Prog C++

HS 2023, Prof. Dr. Christian Werner

Fabian Steiner, 20. April 2024

0.0.0.

1 Basics

1.1 Trivia

Entstanden durch Stroustrup 1986.

1.2 Grundsätzlich

C++ ist sehr ähnlich zu C. Es gibt aber einige wichtige Änderungen zu C:

- zu benutzender Compiler heisst jetzt “clang++”
- Nativen Bool Typ : **Bool**
- richtiges Konstanten Schlüsselwort : **constexpr**
- Standartbibliotheken haben kein .h mehr beim Aufrufen
- C Standartbibliotheken Können weiter verwenden aber haben ein C im Header
- Char-Literale (z. B. ’A’) haben in C++ den Datentyp char (in C war es int)
- Es gibt benannte Namensräume. Wichtigster Namensraum für die STL: std
- ...

1.3 Sourcecode Hello world CPP

```
#include <iostream>
using namespace std;
int main()
{
    cout << "Hello World!" << endl;
    return 0;
}
```

1.4 Streams

Ein Stream repräsentiert einen generischen sequentiellen Datenstrom. Z.b: ein Eingabefeld, Dateien oder Netzwerktraffic.
Die wichtigsten Operatoren sind:

- << Inserter → Daten einfügen
- >> Extractor → Daten herausholen

Für definierte Klassen sind diese Operatoren bereits definiert.

1.4.1 Standardstreams

- **cin** standard Eingabe
- **cout** standard Ausgabe
- **cerr** standard Fehlerausgabe
- **clog** mit cerr gekoppelt aber etwas anders

1.4.2 Streamformatierung

Formatierungen der Streams kann mit folgenden Schüsselwörtern erreicht werden:

Flag	Wirkung
boolalpha	bool-Werte werden textuell ausgegeben
dec	Ausgabe erfolgt dezimal
fixed	Gleitkommazahlen im Fixpunktformat
hex	Ausgabe erfolgt hexadezimal
internal	Ausgabe innerhalb Feld
left	linksbündig
oct	Ausgabe erfolgt oktal
right	rechtsbündig
scientific	Gleitkommazahl wissenschaftlich
showbase	Zahlenbasis wird gezeigt
showpoint	Dezimalpunkt wird immer ausgegeben
showpos	Vorzeichen bei positiven Zahlen anzeigen
skipws	Führende Whitespaces nicht anzeigen
unitbuf	Leert Buffer des Outputstreams nach Schreiben
uppercase	Alle Kleinbuchstaben in Grossbuchstaben wandel

2 Objektorientierung

Objektorientierung soll es erleichtern eine Abbildung der Realität zu erstellen. Viele Dinge aus der Wirklichkeit können als Modell dargestellt / simplifiziert werden.
Diese Modelle können in Software in sogenannte Objekte “umgewandelt” werden.

2.1 Objekte

Objekte stellen Dinge, Sachverhalte oder Prozesse dar. Sie sind ein rein gedankliches Konzept.
Sie kennzeichnen sich durch:

- eine Identität, welche es erlaubt Objekte voneinander zu unterscheiden
- statische Eigenschaften zur Darstellung des Zustandes des Objekts in Form von Attributen
- dynamische Eigenschaften zur Darstellung des Verhaltens des Objekts in Form von Methoden

2.2 Klassen & Instanzen

Ähnliche Objekte können in **Klassen** zusammengefasst werden, um Programmieraufwand tiefer zu halten. Verwendete Objekte werden als **Instanz** eingesetzt. Diese Objekte haben dieselben Attribute, allerdings andere Werte.
Eine Beispielklasse:

```
 #include <String>//Stl Strings
class Student {

    public:
        int IDNumber;
        void doStuff(int time = 0);
    private:

        std::string name;//String
        float bierkonsum;
        long int investedTimeInET;
};
//implementierung der doStuff Funktion
void Student::doStuff(int time = 0) {
    //ToDo
}

int main(){
    //instanz erzeugen
    Student typETStudent;
    //Attribute veraendern
    typETStudent.IDNumber = 1;
    //Methode rufen
    typETStudent.doStuff();
    return 0;
}
```

2.2.1 Header Datei

Reihenfolge in der Header Datei

- Dateikommentar mit Lizenzvereinbarung.
- “Includes” des verwendete System Header
- “Includes” der Projektbezogenen Header
- Konstanten
- typdefs und definition von Strukturen
- Allenfalls extern-Deklraration von Global Variablen
- Funktionsprototypen, ink, Kommentar der Schnittstelle, bzw. Klassendeklarationen

Pro header Datei sollte nur eine Klasse deklariert sein.

