CH08-320142

Object-oriented Programming I

OOP I

Lecture 1 & 2

Dr. Kinga Lipskoch

Fall 2017

Who am I?

- PhD in Computer Science at the Carl von Ossietzky University of Oldenburg
- University lecturer at the Department of Computer Science
- Joined Jacobs University in January 2013
- Office: Research I, Room 94
- Telephone: +49 421 200-3148
- E-Mail: k.lipskoch@jacobs-university.de
- Office hours: Mondays 10:00 - 12:00

Course Goals

- Learn the basic aspects of object-oriented programming
- Learn the basic details of the C++ programming language
- Write and test simple programs
- "Hands on": not just theory, but practice lab sessions to apply what you have learned in the lectures

Course Details

- 3 weeks = 24 hours
- Every week will consist of
  - 2 lectures: Thu/Fri afternoon, 14:15 - 16:15
  - 2 lab sessions: Thu/Fri afternoon, 16:15 - 18:30
- During each lab session you will have to solve a programming assignment sheet (consisting of multiple exercises) related to the corresponding lecture

Course Resources

Textbooks:

- Frank B. Brokken, C++ Annotations Version 10.8.1
  http://www.icce.rug.nl/documents/cplusplus/
- Bruce Eckel: Thinking in C++, Volume I: Introduction to Standard C++, Second Edition, 2000, Prentice Hall PTR

Slides, program assignments, and code will be posted on Grader

- https://grader.eecs.jacobs-university.de/

Grading Policy

- 35% – average grade of the assignments
- 65% – grade of the final exam
- 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

Programming Assignments

- There will be presence assignments that need to be solved in the lab
- Other assignments are due on the following Tuesday and Wednesday morning at 10:00 h
- These assignments are structured in a way that you can solve them during the lab session
- Solutions have to be submitted via web interface to `https://grader.eecs.jacobs-university.de`
- Assignments are graded by the TAs

Grading Criteria for Assignments

- Grading criteria
  https://grader.eecs.jacobs-university.de/courses/320142/2017_2r2/Grading-Criteria-C++.pdf
- Weekly tutorials given by the TAs to help
- Grand tutorial a few days before the exam

Lab Sessions

- TAs will be available to help you in case of problems
- Optimize your time: solve the assignments during lab sessions
- Do not copy the solutions for the assignments, it is not only illegal, but you will certainly fail the written final exam without practice in programming

Missing Homework, Quizzes, Exams according to AP

- https://www.jacobs-university.de/sites/default/files/bachelor_policies_v1.1.pdf (page 9)
- Illness must be documented with a sick certificate
- Sick certificates and documentation for personal emergencies must be submitted to the Student Records Office by the third calendar day
- Predated or backdated sick certificates will be accepted only when the visit to the physician precedes or follows the period of illness by no more than one calendar day
- Students must inform the Instructor of Record 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 Instructor of Record in order to clarify the make-up procedure
- Make-up examinations have to be taken and incomplete coursework has to be submitted by no later than the deadline for submitting incomplete coursework as published in the Academic Calendar

Syllabus of the Course

- Introduction to objects and classes
- Overloading and references
- Constructors, destructors, passing objects
- Copy constructors
- Dynamic object creation
- Conditional compilation, namespaces, inline functions
- Static members
- Inheritance
- Function and operator overloading
- Polymorphism

Programming Environment (1)

- C++ is available on practically every operating system
- Commercial
  - Microsoft C++, Intel C++, Portland
- As well as free compilers available
  - g++
- g++ available on many platforms
- g++ `6.3.0` will be used on Grader

Programming Environment (2)

- We will refer to the Unix operating system and related GNU tools (`g++`, `gdb`, some editor, IDE, etc.)
- Install a C++ compiler on your notebook
- IDE (Integrated Development Environment) may be helpful
- CodeLite `http://www.codelite.org/`
  - powerful IDE
  - runs on Linux, Mac, Windows

Programming Environment (3)

- Linux
  - preferred environment
  - prepare to get to know Unix/Linux
  - g++ should be already on your machine
  - you can install CodeLite from https://downloads.codelite.org/
  - any other IDE or editor is also fine

Programming Environment (4)

- Mac OS
  - it is a Unix system as well
  - XCode (is on MacOS DVD)
  - if the DVD is not available you might need to register as Apple Developer to be able to download XCode
  - CodeLite `http://downloads.codelite.org/`

