Einfuehrung in Robotik und das Zusammenspiel von Software, Elektronik und Mechanik

2 SemWstd. Sommersemester 2021

Univ.-Lekt. Daniel Schatzmayr

daniel.schatzmayr@uni-ak.ac.at

robotic@uni-ak.ac.at

dɪ:'ʌngewʌndtə

C/C++

https://base.uni-ak.ac.at/cloud/index.php/f/8815164

```
// Start of your program

int main ()
{
    return 0;
}
```

```
// Arduino default code

void setup() {
  // put your setup code here, to run once:

}

void loop() {
  // put your main code here, to run repeatedly:

}
```

```
// Why does Arduino not need the main?

#include <Arduino.h>

int main(void)
{
    init();

    setup();

    for (;;)
    {
        loop();
    }

    return 0;
}
```

C++ keywords

This is a list of reserved keywords in C++. Since they are used by the language, these keywords are not available for re-definition or overloading.

A - C	D - P	R - Z
alignas (since C++11)	decltype (since C++11)	reflexpr (reflection TS)
alignof (since C++11)	default (1)	register (2)
and	delete (1)	reinterpret_cast
and_eq	do	requires (since C++20)
asm	double	return
atomic_cancel (TM TS)	dynamic_cast	short
atomic_commit (TM TS)	else	signed
atomic_noexcept (TM TS)	enum	sizeof (1)
auto (1)	explicit	static
bitand	export (1) (3)	static_assert (since C++11)
bitor	extern (1)	static_cast
bool	false	struct (1)
break	float	switch
case	for	synchronized (TM TS)
catch	friend	template
char	goto	this
char8_t (since C++20)	if	thread_local (since C++11)
char16_t (since C++11)	inline (1)	throw
char32_t (since C++11)	int	true
class (1)	long	try
compl	mutable (1)	typedef
concept (since C++20)	namespace	typeid
const	new	typename
consteval (since C++20)	noexcept (since C++11)	union
constexpr (since C++11)	not	unsigned
constinit (since C++20)	not_eq	using (1)
const_cast	nullptr (since C++11)	virtual
continue	operator	void
co_await (since C++20)	or	volatile
co_return (since C++20)	or_eq	wchar_t
co_yield (since C++20)	private	while
	protected	xor
	public	xor_eq

- (1) — meaning changed or new meaning added in C++11.
- (2) — meaning changed in C++17.
- (3) — meaning changed in C++20.

Note that `and`, `bitor`, `or`, `xor`, `compl`, `bitand`, `and_eq`, `or_eq`, `xor_eq`, `not`, and `not_eq` (along with the digraphs `<%`, `%>`, `<:`, `:>`, `%:`, and `%:%:`) provide an alternative way to represent standard tokens.

In addition to keywords, there are *identifiers with special meaning*, which may be used as names of objects or functions, but have special meaning in certain contexts: `final` (C++11), `override` (C++11), `transaction_safe` (TM TS), `transaction_safe_dynamic` (TM TS), `import` (C++20), `module` (C++20)

