**C++ Basics**

1. Heap und Stack

Heap und Stack sind Teile des Arbeitsspeichers. Auf den Stack können immer nur neue Daten obendrauf gelegt werden. Werden Daten wieder freigegeben, werden sie von oben nach unten wieder entfernt. Dieses Prinzip nennt sich LIFO („Last in, First out“). Durch seine Struktur kann der Stack sehr effizient verwaltet werden, weshalb Stack-Operationen auch sehr schnell sind.

Jeder Thread eines Programmes erhält einen eigenen Speicherbereich mit fixer Größe für den Stack zugewiesen. Dort werden Informationen zum Programmablauf (z.B. Funktionsparameter) und lokale Variablen gespeichert. Beim Anlegen neuer lokaler Variablen (z.B. in einer Funktion) wächst der Stack und beim Verlassen des Sichtbarkeitsbereichs (z.B. beim Verlassen einer Funktion) schrumpft (Speicher wird freigegeben) der Stack entsprechend und der Speicher wird automatisch aufgeräumt.



Variable wird auf dem Stack angelegt.

Die Eigenschaften des Stacks sind also:

* Begrenzte Größe
* LIFO Datenstruktur (zuletzt angelegte Daten werden als erstes wieder freigegeben, deshalb auch „Stack“ bzw. „Stapel)
* Wächst und schrumpft mit dem Programmverlauf
* Wird verwendet für lokale Variablen und Funktionsparameter
* Kein explizites Freigeben des Speichers nötig
* Das Ablegen und Entfernen von Elementen ist sehr effizient

Der Heap ist nicht so strukturiert wie der Stack. Anders als der Stack kann der Heap bis zur Speichergrenze auf der Prozessebene anwachsen. Der Heap ist außerdem auch langsamer als der Stack. Auf dem Heap angelegter Speicher muss auch explizit freigegeben werden (durch den Programmiere oder z.B. den Garbage Collector, variiert nach Programmiersprache). Für den Zugriff auf den Heap werden Zeiger verwendet, auch wenn das nicht immer direkt ersichtlich ist. In Java und C# werden z.B. Referenzen für Objekte, die auf dem Heap liegen, verwendet. Intern wird jedoch immer noch mit Zeigern, welche die Speicheradresse des Objektes beinhalten, gearbeitet. Auf Objekte, welche auf dem Heap angelegt wurden, lässt sich global zugreifen, da sie nicht auf den lokalen Sichtbarkeitsbereich beschränkt sind (sofern ein Zeiger oder eine Referenz auf dieses Objekt vorhanden ist). In C++ wird dynamischer Speicher auf dem Heap mit „new“ erzeugt. Danach muss dieser dynamische Speicher jedoch explizit mit „delete“ freigegeben werden. Das Zurücksetzen des Pointers beugt hier einem Zugriff auf bereits (durch delete) freigegebenen Speicher vor, denn „delete“ setzt den Zeiger nicht auf NULL zurück.



Variable mit ‚new‘ auf dem Heap erstellen. Wert wird über den Pointer geändert, der auf diese Variable zeigt. Am Ende wird dieser dynamische Speicher mit ‚delete‘ wieder freigegeben.

Die Eigenschaften des Heaps sind also:

* Der Heap kann innerhalb der Prozessgrenze beliebig groß werden
* Anlegen und freigeben von Objekten ist vergleichsweise langsam
* Auf dem Heap angelegte Objekte können global verfügbar gemacht werden
* In Programmiersprachen ohne Garbage Collector muss der Speicher manuell freigegeben werden, wenn er nicht mehr benötigt wird

Quellen: <https://lerneprogrammieren.com/stack-und-heap/>

2. Output/ Input

Text auf der Konsole mit ‚cout‘ ausgeben.



Mit ‚cin‘ wird der Input des Users eingelesen.



Hier wird der User Input in die Variable ‚i‘ geschrieben und dann auf der Konsole ausgegeben.

3. Const und unsigned



Eine Konstante wird mit ‚const‘ erstellt. Konstanten werden gebraucht, wenn die Variable im Programm nicht geändert wird (im Beispiel PI, da PI einen festen Wert hat und nicht geändert werden muss).

„Unsigned“-Variablen verzichten auf den negativen Bereich, da bei „normalen“(signed) Variablen die Hälfte des Speicherplatzes für den negativen Bereich draufgeht. Der Wertebereich eines signed 32-bit Integers reicht von -2,147,483,648 bis 2,147,483,647, bei einem unsigned 32-bit Integers reicht dieser Wertebereich jedoch von 0 bis 4,294,967,295. In dem Code wird für ‚k‘ das Maximum des unsigned 32-bit Integers ausgegeben, da -1 außerhalb der Reichweite eines unsigned 32-bit Integers liegt, weshalb er auf das Maximum des unsigned 32-bit Integers springt.

In C++ lassen sich diese Grenzen auch mit schon in C++ existierenden Konstanten abfragen, weshalb Code wie dieser hier

auch ‚true‘ bzw. 1 zurückgibt.

Quellen: <https://docs.microsoft.com/de-de/cpp/cpp/data-type-ranges?view=msvc-170>

4. Variablen



In diesem Beispiel kann man sehen, wie eine globale Variable(x) mit 5 initialisiert wird und an zwei verschieden Stellen um 1 inkrementiert wird. Nach dem die globale Variable an beiden Stellen inkrementiert wurde ist x = 7.



Hier wird ein Alias für eine Variable (x) erstellt. Der Alias wird mit einem ‚&‘ erstellt. Jetzt kann man über y und x auf ein und dieselbe Stelle im Arbeitsspeicher zugreifen.



Mit dem ‚static‘ Keyword können lokale Variable erstellt werden die, nachdem der Block, in dem die Variable deklariert wurde, den Gültigkeitsbereich verlässt immer noch im Speicher vorhanden ist.

Deswegen ist der Ouput des Beispielscodes auch:





Möchte man Zufallszahlen in C++ generieren benötigt man einen sogenannten ‚Seed‘ wofür meistens die aktuelle Systemzeit (‚time(0)‘) genutzt wird. Mit dem Aufruf von ‚rand()‘ wird dann eine Zufallszahl generiert. Mit dem Modulo Operator kann man den Bereich der generierten Zufallszahl auch noch beschränken.

‚C-Style‘ Cast:



Der ‚C-Style‘ Cast ist ein Cast, der in C++ und C funktioniert, in C++ jedoch nicht empfohlen wird, da es bessere Alternativen (wie ‚static\_cast‘ gibt) gibt.

Static Cast:



Der ‚static\_cast‘ wandelt Datentypen ineinander um. Dabei steht in den spitzen Klammern ‚<>‘ immer in was das Quellobjekt, welches in den Klammern ‚()‘ steht, gecastet werden soll.

Const Cast:



Mit ‚const\_cast‘ kann die Konstantheit eines Objektes aufgehoben werden, wobei das Objekt [trotz der aufgehobenen Konstantheit] nicht verändert werden sollte.

Aufgebaut ist der ‚const\_cast‘ wie folgt: In den ‚<>‘ steht, zu was man casten will und in den den ‚()‘ steht, was gecastet werden soll.

Im Beispiel erwartet die Funktion einen Zeiger auf einen nicht-konstanten String, wobei der zu übergebene String jedoch konstant ist und in der Funktion auch nicht geändert wird. Daher kann man beim Funktionsaufruf mit ‚const\_cast‘ die Konstanz des konstanten Zeigers entfernen, um ihn der Funktion übergeben zu können. (Der auskommentierte Funktionsaufruf würde nicht funktionieren, da dort versucht wird, einen konstanten Zeiger zu übergeben, wobei die Funktion einen nicht-konstanten Zeiger erwartet)

Quellen: <https://de.wikibooks.org/wiki/C%2B%2B-Programmierung/_N%C3%BCtzliches/_Casts>

5. Funktionen



In dem Beispiel kann man den Aufbau einer Funktion, sowie die Überladung der ‚print‘-Funktion sehen.

In C++ sind Funktion wie folgt aufgebaut: Zuerst folgt der Rückgabewert (‚void‘, falls nichts zurückgegeben werden soll), dann der Name der Funktion und in den Klammern optionale Parameter, die beim Funktionsaufruf als Argumente übergeben werden müssen.

Da die ‚print‘-Funktion überladen ist, kann man einen int oder double als Argument übergeben.

Die ‚add‘-Funktion nimmt zwei Integer als Argument und gibt das Ergebnis der Addition beider Integer zurück.



Sollten die Funktionen nach der ‚main‘-Funktion deklariert werden benötigt man ‚Funktionsprototypen‘, um die Funktionen nutzen zu können. Diese Funktionsprototypen geben den Rückgabewert, den Funktionsnamen und die Parametertypen der Funktion an.



oder



Standardparameter in C++ werden dem Argument zugewiesen sollte beim Funktionsaufruf kein spezieller Wert für das Argument übergeben worden sein. Dies lässt sich direkt in der Deklaration der Funktion oder in dem Funktionsprototypen machen.

6. Referenzen

„Problem“:



Der Ziel des Codes sollte es sein den Wert einer Variable in einer Funktion zu ändern, jedoch wird in dem Beispiel oben nur der Wert der Variable („call by value“) übergeben, dann in ‚a‘ gespeichert und dann wird a inkrementiert, jedoch nicht x, die Variable, die wir eigentlich inkrementieren wollten.

„Lösung“:



Dieser Code ist die Lösung für das oben präsentierte Problem. Hier steht ein ‚&‘ vor dem Parameter, was bedeutet, dass man eine Referenz erwartet („call by reference“). Über die Referenz lässt sich x innerhalb der Funktion ‚funktion‘ ändern, da keine Kopie des Wertes des Argumentes erstellt wird, sondern der Wert des Argumentes direkt über die Referenz geändert wird.

Hier eine Visualisierung des Problems/der Lösung:



In dem Diagramm kann man sehen, dass beim „Call/pass by reference“ die Speicheradresse des Argumentes übergeben wird, anstatt nur der Wert.

Bild: <https://www.programiz.com/cpp-programming/pointers-function>

7. Headerdateien

Headerdateien enthalten die Deklaration (und manchmal auch schon die Implementation bzw. Initialisierung) von z.B. Funktionen oder Variablen. Headerdateien enden auf .h und werden mit ‚#inlcude“Headerdatei“‘ eingebunden.



Verwendet wird die Headerdatei im C++ Code wie folgt:



Möchte man in einer Headerdatei nun Variablen von einer Datei verwenden, in der man die Headerdatei nutzt, lässt sich dies mit dem Keyword ‚extern‘ machen. Hier ein Beispiel einer solchen Headerdatei:



Die Funktion aus der Headerdatei kann in der ‚main.cpp‘ wie folgt verwendet werden:



Hier wird eine globale Variable ‚z‘ initialisiert. Der Aufruf von ‚funktion‘ gibt ‚5‘ aus, weil die Headerdatei mit dem ‚extern‘ Keyword erkennt, dass es eine Variable ‚z‘ in dem Programm gibt, welches die Headerdatei verwendet.

8. Schleifen



In diesem Beispiel Code kann man die drei Schleifenarten ‚while‘, ‚for‘ und ‚do-while‘ sehen. Sie sind genau so aufgebaut wie in Java.

9. Arrays und Strings



In diesem Beispielcode wird ein Array von der Größe von 100 Integern auf dem Stack erstellt und zuerst nur mit 0 initialisiert, da in den geschweiften Klammern keine spezifische Werte angegeben wurden. Später in der ‚for‘-Schleife wird auf eine spezielle Stelle des Arrays über den Index ‚i‘ zugegriffen und dieser Stelle wird der Wert von ‚i‘ zugewiesen und dann auf der Konsole ausgegeben.



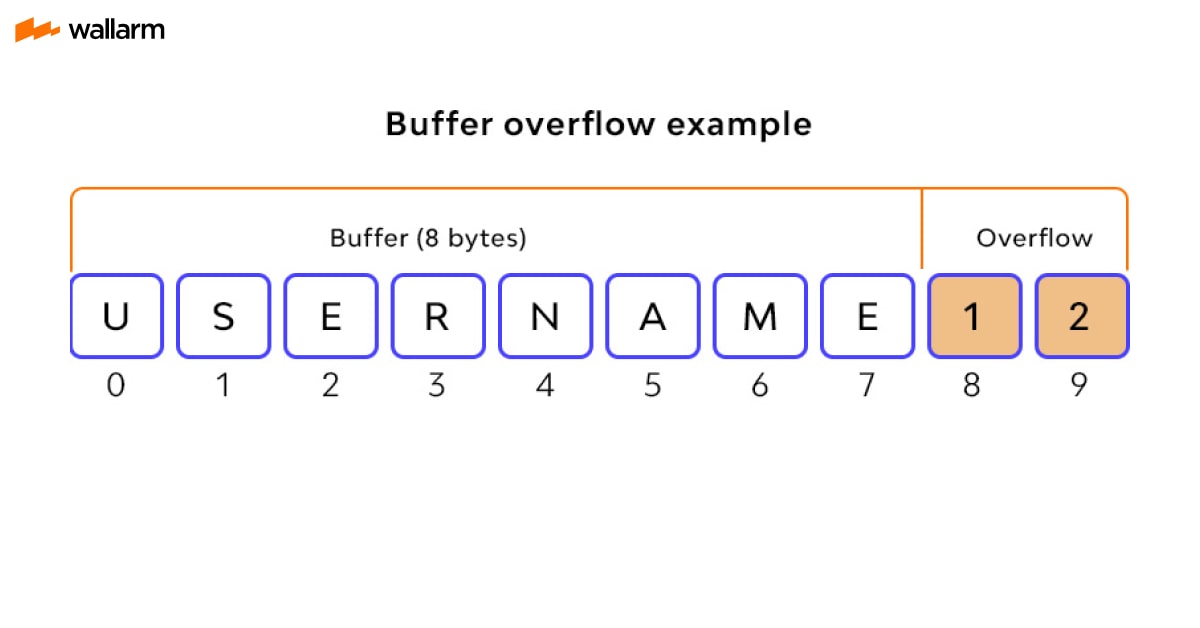
Wichtig: Da der Array mit einer festen Größe auf dem Stack erstellt wurde, darf man nicht versuchen auf einen Index außerhalb der festgelegten Größe des Arrays zugreifen, da dies eine zufällige Stelle im Arbeitsspeicher ausgeben/überschreiben könnte. Daher ist auch der Output des Codes:

661160040 // zahlen[150] zeigt auf irgendeinen Wert im Arbeitsspeicher

100 // Mit zahlen[150] = 100 kann diesen Wert überschreiben, falls dieser Wert gebraucht wird kann das im schlimmsten Fall zum Stopp des Programms führen

