Objektorientiertes Programmieren in C++

2017

Karsten Balzer

880-4659
balzer@rz.uni-kiel.de

11.-13.10.2017

Gliederung

- **Einführung**
- **Grundlagen von C++**
  - Variablendeklaration und Datentypen
  - Kontrollstrukturen und Funktionen → Z.B1
  - Ein- und Ausgaben, Arbeiten mit Dateien, Zeichenketten (Strings)
  - Arrays, dynamische Speicherbelegung (Zeiger) und Strukturen → Z.B2
- **Objektorientierte Programmierung**
  - Grundbegriffe der objektorientierten Programmierung
  - Klassen I.: Definition und Aufbau (Objekte, Members usw.) → Z.C1
  - Klassen II.: Überladung von Funktionen und Operatoren
  - Klassen III.: Freundschaft und Vererbung → Z.C2
- **Weitere Konzepte**
  - Namespaces und Templates
  - [Parallelisierung mit OpenMP/MPI, Beispiele]

- **Das Standardwerk**

– Bjarne Stroustrup, *The C++ Programming Language* (Addison-Wesley, 2013 [4th edition])

- **Ansonsten z.B.**

– Ulrich Breymann, *C++: Einführung und professionelle Programmierung* (Carl Hanser Verlag, 2003)

– E. Balagurusamy, *Object Oriented Programming with C++* (MC Graw Hill, 2014 [6th edition])

– RRZN-Handbuch, *C++ für Programmierer* (2012), über die UB Kiel erhältlich

– Online-Referenz und -Tutorial:
http://www.cplusplus.com/doc/tutorial/
www.cplusplus.com/files/tutorial.pdf

- **ANSI-C**

– http://openbook.rheinwerk-verlag.de/c_von_a_bis_z/

Objektorientiertes Programmieren in C++

Teil A: Einführung

Karsten Balzer

880-4659
balzer@rz.uni-kiel.de

- **C++ als Spracherweiterung von C**

– C++ ist traditionelle prozedurale Programmiersprache mit einigen zusätzlichen Funktionen
– Zusätzliche Konstrukte für das objektorientierte Programmieren
– Einige zusätzliche Konstrukte zur Verbesserung der allgemeinen Syntax
– C++ ist erweiterbare Programmiersprache: Definition von neuen Typen möglich, die sich wie die vordefinierten Typen verhalten

– Ungefähr 30 neue Schlüsselwörter im Vergleich zu ANSI-C. Zum Beispiel:

```
bool, class, delete, false, friend, inline, namespace, new,
operator, private, protected, public, template, this, true,
typename, using, virtual
```

- **Entwicklung von C++**

- C++ wird Anfang der 1980er von Bjarne Stroustrup an den AT&T Bell Labs in Murray Hill, NJ, USA als Erweiterung von C entwickelt (als Vorbild dient Simula67) [www.stroustrup.com]
- Ursprünglicher Name "C with classes"
- 1983: Umbenennung in "C++" in Anlehnung an den Inkrementoperator
- 1985 erschien die erste Version von C++
- 1987: Einführung des GNU C++-Compiler (damit einer der ältesten C++-Compiler)
- Nach der koninuiertlichen Weiterentwicklung in den 1990er wird die Programmiersprache 1998 durch die ISO (International Organization for Standardization) standardisiert (Version C++98)
- 2003: Nachbesserung der Norm von 1998 [ISO/IEC 14882:2003] (Version C++03)
- 2011: Veröffentlichung eines neuen Standards [ISO/IEC 14882:2011] (Inoffiziell Version C++11)

Objektorientiertes Programmieren in C++

Teil B: Grundlagen von C++

Karsten Balzer

880-4659
balzer@rz.uni-kiel.de

C | A | U
Christian-Albrechts-Universität zu Kiel
Rechenzentrum

- **Kommentare in C**

```
/* comment text */
```

– Beispiel:

```
/*
This is a comment in a regular C-program,
which runs over a few lines.*/
```

- **Kommentare in C++**

```
/* comment text */
// single-line comment
```

– Alles ab `//` bis zum Zeilenende ist ein Kommentar.
– Beispiele:

```
int i; // run time variable
float sum=s1+s1; // sum denotes the sum of s1 and s2
```

C A U
Christian-Albrechts-Universität zu Kiel
Rechenzentrum

- **C**

```
/* a simple C-program */

#include <stdio.h>

int main() {
printf("Hello world!\n");
return 0;
}
```

```
/* a simple C-program */

#include <stdio.h>

int main() {
int x=10;
double y=2.718281;
printf("x=%d, y=%.3e\n",x,y);
return 0;
}
```

Kommandozeilenausgaben in C++

- **C**

```
/* a simple C-program */

#include <stdio.h>

int main() {
printf("Hello world!\n");
return 0;
}
```

```
/* a simple C-program */

#include <stdio.h>

int main() {
int x=10;
double y=2.718281;
printf("x=%d, y=%.3e\n",x,y);
return 0;
}
```

- **C++**

```
/* a simple C++-program */

#include <iostream>
using namespace std;

double a=2.718281; // global

int main() {
bool b=true; // local; not false
bool B=true; // case-sensitive
char c='G';
char str[]="job";
cout << "Hello world!\n";
cout << "Hello students!" <<
endl;
// cout handles different
// data types
cout << "a=" << a << endl;
cout << b << " " << c << "\n";
cout << "str=" << str << endl;
return 0;
}
```

- **Ausgabeformatierung in C++**

```
/* a simple C++-program */

#include <iostream>
#include <iomanip>
using namespace std;

double a=2.718281; // global

int main() {
   bool b=true; // local; not false
   bool B=true; // case-sensitive
   char c='G';
   char str[]="job";
   cout << "Hello world!\n";
   cout << "Hello students!" << endl;
   // cout handles different
   // data types
   cout << "a=" << a << endl;
   cout << b << " " << c << "\n";
   cout << "str=" << str << endl;
   cout << "a=" << setprecision(10) << scientific << a << endl;
   cout << "a=" << setprecision(4) << fixed << a << endl;
   return 0;
   }
```

- **Programmübersetzung**

– **G**nu **C**ompiler **C**ollection (GCC): Wir nutzen aus dieser Compiler-Suite den Compiler für C++-Code:

```
$ g++ -o test.ex test.cpp
```

– Das Zeichen `$` steht hier und im Folgenden immer für den Kommandozeilen-Prompt.