Programming Environment (5)

- Windows
  - CodeLite includes a C++ compiler
    `http://downloads.codelite.org/`
  - choose: CodeLite Installer 10.0 for Windows
  - download and install it
  - includes `gdb` and `g++`
  - Alternatives: Visual C++ Express (free of charge), Eclipse, NetBeans

Programming Environment (6)

- You will upload your solutions to Grader, where your source code will be automatically compiled
- Your programs must compile without any warning with g++
- On your local machine turn on all warnings:
  `g++ -Wall -o file file.cpp`

Agenda Week 1

- What is C++?
- What is object-oriented programming?
- Differences to C
- Function overloading
- Inline functions
- Strings
- Introduction to objects and classes
- Information hiding
- The interface of an object (data members and methods)

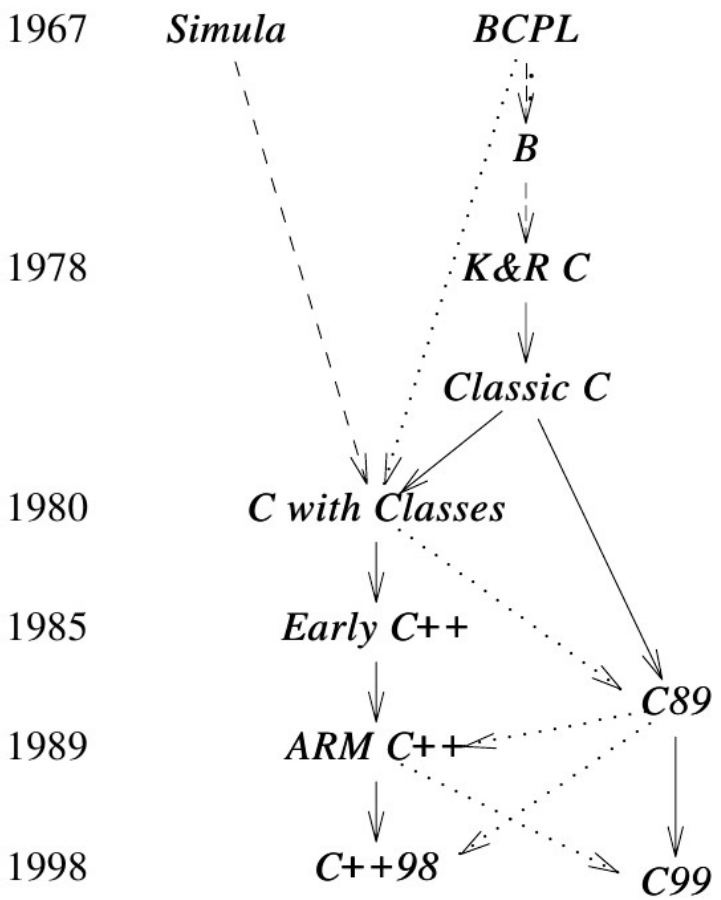
Brief History of C++

C++ is an object-oriented Extension to C

- 1970's B. Kernighan and D. Ritchie create C language at Bell Labs
- 1980's Bjarne Stroustrup creates C successor
  - C with classes
  - Using some pre-compiler C++ code was translated into C and compiled then
- 1998 C++ finally becomes ISO standard (ISO/IEC 14882:1999)
- 2003 C++03 (ISO/IEC 14882:2003)
- 2005 Technical Report (TR1) (many extensions)
- 2011 C++11 (ISO/IEC 14882:2011)
- 2014 C++14 (ISO/IEC 14882:2014(E))
- 2017 C++17 (Draft International Standard in March 2017)

The Way to C++

- Simula has been developed in the 1960s at the Norwegian Computing Center in Oslo, by Ole-Johan Dahl and Kristen Nygaard
- Based on ALGOL 60 it is considered to be the first object-oriented programming language
- Here ARM is not the now more and more famous processor, but Bjarne Stroustrup's ARM (Annotated Reference Manual) which was the de-facto standard, since no official standard existed



C++

C is almost a complete subset of C++

- possibility to clean up some things, but
- compatibility to millions lines of code considered more important
- some differences remain
- char constants:
  - 1 byte in C++ (actually `char`)
  - 4 bytes in C (actually an `int`)
- older C++ standards required prototypes (from standard C11 no implicit need for prototypes)