#include <iostream>  
#include <cstring>  
  
int main() {  
  
 char string[] = "Hello World!"; // 0 Byte am Ende -> 0 Byte != Zeichen "0"  
 char string2[128] = "Test";  
  
 int cmp = strcmp(string, string2);  
  
 int length = strlen(string); // 0 Byte wird benötigt, damit diese Funktion weiß, wann der String endet  
  
 std::cout << "Laenge von " << string << ": " << length << std::endl;  
 std::cout << "String compare von " << string << " und " << string2 << ": " << cmp << std::endl; // 0 = gleich, sonst < 0 oder > 0  
 std::cout << "Zusammenfuegen von " << string << " und " << string2 << ": " << strcat(string, string2) << std::endl;  
 strcpy(string2, string);   
 std::cout << "Inhalt von String 2 (wurde aus String kopiert): " << string2 << std::endl;  
  
 system("pause");  
  
 return 0;  
}

Hier kann man C-Strings sehen, welche in C++ jedoch nicht empfohlen werden, da es eine bessere Alternative gibt. Ein C-String ist ein Array aus verschiedenen Zeichen, wobei der Compiler am Ende der Zeichenkette noch ein ‚0 Byte‘ hinzufügt, damit er weiß, wann das Ender der Zeichenkette ist. Das gefährliche bei C-Strings ist, dass sie (da sie Arrays sind) eine feste ‚Größe‘ bzw. ‚Länge‘ haben und diese nicht ‚überschritten‘ werden darf, wobei sowas aber bei Funktionen wie z.B. ‚strcpy‘ zur Laufzeit passieren kann, da der Source String länger als der Destination String sein kann und somit einen ‚Buffer Overflow‘ auslösen kann. Hier ist eine Visualisierung eines Buffer Overflows:

****

Hier kann man sehr gut sehen, was ein Buffer Overflow ist. Man hat einen Buffer (z.B. den C-String) und versucht in diesen etwas hereinzuschreiben (wie z.B. ‚USERNAME12‘), jedoch ist der Buffer (bzw. der C-String) nicht groß genug (z.B. ist er nur 8 anstatt 10 Bytes groß) und so ‚fließt‘ er aus dem Buffer (bzw. dem C-String).

#include <iostream>  
#include <string>  
  
int main() {  
  
 std::string string = "Hello World!";  
 std::string string2 = string; // Kopie  
 std::string string3("Test"); // int zahlen(40);  
 std::string string4(10, 'a'); // 50mal a

std::string userInput;  
 std::string empty; // String wird schon hier erstellt/ Muss nicht wie in Java noch mit 'new' instanziiert werden

int length = string.length();  
  
 std::cout << "Laenge von " << string << ": " << length << std::endl << string2 << std::endl << string3 << std::endl << string4 << std::endl;  
  
 std::string a = "Hello ";  
 std::string b = "World!";  
 std::string ab = a + b;  
  
 std::cout << ab << std::endl;  
  
 std::string zusammenfuegenVonPrimitivenDatentypen = std::to\_string(10) + std::to\_string(10.5); // Primitive Datentypen müssen zuerst in einen String gewandelt werden  
  
 std::cout << zusammenfuegenVonPrimitivenDatentypen << std::endl;  
  
 if(a == b){ // < und > funktionieren auch; Außnahme: Ä, Ö, Ü  
 std::cout << "A und B sind gleich!" << std::endl;  
 }else{  
 std::cout << "A und B sind nicht gleich!" << std::endl;  
 }  
  
 std::string original = "World!";  
 std::string anfang = "Hello ";  
 original.insert(original.length(), anfang);  
  
 std::cout << original << std::endl;  
  
 original.erase(6, 6);  
 std::cout << original << std::endl;  
  
 std::string satz = "Lorem ipsum Lorem ipsum Lorem ipsum Lorem ipsum";  
 int first = satz.find("Lorem"); // Von links suchen; Wenn nichts gefunden wurde -> -1  
 int last = satz.rfind("Lorem"); // Von rechts suchen  
 std::cout << "Index erstes Lorem: " << first << " Index letztes Lorem: " << last<< std::endl;  
  
 string[4] = 'Z'; // Gefährlich, Index kann über Länge des String reichen und Sachen im Arbeitsspeicher überschreiben  
 std::cout << "Manipulierter String: " << string << std::endl;  
  
 // string.at(400) = 'Z'; // Bessere Alternative -> Zu hoher Index = Programm stürzt ab

std::getline(std::cin, userInput);  
 std::cout << userInput << std::endl;

system("pause");  
  
 return 0;  
}

Dazu gibt es jedoch noch eine bessere Variante Strings zu erstellen und das sind die C++ Strings (std::string). Im Code oben kann man einige Methoden, die mit dem C++ String verwendet werden können, sehen. Der C++ String ist relativ gleich zu dem String in Java, dennoch gibt es auch hier (wie bei den C-Strings) eine kleine ‚Schwachstelle‘ und das wäre das Zugreifen mit ‚[]‘ auf einen zu hohen Index, was dann dafür sorgt, dass ein zufälliger Wert im Arbeitsspeicher überschrieben wird. Besser ist es hier die ‚.at(index)‘-Methode zu verwenden, da diese zur Laufzeit einen Fehler schmeißt, falls der Index ‚out of range‘ ist, was besser ist als das Programm zur Laufzeit zu ‚zerstören‘. Am Ende fällt vielleicht auf, dass anstatt ‚cin‘ ‚getline‘ für User Input benutzt wird, was daran liegt, dass ‚cin‘ einen Abstand als Terminator sieht und man so keine Wörter/Sätze mit Abständen eingeben kann. Mit der ‚getline‘ Funktion geht dies jedoch und der User Input lässt sich auch direkt in einen String/eine Variable schreiben.

10. Weiteres

Hier kann man ein Switch-Case sehen, was genau wie in Java funktioniert (case, default, break, es gibt Fallthroughs).

Hinter dem ‚case‘ steht die Bedingung, man benötigt immer ein ‚break‘ (außer für ein Fallthrough) und falls keiner der Fälle eintritt kann man dies auch mit ‚default‘ abfangen.



Für mathematische Funktionen kann man die C- und C++- Mathebibliotheken verwenden.

Hier kann man 2 Headerdateien sehen, die sich gegensteig inkludieren. Doch, damit das funktioniert werden ‚Includeguards‘ benötigt, welche darauf achten, dass sich die Headerdateien nicht unendlich oft inkludieren, sondern nur einmal. Dies machen sie mit Makros und bedingter Kompilierung.

‚#pragma once‘ ist eine weitere Art eine Headerdatei in einer anderen Headerdatei, die diese Headerdatei inkludiert, zu inkludieren. Jedoch ist ‚#pragma once‘ kein Standard und deshalb muss es nicht jeder Compiler haben. Dafür brauch man in der anderen Headerdatei nur die Headerdatei mit ‚#pragma once‘ inkludieren.



Mit Typedefs kann man verschiedenen Datentypen in C++ Aliasnamen geben. Das ist nützlich, wenn man (wie hier im Beispiel) eine längere Deklaration des Datentypen hat (im Beispiel ‚unsigned‘) und diese nicht immer ausschreiben möchte. Nachdem man einer Deklaration einen Aliasnamen gegeben hat, kann man diesen Aliasnamen benutzen, anstatt die komplette Deklaration immer ausschreiben zu müssen.





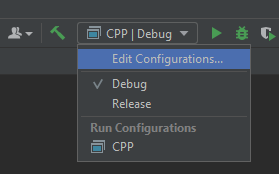
Hier wird ein Typedef verwendet um sich die ‚ständige‘ Deklaration von einem unsigned int zu sparen.

#include <iostream>  
#include <chrono>  
  
bool isPrime(int number){  
  
 if(number < 2) return false;  
 if(number == 2) return true;  
 if(number % 2 == 0) return false;  
 for(int i=3; (i\*i)<=number; i+=2){  
 if(number % i == 0 ) return false;  
 }  
 return true;  
  
}  
  
int main() {  
  
 std::chrono::high\_resolution\_clock::time\_point start = std::chrono::high\_resolution\_clock::now();  
  
 int sum = 0;  
 for(int i = 2; i < 100000; i++){  
 if(isPrime(i)){  
 sum += i;  
 }  
 }  
  
 std::chrono::high\_resolution\_clock::time\_point ende = std::chrono::high\_resolution\_clock::now();  
  
 std::chrono::high\_resolution\_clock::duration diff = ende - start;  
  
 int ms = std::chrono::duration\_cast<std::chrono::milliseconds>(diff).count();  
  
 std::cout << sum << " Dauer: " << ms << " Millisekunden" << std::endl;  
  
 system("pause");  
  
 return 0;  
}

Zeitmessen mit ‚std::chrono‘.

Verschönern kann man den Code auch mit dem ‚auto‘-Keyword, durch welches der Compiler selbst den Rückgabetyp herausfinden muss, damit der Programmierer ihn nicht angeben muss.

#include <iostream>  
#include <chrono>  
  
bool isPrime(int number){  
  
 if(number < 2) return false;  
 if(number == 2) return true;  
 if(number % 2 == 0) return false;  
 for(int i=3; (i\*i)<=number; i+=2){  
 if(number % i == 0 ) return false;  
 }  
 return true;  
  
}  
  
int main() {  
  
 auto start = std::chrono::high\_resolution\_clock::now();  
  
 int sum = 0;  
 for(int i = 2; i < 100000; i++){  
 if(isPrime(i)){  
 sum += i;  
 }  
 }  
  
 auto ende = std::chrono::high\_resolution\_clock::now();  
  
 auto diff = ende - start;  
  
 int ms = std::chrono::duration\_cast<std::chrono::milliseconds>(diff).count();  
  
 std::cout << sum << " Dauer: " << ms << " Millisekunden" << std::endl;  
  
 system("pause");  
  
 return 0;  
}

Code kann in C++ in 2 verschiedenen Modi compiliert werden. Einmal in ‚Debug‘ und einmal ‚Release‘. Der ‚Debug‘ Modus ermöglicht es einem den Code [durch Debug Symbols] zu debuggen und hat keine Optimierungen, anders als der ‚Release‘ Modus, bei welchem der Compiler Optimierungen vornimmt.

#include <iostream>  
#include <fstream>  
  
int main(int argc, char \*argv[]) {  
  
 std::ofstream log(R"(D:\Windows\Desktop\Workspaces\CppWorkspace\CPP\log.txt)", std::ios::app);  
  
 std::cerr.rdbuf(log.rdbuf());  
  
 std::cerr << "Ein Fehler ist aufgetreten!" << std::endl;  
  
 system("pause");  
  
 return 0;  
}

‚cerr‘ in eine Log Datei ‚umleiten‘.

11. Makros und der Präprozessor

Um Makros zu verstehen ist wichtig zu wissen, was im Hintergrund passiert, wenn man das Programm ausführt:

Zuerst kompiliert der Compiler den C++ Code in für Maschinen lesbaren Code. Dieser, für Maschinen lesbare Code, steht dann in OBJ-Dateien und muss noch zusammengefügt werden, worum sich der Linker kümmert. Der Linker fügt dann alles so zusammen, dass es am Ende eine .exe Datei gibt. Vor dem Kompilieren geht der Präprozessor aber nochmal über den Code und verändert ihn, meistens indem er Anweisungen kopiert und an anderer Stelle wieder einfügt. Präprozessor Befehle erkennt man daran, dass sie mit einem Rauten Symbol ‚#‘ anfangen.

Ein Code der also so aussieht, wird vor dem Kompilieren von dem Präprozessor zu folgendem Code geändert:

Makros können außerdem schon das Semikolon enthalten und somit wird dies nicht beim Makroaufruf benötigt. Durch Makros kann man lange Statements, die man öfters benötigt, auch sehr gut abkürzen. Somit kann man sich z.B. das Ausgeben auf der Konsole erleichtern, was öfters bei größeren Projekten wie einer Game Engine wichtig ist, da Logging dort eine große Rolle spielt:



Außerdem kann man mit ‚\‘ einen Zeilenumbruch ‚erzeugen‘ um einen mehrzeiligen Makro zu erzeugen

#include <iostream>  
  
#define MAIN int main() \  
{\  
 std::cout << "Hello World" << std::endl;\  
}  
  
MAIN

Möchte man ein Makro ‚undefinieren‘ (Definition des Makros wird entfernt) kann man das mit ‚#undef‘ machen, was im Code wie folgt aussieht (Beim 2. Aufruf von der ‚GEHEIMZAHL‘ gibt es einen Fehler, da der Präprozessor die Definition für ‚GEHEIMZAHL‘ nicht mehr kennt und daher nicht weiß womit er ‚GEHEIMZAHL‘ ersetzen soll):

Quellen: <https://www.youtube.com/watch?v=A6IHJVMzulg&ab_channel=Pilzschaf>, <https://www.youtube.com/watch?v=j3mYki1SrKE&ab_channel=TheCherno>,<https://docs.microsoft.com/de-de/cpp/preprocessor/hash-undef-directive-c-cpp?view=msvc-170>

12. Bedingte Kompilierung

Bedingte Kompilierung lässt den Nutzer, je nach Bedingung Quelltextblöcke ein oder ausschließen. Mit #ifdef(,#elif) und #endif kann man Bereiche von Code eingrenzen, die nur kompiliert werden sollen, wenn ein bestimmter Makro vorher definiert wurde, was dann ungefähr so aussieht:



In dem Beispiel wird der ‚WIN‘-Makro mit einer 1 definiert und später frage ich auch den Wert dieses Makros ab um dann den entsprechenden Code auszuführen.

