PROGRAMMIEREN II

DHBW Stuttgart Campus Horb INF2017

GENERIERUNG EINES MINESWEEPER FELDS

- Generiere ein Feld mit 16x16 Feldgröße und 99 Minen

- Fülle die Felder drumherum aus, basierend darauf wie viele Mienen um Sie herum sind

- Gib das Ergebnis am Ende aus

- Bonus: Implementiere die Spiellogik :-)

AGENDA

- Quiz
- Smart Pointer
- Exceptions
- Lambdas
- Noch mehr Aufgaben

WAS SAGT DAS AUS?

```
x = (y < 0) ? 10 : 20;
```

WAS SAGT DAS AUS?

```
(a == 0 ? a : b) = 1;
```

LVALUE :-)

```
if (a == 0)
        a = 1;
else
        b = 1;
```

C++ URLS

```
void foo() {
    http://stackoverflow.com/
    int bar = 4;

    ...
}
```

GOTO LABELS :)

```c
int do_something()
{
    printf("Complex stuff\n");
    return 0;
}
int do_something_else()
{
    printf("Complex stuff, too\n");
    return 0;
}
int main()
{
    int err = do_something();
    if (err)
    {
        goto error;
    }
    err = do_something_else();
    if (err)
    {
        goto error;
    }
    printf("all went well!");
error:
    return -1; // exit in error case
    return 0x0; // exit program normally
}
```

WAS BEDEUTET VIRTUAL?

```
#include <iostream>
class Profile
{
  public:
    virtual std::string &getName() = 0;
    virtual char getGender() = 0;
    virtual int getAge() = 0;
};
int main()
{
    return 0x00;
}
```

WAS BEDEUTET DAS =0

```
#include <iostream>
class Profile
{
  public:
    virtual std::string &getName() = 0;
    virtual char getGender() = 0;
    virtual int getAge() = 0;
};
int main()
{
    return 0x00;
}
```

WIE HEIßT SO EIN CONSTRUCT IN JAVA?

```
#include <iostream>
class Profile
{
  public:
    virtual std::string &getName() = 0;
    virtual char getGender() = 0;
    virtual int getAge() = 0;
};
int main()
{
    return 0x00;
}
```

WOZU DAS OVERRIDE?

```
class Auto
{
public:
    void virtual gibGas()
    {
        printf("Wrumm Wrumm\n");
    }
};
class Sportwagen : public Auto
{
public:
    void virtual gibGas() override
    {
        printf("Sportlich Wrumm Wrumm\n");
    }
};
```

SMART POINTER

```
void SomeMethod()
{
   ClassA *a = new ClassA;
   SomeOtherMethod();
   delete a;
}
```

Gibt es ein Problem mit diesem Code?

SMART POINTER ERKLÄRUNG

- Smart Pointer wrappen normale Pointer, um das Handling mit ihnen weniger Fehleranfällig zu machen

- Sie sollten in der Regel vor normalen Pointern bevorzugt werden

- Existieren in verschiedenen Arten

SMART POINTER (ABSTRAKTES) BEISPIEL

```
SomeSmartPtr<MyObject> ptr(new MyObject());
ptr->DoSomething(); // Use the object in some way.
```

SMART POINTER ARTEN

- Kommen ursprünglich aus Boost, sind aber mit C++11 in den Standard übergegangen
- Es gibt noch den bösen Bruder und nicht besonders smarten std::auto_ptr aus C++98
- Arten von Smart Pointern:
  - Unique-Pointer
  - Shared-Pointer
  - Weak-Pointer (Gefährlich)

UNIQUE POINTER

```
void f()
{
    {
        std::unique_ptr<MyObject> ptr(new MyObject());
        ptr->DoSomethingUseful();
    }
}
```

- unique_ptr kann nicht kopiert werden
- Nützlich wenn man ein Objekt nur in einem Scope braucht, oder seine Lebenszeit als Member eines anderen Objekts beschreibt

SHARED_PTR

```
void f()
{
    typedef std::shared_ptr<MyObject> MyObjectPtr; // nice short alias
    MyObjectPtr p1; // Empty

    {
        MyObjectPtr p2(new MyObject());
        // There is now one "reference" to the created object
        p1 = p2; // Copy the pointer.
        // There are now two references to the object.
    } // p2 is destroyed, leaving one reference to the object.
} // p1 is destroyed, leaving a reference count of zero.
  // The object is deleted.
```