2.2.2 Reihenfolger der Implementation

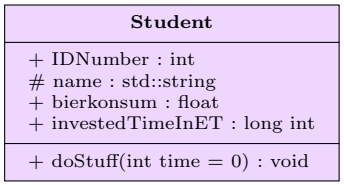
- Dateikommentar mit Lizenzvereinbarung
- “includes” der eigenen Header
- “includes” der Projektbezogenen Header
- “includes” der verwendeten System Header
- allenfalls globale und statischen Variablen
- Präprozessor-Direktiven
- Funktionsprototypen von lokalen, internen Funktionen (in nameless Namespace)
- Definition von Funktionen und Klassen

2.3 UML

Die “Unified Modeling Language” ist eine normierte Sprache um Objekte Grafisch darzustellen. Sie geht extrem ins Detail, allerdings

2.3.1 UML-Klassendiagramm Notation

Besteht immer aus 3 Boxen:



Zu oberst der Name, in der Mitte Attribute(Variablen) und unten Methoden.
Die Methoden fangen mit einem Symbol an welches die Sichtbarkeit definiert:

- + public (überall sichtbar)
- - private (nur in aktueller klasse sichtbar)
- # protected (in aktueller und Unterklassen sichtbar)

2.3.2 class & struct

Eine class und struct können fast identisch verwendet werden. Eine class hat standartmässig sichtbarkeit private, struct public(wenn nichts definiert wird).

2.4 Inkludierte Dokumentation

Mann im H-File direkt die Dokumentation zu der Schnittstelle schreiben. Das ermöglicht das einfachere Verwenden der Schnittstelle. Dies wird in einem Blockkommentar, nach folgendem Muster realisiert:

```
/**
 * @brief Kurzzusammefassung der Klasse
 *
 */
class stuff{
public:
    /**
     * @brief Funktion von x
     *
     */
    double x;
    /**
     *
     * @brief Kurzzusammefassung der Funktion
     *
     * @param eingang Funktion des Parameter
     * @return nix, aber falls...
     *
     */
    void doStuff(stuff eingang);
};
```

Es kann immer mehr geschrieben werden, es sollte sich aber auf das nötige beschränkt werden.

3 Konstruktoren und Destruktoren

Konstruktoren und Destruktoren sind spezielle Methoden von Klassen. Diese haben immer denselben Namen wie die Klasse und auch **kein** Rückgabetyp(auch nicht void).
Aufrufparameter haben folgende Bedeutung:

- **“Keine”** : Default Konstruktor
- **“const-Referenz auf eigene Klasse”** : copy Konstruktor
- .....

3.1 Konstruktoren

Konstruktoren bereiten die Instanz auf ihre Funktion vor.

Wichtig ist das ein Konstruktor immer **alle** Attribute der Klasse initialisiert. Konstruktoren werden wie folgt aufgerufen:

3.1.1 User-Defined

- Default → ohne Aufrufparameter
- Copy → mit einer const Referenz auf eine andere Instanz
- sonstige → werden anhand ihrer Aufrufparameter unterschieden, Sprechweise:“überladen”

Werden verschiedene Konstruktoren definiert, der Default aber soll erhalten bleiben, muss dieser in der Klassendefinition zuerst stehen.

3.1.2 Implizit

Falls ein expliziter nicht angegeben wurde, dieser jedoch aus technischen Gründen benötigt wird:

- Default → macht nichts, alle Parameter bleibe uninitialisiert
- Copy → dieser kopiert alle Attribute der anderen Instanz eins-zu-eins(bytewise). Problematisch, wenn die Instanz Pointer auf etwas hat.
- ...

Wenn ein Objekt ein Pointer auf einen Wert oder allokierten Speicher hat, muss eine spezielle Copy Methode verwendet werden, da der Pointerwert einfach kopiert wird. Falls dann das erste Objekt den Speicher frei gibt, dann entsteht ein falscher Speicherzugriff. DH mus dem Objekt ein separater Speicherbereich gegeben werden.
Konstruktoren werden immer dann gerufen (evtl. auch **implizit**) wenn ein neues Objekt in den Speicher gelegt wird.

3.2 Destruktor

Destruktoren entfernen das Objekt aus dem Speicher. Hat keine Aufrufparameter und auch kein Rückgabewert. Wenn der Destruktor fertig ist, sollte kein Speicher mehr vom Objekt belegt sein.
Destruktoren sollten immer **Virtuell definiert werden** und der Funktionsname beginnt mit einer Tilde. Der Destruktor wird evtl. auch Implizit aufgerufen z.b., wenn aus einer Funktion wieder herausgesprungen wird.