Evolution of C++ Standarts

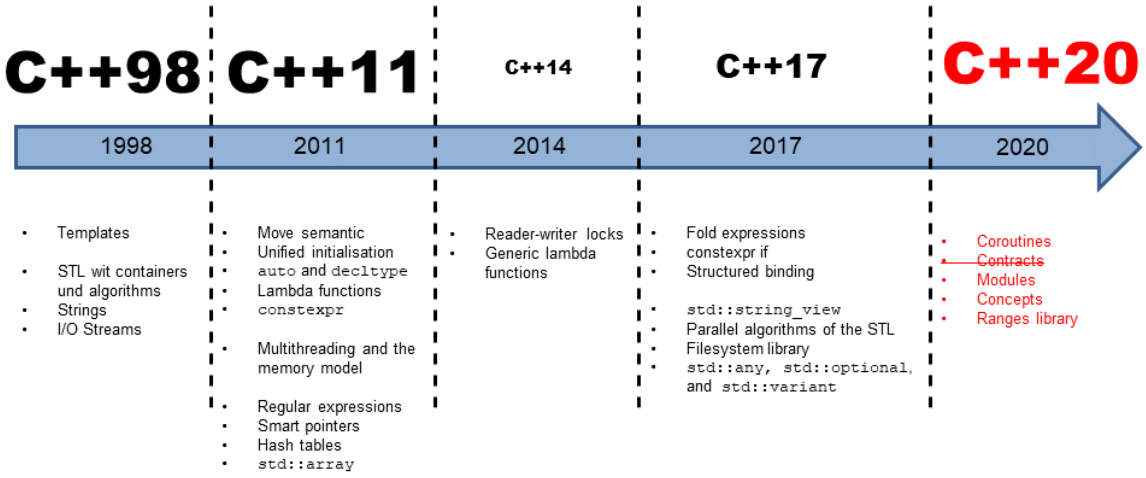


Image from: https://www.modernescpp.com/index.php/c-20-an-overview

Family Tree of Programming Languages

