Programmeertalen: C++

Bas Terwijn

Universiteit van Amsterdam

March 27 2020

Programmeertalen: waar zijn we nu?

	Taal	**Hoorcollege**			
Week 1	Bash	di 4/2	11:00-13:00	CWI Turingzaal	Bas
Week 2	Haskell	di 11/2	11:00-13:00	CWI Turingzaal	Ana
Week 3	Prolog	di 18/2	11:00-13:00	CWI Turingzaal	Koen
Week 4	Python	di 25/2	11:00-13:00	CWI Turingzaal	Bas
Week 5	Erlang	di 3/3	11:00-13:00	CWI Turingzaal	Ana
Week 6	Go	di 10/3	11:00-13:00	CWI Turingzaal	Koen
Week 7	**C++**	**di 17/3**	**11:00-13:00**	**CWI Turingzaal**	**Bas**
Week 8	Tentamen	di 24/3	9:00-11:00	USC Sporthal 2	**Postponed!**

COVID-19: some deadlines are extended, see Canvas (announcements)

Programming Paradigms

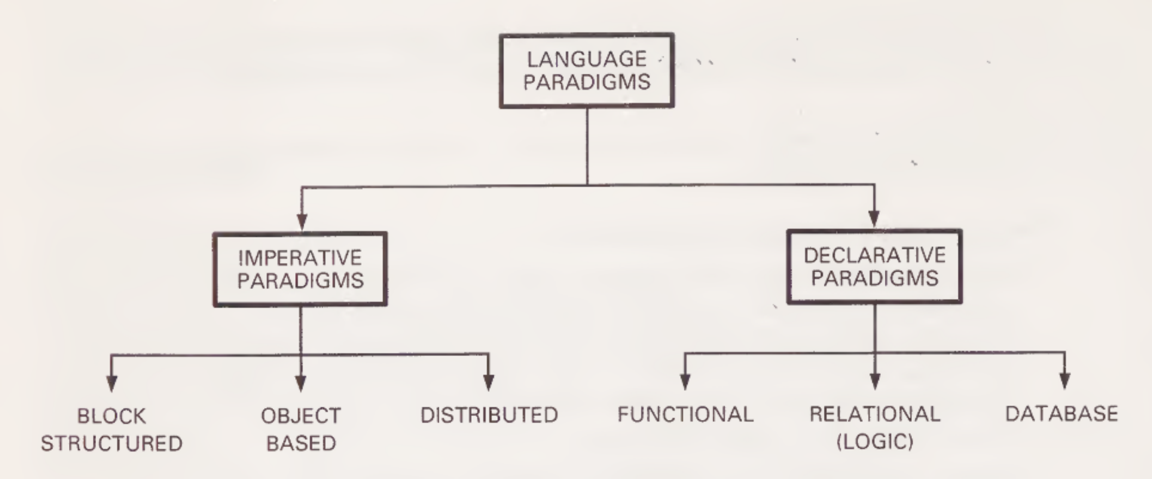


Figure: Programming Languages: Paradigm and Practice, Doris Appleby

Popular Programming Languages

- TIOBE Programming Community Index
  - count number of search query hits
  - *https://www.tiobe.com/tiobe-index*
- PYPL PopularitY of Programming Language
  - number of search queries (Google Trends)
  - *https://pypl.github.io/PYPL.html*
- RedMonk Programming Language Rankings
  - GitHub usage, Stack Overflow discussions
  - *https://redmonk.com/sogrady/2020/02/28/language-rankings-1-20*

Popular Programming Languages

Imperative	**Declarative**
How?	What? (less code)
state-change/side-effects	no state-change/side-effects (easier parallel programming)
iterative	recursion (complex? tail recursion for performance)

C/C++ Usage

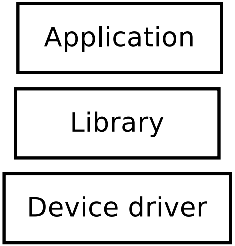
- Operating systems:
  - Unix, Windows, Linux, Mac OS, Android, ...
- Compilers, virtual machines, tools, libraries:
  - Bash, Haskell, SWI-Prolog, Python, Erlang, Go, ...
- Embedded systems:
  - routers, cameras, washing/coffee machine, cars, ...
- High-performance Computing/Gaming

high speed, low memory usage, long battery life, low energy cost, environmentally friendly

