Abschnitt II

- Überladen von Funktionen und Operatoren
- Ausnahmebehandlung

Überladene Funktionsnamen

- **Wünschenswert für gleiche Funktionen auf verschiedenen "Objekten" auch gleiche Namen vergeben zu können**

  Beispiel:
  - `"+"  für int-, float-, double-, char-, etc. Addition`
  - `void print(int);  //Gib einen int aus`
  - `void print(const char*);  //Gib einen C-String aus`

- **Der gleiche Name beschreibt eine "Verwandschaft" von Funktionen**
- **Funktionsunterscheidung in C++ : Name plus Typen der Parameter!**

```
void print(double); void print(long);

void f()
{
   print(1L);  // print(???)
   print(1.0)  // print(???)
   print(1);   // print(???)
}
```

Überladene Funktionsnamen
Finden der richtigen Version

- **Test einer Reihe von Kriterien:**
  - Genaue Übereinstimmung: ohne oder nur mit trivialen Konvertierungen (z.B. Feldname nach Zeiger, Funktionsname nach Funktionszeiger und T nach const T)
  - Übereinstimmung mit Promotionen; das heißt integrale Promotionen (bool nach int, char nach int, short nach int und ihre unsigned Gegenstücke), float nach double und double nach long double.
  - Übereinstimmung mit Standard-Konvertierungen (z.B. int nach double, double to int, Abgeleitet* nach Basis *, T* nach void* , int nach unsigned int
  - Übereinstimmung mit benutzerdefinierten Konvertierungen
  - Übereinstimmung mit "..." in einer Funktionsdeklaration
- **Mehrere Übereinstimmungen der gleichen Stufe: Aufruf mehrdeutig**
- **Rückgabetypen werden bei der Auflösung von Überladungen nicht berücksichtigt**
  - Auflösung für einen einzelnen Operator oder Funktionsaufruf soll kontextunabhängig sein

Überladene Funktionsnamen
Auflösung mit Konvertierungen

```
void print(int);
void print(const char*);
void print(double);
void print(long);
void print(char);


void h(char c, int i, short s, float f)
{
   print(c);      // Genaue Übereinstimmung: benutze print(char)
   print(i);      // Genaue Übereinstimmung: benutze print(int)
   print(s);      // Integrale Promotion: benutze print(int)
   print(f);      // float nach double Promotion: print(double)

   print('a');    // Genaue Übereinstimmung: print(char)
   print(49);     // Genaue Übereinstimmung: print(int)
   print(0.0);    // Genaue Übereinstimmung: print(double)
   print("a");    // Genaue Übereinstimmung: print(const char*)
}
```

Überladen
Anwendungsbeispiel: Potenzieren von Zahlen

- **Sicherstellen, daß der einfachste Algorithmus (Funktion) benutzt wird, wenn die Effizienz oder die Genauigkeit der Berechnung für die beteiligten Typen signifikant unterschiedlich ist. Beispiel:**

```
int pow(int, int);
double pow(double, double);

complex pow(double,  complex);
complex pow(complex, int);
complex pow(complex, double);
complex pow(complex, complex);

void k(complex z)
{
   int      i  = pow(2,2);      // Benutze pow(int,     int)
   double   d  = pow(2.0,2.0);  // Benutze pow(double,  double)
   complex  z2 = pow(2,z);      // Benutze pow(double,  complex)
   complex  z3 = pow(z,2);      // Benutze pow(complex, int)
   complex  z4 = pow(z,z);      // Benutze pow(complex, complex)
}
```


Überladen von Operatoren

```
void f( )
{
   rational a = rational(1, 3);
   rational b = rational(1, 5);

   rational c = a + b;
   rational e = a*b + rational(1, 2);
   b = b + c*a;
}
```

```
class rational {  // Rationale Zahlen
{
   int    Zaehler;
   int    Nenner;
public:
   rational (int z, int n) : Zaehler(z), Nenner(n) {}
   rational operator+(rational);
   rational operator*(rational);
};
```

Operator-Funktionen

Für die folgenden Operatoren können neue Funktionen definiert werden:

+	-	*	/	%	^	&
|	~	!	=	<	>	+=
-=	*=	/=	%=	^=	&=	|=
<<	>>	>>=	<<=	==	!=	<=
>=	&&	||	++	--	->*	,
->	[]	( )	new	new[]	delete	delete[]

- **Die Operatoren :: (Bereichsauflösung), . (Elementauswahl) und .* (Elementauswahl durch einen Funktionszeiger) sind nicht durch den Benutzer definierbar**
- **Man kann neben den oben aufgeführten Operatoren keine neuen/ weiteren Operatoren dazudefinieren.**