Ein sinnvoller Einsatz wäre:

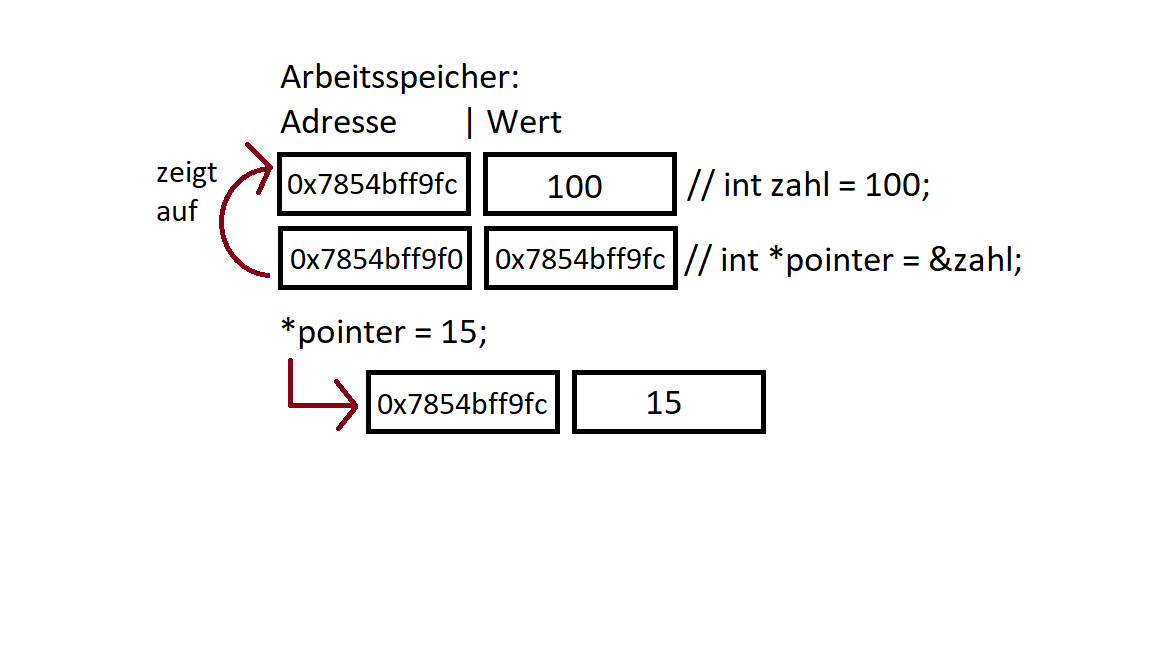


Hier wird ‚Hello World‘ nur ausgegeben, wenn der ‚DEBUG\_MODE‘ definiert ist. Sowas ist nützlich, wenn man ein großes Projekt hat, wo debuggen wichtig ist, was man dann aber auch veröffentlichen möchte, wo dann keine Infos auf der Konsole ausgegeben werden sollen, was man damit erreicht, dass man den Wert von ‚DEBUG\_MODE‘ auf ‚0‘ setzt.



Der folgende Code zeigt eine weitere nützliche Präprozessordirektive, die #error-Anweisung. Sie ist da, um den Entwickler früh vor einer Programminkonsistenz oder einen Verstoß gegen eine Einschränkung zu informieren, was in diesem Fall das Doppelte definieren eines Betriebssystems ist.

13. Pointer



Pointer (bzw. Zeiger) zeigen auf einen Speicherplatz im RAM. An diesem Speicherplatz können Variablen, Objekte etc. liegen. In dem Beispiel wird eine Variable auf dem Stack mit ‚int zahl = 100;‘ erstellt. Der Pointer ‚pointer‘ welcher mit ‚int \*pointer = &zahl;‘ erstellt wird, zeigt dann auf den Speicherort dieser Variable. Mit ‚&‘ wird die Adresse einer Variable ermittelt, was in dem Fall benutzt wird um dem Pointer mitzuteilen an welcher Speicheradresse die Variable liegt. Möchte man nun wissen, worauf der Pointer zeigt (also auf welche Adresse), kann man das einfach machen in dem man ‚pointer‘ ausgibt. Den Wert hinter der Adresse auf den der Pointer zeigt kriegt man mit ‚\*pointer‘ heraus.

Hier ein kleines Beispielprogramm, was gut ist um ein Grundverständnis für die Funktionsweise von Pointern zu entwickeln:



Ein sinnvolles Beispiel für Pointer wäre das Verwalten von dynamischem Speicher:



Da Pointer alles referenzieren können, können sie auch auf weitere Pointer referenzieren, was dann wie folgt aussieht:



(Pointer auf einen Pointer auf einen Pointer)

Quelle: <https://manderc.com/operators/pointerarithmetic/index.php>

13.1. Pointer-Arithmetik

Die Pointer-Arithmetik spezifiziert spezielle Verhaltensregeln bei Additions-, Subtraktions-, Zuweisungs- und Vergleichs-Operatoren. Hier ein Beispiel zur Addition mit einem Pointer:



Hier wird der Anfang des Arrays ‚zahlen‘ in einem Pointer ‚anfang‘ gespeichert. Dieser Pointer wird in einer For-Schleife inkrementiert. Dies zeigt dann (vom Anfang des Arrays) auf die Stelle ‚i‘ des Arrays. Der Stelle ‚i‘ des Arrays ‚zahlen‘ wird dann der Wert von ‚i‘ zugewiesen. Der Output gibt dann die Zahlen von ‚0-99‘ aus. Der Array wurde mit Hilfe von Pointer-Arithmetik gefüllt und letztendlich ausgelesen.

13.2. Null-Pointer

Pointer werden auf null (NULL (Makro für 0) bzw. 0) gesetzt, um zu zeigen, dass sie auf nichts zeigen. Dies lässt sich gut mit ‚0‘ und/oder ‚NULL‘ machen, jedoch ist die neuste und beste Variante einen Pointer auf null zu setzen ‚nullptr‘. Hier ein kleines Beispielprogramm zu Null-Pointer:



Einige Vorteile von ‚nullptr‘ sind:

- Vom Typ ‚std::nullptr\_t‘ -> Kein Integer

- Keyword welches eine Null-Pointer Konstante spezifiziert

- Hat (anders als ‚NULL‘) keine Probleme implizit in 0 (bzw. einen Integer) bei z.B. einem Funktionsaufruf gecastet zu werden.

Quelle: <https://learn.microsoft.com/de-de/cpp/cpp/nullptr?view=msvc-170>

13.3. Casten von Pointern



Hier wird der Wert, auf den der Pointer ‚pA‘ zeigt, zu einem Integer mit einem ‚static\_cast‘ gecastet.

13.4. Void Pointer

Void Pointer haben keinen ihnen zugewiesenen Datentypen. Sie können die Adresse eines Typen halten/ auf irgendeinen Datentypen im RAM zeigen, jedoch können sie diesen Wert nicht ändern. Ein Void Pointer lässt sich in jeden Datentypen casten.



In diesem Programm zeigt ein Void Pointer auf einen Float. Den Wert dieses Floats kann der Void Pointer nicht ändern, jedoch kann man diesen Void Pointer in einen Float Pointer casten und den Wert des referenzierten Floats über den Float Pointer ändern.

13.5. ‚Const‘ bei Pointern

‚const‘ bei Pointern kann dafür sorgen, dass:

- der referenzierte Wert nicht geändert werden darf, die Referenz selbst darf jedoch geändert werden

- der referenzierte Wert geändert werden darf, die Referenz selbst darf jedoch nicht geändert werden

- die Referenz und der referenzierte Wert nicht geändert werden dürfen

13.6. Funktionspointer

Funktionspointer verweisen auf die Adresse einer aufzurufenden Funktion. Um einen Funktionspointer zu erstellen, muss man den Rückgabetypen und die Parameter der Funktion angeben, sowie den Namen des Funktionspointers. Hier ein Beispielprogramm:



14. Dynamischer Speicher

In C++ erstellt man mit dem ‚new‘-Keyword, anders als in Java, dynamischen Speicher. Dieser Speicher wird dann auf dem Heap reserviert. Dynamischen Speicher zu nutzen ist sinnvoll, falls etwas wie ein Array zur Laufzeit (durch z.B. Benutzereingabe) erstellt wird. Dynamischer Speicher wird mit Pointer verwaltet. Außerdem muss Dynamischer Speicher später auch freigegeben (‚gelöscht‘) werden. Hier ein Beispielprogramm:

#include <iostream>  
  
int main() {  
  
 int \*p = new int; // Pointer zeigt auf dynamische Variable im Arbeitsspeicher  
  
 \*p = 10;  
  
 std::cout << \*p << std::endl;  
  
 delete p;  
  
 int k = 0;  
 std::cin >> k;  
  
 int \*array = new int[k]; // Wird nicht zur Compilezeit erstellt, sondern zur Laufzeit  
 array[0] = 15;  
 std::cout << array[0] << std::endl;  
  
 delete[] array;  
  
 system("pause");  
  
 return 0;  
}

In diesem Beispielprogramm wird zuerst ein Pointer auf einen Integer, für den auf dem Heap Speicher reserviert wird, erstellt. Dann wird über diesen Pointer dem dynamischen Speicher auf dem Heap ein Wert (10) zugewiesen. Danach wird dieser Speicher wieder mit ‚delete‘ freigegeben. Das selbe wird dann mit dem Array ‚array‘ gemacht, wobei man für das Freigeben von Arrays kein ‚delete‘ sondern ‚delete[]‘ benutzt.

‚malloc‘ und ‚free‘ sind die C Varianten von ‚new‘ und ‚delete‘.

‚Delete‘ und ‚free‘ sind sich sehr ähnlich, da man bei beiden den freizugebenen Speicher zum Freigeben angibt. Jedoch muss man bei ‚malloc‘ (um dynamischen Speicher zu erstellen) die Größe des zu reservierenden Speichers ‚manuell‘ (mit z.B. ‚sizeof‘) angeben. In diesem Beispielprogramm wird Speicher für 10 Integer reserviert und dann in einen Integer Pointer gecastet, da ‚malloc‘ einen Void Pointer auf den reservierten Speicher zurückgibt:



15. Function Templates

Mit Function Templates (‚Schablonen‘) kann man beliebige Datentypen an eine Funktion übergeben. Dafür muss man dem Compiler erstmal mit ‚template<typename T>‘ (‚T‘ wird am häufigsten für den beliebigen Datentypen genutzt) sagen, dass ‚T‘ hier einen beliebigen Datentypen repräsentiert. Danach kann man seine Funktion ganz normal schreiben, aber anstatt den Datentyp eines Parameters zu spezifizieren kann man ‚T‘ verwenden. So kann jeder Datentyp als Argument an diese Funktion übergeben werden. In der Praxis sieht das dann wie folgt aus:

#include <iostream>  
  
template<typename T> // Platzhalter für einen Datentypen (Compiler setzt später passenden Datentypen ein)  
T min(T a, T b){  
 return a < b ? a : b;  
}  
  
int main() {  
  
 std::cout << min(8, 10) << std::endl;  
  
 std::cout << min(1.53, 7.26) << std::endl;  
  
 system("pause");  
  
 return 0;  
}

Jedoch ist diese Funktion nicht für jeden Datentypen sinnvoll. Wenn man z.B. zwei char-Arrays lexikografisch vergleichen möchte, wird das mit dieser Funktion nicht gehen, da dann der im Speicher weiter vorne liegende char ausgegeben wird. Dafür kann man diese Function Template aber für den ‚char-Fall‘ spezialisieren (ähnlich wie Überladen). Man spezialisiert ein Template wie folgt:



16. Goto

Mit ‚goto‘ kann man überall an jede Stelle in einer Funktion springen. Dafür erstellt man ein Label mit ‚label:‘ an der Stelle, zu der man springen möchte, danach kann man zu diesem Label mit ‚goto label;‘ springen.



‚goto‘ wird nicht empfohlen, weil ‚goto‘ Code sehr unübersichtlich macht. Daher wird ‚goto‘ eher genutzt um aus verschachtelten Schleifen zu kommen.

17. Namespaces

Namespaces deklarieren einen Gültigkeitsbereich für die darin enthaltenen Bezeichner wie Funktionen. Sie sollen Namenskonflikte vermeiden, da manche Funktionen wie z.B. die ‚print‘ Funktion oft gebraucht wird. Auf die Dinge innerhalb eines Namespaces kann man mit einem doppeltem Doppelpunkt zugreifen oder mit ‚using namespace …;‘(nicht empfohlen!).

#include <iostream>  
  
namespace a{  
 char n = 'a';  
  
 void print(){  
 std::cout << "A" << std::endl;  
 }  
}  
  
namespace b{  
 char n = 'b';  
  
 void print(){  
 std::cout << "B" << std::endl;  
 }  
}  
  
int main() {  
  
 std::cout << a::n << std::endl << b::n << std::endl;  
  
 a::print();  
  
 b::print();  
  
 system("pause");  
  
 return 0;  
}

Quelle: <https://learn.microsoft.com/de-de/cpp/cpp/namespaces-cpp?view=msvc-170&viewFallbackFrom=vs-2019>

18. Vector

Vectoren sind in C++ das Gegenstück zu der ArrayList(Arrays die dynamisch anwachsen) in Java.

#include <iostream>  
#include <vector>  
  
int main() {  
  
 std::vector<int> k(10);  
  
 for(int i = 0; i < k.size(); i++){  
 k[i] = i;  
 }  
  
 k.push\_back(10);  
  
 for(int i = 0; i < k.size(); i++){  
 std::cout << k.at(i) << std::endl;  
 }  
  
 system("pause");  
  
 return 0;  
}

In diesem Beispielcode wird ein Vector mit 10 Elementen erstellt. Nach der Initialisierung dieser 10 Elemente wird mit der ‚push\_back‘-Methode ein 11 Element mit dem Wert ‚10‘ an den Vector ‚angehangen‘ -> Der Vector wurde vergrößert bzw. erweitert.

19. Programmargumente

Mit Programmargumenten lassen sich beim Start zusätzlich angegebene Zeichen/Wörter auslesen. Dafür muss man nur zwei Parameter in der main-Methode ergänzen. Der eine Parameter sagt aus, wieviel Argumente übergeben wurde, und der andere gibt an, was für Argumente übergeben wurden.



20. Inline

Mit dem inline-Keyword wird jeder Funktionsaufruf mit der Funktionsdefinition ersetzt. Dadurch kann das Programm schneller werden, da es manchmal mehr Resourcen kostet ist die Funktion aufzurufen und dann auszuführen, als sie ‚direkt‘ aufzurufen.

(Heutzutage ist das jedoch nicht sehr sinnvoll, da die meisten Compiler das Programm selber optimisieren)

21. Enum

Enumerationen werden in C++ mit dem Keyword ‚enum‘ erstellt.

22. Datei schreiben/ auslesen

Mit der Bibliothek ‚fstream‘ kann man in eine Datei schreiben. Zuerst muss man einen ‚Output file stream‘ mit ‚ std::ofstream‘ erstellen, welchen man dann mit der ‚open‘ Methode öffnet. Mit ‚<<‘ kann man dann wie beim ‚cout‘ etwas ausgeben, in diesem Fall schreibt man es in die Datei. Danach muss man den ‚Output file stream‘ nur noch mit der ‚close‘-Methode schließen.

