Conventions

- Name conventions
- Variable declaration
- File conventions

- `for(int j=0; j<dim; ++j) { ... }`

Name conventions

- Local variables
  - `lowercase_with_underscores`
- Global variables
  - `underscore_also_at_the_end_`
- Preprocessor constants
  - `UPPERCASE_WITH_UNDERSCORES`
- In header files
  - `_NAME_OF_THE_CLASS_`
- Functions / methods
  - `firstWordLowercaseNoUnderscores`
- Structures / Classes
  - `FirstWordUppercaseNoUnderscores`

Variable declaration

```
 1 #include <iostream>
 2 using std::cout;
 3 using std::endl;
 4
 5 int main() {
 6   double sum = 0;
 7
 8   for (int j=1; j<=100; ++j) {
 9     sum = sum + j;
10   }
11
12   cout << sum << endl;
13 }
```

- In C++, variables can be declared everywhere
  - Risk: Code can become unclear!
- Convention: At the beginning of the block
  - It is much more clear!
- Two exceptions:
  - Counter variables in `for` loops
    - Usually directly declared in the loop
    - Usually local variable (used only in the loop)
  - `assert` before a declaration is fine
- Code above computes $\sum_{j=1}^{100} j = 5050$
  - Counter `j` exists only in lines 8–10

Bad code 1/2

```
 1 #include <stdio.h>
 2
 3 int main() {
 4   int a[2] = {0, 1};
 5   int b[2] = {2, 3};
 6   int c[3] = {4, 5};
 7   int i = 0;
 8
 9   printf("a = (%d,%d), b = (%d,%d), c = (%d,%d), i = %d\n",
10          a[0], a[1], b[0], b[1], c[0], c[1], i);
11
12   a[i] = b[i] = c[i];
13
14   printf("a = (%d,%d), b = (%d,%d), c = (%d,%d), i = %d\n",
15          a[0], a[1], b[0], b[1], c[0], c[1], i);
16
17   c[0] = 9;
18   i = 0;
19
20   a[i] = b[i++] = c[i];
21
22   printf("a = (%d,%d), b = (%d,%d), c = (%d,%d), i = %d\n",
23          a[0], a[1], b[0], b[1], c[0], c[1], i);
24
25   return 0;
26 }
```

- Bad style: Not all lines are easily understandable!
- **Be careful:** Behavior of `b[i++]` is undefined!

  ```
  warning:  unsequenced modification and
  access to 'i'
  ```

- **Resulting output:**

  ```
  a = (0,1), b = (2,3), c = (4,5), i = 0
  a = (4,1), b = (4,3), c = (4,5), i = 0
  a = (4,9), b = (9,3), c = (9,5), i = 1
  ```

Bad code 2/2

```
 1 #include <cstdlib>
 2 #include <cstdio>
 3 int main(){
 4   int i=0;
 5   int n=5;
 6   int* a=(int*)malloc((n+1)*sizeof(int));
 7   int*b=(int*)malloc((n+1)*sizeof(int));
 8   int *c=(int*)malloc((n+1)*sizeof(int));
 9   int * d=(int*)malloc((n+1)*sizeof(int));
10   while(i<n){
11     a[i]=b[i]=c[i]=d[i]=i++;}
12   printf("a[%d] = %d\n",n-1,n-1);
13 }
```

- Please write code for humans!
  - Use blank spaces before/after
    - assignments and type casting operators
    - arithmetic operations (sometimes)
    - brackets (sometimes, especially if nested)
  - Use empty lines to separate conceptually different parts of code
    - Declarations / Memory allocation / Actions
- Good code performs one action per line!
  - Avoid multiple assignments (although they are allowed in C/C++)
- Opt for count-controlled loops!
  - Condition-controlled loops usually less clear

Same code, but more readable!

```
 1 #include <cstdlib>
 2 #include <cstdio>
 3
 4 int main(){
 5   int n = 5;
 6
 7   int* a = (int*) malloc( (n+1)*sizeof(int) );
 8   int* b = (int*) malloc( (n+1)*sizeof(int) );
 9   int* c = (int*) malloc( (n+1)*sizeof(int) );
10   int* d = (int*) malloc( (n+1)*sizeof(int) );
11
12   for(int i=0; i<n; ++i){
13     a[i] = i - 1;
14     b[i] = i - 1;
15     c[i] = i - 1;
16     d[i] = i - 1;
17   }
18
19   printf("a[%d] = %d\n",n-1,a[n-1]);
20 }
```

