

# Systemnahe Programmierung in C/C++

Peter Pehler

`peter.pehler@symetrion.com`

Lehrstuhl für Datenbanken und Informationssysteme  
Fakultät für Mathematik und Informatik

2006-12-13

# Agenda

- 1 new und delete Operatoren
- 2 Allokatoren
- 3 Exception
- 4 Aufgabe

1 new und delete Operatoren

2 Allokatoren

3 Exception

4 Aufgabe

# new, delete Operatorfunktionen

- Durch das Einbinden von `<new>` werden folgende Operatorfunktionen deklariert:

```
void* operator new(std::size_t) throw (std::bad_alloc);
void* operator new[](std::size_t) throw (std::bad_alloc);
void operator delete(void*) throw();
void operator delete[](void*) throw();
void* operator new(std::size_t, const std::nothrow_t&) throw();
void* operator new[](std::size_t, const std::nothrow_t&) throw();
void operator delete(void*, const std::nothrow_t&) throw();
void operator delete[](void*, const std::nothrow_t&) throw();

inline void* operator new(std::size_t, void* __p) throw() { return __p; }
inline void* operator new[](std::size_t, void* __p) throw() { return __p; }

inline void operator delete (void*, void*) throw() { }
inline void operator delete[](void*, void*) throw() { }
```

- Enthalten sind, aus der letzten Vorlesung bekannten,
  - scalar- und array-new jeweils mit und ohne `std::bad_alloc`
  - scalar- und array-delete jeweils ohne Exception

- Durch das Einbinden von `<new>` werden folgende Operatorfunktionen deklariert:

```
void* operator new(std::size_t) throw (std::bad_alloc);
void* operator new[](std::size_t) throw (std::bad_alloc);
void operator delete(void*) throw();
void operator delete[](void*) throw();
void* operator new(std::size_t, const std::nothrow_t&) throw();
void* operator new[](std::size_t, const std::nothrow_t&) throw();
void operator delete(void*, const std::nothrow_t&) throw();
void operator delete[](void*, const std::nothrow_t&) throw();

inline void* operator new(std::size_t, void* __p) throw() { return __p; }
inline void* operator new[](std::size_t, void* __p) throw() { return __p; }

inline void operator delete (void*, void*) throw() { }
inline void operator delete[](void*, void*) throw() { }
```

- Enthalten sind, aus der letzten Vorlesung bekannten,
  - scalar- und array-new jeweils mit und ohne `std::bad_alloc`
  - scalar- und array-delete jeweils ohne Exception

- Als ersten Parameter bekommt `new` immer die Größe des benötigten Speicherbereiches in Bytes übergeben.
- Die Zeile

```
int* pInt = new int;
```

setzt der Compiler um als (pseudo-Code)

```
int* pInt = operator new( benoetigter_speicher_in_byte );
```

- Dabei gilt für

```
int* pInt = new int;
```

nicht zwangsweise:

```
int* pInt = operator new( sizeof(int) );
```

(siehe array-new)

# new: size Parameter

- Als ersten Parameter bekommt `new` immer die Größe des benötigten Speicherbereiches in Bytes übergeben.
- Die Zeile

```
int* pInt = new int;
```

setzt der Compiler um als (pseudo-Code)

```
int* pInt = operator new( benoetigter_speicher_in_byte );
```

- Dabei gilt für

```
int* pInt = new int;
```

nicht zwangsweise:

```
int* pInt = operator new( sizeof(int) );
```

(siehe array-new)

# new: size Parameter

- Als ersten Parameter bekommt `new` immer die Größe des benötigten Speicherbereiches in Bytes übergeben.
- Die Zeile

```
int* pInt = new int;
```

setzt der Compiler um als (pseudo-Code)

```
int* pInt = operator new( benoetigter_speicher_in_byte );
```

- Dabei gilt für

```
int* pInt = new int;
```

nicht zwangsweise:

```
int* pInt = operator new( sizeof(int) );
```

(siehe array-new)

- Als ersten Parameter bekommt `new` immer die Größe des benötigten Speicherbereiches in Bytes übergeben.
- Die Zeile

```
int* pInt = new int;
```

setzt der Compiler um als (pseudo-Code)

```
int* pInt = operator new( benoetigter_speicher_in_byte );
```

- Dabei gilt für

```
int* pInt = new int;
```

nicht zwangsweise:

```
int* pInt = operator new( sizeof(int) );
```

(siehe array-new)

- Wie sich als Parameter `nothrow` übergeben läßt, kann eine beliebige Parameterliste übergeben werden. Beispiel:

```
struct special_t {} special;

void* operator new( std::size_t size, const special_t& s, std::size_t limit )
{
    /*Implementation*/
}

...
string* pString = new (special, 5) string;
```

- Veranschaulichung:

`new ( param1, param2, ... ) TYPE;`  
`operator new( size_t, param1, param2, ...)`

..... Berechnung des Speicherbedarfs, `sizeof( TYPE ) + sizeof( HILFSDATEN )`

- Wie sich als Parameter `nothrow` übergeben läßt, kann eine beliebige Parameterliste übergeben werden. Beispiel:

```
struct special_t {} special;

void* operator new( std::size_t size, const special_t& s, std::size_t limit )
{
    /*Implementation*/
}

...
string* pString = new (special, 5) string;
```

- Veranschaulichung:

`new ( param1, param2, ... ) TYPE;`  
`operator new( size_t, param1, param2, ...)`

..... Berechnung des Speicherbedarfs, `sizeof( TYPE ) + sizeof( HILFSDATEN )`

- Wie sich als Parameter `nothrow` übergeben läßt, kann eine beliebige Parameterliste übergeben werden. Beispiel:

```
struct special_t {} special;

void* operator new( std::size_t size, const special_t& s, std::size_t limit )
{
    /*Implementation*/
}

...
string* pString = new (special, 5) string;
```

