



IMS  
Institut für Mikroelektronische Systeme  
Leibniz Universität Hannover



Leibniz  
Universität  
Hannover

# Programmierpraktikum Technische Informatik (C++)



12.5.2022



# Überblick

- Pointer und Memory Management
- Aufgaben zu Pointern
- Exceptions

## Pointer und Speicherverwaltung

MAN, I SUCK AT THIS GAME.  
CAN YOU GIVE ME  
A FEW POINTERS?

I HATE YOU.

0x3A28213A  
0x6339392C,  
0x7363682E.



# Pointer

- Pointer (deutsch: Zeiger) sind Objekte, die Adressen anderer Objekte enthalten
  - Sollten aus C bereits bekannt sein

# Pointer

- Pointer (deutsch: Zeiger) sind Objekte, die Adressen anderer Objekte enthalten
  - Sollten aus C bereits bekannt sein
- Ähnlich zu Referenzen, aber klare Unterscheidung zum Objekt
  - Müssen im Gegensatz zu Referenzen nicht auf ein Objekt zeigen
  - Für Zugriff auf das Objekt ist explizite Dereferenzierung notwendig

# Pointer

- Pointer (deutsch: Zeiger) sind Objekte, die Adressen anderer Objekte enthalten
  - Sollten aus C bereits bekannt sein
- Ähnlich zu Referenzen, aber klare Unterscheidung zum Objekt
  - Müssen im Gegensatz zu Referenzen nicht auf ein Objekt zeigen
  - Für Zugriff auf das Objekt ist explizite Dereferenzierung notwendig
- Pointer werden im Folgenden nur oberflächlich wiederholt. Unklarheiten über grundlegende Konzepte anderweitig nachschlagen
  - <http://cslibrary.stanford.edu/104/>

# Definition von Pointern

```
void increment(int* p) {  
    if(p) {  
        *p = *p + 1;  
    }  
}
```

```
int main() {  
    int x = 5;  
    int* px = &x;  
    increment(px);  
    std::cout<<x<<"\n";  
}
```

## Definition von Pointern

```
void increment(int* p) {
    if(p) {
        *p = *p + 1;
    }
}
```

```
int main() {
    int x = 5;
    int* px = &x;
    increment(px);
    std::cout << x << "\n";
}
```

Variable	Adresse	Wert
x	0x003F008	5

- Der Adressoperator & gibt die Adresse eines Objektes zurück



## Definition von Pointern

```
void increment(int* p) {
    if(p) {
        *p = *p + 1;
    }
}
```

```
int main() {
    int x = 5;
    int* px = &x;
    increment(px);
    std::cout << x << "\n";
}
```

Variable	Adresse	Wert
x	0x003F008	5
px	0x003F010	0x003F008

- *T\** definiert einen Pointer auf *T*
  - *T\*\** definiert einen Pointer auf einen Pointer auf *T*
  - Kann beliebig geschachtelt werden
- Pointer können wie andere Typen per Value oder per Referenz übergeben werden
  - Übergabe als *const*-Referenz unnötig, da Kopien für Pointer billig sind
  - *const*-Definitionen von Pointern können etwas komplex werden...

## Const

- Pointer können `const` sein und/oder auf `const`-Objekte zeigen
  - Adressiertes Objekt modifizierbar, auch wenn Pointer `const` ist
- `const` kann auch nach dem Typ stehen
  - Default, `const` vor dem Typ ist lediglich vereinfachte Schreibweise

# Const

- Pointer können `const` sein und/oder auf `const`-Objekte zeigen
  - Adressiertes Objekt modifizierbar, auch wenn Pointer `const` ist
- `const` kann auch nach dem Typ stehen
  - Default, `const` vor dem Typ ist lediglich vereinfachte Schreibweise
- Regel: `const` steht **nach** dem Typbestandteil, auf den es sich bezieht
  - Typen von hinten nach vorne lesen
  - Ausnahme: `const` am Anfang das Typs behandeln als würde es hinter dem **ersten** Typbestandteil stehen

# Const

- Pointer können **const** sein und/oder auf **const**-Objekte zeigen
  - Adressiertes Objekt modifizierbar, auch wenn Pointer **const** ist
- **const** kann auch nach dem Typ stehen
  - Default, **const** vor dem Typ ist lediglich vereinfachte Schreibweise
- Regel: **const** steht **nach** dem Typbestandteil, auf den es sich bezieht
  - Typen von hinten nach vorne lesen
  - Ausnahme: **const** am Anfang das Typs behandeln als würde es hinter dem **ersten** Typbestandteil stehen

```
int  const *
int*  const
const int*
const int  * const
int  const * const
int * const *
using IntPtr = int*;
const IntPtr*
```

# Const

- Pointer können **const** sein und/oder auf **const**-Objekte zeigen
  - Adressiertes Objekt modifizierbar, auch wenn Pointer **const** ist
- **const** kann auch nach dem Typ stehen
  - Default, **const** vor dem Typ ist lediglich vereinfachte Schreibweise
- Regel: **const** steht **nach** dem Typbestandteil, auf den es sich bezieht
  - Typen von hinten nach vorne lesen
  - Ausnahme: **const** am Anfang das Typs behandeln als würde es hinter dem **ersten** Typbestandteil stehen

```
int  const *  
int*  const  
const int*  
const int  * const  
int  const * const  
int * const *  
using IntPtr = int*;  
const IntPtr*
```

# Const

- Pointer können **const** sein und/oder auf **const**-Objekte zeigen
  - Adressiertes Objekt modifizierbar, auch wenn Pointer **const** ist
- **const** kann auch nach dem Typ stehen
  - Default, **const** vor dem Typ ist lediglich vereinfachte Schreibweise
- Regel: **const** steht **nach** dem Typbestandteil, auf den es sich bezieht
  - Typen von hinten nach vorne lesen
  - Ausnahme: **const** am Anfang das Typs behandeln als würde es hinter dem **ersten** Typbestandteil stehen

```
int const *
int* const
const int*
const int * const
int const * const
int * const *
using IntPtr = int*;
const IntPtr*
```

# Const

- Pointer können **const** sein und/oder auf **const**-Objekte zeigen
  - Adressiertes Objekt modifizierbar, auch wenn Pointer **const** ist
- **const** kann auch nach dem Typ stehen
  - Default, **const** vor dem Typ ist lediglich vereinfachte Schreibweise
- Regel: **const** steht **nach** dem Typbestandteil, auf den es sich bezieht
  - Typen von hinten nach vorne lesen
  - Ausnahme: **const** am Anfang das Typs behandeln als würde es hinter dem **ersten** Typbestandteil stehen

```
int const *
int* const
const int*
const int * const
int const * const
int * const *
using IntPtr = int*;
const IntPtr*
```

Pointer auf konstanten **int**

# Const

- Pointer können **const** sein und/oder auf **const**-Objekte zeigen
  - Adressiertes Objekt modifizierbar, auch wenn Pointer **const** ist
- **const** kann auch nach dem Typ stehen
  - Default, **const** vor dem Typ ist lediglich vereinfachte Schreibweise
- Regel: **const** steht **nach** dem Typbestandteil, auf den es sich bezieht
  - Typen von hinten nach vorne lesen
  - Ausnahme: **const** am Anfang das Typs behandeln als würde es hinter dem **ersten** Typbestandteil stehen

```
int  const *
int* const
const int*
const int  * const
int  const * const
int * const *
using IntPtr = int*;
const IntPtr*
```



# Const

- Pointer können **const** sein und/oder auf **const**-Objekte zeigen
  - Adressiertes Objekt modifizierbar, auch wenn Pointer **const** ist
- **const** kann auch nach dem Typ stehen
  - Default, **const** vor dem Typ ist lediglich vereinfachte Schreibweise
- Regel: **const** steht **nach** dem Typbestandteil, auf den es sich bezieht
  - Typen von hinten nach vorne lesen
  - Ausnahme: **const** am Anfang das Typs behandeln als würde es hinter dem **ersten** Typbestandteil stehen

```
int const *
int* const
const int*
const int * const
int const * const
int * const *
using IntPtr = int*;
const IntPtr*
```

Konstanter Pointer auf **int**

# Const

- Pointer können **const** sein und/oder auf **const**-Objekte zeigen
  - Adressiertes Objekt modifizierbar, auch wenn Pointer **const** ist
- **const** kann auch nach dem Typ stehen
  - Default, **const** vor dem Typ ist lediglich vereinfachte Schreibweise
- Regel: **const** steht **nach** dem Typbestandteil, auf den es sich bezieht
  - Typen von hinten nach vorne lesen
  - Ausnahme: **const** am Anfang das Typs behandeln als würde es hinter dem **ersten** Typbestandteil stehen

```
int  const *
int* const
const int*
const int  * const
int  const * const
int * const *
using IntPtr = int*;
const IntPtr*
```

# Const

- Pointer können **const** sein und/oder auf **const**-Objekte zeigen
  - Adressiertes Objekt modifizierbar, auch wenn Pointer **const** ist
- **const** kann auch nach dem Typ stehen
  - Default, **const** vor dem Typ ist lediglich vereinfachte Schreibweise
- Regel: **const** steht **nach** dem Typbestandteil, auf den es sich bezieht
  - Typen von hinten nach vorne lesen
  - Ausnahme: **const** am Anfang das Typs behandeln als würde es hinter dem **ersten** Typbestandteil stehen

```
int const *
int* const
const int*
const int * const
int const * const
int * const *
using IntPtr = int*;
const IntPtr*
```

Pointer auf konstanten **int**

# Const

- Pointer können **const** sein und/oder auf **const**-Objekte zeigen
  - Adressiertes Objekt modifizierbar, auch wenn Pointer **const** ist
- **const** kann auch nach dem Typ stehen
  - Default, **const** vor dem Typ ist lediglich vereinfachte Schreibweise
- Regel: **const** steht **nach** dem Typbestandteil, auf den es sich bezieht
  - Typen von hinten nach vorne lesen
  - Ausnahme: **const** am Anfang das Typs behandeln als würde es hinter dem **ersten** Typbestandteil stehen

```
int  const *
int*  const
const int*
const int * const
int  const * const
int *  const *
using IntPtr = int*;
const IntPtr*
```

# Const

- Pointer können **const** sein und/oder auf **const**-Objekte zeigen
  - Adressiertes Objekt modifizierbar, auch wenn Pointer **const** ist
- **const** kann auch nach dem Typ stehen
  - Default, **const** vor dem Typ ist lediglich vereinfachte Schreibweise
- Regel: **const** steht **nach** dem Typbestandteil, auf den es sich bezieht
  - Typen von hinten nach vorne lesen
  - Ausnahme: **const** am Anfang das Typs behandeln als würde es hinter dem **ersten** Typbestandteil stehen

```
int const *
int* const
const int*
const int * const
int const * const
int * const *
using IntPtr = int*;
const IntPtr*
```

Konstanter Pointer auf konstanten **int**

# Const

- Pointer können **const** sein und/oder auf **const**-Objekte zeigen
  - Adressiertes Objekt modifizierbar, auch wenn Pointer **const** ist
- **const** kann auch nach dem Typ stehen
  - Default, **const** vor dem Typ ist lediglich vereinfachte Schreibweise
- Regel: **const** steht **nach** dem Typbestandteil, auf den es sich bezieht
  - Typen von hinten nach vorne lesen
  - Ausnahme: **const** am Anfang das Typs behandeln als würde es hinter dem **ersten** Typbestandteil stehen

```
int  const *
int*  const
const int*
const int  * const
int  const * const
int * const *
using IntPtr = int*;
const IntPtr*
```

# Const

- Pointer können **const** sein und/oder auf **const**-Objekte zeigen
  - Adressiertes Objekt modifizierbar, auch wenn Pointer **const** ist
- **const** kann auch nach dem Typ stehen
  - Default, **const** vor dem Typ ist lediglich vereinfachte Schreibweise
- Regel: **const** steht **nach** dem Typbestandteil, auf den es sich bezieht
  - Typen von hinten nach vorne lesen
  - Ausnahme: **const** am Anfang das Typs behandeln als würde es hinter dem **ersten** Typbestandteil stehen

```
int const *
int* const
const int*
const int * const
int const * const
int * const *
using IntPtr = int*;
const IntPtr*
```

Konstanter Pointer auf konstanten **int**

# Const

- Pointer können **const** sein und/oder auf **const**-Objekte zeigen
  - Adressiertes Objekt modifizierbar, auch wenn Pointer **const** ist
- **const** kann auch nach dem Typ stehen
  - Default, **const** vor dem Typ ist lediglich vereinfachte Schreibweise
- Regel: **const** steht **nach** dem Typbestandteil, auf den es sich bezieht
  - Typen von hinten nach vorne lesen
  - Ausnahme: **const** am Anfang des Typs behandeln als würde es hinter dem **ersten** Typbestandteil stehen

```
int const *
int* const
const int*
const int * const
int const * const
int * const *
using IntPtr = int*;
const IntPtr*
```



# Const

- Pointer können **const** sein und/oder auf **const**-Objekte zeigen
  - Adressiertes Objekt modifizierbar, auch wenn Pointer **const** ist
- **const** kann auch nach dem Typ stehen
  - Default, **const** vor dem Typ ist lediglich vereinfachte Schreibweise
- Regel: **const** steht **nach** dem Typbestandteil, auf den es sich bezieht
  - Typen von hinten nach vorne lesen
  - Ausnahme: **const** am Anfang das Typs behandeln als würde es hinter dem **ersten** Typbestandteil stehen

```
int const *  
int* const  
const int*  
const int * const  
int const * const  
int * const *  
using IntPtr = int*;  
const IntPtr*
```