C/C++ Close to the Metal

- Can give control over low level things (bits and bytes in memory)
- Compiler with strong optimizer to machine code
- In general no other high-level language should be faster (sacrifice readability)
- A programmer doesn't pay for what she doesn't use
- Zero cost abstractions (processor doesn't know about abstractions, e.g. classes)

The C/C++ language has a reputation of being complex (more options to choose from), but application level programming is **not much** more complex than in other languages (Java).

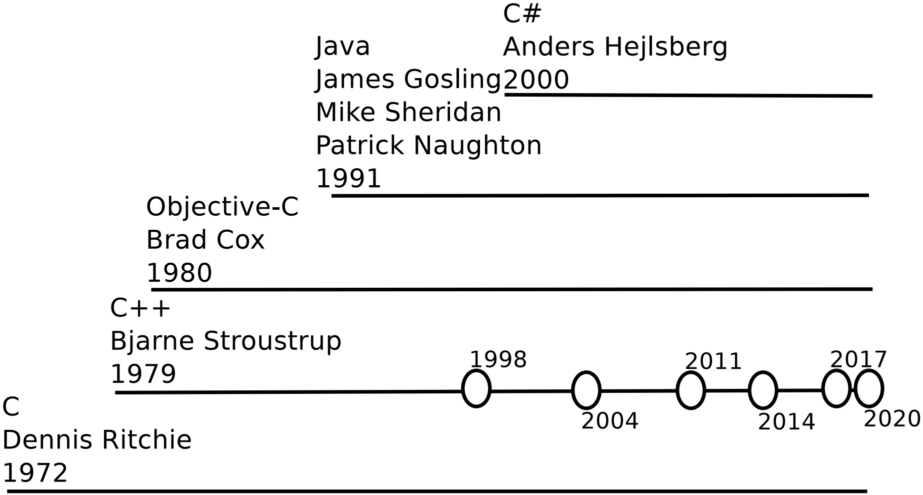


Application

Library

Device driver

Time Line



C#
Anders Hejlsberg
2000

Java
James Gosling
Mike Sheridan
Patrick Naughton
1991

Objective-C
Brad Cox
1980

C++
Bjarne Stroustrup
1979

C
Dennis Ritchie
1972

Hello World

src/helloworld.cpp

```cpp
#include <iostream>

int main(int argc,char* argv[])
{
    std::cout<< "Hello" << " World!" << '\n';
    return 0;
}
```

```
$ g++ -std=c++17 -Wall -Wextra -pedantic -O3 helloworld.cpp -o helloworld
$ ./helloworld
Hello World!
```

-std=c++17	use the 2017 revision of the language
-Wall -Wextra -pedantic	turn on a decent amount of warnings
-O3	optimize the code (use when releasing it)
-o helloworld	create output file 'helloworld' (default: 'a.out')

Type System

- Statically typed (compiler errors help spot bugs)
- Weakly typed (some implicit type conversions: `bool b=5;`)
- Fundamental types:
  - `bool`, `char`, `int`, `float`, `double`
  - initialized to indeterminate value, so: (`int sum=0;`)
  - type modifiers: `const`, `*`, `&`, `short`, `long`, `signed`, `unsigned`
- Type deduction:
  - `int i=5;`
  - `auto i=5;`

Conditions

src/conditionals.cpp

```
if (a<b)  // if statement
{ c=10; }
else
{ c=20; }

a<b ? c=10 : c=20; // conditional/ternary operator

switch (a) // switch statement
{
 case 0: c=10; break;
 case 1: c=20; break;
 default:
     c=100;
}
```