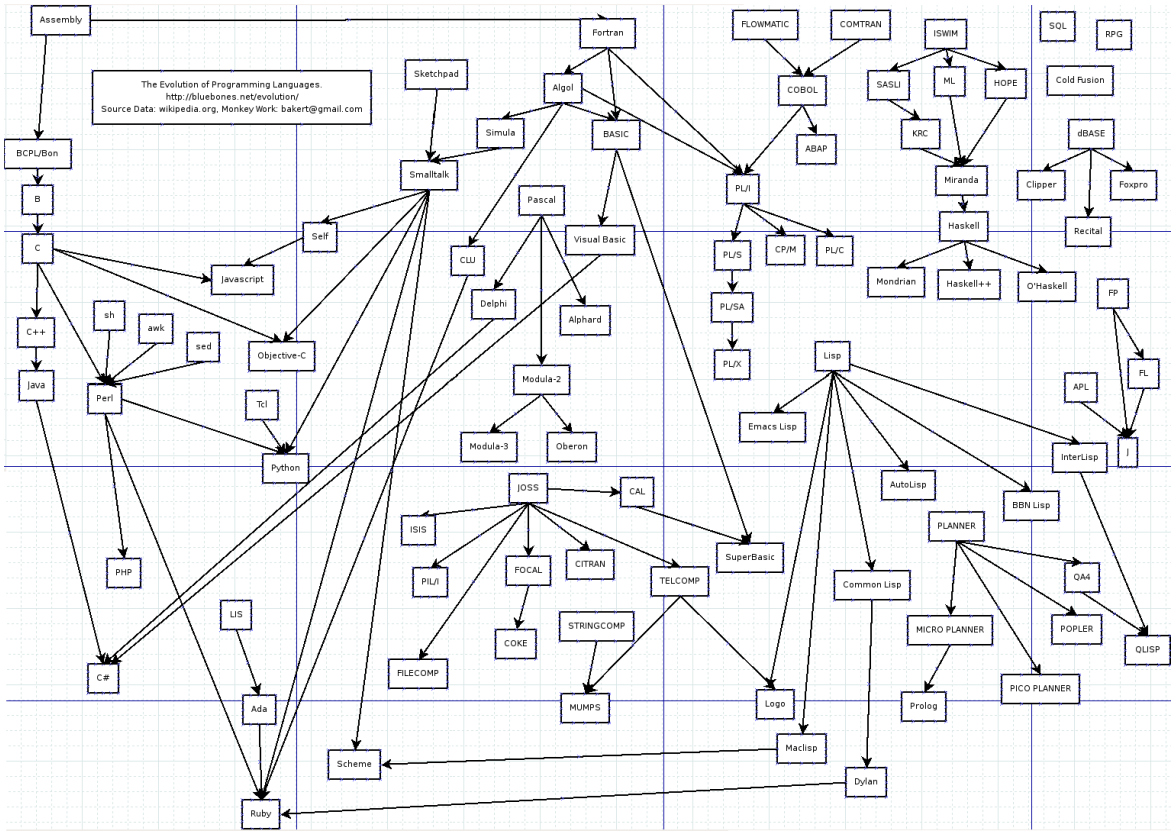
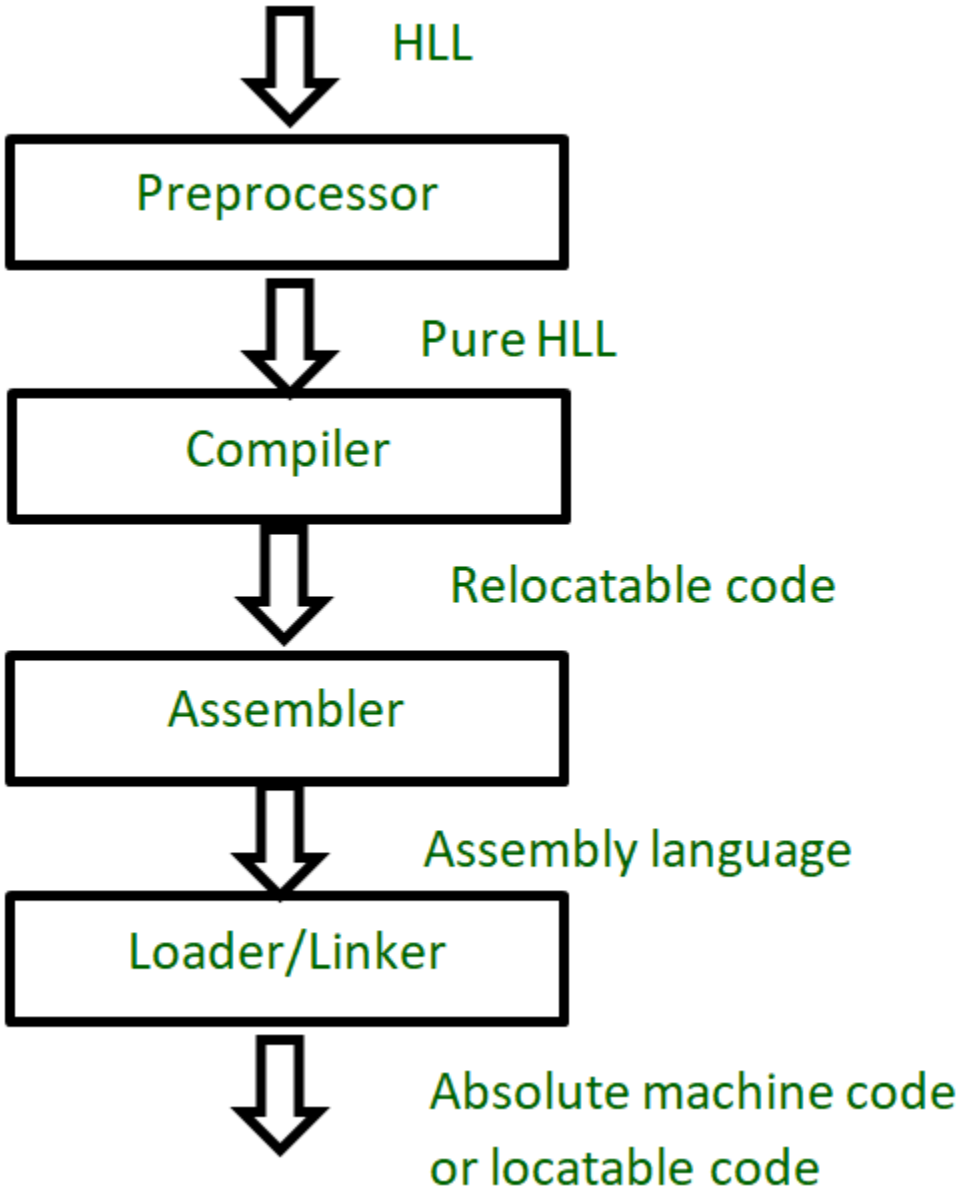


Image from: http://knowtechstuff.blogspot.com/2012/02/evolution-of-c-programming-language.html

