Einführung in die Programmierung mit C++

Nutzerdefinierte Datentypen / Klassen

Uwe Naumann

Informatik 12:
Software and Tools for Computational Engineering (STCE)

RWTH Aachen

Inhalt

Outline

Nutzerdefinierte Datentypen / Klassen

Deklaration, Allokation, Gültigkeitsbereiche


```
1  #include<iostream>
2
3  struct pair {
4    int f=0; bool s=false;
5  };
6
7  int main() {
8    pair p;
9    p.f=42; p.s=true;
10   std::cout << p.f << " "
11       << p.s << std::endl;
12   return 0;
13 }
```

- Nutzerdefinierten Datentypen (auch: Klassen) sind Kollektionen von Variablen (später auch Funktionen) (potentiell) verschiedener Typen.
- Deren Gültigkeit ist auf den einschließenden Gültigkeitsbereich (hier: global) begrenzt.
- Instanzierung von Klassen alloziert Objekte des entsprechenden Typs (hier im statischen Speicher); deren Elemente (auch: Objektvariablen) sollten explizit initialisiert werden.
- Objekte existieren im einschließenden Gültigkeitsbereich (hier: main; der belegte statische Speicher wird bei Verlassen des Gültigkeitsbereichs automatisch freigegeben.
- Zugriff auf die Objektvariablen erfolgt mithilfe des . Operators.

globale Variable:

```
1  #include<iostream>
2
3  struct pair {
4    int f=0; bool s=false;
5  } p;
6
7  int main() {
8    p.f=42; p.s=true;
9    std::cout << p.f << " "
10       << p.s << std::endl;
11   return 0;
12 }
```

lokale Klasse:

```
1  #include<iostream>
2
3  int main() {
4    struct pair {
5      int f=0; bool s=false;
6    } p;
7    p.f=42; p.s=true;
8    std::cout << p.f << " "
9         << p.s << std::endl;
10   return 0;
11 }
```

```
1  #include<iostream>
2
3  struct pair {
4    int f=0; bool s=false;
5  };
6
7  int main() {
8    pair p,*pp=&p,&pr=p;
9    pp->f=42; pr.s=true;
10   std::cout << p.f << " "
11       << pp->s << std::
           endl;
12   return 0;
13 }
```

- Zeiger auf Objekte können definiert werden. Zugriff auf die Elemente eines Objekts über einen Zeiger darauf erfolgt durch Verwendung des -> Operators.
- Referenzen auf Objekte können definiert werden. Zugriff auf die Elemente eines Objekts über eine Referenz dafür erfolgt durch Verwendung des . Operators.

→ Speicherlayout: gdb

```
1  #include<iostream>
2
3  struct pair {
4    int f=0; bool s=false;
5  };
6
7  int main() {
8    const size_t n=3;
9    pair p[n];
10   for (size_t i=0;i<n;i++) {
11     p[i].f=42-i;
12     p[i].s=i%2;
13   }
14   std::cout << p->f << " "
15       << (*(p+2)).s << std::endl;
16   return 0;
17 }
```

- Statische Felder von Objekten können definiert werden. Zugriff auf die Elemente des i-ten Eintrags erfordert Dereferenzierung gefolgt von Verwendung des . Operators.
- Aufgrund der Implementierung statischer Felder als konstante Zeiger auf den ersten Eintrag ergeben sich diverse Alternativen für den Datenzugriff, z.B. p[i].f oder (p+i)->f oder (*(p+i)).f.

