

CH-230-A

Programming in C and C++

C/C++

Tutorial 1

Dr. Kinga Lipskoch

Fall 2019

Who are We?

- ▶ Dr. Kinga Lipskoch, lecturer for Computer Science
- ▶ Dr. Sergey Kosov, lecturer for Data Engineering
- ▶ Dr. Szymon Krupiński, lecturer for Computer Science
- ▶ Lectures are common for all students given by Dr. Kinga Lipskoch
- ▶ For tutorials students are divided into groups
- ▶ Each lecturer is responsible for own tutorial group

Course Goals

- ▶ Learn the basic and some advanced aspects of procedural and object-oriented programming
- ▶ Learn the details of the C and C++ programming languages
- ▶ Write, run, test, debug programs using C and C++

Course Details

- ▶ Every week will consist of
 - ▶ 2 tutorials: Tuesdays, 8:15 - 11:00
 - ▶ 1 lecture: Thursdays, 11:15 - 12:30
- ▶ During each tutorial you will have to solve a programming assignment sheet (consisting of multiple exercises) related to the corresponding lecture
- ▶ **Presence assignments** need to be submitted during the tutorial time
- ▶ **Other assignments** can be submitted before next Monday at 23:00
- ▶ Help session may be offered by TAs before the deadline: Sundays, 19:00 - 21:00, Lecture Hall, Research 1

Course Resources

- ▶ Homepage of course: https://grader.eecs.jacobs-university.de/courses/ch_230_a/2019_2/
Slides, assignment sheets and practice sheets will be posted there
- ▶ Program assignments will be received during the tutorial
Presence exercises have to be submitted during the tutorial
- ▶ Offline questions: Office hours on Mondays 10:00 - 12:00 or ask for another appointment individually
- ▶ Do not hesitate, and do not wait until you are left too much behind

Literature

Some example textbooks for C and C++ are:

- ▶ Brian Kernighan, Dennis Ritchie:
The C Programming Language
- ▶ Steve Oualline:
Practical C Programming
- ▶ Bruce Eckel:
Thinking in C++: Introduction to Standard C++
- ▶ Bruce Eckel, Chuck Allison:
Thinking in C++: Practical Programming
- ▶ Bjarne Stroustrup:
The C++ Programming Language
- ▶ Michael Dawson:
Beginning C++ Through Game Programming

Beyond this Programming Course

- ▶ Programming skills might be one of the key career parameters
- ▶ Programming itself is a vast and multi-faceted activity - mixture of right mental attitude, skills and experience
- ▶ This course is only there to get you started
- ▶ Internet has a plenty of resources and cookbook recipees - but good initial skills are necessary to profit from them
- ▶ Several online portals allow you to hone your skills by solving problems and collect "badges" or have "reputation" score
 - ▶ <https://leetcode.com>
 - ▶ <https://hackerrank.com>
 - ▶ <https://stackoverflow.com>
 - ▶ <https://www.spoj.com>
 - ▶ ...

Submission of Solutions

- ▶ Use Grader
<https://grader.eecs.jacobs-university.de/>
with your CampusNet credentials
- ▶ Submit *.c or *.cpp and *.h files depending on the problem
(which language to use will be fixed for each problem)
- ▶ Pay attention to deadlines and submit before

Grader not Publicly Visible

- ▶ You can access Grader from campus without any additional connection or software
- ▶ To access Grader from outside of campus you need to use a VPN (Virtual Private Network) connection
- ▶ Instructions from the Jacobs IRC IT team on how to install a VPN client:

<https://teamwork.jacobs-university.de/display/ircit/VPN+Access>

Grading

- ▶ Each problem will be graded (percentages)
- ▶ To be able to attend the final exam you need to have $\geq 50\%$ as average over all assignments
- ▶ In the (written) final exam you will be asked to solve exercises similar to ones in the assignments
- ▶ The final exam will take place at the end of the semester scheduled by the Student Records Office

Grading Criteria for Assignments

- ▶ Assignments are graded by the TAs
- ▶ Not just the solution counts, but programming style and form
- ▶ TAs will grade according to document
https://grader.eecs.jacobs-university.de/courses/ch_230_a/2019_2/Grading-Criteria-C-C++.pdf
- ▶ They will use rules to grade your solutions
- ▶ Two types of assignments:
 - ▶ **Automatically graded problems with testcases only** - for example, 10 uploaded testcases and your solution passes only 7 then your grade will be 70%
 - ▶ **Manually graded problems** - with feedback from TAs and a grade between 0% and 100%

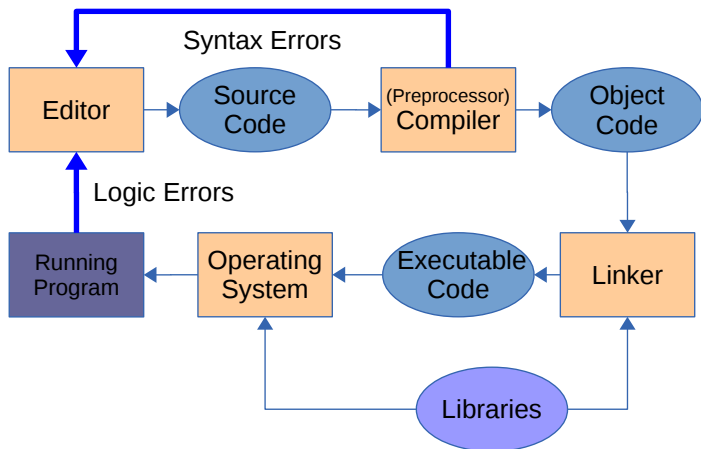
Missing Homework, Quizzes, Exams according to AP

- ▶ https://www.jacobs-university.de/sites/default/files/bachelor_policies_v3.pdf (pages 15 - 16)
- ▶ Illness must be documented with a sick certificate
- ▶ Sick certificates need to verify the date and time of the in-person visit occasioned the confirmation that the student is unable to fulfill his/her academic obligation (either attend class/lab or take the examination)
- ▶ Sick certificates and documentation must be submitted to Registrar Services by the third calendar day from the beginning of illness/of the emergency
- ▶ These three days include the first day of the illness/of the emergency
- ▶ If the third calendar day is a Saturday, Sunday or a public holiday, the deadline is extended to the next working day.
- ▶ Predated or backdated sick certificates, i.e., when the visit to the physician takes place outside of the documented sickness period will be accepted provided that the visit to the physician precedes or follows the period of illness by no more than one working day
- ▶ Regardless of the reason for their absence, students must inform the IoR before the beginning of the examination or class/lab session that they will not be able to attend
- ▶ The day after the excuse ends, students must contact the IoR
- ▶ Failure to complete a module will lead to a continued incomplete of the module until the missing requirements are fulfilled or definitively failed

Prerequisite for Algorithms and Data Structure

- ▶ This course is a prerequisite for the second semester course "Algorithms and Data Structures" (ADS)
- ▶ If you fail or do not take the exam for this course you will not be allowed to proceed with the ADS course
- ▶ If you fail the first exam (December) you can take the so-called make-up exam (January)

Program Development Cycle



Integrated Development Environment (IDE)

- ▶ You can use the editor of your choice and compile from the terminal
- ▶ For C: `gcc -Wall -o executable program.c`
- ▶ For C++: `g++ -Wall -o executable program.cpp`
- ▶ If you do not know any of the above, you can use [Code::Blocks](#) or [Visual Studio Code](#)

IDE Installation

- ▶ Alternative 1: [Code::Blocks](#)
 - ▶ Download and install Code::Blocks from:
<http://codeblocks.org/downloads/26>
 - ▶ If you are a Windows user download (contains IDE + compiler)
[codeblocks-17.12mingw-setup.exe](#)
- ▶ Alternative 2: [Visual Studio Code](#)
 - ▶ Download and install Visual Studio Code https://code.visualstudio.com/download?wt.mc_id=DX_841432
 - ▶ Download and install compiler Mingw-w64
<http://mingw-w64.org/doku.php/download/mingw-builds>
 - ▶ Assuming that you installed Mingw-w64 to this path:
C:\Mingw-w64, Windows users have to add to the environment variable PATH in the following: C:\Mingw-w64\mingw32\bin\
- ▶ Alternative 3: [Visual Studio Community 2019](#)
 - ▶ Only for Windows users
 - ▶ Download and install <https://visualstudio.microsoft.com/thank-you-downloading-visual-studio/?sku=Community&rel=16>

Different Compilers Behave Differently

- ▶ Different compilers behave differently
- ▶ Even different versions of the same compiler may deliver different results in terms of the compilation process
- ▶ Make sure that your solution runs without warnings or errors on Grader
- ▶ If errors of warning appear, you can fix them and resubmit the solution
- ▶ The Grader server runs gcc and g++ version 8.3.0

About C

- ▶ Widely used general purpose language
- ▶ Advantages: small, efficient, portable, structured
- ▶ Disadvantages: not user-friendly
- ▶ C is an imperative language
- ▶ You will find many of its characteristics in other imperative languages, such as Pascal or Fortran, but also in scripting languages such as Perl, PHP, Python, etc.

Imperative Languages

- ▶ Computation is described in terms of:
 - ▶ State (variables)
 - ▶ Operations to change this state (assignments, loops, etc.)
- ▶ Imperative: first do this, then do that, ...
- ▶ There exists other approaches (functional programming, logic programming, object-oriented programming, etc.)

The First Program

- ▶ A true classic: Hello world
- ▶ Open editor → New file
- ▶ Type text below, then save as `hello.c`

```
1  /* This is my first C program */
2  #include <stdio.h>
3
4  int main() {
5      printf("Hello world\n");
6      return 0;
7  }
```

The printf Library Function

- ▶ `printf` is a library function used to output data
- ▶ To use `printf`, the header file `stdio.h` has to be included
- ▶ `stdio` stands for Standard I/O
- ▶ `stdio` contains the specification of many general purpose functions for I/O
- ▶ `printf` is a very rich and powerful function
- ▶ Basic use: printing a sequence of characters
- ▶ The following line calls the `printf` function
`printf("Hello world\n");`
- ▶ The sequence of characters is called string
- ▶ The sequence is surrounded by quotes

Basic Data Types of C

Data type	C identifier
Character	<code>char</code>
Integer number	<code>int</code>
Floating point number	<code>float</code>
Double precision number	<code>double</code>
No type	<code>void</code>

Moreover there exist some modifiers that can be applied to the basic data types

Formatting Specification (1)

- ▶ Specify the type of the data to be printed

```
1      int a = 45;  
2      printf("The value is %d\n", a);
```

- ▶ This will print the following:
The value is 45
- ▶ Each formatting specification must be matched by a parameter
- ▶ To specify wrong control strings is another common error in C programs

Formatting Specification (2)

Formatting specification starts with a % character

Conversion

Meaning

%c	Single character
%d or %i	Signed decimal integer
%f	Floating point (decimal notation)
%e	Floating point (exponential notation)
%lf	Double (decimal notation)
%s	String
%%	print the percent sign itself
%p	print address of pointer

CH-230-A

Programming in C and C++

C/C++

Tutorial 2

Dr. Kinga Lipskoch

Fall 2019

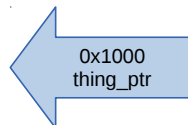
Pointers: Introduction (1)

- ▶ Every data is stored in memory
- ▶ The memory is organized as a sequence of increasingly numbered cells
 - ▶ The number of a cell is its address
 - ▶ The address is determined when the variable is declared



0x1000

A thing



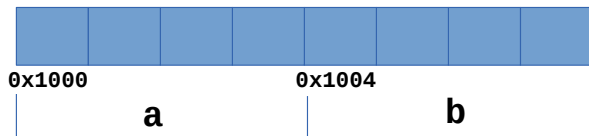
A pointer

Pointer Comparison

- ▶ You can compare pointers and data to street addresses and houses
- ▶ Houses may vary in size, while the address which points to the house is approximately of the same size and small
- ▶ More than one pointer may point to the same thing

Pointers: Introduction (2)

- ▶ Consider the following declaration
`int a = 145, b = 98;`
- ▶ Supposing that an `int` occupies 4 bytes the following situation could arise



The address of `a` is `0x1000` and the address of `b` is `0x1004`.

Pointers

- ▶ A pointer is a variable which stores the address of another variable
- ▶ In the previous example, a pointer to a would store 0x1000, as that is its address
- ▶ Pointers are variables: thus they can be read, written and so on
- ▶ Being variables, they are stored in memory and they have an address as well

Pointers: Declaration and Initialization

- ▶ To declare a pointer, it is necessary to specify the type of the pointed object

```
int *thing_ptr;
```

- ▶ In this case `thing_ptr` will hold the address of an `int`
- ▶ The operator `&` returns the address of a variable
`thing_ptr = &a;`

Do We Need Pointers?

- ▶ Pointers are a critical aspect of C
- ▶ Extremely powerful, allow to solve quickly, efficiently and shortly difficult problems
- ▶ Extremely powerful: if you misuse them you can compromise the integrity of your program
 - ▶ it is the general source of crashes of programs
- ▶ Students usually do not like them, because the different meanings of * is confusing
- ▶ You cannot claim you know C if you do not master them

Managing a Variable via a Pointer

- ▶ The following pieces of code do the same

```
1      int a = 10;  
2      a = a + 5;
```

```
1      int a = 10;  
2      int *ptr;  
3      ptr = &a;  
4      *ptr = *ptr + 5;
```

- ▶ Note the different meanings of *:
 - ▶ Declaration: * declares ptr as pointer to integer
 - ▶ Next line: * dereference (or content-) operator

The Dereference Operator (The Content Operator)

- ▶ The `*` operator allows you to access the content of the pointed object (content-operator)
- ▶ In the previous example we do not increase the pointer, but the content of the pointed object: `a`
- ▶ Note that

```
1      int *ptr;  
2      ptr = &a;  
3      ptr = ptr + 5;
```

is accepted by the compiler but has a completely different meaning!