- **Programmausführung** TUTORIAL #0a

```
$ ./test.ex
```

- **Ausgabe**

```
Hello world!
Hello students!
a=2.71828
1 G
str=job
a=2.7182810000e+00
a=2.7183
```

C | A | U
Christian-Albrechts-Universität zu Kiel
Rechenzentrum

- **C++-Beispiel** TUTORIAL #0b

```
/* a simple C++-program */

#include <iostream>
#include <iomanip>
using namespace std;

double a=2.718281; // global

int main() {
   bool b=true; // local; not false
   bool B=true; // case-sensitive
   char c='G';
   char str[]="job";
   cout << "Hello world!\n";
   cout << "Hello students!" << endl;
   // cout handles different
   // data types
   cout << "a=" << a << endl;
   cout << b << " " << c << "\n";
   cout << "str=" << str << endl;
   cout << "a=" << setprecision(10) << scientific << a << endl;
   cout << "a=" << setprecision(4) << fixed << a << endl;
   cout << "Insert a double and a string:" << endl;
   cin >> a >> str;
   cout << "a=" << a << ", str=" << str << endl;
   return 0;
   }
```

Datentypen in C++

- **Standard-Datentypen**

– C++ besitzt alle Datentypen, die in C vorhanden sind:

```
char, unsigned char, int, short, long, unsigned int, unsigned short,
unsigned long, float, double
```

- **Zusätzliche Datentypen**

– Zusätzlich gibt es zwei weitere Datentypen

– Datentyp "`bool`":
Boolsche Zahl; belegt nur ein Byte; Wert ist entweder wahr (1; "true") oder falsch (0; "false")

```
bool answer=true; // or type answer=1 which is the same
```

– Datentyp "`class`":
Erlaubt es, Daten in eine Klasse zu "verpacken"; Klassen definieren Objekte in C++.

```
class user {
   char name[];
   };
```

C | A | U
Christian-Albrechts-Universität zu Kiel
Rechenzentrum

- **Verzweigungen**

– "if-else", bedingter Ausdruck und "switch-case"

```
if (x<10) {...} else if (x>50) {...} else {...}

x=(x<10) ?  100 :  200;

switch (x) {
      case 5:  {...; break;}
      default:  {...; break;}
      }
```

- **Schleifen**

– "while-do" und "do-while"

```
while (x<10) {...}

do {...} while (x<10);
```

– Variablen können in C++ überall deklariert werden. Jedoch muss die Deklaration vor der ersten Verwendung stehen TUTORIAL #0c

```
// not possible in ANSI-C
// use with care outside of loops
for(int i=0;i<10;i++) cout << i << endl;
```

Operationen und Funktionen (Prozeduren)

- **Grundlegende Operationen wie in C**

– Zum Beispiel:

```
int a=b+c; // or -, *, /, %
int a+=b; // or -=, *=, /=, %=, etc.
int a=1; a++; // or -- (increment or decrement a)
if (a<1) {...} // or <=, >, >=, ==, !=, etc.
if ((a==1)&&(b==2)) {...} // or ||
```

- **Funktionendeklarationen**

– Deklaration wie in C möglich

```
int max(int a, int b) { // call by value
   return (a>b) ?  a :  b;
   }

void add(int& a, int b) { // call by reference (for a)
   a+=b;
   }
```

Aber es gibt einige Besonderheiten in C++!

- **Besonderheiten I**

– Angabe von Default-Werten für Funktionsparameter ist möglich (Reihenfolge wichtig und nur in Funktionsprototypen!)

```
// cylinder
double volume(double height, double radius=1.0) {
   return PI*radius*radius*height;
   }
```

– Überladen von Funktionen TUTORIAL #1

```
// cylinder
double volume(double height, double radius=1.0) {
   return PI*radius*radius*height;
   }

// hollow cylinder
double volume(double height, double r1, double r2) {
   return PI*(r1*r1-r2*r2)*height;
   }

// hollow cylinder (this produces a compiler error)
double volume(double height, double r1, double r2=0.0) {
   return PI*(r1*r1-r2*r2)*height;
   }
```

C | A | U
Christian-Albrechts-Universität zu Kiel
Rechenzentrum

- **Besonderheiten II**

– inline-Funktionen werden mit dem zusätzlichen Schlüsselwort `inline` definiert

```
// cylinder
inline double volume(double height, double r) {
   return PI*r*r*height;
   }
```

– Der Aufruf von Funktionen benötigt eine gewisse Zeit
– Compiler versucht, anstelle des Funktionsaufrufs direkt den Code der Funktion an die Aufrufstelle einzufügen. Es entfällt damit der komplette Overhead für den Aufruf
– inline-Funktionen entsprechen in etwa den Präprozessor-Makros (`#define`-Direktiven). Der Hauptunterschied ist, dass die Parameter bei inline-Funktionen typisiert sind
– Hinweis: `inline` ist nur eine Empfehlung an den Compiler (!), `inline` nur für kleine Funktionen verwenden
– Testen des inlining von Funktionen? TUTORIAL #X1

Zusammenfassung (Z.B1)

- **Änderungen gegenüber ANSI-C**

– Kommentare können auf 2 Arten gekennzeichnet werden
– Unterschiedliche Prinzipien der Ein- und Ausgabe (`<iostream>`)
– "`bool`" und "`class`" treten als zusätzliche Datentypen auf
– Unterschiedliche Platzierung von Variablendeklarationen
– An eine Funktionen kann eine variable Anzahl von Parametern übergeben werden (Polymorphie)
– Funktionen können überladen werden (dies wird wichtig bei der objektorientierten Programmierung zur Definition abstrakter Operationen)
– "Inlining" von Funktionen möglich (nur Empfehlung an den Compiler)

C | A | U
Christian-Albrechts-Universität zu Kiel
Rechenzentrum

- **Ein- und Ausgaben**

– mit den Direktiven fstream, ofstream und ifstream

```
#include <cstdlib>
#include <iostream>
#include <fstream> // or use separately <ofstream>, <ifstream>

using namespace std;

int main()
   {
   ofstream fout;
   fout.open("output.dat");
   fout << 0.123456789 << " " << "Dies ist ein Text!" << endl;
   fout.close();
   system("cp output.dat input.dat");
   string str1, str2;
   ifstream fin;
   fin.open("input.dat");
   fin >> str1;
   getline(fin,str2);
   fin.close();
   cout << str1 << endl;
   cout << str2 << endl;
   fstream f;
   f.open("ouput.dat",ios::out); f.close();
   f.open("input.dat",ios::in); f.close();
   return 0;
   }
```

C | A | U
Christian-Albrechts-Universität zu Kiel
Rechenzentrum

- **<fstream>-Flags**

– `ios::in` — für Input-Operationen öffnen
– `ios::out` — für Output-Operationen öffnen
– `ios::binary` — im binary-Modus öffnen
– `ios::ate` — Anfangsposition wird an das Ende der Datei gesetzt (ansonsten an den Anfang der Datei)
– `ios::app` — Output-Operationen werden am Ende der Datei ausgeführt (Anhängen von Daten)
– `ios::trunc` — Wenn die Datei für Output-Operationen geöffnet wird und bereits existierte, wird der bisherige Inhalt gelöscht und durch die neuen Daten ersetzt

– Flags sind mit dem Oder-Operator | kombinierbar TUTORIAL #2

```
fstream f;
f.open("example.bin", ios::in | ios::binary)
f.close();
```

C | A | U
Christian-Albrechts-Universität zu Kiel
Rechenzentrum

- **Zeichenketten**

– C++ besitzt keinen Datentyp für Strings
– Es gibt jedoch eine Klassenbibliothek <string>, mit denen man Strings definieren kann

```
#include <iostream>
#include <string>

using namespace std;

int main()
   {
   string s1, s2, s3;
   s1="Max";
   s2="Mustermann";
   s3=s1+" "+s2; // append strings
   cout << "first name(s):  " << s1 << endl;
   cout << "last name:  " << s2 << endl;
   cout << "full name:  " << s3 << endl;
   return 0;
   }
```