- Please write code for humans!
  - Use blank spaces before/after
    - assignments and type casting operators
    - arithmetic operations (sometimes)
    - brackets (sometimes, especially if nested)
  - Use empty lines to separate conceptually different parts of code
    - Declarations / Memory allocation / Actions
- Good code performs one action per line!
  - Avoid multiple assignments (although they are allowed in C/C++)
- Opt for count-controlled loops!
  - Condition-controlled loops usually less clear

File conventions

- Each C++ program consists of several files
  - C++ file for the main program `main.cpp`
  - Convention: for each class used in the program
    - Header file `myClass.hpp`
    - Source file `myClass.cpp`
- Header file `myClass.hpp` consists of
  - `#include` for all needed libraries
  - Class definition
  - Method signatures (without body)
  - Comments about the methods
    - What does a method do?
    - What is its input? What is its output?
    - Specify default parameter and optional input
- `myClass.cpp` contains method implementations
- Why splitting the code into several files?
  - Clarity and readability of the code
  - Creation of libraries
- Header files begin with

```
#ifndef _MY_CLASS_
#define _MY_CLASS_
```

- Header files end with

```
#endif
```

- This approach allows for multiple linking!
- Important: Avoid `using` in header files!
  - In particular, avoid also `using std::`...

triangle.hpp

```
 1 #ifndef _TRIANGLE_
 2 #define _TRIANGLE_
 3
 4 #include <cmath>
 5
 6 // The class Triangle stores a triangle in R2
 7
 8 class Triangle {
 9 private:
10   // the coordinates of the nodes
11   double x[2];
12   double y[2];
13   double z[2];
14
15 public:
16   // define or change the nodes of a triangle,
17   // e.g., triangle.setX(x1,x2) writes the
18   // coordinates of the node x of the triangle.
19   void setX(double, double);
20   void setY(double, double);
21   void setZ(double, double);
22
23   // return the area of the triangle
24   double getArea();
25 };
26
27 #endif
```

triangle.cpp

```
 1 #include "triangle.hpp"
 2
 3 void Triangle::setX(double x0, double x1) {
 4     x[0] = x0; x[1] = x1;
 5 }
 6
 7 void Triangle::setY(double y0, double y1) {
 8     y[0] = y0; y[1] = y1;
 9 }
10
11 void Triangle::setZ(double z0, double z1) {
12     z[0] = z0; z[1] = z1;
13 }
14
15 double Triangle::getArea() {
16     return 0.5*fabs( (y[0]-x[0])*(z[1]-x[1])
17                    - (z[0]-x[0])*(y[1]-x[1]) );
18 }
```

- Creation of object code from source (option `-c`)
  - `g++ -c triangle.cpp` creates `triangle.o`
- Compilation `g++ triangle.cpp` leads to an error
  - The linker `ld` fails, because no `main` is available

```
Undefined symbols for architecture x86_64:
"_main", referenced from:
   implicit entry/start for main executable
ld: symbol(s) not found for architecture x86_64
```

triangle_main.cpp

```
 1 #include <iostream>
 2 #include "triangle.hpp"
 3
 4 using std::cout;
 5 using std::endl;
 6
 7 int main() {
 8   Triangle tri;
 9   tri.setX(0.0,0.0);
10   tri.setY(1.0,0.0);
11   tri.setZ(0.0,1.0);
12   cout << "Area = " << tri.getArea() << endl;
13   return 0;
14 }
```

- Compilation `g++ triangle_main.cpp triangle.o`
  - Creation of object code from `triangle_main.cpp`
  - Inclusion of additional object code `triangle.o`
  - Linking with inclusion of the standard library
- Use of `make` as described for C is also possible

Constructor & Destructor

- Constructor
- Destructor
- Overloading of methods
- Optional input and default parameters
- Nesting of classes

- `this`
- `ClassName(...)`
- `~ClassName()`
- Operator :

Constructor & Destructor

- Constructor = Automatic call with declaration
  - Can be used to initialize an object
  - Can be called in different ways (see, e.g., below)
  - Formally: `className(input)`
    - No output, possibly some input
    - Different constructors have different input
    - Standard constructor `className()`
- Destructor = Automatic call with lifetime end
  - Deallocation of dynamically allocated memory
  - Only standard destructor: `~className()`
    - No input, no output
- Constructors can be overloaded, e.g.,
  - Constructor of a class for vectors in $\mathbb{R}^n$
    - No input $\Rightarrow$ Vector of length 0 (standard constructor)
    - Input `dim` $\Rightarrow$ Vector of length `dim` and entries initialized with 0
    - Input `dim`, `val` $\Rightarrow$ Vector of length `dim` and entries initialized with `val`