Flow Control

src/control-flow.cpp

```cpp
for (int i=0; i<10; ++i) // for-loop (break; continue;)
{ ... }

while (a<b) // while-loop
{ ... }

do // do-while loop
{
  ...
} while (a<b)

try // exceptions
{
    throw my_exception("problem detected");
}
catch ( runtime_error& e )
{  std::cout<< e.what() << '\n'; }
catch ( exception& e )
{  std::cout<< e.what() << '\n'; }
```

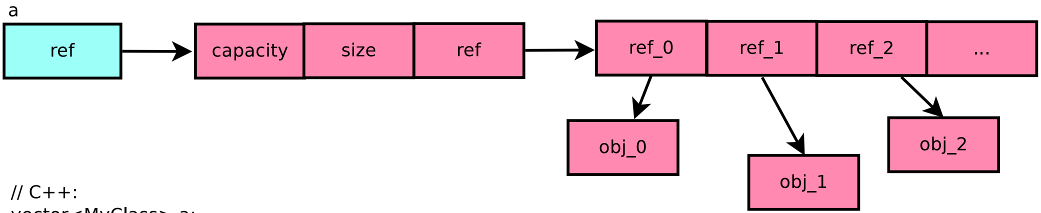
Value Semantics

src/value-semantics.cpp

```cpp
// Java
int v=5;                  // stack allocated
MyClass v=new MyClass();  // dynamic/heap allocated (at runtime)
                          // garbage collected when unreachable
                          // JVM tracing collector finds unreachables
// C++
int v=5;                  // stack allocated
MyClass v;                // stack allocated, preferred!

MyClass* v=new MyClass(); // dynamic/heap allocated (at runtime)
delete v;                 // requires deallocation (error prone)

unique_ptr<MyClass> v = std::make_unique<MyClass>(); // dynamic/heap allocated
                                                     // garbage collected when out of scope
                                                     // can't be copied

shared_ptr<MyClass> v = std::make_shared<MyClass>(); // dynamic/heap allocated
                                                     // garbage collected by reference counter
```

Value Semantics, Dynamic Array

```
// Java:
ArrayList<MyClass> a=new ArrayList<MyClass>();
```



a

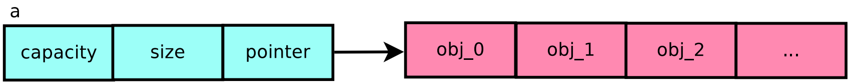
ref

capacity size ref

ref_0 ref_1 ref_2 ...

obj_0 obj_1 obj_2

```
// C++:
vector<MyClass> a;
```



a

capacity size pointer

obj_0 obj_1 obj_2 ...

Pointers vs References

src/reference.cpp

```cpp
#include <iostream>

void function(int  by_value,
              int* by_reference1, // uses "pointer"
              int& by_reference2) // uses "reference", prefered!
{
    by_value       = 100;
    *by_reference1 = 200;
    by_reference2  = 300;
}

int main()
{
    int a=1,b=2,c=3;
    function( a,
              &b,
              c   );
    std::cout<< a <<' '<< b <<' '<< c <<'\n'; // 1 200 300
}
```

Pointers vs References

- A reference is a simplified pointer (to avoid common bugs)

cannot be uninitialized	MyClass *p;
cannot be re-assign	p=&otherObject;
no pointer arithmetic	*(p+4);
no dynamic memory allocation	p=new MyClass();
no pointer to pointer to ...	MyClass** pp=&p;

Const Reference

src/const-reference.cpp

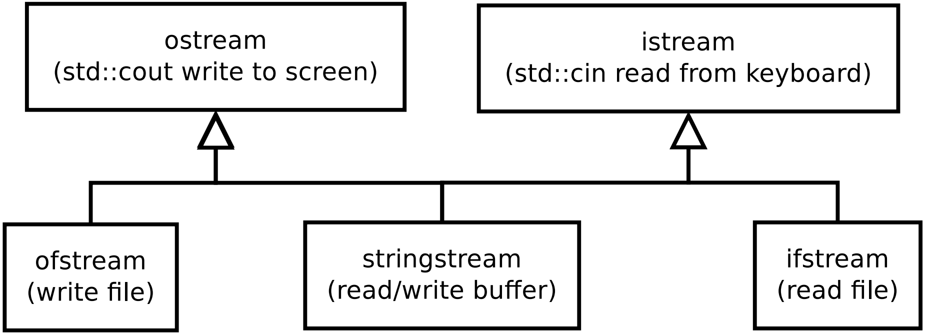
```cpp
// copies MyLargeClass, slow
void call_by_value(MyLargeClass m)
{ ... }

// passes reference to object, but object maybe be changed
void call_by_reference(MyLargeClass& m)
{ ... }

// passes reference to object, and object cannot change, preferred!
void call_by_const_reference(const MyLargeClass& m)
{ ... }

int main()
{
    MyLargeClass m;
    call_by_const_reference( m );
    ...
}
```