Pointer auf konstanten Pointer auf **int**

# Const

- Pointer können **const** sein und/oder auf **const**-Objekte zeigen
  - Adressiertes Objekt modifizierbar, auch wenn Pointer **const** ist
- **const** kann auch nach dem Typ stehen
  - Default, **const** vor dem Typ ist lediglich vereinfachte Schreibweise
- Regel: **const** steht **nach** dem Typbestandteil, auf den es sich bezieht
  - Typen von hinten nach vorne lesen
  - Ausnahme: **const** am Anfang das Typs behandeln als würde es hinter dem **ersten** Typbestandteil stehen

```
int    const *
int*   const
const int*
const int    * const
int    const * const
int * const *
using IntPtr = int*;
const IntPtr*
```

## Const

- Pointer können **const** sein und/oder auf **const**-Objekte zeigen
  - Adressiertes Objekt modifizierbar, auch wenn Pointer **const** ist
- **const** kann auch nach dem Typ stehen
  - Default, **const** vor dem Typ ist lediglich vereinfachte Schreibweise
- Regel: **const** steht **nach** dem Typbestandteil, auf den es sich bezieht
  - Typen von hinten nach vorne lesen
  - Ausnahme: **const** am Anfang das Typs behandeln als würde es hinter dem **ersten** Typbestandteil stehen

```
int const *
int* const
const int*
const int * const
int const * const
int * const *
using IntPtr = int*;
const IntPtr*
```

Pointer auf konstanten Pointer auf **int**  
(**int \* const \***, **nicht: const int\*\***)

## Definition von Pointern

```
void increment(int* p) {  
    if(p) {  
        *p = *p + 1;  
    }  
}
```

```
int main() {  
    int x = 5;  
    int* px = &x;  
    increment(px);  
    std::cout << x << "\n";  
}
```

- Pointer wie Iteratoren mit *\*var* dereferenzieren
  - Gibt Referenz auf das adressierte Objekt zurück
  - Kann verwendet werden, um den Wert des Objekts auszulesen und/oder zu ändern
- Beispiel kann kürzer als `*p += 1` geschrieben werden
  - Noch kürzer, aber schwerer zu lesen: `++(*p)`

## Definition von Pointern

```
void increment(int* p) {
    if(p) {
        *p = *p + 1;
    }
}
```

```
int main() {
    int x = 5;
    int* px = &x;
    increment(px);
    std::cout << x << "\n";
}
```

- Wichtigster Unterschied zwischen Referenzen und Pointern:  
Pointer müssen nicht auf ein Objekt zeigen

# Nullpointer

- Nullpointer haben speziellen Wert (Null), um zu signalisieren, dass sie nicht auf ein Objekt zeigen

# Nullpointer

- Nullpointer haben speziellen Wert (Null), um zu signalisieren, dass sie nicht auf ein Objekt zeigen
- In C und C++03: Macro NULL zum Erzeugen von Nullpointern
  - `int* foo = NULL;`
  - Probleme mit Overloads, da NULL `int` oder Pointer sein kann
    - Was macht `foo(NULL);`, wenn `int foo(int)` und `int foo(char*)` existieren?

# Nullpointer

- Nullpointer haben speziellen Wert (Null), um zu signalisieren, dass sie nicht auf ein Objekt zeigen
- In C und C++03: Macro NULL zum Erzeugen von Nullpointern
  - `int* foo = NULL;`
  - Probleme mit Overloads, da NULL `int` oder Pointer sein kann
    - Was macht `foo(NULL);`, wenn `int foo(int)` und `int foo(char*)` existieren?
    - Antwort: Ruft `foo(int)` auf



# Nullpointer

- Nullpointer haben speziellen Wert (Null), um zu signalisieren, dass sie nicht auf ein Objekt zeigen
- In C und C++03: Macro NULL zum Erzeugen von Nullpointern
  - `int* foo = NULL;`
  - Probleme mit Overloads, da NULL `int` oder Pointer sein kann
    - Was macht `foo(NULL);`, wenn `int foo(int)` und `int foo(char*)` existieren?
    - Antwort: Ruft `foo(int)` auf
- Seit C++11: `nullptr`
  - Konstante vom Typ `nullptr_t`, konvertiert implizit in alle Pointertypen
  - `int* foo = nullptr;`
  - Keine implizite Konvertierung in Integertypen

# Nullpointer

- Nullpointer haben speziellen Wert (Null), um zu signalisieren, dass sie nicht auf ein Objekt zeigen
- In C und C++03: Macro NULL zum Erzeugen von Nullpointern
  - `int* foo = NULL;`
  - Probleme mit Overloads, da NULL `int` oder Pointer sein kann
    - Was macht `foo(NULL);`, wenn `int foo(int)` und `int foo(char*)` existieren?
    - Antwort: Ruft `foo(int)` auf
- Seit C++11: `nullptr`
  - Konstante vom Typ `nullptr_t`, konvertiert implizit in alle Pointertypen
  - `int* foo = nullptr;`
  - Keine implizite Konvertierung in Integertypen

Immer `nullptr` statt NULL verwenden!

## Definition von Pointern

```
void increment(int* p) {  
    if(p) {  
        *p = *p + 1;  
    }  
}
```

```
int main() {  
    int x = 5;  
    int* px = &x;  
    increment(px);  
    std::cout<<x<<"\n";  
}
```

- Wichtigster Unterschied zwischen Referenzen und Pointern:  
Pointer müssen nicht auf ein Objekt zeigen
- Pointer können als Bedingung verwendet werden
- Boolescher Wert eines Pointers ist **true**, wenn er kein Nullpointer ist
  - **if(p)** überprüft, ob p nicht nullptr ist; **if(!p)** überprüft, ob er gleich nullptr ist.

# Pointerarithmetik

- Bisher entsprechen Pointer im Wesentlichen Objektreferenzen in Sprachen wie Java
- Wesentlicher Unterschied: Möglichkeit Pointerarithmetik auszuführen

## Pointerarithmetik

- Bisher entsprechen Pointer im Wesentlichen Objektreferenzen in Sprachen wie Java
- Wesentlicher Unterschied: Möglichkeit Pointerarithmetik auszuführen
- ++*ptr* schiebt *ptr* ein Objekt weiter
  - Setzt *ptr* auf Adresse des nächsten Objekts, nicht um ein Byte höhere Adresse
  - Differenz in Bytes zwischen alter und neuer Adresse entspricht der Größe des adressierten Typs (`sizeof(T)`)
  - *ptr*++, --*ptr*, *ptr*--, *ptr*+n und *ptr*-n entsprechend

## Pointerarithmetik

- Bisher entsprechen Pointer im Wesentlichen Objektreferenzen in Sprachen wie Java
- Wesentlicher Unterschied: Möglichkeit Pointerarithmetik auszuführen
- `++ptr` schiebt `ptr` ein Objekt weiter
  - Setzt `ptr` auf Adresse des nächsten Objekts, nicht um ein Byte höhere Adresse
  - Differenz in Bytes zwischen alter und neuer Adresse entspricht der Größe des adressierten Typs (`sizeof(T)`)
  - `ptr++`, `--ptr`, `ptr--`, `ptr+n` und `ptr-n` entsprechend
- **Wichtige Einschränkung:** Arithmetik nur legal, wenn der Ergebnispointer im selben Array landet
  - Für `std::vector` und C-Style Arrays (später) muss der Pointer zwischen Start des Arrays und einem Element hinter dem Ende liegen
  - Für andere Objekte in der Regel keine legale Pointerarithmetik

## Pointerarithmetik II

- $ptr1 - ptr2$  gibt den Abstand zweier Pointer an
  - Es gelten dieselben Einschränkungen wie für Inkrement
  - Differenz in Vielfachen von `sizeof(T)`

## Pointerarithmetik II

- $ptr1 - ptr2$  gibt den Abstand zweier Pointer an
  - Es gelten dieselben Einschränkungen wie für Inkrement
  - Differenz in Vielfachen von `sizeof(T)`
- $ptr[n]$  entspricht  $*(ptr + n)$



## Pointerarithmetik II

- $ptr1 - ptr2$  gibt den Abstand zweier Pointer an
  - Es gelten dieselben Einschränkungen wie für Inkrement
  - Differenz in Vielfachen von `sizeof(T)`
- $ptr[n]$  entspricht  $*(ptr + n)$
- Pointer können mit `<`, `<=`, `>`, `>=` verglichen werden
  - Wenn die Pointer zum selben Array gehören

## Pointerarithmetik II

- $ptr1 - ptr2$  gibt den Abstand zweier Pointer an
  - Es gelten dieselben Einschränkungen wie für Inkrement
  - Differenz in Vielfachen von `sizeof(T)`
- $ptr[n]$  entspricht  $*(ptr + n)$
- Pointer können mit `<`, `<=`, `>`, `>=` verglichen werden
  - Wenn die Pointer zum selben Array gehören
- `==` und `!=` als einzige Operationen auch für voneinander unabhängige Pointer

## Pointerarithmetik II

- $ptr1 - ptr2$  gibt den Abstand zweier Pointer an
  - Es gelten dieselben Einschränkungen wie für Inkrement
  - Differenz in Vielfachen von `sizeof(T)`
- $ptr[n]$  entspricht  $*(ptr + n)$
- Pointer können mit `<`, `<=`, `>`, `>=` verglichen werden
  - Wenn die Pointer zum selben Array gehören
- `==` und `!=` als einzige Operationen auch für voneinander unabhängige Pointer
- In der Realität funktionieren arithmetische und Vergleichsoperationen auch für unabhängige Objekte
  - Verwendung führt häufig zu schwierig zu findenden Bugs

# Dynamisch allozierter Speicher

- Bisher nur automatisch allozierte Objekte
  - Werden am Ende des jeweiligen Scopes gelöscht
  - Parameter und lokale Variablen

# Dynamisch allozierter Speicher

- Bisher nur automatisch allozierte Objekte
  - Werden am Ende des jeweiligen Scopes gelöscht
  - Parameter und lokale Variablen
- Container mit zur Compilezeit unbekannter Länge und Struktur (z.B. `std::map`) können auf diese Weise nicht implementiert werden
  - Variablen müssen in C++ zur Compilezeit festgelegte Länge haben

# Dynamisch allozierter Speicher

- Bisher nur automatisch allozierte Objekte
  - Werden am Ende des jeweiligen Scopes gelöscht
  - Parameter und lokale Variablen
- Container mit zur Compilezeit unbekannter Länge und Struktur (z.B. `std::map`) können auf diese Weise nicht implementiert werden
  - Variablen müssen in C++ zur Compilezeit festgelegte Länge haben
- Lösung: Dynamisch allozierter Speicher
  - Allozierter Speicher nicht direkt an Lebenszeit des enthaltenden Scopes gebunden
  - Container wie `std::map` halten üblicherweise Pointer auf dynamisch allozierten Speicher

# Verwendung von dynamisch alloziertem Speicher

```
#include <string>
#include <iostream>
std::string* foo() {
    std::string* result = new std::string;
    return result;
}

int main() {
    std::string* p = foo();
    std::cout<<p<<"\n";
}
```

# Verwendung von dynamisch alloziertem Speicher

```
std::string* foo() {  
    std::string* result = new std::string;  
    return result;  
}
```

```
int main() {  
    std::string* p = foo();  
    std::cout<<p<<"\n";  
}
```



# Verwendung von dynamisch alloziertem Speicher

```
std::string* foo() {  
    std::string* result = new std::string;  
    return result;  
}
```

```
int main() {  
    std::string* p = foo();  
    std::cout<<p<<"\n";  
}
```

- **new** *T* alloziert und defaultkonstruiert ein Objekt vom Typ *T* und gibt einen Pointer auf dieses zurück
  - Mit **new** *T(args)* oder **new** *T{args}* kann nichtdefaultkonstruiertes Objekt erstellt werden
- Die Lebensdauer des allozierten Objektes endet nicht mit dem umschließenden Scope

## Verwendung von dynamisch alloziertem Speicher

```
std::string* foo() {  
    std::string* result = new std::string;  
    return result;  
}
```

```
int main() {  
    std::string* p = foo();  
    std::cout<<p<<"\n";  
}
```

- **new** *T* alloziert und defaultkonstruiert ein Objekt vom Typ *T* und gibt einen Pointer auf dieses zurück
  - Mit **new** *T(args)* oder **new** *T{args}* kann nichtdefaultkonstruiertes Objekt erstellt werden
- Die Lebensdauer des allozierten Objektes endet nicht mit dem umschließenden Scope
- Der Code weist dennoch einen gravierenden Bug auf

## Verwendung von dynamisch alloziertem Speicher

```
std::string* foo() {  
    std::string* result = new std::string;  
    return result;  
}
```

```
int main() {  
    std::string* p = foo();  
    std::cout<<p<<"\n";  
}
```

- Der allozierte String wird niemals wieder freigegeben
  - C++ enthält keinen Garbage Collector, mit `new` allozierter Speicher muss vom Programmierer wieder freigegeben werden
  - Leakt Speicher, kann bei häufigem Vorkommen zu Programmabstürzen aufgrund mangelnden Hauptspeichers führen

## Verwendung von dynamisch alloziertem Speicher

```
std::string* foo() {  
    std::string* result = new std::string;  
    return result;  
}
```

```
int main() {  
    std::string* p = foo();  
    std::cout<<p<<"\n";  
    delete p;  
}
```