Constructor: An example

```
 1 #include <iostream>
 2 #include <string>
 3 using std::cout;
 4 using std::string;
 5
 6 class Student {
 7 private:
 8   string lastname;
 9   int student_id;
10 public:
11   Student() {
12     cout << "Student generated\n";
13   }
14   Student(string name, int id) {
15     lastname = name;
16     student_id = id;
17     cout << "Student (" << lastname << ", ";
18     cout << student_id << ") registered\n";
19   }
20 };
21
22 int main() {
23   Student demo;
24   Student var("Praetorius",12345678);
25   return 0;
26 }
```

- Constructors have no return values (lines 11+14)
  - Name `className(input)`
  - Standard constructor `Student()` without input (line 11)
- Output

  ```
  Student generated
  Student (Praetorius, 12345678) registered
  ```

Name conflicts & pointer `this`

```
 1 #include <iostream>
 2 #include <string>
 3 using std::cout;
 4 using std::string;
 5
 6 class Student {
 7 private:
 8   string lastname;
 9   int student_id;
10 public:
11   Student() {
12     cout << "Student generated\n";
13   }
14   Student(string lastname, int student_id) {
15     this->lastname = lastname;
16     this->student_id = student_id;
17     cout << "Student (" << lastname << ", ";
18     cout << student_id << ") registered\n";
19   }
20 };
21
22 int main() {
23   Student demo;
24   Student var("Praetorius",12345678);
25   return 0;
26 }
```

- `this` gives a pointer to the current object
  - `this->` allows access to the member of the current object
- Name conflict in constructor (line 14)
  - Input variable are called like class members
  - Lines 14–16: Solution of the conflict via `this->`

Destructor: An example

```
 1 #include <iostream>
 2 #include <string>
 3 using std::cout;
 4 using std::string;
 5
 6 class Student {
 7 private:
 8   string lastname;
 9   int student_id;
10 public:
11   Student() {
12     cout << "Student generated\n";
13   }
14   Student(string lastname, int student_id) {
15     this->lastname = lastname;
16     this->student_id = student_id;
17     cout << "Student (" << lastname << ", ";
18     cout << student_id << ") registered\n";
19   }
20   ~Student() {
21     cout << "Student (" << lastname << ", ";
22     cout << student_id << ") deregistered\n";
23   }
24 };
25
26 int main() {
27   Student var("Praetorius",12345678);
28   return 0;
29 }
```

- Lines 20–23: Destructor (without input or output)
- Output

```
Student (Praetorius, 12345678) registered
Student (Praetorius, 12345678) deregistered
```

Methods: short syntax

```
 1 #include <iostream>
 2 #include <string>
 3 using std::cout;
 4 using std::string;
 5
 6 class Student {
 7 private:
 8   string lastname;
 9   int student_id;
10 public:
11   Student() : lastname("nobody"), student_id(0) {
12     cout << "Student generated\n";
13   }
14   Student(string name, int id) :
15     lastname(name), student_id(id) {
16       cout << "Student (" << lastname << ", ";
17       cout << student_id << ") registered\n";
18   }
19   ~Student() {
20     cout << "Student (" << lastname << ", ";
21     cout << student_id << ") deregistered\n";
22   }
23 };
24
25 int main() {
26   Student test;
27   return 0;
28 }
```

- Lines 11 and 14–15: Short syntax for assignment
  - Call of the corresponding constructors
  - Code less readable
- Output

  ```
  Student generated
  Student (nobody, 0) deregistered
  ```

One more example

```
 1 #include <iostream>
 2 #include <string>
 3 using std::cout;
 4 using std::string;
 5
 6 class Test {
 7 private:
 8   string name;
 9 public:
10   void print() {
11     cout << "Name " << name << "\n";
12   }
13   Test() : name("Standard") { print(); }
14   Test(string n) : name(n)  { print(); }
15   ~Test() {
16     cout << "Delete " << name << "\n";
17   }
18 };
19
20 int main() {
21   Test t1("Object1");
22   {
23     Test t2;
24     Test t3("Object3");
25   }
26   cout << "Block end" << "\n";
27   return 0;
28 }
```

- Output:

```
Name Object1
Name Standard
Name Object3
Delete Object3
Delete Standard
Block end
Delete Object1
```