C++ and OOP

- C++ is an improved version of C, which includes constructs supporting Object-Oriented Programming (OOP)
  - Improved version of C means that
    - Almost all the code written in C will be compiled by a C++ compiler
    - C++ fixes some C "holes"
    - This may need minor code adjustments
    - Libraries written for C can be used for C++
    - The meaning of C constructs is not altered
- C++ is a multi-paradigm language
- It does not force you to exclusively use OOP, but it lets you mix OOP and classic imperative programming (differently from other pure OO languages like Java, Smalltalk, etc.)

What is OOP?

- OOP is a programming paradigm, i.e., a way to organize the solution of your computational needs
  - But definitely not the only way
- Pros and cons
  - Increases productivity at very different levels (code reuse, generic programming, hopefully simpler design process)
  - OOP is not the solution for every need
  - OOP could be not easy to learn and takes a long time before one can master it properly

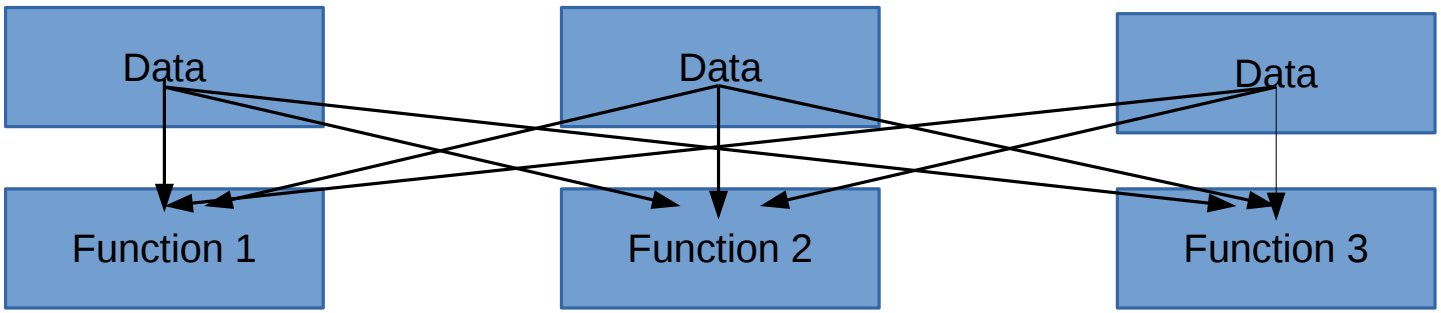
Imperative Programming vs. OOP (1)

Imperative programming relies on a top-down approach

- Iteratively divide the given problems into simpler sub-problems (simpler from a logic point of view)
- When a sub-problem is simple enough, code it
  - Again, simple means simple from a logic point of view, and not from the point of view of the number of lines of code
- Interactions between sub-problems happen by mean of function calls
  - You have seen this while programming in C

Problems with Imperative Programming using Functions

- `account.c`
- 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

Cornerstones of OOP

- Data abstraction (hiding of information)
- Encapsulation (hiding of internal workings)
- Inheritance (relation between class and subclass)
- Polymorphism (ability to use the same syntax for objects of different types)

Imperative Programming vs. OOP (2)

- OOP follows a bottom-up process
- Given a problem you should first ask yourself
  - Which are the entities which characterize this problem?
  - What are their characteristics? (member data)
  - How do they interact? (methods/functions)
  - Entities (objects) interact by mean of messages exchanged between them
    - A message is a request to execute a method

OOP Allows Better Modeling

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

OOP Characteristics (by Alan Kay, inventor of Smalltalk)

1. Everything is an object
2. A program is a collection of objects exchanging messages
3. Each object has a memory made by other objects
4. Every object has a type
5. Objects of the same type can receive the same messages

