C++

- What is C++?
- How to write a program in C++?
- Hello World! with C++

- main
- cout, cin, endl
- using std::
- Scope operator ::
- Operators «, »
- #include <iostream>

What is C++

- Extension of C
  - Developed since 1979 at AT&T
  - Inventor: Bjarne Stroustrup
- C++ is compatible with C
  - No syntax errors (but possibly several warnings)
  - Stronger access control for 'structures'
    - Encapsulation (information hiding)
- Compiler:
  - Freely available in Unix/MacOS: g++
  - Microsoft Visual C++ Compiler
  - Borland C++ Compiler

Object-oriented programming language

- C++ is object-oriented C
  - Originally referred to as *C with classes*
- Object = Collection of data and functions
  - Functionality depends on the data
    e.g., multiplication for scalars, vectors, matrices
- Some online references
  - https://en.cppreference.com/w/
  - http://www.cplusplus.com

How to create a program in C++?

- Start your favorite text editor
  e.g., nano, emacs, vim, gedit, atom, ...
- Open a (new) file name.cpp
  - The filename extension .cpp is typical for programs in C++
- Write the *source code* (= program)
- Don't forget to save the file
- Compile the code, e.g., open a shell and type
  g++ name.cpp
- If there are no errors, one gets the *executable*
  a.out (a.exe under Windows)
- This can be executed with a.out or ./a.out
- Compile with g++ name.cpp -o output creates the executable output instead of a.out

Hello World!

```
1 #include <iostream>
2
3 int main() {
4   std::cout << "Hello World!\n";
5   return 0;
6 }
```

- C++ library for input/output is iostream
- main has compulsorily a return value of type int
  - int main()
  - int main(int argc, char* argv[])
    - In particular, note return 0; in line 5
- Scope operator :: characterizes the *name space*
  - All functions of the standard library have std
- std::cout is the standard function to print text to the screen
  - Operator << passes his right argument to cout

```
1 #include <iostream>
2 using std::cout;
3
4 int main() {
5   cout << "Hello World!\n";
6   return 0;
7 }
```

- using std::cout; in line 2
  - cout belongs to *name space* std
  - One can abbreviate std::cout with cout

Shell input for main

```
#include <iostream>
using std::cout;
using std::endl;

int main(int argc, char* argv[]) {
  int j = 0;
  cout << "This is " << argv[0] << endl;
  cout << "got " << argc-1 << " inputs:" << endl;
  for (j=1; j<argc; ++j) {
    cout << j << ": " << argv[j] << endl;
  }
  return 0;
}
```

- `<<` works with different types and can produce multiple output
- `endl` is equivalent to `\n` in C
- Shell can pass input as C strings to a program
  - The parameters are separated by blank spaces
  - argc = Number of parameters
  - argv = Vector with input strings
  - argv[0] = Program name
  - i.e., argc−1 effective input parameters
- Output for shell input `./a.out Hello World!`

```
This is ./a.out
got 2 inputs:
1: Hello
2: World!
```


Read input / Print output

```
#include <iostream>
using std::cin;
using std::cout;
using std::endl;

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

  cout << "Please enter an integer: ";
  cin >> x;
  cout << "Please enter two double: ";
  cin >> y >> z;

  cout << x << " * " << y << " / " << z;
  cout << " = " << x*y/z << endl;

  return 0;
}
```

- `std::cin` is the standard function to read input from the keyboard
  - Operator `>>` writes input to the variable given in its right argument
- Possible input/output of the program:

```
Please enter an integer: 2
Please enter two double: 3.6 1.3
2 * 3.6 / 1.3 = 5.53846
```

- `cin` / `cout` are equivalent to `printf` / `scanf` in C
  - But easier to use
  - Use of neither placeholder nor pointer is required

Data type `bool`

- `bool`
- `true`
- `false`

Data type `bool`

```
#include <iostream>
using std::cout;

int main() {
  double var = 0.3;
  bool tmp = var;

  if (1) {
    cout << "1 is true\n";
  }
  if (var) {
    cout << var << " is also true\n";
  }
  if (tmp == true) {
    cout << tmp << " is also true\n";
    cout << "sizeof(bool) = " << sizeof(bool) << "\n";
  }
  if (0) {
    cout << "0 is true\n";
  }
  return 0;
}
```

- C → No specific data type for logical values
  - Evaluation of logical expressions returns 1 for true, 0 for false
  - All nonzero numbers are interpreted as true
- C++ → Data type `bool` for logical values
  - Value `true` for true, `false` for false
  - All nonzero numbers are interpreted as true
- Output:

```
1 is true
0.3 is also true
1 is also true
sizeof(bool) = 1
```