Nesting of classes

```
 1 #include <iostream>
 2 using std::cout;
 3 using std::endl;
 4
 5 class Class1 {
 6 public:
 7   Class1() { cout << "Constr Class1" << endl; }
 8   ~Class1() { cout << "Destr Class1" << endl; }
 9 };
10
11 class Class2 {
12 private:
13    Class1 obj1;
14 public:
15    Class2() { cout << "Constr Class2" << endl; }
16    ~Class2() { cout << "Destr Class2" << endl; }
17 };
18
19 int main() {
20   Class2 obj2;
21   return 0;
22 }
```

- Classes can be nested
  - Standard constructor/destructor are automatically called
  - Constructors of the member first
  - Destructors of the member at the end
- Output:

```
Constr Class1
Constr Class2
Destr Class2
Destr Class1
```

vector_first.hpp

```
 1 #ifndef _VECTOR_FIRST_
 2 #define _VECTOR_FIRST_
 3
 4 #include <cmath>
 5 #include <cstdlib>
 6 #include <cassert>
 7 #include <iostream>
 8
 9 // The class Vector stores vectors in Rd
10
11 class Vector {
12 private:
13   // dimension of the vector
14   int dim;
15   // dynamic coefficient vector
16   double* coeff;
17
18 public:
19   // constructors and destructor
20   Vector();
21   Vector(int dim, double init = 0);
22   ~Vector();
23
24   // return vector dimension
25   int size();
26
27   // read and write vector coefficients
28   void set(int k, double value);
29   double get(int k);
30
31   // compute Euclidean norm
32   double norm();
33 };
34
35 #endif
```

vector_first.cpp 1/2

```
 1 #include "vector_first.hpp"
 2
 3 Vector::Vector() {
 4   dim = 0;
 5   coeff = (double*) 0;
 6   std::cout << "allocate empty vector" << "\n";
 7 }
 8
 9 Vector::Vector(int dim, double init) {
10   assert(dim>0);
11   this->dim = dim;
12   coeff = (double*) malloc(dim*sizeof(double));
13   assert(coeff != (double*) 0);
14   for (int j=0; j<dim; ++j) {
15     coeff[j] = init;
16   }
17   std::cout << "allocate vector, length " << dim << "\n";
18 }
```

- Implementation of three constructors (lines 3+9)
  - Standard constructor (line 3)
  - Declaration `Vector var(dim,init);`
  - Declaration `Vector var(dim);` with `init = 0`
  - Optional input via default parameter (line 9)
    - Defined in `vector.hpp` (see previous slide)
- **Attention:** `g++` requires explicit type casting for pointers, e.g., `malloc` (line 12)
- In C++ variables can be declared everywhere
- In C (original standard) only at the beginning of a block, which leads to more readable code
- Declaration right before use for local counter variables is acceptable (line 14)

vector_first.cpp 2/2

```
 9 Vector::Vector(int dim, double init) {
10   assert(dim>0);
11   this->dim = dim;
12   coeff = (double*) malloc(dim*sizeof(double));
13   assert(coeff != (double*) 0);
14   for (int j=0; j<dim; ++j) {
15     coeff[j] = init;
16   }
17   std::cout << "allocate vector, length " << dim << "\n";
18 }
19
20 Vector::~Vector() {
21   if (dim > 0) {
22     free(coeff);
23   }
24   std::cout << "free vector, length " << dim << "\n";
25 }
26
27 int Vector::size() {
28   return dim;
29 }
30
31 void Vector::set(int k, double value) {
32   assert(k>=0 && k<dim);
33   coeff[k] = value;
34 }
35
36 double Vector::get(int k) {
37   assert(k>=0 && k<dim);
38   return coeff[k];
39 }
40
41 double Vector::norm() {
42   double norm = 0;
43   for (int j=0; j<dim; ++j) {
44     norm = norm + coeff[j]*coeff[j];
45   }
46   return sqrt(norm);
47 }
```

main.cpp

```
 1 #include "vector_first.hpp"
 2 #include <iostream>
 3
 4 using std::cout;
 5
 6 int main() {
 7   Vector vector1;
 8   Vector vector2(20);
 9   Vector vector3(100,4);
10   cout << "Norm = " << vector1.norm() << "\n";
11   cout << "Norm = " << vector2.norm() << "\n";
12   cout << "Norm = " << vector3.norm() << "\n";
13
14   return 0;
15 }
```

- Compile with

```
g++ -c vector_first.cpp
g++ main.cpp vector_first.o
```

- Output:

```
allocate empty vector
allocate vector, length 20
allocate vector, length 100
Norm = 0
Norm = 0
Norm = 40
free vector, length 100
free vector, length 20
free vector, length 0
```