→ Speicherlayout: gdb

Felder von Objekten

Varianten


```cpp
#include<iostream>
#include<array>

struct pair {
  int f=0; bool s=false;
};

int main() {
  const size_t n=3;
  std::array<pair,n> p;
  for (size_t i=0;i<n;i++) {
    p[i].f=42-i;
    p[i].s=i%2;
  }
  std::cout << p[1].f << " "
    << p[2].s << std::endl;
  return 0;
}
```

```cpp
#include<iostream>
#include<vector>

struct pair {
  int f=0; bool s=false;
};

int main() {
  size_t n=3;
  std::vector<pair> p;
  for (size_t i=0;i<n;i++) {
    pair q; q.f=42-i; q.s=i%2;
    p.push_back(q);
  }
  std::cout << p[1].f << " "
    << p[2].s << std::endl;
  return 0;
}
```

Klassen

Schachtelung von Objekten


```
1  #include<iostream>
2
3  namespace toy {
4    struct cond {
5      bool c=false;
6    };
7  }
8
9  struct pair {
10   int f=0; toy::cond s;
11 };
12
13 int main() {
14   pair p;
15   p.f=42; p.s.c=true;
16   std::cout << p.f << " "
17         << p.s.c << std::endl;
18   return 0;
19 }
```

- Klassen können Namensbereichen zugeordnet sein; Zugriff auf den Typ erfordert Spezifikation des Namensbereichs.
- Klassen können Variablen anderer Klassen enthalten.
- Zugriff auf die Objektvariablen erfolgt mittels Sequenzen des . Operators.

Outline

```
1  #include<iostream>
2
3  struct pair {
4    struct cond {
5      bool c=false;
6    };
7    int f=0; cond s;
8  };
9
10 int main() {
11   pair p; pair::cond c;
12   p.f=42; c.c=true; p.s=c;
13   std::cout << p.f << " "
14         << p.s.c << std::endl;
15   return 0;
16 }
```

- Klassen können inerhalb von Klassen definiert werden.
- Zugriff auf die enthaltene Klasse erfordert Angabe des Names der enthaltenden Klasse.
- Objekte gleichen Typs (Instanzen derselben Klasse) die ausschließlich statischen Speicher belegen können einander zugewiesen werden.

```
1  #include<iostream>
2
3  struct pair {
4    int f=0; bool s=false;
5  };
6
7  void print(const pair& p) {
8    using namespace std;
9    cout << p.f << " "
10         << p.s << endl;
11 }
12
13 int main() {
14   pair p;
15   p.f=42; p.s=true;
16   print(p);
17   return 0;
18 }
```

- Objekte können an Funktionen "by [const] value" sowie "by [const] reference" übergeben werden.
- Übergabe "by [const] reference" vermeidet das Anlegen lokaler Kopien, was für große (hoher Speicherbedarf) Objekte zu verbesserter Effizienz des Programms führt.

→ Live: Fallstudie

Übergabe von Objekten an Funktionen

Fallstudie


```
1  #include<iostream>
2
3  struct pair {
4    int f=0; double s[100000];
5  };
6
7  void print(const pair& p) {
8    using namespace std;
9    cout << &p.s << endl;
10 }
11
12 int main() {
13   pair p;
14   for (size_t i=0;i<100000;print(p),i++) std::cout << i << ": ";
15   return 0;
16 }
```

`/usr/bin/time -v` erlaubt Laufzeitmessung und Bestimmung des Speicherbedarfs.

Outline

```
1  #include<iostream>
2
3  struct pair {
4    int f=0; bool s=false;
5
6    void print() {
7      using namespace std;
8      cout << f << " "
9           << s << endl;
10   }
11 };
12
13 int main() {
14   pair p;
15   p.f=42; p.s=true;
16   p.print();
17   return 0;
18 }
```

- Objektfunktionen haben direkten Zugriff auf die Objektvariablen.
- Sie dienen der Modifikation / Abfrage des Zustands von Objekten (Werte der Objektvariablen).
- Ihre Deklaration muss innerhalb der Klassendeklaration geschehen.
- Ihre Definition kann außerhalb der Klassendeklaration geschehen, z.B

```
struct pair { ... void print(); };
void pair::print() { ... }
```