3.3 Beispiel

```
#include <iostream>
class storage{
    public:
        storage();
        virtual ~storage();
        void add(int in);
        void nix(storage inC);
    private:
        int* data;
        int size;
};//eine Klasse die Int Werte speichertaehnlich wie ein array
// ---- End h File
storage::storage(){//Konstruktor
    data = nullptr;
    size = 0;
}
storage::~storage(){//Destruktor
    delete[] data;
    data = nullptr;
    size = 0;//Speicher der allokiert wurde sollte hier
     freigegeben werden
}
void storage::add(int in){
    //todo
}
void storage::nix(storage inC){
    //todo
}
// ---- End cpp File
int main(){

    storage* i1 = new storage;//default konstruktor
    storage i2;//default konstruktor
    i1->nix(i2);//Call-by-Value. Eine Kopie von i2 wird auf den
     Stack gelegt -> Copy-Konstruktor!

    delete i1;//destruktor von i1 wird explizit aufgerufen
    return;//destruktor von i2 wird implizit aufgerufen
}
```

4 Initialisierungslisten und direkte Initialisierung

Die Initialisierung von Datentypen kann auf verschiedene Arten erreicht werden. Es wird zwischen Zuweisung und Initialisierung unterschieden.

4.1 POD's

“plain old datastructure” / built in

```
// Zuweisung fuer POD:

int i; // Speicher fuer Instanz wird alloziert, Wert ist noch
     uninitialisiert
i = 5; // Wert (der an anderer Stelle im Speicher stehen muss)
     wird zugewiesen

// Initialisierung fuer POD:

int i = 5; // Instanz wird mit Wert 5 in den Speicher gelegt (C-
     Style)
int i(5); // Instanz wird mit Wert 5 in den Speicher gelegt (C++-
     Style bis C++11)
int i{5}; // Instanz wird mit Wert 5 in den Speicher gelegt (neuer
      C++-Style)
// Aus Effizienzgruenden bevorzugen wir (fast) immer die (direkte)
     Initialisierung!
```

4.2 Non PODs

```
// Mitarbeiter m;
// Vorsicht: Falls sizeof(Mitarbeiter) gross ist, muss hier viel
     kopiert werden:
m = stefan;

// Initialisierung fuer non-POD:
Mitarbeiter m = stefan; //Copy-Initialisierung (copy ctor)
Mitarbeiter m(stefan); //Copy-Initialisierung, C++-Schreibweise
Mitarbeiter m{stefan}; //Copy-Initialisierung, bevorzugte C++-
     Schreibweise seit c++11, aber: noch immer muss kopiert werden
     !

//Besser: direkte Initialisierung auch fuer non-PODs:
Mitarbeiter m("Stefan Melcher", 182.0, 23, 12, 2003); // (vor C
     ++11)
Mitarbeiter m{"Stefan Melcher", 182.0, 23, 12, 2003}; //
     bevorzugte Schreibw. seit C++11
// Die Instanz wird ohne Umwege mit den gewuenschten Werten in den
      Speicher geschrieben, sofern ein geeigneter user-defined
     ctor vorhanden ist!
```

4.3 User defined Ctor

Um eine Klasse richtig initialisieren zu können muss der Ctor korrekt definiert werden. Eine sogenannte Initialisierungsliste:

```
student::student(int semester,
                 long int eltVerzweiflung) :
    semester{ _semester},
    eltVerzweiflung{_eltVerzweiflung}
{//die Reihenfolge der Initialisierung ist durch die Reihenfolge
     im Speicher gegeben!
    //Rumpf kann leer bleiben Initialisierung bereits fertig
}
```

5 Assertions

Assertions sollten Annahmen über korrekte Wertebelegung darstellen. Sie sind kein C spezifisches Konzept und sollten nur zu Testzwecken eingesetzt werden. Im Releasebuild sollten sie deaktiviert werden. Dafür gibt es spezifische Präprozessorflags. Der Header assert.h bzw. cassert stellt hierfür ein Präprozessor-Makro “assert(Bedingung)” zur Verfügung, um solche Zusicherungen auszudrücken.
Beispiel:

```
#include <cassert>
//#define NDEBUG // Deaktiviert Assertions im Projekt
bool testStuff(int* in1){

    assert(in1 != nullptr);
    //assertion

    ;//restliche implementation einer Funktion

}
```

Das obige Beispiel hat eine Assertion. Diese würde im Test, falls ein Nullpointer übergeben wird ein Fehler ausgegeben.