- Mit **new** allozierter Speicher **muss** wieder freigegeben werden

# Verwendung von dynamisch alloziertem Speicher

```
std::string* foo() {  
    std::string* result = new std::string;  
    return result;  
}
```

```
int main() {  
    std::string* p = foo();  
    std::cout<<p<<"\n";  
    delete p;  
}
```

- Mit **new** allozierter Speicher **muss** wieder freigegeben werden
- **delete** gibt mit **new** allozierten Speicher wieder frei
  - Gibt dem Objekt die Möglichkeit aufzuräumen
  - Nur für mit **new** allozierten Speicher, nicht für mit `malloc` allozierten Speicher verwendbar
  - **Achtung:** **delete** auf Instanzen benötigt Definition der Klasse, Forward-Deklaration reicht nicht

# Probleme mit manuellem Speichermanagement

- Manuelles Speichermanagement ist extrem anfällig für Bugs
- Freigeben von Speicher kann leicht vergessen oder übersprungen werden
  - Wenn Speicher in einer externen Funktion alloziert wird ohne kenntlich zu machen, dass der Speicher vom Programmierer wieder freigegeben werden muss
  - In Funktionen mit mehreren `return`-Statements
  - In Fällen der Ausnahmebehandlung

## Probleme mit manuellem Speichermanagement

- Manuelles Speichermanagement ist extrem anfällig für Bugs
- Freigeben von Speicher kann leicht vergessen oder übersprungen werden
  - Wenn Speicher in einer externen Funktion alloziert wird ohne kenntlich zu machen, dass der Speicher vom Programmierer wieder freigegeben werden muss
  - In Funktionen mit mehreren `return`-Statements
  - In Fällen der Ausnahmebehandlung
- Freigeben des Speicherbereiches muss kenntlich gemacht werden
  - Dereferenzieren nach einem `delete` ist undefiniertes Verhalten

## Probleme mit manuellem Speichermanagement

- Manuelles Speichermanagement ist extrem anfällig für Bugs
- Freigeben von Speicher kann leicht vergessen oder übersprungen werden
  - Wenn Speicher in einer externen Funktion alloziert wird ohne kenntlich zu machen, dass der Speicher vom Programmierer wieder freigegeben werden muss
  - In Funktionen mit mehreren `return`-Statements
  - In Fällen der Ausnahmebehandlung
- Freigeben des Speicherbereiches muss kenntlich gemacht werden
  - Dereferenzieren nach einem `delete` ist undefiniertes Verhalten
- Speicher darf nur einmal freigegeben werden
  - Aufruf von `delete` auf bereits freigegebenem Speicher ist undefiniertes Verhalten



## RAII

- In C++ wird üblicherweise automatisiertes Ressourcenmanagement verwendet
- Läuft unter dem Begriff RAII (Resource Acquisition is Initialization)
  - Wahrscheinlich das wichtigste C++ Idiom überhaupt
  - und eines der am schlechtesten benannten
  - Korrekte Anwendung von RAII wesentlicher Unterschied zwischen gutem und schlechtem C++ Code

## RAII

- In C++ wird üblicherweise automatisiertes Ressourcenmanagement verwendet
- Läuft unter dem Begriff RAII (Resource Acquisition is Initialization)
  - Wahrscheinlich das wichtigste C++ Idiom überhaupt
  - und eines der am schlechtesten benannten
  - Korrekte Anwendung von RAII wesentlicher Unterschied zwischen gutem und schlechtem C++ Code
- Ressourcen (z.B. dynamisch allozierter Speicher) werden von Objekten gemanaged
  - Lebensdauer dieser Ressourcenmanagementobjekte auf andere Weise gehandhabt
  - Üblicherweise lokale Variablen mit scopegebundener Lebenszeit
  - Alternativ: Rekursive Anwendung von RAII, Objekt wird in ein anderes Managementobjekt geschachtelt

## RAII II

- Objekt wird durch Aufruf eines Konstruktors bei Definition initialisiert
  - Vor dem Konstruktoraufruf existiert keine Möglichkeit, auf das Objekt zuzugreifen
  - Keine Initialisierung von primitiven Datentypen

## RAII II

- Objekt wird durch Aufruf eines Konstruktors bei Definition initialisiert
  - Vor dem Konstruktoraufruf existiert keine Möglichkeit, auf das Objekt zuzugreifen
  - Keine Initialisierung von primitiven Datentypen
- Am Ende seiner Lebenszeit wird das Objekt durch Aufruf eines sogenannten Destruktors deinitialisiert
  - Gibt dem Objekt die Möglichkeit aufzuräumen
  - Beispielsweise Freigabe dynamisch allozierten Speichers
  - Ebenso Freigabe anderer Ressourcen, z.B. Datei-Handles
  - Zu keinem Zeitpunkt Zugriff auf uninitialisiertes Objekt möglich

## RAII II

- Objekt wird durch Aufruf eines Konstruktors bei Definition initialisiert
  - Vor dem Konstruktoraufruf existiert keine Möglichkeit, auf das Objekt zuzugreifen
  - Keine Initialisierung von primitiven Datentypen
- Am Ende seiner Lebenszeit wird das Objekt durch Aufruf eines sogenannten Destruktors deinitialisiert
  - Gibt dem Objekt die Möglichkeit aufzuräumen
  - Beispielsweise Freigabe dynamisch allozierten Speichers
  - Ebenso Freigabe anderer Ressourcen, z.B. Datei-Handles
  - Zu keinem Zeitpunkt Zugriff auf uninitialisiertes Objekt möglich
- Aufruf der Destruktoren in umgekehrter Definitionsreihenfolge
  - Zuletzt erstellte Objekte werden zuerst zerstört

## RAII II

- Objekt wird durch Aufruf eines Konstruktors bei Definition initialisiert
  - Vor dem Konstruktoraufruf existiert keine Möglichkeit, auf das Objekt zuzugreifen
  - Keine Initialisierung von primitiven Datentypen
- Am Ende seiner Lebenszeit wird das Objekt durch Aufruf eines sogenannten Destruktors deinitialisiert
  - Gibt dem Objekt die Möglichkeit aufzuräumen
  - Beispielsweise Freigabe dynamisch allozierten Speichers
  - Ebenso Freigabe anderer Ressourcen, z.B. Datei-Handles
  - Zu keinem Zeitpunkt Zugriff auf uninitialisiertes Objekt möglich
- Aufruf der Destruktoren in umgekehrter Definitionsreihenfolge
  - Zuletzt erstellte Objekte werden zuerst zerstört

Ressourcen (z.B. Speicher) in C++ **immer** über entsprechende RAI-Objekte managen!

## RAII Beispiel

```
int main()
{
    std::vector<std::string> message = {"hello"};
    std::string world = "world";
    message.emplace_back(world);
    message.push_back(std::string("\n"));
    if(!message.empty())
    {
        std::fstream fs("output.txt");
        for(std::string elem: message)
        {
            fs<<elem;
        }
    }
}
```

## RAII Beispiel

```
int main()
{
    std::vector<std::string> message = {"hello"};
    std::string world = "world";
    message.emplace_back(world);
    message.push_back(std::string("\n"));
    if(!message.empty())
    {
        std::fstream fs("output.txt");
        for(std::string elem: message)
        {
            fs<<elem;
        }
    }
}
```



## RAII Beispiel

```
int main()
{
    std::vector<std::string> message = {"hello"};
    std::string world = "world";
    message.emplace_back(world);
    message.push_back(std::string("\n"));
    if(!message.empty())
    {
        std::fstream fs("output.txt");
        for(std::string elem: message)
        {
            fs<<elem;
        }
    }
}
```

## RAII Beispiel

```
int main()
{
    std::vector<std::string> message = {"hello"};
    std::string world = "world";
    message.emplace_back(world);
    message.push_back(std::string("\n"));
    if(!message.empty())
    {
        std::fstream fs("output.txt");
        for(std::string elem: message)
        {
            fs<<elem;
        }
    }
}
```

## RAII Beispiel

```
int main()
{
    std::vector<std::string> message = {"hello"};
    std::string world = "world";
    message.emplace_back(world);
    message.push_back(std::string("\n"));
    if(!message.empty())
    {
        std::fstream fs("output.txt");
        for(std::string elem: message)
        {
            fs<<elem;
        }
    }
}
```

- Normalerweise hier `const std::string& elem` verwenden

## RAII Beispiel

```
int main()
{
    std::vector<std::string> message = {"hello"};
    std::string world = "world";
    message.emplace_back(world);
    message.push_back(std::string("\n"));
    if(!message.empty())
    {
        std::fstream fs("output.txt");
        for(std::string elem: message)
        {
            fs<<elem;
        }
    }
}
```

## RAII Beispiel

```
int main()
{
    std::vector<std::string> message = {"hello"};
    std::string world = "world";
    message.emplace_back(world);
    message.push_back(std::string("\n"));
    if(!message.empty())
    {
        std::fstream fs("output.txt");
        for(std::string elem: message)
        {
            fs<<elem;
        }
    }
}
```

- Zerstörung von `fs` schließt die Datei

## RAII Beispiel

```
int main()
{
    std::vector<std::string> message = {"hello"};
    std::string world = "world";
    message.emplace_back(world);
    message.push_back(std::string("\n"));
    if(!message.empty())
    {
        std::fstream fs("output.txt");
        for(std::string elem: message)
        {
            fs<<elem;
        }
    }
}
```

## RAII Beispiel

```
int main()
{
    std::vector<std::string> message = {"hello"};
    std::string world = "world";
    message.emplace_back(world);
    message.push_back(std::string("\n"));
    if(!message.empty())
    {
        std::fstream fs("output.txt");
        for(std::string elem: message)
        {
            fs<<elem;
        }
    }
}
```

## RAII Beispiel

```
int main()
{
    std::vector<std::string> message = {"hello"};
    std::string world = "world";
    message.emplace_back(world);
    message.push_back(std::string("\n"));
    if(!message.empty())
    {
        std::fstream fs("output.txt");
        for(std::string elem: message)
        {
            fs<<elem;
        }
    }
}
```



## RAII Beispiel

```
int main()
{
    std::vector<std::string> message = {"hello"};
    std::string world = "world";
    message.emplace_back(world);
    message.push_back(std::string("\n"));
    if(!message.empty())
    {
        std::fstream fs("output.txt");
        for(std::string elem: message)
        {
            fs<<elem;
        }
    }
}
```

- Zerstörung von `message` zerstört automatisch die enthaltenen Strings

# Smartpointer

- Verwendung von Smartpointern zur Speicherverwaltung
  - Klassen, die dank Operator-Überladung wie Pointer verwendet werden
  - Dynamisch allozierter Speicher wird automatisch über die Lebenszeit des Smartpointers gemanaged
  - Nur Dereferenzierung und Vergleich, keine Pointerarithmetik direkt auf dem Smartpointer

# Smartpointer

- Verwendung von Smartpointern zur Speicherverwaltung
  - Klassen, die dank Operator-Überladung wie Pointer verwendet werden
  - Dynamisch allozierter Speicher wird automatisch über die Lebenszeit des Smartpointers gemanaged
  - Nur Dereferenzierung und Vergleich, keine Pointerarithmetik direkt auf dem Smartpointer
- Normale Pointer im Folgenden als Rawpointer bezeichnet

# Smartpointer

- Verwendung von Smartpointern zur Speicherverwaltung
  - Klassen, die dank Operator-Überladung wie Pointer verwendet werden
  - Dynamisch allozierter Speicher wird automatisch über die Lebenszeit des Smartpointers gemanaged
  - Nur Dereferenzierung und Vergleich, keine Pointerarithmetik direkt auf dem Smartpointer
- Normale Pointer im Folgenden als Rawpointer bezeichnet
- Seit C++11 verschiedene Smartpointer im C++-Standard:
  - `std::unique_ptr`
  - `std::shared_ptr`
  - `std::weak_ptr`

# Smartpointer

- Verwendung von Smartpointern zur Speicherverwaltung
  - Klassen, die dank Operator-Überladung wie Pointer verwendet werden
  - Dynamisch allozierter Speicher wird automatisch über die Lebenszeit des Smartpointers gemanaged
  - Nur Dereferenzierung und Vergleich, keine Pointerarithmetik direkt auf dem Smartpointer
- Normale Pointer im Folgenden als Rawpointer bezeichnet
- Seit C++11 verschiedene Smartpointer im C++-Standard:
  - `std::unique_ptr`
  - `std::shared_ptr`
  - `std::weak_ptr`

Manuell allozierter Speicher sollte immer durch einen Smartpointer verwaltet werden!