Ein- und zweistellige Operatoren

- **zweistelliger Operator**
  - nichtstatische Elementfunktion mit einem Parameter: aa.operator@(bb) oder
  - Nicht-Elementfunktion mit zwei Parametern: operator@(aa,bb)
- **einstelliger Operator, egal ob Präfix- oder Postfix-Operator**
  - nicht-statische Elementfunktion ohne Parameter oder
  - Nicht-Elementfunktion mit einem Parameter.
- **Präfix-Operator @**
  - aa.operator@() oder als operator@ (aa)
- **Postfix-Operator @**
  - aa.operator@ (int) oder als operator@ (aa, int)
- **Ein Operator kann nur entsprechend der für ihn in der Grammatik definierten Syntax deklariert werden**
  - kein einstelliges % und kein dreistelliges +

Beispiele

```
class X {
// Elementfunktionen mit implizitem this Zeiger)
   X*operator&();       // einstelliger Präfix-Operator & (Adresse)
   X operator&(X);           // zweistelliges & (Und)
   X operator++(int);        // Postfix-Inkrement
   X operator&(X,X);         // Fehler dreistellig
   X operator/();            // Fehler einstelliges /
};

// Nicht-Elementfunktionen
X operator-(X);              // einstelliges Präfix-Minus
X operator-(X,X);            // zweistelliges Minus
X operator--(X&, int);       // Postfix-Dekrement
X operator-();               // Fehler: kein Operand
X operator-(X,X,X);          // Fehler: dreistellig
X operator%(X);              // Fehler: einstelliges %
```


Umfangreicherer Typ für rationale Zahlen

- **Erwartung:**

```
void f()
{
  rational  a = rational(1,2);
  rational  b = 3;
  rational  c = a+2;
  rational  d = 2+b;
  rational  e = -b – c;
  b = c*2*c;
}
```

- **zusätzliche Operatoren**
  - == zum Vergleichen
  - Satz an mathematischen Funktionen wie pow() und sqrt().

Element- und Nicht-Elementoperatoren

- **Stroustrup: Designempfehlung: Minimierung der Anzahl der Funktionen, die direkt die Repräsentation eines Objekts manipulieren**
  - Nur Operatoren, die schon von ihrem Prinzip her ihr erstes Argument modifizieren (z.B. +=), werden innerhalb der Klasse definiert.
  - Operatoren, die nur auf der Basis der Werte ihrer Argumente einen neuen Wert erzeugen (z.B. +), werden außerhalb der Klasse definiert und benutzen die wesentlichen Operatoren in ihrer Implementierung:

```
class rational {
  int  zaehler, nenner;
public:
  rational & operator+=(rational a); // Benötigt den Zugriff auf Interna
  // ...
};


rational operator+(rational a, rational b)
{
  rational r = a;
  return r += b; // Zugriff über +=
}
```

Beispiel

```
class rational {
   int    zaehler, nenner;
public:
   rational & operator+=(rational a);    // Benötigt den Zugriff auf Interna
   // ...
};

rational operator+(rational a, rational b)
{
   rational r = a;
   return r += b; // Zugriff über +=
}
```

- **Mit diesen Deklarationen kann man folgendes schreiben:**

```
void f(rational x, rational y, rational z)
{
  rational r1 = x + y + z;      // r1 =operator+(operator+(x,y),z)
  rational r2 = x;              // r2 = x
  r2 +=     y;                  // r2.operator+=(y)
  r2 +=     z;                  // r2.operator+=(z)
}
```

Gemischte Arithmetik
Beispiel: rational d = 2+b;

```
class rational {
   int    zaehler, nenner;
public:
   rational & operator+=(rational a)    { /* ... */ }
   rational & operator+=(int a)         { /* ... */ }
};

rational operator+(rational a, rational b)
{
   rational r = a;
   return   r+= b;  // ruft rational::operator+=(rational) auf
}

rational operator+(rational a, int b)
{
   rational r = a;
   return   r+= b;  // ruft rational::operator+=(int) auf
}

rational operator+(int a, rational b)
{
   rational r = b;
   return   r+= a;  // ruft rational::operator+=(int) auf
}
```

Initialisierung und Zuweisung mit einem Skalar

- **~~> Konvertierung eines Skalar (z.B. int) in einen rational**

```
rational   b = 3;   // entspricht: zaehler=3, nenner=1
```

- **Ein Konstruktor mit einem einzelnen Argument spezifiziert eine Konvertierung von seinem Argument-Typ auf den Typ des Konstruktors.**

```
class rational
{
  int  zaehler, nenner;
public:
  rational(int r) : zaehler(r), nenner(1) {}
  // ...
}
```

- **Beispiel:**
  - rational b = 3;

  bedeutet
  - rational b = rational(3);