(the pointer itself will be increased by the size of 5 integers meaning $5 * 4 = 20$ bytes)

An Example

```
1      #include <stdio.h>
2      int main() {
3          int a = 7;
4          int *ptr;
5          ptr = &a;
6
7          printf("Address of a: %p\n", ptr);
8          printf("Value of a: %d\n", *ptr);
9          return 0;
10     }
```

Simple Use of Pointers

```
1 #include <stdio.h>
2 int main() {
3     int  thing_var; /* def. var. for a thing */
4     int *thing_ptr; /* def. pointer to a thing */
5     thing_var = 2; /* assign a value to a thing */
6     printf("Thing %d\n", thing_var);
7     thing_ptr = &thing_var; /* make the pointer
8                               point to thing_var */
9     *thing_ptr = 3; /* thing_ptr points to
10                    thing_var, so thing_var changes to 3 */
11    printf("Thing %d\n", thing_var);
12    printf("Thing %d\n", *thing_ptr);
13    /* another way */
14    return 0;
15 }
```

Comments

- ▶ It is obvious that it is necessary to specify the type of the pointed variable
 - ▶ Previous example: how many bytes for the integer: 2 of 4?
Knowing the type the answer can be given
- ▶ The following does not work

```
1      int a = 45;
2      int *int_ptr = &a;
3      char *char_ptr;
4      char_ptr = int_ptr; /* WRONG */
```

Misuse of Pointers

- ▶ `*thing_var`
 - ▶ illegal: Get the object pointed to by the variable `thing_var`, `thing_var` is not a pointer, so operation is invalid
- ▶ `&thing_ptr`
 - ▶ legal, but strange. `thing_ptr` is a pointer: Get address of pointer to a object, result is a pointer to a pointer

Arrays (1)

- ▶ Arrays (or vectors) can be defined in different ways:
 - ▶ An indexed variable
 - ▶ A contiguous memory area holding elements of the same type
- ▶ Arrays are useful if you have to deal with many variables of the same type
 - ▶ Logically related to each other

The Need for Arrays

- ▶ Write a program that reads all the students grades and prints the maximum and minimum, the mean and the variance
 - ▶ Assume at most 200 students
- ▶ Should we declare 200 float variables?
- ▶ From a logical point of view, all the grades belong to the same set of data

Arrays (2)

- ▶ An array has a common name and a shared type
- ▶ An array has a dimension
- ▶ Elements are numbered sequentially
- ▶ Elements can be accessed (read/write) in an array by using their position (also called index or subscript)

Arrays in C

- ▶ In C you declare an array by specifying the dimension between square brackets

```
int data[4];
```

- ▶ The former is an array of 4 elements
- ▶ The first one is at position 0, the last one is at position 3

```
1      data[0] = 5;  
2      data[1] = 7;  
3      data[2] = 3;  
4      data[3] = 8;
```

Strings

- ▶ Strings are sequences of characters
- ▶ Many languages have strings as a built-in type, but C does not
- ▶ C has character arrays, with the special character `'\0'` to indicate the end of the string

```
1  #include <stdio.h>
2  int main() {
3      char name[30] = "Hello World";
4      printf("%s\n", name);
5      name[5] = '\0';
6      printf("%s\n", name);
7      return 0;
8  }
```

Reading Strings from the Keyboard

Standard function `fgets` may be used to read a string from the keyboard.

```
fgets(name, sizeof(name), stdin);
```

- ▶ `name`
 - ▶ the name of the character array
- ▶ `sizeof(name)`
 - ▶ maximum number of characters to read
- ▶ `stdin`
 - ▶ is the file/stream to read from
 `stdin` means standard input (keyboard)

Example Program

```
1  #include <string.h>
2  #include <stdio.h>
3  int main() {
4      char line[100];
5      printf("Enter a line: ");
6      fgets(line, sizeof(line), stdin);
7      printf("You entered: %s\n", line);
8      printf("The length of the line is: %s\n",
9             strlen(line));
10     return 0;
11 }
```

This example contains three errors, one syntax error, and two logical errors. One logical error might crash your program, the other will just count the number of characters you have put wrong.

Reading Numbers with fgets and sscanf

Use fgets to read a line of input and sscanf to convert the text into numbers

```
1  #include <stdio.h>
2  int main() {
3      char line[100];
4      int value;
5      printf("Enter a value: ");
6      fgets(line, sizeof(line), stdin);
7      sscanf(line, "%d", &value);
8      printf("You entered: %d\n", value);
9      return 0;
10 }
```

Reading Numbers with scanf

```
1  #include <stdio.h>
2  int main() {
3      int value;
4      printf("Enter a value: ");
5      scanf("%d", &value);
6      printf("You entered: %d\n", value);
7      return 0;
8  }
```

Problems with scanf (1)

```
1  #include <stdio.h>
2  int main() {
3      int nr;
4      char ch;
5      scanf("%d", &nr);
6      // getchar();
7      scanf("%c", &ch);
8      printf("nr=%d, ch=%c\n", nr, ch);
9      return 0;
10 }
```

Assuming that you want to input 12 for `nr` and `'a'` for `ch` then you will experience `nr=12` and `c='\n'` which is not correct, therefore using `scanf` is not advisable in this context

One solution to solve this is to remove the comments in line 6

Problems with `scanf` (2)

- ▶ "The function `scanf` works like `printf`, except that `scanf` reads numbers instead of writing them. `scanf` provides a simple and easy way of reading numbers that almost never works. The function `scanf` is notorious for its poor end-of-line handling, which makes `scanf` useless for all but an expert."
- ▶ "However we have found a simple way to get around the deficiencies of `scanf`, we do not use it."

From: Steve Oualline, Practical C Programming, O'Reilly, 3rd edition

CH-230-A

Programming in C and C++

C/C++

Tutorial 3

Dr. Kinga Lipskoch

Fall 2019

for: Example Revised

```
1 #include <stdio.h>
2 int main() {
3     int idx, n, sum = 0;
4     printf("Type a positive number ");
5     scanf("%d", &n);
6     for (idx = 1; idx <= n; idx++) {
7         printf("Processing %d..\n", idx);
8         sum += idx;
9     }
10    printf("The sum is %d\n", sum);
11    return 0;
12 }
```

Boolean Operators and if

```
1  for (n = 0; n < 3; n++) {  
2      for (i = 0; i < 10; i++) {  
3          if (n < 1 && i == 0) {  
4              printf("n is < 1, i is 0\n");  
5          }  
6          if (n == 2 || i == 5) {  
7              printf("HERE n: %d i:%d\n", n, i);  
8          }  
9          else {  
10             printf("n:%d, i:%d\n", n, i);  
11         }  
12     }  
13 }
```

Easier or Harder to Read?

```
1 for (n = 0; n < 3; n++)
2   for (i = 0; i < 10; i++) {
3     if (n < 1 && i == 0) {
4       printf("n is < 1, i is 0\n"); }
5     if (n == 2 || i == 5) {
6       printf("HERE n: %d i:%d\n", n, i); }
7     else {
8       printf("n:%d, i:%d\n", n, i); }}
```

Iterations: `do ... while`

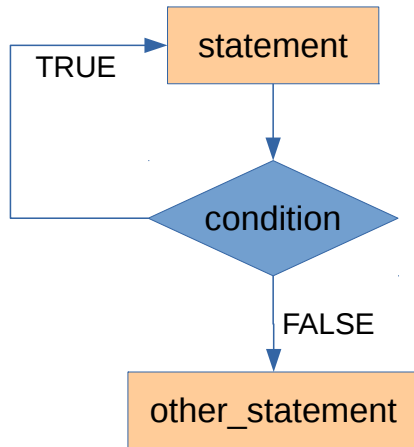
- ▶ General syntax:

```
1  do
2      statement;
3  while (condition);
```

```
1  do {
2      statement1;
3      statement2;
4  } while (condition);
```

- ▶ In this case the end condition is evaluated at the end
- ▶ The body is always executed at least once

do ... while: Flow Chart



do ... while: Example

```
1 #include <stdio.h>
2 int main() {
3     int n, sum = 0;
4     do {
5         printf("Enter number (<0 ends)");
6         scanf("%d", &n);
7         sum += n;
8     } while (n >= 0);
9     sum -= n; /* Remove last negative value */
10    printf("The sum is %d\n", sum);
11    return 0;
12 }
```

Because the statement is executed first and the condition is checked second in do while loop. So, -5 will also be added here.

Jumping Out of a Cycle: `break`

- ▶ The keyword `break` allows to jump out of a cycle when executed
- ▶ We have already seen this while discussing `switch`

```
1 int num, i = 0;
2 scanf("%d", &num);
3 while (i < 50) {
4     printf("%d\n", i);
5     i++;
6     if (i == num)
7         break;
8 }
```


Jumping Out of a Cycle: `continue`

- ▶ `continue` jumps to the expression governing the cycle
- ▶ The expression is evaluated again and so on

```
1 char c;  
2 /* code assumes that the input is  
3    provided in one line like:  
4    "abf23cdef" followed by enter */  
5 while ((c = getchar()) != '\n') {  
6     // ignore the letter b  
7     if (c == 'b')  
8         continue;  
9     printf("%c", c);  
10 }
```

Jumping Out of a Cycle

- ▶ Do not abuse `break` and `continue`
- ▶ You can always obtain the same result without using them
 - ▶ This at the price of longer coding
- ▶ By using them your code gets more difficult to read
- ▶ When you are experienced you will master their use
 - ▶ Meanwhile, learn the basics

Iterations: General Comments

- ▶ Inside the body of the loop you must insert an instruction that can cause the condition to become false
- ▶ If you do not do that, your program will fall into an infinite loop and will be unable to stop (Press Ctrl-C to stop such a program)
- ▶ `do ... while` is far less used than `while` and `for`
- ▶ The same constructs are provided in the majority of other programming languages

Arrays in C

- ▶ See first lecture for introduction
- ▶ In C you declare an array by specifying the size between square brackets
- ▶ Example: `int my_array[50];`
- ▶ The former is an array of 50 elements
- ▶ The first element is at position 0, the last one is at position 49

Accessing an Array in C

- ▶ To write an element, you specify its position

```
1  my_array[2] = 34;  
2  my_array[0] = my_array[2];
```

- ▶ Pay attention: if you specify a position outside the limit, you will have unpredictable results segmentation fault, bus error, etc.
- ▶ And obviously wrong
- ▶ Note the different meaning of brackets
- ▶ Brackets in declaration describe the dimension, while in program they are the index operator

Arrays with Initialization

- ▶ C allows also the following declarations:

```
1  int first_array[]    = {12, 45, 7, 34};  
2  int second_array[4] = {1, 4, 16, 64};  
3  int third_array[4]  = {0, 0};
```

- ▶ It is not possible to specify more values than the declared size of the array
- ▶ The following is wrong:

```
1  int wrong[3] = {1, 2, 3, 4};
```

Typical Structure of a C Program

```
1  #include <stdio.h>
2  int rect_area(int length, int width);
3  float b_func(int a, int b);
4  int main() {
5      ...
6      c = rect_area(5, 7);
7      b_func(11, 6);
8      return 0;
9  }
10 int rect_area(int length, int width) {
11     ... /* do some operations */
12     return area;
13 }
14 float b_func(int a, int b) {
15     ... /* do some operations */
16     return c;
17 }
```

Calling a Function

- ▶ To call a function you insert its name
 - ▶ Function call is a statement
- ▶ You have to provide suitable parameters
 - ▶ Number and type of parameters must match function declaration
- ▶ The result of a function can be ignored

An Example

```
1 #include <math.h>
2 #include <stdio.h>
3 int main() {
4     double number, root;
5     scanf("%lf", &number);
6     if (number >= 0) {
7         root = sqrt(number);
8         printf("Square root is %f\n", root);
9         sqrt(number); /* useless but legal */
10        /* What can I print now? */
11    }
12    else
13        printf("Cannot calc square root\n");
14    return 0;
15 }
```

```
gcc -Wall -lm -o example example.c
```

Finding the Maximum Value in an Array

```
1  /*  v[100]: array of ints
2      dim: number of elements in v
3      Returns the greatest element in v
4  */
5  int findmax(int v[100], int dim) {
6      int i, max;
7
8      max = v[0];
9      for (i = 1; i < dim; i++) {
10         if (v[i] > max)
11             max = v[i];
12     }
13     return max;
14 }
```

Looking for an Element

```
1  /*  v[100]: array of ints
2      dim: number of elements in v
3      t: element to find
4      Returns -1 if t is not present in v or
5          its position in v
6  */
7  int find_element(int v[100], int dim, int t) {
8      int i;
9      for (i = 0; i < dim; i++) {
10         if (v[i] == t)
11             return i;
12     }
13     return -1;
14 }
```

Flow of Execution

```
1  #include <stdio.h>
2
3  int main() {
4      int array[] = {2, 4, 8, 16, 32};
5      int result;
6
7      result = find_element(array, 5, 37);
8      if (result == -1)
9          printf("37 is not present\n");
10
11     return 0;
12 }
```

Pointers and Address Arithmetic

- ▶ The arithmetic operators for sum and difference (+, -, ++, --, etc) can be applied also to pointers
 - ▶ After all a pointer stores an address, which is an integer
- ▶ These operators are subject to the "address arithmetic".
- ▶ Increasing a pointer means that the pointer will point to the following element
 - ▶ You can also add a number other than 1
- ▶ From a logic point of view the pointer is increased by one. From a physical point of view, the increment depends on the size of the pointed type

Address Arithmetic: Example (1)

```
1 int main() {
2     char a_string[] = "This is a string\0";
3     char *p;
4     int count = 0;
5     printf("The string: %s\n", a_string);
6     for (p = &a_string[0]; *p != '\0'; p++)
7         count++;
8     printf("The string has %d chars.\n", count);
9     p--;
10    printf("Printing the reverse string: ");
11    while (count > 0) {
12        printf("%c", *p);
13        p--;
14        count--;
15    }
16    printf("\n");
17    return 0;
18 }
```

Address Arithmetic: Example (2)

```
1 int main() {
2     char a_string[] = "This is a string\0";
3     char *p;
4     int count = 0;
5     printf("The string: %s\n", a_string);
6     p = a_string;
7     while (*p != '\0') {
8         p++;
9         count++;
10    }
11    printf("The string has %d characters.\n", count);
12    printf("Printing the reverse string: ");
13    p--;
14    while (count > 0) {
15        printf("%c", *p);
16        p--;
17        count--;
18    }
19    printf("\n");
20    return 0;
21 }
```

Increasing a Pointer will Increase the Memory Address Depending on the Size of Type

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main() {
4     char ch_arr[2] = {'A', 'B'};
5     char *ch_ptr;
6     float f_arr[2] = {1.1, 2.2};
7     float *f_ptr;
8
9     ch_ptr = &ch_arr[0];           /* same as ch_ptr = ch_arr */
10    printf("%p\n", ch_ptr);         /* address of 1st elem */
11    ch_ptr++;                       /* increase pointer */
12    printf("%p\n", ch_ptr);         /* address of 2nd elem */
13    printf("%c\n", *ch_ptr);        /* content of 2nd elem */
14
15    f_ptr = f_arr;                  /* same as &f_arr[0] */
16    printf("%p\n", f_ptr);          /* address of 1st elem */
17
18    f_ptr++;                        /* increase pointer */
19    printf("%p\n", f_ptr);          /* address of 2nd elem */
20    printf("%f\n", *f_ptr);         /* content of 2nd elem */
21    return 0;
22 }
```


CH-230-A

Programming in C and C++

C/C++

Tutorial 4

Dr. Kinga Lipskoch

Fall 2019

Passing Arrays to Functions

- ▶ An array does not store its size
- ▶ This has to be provided as a parameter, or by making assumptions on the contents of the array (like for strings)
- ▶ The name of an array is a pointer to the first element of the array, i.e., when an array is passed to a function, a copy of the address of the first element is given
- ▶ Modifications to the elements are seen outside
- ▶ Modifications to the array are not seen outside
- ▶ Can you explain why?

Passing Arrays to Functions: Example

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  void strange_function(int v[], int dim) {
4      int i;
5      for (i = 0; i < dim; i++)
6          v[i] = 287;
7      // v = (int *) malloc(sizeof(int) * 1000);
8  }
9  int main() {
10     int array[] = {1, 2, 9, 16};
11     int *p = &array[0];
12     strange_function(array, 4);
13     printf("%d %p %p\n", array[0], p, array);
14     return 0;
15 }
```