#include <iostream>  
#include <fstream>  
  
int main(int argc, char \*argv[]) {  
  
 std::ofstream file;  
  
 file.open(R"(D:\Windows\Desktop\Workspaces\CppWorkspace\CPP\test.txt)");  
  
 file << "Hello World!";  
  
 file.close();  
  
 system("pause");  
  
 return 0;  
}

Um eine Datei einzulesen, benötigt man den ‚Input file stream‘. Den muss man dann auch wieder mit der ‚open‘-Methode öffnen, jedoch benötigt man hier einen String in den der Content der Datei geschrieben werden kann, was man meistens in einem ‚while‘-Loop mit der ‚getline‘-Methode macht, welche automatisch jede Zeile einer Datei in einen String schreibt und ‚false‘ returned, wenn es keine Zeile zum Auslesen mehr gibt. Am Ende muss man den ‚Input file stream‘ nur noch mit der ‚close‘-Methode schließen.

#include <iostream>  
#include <fstream>  
#include <string>  
  
int main(int argc, char \*argv[]) {  
  
 std::ifstream file;  
  
 std::string line;  
  
 file.open(R"(D:\Windows\Desktop\Workspaces\CppWorkspace\CPP\test.txt)");  
  
 if(file.is\_open()){  
 while(std::getline(file, line)){  
 std::cout << line << std::endl;  
 }  
 file.close();  
 }else{  
 std::cerr << "Fehler beim Öffnen der Datei!" << std::endl;  
 }  
  
 system("pause");  
  
 return 0;  
}

Da beim Auslesen/ Schreiben diverse Fehler auftreten können, sollte man mit ‚is\_open()‘ prüfen, ob die Datei erfolgreich geöffnet wurde.

Außerdem gibt es noch verschiedene Modi, die man zum Öffnen verwenden kann.

#include <iostream>  
#include <fstream>  
  
int main(int argc, char \*argv[]) {  
  
 std::ofstream file;  
  
 file.open(R"(D:\Windows\Desktop\Workspaces\CppWorkspace\CPP\test.txt)", std::ios::app);  
  
 if(file.is\_open()){  
 file << "Hello World!";  
 }else{  
 std::cerr << "Fehler!" << std::endl;  
 }  
 file.close();  
  
 system("pause");  
  
 return 0;  
}

Die wichtigsten Modi sind ‚app‘(append), ‚trunc‘(Datei wird beim Öffnen komplett überschrieben), ‚ate‘(geht zum Ende der Datei, wie ‚app‘, jedoch kann man die Zeigerposition nachher noch verändern) und ‚binary‘ (Datei wird in Binär-Format geöffnet).

Zip-Dateien kann man nicht so einlesen, wie normale Text Dateien. Zip-Dateien muss man im Binär-Format öffnen und den Inhalt dann in einen Buffer schreiben. Die Größe des Buffers wird durch ‚das Ende‘ der Datei (welches man hat, da die Datei auch im ‚ate‘-Modus geöffnet wurde) bestimmt. Auch das Ausgeben funktioniert dadurch, dass man durch den Buffer iteriert, solange ‚i‘ kleiner ‚size‘ (die Größe der Datei, die auch zum Erstellen des Buffers genutzt wurde) ist.

#include <iostream>  
#include <fstream>  
  
int main(int argc, char \*argv[]) {  
  
 char \*buffer;  
  
 std::ifstream f;  
  
 f.open(R"(D:\Windows\Desktop\Workspaces\CppWorkspace\CPP\test.zip)", std::ios::binary | std::ios::ate);  
  
 if(f.is\_open()){  
 int size = f.tellg(); // Ende des 'Cursors', gibt in diesem Fall die Dateigröße an, da er mit 'ate' nach ganz hinten gesetzt wird  
 buffer = new char[size];  
  
 f.seekg(0, std::ios::beg); // 'Cursor' an den Anfang  
 f.read(buffer, size);  
 f.close();  
  
 std::cout << "Inhalt der Datei: " << std::endl;  
 for(int i = 0; i < size; i++){  
 std::cout << buffer[i];  
 }  
 std::cout << std::endl;  
  
 delete buffer;  
 buffer = nullptr;  
 }else{  
 std::cerr << "Fehler beim Öffnen der Datei!" << std::endl;  
 }  
  
 system("pause");  
  
 return 0;  
}

23. Klasse erstellen

#include <iostream>  
  
class Mensch{  
public:  
 std::string name;  
 int alter;  
  
 Mensch(int alter\_, std::string name\_){ // Konstruktor  
 alter = alter\_;  
 name = name\_;  
 }  
private:  
};  
  
int main() {  
  
 Mensch mensch(15, "Test"); // Objekt ist schon instanziiert  
  
 std::cout << mensch.name << std::endl;  
 std::cout << mensch.alter << std::endl;  
  
 system("pause");  
  
 return 0;  
}

Eine Klasse wird mit dem Keyword ‚class‘ erstellt. Mit ‚public:‘ oder ‚private:‘ kann man die [auf das Keyword] folgenden Attribute public oder private machen. Der Konstruktor wird, wie in Java, einfach mit dem Klassennamen erstellt. Für das Instanziieren muss man nur den Klassenamen und den Objekt- bzw. Instanzennamen schreiben und falls ein Konstruktor benötigt sein sollte, ruft man diesen mit ‚()‘ hinter dem Objekt- bzw. Instanzennamen auf.

23. Structs

Structs sind genau dasselbe wie Klassen, jedoch sind die Attribute eines Structs, anders als bei einer Klasse, standardmäßig public.



Structs werden meistens verwendet, um ‚verwandte‘ bzw. ‚ähnliche‘ Variablen zu halten, auch wenn sie theoretisch Methoden beinhalten können.

Structs ‚halten‘ einfach nur Daten, Klassen für Objekte mit Funktion.

23. Klassenprototypen

Bei Klassenprototypen wird, wie bei Funktionsprototypen, zuerst der Klassenprototypen erstellt und dieser wird dann wo anders implementiert.

In diesem Beispiel ist der Prototyp in der ‚Mensch.hpp‘:

#pragma once  
#include <string>  
  
class Mensch{ // Prototyp  
private:  
 std::string name;  
 int alter;  
public:  
 Mensch(int alter\_, std::string name\_); // Ohne Rumpf, da Prototyp  
 std::string getName();  
 int getAlter();  
};

Implementiert werden die Methoden und der Konstruktor dann in der ‚Mensch.cpp‘:

#pragma once  
#include "Mensch.hpp"  
  
Mensch::Mensch(int alter\_, std::string name\_){ // Konstruktor implementieren  
 alter = alter\_;  
 name = name\_;  
}  
  
std::string Mensch::getName() {  
 return name;  
}  
  
int Mensch::getAlter() {  
 return alter;  
}

In der ‚main.cpp‘ kann man diese [im Nachhinein implementierte] Klasse dann ganz normal nutzen:

#include <iostream>  
#include "Mensch.hpp"  
  
int main() {  
  
 Mensch mensch(15, "Test");  
  
 std::cout << mensch.getAlter() << std::endl;  
 std::cout << mensch.getName() << std::endl;  
  
 system("pause");  
  
 return 0;  
}

23. Static

Statische Elemente existieren nur einmal pro Klasse anders als nicht statische Elemente, welche immer einmal pro Instanz existieren. So kann man ‚aus einer Klasse‘ immer auf eine statische Variable zugreifen, welche Instanz unabhängig ist. In diesem Beispiel gibt es eine statische Variable ‚anzahlMenschen‘ und eine statische Methode ‚getAnzahlMenschen‘, welche die statische Variabel zurückgibt.

Mensch.hpp:

#pragma once  
#include <string>  
  
class Mensch{ // Prototyp  
private:  
 static int anzahlMenschen;  
 std::string name;  
 int alter;  
public:  
 Mensch(int alter\_, std::string name\_); // Ohne Rumpf, da Prototyp  
 static int getAnzahlMenschen();  
 std::string getName();  
 int getAlter();  
};

Mensch.cpp:

#pragma once  
#include "Mensch.hpp"  
  
int Mensch::anzahlMenschen = 0;  
  
Mensch::Mensch(int alter\_, std::string name\_){ // Konstruktor implementieren  
 alter = alter\_;  
 name = name\_;  
 anzahlMenschen++;  
}  
  
std::string Mensch::getName() {  
 return name;  
}  
  
int Mensch::getAlter() {  
 return alter;  
}  
  
int Mensch::getAnzahlMenschen() {  
 return anzahlMenschen;  
}

main.cpp:

#include <iostream>  
#include "Mensch.hpp"  
  
int main() {  
  
 Mensch mensch(15, "Test");  
  
 std::cout << mensch.getAlter() << std::endl;  
 std::cout << mensch.getName() << std::endl;  
  
 Mensch mensch2(17, "Test123");  
  
 std::cout << Mensch::getAnzahlMenschen() << std::endl;  
  
 system("pause");  
  
 return 0;  
}

(Mensch::getAnzahlMenschen() gibt ‚2‘ zurück)

23. Initialisierungslisten

Mit Initialisierungslisten kann man Attribute einer Klasse direkt mit den Werten der Argumente initialisieren. Dafür muss man hinter dem Konstruktor, in dem die Parameter stehen, ein ‚:‘ stehen, hinter welchem dann die Attribute und die Argumente in Klammern stehen. Dadurch sieht die ‚Mensch.cpp‘ jetzt so aus:

#pragma once  
#include "Mensch.hpp"  
  
int Mensch::anzahlMenschen = 0;  
  
Mensch::Mensch(int alter, std::string name)  
 : alter(alter), name(name)  
{  
 anzahlMenschen++;  
}  
  
Mensch::Mensch() : alter(-1), name("N/A"){  
 anzahlMenschen++;  
}  
  
std::string Mensch::getName() {  
 return name;  
}  
  
int Mensch::getAlter() {  
 return alter;  
}  
  
int Mensch::getAnzahlMenschen() {  
 return anzahlMenschen;  
}

Jetzt benötigt man auch keine ‚\_‘ mehr, da keine Namenskonflikte entstehen können.

23. Const bei Klassen

In C++ sorgt ‚const‘ vor einer Klasse dafür, dass man die Attribute der Klasse nicht manipulieren kann. Um sicherzustellen, dass das immer eingehalten werden kann, müssen auch Methoden, damit sie (von einer mit ‚const‘ instanziierten Instanz) aufgerufen werden können, mit ‚const‘ deklariert werden.

Das ‚const‘ wird daher im Prototypen in der Mensch.hpp hinter die Methoden geschrieben:



In der Implementation in der Mensch.cpp muss das natürlich auch geschehen:

std::string Mensch::getName() const {  
 return name;  
}  
  
int Mensch::getAlter() const {  
 return alter;  
}

Genutzt werden kann die Instanz dann wie folgt:

#include <iostream>  
#include "Mensch.hpp"  
  
int main() {  
  
 const Mensch mensch(15, "Test");  
  
 std::cout << mensch.name << std::endl;  
 // mensch.name = "123"; Fehler, da Attribute nicht manipuliert werden dürfen  
 std::cout << mensch.getAlter() << std::endl;  
  
 system("pause");  
  
 return 0;  
}

23. Const reference

#include <iostream>  
#include "Mensch.hpp"  
  
void printMensch(const Mensch &mensch){  
 std::cout << "Alter: " << mensch.getAlter() << std::endl;  
 std::cout << "Name: " << mensch.getName() << std::endl;  
}  
  
int main() {  
  
 const Mensch mensch(15, "Test");  
  
 printMensch(mensch);  
  
 system("pause");  
  
 return 0;  
}

Eine konstante Referenz ist komplett konstant, d.h. die Werte können nicht verändert werden. Für Objekte (wie in dem Beispiel) heißt das, dass man (wie bei konstanten Klassen) die Attribute des Objektes nicht ändern kann.

23. Destruktor

Ein Destruktor wird aufgerufen, wenn ein Objekt ‚zerstört‘ wird. Sinnvoll ist das, wenn die Instanz dynamische Variablen auf dem Heap angelegt hat. Diese müssen nämlich zerstört werden, jedoch nicht vorzeitig. Dafür ist der Destruktor da, da er genau dann aufgerufen wird, wenn die Instanz zerstört wird, wobei er dann die dynamischen Attribute mit ‚delete‘ löschen kann. In der Mensch.hpp wird der Destruktor im Prototypen ergänzt:



Der Destruktor ist dem Konstruktor sehr ähnlich außer, dass vor dem Destruktor ein ‚~‘ steht.

Implementiert wird er in der Mensch.cpp:



Das nächste Mal, dass man ein Programm mit einer ‚Mensch‘ Instanz beendet wird, wird ‚Destruktor‘ ausgegeben.

23. Copy Constructor

Ein Copy Constructor ist ein nicht sichtbarer Konstruktor, der es erlaubt alle Attribute aus einer Instanz einer Klasse in eine andere Instanz zu kopieren. Jedoch versucht die kopierte Instanz dann auch im Destruktor denselben dynamischen Speicher zu löschen bzw. freizugeben (falls Speicher im Destruktor freigegeben wird), was zu einem Fehler führt. Mit einem überschriebenem Copy Constructor kann man der kopierten Instanz eigenen dynamischen Speicher zuweisen, welcher dann im Destruktor der kopierten Instanz auch wieder freigegeben wird.

DynamicArray.hpp:

#pragma once  
  
class DynamicArray{  
private:  
 int \*data;  
 int size;  
public:  
 DynamicArray();  
 DynamicArray(int size);  
 ~DynamicArray();  
 DynamicArray(const DynamicArray &otherDynamicArray);  
  
 int getSize() const;  
 int get(int index) const;  
 void set(int index, int value);  
};

DynamicArray.cpp:

#pragma once  
#include "DynamicArray.hpp"  
  
DynamicArray::DynamicArray() {  
 size = 10;  
 data = new int[size];  
}  
  
DynamicArray::DynamicArray(int size) : size(size) {  
 data = new int[size];  
}  
  
DynamicArray::DynamicArray(const DynamicArray &otherDynamicArray) {  
 size = otherDynamicArray.size;  
 data = new int[size]; // neuen dynamischen Speicher erzeugen  
 for(int i = 0; i < size; i++){  
 data[i] = otherDynamicArray.data[i]; // Werte aus der Originalinstanz in den neuen dynamischen Speicher kopieren  
 }  
}  
  
DynamicArray::~DynamicArray() {  
 delete data;  
}  
  
int DynamicArray::getSize() const {  
 return size;  
}  
  
int DynamicArray::get(int index) const {  
 return data[index];  
}  
  
void DynamicArray::set(int index, int value) {  
 data[index] = value;

main.cpp:

#include <iostream>  
#include "DynamicArray.hpp"  
  
int main() {  
  
 DynamicArray d(100);  
  
 d.set(5, 105);  
  
 DynamicArray e(d); // 'e' übernimmt dabei alle Eigenschaften/ Werte von 'd'  
  
 std::cout << d.get(5) << std::endl;  
 std::cout << e.get(5) << std::endl;  
  
 system("pause");  
  
 return 0;  
}

Außerdem lässt sich die Möglichkeit, den Copy Constructor zu nutzen, mit einem ‚= delete‘ hinter der Definition ‚verbieten‘.