- Veranschaulichung:

`new ( param1, param2, ... ) TYPE;`  
`operator new( size_t, param1, param2, ...)`

..... Berechnung des Speicherbedarfs, `sizeof( TYPE ) + sizeof( HILFSDATEN )`

- Um Speicher für mehrere Instanzen bereitzustellen, existiert der array-new Operator

```
void* operator new[]( std::size_t size )
```

- Die Übergabe von Parametern an `new[]` ist äquivalent zu `new`.
- Die Berechnung von des ersten Paramters 'size' wird hier an einem Beispiel vorgestellt ( 32Bit, g++ ):

```
class CMyClass{
    uint32_t mValue;
public:
    CMyClass(){ std::cout << __PRETTY_FUNCTION__ << std::endl;}
};

...
CMyClass* pArray = new CMyClass[5];
```

- Welchen Wert hat `size` für den daraus resultierenden Aufruf von `void* operator new[](std::size_t size) throw(std::bad_alloc)?`

- Um Speicher für mehrere Instanzen bereitzustellen, existiert der array-new Operator

```
void* operator new[]( std::size_t size )
```

- Die Übergabe von Parametern an `new[]` ist äquivalent zu `new`.
- Die Berechnung von des ersten Paramters 'size' wird hier an einem Beispiel vorgestellt ( 32Bit, g++ ):

```
class CMyClass{
    uint32_t mValue;
public:
    CMyClass(){ std::cout << __PRETTY_FUNCTION__ << std::endl;}
};

...
CMyClass* pArray = new CMyClass[5];
```

- Welchen Wert hat size für den daraus resultierenden Aufruf von `void* operator new[](std::size_t size) throw(std::bad_alloc)?`

- Um Speicher für mehrere Instanzen bereitzustellen, existiert der array-new Operator

```
void* operator new[]( std::size_t size )
```

- Die Übergabe von Parametern an `new[]` ist äquivalent zu `new`.
- Die Berechnung von des ersten Paramters 'size' wird hier an einem Beispiel vorgestellt ( 32Bit, g++ ):

```
class CMyClass{
    uint32_t mValue;
public:
    CMyClass(){ std::cout << __PRETTY_FUNCTION__ << std::endl;}
};

...
CMyClass* pArray = new CMyClass[5];
```

- Welchen Wert hat `size` für den daraus resultierenden Aufruf von `void* operator new[](std::size_t size) throw(std::bad_alloc)?`

- Um Speicher für mehrere Instanzen bereitzustellen, existiert der array-new Operator

```
void* operator new[]( std::size_t size )
```

- Die Übergabe von Parametern an `new[]` ist äquivalent zu `new`.
- Die Berechnung von des ersten Paramters 'size' wird hier an einem Beispiel vorgestellt ( 32Bit, g++ ):

```
class CMyClass{
    uint32_t mValue;
public:
    CMyClass(){ std::cout << __PRETTY_FUNCTION__ << std::endl;}
};

...
CMyClass* pArray = new CMyClass[5];
```

- Welchen Wert hat `size` für den daraus resultierenden Aufruf von

```
void* operator new[](std::size_t size) throw(std::bad_alloc)?
```

- Zweites Beispiel:

```
class CMyClass{
    uint32_t mValue;
public:
    CMyClass(){ std::cout << __PRETTY_FUNCTION__ << std::endl;}
    ~CMyClass(){ std::cout << __PRETTY_FUNCTION__ << std::endl;}
};

...
CMyClass* pArray = new CMyClass[5];
```

- Welchen Wert hat `size` für den daraus resultierenden Aufruf von `void* operator new[](std::size_t size) throw(std::bad_alloc)?`
- `CMyClass` besitzt einen Destruktor. Daraus folgt:
  - beim Abräumen des Arrays muß für jedes Element der Destruktor durchlaufen werden,
  - die Information, wieviele Elemente im Array liegen, muß bis zum Freigabezeitpunkt gerettet werden.
- $size = n * sizeof(CMyClass) + sizeof(size_t) = 5 * 4 + 4 = 24$  (32Bit-System, g++, libstdc++-v3)
- In diesem Beispiel ergeben sich 20% overhead.

- Zweites Beispiel:

```
class CMyClass{
    uint32_t mValue;
public:
    CMyClass(){ std::cout << __PRETTY_FUNCTION__ << std::endl;}
    ~CMyClass(){ std::cout << __PRETTY_FUNCTION__ << std::endl;}
};

...
CMyClass* pArray = new CMyClass[5];
```

- Welchen Wert hat `size` für den daraus resultierenden Aufruf von

`void* operator new[](std::size_t size) throw(std::bad_alloc)?`

- CMyClass besitzt einen Destruktor. Daraus folgt:

- beim Abräumen des Arrays muß für jedes Element der Destruktor durchlaufen werden,
- die Information, wieviele Elemente im Array liegen, muß bis zum Freigabezeitpunkt gerettet werden.
- $size = n * sizeof(CMyClass) + sizeof(size_t) = 5 * 4 + 4 = 24$   
(32Bit-System, g++, libstdc++-v3)
- In diesem Beispiel ergeben sich 20% overhead.