Dynamic Memory Allocation

- ▶ What if we do not know the dimension of the array while coding?
- ▶ Dynamic memory allocation allows you to solve this problem
 - ▶ And many others
 - ▶ But can also cause a lot of troubles if you misuse it

Pointers and Arrays

There is a strong relation between pointers and arrays

- ▶ Indeed an array is nothing but a pointer to the first element in the sequence
- ▶ We are looking at this in detail

Specifying the Dimension on the Fly

To specify the dimension on the fly you can use the `malloc()` function defined in the header file `stdlib.h`

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main() {
4     int *dyn_array, how_many, i;
5     printf("How many elements? ");
6     scanf("%d", &how_many);
7     dyn_array =
8         (int*) malloc(sizeof(int) * how_many);
9     if (dyn_array == NULL)
10         exit(1);
11     for (i = 0 ; i < how_many; i++) {
12         printf("\nInput number %d:", i);
13         scanf("%d", &dyn_array[i]);
14     } return 0;
15 }
```

The malloc() Function (1)

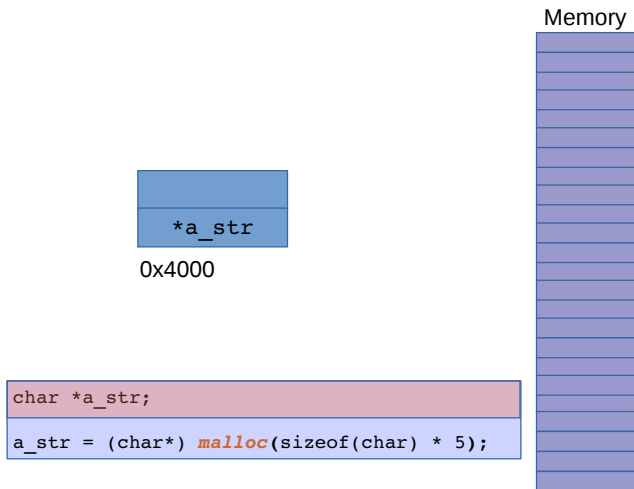
- ▶ `void * malloc(unsigned int);`
- ▶ `malloc` reserves a chunk of memory
- ▶ The parameter specifies how many bytes are requested
- ▶ `malloc` returns a pointer to the first byte of such a sequence
- ▶ The returned pointer must be forced (cast) to the required type

The malloc() Function (2)

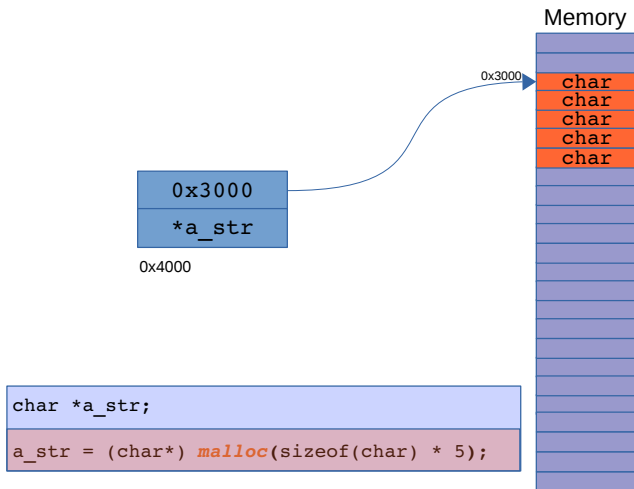
```
1 pointer    = (cast) malloc(number of bytes);  
2  
3  
4 char* a_str;  
5 a_str = (char*) malloc(sizeof(char) * how_many);
```

- ▶ malloc returns a `void *` pointer (i.e., a generic pointer) and this is assigned to a non `void *` pointer
- ▶ If you omit the casting you will get a warning concerning a possible incorrect assignment

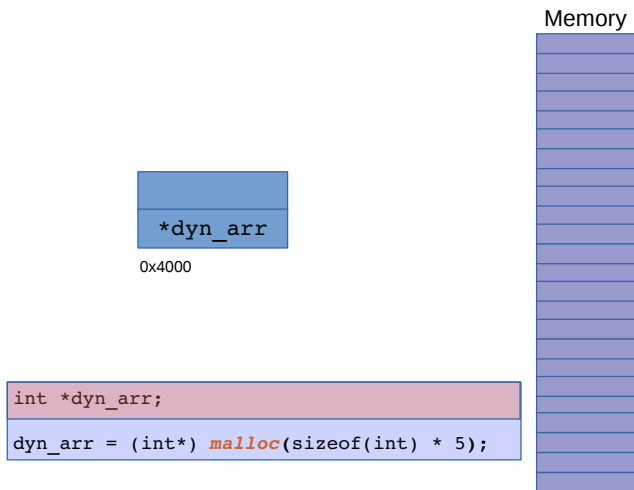
Dynamically Allocating Space for an Array of `char`



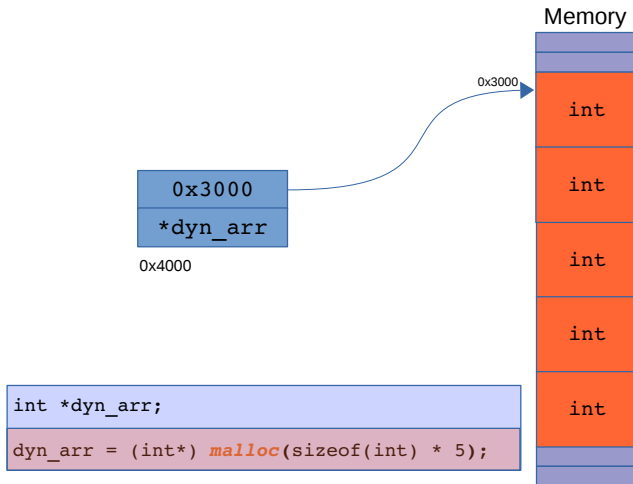
Dynamically Allocating Space for an Array of `char`



Dynamically Allocating Space for an Array of `int`



Dynamically Allocating Space for an Array of `int`



malloc() and free()

- ▶ All the memory you reserve via `malloc`, must be released by using the `free` function
- ▶ If you keep reserving memory without freeing, you will run out of memory

```
1  float *ptr;  
2  int number;  
3  ...  
4  ptr = (float*) malloc(sizeof(float) *  
    number);  
5  ...  
6  free(ptr);
```

Rules for `malloc()` and `free()`

- ▶ The following points are up to you (the compiler does not perform any control)
 1. Always check if `malloc` returned a valid pointer (i.e., not `NULL`)
 2. Free allocated memory just once
 3. Free only dynamically allocated memory
- ▶ Not following these rules will cause endless troubles
- ▶ `sizeof()` is compile time operator, it does not work on allocated memory

Review: Pointers, Arrays, Values

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main() {
4     int length[2] = {7, 9};
5     int *ptr1, *ptr2; int n1, n2;
6     ptr1 = &length[0];
7     // &length[0] is pointer to first elem
8     ptr2 = length;
9     // length is pointer to first elem therefore
10    // same as above
11    n1 = length[0];
12    // length[0] is value
13    n2 = *ptr2;
14    // *ptr2 is value therefore same as above
15    printf("ptr1: %p, ptr2: %p\n", ptr1, ptr2);
16    printf("n1: %d, n2: %d\n", n1, n2);
17    return 0;
18 }
```

Multi-dimensional Arrays

- ▶ It is possible to define multi-dimensional arrays
 - ▶ Mostly used are bidimensional arrays, i.e., tables or matrices
- ▶ As for arrays, to access an element it is necessary to provide an index for each dimension
 - ▶ Think of matrices in mathematics

Multi-dimensional Arrays in C

- ▶ It is necessary to specify the size of each dimension
 - ▶ Dimensions must be constants
 - ▶ In each dimension the first element is at position 0

```
1 int matrix[10][20];    /* 10 rows, 20 cols */
2 float cube[5][5][5];  /* 125 elements */
```

- ▶ Every index goes between brackets

```
1 matrix[0][0] = 5;
```

Multi-dimensional Arrays in C: Example

```
1 #include <stdio.h>
2 int main() {
3     int table[50][50];
4     int i, j, row, col;
5     scanf("%d", &row);
6     scanf("%d", &col);
7     for (i = 0; i < row; i++)
8         for (j = 0; j < col; j++)
9             table[i][j] = i * j;
10    for (i = 0; i < row; i++)
11    {
12        for (j = 0; j < col; j++)
13            printf("%d ", table[i][j]);
14        printf("\n");
15    }
16    return 0;
17 }
```

The main Function (1)

- ▶ Can return an `int` to the operating system
 - ▶ Program exit code (can be omitted)
 - ▶ print exit code in shell: `$> echo $?`
- ▶ Can accept two parameters:
 - ▶ An integer (usually called `argc`)
 - ▶ A vector of strings (usually called `argv`)
 - ▶ `argc` specifies how many strings contains `argv`

The main Function (2)

```
1 #include <stdio.h>
2 int main(int argc, char *argv[]) {
3     int i;
4     for (i = 1; i < argc; i++)
5         printf("%d %s\n", i, argv[i]);
6     return 0;
7 }
```

- ▶ Compile it and call the executable paramscounter
- ▶ Execute it as follows:
\$> ./paramscounter first what this
- ▶ It will print first, what and this, one word per line
- ▶ Note that argc is always greater or equal than one
- ▶ The first parameter is the program's name

CH-230-A

Programming in C and C++

C/C++

Tutorial 5

Dr. Kinga Lipskoch

Fall 2019

The const Keyword

- ▶ The modifier `const` can be applied to variable declarations
- ▶ It states that the variable cannot be changed
 - ▶ i.e., it is not a variable but a constant
- ▶ When applied to arrays it means that the elements cannot be changed

const Examples

```
1  const double e = 2.71828182845905;  
2  const char str[] = "Hello world";  
3  e = 3;           /* error */  
4  str[0] = 'h';    /* error */
```

- ▶ You can also use `#define` of the preprocessor
- ▶ But defines do not have type checking, while constants do

More const Examples

- ▶ `const char *text = "Hello";`
 - ▶ Does not mean that the variable `text` is constant
 - ▶ The data pointed to by `text` is a constant
 - ▶ While the data cannot be changed, the pointer can be changed
- ▶ `char *const name = "Test";`
 - ▶ `name` is a constant pointer
 - ▶ While the pointer is constant, the data the pointer points to may be changed
- ▶ `const char *const title = "Title";`
 - ▶ Neither the pointer nor the data may be changed

Dealing with Big Projects

- ▶ Functions are a first step to break big programs in small logical units
- ▶ A further step consists in breaking the source into many files
 - ▶ Smaller files are easy to handle
 - ▶ Objects sharing a context can be put together and easily reused
- ▶ C allows to put together separately compiled files to have one executable

Declarations and Definitions

- ▶ **Declaration:** introduces an object. After declaration the object can be used
 - ▶ Example: functions' prototypes
- ▶ **Definition:** specifies the structure of an object
 - ▶ Example: function definition
- ▶ Declarations can appear many times, definitions just once

Building from Multiple Sources

- ▶ C compilers can compile multiple sources files into one executable
- ▶ For every declaration there must be one definition in one of the compiled files
 - ▶ Indeed also libraries play a role
 - ▶ This control is performed by the linker
- ▶ `gcc -o name file1.c file2.c file3.c`

Libraries

- ▶ Libraries are collection of compiled definitions
- ▶ You include header files to get the declarations of objects in libraries
- ▶ At linking time libraries are searched for unresolved declarations
- ▶ Some libraries are included by gcc even if you do not specifically ask for them

Linking Math Functions: Example

```
1 #include <math.h>
2 #include <stdio.h>
3
4 int main() {
5     double n;
6     double sn;
7
8     scanf("%lf\n", &n); /* double needs %lf */
9     sn = sqrt(n);
10    /* conversion from double to float ok */
11    printf("Square root of %f is %f\n", n, sn);
12    return 0;
13 }
14
15 gcc -lm -o compute compute.c
```

Compilers, Linkers and More

- ▶ Different compilers differ in many details
 - ▶ Libraries names, ways to link against them, types of linking
- ▶ Check your documentation
- ▶ But preprocessing, compilation and linking are common steps

Recursive Functions (1)

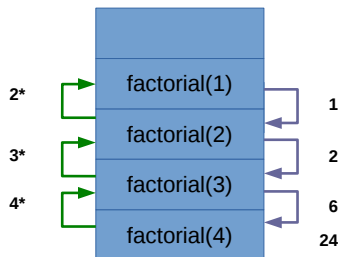
- ▶ Can a function call other functions?
 - ▶ Yes, indeed function calls appear only inside other functions (and everything starts with the execution of `main`)
- ▶ Can a function call itself?
 - ▶ Yes, but in this case special care should be taken
- ▶ A function which calls itself is called a **recursive function**
- ▶ Function *A* calls function *A*
- ▶ At a certain point function *B* calls *A*
 - ▶ *A* calls *A* then *A* calls *A* then *A* calls *A* ...
- ▶ When coding recursive functions attention should be paid to avoid endless recursive calls

Recursive Functions (2)

- ▶ Recursion theory can be studied for a longer time: here we will just scratch its surface from a basic coding standpoint
- ▶ Every recursive function must contain some code which allows it to terminate without entering the recursive step
 - ▶ Usually called **inductive base** or **base case**
- ▶ When recursion is executed, the new call should be driven "towards the inductive case"

Stack of Calls: Example

```
1 int factorial(int n) {  
2     if ((n == 0) || (n == 1))  
3         return 1;  
4     else  
5         return n * factorial(n - 1);  
6 }
```

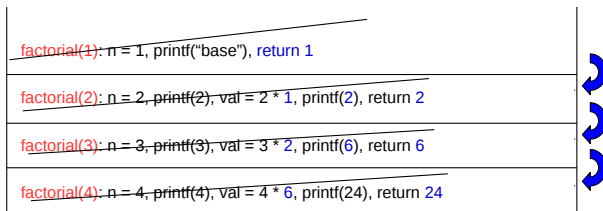
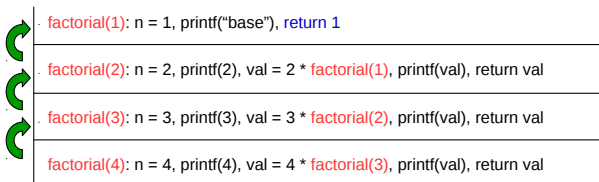


Tracing the Stack of Calls (1)

```
1 int factorial(int n) {
2     int val;
3     if ((n == 0) || (n == 1)) {
4         printf("base\n");
5         return 1;
6     } else {
7         printf("called with par = %d\n", n);
8         val = n * factorial(n - 1);
9         printf("returning %d\n", val);
10        return val;
11    }
12 }
13 int main() {
14     printf("%d\n", factorial(4));
15     return 0;
16 }
```

Tracing the Stack of Calls (2)

From the main: call `factorial(4)`



One More Example: Fibonacci Numbers

$$F(N) = \begin{cases} 1, & N \leq 1 \\ F(N-1) + F(N-2), & N > 1 \end{cases}$$

```
1 int fibonacci(int n) {  
2     if ((n == 0) || (n == 1))  
3         return 1;  
4     else  
5         return fibonacci(n-1) + fibonacci(n-2);  
6 }
```

The C Preprocessor (1)

- ▶ Before compilation, C source files are being preprocessed
- ▶ The preprocessor replaces tokens by an arbitrary number of characters
- ▶ Offers possibility of:
 - ▶ Use of named constants
 - ▶ Include files
 - ▶ Conditional compilation
 - ▶ Use of macros with arguments

The C Preprocessor (2)

- ▶ The preprocessor has a different syntax from C
- ▶ All preprocessor commands start with #
- ▶ A preprocessor directive terminates at the end-of-line
 - ▶ Do not put ; at the end of a directive
- ▶ It is a common programming practice to use all uppercase letters for macro names

The C Preprocessor: File Inclusion

- ▶ `#include <filename>`
 - ▶ includes file, follows implementation defined rule where to look for file, for Unix is typically `/usr/include`
 - ▶ Ex: `#include <stdio.h>`
- ▶ `#include "filename"`
 - ▶ looks in the directory of the source file
 - ▶ Ex: `#include "myheader.h"`
- ▶ Included files may include further files
- ▶ Typically used to include prototype declarations

The C Preprocessor: Motivation for Macros (1)

- ▶ Motivation for using named constants/macros
- ▶ What if the size of arrays has to be changed?

```
1 int data[10];
2 int twice[10];
3 int main()
4 {
5     int index;
6     for(index = 0; index < 10; ++index) {
7         data[index] = index;
8         twice[index] = index * 2;
9     }
10    return 0;
11 }
```


The C Preprocessor: Motivation for Macros (2)

More generic program if using named constants/macros

```
1 #define SIZE 20
2 int data[SIZE];
3 int twice[SIZE];
4 int main()
5 {
6     int index;
7     for(index = 0; index < SIZE; ++index) {
8         data[index] = index;
9         twice[index] = index * 2;
10    }
11    return 0;
12 }
```

Works but it no type information is associated with macros, so using `const` for this problem is a better solution.