Compiler Tool-Chain



Back to Coding: (https://godbolt.org/)

#includes

```cpp
#include <iostream> // <standard_directory> includes search
#include "iostream" // “relative” includes search

int main()
{
    // blabla bla
    // Uses Comments to explain what you are doing
    // Try avoiding leaving deactivated code as comments
    /*
       More form of comments
       std::cout got included from the iostream library
    */

    std::cout << "Hello World" << std::endl;

    return 0;
}
```

Header Files *.h and Source Files *.cpp

C++ Separate Header and Implementation FilesC++ classes (and often function prototypes) are normally split up into two files. The header file has the extension of .h and contains class definitions and functions. The implementation of the class goes into the .cpp file. By doing this, if your class implementation doesn't change then it won't need to be recompiled. Most IDE's will do this for you – they will only recompile the classes that have changed. This is possible when they are split up this way, but it isn't possible if everything is in one file (or if the implementation is all part of the header file).

```
//File: Num.h
class Num
{
  private:
    int num;
  public:
    Num(int n);
    int getNum();
};
```

```
//File: Num.cpp
#include "Num.h"

Num::Num() : num(0) { }

Num::Num(int n) : num(n) {}

int Num::getNum()
{
  return num;
}
```

```
//File: main.cpp
#include <iostream>
#include "Num.h"

using namespace std;

int main()
{
  Num n(35);
  cout << n.getNum() << endl;
  return 0;
}
```

#defines and Pre-Compiler statements

```
#include <iostream>

#define DANIEL

int main()
{

#ifdef DANIEL
    std::cout << "Hallo Daniel" << std::endl;
#else
    std::cout << "Hallo not Daniel" << std::endl;
#endif

}
```

```
#include <iostream>

#define MY_TEXT   "Hallo this is my Text"

int main()
{
    std::cout << MY_TEXT << std::endl;
}
```

#define Macros

```
#include <iostream>

#define abs(x)  (x < 0 ? -x : x)

int main()
{
    std::cout << abs(-10) << std::endl;
}
```

Variables

Int Integer (Ganzezahlen) …-1,0,1,2,3…255…

signed und unsigned int

Float Gleitkommazahlen 3.14159

Bool Booleanzahle … true (1), false (0)

Char/Byte (0-255 bzw 0x00-0xFF)

Selbstdefinierte Variablen

zB: typedefs, structs und class/objects

Variablen koennen Einzeln oder als Felder/Arrays[] definiert werden

---

```cpp
#include <iostream>

int myGlobalVariable = 155;

int main()
{
    int myLocalVariable = 1980;

    std::cout << myGlobalVariable << std::endl;
    std::cout << myLocalVariable << std::endl;
}
```

Arrays []

```cpp
int main()
{

    int myIntArray[10] = {0,1,2,3,4,5,6,7,8,9};

    myIntArray[0] = myIntArray[2] + myIntArray[5];

    return myIntArray[0];

}
```

Operators

- **Assignment operator (=)**
- **Arithmetic operators (+, -, *, /, %)**
- **Compound assignment (+=, -=, *=, /=, %=, >>=, <<=, &=, ^=, |=)**
- **Relational and comparison operators (==, !=, >, <, >=, <=)**
- **Logical operators (!, &&, ||)**
- **Bitwise operators (&, |, ^, ~, <<, >>)**
- **Conditional ternary operator (?)**