5.1 This-Pointer

Mit dem “this Pointer” kann ein Pointer des aufrufenden Objekt zurückgegeben werden. Bsp:

```
class Stuff{
    public:
    Stuff& funktion(int valIn);
    private:
    int value;
};
Stuff& Stuff::funktion(int valIn){
    //do Stuff with val z.b. addieren auf interne Variabel
    this->funktion(0);// auch moeglich
    return *this;//this pointer
}
int main(){
    Stuff memb;
    memb.funktion(1).funktion(2).funktion(3);
}
```

Nacheinander werden diese Funktionen durchlaufen da Funktion(2) vom zurückgegebenen Pointer aus Funktionsaufruf 1 aus gerufen wird. Die funktion wird “Kaskadiert”

6 Overloading

Operator Overloading bedeutet das bestehende Operatoren “überladen” werden im Sinne neu definiert / drüber laden und erhalten neue bedeutung für eine Klasse.

6.0.1 Methodenoverloading

Methoden dürfen in Cpp gleich heissen, sie müssen aber unterschiedliche Aufrufparameter haben.

```
int addiere(int v1, int v2);
double addiere(double d1, double d2);

int i = addiere(2, 7); // ruft addiere(int, int)
double d = addiere(2.3, 2.0);// ruft addiere(double, double)
```

6.1 Operator overloading

Es können die inkludierten Operatoren von Cpp überladen werden. Erlaubte Operatoren sind:

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

Nicht erlaubte Operatoren sind: “.” “.*” “::” und “?:”.
Es wird zwischen overloading global und in Methode unterschieden.

6.1.1 Global

Globales overloading funktioniert mit einer als “friend” deklarierten Funktion. Als friend deklarierte Funktionen agieren als globale Funktion und haben vollen zugriff auf alle Parameter einer Klasse. Sie sind daher zu vermeiden da sie eher ein Workaround sind und zu Problemen führen können.

Syntax : “return-type” operator“typ”(type val1, type val2);
BSP:

```cpp
// H file
class Dozent{
    friend Dozent& operator+(Dozent& in1, Dozent& in2);//der +
     Operator wird ueberschriben
    private:
    int numb;
};

Dozent& operator+(Dozent& in1, Dozent& in2);

// ------ end H file ------

// ------ start cpp ------
Dozent& operator+(Dozent& in1, Dozent& in2){
        in1.numb *= in2.numb;
        return in1;
}
```

Die Funktion kann von überall aus dem Programm ausgeführt werden. Es sollte nur dann verwendet werden wenn der erste Parameter (links) nicht eine Instanz der Klasse ist da der Erste Aufrufparameter immer die Instanz sein muss.

6.1.2 Methoden

Ein Operator kann auch als Methode einer Klasse definiert werden. Sie haben nur noch ein Aufrufparameter.
Syntax : "return-type" operator"typ"(type in);
Der 2. Parameter ist das Objekt selbst, welche die Methode aufruft.

```cpp
// H file
class Dozent{
    friend Dozent& operator+(int in1, Dozent& in2);
    public: // friends immer vors public
    Dozent& operator+(Dozent& in1);//der + Operator wird
     ueberschriben(Falls eine andere Instanz addiert wird)
    Dozent& operator+(int in1);//der + Operator wird ueberschriben
     (falls ein Int addiert wird)
    private:
    int number;
};
Dozent& operator+(int in1, Dozent& in2);//dieser Fall muss so
     geloest werden

// ------ end H file ------

// ------ start cpp ------
Dozent& operator+(int in1, Dozent& in2){
        in2.number += in1;
        return in2;
}
Dozent& Dozent::operator+(Dozent& in2){
    number += in2.number;
    return *this;
}
Dozent& Dozent::operator+(int in2){
    number += in2;
    return *this;
}
// ------ end Cpp file ------
main(){
    Dozent D1;
    Dozent D2;
    D1 + D2;
    D1 + 1;
    1  + D1;//durch Global overloading abgedeckt
}
```

Die obigen Funktionen machen alle dasselbe. Sie unterscheiden sich durch die unterschiedlichen Argumente. Diese decken verschiedene Eingabetypen ab. Es wäre schlechter Stil, die Typkonvertierung dem Compiler implizit zu überlassen! Ein Fall, der bei dem das anderstypige Argument zuerst kommt, muss noch per Globalem Overloading gelöst werden, da der Erste Aufrufparameter, beim overloading mit Methoden, immer zuerst kommt.

6.2 Default Argumente

Default Argumente werden, als Parameterwert verwendet, wenn keiner mitgegeben wird. Parameter werden von links nach rechts werten belegt. Daher, wenn manche Aufrufparameter keinen Default wert erhalten, müssen diese weiter links, als welche mit Default wert stehen.

```cpp
void func(int a = 0, int b = 0, int c = 0){}
func();
func(0);
func(0, 0);
func(0, 0, 0);
//Alle Funktionsaufrufe valide

void func2(int a, int b = 0, int c = 0);//korrekte Reihenfolge
```

Default Argumente können überall verwendet werden, bei Aufrufparameter von Operatoroverloading macht es lediglich keinen Sinn.

