# Begriffe

* Herkunft C, C ist Subset, die meisten C-Programme sind auch C++ Programme.
* kann C Bibliotheken linken
* 1990, 1998 C++98, 03 C++03 (Korrekturen), 2011 C++11, 2014 C++14, 2017 planned

# Was ist neu, grosse Dinge?

Namespaces

Classes, Polymorphism, Inheritence, Overloading

Templates

Strongly Typing

Exceptions

# Was ist neu, kleine Dinge?

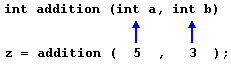
* p7: #include <iostream>
* p11: <inttypes.h> -> <cstdint> (C++11)
* p12: bool mit den zwei Boolean Literals „true“, „false“
* p15: int a = 0; int a(0); Konstruktor-Initialisierung
* p15: string: kein Fundamentaltyp, realisiert als Klasse (analog zu Java)
* p19: Neues Literal „nullptr“;
* p20: #define vermeiden, ersetzen durch const, Vorteil: ist typisiert
* p20: #define Makro: Makros vermeiden, neu mit „inline“
* p26: casting neu mit Funktionsnotation i = (int)f i = int (f)
* p29: cout, insertion operator <<, endl flushes out buffer

## Aufgabe

p34: Was passiert bei if (x)? Geht das immer noch? Testprogramm schreiben.

# Passing by value / reference

p47:



void exchange(int a, int b) -> wie machen

C-Style: mit Zeiger:

void exchange(int\* a, int\* b)

exchange(&x, &y)

Nachteile : Adressen der Aktualparameter müssen übergeben werden, in Funktion müssen alle Zugriffe dereferenziert werden.

Was möchte man? Aufrufe nicht ändern, Funktionen nicht ändern.

void change(int& a, int& b)

## Aufgabe

Implementiere Funktion exchange, soll zwei Werte vertauschen, Ausgabe machen.

Was passiert, wenn eine Referenz nur deklariert aber nicht initialisiert wird?

Was passiert bei: int x = 100; int& y = x; y = 200

## Referenzen versus Pointer

Fazit : Referenzen sind ähnlich wie Zeiger, aber

Zeiger Referenz

Zuweisung immer kann nur einmal zugewiesen werden

Initialisierung egal muss initialisiert werden bei der Deklaration

Zeiger belegen Speicherplatz (in den globals oder auf Stack, 4 Bytes typ.)

gleicher Speicherplatz wie referenzierte Var.

Indirektion, mehrere Levels \*\*\* keine Indirektion

Array of keine Arrays

## Effizienz

call by value: Werte werden kopiert

call by reference: nur Referenzen werden übergeben, Nachteil: es wird auf ursprüngliche Objekte zugegriffen.

# Klassen

p86 und p87 behandeln, Beispiel Rectangle

Klassenname, Element1, Element2...

Elemente sind Daten und Funktionen, Daten sind Standardtypen und komplexe Typen (Klassen)

Jeder Member hat Zugriffskontrolle public, private, protected (default ist private)

Slide 1: hier wird gleich eine Variable rect definiert.

Slide 2: Scope Operator :: , area() direkt in Definition der Klasse definiert, set\_values ausserhalb.

Slide 3: Daten und Funktionen sind Elemente eines Objektes, Witz von OO

Klasse kann Konstruktoren haben,

Slide 4: set\_values direkt im Konstruktor machen, Konstruktor wird bei der Deklaration aufgerufen (automatisch vom Compiler), Was ist der Unterschied zwischen Konstruktor und set\_values?

## Überladen von Konstruktoren

Slide5: Konstruktor ohne Parameter, mit zwei Parametern

Wenn kein Konstruktor -> Default-Konstruktor

Wenn Konstruktor -> kein Default-Konstruktor

## Aufgabe

In eclipse Klasse Led schreiben, UML auf Flipchart

**class** Led {

**int** pin;

**int** times;

string name;

**public**:

**Led**;

**Led**(**int** pin, string name) ;

**Led**(**int** pin, **int** times, string name) ;

**virtual** **~Led**();

**int** **getPin**();

**void** **setPin**(**int** pin);

**void** **blink**();

};

Erzeugt gleich Led.h und Led.cpp, Namen können in Wizard spezifiziert werden.

