returns an iterator to the start of the vector.
- `end()` returns an iterator to the end of the vector.

Algorithms

The header algorithm defines a collection of functions especially designed to be used on ranges of elements.They act on containers and provide means for various operations for the contents of the containers.

Algorithm:
- Sorting
- Searching
- Important STL Algorithms
- Useful Array algorithms
- Partition Operations

Numeric: valarray class

Containers

Containers or container classes store objects and data. There are in total seven standard “first-class” container classes and three container adaptor classes and only seven header files that provide access to these containers or container adaptors:

- *Sequence Containers:* implement data structures which can be accessed in a sequential manner.
  - vector, implement C-vector typical operations but are able to dynamically grow and shrink
  - list
  - deque (double ended queue)
  - arrays
  - forward_list(Introduced in C++11)
- *Container Adaptors :* provide a different interface for sequential containers.
  - queue
  - priority_queue
  - stack
- *Associative Containers :* implement sorted data structures that can be quickly searched (O(log n) complexity).
  - set
  - multiset
  - map

– multimap

Sequential Container: Example

Source Code 2.87: STL class Example (Exercise: 19.2)

```
/*
============================================================================
 Name        : sortvec.h
 Author      : A. Pretschner
 Version     :
 Copyright   :
 Description : template class SortVec
 ============================================================================
 */

#include <vector>
#include <functional>  // for Comparator-class less<T>
using namespace std;

template <class T, class Compare = less<T> >
class SortVec : public vector<T>  {
  public:
   SortVec() {}
   SortVec(int n, const T& x = T()) : vector<T>(n,x) {}

   void insert(const T& obj);        // Sorted insertion
   int  search(const T& obj);        // search Object
   void merge(const SortVec& v);     // merge
};


// ----------------------------------------------------------
// insert()
//
// Insert new object at the end of the vector
// then sorting that object into vector.

template <class T, class Compare >
void SortVec<T, Compare>::insert(const T& obj) {
   typename SortVec::iterator  pos, temp;
   this->push_back(obj);                // insert at the end

   pos = this->end()-1;                 // last position

   while( pos > this->begin()) {         // sorting:
     --pos;
         if( Compare()(obj,*pos) )    // swap:
     { temp = pos; *(++temp) = *pos;  *pos = obj; }
     else   break;
   }
}

// ----------------------------------------------------------
// search()
//
```

```
// Search given object in container with the binary search algorithm.
// Hereby the object is compared with the element value
// at the middle vector position. Continuing searching left side if the object is less
// than the element otherwise in the right part of the vector.
// The binary search algorithm runs logarithmically fast.
// Return-value: Index of the found object
//               otherwise  -1.

template <class T, class Compare >
int SortVec<T, Compare>::search(const T& obj){
  int first = 0, last = this->end() - this->begin() - 1, mid;

  while( first < last )  {
     mid = (first + last + 1)/ 2;
                               // search in lower part:
         if( Compare()(obj,(*this)[mid]) )
            last = mid - 1;
     else
        first = mid;           // respectively in upper part.
  }

  if( obj == (*this)[first] )  // found?
     return first;             // Yes.
  else
     return -1;                // No.
}

// ---------------------------------------------------------
// merge()

template <class T, class Compare >
void SortVec<T,Compare>::merge(const SortVec<T,Compare>& v){
  SortVec temp;                     // Temporary Vector
  typename SortVec::iterator pos = this->begin();  // Iterator
  unsigned int n1 = 0, n2 = 0;

        // copy the respective smaller object into temp:
  while(n1 < this->size() &&  n2 < v.size())
    if( Compare()(pos[n1], v[n2]) )   // if( pos[n1] < v[n2] )
       temp.push_back(pos[n1++]);
    else
        temp.push_back(v[n2++]);
                                    // attache the rest:
  while( n1 < this->size())
    temp.push_back(pos[n1++]);

  while( n2 < v.size())
    temp.push_back(v[n2++]);

  this->swap(temp);      // more effective then  *this = temp;
}
```

Source Code 2.88: Test application for STL class Example (Exercise: 19.3)

```
/*
============================================================================
 Name         : sortvec_t.cpp
 Author       : A. Pretschner
 Version     :
 Copyright   :
 Description : test application for  SortVec
 ============================================================================
 */


#include "sortvec.h"
#include <iostream>
using namespace std;

typedef SortVec<int> IntSortVec;

void display(const IntSortVec& c);

int main(){
   IntSortVec v, w;             // Default-constructor

   v.insert(2);                 // insert
   v.insert(7);
   v.insert(1);

   display(v);

   int key;
   cout << "Key? " << endl;
   cin >> key;

   int i = v.search(key);         // search
   if( i > -1)
       cout << "Key "  << key
            << " at Index " << i << " found!" << endl;
   else
       cout << "Key "  << key << " not found!" << endl;

   cout << "\nNow insert into second vector: " << endl;
   w.insert(3);                   // insert
   w.insert(9);
   display(w);

   cout << "\nMerge Vectors: " << endl;
   v.merge(w);                    // merge

   display(v);
   return 0;
}


void display(const IntSortVec& c){
   SortVec<int>::const_iterator pos;         // Iterator

   for( pos = c.begin(); pos != c.end(); pos++)
     cout << *pos << ' ';
   cout << endl;
}
```

Example output:

```
1 2 7
Key?
2
Key 2 at Index 1 found!

Now insert into second vector:
3 9

Merge Vectors:
1 2 3 7 9
```

Hints:

- `this->push_back()` insert at end
- `this->insert()` before given position
- `this->push_front()` insert at the begin
- `this->front()` access to first element
- `this->back()` access to last element

Sequential Container: Example for deletion

Source Code 2.89: Test application for STL class Example (Exercise: 19.4)

```
/*
============================================================================
 Name        : con-list_t.cpp
 Author      : A. Pretschner
 Version     :
 Copyright   :
 Description : test application for  list operations
 ============================================================================
 */

#include <list>
#include <cstdlib>
#include <iostream>
#include <ctime>
using namespace std;

typedef list<int> INTLIST;
int display( const INTLIST& c);

int main(){
  INTLIST ls, sls;

  srand( (unsigned int)time(NULL));  // init random number
  int i;
  for( i=1; i<=3; i++)
    ls.push_back( rand()%10 );       // z.B.  1 7 4

  cout << "First List ls: " << endl;
  display(ls);

  ls.push_back(ls.front());          // 1 7 4 1
  cout << "Attach First Element:" << endl;
  display(ls);
  ls.reverse();                     // 1 4 7 1
  cout << "Invert:" << endl;
  display(ls);

  ls.sort();                        // 1 1 4 7
  cout << "Sort:" << endl;
  display(ls);

  for( i=1; i<=3; i++)
    sls.push_back( rand()%10 );     // z.B. 0 9 4
  cout << "New List sls:" << endl;
  display(sls);

  // first Object of sls before last object of ls:
  INTLIST::iterator pos = ls.end();
  ls.splice(--pos, sls, sls.begin());   // 1 1 4 0 7

  cout << "First Object of  sls prior to last object of ls:" << endl;
  display(ls);

  cout << "Second List sls:" << endl;
  display(sls);                      // 9 4

  ls.sort();                         // 0 1 1 4 7
  sls.sort();                        // 4 9
  ls.merge(sls);                     // 0 1 1 4 4 7 9
  cout << "Sort and  Merge:" << endl;
  display(ls);

  ls.unique();                       // 0 1 4 7 9
  cout << "Make unambiguous:" << endl; display(ls);

  return 0;
}


int display(const INTLIST& c){
   int z = 0;                           // counter
   INTLIST::const_iterator pos;         // Iterator

   for( pos = c.begin(); pos != c.end(); pos++, z++)
     cout << *pos << ' ';
   cout << endl;
   cin.get();

   return z;
}
```

Example output:

```
7  Harry
4  Hermine
1  Ron
```

Sequential Container: List operations

Source Code 2.90: Test application for STL class list (Exercise: 19.5)

```
/*
============================================================================
 Name        : con-list_t.cpp
 Author      : A. Pretschner
 Version     :
 Copyright   :
 Description : test application for  list operations
 ============================================================================
 */

#include <list>
#include <cstdlib>
#include <iostream>
#include <ctime>
using namespace std;

typedef list<int> INTLIST;
int display( const INTLIST& c);

int main(){
  INTLIST ls, sls;

  srand( (unsigned int)time(NULL));  // init random number
  int i;
  for( i=1; i<=3; i++)
    ls.push_back( rand()%10 );

  cout << "First List ls: " << endl;
  display(ls);

  ls.push_back(ls.front());
  cout << "Attach First Element:" << endl;
  display(ls);

  ls.reverse();
  cout << "Invert:" << endl;
  display(ls);

  ls.sort();
  cout << "Sort:" << endl;
  display(ls);

  for( i=1; i<=3; i++)
    sls.push_back( rand()%10 );
  cout << "New List sls:" << endl;
  display(sls);

  // first Object of sls before last object of ls:
  INTLIST::iterator pos = ls.end();
  ls.splice(--pos, sls, sls.begin());

  cout << "First element of sls before last element of ls:" << endl;
  display(ls);

  cout << "Second List sls:" << endl;
  display(sls);
  ls.sort();
  sls.sort();
  ls.merge(sls);
  cout << "Sort and  Merge:" << endl;
  display(ls);

  ls.unique();
  cout << "Make unambiguous:" << endl; display(ls);

  return 0;
}


int display(const INTLIST& c){
   int z = 0;                           // counter
   INTLIST::const_iterator pos;         // Iterator

   for( pos = c.begin(); pos != c.end(); pos++, z++)
     cout << *pos << ' ';
   cout << endl;
   cin.get();

   return z;
}
```

Example output:

```
First List ls:
0 2 1

Attach First Element:
0 2 1 0

Invert:
0 1 2 0

Sort:
0 0 1 2

New List sls:
5 0 5

First Object of  sls prior to last object of ls:
0 0 1 5 2

Second List sls:
0 5

Sort and  Merge:
0 0 0 1 2 5 5

Make unambiguous:
0 1 2 5
```