6.3 Getter & Settermethoden

Getter und Settermethoden werden Typischerweise verwendet um lese und Schreibzugriffe auf eine Klasse zu regeln. Attribute sind typischerweise privat. Mit einer Get oder Set methode kann dieser kontrollierter angepasst werden.

```cpp
#include <string>
class Stuff{
    public:
    const std::string& getName() const;
    protected://erlaubt schreibenden Zugriff von Unterklassen
    void setName(const std::string& in);
    private:
    std::string name;//privates Attribut
};
const std::string& Stuff::getName() const{
    return name;
}
void Stuff::setName(const std::string& in){
    name = in;
}
```

7 Vererbung

Vererbung erlaubt es Attribute und Methoden von anderen Klassen zu übernehmen. Die Oberklasse, Baisklasse oder superklasse "vererbt" an eine Unterklasse, derived class oder eine Spezialisierung. Bei der Vererbung werden immer **Alle** Attribute oder Methoden weitergegeben.

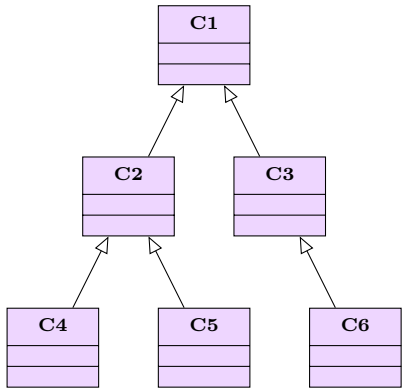
7.0.1 UML

In einem UML Diagramm wird vererbung wie folgt dargestellt:

Unterklasse erbt (die Attribute und Methoden) von Superklasse. Dies stellt eine "ist ein beziehung" dar. Es ist sehr wichtig, dass die Pfeilspitze **genau so** aussieht wie im in der Darstellung da ansonsten Verwechslungsgefahr mit anderen Mechanismen entstehen könnte.

7.0.2 UML++

Vererbung ist auch über mehrere Stufen Möglich:



C4 Erbt alles von C2 und C1, ...

7.0.3 Syntax

```cpp
class Unterklasse : public : Superklasse{
    ...
} ;
```

Unterklasse erbt Superklasse.

Die Sichtbarkeit ist durch das Public definiert. Es ist möglich die Sichtbarkeit aller Attributen und Methoden einzuschränken. Mit **Public** wird alles mit der originalen Sichtbarkeit übernommen. Mit **protected** wird alles was public ist protected. Mit **private** wird alles private (nicht sehr nützlich).
Wird eine Vererbung private durchgeführt, sind keine Attribute oder Methoden mehr sichtbar.

7.0.4 C/D tor chaining

Grundsätzlich initialisieren ctors nur die eigene Klasse. Dh Superklassen initialisieren sich selbst. Eine Subklasse initialisiert sich, indem sie zunächst einen ctor der Superklasse aufruft und dann sich selbst **IMPLIZIT**. Man das auch explizit machen:

```cpp
Unterklasse::unterklasse(int _initvalOberklasse,
                         int _initvallUnterklasse):
    Oberklasse{_initvalOberklasse},
    unterklassenAttribut{_initvallUnterklasse}
{
}
```

Note: Da der Konstruktor aufgerufen wird kann auch ein evtl Private Attribut der Oberklasse initialisiert werden. Gleiches gilt mit dem Dtor, nur umgekehrte Reihenfolge (erst wird der Dtor von der subklasse aufgerufen, dann.....).

7.1 Überschreiben von Methoden

Geerbte Funktionen können "überschrieben" werden. Die Unterklasse definiert eine neue Methode mit demselben Namen. Allerdings kann es dann zu Mehrdeutigkeit kommen:
Im folgenden Beispiel haben eine Subklasse und eine Oberklasse eine Print Funktion welche **Nicht** virtual gekennzeichnet ist:

```
int main() {
    Subclass p;          // Klassenerstellung
    p.print();           // print von Subclass, ok
    Subclass* sPtr = &p;// Subclass Pointer
    sPtr->print();       // print von Subclass, ok
    Topclass* tPtr = &p;// !! Topclass Pointer !!
    pPtr->print();       // print von Topclass!!!!!!
    Subclass& sRef = p; // Subclass ref
    sRef.print();        // print von Subclass, ok
    Topclass& tRef = p; // !! Topclass ref !!
    pRef.print();        // print von Topclass!!!!!!
}
```

Dieses Beispiel soll zeigen, dass Pointer, welche von auf einer Oberklasse auf eine Unterklasse zeigen auf die "Eigenen" Funktionen zeigt, solange diese nicht überschrieben worden sind. Dies kann als Verhalten gewünscht sein.