C | A | U
Christian-Albrechts-Universität zu Kiel
Rechenzentrum

- **Manipulation von Zeichenketten**

– Suchmöglichkeiten

```
string s="Hallo world!"
int position=s.find('w'); // position of the first appearance
if (s.find (' ')!=string::npos) {
   cout << "Contains at least one space!" << endl;
   }
```

– Substrings

```
string s="Hallo world!"
string s1=s.substr(7); // --> "world!"
string s2=s.substr(7,5); // --> "world"
```

– Löschen und Ersetzen

```
string s="This is a test!"
string s1=s.erase(5,5); // --> "This test!"
string s2=s.replace(5,2,"was"); // --> "This was a test!"
```

– Strings einfügen

```
string s="This is a test!"
string s1=s.insert(10,"crucial ");
// --> "This is a crucial test!"
```

- **Möglichkeiten komplexere Strings zu generieren**

– "String streams" ermöglichen zusätzliche Flexibilität im Umgang mit Zeichenketten (Funktionalität ähnlich wie `cout`)

```
#include <sstream>

...

ostringstream s;
s << "His age is" << 27 << ".";
cout << s.str() << endl; // --> "His age is 27."
```

– Zerlegen von Zeichenketten TUTORIAL #X2

```
#include <sstream>

...

int year, month, day;
char dash1, dash2; // Dummy variables to read delimiters
string iso_date; // date in ISO format
cout << "Enter date (yyyy-mm-dd):  " << flush;
getline(cin, iso_date);
istringstream s(iso_date); // load value into string stream
s >> year >> dash1 >> month >> dash2 >> day;
// --> year, month, day
```

C | A | U
Christian-Albrechts-Universität zu Kiel
Rechenzentrum

- **Arrays**

– Definition ähnlich wie in C

```
int array1[5]={1,2,0,7,9}; // initialization
int array1[0]=2;
int value1=array1[4];

int array2[4][7]; // multidimensional array
int value2=array2[1][1];

int array3[4*7]; // pseudo-multidimensional array
int value3=array3[1*4+1];

int procedure(int a[]); // call by reference
b=procedure(array3);

double array4[]={1.1,2.7}; // square brackets can be left empty
double value4=array4[5]; // no compilation but runtime errors!
```

- **Pointers I**

– Referenz- und Dereferenzoperator

```
a=&b; // reference (a is "address of" b)
c=*a; // dereference (c is "value pointed by" a)
```

- **Pointers II** TUTORIAL #3

– Deklaration von Variablen des Typs “Pointer”

```
int *a;
char *b;
double *c; // a,b,c are of the same size in memory
// but the memory space they point to is of different size!


// example 1
int value;
int *p;
p=&value;
*p=10;
cout << value << endl;

// example 2
int value1=7, value2=5;
int *p1, *p2;
p1=&value1;
p2=&value2;
*p1=10;
*p2=*p1;
p1=p2;
*p1=20;
cout << value1 << " " << value2 << endl;
```

C | A | U
Christian-Albrechts-Universität zu Kiel
Rechenzentrum

- **Pointers III**

– Das Konzept von Arrays ist sehr verknüpft mit dem des Pointers (Pointer Arithmetik)

```
// example 3
int numbers[5];
int *p;
p=numbers; *p=10;
p++; *p=20;
p=&numbers[2]; *p=30;
p=numbers+3; *p=40;
p=numbers; *(p+4)=50;
for(int n=0;n<5;n++) cout << numbers[n] << " ";
```

– Pointers auf Pointers

```
// example 4
char a;
char *b;
char **c;

a='z';
b=&a;
c=&b;

cout << **c << endl;
```

C | A | U
Christian-Albrechts-Universität zu Kiel
Rechenzentrum

- **ANSI-C**

– Für die dynamische Speicherbelegung stehen die Funktionen `malloc`, `calloc`, `realloc` und `free` zur Verfügung, die auch in C++ unter dem header `<cstdlib>` verwendet werden können

```
int N=50;
double *v1, *v2;

v1=(double*)malloc(N*sizeof(double));

v2=(double*)calloc(N,sizeof(double)); // initialization with zeros

// (increase) decrease memory (without initialization)
v2=(double*)realloc(v2,(2*N)*sizeof(double));

free(v1); free(v2);
```

- **C++**

– Zur Vereinfachung stellt C++ die beiden Operatoren "`new`" und "`delete`" zur Verfügung

```
int N=50;
double *array;
array=new double[N]; // allocate memory

delete [] array; // free memory
```

- **Definition und Benutzung von Strukturen**

– Strukturen sind sinnvoll, um Informationen einer bestimmten Gruppe zusammenzufassen
– Syntax zunächst wie in ANSI-C

```
struct product {
   float weight; // an element of the struct
   float price;
   string unit;
   } banana;

struct product apple; // declaration of ANSI-C type
product pear; // this is possible in C++

apple.weight=1.0; // .  is element operator
apple.price=2.49;
apple.unit="kg";

product melon={1.5,1.79,"kg"};
```

– Verwenden von Strukturen als Argumente

```
void show(struct product x) {
   cout << x.price/x.weight << " EUR/" << x.unit << endl;
   }
show(apple); // pass a struct to a function
```

- **Elementfunktionen** TUTORIAL #4

– In C++ können innerhalb von Strukturen sowohl Variablen als auch Funktionen definiert werden

```
struct product {
   float weight; // an element of the struct
   float price;
   string unit;

   float price_per_unit(void) { // a member function
      return price/weight;
      }

   void tag(void); // a member function with external declaration

   };

void product::tag(void) { // ::  is operator of scope
   cout << price << " EUR" << endl;
   }


...

cout << melon.price_per_unit() << endl;
melon.tag();
```

Zusammenfassung (Z.B2)

- **Änderungen gegenüber ANSI-C**

– Direktiven für Datei-Ein- und Ausgaben werden über die Klasse `<fstream>` bereitgestellt

– C++ hat keinen Datentyp für Strings

– Die Klassen `<string>` und `<sstream>` erlauben das Deklarieren und Arbeiten mit Strings

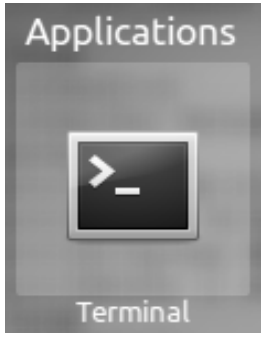
– Dynamische Speicherbelegung wird in C++ deutlich strukturierter durch die Schlüsselwörter `new` and `delete`. Nur eine sorgfältige Programmierung verhindert Speicherlecks!