## unique\_ptr

- `std::unique_ptr<T>` ist primärer Smartpointertyp in C++

## unique\_ptr

- `std::unique_ptr<T>` ist primärer Smartpointertyp in C++
- Modelliert eindeutige Besitzverhältnisse
  - Bedeutet: Es gibt genau ein Objekt, das die Lebenszeit des allozierten Speichers bestimmt

## unique\_ptr

- `std::unique_ptr<T>` ist primärer Smartpointertyp in C++
- Modelliert eindeutige Besitzverhältnisse
  - Bedeutet: Es gibt genau ein Objekt, das die Lebenszeit des allozierten Speichers bestimmt
- Am Ende der Lebenszeit des `std::unique_ptr` wird der allozierte Speicher freigegeben



## unique\_ptr

- `std::unique_ptr<T>` ist primärer Smartpointertyp in C++
- Modelliert eindeutige Besitzverhältnisse
  - Bedeutet: Es gibt genau ein Objekt, das die Lebenszeit des allozierten Speichers bestimmt
- Am Ende der Lebenszeit des `std::unique_ptr` wird der allozierte Speicher freigegeben
- Kann nicht kopiert, sondern nur verschoben werden

## unique\_ptr

```
#include <string>
#include <memory>
#include <iostream>
std::unique_ptr<std::string> makeString() {
    return std::make_unique<std::string>("hello world");
}
void consumeString(std::unique_ptr<std::string> p);

int main() {
    std::unique_ptr<std::string> p = makeString();
    std::cout<<p.get()<<": " <<*p<< "(" <<p->size()<<")\n";

}
```

## unique\_ptr

```
std::unique_ptr<std::string> makeString() {
    return std::make_unique<std::string>("hello world");
}

void consumeString(std::unique_ptr<std::string> p);

int main() {
    std::unique_ptr<std::string> p = makeString();
    std::cout<<p.get()<<": " <<*p<< "(" <<p->size()<<")\n";
}
}
```

- `std::make_unique<std::string>(args)` erstellt einen `std::unique_ptr`, der auf einen neu allozierten String zeigt
  - Allozierter Speicher wird somit vom `unique_ptr` verwaltet
  - Verfügbar seit C++14, vorher: `std::unique_ptr<T>(new T(args))`

## unique\_ptr und Kopien

- Eindeutige Besitzverhältnisse verbieten auf natürliche Weise Kopien
  - Exklusiver Besitz einer Resource durch zwei `std::unique_ptr` macht keinen Sinn

## unique\_ptr und Kopien

- Eindeutige Besitzverhältnisse verbieten auf natürliche Weise Kopien
  - Exklusiver Besitz einer Resource durch zwei `std::unique_ptr` macht keinen Sinn
- `std::unique_ptr` kann dementsprechend nicht kopiert werden
- Wird dennoch im Beispiel per Value zurückgegeben

## unique\_ptr und Kopien

- Eindeutige Besitzverhältnisse verbieten auf natürliche Weise Kopien
  - Exklusiver Besitz einer Resource durch zwei `std::unique_ptr` macht keinen Sinn
- `std::unique_ptr` kann dementsprechend nicht kopiert werden
- Wird dennoch im Beispiel per Value zurückgegeben
- Nutzt eine Neuerung in C++11: Rvalue-Referenzen
  - Identifizieren einen Wert als Rvalue, wird durch `Type&&` deklariert
    - Zur Erinnerung: `Type&` deklariert eine Lvalue-Referenz
  - Später mehr dazu

## unique\_ptr und Kopien

- Eindeutige Besitzverhältnisse verbieten auf natürliche Weise Kopien
  - Exklusiver Besitz einer Resource durch zwei `std::unique_ptr` macht keinen Sinn
- `std::unique_ptr` kann dementsprechend nicht kopiert werden
- Wird dennoch im Beispiel per Value zurückgegeben
- Nutzt eine Neuerung in C++11: Rvalue-Referenzen
  - Identifizieren einen Wert als Rvalue, wird durch *Type*&& deklariert
    - Zur Erinnerung: *Type*& deklariert eine Lvalue-Referenz
  - Später mehr dazu
- Rvalues werden kurz nach ihrer Nutzung zerstört

## unique\_ptr und Kopien

- Eindeutige Besitzverhältnisse verbieten auf natürliche Weise Kopien
  - Exklusiver Besitz einer Resource durch zwei `std::unique_ptr` macht keinen Sinn
- `std::unique_ptr` kann dementsprechend nicht kopiert werden
- Wird dennoch im Beispiel per Value zurückgegeben
- Nutzt eine Neuerung in C++11: Rvalue-Referenzen
  - Identifizieren einen Wert als Rvalue, wird durch *Type*&& deklariert
    - Zur Erinnerung: *Type*& deklariert eine Lvalue-Referenz
  - Später mehr dazu
- Rvalues werden kurz nach ihrer Nutzung zerstört
- Bei per-Value-Rückgabe eines Rvalue-unique\_ptr wird der Besitz der Resource an den neuen unique\_ptr übertragen



## unique\_ptr

```
std::unique_ptr<std::string> makeString() {  
    return std::make_unique<std::string>("hello world");  
}  
  
void consumeString(std::unique_ptr<std::string> p);  
  
int main() {  
    std::unique_ptr<std::string> p = makeString();  
    std::cout<<p.get()<<": " <<*p<< "(" <<p->size()<<")\n";  
  
}
```

- Besitz des allozierten Speichers wird auf das temporäre, von **return** zurückgegebene Objekt verschoben

## unique\_ptr

```
std::unique_ptr<std::string> makeString() {  
    return std::make_unique<std::string>("hello world");  
}  
  
void consumeString(std::unique_ptr<std::string> p);  
  
int main() {  
    std::unique_ptr<std::string> p = makeString();  
    std::cout<<p.get()<<": " <<*p<< "(" <<p->size()<<")\n";  
  
}
```

- Besitz des allozierten Speichers wird auf das temporäre, von **return** zurückgegebene Objekt verschoben
- Der allozierte Speicher wird weiterverschoben und gehört nun p
- Optimierende Compiler können beide Verschiebungen wegoptimieren und den allozierten Speicher direkt in p ablegen

## unique\_ptr

```
std::unique_ptr<std::string> makeString() {  
    return std::make_unique<std::string>("hello world");  
}  
  
void consumeString(std::unique_ptr<std::string> p);  
  
int main() {  
    std::unique_ptr<std::string> p = makeString();  
    std::cout<<p.get()<<": " <<*p<< "(" <<p->size()<<")\n";  
  
}
```

- Mit dem Dereferenzierungsoperator kann wie bei Rawpointern auf den adressierten Speicher zugegriffen werden

## unique\_ptr

```
std::unique_ptr<std::string> makeString() {  
    return std::make_unique<std::string>("hello world");  
}  
  
void consumeString(std::unique_ptr<std::string> p);  
  
int main() {  
    std::unique_ptr<std::string> p = makeString();  
    std::cout<<p.get()<<" : " <<*p<<" (" <<p->size()<<" )\n";  
  
}
```

- Mit dem Dereferenzierungsoperator kann wie bei Rawpointern auf den adressierten Speicher zugegriffen werden
- Auf den in einem Smartpointer enthaltenen Rawpointer kann mit der `.get()`-Methode zugegriffen werden

## unique\_ptr

```
std::unique_ptr<std::string> makeString() {  
    return std::make_unique<std::string>("hello world");  
}  
  
void consumeString(std::unique_ptr<std::string> p);  
  
int main() {  
    std::unique_ptr<std::string> p = makeString();  
    std::cout<<p.get()<<": " <<*p<< "(" <<p->size()<<")\n";  
}
```

- Am Ende des Scopes endet die Lebenszeit von p
- Besitz des allozierten Speichers wurde weder an einen anderen Smartpointer weitergegeben, noch mit `.release()` manuell entfernt
- Zerstörung von p gibt den allozierten Speicher automatisch frei

## unique\_ptr

```
std::unique_ptr<std::string> makeString() {
    return std::make_unique<std::string>("hello world");
}

void consumeString(std::unique_ptr<std::string> p);

int main() {
    std::unique_ptr<std::string> p = makeString();
    std::cout<<p.get()<<": " <<*p<< "(" <<p->size()<<")\n";
    consumeString(p);
}
```

## unique\_ptr

```
std::unique_ptr<std::string> makeString() {
    return std::make_unique<std::string>("hello world");
}

void consumeString(std::unique_ptr<std::string> p);

int main() {
    std::unique_ptr<std::string> p = makeString();
    std::cout<<p.get()<<": " <<*p<< "(" <<p->size()<<")\n";
    consumeString(p);
}
```

- Versucht, p (einen Lvalue) per Value zu übergeben
- Compilerfehler: `std::unique_ptr` kann nicht kopiert werden

## Rvalues und move

- `std::unique_ptr` können nur per Value übergeben werden, wenn sie Rvalues sind
  - Trifft in der Regel nicht auf Variablen zu



## Rvalues und move

- `std::unique_ptr` können nur per Value übergeben werden, wenn sie Rvalues sind
  - Trifft in der Regel nicht auf Variablen zu
- C++ bietet daher `std::move`
  - `std::move(val)` gibt eine Rvalue-Referenz zurück
  - Auch wenn `val` ein Lvalue ist
  - Im Header `utility`
  - Äquivalent zu `static_cast<T&&>(val)`

## Rvalues und move

- `std::unique_ptr` können nur per Value übergeben werden, wenn sie Rvalues sind
  - Trifft in der Regel nicht auf Variablen zu
- C++ bietet daher `std::move`
  - `std::move(val)` gibt eine Rvalue-Referenz zurück
  - Auch wenn `val` ein Lvalue ist
  - Im Header `utility`
  - Äquivalent zu `static_cast<T&&>(val)`
- **Achtung:** Verwendung einer Variablen als Rvalue hinterlässt diese in unbestimmtem Zustand
  - Zustand muss gültig sein
  - Häufig: Abhängig von den verwendeten Operationen entweder im Originalzustand, oder v.a. bei Containern leer

## Rvalues und move

- `std::unique_ptr` können nur per Value übergeben werden, wenn sie Rvalues sind
  - Trifft in der Regel nicht auf Variablen zu
- C++ bietet daher `std::move`
  - `std::move(val)` gibt eine Rvalue-Referenz zurück
  - Auch wenn `val` ein Lvalue ist
  - Im Header `utility`
  - Äquivalent zu `static_cast<T&&>(val)`
- **Achtung:** Verwendung einer Variablen als Rvalue hinterlässt diese in unbestimmtem Zustand
  - Zustand muss gültig sein
  - Häufig: Abhängig von den verwendeten Operationen entweder im Originalzustand, oder v.a. bei Containern leer
- Vor erneuter Verwendung die Variable neu befüllen

## unique\_ptr

```
std::unique_ptr<std::string> makeString() {  
    return std::make_unique<std::string>("hello world");  
}  
  
void consumeString(std::unique_ptr<std::string> p);  
  
int main() {  
    std::unique_ptr<std::string> p = makeString();  
    std::cout<<p.get()<<": " <<*p<< "(" <<p->size()<<")\n";  
    consumeString(p);  
}
```

- Versucht, p (einen Lvalue) per Value zu übergeben
- Compilerfehler: `std::unique_ptr` kann nicht kopiert werden

## unique\_ptr

```
std::unique_ptr<std::string> makeString() {
    return std::make_unique<std::string>("hello world");
}

void consumeString(std::unique_ptr<std::string> p);

int main() {
    std::unique_ptr<std::string> p = makeString();
    std::cout<<p.get()<<": " <<*p<< "(" <<p->size()<<")\n";
    consumeString(std::move(p));
}
```

- Versucht, p (einen Lvalue) per Value zu übergeben
  - Compilerfehler: `std::unique_ptr` kann nicht kopiert werden
  - `std::move` wandelt p in einen Rvalue um, Funktionsaufruf kann p die Daten stehlen
- p ist nach dem Funktionsaufruf leer

## unique\_ptr

```
std::unique_ptr<std::string> makeString() {
    return std::make_unique<std::string>("hello world");
}

void consumeString(std::unique_ptr<std::string> p);

int main() {
    std::unique_ptr<std::string> p = makeString();
    std::cout<<p.get()<<": ";<<*p<< "("<<p->size()<<")\n";
    consumeString(std::move(p));
    p = std::make_unique<std::string>("hello world again");
}
```

- `p = ...` gibt die enthaltene Resource (sofern vorhanden) frei und reinitialisiert `p` mit dem übergebenen Smartpointer
- `p = nullptr` würde `p` auf einen Nullpointer setzen.

