Einführung C++

- heute nur kurze Einführung in C++
- gutes C++-Tutorial unter:

`http://www.cpp-tutor.de`
`http://www.cplusplus.com/doc/tutorial/`

Einfache Datentypen

- Einfache Datentypen
  - Ganzzahl: `int, unsigned, short, char`: -15, 7, 'a'
  - Fließkomma: `double, float`: -3.5, 0.f, 1e-4
  - Wahrheitswerte: `bool/boolean`: true, false
  - Achtung! Variablen werden nicht automatisch initialisiert!
- Operatoren
  - arithmetische Operatoren `+, -, *, /, %, ++, --`
  - logische Operatoren `&&, ||, !, <, ==, >=`
  - Zuweisungsoperatoren `=, +=, *=, (&=, ...)`
  - (Binäre Operatoren `&, |, ^, ~, <<, >>`)
- Datentyp-Konvertierung
  - automatische Konvertierung
    `int` $\Leftrightarrow$ `unsigned`, `int` $\Rightarrow$ `double`
  - explizite Konvertierung `(int)0.5`
- beliebte Fehler
  - `double x = 2/3;` besser: `x = 2./3.;`
  - `unsigned u = 2-3;` oder `while (u >= 0) ...`

Einfache Datentypen

- Einfache Datentypen
  - Ganzzahl: `int, unsigned, short, char`: -15, 7, 'a'
  - Fließkomma: `double, float`: -3.5, 0.f, 1e-4
  - Wahrheitswerte: `bool/boolean`: true, false
  - Achtung! Variablen werden nicht automatisch initialisiert!
- Operatoren
  - arithmetische Operatoren `+, -, *, /, %, ++, --`
  - logische Operatoren `&&, ||, !, <, ==, >=`
  - Zuweisungsoperatoren `=, +=, *=, (&=, ...)`
  - (Binäre Operatoren `&, |, ^, ~, <<, >>`)
- Datentyp-Konvertierung
  - automatische Konvertierung
    `int` $\Leftrightarrow$ `unsigned, int` $\Rightarrow$ `double`
  - explizite Konvertierung `(int)0.5`
- beliebte Fehler
  - `double x = 2/3;` besser: `x = 2./3.;`
  - `unsigned u = 2-3;` oder `while (u >= 0) ...`

Einfache Datentypen

- Einfache Datentypen
  - Ganzzahl: `int, unsigned, short, char`: -15, 7, 'a'
  - Fließkomma: `double, float`: -3.5, 0.f, 1e-4
  - Wahrheitswerte: `bool/boolean`: true, false
  - Achtung! Variablen werden nicht automatisch initialisiert!
- Operatoren
  - arithmetische Operatoren `+, -, *, /, %, ++, --`
  - logische Operatoren `&&, ||, !, <, ==, >=`
  - Zuweisungsoperatoren `=, +=, *=, (&=, ...)`
  - (Binäre Operatoren `&, |, ^, ~, <<, >>`)
- Datentyp-Konvertierung
  - automatische Konvertierung
    `int` $\Leftrightarrow$ `unsigned`, `int` $\Rightarrow$ `double`
  - explizite Konvertierung `(int)0.5`
- beliebte Fehler
  - `double x = 2/3;` besser: `x = 2./3.;`
  - `unsigned u = 2-3;` oder `while (u >= 0) ...`

Einfache Datentypen

- Einfache Datentypen
  - Ganzzahl: `int, unsigned, short, char`: -15, 7, 'a'
  - Fließkomma: `double, float`: -3.5, 0.f, 1e-4
  - Wahrheitswerte: `bool/boolean`: true, false
  - Achtung! Variablen werden nicht automatisch initialisiert!
- Operatoren
  - arithmetische Operatoren `+, -, *, /, %, ++, --`
  - logische Operatoren `&&, ||, !, <, ==, >=`
  - Zuweisungsoperatoren `=, +=, *=, (&=, ...)`
  - (Binäre Operatoren `&, |, ^, ~, <<, >>`)
- Datentyp-Konvertierung
  - automatische Konvertierung