24.1 this

Das Keyword ‚this‘ ist ein Pointer auf das aktuelle Objekt. In dem Beispiel wird ‚this‘ verwendet, um eine Initialisierungsliste zu vermeiden:



Hier verweist ‚(\*this)‘ (dereferenziert, weil) auf auf das aktuelle Objekt. Damit greift man auf das Attribut des aktuell verwendeten Objektes zu und weißt diesem Attribut den Wert des Parameters zu.

24.2 Arrow Operator

Anstatt jedesmal ‚(\*this).x‘ zu schreiben, kann das ganze auch mit dem ‚arrow operator(->)‘ abgekürzt werden. Mit dem arrow operator kann das dereferenzieren nun abgekürzt werden. So brauch man nicht mehr die ganzen Klammern zum dereferenzieren und aufrufen eines Attributes ‚(\*(p)).n‘, sondern man kann es mit dem Pfeiloperator wie folgt abkürzen: ‚p->n‘. Nun sieht der Konstruktor (mit dem Pfeiloperator) wie folgt aus:



25. Move Constructor

In dem Beispiel, wird jetzt der ‚+‘ Operator überladen, um zwei DynamicArrays miteinander zu addieren.

DynamicArray.hpp:



DynamicArray.cpp:

DynamicArray DynamicArray::operator+(const DynamicArray &otherDynamicArray) {  
 if(size != otherDynamicArray.size){  
 std::cerr << "Fehler! Dynamic Arrays müssen gleich groß sein!" << std::endl;  
 exit(-1);  
 }  
 DynamicArray retVal(size);  
 for(int i = 0; i < size; i++){  
 retVal[i] = data[i] + otherDynamicArray.data[i]; // Werte werden addiert  
 }  
 return retVal;  
}

Genutzt werden kann diese Funktion nun wie folgt:

#include <iostream>  
#include "DynamicArray.hpp"  
  
int main() {  
  
 DynamicArray d(100);  
  
 d.set(5, 105);  
  
 DynamicArray e(d);  
  
 DynamicArray f = d + e; // retVal(von + Operator) wird nicht direkt in 'f' gespeichert, sondern der retVal wird in 'f' kopiert  
  
 std::cout << f[5] << std::endl;  
  
 system("pause");  
  
 return 0;  
}

Hier werden die neuen Werte nach der Operation jedoch nicht direkt ‚in f bewegt‘, sondern sie werden in f kopiert, was den ganzen Vorgang langsamer macht. Das ist unnötig, da der Rückgabewert von ‚d + e‘ danach eh ‚verworfen‘ wird. Daher kann man einen ‚Move Constructor‘ erstellen in dem man dann angibt, was mit den Werten (‚d + e‘ bzw. ‚retVal‘) geschehen soll. Dafür muss erst der Prototyp in Mensch.hpp geschrieben werden:



Implementiert werden kann der ‚Move Constructor‘ in der Mensch.cpp wie folgt:

DynamicArray::DynamicArray(DynamicArray &&other) { // Move Constructor  
 // Daten werden direkt verschoben, anstatt kopiert  
 data = other.data;  
 size = other.size;  
 // Damit Destruktor 'data' nicht für ungültig erklärt bzw. Fehler wirft  
 other.data = nullptr;  
}

Dabei wird ‚other.data‘ auf ‚nullptr‘ gesetzt, da der Destruktor von ‚other‘ versuchen wird ‚data‘ zu löschen, was nicht geht, da ‚other.data‘ nach ‚data‘ verschoben wurde.

26.1 Operator Overloading

Mit Operator Overloading kann man die ‚Bedeutung‘ bzw. Funktionsweise eines Operator ändern. Dies macht man mithilfe des ‚operator‘ Keywords. Davor steht der Typ und hinter dem Keyword kommt das Symbol und in Klammern die Parameter Liste.

DynamicArray.hpp:



DynamicArray.cpp:



In der Implementation wird gesagt, dass der Operator ‚[]‘ eine Integer Referenz auf den Index, der in den ‚[]‘ genannt werden soll, in ‚data‘ zurückgeben soll.

26.2 (Copy) assignment operator

Beim Copy assignment operator sind ‚Ziel‘ und ‚Quelle‘ einer Zuweisung derselben Klasse. Falls der Programmierer für diesen Fall nichts implementiert, wird eine Standardversion ausgeführt die jedes Attribut kopiert. Der copy assignment operator unterscheidet sich zum copy constructor, da er die Attribute des ‚Zieles‘ selber löschen muss und die Selbstzuweisung selber handeln muss, wobei der copy constructor die Zuweisung zu nicht initialisierten Objekten übernimmt.

DynamicArray.hpp:



DynamicArray.cpp:

DynamicArray& DynamicArray::operator=(const DynamicArray &other) {  
 if(&other == this) return \*this;  
 if(other.size != size){  
 delete[] data;  
 size = other.size;  
 data = new int[size];  
 }  
 for(int i = 0; i < size;i++){  
 data[i] = other.data[i];  
 }  
 return \*this;  
}

main.cpp:



(In dem Beispiel ist die Quelle ‚f‘ und das Ziel ‚d’, d.h ‚d‘ soll die Werte von ‚f‘ übernehmen)

Quelle: <https://en.wikipedia.org/wiki/Assignment_operator_(C++)>

26.3 Move assignment operator

Der move assignment operator ist dem move constructor sehr ähnlich, jedoch sorgt er hier dafür, dass die Werte, die eh bald zerstört werden (R-Value), in ein bestehendes Objekt ‚geschrieben‘ bzw. ‚bewegt‘ werden. Dafür überlädt man auch wieder den ‚=‘-Operator wie folgt:

DynamicArray.hpp:



DynamicArray.cpp:



main.cpp:

DynamicArray d(100);  
d[5] = 100;  
  
DynamicArray e(d);  
  
DynamicArray f = d + e;  
  
d = f + e;  
  
std::cout << d[5] << std::endl; // 300

Quelle: <https://learn.microsoft.com/en-us/cpp/cpp/move-constructors-and-move-assignment-operators-cpp?view=msvc-170>

27. Exception Handling

In C++ kann man alles als Exception schmeißen. In diesem Beispiel wird dafür ein Integer geschmissen, der dann auch abgefangen wird.

Hier wird der Fehler mit ‚throw‘ geworfen, sollte die Bedingung eintreten:



Catchen tut man den Fehler wie folgt:



Da beim Auftreten des Fehlers eine 10 geworfen wird, wird im ‚catch‘ eine 10 ausgegeben.

28.1 Vererbung

Zuerst brauchen wir eine neue Klasse, von der wir dann erben können. Dafür erstellen wir uns eine Lebewesen.hpp (Deklaration) und eine Lebewesen.cpp (Implementation).

Lebewesen.hpp:



Lebewesen.cpp:



In der Mensch.hpp können wir jetzt wie folgt von der Lebewesen Klasse erben:



(‚: public Klasse‘)

In der main.cpp können wir diese von Lebewesen geerbte Methode über ein ‚Mensch‘-Objekt aufrufen:

#include "DynamicArray.hpp"  
#include "Mensch.hpp"  
  
int main() {  
  
 Mensch m;  
 m.makeSound();  
  
 system("pause");  
  
 return 0;  
}

28.2 public/protected/private bei Vererbung

Klassen kann man nicht nur public sondern auch protected und private erben.

- Ist ein Element von der zu erbenden Klasse private, wird dies gar nicht übergeben.

- Ist ein Element von der zu erbenden Klasse public oder protected und wird mit der private als Vererbungsart geerbt wird es in der Klasse, die erbt, private

- Ist ein Element von der zu erbenden Klasse public oder protected und wird mit protected als Vererbungsart geerbt wird es in der Klasse, die erbt, protected

- Ist ein Element von der zu erbenden Klasse public(protected) und wird mit public als Vererbungsart geerbt wird es in der Klasse, die erbt, public(protected)

Ein Bild, das Tisch enthält.

Automatisch generierte Beschreibung

28.3 virtual und override

‚Problem‘:

Überschreibt die Mensch Klasse jetzt die ‚makeSound‘-Methode





und hat dann einen Pointer ‚Lebewesen‘ auf eine ‚Mensch‘ Klasse und ruft die Methode dann über den Pointer wie folgt auf



wird die nicht überschriebene Version (‚x‘) ausgegeben.

Lösen tut man das mit dem ‚virtual‘-Keyword.

Lebewesen.hpp:



Dazu kann man in der erbenden Klasse ein ‚override‘ hinter die Überschriebene Methode schreiben, da der Compiler direkt einen Fehler wirft, wenn diese Keyword hinter eine Methode geschrieben wird, die nicht versucht eine virtual-Methode einer Super Klasse zu überschreiben.

28.4 pure virtual

Eine ‚pure virtual‘-Methode kann mit ‚=0‘ hinter einer virtual-Methode erstellt werden. Anders als eine ‚virtual‘-Methode, die die Möglichkeit gibt, die Funktionen der Super-Klasse in einer Sub-Klasse zu überschreiben, muss man bei einer ‚pure virtual‘-Methode die Funktionen der Super Klasse in einer Sub-Klasse überschreiben. Die Super-Klasse selbst kann eine ‚pure virtual‘-Methode also nicht selbst implementieren, das muss immer eine Sub-Klasse machen.

Lebewesen.hpp:



main.cpp:



Hier wird jetzt auch die ‚Mensch‘-Implementation der ‚makeSound‘-Methode ausgeführt, da die Klasse, die eine ‚pure virtual‘-Methode deklariert, diese nicht selber implementieren darf (und wenn sie von dieser Klasse implementiert wird, wird die Implementation einfach ignoriert).

Dies sorgt aber auch dafür, dass man die Super-Klasse nicht instanziieren kann.

28.5 virtual Destruktor

Jede Klasse, die eine virtuelle Funktion definiert, sollte zugleich auch einen virtuellen Destruktor definieren. Denn wenn man dann einen Pointer vom Typen Lebewesen (Basisklasse bzw. Super-Klasse) auf eine Sub-Klasse hat, wird der Destruktor dieser Sub-Klasse nicht aufgerufen, da die Super-Klasse einen virtuellen Destruktor benötigt, weil sonst einfach der Destruktor von der Super-Klasse aufgerufen wird. Definiert man nun also einen virtuellen Destruktor in der Super-Klasse weiß ‚delete‘, dass er den Destruktor der Sub-Klasse aufrufen muss.



Hier (bei ‚delete‘) würde der Destruktor von ‚Lebewesen‘ aufgerufen werden, weshalb man einen virtuellen Destruktor in ‚Lebewesen‘ definieren muss, damit der Destruktor der Sub-Klasse (‚Mensch‘) beim deleten aufgerufen wird.

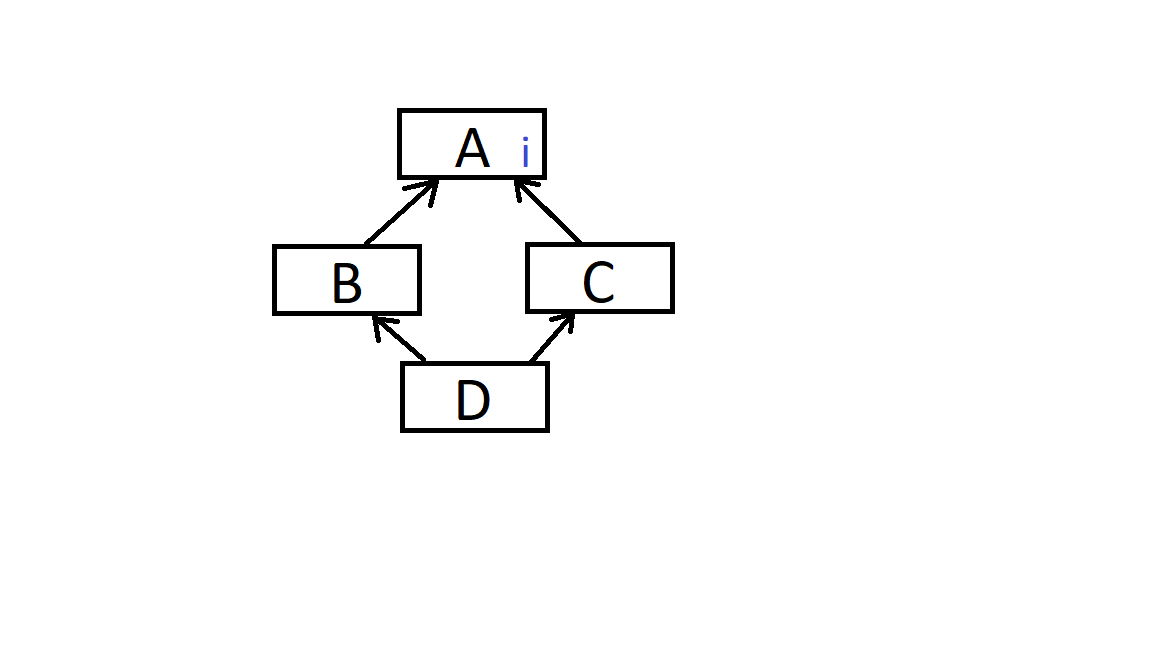


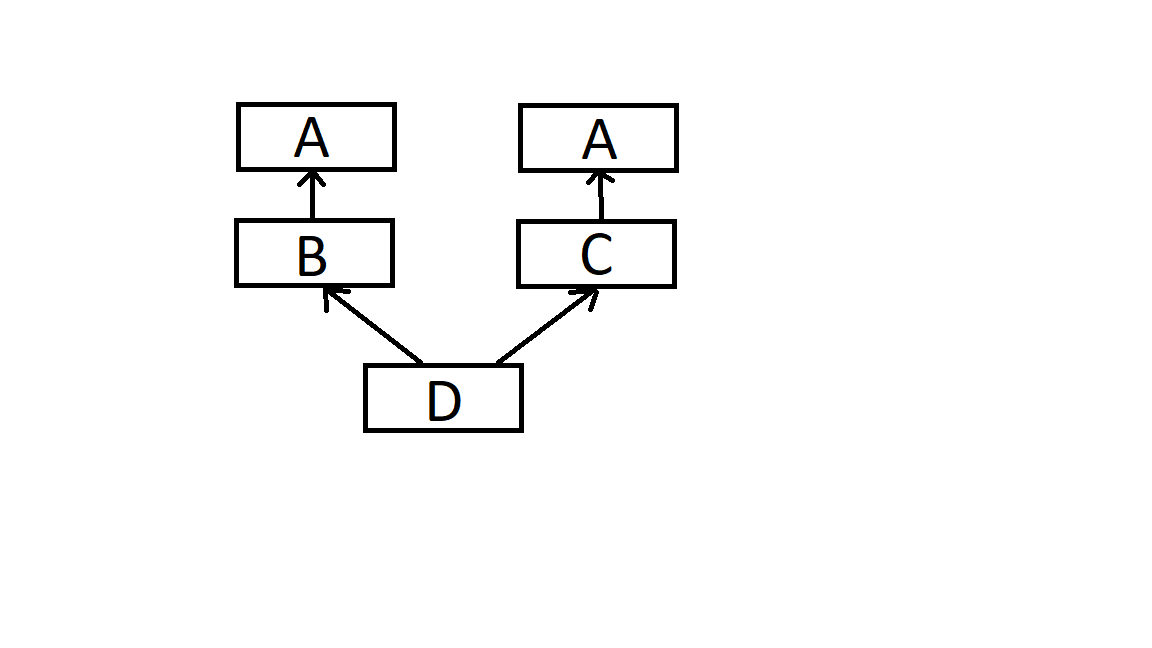
28.6.1 Mehrfachvererbung

Bei der Mehrfachvererbung erbt eine Klasse von mehreren Klassen.

class A{  
public:  
 int a = 10;  
 int i;  
};  
class B{  
public:  
 int b = 15;  
 int i;  
};  
class C: public A, public B{  
  
};  
  
int main() {  
  
 C c;  
  
 c.A::i = c.a;  
 c.B::i = c.b;  
  
 std::cout << c.A::i << std::endl;  
 std::cout << c.B::i << std::endl;  
  
 system("pause");  
  
 return 0;  
}

In diesem Beispiel wird ein Problem von Mehrfachvererbung gezeigt. Nämlich ist ‚i‘ in beiden Basisklassen (A, B) definiert, dadurch weiß der Compiler bei der Zuweisung ‚c.i = 10‘ nicht, welches ‚i‘ gemeint ist. Das kann man jedoch mit dem Klassennamen und ‚::‘ vor dem Attribut spezifizieren.