Associative Container: Sets and Multisets

Source Code 2.91: Test application for STL class list (Exercise: 19.6)

```
/*
============================================================================
 Name        : sortvec.h
 Author      : A. Pretschner
 Version     :
 Copyright   :
 Description : test application for  associative container: set
 ============================================================================
 */
#include <set>
#include <cstdlib>
#include <ctime>
#include <iostream>
using namespace std;

typedef set<int> IntSet;              // Set for int-values
typedef multiset<int> IntMultiSet;    // Multiset for int-values

int main(){
   IntSet  lotto;                     // create Set

   srand( (unsigned int)time(NULL)); // init random number generator
   while( lotto.size() < 6)           // insert
     lotto.insert( 1 + rand()%49 );

   cout << "Your Lotto number: " << endl;
   for( auto pos = lotto.begin(); pos != lotto.end(); pos++)
       cout << *pos << "   ";
   cout << endl << endl;

   IntMultiSet  ms;                   // create Multiset

   for( int i=0; i < 10; i++)         // insert
     ms.insert( rand()%10 );

   cout << "Here 10 random numbers "
        << "between 0 and 10: " << endl;
   for( auto mpos = ms.begin(); mpos != ms.end(); mpos++)
       cout << *mpos << "   ";
   cout << endl;
   return 0;
}
```

Example output:

```
Your Lotto number:
3   6   9   12   24   46

Here 10 random numbers between 0 and 10:
1   3   3   3   4   5   6   8   8   9
```

Associatve Container: Maps and Multimaps

Source Code 2.92: Test application for STL class list (Exercise: 19.7)

```
/*
============================================================================
 Name        : map_t.cpp
 Author      : A. Pretschner
 Version     :
 Copyright   :
 Description : test application for  associative container: maps
 ============================================================================
 */

#include <map>
#include <string>
#include <iostream>
using namespace std;

typedef multimap<int, string> MULTI_MAP;
typedef MULTI_MAP::iterator ITERATOR;

int main(){
   MULTI_MAP  m;            // create Multimap.
   ITERATOR pos;            // Iterator.
                            // insert:
   m.insert(pair<int, string>(7, "Harry") );
   m.insert(pair<int, string>(3, "Hermine"));
   m.insert(pair<int, string>(1, "Bellatrix"));
   m.insert(pair<int, string>(1, "Ron"));

   cout << "The Multimap: " << endl;

   for(pos = m.begin(); pos!= m.end(); pos++)
       cout << pos->first << "  "
            << pos->second << endl;
   cout << endl;

   pos = m.find(3);       // search pair to key
   if( pos != m.end())
     cout << pos->first << "  " << pos->second << endl;

   int key = 1;             // determine object number
   cout << "To key " << key << " exist "
        <<  m.count(key) << " Objects " << endl;

   return 0;
}
```

Example output:

```
The Multimap:
1  Bellatrix
1  Ron
3  Hermine
7  Harry

3  Hermine
To key 1 exist 2 Objects
```

Initializing lists and Range-for-loops

Source Code 2.93: Class Account initializing (Exercise: 19.8)

```
/*
 ============================================================================
 Name        : account2.h
 Author      : A. Pretschner
 Version     : Kirch & Prinz, 7. Edition
 Copyright   : Your copyright notice
 Description : Definition of class Account
 ============================================================================
 */


#ifndef _ACCOUNT_H
#define _ACCOUNT_H

#include <iomanip>
#include <string>
#include <iostream>

using namespace std;

class Account {
    private:
      string name;
      unsigned long nbr;
      double status;

    public:
      Account(const string s = "X", unsigned long n = 1111111L,  double st = 0.0) {
              name = s;  nbr = n;    status = st;
      }

      long getNr() const { return nbr; }
      void setNr(unsigned long n){ nbr = n; }
      // etc.

      void display() const{
          cout << fixed << setprecision(2)
               << "----------------------------------\n"
               << "Owner:       " << name  << endl
               << "Number:      " << nbr    << endl
               << "Status:      " << status << endl
               << "----------------------------------\n"
               << endl;
      }
};


#endif
```

Source Code 2.94: Test application for list initializing (Exercise: 19.9)

```
/*
============================================================================
 Name        : initList_t.cpp
 Author      : A. Pretschner
 Version     :
 Copyright   :
```

```
 Description : test application for  initializing container and range-for-loops
 ===========================================================================
 */


#include <vector>
#include <map>
#include <string>
#include <iostream>
#include "account2.h"
using namespace std;

int main(){
   int arr[] { 1, 2, 3, 4, 5 };       // initialize int-Array.
   for( int& el : arr){                // element reference el.
       el += 10;                      // change each element
       cout << el << "  ";            // and show.
   }
   cout << endl;

   // initialize vector-Object with Account-Elements :
   vector<Account> kontoVec { Account("Beatrix", 12345UL, 1950.99),
                            Account("Clara", 54321UL, 1150.99),
                            Account("Julia",  98765UL, 950.99)  };
   // show Vector:
   for( const auto& k : kontoVec)       // k is  read-only-Reference
       k.display();

   // initialize  Multimap .
   multimap<int,string> mmap = { { 5, "Jeany" }, { 3, "Vivi" }, { 5, "Anna" } };

   cout << "The Multimap contains " << mmap.size() << " Elements:" << endl;
   for( const auto& pair : mmap)
       cout << pair.first << "  " << pair.second << endl;
   cout << endl;
   return 0;
}
```

Example output:

```
11  12  13  14  15
----------------------------------
Owner:       Beatrix
Number:      12345
Status:      1950.99
----------------------------------

----------------------------------
Owner:       Clara
Number:      54321
Status:      1150.99
----------------------------------

----------------------------------
Owner:       Julia
Number:      98765
Status:      950.99
----------------------------------
```

```
The Multimap contains 3 Elements:
3  Vivi
5  Jeany
5  Anna
```

Hash-Tables

A hash table is traditionally implemented with an array of *linked lists*. When we want to insert a *key/Value* pair, we map the key to an index in the array using the hash function. The value is then inserted into the linked list at that position.
Note: The elements in the linked list at a particular index of the array do not have the same key. Rather, hash function(key) is the same for these values. Therefore, in order to retrieve the value for a specific key, we need to store in each node both the exact key and the value.
To summarize, a hash table will be implemented with an array of linked lists, where each node in the linked list holds two pieces of data: the value and the original key. In addition, we will want to note the following design criteria:

- We want to use a good hash function to ensure that the keys are well distributed. If they are not well distributed, then we would get a lot of collision and the speed to find an element would decline.
- No matter how good hash function is, we will still have collisions, so we need a method for handling them. this often means chaining via a linked list, but it's not the only way.
- We may also wish to implement methods to dynamically increase or decrease the hash table size depending on capacity. For example, when the ratio of the number of elements to the table size exceeds a certain threshold, we may wish to increase the hash table size. This would mean creating a new hash table and transferring the entries from the old table to the new table. Because this is an expensive operation, we want to be careful to not do it too often.

```
class Customer {
  private:
    int id;
    string name; //key and customer name
  public:
    Customer(int i=0, const string& s="X") {id=i; name=s;}

    //Access methods getID() and operator methods
    friend bool operator == (const Customer& a,
                             const Customer& b){
      return a.id == b.id;
    }

    // more operator << ...
};

namespace std { //the same name space in which hash<T> is declared
  template <> struct hash<Customer>{
    size_t operator() (const Customer& ct) const{
      hash<int> int_hash;
      return int_hash(ct.getID());
    }
  };
}
```

Objects in associative containers can be managed with *hash tables*. Hash tables contain elements of variable length. This elements are called **buckets**, which are indexed starting by 0.

Each **bucket** is a link list and can store multiple objects.
During hash table creation a finite number *b* of buckets is initialized. If *n* is the number of stored objects in the hash table, the **load factor** is defined by `load_factor = n/b;`. If the load factor exceeds a given value (default value = 1), the number of buckets is increased and the hash table will be reorganized.
The **hash-functions** convert the object key into integer value of type `size_t`. The functional value `modulo b` is equal to the bucket number in which the object is stored. Thus, objects, which keys have the same hash-value are stored in the same bucket.
A **hash-function** can be realized as structure of type `hash<T>` and can be parameterized with keys of type `T` and operator overloading `operator()`. There exists specialized overloaded templates for different data types, strings and pointers. Above example shows the hash table specialization `hash<Customer>` for the defined `class Customer`. The operator function declares a functional object of type `hash<int>` which calculates a value of type `size_t` of the key `id`.

Has-Tables: Example

Source Code 2.95: Test application for hash tables (Exercise: 19.10)

```
/*
============================================================================
 Name        : unordered-set_t.cpp
 Author      : A. Pretschner
 Version     :
 Copyright   :
 Description : test application for  template unordered_set<T>
 ============================================================================
 */


#include <unordered_set>
#include <string>
#include <iostream>
using namespace std;

class Customer {
  private:
    int id; string name;   // key and  customer name
  public:
    Customer(int i=0, const string& s="X"){id = i; name = s;}

    // access methods getID() etc. and operator functions:
    int getID() const { return id; }

    friend bool operator==(const Customer& a,
                           const Customer& b){
            return a.id == b.id;
    }

    friend ostream& operator<<(ostream& os, const Customer& ct){
        os << "ID = " << ct.id << ", Name = "<< ct.name; return os;}
};

namespace std {     //the same name space in which hash<T> is declared
  template<> struct hash<Customer>  {
     size_t operator()(const Customer& ct) const  {
        hash<int> int_hash;
        return int_hash(ct.getID());
     }
  };
}

typedef unordered_set<Customer> HashTable;

int main(){
    HashTable hashtable;
    hash<Customer> hash = hashtable.hash_function();

    hashtable.max_load_factor(2);   // Max. load factor : 2 for testing

    for(int i= 1; i <= 15; i++)
       hashtable.emplace(12 + i);

    //for( auto &cust : hashtable)
    //    cout << cust << endl;
    //cin.get();
                                    // Bucketweise ausgeben:
    HashTable::const_local_iterator loc_it;

    for(size_t i = 0; i < hashtable.bucket_count(); i++)    {
      cout << "Bucket number: " << i << endl;
      if(hashtable.bucket_size(i) > 0)      {
        loc_it = hashtable.begin(i);
        while(loc_it!=hashtable.end(i))
              cout  << "Customer: " << *loc_it++  << endl;
      }
    }

    cout << "\nCount Buckets: "
         << hashtable.bucket_count()  << endl;

                                             // Bucket suchen:
    cout <<  "Search Customer with id  21. "  << endl;

    auto it = hashtable.find(21);

    cout << "Customer: " << *it << endl;
    size_t nr = hashtable.bucket(it->getID());
    cout << "Bucket number: " << nr << endl;
    cout << "Number of objects in bucket: "
         << hashtable.bucket_size(nr) << endl;
    cout << "Hash-value: " << hash(*it) << endl;

    return 0;
}
```

Example output:

```
Bucket number: 0
Customer: ID = 17, Name = X
Bucket number: 1
Customer: ID = 18, Name = X
Bucket number: 2
Customer: ID = 19, Name = X
```

```
Bucket number: 3
Customer: ID = 20, Name = X
Bucket number: 4
Customer: ID = 21, Name = X
Bucket number: 5
Customer: ID = 22, Name = X
Bucket number: 6
Customer: ID = 23, Name = X
Bucket number: 7
Customer: ID = 24, Name = X
Bucket number: 8
Customer: ID = 25, Name = X
Bucket number: 9
Customer: ID = 26, Name = X
Bucket number: 10
Customer: ID = 27, Name = X
Bucket number: 11
Bucket number: 12
Bucket number: 13
Customer: ID = 13, Name = X
Bucket number: 14
Customer: ID = 14, Name = X
Bucket number: 15
Customer: ID = 15, Name = X
Bucket number: 16
Customer: ID = 16, Name = X

Count Buckets: 17
Search Customer with id  21.
Customer: ID = 21, Name = X
Bucket number: 4
Number of objects in bucket: 1
Hash-value: 21
```

Hints

- The class comparison operator == has to be overloaded.
- A hash function has to be defined. Because of template `hash<T>` for unsorted associative container a specialized template has to be created and the operator `operator ()` has to be overloaded
- Because of the limited *load_factor* a unordered set of 15 `Customer` objects with maximum of 2 objects of each bucket is created.