    `int` $\Leftrightarrow$ `unsigned, int` $\Rightarrow$ `double`
  - explizite Konvertierung `(int)0.5`
- beliebte Fehler
  - `double x = 2/3;` besser: `x = 2./3.;`
  - `unsigned u = 2-3;` oder `while (u >= 0) ...`

Einfache Datentypen

- Einfache Datentypen
  - Ganzzahl: `int, unsigned, short, char`: -15, 7, 'a'
  - Fließkomma: `double, float`: -3.5, 0.f, 1e-4
  - Wahrheitswerte: `bool/boolean`: true, false
  - Achtung! Variablen werden nicht automatisch initialisiert!
- Operatoren
  - arithmetische Operatoren `+, -, *, /, %, ++, --`
  - logische Operatoren `&&, ||, !, <, ==, >=`
  - Zuweisungsoperatoren `=, +=, *=, (&=, ...)`
  - (Binäre Operatoren `&, |, ^, ~, <<, >>`)
- Datentyp-Konvertierung
  - automatische Konvertierung

    `int` $\Leftrightarrow$ `unsigned, int` $\Rightarrow$ `double`
  - explizite Konvertierung `(int)0.5`
- beliebte Fehler
  - `double x = 2/3;` besser: `x = 2./3.;`
  - `unsigned u = 2-3;` oder `while (u >= 0) ...`

Komplexe Datentypen

- Vektoren
  - Vektoren von Variablen gleichen Typs
    ```
    std::vector<double> v(10);
    ```
  - Mehrdimensionale Felder
    ```
    std::vector<std::vector<double> > m(5);
    ```
  - Index startet **immer** mit 0
  - Zugriff über `v[0]...v[9]`, `m[0][0]...m[4][2]`
  - Schneller Zugriff auf die Elemente, nachträgliche Änderung der Größe per `resize`
- Strukturen
  - Zusammenfassung verschiedener Typen zu einer Einheit
    ```
    struct{int a; double b; char c;} s;
    typedef struct{int a; double b; char c;} T;
    ```
  - Zugriff über .-Operator
    ```
    s.a = 7; s.b = M_PI; s.c = 'o';
    ```
- andere Operatoren sind für Strukturen i.A. nicht vorhanden

Komplexe Datentypen

- Vektoren
  - Vektoren von Variablen gleichen Typs
    `std::vector<double> v(10);`
  - Mehrdimensionale Felder
    `std::vector<std::vector<double> > m(5);`
  - Index startet **immer** mit 0
  - Zugriff über `v[0]...v[9], m[0][0]...m[4][2]`
  - Schneller Zugriff auf die Elemente, nachträgliche Änderung der Größe per `resize`
- Strukturen
  - Zusammenfassung verschiedener Typen zu einer Einheit
    `struct{int a; double b; char c;} s;`
    `typedef struct{int a; double b; char c;} T;`
  - Zugriff über .-Operator
    `s.a = 7; s.b = M_PI; s.c = 'o';`
- andere Operatoren sind für Strukturen i.A. nicht vorhanden

Komplexe Datentypen

- Vektoren
  - Vektoren von Variablen gleichen Typs
    `std::vector<double> v(10);`
  - Mehrdimensionale Felder
    `std::vector<std::vector<double> > m(5);`
  - Index startet **immer** mit 0
  - Zugriff über `v[0]...v[9], m[0][0]...m[4][2]`
  - Schneller Zugriff auf die Elemente, nachträgliche Änderung der Größe per `resize`
- Strukturen
  - Zusammenfassung verschiedener Typen zu einer Einheit
    `struct{int a; double b; char c;} s;`
    `typedef struct{int a; double b; char c;} T;`
  - Zugriff über .-Operator
    `s.a = 7; s.b = M_PI; s.c = 'o';`
- andere Operatoren sind für Strukturen i.A. nicht vorhanden

Zeiger und Referenzen

- Zeiger, Referenzen
  - Zeiger auf Speicherbereich `double*, char*`
  - Referenzen auf bereits existierende Objekte `double&`
- Operatoren
  - Neues Objekt bzw. Array auf dem Heap erzeugen `new`