Higher Order Functions

src/func-higher-order.cpp

```cpp
#include <vector>
#include <algorithm>
#include <iostream>
using namespace std;

bool even(int x)
{   return x%2==0; }

int main()
{
    vector<int> data{1,2,3,4,5}; // dynamic array
    cout<< count_if(data.begin() ,data.end(), even) <<'\n'; // 2

    // lambda, anonymous function
    cout<< count_if(data.begin() ,data.end(),  [](int x) { return x%2==1;   }  ) <<'\n'; // 3

    int div=3;
    // lambda that captures local variables by reference (closure)
    cout<< count_if(data.begin() ,data.end(), [&](int x) { return x%div==0; }  ) <<'\n'; // 1
}
```

Streams


ostream
(std::cout write to screen)
istream
(std::cin read from keyboard)
ofstream
(write file)
stringstream
(read/write buffer)
ifstream
(read file)

Streams

src/streams.cpp

```cpp
#include <iostream>
#include <fstream>
#include <sstream>
void read_write_int(std::istream& is, std::ostream& os)
{   while (is.peek() != EOF )          // test end of stream
    {   int i; is >> i;                // read int
        if (!is.fail()) os<< i <<' ';  // test status, write int
        else { is.clear(); is.get(); } // clear error status, read 1 char
    }
}

int main()
{   std::stringstream in_ss{"ABC 10 20___30"}, out_ss;
    if (std::ofstream out_file("file.txt"); out_file.is_open())
    {   read_write_int( in_ss , out_file );    // read in_ss write to out_file
        out_file.close();
        if ( std::ifstream in_file{"file.txt"}; in_file.is_open())
        {   read_write_int( in_file, out_ss ); // read in_file write to out_ss
            std::cout<< out_ss.str() <<'\n';   // 10 20 30
        }
    }
}
```

Classes, Constructor Destructor

src/classes-destructor.cpp

```cpp
class MyClass
{
    int a;
    Resource resource;
public:
    MyClass() // constructor
    {   a=0; resource=claim_some_resource(); } //initializes members

    MyClass(int a) // constructor
      : a{a}, resource{claim_some_resource()} //initializes members, preferred!
    { }

    ~MyClass() // destructor
    {   free_resource( resource ); }
};

int main()
{
    MyClass m{42}; // claims resource
}   // out of scope: frees resource (guaranteed, RAII)
```

RAII: Resource Acquisition Is Initialization

Classes, Operator Overloading

src/classes-operators.cpp

```cpp
#include <iostream>
using namespace std;

class MyClass
{ public:
    int x;
    int operator()(int a,int b) { return x*a+b; } // has to be defined inside class
};

bool          operator==(const MyClass& m1, const MyClass& m2) { return m1.x==m2.x;         }
MyClass       operator+ (const MyClass& m1, const MyClass& m2) { return MyClass{m1.x+m2.x}; }
MyClass&      operator+=(      MyClass& m1, const MyClass& m2) { m1.x+=m2.x; return m1;     }
std::ostream& operator<<(std::ostream&  os, const MyClass& m ) { return os<<m.x;            }

int main()
{
    MyClass m1{1}, m2{2};
    if ( !(m1==m2) ) std::cout<< m1+m2 <<'\n'; // 3
    std::cout<< (m1+=m1+=m2)  <<'\n'; // 6
    std::cout<<  m1(1,2)      <<'\n'; // 8
}
```