The C Preprocessor: Macro Substitution (1)

- ▶ Definition of macro
 - ▶ `#define` NAME replacement_text
- ▶ Any name may be replaced with any replacement text
 - ▶ Ex: `#define` FOREVER `for` (`;;`) defines new word FOREVER to be an infinite loop
 - ▶ Ex: `#define` ODD(A, B) { `unsigned char` abit=A & 1; \ `unsigned char` bbit=B & 1; \ ... }

The C Preprocessor: Macro Substitution (2)

- ▶ Possible to define macros with arguments
 - ▶ `#define MAX(A, B) ((A) > (B) ? (A) : (B))`
- ▶ Each formal parameter (A or B) will be replaced by corresponding argument
 - ▶ `x = MAX(p+q, r+s);` will be replaced by
 - ▶ `x = ((p+q) > (r+s) ? (p+q) : (r+s));`
- ▶ It is type independent

The C Preprocessor: Macro Substitution (3)

- ▶ Why are the () around the variables important in the macro definition?
 - ▶ `#define SQR(A) (A)*(A)`
- ▶ Write a small program using this and see the effect without () in (A)*(A) by calling `SQR(5+1)`
- ▶ Try also `gcc -E program.c` sends the output of the preprocessor to the standard output
- ▶ What happens if you call `SQR(++i)`?

The C Preprocessor: Macro Substitution (4)

- ▶ Spacing and syntax in macro definition is very important
- ▶ See the preprocessor output of the following source code

```
1 #include <stdio.h>
2 #define MAX =10
3 int main()
4 {
5     int counter;
6     for(counter =MAX; counter > 0; --counter)
7         printf("Hi there!\n");
8     return 0;
9 }
```

wrong_macro.c

CH-230-A

Programming in C and C++

C/C++

Tutorial 6

Dr. Kinga Lipskoch

Fall 2019

The C Preprocessor: Macro Substitution (5)

- ▶ Defined names can be undefined using
 - ▶ `#undef NAME`
- ▶ Formal parameters are not replaced within quoted strings
- ▶ If parameter name is preceded by `#` in replacement text, the actual argument will be put inside quotes
 - ▶ `#define DPRINT(expr) printf(#expr " = %g\n", expr)`
 - ▶ `DPRINT(x/y)` will be expanded to
 - ▶ `printf("x/y" " = %g\n", x/y);`

The C Preprocessor: Conditional Inclusion (1)

- ▶ Preprocessing can be controlled by using conditional statements which will be evaluated while preprocessor runs
- ▶ Enables programmer to selectively include code, depending on conditions
- ▶ `#if`, `#endif`, `#elif` (i.e., else if), `#else`

```
1 #if defined(DEBUG)    // short: #ifdef DEBUG
2     printf("x: %d\n", x);
3 #endif
```


The C Preprocessor: Conditional Inclusion (2)

- ▶ `#ifdef`, `#ifndef` are special constructs that test whether name is (not) defined
- ▶ `gcc` allows to define names using the `-D` switch
- ▶ **Ex:** `gcc -DDEBUG -c program.c`
- ▶ Previous line is equivalent to
`#define DEBUG`

The C Preprocessor: Conditional Inclusion (3)

- ▶ Write a small program in which you illustrate the use of conditional inclusion for debugging purposes
- ▶ **Ex:** If the name `DEBUG` is defined then print on the screen the message "This is a test version of the program"
- ▶ If `DEBUG` is not defined then print on the screen the message "This is the production version of the program"
- ▶ Also experiment with `gcc -D`

The Structure of a Header File with Conditional Inclusion

```
1  /* Student.h */
2  #ifndef _LIST_H
3  #define _LIST_H
4  struct list {
5      int info;
6      struct list *next;
7  };
8  void printList(struct list *);
9  struct list * push_front(struct list *, int);
10 ...
11 #endif // this matches the initial #ifndef
```

Bit Operations

- ▶ The bit is the smallest unit of information
 - ▶ Represented by 0 or 1
- ▶ Eight bits form one byte
 - ▶ Which data type could be used for representation?
- ▶ Low-level coding like writing device drivers or graphic programming require bit operations
- ▶ Data representation
 - ▶ Octal (format %o), hexadecimal (format %x, representation prefix 0x)
- ▶ In C you can manipulate individual bits within a variable

Bitwise Operators (1)

Power	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
Decimal	128	64	32	16	8	4	2	1
Binary number	0	1	0	1	1	1	0	1

- ▶ Allow you to store and manipulate multiple states in one variable
- ▶ Allows to set and test individual bits in a variable

Bitwise Operators (2)

Operator	Function	Use
~	bitwise NOT	~expr
<<	left shift	expr1 << expr2
>>	right shift	expr1 >> expr2
&	bitwise AND	expr1 & expr2
^	bitwise XOR	expr1 ^ expr2
	bitwise OR	expr1 expr2
&=	bitwise AND assign	expr1 &= expr2
^=	bitwise XOR assign	expr1 ^= expr2
=	bitwise OR assign	expr1 = expr2

Bitwise and Logical AND

```
1 #include <stdio.h>
2 int main()
3 {
4     int i1, i2;
5     i1 = 6; // set to 4 and suddenly check 3 fails
6     i2 = 2;
7     if ((i1 != 0) && (i2 != 0))
8         printf("1: Both are not zero!\n");
9     if (i1 && i2)
10        printf("2: Both are not zero!\n");
11    // wrong check
12    if (i1 & i2)
13        printf("3: Both are not zero!\n");
14    return 0;
15 }
```

The Left-Shift Operator

- ▶ Moves the data to the left a specified number of bits
- ▶ Shifted out bits disappear
- ▶ New bits coming from the right are 0's
- ▶ Ex: `10101101 << 3` results in `01101000`

The Right-Shift Operator

- ▶ Moves the data to the right a specified number of bits
- ▶ Shifted out bits disappear
- ▶ New bits coming from the right are:
 - ▶ 0's if variable is unsigned
 - ▶ Value of the sign bit if variable is signed
- ▶ Ex:
 - ▶ $7 = 00000111 \gg 2$ results in 00000001
 - ▶ $-7 = 11111001 \gg 2$ results in 11111110

Using Masks to Identify Bits

MASK

0	0	0	0	0	1	0	0
---	---	---	---	---	---	---	---

&

flag

0	1	1	0	1	1	0	1
---	---	---	---	---	---	---	---

=

					1		
--	--	--	--	--	---	--	--

Using Masks

- ▶ Bitwise AND often used with a mask
- ▶ A mask is a bit pattern with one (or possibly more) bit(s) set
- ▶ Think of 0's as opaque and the 1's being transparent, only the mask 1's are visible
- ▶ If `result > 0` then at least one bit of mask is set
- ▶ If `result == MASK` then the bits of the mask are set

binary.c

```
1  #include <stdio.h>
2  char str[sizeof(int) * 8 + 1];
3  const int maxbit = sizeof(int) * 8 - 1;
4  char* itobin(int n, char* binstr) {
5      int i;
6      for (i = 0; i <= maxbit; i++) {
7          if (n & 1 << i) {
8              binstr[maxbit - i] = '1';
9          }
10         else {
11             binstr[maxbit - i] = '0';
12         }
13     }
14     binstr[maxbit + 1] = '\\0';
15     return binstr;
16 }
17 int main()
18 {
19     int n;
20     while (1) {
21         scanf("%i", &n);
22         if (n < 0) break;
23         printf("%6d: %s\\n", n, itobin(n, str));
24     }
25     return 0;
26 }
```

I didn't understand this one.

How to Turn on a Particular Bit

- ▶ To turn on bit 1 (second bit from the right), why does `flags += 2` not work?
 - ▶ If `flags = 2 = 000000010(2)`
 - ▶ Then `flags += 2` will result in
 - ▶ `flags = 4 = 00000100(2)` which "unsets" bit 1
- ▶ Correct usage:
 - ▶ `flags = flags | 2` is equivalent to
 - ▶ `flags |= 2` and turns on bit 1

How to Toggle a Particular Bit

- ▶ To toggle bit 1
 - ▶ `flags = flags ^ 2;`
 - ▶ `flags ^= 2;` toggles on bit 1
- ▶ General form
 - ▶ `flags ^= MASK;`

How to Test a Particular Bit

- ▶ To test bit 1, why does `flags == 2` not work?
- ▶ Testing whether any bit of MASK are set:
 - ▶ `if (flags & MASK) ...`
- ▶ Testing whether all bits of MASK are set:
 - ▶ `if ((flags & MASK) == MASK) ...`

CH-230-A

Programming in C and C++

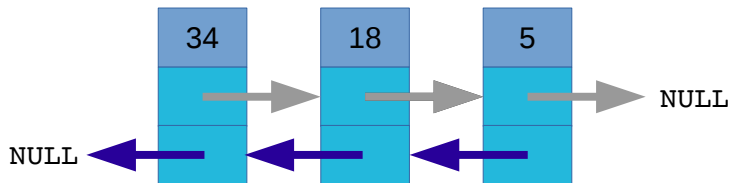
C/C++

Tutorial 7

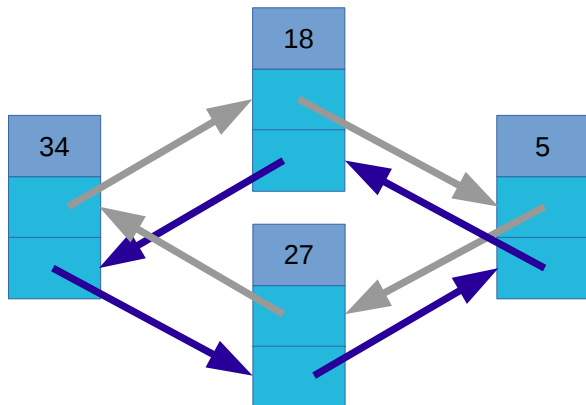
Dr. Kinga Lipskoch

Fall 2019

Doubly Linked Lists



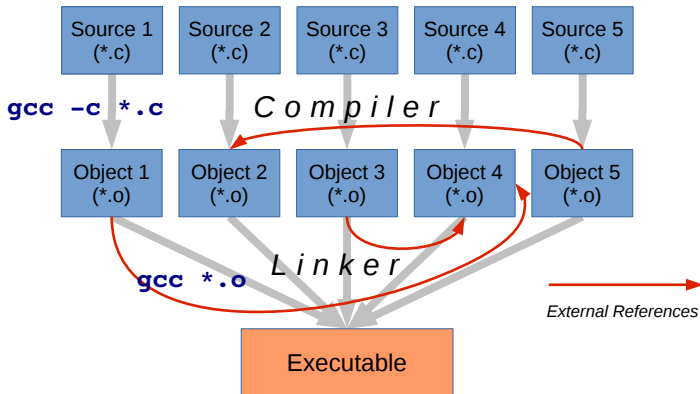
Circular Doubly Linked Lists



Building from Multiple Sources

- ▶ C compilers can compile multiple sources files into one executable
- ▶ For every declaration there must be one definition in one of the compiled files
 - ▶ Indeed also libraries play a role
 - ▶ This control is performed by the linker
- ▶ `gcc -o name file1.c file2.c file3.c`

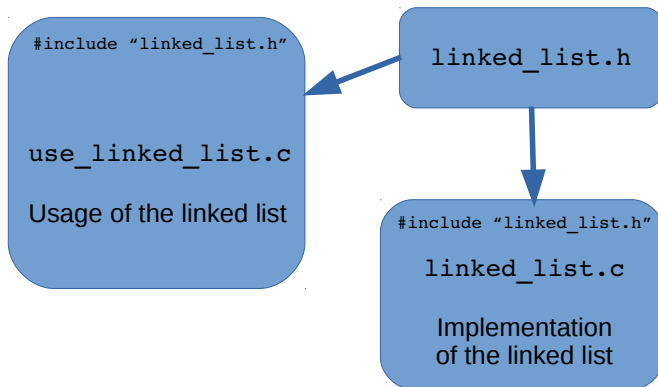
Linking



Linked List Header File

```
1  /*****
2  *
3  * A simply linked list is linked from node structures
4  * whose size can grow as needed. Adding more elements
5  * to the list will just cause it to grow and removing
6  * elements will cause it to shrink.
7  *-----*
8  * struct ll_node
9  *     used to hold the information for a node of a
10 *     simply linked list
11 *-----*
12 * Function declaration (routines)
13 *
14 *     push_front -- add an element in the beginning
15 *     push_back  -- add an element in the end
16 *     dispose_list -- remove all the elements
17 *     ...
18 *****/
```

Definition Import via #include



Compile Linked List from Multiple Sources

- ▶ Create a project with your IDE, add all files including the header file and then compile and execute
- ▶ or
- ▶ **Compile:** `gcc -Wall -o use_linked_list linked_list.c use_linked_list.c`
- ▶ **Execute:** `./use_linked_list`