– In C++ lassen sich innerhalb von Strukturen auch Memberfunktionen definieren (und die Deklaration einer Instanz kann ohne das Schlüsselwort "`struct`" erfolgen)

- **VMware Ubuntu 14.04**

ubuntu

– kurs: pw “geheim”
– Shell/Kommandozeile: Terminal
– Source-Code Editor: Kate



- **Eventuell noch zu installieren:**

– g++-Compiler und Gnuplot:

```
$ sudo apt-get install g++
$ sudo apt-get install gnuplot-x11
```

Objektorientiertes Programmieren in C++

Teil C: Objektorientierte Programmierung

Karsten Balzer

880-4659
balzer@rz.uni-kiel.de

– Objektorientierte Programmierung (OOP) im Gegensatz zur Prozeduralorientierten Programmierung (POP)

- **POP**

– "top-down"-Programmierstil. Problem wird als sequenzielle Aneinanderreihung von zu erledigenden Teilaufgaben formuliert
– Nachteile: freier und damit ungeschützer Fluss von Daten, schlechte Abbildung von "realen" Gegebenheiten und Beziehungen

- **OOP**

– "bottom-up"-Programmierstil. Problem wird als Zusammenstellung einer Anzahl von Objekten (d.h. Bausteinen von Klassen) betrachtet, die miteinander interagieren
– Vorteile: Daten werden als kritische Einheiten behandelt, Kapselung von Daten und zugeordneten Funktionen, Wiederverwendbarkeit

C++ ermöglicht Beides, POP und OOP!

4 Säulen der objektorientierten Programmierung

- **Abstrahierung** (*Abstraction*)
  - Gruppieren wesentlicher Eigenschaften (ohne das Einbinden von unnötigen Details)
- **Kapselung** (*Encapsulation*)
  - Verhinderung von direkten Datenzugriffen durch das Programm
- **Polymorphismus** (*Polymorphism*)
  - Bedeutet soviel wie "Ein Name, mehrere Formen"
  - Erlaubt mehrere Funktionen oder Prozeduren mit dem selben Namen und damit auch die Überladung von Operatoren
- **Vererbung** (*Inheritance*)
  - Prozess bei dem Objekte einer Klasse Eigenschaften von Objekten einer anderen Klasse erwerben

Resultat: Hohe Wiederverwendbarkeit von Programmteilen!

C | A | U
Christian-Albrechts-Universität zu Kiel
Rechenzentrum

- **Grundlagen I**

– Eine Klasse ist ein erweitertes Konzept einer Struktur
– Eine Instanz einer Klasse wird als “Objekt” bezeichnet. In der Sprache von Variablen ist die Klasse der Typ und ein Objekt die Variable
– Die Bausteine einer Klasse werden als Elemente (“Member”) bezeichnet. Wie in Strukturen gibt Membervariablen und Memberfunktionen
– Memberfunktionen einer Klasse sind per default inline-Funktionen
– Deklariert werden Klassen mit dem Schlüsselwort “`class`”

```
class rectangle {
   int x, y;
   int area(void);
   };

int rectangle::area(void) {
   return x*y;
   }
```

- **Grundlagen II**

– Zugriffsbeschränkungen (access specifiers)
– Schlüsselwörter “`private`”, “`public`”, “`protected`” regeln die Zugriffrechte auf die Elemente einer Klasse

– `private` Members sind nur anderen Members derselben Klasse (oder deren Freunden) zugänglich
– `public` Members sind von überall erreichbar, wo auch das Objekt sichtbar ist
– `protected` Members sind anderen Members derselben Klasse (oder deren Freunden) zugänglich, aber auch von Members abgeleiteter Klassen

```
class rectangle {
   int x, y;
   public:
      int area(void);
   };
```

C | A | U
Christian-Albrechts-Universität zu Kiel
Rechenzentrum

- **Grundlagen III**

– Eine voll funktionsfähige Klasse TUTORIAL #5

```
class rectangle {
   int x, y; // private by default (in contrast to structs!)
   public:
      void set(int xx, int yy);
      int area(void);
   };

void rectangle::set(int xx, int yy) {
   x=xx;
   y=yy;
   }

int rectangle::area(void) {
   return x*y;
   }
```

– Verwendung der Klasse im Hauptprogramm

```
rectangle r;
r.set(4,5);
cout << r.area() << endl;
```

- **Grundlagen IV**

– Für die Initialisierung von Elementen stellt C++ eine besondere Memberfunktion bereit, den Konstruktor (constructor)

– Der Konstruktor wird automatisch bei der Definition eines Objektes aufgerufen (und hat keinen Rückgabetyp)

```
class rectangle {
   int x, y;
   public:
      rectangle(int xx, int yy);
      void set(int xx, int yy);
      int area(void);
   };

rectangle::rectangle(int xx, int yy) {
   x=xx;
   y=yy;
   }
```

– Verwendung im Hauptprogramm

```
rectangle r(4,5); // initiallization via the constructor
```

C | A | U
Christian-Albrechts-Universität zu Kiel
Rechenzentrum

- **Grundlagen V**

– Default-Konstruktor (default constructor)
– Wird kein Konstruktor deklariert, dann nimmt der Compiler an, dass es einen Default-Konstruktor ohne Argumente gibt
– Um ein Objekt einer Klasse zu definieren, genügt dann ein Aufruf ohne Argumente

```
rectangle r;
```

– Aber sobald ein eigener Konstruktor definiert wurde, stellt der Compiler kein impliziten Default-Konstruktor mehr zur Verfügung

```
rectangle r; // not valid if a constructor is declared!
```

– Das Überladen des Konstruktors ist möglich

```
rectangle::rectangle(); // my own "default" constructor
rectangle::rectangle(int xx, int yy); // my second constructor
```

- **Grundlagen VI**

– Deinitialisierung mit dem Destruktor (destructor)

– Der Destruktor wird, genauso wie der Konstruktor, automatisch aufgerufen, jedoch beim Löschen eines Objekts

```
class rectangle {
   int *x, *y;
   public:
      rectangle(int xx, int yy);
      ~rectangle();
      void set(int xx, int yy);
      int area(void);
   };

rectangle::rectangle(int xx, int yy) {
   x=new int; *x=xx;
   y=new int; *y=yy;
   }

rectangle::~rectangle() {
   delete x;
   delete y;
   }
```

- **Weitere Memberfunktionen**

– Neben dem (Default-)Konstruktor und dem Destruktor gibt es noch drei weitere Memberfunktionen, die ebenfalls implizit deklariert werden, wenn keine entsprechende, eigene Funktion definiert wird

– "Default-Destruktor" (default destructor)

– "Kopier-Konstruktor" (copy constructor)

– "Kopier-Zuweisungsoperator" (copy assignment operator)

– Beispiel eines expliziten Kopier-Konstruktors TUTORIAL #6

```
rectangle::rectangle(const rectangle& r) {
   x=r.x;
   y=r.y;
   }
```

– Verwendung im Hauptprogramm

```
rectangle r1(6,7);
rectangle r2(r1);
cout << r2.area() << endl;
```

- **Zeiger auf Klassen** TUTORIAL #6

– Klassen im dynamischen Kontext

```
int N=10;
rectangle *s;

s=new rectangle[N];
s[1].set(2,7);
cout << "s[1].area()=" << s[1].area() << endl;

delete [] s;
```