Level	Precedence group	Operator	Description	Grouping
1	Scope	`::`	scope qualifier	Left-to-right
2	Postfix (unary)	`++ --`	postfix increment / decrement	Left-to-right
		`()`	functional forms	
		`[]`	subscript	
		`. ->`	member access	
3	Prefix (unary)	`++ --`	prefix increment / decrement	Right-to-left
		`~ !`	bitwise NOT / logical NOT	
		`+ -`	unary prefix	
		`& *`	reference / dereference	
		`new delete`	allocation / deallocation	
		`sizeof`	parameter pack	
		`(type)`	C-style type-casting	
4	Pointer-to-member	`.* ->*`	access pointer	Left-to-right
5	Arithmetic: scaling	`* / %`	multiply, divide, modulo	Left-to-right
6	Arithmetic: addition	`+ -`	addition, subtraction	Left-to-right
7	Bitwise shift	`<< >>`	shift left, shift right	Left-to-right
8	Relational	`< > <= >=`	comparison operators	Left-to-right
9	Equality	`== !=`	equality / inequality	Left-to-right
10	And	`&`	bitwise AND	Left-to-right
11	Exclusive or	`^`	bitwise XOR	Left-to-right
12	Inclusive or	`|`	bitwise OR	Left-to-right
13	Conjunction	`&&`	logical AND	Left-to-right
14	Disjunction	`||`	logical OR	Left-to-right
15	Assignment-level expressions	`= *= /= %= += -= >>= <<= &= ^= |=`	assignment / compound assignment	Right-to-left
		`?:`	conditional operator	
16	Sequencing	`,`	comma separator	Left-to-right

If Statements

```
int a = 2;
int sensorValue = analogRead(sensorPin);
int c = a + sensorValue;

if(c > 100)
{
    Serial.print(" c ist grosser als 100: ");
}else
{
    Serial.print(" c ist kleiner oder gleich 100");
}
```

operator	description
==	Equal to
!=	Not equal to
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to

&& OPERATOR (and)		
a	b	a && b
true	true	true
true	false	false
false	true	false
false	false	false

|| OPERATOR (or)		
a	b	a || b
true	true	true
true	false	true
false	true	true
false	false	false

Loops / for and while loops

```
int a = 0;

for(int i = 0; i < 100; i++)
{
    a = a + i;
}

Serial.print(" a ist: ");
Serial.print(a);

a = 0;

while( a < 1000)
{
    a = a + a;
}
```

```
#include <iostream>

using namespace std;

int main()
{
    int a = 0;

    for(int i = 0; i < 7 ;i++)
    {
        cout << a << " ";
        a = a + 1;
    }

    cout << endl; // do a line break

    while( a > 0)
    {
        a--;
        cout << a << " ";
    }

    return 1;
}
```

Function-definition:

```cpp
#include <iostream>

int HalloDoMath(int a, int b)
{
    int myReturnValue = a + b;
    return myReturnValue;
}

int main()
{
    std::cout << HalloDoMath(12, 8);
    return 1;
}
```

```cpp
#include <iostream>

// forward declaration
int HalloDoMath(int a, int b);

int main()
{
    std::cout << HalloDoMath(12, 8);
    return 1;
}

int HalloDoMath(int a, int b)
{
    int myReturnValue = a + b;
    return myReturnValue;
}
```

Pointers, the Address-Operator & and de-referencing *

```cpp
#include <iostream>

using namespace std;

int main()
{
    int a = 1505;

    cout << "Value: " << a << " Address in Memory: " << &a << endl;
}
```

```cpp
#include <iostream>

using namespace std;

int main()
{
    int a = 1505;
   // Output as hex value
    cout << "0x" << hex << a << " " << dec << a << endl;
}
```

```cpp
#include <iostream>

using namespace std;

int a = 1505;
int b = 1980;

int main()
{
    cout << a << " " << b << endl;

    cout << &a << " " << &b << endl;

    cout << &a + 1 << endl;

    cout << *(&a + 1) << endl;

}
```

Pass by value, pointer and reference

```cpp
#include <iostream>

using namespace std;

int MyFunction(int a, int* b, int& c)
{
    a = a + 1;
    *b = *b + 1;
    c = c + 1;

    return a + *b + c;
}

int main()
{
    int a = 1;
    int b = 2;
    int c = 3;

    cout << a << " " << b << " " << c << endl;

    cout << MyFunction(a, &b, c) << endl;

    cout << a << " " << b << " " << c << endl;
}
```