Task 19.1: *In data communication between remote devices messages are transported through different sub nets. Routers are active devices for queuing and forwarding messages between the source and target network of the communication peers. Special routing algorithms are used normally for managing large address tables. One simple routing algorithm without address tables is know as Hot-Potato-Algorithm. Hereby the router is trying to forward the incoming message as fast as possible to the shortest output queue (https://en.wikipedia.org/wiki/Hot-potato_and_cold-potato_routing).*

- *Create container `class VecQueue<T>` which is parameterized with message of type `T`. The class consists of one vector of queues of type `vector< queue<T> >` and one data field to store the current count of queues.*
- *The constructor initializes with argument `n` given number of empty queues. The methods `size()`, `empty()`, `push()` and `pop()` are to be declared.*
- *The method `size()` is to be overloaded: without argument returning the count of all messages, with argument `i` of type `int` returning count of messages of the i-th queue.*

Appropriately define `empty()` and `empty(int i)` which return true or false in case all queues or the i-th queue is empty.

- *Getting messages shall be simulated with `pop()` and `pop(i)` where i takes message from i-th queue*
- *In the `main()` test application declare container of type `VecQueue<int>`. A message here contains a random number in range of 0 to 99 which have to be generated in a loop. Some messages shall be directed to the output some messages shall remain in the queues (see below). For that purpose use `pop (i)`.*

Hints:

```
Initialized are 9 Queues.

Filling up with help of Hot-Potato-Algorithm.

Delete some elements from random chosen queues ...

Remaing elements are:
1.Queue:   86   6  28  24  86  67
2.Queue:   80  64  25  35  92  49  50
3.Queue:   25  84  52  97  61
4.Queue:   67  26  64  71  13
5.Queue:   36  44  27   6  68  33  35  57
6.Queue:   10  95  15  38  17  17  74  49
7.Queue:
8.Queue:   81  23  82  54  35
9.Queue:   46  75  61  96  80  69
```

Task 19.2: *The access to one online portal shall be secured by passwords. For that purpose each new user gets user account with user name and password data. The user name and the hash value of the password are stored internally in an hash table.*

- *Define the `class WebAccount` with the user name data field of type `string` and one flag indicating if the user is logged in. Create the appropriate constructors and access methods.*
- *Define the `class UserTable` with following data fields:*
  - *Hash-functionof type `hash<string>`*
  - *Object of type unsorted `Multimap`, holding the hash values of type `size_t` and is initialized with type `WebAccount`*
- *Below described methods are to be declared as `public`.*
- *Function `find()` for getting the user account position in the hash-table and function `isUser()` for checking if the user account exists, are to be declared as `private`. Bot methods have two arguments - the user name and the password. Method `find()` determines first with call to `equal_range()` the range of hash value of the given password. In that range then, then methods iterate to find the appropriate user name.*
- *Test `class UserTable` in `main()` by defining one object of type `Multimap` and integrate the following menu. Password and user name are input always in dialog.*

Hints:

- `adduser()` - adding user account into hash table if not exists, no automatic login
- `login()` - user login if exists, `flag` is set to `true`
- `removeUser()` - delete user from hash table
- `displayActiveUsers()` - shows all logged in users.

Hints

```
1 = Create new web account
 2 = Login
 3 = Logout
 4 = Delete web account
 5 = Show active users
 0 = Exit program
```

Lab 2.10: Miscellaneous

Bitmanipulation

In C, following 6 operators are bitwise operators (work at bit-level):

& (`bitwise AND`) Takes two numbers as operands and does AND on every bit of two numbers. The result of AND is 1 only if both bits are 1.

| (`bitwise OR`) Takes two numbers as operands and does OR on every bit of two numbers. The result of OR is 1 any of the two bits is 1.

$\wedge$ (`bitwise XOR`) Takes two numbers as operands and does XOR on every bit of two numbers. The result of XOR is 1 if the two bits are different.

$<<$ (`left shift`) Takes two numbers, left shifts the bits of the first operand, the second operand decides the number of places to shift.

$>>$ (`right shift`) Takes two numbers, right shifts the bits of the first operand, the second operand decides the number of places to shift.

(`bitwise NOT`) Takes one number and inverts all bits of it

Following C-program demonstrates the operations:

```c
/*
=============================================================================
 Name        : bit-test.c
 Author      : A. Pretschner
 Version     :
 Copyright   :
 Description : C program to show bitmanipulations.
 =============================================================================
 */

#include<stdio.h>
int main(){
        unsigned char a = 5, b = 9; // a = 5(00000101), b = 9(00001001)
        printf("a = %d, b = %d\n", a, b);
        printf("a&b = %d\n", a&b); // The result is 00000001
        printf("a|b = %d\n", a|b); // The result is 00001101
        printf("a^b = %d\n", a^b); // The result is 00001100
        printf("~a = %d\n", a = ~a); // The result is 11111010
        printf("b<<1 = %d\n", b<<1); // The result is 00010010
        printf("b>>1 = %d\n", b>>1); // The result is 00000100
        return 0;
}
```

Output:

```
a = 5, b = 9
a&b = 1
a|b = 13
a^b = 12
~a = 250
b<<1 = 18
b>>1 = 4
```

Following are interesting facts about bitwise operators:

- The *left shift* and *right shift* operators should not be used for negative numbers If any of the operands is a negative number, it results in undefined behaviour. For example results of both $-1 << 1$ and $1 << -1$ is undefined. Also, if the number is shifted more than the size of integer, the behaviour is undefined. For example, $1 << 33$ is undefined if integers are stored using 32 bits.

- The *bitwise XOR* operator is the most useful operator from technical interview perspective. It is used in many problems. A simple example could be "Given a set of numbers where all elements occur even number of times except one number, find the odd occurring number" This problem can be efficiently solved by just doing *XOR* of all numbers.

```
/*
============================================================================
 Name        : find-odd.c
 Author      : A. Pretschner
 Version     :
 Copyright   :
 Description : Function to return the only odd occurring element.
 ============================================================================
 */

int findOdd(int arr[], int n) {
int res = 0, i;
for (i = 0; i < n; i++)
        res ^= arr[i];
return res;
}

int main(void) {
int arr[] = {12, 12, 14, 90, 14, 14, 14};
int n = sizeof(arr)/sizeof(arr[0]);
printf ("The odd occurring element is %d ", findOdd(arr, n));
return 0;
}
```

- The bitwise operators should not be used in place of logical operators. The result of logical operators (&&, || and !) is either 0 or 1, but bitwise operators return an integer value. Also, the logical operators consider any non-zero operand as 1. For example, consider the following program, the results of & and && are different for same operands.

```
/*
============================================================================
 Name        : find-odd.c
 Author      : A. Pretschner
 Version     :
 Copyright   :
 Description : bitwise operators and their logical output.
 ============================================================================
 */

int main() {}
int x = 2, y = 5;
(x & y)? printf("True ") : printf("False ");
(x && y)? printf("True ") : printf("False ");
return 0;
}
// Output: False True
```

- The left-shift and right-shift operators are equivalent to multiplication and division by 2 respectively.
- The & operator can be used to quickly check if a number is odd or even, the value of expression (`x & 1`) would be non-zero only if x is odd, otherwise the value would be zero.
- The operator should be used carefully, the result of operator on a small number can be a big number if the result is stored in an unsigned variable. And result may be nega-

tive number if result is stored in signed variable (assuming that the negative numbers are stored in 2's complement form where leftmost bit is the sign bit)

Bit Masks

The &-operator is often used to delete specific bits, which are declared by *MASK*. In `c = c & 0x7F;` the `MASK 0x7F` sets the most significant bits to 1 and all least significant bits to 0. Thus, a bits in c are deleted with exception of the 7 least significant bits. The variable c may be of each integer type. If more than 1 byte is used, than the `MASK 0x7F` is filled up in the most significant bits with 0.

Settings and inverting bits

The |-operator sets specific bits. The following example shows the transformation of one character. In ASCII-code the distinction between upper and lower character is only the 5-th bit.

```
/*
============================================================================
 Name        : bitmask.c
 Author      : A. Pretschner
 Version     :
 Copyright   :
 Description : bitmask operations.
 ============================================================================
 */
 #define MASK 0x20
 char c = 'A';
 c = c | MASK;
```

The bit calculation:

$$\begin{array}{c|cccccccc|c}
 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 'A' = 0x41 \\
| & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & MASK = 0x20 \\
\hline
 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 'a' = 0x61
\end{array}$$

Last but not least the ∧-operator inverts specific bits. Thus, each 0-bit is set to 1 and each 1-bit is set to 0. In `c = c ^ 0xAA;` each second bit is inverted.
Summarizing for any statement x and any `MASK` is valid:

- `x & MASK` sets all bits in x to zero, which position are marked in `MASK` with 0
- `x | MASK` sets all bits in x to 1, which position are marked in `MASK` with 1
- `x ^ MASK` inverts all bits, which positions in `MASK` are set to 1.