## shared\_ptr

- `std::shared_ptr` funktioniert wie `std::unique_ptr`, erlaubt aber geteilte Besitzverhältnisse

## shared\_ptr

- `std::shared_ptr` funktioniert wie `std::unique_ptr`, erlaubt aber geteilte Besitzverhältnisse
- Erzeugung mit `std::make_shared<T>(args)`
  - `auto p = std::make_shared<std::string>("Hello World!");`
- Mehrere `std::shared_ptr` auf denselben Speicherbereich möglich



## shared\_ptr

- `std::shared_ptr` funktioniert wie `std::unique_ptr`, erlaubt aber geteilte Besitzverhältnisse
- Erzeugung mit `std::make_shared<T>(args)`
  - `auto p = std::make_shared<std::string>("Hello World!");`
- Mehrere `std::shared_ptr` auf denselben Speicherbereich möglich
- Der Speicher wird freigegeben, wenn die Lebenszeit des letzten darauf zeigenden `shared_ptr` endet
  - Verwendet Reference Counting
  - Zählt mit, wieviele `shared_ptr` noch leben
  - Potentiell: Performancekosten durch zusätzliche Allokation des Reference Counts

## shared\_ptr II

- Weitere `shared_ptr` müssen als Kopien eines bestehenden `shared_ptr` erzeugt werden
  - Werden mehrere `shared_ptr` unabhängig voneinander aus einem Rawpointer erstellt, funktioniert das Reference Counting nicht

## shared\_ptr II

- Weitere `shared_ptr` müssen als Kopien eines bestehenden `shared_ptr` erzeugt werden
  - Werden mehrere `shared_ptr` unabhängig voneinander aus einem Rawpointer erstellt, funktioniert das Reference Counting nicht
- Reference Count muss für jede Kopie erhöht werden, für jede Zerstörung verringert
  - Muss threadsicher geschehen, daher bedeutender Laufzeitoverhead

## shared\_ptr II

- Weitere `shared_ptr` müssen als Kopien eines bestehenden `shared_ptr` erzeugt werden
  - Werden mehrere `shared_ptr` unabhängig voneinander aus einem Rawpointer erstellt, funktioniert das Reference Counting nicht
- Reference Count muss für jede Kopie erhöht werden, für jede Zerstörung verringert
  - Muss threadsicher geschehen, daher bedeutender Laufzeitoverhead
- Nachteil von Reference Counting: Keine Erkennung von Zyklen
  - Aufeinander verweisende `shared_ptr` halten gegenseitig den Reference Count über 0

## shared\_ptr II

- Weitere shared\_ptr müssen als Kopien eines bestehenden shared\_ptr erzeugt werden
  - Werden mehrere shared\_ptr unabhängig voneinander aus einem Rawpointer erstellt, funktioniert das Reference Counting nicht
- Reference Counting
  - Muss threadsicher sein
  - Nachteil von Reference Counting: Reference Counting verringert
  - Aufeinander verweisen: Reference Count über 0

```
struct foo{
    std::shared_ptr<foo> p;
    foo(const std::shared_ptr<foo>& pf):
        p(pf) {}
    foo():p() {}
};

auto a = std::make_shared<foo>();
auto b = std::make_shared<foo>(a);
a.p = b;
```

## shared\_ptr II

- Weitere `shared_ptr` müssen als Kopien eines bestehenden `shared_ptr` erzeugt werden
  - Werden mehrere `shared_ptr` unabhängig voneinander aus einem Rawpointer erstellt, funktioniert das Reference Counting nicht
- Reference Count muss für jede Kopie erhöht werden, für jede Zerstörung verringert
  - Muss threadsicher geschehen, daher bedeutender Laufzeitoverhead
- Nachteil von Reference Counting: Keine Erkennung von Zyklen
  - Aufeinander verweisende `shared_ptr` halten gegenseitig den Reference Count über 0
- Zyklen können durch Verwendung von `std::weak_ptr<T>` aufgebrochen werden
  - [http://en.cppreference.com/w/cpp/memory/weak\\_ptr](http://en.cppreference.com/w/cpp/memory/weak_ptr)

## Wahl des richtigen Smartpointertyps

- Dynamisch allozierter Speicher sollte **immer** durch Smartpointer verwaltet werden
  - Manuelle Verwaltung birgt hohes Risiko von Bugs

## Wahl des richtigen Smartpointertyps

- Dynamisch allozierter Speicher sollte **immer** durch Smartpointer verwaltet werden
  - Manuelle Verwaltung birgt hohes Risiko von Bugs
- `std::unique_ptr` modelliert einfachstes Besitzverhältnis (hierarchisch) und hat nur minimalen Overhead
  - Overhead wird von den Kosten einer Speicherallokation in den Schatten gestellt
  - Immer verwenden, außer wenn geteilter Besitz notwendig ist



## Wahl des richtigen Smartpointertyps

- Dynamisch allozierter Speicher sollte **immer** durch Smartpointer verwaltet werden
  - Manuelle Verwaltung birgt hohes Risiko von Bugs
- `std::unique_ptr` modelliert einfachstes Besitzverhältnis (hierarchisch) und hat nur minimalen Overhead
  - Overhead wird von den Kosten einer Speicherallozierung in den Schatten gestellt
  - Immer verwenden, außer wenn geteilter Besitz notwendig ist
- `std::shared_ptr` verwenden, wenn Lebenszeit des Speichers von mehreren Objekten abhängt

## Wahl des richtigen Smartpointertyps

- Dynamisch allozierter Speicher sollte **immer** durch Smartpointer verwaltet werden
  - Manuelle Verwaltung birgt hohes Risiko von Bugs
- `std::unique_ptr` modelliert einfachstes Besitzverhältnis (hierarchisch) und hat nur minimalen Overhead
  - Overhead wird von den Kosten einer Speicherallokation in den Schatten gestellt
  - Immer verwenden, außer wenn geteilter Besitz notwendig ist
- `std::shared_ptr` verwenden, wenn Lebenszeit des Speichers von mehreren Objekten abhängt
- `std::weak_ptr` nur zum Verhindern von Zyklen verwenden

# Smartpointer und Rawpointer

- Dynamisch allozierter Speicher sollte immer durch Smartpointer verwaltet werden
  - Bedeutet nicht, dass er nur durch Smartpointer adressiert wird

## Smartpointer und Rawpointer

- Dynamisch allozierter Speicher sollte immer durch Smartpointer verwaltet werden
  - Bedeutet nicht, dass er nur durch Smartpointer adressiert wird
- Smartpointer nur in Objekten/Funktionen verwenden, die die Lebenszeit beeinflussen

## Smartpointer und Rawpointer

- Dynamisch allozierter Speicher sollte immer durch Smartpointer verwaltet werden
  - Bedeutet nicht, dass er nur durch Smartpointer adressiert wird
- Smartpointer nur in Objekten/Funktionen verwenden, die die Lebenszeit beeinflussen
- Übergabe an Funktionen in der Regel als Rawpointer
  - Funktion wird beendet, bevor der Smartpointer zerstört werden kann
  - Unwichtig, ob Verwaltung durch `unique_ptr` oder `shared_ptr`
  - Spart Overhead
  - Ausnahme: Übergabe als Smartpointer, falls die Funktion den Pointer extern speichert

## Smartpointer und Rawpointer

- Dynamisch allozierter Speicher sollte immer durch Smartpointer verwaltet werden
  - Bedeutet nicht, dass er nur durch Smartpointer adressiert wird
- Smartpointer nur in Objekten/Funktionen verwenden, die die Lebenszeit beeinflussen
- Übergabe an Funktionen in der Regel als Rawpointer
  - Funktion wird beendet, bevor der Smartpointer zerstört werden kann
  - Unwichtig, ob Verwaltung durch `unique_ptr` oder `shared_ptr`
  - Spart Overhead
  - Ausnahme: Übergabe als Smartpointer, falls die Funktion den Pointer extern speichert
- Für Datenmember abhängig von den Besitzverhältnissen

## Link

- Weitere Hinweise zu dynamischer Speicherverwaltung und `std::move`:  
<https://dei.spdns.de/cpp/>

## C-Style Arrays

- Ähnlich zu C können in C++ mit `type varname[length]` Arrays definiert werden
  - In C++ nur mit zur Compilezeit bekannter Länge
  - C-Style Arrays werden stackalloziert und sind kontinuierlich
  - Typ des Arrays: `type[length]`



## C-Style Arrays

- Ähnlich zu C können in C++ mit `type varname[length]` Arrays definiert werden
  - In C++ nur mit zur Compilezeit bekannter Länge
  - C-Style Arrays werden stackalloziert und sind kontinuierlich
  - Typ des Arrays: `type[length]`
- C-Style Arrays konvertieren implizit in Pointer auf das erste Element des Arrays

## C-Style Arrays

- Ähnlich zu C können in C++ mit `type varname[length]` Arrays definiert werden
  - In C++ nur mit zur Compilezeit bekannter Länge
  - C-Style Arrays werden stackalloziert und sind kontinuierlich
  - Typ des Arrays: `type[length]`
- C-Style Arrays konvertieren implizit in Pointer auf das erste Element des Arrays
- Mit `new T[N]` kann ein Array dynamisch alloziert werden
  - Wird mit `delete[] ptr` (**nicht** `delete`) wieder freigegeben
  - `std::unique_ptr<T[]>` verwendet korrekte Deallokation
  - Allokation besser mit `std::make_unique<T[]>(...)`

## C-Style Arrays

- Ähnlich zu C können in C++ mit `type varname[length]` Arrays definiert werden
  - In C++ nur mit zur Compilezeit bekannter Länge
  - C-Style Arrays werden stackalloziert und sind kontinuierlich
  - Typ des Arrays: `type[length]`
- C-Style Arrays konvertieren implizit in Pointer auf das erste Element des Arrays
- Mit `new T[N]` kann ein Array dynamisch alloziert werden
  - Wird mit `delete[] ptr` (**nicht** `delete`) wieder freigegeben
  - `std::unique_ptr<T[]>` verwendet korrekte Deallokation
  - Allokation besser mit `std::make_unique<T[]>(...)`
- `std::vector` ist fast immer eine bessere Alternative, weiterhin existiert `std::array` als sicherer Wrapper für C-style Arrays

## C-Style Arrays

- Ähnlich zu C können in C++ mit `type varname[length]` Arrays definiert werden
  - In C++ nur mit zur Compilezeit bekannter Länge
  - C-Style Arrays werden stackalloziert und sind kontinuierlich
  - Typ des Arrays: `type[length]`
- C-Style Arrays konvertieren implizit in Pointer auf das erste Element des Arrays
- Mit `new T[N]` kann ein Array dynamisch alloziert werden
  - Wird mit `delete[] ptr` (**nicht** `delete`) wieder freigegeben
  - `std::unique_ptr<T[]>` verwendet korrekte Deallokation
  - Allokation besser mit `std::make_unique<T[]>(...)`
- `std::vector` ist fast immer eine bessere Alternative, weiterhin existiert `std::array` als sicherer Wrapper für C-style Arrays

`std::vector` oder `std::array` statt C-Style Arrays verwenden!

# String Literale

- String Literale sind (nahezu) rückwärts kompatibel zu C vom Typ `char const[N]`
  - Nullterminierte Zeichenkette, d.h. das Ende wird mit einem extra Character mit Wert 0 markiert
  - `N` gibt Länge des Literals inklusive dem terminalen Nullcharacter an
  - Typ von `"hello"` ist beispielsweise `char const[6]`

# String Literale

- String Literale sind (nahezu) rückwärts kompatibel zu C vom Typ `char const[N]`
  - Nullterminierte Zeichenkette, d.h. das Ende wird mit einem extra Character mit Wert 0 markiert
  - `N` gibt Länge des Literals inklusive dem terminalen Nullcharacter an
  - Typ von `"hello"` ist beispielsweise `char const[6]`
- Konvertieren implizit zu `const char*` und aufgrund einer Sonderregelung zu `char*`

# String Literale

- String Literale sind (nahezu) rückwärts kompatibel zu C vom Typ `char const[N]`
  - Nullterminierte Zeichenkette, d.h. das Ende wird mit einem extra Character mit Wert 0 markiert
  - `N` gibt Länge des Literals inklusive dem terminalen Nullcharacter an
  - Typ von `"hello"` ist beispielsweise `char const[6]`
- Konvertieren implizit zu `const char*` und aufgrund einer Sonderregelung zu `char*`
- Außer in extrem performancesensitivem Code immer direkt in einen `std::string` schreiben, nicht mit C-Style Strings arbeiten
  - Code ist erst dann performancesensitiv, wenn dies durch Messungen erwiesen wurde!

## Weiterführendes

- Im Folgenden: Kurze Erläuterung verbreiteter Low-Level-Konstrukte



## Weiterführendes

- Im Folgenden: Kurze Erläuterung verbreiteter Low-Level-Konstrukte
- **union**
  - Erlauben das Speichern verschiedener Variablen an der selben Speicherstelle
  - **Achtung:** Lesen eines Members nur nach vorherigem Schreiben erlaubt (nach Standard-C++, Compiler sind häufig weniger streng)

## Weiterführendes

- Im Folgenden: Kurze Erläuterung verbreiteter Low-Level-Konstrukte
- **union**
  - Erlauben das Speichern verschiedener Variablen an der selben Speicherstelle
  - **Achtung:** Lesen eines Members nur nach vorherigem Schreiben erlaubt (nach Standard-C++, Compiler sind häufig weniger streng)
- Placement **new**
  - Erlaubt die Erstellung eines Objektes an einer bereits existierenden (z.B. über `malloc` allozierten) Speicherstelle
  - Primär für eigene Speicherverwaltungsmechanismen und Unions wichtig, ansonsten selten verwendet

## Weiterführendes

- Im Folgenden: Kurze Erläuterung verbreiteter Low-Level-Konstrukte
- **union**
  - Erlauben das Speichern verschiedener Variablen an der selben Speicherstelle
  - **Achtung:** Lesen eines Members nur nach vorherigem Schreiben erlaubt (nach Standard-C++, Compiler sind häufig weniger streng)
- Placement **new**
  - Erlaubt die Erstellung eines Objektes an einer bereits existierenden (z.B. über `malloc` allozierten) Speicherstelle
  - Primär für eigene Speicherverwaltungsmechanismen und Unions wichtig, ansonsten selten verwendet
- **volatile**
  - Typmodifizier (wie **const**), verhindert gewisse Compileroptimierungen
  - Primär für Zugriff auf Memory-Mapped Registers und durch Interrupts gesetzte Variablen



IMS  
Institut für Mikroelektronische Systeme  
Leibniz Universität Hannover



Leibniz  
Universität  
Hannover

# Aufgaben zu Pointern

## Aufgabe A

Was gibt der folgende Code aus?

```
1: int i = 4;  
2: int* p1 = &i;  
3: *p1 = *p1 * *p1;  
4: std::cout << i << std::endl;
```

## Aufgabe A

Was gibt der folgende Code aus?

```
1: int i = 4;  
2: int* p1 = &i;  
3: *p1 = *p1 * *p1;  
4: std::cout << i << std::endl;
```