Implementiere die drei Konstruktoren und die Funktionen in led.cpp

Zusammen:

Testprogramm schreiben, blinken lassen

**void** **Led::blink**() {

**for** (**int** i = 0; i < times; i++)

cout << "led '" << name << "' blinks on pin" << pin << **endl**;

}

**int** **main**() {

Led led0(4, "led0"), led1(2,3,"led1");

// cout << "led1 blinks on pin " << led1.getPin() << endl; // prints LED

led1.blink();

**return** 0;

}

## Aufgaben 1 lösen

# Organisation

Definition der Klasse immer in separatem Headerfile. Member-Funktionen gleich dort wenn kurz.

Viele Klassen haben nur Headerfile!

Nur Standardklassen ohne Extension #include <string>

Name grundsätzlich irgendetwas, Empfehlung \*.hpp

Headerfile in eigenes Verzeichnis include, dieses unter Project Properties einbinden (C/C++ Build -> Settings)

Namespaces: Zeigen wie machen, namespace driver {...}

using namespace driver; // importiert gesamten Namespace

using namespace driver::Led; // importiert die Klasse Led

## Aufgabe

Umbauen mit hpp in include-File und Namespace

# Mehr zu Konstruktoren

Zeigen in led.hpp

1. Alle Konstruktoren mit explicit assignment
2. Alle Konstruktoren mit implicit assignment through initializer list
3. Kombination
4. Konstruktoren mit delegating (c++11)

Achtung: was passiert bei Led led0;? Defaultkonstruktor wird aufgerufen, keine Klammern! Objekt wird auf Stack alloziert, verschwindet dann wieder.

Achtung: Fehler wenn Led led0(); In diesem Fall wird nicht der Konstruktor aufgerufen, sondern eine neue Funktion deklariert mit Namen led0 und einem Rückgabewert Led. Das darf auch mitten im Code passieren.

## Aufgabe

Alles implementieren und testen

# Pointer to Classes

ppt Folie 6, daneben auf Flipchart Tabelle mit allen relevanten Operatoren (aus Tutorial).

Dabei gibt es ein Beispiel mit baz = new Rectangle[2] { {2,5}, {3,6} };

In diesem Fall werden das Array und auch die darin erzeugten Objekte auf dem Heap erzeugt.

# Überladen

Repetition von Funktionen, ist häufig bei Konstruktoren

Überladen von Operatoren

ppt Folie 7 erklären, warum so viele CVector, CVector::operator ist wegen Scope, dann bei Parametertyp, diese ist eine Referenz, und zuletzt für Returntyp

Tabelle in Tutorial zeigen zu Operatoren, jeweils als Memberfunktion oder z.T. als nicht Memberfunktion.

## Aufgabe

Was macht Sinn bei Led? Ideen?

==, <, > vergleichen wie oft noch geblinkt werden muss, ~ Blinkmuster drehen

+ Blinkzeiten addieren

<< überladen für Led, demonstrieren

Zeigen wie gemacht wird, friend nur in der Deklaration, wenn in cpp getrennt definiert, fällt dort friend weg. Erster Parameter wäre this, ist falsch.

# Static Members

Heissen auch Klassenvariablen und –Funktionen

Haben Klassenscope, dürfen nicht direkt in der Klassendeklaration initialisiert werden. Sind Member der Klasse aber auch Member eines einzelnen Objektes, Bsp: Led::count, led0.count

static Funktionen können in hpp oder cpp implementiert werden.

static Variablen müssen in cpp definiert und initialisiert werden. In hpp werden sie ja nur deklariert (und diese Deklaration wird in vielen Files inkludiert), fehlt diese Deklaration, meldet der Compiler „… was not declared in this scope“.

In cpp muss an einem einzigen Ort die Variable noch definiert und initialisiert werden. Das darf nicht im Headerfile sein, sonst mehrere Definitionen. Die Definition muss aber irgendwo sein, sonst heisst es „undefined reference to …“. Der Ort ist egal, weil die Deklaration ja gemacht wurde.

## Aufgabe

Füge count in Led ein, soll private sein, Zugriffsmethode darauf, getCount

# Const

const MyClass obj; object ist read-only, nur const Funktionen können aufgerufen werden, siehe nächste Zeile

int get() const {…} ; Funktion ist const, in Led getPin const deklarieren, es können keine Members verändert werden.