Cygwin

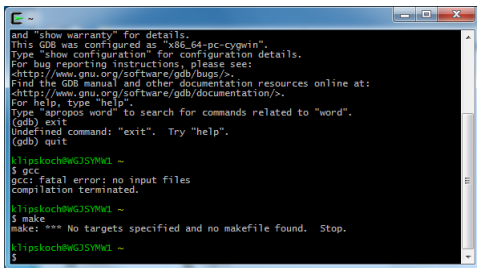
- ▶ Cygwin is a Unix-like environment and command-line interface for Microsoft Windows
- ▶ Cygwin provides native integration of Windows-based applications, data, and other system resources with applications, software tools, and data of the Unix-like environment
- ▶ Thus it is possible to launch Windows applications from the Cygwin environment, as well as to use Cygwin tools and applications within the Windows operating context

- ▶ Go to <https://cygwin.com/install.html>, download setup-x86_64.exe and install it
- ▶ During installation add gdb, gcc-core and make listed under Devel



Install Cygwin on Windows (2)

- ▶ Once installed under `C:/cygwin64` you will have a Unix-like environment
- ▶ You can use it to compile and debug your code using `gcc` and `gdb`



```
and "show warranty" for details.  
This GDB was configured as "x86_64-pc-cygwin".  
Type "show configuration" for configuration details.  
For bug reporting instructions, please see:  
<http://www.gnu.org/software/gdb/bugs/>.  
Find the GDB manual and other documentation resources online at:  
<http://www.gnu.org/software/gdb/documentation/>.  
For help, type "help".  
Type "apropos word" to search for commands related to "word".  
(gdb) exit  
Undefined command: "exit". Try "help".  
(gdb) quit  
  
klipskoch@WGJSYMW1 ~  
$ gcc  
gcc: fatal error: no input files  
compilation terminated.  
  
klipskoch@WGJSYMW1 ~  
$ make  
make: *** No targets specified and no makefile found. Stop.  
  
klipskoch@WGJSYMW1 ~  
$
```

make (1)

- ▶ make is special utility to help programmer compiling and linking programs
- ▶ Programmers had to type in compile commands for every change in program
- ▶ With more modules more files need to be compiled
 - ▶ Possibility to write script, which handles sequence of compile commands
- ▶ Inefficient

make (2)

- ▶ Compiling takes time
- ▶ For only small change in one module, not necessary to recompile other modules
- ▶ `make`: compilations depends upon whether file has been updated since last compilation
- ▶ Also possible to specify dependencies
- ▶ Also possible to specify commands to compile (e.g., depending of suffix of source)

Makefile (1)

- ▶ A makefile has the name "Makefile"
- ▶ Makefile contains following sections:
 - ▶ Comments
 - ▶ Macros
 - ▶ Explicit rules
 - ▶ Default rules

Makefile (2)

- ▶ Comments
 - ▶ Any line that starts with a `#` is a comment
- ▶ Macro format
 - ▶ `name = data`
 - ▶ `Ex: OBJ=linked_list.o use_linked_list.o`
 - ▶ Can be referred to as `$(OBJ)` from now on

Makefile (3)

Explicit rules

- ▶ `target:source1 [source2] [source3]`
 `command1`
 `[command2]`
 `[command3]`
- ▶ `target` is the name of file to create
- ▶ File is created from `source1` (and `source2`, ...)
- ▶ `use_linked_list: use_linked_list.o linked_list.o`
 `gcc -o use_linked_list`
 `use_linked_list.o linked_list.o`

Makefile (4)

Explicit rules

- ▶ target:

command

Commands are unconditionally executed each time make is run

- ▶ Commands may be omitted, **built-in rules** are used then to determine what to do

`use_linked_list.o: linked_list.h use_linked_list.c`

- ▶ Create `use_linked_list.o` from `linked_list.h` and `use_linked_list.c` using standard suffix rule for getting to `use_linked_list.o` from `linked_list.c`

- ▶ `$(CC) $(CFLAGS) -c file.c`

Example Makefile (1)

- ▶ Header file with `struct` definition and function prototypes
 - ▶ `header_file.h`
- ▶ Implementation file with usage of the `struct` and function definitions
 - ▶ `implementation.c`
- ▶ Main function where implemented behaviour can be used
 - ▶ `main.c`
- ▶ Makefile with different targets for different purposes
 - ▶ `Makefile.txt`

Run Makefile

- ▶ `make`
Default makefile called `Makefile` and default target `all`
- ▶ `make TargetName`
Default makefile called `Makefile` and target `TargetName`
- ▶ `make -f MyMakeFile.txt`
Makefile called `MyMakeFile.txt` and default target `all`
- ▶ `make -f MyMakeFile.txt TargetName`
Makefile called `MyMakeFile.txt` and default target `TargetName`

Example Makefile (2)

```
1 CC = gcc
2 CFLAGS = -Wall
3
4 OBJ = linked_list.o use_linked_list.o
5
6 all: use_linked_list
7
8 use_linked_list: $(OBJ)
9                 $(CC) $(CFLAGS) -o use_linked_list $(OBJ)
10
11 use_linked_list.o: linked_list.h use_linked_list.c
12
13 linked_list.o: linked_list.h linked_list.c
14
15 clean:
16     rm -f use_linked_list *.o
```

Function Pointers

- ▶ A pointer may not just point to a variable, but may also point to a function
- ▶ In the program it is assumed that the function does what it has to do and you use it in your program as if it was there
- ▶ The decision which function will actually be called is determined at run-time

Function Pointer Syntax

- ▶ `void (*foo)(int);`
 - ▶ `foo` is a pointer to a function taking one argument, an integer, and that returns `void`
- ▶ `void *(*foo)(int *);`
 - ▶ `foo` is a pointer to a function that returns a `void *` and takes an `int *` as parameter
- ▶ `int (*foo_array[2])(int);`
 - ▶ `foo_array` is an array of two pointer functions having an `int` as parameter and returning an `int`
- ▶ Easier and equivalent:
`typedef int (*foo_ptr_t)(int);`
`foo_ptr_t foo_ptr_array[2];`

Function Pointers: Simple Examples

```
1 void (*func) (void);    /* define pointer to function */
2 void a(void) { printf("func a\n"); }
3 void b(void) { printf("func b\n"); }
4
5 int main() {
6     func = &a;    // calling func() is the same as calling a()
7     func = a;    // calling func() is the same as calling a()
8     func();
9 }
```

One may have an [array of function pointers](#):

```
1 int func1(void);
2 int func2(void);
3 int func3(void);
4 int (*func_arr[3])(void)
5                                     = {func1, func2, func3};
```

Another Function Pointer Example

```
1 #include <stdio.h>
2 void output(void) {
3     printf("%s\n", "Please enter a number:");
4 }
5 int sum(int a, int b) {
6     return (a + b);
7 }
8 int main() {
9     int x, y;
10    void (*fptr1)(void);
11    int (*fptr2)(int, int);
12    fptr1 = output;
13    fptr2 = sum;
14    fptr1();    // cannot see whether function or pointer
15    scanf("%d", &x);
16    (fptr1)();    // some prefer this to show it is pointer
17    (*fptr1)();    // complete syntax, same as above
18    scanf("%d", &y);
19    printf("The sum is %d.\n", fptr2(x, y));
20 }
```

Alternatives for Usage

```
1 int (*fct) (int, int);
2 /* define pointer to a fct */
3 int plus(int a, int b) {return a+b;}
4 int minus(int a, int b) {return a-b;}
5 int a=3; int b=4;
6 fct = &plus;
7 /* calling fct() same as calling plus() */
8 printf("fct(a,b):%d\n", fct(a,b)); /* 7 */
```

or

```
1 printf("fct(a,b):%d\n", (*fct)(a,b)); /* 7 */
2 fct = &minus;
3 /* calling fct() same as calling minus() */
4 printf("fct(a,b):%d\n", fct(a,b)); /* -1 */
```


CH-230-A

Programming in C and C++

C/C++

Tutorial 8

Dr. Kinga Lipskoch

Fall 2019

Queues

FIFO uses list, LIFO uses stack.

- ▶ A queue is a FIFO (First-In First-Out) data structure, often implemented as a simply linked list
- ▶ However:
 - ▶ New items can only be added to end of list
 - ▶ Items can be removed from the list only from the beginning
 - ▶ Just think of line waiting in front of the movies

Operations on the Queue

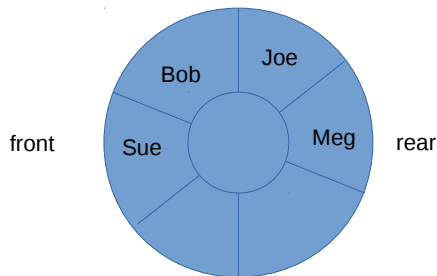
- ▶ Initialize queue
- ▶ Determine whether queue is empty
- ▶ Determine whether queue is full
- ▶ Determine number of items in queue
- ▶ Add item to queue (always at end)
- ▶ Remove item from queue (always from front)
- ▶ Empty queue

Data Representation

- ▶ Array might be used for queue
 - ▶ Simple implementation, but all elements need to be moved each time item is removed from queue
- ▶ Wrap-around array
 - ▶ Instead of moving elements, use array where indexes wrap around
 - ▶ Front and rear pointers point to begin and end of queue

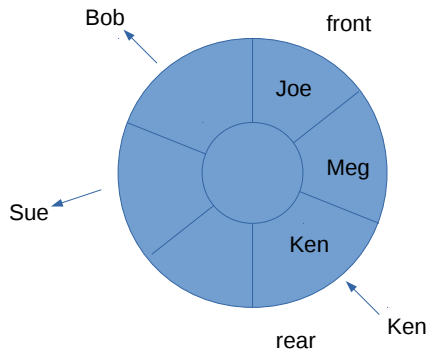
Queue (1)

4 people in the queue



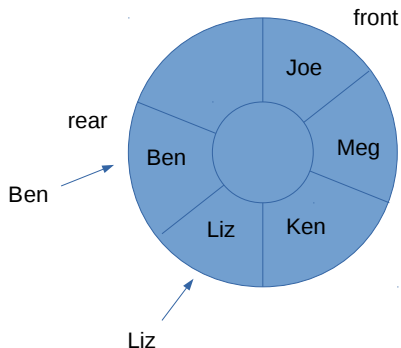
Queue (2)

Sue and Bob leave, while Ken joins queue



Queue (3)

Circular queue wraps around



Queue Implementation (1)

- ▶ Use linked list or circular linked list
- ▶ Should work with anything, but let's start with integers

```
typedef int Item;
```

- ▶ Linked list is built from nodes

```
1 struct node {  
2     Item item;  
3     struct node *next;  
4 };  
5 typedef struct node Node;
```


Queue Implementation (2)

- ▶ Queue needs to keep track of front and rear items
- ▶ Just use pointers for this
- ▶ Counter to keep track of items in queue

```
1 struct queue {  
2     Node *front;  
3     Node *rear;  
4     int items;  
5 };  
6 typedef struct queue Queue;
```

Header Files and Conditional Inclusion

- ▶ Have seen that conditional statements can control preprocessing itself
- ▶ To make sure that contents of file `myheader.h` is included only once

```
1 #ifndef _MYHEADER_H
2 #define _MYHEADER_H
3
4     // contents of myheader.h goes here
5
6 #endif
```

Interface and Complete Implementation

- ▶ Header file contains data types and prototypes
 - ▶ `queue.h`
 - ▶ Needs to be included by implementation (and users of queue)
- ▶ Implementation of queue
 - ▶ `queue.c`
- ▶ User of queue
 - ▶ `testqueue.c`
- ▶ Makefile with targets like `all`, `testqueue`, `doc`, `clean`, `clobber`
 - ▶ `Makefile`
- ▶ Configuration file for doxygen
 - ▶ `Doxyfile`
- ▶ Testcase input and output
 - ▶ `test1.in` `test1.out`

Adding an Item to a Queue

1. If queue is full do not do anything
2. Create a new node
3. Copy item to the node
4. Set next pointer to NULL
5. Set front node if queue was empty
6. Set current rear node's next pointer to new node if queue already exists
7. Set rear pointer to new node
8. Add 1 to item count

Removing an Item from a Queue

1. If queue is empty do not do anything
2. Copy item to waiting variable
3. Reset front pointer to the next item in queue
4. Free memory
5. Reset front and rear pointers to NULL, if last item is removed
6. Decrement item count

File Handling in C

- ▶ Input and output can come from/go into files
- ▶ C treats files as streams of data
- ▶ A stream is a sequence of bytes (either incoming or outgoing)
- ▶ The language does not provide basic constructs for file handling, but rather the standard library does

Communicating with Files

- ▶ Communication with files from the outside
- ▶ Output redirection
 - ▶ `file > outputfile`
- ▶ Input redirection
 - ▶ `file < inputfile`

Working with Files

- ▶ The paradigm is the following:
 - ▶ Open the file
 - ▶ Read/write
 - ▶ Close the file
- ▶ In C the information concerning a file are stored in a `FILE` structure (defined in `stdio.h`)
- ▶ The C `stdio` library implements buffered I/O: Data is first written to an internal buffer, which is eventually written to a file

Standard Streams

- ▶ `stdin`
 - ▶ Standard input is stream data (often text) going into a program
 - ▶ Unless redirected, standard input is expected from the keyboard which started the program
- ▶ `stdout`
 - ▶ Standard output is the stream where a program writes its output data
 - ▶ Unless redirected, standard output is the text terminal which initiated the program
- ▶ `stderr`
 - ▶ Standard error is another output stream typically used by programs to output error messages or diagnostics
 - ▶ It is a stream independent of standard output and can be redirected separately

File Modes

Streams can be handled in two modes: (only important for MS Windows)

- ▶ Text streams: sequence of characters logically organized in lines. Lines are terminated by a newline ('\n')
 - ▶ Sometimes pre/post processed
 - ▶ Example: text files
- ▶ Binary streams: sequence of raw bytes
 - ▶ Examples: images, mp3, user defined file formats, etc.