- Zeile 2: p1 wird auf die Adresse von i gesetzt

## Aufgabe A

Was gibt der folgende Code aus?

```
1: int i = 4;  
2: int* p1 = &i;  
3: *p1 = *p1 * *p1;  
4: std::cout << i << std::endl;
```

- Zeile 2: p1 wird auf die Adresse von i gesetzt
- Zeile 3: i wird in der letzten Zeile über p1 quadriert

## Aufgabe A

Was gibt der folgende Code aus?

```
1: int i = 4;  
2: int* p1 = &i;  
3: *p1 = *p1 * *p1;  
4: std::cout << i << std::endl;
```

- Zeile 2: p1 wird auf die Adresse von i gesetzt
- Zeile 3: i wird in der letzten Zeile über p1 quadriert
- i hat am Ende den Wert  $4^2$ , also 16



## Aufgabe B

Ist eine der folgenden Definitionen illegal?

```
1: int i = 0;  
2: double* dp = &i;  
3: int* ip = 0;  
4: int* p = &i;
```

## Aufgabe B

Ist eine der folgenden Definitionen illegal?

```
1: int i = 0;  
2: double* dp = &i;  
3: int* ip = 0;  
4: int* p = &i;
```

## Aufgabe B

Ist eine der folgenden Definitionen illegal?

```
1: int i = 0;
2: double* dp = &i;
3: int* ip = 0;
4: int* p = &i;
```

- Zeile 2: Illegal, da &i ein Pointer auf `int` und nicht auf `double` ist  
error: cannot convert '`int*`' to '`double*`' in initialization

## Aufgabe B

Ist eine der folgenden Definitionen illegal?

```
1: int i = 0;  
2: double* dp = &i;  
3: int* ip = 0;  
4: int* p = &i;
```

## Aufgabe B

Ist eine der folgenden Definitionen illegal?

```
1: int i = 0;
2: double* dp = &i;
3: int* ip = 0;
4: int* p = &i;
```

- Zeile 3: Legal, initialisiert ip als Nullpointer
- Besser wäre `int* ip = nullptr;`

## Aufgabe B

Ist eine der folgenden Definitionen illegal?

```
1: int i = 0;
2: double* dp = &i;
3: int* ip = 0;
4: int* p = &i;
```

## Aufgabe B

Ist eine der folgenden Definitionen illegal?

```
1: int i = 0;
2: double* dp = &i;
3: int* ip = 0;
4: int* p = &i;
```

- Zeile 4: Legal, initialisiert p als Pointer auf i

## Aufgabe C

```
1: int id    = 1024;
2: int* pi   = &id;
3: int* pi2  = new int(2048);
4: using IntPtr = std::unique_ptr<int>;
5: IntPtr p0(id);
6: IntPtr p1(pi);
7: IntPtr p2(pi2);
8: IntPtr p3(&id);
9: IntPtr p4(std::make_unique<int>(2048));
10: IntPtr p5(p2.get());
11: IntPtr p6(p4);
```



## Aufgabe C

```

1: int id    = 1024;
2: int* pi   = &id;
3: int* pi2  = new int(2048);
4: using IntPtr = std::unique_ptr<int>;
5: IntPtr p0(id);
6: IntPtr p1(pi);
7: IntPtr p2(pi2);
8: IntPtr p3(&id);
9: IntPtr p4(std::make_unique<int>(2048));
10: IntPtr p5(p2.get());
11: IntPtr p6(p4);

```

## Aufgabe C

```

1: int id    = 1024;
2: int* pi   = &id;
3: int* pi2  = new int(2048);
4: using IntPtr = std::unique_ptr<int>;
5: IntPtr p0(id);
6: IntPtr p1(pi);
7: IntPtr p2(pi2);
8: IntPtr p3(&id);
9: IntPtr p4(std::make_unique<int>(2048));
10: IntPtr p5(p2.get());
11: IntPtr p6(p4);

```

- Versucht einen `unique_ptr` mit einem `int` zu initialisieren
- Illegal, da kein passender Konstruktor existiert
- `error: invalid conversion from 'int' to 'std::unique_ptr<int>::pointer aka int*'`

## Aufgabe C

```
1: int id    = 1024;
2: int* pi   = &id;
3: int* pi2  = new int(2048);
4: using IntPtr = std::unique_ptr<int>;
5: IntPtr p0(id);
6: IntPtr p1(pi);
7: IntPtr p2(pi2);
8: IntPtr p3(&id);
9: IntPtr p4(std::make_unique<int>(2048));
10: IntPtr p5(p2.get());
11: IntPtr p6(p4);
```

## Aufgabe C

```

1: int id    = 1024;
2: int* pi   = &id;
3: int* pi2  = new int(2048);
4: using IntPtr = std::unique_ptr<int>;
5: IntPtr p0(id);
6: IntPtr p1(pi);
7: IntPtr p2(pi2);
8: IntPtr p3(&id);
9: IntPtr p4(std::make_unique<int>(2048));
10: IntPtr p5(p2.get());
11: IntPtr p6(p4);

```

- Initialisierung eines `unique_ptr` mit einem Raw-Pointer zwar legal
- Aber: `pi` zeigt auf Speicher, der nicht dynamisch alloziert ist
- Bei Zerstörung von `p1` wird versucht den Speicher freizugeben, führt zu undefiniertem Verhalten
- Wahrscheinliche Folge: Programmabsturz

## Aufgabe C

```
1: int id    = 1024;
2: int* pi   = &id;
3: int* pi2  = new int(2048);
4: using IntPtr = std::unique_ptr<int>;
5: IntPtr p0(id);
6: IntPtr p1(pi);
7: IntPtr p2(pi2);
8: IntPtr p3(&id);
9: IntPtr p4(std::make_unique<int>(2048));
10: IntPtr p5(p2.get());
11: IntPtr p6(p4);
```

## Aufgabe C

```
1: int id    = 1024;
2: int* pi   = &id;
3: int* pi2  = new int(2048);
4: using IntPtr = std::unique_ptr<int>;
5: IntPtr p0(id);
6: IntPtr p1(pi);
7: IntPtr p2(pi2);
8: IntPtr p3(&id);
9: IntPtr p4(std::make_unique<int>(2048));
10: IntPtr p5(p2.get());
11: IntPtr p6(p4);
```

- Alles Ok, initialisiert p2 mit einer auf dynamisch allozierten Speicher zeigenden Adresse

## Aufgabe C

```
1: int id    = 1024;
2: int* pi   = &id;
3: int* pi2  = new int(2048);
4: using IntPtr = std::unique_ptr<int>;
5: IntPtr p0(id);
6: IntPtr p1(pi);
7: IntPtr p2(pi2);
8: IntPtr p3(&id);
9: IntPtr p4(std::make_unique<int>(2048));
10: IntPtr p5(p2.get());
11: IntPtr p6(p4);
```

## Aufgabe C

```

1: int id    = 1024;
2: int* pi   = &id;
3: int* pi2  = new int(2048);
4: using IntPtr = std::unique_ptr<int>;
5: IntPtr p0(id);
6: IntPtr p1(pi);
7: IntPtr p2(pi2);
8: IntPtr p3(&id);
9: IntPtr p4(std::make_unique<int>(2048));
10: IntPtr p5(p2.get());
11: IntPtr p6(p4);

```

- Selbes Problem wie bei p1, versucht stackallozierten Speicher freizugeben



## Aufgabe C

```

1: int id    = 1024;
2: int* pi   = &id;
3: int* pi2  = new int(2048);
4: using IntPtr = std::unique_ptr<int>;
5: IntPtr p0(id);
6: IntPtr p1(pi);
7: IntPtr p2(pi2);
8: IntPtr p3(&id);
9: IntPtr p4(std::make_unique<int>(2048));
10: IntPtr p5(p2.get());
11: IntPtr p6(p4);

```

## Aufgabe C

```
1: int id    = 1024;
2: int* pi   = &id;
3: int* pi2  = new int(2048);
4: using IntPtr = std::unique_ptr<int>;
5: IntPtr p0(id);
6: IntPtr p1(pi);
7: IntPtr p2(pi2);
8: IntPtr p3(&id);
9: IntPtr p4(std::make_unique<int>(2048));
10: IntPtr p5(p2.get());
11: IntPtr p6(p4);
```

- Alles ok, initialisiert p4 direkt mit neualloziertem Speicher
- Standardvorgehen für Speicherallokation

## Aufgabe C

```
1: int id    = 1024;
2: int* pi   = &id;
3: int* pi2  = new int(2048);
4: using IntPtr = std::unique_ptr<int>;
5: IntPtr p0(id);
6: IntPtr p1(pi);
7: IntPtr p2(pi2);
8: IntPtr p3(&id);
9: IntPtr p4(std::make_unique<int>(2048));
10: IntPtr p5(p2.get());
11: IntPtr p6(p4);
```

## Aufgabe C

```
1: int id    = 1024;
2: int* pi   = &id;
3: int* pi2  = new int(2048);
4: using IntPtr = std::unique_ptr<int>;
5: IntPtr p0(id);
6: IntPtr p1(pi);
7: IntPtr p2(pi2);
8: IntPtr p3(&id);
9: IntPtr p4(std::make_unique<int>(2048));
10: IntPtr p5(p2.get());
11: IntPtr p6(p4);
```

- Initialisiert p5 mit dem selben Pointer wie p2
- Beide Objekte versuchen am Ende ihrer Lebenszeit den Speicher freizugeben
- Doppelte Freigabe von Speicher führt üblicherweise zu Korruption in den Datenstrukturen der Speicherverwaltung

## Aufgabe C

```
1: int id    = 1024;
2: int* pi   = &id;
3: int* pi2  = new int(2048);
4: using IntPtr = std::unique_ptr<int>;
5: IntPtr p0(id);
6: IntPtr p1(pi);
7: IntPtr p2(pi2);
8: IntPtr p3(&id);
9: IntPtr p4(std::make_unique<int>(2048));
10: IntPtr p5(p2.get());
11: IntPtr p6(p4);
```

## Aufgabe C

```

1: int id    = 1024;
2: int* pi   = &id;
3: int* pi2  = new int(2048);
4: using IntPtr = std::unique_ptr<int>;
5: IntPtr p0(id);
6: IntPtr p1(pi);
7: IntPtr p2(pi2);
8: IntPtr p3(&id);
9: IntPtr p4(std::make_unique<int>(2048));
10: IntPtr p5(p2.get());
11: IntPtr p6(p4);

```

- Versucht einen unique\_ptr zu kopieren
- Illegal, da unique\_ptr nur verschoben, aber nicht kopiert werden können
- Korrekter Ansatz: `IntPtr p6(std::move(p4));`



IMS  
Institut für Mikroelektronische Systeme  
Leibniz Universität Hannover



Leibniz  
Universität  
Hannover

# Exceptions



# Ausnahmebehandlung

```
#include <string>
#include <iostream>
#include <vector>
#include <fstream>

void copyFile(const std::string& inFile, const std::string& outFile) {
    std::ifstream src(inFile);
    std::ofstream dst(outFile);
    dst << src.rdbuf();
}
```

- Sieht korrekt aus



# Ausnahmebehandlung

```
#include <string>
#include <iostream>
#include <vector>
#include <fstream>
void copyFile(const std::string& inFile, const std::string& outFile) {
    std::ifstream src(inFile);
    std::ofstream dst(outFile);
    dst << src.rdbuf();
}
```

- Sieht korrekt aus
- Was passiert, wenn `inFile` nicht lesend geöffnet werden kann?

# Ausnahmebehandlung

```
#include <string>
#include <iostream>
#include <vector>
#include <fstream>
void copyFile(const std::string& inFile, const std::string& outFile) {
    std::ifstream src(inFile);
    std::ofstream dst(outFile);
    dst << src.rdbuf();
}
```

- Sieht korrekt aus
- Was passiert, wenn `inFile` nicht lesend geöffnet werden kann?
  - Weil die Datei nicht existiert
  - Weil die Datei von einer anderen Anwendung exklusiv geöffnet ist

# Ausnahmebehandlung

```
#include <string>
#include <iostream>
#include <vector>
#include <fstream>
void copyFile(const std::string& inFile, const std::string& outFile) {
    std::ifstream src(inFile);
    std::ofstream dst(outFile);
    dst << src.rdbuf();
}
```

- Sieht korrekt aus
- Was passiert, wenn `inFile` nicht lesend geöffnet werden kann?
  - Weil die Datei nicht existiert
  - Weil die Datei von einer anderen Anwendung exklusiv geöffnet ist
- Was passiert, wenn `outFile` nicht erzeugt werden kann?

# Ausnahmebehandlung

```
#include <string>
#include <iostream>
#include <vector>
#include <fstream>

void copyFile(const std::string& inFile, const std::string& outFile) {
    std::ifstream src(inFile);
    std::ofstream dst(outFile);
    dst << src.rdbuf();
}
```

- Sieht korrekt aus
- Was passiert, wenn `inFile` nicht lesend geöffnet werden kann?
  - Weil die Datei nicht existiert
  - Weil die Datei von einer anderen Anwendung exklusiv geöffnet ist
- Was passiert, wenn `outFile` nicht erzeugt werden kann?
  - Weil der Pfad nicht existiert
  - Weil eine andere Anwendung darauf zugreift

# Ausnahmebehandlung

```
#include <string>
#include <iostream>
#include <vector>
#include <fstream>

void copyFile(const std::string& inFile, const std::string& outFile) {
    std::ifstream src(inFile);
    std::ofstream dst(outFile);
    dst << src.rdbuf();
}
```

Berücksichtigung von unerwarteten Fehlern elementar für die Entwicklung stabiler Software!