– Der Pfeiloperator "->" (funktioniert genauso wie bei Strukturen)

```
rectangle *p;
rectangle r(6,7);
p=&r;

p->set(6,8); // memberfunction set of object pointed by p

cout << "r.area()=" << r.area() << endl;
```

Zusammenfassung (Z.C1)

- **Von Strukturen zu Klassen**

– Klassen erweitern das Konzept einer Struktur in C++
– Die Deklaration von Klassen erfolgt über das Schlüsselwort “`class`”
– Gegenüber von Strukturen ermöglichen Klassen Zugriffsbeschränkungen auf Members (Schlüsselwörter `private`, `public` und `protected`)
– Operationen und Methoden stellen Schnittstelle nach außen her (“black box”-Konzept)
– Zur Initialisierung, zum Kopieren und zur Speicherfreigabe gibt es innerhalb von Klassen spezielle, implizite Default-Memberfunktionen

- **Wirkung von Operatoren**

– Der "+"-Operator für Standard-Datentypen

```
int a,b,c;
c=a+b;
```

– Wirkung des "+"-Operators für eine Klasse `vector`?

```
class vector {
   int dim;
   double *v;
   public:
      ...
   } a,b,c;

c=a+b; // we know what this should mean (if dim is the same)
```

– Wirkung des "+"-Operators für die Klasse `rectangle`?

```
class rectangle {
   int x, y;
   public:
      ...
   } a,b,c;

c=a+b; // what does this mean?
```

- **Prinzip der Operatorüberladung**

– Durch die Operatorüberladung können Operatoren für Klassen oder Strukturen definiert und mit der Operatorsyntax verwendet werden

– Dies ist besonders nützlich bei Datentypen, die gute Definitionen der speziellen Operatorbedeutung aufweisen, sodass der Benutzer wenige Ausdrücke effizient einsetzen kann.

– Zu den überladbaren Operatoren gehören

```
+ - * / % (arithmetische Operatoren)
++ -- (Inkrement, Dekrement)
== != < > <= >= (Vergleichsoperatoren)
&& || !  (logische Operatoren)
= += -= *= /= (Zuweisungsoperatoren)
() [] (Funktionsaufruf, Indexoperator)
new delete new[] delete[] (Speicherbelegungsoperatoren)
```

– Wie wird in C++ das Überladen von Operatoren implementiert?

Überladung von Operatoren

- **Operatorüberladung in C++** TUTORIAL #7

– Beispiel für einer Klasse zum Arbeiten mit Vektoren

```
class vector {
   public:
      double x, y; // a two-dimensional vector (x,y)

      vector() { x=0.0; y=0.0; }
      vector(double xx, double yy) { x=xx; y=yy; }

      vector operator+(vector arg);
   };

vector vector::operator+(vector arg) {
   vector tmp; // a temporary object
   tmp.x=x+arg.x;
   tmp.y=y+arg.y;
   return tmp;
   }
```

– Verwendung (mit Default-Kopier-Zuweisungsoperator)

```
vector a(1.2,7.8);
vector b(0.9,4.1);

vector c=a+b; // implicit call

vector c=a.operator+(b); // explicit call
```

- **Das Schlüsselwort "this"** TUTORIAL #7

– "this" stellt ein Pointer auf das Objekt dar, dessen Memberfunktion ausgeführt wird
– Es ist ein Pointer auf das Objekt selbst
– Anwendung I ("Bin ich es"-Abfrage?)

```
bool myclass::isitme(myclass& arg) {
   if (&arg==this) return true;
   else return false;
   }
```

– Anwendung II (Zuweisungsoperator)

```
vector& vector::operator=(const vector& arg) {
   x=arg.x;
   y=arg.y;
   return *this; // to avoid a temporary object
   }
```

- **Statische Members**

– Statische Members sind Members (entweder Datenvariable oder Funktionen) einer Klasse, die "globale Gültigkeit" haben
– Sind unabhängig von den Instanzen (Objekten) der Klasse
– Zum Beispiel geeignet, um dynamisch festzustellen, wieviele Objekte einer Klasse momentan definiert sind

```
class myclass {
   public:
       static int n;
       myclass() { n++; }
       ~myclass() { n--; }
   };

int myclass::n=0;
```

– Verwendung im Hauptprogramm

```
cout << myclass.n << endl; // n=0
myclass a;
cout << myclass.n << endl; // --> n=1
{
myclass b;
cout << myclass.n << endl; // --> n=2
}
cout << myclass.n << endl; // --> n=1
```

- **Sog. "befreundete Funktionen" (friend functions)**

– Auf private und protected Members einer Klasse kann von außerhalb der Klasse nicht zugegriffen werden

– Diese Regel gilt aber nicht für "friends" (entweder externe Funktionen oder Klassen)

– Freundschaften werden durch einen Funktionsprototyp innerhalb der Klasse mit dem zusätzlichen Schlüsselwort "`friend`" angezeigt

– Im Allgemeinen zählen "friend functions" nicht zur objektorientierten Programmierung, d.h. wenn möglich sollten stets Members derselben Klasse benutzt werden, um Operationen durchzuführen

– Keine Transitivität

– Anwendungsbereiche

  → Ausführung von Operationen an der Schnittstelle zweier Klassen

  → insb. Konvertierung eines Objekts einer Klasse in ein Objekt einer anderen Klasse

C | A | U
Christian-Albrechts-Universität zu Kiel
Rechenzentrum

- **"Befreundete Klassen" (friend classes)**

– Beispiel: Konvertierung eines Objekts der Klasse "square" in ein Objekt der Klasse "rectangle"

```
class square;

class rectangle{ // does not have square as friend!
   private:
      int x, y;
   public:
      convert(square arg) {
         x=arg.x;
         y=arg.x;
         }
   };

class square{
   private:
      int x;
   public:
      friend class rectangle;
   };


square s(5);
rectangle r(3,7);
r.convert(s);
```

C | A | U
Christian-Albrechts-Universität zu Kiel
Rechenzentrum

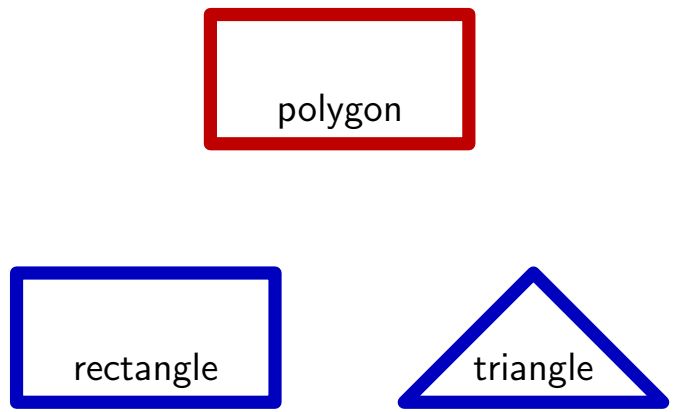
- **Vererbung (inheritance)**

– Die Möglichkeit von Klassen-Vererbungen gehört zu den zentralen Eigenschaften von C++