- Zweites Beispiel:

```
class CMyClass{
    uint32_t mValue;
public:
    CMyClass(){ std::cout << __PRETTY_FUNCTION__ << std::endl;}
    ~CMyClass(){ std::cout << __PRETTY_FUNCTION__ << std::endl;}
};

...
CMyClass* pArray = new CMyClass[5];
```

- Welchen Wert hat `size` für den daraus resultierenden Aufruf von

`void* operator new[](std::size_t size) throw(std::bad_alloc)?`

- `CMyClass` besitzt einen Destruktor. Daraus folgt:

- beim Abräumen des Arrays muß für jedes Element der Destruktor durchlaufen werden,
- die Information, wieviele Elemente im Array liegen, muß bis zum Freigabezeitpunkt gerettet werden.
- $size = n * sizeof(CMyClass) + sizeof(size\_t) = 5 * 4 + 4 = 24$   
(32Bit-System, g++, libstdc++-v3)
- In diesem Beispiel ergeben sich 20% overhead.

- Zweites Beispiel:

```
class CMyClass{
    uint32_t mValue;
public:
    CMyClass(){ std::cout << __PRETTY_FUNCTION__ << std::endl;}
    ~CMyClass(){ std::cout << __PRETTY_FUNCTION__ << std::endl;}
};

...
CMyClass* pArray = new CMyClass[5];
```

- Welchen Wert hat `size` für den daraus resultierenden Aufruf von

`void* operator new[](std::size_t size) throw(std::bad_alloc)?`

- `CMyClass` besitzt einen Destruktor. Daraus folgt:

- beim Abräumen des Arrays muß für jedes Element der Destruktor durchlaufen werden,
  - die Information, wieviele Elemente im Array liegen, muß bis zum Freigabezeitpunkt gerettet werden.
- $size = n * sizeof(CMyClass) + sizeof(size\_t) = 5 * 4 + 4 = 24$   
(32Bit-System, g++, libstdc++-v3)
- In diesem Beispiel ergeben sich 20% overhead.

- Zweites Beispiel:

```
class CMyClass{
    uint32_t mValue;
public:
    CMyClass(){ std::cout << __PRETTY_FUNCTION__ << std::endl;}
    ~CMyClass(){ std::cout << __PRETTY_FUNCTION__ << std::endl;}
};

...
CMyClass* pArray = new CMyClass[5];
```

- Welchen Wert hat `size` für den daraus resultierenden Aufruf von

`void* operator new[](std::size_t size) throw(std::bad_alloc)?`

- `CMyClass` besitzt einen Destruktor. Daraus folgt:

- beim Abräumen des Arrays muß für jedes Element der Destruktor durchlaufen werden,
- die Information, wieviele Elemente im Array liegen, muß bis zum Freigabezeitpunkt gerettet werden.

- $size = n * sizeof(CMyClass) + sizeof(size\_t) = 5 * 4 + 4 = 24$   
(32Bit-System, g++, libstdc++-v3)

- In diesem Beispiel ergeben sich 20% overhead.

- Zweites Beispiel:

```
class CMyClass{
    uint32_t mValue;
public:
    CMyClass(){ std::cout << __PRETTY_FUNCTION__ << std::endl;}
    ~CMyClass(){ std::cout << __PRETTY_FUNCTION__ << std::endl;}
};

...
CMyClass* pArray = new CMyClass[5];
```

- Welchen Wert hat `size` für den daraus resultierenden Aufruf von `void* operator new[](std::size_t size) throw(std::bad_alloc)?`
- `CMyClass` besitzt einen Destruktor. Daraus folgt:
  - beim Abräumen des Arrays muß für jedes Element der Destruktor durchlaufen werden,
  - die Information, wieviele Elemente im Array liegen, muß bis zum Freigabezeitpunkt gerettet werden.
- $size = n * sizeof(CMyClass) + sizeof(size_t) = 5 * 4 + 4 = 24$  (32Bit-System, g++, libstdc++-v3)
- In diesem Beispiel ergeben sich 20% overhead.

- Wie in Folie 4 dargestellt, lässt sich der operator delete durch

```
void operator delete(void* p);
```

überladen.

- Nach dem C++-Standard kann der operator delete auch durch

```
void operator delete(void* p, std::size_t size);
```

überladen werden.

- Der Compiler stellt dabei sicher, dass scalar-new und scalar-delete als auch array-new und array-delete jeweils mit dem gleichen Wert für size aufgerufen werden.

# delete Operatorfunktion

- Wie in Folie 4 dargestellt, lässt sich der operator delete durch

```
void operator delete(void* p);
```

überladen.

- Nach dem C++-Standard kann der operator delete auch durch

```
void operator delete(void* p, std::size_t size);
```

überladen werden.

- Der Compiler stellt dabei sicher, dass scalar-new und scalar-delete als auch array-new und array-delete jeweils mit dem gleichen Wert für size aufgerufen werden.

- Wie in Folie 4 dargestellt, lässt sich der operator delete durch

```
void operator delete(void* p);
```

überladen.

- Nach dem C++-Standard kann der operator delete auch durch

```
void operator delete(void* p, std::size_t size);
```

überladen werden.

- Der Compiler stellt dabei sicher, dass scalar-new und scalar-delete als auch array-new und array-delete jeweils mit dem gleichen Wert für size aufgerufen werden.

- Zusätzlich zu den Operatorfunktionen von new und delete, lassen sich new und delete als statische Operatormethoden überladen.

Deklaration:

```
class MyClass
{
public:
    static void* operator new( std::size_t size ) throw( std::bad_alloc);
    static void* operator new[]( std::size_t size ) throw( std::bad_alloc);
    ...
    static void operator delete( void* p, std::size_t size ) throw( );
    static void operator delete[]( void* p, std::size_t size ) throw( );
    ...
};
```

- Die Übergabe von zusätzlichen Parametern an operator new erfolgt auch hier nach dem auf Folie 10 dargestellten Prinzip.