Beispiel zeigen in Tutorial

# Templates

Primär für Biblitheken, Container Klassen zeigen, Templates in Tutorial

#include <vector>

std::vector <int,5> a={1,2,3};

myints::size\_type sz = a.size();

myints::iterator it = a.begin();

while (it++ != a.end()) cout<<\*it<<endl;

## Aufgaben 2 lösen

Meine Lösung zeigen, pwm.cpp und pwm.hpp, überladene Operatoren

Statische Variable zum Zählen der Objekte, list<PWM> für das Speichern aller Objekte, Funktion, die die Periode aller Instanzen ändert. Iterator von list.begin() bis list.end(), warum ++i -> kann schneller sein als i++, spielt hier aber keine Rolle.

Zeit geben, um eigene Lösungen zu korrigieren.

## Aufgabe

Klasse BatteryLevel erstellen, UML an Tafel

led’s direkt in hpp deklarieren und in Konstruktor erzeugen, ist Komposition, wird erzeugt und gelöscht bei Austritt aus Scope.

Verbesserungen in BatteryLevel mit std::vector, hinweisen auf Initialisierung

# EWARM

Kochbuchrezept zeigen für neue Projekte, Organisation der Bibliotheken und Headerfiles.

## Aufgabe

Klassen DigitalIn und Out verbessern, UML an Tafel, Initialisierung soll in Konstruktor gemacht werden. Einsatz der HAL, Einsatz von Enums

# Enum

Zeigen in Tutorial

Unscoped enum:

**enum** States1 {*init*, *start*, *run*, *end*}; // unscoped

**enum** States2 {*init2*, *start2*, *run2*, *end2*}; // unscoped

Wenn enums global -> grosses Durcheinander

Scopes enums

**enum** **class** States3 : **char** {*init*='i', *start*, *run*, *end*}; // scoped

**enum** **class** States4 {*init*, *start*, *run*, *end*}; // scoped

Zeigen wie Ausgabe gemacht werden kann und wie Zuweisungen passieren. Bei unscoped Enums können die einzelnen Enumeratoren nur ein einziges Mal definiert werden! z.B. „init“, „reset“.

# Friend Klassen und Vererbung

In Tutorial Friend Funktionen und Klassen zeigen, Achtung was mit friend markiert wird, ist nicht Teil dieser Klasse. Stiftet oft Verwirrung.

Beispiel mit Square und Rectangle: Rectangle darf auf die Felder von Square zugreifen.

Vererbung zeigen, zeigen was nicht vererbt wird.

Zeigen wie Konstruktoren der Oberklasse aufgerufen werden am Bsp. von Mother, Daugther, Son.

Mehrfache Vererbung ist möglich, eher selten.

# Polymorphismus

Beispiel machen mit Actuator <- Motor / Solenoid <- DCMotor / Stepper

Methode run() in Actuator als virtual. Dazu auch Bsp. aus Tutorial zeigen. Was soll das alles? Liste mit 10 Aktoren, auf alle run() anwenden.

Virtual Funktion: kann überschrieben werden (muss nicht), überschriebene Funktionen können Schlüsselwort virtual weglassen

Abstrakte Klassen mit =0, wenn mindestens eine virtual Methode =0 , dann abstrakte Klasse.

Abstrakte Klasse kann nicht instantiiert werden, aber man kann einen Pointer darauf haben.

Vergleichen mit Interfaces in Java.

Zeigen wie mit Typdescriptoren realisiert.

## Aufgaben 3

# Repetition

Mit normalen Variablen (statisch, automatisch, new) lässt sich kein Polymorphismus realisieren, dazu sind Pointer notwendig. Wo werden diese Variablen erzeugt? Was gilt es dabei zu berücksichtigen? Nächstes Programm zusammen entwickeln.

# Stack / Heap

#include <iostream>

using namespace std;

int a = 10;

int main() {

int b = 20;

cout << "a=" << a << " at addr=" << &a << endl;

cout << "b=" << b << " at addr=" << &b << endl;

return 0;

}

Wo werden Objekte erzeugt ? Java immer auf dem Heap. C auf Heap oder Stack.

class Test {

public:

int a;

Test() : Test(0) {};

Test(int a) : a(a) {cout << "calling constructor" << endl;}

~Test() {cout << "calling destructor" << endl;}

};

int main() {

int b = 20;

cout << "a=" << a << " at addr=" << &a << endl;

cout << "b=" << b << " at addr=" << &b << endl;

Test\* testObj0 = new Test();

cout << "testObj0 at addr=" << &(\*testObj0) << endl;

cout << "testObj0.a at addr=" << &(testObj0->a) << endl;

delete testObj0; // or at end of scope

Test testObj1(100);

cout << "testObj1 at addr=" << &testObj1 << endl;

cout << "testObj1.a at addr=" << &testObj1.a << endl;

return 0;

}

Adressen zusammen anschauen, testObj0 am Ende des Scope automatisch löschen oder mit delete.