- Pointer mit Reference-Counting
- Nützlich, wenn die Lebensdauer des Objekts komplizierter ist
- Gefahren: Dangling Pointers und Circular References

WEAK_PTR

- std::weak_ptr ist eine Referenz auf ein Objekt, welche nicht mit ins Reference Counting eingeht
- Wenn man ein Konstrukt hat, wo man diesen Pointer braucht, sollte man seinen Programmentwurf wirklich hinterfragen

ERZEUGUNGSVARIANTEN

- std::make_shared<Class>(Construktur Args)
- std::make_unique<Class>(Construktor Args)
- std::shared_ptr<Class>(new Class(Constructor…))
- std::unique_ptr<Class>(new Class(Constructor…))

SMART POINTER BEISPIEL

```cpp
#include <memory>

typedef struct my_struct
{
    my_struct(int a, int b) : x(a), y(b) {}
    int x;
    int y;
} my_struct_t;

typedef std::shared_ptr<my_struct_t> my_struct_ptr;

int main()
{
    my_struct_ptr smart_ptr = std::make_shared<my_struct_t>(187, 4711);
    return 0x00;
}
```

SMART POINTER ZUSAMMENFASSUNG

- Klassische Pointer und „new“ Aufrufe sind eher oldschool und kommen in modernem C++ Code nicht mehr (so oft) vor

- Trotz Reference-Counting keine Garbage Collection —> Mehr Kontrolle, Performance Vorteile

AUTO_PTR

- Niemals benutzen
- Wie Unique_ptr nur dass er kopiert werden kann und dabei die Ownership auf das Objekt übergibt
- Der alte Unique_Ptr wird ungültig ohne es zu merken

DAS PROBLEM MIT GOTO

```c
int main()
{
    int err = do_something();
    if (err)
    {
        goto error;
    }
    err = do_something_else();
    if (err)
    {
        goto error;
    }
    printf("all went well!");
error:
    return -1; // exit in error case
    return 0x0; // exit program normally
}
```

DAS GOTO PROBLEM

- Eigentlich ist nicht goto das Problem
- Es führt nur meistens zu Spagetti Code und schlechter Lesbarkeit
- Es gibt keinen Grund mehr in OOP für dieses Pattern, da diese Jumps besser mit Exceptions umgesetzt werden können

MERKSATZ

- „Wenn Sie goto in Ihrem C++ Code verwenden, werden Ihre Kollegen Sie ähnlich ungläubig anschauen, wie wenn Sie Ihr Schnitzel in der Kantine mit der Streitaxt zerteilen“ - Unknown

EXCEPTIONS

- Zu deutsch: Ausnahme
- Bietet eine Möglichkeit den Programmfluss durch Kontrollsprünge zu verändern
- Die Kontrolle wird an Funktionen gegeben, die man Handler nennt

EXCEPTION BEISPIEL

```cpp
#include <iostream>

double divide(double a, double b)
{
    if (b == 0)
    {
        throw - 1;
    }
    return a / b;
}

int main()
{
    try
    {
        auto x = divide(4, 2);
        printf("Divided %f through %f result was: %f\n", 4.0, 2.0, x);
        auto y = divide(69.69, 0);
        printf("I am not sure if this gets printed\n");
        return 0;
    }
    catch (int ex)
    {
        printf("Caught exception!\n");
        return ex;
    }
}
```

EXCEPTION BEISPIEL ERKENNTNISSE

- Schlüsselwort try gibt den überwachten Block an
- Schlüsselwort throw wirft eine entsprechende Ausnahme
- Schlüsselwort catch leitet eine Handler-Funktion ein
- Exceptions führen zu einem geregeltem Stack Unwinding
- Das schlimmste: Jeder Typ kann als Exception geworfen / gefangen werden

DAS ÜBEL!

```
throw -1;
```

DIE LÖSUNG

- Innerhalb der C++ Standardlib erben alle geworfenen Exceptions von std::exception