Klassen

Schnittstelle


```
1  #include<iostream>
2
3  struct pair {
4    int f=0; bool s=false;
5    int get_f() const;
6    void set_f(const int&);
7    bool get_s() const;
8    void set_s(const bool&);
9  };
10
11 int pair::get_f() const { return f; }
12 void pair::set_f(const int &i) { f=i; }
13 bool pair::get_s() const { return s; }
14 void pair::set_s(const bool &i) { s=i; }
15
16 int main() {
17   pair p;
18   p.set_f(42); p.set_s(true);
19   std::cout << p.get_f() << " "
20       << p.get_s() << std::endl;
21   return 0;
22 }
```

- Spezielle set- und get-Routinen erlauben die Entkopplung des Datenlayouts von der Nutzerschnittstelle.
- Das **const** als Teil der Signatur der get-Routinen markiert diese als “den Zustand des Objekts nicht modifizierend”.
- Zugriffsrechte (siehe später) ermöglichen das Verbot eines direkten Zugriffs auf die Objektvariablen.
- Alternativ kann man sowohl Lese- als auch Schreibzugriff durch Rückgabe einer (nicht konstanten) Referenz auf die Objektvariablen ermöglichen, z.B.

```
int& pair::first() { return f; }
```

Klassen

Zugriffsrechte


```
1  #include<iostream>
2
3  class pair {
4    int f=0; bool s=false;
5  public:
6    int& first(); bool& second();
7  };
8
9  int& pair::first() { return f; }
10 bool& pair::second() { return s; }
11
12 int main() {
13   pair p;
14   p.first()=42; p.second()=true;
15   std::cout << p.first() << " "
16        << p.second() << std::endl;
17   return 0;
18 }
```

- Als **struct** definierte Klassen erlauben externen Zugriff auf alle Objektvariablen und -funktionen (alles **public**).
- Als **class** definierte Klassen verbieten externen Zugriff auf alle Objektvariablen und -funktionen (alles **private**). Zugriffsrechte müssen mittels **public**: explizit erteilt werden.
- Objektvariablen sollten in der Regel **private** sein ($\rightarrow$ Datenkapselung).
- Objektfunktionen, die **public** deklariert sind, definieren die Schnittstelle der Klasse (der abgeleiteten Objekte).

In Ausnahmefällen kann man mittels **friend** externen Funktionen bzw. Klassen Zugriff auf die privaten Variablen und Funktionen gewähren (kann meistens / sollte vermieden werden).

Outline

```
1  #include<iostream>
2
3  template<typename F, typename S>
4  class pair {
5    F f; S s;
6  public:
7    F& first(); S& second();
8  };
9
10 template<typename F, typename S>
11 F& pair<F,S>::first() { return f; }
12 template<typename F, typename S>
13 S& pair<F,S>::second() { return s; }
14
15 int main() {
16   pair<int,bool> p;
17   p.first()=42; p.second()=true;
18   std::cout << p.first() << " "
19       << p.second() << std::endl;
20   return 0;
21 }
```

- Typgenerische Klassen ermöglichen die Instanzierung durch den Compiler bei Verwendung mit spezifischen Typen.
- Bei externer Definition der typgenerischen Objektfunktionen ist die Zugehörigkeit zur entsprechenden typgenerischen Klasse anzugeben.
- Typgenerik sollte fixer Typisierung aufgrund der verbesserten Flexibilität vorgezogen werden.

```
1  #include<iostream>
2
3  template<int N, typename T>
4  struct ensemble {
5    T v[N];
6  };
7
8  int main() {
9    ensemble<3,int> e;
10   e.v[0]=1; e.v[1]=2; e.v[2]=3;
11   for (size_t i=0;i<4;i++)
12     std::cout << *(e.v+i) << " ";
13   std::cout << std::endl;
14   return 0;
15 }
```

- Neben Typgenerik können Klassen auch für verschiedene (zur Übersetzungszeit variable) **int**-Werte instanziert werden, z.B. für variable Größen eines enthaltenen statischen 1D-Feldes.
- statische 1D-Feldgrößen werden - wie gehabt - bei Elementzugriff nicht durch den Compiler überprüft, z.B. könnte durch nebenstehendes Programm

  `1 2 3 1367297536`

  ausgegeben werden.

