C++ Building Blocks

1. Programming Paradigms

Definition 0.1 Procedural Programming: Is a programming paradigm based on procedures, also known as routines, or functions. Any given procedure might be called at any point during a programs execution, including by other procedures as well.

Definition 0.2 Modular Programming: Modular programming involves dividing our objects into modules of common characteristics and then piplining the modules to obtain the defined output. Modules consit of an interface and an implementation. The interface describes the functionality of the module to the outer world and explains how other modules can make use of it.
E.g. separating files, Libraries.

Definition 0.3 Object Oriented Programming (OOP): Builds on modular programming but focuses rather on data itself than algorithms.
Programs are organized as objects: data structures consisting of data fields and member functions.

Definition 0.4 Generic Programming: Style of programming in which algorithms are written in terms of types "to be specified later" that are then instantiated when needed for specific types provided as parameters.
E.g. Templates

2. Building a C++ Program

Definition 0.5 Compiler: a computer program (or set of programs) that translates source code of a high-level programming language, e.g. C++ into a low level language (e.g. assembly language [def. 0.6] or direct into machine language).

Definition 0.6 Assembly Language (asm): Is a low-level programming language that can be translated one-to-one from machine code to a readable text.
In contrast to high level programming languages each assembly language is (usually) computer architecture specific.

Definition 0.7 Assembler: Assembly language gets converted into machine code by a program called assembler.

Definition 0.8 Object File `file.o`: The compiler creates an object file for each source file. The object file may contain executable code as well as information for the linker [def. 0.9].

Definition 0.9 Linker: The Linker is a computer program that takes one or many object files and combines them into one single executable file, library, or another object file [def. 0.8].

Note: System Libraries

Are usually linked by default.

2.1. Decleration Pradigm

C++ Basics

Libraries

1. Static Libaries/Archieves *.lib/*.so/.ar

Static Libraries/Achieves are nothing else but a couple of object files *.o packed together, that can be linked into your program.

1.1. Creating Static Libaries

1. Creating an achieve from object files *.o

```
ar [options] libname.a file1.o file2.o ...
```

-c: Create an archive/library and do not warn if the library has to be created.

-r: Add object files *.o to the library and replace any existing files with the same name.

-u: Replace the member only if the modification time of the file member is more recent then the time of the file in the archive.

2. Add index/table of contents (not needed on all platforms)

```
ranlib libname.a
```

Note

A good combination to use is ruc.

Note: ranlib

ranlib generates an index to the contents of an archive and stores it in the archive. The index lists each symbol defined by a member of an archive that is a relocatable object file. An archive with such an index speeds up linking to the library and allows routines in the library to call each other without regard to their placement in the archive.

1.2. Compiling&linking achieves into our Program

linking the library/archive into our program

```
gcc|g++|... -static main.cpp [-Ldir]  -llibname.a
```

-L: Specify the path from which to include the library

2. Dynamic Libaries *.ddl/*.a

3. Header Only Libaries *.h

Object Oriented Programming (OOP)

Definition 1.1 Classes:

```
class MyClass [:[visibility_mode] BaseClass]{
  public:
  MyClass()                  ⟵ Constructor
  ~MyClass()                 ⟵ Destructor
  public:
      ...
  protected:
      ...
  private:
      ...
  friend:
      ...
}
```

- Use capital letters for class names
- Use underscore suffixes for member variables `var_`

Standard Template Library (STD)

The C++ library includes the same definitions as the C language library organized in the same structure of header files, with the following differences:

1. Each header file in C++ has the same name as the C language version but
   - with a "c" prefix
   - and no ".h" extension

   For example, the C++ equivalent for the C language header file `<stdlib.h>` is `<cstdlib>`.
2. Every element of the C++ libraries is defined within the std namespace and not globally.

C-include: `#include<library.h>`
C++-include: `#include<clibrary>`

Timer

4.1. #include⟨chrono⟩

In order to obtain the time passed between two time points, in seconds use:

```
const auto t1 = std::chrono::steady_clock::now();
// ...
const auto t2 = std::chrono::steady_clock::now();
std::chrono::duration<double> diff = t2-t1;
```

or in order to get a different unit use

```
const double diff =
↪ std::chrono::duration_cast<unit>(t2-t1).count();
```

Where unit corresponds to:

- `std::chrono::nanoseconds`
- `std::chrono::microseconds`
- `std::chrono::milliseconds`
- `std::chrono::seconds`
- `std::chrono::minutes`
- `std::chrono::hours`

Input and Output

C++ includes two input/output libraries:

- The standard set of C Input/Output functions
- A modern, stream-based object-oriented I/O library

Note

The C/C++ language did not build the input/output facilities into the language. That is, there exist no specific keywords like read or write.
Instead, it left the I/O to the compiler as external library functions.