- Zusätzlich zu den Operatorfunktionen von new und delete, lassen sich new und delete als statische Operatormethoden überladen.

Deklaration:

```
class CMyClass
{
public:
    static void* operator new( std::size_t size ) throw( std::bad_alloc);
    static void* operator new[]( std::size_t size ) throw( std::bad_alloc);
    ...
    static void operator delete( void* p, std::size_t size ) throw( );
    static void operator delete[]( void* p, std::size_t size ) throw( );
    ...
};
```

- Die Übergabe von zusätzlichen Parametern an operator new erfolgt auch hier nach dem auf Folie 10 dargestellten Prinzip.

# new/delete Operatormethode

- Ist eine Operatormethode für new oder delete überladen, wird diese vorrangig gegenüber der entsprechenden Operatorfunktion verwendet:

```
1  class MyBase{
2      int mValue;
3      public:
4          MyBase();
5          virtual ~MyBase();
6
7          static void* operator new( std::size_t size ) throw( std::bad_alloc);
8          static void operator delete( void* p, std::size_t size);
9  };
10
11 class MyDerived : public MyBase{
12     int mValue;
13     public:
14         MyDerived();
15         ~MyDerived();
16 };
17
18 void* operator new( std::size_t size) throw(std::bad_alloc);
19 void operator delete( void* p, std::size_t size) throw();
20
21 int main( int argc, char* argv[] )
22 {
23     MyBase* pBase = new MyBase;
24     MyDerived* pDerived = new MyDerived;
25     delete pBase;
26     delete pDerived;
27 }
```

# new/delete Operatormethode

- Ist eine Operatormethode für new oder delete überladen, wird diese vorrangig gegenüber der entsprechenden Operatorfunktion verwendet:

```
1  class MyBase{
2      int mValue;
3      public:
4          MyBase();
5          virtual ~MyBase();
6
7          static void* operator new( std::size_t size ) throw( std::bad_alloc);
8          static void operator delete( void* p, std::size_t size);
9  };
10
11 class MyDerived : public MyBase{
12     int mValue;
13     public:
14         MyDerived();
15         ~MyDerived();
16 };
17
18 void* operator new( std::size_t size) throw(std::bad_alloc);
19 void operator delete( void* p, std::size_t size) throw();
20
21 int main( int argc, char* argv[] )
22 {
23     MyBase* pBase = new MyBase;
24     MyDerived* pDerived = new MyDerived;
25     delete pBase;
26     delete pDerived;
27 }
```

- Welche Methoden/Funktionen werden in welcher Reihenfolge durchlaufen?



- Wird eine operator new Methode überladen, müssen alle Varianten überladen werden. Beispiel:

```
class CMyBase{
    int mValue;
public:
    CMyBase();
    virtual ~CMyBase();

    static void* operator new( std::size_t size ) throw( std::bad_alloc);
    static void operator delete( void* p, std::size_t size) throw();
};
...
char buffer[100];
CMyBase* pBase = new (buffer) CMyBase;
```

- Der Compiler (gcc-Version 4.1.1) sucht nach einer operator new Methode, die der zusätzlicher Parameter char\* (Typkonvertierung möglich) übergeben werden kann.

- Wird eine operator new Methode überladen, müssen alle Varianten überladen werden. Beispiel:

```
class CMyBase{
    int mValue;
public:
    CMyBase();
    virtual ~CMyBase();

    static void* operator new( std::size_t size ) throw( std::bad_alloc);
    static void operator delete( void* p, std::size_t size) throw();
};
...
char buffer[100];
CMyBase* pBase = new (buffer) CMyBase;
```

- Der Compiler (gcc-Version 4.1.1) sucht nach einer operator new Methode, die der zusätzlicher Parameter char\* (Typkonvertierung möglich) übergeben werden kann.

- Wird eine operator new Methode überladen, müssen alle Varianten überladen werden. Beispiel:

```
class CMyBase{
    int mValue;
public:
    CMyBase();
    virtual ~CMyBase();

    static void* operator new( std::size_t size ) throw( std::bad_alloc);
    static void operator delete( void* p, std::size_t size) throw();
};
...
char buffer[100];
CMyBase* pBase = new (buffer) CMyBase;
```

- Der Compiler (gcc-Version 4.1.1) sucht nach einer operator new Methode, die der zusätzlicher Parameter char\* (Typkonvertierung möglich) übergeben werden kann.

- Soll lediglich Auswahl an Operatormethoden überladen werden, kann man sich mit Methodentemplates behelfen:

```
1 class CMyBase{
2     public:
3
4         static void* operator new( std::size_t size ) throw( std::bad_alloc );
5         static void* operator new( std::size_t size, const std::nothrow_t& ) throw();
6
7         template< typename T >
8         static void* operator new( std::size_t size, T& t ) throw( std::bad_alloc )
9         { return ::operator new(size,t); }
10        ...
11};
```

- Alternativ zu Zeile 8 { return ::operator new(size,t); } kann man auch die array new Funktion verwenden:

```
{ return ::new (t) char[size]; }
```

- Soll lediglich Auswahl an Operatormethoden überladen werden, kann man sich mit Methodentemplates behelfen:

```
1 class CMyBase{
2     public:
3
4         static void* operator new( std::size_t size ) throw( std::bad_alloc );
5         static void* operator new( std::size_t size, const std::nothrow_t& ) throw();
6
7         template< typename T >
8         static void* operator new( std::size_t size, T& t ) throw( std::bad_alloc )
9         { return ::operator new(size,t); }
10        ...
11};
```

- Alternativ zu Zeile 8 { return ::operator new(size,t); } kann man auch die array new Funktion verwenden:

```
{ return ::new (t) char[size]; }
```

- Ein überladener new Operator
  - 1 ruft nicht den Konstruktor auf;
  - 2 darf keinen bestimmten Wert für `size` erwarten;
  - 3 muß als Ausnahme eine Instanz von `std::bad_cast` (oder davon abgeleitet) werfen, wenn keine Instanz von `std::nothrow_t` übergeben wurde;
  - 4 darf keine Ausnahme werfen, wenn eine Instanz von `std::nothrow_t` übergeben wurde.
  - 5 muß den aktuellen `new_handler` rufen, falls installiert.
- Insbesondere die standardkonforme Unterstützung des `new_handler` (Punkt 5) kann zu Problemen in der Implementierung eines `new` Operator führen.