Classes

- Classes
- Instances
- Objects

- `class`
- `struct`
- `private`, `public`
- `string`
- `#include <cmath>`
- `#include <cstdio>`
- `#include <string>`

Classes & Objects

- Classes are (programmer-defined) data types
  - Extensions of `struct` in C
  - They consist of data and methods
  - Methods = Functions on the data of the class
- Declaration etc. as for structures
  - Access to members via point operator (if accessing the data is allowed)
    - Access control = Encapsulation
- Formal Syntax: `class ClassName{ ... };`
- Object = Instance of a class
  - Variables of the new data type
  - Methods are stored only 1x in the memory
- Later: Methods can be overloaded
  - i.e., the functionality of the method depends on the input type
- Later: Operators can be overloaded
  - e.g., $x + y$ for vectors
- Later: Classes can be derived from existing classes
  - So-called *inheritance*
  - e.g., $\mathbb{C} \supset \mathbb{R} \supset \mathbb{Q} \supset \mathbb{Z} \supset \mathbb{N}$, where $\mathbb{R}$ inherits methods from $\mathbb{C}$ etc.

Access control

- Classes (and objects) contribute to abstraction
  - Knowledge of implementation details not important
- Users should have less information as possible
  - So-called *black-box* programming
  - Only input/output should be known
- Access must be secured
- Keywords `private`, `public` and `protected`
- `private` (standard)
  - Access allowed only for methods of the same class
- `public`
  - Access from 'outside' allowed
- `protected`
  - Access from 'outside' partially allowed ($\rightsquigarrow$ Inheritance)

Example 1/2

```
1  class Triangle {
2  private:
3    double x[2];
4    double y[2];
5    double z[2];
6
7  public:
8    void setX(double, double);
9    void setY(double, double);
10   void setZ(double, double);
11   double area();
12 };
```

- Triangle in $\mathbb{R}^2$ with vertices `x`, `y`, `z`
- Users cannot directly read/write `x`,`y`,`z`
  - Possible only via `get`/`set` functions in `public` part
- Users can call the method `area`
- Users must not know how the data are managed internally
  - The data structure can be changed if needed without affecting users' approach
  - e.g., a triangle can be defined also via a vertex and two vectors
- Line 2: `private:` can be omitted
  - All members/methods are `private` by default
- Line 7: after `public:`, free access

Example 2/2

```
1  class Triangle {
2  private:
3    double x[2];
4    double y[2];
5    double z[2];
6
7  public:
8    void setX(double, double);
9    void setY(double, double);
10   void setZ(double, double);
11   double getArea();
12 };
13
14 int main() {
15   Triangle tri;
16
17   tri.x[0] = 1.0; // Syntax error!
18
19   return 0;
20 }
```

- Lines 8–11: Declaration of `public` methods
- Line 15: Declaration of object `tri` of type `Triangle`
- Line 17: Access to a `private` member
- The compilation process yields an error
  `triangle2.cpp:17: error: 'x' is a private member of 'Triangle'`
  `triangle2.cpp:3: note: declared private here`
- Hence: Use of get/set-functions

Method implementation 1/3

```
1  #include <cmath>
2
3  class Triangle {
4  private:
5    double x[2];
6    double y[2];
7    double z[2];
8  public:
9    void setX(double, double);
10   void setY(double, double);
11   void setZ(double, double);
12   double getArea();
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 }
```

- Implementation as any other function
  - Direct access to class members
- Signature: `type ClassName::fctName(input)`
  - `type` = Return value (`void`, `double` etc.)
  - `input` = Input parameters as in C
- Important: `ClassName::` before `fctName`
  - i.e., the method `fctName` belongs to `ClassName`
- Inside `ClassName::fctName`, direct access to all class members is allowed (lines 16–17)
  - Also to `private` members
- Line 1: Inclusion of the C library `math.h`

Method implementation 2/3

```
#include <cmath>

class Triangle {
private:
  double x[2];
  double y[2];
  double z[2];

public:
  void setX(double, double);
  void setY(double, double);
  void setZ(double, double);
  double getArea();
};

void Triangle::setX(double x0, double x1) {
  x[0] = x0; x[1] = x1;
}

void Triangle::setY(double y0, double y1) {
  y[0] = y0; y[1] = y1;
}

void Triangle::setZ(double z0, double z1) {
  z[0] = z0; z[1] = z1;
}

double Triangle::getArea() {
  return 0.5*fabs( (y[0]-x[0])*(z[1]-x[1])
                 - (z[0]-x[0])*(y[1]-x[1]) );
}
```