6. Stream Based I/O #include⟨ios⟩

Definition 1.2 The Stream Object `stream`:
Is the basic data type in order to represent I/O-operations in C++. A stream is basically a sequence of bytes flowing in and out of a program Thus streams act as intermediaries between the program and an actual IO devices s.a. files, consoles, disks, networks, other programs,... The basics steps to perform input and output in C++ consist of:

1. Constructing a stream object
2. Connect the stream object to an I/O device
3. Perform I/O operations on the stream via the streams public methods/operators
4. Disconnect the stream object from the I/O device and free stream object.

Definition 1.3 `std::ios_base`: Is the abstract base class of all stream classes.

6.1. #include⟨iostream⟩

Defines standard input/output stream objects

- `cin`: standard input stream object
- `cout`: standard output stream object
- `cerr`: standard output stream object for errors
- `clog`: standard output stream object for logging

make this a subsubsubsection

Setting the output precision

```
std::cout.precision(17);
```

6.2. #include⟨fstream⟩

Defines file types for manipulating files

- `ofstream`: used to create or write to files
- `ifstream`: used to read from files
- `fstream`: able to do both of the above

Checking for existence

```
std::ifstream fh(filename);
fh.good();
```

opening files

```
stream_type fh;
fh.open(file_path, mode)
// do something
fh.cloes();
```

modes

- `ios::app`: append to the file
- `ios::ate`: open the file and move the read/write pointer to the end of the file
- `ios::in`: open a file for reading
- `ios::out`: open a file for writing
- `ios::trunc`: if the file already exists, delete its contents

Note

All this modes may be combined using the | symbol

6.3. Reading and Writing

As fstream objects are streams we may use the same stream operators we use for `cin/cout`.

Writing to a file

```
fh << value;
```

6.4. #include⟨iomanip⟩

Defines functions in order to manipulate input/output

- `setfill`: set the streams fill character
- `setw`: set the streams fill width
- `setprec`: set the streams output precision for floating-point values

change this better

Setting the output precision

```
std::cout.precision(17);
```

```
ostream << std::setprecision(17) << ...
```

7. C-Style I/O #include⟨stdio.h⟩

7.1. fprintf

Definition 1.4: Writes C-strings to the specified output stream:

```
int fprintf(FILE *stream, const char *format, spec_arg1,...)
```

- `stream`: Pointer to a `FILE` object that identifies an output stream
- `format`: is a C-string that contains the text to be written to the stream.
  It can optionally contain embedded `format specifiers` that are replaced by the values specified in subsequent additional arguments and formatted as requested.
- `spec_arg`: depending on the format string, the function expects a sequence of additional arguments equaling the number of values specified in the format specifiers.
  Each argument specifies the value to be used to replace a `format specifier` in the format string (or a pointer to a storage location, for n).

7.1.1. Format Specifiers

Definition 1.5: Specifies format, type and other attributes of a supplied specifier argument and has the following form:

```
%[flags][width][.precision][length]specifier
```

specifiers

specifierspecifies type and interpretation of its argument:

Specifier	Directive	Example
`i(d)`	signed decimal integer	392
`u`	unsigned decimal integer	7326
`f(F)`	decimal float (uppercase)*	392.65
`e(E)`	Scientific notat. (uppercase)*	3.9265e+2
`g(G)`	Use shortest f/e (F/E)	392.65
`c`	Character	c
`s`	String of chars	sample
`p`	pointer address	b8000000
`n`	nothing specified by int	
`%`	writes % sign	%

Note*

Uppercase corresponds to things like nan/NAN and e/E.

add rest: http://www.cplusplus.com/reference/cstdio/fprintf/

7.2. printf

Definition 1.6 printf: Writes C-strings to the standard output stream:

```
int fprintf(const char *format, spec_arg1,...)
```

Note

Its basically a shorthand for:

```
int fprintf(stdout, const char *format, spec_arg1,...)
```

Type Conversions

8.1. #include⟨cstdlib⟩

Declares a set of general purpose functions for:

- Converting strings
  - `std::atoI()`
  - `std::atof()`
  - `std::atol()`
  - `std::atoI()`
- Mathematical Tools:
  - `std::abs()`: returns abs of integer and long integer values

Math

cmath

- `std::cbrt`: Calcualte the cubic root $\sqrt[3]{\cdot}$

9.1. #include⟨cmath⟩

Declares a set of functions to compute common mathematical operations and transformations

Others

- `abs()`: returns absolute value of given type.