- Sieht korrekt aus
- Was passiert, wenn `inFile` nicht lesend geöffnet werden kann?
  - Weil die Datei nicht existiert
  - Weil die Datei von einer anderen Anwendung exklusiv geöffnet ist
- Was passiert, wenn `outFile` nicht erzeugt werden kann?
  - Weil der Pfad nicht existiert
  - Weil eine andere Anwendung darauf zugreift

# Ausnahmebehandlung II

- Fehler häufig nicht an der Stelle ihres Auftretens behandelbar

## Ausnahmebehandlung II

- Fehler häufig nicht an der Stelle ihres Auftretens behandelbar
- Korrektes Vorgehen, wenn eine Operation in `CopyFile` fehlschlägt?
  - Abhängig vom aufrufenden Kontext
  - Kann konsequenzenlos oder fataler Fehler sein

## Ausnahmebehandlung II

- Fehler häufig nicht an der Stelle ihres Auftretens behandelbar
- Korrektes Vorgehen, wenn eine Operation in `CopyFile` fehlschlägt?
  - Abhängig vom aufrufenden Kontext
  - Kann konsequenzenlos oder fataler Fehler sein
- Fehler müssen häufig aus einer Funktion hinauskommuniziert werden



## Ausnahmebehandlung II

- Fehler häufig nicht an der Stelle ihres Auftretens behandelbar
- Korrektes Vorgehen, wenn eine Operation in `CopyFile` fehlschlägt?
  - Abhängig vom aufrufenden Kontext
  - Kann konsequenzenlos oder fataler Fehler sein
- Fehler müssen häufig aus einer Funktion hinauskommuniziert werden
- Im Wesentlichen zwei Ansätze, um Fehler zu kommunizieren
  - Return Codes
  - Exceptions

## Return Codes

- Bereits aus C bekannt
- Funktion signalisiert Erfolg und Misserfolg über den Rückgabewert

## Return Codes

- Bereits aus C bekannt
- Funktion signalisiert Erfolg und Misserfolg über den Rückgabewert
- Häufig: Rückgabebetyp `int` oder `enum`
  - 0 signalisiert Erfolg
  - Andere Rückgabewerte sind Error Codes, die die Art des aufgetretenen Fehlers spezifizieren

## Return Codes

- Bereits aus C bekannt
- Funktion signalisiert Erfolg und Misserfolg über den Rückgabewert
- Häufig: Rückgabebetyp `int` oder `enum`
  - 0 signalisiert Erfolg
  - Andere Rückgabewerte sind Error Codes, die die Art des aufgetretenen Fehlers spezifizieren
- Aufrufender Code überprüft den Rückgabewert und kann entsprechend reagieren

## Nachteile von Return Codes

- Aufrufender Code muss den Return Code überprüfen
  - Vergessene Überprüfung führt häufig zu inkonsistentem Programmzustand und schwierig zu debuggenden Problemen

## Nachteile von Return Codes

- Aufrufender Code muss den Return Code überprüfen
  - Vergessene Überprüfung führt häufig zu inkonsistentem Programmzustand und schwierig zu debuggenden Problemen

```
int process(Object& obj)
{
    obj.initialize();
    obj.doWork();
    obj.deinitialize();
    return 0;
}
```

Funktionieren doWork und deinitialize, falls initialize fehlschlägt?

## Nachteile von Return Codes

- Aufrufender Code muss den Return Code überprüfen
  - Vergessene Überprüfung führt häufig zu inkonsistentem Programmzustand und schwierig zu debuggenden Problemen



## Nachteile von Return Codes

- Aufrufender Code muss den Return Code überprüfen
  - Vergessene Überprüfung führt häufig zu inkonsistentem Programmzustand und schwierig zu debuggenden Problemen
  - Verwendung von 0 (entspricht `false`) als Erfolgssignal ist unintuitiv



## Nachteile von Return Codes

- Aufrufender Code muss den Return Code überprüfen
  - Vergessene Überprüfung führt häufig zu inkonsistentem Programmzustand und schwierig zu debuggenden Problemen
  - Verwendung von 0 (entspricht `false`) als Erfolgssignal ist unintuitiv
  - Durch die Vielzahl von `if`-Statements wird der Code unübersichtlich

## Nachteile von Return Codes

- Aufrufender Code muss den Return Code überprüfen

- Vergessene Überprüfung  
debuggenden
- Verwendung von  
Veränderung
- Durch die Vielzahl

```
int process(Object& obj) {  
    int result = 0;  
    if(!obj.initialize()) {  
        if(obj.doWork())  
            result = -2;  
        if(obj.deinitialize())  
            result = -3;  
    }  
    else  
        result = -1;  
    return result;  
}
```

and und schwierig zu

In echtem Code Konstanten/Enums für die Fehlercodes verwenden, nicht hart in die Funktion kodieren

## Nachteile von Return Codes

- Aufrufender Code muss den Return Code überprüfen
  - Vergessene Überprüfung führt häufig zu inkonsistentem Programmzustand und schwierig zu debuggenden Problemen
  - Verwendung von 0 (entspricht `false`) als Erfolgssignal ist unintuitiv
  - Durch die Vielzahl von `if`-Statements wird der Code unübersichtlich
- Auswertung des Rückgabewertes bibliotheksabhängig
  - Häufig signalisiert 0 erfolgreiche Ausführung
  - Manchmal ist die Rückgabe ein boolescher Wert, 0 signalisiert also Misserfolg
  - Andere Variante: Positive Werte signalisieren Erfolg (mit encodierten Informationen), negative Misserfolg

## Nachteile von Return Codes

- Aufrufender Code muss den Return Code überprüfen
  - Vergessene Überprüfung führt häufig zu inkonsistentem Programmzustand und schwierig zu debuggenden Problemen
  - Verwendung von 0 (entspricht `false`) als Erfolgssignal ist unintuitiv
  - Durch die Vielzahl von `if`-Statements wird der Code unübersichtlich
- Auswertung des Rückgabewertes bibliotheksabhängig
  - Häufig signalisiert 0 erfolgreiche Ausführung
  - Manchmal ist die Rückgabe ein boolescher Wert, 0 signalisiert also Misserfolg
  - Andere Variante: Positive Werte signalisieren Erfolg (mit encodierten Informationen), negative Misserfolg
- Rückgabewert wird für Return Code verwendet
  - Existiert eigentlicher Rückgabewert, muss für diesen eine Zielvariable per Pass-by-Reference (oder Pointer) übergeben werden
  - Verhindert Functionchaining: `foo(bar(5))`
  - Konstruktoren haben keinen Rückgabewert

## Der Callstack

Um Exceptions zu verstehen, ist ein Verständnis des Callstacks wichtig:

- Die Aufrufhierarchie eines Programmes lässt sich als Stack abbilden

## Der Callstack

Um Exceptions zu verstehen, ist ein Verständnis des Callstacks wichtig:

- Die Aufrufhierarchie eines Programmes lässt sich als Stack abbilden
- Bei Funktionsaufruf wird die aufgerufene Funktion auf den Stack gelegt
  - Stackeintrag (der sog. Stackframe) der Funktion enthält lokale Variablen
  - Gilt im C++-Modell prinzipiell nicht nur für Funktionen, sondern auch für Compound-Statements

## Der Callstack

Um Exceptions zu verstehen, ist ein Verständnis des Callstacks wichtig:

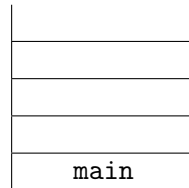
- Die Aufrufhierarchie eines Programmes lässt sich als Stack abbilden
- Bei Funktionsaufruf wird die aufgerufene Funktion auf den Stack gelegt
  - Stackeintrag (der sog. Stackframe) der Funktion enthält lokale Variablen
  - Gilt im C++-Modell prinzipiell nicht nur für Funktionen, sondern auch für Compound-Statements
- Bei Verlassen der Funktion wird der oberste Stackeintrag entfernt
  - Automatische Zerstörung aller lokalen RAII-Objekte in diesem Schritt

# Callstack-Visualisierung

```
void bar(int& x)
{
    x -= 2;
}
```

```
void foo(int& x)
{
    x += 3;
    bar(x);
}
```

```
int main()
{
    int x = 0; ←
    foo(x);
    x += 1;
    bar(x);
    return x;
}
```



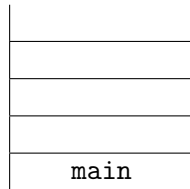


# Callstack-Visualisierung

```
void bar(int& x)
{
    x -= 2;
}
```

```
void foo(int& x)
{
    x += 3;
    bar(x);
}
```

```
int main()
{
    int x = 0;
    foo(x); ←
    x += 1;
    bar(x);
    return x;
}
```

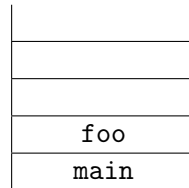


# Callstack-Visualisierung

```
void bar(int& x)
{
    x -= 2;
}
```

```
void foo(int& x)
{
    x += 3; ←
    bar(x);
}
```

```
int main()
{
    int x = 0;
    foo(x);
    x += 1;
    bar(x);
    return x;
}
```

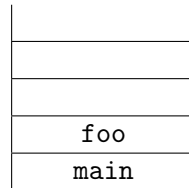


# Callstack-Visualisierung

```
void bar(int& x)
{
    x -= 2;
}
```

```
void foo(int& x)
{
    x += 3;
    bar(x); ←
}
```

```
int main()
{
    int x = 0;
    foo(x);
    x += 1;
    bar(x);
    return x;
}
```



# Callstack-Visualisierung

```
void bar(int& x)
{
    x -= 2; ←
}
```

```
void foo(int& x)
{
    x += 3;
    bar(x);
}
```

```
int main()
{
    int x = 0;
    foo(x);
    x += 1;
    bar(x);
    return x;
}
```

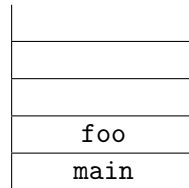
bar
foo
main

# Callstack-Visualisierung

```
void bar(int& x)
{
    x -= 2;
}
```

```
void foo(int& x)
{
    x += 3;
    bar(x); ←
}
```

```
int main()
{
    int x = 0;
    foo(x);
    x += 1;
    bar(x);
    return x;
}
```

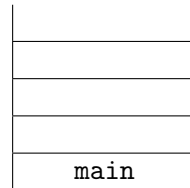


# Callstack-Visualisierung

```
void bar(int& x)
{
    x -= 2;
}
```

```
void foo(int& x)
{
    x += 3;
    bar(x);
}
```

```
int main()
{
    int x = 0;
    foo(x); ←
    x += 1;
    bar(x);
    return x;
}
```

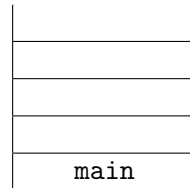


# Callstack-Visualisierung

```
void bar(int& x)
{
    x -= 2;
}
```

```
void foo(int& x)
{
    x += 3;
    bar(x);
}
```

```
int main()
{
    int x = 0;
    foo(x);
    x += 1; ←
    bar(x);
    return x;
}
```

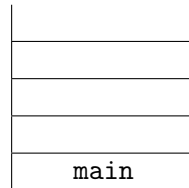


# Callstack-Visualisierung

```
void bar(int& x)
{
    x -= 2;
}
```

```
void foo(int& x)
{
    x += 3;
    bar(x);
}
```

```
int main()
{
    int x = 0;
    foo(x);
    x += 1;
    bar(x); ←
    return x;
}
```



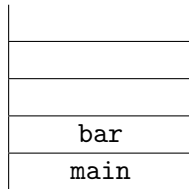


# Callstack-Visualisierung

```
void bar(int& x)
{
    x -= 2; ←
}
```

```
void foo(int& x)
{
    x += 3;
    bar(x);
}
```

```
int main()
{
    int x = 0;
    foo(x);
    x += 1;
    bar(x);
    return x;
}
```

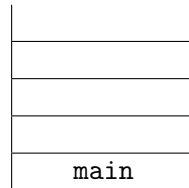


# Callstack-Visualisierung

```
void bar(int& x)
{
    x -= 2;
}
```

```
void foo(int& x)
{
    x += 3;
    bar(x);
}
```

```
int main()
{
    int x = 0;
    foo(x);
    x += 1;
    bar(x); ←
    return x;
}
```

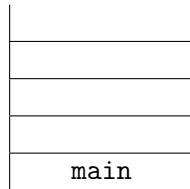


# Callstack-Visualisierung

```
void bar(int& x)
{
    x -= 2;
}
```

```
void foo(int& x)
{
    x += 3;
    bar(x);
}
```

```
int main()
{
    int x = 0;
    foo(x);
    x += 1;
    bar(x);
    return x; ←
```



# Exceptions

- Aufgrund der Probleme mit Return Codes unterstützen viele Programmiersprachen sogenannte Exceptions
- Exceptions erlauben eine fast vollständig vom normalen Kontrollfluss getrennte Behandlung von Fehlern
- Im Fehlerfall werden Exceptions geworfen

# Exceptions

- Aufgrund der Probleme mit Return Codes unterstützen viele Programmiersprachen sogenannte Exceptions
- Exceptions erlauben eine fast vollständig vom normalen Kontrollfluss getrennte Behandlung von Fehlern
- Im Fehlerfall werden Exceptions geworfen
- Exceptions propagieren entlang des Callstacks nach unten, bis sie auf einen Handler stoßen
  - RAI-Objekte werden dabei ordnungsgemäß zerstört
  - Ist zusätzliches Problem bei manuellem Speichermanagement in C++