28.6.2 Deadly Diamond of Death

Verhältnis der Klassen zueinander (B und C erben von A, A hat eine Variable namens ‚i‘, D erbt von B und C)

Im Speicher trifft eher diese Darstellung zu, da im Speicher B und C nicht beide auf eine Klasse A zugreifen, sondern sie greifen beide auf unterschiedliche A Klassen zu, so dass es sozusagen zwei Varianten von A gibt, nämlich einmal Variante B A und Variante C A. Möchte man über D nun auf ‚i‘ in A zugreifen muss man sich für eine der beiden Varianten entscheiden, also ob man auf B A und dann i oder auf C A und dann i zugreift.

Ein Beispielprogramm:



28.6.3 Virtuelle Vererbung

Um das ‚Deadly Diamond of Death’ Problem zu lösen, kann man virtuelle Vererbung nutzen, wobei es dann nicht wie oben zwei Varianten von A gibt, sondern B und C auf dieselbe Variante von A zugreifen. Dafür muss man beim Erben von A nur ein ‚virtual‘ vor A schreiben. Theoretisch kann man immer noch über B oder C (‚.B::i‘ bzw. ‚.C::i‘) auf i zugreifen, jedoch greift man jetzt immer auf dasselbe ‚i‘ zu.



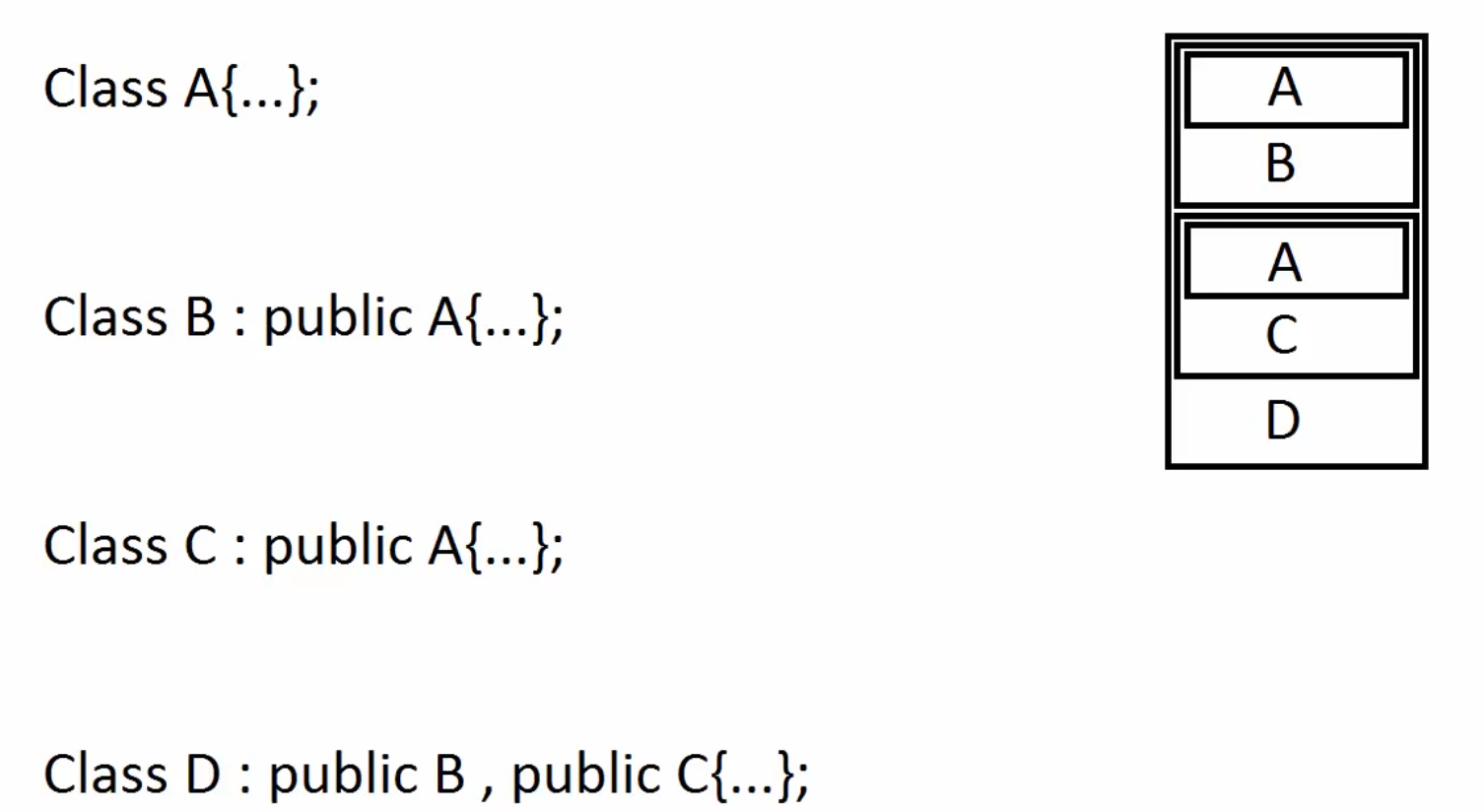
29. Klassenspeicherlayout

(Ein Kasten mit Klassennamen, repräsentiert eine komplette Klasse, also mit Attributen, Methoden etc.)



Erbt man von einer Klasse, so liegt im Speicher zuerst die Klasse von der man erbt und danach liegt im Speicher die Klasse die von der Klasse erbt.

Der Deadly Diamond of Death sieht im Speicher wie folgt aus:



Dort kann man sehen, dass es die Superklasse wirklich 2mal abgespeichert wird, daher existiert auch jedes Attribut 2mal.

30.1 Static Cast

Beim Downcasting in C++ zeigt ein Pointer einer Superklasse auf ein Objekt einer Subklasse. Downcasten kann man in C++ einfach mit dem ‚static\_cast‘, wenn man genau weiß, worauf der Superklassen-Pointer zeigt, wie in dem Beispiel hier:

Dadurch, dass der Pointer vom Typ der Superklasse ist, kann er nicht nur auf die Attribute der Subklasse zeigen, auf die er zeigt, sondern er kann auch auf die Attribute der Superklasse zeigen, der der Typ des Pointers ist.

Da ‚static\_cast‘ jedoch keine ‚Sicherheitschecks‘ vorher durchführt und daher trotz fehlerhaftem Cast trotzdem weitermacht sollte man ‚static\_cast‘ wirklich nur nutzen, wenn man sich absolut sicher ist, was man downcastet.

30.2 Dynamic Cast

Mit ‚dynamic\_cast‘ kann ein Zeiger sicher in eine abgeleitete Klasse konvertiert werden. Hat das Objekt tatsächlich den Typ, den man für die Umwandlung angegeben hat, dann liefert er wieder die Adresse des Objekt zurück, diesmal natürlich mit dem Datentyp der abgeleiteten Klasse. Hat das Objekt hingegen einen anderen Datentyp als den angegebenen, liefert ‚dynamic\_cast‘ einen Nullzeiger zurück. Nach einem Aufruf von ‚dynamic\_cast‘ müssen Sie also immer prüfen, ob die Umwandlung erfolgreich war. Außerdem benötigt der Datentyp, der gecastet werden soll mindestens eine virtuelle Methode (polymorph).

#include <iostream>  
#include "DynamicArray.hpp"  
#include "Mensch.hpp"  
  
class A{  
public:  
 int i = 10;  
 virtual void doSmh(){};  
};  
class B : public A{  
public:  
 int j = 15;  
};  
class C : public A{};  
  
int main() {  
  
 A \*aB = new B();  
 A \*aC = new C();  
  
 B \*bP = dynamic\_cast<B\*>(aB);  
 B \*bP1 = dynamic\_cast<B\*>(aC);  
  
 if(B \*bP = dynamic\_cast<B\*>(aB)){  
 std::cout << "Ist B!" << std::endl; // wird ausgegeben  
 }else{  
 std::cout << "Nicht B!" << std::endl;  
 }  
  
 if(B \*bP1 = dynamic\_cast<B\*>(aC)){  
 std::cout << "Ist B!" << std::endl;  
 }else{  
 std::cout << "Nicht B!" << std::endl; // wird ausgegeben  
 }  
  
 std::cout << bP << std::endl << bP1 << std::endl;  
  
 system("pause");  
  
 return 0;  
}

Quelle: <https://de.wikibooks.org/wiki/C%2B%2B-Programmierung/_N%C3%BCtzliches/_Casts#:~:text=Mit%20dynamic_cast%20kann%20ein%20Zeiger,dem%20Datentyp%20der%20abgeleiteten%20Klasse>.

30.3 Reinterpret Cast

Mit dem ‚reinterpret\_cast‘ kann man Ganzzahlen unterschiedlich interpretieren. So kann man z.B. einen Integer-Pointer als Float-Pointer interpretieren.

Dies kann nützlich sein, falls man Informationen in anderer Form darstellen möchte.



30.4 Const Cast

Mit dem ‚const\_cast‘ kann man das ‚const‘ einer Variable wieder wegnehmen (bzw. die Konstantheit wieder aufheben) und damit theoretisch wieder in die Variable schreiben. Jedoch sollte man dies nicht machen, da manche Compiler Optimierungen an Konstanten vornehmen, was dann später beim Beschreiben einer vorherigen Konstante zu Problemen führen könnte.

#include <iostream>  
#include "DynamicArray.hpp"  
#include "Mensch.hpp"  
  
int main() {  
  
 const char c1[] = "Test";  
  
 char \*c2 = const\_cast<char\*>(c1);  
  
 c2[0] = 't'; // undefined behaviour, da manche compiler optimierungen an const variablen vornehmen könnten, was später dann zu Fehlern führen kann  
  
 std::cout << c1 << std::endl; // test  
  
 system("pause");  
  
 return 0;  
}

Quelle: <https://de.wikibooks.org/wiki/C%2B%2B-Programmierung/_N%C3%BCtzliches/_Casts#:~:text=Mit%20dynamic_cast%20kann%20ein%20Zeiger,dem%20Datentyp%20der%20abgeleiteten%20Klasse>.

31. Klassentemplates

Wie Funktionstemplates, kann man mit Klassentemplates dafür sorgen, dass eine Klasse mehrere Typen ‚annehmen‘ kann, was durch einen Typ-Parameter erreicht wird. Durch diesen Typ-Parameter, den man in der ganzen Definition und Implementierung angeben muss, kann die Klasse mit jedem beliebigem Typen arbeiten. Außerdem müssen Definition und Implementierung, bei denen Klassentemplates verwendet werden, in einer Datei sein. D.h. man kann nicht mehr zwei Dateien (einmal für die Definition, einmal für die Implementierung) haben.