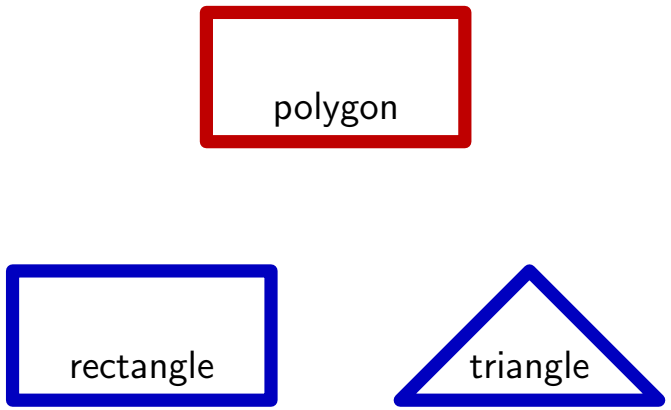
– Vererbungen ermöglichen es, Klassen zu generieren, die von anderen Klassen abgeleitet werden

– Dabei werden automatisch einige Members der "Eltern-Klasse" (der parents) und die eigenen Members eingebunden

– Einfaches Beispiel



- Die Objekte `rectangle` und `triangle` haben gemeinsame Eigenschaften; beide werden beschrieben durch zwei Parameter, Höhe und Breite
- Dieser Sachverhalt kann durch eine Klasse `polygon` representiert werden, von der die Klassen `rectangle` und `triangle` abgeleitet werden
- Die Klasse übernimmt dabei die Rolle einer sogennanten "Basisklasse" und enthält Members, die beide Typen von Polygonen gemeinsam haben
- In unserem Falle sind dies die Parameter `width` und `height`
- Die abgeleiteten Klassen enthalten dann Members, die je nach Polygon unterschiedlich sind



C | A | U
Christian-Albrechts-Universität zu Kiel
Rechenzentrum

- **Grundlagen I**

– Abgeleitete Klassen erben alle zugänglichen Members der Basisklasse

– Dies bedeutet, dass, wenn eine Basisklasse einen public oder protected Member `A` besitzt und an eine weitere Klasse mit einem Member `B` vererbt wird, dann besitzt die abgeleitete Klasse beide Members, `A` und `B`

– Syntax für Klassenvererbungen

```
class derived_class_name : public base_class_name
   {
   ...
   };
```

– Die Zugriffsbeschränkung (access specifier) `public` kann dabei durch `protected` oder `private` ersetzt werden (siehe weiter unten)

- **Grundlagen II** TUTORIAL #8

– Beispiel

```
class polygon {
   protected:
      double width, height;
   public:
      void set(double a, double b) { width=a; height=b; }
   };

class rectangle : public polygon {
   public:
      double area () { return (width*height); }
   };

class triangle : public polygon {
   public:
      double area () { return (width*height/2); }
   };
```

– Verwendung im Hauptprogramm

```
rectangle r;
triangle t;
r.set(4.0,5.0);
t.set(4.0,5.0);
cout << r.area() << endl; // --> 20.0
cout << t.area() << endl; // --> 10.0
```

- **Grundlagen III**

– Zugriff auf Members der Basisklasse

von	public	protected	private
Members der Basisklasse	ja	ja	ja
Members einer abgel. Klasse	ja	ja	nein
"Nicht-Members"	ja	nein	nein

– Steuerung von maximalen Zugriffsrechten

```
class rectangle : public polygon { ... }
```

→ `public`: Alle Members der Basisklasse werden mit den urprünglichen Zugriffsbeschränkungen vererbt

→ `protected`: Alle Members der Basisklasse werden mit der Zugriffsbeschränkung `protected` vererbt

→ `private`: Alle Members der Basisklasse werden mit der Zugriffsbeschränkung `private` vererbt

C | A | U
Christian-Albrechts-Universität zu Kiel
Rechenzentrum

- **Grundlagen IV**

– Was kann von der Basisklasse vererbt werden?
– Im Prinzip vererben Basisklassen alle Members **außer** ihre eigenen Konstruktoren und Destuktoren, ihre eigenen `operator=()`-Members sowie ihre eigenen Freunde
– Bemerkung: Default-Konstruktoren bzw. Default-Destruktoren werden jedoch aufgerufen, wenn ein neues Objekt einer abgeleiteten Klasse deklariert bzw. gelöscht wird

– Mehrfachvererbungen: Es ist möglich, dass eine Klasse Members von mehreren Basisklassen erbt
– Syntax

```
class derived_class : public base_class1, protected base_class2
   {
   ...
   };
```

- **Pointers auf Basisklassen**

– Ein Pointer auf eine abgeleitete Klasse ist Typ-kompatibel mit einem Pointer auf dessen Basisklasse

– Beispiel

```
rectangle rect;
triangle trgl;

polygon *ppoly1=&rect; // points to an object of class polygon
                       // and is referenced to rect
polygon *ppoly2=&trgl;

ppoly1->set(4.0,5.0);
ppoly2->set(4.0,5.0);

cout << rect.area() << endl; // --> 20.0
cout << trgl.area() << endl; // --> 10.0
```

– Nachteil: `ppoly1->area()` ist nicht definiert! (Siehe Definition der Klasse `polygon`)

- **Virtuelle Members** TUTORIAL #9

– Ein Member einer Klasse, dass in einer abgeleiteten Klasse erneut definiert werden kann, wird als "virtuelles Member" bezeichnet
– Die Deklaration von virtuellen Members geschieht über das Schlüsselwort `virtual`
– Beispiel

```
class polygon {
   protected:
      double width, height;
   public:
      void set(double a, double b)
         { width=a; height=b; }
      virtual double area()
         { return (0.0); }
   };
```

– Verwendung im Hauptprogramm

```
cout << ppoly1->area() << endl; // --> 20.0

polygon poly;
polygon *ppoly3=&poly;
cout << ppoly3->area() << endl; // --> 0.0
```

C | A | U
Christian-Albrechts-Universität zu Kiel
Rechenzentrum

- **Abstrakte Basisklassen**

– Eine Klasse, die ein virtuelles Member deklariert oder vererbt, wird als "polymorphe Klasse" bezeichnet

– Eine abstrakte Basisklasse ist eine polymorphe Klasse, bei der zusätzlich noch die Implementation von virtuellen Members weggelassen wird

– Beispiel

```
class polygon {
   protected:
      double width, height;
   public:
      void set(double a, double b)
         { width=a; height=b; }
      virtual double area()=0.0;
   };
```

– Die Funktion `area()` wird in dieser Form auch als "rein virtuelle Funktion" bezeichnet (pure virtual function)

– Diese Funktionalität zahlt sich z.B. aus bei Arrays von Objekten oder bei dynamisch alloziierten Objekten

C | A | U
Christian-Albrechts-Universität zu Kiel
Rechenzentrum

- **Header- und Implementierungsdateien**

– Empfehlungen zur Generierung eines übersichtlichen Quellcodes

– Klassen sollten jeweils in einer separaten Header-Datei, z.B. in "`myclass.h`", definiert werden

```
#include "myclass.h"
```

– Beachte: Die Header-Dateien der C++-Standardbibliotheken haben keine Endung "`.h`".