# Exceptions

- Aufgrund der Probleme mit Return Codes unterstützen viele Programmiersprachen sogenannte Exceptions
- Exceptions erlauben eine fast vollständig vom normalen Kontrollfluss getrennte Behandlung von Fehlern
- Im Fehlerfall werden Exceptions geworfen
- Exceptions propagieren entlang des Callstacks nach unten, bis sie auf einen Handler stoßen
  - RAII-Objekte werden dabei ordnungsgemäß zerstört
  - Ist zusätzliches Problem bei manuellem Speichermanagement in C++
- Berücksichtigung der Exceptions erst notwendig, wenn sie gehandled werden
  - Im Gegensatz zu Return Codes keine manuelle Weitergabe durch jede Aufrufebebene notwendig

# Throw

- Mit **throw** *expr* wird eine Exception geworfen
  - Typ von *expr* prinzipiell beliebig
  - Konvention: Unterklasse von `std::exception`
  - Diverse Exceptionklassen in `stdexcept`

# Throw

- Mit **throw** *expr* wird eine Exception geworfen
  - Typ von *expr* prinzipiell beliebig
  - Konvention: Unterklasse von `std::exception`
  - Diverse Exceptionklassen in `stdexcept`
- Lebensdauer von temporären geworfenen Objekten wird verlängert, bis der Fehler gehandled ist



# Throw

- Mit **throw** *expr* wird eine Exception geworfen
  - Typ von *expr* prinzipiell beliebig
  - Konvention: Unterklasse von `std::exception`
  - Diverse Exceptionklassen in `stdexcept`
- Lebensdauer von temporären geworfenen Objekten wird verlängert, bis der Fehler gehandled ist
- Scope wird ansonsten ordnungsgemäß verlassen
  - Zerstörung lokaler Objekte für alle Scopes bis zum Handler

## Try-Catch

```
#include <stdexcept>
#include <iostream>
int foo() {
    try {
        throw std::runtime_error("throwExcept called");
    } catch(std::exception& e){
        std::cout<<e.what()<<std::endl;
        throw;
    }
}
```

- In Release-Code müssen Exceptions in der Regel gefangen werden
  - Nichtgefangene Exceptions führen zu Programmabbruch
- Zum Fangen von Exceptions dient try-catch

# Try-Catch Syntax

```
try
{
    regularCodePath
}
catch(exceptionDefinition)
{
    exceptionalCodePath
}
```

- Geschweifte Klammern sind notwendig, erzeugen immer einen Scope

## Try-Catch Syntax

```
try
{
    regularCodePath
}
catch(exceptionDefinition)
{
    exceptionalCodePath
}
```

- Geschweifte Klammern sind notwendig, erzeugen immer einen Scope
- *regularCodePath* wird immer ausgeführt
  - Bis zu dem Punkt, an dem eine Exception geworfen wird

## Try-Catch Syntax

```
try
{
    regularCodePath
}
catch(exceptionDefinition)
{
    exceptionalCodePath
}
```

- Geschweifte Klammern sind notwendig, erzeugen immer einen Scope
- *regularCodePath* wird immer ausgeführt
  - Bis zu dem Punkt, an dem eine Exception geworfen wird
- *exceptionalCodePath* wird nur ausgeführt, wenn eine Exception gefangen wird
- *exceptionDefinition* definiert die zu fangende Exception
  - Normale Variablendeklaration
  - Nur Exceptions mit kompatibelem Typ werden von dem `catch` gefangen
  - Inkompatible Exceptions propagieren weiter

## Try-Catch

```
#include <stdexcept>
#include <iostream>
int foo() {
    try {
        throw std::runtime_error("throwExcept called");
    } catch(std::exception& e){
        std::cout<<e.what()<<std::endl;
        throw;
    }
}
```

- Fängt Exceptions vom Typ `std::exception` per Referenz
  - `std::runtime_error` abgeleitet von `std::exception`
  - Polymorphe Klasse, Fangen als Referenz erhält Polymorphie (mehr dazu später)
  - Kann auch unbenannt sein: `catch(std::exception&)`
  - Kopie kann fehlschlagen, würde zu Programmabbruch führen

`throw` by-Value, `catch` by-Reference!

## Try-Catch

```
#include <stdexcept>
#include <iostream>
int foo() {
    try {
        throw std::runtime_error("throwExcept called");
    } catch (std::exception& e) {
        std::cout << e.what() << std::endl;
        throw;
    }
}
```

- Mit `catch(...)` können beliebige Typen gefangen werden
- In der Regel: Nur Unterklassen von `std::exception` werfen und fangen

## Try-Catch

```
#include <stdexcept>
#include <iostream>
int foo() {
    try {
        throw std::runtime_error("throwExcept called");
    } catch(std::exception& e){
        std::cout<<e.what()<<std::endl;
        throw;
    }
}
```

- Im Catch-Block kann die Exception mit `throw`; erneut geworfen werden
  - Sinnvoll, wenn die Exception nicht gehandelt werden kann
- Rethrow auch mit `throw e`; möglich
  - Nicht empfehlenswert, da Ursprung der Exception nicht mitgeschleift wird (für Debugging)



# Nachteile von Exceptions

- Exceptions haben nicht nur Vorteile gegenüber Return Codes

# Nachteile von Exceptions

- Exceptions haben nicht nur Vorteile gegenüber Return Codes
- Komplexität
  - Mit Exceptions: Behandlung von Fehlerfällen nicht lokal
  - Prüfung, ob alle Fehlerfälle abgedeckt sind, ist schwieriger

# Nachteile von Exceptions

- Exceptions haben nicht nur Vorteile gegenüber Return Codes
- Komplexität
  - Mit Exceptions: Behandlung von Fehlerfällen nicht lokal
  - Prüfung, ob alle Fehlerfälle abgedeckt sind, ist schwieriger
  - Problematisch für systemkritische Software
    - Strikte Verifikationsprozesse reduzieren das Risiko unbehandelter Fehler bei Return Codes
    - Unbehandelte Exception in z.B. der Steuerung einer Rakete kann katastrophal enden

# Nachteile von Exceptions

- Exceptions haben nicht nur Vorteile gegenüber Return Codes
- Komplexität
  - Mit Exceptions
  - Prüfung, ob alle
  - Problematisch
    - Strikte Ver
    - Unbehand



bei Return Codes  
ophal enden

## Nachteile von Exceptions

- Exceptions haben nicht nur Vorteile gegenüber Return Codes
- Komplexität
  - Mit Exceptions: Behandlung von Fehlerfällen nicht lokal
  - Prüfung, ob alle Fehlerfälle abgedeckt sind, ist schwieriger
  - Problematisch für systemkritische Software
    - Strikte Verifikationsprozesse reduzieren das Risiko unbehandelter Fehler bei Return Codes
    - Unbehandelte Exception in z.B. der Steuerung einer Rakete kann katastrophal enden
- Laufzeitoverhead
  - Overhead für normalen Codefluss gering bis nicht vorhanden
  - Werfen und Fangen von Exceptions in der Regel sehr teuer
  - Für Operationen, die häufig fehlschlagen, möglicherweise nicht optimal
  - **Niemals Exceptions für normalen Codefluss verwenden!**
    - Wirklich **nie!**

# Assertions

- Ausnahmefälle haben zwei Ursachen
  - Fehlerhafte Eingaben
  - Programmierfehler

# Assertions

- Ausnahmefälle haben zwei Ursachen
  - Fehlerhafte Eingaben
  - Programmierfehler
- Idealszenario: Programmierfehler treten in fertiger Software nicht auf

# Assertions

- Ausnahmefälle haben zwei Ursachen
  - Fehlerhafte Eingaben
  - Programmierfehler
- Idealszenario: Programmierfehler treten in fertiger Software nicht auf
- Überprüfungen zum Abfangen von Fehlern kosten Leistung
  - Unnötige Überprüfungen daher nicht wünschenswert



# Assertions

- Ausnahmefälle haben zwei Ursachen
  - Fehlerhafte Eingaben
  - Programmierfehler
- Idealszenario: Programmierfehler treten in fertiger Software nicht auf
- Überprüfungen zum Abfangen von Fehlern kosten Leistung
  - Unnötige Überprüfungen daher nicht wünschenswert
- Assertions dienen zum Abfangen von Programmierfehlern
  - Werden in Release-Versionen in der Regel rauskompiliert
  - Implementiert über Macros

# Assertions

- Ausnahmefälle haben zwei Ursachen
  - Fehlerhafte Eingaben
  - Programmierfehler
- Idealszenario: Programmierfehler treten in fertiger Software nicht auf
- Überprüfungen zum Abfangen von Fehlern kosten Leistung
  - Unnötige Überprüfungen daher nicht wünschenswert
- Assertions dienen zum Abfangen von Programmierfehlern
  - Werden in Release-Versionen in der Regel rauskompiliert
  - Implementiert über Macros
- Verwendung: `assert(expr) ;`
  - Programm wird beendet, wenn *expr* zu `false` evaluiert
  - Definiert in `cassert`
  - Wird durch Definition des Macros `NDEBUG` deaktiviert
  - **Achtung:** Seiteneffekte in *expr* können zu unterschiedlichem Verhalten zwischen Debug und Releasebuild führen

# Assertions

- Ausnahmefälle haben zwei Ursachen
  - Fehlerhafte Eingaben
  - Programmierfehler
- Idealszenario: Programmierfehler treten in fertiger Software nicht auf

- Überprüfungen zu
  - Unnötige Überprüfungen

- Assertions dienen

- Werden in Release-Builds deaktiviert
  - Implementiert als `assert`

- Verwendung: `assert`

- Programm wird beendet, wenn `expr` zu `false` evaluiert
  - Definiert in `cassert`
  - Wird durch Definition des Macros `NDEBUG` deaktiviert
  - **Achtung:** Seiteneffekte in `expr` können zu unterschiedlichem Verhalten zwischen Debug und Releasebuild führen

```
#include <cassert>
#include <string>
void processString(std::string* dataPtr)
{
    assert(dataPtr != nullptr);
    assert(!dataPtr->empty());
    //do work...
}
```



IMS  
Institut für Mikroelektronische Systeme  
Leibniz Universität Hannover



Leibniz  
Universität  
Hannover

# Allgemeines zu den Aufgaben

## Dateien im Programmierprojekt II

- Zur Vermeidung von Konflikten sollten die folgenden Punkte beachtet werden:
  - 1 Zu Aufgaben gehörige Dateien nur durch Pull von `common` in das Repository bringen, nicht `ppti-common` klonen und Dateien manuell in das eigene Repository kopieren
  - 2 Nur die Dateien verändern, in denen Aufgaben bearbeitet werden sollen
  - 3 Die Verzeichnisstruktur nicht verändern

## Dateien im Programmierprojekt II

- Zur Vermeidung von Konflikten sollten die folgenden Punkte beachtet werden:
  - 1 Zu Aufgaben gehörige Dateien nur durch Pull von `common` in das Repository bringen, nicht `ppti-common` klonen und Dateien manuell in das eigene Repository kopieren
  - 2 Nur die Dateien verändern, in denen Aufgaben bearbeitet werden sollen
  - 3 Die Verzeichnisstruktur nicht verändern
- Wenn dies in der Vergangenheit missachtet: Neue Aufgabe frühzeitig pullen und überprüfen, ob alles sauber zusammengefügt wurde
  - Bei Problemen: Melden Sie sich bei uns!

## Weitere Regeln für die C++-Programmierung

- In Konstruktoren: Alle Membervariablen in der Initialisierungsliste initialisieren!
  - Reihenfolge entsprechend der Definitionsreihenfolge der Member in der Klasse

## Weitere Regeln für die C++-Programmierung

- In Konstruktoren: Alle Membervariablen in der Initialisierungsliste initialisieren!
  - Reihenfolge entsprechend der Definitionsreihenfolge der Member in der Klasse
- Für Nullpointer `nullptr` und nicht `NULL` oder `0` verwenden!



## Weitere Regeln für die C++-Programmierung

- In Konstruktoren: Alle Membervariablen in der Initialisierungsliste initialisieren!
  - Reihenfolge entsprechend der Definitionsreihenfolge der Member in der Klasse
- Für Nullpointer `nullptr` und nicht `NULL` oder `0` verwenden!
- Kein manuelles Speichermanagement verwenden
  - Speicher immer durch RAII-Objekte, also Container oder Smartpointer, verwalten lassen
  - Keine direkten Aufrufe von `new` und `delete` im eigenen Code, stattdessen `std::make_unique` oder `std::make_shared` verwenden

## Weitere Regeln für die C++-Programmierung

- In Konstruktoren: Alle Membervariablen in der Initialisierungsliste initialisieren!
  - Reihenfolge entsprechend der Definitionsreihenfolge der Member in der Klasse
- Für Nullpointer `nullptr` und nicht `NULL` oder `0` verwenden!
- Kein manuelles Speichermanagement verwenden
  - Speicher immer durch RAII-Objekte, also Container oder Smartpointer, verwalten lassen
  - Keine direkten Aufrufe von `new` und `delete` im eigenen Code, stattdessen `std::make_unique` oder `std::make_shared` verwenden

Nichtbeachtung ohne **überzeugende** Begründung kann zu Punktabzug führen!