    ```
    double* p = new double;
    double* v = new double[20];
    ```
  - Objekt auf dem Heap löschen `delete`

    ```
    delete p; delete[] v;
    ```
  - Adresse eines Objektes `&`

    ```
    double x = 10.; double* p = &x;
    ```
  - Referenz auf ein Objekt `&`

    ```
    double x = 10.; double& r = x;
    ```
  - Dereferenzierung `*, ->`

    ```
    *p = x + 5.;
    T* t = new T; t->a = 0; (*t).c = '\n';
    ```

Zeiger und Referenzen

- Zeiger, Referenzen
  - Zeiger auf Speicherbereich `double*, char*`
  - Referenzen auf bereits existierende Objekte `double&`
- Operatoren
  - Neues Objekt bzw. Array auf dem Heap erzeugen `new`

    ```
    double* p = new double;
    double* v = new double[20];
    ```
  - Objekt auf dem Heap löschen `delete`

    ```
    delete p; delete[] v;
    ```
  - Adresse eines Objektes `&`

    ```
    double x = 10.; double* p = &x;
    ```
  - Referenz auf ein Objekt `&`

    ```
    double x = 10.; double& r = x;
    ```
  - Dereferenzierung `*, ->`

    ```
    *p = x + 5.;
    T* t = new T; t->a = 0; (*t).c = '\n';
    ```

Zeiger und Referenzen

- Zeiger, Referenzen
  - Zeiger auf Speicherbereich `double*, char*`
  - Referenzen auf bereits existierende Objekte `double&`
- Operatoren
  - Neues Objekt bzw. Array auf dem Heap erzeugen `new`

    ```
    double* p = new double;
    double* v = new double[20];
    ```
  - Objekt auf dem Heap löschen `delete`

    ```
    delete p; delete[] v;
    ```
  - Adresse eines Objektes `&`

    ```
    double x = 10.; double* p = &x;
    ```
  - Referenz auf ein Objekt `&`

    ```
    double x = 10.; double& r = x;
    ```
  - Dereferenzierung `*, ->`

    ```
    *p = x + 5.;
    T* t = new T; t->a = 0; (*t).c = '\n';
    ```

Zeiger und Referenzen

- Zeiger, Referenzen
  - Zeiger auf Speicherbereich `double*, char*`
  - Referenzen auf bereits existierende Objekte `double&`
- Operatoren
  - Neues Objekt bzw. Array auf dem Heap erzeugen `new`

    ```
    double* p = new double;
    double* v = new double[20];
    ```
  - Objekt auf dem Heap löschen `delete`

    ```
    delete p; delete[] v;
    ```
  - Adresse eines Objektes `&`

    ```
    double x = 10.; double* p = &x;
    ```
  - Referenz auf ein Objekt `&`

    ```
    double x = 10.; double& r = x;
    ```
  - Dereferenzierung `*, ->`

    ```
    *p = x + 5.;
    T* t = new T; t->a = 0; (*t).c = '\n';
    ```

Zeiger und Referenzen

- Zeiger, Referenzen
  - Zeiger auf Speicherbereich `double*, char*`
  - Referenzen auf bereits existierende Objekte `double&`
- Operatoren
  - Neues Objekt bzw. Array auf dem Heap erzeugen `new`

    ```
    double* p = new double;
    double* v = new double[20];
    ```
  - Objekt auf dem Heap löschen `delete`

    ```
    delete p; delete[] v;
    ```
  - Adresse eines Objektes `&`

    ```
    double x = 10.; double* p = &x;
    ```
  - Referenz auf ein Objekt `&`

    ```
    double x = 10.; double& r = x;
    ```
  - Dereferenzierung `*, ->`

    ```
    *p = x + 5.;
    T* t = new T; t->a = 0; (*t).c = '\n';
    ```

Zeiger und Referenzen

- Zeiger, Referenzen
  - Zeiger auf Speicherbereich `double*, char*`
  - Referenzen auf bereits existierende Objekte `double&`
- Operatoren
  - Neues Objekt bzw. Array auf dem Heap erzeugen `new`