Opening a File

- ▶ To open a file the `fopen` function is used
`FILE *fopen(const char * name, const char * mode)`
- ▶ `name`: name of the file (OS level)
- ▶ `mode`: indicates the type of the file and the operations that will be performed

```
FILE *fptr;  
fptr = fopen("myfile.txt", "r");
```

Mode Strings

String	Meaning
"r"	Open for reading, positions at the beginning
"r+"	Open for reading and writing, positions at the beginning
"w"	Open for writing, truncate if exists, positions at the beginning
"w+"	Open for reading and writing, truncate if exists, positions at the beginning
"a"	Open for appending, does not truncate if exists, positions at the end
"a+"	Open for appending and reading, does not truncate if exists, positions at the end

A `b` or `t` can be added to indicate it is a binary/text file

Closing a File

- ▶ `int fclose(FILE *fp);`
- ▶ Forgetting to close a file might result in a loss of data
- ▶ After a file is closed it is not possible anymore to read/write

```
1      FILE *fptr;  
2      fptr = fopen("myfile.txt", "r");  
3      if (fptr == NULL) {  
4          printf("Some error occurred!\n");  
5          exit(1);  
6      }  
7      ...  
8      /* do some operations */  
9      fclose(fptr);  
10     ...
```

Reading/Writing

Prototype	Use
<code>int getc(FILE *fp)</code>	Returns next <code>char</code> from <code>fp</code>
<code>int putc(int c, FILE *fp)</code>	Writes a <code>char</code> to <code>fp</code>
<code>int fscanf(FILE* fp, char * format, ...)</code>	Gets data from <code>fp</code> according to the format string
<code>int fprintf(FILE* fp, char * format, ...)</code>	Outputs data to <code>fp</code> according to the format string

Line Input and Line Output

```
char *fgets(char *line, int max, FILE *fp);
```

- ▶ Already seen with stdin
- ▶ Used for files as well

```
int fputs(char *line, FILE *fp);
```

- ▶ Outputs/writes a string to a file

Files: Example 1

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main() {
4     char ch;
5     FILE *fp;
6     fp = fopen("file.txt", "r");
7     if (fp == NULL) {
8         printf("Cannot open file!\n");
9         exit(1);
10    }
11    ch = getc(fp);
12    while (ch != EOF) {
13        putchar(ch);
14        ch = getc(fp);
15    }
16    fclose(fp);
17    return 0;
18 }
```


Files: Example 2

```
1 # include <stdio.h>
2 # include <stdlib.h>
3 int main () {
4     char ch;
5     FILE * fp;
6     fp = fopen("file.txt", "r") ;
7     if (fp == NULL) {
8         printf("Cannot open file!\n");
9         exit(1);
10    }
11    while((ch=getc(fp))!=EOF) {
12        putchar(ch);
13    }
14    fclose(fp);
15    return 0;
16 }
```

Files: Example 3

```
1 # include <stdio.h>
2 # include <stdlib.h>
3 int main () {
4     char ch;
5     FILE * fp;
6     fp = fopen("file.txt", "r") ;
7     if (fp == NULL) {
8         printf("Cannot open file!\n");
9         exit(1);
10    }
11    while(!feof(fp)) {
12        ch=getc(fp);
13        if (ch!=EOF)
14            putchar(ch);
15    }
16    fclose(fp);
17    return 0;
18 }
```

CH-230-A

Programming in C and C++

C/C++

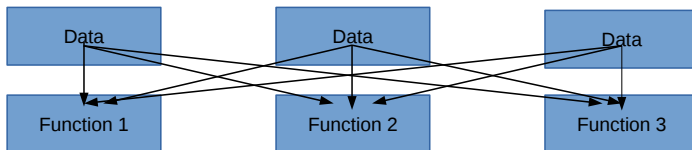
Tutorial 9

Dr. Kinga Lipskoch

Fall 2019

Problems with Imperative Programming using Functions

- ▶ `account.c` It is a link for "account" programming which shows the problem with imperative programming using Functions.
- ▶ Functions can use data that is generally accessible, but do not make sense
- ▶ Possible to apply invalid functions to data
- ▶ No protection against semantic errors
- ▶ Data and functions are kept apart



Disadvantages of Imperative Programming

- ▶ Lack of protection of data
 - ▶ data is **not protected**
 - ▶ transferred as parameters from function to function
 - ▶ can be manipulated anywhere
 - ▶ difficult to follow how changes affect other functions
- ▶ Lack of overview in large systems
 - ▶ huge collection of unordered functions
- ▶ Lack of source code reuse
 - ▶ difficult to find existing building blocks

OOP Allows Better Modeling

important.

The OOP approach allows the programmer to think in terms of the problem rather than in terms of the underlying computational model

C++ is an OOP program.

An Example: A Program for Printing the Grades of this Course

- ▶ Write a program which reads the names of the students and their grades, and then prints the list in some order (e.g., ascending order)
- ▶ Assumptions:
 - ▶ Less than 100 students will attend this course
 - ▶ For every student we log the complete name, the grade and the year of birth

An Imperative (C like) Solution (1)

- ▶ Three so called aligned vectors, one of strings, one of floats, one of integers (name, grade, and year of birth)
- ▶ One function which fills the vectors and one function which sorts the elements (comparison based on the grade and consequent swap of all corresponding information)
 - ▶ Could also use a C struct to group all the data together

A Classic Solution (2)

```
1 for (i = 0; i < Nstud; i++) {
2     scanf("%s", names[i]);
3     scanf("%f", &grades[i]);
4     ...
5 }
6 void sort(char** names, float*
    grades, int* years, int Nstud) {
7     ...
8     if (grade[j] < grade[k]) {
9         /* swap elements */
10        strcpy(tmpstr, name[j]);
11        strcpy(name[j], name[k]);
12        strcpy(name[k], tmpstr);
13        tmpgrade = grade[j];
14        grades[j] = grades[k];
15        grades[k] = tmpgrade;
16        ...
17    }
18 }
```

All this process
indicates the swapping.

Name	Grade	Year
XY	1.0	1978

A Classic Solution (3)

```
1 struct student {
2     char name[40];
3     double grade;
4     int year;
5 };
6
7 struct student S[100];
8
9 for (i = 0; i < Nstud; i++) {
10     scanf("%s", S[i].name);
11     scanf("%f", &S[i].grade);
12     ...
13 }
14 void sort(struct student S*, int Nstud) {
15     ...
16     if (S[j].grade < S[k].grade) {
17         /* swap elements */
18         strcpy(tmpstr, S[j].name);
19         strcpy(S[j].name, S[k].name);
20         strcpy(S[k].name, tmpstr);
21         tmpgrade = S[j].grade;
22         S[j].grade = S[k].grade;
23         S[k].grade = tmpgrade;
24         ...
25     }
26 }
```

Name	Grade	Year
XY	1.0	1978

A Possible OO Solution

- ▶ Which are the entities?
 - ▶ Students
- ▶ What is their interesting data?
 - ▶ Name, grade, date of birth
- ▶ What kind of operations do we have on them?
 - ▶ Set the name/grade/date
 - ▶ Get the grade (to sort)
 - ▶ Print the student's data to screen
- ▶ Then: build a model for this entity and write a program which solves the problem by using it

OOP Jargon

- ▶ You wish to model entities which populate your problem
- ▶ Such models are called **classes**
- ▶ Being a model, a **class** describes all the entities but itself it is not an entity
 - ▶ The class of cars (**Car**): every car has a color, a brand, an engine size, etc.
- ▶ Specific **instances** of a class are called **objects**
 - ▶ John's car is an instance of the class **Car**: it is red, its brand is XYZ, it has a 2.0 l engine
 - ▶ Mark's car is another instance of **Car**: it is blue, its brand is ZZZ, it has a 4.2 l engine

Class Clients (or Users)

- ▶ We will often talk about class clients
- ▶ They are programmers using that class
 - ▶ It could be yourself, your staff mate, your company colleagues, or a third party which makes use of your developed libraries
 - ▶ Not the program user (to whom, whether the program is written in an OOP language or not, can be completely transparent)
- ▶ You develop a class and put it in a repository
- ▶ From that point, someone who uses it is a client

Hello World (1)

```
1 #include <iostream>
2
3 int main(int argc, char** argv) {
4     std::cout << "Hello World!" << std::endl;
5     return 0;
6 }
```

Hello World (2)

```
1 #include <iostream>
2
3 int main(int argc, char** argv) {
4     std::cout << "Hello World!" << std::endl;
5     // this is a one line comment
6     return 0;
7 }
```

- ▶ `<iostream>`: C++ preprocessor naming convention
- ▶ `std::cout`: used from the `std` namespace
- ▶ `//`: one line comments specific to C++ (but have found their way to C as well)

The C++ Preprocessor

Runs before the compiler, works as the C preprocessor but:

- ▶ C++ standard header files have to be included omitting the extension
- ▶ The file `iostream` is then included as follows
`#include <iostream>`
- ▶ C standard header files have to be included omitting the extension and inserting a c as first letter
`#include <cstdlib>`
- ▶ Other files have to be included as in C
`#include <pthread.h>`
`#include "myinclude.h"`

C++ Comments

- ▶ C++ allows to insert one-line comments and multi-line comments

```
// this text will be ignored  
int a; // some words on a line  
/* multi-line comment */
```

- ▶ Like in C, C++ comments are removed from the source by the preprocessor
- ▶ The programmer is free to use both styles

cout: The First Object we Meet

- ▶ C++ provides some classes for dealing with I/O
- ▶ `cout` (console out) is an instance of the built-in `ostream` class, it is declared inside the `iostream` header
- ▶ The inserter operator `<<` is used to send data to a stream
`cout << 3 + 5 << endl; // prints 8`
- ▶ Inserter operators can be concatenated
- ▶ The `endl` modifier writes an EOL (End Of Line)
- ▶ Data sent to `cout` will appear on the screen
- ▶ The stream `cerr` can be used to send data to the standard error stream (`stdin`, `stdout`, `stderr` in the C library)

Operators with Different Meaning

- ▶ `<<` has a different meaning in C
- ▶ C++ allows the programmer to define how operators should behave when applied to user defined classes
 - ▶ This is called operator overloading (will be covered later)
 - ▶ In C, the `<<` operator only allows to shift bits into integer variables

Compile and Execute

- ▶ The g++ compiler provided by the GNU software foundation is one of the best available (and for free)
- ▶ Built on the top of gcc, its use is very similar
- ▶ C++ source files have extension
 - ▶ .cpp, .cxx, .cc or .C
 - ▶ self-written header files have the usual .h extension
- ▶ Adhere to these conventions
- ▶ Even if gcc would compile the files (it will recognize them as C++ source files by the extension), use g++ instead, as it will include the standard C++ libraries while linking

Compiling a C++ Program

- ▶ Compiling `hello.cpp` to an executable
`g++ -Wall -o hello hello.cpp`
- ▶ Running the executable program
`./hello`

cin : Console Input (1)

- ▶ `cin` is the companion stream of `cout` and provides a way to get input
 - ▶ as `cout`, it is declared in `iostream`
- ▶ The overloaded operator `>>` (extractor) gets data from the stream

```
float f;
```

```
cin >> f;
```

- ▶ Warning: it does not remove endlines
- ▶ If you are reading both numbers and strings you have to pay attention

Boolean and String as Types

- ▶ `bool` as distinct type
(also now in C, you need to include `stdbool.h`)

```
bool c;  
c = true;  
cout << c << endl;
```

- ▶ `string` as distinct type

```
string s;  
s = "Hello, I am a C++ string";  
cout << s << endl;
```

cin : Console Input (2)

- ▶ There is one `getline` function and one `getline` method
- ▶ The function `getline` is a global function and reads a string from an input stream

```
string str;
```

```
getline(cin, str);
```

- ▶ The method `getline` gets a whole line of text (ended by `'\n'` and it removes the separator)
- ▶ It reads a C string (a character array that ends with a `'\0'`)

```
char buf[50];
```

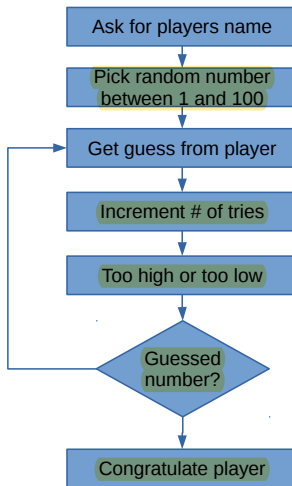
```
string s;
```

```
cin.getline(buf, 50);
```

```
s = string(buf);
```

```
// convert to a C++ string
```


A Simple Guessing Game



How to Pick a Random Number

```
1 #include <iostream>
2 #include <cstdlib>
3 #include <ctime>
4 using namespace std;
5 int main() {
6     int die;
7     int count = 0;
8     int randomNumber;
9     // init random number generator
10    srand(static_cast<unsigned int>(time(0)));
11    while (count < 10) {
12        count++;
13        randomNumber = rand();
14        die = (randomNumber % 6) + 1;
15        cout << count << ": " << die << endl;
16    }
17    return 0;
18 }
```

C++ Extensions to C

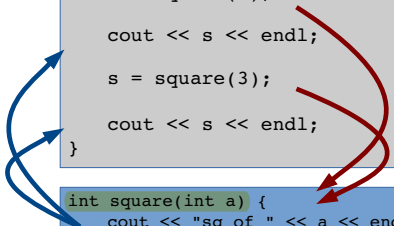
- ▶ Inline functions
 - ▶ available in C since the standard C99
- ▶ Overloading
- ▶ Variables can be declared anywhere
 - ▶ possible in C since the standard C99
- ▶ References

Inline Functions (1)

- ▶ For each call to a function you need to setup registers (setup stack), jump to new code, execute code in function and jump back
- ▶ To save execution time macros (i.e., `#define`) have often been used in C
- ▶ A preprocessor does basically string replacement
- ▶ Disadvantage: it is error prone, no type information
- ▶ `inline.cpp`

Inline Functions (2)

```
int main() {  
    int s;  
    s = square(5);  
  
    cout << s << endl;  
  
    s = square(3);  
  
    cout << s << endl;  
}  
  
int square(int a) {  
    cout << "sq of " << a << end;  
    return a * a;  
}
```



The diagram illustrates the execution flow between the `main` and `square` functions. Two blue arrows originate from the `square(5)` and `square(3)` calls in the `main` function, pointing to the `square` function definition. Two red arrows originate from the `return a * a;` line in the `square` function, pointing back to the assignment statements `s = square(5);` and `s = square(3);` in the `main` function, indicating the return of the calculated values.