7.2 Virtual, scopeoperator, override und final

7.2.1 virtual

Mit virtual kann eine Methode kennzeichnen, dass diese Methode von einer Unterklasse überschrieben wird. Somit würde das obige Beispiel nicht mehr funktionieren und es wird immer die Printfunktion der Unterklasse aufgerufen. In einer Unterklasse ist virtual nicht mehr zwingend zu schreiben. Eine Klasse mit einer virtuellen Funktion wird als Polymorph bezeichnet.

7.2.2 Virtual beim Dtor

Ist eine Methode als virtual deklariert, so muss der Dtor **ZWINGEND** auch als virtual deklariert werden.

7.2.3 Override

Soll eine Methode eine geerbte Methode ersetzen so muss diese mit "Override" gekennzeichnet werden. Solch eine Methode ist automatisch auch virtual. Der Compiler kann so überprüfen, ob eine virtual Methode vorhanden ist.

7.2.4 Final

Final kann zum einen verbieten, dass eine Methode einer Klasse überschrieben wird (Logischerweise geht das nur bei virtuellen Methoden). Zum anderen kann sie auch das weitere Erben einer Klasse verbieten.

7.2.5 Scopeoperator

Per Scopeoperator kann (::) eine Methode aus einer Oberklasse gezielt aufgerufen werden.

Scope Operator und Virtual

```
class Subexample : public Uppclass{
    virtual ~Subexample(); // Correct Dtor
    virtual void newMethod(); //gets overriden when inherited
    void oldVirtualMethod();//still Virtual
    void getsOverridden() override;// overrides method in uppclass
    virtual void iAmPerfect() final;//cant be overridden
};
void Subexample::newMethod(){
    Uppclass::amethodfromUpclass();
    //calls a class from a class higher in the chain
}
```

7.2.6 RTTI / Typing / Binding

Wird kein Virtual verwendet, nennt man das static binding. Mit virtual wird es Dynamic binding genannt, da erst bei Laufzeit bekannt ist um welchen Pointer es sich handelt. Um bei Laufzeit herauszufinden um welchen Typ es sich handelt kann die Run-time Type Information verwendet werden. Mit dem "typeid" operator kann zur Laufzeit überprüft werden um welchen Typ es sich handelt. typeid gibt ein std::type_info zurück.

```
#include <typeinfo>
int main() {
    int a{0};
    float b{0.0f};
    if (typeid(a) != typeid(b)) {// is true
    }// because they have a different type
}
```

7.3 Abstrakte Klassen

Wenn Klassen zwar festlegt, dass eine Methode da ist, diese aber noch nicht implementieren, nennt man das eine abstrakte Methode. Man spricht dann von einer abstrakten Klasse. Eine abstrakte Methode wird in UML ***kursiv*** dargestellt. Es ist egal, ob die abstrakten Methoden selbst definiert sind oder geerbt. Abstrakte Klassen kann man nicht instanziieren. Es gibt kein spezielles Schlüsselwort dafür, man weist der Methode bei Definition 0 zu:

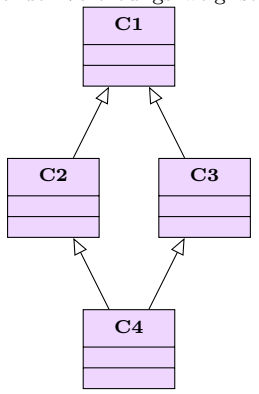
```
class ModernArt{// abstrakte klasse (wegen Methode)
    virtual void aMethod() = 0;//abstrakte methode
};
```

7.4 Organisation der H files

Generell gilt immer noch: Jedes H File hat ein Cpp File. Allerdings können Unterklassen in ein H File zusammengefasst werden, falls diese nicht allzu Umfänglich sind. Ansonsten sollt ein neues H File erstellt werden. Für abstrakte Klassen fällt die Implementierung in einem Cpp file weg.

7.5 Mehrfachvererbung

Folgender Vererbungszweig ist möglich:



Solche Gebilde sind zwar möglich, aber zu verhindern da schnell sehr kompliziert und kaum lesbar. Es entsteht schnell Mehrdeutigkeit, durch die verschiedenen Erbzweige. Bei der Definition einer Klasse kann eine zweite Oberklasse diese mit einem Komma getrennt angegeben werden.

```
class Unterklasse : public Oberklasse1, public Oberklasse2{};
```

8 Memorymanagement

Es gibt verschiedene Gründe warum, nicht immer gleich viel Speicher verwendet wird z.B. da nicht unbegrenzt vorhanden ist. Speicher wird nur dann verwendet wenn er wirklich gebraucht wird. Das Memorymanagement muss aktiv beachtet werden. Es wird zwischen automatischen und manuellen Speichermanagement unterschieden.