Outline

Klassen

Konstruktoren / dynamische Initialisierung


```
1  #include<iostream>
2
3  template<typename F, typename S>
4  class pair {
5    F f; S s;
6  public:
7    pair(const F &f, const S &s)
8      : f(f), s(s) /* Initialisierung */ {}
9    F& first(); S& second();
10 };
11
12 template<typename F, typename S>
13 F& pair<F,S>::first() { return f; }
14 template<typename F, typename S>
15 S& pair<F,S>::second() { return s; }
16
17 int main() {
18   pair<int,bool> p(42,true);
19   std::cout << p.first() << " "
20       << p.second() << std::endl;
21   return 0;
22 }
```

- Nutzerdefinierte Konstruktoren werden bei Allozierung von Objekten aufgerufen. Ihr Name ist gleich dem der Klasse. Sie müssen innerhalb der Klasse deklariert werden und können innerhalb (wie hier) oder außerhalb der Klasse definiert werden.
- Konstruktoren verschiedener Signaturen können zur dynamischen (zur Laufzeit des Programms) Initialisierung der Objektvariablen eingesetzt werden.
- Ein Standardkonstruktor wird durch den Compiler automatisch generiert. Er alloziert alle Objektvariablen ohne diese zu initialisieren.

Klassen

Destruktor


```
1  #include<iostream>
2
3  template<typename F, typename S>
4  class pair {
5    F f; S s;
6  public:
7    pair(const F &f, const S &s)
8      : f(f), s(s) {}
9    ~pair() {
10     std::cout << "Later!" << std::endl;
11   }
12   F& first(); S& second();
13 };
14
15 // ... definition of first and second
16
17 int main() {
18   pair<int,bool> p(42,true);
19   std::cout << p.first() << " "
20       << p.second() << std::endl;
21   return 0;
22 }
```

- Ein (potentiell nutzerdefinierter) Destruktor `~T()` mit fixer Signatur wird bei Deallozierung von Objekten der Klasse `T` aufgerufen, z.B. hier Ausgabe:

  ```
  42 1
  Later!
  ```

- Ein Standarddestruktor wird durch den Compiler automatisch generiert. Er deallokiert alle Objektvariablen und gibt so den durch das Objekt belegten statischen Speicher frei.
- Auf dem aktuellen Stand der Lehrveranstaltung ist die Definition eines speziellen Destruktors noch nicht notwendig. Das wird sich ändern ...

Outline

```
1  #include<iostream>
2
3  template<typename F, typename S>
4  class pair {
5    F f; S &s;
6  public:
7    pair(const F &f, S &s) : f(f), s(s) {}
8    F& first(); S& second();
9  };
10
11 template<typename F, typename S>
12 F& pair<F,S>::first() { return f; }
13 template<typename F, typename S>
14 S& pair<F,S>::second() { return s; }
15
16 int main() {
17   bool b=true;
18   pair<int,bool> p(42,b);
19   std::cout << p.first() << " "
20       << p.second() << std::endl;
21   return 0;
22 }
```

- Referenzen innerhalb von Objekten müssen (wie immer) bei Konstruktion initialisiert werden.
- Entsprechende Konstruktoren werden benötigt.
- Konstante Referenzen müssen auch mit solchen initialisiert werden, z.B.

```
1
2    ... const S &s;
3  public:
4    pair(const F &f, const S &s) ...
```

und second() muss eine konstante Referenz zurückgeben.

```
1  ...
2  template<typename F, typename S>
3  class pair {
4    F f; S *s=nullptr;
5  public:
6    pair(const F &f) : f(f) {}
7    pair(S *s) : s(s) {}
8    ... S& second();
9  };
10 ...
11 template<typename F, typename S>
12 S& pair<F,S>::second() {
13   if (s) return *s; else assert(false);
14 }
15
16 int main() {
17   bool b=true; { pair<int,bool> p(&b);
18   std::cout << p.second() << std::endl; }
19   { pair<int,bool> p(42);
20   std::cout << p.second() << std::endl; }
21   return 0;
22 }
```

- Zeiger innerhalb von Objekten sollten (wie immer) bei Konstruktion initialisiert werden.
- Im Allgemeinen bietet sich eine Initialisierung mit nullptr an.
- Spezielle Konstruktoren werden für andere Initialwerte benötigt.
- Zeiger auf Konstanten erlauben nur Lesezugriff auf (nicht konstante ...) Variablen, z.B.

```
1
2    ... const S *s;
3  public:
4    pair(S *s) ...
```

und second() muss eine konstante Referenz zurückgeben.