```
int main() {  
    int s;  
    cout << "sq of " << 5 << end;  
    s = 5 * 5;  
  
    cout << s << endl;  
  
    cout << "sq of " << 3 << end;  
    s = 3 * 3;  
  
    cout << s << endl;  
}
```

Function Overloading

```
1 #include <iostream>
2 using namespace std;
3 int division(int dividend, int divisor) {
4     return dividend / divisor;
5 }
6 float division(float dividend, float divisor) {
7     return dividend / divisor;
8 }
9 int main() {
10     int ia = 10;
11     int ib = 3;
12     float fa = 10.0;
13     float fb = 3.0;
14
15     cout << division(ia, ib) << endl;
16     cout << division(fa, fb) << endl;
17     return 0;
18 }
```

Output: 3 3.33333

Variable Declaration "Everywhere"

```
1 void function() {  
2     .....  
3     printf("C-statements...\n");  
4     .....  
5     int x = 5;           Here we can notice that the variable is declared after printf.  
6     // now allowed, works in C  
7     // as well since standard C99  
8 }
```

No "Real" References in C (1)

Accessing a variable in C

- ▶ `int a;` `// variable of type integer`
- ▶ `int b = 9;` `// initialized variable of type integer`
- ▶ `a = b;` `// assign one variable to another`
- ▶ `b = 5;` `// assignment of value to variable`

No "Real" References in C (2)

Accessing variable via pointers

- ▶ `int a;` `// variable of type integer`
- ▶ `int b = 5;` `// initialized variable`
- ▶ `int* ptr;` `// pointer to integer`
- ▶ `ptr = &a;` `// address of a is assigned to ptr`
`// (it points to a)`
- ▶ `*ptr = b;` `// assign b to content where ptr`
`points to a is now 5`

References in C++

A reference can be seen as additional name or as an alias of the variable

```
▶ int a;  
▶ int b = 5;           // initialized variable  
▶ int& ref = a;        // reference to variable  
                        // of type int  
▶ ref = 7;             // assignment of variable a  
                        // via reference ref
```

"Real" Call-by-Reference (1)

```
1 #include <stdio.h>
2 void swap_cpp(int &a, int &b);      // prototype
3 void swap_c(int *a, int *b);      // prototype
4 void swap_wrong(int a, int b);    // prototype
5 int main(void) {
6     int a_cpp = 3, b_cpp = 5,
7     a_c = 3, b_c = 5,
8     a = 3, b = 5;
9     swap_cpp(a_cpp, b_cpp);
10    swap_c(&a_c, &b_c);
11    swap_wrong(a, b);
12    printf("C++: a=%d, b=%d\n", a_cpp, b_cpp);
13    printf("C: a=%d, b=%d\n", a_c, b_c);
14    printf("Wrong: a=%d, b=%d\n", a, b);
15    return 0;
16 }
```

"Real" Call-by-Reference (2)

```
1 void swap_cpp(int &a, int &b) {
2     // real Call-by-Reference
3     int help = a;
4     a = b;
5     b = help;
6 }
7 void swap_c(int *a, int *b) {
8     // not real Call-by-Reference
9     // Call-by-Value via Pointer
10    int help = *a;
11    *a = *b;
12    *b = help;
13 }
14 void swap_wrong(int a, int b) {
15     // Call-by-Value
16     int help = a;           // no swapping of passed
17     a = b;                  // parameters,
18     b = help;               // since only copies are swapped
19 }
```

Constant References

- ▶ References are not only useful if arguments are to be modified
- ▶ No copying of (possibly large) data objects will happen
- ▶ Using references saves time
- ▶ To show that parameters are not going to be modified constant references should be used

```
void writeout(const int &a, const int &b) { ... }
```

- ▶ `ref_timing.cpp`

Dynamic Memory Allocation

C++ has an operator for dynamic memory allocation

- ▶ It replaces the use of the C `malloc` functions
- ▶ `alloc_in_c.c`
 - ▶ Easier and safer
- ▶ The operator is called `new`
 - ▶ It can be applied both to user defined types (classes) and to native types
 - ▶ `operator_new.cpp`
 - ▶ use `-std=c++0x` switch to compile program according to the standard C++11
 - ▶ use `-std=c++14` switch to compile program according to the standard C++14

Operators new and delete

- ▶ new
 - ▶ primitive types are initialized to 0
 - ▶ returned type is a pointer to the allocated type
- ▶ delete releases allocated memory
 - ▶ `delete ptr_1; // releases int`
 - ▶ `delete [] ptr_7; // releases int-array`
- ▶ Memory that has been allocated via `new []` must be released by `delete []`
- ▶ C: `malloc()` --> `free()`
- ▶ C++: `new` --> `delete`

CH-230-A

Programming in C and C++

C/C++

Tutorial 10

Dr. Kinga Lipskoch

Fall 2019

New Header Files

- ▶ `stdlib.h` or `math.h` can still be included in C++, but `<cstdlib>` or `<cmath>` is preferred
- ▶ Functions are then put into the `std` namespace
- ▶ Header files explicitly created for C++

Namespaces

- ▶ C has only one global namespace
- ▶ Name collisions avoided by using prefixes
 - ▶ `jpeg_xxx`
- ▶ C++: `using namespace std;`
- ▶ Standard C++ libraries are all inside of the `std` namespace

structs and classes

```
1 struct article {           // the C way
2     int id;
3     float price;
4 };
5 int add_article(struct article*, int id, float price);
6 ...
7 struct article a;
8 add_article(&a, 1234, 9.99);
```

```
1 class Article {           // the C++ way
2     int id;
3     float price;
4     int add(float id, int price);
5 };
6 Article s;
7 s.add(1234, 9.99);
```

The string Class (1)

- ▶ `string` is another class provided by the standard C++ library
- ▶ It handles a sequence of characters which may dynamically grow or shrink
- ▶ Strings can be created in different ways

```
1 string empty;           // empty string
2 string a("this is also a string");
3 string b = "also this one";
4 empty = a;              // now they hold the same
5 empty += " 8";          // appending to a string
```

The string Class (2)

- ▶ The `string` class has many methods performing useful operations
 - ▶ Appending, inserting, removing, concatenating, replacing, searching, comparing and more
- ▶ We are not covering all of them on the slides
- ▶ See Chapter 5 in the *C++ Annotations* book or check operators and methods on www.cplusplus.com

Example with Strings

```
string_tester.cpp
```

Deficiencies of C Structures

`cstruct.c`

- ▶ Any function is able to read and also write the variables one and two
- ▶ Uncontrolled access to the account
- ▶ Clients are able to directly manipulate data
- ▶ No guarantee that access is done in the “right way”

struct in C++

Member functions, methods are part of the struct itself

```
1 struct account {  
2     char name[100];  
3     unsigned int no;  
4     double balance;  
5  
6     // functions inside struct  
7     void createAccount(const char *name, ....);  
8     void deposit(double amount);  
9     void drawout(double amount);  
10    void transfer(struct account *to, double  
        balance);  
11 };
```


How to Define New Classes

- ▶ The keyword `class` is used to define a new class
 - ▶ `struct` with methods
- ▶ Two other keywords used when defining classes:
 - ▶ `private`: to define what is internal to the class
 - ▶ `public`: to define what can be used from outside the class
- ▶ There exists a third keyword, `protected`, which will be introduced when we will talk about `inheritance` in more detail

Information Hiding

- ▶ While designing a class it is necessary to devise which information should be visible and which one should not be visible to class users
 - ▶ This choice has to be done for both data members and methods
- ▶ The visible (`public`) subset of data and methods is called the **interface** of the class

Information Hiding: Why?

- ▶ **Protection:**
 - ▶ Users are not allowed to use class data not belonging to themselves (data integrity)
- ▶ **Modularity:**
 - ▶ An interface is a **contract** between the class developer and the class user
 - ▶ As long as the interface does not change, the private part of the class can be changed without the need to modify the code that uses that class

private and public

- ▶ **General rule:** data should be kept private and methods should be provided to access (read and write) them
- ▶ There may be exceptions to this principle, mainly due to efficiency needs
- ▶ Methods providing functionality needed by class users will be `public`
- ▶ Methods used to implement these functionality should be `private`

Critters

- ▶ Critter have several properties (name, color, hunger)
- ▶ Data concerning these properties will be kept private
- ▶ Methods should be provided to write those data (setter methods)
- ▶ A method to get data (e.g., name for sorting – getter method)
- ▶ An additional method can be to print the data to the screen

Implementation of the class Critter

- ▶ It is common to split the coding into two components
 - ▶ A header file specifies how the class looks like, i.e., its data members and methods
 - ▶ Class declaration
 - ▶ A C++ file defines how the methods are actually implemented
 - ▶ Class definition
- ▶ `Critter.h`
- ▶ `Critter.cpp`

Compile the class Critter

Critter.cpp can be compiled but:

- ▶ It is just a model (no instances up to now)
- ▶ No main function, so it is necessary to instruct the compiler to avoid the linking stage
- ▶ `g++ -Wall -c Critter.cpp` generates `Critter.o`

A Test Program

- ▶ `testcritter.cpp`
- ▶ Putting all together:
`g++ -c testcritter.cpp`
`g++ testcritter.o Critter.o -o testcritter`
- ▶ Could also be done by just one command:
`g++ testcritter.cpp Critter.cpp -o testcritter`
- ▶ Execute:
`./testcritter`

Some Comments on `testcritter.cpp`

An instance of type `Critter` has been created

- ▶ Classes define Abstract Data Types (ADT)
- ▶ Once defined, they are types as language defined types, so it is possible to pass them as parameters to functions, declare pointers to ADT, etc.

How to Invoke Methods

- ▶ Public methods can be invoked by using the selection operator, which is a dot

```
Critter c;  
c.setName("Gremlin");
```

- ▶ Methods must be applied to instances and not to classes

```
Critter.setName("Gremlin"); // wrong!
```

- ▶ With the notable exception of static elements (to be covered later)
- ▶ Method invocation evokes procedure call

How to Access Data Members

- ▶ With the selection operator it is also possible to access (read write) data members, provided they are accessible

```
Critter c;
```

```
c.name = "Bitey";    // wrong: private
```

- ▶ Note the similarity with the selection operator used to access a C `struct`

Specifying the Definition of a Class

Methods defined in the header file are usually implemented in a different source file

- ▶ Include the header (the compiler needs to know the shape of a class before checking methods)
- ▶ When defining a method specify the name of the class it belongs to:

```
void Critter::setName(string name) { ... }
```

- ▶ There can be more methods called `setName` in different classes, so it is necessary to specify which one it is being defined

Defining a Method

When implementing a method it is not necessary to use the selection operator to call methods of the same class or to access data members

- ▶ Access defaults to the local instance

```
1 void Critter::setName(string newname) {  
2     name = newname;  
3 }
```

More on Methods

- ▶ There is a strong correspondence between method calls and function calls
- ▶ A method:
 - ▶ Can accept parameters
 - ▶ Returns a typed value to the caller
 - ▶ Can call functions and other methods, including itself
 - ▶ Can include iterative cycles, local variables declaration, etc.
- ▶ A method is allowed to access data members and other methods in its class

Instances of the Same Class

- ▶ Instances of the same class have the same set of data, but they are replicated so that they do not overlap

```
1  Critter a, b;  
2  a.setHunger(1);  
3  b.setHunger(4);  
4  cout << a.getHunger() << " "  
5      << b.getHunger();
```

will print

1 4

- ▶ a and b have a different memory space, so their modifications are independent

When Should Data Members be `public`?

- ▶ The interface of a class should be *minimal*
 - ▶ This gives least commitments in what you should keep untouched in order to avoid modifying client code
- ▶ Exceptions: if you need to access a data very frequently, the use of setter and getter methods may result in a bottleneck (after all it is a function call)
- ▶ In those cases you could consider to make a data member `public` (but you can also declare the method as `inline`)

CH-230-A

Programming in C and C++

C/C++

Tutorial 11

Dr. Kinga Lipskoch

Fall 2019

C++ References (1)

A reference is a constant pointer which is automatically dereferenced and that has to be initialized when it is created

- ▶ Constant means that it cannot be modified to reference a different entity
 - ▶ But you can of course modify what it is pointing to
- ▶ Cannot reference NULL

```
1  int a = 3;
2  int &reference = a;
3  reference++;
4  cout << a;           // prints 4
```

C++ References (2)

- ▶ References create a synonym (alias)
 - ▶ Previous example: acting on a reference is the same that acting on the variable `a`
- ▶ The first use of references is for creating functions and methods having out parameters
 - ▶ Indeed C++ references can be used even if you do not exploit the object-oriented capabilities of the language
- ▶ `outparameters.cpp`

outparameters.cpp

```
1  #include <iostream>
2  using namespace std;
3  /* This function takes two parameters. Only modifications
4     done on the first one are visible outside. */
5  void oldStyle(int *outval, int inval)
6  {
7     cout << "Inside oldStyle" << endl;
8     inval++;
9     (*outval)++;           // need to dereference
10 }
11 /* Also this function takes two parameters. Again, only
12    modifications done on the first one are visible outside. */
13 void newStyle(int &outval, int inval)
14 {
15     cout << "Inside newStyle" << endl;
16     inval++;
17     outval++;             // no need to dereference
18 }
19 int main(int argc, char** argv)
20 {
21     int a = 0, b = 0;
22     cout << a << " " << b << endl;
23     oldStyle(&a, b);       // needs to take the address
24     cout << a << " " << b << endl;
25     a = b = 0;             // reset to initial values
26     newStyle(a, b);        // no specific syntax to pass the parameter
27     cout << a << " " << b << endl;
28     return 0;
29 }
```

Passing const Object References

The usual way to pass an input parameter to a function or method, is to pass it as a `const` reference

- ▶ Improved efficiency + cannot modify
 - ▶ No need to create a temporary copy of the object
- ▶ No need to define a copy constructor (more soon)

```
1 void method(const string& byvaluepar) {  
2     // use it as a constant object  
3 }
```

- ▶ All previous examples should be rewritten according to this indication

Passing Objects by Reference

- ▶ Another case: use a reference when you wish to pass an object by reference, i.e., if modifications have to be seen outside

```
1 void modifyString(string& tomod) {  
2     tomod.assign("new value");  
3     // non const as modification has to be seen  
4 }
```

- ▶ The use is consistent with basic data types

Dynamic Memory Allocation

- ▶ C++ has an operator for dynamic memory allocation
 - ▶ It replaces the use of the C `malloc` function
 - ▶ Easier and safer
- ▶ The operator is called `new`
 - ▶ It can be applied both to user defined types (classes) and to native types

Using `new` for Predefined Data Types

- ▶ The operator returns a pointer to a specified type
- ▶ It automatically calculates the amount of memory necessary
 - ▶ It only requires the type and the number of "objects" to hold
- ▶ `newarrays.cpp`
- ▶ Note: same syntax for pointers to different types (no casting needed)

Dynamic Memory Deallocation

- ▶ As `malloc` has the companion function `free`, the operator `new` is coupled with the operator `delete`, which removes an object from memory
- ▶ `delete` requires the address of the object(s) to be deleted from the memory

```
1  int *a, *b;  
2  a = new int;  
3  b = new int[40];  
4  delete a;  
5  delete []b;
```

More on delete

- ▶ When `delete` is called to remove an object, the destructor is invoked before removing it
- ▶ Calling `delete` twice (or more) on the same object will result in an undefined behavior
- ▶ Calling `delete` on a `NULL` pointer will do nothing
 - ▶ Thus it could be advisable to set a pointer to `NULL` after calling `delete` (further `delete` will have no effect)
- ▶ Do not mix calls to `new` / `delete` with `malloc` / `free` – similar purpose, but predictably very bad result

new and delete for Arrays of Objects

- ▶ It is possible to dynamically create arrays of objects (instances of classes)
 - ▶ There must be a default (empty) constructor for the class (which will be called for every element of the array)
 - ▶ An array of objects must be explicitly deleted, by using the following syntax
`delete []ptr;`
 - ▶ In this way the compiler is able to call the destructor for every element before freeing memory
 - ▶ There is no need to specify how many elements
- ▶ `studentsrevised.cpp`

Even More on delete

- ▶ When you create objects via `new`, you should destroy them via `delete`
 - ▶ All the memory you get from the operating system should be returned
 - ▶ Again, your programs must avoid memory leaks
- ▶ Most of the bugs in early stage are due to bad / misplaced calls of `delete`
 - ▶ Many memory related errors cause severe problems

Constants (1)

- ▶ As in C, the keyword `const` is used to define values that do not change
- ▶ In C++ the use of constants is wider
 - ▶ Constants should be used instead of the preprocessor `#define` directive