Method implementation 3/3

```
#include <cmath>

class Triangle {
private:
  double x[2];
  double y[2];
  double z[2];

public:
  void setX(double x0, double x1) {
    x[0] = x0;
    x[1] = x1;
  }
  void setY(double y0, double y1) {
    y[0] = y0;
    y[1] = y1;
  }
  void setZ(double z0, double z1) {
    z[0] = z0;
    z[1] = z1;
  }
  double getArea() {
    return 0.5*fabs( (y[0]-x[0])*(z[1]-x[1])
                   - (z[0]-x[0])*(y[1]-x[1]) );
  }
};
```

- Method can be implemented inside the class definition
- Usually less clear code ⇒ It should be avoided

Call of methods

```cpp
#include <iostream>
#include "triangle4.cpp" // Code of slide 202

using std::cout;
using std::endl;

// void Triangle::setX(double x0, double x1)
// void Triangle::setY(double y0, double y1)
// void Triangle::setZ(double z0, double z1)

// double Triangle::getArea() {
//     return 0.5*fabs( (y[0]-x[0])*(z[1]-x[1])
//                     - (z[0]-x[0])*(y[1]-x[1]) );
// }

int main() {
  Triangle tri;
  tri.setX(0.0,0.0);
  tri.setY(1.0,0.0);
  tri.setZ(0.0,1.0);
  cout << "Area = " << tri.getArea() << endl;
  return 0;
}
```

- Call like for member access for C structures
  - Realization via function pointer possible in C
- `getArea` acts on members of `tri`
  - i.e., `x[0]` in method code refers to `tri.x[0]`
- **Output:** `Area = 0.5`

Class `string`

```cpp
1  #include <iostream>
2  #include <string>
3  #include <cstdio>
4  using std::cout;
5  using std::string;
6
7  int main() {
8    string str1 = "Hello";
9    string str2 = "World";
10   string str3 = str1 + " " + str2;
11
12   cout << str3 << "! ";
13   str3.replace(6,4, "Peter");
14   cout << str3 << "! ";
15
16   printf("%s?\n",str3.c_str());
17
18   return 0;
19 }
```

- **Output:** `Hello World! Hello Peter! Hello Peter?`
- Line 3: Inclusion of C library `stdio.h`
- Important: `string` $\neq$ `char*`, more powerful!
- `string` includes a collection of useful methods
  - '+' to combine strings
  - `replace` to replace sub-strings
  - `length` to read string lengths
  - `c_str` returns pointer to `char*`
- `http://www.cplusplus.com/reference/string/string/`

Structures

```cpp
struct MyStruct {
  double x[2];
  double y[2];
  double z[2];
};

class MyClass {
  double x[2];
  double y[2];
  double z[2];
};

class MyStructClass {
public:
  double x[2];
  double y[2];
  double z[2];
};

int main() {
  MyStruct var1;
  MyClass var2;
  MyStructClass var3;

  var1.x[0] = 0;
  var2.x[0] = 0; // Syntax error
  var3.x[0] = 0;

  return 0;
}
```

- Structures = Classes with `public` members
  - i.e., `MyStruct` = `MyStructClass`
- Better directly using `class`

Functions

- Default parameters & Optional input
- Overloading

Default parameters 1/2

```
1  void f(int x, int y, int z = 0);
2  void g(int x, int y = 0, int z = 0);
3  void h(int x = 0, int y = 0, int z = 0);
```

- Set up of default values for input parameters
  - Via `= value`
  - The input parameter is then optional
  - If not passed, default value is assigned
- Example: Line 1 allows for the calls
  - `f(x,y,z)`
  - `f(x,y)` (`z` receives the default value `z = 0`)

```
1  void f(int x = 0, int y = 0, int z); // Wrong
2  void g(int x, int y = 0, int z);     // Wrong
3  void h(int x = 0, int y, int z = 0); // Wrong
```

- Optional (= with default value) parameters must follow required parameters
  - i.e., after an optional parameter, no required parameter can appear

Default parameters 2/2

```
1  #include <iostream>
2  using std::cout;
3
4  void f(int x, int y = 0);
5
6  void f(int x, int y = 0) {
7      cout << "x=" << x << ", y=" << y << "\n";
8  }
9
10 int main() {
11     f(1);
12     f(1,2);
13     return 0;
14 }
```