- Diese Regel sollte man innerhalb seines Programms auch befolgen

- Überschreiben von exception::what , um die Fehlerursache anzugeben

CUSTOM EXCEPTION

```cpp
#include <iostream>
#include <exception>
using namespace std;
class too_young_exception : public exception
{
    virtual const char *what() const throw() override
    {
        return "The person you wanted to date is not an adult!";
    }
};
int main()
{
    try
    {
        printf("How old are you? - %i\n", 16);
        too_young_exception ex;
        throw ex;
    }
    catch (exception &e)
    {
        cout << e.what() << '\n';
    }
    return 0;
}
```

EXCEPTIONS DER STANDARDLIB

- std::bad_alloc
- std::bad_cast
- std::bad_exception
- std::typeid
- std::bad_function_call
- std::bad_weak_ptr

FÜR DIE VERERBUNG VORGESEHENE TYPEN

- std:: logic_error, wenn man ein interner Logikfehler festgestellt wird

- std::runtime_error, wenn man einen Fehler zur Laufzeit feststellt, der eher technisch ist (ungültige Pointer, fehlgeschlagene Lesevorgänge, …)

REIHENFOLGE VON CATCH BLÖCKEN

- Die Reihenfolge von Catch-Blöcken ist sehr wichtig

- Der erste Catch-Block der zutrifft wird ausgelöst, alle anderen werden ignoriert

- Folglich muss der „Elipsen“-Catch-Block (…) der letzte in der Reihe sein

MÖGLICHKEITEN FÜR EINEN CATCH-HANDLER

- Genau passender Typ
- Typ der durch implizierte Konvertierung passt
- Referenz auf einen passenden Typen
- (Referenz auf) Klasse / Struktur die auf eine base-class passt
- Elipse (…) passt auf alles

MULTIPLE CATCH BLÖCKE

```
// ...
try
{
    // ...
}
catch( ... )
{
    // Handle exception here.
}
// Error: the next two handlers are never examined.
catch( const char * str )
{
    cout << "Caught exception: " << str << endl;
}
catch( CExcptClass E )
{
    // Handle CExcptClass exception here.
}
```

ZERO COST EXCEPTIONS

- Die meisten C++ Compiler implementieren ein Pattern, das dafür sorgt, dass ein try-Block ohne Performance-Verlust durchlaufen werden kann

- Damit ist ein Try{}catch{} block im OK-Fall schneller als der return Type check (if(error))

- Exception Fall dafür deutlich teurer —> Merken

NOEXCEPT KEYWORD

```
void f() noexcept; // the function f() does not throw
```

```
void (*fp)() noexcept(false); // fp points to a function that may throw
```

```
void g(void pfa() noexcept);  // g takes a pointer to function that doesn't throw
```

NOEXEPT KEYWORD ERKLÄRUNG

- Jede Funktion in CPP kann einen von zwei Zuständen haben:
  - a) Does not throw
  - b) May throw
- NOEXCEPT(expression) gibt an, ob eine Funktion Status a) oder b) hat
- Wenn expression = true, dann wirft der Aufruf der Funktion garantiert keine Exception, ansonsten wirft sie eventuell eine

WENN'S DANN DOCH KNALLT

```
// whether foo is declared noexcept depends on if the expression
// T() will throw any exceptions
template <class T>
  void foo() noexcept(noexcept(T())) {}

void bar() noexcept(true) {}
void baz() noexcept { throw 42; }  // noexcept is the same as noexcept(true)

int main()
{
    foo<int>();  // noexcept(noexcept(int())) => noexcept(true), so this is fine

    bar();  // fine
    baz();  // compiles, but at runtime this calls std::terminate
}
```

NOEXCEPT ALS OPERATOR

```
noexcept(may_throw()) —> returns false
noexcept(no_throw()) —> returns true
```

STD::TERMINATE

- Ruft den registrierten std::terminate_handler auf

- Standardmäßig std::abort

STD::ABORT

- Beendet das Programm mit SIGABRT
- Wichtig für euch: Cleanup Funktionen werden nicht unbedingt aufgerufen
- Es gibt kaum einen schlechteren Weg ein Programm zu beenden

SIGNALE

Constant Explanation

SIGTERM termination request, sent to the program

SIGSEGV invalid memory access (segmentation fault)

SIGINT external interrupt, usually initiated by the user

SIGILL invalid program image, such as invalid instruction

SIGABRT abnormal termination condition, as is e.g. initiated by std::abort()