Test getAnObject1(int val){

Test obj(val);

return obj;

}

In main()

Test testObj2 = getAnObject1(0);

cout << "testObj2 at addr=" << &testObj2 << endl;

cout << "testObj2.a at addr=" << &testObj2.a << endl;

Das funktioniert. Aber was passiert eigentlich ? Call-by-Value und Return-by-Value. Dabei wird das ganze Objekt kopiert. Das lokale Objekt wird dann gelöscht durch den Destruktor!

Aber Achtung :

However, the compiler is free to generate different code that yields the same observable behaviour. In particular, the standard grants the compilers the right to elide the copying of the return value, and reuse the same storage location for both objects (the local object and the receiving object of the return value). In doing so, the compiler might not need to call the copy constructor, nor the destructor (since it’s reusing the same memory location).

However, this optimization (called “named return value optimization”, NRVO) is not guaranteed by the standard (and in fact it’s not possible to perform everywhere). You cannot assume that it will happen for correctness. In particular, your object still needs a well-defined copy constructor and destructor, otherwise the program is ill-formed.

On the other hand, you can reasonably expect all modern compilers to perform this optimization where ever it is possible. You can therefore (usually) rely on this optimization from a performance point of view.

Aber was passiert hier ?

Test\* getAnObject2(int val){

Test obj(val);

return &obj;

}

Test& getAnObject3(int val){

Test obj(val);

return obj;

}

Wir referenzieren ein Objekt auf dem Stack. Testen wir mit einer Funktion:

void sum () {

Test obj1 = getAnObject1(10);

Test obj2 = getAnObject1(100);

cout << "sum1 = " << (obj1.a + obj2.a) << endl;

Test\* obj3 = getAnObject2(10);

Test\* obj4 = getAnObject2(100);

cout << "sum2 = " << (obj3->a + obj4->a) << endl;

Test& obj5 = getAnObject3(10); // Test obj5 would be ok

Test& obj6 = getAnObject3(100);

cout << "sum3 = " << (obj5.a + obj6.a) << endl;

}

Diese Schwierigkeit hat man nicht auf dem Heap!

# Special Members

Default constructor: wenn keiner vorhanden, kann z.B. folgendes nicht deklariert werden Led led1;

Destructor: wenn in Objekt Zeiger vorhanden sind, diesen Speicher freigeben. Wird für alle Speicherklassen (static, auto, new) aufgerufen, automatisch am Ende des Scopes oder mit delete.

Copy constructor: Anwendung zeigen

Gibt es auch in Klasse PWM, beim Einfügen in die Liste mit push\_back(\*this) wird der Copy Constructor aufgerufen. Es gibt also eine Kopie. Und darum wurden die Perioden der pwm’s auch nicht verändert!

Macht shallow copy. Alle Member werden kopiert, auch Zeiger. Wenn deep copy -> überschreiben.

Copy assignment:

Move Assignment und Move constructor: für Objekte ohne Namen. Beispiele anschauen.

## Lvalue und rvalue

Lvalue: lokalisierbar im Speicher, ist Variable, auf der linken Seite einer Zuweisung

Rvalue: lesbar, readable, auf der rechten Seite einer Zuweisung

Lvalue = rvalue

Deklaration mit &&

String&& str heisst str ist rvalue reference

## Beispiel Klasse PWM

Ausgabe in Konstruktor

Ausgabe in Destruktor

Copy Constructor überschreiben und Ausgabe machen:

PWM (const PWM& pwm) {cout << "calling copy constructor" << endl;}

Aber Achtung : Jetzt wird nicht kopiert !

Warum hat es 4 Mitglieder in der Liste -> im Destruktor aus Liste löschen, count—

Liste deklarieren als PWM\*.

Push\_back(this).

In setPeriod: iterator ist Zeiger auf Element, also Zeiger auf Zeiger auf PWM, dereferenzieren mit (\*i)->period.

# Exceptions

In Tutorial zeigen.

Was machen bei bad\_alloc? Terminieren, Reset?