Example: Bitmasks

```
/*
============================================================================
 Name        : paraity_t.cpp
 Author      : A. Pretschner
 Version     :
 Copyright   :
 Description : definition and test for  parity(),         return value 0 if 1-bit count is even , else 1
 ============================================================================
 */


unsigned int bit0( unsigned int x ) {
   return (x & 1);
}
```

```
int parity( unsigned int n){
   unsigned int par = 0;
   for( ; n != 0; n >>=1 )
     par ^= bit0(n);
   return par;
}
```

```
#include <iostream>
using namespace std;

int main(){
    unsigned int x;

    cout << "Insert positive integer number:  ";
    cin >> x;

    if( parity(x) != 0)
        cout << "Number " << x
             << " has odd count of 1-bits";
    else
        cout << "Number " << x
             << " has even count of 1-bits";
    cout << endl;

    return 0;
}
```

Creating own Bit Masks

Example `x = x & ~3`: Mask 3 sets only bits 0 and 1, so that Mask 3 contains only of 1-bits with except of both least significant bits, which are set to 0. Thus in the example the two least significant bits are set to 0.
Mask 3 is independent of the cpu word length and should be used instead of Mask 0xFFFC.
Other examples are Masks, which are used to manipulate only one bit. Here left-shifts of 1 can be used:

- `x=x | (1<<6)`; sets the 6-th bit in x,
- `x=x & ~(1 << 6)`; deletes 6-th bit in x

With variable bit setting:

```
int setBit(int x, unsinged int n){
  if (n < 8*sizeof(int))
    return (x | (1<<n));
}
```

Bit Fields

A Bit field declares a member with explicit width, in bits. Adjacent bit field members may be packed to share and straddle the individual bytes.
A bit field declaration is a `struct` or `union` member declaration which uses the following declarator:

```
identifier(optional) : width
```
with:

identifier	the name of the bit field that is being declared. The name is optional: nameless bit fields introduce the specified number of bits of padding
width	an integer constant expression with a value greater or equal to zero and less or equal the number of bits in the underlying type. When greater than zero, this is the number of bits that this bit field will occupy. The value zero is only allowed for nameless bit fields and has special meaning: it specifies that the next bit field in the class definition will begin at an allocation unit's boundary.

Hints: Bit fields can have only one of four types (possibly const or volatile qualified):

- `unsigned int`, for unsigned bit fields (`unsigned int b:3;` has the range 0..7)
- `signed int`, for signed bit fields (`signed int b:3;` has the range -4..3)
- `int`, for bit fields with implementation-defined signedness (Note that this differs from the meaning of the keyword int everywhere else, where it means ”signed int”). For example, `int b:3;` may have the range of values 0..7 or -4..3.
- `_Bool`, for single-bit bit fields (`bool x:1;` has the range 0..1 and implicit conversions to and from it follow the boolean conversion rules.

Example a):

```
#include <stdio.h>
struct S {
 // three-bit unsigned field,
 // allowed values are 0...7
 unsigned int b : 3;
};

int main(void) {
    struct S s = {7};
    ++s.b; // unsigned overflow
    printf("%d\n", s.b); // output: 0
}
```

Example b):

```
#include <stdio.h>
struct S {
    // will usually occupy 4 bytes:
    // 5 bits: value of b1
    // 11 bits: unused
    // 6 bits: value of b2
    // 2 bits: value of b3
    // 8 bits: unused
    unsigned b1 : 5, : 11, b2 : 6, b3 : 2;
};

int main(void) {
    printf("%zu\n",sizeof(struct S)); // usually prints 4
}
```

Example c):

```
#include <stdio.h>
struct S {
    // will usually occupy 8 bytes:
    // 5 bits: value of b1
    // 27 bits: unused
    // 6 bits: value of b2
    // 15 bits: value of b3
    // 11 bits: unused
    unsigned b1 : 5;
    unsigned :0; // start a new unsigned int
    unsigned b2 : 6;
    unsigned b3 : 15;
};

int main(void) {
    printf("%zu\n", sizeof(struct S)); // usually prints 8
}
```

For bit fields exist some restrictions:

- The address operator can not be used, vector of bit fields are not allowed.
- The bit field alignment depends of running machine.

Below the example of the ATM-communication cell is shown:

General Flow Control		Virtual Path Identifier	
Virtual Path Identifier		Virtual Channel Identifier	
Virtual Channel Identifier			
Virtual Channel Identifier		Payload Type	Cell Los Priority
Header Error Control			

```
struct ATM_Cell{
  unsigned GFC : 4;   // general flow control
  unsigned VPI : 8;   // virtual path identifier
  unsigned VCI : 16;  // virtual channel identifier
  unsigned PT  : 3;   // payload type
  unsigned CLP : 1;   // cell loss priotity
  unsigned HEC : 8;   // header error control
  char payload[48];   // user information
}
```

Task 20.1: *Write a function `putBits()` which returns a bit pattern of type `unsigned int`. Independently of the machine word length 16 bit shall be output. The argument of the function is the number to show, the function returns nothing.*

- *Write a "teaching program" for bit operations. Read from keyboard the variables x and y, then the applied function `putBits()` shall output `x, x&y, x|y, x^y` and `~x`*
- *Next shift the value x to right and to left by certain bit positions, input with the keyboard. In case of none valid input position, 1 shall be used instead.*
- *A running program example is shown below.*

Hint:

```
     ******  BIT-OPERATORS  ******

Input two integer numbers.

1. number --> 57
2. number --> -3

Bit pattern of     57 = x :    0000 0000 0011 1001
Bit pattern of     -3 = y :    1111 1111 1111 1101
Bit pattern of     x & y  :    0000 0000 0011 1001
Bit pattern of     x | y  :    1111 1111 1111 1101
Bit pattern of     x ^ y  :    1111 1111 1100 0100

By how many positions should x be shifted?
```

```
Count --> 4

Bit pattern of     x <<  4 :    0000 0011 1001 0000
Bit pattern of     x >>  4 :    0000 0000 0000 0011
Repeat (y/n)?
```

Task 20.2: *For spy defense in data communication the transfer data shall be encrypted. The sender encrypts the transfer data with help of filters. The receiver decrypts with the same filter.*

- *Define `swapBits()` for swapping two bits of an integer variable. Arguments of the function are the the `int` value and bit positions used for swapping. The return value of this function is the new `int` value. In case of invalid bit position the unchanged `int` value is returned.*
- *Write a filter for all character values, not containing any control sequences (ASCII-Code > 32) which swap the bits at bit position 5,6 and 0,4 and 1,3.*
- *First, encrypt the values and write them into a file, second read the file and let pass the values by the filter and output the decrypted values on screen.*

OS - Laboratory Training Files

Lab 3.1: Critical Sections

To obtain a mutual exclusion, bounded waiting, and progress there have been several algorithms implemented, two of them are Dekker's Algorithm and Peterson's Algorithm. To understand the algorithm let's understand the solution to the critical section problem first.
A process is generally represented as :

Source Code 3.96: Process flow (Exercise: 21.1)

```
do {
    //entry section
        critical section
    //exit section
        remainder section
} while (TRUE);
```

The solution to critical section problem must ensure following three conditions:

- Mutual Exclusion
- Progress
- Bounded Waiting

One of solution for ensuring above all factors is Peterson's solution.

Explanation of Peterson's algorithm

Peterson's Algorithm is used to synchronize two processes. It uses two variables, a **bool array flag** of size 2 and an int variable **turn** to accomplish it. In the solution **i** represents the **Consumer** and **j** represents the **Producer**. Initially the flags are false. When a process wants to execute it's *critical section*, it sets it's **flag** to true and **turn** as the index of the other process. This means that the process wants to execute but it will allow the other process to run first. The process *performs busy waiting* until the other process has finished it's own critical section.
After this the current process enters it's critical section and adds or removes a random number from the shared buffer. After completing the critical section, it sets it's own **flag** to false, indication it does not wish to execute anymore.
The program runs for a fixed amount of time before exiting. This time can be changed by changing value of the the macro **RT**.

Source Code 3.97: Peterson's Algorithm Exercise (Exercise: 21.2)

```
/*
============================================================================
 Name        : peterson.c
 Author      : A. Pretschner
 Version     :
 Copyright   : Nabaneet Roy/ GeeksforGeeks
 Description : C program to implement Peterson's Algorithm
               for producer-consumer problem. (needs -lpthread)
 ============================================================================
 */

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

```
#include <unistd.h>
#include <time.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdbool.h>
#include <sys/time.h>
#include <sys/wait.h>

#define BSIZE 8 // Buffer size
#define PWT 2         // Producer wait time limit
#define CWT 10         // Consumer wait time limit
#define RT 10         // Program run-time in seconds

//defining 4 shared memories for
// 1 - flag
// 2 - turn
// 3 - buffer
// 4 - time stamp

int shmid1, shmid2, shmid3, shmid4;

//identifying keys for the memories access channel
key_t k1 = 5491, k2 = 5812, k3 = 4327, k4 = 3213;

bool* SHM1;
int* SHM2;
int* SHM3;

int myrand(int n){ // Returns a random number between 1 and n
        time_t t;
        srand((unsigned)time(&t));
        return (rand() % n + 1);
}

int main(){
        shmid1 = shmget(k1, sizeof(bool) * 2, IPC_CREAT | 0660);         // flag
        shmid2 = shmget(k2, sizeof(int) * 1, IPC_CREAT | 0660);         // turn
        shmid3 = shmget(k3, sizeof(int) * BSIZE, IPC_CREAT | 0660); // buffer
        shmid4 = shmget(k4, sizeof(int) * 1, IPC_CREAT | 0660);         // time stamp

        if (shmid1 < 0 || shmid2 < 0 || shmid3 < 0 || shmid4 < 0) {
                perror("Main shmget error: ");
                exit(1);
        }
        SHM3 = (int*)shmat(shmid3, NULL, 0);
        int ix = 0;
        while (ix < BSIZE) // Initializing buffer
                SHM3[ix++] = 0;

        struct timeval t;
        time_t t1, t2;
        gettimeofday(&t, NULL);
        t1 = t.tv_sec;

        int* state = (int*)shmat(shmid4, NULL, 0);
```

```
*state = 1;
int wait_time;

int i = 0; // Consumer
int j = 1; // Producer

/*Producer code*/
if (fork() == 0){
        SHM1 = (bool*)shmat(shmid1, NULL, 0);
        SHM2 = (int*)shmat(shmid2, NULL, 0);
        SHM3 = (int*)shmat(shmid3, NULL, 0);
        if (SHM1 == (bool*)-1 || SHM2 == (int*)-1 || SHM3 == (int*)-1) {
                perror("Producer shmat error: ");
                exit(1);
        }

        bool* flag = SHM1;
        int* turn = SHM2;
        int* buf = SHM3;
        int index = 0;

        while (*state == 1) {
                flag[j] = true;
                printf("Producer is ready now.\n\n");
                *turn = i;
                while (flag[i] == true && *turn == i)
                        ;