Algorithms & Data Structures

2. Generic Container

2.1. C-Style Array
2.2. #include⟨array⟩
2.3. #include⟨vector⟩
2.4. #include⟨map⟩

Definition 2.1 Map:

define new code box for this

3. Generic Algorithms #include⟨algorithm⟩

Implement a big number of useful algorithms.
Rely only on the existence of iterators ⇒ do not depend on a specific container type.

Note: C-style Arrays

Can use generic algorithms by using pointers i.e.

```
algorithm(arr, arr+N);
```

3.1. for_each

Applies a function `my_function` onto each element of a given container and returns the return value of the function call for the last element.

```
for_each(it first, it last, my_function);
```

Create sequences of $2^1, 2^2, \dots, 2^5$ Listening 2.1

```
vector<int> exp{1, 2, 3, 4, 5};
int base = 2;
std::for_each(exp.begin(), exp.end(), [base](int &val) { val
↪ = pow(base, exp); });
```

add somewhere in algorithms section

Title Listening 2.1

```
void get_metrics(vector<unsigned long long> cycles, Metrics
↪ &metrics) {
  double sum = std::accumulate(std::begin(cycles),
  ↪ std::end(cycles), 0.0);
  double mean = sum / cycles.size();

  double accum = 0.0;
  std::for_each(std::begin(cycles), std::end(cycles),
  ↪ [&](const double val) {
    accum += (val - mean) * (val - mean);
                                                        });

  double  std = std::sqrt(accum / (cycles.size() - 1));
  metrics.mean = mean;
  metrics.std = std;
}
```

C++11

4. Special Pointers

4.1. Nullpointer

In c++ NULL is just a define for zero: *#define NULL 0*.
Assume: now want to pass a NULL pointer to a function for example for debuging or if don't know yet which concrete pointer to use.
Problem: if we have an overloded function, the function now no longer knows what NULL is.

```
void f(int v) std::cout << "non-pointer overload";
void f(int* v) std::cout << "pointer overload";

int main(){ f(NULL); }
```

f(NULL) will print "non-pointer overload" eventought we would have expected/wanted the "pointer overload"-version.
Solution: f(nullptr) will print "pointer overload" as expected.

Add different ways of passing functions to functions:
https://vittorioromeo.info/index/blog/passing_functions_to_functions.html

5. Function Arithmetics

5.1. Lambd Functions

Are closeures or anonymous functions that are used where you used to (before C++11) create Functors (simple small function objects) that are not meant to be reused.

```
[auto functionHandle =
↪  ][captures](parameters)[->retrurnType]{body}
```

captures:

- []: No external reference.
- [=]: Capture all values from the enclosing scope by value.
- [&]: ——"——by reference.
- [this]: Capture all data members of the enclosing calss.
- [var1, &var2, ...]: Specification for single values.

5.2. std::function

Is a templated object that is used to store and call any callable type, such as functions, objects, lambdas and the result of std::bind

```
#include <functional>
std::function< returnType(argType) > f;
```

5.3. std::bind

std::bind is a template function that that binds a set of arguments to a function and returns a std::function object.

```
#include <functional>
[(std::function <returnType(argTyppe)>|auto) f = ]
↪  std::bind(callableObj, arg1, ...)
```

Listening 2.1 : *Example* *cpp*

```
using namespace std;
void execute(
  const vector<function<void() >>& fs){
  for (auto& f : fs) f();
}

vector<function<void ()>> vec;
function <void ()> f
   = bind(callable, args);
vec.push_back(f);
```

Assume we define a function that takes an vector of void functions as parameter. If we now want to pass a non-void callable to the vector, we can use bind to define a new void function.

Placeholders

std::bind allows us also to use placeholders, which are simply numbers (indicating the arg position in the new function) followed by an underscore.

```
std::bind(callable, 2_, 1_) // arg1 ↔ arg2 reversed
```

5.4. std::future

6. Rvalue Reference

Rvalues vs. Lvazlues Listening 2.2

```
int a = 5;  // a is a lvalue
int& b = a; // b is a (lvalue) reference
int&& c;    // c is a rvalue reference
```

Overloading for r-and l-value references

```
Given: overloaded function printInt:
void printInt(int& i) std::cout << "lv-ref" << i;
void printInt(int&& i) std::cout << "rv-ref" << i;

printInt(a); //Calls printInt(int& i)
printInt(6); //Calls printInt(int&& i)
```

Attention

If we would also define a function void printInt(int a) we would get an compiler error.
This is because for:

1. printInt(a) the compiler does not know if to call the reference or the non-reference version.
2. printInt(6) the compiler does not know if to call the non-reference or the rvalue-reference version.