AUFWACHEN!

Was wird durch das folgende Programm ausgegeben?

```
1  #include<iostream>
2
3  template<typename T>
4  class pair {
5    const pair* f=nullptr;
6    const T& s;
7  public:
8    pair(const pair& f, const T& s) : f(&f), s(s) {}
9    void print() {
10     std::cout << f->f->f->s+1 << s-1 << std::endl;
11   }
12 };
13
14 int main() {
15   pair<int> p(p,3),*pp=&p; pp->print();
16   return 0;
17 }
```

Wach?

```
1  #include<iostream>
2
3  template<typename T>
4  class pair {
5    const pair* f=nullptr;
6    const T& s;
7  public:
8    pair(const pair& f, const T& s) : f(&f), s(s) {}
9    void print() {
10     std::cout << f->f->f->s+1 << s-1 << std::endl;
11   }
12 };
13
14 int main() {
15   pair<int> p(p,3),*pp=&p; pp->print();
16   return 0;
17 }
```

Was sonst ...?

Outline

Klassen

Klassenvariablen


```
1  #include<iostream>
2
3  template<typename F, typename S>
4  class pair {
5    F f; S s;
6  public:
7    static std::string type;
8    F& first(); S& second();
9  };
10
11 template<typename F, typename S>
12 std::string pair<F,S>::type="pair";
13 // ... definition of first and second
14
15 int main() {
16   pair<int,bool> p; pair<float,char> q;
17   std::cout << p.type << " " << q.type
18     << " " << pair<int,bool>::type << " "
19     << pair<float,char>::type << std::endl;
20   return 0;
21 }
```

- Klassenvariablen (**static**) werden durch alle Instanzen (Objekte) geteilt.
- Deren Initialisierung erfolgt einmalig außerhalb der Klassendeklaration.
- Objektfunktionen können auf Klassenvariablen zugreifen.
- Externer Zugriff ist nur auf **public** Klassenvariablen möglich.
- Der Zugriff auf Klassenvariablen erfolgt über das Objekt (. oder, bei Zeiger auf Objekt, −> Notation) bzw. den Klassennamen (:: Notation) unter Beachtung potentieller Typgenerik.

```
1  template<typename F, typename S>
2  class pair {
3    static std::string t;
4    F f; S s;
5  public:
6    static const std::string& type() {
7      return t;
8    }
9    F& first(); S& second();
10 };
11
12 template<typename F, typename S>
13 std::string pair<F,S>::t="pair";
14 // ... definition of first and second
15
16 int main() {
17   pair<int,bool> p;
18   std::cout << p.type() << " " <<
19       pair<int,bool>::type() << std::endl;
20   return 0;
21 }
```

- Klassenfunktionen (**static**) werden durch alle Instanzen (Objekte) geteilt
- Deren Deklaration erfolgt innerhalb der Klassendeklaration.
- Die Definition kann inner- oder außerhalb der Klassendeklaration erfolgen.
- Klassenfunktionen erlauben keinen Zugriff auf Objektdaten. Sie werden daher meist für den Lese-/Schreibzugriff auf Klassenvariablen verwendet.