- Ein überladener new Operator
  - 1 ruft nicht den Konstruktor auf;
  - 2 darf keinen bestimmten Wert für `size` erwarten;
  - 3 muß als Ausnahme eine Instanz von `std::bad_cast` (oder davon abgeleitet) werfen, wenn keine Instanz von `std::nothrow_t` übergeben wurde;
  - 4 darf keine Ausnahme werfen, wenn eine Instanz von `std::nothrow_t` übergeben wurde.
  - 5 muß den aktuellen `new_handler` rufen, falls installiert.
- Insbesondere die standardkonforme Unterstützung des `new_handler` (Punkt 5) kann zu Problemen in der Implementierung eines `new` Operator führen.

- Ein überladener new Operator
  - 1 ruft nicht den Konstruktor auf;
  - 2 darf keinen bestimmten Wert für `size` erwarten;
  - 3 muß als Ausnahme eine Instanz von `std::bad_cast` (oder davon abgeleitet) werfen, wenn keine Instanz von `std::nothrow_t` übergeben wurde;
  - 4 darf keine Ausnahme werfen, wenn eine Instanz von `std::nothrow_t` übergeben wurde.
  - 5 muß den aktuellen `new_handler` rufen, falls installiert.
- Insbesondere die standardkonforme Unterstützung des `new_handler` (Punkt 5) kann zu Problemen in der Implementierung eines `new` Operator führen.

- Ein überladener `new` Operator
  - 1 ruft nicht den Konstruktor auf;
  - 2 darf keinen bestimmten Wert für `size` erwarten;
  - 3 muß als Ausnahme eine Instanz von `std::bad_cast` (oder davon abgeleitet) werfen, wenn keine Instanz von `std::nothrow_t` übergeben wurde;
  - 4 darf keine Ausnahme werfen, wenn eine Instanz von `std::nothrow_t` übergeben wurde.
  - 5 muß den aktuellen `new_handler` rufen, falls installiert.
- Insbesondere die standardkonforme Unterstützung des `new_handler` (Punkt 5) kann zu Problemen in der Implementierung eines `new` Operator führen.

- Ein überladener `new` Operator
  - 1 ruft nicht den Konstruktor auf;
  - 2 darf keinen bestimmten Wert für `size` erwarten;
  - 3 muß als Ausnahme eine Instanz von `std::bad_cast` (oder davon abgeleitet) werfen, wenn keine Instanz von `std::nothrow_t` übergeben wurde;
  - 4 darf keine Ausnahme werfen, wenn eine Instanz von `std::nothrow_t` übergeben wurde.
  - 5 muß den aktuellen `new_handler` rufen, falls installiert.
- Insbesondere die standardkonforme Unterstützung des `new_handler` (Punkt 5) kann zu Problemen in der Implementierung eines `new` Operator führen.

- Ein überladener delete Operator
  - 1 ruft nicht den Destruktor auf;
  - 2 darf keinen bestimmten Wert für size erwarten;
  - 3 darf keine Ausnahme werfen.

- Ein überladener delete Operator
  - 1 ruft nicht den Destruktor auf;
  - 2 darf keinen bestimmten Wert für size erwarten;
  - 3 darf keine Ausnahme werfen.

- Ein überladener delete Operator
  - 1 ruft nicht den Destruktor auf;
  - 2 darf keinen bestimmten Wert für size erwarten;
  - 3 darf keine Ausnahme werfen.

1 new und delete Operatoren

2 Allokatoren

3 Exception

4 Aufgabe

- Allokatoren sind eine Abstraktionsschicht um Arbeitsspeicher anzufordern und frei zu geben.
- Alle Container der STL sind mit Allokatoren parameterisierbar.
- Über die standardisierte Schnittstelle sind die Details ( wie wird Speicher angefordert, Garbage-Collection, etc.) verborgen.
- Eigene Logiken für die Reservierung und Freigabe von Speicher sollten durch die Erstellung eines, dem Standard konformen, Allokator umgesetzt werden.

- Allokatoren sind eine Abstraktionsschicht um Arbeitsspeicher anzufordern und frei zu geben.
- Alle Container der STL sind mit Allokatoren parameterisierbar.
- Über die standardisierte Schnittstelle sind die Details ( wie wird Speicher angefordert, Garbage-Collection, etc.) verborgen.
- Eigene Logiken für die Reservierung und Freigabe von Speicher sollten durch die Erstellung eines, dem Standard konformen, Allokator umgesetzt werden.

- Allokatoren sind eine Abstraktionsschicht um Arbeitsspeicher anzufordern und frei zu geben.
- Alle Container der STL sind mit Allokatoren parameterisierbar.
- Über die standardisierte Schnittstelle sind die Details ( wie wird Speicher angefordert, Garbage-Collection, etc.) verborgen.
- Eigene Logiken für die Reservierung und Freigabe von Speicher sollten durch die Erstellung eines, dem Standard konformen, Allokator umgesetzt werden.