    ```
    double* p = new double;
    double* v = new double[20];
    ```
  - Objekt auf dem Heap löschen `delete`

    ```
    delete p; delete[] v;
    ```
  - Adresse eines Objektes `&`

    ```
    double x = 10.; double* p = &x;
    ```
  - Referenz auf ein Objekt `&`

    ```
    double x = 10.; double& r = x;
    ```
  - Dereferenzierung `*, ->`

    ```
    *p = x + 5.;
    T* t = new T; t->a = 0; (*t).c = '\n';
    ```

Blöcke

- Bedingungen:
  - einzelne Verzweigung `if`

```
if (cond) {bodyTRUE}
if (cond) {bodyTRUE} else {bodyFALSE}
```

  - mehrfache Verzweigung `switch`

```
switch (i){
  case 0:
    body0
    break; ← nicht vergessen!
  ...
  default:
    bodyDEFAULT
    break;
}
```

  - ?-Operator:

```
cond ? bodyTRUE : bodyFALSE
int b = a > 0 ? a : -a;
```

Blöcke

- Bedingungen:
  - einzelne Verzweigung `if`

```
if (cond) {bodyTRUE}
if (cond) {bodyTRUE} else {bodyFALSE}
```

  - mehrfache Verzweigung `switch`

```
switch (i){
  case 0:
    body0
    break; ← nicht vergessen!
  ...
  default:
    bodyDEFAULT
    break;
}
```

  - ?-Operator:

```
cond ? bodyTRUE : bodyFALSE
int b = a > 0 ? a : -a;
```

Blöcke

- Bedingungen:
  - einzelne Verzweigung `if`

    ```
    if (cond) {bodyTRUE}
    if (cond) {bodyTRUE} else {bodyFALSE}
    ```
  - mehrfache Verzweigung `switch`

    ```
    switch (i){
      case 0:
        body0
        break; ← nicht vergessen!
      ...
      default:
        bodyDEFAULT
        break;
    }
    ```
  - ?-Operator:

    ```
    cond ? bodyTRUE : bodyFALSE
    int b = a > 0 ? a : -a;
    ```

Blöcke

- Schleifen
  - while-Schleife:
    `while (cond) {body}`
  - do-while-Schleife:
    `do {body} while (cond);`
  - for-Schleife:
    `for (init ; cond ; statement) {body}`
- Beispiele:

```
while (a < b){
  a = a + 1;
}

for (int i = 0; i < 10; ++i){
  sum += v[i];
}
```

Blöcke

- Schleifen
  - while-Schleife:
    `while (cond) {body}`
  - do-while-Schleife:
    `do {body} while (cond);`
  - for-Schleife:
    `for (init ; cond ; statement) {body}`
- Beispiele:

```
while (a < b){
  a = a + 1;
}

for (int i = 0; i < 10; ++i){
  sum += v[i];
}
```

Blöcke

- Funktionen:
- Häufig benötigte Aufgaben als eigene Funktion implementieren, die an entsprechender Stelle im Code aufgerufen wird
- Vorteil: einfachere Fehlersuche, Fehler muß nur an einer Stelle im Code gesucht werden
- bessere Lesbarkeit des Codes, er wird deutlich kürzer

Blöcke

- Beispiel: Berechnung des Skalarproduktes zweier Vektoren

```
double scalarProd(std::vector<double> x,
std::vector<double> y){
  double t = 0;
  for (int i=0;i<x.size();++i){
  t+=x[i]*y[i];
  }
  return t;
}
```