Konstruktoren und Konvertierungen (1) §11.3.5

- **Problemstellung: drei Versionen von jedem der vier arithm. Operatoren**
  - `rational operator+(rational, int);`
  - `rational operator+(int, rational);`
  - `rational operator+(rational, rational);`
- **Dies wird schnell aufwendig und damit fehleranfällig, wenn weitere (Standard-)Typen hinzukommen**
- **Alternative:**
  - Bereitstellen eines Konstruktors, der einen "int" in einen "rational" umwandelt
  - "Sich auf die Konvertierung verlassen"
- **Konsequenz: es wird nur eine Version eines Operators benötigt**

```
bool operator==(rational, rational);

void f(complex x, complex y)
{
     x == y;         // bedeutet: operator==(x,y)
     x == 3;         // bedeutet: operator==(x, rational(3))
     3 == x;         // bedeutet: operator==(rational(3), x)
}
```


Konstruktoren und Konvertierungen (2) Element- / Nicht-Elementfunktionen

```
class rational {
public:
   rational operator+(rational);   // '+'  hier als Elementfunktion!
};
```

- **kann automatisch konvertieren: a+b, b+2, nicht jedoch 2+b!**
  - 2 ist kein Klassenobjekt!
- **Lösung über nicht-Elementfunktion, Problem dabei jedoch**
  - Zugriff auf Interna (private-Elemente) nicht möglich
  - Problemumgehung: z.B. über den += Operator
  - oder: Deklaration dieser Funktion als Freund der Klasse (friend), dem damit der Zugriff auf Interna erlaubt wird:

```
class rational {
public:
   friend rational operator+(rational, rational);
};
```

Konstruktoren und Konvertierungen (3)
Einbindung des 'Freunds-+' im Zusammenhang

```
class rational
{
   int         zaehler;
   unsigned    nenner;
   void        ggt();
public:
   // …
   friend rational operator+(rational, rational);
}; // Nicht-Elementfunktion als 'Freund' deklarieren

rational operator+(rational a, rational b)
{  // Operator '+' hier als Nicht-Elementfunktion implementiert
   rational    r;

   r.zaehler = a.zaehler * b.nenner  +  b.zaehler * a.nenner;
   r.nenner  = a.nenner * b.nenner;

   r.ggt();
   return r;
}
```


Konstruktoren und Konvertierungen (4)
Kommentare

- **Aspekte bei der Nutzung der automatischen Konvertierung**
  - Konvertierung kann einen Mehraufwand erzeugen
    (Es wird ein temporäres Objekt vom jeweiligen Typ erzeugt und später wieder zerstört)
  - Bei den Elementfunktionen kann evtl. für bestimmte Typen ein einfacherer Algorithmus benutzt werden
- **Mittelweg**
  - Nutzen der automatischen Konvertierung soweit möglich
  - Implementieren von spezifischen Operatorvarianten soweit notwendig
- **Bei mehreren Varianten einer Funktion oder eines Operators**
  - Compiler muß "die Richtige" anhand ihrer Argumenttypen und der verfügbaren Standard- oder benutzerdefinierten Konvertierungen herausfinden
  - Falls keine "beste Übereinstimmung" existiert, ist der Ausdruck nicht eindeutig und damit ein Fehler!

Konstruktoren und Konvertierungen (5)
Implizite benutzerdefinierte Konvertierungen auf linker Seite

- Es werden keine impliziten benutzerdefinierten Konvertierungen auf die linke Seite eines . oder -> angewendet

```
void g(rational r)
{
    3+r;                // OK: rational(3)+z
    3.operator+=(z);    // Fehler: 3 ist kein Klassenobjekt
    3+=z;               // Fehler: 3 ist kein Klassenobjekt
}
```

Weitere Konstruktoren

- benutzerdefinierte Konvertierung
- Konstruktor für zwei int
- Defaultkonstruktor, der ein rational mit (0,1) initialisiert

```
class rational
{
   int         zaehler, nenner;
public:
   rational() : zaehler(0), nenner(1) {}
   rational(int r) : zaehler(r), nenner(1) {}
   rational(int z, int n) : zaehler(z), nenner(n) {}
   // ...
};
```

- Durch Default-Argumente können wir abgekürzt schreiben:

```
class rational
{
   int         zaehler, nenner;
public:
   rational(int z=0, int n=1) : zaehler(z), nenner(n) {}
   // ...
};
```

Entwurfsvorgang -> §23.4.3

- Finden Sie die Konzepte/Klassen und deren elementarste Beziehungen
- Verfeinern Sie die Klassen durch Spezifikation ihrer Menge von Operatoren
- Verfeinern Sie die Klassen durch Spezifikation ihrer Abhängigkeiten
- Spezifizieren Sie die Schnittstellen

Ausnahmen
Ausnahmebehandlung

Problemsituation Ausnahmefall

```
void CFifo::enqueue(const int &NewItem)
{
   if (ElemCount < fifosize)
   {
     Data[NxtIn] = NewItem;
     NxtIn       = Next(NxtIn);
     ElemCount ++;
   }
   else
   {                  // Fehler: Fifo voll
   }
}
```

Was tun?