SIGFPE erroneous arithmetic operation such as divide by zero

SIGNALE BEISPIELE

```cpp
#include <csignal>
#include <iostream>

namespace
{
  volatile std::sig_atomic_t gSignalStatus;
}

void signal_handler(int signal)
{
  gSignalStatus = signal;
}

int main()
{
  // Install a signal handler
  std::signal(SIGINT, signal_handler);

  std::cout << "SignalValue: " << gSignalStatus << '\n';
  std::cout << "Sending signal " << SIGINT << '\n';
  std::raise(SIGINT);
  std::cout << "SignalValue: " << gSignalStatus << '\n';
}
```

UNDEFINED BEHAVIOR

- In einem Signal Handler darf man fast nur noch ein paar letzte Aufräum- oder Loggingaktionen machen
- Das Verhalten ist undefiniert für
  - Allokationen
  - Casts
  - Library calls
  - …
  - Volle Liste: https://en.cppreference.com/w/cpp/utility/program/signal
- Noch dazu: Eher C Style

KLEINE PROGRAMMIERAUFGABE

- Schreibe ein Programm das (ausnahmsweise) absichtlich SIGSEGV auslöst und im Handler den Programmierer beleidigt (ca. 15 Minuten)

FUNCTION

```
#include <functional>
#include <iostream>

int do_something_smart(int x, int y)
{
    return x + y;
}

int main()
{
    std::function<int(int,int)> smart = do_something_smart;
}
```

FUNCTION SYNTAX

- std::function<return_type(arg_type,arg_type…)

LAMBDAS

```
#include <iostream>

int main(){
  auto x = [](){std::cout << "Hello from lambda" << std::endl;};
  x();
  return 0;
}
```

LAMBDAS

Definiert ein Closure: ein unbenanntes (anonymes) Funktions-Literal, welches Variablen über deren Erstellungskontext hinaus einfangen und "am Leben erhalten" kann.

LAMBDAS HEIßT:

- Eine anonyme Funktion die in eine Variable gespeichert werden kann übergeben und anderswo aufgeführt werden kann

LAMBDA SYNTAX (EINFACH)

[capture **]** **(** params **)** **{** body **}**

CAPTURE

spezifiziert, welche Symbole erfasst und für den Funktionskörper sichtbar gemacht werden.

Eine Liste der Symbole kann wie folgt übergeben werden:

- **[a,&b]**, a wird kopiert und b als Referenz erfasst.
- **[this]** erfasst den **this**-Zeiger als Kopie.
- **[&]** erfasst alle im Funktionskörper verwendeten Symbole als Referenzen.
- **[=]** erfasst alle im Funktionskörper verwendeten Symbole als Kopien.
- **[]** es wird nichts erfasst.

PARAMETER

- Genau wie bei „normalen“ Funktionen

BODY

- Genau wie bei „normalen“ Funktionen

SUPER SIMPLE LAMBDA

```
#include <iostream>

int main(){
  auto x = [](){std::cout << "Hello from lambda" << std::endl;};
  x();
  return 0;
}
```

HOW TO (NOT) USE A LAMBDA

```cpp
#include <iostream>
#include <functional>
void sneaky_call(std::function<void()> method)
{
  try
  {
    method();
  }
  catch (...)
  {
    //... silently take it
  }
}
int main()
{
  auto x = []() { std::cout << "Hello from lambda" << std::endl; };
  sneaky_call(x);
  return 0;
}
```

TYPISCHE ANWENDUNGSGEBIETE

- Callbacks (z.B. nach asynchronen Lese- oder Schreibeoperationen)
- Templateprogrammierung
- Strukturierung von Code

AUFGABE FÜR DAS NÄCHSTE MAL

- Lies die csv Datei im Repository ein
- Baue intelligente Datenstrukturen, um die Daten im Speicher zu halten
- Das System soll die folgenden Abfragen unterstützen und per Menü anbieten
  - Total profit pro Region, Country, Item Type oder Sales Channel
  - Wie oft wurde ein Item Type in den Regionen verkauft? Absolut und prozentual
  - Welcher Item Type ist der beliebteste in Land X?
  - Wie oft wurde in Land X Online oder Offline bestellt (insgesamt, prozentual)?
  - Weitere nach Langeweile ;-)
- Nutzt erstmal die Sample Datei mit 100 Einträgen und wechselt dann auf die mit 15.000.000 Einträgen