Blöcke

- Struktur:
  `ret_type func_name(param_list){body}`
  - `ret_type`: Typ des zurückgegebenen Wertes
    - beliebige Typen erlaubt (Ausnahme: C-Arrays)
    - keine Rückgabewert: `void`
  - `func_name` Name der Funktion
  - `param_list` Liste der Parameter
    - Komma-separierte Liste `int a, double b`
    - Wertparameter: Wert wird *kopiert* `double x`
    - Referenzparameter: `double& x`
    - Standard-Werte: `int a = 0, double b = 0.`
  - `body` Inhalt der Funktion

main-Funktion

- wichtigste Funktion `main`:
  `int main (int argc, char* argv[]){ }`
  - Hauptfunktion, wird bei jedem Programmstart aufgerufen
  - `argc` und `argv` sind Kommandozeilen-Parameter
    - `argc` gibt die Anzahl von Parametern an
    - `argv[i]` sind die Paramter (als `char*`)
    - `argv[0]` ist immer der Programmname selbst
    - Achtung! Vor `argv[i]` immer `argc > i` prüfen!
  - Funktionen zur Umwandlung der Parameter:
    - `int atoi(char*);`
    - `double atof(char*);`
  - Rückgabewert
    - 0 bei Erfolg
    - meist 1 oder -1 im Fehlerfall

Klassen

- Klassen bilden abgeschlossene Einheiten von
  - Variablen (Attribute, Member-Variablen)
  - Funktionen (Methoden, Schnittstellen)
- Konzepte von Klassen
  - Abgeschlossenheit, Kapselung
  - Zugriffsrechte `public, protected, private`
  - typisch: Funktionen `public` oder `protected`, Variablen `protected` oder `private`
  - Vererbung, Polymorphie
  - Klasse (statisch) und Objekt (dynamisch)
- Aufteilung in Deklaration und Definition
  - Deklaration im Header (.h oder .hpp)
  - Definition im Programmcode (.cc oder .cpp)

Klassen

- Klassen bilden abgeschlossene Einheiten von
  - Variablen (Attribute, Member-Variablen)
  - Funktionen (Methoden, Schnittstellen)
- Konzepte von Klassen
  - Abgeschlossenheit, Kapselung
  - Zugriffsrechte `public, protected, private`
  - typisch: Funktionen `public` oder `protected`, Variablen `protected` oder `private`
  - Vererbung, Polymorphie
  - Klasse (statisch) und Objekt (dynamisch)
- Aufteilung in Deklaration und Definition
  - Deklaration im Header (.h oder .hpp)
  - Definition im Programmcode (.cc oder .cpp)

Klassen

- Klassen bilden abgeschlossene Einheiten von
  - Variablen (Attribute, Member-Variablen)
  - Funktionen (Methoden, Schnittstellen)
- Konzepte von Klassen
  - Abgeschlossenheit, Kapselung
  - Zugriffsrechte `public, protected, private`
  - typisch: Funktionen `public` oder `protected`, Variablen `protected` oder `private`
  - Vererbung, Polymorphie
  - Klasse (statisch) und Objekt (dynamisch)
- Aufteilung in Deklaration und Definition
  - Deklaration im Header (.h oder .hpp)
  - Definition im Programmcode (.cc oder .cpp)

Klassen

- Konstruktoren
  - werden beim Erstellen eines Objektes aufgerufen
  - initialisieren alle Attribute des Objektes
  - ⇒ Spezielle Konstruktor-Syntax
  - legen oft dynamischen Speicher an
  - haben den selben Namen wie die Klasse
  - haben keinen Rückgabetyp (auch nicht `void`)
  - können verschiedene Parameter besitzen
  - häufig: Standard-Konstruktoren, Copy-Konstruktoren
- Destruktoren
  - werden beim Zerstören eines Objektes aufgerufen
  - sollten dynamischen Speicher wieder freigeben
  - haben den Namen `~Klassenname()`
  - haben keine Parameter und keinen Rückgabetyp
  - können auch weggelassen werden (Default-Destruktor)

Klassen

- Konstruktoren
  - werden beim Erstellen eines Objektes aufgerufen
  - initialisieren alle Attribute des Objektes
  - $\Rightarrow$ Spezielle Konstruktor-Syntax
  - legen oft dynamischen Speicher an
  - haben den selben Namen wie die Klasse
  - haben keinen Rückgabetyp (auch nicht `void`)
  - können verschiedene Parameter besitzen
  - häufig: Standard-Konstruktoren, Copy-Konstruktoren
- Destruktoren
  - werden beim Zerstören eines Objektes aufgerufen
  - sollten dynamischen Speicher wieder freigeben
  - haben den Namen `~Klassenname()`
  - haben keine Parameter und keinen Rückgabetyp
  - können auch weggelassen werden (Default-Destruktor)