Dadurch sieht die Dynamic Array Klasse jetzt wie folgt aus:

#pragma once  
  
#include <iostream>  
  
template <typename T>  
  
class DynamicArray{  
private:  
 T \*data;  
 int size;  
public:  
 DynamicArray();  
 DynamicArray(int size);  
 ~DynamicArray();  
 DynamicArray(const DynamicArray<T> &other); // Kopierkonstruktor überschreiben  
 DynamicArray(DynamicArray<T> &&other); // &&otherDynamicArray = 'R-Value', Move Constructor  
  
 int getSize() const;  
 T get(int index) const;  
 void set(int index, T value);  
  
 T& operator[](int index);  
 DynamicArray<T> operator+(const DynamicArray<T> &other);  
 DynamicArray<T>& operator=(const DynamicArray<T> &other);  
 DynamicArray<T>& operator=(DynamicArray<T> &&other);  
};  
  
template <typename T>  
DynamicArray<T>::DynamicArray() {  
 size = 10;  
 data = new T[size];  
}  
  
/\*  
DynamicArray::DynamicArray(int size) : size(size){  
 data = new int[size];  
}  
\*/  
  
template <typename T>  
DynamicArray<T>::DynamicArray(int size){  
 // (\*this).size = size;  
 this->size = size;  
 data = new T[size];  
}  
  
template <typename T>  
DynamicArray<T>::DynamicArray(const DynamicArray<T> &other) {  
 size = other.size;  
 data = new int[size]; // neuen dynamischen Speicher erzeugen  
 for(int i = 0; i < size; i++){  
 data[i] = other.data[i]; // Werte aus der Originalinstanz in den neuen dynamischen Speicher kopieren  
 }  
}  
  
template <typename T>  
DynamicArray<T>::DynamicArray(DynamicArray<T> &&other) { // Move Constructor  
 // Daten werden direkt verschoben, anstatt kopiert  
 data = other.data;  
 size = other.size;  
 // Damit Destruktor 'data' nicht für ungültig erklärt bzw. Fehler wirft  
 other.data = nullptr;  
 std::cout << "Move Constructor wurde aufgerufen! Werte erfolgreich verschoben!" << std::endl;  
}  
  
template <typename T>  
DynamicArray<T>::~DynamicArray() {  
 delete data;  
}  
  
template <typename T>  
int DynamicArray<T>::getSize() const {  
 return size;  
}  
  
template <typename T>  
T DynamicArray<T>::get(int index) const {  
 return data[index];  
}  
  
template <typename T>  
void DynamicArray<T>::set(int index, T value) {  
 data[index] = value;  
}  
  
template <typename T>  
T& DynamicArray<T>::operator[](int index) {  
 return data[index];  
}  
  
template <typename T>  
DynamicArray<T> DynamicArray<T>::operator+(const DynamicArray<T> &other) {  
 if(size != other.size){  
 throw 10;  
 }  
 DynamicArray retVal(size);  
 for(int i = 0; i < size; i++){  
 retVal[i] = data[i] + other.data[i]; // Werte werden addiert  
 }  
 return retVal;  
}  
  
template <typename T>  
DynamicArray<T>& DynamicArray<T>::operator=(const DynamicArray<T> &other) {  
 if(&other == this) return \*this;  
 if(other.size != size){  
 delete[] data;  
 size = other.size;  
 data = new int[size];  
 }  
 for(int i = 0; i < size;i++){  
 data[i] = other.data[i];  
 }  
 return \*this;  
}  
  
template <typename T>  
DynamicArray<T>& DynamicArray<T>::operator=(DynamicArray<T> &&other) {  
 if(&other == this) return \*this;  
 delete[] data;  
 data = other.data;  
 other.data = nullptr;  
 return \*this;  
}

Dadurch kann man die Dynamic Array Klasse jetzt wie folgt verwenden:



31. friend

Mit dem ‚friend‘ Keyword kann eine Klasse auf die privaten Attribute einer anderen Klasse zugreifen. Dafür muss die Klasse, mit den privaten Attributen, die andere Klasse, die auf diese Attribute zugreifen will, als ‚friend‘ Klasse ‚markieren‘. Dies gilt nur einseitig und die Klasse die eine Klasse als ‚friend‘ markiert kann nicht auf die Attribute dieser Klasse zugreifen.



32.1 memset

Mit memset lassen sich einer oder mehrere Werte initialisieren. Dies wird öfters für Arrays genutzt, die mit 0 initialisiert werden sollen, da sie sonst ‚zufällige‘ Zahlen beinhalten würden. Um anzugeben ‚ab was‘ bzw. was man initialisieren möchte muss man als erstes Argument einen void Pointer übergeben. Um das Value zu setzen, übergibt man das Value einfach als zweites Argument als ein Integer, jedoch wird dies als ein ‚unsigend char‘ also 1 Byte interpretiert. Zuletzt übergibt man noch an, wieviel bzw. was (bytes) initialisiert werden soll. Möchte man z.B. ab dem Anfang (den man mit der Übergabe des Pointers macht) 100 Stellen eines int Arrays initialisieren muss man ‚100 \* sizeof(int)‘ als Size angeben.



Quelle: <https://cplusplus.com/reference/cstring/memset/>

32.2 memcpy

‚memcpy‘ wird dazu genutzt, um mehrere Daten schnell zu kopieren, z.B. von einem Array (‚src‘, void-Pointer-Konstante) in einen anderen Array (‚dest‘, void Pointer). Dazu muss man als letztes Argument wie beim ‚memset‘ auch die size angeben, für ein 100 Integer großes Array wäre es also auch: ‚100 \* sizeof(int)‘.

33.1 shared\_ptr

Ein Problem bei Objekten die auf dem Heap liegen ist, dass sie gelöscht werden müssen, was nicht möglich ist wenn z.B. eine Exception geworfen wird, wodurch ein Memory Leak entsteht.

Mit ‚shared\_ptr‘ kann man das verhindern, da diese einen eigenen Destruktor aufrufen, der die ‚Verweisanzahl‘ dann dekrementiert und wenn es keine Verweise mehr auf das Objekt gibt wird das Objekt und der Pointer gelöscht.



Quelle: <https://learn.microsoft.com/de-de/cpp/cpp/how-to-create-and-use-shared-ptr-instances?view=msvc-170>

33.2 unique\_ptr

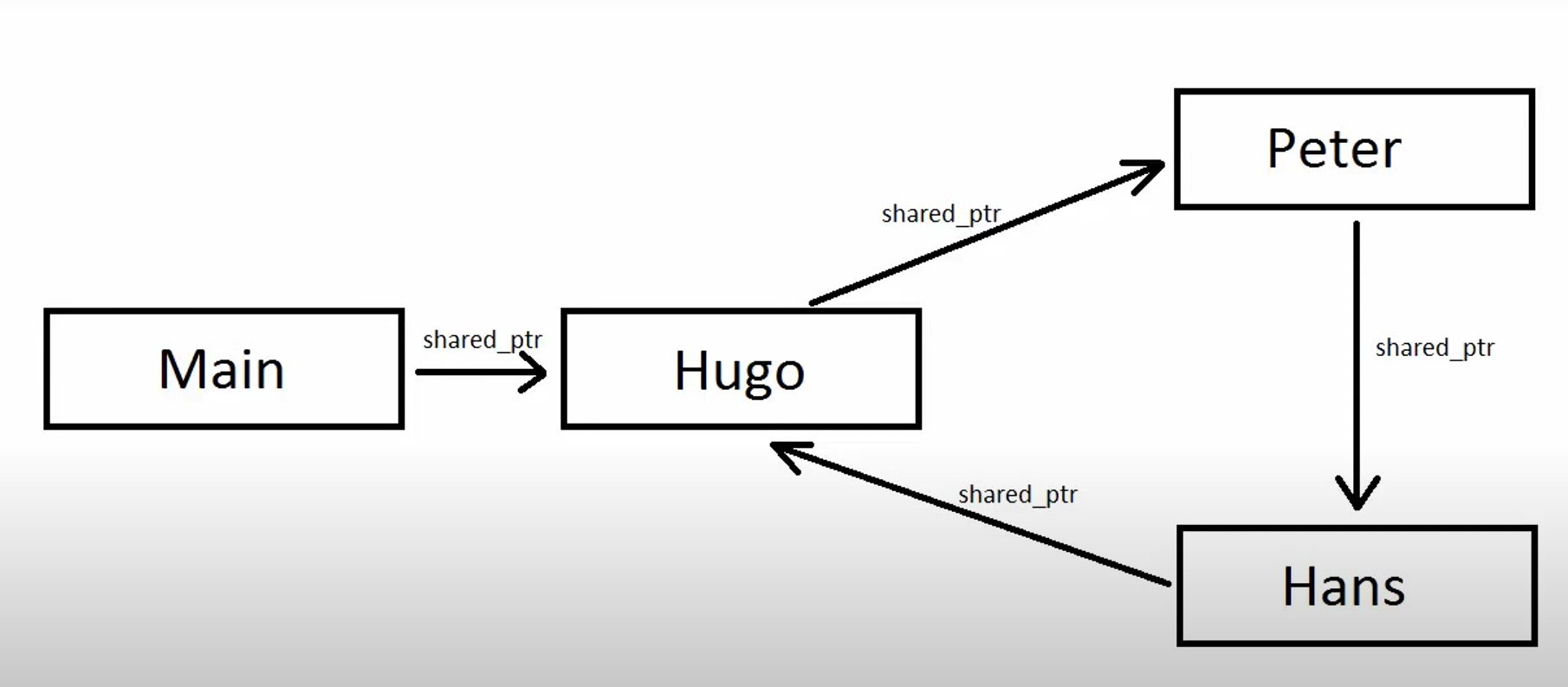
Der ‚unique\_ptr‘ ist das komplette Gegenteil zum ‚shared\_ptr‘, da nur maximal 1 ‚unique\_ptr‘ auf etwas referenzieren kann, d.h. es darf nicht zwei unique Pointer geben, die auf das selbe Objekt zeigen. Jedoch kann man das, worauf der unique Pointer referenziert, in einen anderen unique Pointer mit ‚std::move‘ schieben, sodass der 1. unique Pointer zum null Pointer wird und nur noch der 2. unique Pointer auf das Objekt referenziert.



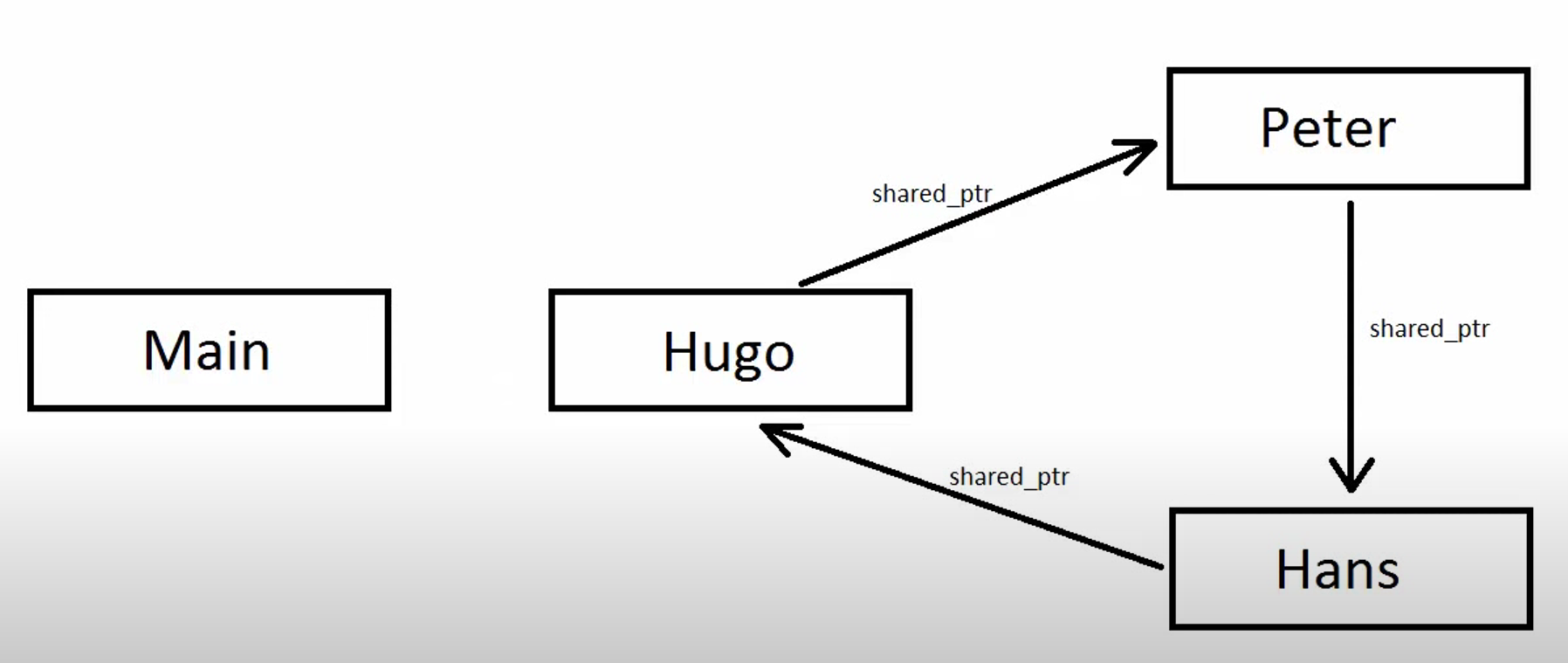
Quelle: <https://learn.microsoft.com/de-de/cpp/cpp/how-to-create-and-use-unique-ptr-instances?view=msvc-170>

33.3 weak\_ptr

Ausgangssituation:



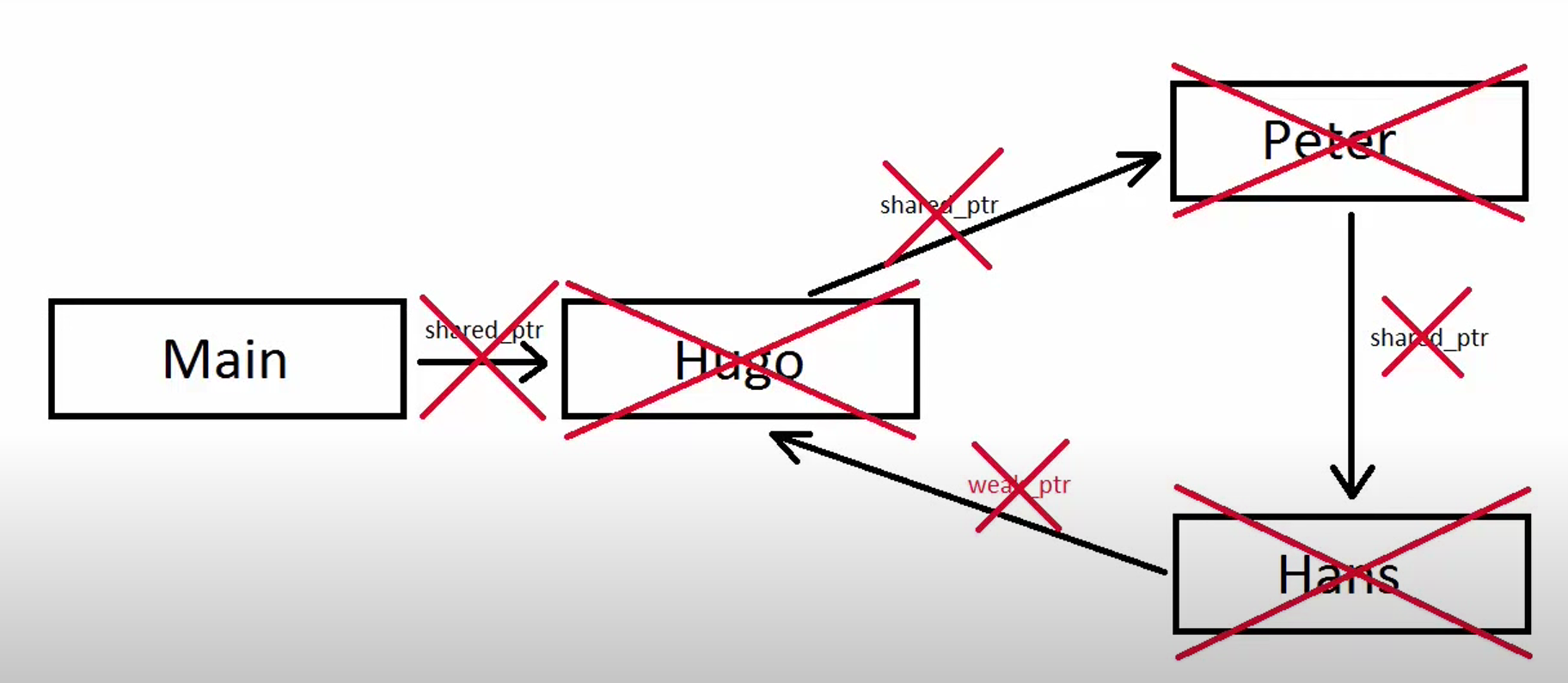
Verlassen des Gültigkeitsbereiches vom shared Pointer auf ‚Hugo‘ oder ähnliches:



Es wurde ein Memory Leak geschaffen, da für die anderen shared Pointer der Referenzcount immer noch größer 0 ist, jedoch kann man diese nicht mehr löschen und sie werden sich selber auch nicht löschen, da der shared Pointer zu ‚Hugo‘ fehlt, was dazu führt, dass der Arbeitsspeicher zugemüllt wird.