⇒ can only define two of the three versions.

There are two main applications of rvalue-references:

1. Move Semantics: to optimize data transferring between the objects, in situations when objects have large contents allocated on the heap.
2. Perfect Forwarding: to optimize data forwarding to other functions using universal references.

6.1. std::move

Given Listening 2.3

```
class MyVector{
  double* arr_; //big array
  size_t size;
public:
MyVector(const MyVector& rhs){ // expens. Copy Constr.
  size = rhs.size;
  arr_ = new double[size];
  for(int i=0; i<size; ++i) arr_[i] = rhs.arr_[i];
  }
MyVector(MyVector&& rhs){ // cheap Move Constr.
  size = rhs.size;
  arr_ = rhs.arr_; /* doesn't do any copying
  $\Rightarrow$ need to set the rhs.size to nullptr so that
↪  arr_ doesn't get destroyed when destructor of rhs is
↪  called.*/
  rhs.size = nullptr;
  }
~MyVector() {delete [] arr_; }
}
```

Passing MyVector to functions Listening 2.4

```
main(){
  MyVector reusable = createMyVector();

  foo(reusable); //Deep copy using copy constructor
  foo(createMyVector()) //Cheap copy using move const.
}
```

How can we make use of this?

If we want to write a generic efficient program: we need to distinguish between copying (lvalues) and moving objects (rvalues):

```
foo(MyVector v);
foo_by_refernce(MyVector& v);
```

Problem we do not want to create lots of different versions of foo.
Solution: c++11 offers the std::move function that will move an lvalue with the move constructor.

```
main(){
  MyVector reusable = createMyVector();

  foo_by_reference(reusable);//Cheap copy with move const.
  foo(reusable);            //Most expensive
  foo(std::move(reusable)); // reusable is destroyed here
  // ⇒ reusable.arr_ = nullptr;
}
```

Notes

1. With move we do no longer need to define foo_by_reference.
2. Attention: after std::move we can no longer use reusable.

Conclusion

1. Move semantic avoids costly and unnecassary deep copying and is used by all stl containters.
2. Move Constructors/Assignment Operators are particularly powerfull where passing by value **and** passing by reference are used.

6.2. std::forward

Given a functions that forwards its arguments to another function, perfect forwarding ensures that the argument passed to the second function as if the first function doesn't exist.

```
template<class T>
void wrapper(T&& arg)
{
    // arg is always lvalue
    foo(std::forward<T>(arg)); // Forward as lvalue or as
    ↪  rvalue, depending on T
}
```

Why is this useful: it avoids excessive copying, and avoids the template author having to write multiple overloads for lvalue and rvalue references.

Implementation of std::forward

```
  template<class T>
    T&& forward(typename remove_reference<T>::type& arg)
      return static_cast<T&&>(arg);
```

Law 2.1 Reference Collapsing: C++11 defines reference collapsing rules for type deduction in order for the compiler to chose the right type:

Compiler Code	$\xrightarrow{\text{collapsing}}$	Type
T& &	⟶	T&
T& &&	⟶	T&
T&& &	⟶	T&
T&& &&	⟶	T&&

Definition 2.2 Universal References T&& ref: T&& ref is a universal reference and not only a rvalue-refrence $\iff$:

1. T is a templated type.
2. Type dedeuction (reference collapsing happens to T).

This give functions the power to take on any value (not only type) lvalue, rvalue, const, non-const,...

Listening 2.2 : *Example* *cpp*

```
void fu(X&& t);
void fu(X& t);

template<typename Arg>
void relay(Arg&& arg)
// Universal Reference
{
    fu(std::forward<Arg>(arg));
}

int main(){
  MyVector v;
  relay(v); // lvalue ref 1
  relay(MyVector()); // rvalue ref 2
}
```

1: T is deduced to be an lvalue reference, std::forward<T> just returns its argument and does nothing.
2: std::forward makes sure that the argument is forwarded as an rvalue reference and thus that fu(X&& t) is called.

Where do we need this?

Essential for libraries such:

1. std::thread
2. std::function which pass arguments to another (user-supplied function).