- Allokatoren sind eine Abstraktionsschicht um Arbeitsspeicher anzufordern und frei zu geben.
- Alle Container der STL sind mit Allokatoren parameterisierbar.
- Über die standardisierte Schnittstelle sind die Details ( wie wird Speicher angefordert, Garbage-Collection, etc.) verborgen.
- Eigene Logiken für die Reservierung und Freigabe von Speicher sollten durch die Erstellung eines, dem Standard konformen, Allokator umgesetzt werden.

- Der Standardallokator ist deklariert als:

```
template< typename T >  
class allocator{  
...  
};
```

- T ist der Typ, für den der Allokator die Speicherbehandlung bereitstellt.
- Für einen Allokator A existiert ein Typ T, für den A die Speicherbehandlung übernimmt.
- Jeder Allokator stellt den Typnamen zur Verfügung, für den er die Speicherbehandlung bereitstellt:

```
typedef allocator< int > allocator_t;  
allocator_t::value_type; //value_type ist int
```

Ein Allokator kann Speicher auf einem privaten heap verwalten, oder in einer gesonderten Struktur. Dies kann erfordern, dass über spezielle Zeiger (smart pointer) auf die Objekte zugegriffen werden muß.

Ein Allokator definiert deshalb folgende vier Typen:

`pointer` verhält sich wie ein Zeiger auf T.

`const_pointer` verhält sich wie ein konstanter Zeiger auf T.

`reference` verhält sich wie eine Referenz auf T.

`const_reference` verhält sich wie eine konstante Referenz auf T.

`allocator_t::pointer` ist also nicht zwangsweise das Gleiche wie `T*`.

Ein Allokator kann Speicher auf einem privaten heap verwalten, oder in einer gesonderten Struktur. Dies kann erfordern, dass über spezielle Zeiger (smart pointer) auf die Objekte zugegriffen werden muß.

Ein Allokator definiert deshalb folgende vier Typen:

`pointer` verhält sich wie ein Zeiger auf T.

`const_pointer` verhält sich wie ein konstanter Zeiger auf T.

`reference` verhält sich wie eine Referenz auf T.

`const_reference` verhält sich wie eine konstante Referenz auf T.

`allocator_t::pointer` ist also nicht zwangsweise das Gleiche wie `T*`.

Ein Allokator kann Speicher auf einem privaten heap verwalten, oder in einer gesonderten Struktur. Dies kann erfordern, dass über spezielle Zeiger (smart pointer) auf die Objekte zugegriffen werden muß.

Ein Allokator definiert deshalb folgende vier Typen:

`pointer` verhält sich wie ein Zeiger auf `T`.

`const_pointer` verhält sich wie ein konstanter Zeiger auf `T`.

`reference` verhält sich wie eine Referenz auf `T`.

`const_reference` verhält sich wie eine konstante Referenz auf `T`.

`allocator_t::pointer` ist also nicht zwangsweise das Gleiche wie `T*`.

Ein Allokator kann Speicher auf einem privaten heap verwalten, oder in einer gesonderten Struktur. Dies kann erfordern, dass über spezielle Zeiger (smart pointer) auf die Objekte zugegriffen werden muß.

Ein Allokator definiert deshalb folgende vier Typen:

`pointer` verhält sich wie ein Zeiger auf T.

`const_pointer` verhält sich wie ein konstanter Zeiger auf T.

`reference` verhält sich wie eine Referenz auf T.

`const_reference` verhält sich wie eine konstante Referenz auf T.

`allocator_t::pointer` ist also nicht zwangsweise das Gleiche wie T\*.

Ein Allokator kann Speicher auf einem privaten heap verwalten, oder in einer gesonderten Struktur. Dies kann erfordern, dass über spezielle Zeiger (smart pointer) auf die Objekte zugegriffen werden muß.

Ein Allokator definiert deshalb folgende vier Typen:

`pointer` verhält sich wie ein Zeiger auf T.

`const_pointer` verhält sich wie ein konstanter Zeiger auf T.

`reference` verhält sich wie eine Referenz auf T.

`const_reference` verhält sich wie eine konstante Referenz auf T.

`allocator_t::pointer` ist also nicht zwangsweise das Gleiche wie `T*`.

`pointer address(reference x) const` Konvertiert eine Instanz vom Typ `reference` zu dem Typ `pointer`.

`const_pointer address(const_reference x) const` sinngemäß zum Vorherigen  
Speicheranforderung erfolgt über die folgende Methode:

```
pointer allocate( size_type n, const void* hint = 0)
```

mit den folgenden Eigenschaften:

- es wird Speicher für `n` T-Objekte reserviert,
- die Objekte werden nicht konstruiert,
- das Resultat ist ein `random access iterator`,
- der Standardallokator verwendet `::operator new(size)`, daher ist die Ausnahme `std::bad_alloc` möglich,
- die Übergabe des Parameter `hint` suggeriert dem Allokator, dass der neue Speicherbereich möglichst nah an dem Speicherbereich liegen soll, auf den `hint` zeigt.

Der mit `allocate` angeforderte Speicherbereich wird mit der Methode `deallocate` wieder freigegeben. `deallocate` ist deklariert als:

```
void deallocate( pointer p, size_type n )
```

mit den folgenden Eigenschaften:

- es wird Speicher für `n` Objekte des Types frei gegeben,
- `n` muß den gleichen Wert haben, wie bei der zugehörigen `allocate` Methode,
- die Objekte werden nicht zerstört (kein Destruktoraufruf),
- die Objekte sind also vorher mittels der Methode `destroy` zu zerstören (siehe Folie 58)

Um auf dem mit `allocate` angeforderte Speicherbereich die Instanzen zu konstruieren, wird die Methode `construct` verwendet:

```
void construct( pointer p, const_reference val )
```

Die Eigenschaften von `construct` sind:

- es wird mittels Copy-Constructor ein Objekt konstruiert,
- das Objekt wird an der Stelle konstruiert, auf die `p` zeigt,
- `p` muss vorher mit `allocate` bestimmt worden sein.