– Bei komplexeren Klassen, sollte die Header-Datei nur die Funktionsprototypen enthalten. Die eigentliche Implementierungen (Konstruktor, Memberfunktionen, etc.) sollten dann in eine "`.cpp`"-Datei mit demselben Namen ausgelagert werden

```
#include "myclass.h"
#include "myclass.cpp"
```

– Normalerweise wird im Hauptprogramm dann nur die "`.cpp`"-Datei eingebunden, und das Include der Header-Datei steht am Anfang dieser Implementierungsdatei TUTORIAL #10

- **Makefiles**

– Dienen zum (effizienten) Übersetzen größerer Programme mit evtl. vielen Quelldateien und Abhängigkeiten
– Ein Makefile ist üblicherweise eine Datei mit dem Namen "`Makefile`", die mit dem Quellcode zusammen abgelegt wird (Beachte die Großschreibung!)
– Programmübersetzung erfolgt dann i.d.R. durch

```
$ make
$ make prog
```

- **Aufbau eines Makefile I** TUTORIAL #11

```
# This is a makefile with target "prog"

CC = g++
CFLAGS = -Wall -O3

prog: main.cpp
	$(CC) $(CFLAGS) -o prog.ex main.cpp

# very important: space in above line must be a tab!!!
```

- **Aufbau eines Makefile II**

– Objects, Regeln und Abhängigkeiten TUTORIAL #12

```
CC = g++
CFLAGS = -Wall -O3

OBJ = myclass1.o myclass2.o


prog: main.cpp $(OBJ)
	$(CC) $(CFLAGS) -o prog.ex main.cpp

%.o: %.cpp
	$(CC) $(CFLAGS) -c $<

clean:
	rm -rf prog.ex $(OBJ)
```

– Vollständige Programmübersetzung

```
$ make clean
$ make
```

– Aktualisierung

```
$ make
```

C | A | U
Christian-Albrechts-Universität zu Kiel
Rechenzentrum

- **Klassen, Klassen, Klassen**

– Überladung von Operatoren
– Das Schlüsselwort “`this`”
– Freundschaft
– Vererbung und Mehrfachvererbungen
– Virtuelle Members und abstrakte Basisklassen
– Strukturierung von (großen) Programmpaketen: “`.h`”- und “`.cpp`”-Dateien und das `Makefile`

Objektorientiertes Programmieren in C++

Teil D: Weitere Konzepte

Karsten Balzer

880-4659
balzer@rz.uni-kiel.de

- **Was sind Templates?**

– Template bedeutet soviel wie “Schablone” oder “Mustervorlage”
– In C++ stellen Templates ein Mittel zur Typparametrierung bereit
– Man unterscheidet zwischen Funktionen-Templates und Klassen-Templates

– Funktionen-Templates sind spezielle Funktionen, die mit einem “generischen Typ” umgehen können. Dabei wird ein sog. “Template-Parameter” als spezieller Parameter verwenden, um im Gegensatz zu Werten auch Typen zu übergeben

– Klassen-Templates ermöglichen, dass Members die Template-Parameter als Typen verwenden

- **Funktionen-Templates I**

– Definition einer Templates für eine Funktion `max()`

```
template <typename T>
T max(T& a, T& b) {
   if (a>b) return a;
   else return b;
   }
```

– Verwendung im Hauptprogramm TUTORIAL #13

```
int a=17, b=9;
int c=max<int>(a,b);

float a=5.136, b=6.7;
float c=max<float>(a,b);


rectangle a(4,5), b(4,5);
// if operator > has been overloaded in class "rectangle"
rectangle c=max<rectangle>(a,b);


rectangle a(4,5), b(4,5);
// usually the compiler knows which data type to instantiate
rectangle c=max(a,b);
```

C | A | U
Christian-Albrechts-Universität zu Kiel
Rechenzentrum

- **Funktionen-Templates II**

– Templates mit mehreren Typ Parametern unterschiedlicher Art

```
template <typename T, typename U>
U min(T& a, U& b) {
   if (a>b) return a;
   else return b;
   }
```

– Verwendung im Hauptprogramm

```
int i=37;
long j=149;
long k;

k=min(i,j);
```

- **Klassen-Templates**

– Bei Klassen-Templates können die Members Template-Paramteter als Typen verwenden

```
template <typename T>
class pair {
   private:
      T values[2];
   public:
      set(T v1, T v2) {
         value[0]=v1;
         value[1]=v2;
         }
      // further member functions ...
   }
```

– Verwendung im Hauptprogramm

```
pair<int> a;

pair<rectangle> b;
```

C | A | U
Christian-Albrechts-Universität zu Kiel
Rechenzentrum

- **Template-Spezialisierung** TUTORIAL #14

– Um eine andere Implementierung für ein Template zu definieren, falls ein bestimmter Typ als Template-Parameter übergeben wird

```
template <typename T> class count { // class template
   T value;
   public:
      count() { value=0; }
      void increase();
      T get();
   };

template <typename T> void count<T>::increase() { value+=1; }
template <typename T> T count<T>::get() { return value; }

template <> class count<char> { // class template specialization
   char value;
   public:
      count() { value='a'; }
      void increase();
      char get();
   };

void count<char>::increase() { value+=1; }
char count<char>::get() { return value; }
```

- **Was sind Namespaces?**

– Namensräume (namespaces) erlauben es, Einheiten wie Klassen, Objekte und Funktionen unter einem Namen zusammenzufasssen (Abgrenzung von Gültigkeitsbereichen)

```
namespace math {
   double e, m;
   }

namespace physics {
   double e, m;
   }
```

– Verwendung

```
cout << math::e << endl;
cout << physics::e << endl;
```

– Das Schlüsselwort "using" wird verwendet um einen Namensraum "sichtbar", d.h. bekannt, zu machen

```
math::e=2.718; physics::e=1.602e-19;
using namespace math;
cout << e << endl; // --> 2.71
```

C | A | U
Christian-Albrechts-Universität zu Kiel
Rechenzentrum

- **Verwendung von Namespaces**

– Wir haben schon oft mit dem Namensraum “std” der C++-Standardbibliothek gearbeitet

– Zum Beispiel sind die Standardausgabe “cout” und der Zeilenvorschub “endl” Elemente dieses Namensraumes

```
std::cout << "Hello world!" << std::endl;
```

```
using namespace std;
cout << "Hello world!" << endl;
```

– Namespace alias

```
namespace mystd=std;
mystd::cout << "Hello world!" << mystd::endl;
```

C | A | U
Christian-Albrechts-Universität zu Kiel
Rechenzentrum

- **Warum Parallelisierung?**

– Vorteile:
– Nutzung von Recourcen mehrerer CPUs und unterschiedlicher Rechnerarchitekturen
– Nutzung von mehr Arbeitsspeicher (RAM)
– Ermöglicht das schnellere Lösen eines Problems
– Ermöglicht das Lösen von Problemen, welche mit einem seriellen Programm nicht möglich sind