                // Critical Section Begin
                index = 0;
                while (index < BSIZE) {
                        if (buf[index] == 0) {
                                int tempo = myrand(BSIZE * 3);
                                printf("Job %d has been produced\n", tempo);
                                buf[index] = tempo;
                                break;
                        }
                        index++;
                }
                if (index == BSIZE)
                        printf("Buffer is full, nothing can be produced!!!\n");
                printf("Buffer: ");
                index = 0;
                while (index < BSIZE)
                        printf("%d ", buf[index++]);
                printf("\n");
                // Critical Section End

                flag[j] = false;
                if (*state == 0)
                        break;
                wait_time = myrand(PWT);
                printf("Producer will wait for %d seconds\n\n", wait_time);
                sleep(wait_time);
        }
        exit(0);
```

```
	}

	/*Consumer code*/
	if (fork() == 0){
		SHM1 = (bool*)shmat(shmid1, NULL, 0);
		SHM2 = (int*)shmat(shmid2, NULL, 0);
		SHM3 = (int*)shmat(shmid3, NULL, 0);
		if (SHM1 == (bool*)-1 || SHM2 == (int*)-1 || SHM3 == (int*)-1) {
			perror("Consumer shmat error:");
			exit(1);
		}

		bool* flag = SHM1;
		int* turn = SHM2;
		int* buf = SHM3;
		int index = 0;
		flag[i] = false;
		sleep(5);
		while (*state == 1) {
			flag[i] = true;
			printf("Consumer is ready now.\n\n");
			*turn = j;
			while (flag[j] == true && *turn == j)
				;

			// Critical Section Begin
			if (buf[0] != 0) {
				printf("Job %d has been consumed\n", buf[0]);
				buf[0] = 0;
				index = 1;
				while (index < BSIZE) {// Shifting remaining jobs forward
					buf[index - 1] = buf[index];
					index++;
				}
				buf[index - 1] = 0;
			} else
				printf("Buffer is empty, nothing can be consumed!!!\n");
			printf("Buffer: ");
			index = 0;
			while (index < BSIZE)
				printf("%d ", buf[index++]);
			printf("\n");
			// Critical Section End

			flag[i] = false;
			if (*state == 0)
				break;
			wait_time = myrand(CWT);
			printf("Consumer will sleep for %d seconds\n\n", wait_time);
			sleep(wait_time);
		}
		exit(0);
	}
	// Parent process will now for RT seconds before causing child to terminate
	while (1) {
		gettimeofday(&t, NULL);
```

```
                t2 = t.tv_sec;
                if (t2 - t1 > RT){ // Program will exit after RT seconds
                        *state = 0;
                        break;
                }
        }
        // Waiting for both processes to exit
        wait(NULL);
        wait(NULL);
        printf("The clock ran out.\n");
        return 0;
}
```

Hints:

- `shmget` - returns the identifier of the System V shared memory segment associated with the value of the argument `key` http://man7.org/linux/man-pages/man2/shmget.2.html
- `key_t` - *key* key is one of the identifier of `key_t` type to identifying the message queue in the kernel apart from message queue id. Normally the `key` is obtained by

```
key_t key;
key = ftok("file.c", 'b'));/*instead of taking random 1234,
                             you are generating key from file based on proj_id */
msgid=msgget(key_t, 0666 | IPC_CREAT);
```

- **IPC** - Inter-Process-facilities are managing three resources: *shared memory segments*, *message queues* and *semaphore arrays*. The owner of the program owns the access rights to these resources. If you want to change the System V shared memory segment after creating one in first run you need to release that memory segment identified with the `key` before. In the shown code sequence the shared memory for the buffer is identified with the key `k3 = 4327` (in $hex = 0x000010e7$) and the id `shmid3`. To change the buffer size `BSIZE = 8` to `BSIZE = 10` do the following:

```
itworker@itworker:~/eclipse/cpp-photon/eclipse> ipcs

------ Message Queues --------
key        msqid      owner      perms      used-bytes   messages

------ Shared Memory Segments --------
key        shmid      owner        perms      bytes      nattch     status
0x00001573 2064419    itworker     660        2          2
0x000016b4 2097188    itworker     660        4          2
0x000010e7 3112997    itworker     660        32         5
0x00000c8d 2162726    itworker     660        4          3

ipcrm -M 0x000010e7
```

`ipcs` show information on IPC facilities and `ipcrm` remove certain IPC resources

- `shmat` - attaches the shared memory segment identified by shmid to the address space of the calling process https://linux.die.net/man/2/shmat
- `fork()` - creates a new process by duplicating the calling process. The new process is referred to as the child process. The calling process is referred to as the parent process http://man7.org/linux/man0-pages/man2/fork.2.html.
- `wait()` - blocks the calling process until one of its child processes exits or a signal is received. After child process terminates, parent continues its execution after wait system call instruction http://man7.org/linux/man-pages/man2/waitpid.2.html.

Task 21.1: *Alternate the above program in such way, that a) the buffer is fast running over and b) the buffer will never running over*

Task 21.2: *Extend the Example program in using a second buffer. The first buffer is used as a byte coded order buffer and the second buffer contains the byte coded answers. As suggestion* *define three questions to which a special answer shall be given (as a kind of protocol handling)*

Submit your answers from Lab 3.1 to Moodle one week after the assignment

Lab 3.2: Threads, Locks and Mutexe

Before you work out the Dekker's Solution in the next section have a glance to the following test applications. Example Threads (Exercise: 22.1) starts two threads and shows their id's. The `mutex` locks the critical section of the `class Thread`.

Source Code 3.98: Threads (Exercise: 22.1)

```
/*
============================================================================
 Name        : threadID.cpp
 Author      : A. Pretschner
 Version     :
 Copyright   :
 Description : Determines thread id
               Hint: Do not forget to set -lpthread
 ============================================================================
 */

#include <thread>
#include <mutex>
#include <iostream>
using namespace std;

mutex mtx;
class ThreadID
{
  public:
    void operator()()
    {
      mtx.lock();
      cout << "\nActive Thread: " << this_thread::get_id();
      mtx.unlock();
    }
};

int main()
{
   ThreadID a;
   thread t1(a), t2(a);

   mtx.lock();
   cout << "\nThread-ID of t1: " << t1.get_id();
   cout << "\nThread-ID of t2: " << t2.get_id();
   mtx.unlock();

   t1.detach();
   mtx.lock();
   cout << "\nThread t1 detached.\nThread-ID of t1: "
        << t1.get_id();
   mtx.unlock();

   t2.join();
   mtx.lock();
   cout << "\nAfter call of join()."
        << "\nThread-ID of t2: " << t1.get_id() << endl;
   mtx.unlock();

   return 0;
}
```

Note:

- `thread` - Class to represent individual threads of execution. A thread of execution is a sequence of instructions that can be executed concurrently with other such sequences in multithreading environments, while sharing a same address space http://www.cplusplus.com/reference/thread/thread/.The `thread` constructor constructs a thread object http://www.cplusplus.com/reference/thread/thread/thread/. In above case the initialization constructor is used. `template <class Fn, class... Args>`. `Fn` means A pointer to function, pointer to member, or any kind of move-constructible function object (i.e., an object whose class defines `operator()`, including closures and function objects). The return value (if any) is ignored. In above case the `void operator()` was defined.
- `mutex` -The mutex class is a synchronization primitive that can be used to protect shared data from being simultaneously accessed by multiple threads. `mutex` offers exclusive, non-recursive ownership semantics https://en.cppreference.com/w/cpp/thread/mutex

Hints: Visualizing Threads in OS

- Viewing threads of a process in Linux Server by : `htop`, `ps` or `pstree`

`htop` If not yet already done install the program via *zypper* or *aptget*. For showing the executed thread in one process you have to return the current PID of the application like:

```
Process ID: 9164

Aktive Thread: 140453167822592
Aktive Thread: 140453159429888
```

then set up the `htop` program to show all threads.

`ps` `ps -T -p 9164` or `cat /proc/9164/task/9164/status`

`htop` see above

`pstree` `pstree -p 9165`

Source Code 3.99: Function Objects (Exercise: 22.2)

```
/*
============================================================================
 Name        : funObj.cpp
 Author      : A. Pretschner
 Version     :
 Copyright   :
 Description : Initializing threads with Function Objects
 ============================================================================
 */

 #include <thread>
#include <iostream>
using namespace std;

struct FuncObj
{
    FuncObj(){                        // Default-Constructor
      cout << "\nConstructor constructs Object: "
           << (void*)this << endl;}
    FuncObj(const FuncObj& mt){     // Copy-Constructor
      cout <<"\nCopy Constructor constructs Object: "
           << (void*) this << endl;}

    FuncObj(FuncObj&& mt){          // Move-Constructor
      cout << "\nMove-Constructor constructs Object: "
           << (void*) this << endl;}

    ~FuncObj(){
      cout << "\nDestructor destroys Object: "
          << (void*) this << endl; }

    void operator()() {
      cout << "\n()-Operator call for Object: "
          << (void*) this << ". \n()-Operator finished. "
          << endl;
    }
};

int main(){
    FuncObj a;
    thread t(a);
    t.join();

    return 0;
}
```

In the next example the *comsumer-producer-problem* is shown. Two threads have congruent access to the same buffer (reading from /Writing to the buffer). Using a mutex does not solve the problem finally. Better is if:

- the consumer is sleeping (in blocked state) if the buffer is empty and will be wakened by the producer if data arrive in the buffer.
- the producer is sleeping if the buffer is full and will be wakened by the consumer when the buffer is empty.

Source Code 3.100: Producer-Consumer-Problem (Exercise: 22.3)

```
/*
============================================================================
 Name        : producer_consumer.h
 Author      : A. Pretschner
 Version     :
 Copyright   :
 Description : Class Definitions and thread conditions
 ============================================================================
 */