8.1 Automatisch

Automatisches Speichermanagement wird anhand der Lebenszeit einer Variable festgestellt und während dem Programmieren bereits festgelegt. Der Speicher liegt auf dem sogenannten **Stapel / Stack**. Wird eine Funktion aufgerufen werden die benötigten Variablen auf dem Stack definiert. Ebenso wird eine Return Adresse zur Funktion, die die Funktion aufgerufen hat abgelegt. Wie die Daten angeordnet werden, ist Architektur-abhängig. Wird die Funktion wieder verlassen, werden die Variablen vom Stack wieder entfernt und zur aufrufenden Funktion zurückgesprungen.

8.2 Manuell/dynamisch

Man kann auch manuell Speicher anfordern. Zum Beispiel, falls zur Compilezeit nicht bekannt ist wie viel Speicher gebraucht wird. Dieser Speicher ist auf dem sogenannten **Haufen / Heap**.
Der Heap ist ein Speicherbereich der vom Betriebssystem bereitgestellt und verwaltet wird.
Zur Laufzeit wird Speicherdynamisch angefordert. Dieser bleibt so lange belegt bis dieser Manuell wieder freigegeben wird.
Es kann aber auch sein, falls der Speicher stark fragmentiert ist, das kein Speicher bereitgestellt werden kann. Generell ist bei sehr vollem Speicher das Arbeiten erschwert.

8.2.1 Speicherlecks

Falls ein Programm unkontrolliert Speicher aufnimmt oder den ihm zur Verfügung gestellte nicht wieder freigibt, spricht man von einem Speicherleck. Diese sind zu verhindern da diese zu einer Systemüberlastung führen und Abstürzen führen.

8.3 Speichermanagement funktionen

Es gibt einige Funktionen für Speichermanagement. In c und C++ sind diese leicht verschieden. Generell geben diese einen Pointer zurück, welcher auf freien Speicher zeigt. **Diese müssen immer auf einen nullpointer geprüft werden da keine garantie vorhanden ist das überhaupt speicher vorhanden ist!** Generell sollte mit solchen Pointer immer mistrauisch gearbeitet werden und nach ambschluss immer wider freigegeben werden.

8.3.1 C: malloc( ) calloc( ), free( ) (Funktionen)

Malloc() gibt einen Voidpointer(pointercasting nicht vergessen) zurück welche auf eine angeforderte Grösse an Bytes Speicher zeigt. Der Speicher ist **nicht** initialisiert Calloc() macht dasselbe, initialisiert aber den Speicher auf 0.
free() gibt den Speicherbereich wieder frei.

```
#include <stdio.h>
#include <stdlib.h>

void *malloc(size_t size);
//Alokiert Speicherbereich size in bytes
void *calloc(size_t size);
//Alokiert Speicherbereich size in bytes und setzt diesen 0
free(void *ptr);
//Gibt speicherbereich auf welcher ptr zeigt wieder frei

int main(){
    int* iPtr = NULL;
    iPtr = (int*)malloc(3*sizeof(int));
    //Speicher fuer 3 Ints reservieren
    //Insert Nullpointer check here
    free(iPtr);
}   //Speicher wider freigegen
```

Falls kein Speicher allokiert werden kann, wird ein NULL-pointer zurückgegeben.

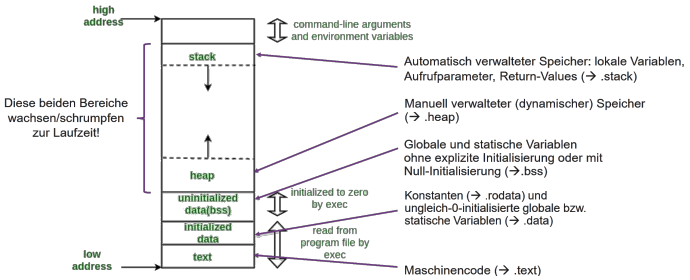
8.3.2 C++: new, delete, new[ ], delete[ ] (Operatoren)