Lösung:

Einen ‚weak\_ptr‘ benutzen. Ein ‚weak\_ptr‘ verhält sich genauso wie ein ‚shared\_ptr‘ außer, dass er den Referenzcount nicht erhöt, wenn er ein Objekt referenziert.



Wird ‚Hugo‘ von ‚Hans‘ nun also mit einem weak Pointer referenziert wird der Referenzcounter von ‚Hugo‘ nicht erhöht, wodurch man durch das Löschen vom einzigen shared Pointer der auf ‚Hugo‘ zeigt Hugos Referenzcounter auf 0 setzt, wodurch Hugo gelöscht wird. Darauf folgt eine Kettenreaktion und alle anderen Objekte werden auch gelöscht, da der Referenzcount überall 0 ist.

34.1 Vector Aufbau



Bei Vectoren kann der überladene Operator ‚[]‘ nicht verwendet werden, um den Vector zu erweitern, sollte der Index größer als die Size sein. Daher ist ein Zugriff auf einen Index größer als Size undefined Behaviour.

Außerdem verwenden Vectoren intern auch einen ‚capacity‘ Wert um etwas mehr Platz für neue Heap Allokationen(push\_backs) zu schaffen.

34.2 list und forward\_list

Der einzige Unterschied zwischen einer list und einer forward\_list ist, dass man durch eine list auch von hinten iterieren kann (wodurch man auch gleichzeitig Zugriff auf push\_back und push\_front hat), anders als bei einer forward\_list, bei der man nur von vorne iterieren kann (deswegen kann man auch nicht ‚rbegin‘ oder ‚rend‘ aufrufen). Da man bei einer Liste von vorne oder hinten iterieren kann, kann das Iterieren über eine Liste länger dauern als z.B. bei einem Vektor.

In diesem Beispielcode sind nochmal einige Methoden beider Listen zu sehen:

#include <iostream>  
#include <list>  
#include <forward\_list>  
  
int main(){  
  
 std::list<int> list; // Jedes Element besteht aus zwei Zeigern, einer zeigt auf das vorherige- der andere auf das nächste Element, man kann von vorne und von hinten iterieren  
  
 list.assign(5, 27); // 5 Werte mit dem Wert 27  
  
 list.push\_front(6); // Neuen Wert am Anfang hinzufügen  
 list.push\_back(10); // Neuen Wert am Ende hinzufügen  
  
 // Löschen list.pop\_back()/ list.pop\_front()  
  
 for(auto i = list.begin(); i != list.end(); i++){  
 std::cout << \*i << std::endl; // 6 27 27 27 27 27 10  
 }  
  
 list.unique(); // Entfernt alle Duplikate  
  
 std::cout << std::endl;  
  
 for(auto i = list.begin(); i != list.end(); i++){  
 std::cout << \*i << std::endl; // 6 27 10  
 }  
  
 list.sort(); // Liste sortieren  
  
 std::cout << std::endl;  
  
 for(auto i = list.begin(); i != list.end(); i++){  
 std::cout << \*i << std::endl; // 6 10 27  
 }  
  
 std::cout << std::endl;  
  
 for(auto i = list.rbegin(); i != list.rend(); i++){ // reverse Iterator  
 std::cout << \*i << std::endl; // 27 10 6  
 // Möglichkeit um Liste mittendrin zu bearbeiten:  
 if(\*i == 27){  
 // Code...  
 }  
 }  
  
 list.remove(10);  
  
 std::cout << std::endl;  
  
 for(auto i = list.begin(); i != list.end(); i++){  
 std::cout << \*i << std::endl; // 6 27  
 }  
  
 std::forward\_list<int> singleLinkedList; // Jedes Element zeigt auf das darauffolgende Element, man kann nur von vorne iterieren  
  
 singleLinkedList.assign(5, 17); // 5 Werte mit dem Wert 17  
 singleLinkedList.push\_front(10); // Eine 10 an den Anfang  
 singleLinkedList.push\_front(20); // Eine 20 an den Anfang  
  
 std::cout << std::endl;  
  
 for(auto i = singleLinkedList.begin(); i != singleLinkedList.end(); i++){  
 std::cout << \*i << std::endl; // 20 10 17 17 17 17 17  
 }  
  
 system("pause");  
  
 return 0;  
}

34.3 Iterator

Iteratoren können durch Container iterieren oder auf spezielle Elemente in einem Container zugreifen. Mit ‚++‘ oder ‚--‘ kann man z.B. auf das nächste Element bzw. das vorherige Element zugreifen. Damit der Iterator auf den Anfang bzw. das Ende eines Containers zeigt kann man ‚begin()‘ bzw. ‚end()‘ benutzen. In diesem Beispielcode iteriert ein Iterator über eine Liste:



(Für ein Zugriff auf so viele Namespaces sollte man außerdem am besten ‚auto‘ verwenden, so kann man auch einfach den Container wechseln und trotzdem noch drüber iterieren ohne etwas anders zu ändern.)

34.4 deque

Deque bietet schnellen Zugriff auf jedes Element und effizientes Einfügen und Löschen von Elementen von hinten und von Vorne des Containers.

Intern in einem deque zeigen verschiedene Arrays aufeinandern. Es ist sozusagen eine Liste, die aus Vektoren besteht.

#include <iostream>  
#include <deque>  
  
int main(){  
  
 // Arrays die jeweils auf das nächste Array zeigen  
 std::deque<int> deque; // Elemente referenzieren einander nicht wie bei einer Liste, kein zusammenhängender Block wie bei einem Vektor  
  
 deque.push\_back(9);  
 deque.push\_back(4);  
 deque.push\_front(42);  
  
 while(!deque.empty()){  
 std::cout << deque.front() << std::endl; // 42 9 4  
 deque.pop\_front();  
 }  
  
 deque.push\_back(9);  
 deque.push\_back(4);  
 deque.push\_front(42);  
  
 std::cout << std::endl;  
  
 while(!deque.empty()){  
 std::cout << deque.back() << std::endl; // 4 9 42  
 deque.pop\_back();  
 }  
  
 system("pause");  
  
 return 0;  
}

34.5 stack

Beim stack ist das, was man zuletzt mit ‚push()‘ reingetan hat das letzte Element. Packt man jetzt also z.B. wie im Beispielcode als letztes Element eine 4 rein, so gibt stack.top() eine 4 zurück.



34.6.1 queue

Eine queue ist das genaue Gegenteil zum stack, da hier das ‚älteste‘ Element zuerst zurückgegeben wird. Packt man jetzt also z.B. wie im Beispielcode als erstes Element eine 17, so wird beim Aufruf von queue.front() auch das erste Element zurückgegeben.



34.6.2 priority\_queue

Das Element(Objekt) mit der höchsten Priorität wird zurückgegeben.

#include <iostream>  
#include <queue>  
  
class Person{  
private:  
 int priority; // Priorität  
public:  
 std::string name;  
 Person(int priority, std::string name) : priority(priority), name(name){  
 // Code...  
 }  
 friend bool operator<(const Person &p1, const Person &p2){ // Operator überladen damit C++ weiß wie wir das Objekt ordnen wollen  
 return p1.priority < p2.priority;  
 }  
};  
  
int main(){  
  
 std::priority\_queue<Person> priorityQueue;  
  
 priorityQueue.push(Person(12, "Test1"));  
 priorityQueue.push(Person(2, "Test2"));  
 priorityQueue.push(Person(4, "Test3"));  
 priorityQueue.push(Person(20, "Test4"));  
  
 std::cout << priorityQueue.top().name << std::endl;  
  
 system("pause");  
  
 return 0;  
}

Eine priority\_queue ordnet Objekte nach ihrer Priorität. Diese Priorität kann man bei eigenen Objekten bzw. Klassen bestimmen, indem man den ‚<‘ Operator überlädt. In diesem Beispielcode wird festgelegt, dass der mit der höchsten Priority ganz oben sein soll, in dem Beispiel ist ‚Test4‘ also ganz oben

34.7.1 set

Elemente in Sets werden immer sortiert. Außerdem darf kein Element doppelt vorhanden sein. Versucht man jedoch trotzdem ein Element doppelt hinzuzufügen, passiert nichts. Auch kann man angeben, wie das Set sortiert werden soll.

#include <iostream>  
#include <set>  
#include <functional>  
  
int main(){  
  
 std::set<int> set;  
  
 set.insert(5);  
 set.insert(72);  
 set.insert(17);  
 set.insert(18);  
 set.insert(5);  
  
 for(auto i = set.begin(); i != set.end(); i++){  
 std::cout << \*i << std::endl; // 5 17 18 72  
 }  
  
 std::set<int, std::greater<int>> setGreater;  
  
 setGreater.insert(5);  
 setGreater.insert(72);  
 setGreater.insert(17);  
 setGreater.insert(18);  
 setGreater.insert(5);  
  
 std::cout << std::endl;  
  
 for(auto i = setGreater.begin(); i != setGreater.end(); i++){  
 std::cout << \*i << std::endl; // 72 18 17 5  
 }  
  
 system("pause");  
  
 return 0;  
}

In dem zweiten set z.B. wird mit ‚std::greater<int>‘ gesagt, dass nach dem größer als Zeichen sortiert werden soll, die größten Zahlen also ‚vorne‘ sind.

34.7.2 multiset

Das multiset ist dem set sehr ähnlich, mit der einzigen Ausnahme, dass hier doppelte Elemente erlaubt sind. 

34.8.1 map

Eine map besteht in C++ aus pairs. Diese stellen den Key und das Value dar. Wie in Java werden die pairs auch hier nach ihren Keys sortiert, wie man im Beispielcode erkennen kann. 

Außerdem kann man erkennen, dass man beim iterieren oder beim Aufrufen der find() Methode das was man zurückbekommt zuerst dereferenzieren muss und dann kann man mit ‚second‘ auf das Value zugreifen. ‚first‘ würde den Key zurückgeben.

34.8.2 multimap

Multimaps können, ähnlich wie bei sets und multisets, mehrere Pairs mit gleichen Keys speichern.



In dem Beispiel gibt es in der multimap z.B. zwei Pairs mit dem selben Key. Um alle Values mit einem Key x herauszufinden kann man über das return value von der ‚equal\_range()‘ Funktion iterieren und sich ‚second‘ ausgeben lassen. In dem Fall von dem Beispiel wird ‚Test 2.1‘ und ‚Test 2.2‘ auf der Konsole ausgegeben.

35.1 hash

Objekte kann man (meistesn) über ein Hash identifizieren, indem man eine Hashfunktion schreibt, die einen Hash generiert, welches bei gleichen Attributen gleich ist. Um zu entscheiden, wie der Hash für ein eigenes Objekt erstellt werden soll, muss man im std namespace den ‚()‘ Operator überschreiben und einen Wert zurückgeben, der den Hash Wert repräsentiert. Nun kann man jedem Objekt einen besonderen Hash zuweisen.



35.2 unordered\_map, unordered\_multimap, unordered\_set, unordered\_multiset

Die unordered Varianten der Maps und Sets sind optimierte Varianten. Um die unordered Varianten der Maps und Sets benutzen zu können benötigt man eine Hash Funktion, die die Hashes für Objekte bildet und man muss den ‚==‘ Operator in der Klasse überschreiben.

Intern haben die unordered Varianten mehrere ‚Buckets‘. Diese kann man sich wie eigene Container vorstellen. Sie können mehrere Objekte vom gleichen Hash halten.

Am Anfang kann so eine unordered Variante z.B. 5 Buckets haben. Jedoch ist ein Hash viel größer als 5, daher wird der Hash % die Anzahl der Buckets gerechnet und dann in den richtigen Bucket gepackt. Bei einer Hash Kollision oder dem gleichen Ergebnis bei der % Operation kann es passieren, dass (eigentlich) unterschiedliche Objekte in demselben Bucket landen.

Ein Nachteil der unordered Varianten ist jedoch, dass sie langsamer beim Iterieren sind. Daher sollte man zum Iterieren doch eher die normalen Varianten der Maps und Sets nutzen. Außerdem ist die Reihenfolge beim Iterieren nicht festgelegt, weil die Buckets immer umsortiert werden müssen, was dazu führt, dass sie in jeder Iteration anders ist bzw. sein kann. Ein weiterer Nachteil ist, dass man die Keys nicht verändern darf, da man dann einen ‚fehlerhaften‘ Hash hat, der also zu keinem Value passt.

#include <iostream>  
#include <unordered\_map>  
#include <string>  
  
class Car {  
public:  
 int ps;  
 int baujahr;  
 bool spoiler;  
  
 Car(int ps, int baujahr, bool spoiler)  
 : ps(ps), baujahr(baujahr), spoiler(spoiler){  
  
 }  
 friend bool operator==(const Car &c1, const Car &c2){ // Wichtig für unordered Varianten  
 return c1.ps == c2.ps && c1.baujahr == c2.baujahr && c1.spoiler && c2.spoiler;  
 }  
};  
  
namespace std{  
 template<> struct hash<Car>{  
 size\_t operator()(const Car &car)const{  
 hash<int> intHash;  
 hash<bool> boolHash;  
  
 return intHash(car.ps) + intHash(car.baujahr) \* 7 + boolHash(car.spoiler);  
 }  
 };  
}  
  
int main(){  
  
 std::unordered\_map<Car, std::string> owners;  
  
 Car c(500, 2012, true);  
  
 owners.insert(std::pair<Car, std::string>(c, "Peter"));  
  
 std::cout << owners.find(c)->second << std::endl;  
  
 system("pause");  
  
 return 0;  
}