#include <thread>
#include <mutex>
#include <condition_variable>
#include <ctime>       // for time()
#include <cstdlib>     // for rand(), srand() und NULL
#include <iostream>
using namespace std;

class Buffer {
   private:
    unsigned buf;
    bool empty;    // true, if buffer is empty else  false.
    bool done;     // after last write-operation true

    mutex mtx;
    condition_variable cond;
   public:
    Buffer(): buf(0), empty(true), done(false){} //constructor

    void put(unsigned val) {
      unique_lock<mutex> lock(mtx);
      while(!empty)          // while buffer not empty:
        cond.wait(lock);     // sleep.

      buf = val;             // buffer empty: write.
      empty = false;         // buffer not empty.
      cond.notify_one();     // awake next thread .
    }

    unsigned get(){
      unique_lock<mutex> lock(mtx);
      while(empty)           // Wait, if buffer is empty.
        cond.wait(lock);

      unsigned res = buf;    // buffer read.
      empty = true;          // buffer empty again.
      cond.notify_one();     // awake next thread.
      return res;
     }

    void setDone(){ done = true; cond.notify_one();}
    bool isDone() const{ return done; }
};

// Producer and Consumer classes
// -------------------------------------------

class Producer{
  private:
   Buffer& bufref; //reference to buffer
  public:
   Producer(Buffer& b):bufref(b){     //constructor
           srand((unsigned)time(NULL));
   }

   void operator()(){                //operator for thread function
     for(int i=0; i < 10; i++)
             bufref.put(rand());
     bufref.setDone();               // writing finished
   }
};

class Consumer {
  private:
    Buffer& bufref;
   public:
     Consumer(Buffer& b) : bufref(b){}  //constructor

     void operator()() {
     int val;
     while(!bufref.isDone()){
              val = bufref.get();
              cout << "Customer read " << val << endl;
     }
  }
};
```

The main application for the consumer-producer-problem below show the write- and read-operation of an random number into the buffer from both sides.

Source Code 3.101: Application for P-C-P (Exercise: 22.4)

```
/*
============================================================================
 Name        : producer_consumer_t.cpp
 Author      : A. Pretschner
 Version     :
 Copyright   :
 Description : test application - thread conditions
 ============================================================================
 */
#include "consumer_producer.h"

int main(){
        //initialize buffer and classes
        Buffer b;
        Producer p(b);
        Consumer c(b);

        //initialize threads
        thread t1(p);
        thread t2(c);

        //run threads
        t1.join();
        t2.join();
        return 0;
}
```

Note:
In above example both `Producer` and `Consumer` hold a reference to the `Buffer` object. The `operator()` is overloaded in both cases to read/ write a random number 10 times to and from the buffer.

Finally a locking mechanism is shown below:

Source Code 3.102: Locks (Exercise: 22.5)

```
/*
============================================================================
 Name        : producer_consumer_t.cpp
 Author      : A. Pretschner
 Version     :
 Copyright   :
 Description : usage of lock_guard
 ===========================================================================
 */

#include <thread>
#include <mutex>
#include <iostream>
using namespace std;

mutex mtx;
unsigned long sum = 0;

class Add {
    int fct;

    long nextSum(){       // long time lasting calculation: sum += fct.
        for(int i=1; i <= fct; i++)
            sum++;
        return sum;
    }

  public:           // the lock secures the not interruptible calculation
    Add(int n = 256) : fct(n){}

    void operator()(int m)  {
       lock_guard<mutex> lock(mtx);   // create lock

       for(int i=1; i <= m; i++)   {
         sum = nextSum();
                                      // result check
         if( sum % fct != 0 )
            cout << "\nThread-ID " << this_thread::get_id()
                 << ": Result not divisible by "
                 << fct << " !";
       }
       cout << "Intermediate result: " << sum << endl;
    }
};

int main() {
       Add d;                 // calculate multiples of 256
       thread t1(d,10);       // and add separately:
       thread t2(d,20);       // 10 * 256 + 20*256

       t1.join();
       t2.join();

       cout << " Final result = " << sum << endl;
       return 0;
}
```

Task 22.1: *Create random numbers in different threads and output those numbers avoiding congruent memory access. Define for that matter the `class RandomFunctor` in an header file describing the Function Objects. The class owns a static data element `count` for the running thread number. The operator function `operator()` contains as argument the number of the random numbers, increments the thread number and shows the random number with the according thread-id. Synchronize the data access with the lock-mechanism. In the source file define the `mutex` and initialize the data element of `RandomFunctor`. The `main()` code contains* *the Function Object of class type `RandomFunctor` and initializes several threads. Output the main thread-id too (see listing below).*

Source Code 3.103: Output for RandomFunctor (Exercise: 22.6)

```
1th Thread is executed:
Thread-ID = 140305604011776
20 Random numbers:
  83  86  77  15  93  35  86  92  49  21
  62  27  90  59  63  26  40  26  72  36

2th Thread is executed:
Thread-ID = 140305612404480
10 Random numbers:
  11  68  67  29  82  30  62  23  67  35

In main() function: Thread-ID = 140305629857472
```

Note: Suggestion for possible used functions

- `rand()` - Returns a pseudo-random integral number in the range between 0 and RAND_MAX. This number is generated by an algorithm that returns a sequence of apparently non-related numbers each time it is called. This algorithm uses a seed to generate the series, which should be initialized to some distinctive value using function srand (http://www.cplusplus.com/reference/cstdlib/rand/).
- `setw()` - Sets the field width to be used on output operations. Behaves as if member width were called with n as argument on the stream on which it is inserted/extracted as a manipulator (it can be inserted/extracted on input streams or output streams) (http://www.cplusplus.com/reference/iomanip/setw/).

Task 22.2: *One client uses the server service of time assuming recherche tasks. The message communication uses one common message buffer. The request is identified by number, the replay is a simple string. Define the Message Buffer as `class ClientServerStream` with the data fields of type `int` for the request and type `string` for the replay. Additionally define two boolean variables for the server status with `true` for active and the client status with `true` for read last request. Define the methods `request()` and `respond()` accordingly.Define the `class Server` and `Client` which both hold a object reference to the `lientServerstream` type. Overload the object's function `operator()` that requests and answers are evaluated loop wise. Test the program by initializing the communication buffer and start threads for Server and Client accordingly.*

Hints: for `request()` and `response()` methods

- `request()` sleeps while the Server is active, than stores the given request number from the argument inside the classes data structure, awakes the Server and waits until the Answering is finished, outputs the response and waits for next request.
- `response()` sleeps while the Client is active, than reads the request number from the data field, writes the answer into the data storage of the class, deletes the Server-Busy-Flag and leaves the method.

Source Code 3.104: Output for Client-Server (Exercise: 22.7)

```
Client´s response:  Request Number 1

Client´s response:  Request Number 2

Client´s response:  Request Number 3
```

```
Program finished.
```

Note: Suggestion for possible used functions

- `notify_one()` - Unblocks one of the threads currently waiting for this condition. If no threads are waiting, the function does nothing. If more than one, it is unspecified which of the threads is selected. (http://www.cplusplus.com/reference/condition_variable/condition_variable/notify_one/).
- `notify_all()` - Unblocks all threads currently waiting for this condition. If no threads are waiting, the function does nothing. (http://www.cplusplus.com/reference/condition_variable/condition_variable/notify_all/).

Task 22.3: *Supposed many threads have read and write access to one common buffer. Now it's possible to use Locks to avoid their congruent access. But if one thread wants to write something many other reading threads can try to starve the read thread. A writing thread should be allowed as soon as possible to write. For that matter define in a Header file a* `class Database` *with following elements:* `String data` *for the common ressource, two counter* `cntReader` *for Reader and* `cntWriter` *for the Writer thread, two boolean flags* `busyReader` *and* `busyWriter` *announcing which thread is active.*

Hints: for `class Database()` methods

- `Database()` - the constructor sets the `counter` to 0 and the flags to false.
- `write()` - manages the `cntWriter` and `busy-Writer-Flag`. The method is blocked using the condition variable `okToWrite` while one other thread is active, than writes the given string argument at the end of the class field `data` and outputs `data` together with the number of Writers, finally unblocks the access for the next write access and all other read access.
- `read()` - manages `cntReader` and `busy-Reader-Flag`. The method is blocked while one Writer is active or the resource `data` is empty, then reads the resource and outputs the total number of Readers, finally unblocks for the next read access.

Submit your answers from Lab 3.2 to Moodle one week after the assignment

Lab 3.3: Variation of Dekker's Solution

Dekker's algorithm was the first provably-correct solution to the critical section problem. It allows two threads to share a single-use resource without conflict, using only shared memory for communication. It avoids the strict alternation of a naïve turn-taking algorithm, and was one of the first mutual exclusion algorithms to be invented.
Although there are many versions of Dekker's Solution, the final 5th version is the one that satisfies all of the above conditions and is the most efficient of them all. The versions do not yet compile! Find the solution for running all of the 5 Dekker Versions. Try to adopt Locks (Exercise: 23.0)
Note - Dekker's Solution, mentioned here, ensures mutual exclusion between two processes only, it could be extended to more than two processes with the proper use of arrays and variables.
Algorithm - It requires both an array of Boolean values and an integer variable:

Source Code 3.105: Critical Sections (Exercise: 23.1)

```
var flag: array [0..1] of boolean;
turn: 0..1;
repeat
```

```
        flag[i] := true;
        while flag[j] do
                if turn = j then
                begin
                        flag[i] := false;
                        while turn = j do no-op;
                        flag[i] := true;
                end;

                critical section

        turn := j;
        flag[i] := false;

                remainder section

until false;
```

First Version of Dekker's Solution The idea is to use common or shared thread number between processes and stop the other process from entering its critical section if the shared thread indicates the former one already running.

Source Code 3.106: Dekker's Algorithm 1. Version(Exercise: 23.2)

```
/*
============================================================================
 Name        : dekker1.c
 Author      : A. Pretschner
 Version     :
 Copyright   : Nabaneet Roy/ GeeksforGeeks
 Description : C program to implement Dekker's Algorithm
               for producer-consumer problem.
 ============================================================================
 */
int Main()
{

        int thread_number = 1;
        startThreads();
        return 0;
}

Thread1()
{
        do {

                // entry section
                // wait until threadnumber is 1
                while (threadnumber == 2)
                        ;

                // critical section

                // exit section
                // give access to the other thread
                threadnumber = 2;
                // remainder section

        } while (completed == false)
}

Thread2()
{

        do {

                // entry section
                // wait until threadnumber is 2
                while (threadnumber == 1)
                        ;