- Programm beenden
- einen Wert, der Fehler bedeutet, zurückliefern
- einen gültigen Wert zurückliefern und das Programm in einem ungültigen Zustand hinterlassen
- eine Funktion aufrufen, die für den Fehlerfall bereitgestellt wurde

- **Aspekte bei Ausnahmen**
  - Aufrufende Routine kann den Fehler noch nicht vorhersehen (z.B. fopen)
  - Aufgerufene Routine kann den Fehler selbst nicht behandeln (s.o.)
  - "Begleitende" Fehlerbehandlung macht den Code unhandlich, unelegant, schwer verständlich; Man möchte i.w. eigentlich den Hauptfall betrachten
  - Eine "passende" Stelle im Programm soll sich um Ausnahmen kümmern

Ausnahmebehandlungskonzept in C++: "Ausnahmen fangen"

- **Idee: Aufgerufene Routine erzeugt im Fehlerfall eine Ausnahme**
  - throw "Ausnahme"
- **"Irgendeine", in der Aufrufhierarchie übergeordnete Funktion zeigt daß sie ggf. auf Ausnahmen reagieren möchte,**
  - try { /* ... */ }
- **fängt den Fehler**
  - catch "Ausnahme"
- **... und führt an dieser Stelle die Ausnahmebehandlung durch**

- **Vorteile:**
  - "Saubere" Programmierung des Gut/Schlecht-Falls in der aufgerufenen Routine
  - Keine Überlastung der Rückgabewerte durch die Kodierung eines Schlecht-Zustands
  - Fehlerbehandlung muß nicht in der direkt übergeordneten Funktion durchgeführt werden, sondern kann an der passendsten Stelle geschehen
- **Falls kein "Fänger" existiert: Programmabbruch!**

Ausnahmebehandlung
Beispiel: throw

```
void CFifo::enqueue(const int &NewItem)
{
   if (ElemCount == fifosize)
     throw Overflow();                  // Fehler: Fifo voll
   Data[NxtIn] = NewItem;
   NxtIn       = Next(NxtIn);
   ElemCount ++;
}
```

- **throw**
  - bricht die Abarbeitung der Routine an dieser Stelle ab
  - der passende Ausnahme-Handler wird gesucht
  - evtl. "dazwischenliegende" Routinen werden übersprungen
  - die Ausführung wird direkt im Handler fortgesetzt

Ausnahmebehandlung
Beispiel: try - catch

```
main()
{  ...
   try {
     switch (Cmd){
     case 'i': cin >> Name;   // i: Eintragen eines neuen Elements
               fifo.enqueue(Name);
               break;
     case 'o': cout << "Eintrag: " << fifo.dequeue() << endl ;
               break;         // O: Rückgabe des nächsten Elements
     }
   } // Ende try

   catch (CFifo::Overflow) {  // Fifo-Overflow Behandlung
     cerr << "Hey, kein Platz mehr verfuegbar!\n";
   }
   catch (CFifo::Underflow) { // Fifo-Underflow Behandlung
     cerr << "Woher soll ich bitte die Daten nehmen???\n";
   }
   ...
}
```

Ausnahmebehandlung
Woher kommen die Ausnahme-Typen?

```
class CFifo {
   int     *Data;
   int     fifosize, ElemCount, NxtIn, NxtOut;
public:
   class   Overflow {};    // definiert eine (öffentliche) Struktur als Klassenelement
   class   Underflow {};   // dto.
           CFifo(int s = 15);
           ~CFifo();
   void    enqueue(const int &NewItem);
   int     dequeue();
};
```

- **Ausnahme-Typen werden mit in der Klasse als Elemente definiert**
- **Ausnahme-Typen können sich auch Fehlerzustände merken:**

```
   class   Range_error {
      int  i;
      Range_error(int ii) { i = ii; }    // Konstruktor
   };
```

- **Verwendung: throw Range_error(i);**

Hinweise zur Ausnahmebehandlung

- **Achtung: throw ist kein einfacher Rücksprung!**
  - Alle zwischen throw und Ausnahme-Handler liegenden Daten auf dem Aufruf-Stack müssen entfernt werden
  - Returnwerte werden nicht erzeugt
  - Strukturierte Programmierung???
- **Besondere Vorsicht bei Funktionen mit variabler Argumentanzahl**
- **Später ...**
  - Möglichkeit zum Aufbauen von Hierarchien von Ausnahmen, die einzeln oder über die Ausnahmegruppe abgefangen werden können
  - Weiterwerfen von Fehlerbehandlungen, wenn der Fehler an einer Stelle nur unvollkommen behandelt werden konnte