Um eine mit `construct` konstruierte Instanzen zu zerstören, wird `destroy` verwendet:

```
void destroy( pointer p )
```

Die Eigenschaften von `destroy` sind:

- es wird mittels Destruktor ein Objekt zerstört,
- der zugehörige Speicherbereich wird nicht freigegeben,
- anschließend kann der Speicher mit `deallocate` freigegeben werden

`size_type max_size() const throw()` Die Methode `max_size` liefert den größten Wert, mit dem `allocate` sinnvollerweise aufgerufen werden kann.

Jeder Allokator definiert folgende Operatormethoden:

```
template< typename T, typename U >  
bool operator==( const allocator<T>&, const allocator<U>& ) throw();  
  
template< typename T, typename U >  
bool operator!=( const allocator<T>&, const allocator<U>& ) throw();
```

Für zwei Allokatoren a1 und a2 gilt:

- $a1 == a2$  liefert true, wenn mit a1 reservierter Speicher mit a2 freigegeben werden kann,
- $a1 != a2$ , Negation zu  $a1 == a2$ .

Jeder Allokator definiert folgendes Template:

```
template< typename T > struct rebind( typedef allocator<U> other; }
```

- Ein Allokator ist für Speicherbehandlung eines Types zuständig.
- Wie erhält man einen Allokator für einen anderen Typ?

```
typedef allocator< SomeType > SomeAllocator_t;  
  
typedef typename SomeAllocator_t::rebind< OtherType>::other OtherAllocator_t;  
  
template<typename TElem, typename TAllocator = std::allocator< TElem > >  
class CMyContainer  
{  
    ...  
    typedef TAllocator ElementAllocator_t;  
    typedef typename TAllocator::template rebind< Helper_t >::other HelperAllocator_t;  
    ...  
}
```

# Allokator - constructor

Für einen Allokator sind folgende Konstruktoren definiert:

```
// Standardkonstruktor
allocator() throw();

// Copy-Konstruktor
template< typename U >
allocator( const allocator< U >& ) throw();

//Destruktor
~allocator() throw();
```

Ein Container kann also einen Allokator wie folgt verwenden:

```
class CMyContainer
{
private:
    struct Helper_t{ ... };

    typedef TAllocator ElementAllocator_t;
    typedef typename TAllocator::template rebind< Helper_t >::other HelperAllocator_t;
public:
    CMyContainer() throw()
    : mElemAlloc(), mHelpAlloc( mElemAlloc )
    { }
protected:
    ElementAllocator_t mElemAlloc;
    HelperAllocator_t mHelpAlloc;
};
```

# Allokator - constructor

Für einen Allokator sind folgende Konstruktoren definiert:

```
// Standardkonstruktor
allocator() throw();

// Copy-Konstruktor
template< typename U >
allocator( const allocator< U >& ) throw();

//Destruktor
~allocator() throw();
```

Ein Container kann also einen Allokator wie folgt verwenden:

```
class CMyContainer
{
private:
    struct Helper_t{ ... };

    typedef TAllocator ElementAllocator_t;
    typedef typename TAllocator::template rebind< Helper_t >::other HelperAllocator_t;
public:
    CMyContainer() throw()
    : mElemAlloc(), mHelpAlloc( mElemAlloc )
    { }
protected:
    ElementAllocator_t mElemAlloc;
    HelperAllocator_t mHelpAlloc;
};
```

Zu jedem Container ist ermittelbar:

- der Typ des verwendeten Allokators,
- das verwendete Allokatorobjekt selbst.

Typdefinition, die ein Container bereit stellt:

```
container::allocator_type;
```

Methode, um den aktuellen Allokator abzufragen:

```
allocator_type get_allocator() const;
```

Unser kleiner Container sieht dann wie folgt aus:

Zu jedem Container ist ermittelbar:

- der Typ des verwendeten Allokators,
- das verwendete Allokatorobjekt selbst.

Typdefinition, die ein Container bereit stellt:

```
container::allocator_type;
```

Methode, um den aktuellen Allokator abzufragen:

```
allocator_type get_allocator() const;
```

Unser kleiner Container sieht dann wie folgt aus:

Zu jedem Container ist ermittelbar:

- der Typ des verwendeten Allokators,
- das verwendete Allokatorobjekt selbst.

Typdefinition, die ein Container bereit stellt:

```
container::allocator_type;
```

Methode, um den aktuellen Allokator abzufragen:

```
allocator_type get_allocator() const;
```

Unser kleiner Container sieht dann wie folgt aus:

# Allokator - Container Koppelung

```
class CMyContainer
{
private:
    struct Helper_t{ ... };

    typedef TAllocator ElementAllocator_t;
    typedef typename TAllocator::template rebind< Helper_t >::other HelperAllocator_t;

public:
    typedef TAllocator allocator_type;

    CMyContainer() throw()
    : mElemAlloc(), mHelpAlloc( mElemAlloc )
    { }

    allocator_type get_allocator() const
    { return mElemAlloc; }

protected:
    ElementAllocator_t mElemAlloc;
    HelperAllocator_t mHelpAlloc;
};
```

- 1 new und delete Operatoren
- 2 Allokatoren
- 3 Exception**
- 4 Aufgabe

# Exception - Deklaration

- Folgender Grundsatz:  
Jeder kann alles als exception werfen!
- Eine exception wird mit dem Schlüsselwort `throw` ausgelöst.

```
int func( int value = 1)
{ throw 5 * value; }
```

- Eine Funktion oder Methode dekariert, ob sie Ausnahmen auslöst:

`void func();` nicht festgelegt, ob eine Ausnahme auftritt

`void func() throw();` es wird keine Ausnahme auftreten

`void func() throw(int);` es kann eine Ausnahme vom Typ `int` auftreten

`void func() throw(int,std::exception);` es kann eine Ausnahme vom Typ `int` oder vom Typ `std::exception` auftreten.