Structs and classes

```cpp
#include <iostream>

using namespace std;

struct MyStruct
{
    int a = 1;
    int b = 2;
    int c = 3;
};

int MyFunction(MyStruct myS)
{
    myS.a =  myS.a + 1;
    myS.b =  myS.b + 1;
    myS.c =  myS.c + 1;

    return  myS.a +  myS.b +  myS.c;
}

int main()
{
    MyStruct myInst;

    cout << myInst.a << " " << myInst.b << " " << myInst.c << endl;

    cout << MyFunction(myInst) << endl;

    cout << myInst.a << " " << myInst.b << " " << myInst.c << endl;

}
```

```cpp
class MyStruct
{
private:

public:
    int a = 1;
    int b = 2;
    int c = 3;
};
```

Constructors and De-structors

```
#include <iostream>

using namespace std;

struct MyStruct
{
    MyStruct() // default constructor
    {
        cout << "Constructor 1 called..." << endl;
    }

    MyStruct(int A, int B, int C) // constructor with parameters
    {
        a = A; b = B; c = C;

        cout << "Constructor 2 called..." << endl;
    }

    MyStruct(const MyStruct& source) // copy constructor
    {
         cout << "Constructor 3 called..." << endl;
    }

    ~MyStruct()
    {
        cout << "Destructor called..." << endl;
    }

    int a = 1;  int b = 2;  int c = 3;
};

int MyFunction(MyStruct myS)
{
    return  myS.a +  myS.b +  myS.c;
}

int main()
{
    MyStruct myInst;

    cout << MyFunction(myInst) << endl;

}
```

Overloaded functions

```
int MyFunction(MyStruct myS)
{
    return  myS.a +  myS.b +  myS.c;
}

int MyFunction(MyStruct *myS)
{
    return  myS->a +  myS->b +  myS->c;
}

int MyFunction(MyStruct &myS, int andSoOn)
{
    return  myS.a +  myS.b +  myS.c + andSoOn;
}
```

The Keyword static

```
#include <iostream>

using namespace std;

void MyFunction1()
{

    int a = 0;

    a++;

    cout << "1: " << a << endl;

}

void MyFunction2()
{

    static int a = 0;

    a++;

    cout << "2: " << a << endl;

}


int main()
{

   for(int i = 0; i < 5; i++)
   {

       MyFunction1();

       MyFunction2();

   }

}
```

```cpp
#include <iostream>

using namespace std;

int InstanceID = 0;

struct MyStruct
{
   // static int InstanceID ;
    int Id = -1;

    MyStruct() // default constructor
    {
        Id = InstanceID;
        InstanceID++;
        cout << "Constructor 1 called of ID: " << Id << endl;
    }

    MyStruct(int A, int B, int C) // constructor with parameters
    {
        Id = InstanceID;
        InstanceID++;

        a = A;
        b = B;
        c = C;

        cout << "Constructor 2 called of ID: " << Id <<endl;
    }

    MyStruct(const MyStruct& source) // copy constructor
    {
        Id = InstanceID;
        InstanceID++;

        cout << "Constructor 3 called of ID: " << Id << endl;
    }

    ~MyStruct()
    {
        cout << "Destructor called of ID: " << Id << endl;
    }

    int a = 1;  int b = 2;   int c = 3;

    MyStruct operator+(const MyStruct& b) // overloaded opertor
```

```
    {
        MyStruct res;
        res.a = this->a + b.a;
        res.b = this->b + b.b;
        res.c = this->c + b.c;
        return res;
    }

    MyStruct operator+(const int b)  // overloaded opertor
    {
        MyStruct res;
        res.a = this->a + b;
        res.b = this->b + b;
        res.c = this->c + b;
        return res;
    }

};

int MyFunction(MyStruct myS)
{
    myS.a =  myS.a + 1;
    myS.b =  myS.b + 1;
    myS.c =  myS.c + 1;

    return  myS.a +  myS.b +  myS.c;
}

//int MyStruct::InstanceID = 0;

int main()
{
    MyStruct myInst;

    cout << MyFunction(myInst + 1) << endl;
}
```

And more like new and delete
c-style cast, static_cast
const
templates
stl libraries
and still there is a lot more...