Klassenvariablen und -funktionen

Fallstudie: Objektzähler


```
1  #include<iostream>
2
3  template<typename F, typename S>
4  class pair {
5    static size_t c;
6    F f; S s;
7  public:
8    pair() { c++; }
9    ~pair() { c--; }
10   static const size_t& count() { return c; };
11   F& first(); S& second();
12 };
13
14 // ... definition of pair<F,S>::c=0, first and second
15
16 int main() {
17   pair<int,bool> p1,p2,p3;
18   pair<float,char> q1,q2;
19   std::cout << pair<int,bool>::count() << std::endl;
20   std::cout << pair<float,char>::count() << std::endl;
21   return 0;
22 }
```

AUFWACHEN!

Was wird nach Aufruf des Programms `bla` mittels `./bla bla bla` ausgegeben?

```
1  #include<iostream>
2
3  struct A {
4    static int c;
5    A() { c++; }
6  };
7
8  int A::c=0;
9
10 int main(int c, char* v[]) {
11   for (int i=0;i<c;i++) A a;
12   std::cout << v[0] << " " << A::c << std::endl;
13   return 0;
14 }
```

Wach?

```
1  #include<iostream>
2
3  struct A {
4    static int c;
5    A() { c++; }
6  };
7
8  int A::c=0;
9
10 int main(int c, char* v[]) {
11   for (int i=0;i<c;i++) A a;
12   std::cout << v[0] << " " << A::c << std::endl;
13   return 0;
14 }
```

```
./bla 3
```

Outline

Nutzerdefinierte Datentypen

union


```
1  #include<iostream>
2
3  union pair {
4    int f; double s;
5  };
6
7  int main() {
8    pair p;
9    p.f=42;
10   std::cout << p.f << " "
11       << p.s << std::endl;
12   p.s=4.2;
13   std::cout << p.f << " "
14       << p.s << std::endl;
15   return 0;
16 }
```

- **union** hat C++ aus C "geerbt."
- Während **struct** der AND-Logik entspricht (alle Elemente in separaten Speicherbereichen), folgt **union** der XOR-Logik (alle Elemente teilen sich Speicherbereich mit derselben Basisadresse)
- **union** ist mit Vorsicht zu genießen, da unerwünschte Effekte durch implizite Typumwandlung auftreten können, z.B.

```
42 2.07508e-322
-858993459 4.2
```

AUFWACHEN!

Was wird durch das folgende Programm ausgegeben?

```
1  #include<iostream>
2
3  union pair {
4    short f; char s[2];
5  };
6
7  int main() {
8    pair p; p.f=42; p.s[0]=48; p.s[1]=49;
9    std::cout << p.f << std::endl
10       << p.s[0] << " " << p.s[1] << std::endl;
11   return 0;
12 }
```

Wach?

Was wird durch das folgende Programm ausgegeben?

```
1  #include<iostream>
2
3  union pair {
4    short f; char s[2];
5  };
6
7  int main() {
8    pair p; p.f=42; p.s[0]=48; p.s[1]=49;
9    std::cout << p.f << std::endl
10       << p.s[0] << " " << p.s[1] << std::endl;
11   return 0;
12 }
```

12592
0 1

Warum?

Übrigens

Übersetzung von

```
1  // naumann@stce.rwth-aachen.de
2
3  #include<iostream>
4
5  struct pair {
6    int f=0; bool s=false;
7  };
8
9  int main() {
10   using namespace std;
11   pair p; p.f=42; p.s=true; cout << p.f << " " << p.s << endl;
12   return 0;
13 }
```

schlägt (korrekterweise) fehl ...

```
... :

...:11:3: error: reference to ‘pair’ is ambiguous
   11 |   pair p; p.f=42; p.s=true; cout << p.f << " " << p.s << endl;
...
...:5:8: note: candidates are: ‘struct pair’
    5 | struct pair {
...
/usr/include/c++/9/bits/stl_pair.h:208:12: note:
‘template<class _T1, class _T2> struct std::pair’
  208 |     struct pair
...
```

Zusammenfassung

Grundlegendes zu Klassen

Schachtelung von Klassen

Zugriffsrechte und Schnittstelle

Typgenerik

Konstruktoren und Destruktor

Referenzen und Zeiger in Objekten

Klassenvariablen und -funktionen

union