Classes, Operator Overloading

```
+  -  *  /  %  ^  &  |  ~  !  =  <  >
+=  -=  *=  /=  %=  ^=  &=  |=
<<  >>  >>=  <<=  ==  !=  <=  >=  <=>
&&  ||  ++  --  ,  ->*  ->  ()  []
```

Classes, Defaults

src/classes-defaults.cpp

```cpp
class MyClass
{
    int a;
    /* implicitly generated by default when not declared:
    MyClass() : a{} // implicit constructor (indeterminate value for a)
    { }

    MyClass(const MyClass& m2) : a{m2.a} // implicit copy constructor
    { }

    MyClass& operator=(const MyClass& m2) // implicit assignment operator
    {  a=m2.a; return *this; }
    */
};

int main()
{
    MyClass m1;      // constructor
    MyClass m2{m1}; // copy constructor
    m2=m1;           // assignment operator
}
```

Classes, Structs

src/classes-structs.cpp

```cpp
#include <iostream>
class Date // default 'private:', use when preserving invariants
{
  int year,month,day; // invariants: 1<=month<=12, day==31 is not valid when month==2
  friend std::ostream& operator<<(std::ostream&,const Date&); // 'friend' grants access to privates
};

std::ostream& operator<<(std::ostream& os,const Date& d)
{ return os<< d.year <<'-'<< d.month <<'-'<< d.day; }

struct Coordinate // default 'public:', use when there is no invariant
{
  int x=0,y=0,z=0; // no invariant: any value for x,y,z is valid
  Coordinate() {}
  int square_length() { return square_length(c) } const // member function
};

int square_length(const Coordinate& c) // free function, preferred!
{ return c.x*c.x+c.y*c.y+c.z*c.z; }
```

YouTube: *Free your Functions, Klaus Iglberger*

Dynamic Polymorphism

src/poly-dynamic.cpp

```cpp
#include <iostream>
class Shape
{ protected:
    double zoom;
  public:
    Shape(double zoom) : zoom{zoom} { }
    virtual void draw() const {}; // virtual for dynamic/late binding
};

class Circle : public Shape // inherits/extends Shape
{   double rad;
  public:
    Circle(double rad,double zoom=1) : Shape{zoom}, rad{rad} { }
    void draw() const { std::cout<<"Circle: "<< rad*zoom; }
};

class Rect : public Shape // inherits/extends Shape
{   double w,h;
  public:
    Rect(double w,double h,double zoom=1) : Shape{zoom}, w{w}, h{h} { }
    void draw() const { std::cout<<"Rect: "<< w*zoom <<","<< h*zoom; }
};
```

Dynamic Polymorphism

src/poly-dynamic-late-bind.cpp

```
void draw_value    (const Shape  shape) { shape.draw(); }
void draw_reference(const Shape& shape) { shape.draw(); }  // works: virtual + reference
void draw_pointer  (const Shape* shape) { shape->draw(); } // works: virtual + pointer

int main()
{
    Circle c{10};
    draw_value(c);     //                calls Shape.draw()
    draw_reference(c); // Circle: 10
    draw_pointer(&c);  // Circle: 10
}
```

Dynamic Polymorphism

src/poly-dynamic-vector.cpp

```cpp
#include <vector>
#include <memory>
void draw_pointer(const Shape* shape) { shape->draw(); }

void draw_all(const std::vector< Shape* >& shapes)
{   for (auto s : shapes) draw_pointer(s); }

void draw_all(const std::vector< std::shared_ptr<Shape> >& shapes)
{   for (auto s : shapes) draw_pointer(s.get()); }

int main()
{
    std::vector<Shape*> shapes{new Rect{1,2},  // homogenous type Shape*
                               new Circle{3}}; // no vector of references (requires initialization)
    draw_all( shapes );
    for (auto s : shapes) delete s; // deallocate shapes

    draw_all( std::vector< std::shared_ptr<Shape> >{ std::make_shared<Rect>(1,2),
                                                     std::make_shared<Circle>(3)} );
    // no vector of unqiue_ptr (can't copy)
}
```