Der New Operator erstellt ein Pointer welcher auf die mitgegebene Grösse zeigt. new[ ] macht dasselbe, ausser das dieser ein Array zurückgibt.
Delete([ ]) gibt den Speicher wieder frei. Dieser kann auch auf den Nullpointer ausgeführt werden. Dieser ist “Delete” egal.

```
int main(){
    int* intPtr = new int;
    //Allokiert fuer den intPtr ein speicherbereich eines Ints
    int* intArrayPtr = new int[10];
    //Allokiert fuer den intPtr ein speicherbereich eines Int
     Array der groesse 10
    if(intPtr == nullptr) return-1;//Nullpointercheck
    delete intPtr;
    //Gibt intPtr wieder frei
    delete[] intArrayPtr;
    //Gibt intArrayPtr wieder frei
}
```

8.4 Speicheraufbau

Der Speicheraufbau bildlich dargestellt sieht ungefähr so aus:



Je nach Architektur kann es auch sein das die hohen und tiefen Adressen anders herum sind.

8.5 Referenzen

Referenzen sind die modernere / eingeschränkte Version von Pointer. Wenn möglich sollte immer mit Referenzen gearbeitet werden, dies ist aber nicht immer möglich.

Referenzen...

- wirken wie ein Alias-Name einer Variablen
- werden wie normale Variablen verwendet
- sind niemals uninitialisiert
- haben niemals den Wert nullptr
- brauchen nicht immer speicher(Implementation abhängig)

Generell helfen Referenzen weniger Fehler in der Programmierung zu machen und weniger Risiken zu haben. Pointer werden allerdings weiter für sehr hardwarenahe Programmierung gebraucht. Sizeof() liefert die Grösse des Typs, auf der die Referenz zeigt.
Beispiel:

```
int i = 42;

//myref ist eine Referenz auf i:
int& myref = i;

//kann auch bei einem Funktionsaufruf verwendet werden:
void myfunc(const DaClass& in1,int& in2);
//hier werden in1& 2 als referenz uebergeben. "DaClass" ist hier
     auch Schreibgeschuetzt(wie ein constpointer)
```

8.6 Speicherplatzbedarf von Objekten

Eine Unterklasse braucht immer mindestens so viel speicher einer Oberklasse.

8.6.1 ZUweisungen

Zuweisungen von Oberklasse zu Unterklasse sind in der regel möglich da alle geerbten Attribute ja vorhanden sind. Umgekehrt geht das allerdings nicht, da der Speicher dafür nicht initialisiert ist.

9 Makefiles

Makefiles sollten generell das umsetzten des Codes in Maschinencode vereinfachen. Es ist eine “Skriptingsprache” welche grob nach dem muster “Erzeugnis/target” : “Abhängigkeiten/Dependency” folgt. Dann folgt indentiert der Befehl, um dieses Erzeugnis zu erzeugen.
Wenn ein Erzeugnis keine Datei zurückgibt, muss dieser als “.PHONY” markiert werden. Das verhindert, dass eine Datei mit demselben Namen das Ausführen verhindert.
Am besten sieht man das an einem Beispiel:

```
all : stuff # erzegt eine definition fuer "make all"

project : main.o lib.o # Projekt linken
  clang++ -Wall -o vector Vector3D.o main.o

lib.o : lib.cpp # lib.cpp compilen
  clang++ -Wall -c lib.cpp

main.o : main.cpp # main.cpp compilen
  clang++ -Wall -c main.cpp

clean : # Projekt cleanup
  rm -f project.exe lib.o main.o

.PHONY : clean all   # Mariert die Targets "clean" und "all" als "
     PHONY" -> Lesbarkeit
```

Der Befehl “make all” baut nun das Projekt.
Sind die o Dateien noch aktuell werden diese nicht erneut kompiliert.
Das Projektverzeichnis kann einfach durch “make clean” aufgeräumt werden.

9.1 Platzhalter

In einem Makefile können auch Platzhalter verwendet werden:

- **$ @** Dateiname des Targets
- **$ <** Dateiname der ersten Dependency des aktuellen Targets
- **$ ∧** Dateiname aller Dependencies des aktuellen Targets durch Leerzeichen getrennt

Mit diesen Mitteln kann ein Professionelleres Makefile erstellt werden:

```
CXX = clang++           # Verwendeter Compiler
CXXFLAGS = -Wall -c     # compilerflags
LDFLAGS = -o            # loader flags
BIN = project       # binary output
OBJS = drink.o drinktest.o  # objectfiles

all: $(BIN)

%.o: %.cpp
  $(CXX) $(CXXFLAGS) $<

$(BIN): $(OBJS)
  $(CXX) $(LDFLAGS) $@ $^

clean:
  rm -f $(OBJS) $(BIN)

.PHONY: clean all
```

Diese kann relativ universell verwendet werden, um kleinere Projekte zu übersetzen.

10 Emotional support meme