                // critical section

                // exit section
                // give access to the other thread
                threadnumber = 1;

                // remainder section

        } while (completed == false)
}
```

The problem arising in the above implementation is lockstep synchronization, i.e each thread depends on the other for its execution. If one of the processes completes, then the second process runs, gives access to the completed one and waits for its turn, however, the former process is already completed and would never run to return the access back to the latter one. Hence, the second process waits infinitely then.

Second Version of Dekker's Solution - To remove lockstep synchronization, it uses two flags to indicate its current status and updates them accordingly at the entry and exit section.

Source Code 3.107: Dekker's Algorithm 2. Version(Exercise: 23.3)

```
/*
============================================================================
 Name        : dekker2.c
 Author      : A. Pretschner
 Version     :
 Copyright   : Deitel & Associates, Inc.
 Description : C program to implement Dekker's Algorithm
               for producer-consumer problem.
 ============================================================================
 */
Main()
{

        // flags to indicate if each thread is in
        // its critial section or not.
        boolean thread1 = false;
        boolean thread2 = false;

        startThreads();
}

Thread1()
```

```
{

        do {

                // entry section
                // wait until thread2 is in its critical section
                while (thread2 == true)
                        ;

                // indicate thread1 entering its critical section
                thread1 = true;

                // critical section

                // exit section
                // indicate thread1 exiting its critical section
                thread1 = false;

                // remainder section

        } while (completed == false)
}

Thread2()
{

        do {

                // entry section
                // wait until thread1 is in its critical section
                while (thread1 == true)
                        ;

                // indicate thread2 entering its critical section
                thread2 = true;

                // critical section

                // exit section
                // indicate thread2 exiting its critical section
                thread2 = false;

                // remainder section

        } while (completed == false)
}
```

The problem arising in the above version is mutual exclusion itself. If threads are preempted (stopped) during flag updation (i.e during current_thread = true) then, both the threads enter their critical section once the preempted thread is restarted, also the same can be observed at the start itself, when both the flags are false.

Third Version of Dekker's Solution - To re-ensure mutual exclusion, it sets the flags before entry section itself.

Source Code 3.108: Dekker's Algorithm 3. Version(Exercise: 23.4)

```
/*
===========================================================================
 Name        : dekker3.c
 Author      : A. Pretschner
 Version     :
 Copyright   : Deitel & Associates, Inc.
 Description : C program to implement Dekker's Algorithm
               for producer-consumer problem.
 ============================================================================
 */
Main()
{

        // flags to indicate if each thread is in
        // queue to enter its critical section
        boolean thread1wantstoenter = false;
        boolean thread2wantstoenter = false;

        startThreads();
}

Thread1()
{

        do {

                thread1wantstoenter = true;

                // entry section
                // wait until thread2 wants to enter
                // its critical section
                while (thread2wantstoenter == true)
                        ;

                // critical section

                // exit section
                // indicate thread1 has completed
                // its critical section
                thread1wantstoenter = false;

                // remainder section

        } while (completed == false)
}

Thread2()
{

        do {

                thread2wantstoenter = true;

                // entry section
                // wait until thread1 wants to enter
                // its critical section
                while (thread1wantstoenter == true)
                        ;
                // critical section

                // exit section
                // indicate thread2 has completed
                // its critical section
                thread2wantstoenter = false;

                // remainder section

        } while (completed == false)
}
```

The problem with this version is deadlock possibility. Both threads could set their flag as true simultaneously and both will wait infinitely later on.

Fourth Version of Dekker's Solution - Uses small time interval to recheck the condition, eliminates deadlock and ensures mutual exclusion as well.

Source Code 3.109: Dekker's Algorithm 4. Version(Exercise: 23.5)

```
/*
============================================================================
 Name        : dekker4.c
 Author      : A. Pretschner
 Version     :
 Copyright   : Deitel & Associates, Inc.
 Description : C program to implement Dekker's Algorithm
               for producer-consumer problem.
 ============================================================================
 */
Main()
{

        // flags to indicate if each thread is in
        // queue to enter its critical section
        boolean thread1wantstoenter = false;
        boolean thread2wantstoenter = false;

        startThreads();
}

Thread1()
{

        do {

                thread1wantstoenter = true;

                while (thread2wantstoenter == true) {

                        // gives access to other thread
                        // wait for random amount of time
                        thread1wantstoenter = false;

                        thread1wantstoenter = true;
                }

                // entry section
                // wait until thread2 wants to enter
                // its critical section

                // critical section

                // exit section
                // indicate thread1 has completed
                // its critical section
                thread1wantstoenter = false;

                // remainder section

        } while (completed == false)
}

Thread2()
{

        do {

                thread2wantstoenter = true;

                while (thread1wantstoenter == true) {

                        // gives access to other thread
                        // wait for random amount of time
                        thread2wantstoenter = false;

                        thread2wantstoenter = true;
                }

                // entry section
                // wait until thread1 wants to enter
                // its critical section

                // critical section

                // exit section
                // indicate thread2 has completed
                // its critical section
                thread2wantstoenter = false;

                // remainder section

        } while (completed == false)
}
```

The problem with this version is the indefinite postponement. Also, random amount of time is erratic depending upon the situation in which the algorithm is being implemented, hence not an acceptable solution in business critical systems.

Dekker's Algorithm : Final and completed Solution - Idea is to use favoured thread notion to determine entry to the critical section. Favoured thread alternates between the thread providing mutual exclusion and avoiding deadlock, indefinite postponement or lockstep synchronization.

Source Code 3.110: Dekker's Algorithm 5. Version(Exercise: 23.6)

```
/*
```

```
============================================================================
 Name        : dekker5.c
 Author      : A. Pretschner
 Version     :
 Copyright   : Deitel & Associates, Inc.
 Description : C program to implement Dekker's Algorithm
               for producer-consumer problem.
 ============================================================================
 */
Main()
{

        // to denote which thread will enter next
        int favouredthread = 1;

        // flags to indicate if each thread is in
        // queue to enter its critical section
        boolean thread1wantstoenter = false;
        boolean thread2wantstoenter = false;

        startThreads();
}

Thread1()
{
        do {

                thread1wantstoenter = true;

                // entry section
                // wait until thread2 wants to enter
                // its critical section
                while (thread2wantstoenter == true) {

                        // if 2nd thread is more favored
                        if (favaouredthread == 2) {

                                // gives access to other thread
                                thread1wantstoenter = false;

                                // wait until this thread is favored
                                while (favouredthread == 2)
                                        ;

                                thread1wantstoenter = true;
                        }
                }

                // critical section

                // favor the 2nd thread
                favouredthread = 2;

                // exit section
                // indicate thread1 has completed
                // its critical section
                thread1wantstoenter = false;

                // remainder section

        } while (completed == false)
}

Thread2()
{

        do {

                thread2wantstoenter = true;

                // entry section
                // wait until thread1 wants to enter
                // its critical section
                while (thread1wantstoenter == true) {

                        // if 1st thread is more favored
                        if (favaouredthread == 1) {

                                // gives access to other thread
                                thread2wantstoenter = false;

                                // wait until this thread is favored
                                while (favouredthread == 1)
                                        ;

                                thread2wantstoenter = true;
                        }
                }

                // critical section

                // favour the 1st thread
                favouredthread = 1;

                // exit section
                // indicate thread2 has completed
                // its critical section
                thread2wantstoenter = false;

                // remainder section

        } while (completed == false)
}
```

This version guarantees a complete solution to the critical solution problem.

Task 23.1: *Program the Dekker's solution into sample programs.*

Submit your answers from Lab 3.3 to Moodle one week after the assignment

Python - Laboratory Training Files

Lab 4.1: Short Introduction into object oriented Python

These exercises include sample programs and easy exercises to learn the language Python. They are not a complete script for the course. Detailed explanations are given in the lecture.
The following exercises and samples are taken Literature:
PYnative: Learn Python with Tutorials, Exercises, and Quizzes `https://pynative.com/`.
What is a Class and Objects in Python?

- Class: The class is a user-defined data structure that binds the data members and methods into a single unit. Class is a blueprint or code template for object creation. Using a class, you can create as many objects as you want.
- Object: An object is an instance of a class. It is a collection of attributes (variables) and methods. We use the object of a class to perform actions. Objects have two characteristics: They have states and behaviors (an object has attributes and methods attached to it). Attributes represent its state, and methods represent its behavior. Using its methods, we can modify its state.

In short, Every object has the following properties.

- Identity: Every object must be uniquely identified.
- State: An object has an attribute that represents a state of an object, and it also reflects the property of an object.
- Behavior: An object has methods that represent its behavior.

Python is an Object-Oriented Programming language, so everything in Python is treated as an object. An object is a real-life entity. It is the collection of various data and functions that operate on those data.
For example, If we design a class based on the states and behaviors of a Person, then States can be represented as instance variables and behaviors as class methods. A real-life example of class and objects.

Class: Person State: Name, Sex, Profession Behavior: Working, Study

Using the above class, we can create multiple objects that depict different states and behavior.

Object 1: Jessa State: Name: Jessa Sex: Female Profession: Software Engineer Behavior: Working: She is working as a software developer at ABC Company Study: She studies 2 hours a day

Object 2: Jon State: Name: Jon Sex: Male Profession: Doctor Behavior: Working: He is working as a doctor Study: He studies 5 hours a day

As you can see, Jessa is female, and she works as a Software engineer. On the other hand, Jon is a male, and he is a lawyer. Here, both objects are created from the same class, but they have different states and behaviors.

Source Code 4.111: To Upper (Exercise: 24.1)

```
class class_name:
    '''This is a docstring. I have created a new class'''
    <statement 1>
    <statement 2>
    .
    .
    <statement N>
```

- class_name: It is the name of the class
- Docstring: It is the first string inside the class and has a brief description of the class. Although not mandatory, this is highly recommended.
- statements: Attributes and methods

Example: Define a class in Python

In this example, we are creating a Person Class with name, sex, and profession instance variables.

Source Code 4.112: To Upper (Exercise: 24.1)

```
class Person:
    def __init__(self, name, sex, profession):
        # data members (instance variables)
        self.name = name
        self.sex = sex
        self.profession = profession

    # Behavior (instance methods)
    def show(self):
        print('Name:', self.name, 'Sex:', self.sex, 'Profession:', self.profession)

    # Behavior (instance methods)
    def work(self):
        print(self.name, 'working as a', self.profession)
```

Create Object of a Class

An object is essential to work with the class attributes. The object is created using the class name. When we create an object of the class, it is called instantiation. The object is also called the instance of a class.

A constructor is a special method used to create and initialize an object of a class. This method is defined in the class. In Python, Object creation is divided into two parts in Object Creation and Object initialization. A constructor is a special method used to create and initialize an object of a class. This method is defined in the class. In Python, Object creation is divided into two parts in Object Creation and Object initialization

> Internally, the `__new__` is the method that creates the object And, using the `__init__()` method we can implement constructor to initialize the object.