# UART in IAR

UML zeichnen, was wollte ich, structure UART\_HandleTypeDef in Klasse speichern, gibt Probleme mit Hal\_init().

Darum global speichern (unschön), in Klasse nur Zeiger drauf und Initialisierung machen.

Zusammen lösen.

Uart debug = Uart(&uart2);

Uart laserScanner = Uart(&uart3);

Was wäre noch besser? Im Moment können wir nur Byteströme senden, Formatierung müssen wir selber machen.

# Logger Framework

# Best Practice in C++

|  |
| --- |
| **#define statements should be avoided.** |
| const double PI = 3.1415;     // NOT: #define PI 3.1415 |
| #define is not part of the C++ language definition and since C++ provides constructs that make #define superflous, these should be preferred.  Instead of #define constants, use const variables, or better yet: Use member functions to access constants (See [[1]](http://geosoft.no/development/cpppractice.html" \l "ref1)).  Instead of typeless macros use functions with template parameters. |

|  |
| --- |
| **The iostream library should be preferred to stdio.** |
| #include <iostream>     // NOT: #include <stdio.h> |
| The stdio has been replaced by the much more powerful iostream library, and when programming C++ the latter should be preferred. |

|  |
| --- |
| **References should be preferred to pointers.** |
|  |
| Pointers should be used if and only if it should be possible for the referred object to be *null*.  Example: If a person is modelled as an object, the persons parents should be *references* to persons (everybody has parents) while siblings should be *pointers* to persons. |

|  |
| --- |
| **const should be used wherever possible.** |
|  |
| The *const* keyword is a documentation aid.  In particular, member functions that does not affect the state of an object is to be declared *const*. These are the only functions which may be invoked on a const object. |

|  |
| --- |
| **Pass-by-value should be avoided for objects.** |
| myMethod (const SomeClass &object) // NOT: myMethod (SomeClass object) |
| There are two reasons for this. First is of performance. Pass by value for objects always involves creating a temporar object using the copy constructor of the objects class and destroying the object again on the method exit.  Second reason is that objects passed by value through a base-class variable will in effect behave as a base-class object without the extended information defined by the derived class. |

|  |
| --- |
| **Variable argument lists (...) should be avoided.** |
|  |
| Variable argument lists prohibit strong type checking as provided by C++. In most cases variable argument lists can be exchanged by function overloading and by using default arguments. |

|  |
| --- |
| **A copy constructor and an assignment operator should always be defined for classes with dynamically allocated memory ([[1]](http://geosoft.no/development/cpppractice.html" \l "ref1), Item 11).** |
|  |
| Unless the copy constructor and the assignment operator is explicitly defined, the compiler will generate them automatically. If the class use dynamically allocated memory, the default generated copy constructor and assignment operator will most often not behave as expected. The exception to this if when multiple objects of the same class indeed should share a data area. In this case it is necessary to make sure that the shared data is not deallocated as long as there are references to it. |

|  |
| --- |
| **The assignment operator should always return a reference to \*this.** |
| MyClass& MyClass::operator= (const MyClass& rhs)  {    ...    return \*this;  } |
| This is done to have the user defined assignment operator mimic assigment of built in types.  For instance will the following statement be both possible and meaningful:  MyClass a, b, c; a = b = c; |

|  |
| --- |
| **"Isa" relationship should be modelled by inheritance, "has-a" should be modelled by containment.** |
| class B : public A     // B "is-a" A  {    ...  }    class B  {    ...     private:      A a\_;     // B "has-a" A  } |

|  |
| --- |
| **Exceptions should be caught by reference.** |
| try {    ...  }  catch (Exception& exception) {    ...  } |

|  |
| --- |
| **A public member function must never return a non-const reference or pointer to member data (**[**[5]**](http://geosoft.no/development/cpppractice.html#ref5)**, Rule 29).** |
|  |
| Returning a non-const reference to memeber data violates the encapsulation of the class. |

[https://isocpp.org/wiki/faq/containers#arrays-are-evil](https://isocpp.org/wiki/faq/containers" \l "arrays-are-evil)

std ::string anstatt array of char

Benütze class statt struct

Studiere Klassenbibliothek:

<http://en.cppreference.com/w/cpp/atomic/atomic>

<http://en.cppreference.com/w/cpp/thread>

<http://en.cppreference.com/w/cpp/container>

smart-Pointers