CH-230-A

Programming in C and C++

C/C++

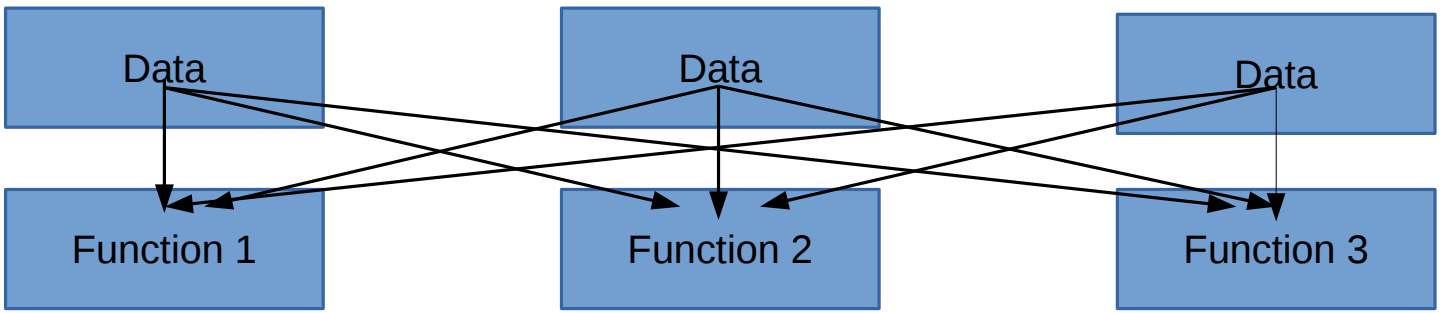
Tutorial 9

Dr. Kinga Lipskoch

Fall 2019

Problems with Imperative Programming using Functions

- `account.c`
- 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

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

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)

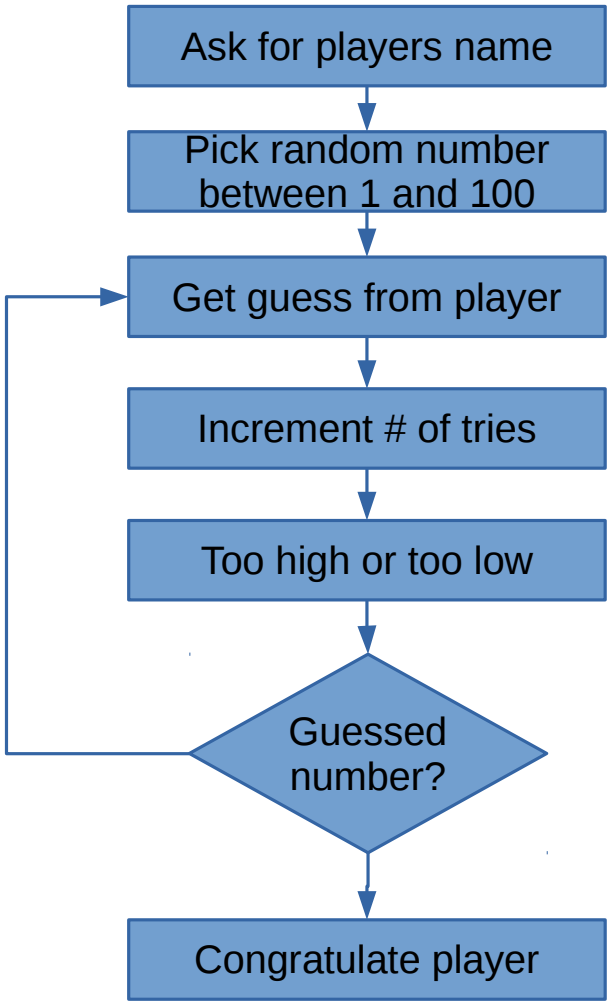
- There is one `getline` function and one `getline` method
- The function `getline` is a global function and reads a string from an input stream

```
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 }
```

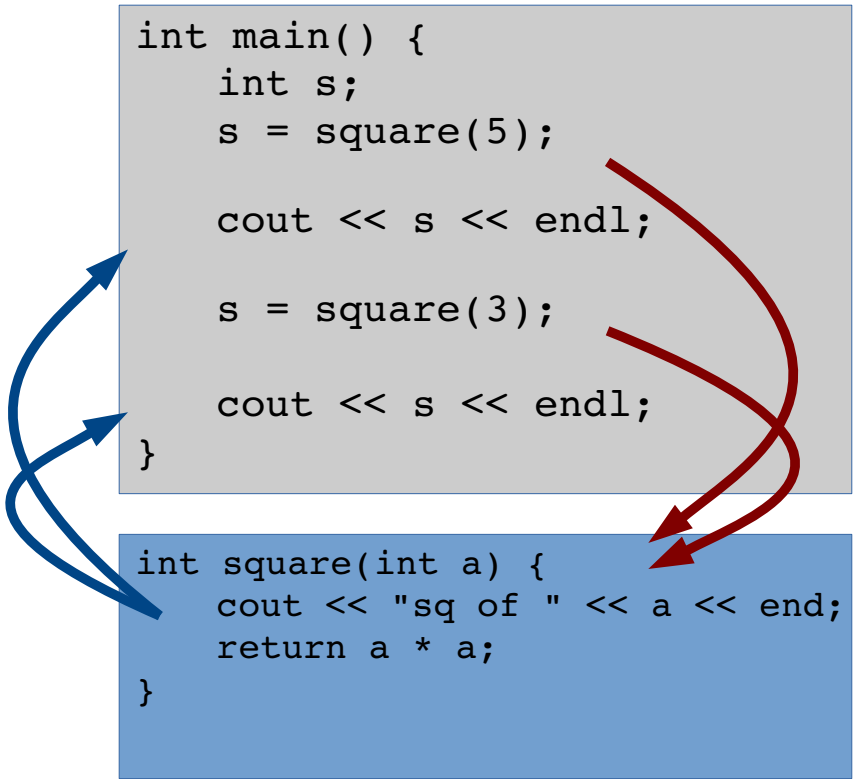
C++ Extensions to C

- 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`