- Default parameter can be defined only once
- Compiling yields a syntax error:

  `default_wrong.cpp:6: error: redefinition of default argument`
- Correction: Define default parameter only in line 4!
- Output after the correction:

  `x=1, y=0`

  `x=1, y=2`
- **Convention:**
  - Default parameter are defined in header file `.hpp`
- No variable name required with forward declaration
  - `void f(int, int = 0);` in line 4 ist fine

Function overloading 1/2

```
1  void f(char*);
2  double f(char*, double);
3  int f(char*, char*, int = 1);
4  int f(char*);                 // Syntax error
5  double f(char*, int = 0);     // Syntax error
```

- Multiple functions can have the same name
  - But different signature
- The input must be unambiguous
- Function call identifies the right version
  - Compiler recognize it with the input parameters
  - Be careful with implicit type casting
- This concept is called *overloading*
- Ordering in declaration is not important
  - i.e., lines 1–3 can arbitrarily be permuted
- Return values can be also different
  - However: choosing different return values but same input parameter is not allowed
    - Lines 1–3: OK
    - Line 4: Syntax error, as input = line 1
    - Line 5: Syntax error, as optional input

Function overloading 2/2

```
#include <iostream>
using std::cout;
using std::endl;

class Car {
public:
  void drive();
  void drive(int km);
  void drive(int km, int h);
};

void Car::drive() {
  cout << "10 km traveled" << endl;
}

void Car::drive(int km) {
  cout << km << " km traveled" << endl;
}

void Car::drive(int km, int h) {
  cout << km << " km traveled in " << h
    << " hour(s)" << endl;
}

int main() {
  Car TestCar;
  TestCar.drive();
  TestCar.drive(35);
  TestCar.drive(50,1);
  return 0;
}
```

- Output:
```
10 km traveled
35 km traveled
50 km travel in 1 hour(s)
```

Overloading vs. default parameters

```
#include <iostream>
using std::cout;
using std::endl;

class Car {
public:
  void drive(int km = 10, int h = 0);
};

void Car::drive(int km, int h) {
  cout << km << " km traveled";
  if (h > 0) {
    cout << " in " << h << " hour(s)";
  }
  cout << endl;
}

int main() {
  Car TestCar;
  TestCar.drive();
  TestCar.drive(35);
  TestCar.drive(50,1);
  return 0;
}
```

- Output:
```
10 km traveled
35 km traveled
50 km traveled in 1 hour(s)
```


Simple error control

- Why access control?
- Avoid runtime error!
- Intentional error-caused termination

- `assert`
- `#include <cassert>`

Why access control?

```
1  class Fraction {
2  public:
3    int numerator;
4    int denominator;
5  };
6
7  int main() {
8    Fraction x;
9    x.numerator = -1000;
10   x.denominator = 0;
11
12   return 0;
13 }
```

- Most of the programming time is usually devoted to the research of runtime errors
- Catch errors with good programming practices!
  - Check function input, abort if not admissible
  - Ensure admissible output
  - Control access to data via mutator functions (`get`/`set` methods)
    - Data should be always `private`
    - Users should not be allowed to bungle data
    - In C = They should not...
    - In C++ = They cannot!
- How to ensure meaningful data values? (line 10...)
  - Prevent possible error sources
- Intentional termination with C library `assert.h`
  - Add `#include <cassert>`
  - Termination with line number information in case of errors

C library `assert.h`

```
1  #include <iostream>
2  #include <cassert>
3  using std::cout;
4
5  class Fraction {
6  private:
7    int numerator;
8    int denominator;
9  public:
10   int getNumerator() { return numerator; };
11   int getDenominator() { return denominator; };
12   void setNumerator(int n) { numerator = n; };
13   void setDenominator(int n) {
14     assert(n != 0);
15     if (n > 0) {
16       denominator = n;
17     }
18     else {
19       denominator = -n;
20       numerator = -numerator;
21     }
22   }
23   void print() {
24     cout << numerator << "/" << denominator << "\n";
25   }
26 };
27
28 int main() {
29   Fraction x;
30   x.setNumerator(1);
31   x.setDenominator(3);
32   x.print();
33   x.setDenominator(0);
34   return 0;
35 }
```

- `assert(condition);` termination if `condition` wrong
- Output:
```
1/3
Assertion failed:  (n>0), function setDenominator,
file assert.cpp, line 14.
```