Dynamic Polymorphism

- Dynamic Polymorphism is flexible:
  - you can have a vector of different sub classes
  - no recompilation when introducing new sub classes
- Dynamic Polymorphism can be inefficient:
  - requires virtual function table memory
  - requires run-time lookup in virtual function table
  - a run-time decision hampers compile-time optimization
  - references/pointers are required (extra memory, fragmentation)

Static Polymorphism, Function Overloading

src/poly-overload.cpp

```cpp
bool test(double d)
{   return d>0; }

bool test(int t)
{   return t%2==0; }

bool test(const MyClass& m)
{   return m.is_valid(); }
```

Static Polymorphism, Function Template

src/poly-max.cpp

```
template <typename T> // template parameter
T max(T a1,T a2) { return a1<a2 ? a2 : a1; }

MyClass max(MyClass a1,MyClass a2) { return (a1-a2)<0 ? a2 : a1; } // specialization

template <typename R,typename T1,typename T2> // multiple template parameters
R max(T1 a1,T2 a2) { return a1<a2 ? a2 : a1; }

int main()
{
    std::cout<< max(1         , 2           ) <<'\n';  // 2
    std::cout<< max(1.0       , 2.0         ) <<'\n';  // 2.0
    std::cout<< max('1'       , '2'         ) <<'\n';  // '2'

    std::cout<< max(MyClass{1}, MyClass{2}  ) <<'\n';  // 2

    std::cout<< max<double,int,double>(1,2.0) <<'\n';  // 2.0
}
```

Static Polymorphism, Non-Type Template Variable

src/poly-non-type.cpp

```cpp
#include <iostream>

template <int N> // non-type template parameter
int factorial()    { return N*factorial<N-1>(); }
template <>      // non-type template specialization
int factorial<0>() { return 1; }

constexpr int factorial(int n) // constexpr function
{   return n<2 ? 1 : n*factorial(n-1); }

int factorial(int n, int value) // tail recursion
{   return n<2 ? value : factorial(n-1, n*value); }

int main()
{
    std::cout<< factorial<10>() <<'\n'; // 3628800  (compile time)
    std::cout<< factorial(10)   <<'\n'; // 3628800  (when possible at compile time)
    std::cout<< factorial(10,1) <<'\n'; // 3628800  (run time)
}
```

Static Polymorphism, Class Template

src/poly-class.cpp

```
#include <iostream>

template <typename T> struct Sum
{
    T sum = 0;
    Sum<T>& add(T t) { sum+=t; return *this; }
};

int main()
{
    Sum<int> sum_int;
    std::cout<< sum_int.add(1).add(2).add(3).sum <<'\n';       // 6
    Sum<char> sum_str;
    std::cout<< sum_str.add('a').add('b').add('c').sum <<'\n'; // '&' (no "abc"!)
}
```

Static Polymorphism, Type Traits

src/poly-class2.cpp

```cpp
#include <iostream>
template<typename T> struct Type_Traits
{ using accum_type = long;          static accum_type init_value() {return 0;} };
template<> struct Type_Traits<char> // Type_Traits specialization for char
{ using accum_type = std::string;   static accum_type init_value() {return "";} };

template <typename T> struct Sum
{   using accum_type= typename Type_Traits<T>::accum_type; // get accumulator type for T
    accum_type sum = Type_Traits<T>::init_value(); // get initial value for T
    Sum<T>& add(T t) { sum+=t; return *this; }
};

int main()
{   Sum<int> sum_int;
    std::cout<< sum_int.add(1).add(2).add(3).sum <<'\n';       // 6
    Sum<char> sum_str;
    std::cout<< sum_str.add('a').add('b').add('c').sum <<'\n'; // "abc"
}
```

Datastructures, STL

- STL: Standard Template Library (static polymorphism)

container	**description**
array<T>	static array (don't use [ ])
vector<T>	dynamic array
queue<T>	dynamic array
list<T>	doubly linked list (fragmentation)
set<Key,Compare>	red-black tree
map<Key,Value,Compare>	red-black tree
unordered_set<Key,Hash,KeyEqual>	hash table
unordered_map<Key,Value,Hash,KeyEqual>	hash table