C | A | U
Christian-Albrechts-Universität zu Kiel
Rechenzentrum

- **OpenMP (Open Multi-Processing)**
  - Compiler-Direktiven-basierte Programmierung
  - Die Direktiven teilen den Prozessoren mit, wie Daten und Berechnungen verteilt werden sollen
  - Die Direktiven erscheinen als spezielle Kommentare in einem seriellen Code
  - Steht nur auf "Shared-Memory" Architekturen zur Verfügung

- **MPI (Message Passing Interface)**
  - Nutzt ein Nachrichtenprotokoll, um Daten zwischen mehreren MPI-Prozessen auszutauschen
  - Jeder MPI-Prozess hat dabei seine eigenen lokalen Variablen
  - Steht auf "Shared-Memory" und "Distributed-Memory" Architekturen zur Verfügung

- **OpenMP und MPI unter C++**
  - Jeweils zwei einfache Beispiele

- **Beispiel 1** TUTORIAL #omp1

```
#include <omp.h>

#include <iostream>
#include <cmath>
using namespace std;

int main()
   {
   int i;
   double *v=new double[50000000];

   #pragma omp parallel for
   for(i=0;i<50000000;i++) {
      v[i]=cos((double)i);
      }

   delete [] v;

   return 0;
   }
```

```
$ g++ -fopenmp -o omp1.ex omp1.cpp
$ export OMP_NUM_THREADS=2
$ time ./omp1.ex
```

- **Beispiel 2** TUTORIAL #omp2

```
#include <omp.h>

#include <iostream>
#include <cmath>
using namespace std;

int main()
   {
   int i;
   double w;
   double sum=0.0;

   #pragma omp parallel for private(w) reduction(+:sum)
   for(i=0;i<50000000;i++) {
      w=sqrt((double)i);
      sum=sum+cos(w);
      }

   cout << sum << endl;

   return 0;
   }
```

```
$ g++ -fopenmp -o omp2.ex omp2.cpp
$ export OMP_NUM_THREADS=2
$ time ./omp2.ex
```

- **Beispiel 1** TUTORIAL #mpi1

```
#include <mpi.h>

#include <iostream>
using namespace std;

int main(int argc, char *argv[])
   {
   int i, j, sum=0;

   int rank, nranks;
   MPI_Init(&argc, &argv); // MPI initialization
   MPI_Comm_rank(MPI_COMM_WORLD, &rank); // get current process id
   MPI_Comm_size(MPI_COMM_WORLD, &nranks); // get number of processes

   cout << "Hello world from process "
        << rank+1 << " of " << nranks << "!" << endl;

   MPI_Finalize(); // MPI deinitialization

   return 0;
   }
```

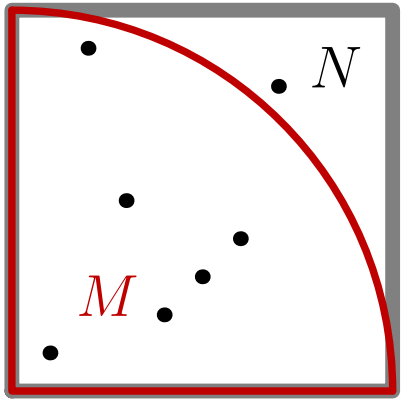
```
$ mpic++ -o mpi1.ex mpi1.cpp
$ time mpirun -np 2 ./mpi1.ex
```

C | A | U
Christian-Albrechts-Universität zu Kiel
Rechenzentrum

- **Beispiel 2 – Monte-Carlo Berechnung von π**

– Wähle zufällig N Punkte in einem Einheitsquadrat und bilde das Verhältnis $4M/N = \pi$

$$r^2 = 1 = x^2 + y^2$$




```
double compute_pi(int N) {
   int M=0;
   for(int n=0;n<N;n++) {
      const double x=((double)rand())/((double)RAND_MAX);
      const double y=((double)rand())/((double)RAND_MAX);
      if (x*x+y*y<=1.0) M++;
      }
   return 4.0*((double)M)/((double)N);
   }
```

- **Beispiel 2 – Monte-Carlo Berechnung von π** TUTORIAL #mpi2

```
#include <mpi.h>
#include <iostream>
#include <iomanip>
#include <cstdlib>
using namespace std;
int main(int argc, char *argv[])
   {
   int rank, nranks;
   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   MPI_Comm_size(MPI_COMM_WORLD, &nranks);
   int N=1000000;
   double local_pi, global_pi=0.0;
   srand(rank); // this is essential!!!
   local_pi=compute_pi(N);
   MPI_Reduce(&local_pi, &global_pi, 1, MPI_DOUBLE, MPI_SUM, 0,
MPI_COMM_WORLD);
   if (rank==0) cout << setprecision(10) << fixed << "pi(" << nranks
<< ")=" << global_pi/((double)nranks) << endl;
   MPI_Finalize();
   return 0;
   }
```

C | A | U
Christian-Albrechts-Universität zu Kiel
Rechenzentrum

- **OpenMP +**

– Leichter zu programmieren und zu debuggen als MPI. Quellcode ist leichter verständlich
– Direktiven können schrittweise eingebaut werden (schrittweise Parallelisierung)
– Der Code kann immer als serieller Code genutzt werden
– Serielle Codeteile benötigen keine Modifikation

- **OpenMP -**

– Code läuft nur auf Rechnern mit “Shared Memory”
– Erfordert einen Compiler der OpenMP unterstützt
– Meistens zur Parallelisierung von Schleifen verwendet

OpenMP vs. MPI I

- **MPI +**

– MPI-Programme laufen sowohl auf “Shared-Memory” als auch auf “Distributed-Memory” Architekturen
– Kann auf eine deutlich größere Anzahl von Problemen angewendet werden
– Jeder Prozess hat seine eigenen lokalen Variablen

- **MPI -**

– Erfordert in der Regel deutliche Modifikationen des Quellcodes
– Kann schwerer sein zu debuggen
– Performance ist begrenzt durch das Kommunikationsnetzwerk zwischen den Rechenknoten

- **Erforderliche Pakete**

– Installiere die folgenden Packages

```
$ sudo apt-get install libcr-dev mpich2 mpich2-doc
```

- **Kompilierung**

– Beispiele für eine Quellcode-Übersetzung

```
$ g++ -fopenmp -o test.ex test.cpp
$ mpic++ -o test.ex test.cpp
$ mpic++ -fopenmp -o test.ex test.cpp
```

– Reiner OpenMP-Aufruf

```
$ export OMP_NUM_THREADS=2; ./test.ex
```

– Reiner MPI-Aufruf

```
$ mpirun -np 2 ./test.ex
```

The End

– Gibt es Anmerkungen, Wünsche, ... ?

- **Abfrage des Datentyps**

– Zur Bestimmung des Datentyps zur Laufzeit eines Programmes kann der folgende Quellcode verwendet werden:

```
#include <iostream>
#include <typeinfo>

using namespace std;

int main()
   {
   double x=5.6;
   cout << typeid(x).name() << endl; // --> d
   }
```