```
1  // avoid #define SIZE 100
2  const int SIZE = 100;
3  // use this instead
```

- ▶ Why? Preprocessor directives can hide bugs which are nasty to find
- ▶ Constants can be inserted into header files, name clashes will be detected by the compiler

Constants (2)

- ▶ Methods or method parameters can be declared as `const`
 - ▶ Does not add information to the outside, but rather force the compiler to check that no modifications are attempted
 - ▶ Useful when dealing with temporarily generated objects
 - ▶ Useful for efficient parameter passing (more soon)
- ▶ `constantparameters.cpp`

const Objects

- ▶ Again, as classes are types, it is possible to declare `const` objects or to declare a method which accepts a `const` object
 - ▶ The syntax is the same
- ▶ For a `const` object it is not possible to modify its public data members

Constant Methods

- ▶ A constant method is a method which does not alter the object. Thus
 - ▶ It cannot modify data members
 - ▶ It may call only other `const` methods
- ▶ Constant methods are the only methods which can be called for constant objects
- ▶ `constclass.cpp`

Multiple Inclusions

- ▶ Class declarations go to header files
- ▶ Header files will be included in all the .cpp files that need their declarations
- ▶ What if a header file is included twice?
 - ▶ A repeated class declaration is an error
 - ▶ But a repeated function declaration is not (as long as the declarations are the same)
- ▶ Should the programmer take care of not including the file twice?
 - ▶ Almost impossible in big projects

Conditional Compilation

- ▶ The preprocessor can be used to avoid multiple inclusions
- ▶ The `#ifdef`, `#ifndef`, `#else`, `#endif` directives allow to exclude some parts of the code according to specified conditions
- ▶ They are to be used with the `#define` that you already know

The Structure of a Header File

```
1  /* Student.h */
2  #ifndef _STUDENT_H
3  #define _STUDENT_H
4  class student {
5      /* your class declaration */
6  };
7  #endif // this matches the initial #ifndef
```

How Does This Work?

- ▶ The first time the header is included the symbol `_STUDENT_H` is not defined
 - ▶ Then the class declaration is compiled and then the symbol is defined
- ▶ In all the subsequent inclusions the symbol is already defined and then the class declaration is skipped
- ▶ You must always protect (or guard) your header files with this mechanism

CH-230-A

Programming in C and C++

C/C++

Tutorial 12

Dr. Kinga Lipskoch

Fall 2019

Namespaces (1)

- ▶ While developing large projects, the risk of running into a name clash is high
 - ▶ Multiple programmers could use the same names for their classes, functions, etc. At the linking stage, name collisions can arise
 - ▶ You can have the same problem when using third party developed libraries
- ▶ Solutions found in the past, consisting on appending specific prefixes, are not appealing

Namespaces (2)

- ▶ A namespace introduces a further level of code protection
- ▶ Elements belonging to the same namespace can refer to each other without any special syntax
- ▶ Elements in different namespaces can refer to each other just by using a designed syntax
 - ▶ They have to explicitly declare that they are referring to a different namespace

Creating a Namespace

- ▶ A namespace is created using the `namespace` keyword at the file level

```
1 namespace CPPcourse {  
2     void f1() { ... }  
3     class class1 { ... };  
4 }
```

- ▶ Namespace declaration can be split over multiple files without creating redefinition problems
- ▶ `namespace.h`
- ▶ `namespace.cpp`

Using Names from a Namespace

Three ways:

- ▶ Import the whole namespace
`using namespace CPPcourse;`
- ▶ Import a specific name from a namespace

```
1 using CPPcourse::FirstExample;  
2 FirstExample a("Try this");
```
- ▶ Using complete name specification
`CPPcourse::FirstExample a("Try this");`

Examples Revised

- ▶ Then in all former examples
`using namespace std;`
was introduced to use standard C++ classes, which are declared in the `std` namespace
- ▶ In header files we have used full name specification
`std::string name;`
- ▶ Never use the `using` directive in a header file, use full name qualification instead
 - ▶ While writing a header file you do not know what your potential client will need in terms of namespaces

Final Remarks on Namespaces

- ▶ If a namespace's name is too "awkward" to use, it is possible to create an **alias**
`namespace shortName = AliasForANameTooLongToBeUsed;`
- ▶ From now on we can use `shortName` instead of the alias it points to
- ▶ Namespaces can be nested
- ▶ More details in Eckel's book (chapter 10)

static Data Members

- ▶ A `static` data member is shared among all the instances of a class
 - ▶ It creates a sort of class variable
- ▶ It exists even if no instances are created
- ▶ Storage must be explicitly allocated outside of class definition
- ▶ Can be useful to define class constants
 - ▶ Using `const` as modifier for a data member does not yield the desired results (as class constant)
- ▶ `staticexample.cpp`

static Methods

- ▶ Also methods can be declared as `static`
- ▶ Static methods can access only static data members and can call only static methods
- ▶ Static methods can be called referring to an instance or to the class
 - ▶ Like `static` data members they are class methods
- ▶ `staticshapes.cpp`

When Should we Use `static`?

No general rules, but some generic indications:

- ▶ When creating class level constants
- ▶ When you devise some information which belongs to the class rather than to instances
- ▶ When a method needs to access data members but it is not logically tied to a specific instance

Inline Methods (1)

- ▶ C is well appreciated as it is an efficient language
 - ▶ The UNIX operating system relies on C
- ▶ C++ cannot give up C efficiency
- ▶ Inline methods are designed to improve the performance of C++ programs
 - ▶ No semantic alterations w.r.t. non-inline methods

Inline Methods (2)

- ▶ A method call is equivalent to a procedure call
 - ▶ Push arguments onto the stack (or register)
 - ▶ Execute a CALL-like instruction
 - ▶ Execute function/method code and then return
 - ▶ Stack cleanup
- ▶ For small methods the overhead of the call could take more time than code execution
 - ▶ Think for example of getter or setter methods, where you have just one instruction as body
 - ▶ Moreover those methods are likely to be called frequently

Inline Methods (3)

- ▶ An `inline` function is expanded in place, rather than called
 - ▶ Instead of a regular call, function code is directly inserted
- ▶ You trade off speed for size
 - ▶ No call overhead, but your code could grow as the body of the function will be copied many times
- ▶ Good candidates for being `inline` are short methods that are frequently called

Inline Methods (4)

How to create inline methods - two possibilities:

- ▶ Put the definition of the method inside the class declaration
`inlineinside.h` `inlineinside.cpp`

- ▶ Use the keyword `inline` and write the definition outside the class declaration

`inlineoutside.h` `inlineoutside.cpp`

Put the `inline` function definition in the same header file where the class is declared

Inline Methods: How Do they Work?

- ▶ When the compiler finds the definition of an inline method it stores its signature and its code in its symbol table
- ▶ When it finds a call to an inline method it checks type correctness and replaces/copies the code
 - ▶ C preprocessor macros offered similar advantages, but no type checking was enforced
 - ▶ Nasty to find bugs which could be generated
 - ▶ Preprocessor macros have no concept of scoping

Inline Methods: Final Remarks

- ▶ Not everything declared as `inline` by the programmer will necessarily be inlined by the compiler (`inline` is just a hint)
 - ▶ If a method includes loops it is unlikely that it will be expanded
- ▶ Defining inline methods outside class declaration increases code readability
- ▶ Multiple inclusions of headers with inline methods will not result in redefinition problems

The Implicit Pointer `this`

- ▶ The reserved keyword `this` is a pointer to the current instance of a class
- ▶ `this` is silently passed by the compiler as an argument to every method call
 - ▶ Except of course to static methods. Why?
- ▶ `thisexample.cpp`
- ▶ Will be very useful when implementing overloaded operators

friend Functions

It is possible to "break" the protection mechanism, i.e., to let a class or a function access non-public data members of a class

- ▶ Is this needed? Sometimes yes, if getting through the getter and setter methods becomes difficult to manage
- ▶ We will see that this will be very important while redefining operators
- ▶ Be aware: when using `friend` elements, you break the information hiding mechanism
- ▶ Do not misuse it

friend: How to Create

- ▶ In the class declaration, declare a "method" with the `friend` modifier
 - ▶ That indicates a function which can access class data members
 - ▶ The function has to be defined later, but remember that it is not a method
 - ▶ `friendexample.cpp`
- ▶ It is also possible to create friend classes, i.e., classes which can access private data of other classes

CH-230-A

Programming in C and C++

C/C++

Tutorial 13

Dr. Kinga Lipskoch

Fall 2019

Abstract Classes (1)

- ▶ It should be evident that classes near to the root of the hierarchy are seldom instantiated
 - ▶ Very general but also very unspecialized
- ▶ Some classes are introduced just to define common behaviors, but are not self sufficient
 - ▶ Think of the class shape in one of the former examples
- ▶ Those classes are useful only for abstraction

Abstract Classes (2)

- ▶ Abstract classes define a set of methods to be shared by a derived class but are not yet implemented
 - ▶ Implementation will be defined in a derived class
 - ▶ Virtual mechanism plays a fundamental role
- ▶ A pure virtual method is a method declared as:
`virtual void something() = 0;`
- ▶ A class having one or more pure virtual methods is **abstract**

Abstract Classes (3)

- ▶ Abstract classes cannot be instantiated
- ▶ Abstract classes can also include non-pure virtual methods
- ▶ Methods and functions can accept pointers to abstract classes
 - ▶ This is their main use: through virtual calls generic code is developed

Shapes Example Revised

- ▶ In the shape example the shape class has not actually represented a shape (instance), but rather collected some data common to all shapes
- ▶ Therefore, Shape is a good candidate to be an abstract class
 - ▶ `shapesrevised.h`
 - ▶ `shapesrevised.cpp`
 - ▶ `testshapesrevised.cpp`

Virtual Destructors?

Destructors are almost always `virtual`

- ▶ If you are manipulating objects via pointers to the base class, then the base class should define its destructor as `virtual`
- ▶ Otherwise just the base class destructor is called
- ▶ Recall that destructors are called from bottom to up
- ▶ Destructors can be pure `virtual`
 - ▶ There are some subtle details concerning this aspect (see Eckel's book, chapter 15)

Virtual Constructors?

- ▶ You cannot have virtual constructors
 - ▶ Remember that constructors are called from the base to the leaves of the derivation tree
- ▶ Inside a constructor you can call a virtual method, but this will execute the local version
 - ▶ No downsearch is performed, as the assembly of the object is still being performed and elements belonging to derived classes are not guaranteed to be properly initialized

Overloading Operators for Casting

- ▶ It is possible to create operators for converting a type to another, thus performing a sort of casting
`operatorconversion.cpp`
- ▶ This can also be done by implementing an ad-hoc constructor taking the type we want convert from
`constructorconversion.cpp`

The explicit Keyword

- ▶ If a constructor is declared with the explicit modifier, it will be used for type conversion only if the typename is explicitly inserted
- ▶ Then it is possible to choose which kind of conversion will take place: constructor driven or operator driven
`explicitconversion.cpp`

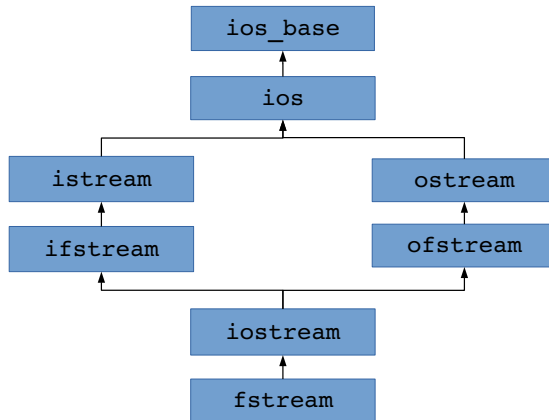
Streams

- ▶ A stream is a flow of data from a source to a destination
 - ▶ Widely used concept in Unix
 - ▶ Think to water flowing in a pipe
- ▶ Standard C++ provides classes for handling streams of data connected to the console or to files
 - ▶ Common interface: learn once use everywhere

iostreams

- ▶ You already used them
- ▶ The instances `cin`, `cout` and `cerr` are declared in the header files included in `<iostream>`
- ▶ Exceptional use due to their wide use
 - ▶ Preprocessor directives for conditional compiling avoid multi-declaration problems
- ▶ Extractors and inserters are overloaded operators designed to work with different data types
 - ▶ Consider to overload them to work with your own developed classes

Class Hierarchy



Output Streams and the Inserter Operator <<

- ▶ Operator << has been overloaded to work with all language data types and many classes
 - ▶ It sends data to an output stream (ostream)
 - ▶ Inserters can be concatenated
 - ▶ Additionally, manipulators can modify the output
 - ▶ endl, flush, hex, oct, dec
- Example: `cout << hex << "0x" << 34 << flush;`

The << Operator

- Converts internal data type into sequence of ASCII characters

```
ostream& operator<<(const char *)
```

```
ostream& operator<<(char)
```

```
ostream& operator<<(int)
```

```
ostream& operator<<(float)
```

```
ostream& operator<<(double)
```

- Returns reference to ostream

Input Streams and the Extractor Operator >>

- ▶ The operator >> has been overloaded to work with predefined language data types
 - ▶ It gets data from an input stream (`istream`)
- ▶ Extractor stops reading when it finds a whitespace
- ▶ The manipulator `ws` removes leading and trailing white space from an `istream`

Line Oriented Input

- ▶ Istreams provide two methods to get a whole line of text:
 - ▶ `get()` get the text but do not remove the delimiter
 - ▶ `getline()` get the text and remove the delimiter
 - ▶ Both accept three parameters: char buffer to store data, buffersize and terminator character
 - ▶ Default value of terminator is `'\n'`
- ▶ It can be useful to grab input as a char sequence and then convert it using C functions

Raw I/O

- ▶ Binary files: images, audio, self-defined formats, etc.
- ▶ Raw I/O member functions are used to write/read binary data to/from streams
 - ▶ Istreams:
 - ▶ `read(char *, int)`
 - ▶ `gcount()` returns the number of characters extracted
 - ▶ Ostreams:
 - ▶ `write(char *, int)`

The State of a Stream

The following member functions can be used to investigate on the state of a stream:

- ▶ `good()` true if `goodbit` is the current state
- ▶ `eof()` true if `endoffile`
- ▶ `fail()` true if `failbit` or `badbit` set
- ▶ `bad()` true if `badbit` set
- ▶ `clear()` set state to `goodbit`

File Streams

`ifstream` and `ofstream` classes can be used to connect a stream to a file

- ▶ Just provide the name of the file as a parameter to the constructor
- ▶ You do not need to open or close the file (up to constructor and destructor)
- ▶ Classes are declared in the `fstream` header file
- ▶ `filestream.cpp`

Open Mode Flags

Flag	Function
<code>ios::in</code>	Open as input
<code>ios::out</code>	Open as output
<code>ios::binary</code>	Open in binary mode
<code>ios::app</code>	Open for appending
<code>ios::ate</code>	Open and go at end
<code>ios::trunc</code>	Open and delete the old if present

Overloading Extractors and Inserters for your Types

- ▶ It can be useful to overload `<<` and `>>` to dump and/or read classes instances to streams
 - ▶ For example to save/retrieve the state of the application to/from a file
- ▶ Add an overloaded operator `<<` or `>>` definition to the class
 - ▶ Should be friend
 - ▶ Returns an `ostream/istream` reference
 - ▶ Should take an `istream/ostream` reference and a (`const`) reference to the class as parameters
 - ▶ `overloadedstream.cpp`