- Utility library

type	**description**
tuple<T1,T2,T3,...>	static list of hetrogenous types
pair<T1,T2>	two hetrogenous types

STL unordered_map

src/unordered-map.cpp

```cpp
#include <unordered_map>
#include <iostream>
struct Key { int a,b; };
std::ostream& operator<<(std::ostream& os,const Key& k)
{ return os<< k.a <<'^'<< k.b; }

struct Hash
{ std::size_t operator()(const Key& m) const { return m.a+m.b;} };

struct Equal
{ bool operator()(const Key& m1, const Key& m2) const { return m1.a==m2.a && m1.b==m2.b;} };

int main()
{
    std::unordered_map<Key, std::string, Hash, Equal> um{ {Key{2,1},"two"} , {Key{2,2},"four"} };
    um[Key{2,3}]="eight";
    for (const auto& i : um)
        std::cout<< i.first <<"=>"<< i.second <<' '; // 2^3=>eight 2^1=>two 2^2=>four
    auto it=um.find(Key{2,3});
    if ( it!=um.end() ) // test if key is found
        std::cout<<" found: "<< it->second; // found: eight
}
```

STL iterator

src/iterator.cpp

```cpp
#include <iostream>

struct My_Iter { int v; };
int& operator*(My_Iter& i)        { return i.v; }          // get current value
My_Iter& operator++(My_Iter& i1)  { ++(i1.v); return i1; } // step to next value
bool operator!=(const My_Iter& i1,const My_Iter& i2) { return i1.v<i2.v; } // compare iterator

class My_Container
{ public:
    My_Iter begin() { return My_Iter{0}; }  // iterator to first element
    My_Iter end()   { return My_Iter{10}; } // iterator one past last element
};

int main()
{
    My_Container c;
    for (My_Iter it=c.begin(); it!=c.end(); ++it)
        std::cout<< *it <<' '; // 0 1 2 3 4 5 6 7 8 9
    for (auto i : c)
        std::cout<< i <<' ';   // 0 1 2 3 4 5 6 7 8 9
}
```

Algorithm

src/algorithms.cpp

```cpp
#include <algorithm> // use instead of raw for-loops, express intend
#include <numeric>
template <typename T> std::ostream& operator<<(std::ostream& os,const std::vector<T>& v)
{ std::for_each(v.begin(), v.end(), [&](const auto& x){ os<<x<<' ';} ); return os; }

int main()
{
    std::vector<int> v(5);                                  // 0 0 0 0 0
    std::iota       (v.begin(), v.end(), 0);                // 0 1 2 3 4
    std::reverse    (v.begin(), v.end());                   // 4 3 2 1 0
    std::sort       (v.begin(), v.end());                   // 0 1 2 3 4
    std::accumulate (v.begin(), v.end(), 0);                // 10
    std::any_of     (v.begin(), v.end(),
                     [](const auto& x) {return x%2;} );     // true
    std::count_if   (v.begin(), v.end(),
                     [](const auto& x) {return x%2;} );     // 2
    std::transform  (v.begin(), v.end(), v.begin(),         // Python map?
                     [](const auto& x) {return x*10;} );    // 0 10 20 30 40
    std::vector<int> v2;
    std::copy_if    (v.begin(), v.end(), back_inserter(v2), // Python filter?
                     [](const auto& x) {return x<25;} );    // v2: 0 10 20
}
```

Assignment

- Individual
  - infix to postfix: “(a+b)+c*d” => “a b + c d * +”
- Team,
  - Matrix, matrix arithmetic
  - MatrixT<int>, matrix with template argument
  - Str, expression as string “(a+b)+c*d”
  - MatrixT<Str>, matrix expression as string (Type Traits)
  - Algorithm, replace raw for-loops with Algorithm functions
- Questions:
  - Canvas Conference, during your scheduled hours (datanose)
  - Canvas Discussions, for questions useful to others
  - programmeertalen-2020@list.uva.nl, for questions useful to you
  - Things may change, check Canvas announcements