Klassen

Inhalt von Matrix.h: Deklaration der Klasse `Matrix` (Auszug)

```
class Matrix{
  public:
    Matrix(unsigned xSizeA=0, unsigned ySizeA=0);
    ~Matrix();
    unsigned xSize() const;
    unsigned ySize() const;
    const double item(unsigned xA,
    unsigned yA) const;
    void item(unsigned xA, unsigned yA,
    double valueA);
    void invert(double epsilonA=1e-12);
  protected:
    unsigned rowsE, colsE;
    double* dataE;
};  ← Semikolon nicht vergessen!
```

Klassen

Inhalt von Matrix.h: Definition der Elemente von `Matrix`

```
Matrix::Matrix(unsigned xSizeA, unsigned ySizeA)
: xSizeE(xSizeA), ySizeE(ySizeA){
   dataE = new double[xSizeE * ySizeE];
}

Matrix::~Matrix(){
   delete[] dataE;
}

inline const double Matrix::item(
   unsigned xA, unsigned yA) const{
   assert(xA < xSizeE && yA < ySizeE);
   return dataE[xA * ySizeE + yA];
}
```

⋮

Standard Template Library im Namespace std

- Zeichenketten `string`
- Containerklassen (Templates)
  - Arrays mit variabler Länge `vector<T>`
  - Mengen mit eindeutigen Werten `set<Key>`
  - assoziative Arrays `map<Key, T>`
  - `list<T>`, Iteratoren, Funktionen (`sort,...`), ...
- Ein/Ausgabe
  - Dateiströme zum Lesen / Schreiben von ASCII-Dateien `ifstream, ofstream`
    Basisklassen `istream, ostream`
  - formatierte Eingabe per `istream::operator >>`
  - formatierte Ausgabe per `ostream::operator <<`
  - Standardein- und -ausgabe `cin, cout, cerr`
  - Spezielle Zeichen `'\n', '\t', '\r', flush, endl`
  - Beispiel:

```
cout << "2 * 2 =\t" << 2*2 << '.' << endl;
```

Standard Template Library im Namespace std

- Zeichenketten `string`
- Containerklassen (Templates)
  - Arrays mit variabler Länge `vector<T>`
  - Mengen mit eindeutigen Werten `set<Key>`
  - assoziative Arrays `map<Key, T>`
  - `list<T>`, Iteratoren, Funktionen (`sort,...`), ...
- Ein/Ausgabe
  - Dateiströme zum Lesen / Schreiben von ASCII-Dateien `ifstream, ofstream`
    Basisklassen `istream, ostream`
  - formatierte Eingabe per `istream::operator >>`
  - formatierte Ausgabe per `ostream::operator <<`
  - Standardein- und -ausgabe `cin, cout, cerr`
  - Spezielle Zeichen `'\n', '\t', '\r', flush, endl`
  - Beispiel:

```
cout << "2 * 2 =\t" << 2*2 << '.' << endl;
```

Standard Template Library im Namespace std

- Zeichenketten `string`
- Containerklassen (Templates)
  - Arrays mit variabler Länge `vector<T>`
  - Mengen mit eindeutigen Werten `set<Key>`
  - assoziative Arrays `map<Key, T>`
  - `list<T>`, Iteratoren, Funktionen (`sort,...`), ...
- Ein/Ausgabe
  - Dateiströme zum Lesen / Schreiben von ASCII-Dateien `ifstream, ofstream`
    Basisklassen `istream, ostream`
  - formatierte Eingabe per `istream::operator >>`
  - formatierte Ausgabe per `ostream::operator <<`
  - Standardein- und -ausgabe `cin, cout, cerr`
  - Spezielle Zeichen `'\n', '\t', '\r', flush, endl`
  - Beispiel:

```
cout << "2 * 2 =\t" << 2*2 << '.' << endl;
```