The Early History of Smalltalk, Alan C. Kay ACM SIGPLAN Notices Volume 28, No. 3, March 1993 Pages 69–95

http://stephane.ducasse.free.fr/FreeBooks/SmalltalkHistoryHOPL.pdf

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 }
```

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)

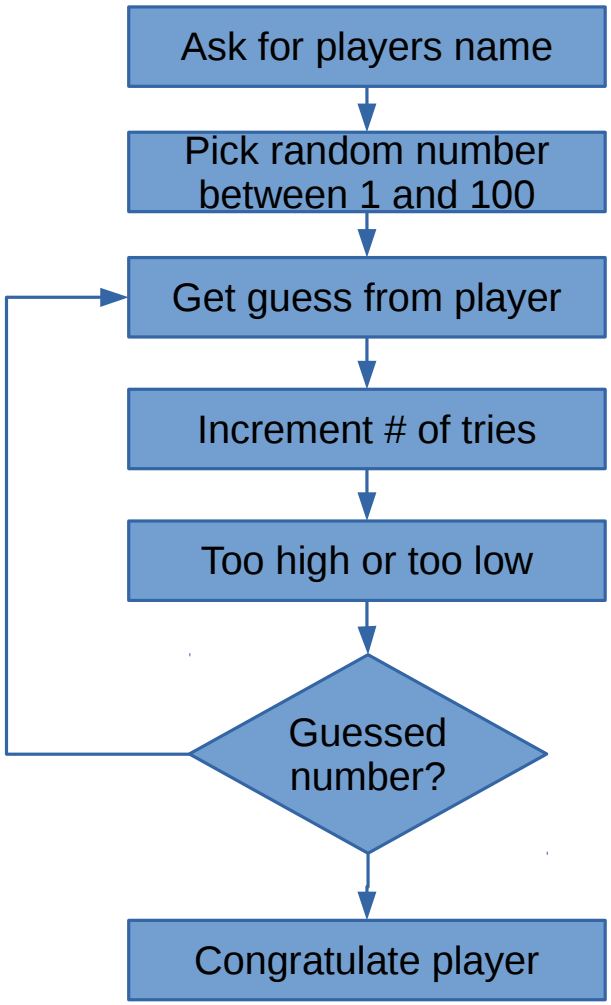
- There is one `getline` function and one `getline` method
- The function `getline` is a global function and reads a string from an input stream

```
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 }
```

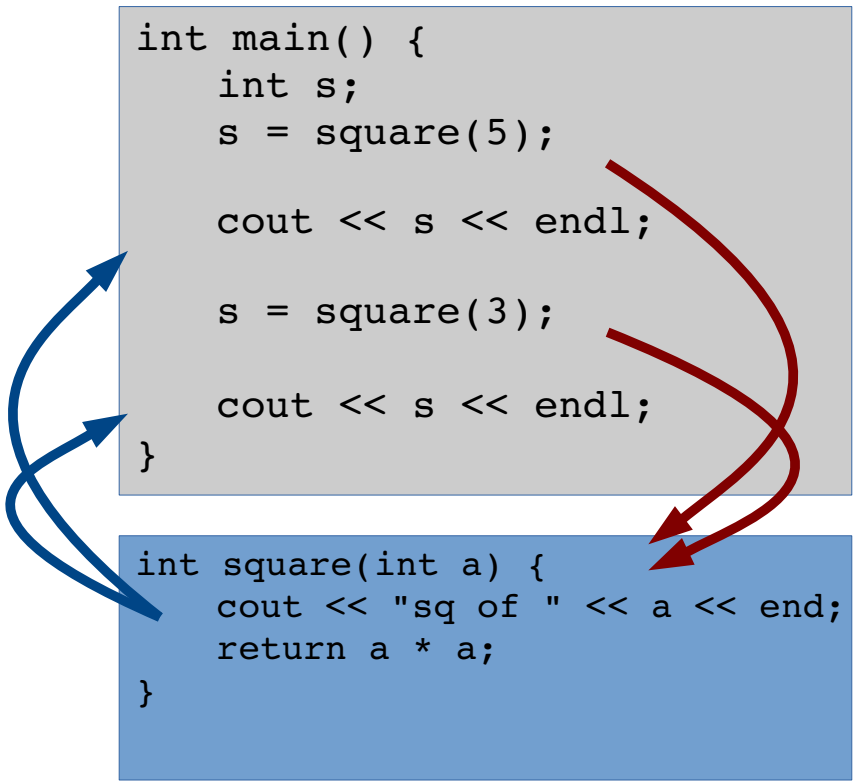
C++ Extensions to C

- 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;
    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;
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 pt`
  `// (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`

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 tescritter.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 }
```