PROGRAMMIEREN II

DHBW Stuttgart Campus Horb INF2018

AGENDA

- Richtig casten
- Templates
- Binärdatenformate

CASTS

- C++ ist eine stark typisierte Sprache
- Manchmal ist es jedoch notwendig, dass Typen umkonvertiert werden
- Dies geschieht manchmal implizit und manchmal explizit

IMPLIZITER CAST

```cpp
#include <iostream>

int main()
{
    const char *c_str = "This is a string";

    std::string cpp_str = c_str;

    std::cout << cpp_str << std::endl;
}
```

EXPLIZITER (C) CAST

```c
#include <iostream>
int main()
{
    double am_double = 3.14;
    int casted_int = (int)am_double;

    printf("%f,%d\n", am_double, casted_int);
}
```

STATIC CAST

- Sicherster Cast
- Verändert die Daten
- Kann nur ausgeführt werden, wenn eine Konvertierungsregel existiert
  - Überladener Cast Operator
  - Konstruktor mit entsprechendem Argument
  - Alle Primitiven und Zeiger (Informationsverlust!)

STATIC CAST BEISPIEL

```cpp
#include <iostream>
struct Auto
{
    std::string brand;
    int power;
    Auto(int p) : power(p), brand("Unknown")
    {
    }
    void print()
    {
        printf("Brand: %s; Power: %d\n", brand.c_str(), power);
    }
};
int main()
{
    int magic_power = 286;
    Auto car = static_cast<Auto>(magic_power);
    car.print();
}
```

CONST CAST

- Hebt die „Konstantheit“ einer Variable auf
- Die Variable sollte dennoch nicht verwendet werden
- Anwedungszweck: Kompatibilität zu altem Code herstellen, indem const Methoden nicht als solche deklariert sind

DYNAMIC CAST

- Wandelt einen Zeiger auf eine Basisklasse auf einen Zeiger einer spezialisierten Klasse um

- Klasse muss Polymorph sein (also mindestens eine virtuelle Methode, Destruktor… besitzen)

- Einziger Cast der zur Laufzeit ausgeführt wird (siehe Late Binding)

- Bei Misserfolg wird nullptr zurückgegeben

DYNAMIC CAST BEISPIEL

```
#include <iostream>
class Auto
{
public:
    virtual ~Auto() {}
};
class Sportwagen : public Auto{};
int main()
{
    // deshalb nicht namespace std
    Auto *auto_ptr = new Auto;
    Auto *sport_wagen = new Sportwagen;

    if(dynamic_cast<Sportwagen*>(sport_wagen)){
        printf("Woop Woop\n");
    }
}
```

REINTERPRET CAST

- Der schlimmste aller casts
- Es gibt keine Sicherheit und keine Prüfungen, das Feld wird einfach Bitweise neu interpretiert

REINTERPRET CAST

```
#include <iostream>
int main()
{
    auto array = new char[sizeof(int32_t)];
    int num = 187;
    memcpy(array, &num, sizeof(int32_t));
    int32_t maybe_num = *reinterpret_cast<int32_t*>(array);
    if (maybe_num == num)
    {
        printf("Woop Woop");
    }
}
```

AUFGABE FÜR ZWISCHENDURCH

- Schreibe ein Programm das eine eingegebene Jahreszahl in die römische Schreibweise setzt

	I	=	1
	V	=	5
	X	=	10
	L	=	50
	C	=	100
	D	=	500
	M	=	1000

ÜBERLADENE FUNKTIONEN

```
// overloaded functions
#include <iostream>
using namespace std;

int sum (int a, int b)
{
  return a+b;
}

double sum (double a, double b)
{
  return a+b;
}

int main ()
{
  cout << sum (10,20) << '\n';
  cout << sum (1.0,1.5) << '\n';
  return 0;
}
```

ÜBERLADENE FUNKTIONEN ERKLÄRUNG

- Zwei identische Funktionen bis auf die Typ-Parameter
- Für diesen Fall hat C++ (Funktions-)Templates eingeführt

TEMPLATE FUNCTION

```
#include <iostream>
template <typename myType>
myType sum(myType a, myType b, myType c)
{
    return a + b + c;
}

int main()
{
    // template type is derived from input parameters
    auto result = sum(1, 5, 3);
    printf("Result: %i\n",result);
    auto double_result = sum(0.444,0.69,4213.22);
    printf("Result: %f \n",double_result);
}
```

NOCH EIN TEMPLATE

```
template <typename type>
size_t write_binary(std::ostream &out, type &arg)
{
    out.write((char*)&arg, sizeof(type));
    return sizeof(type);
}
```

ETWAS KOMPLIZIERTERES TEMPLATE

```
#include <functional>

template <typename... Args>
void times(int i, std::function<void(Args...)> f, Args... args)
{
    for (int c = 0; c < i; ++c)
        f(args...);
}
void some_func(int a, int b)Ï
{
    printf("%d+%d=%d\n", a, b, a + b);
}
int main()
{
    std::function<void(int, int)> f = some_func;
    times(5, f, 1, 2);
}
```

TEMPLATE DEKLARATION

- Eingeleitet durch Keyword template
- Die Template Paremeter werden als typename oder class (gleichbedeutend) beschrieben
- Es können auch „normale“ Typen als Template Parameter verwendet werden
-

BEISPIEL TEMPLATE

```cpp
#include <fstream>
#include <exception>
template <typename typ1>
typ1 readFromStream(std::ifstream &stream, char *dest)
{
    if (stream.is_open())
    {
        stream.read(dest, sizeof(typ1));
        typ1 *ptr = reinterpret_cast<typ1 *>(dest);
        return std::move(*ptr);
    }
    else
    {
        std::runtime_error("Could not read from file");
    }
}
```

PROGRAMM-CODE

```
int main()
{
    char *dest = new char[1024];
    try
    {
        std::ifstream stream("charakter.d2s");
        auto header = readFromStream<long>(stream, dest);
        printf("The correct header is: AA55AA55. File starts with %X\n“,
header);
    }
    catch (std::exception &ex)
    {
        printf("An error occured while reading the file: %s", ex.what());
        return -1;
    }
}
```

WIE FUNKTIONIERT DAS HIER?

```
template<typename From, typename To>
union union_cast {
    From from;
    To   to;

    union_cast(From from)
        :from(from) { }

    To getTo() const { return to; }
};
```

DISKUSSION

- Worauf sollte man bei einem Binärformat achten?

DISKUSSIONSRUNDE LETZTES MAL

- Auf was muss man bei einem Binär-Format achten?
  - Bei Arrays mit variablen Längen zuerst die Länge angeben
  - Version am Anfang
  - App Identifier
  - Fixe Start Bytes
  - Stop Sequenz oder Länge der Nachricht am Anfang mit angeben
  - Checksumme
  - Bei Header+Body Aufteilung möglichst den Header mit Fixer Länge und Info über Gesamtlänge der Nachricht
  - Big || Little Endian
  - Signed oder Unsigned

NÄCHSTE AUFGABE

- Auslesen der folgenden Informationen aus dem Diablo 2 Charakter im Repository
- Das Dateiformat ist in der eingecheckten PDF beschrieben
- Finde die folgenden Informationen über den Helden hinaus und speichere sie in einer entsprechenden Struktur / Klasse
    - Name
    - Status (lebendig, hardcore)
    - Titel
    - Klasse
    - Level
- Tipp: Mit ifstream::ignore können bytes übersprungen werden