```
<object-name> = <class-name>(<arguments>)
jessa = Person('Jessa', 'Female', 'Software Engineer')
```

The complete example:

```
class Person:
    def __init__(self, name, sex, profession):
        # data members (instance variables)
        self.name = name
        self.sex = sex
        self.profession = profession

    # Behavior (instance methods)
    def show(self):
        print('Name:', self.name, 'Sex:', self.sex, 'Profession:', self.profession)

    # Behavior (instance methods)
    def work(self):
        print(self.name, 'working as a', self.profession)

# create object of a class
jessa = Person('Jessa', 'Female', 'Software Engineer')

# call methods
jessa.show()
jessa.work()
```

Class Attributes

When we design a class, we use instance variables and class variables. In Class, attributes can be defined into two parts:

- Instance variables: The instance variables are attributes attached to an instance of a class. We define instance variables in the constructor (the `__init__()` method of a class).
- Class Variables: A class variable is a variable that is declared inside of class, but outside of any instance method or `__init__()` method.

Objects do not share instance attributes. Instead, every object has its copy of the instance attribute and is unique to each object. All instances of a class share the class variables. However, unlike instance variables, the value of a class variable is not varied from object to object. Only one copy of the static variable will be created and shared between all objects of the class.
Accessing properties and assigning values

- An instance attribute can be accessed or modified by using the dot notation: `instance_name.attribute_name`.
- A class variable is accessed or modified using the class name

Example:

```
class Student:
    # class variables
    school_name = 'ABC School'

    # constructor
    def __init__(self, name, age):
        # instance variables
        self.name = name
        self.age = age

s1 = Student("Harry", 12)
# access instance variables
print('Student:', s1.name, s1.age)

# access class variable
print('School name:', Student.school_name)

# Modify instance variables
s1.name = 'Jessa'
s1.age = 14
print('Student:', s1.name, s1.age)
```

```
# Modify class variables
Student.school_name = 'XYZ School'
print('School name:', Student.school_name)
```

Class Methods

In Object-oriented programming, Inside a Class, we can define the following three types of methods.

- Instance method: Used to access or modify the object state. If we use instance variables inside a method, such methods are called instance methods.
- Class method: Used to access or modify the class state. In method implementation, if we use only class variables, then such type of methods we should declare as a class method.
- Static method: It is a general utility method that performs a task in isolation. Inside this method, we don't use instance or class variable because this static method doesn't have access to the class attributes.

Instance methods work on the instance level (object level). For example, if we have two objects created from the student class, They may have different names, marks, roll numbers, etc. Using instance methods, we can access and modify the instance variables. A class method is bound to the class and not the object of the class. It can access only class variables.

Example: Define and call an instance method and class method

Next Example shows one possible implementation:

```
# class methods demo
class Student:
    # class variable
    school_name = 'ABC School'

    # constructor
    def __init__(self, name, age):
        # instance variables
        self.name = name
        self.age = age

    # instance method
    def show(self):
        # access instance variables and class variables
        print('Student:', self.name, self.age, Student.school_name)

    # instance method
    def change_age(self, new_age):
        # modify instance variable
        self.age = new_age

    # class method
    @classmethod
    def modify_school_name(cls, new_name):
        # modify class variable
        cls.school_name = new_name

s1 = Student("Harry", 12)

# call instance methods
s1.show()
s1.change_age(14)

# call class method
Student.modify_school_name('XYZ School')
```

```
# call instance methods
s1.show()
```

Class Naming Convention

Naming conventions are essential in any programming language for better readability. If we give a sensible name, it will save our time and energy later. Writing readable code is one of the guiding principles of the Python language.

We should follow specific rules while we are deciding a name for the class in Python.

- Rule-1: Class names should follow the UpperCaseCamelCase convention
- Rule-2: Exception classes should end in "Error".
- Rule-3: If a class is callable (Calling the class from somewhere), in that case, we can give a class name like a function.
- Rule-4: Python's built-in classes are typically lowercase words

pass Statement in Class

In Python, the pass is a null statement. Therefore, nothing happens when the pass statement is executed.

The pass statement is used to have an empty block in a code because the empty code is not allowed in loops, function definition, class definition. Thus, the pass statement will results in no operation (NOP). Generally, we use it as a placeholder when we do not know what code to write or add code in a future release.

For example, suppose we have a class that is not implemented yet, but we want to implement it in the future, and they cannot have an empty body because the interpreter gives an error. So use the pass statement to construct a body that does nothing.

Example

```
class Demo:
  pass
```

In the above example, we defined class without a body. To avoid errors while executing it, we added the pass statement in the class body.

Object Properties

Every object has properties with it. In other words, we can say that object property is an association between name and value.

For example, a car is an object, and its properties are car color, sunroof, price, manufacture, model, engine, and so on. Here, color is the name and red is the value. Object properties are nothing but instance variables.

Modify Object Properties

Every object has properties associated with them. We can set or modify the object's properties after object initialization by calling the property directly using the dot operator.

```
Obj.PROPERTY = value
```

Example

```
class Fruit:
    def __init__(self, name, color):
        self.name = name
        self.color = color

    def show(self):
        print("Fruit is", self.name, "and Color is", self.color)

# creating object of the class
obj = Fruit("Apple", "red")

# Modifying Object Properties
obj.name = "strawberry"

# calling the instance method using the object obj
```

```
obj.show()
# Output Fruit is strawberry and Color is red
```

Delete object properties

We can delete the object property by using the del keyword. After deleting it, if we try to access it, we will get an error.

```
class Fruit:
    def __init__(self, name, color):
        self.name = name
        self.color = color

    def show(self):
        print("Fruit is", self.name, "and Color is", self.color)

# creating object of the class
obj = Fruit("Apple", "red")

# Deleting Object Properties
del obj.name

# Accessing object properties after deleting
print(obj.name)
# Output: AttributeError: 'Fruit' object has no attribute 'name'
```

In the above example, As we can see, the attribute name has been deleted when we try to print or access that attribute gets an error message.

Delete Objects

In Python, we can also delete the object by using a del keyword. An object can be anything like, class object, list, tuple, set, etc.

Syntax:

```
del object_name
```

Example: Deleting object

```
class Employee:
    depatment = "IT"

    def show(self):
        print("Department is ", self.depatment)

emp = Employee()
emp.show()

# delete object
del emp

# Accessing after delete object
emp.show()

# Output : NameError: name 'emp' is not defined
```

In the above example, we create the object `emp` of the class Employee. After that, using the `del` keyword, we deleted that object.

Lab 4.2: Phyton Exercises: Create a Class with instance attributes

Write a Python program to create a Vehicle class with `max_speed` and `mileage` instance attributes. Create a child class Bus that will inherit all of the variables and methods of the Vehicle class.

Source Code 4.113: (Exercise: 25.1)

```
class Vehicle:
    def __init__(self, name, max_speed, mileage):
        self.name = name
        self.max_speed = max_speed
        self.mileage = mileage
```

Create a Bus class that inherits from the Vehicle class. Give the capacity argument of `Bus.seating_capacity()` a default value of 50. Use the following code for your parent Vehicle class.

Source Code 4.114: (Exercise: 25.2)

```
class Vehicle:
    def __init__(self, name, max_speed, mileage):
        self.name = name
        self.max_speed = max_speed
        self.mileage = mileage

    def seating_capacity(self, capacity):
        return f"The seating capacity of a {self.name} is {capacity} passengers"
```

Define a class attribute”color” with a default value white. I.e., Every Vehicle should be white. Use the following code for this exercise.

Source Code 4.115: (Exercise: 25.3)

```
class Vehicle:

    def __init__(self, name, max_speed, mileage):
        self.name = name
        self.max_speed = max_speed
        self.mileage = mileage

class Bus(Vehicle):
    pass

class Car(Vehicle):
    pass
```

Create a Bus child class that inherits from the Vehicle class. The default fare charge of any vehicle is `seating capacity * 100`. If Vehicle is Bus instance, we need to add an extra 10% on full fare as a maintenance charge. So total fare for bus instance will become the `final amount = total fare + 10\% of the total fare`.
Note: The bus seating capacity is 50. so the final fare amount should be 5500. You need to override the `fare()` method of a Vehicle class in Bus class. Use the following code for your parent Vehicle class. We need to access the parent class from inside a method of a child class.

Source Code 4.116: (Exercise: 25.4)

```
class Vehicle:
    def __init__(self, name, mileage, capacity):
        self.name = name
        self.mileage = mileage
        self.capacity = capacity

    def fare(self):
        return self.capacity * 100

class Bus(Vehicle):
    pass

School_bus = Bus("School Volvo", 12, 50)
```

```
print("Total Bus fare is:", School_bus.fare())
```

Write a program to determine which class a given Bus object belongs to.

Source Code 4.117: (Exercise: 25.5)

```
class Vehicle:
    def __init__(self, name, mileage, capacity):
        self.name = name
        self.mileage = mileage
        self.capacity = capacity

class Bus(Vehicle):
    pass

School_bus = Bus("School Volvo", 12, 50)
```

Determine if `School_bus` is also an instance of the Vehicle class

Source Code 4.118: (Exercise: 25.6)

```
class Vehicle:
    def __init__(self, name, mileage, capacity):
        self.name = name
        self.mileage = mileage
        self.capacity = capacity

class Bus(Vehicle):
    pass

School_bus = Bus("School Volvo", 12, 50)
```

Conclusions about your lab files

Objective: clean up and make an archive of your lab directory

End of the training session

Congratulations. You reached the end of the training session. You now have plenty of working examples you created by yourself, and you can build upon them to create more elaborate things. In this last lab, we will remember what you did and which files you created. You won't keep everything though, as there are lots of things you can easily retrieve again.

Create a lab archive

Go to the directory containing your `~/worksapce` directory:
Now, run a command that will do some clean up and then create an archive with the most important files:

- Python configuration files
- Other source configuration files
- Project examples
- Other custom files

But normally you need no backup, because all configuration files and all necessary source packages are available through out our Moodle Teaching System.