# Exception - Deklaration

- Folgender Grundsatz:  
Jeder kann alles als exception werfen!
- Eine exception wird mit dem Schlüsselwort `throw` ausgelöst.

```
int func( int value = 1)
{ throw 5 * value; }
```

- Eine Funktion oder Methode dekariert, ob sie Ausnahmen auslöst:

`void func();` nicht festgelegt, ob eine Ausnahme auftritt

`void func() throw();` es wird keine Ausnahme auftreten

`void func() throw(int);` es kann eine Ausnahme vom Typ `int` auftreten

`void func() throw(int,std::exception);` es kann eine Ausnahme vom Typ `int` oder vom Typ `std::exception` auftreten.

# Exception - Behandlung

- Eine nicht gefangene Ausnahme führt zum Programmabbruch.
- Gefangen wird eine Ausnahme mit einer `catch` Anweisung.

```
try{
  throw 5;
  nicht_erreichter_aufruf();
}
catch( int i )
{
  //irgendeine Behandlung
}
```

- Es läßt sich auch eine Behandlung definieren, die für alle Ausnahmen gilt:

```
try{
  throw 5.5;
  nicht_erreichter_aufruf();
}
catch( int i )
{
  //Behandlung fuer Ausnahme vom Typ int
}
catch( ... )
{
  //Behandlung aller uebrig gebliebenen Ausnahmen,
  //kein Zugriff auf das geworfene Objekt!
}
```

# Exception - Behandlung

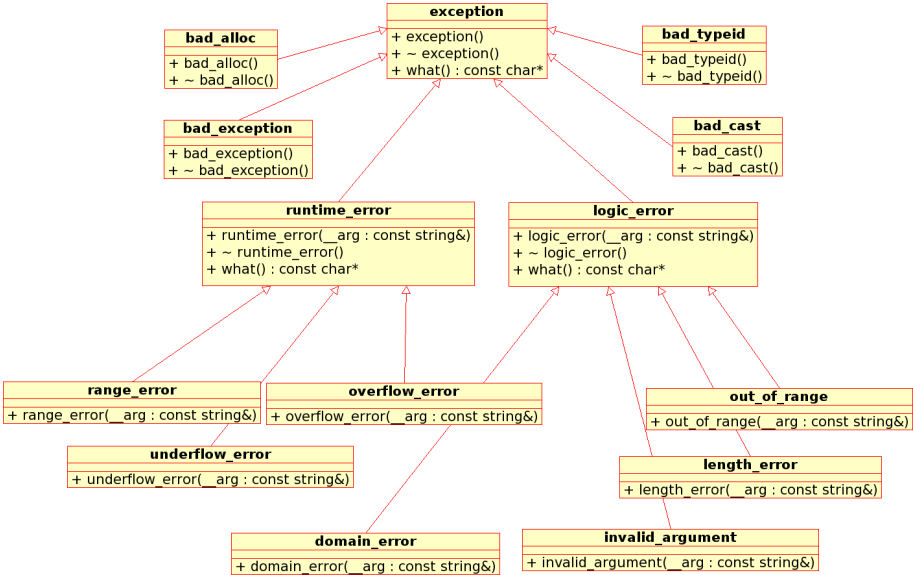
- Eine nicht gefangene Ausnahme führt zum Programmabbruch.
- Gefangen wird eine Ausnahme mit einer `catch` Anweisung.

```
try{
throw 5;
nicht_erreichter_aufruf();
}
catch( int i )
{
//irgendeine Behandlung
}
```

- Es läßt sich auch eine Behandlung definieren, die für alle Ausnahmen gilt:

```
try{
throw 5.5;
nicht_erreichter_aufruf();
}
catch( int i )
{
//Behandlung fuer Ausnahme vom Typ int
}
catch( ... )
{
//Behandlung aller uebrig gebliebenen Ausnahmen,
//kein Zugriff auf das geworfene Objekt!
}
```

# Exception - std::exception



# Exception - std::exception

- Eigene Ausnahmetypen sollten von std::exception erben.
- Ausnahmen als Referenz fangen:

```
class CMyException : public std::exception
{
    int mValue;
public:
    CMyException( int i )
        : mValue( i )
    {};
    ...
};

try{
    throw CMyException( 5 );
    nicht_erreichter_aufruf();
}
catch( const CMyException& e )
{
    //Behandlung fuer Ausnahme
}
catch( const std::exception& e )
{
    //Behandlung aller uebrig gebliebenen Standardausnahmen
}
```

- 1 new und delete Operatoren
- 2 Allokatoren
- 3 Exception
- 4 Aufgabe**

Schreiben Sie einen Allokator mit den folgenden Eigenschaften:

- 1 standardkonform,
- 2 die Speicheranforderungen und Freigaben werden beobachtet (Ausgabe auf `std::cerr`),
- 3 ein `rebind` gibt einen Typ von `std::allocator` zurück.

Wenn Sie Ihren Allokator in einem Container verwenden, sehen Sie die Speicheranforderungen für die Elemente, nicht die für die Hilfsdaten.