Klassen, Templates, STL

recycelte Version von letztem Jahr

Structs

- bauen neue Datentypen aus bestehenden Datentypen
- Syntax:

```
struct Getraenk {
     double groesse;
     double fuellstand;
}

Getraenk cola;
cola.groesse    = 0.4;
cola.fuellstand = 0.4;
```

```
void trinken(Getraenk g) {
     if (g.fuellstand > 0){
          g.fuellstand -= 0.02;
          cout << “Gluck! Gluck!” << endl;
     }
}
```

Klassen

```
class Getraenk {
     double groesse;
     double fuellstand;
public:
     void trinken() {
          if (fuellstand > 0){
               fuellstand -= 0.02;
               cout << “Gluck! Gluck!” << endl;
          }
     }
};
```

- implementieren Methoden, klasseninterne Funktionen, die auf den Attributen arbeiten können.
- kein Zugriff auf ‘private’ Member von außen

```
Getraenk cola;
```

Getraenk: Klasse
cola: Objekt

Getter / Setter

```
double Getraenk::get_groesse(){
    return groesse;
}

void Getraenk::set_groesse(double groesse){
    this->groesse = groesse;
}
```

Konstruktor / Destruktor

```
Getraenk(){
    groesse = 1.0;
    fuellstand = 1.0;
}

Getraenk(double g, double f)
    :groesse(g),fuellstand(f){}

Getraenk(double g):Getraenk(g, g){}
```

```
~Getraenk(){
    cout << “War’s lecker?”
    << endl;
}
```

const correctness

- Methoden können `const` sein. Das impliziert zwei Dinge:
  - kann keine Attribute d. Klasse ändern
  - kann von `const`-Instanzen der Klasse aufgerufen werden

Syntax:

```
double get_fuellstand() const { return fuellstand; }
```

static attributes / methods

Statische Attribute und Methoden gehören zur Klasse, statt zum Objekt.

```
static int num_glaeser = 0;

static void rechnung(){
     cout << “Ihr müsst “ << Getraenk::num_glaeser
     << “ bezahlen.” << endl;
}

Getraenk(){
     groesse = 1.0;
     fuellstand = 1.0;
     Getraenk::num_glaeser++;
}
```

Vererbung

```
class Spirituose: public Getraenk {
     double alkoholgehalt;
public:
     void trinken() {
          if (fuellstand > 0){
               fuellstand -= 0.02;
               cout << “Gluck! Gluck! Hicks!”
               << endl;
          }
     }
};
```

- übernimmt Attribute und Mathoden der Basis-Klasse
- kann Attribute ergänzen
- kann Methoden ergänzen und überladen

Vererbungstypen

Vererbungs-Typ

Member-Typ	**public**	**protected**	**private**
public	public	protected	private
protected	protected	protected	private
private	private	private	private

Polymorphe Pointer

Basis-Klassen-Pointer können auf abgeleitete Instanzen zeigen.
Das dereferenzierte Objekt verhält sich jedoch wie die Basisklasse.
Bsp.:

```
void Getraenk::wasbinich(){cout << “Ein Getränk!” << endl;}
void Spirituose::wasbinich(){cout << “Schnaps!” << endl;}

Getraenk* ptr = new Spirituose();
ptr->wasbinich(); // “Ein Getränk!” (mit Identitätskriese…)
ptr->alkoholgehalt += 0.01; //ERROR
```

virtual methods

Faustregel: nicht-virtuelle Methoden gehören zur Klasse,
virtuelle Methoden gehören zum Objekt

Bsp.: (Die Methoden sind in der Klasse als virtual deklariert. Das virtual-keyword gehört NICHT zur späteren Definition!)

```
virtual void Getraenk::wasbinich(){cout << “Ein Getränk!” << endl;}
virtual void Spirituose::wasbinich(){cout << “Schnaps!” << endl;}

Getraenk* ptr = new Spirituose();
ptr->wasbinich(); // “Schnaps!”
ptr->alkoholgehalt += 0.01; //IMMERNOCH ERROR
```

Abstrakte Basis-Klassen

Idee:

- nicht instanziierbare Klasse
- modelliert abstakte Gegebenheit
- bietet Interface für Kind-Klassen

```
class SomeClass {
[...]
    virtual void abstractMethod() = 0;
[...]
}
```

realisiert durch rein virtuelle Methoden:

- Klassen mit rein virtuellen Methoden sind Abstrakte Basis-Klassen
- Kindklassen sind auch abstrakt, wenn Methode nicht überladen

Templates

Funktionen-Templates

Verallgemeinerungen von Funktionen auf beliebige Datentypen:

```
int square(int x){
    return x*x;
}

template<class T>
T square(T x){
    return x*x;
}
```

Template-Funktionen werden zur Compile-Zeit erzeugt.

Dabei treten Fehler auf, wenn Operationen auf dem Datentyp nicht definiert sind.

Funktionen-Templates (anw.)

```
template<class T>
T square(T x){
    return x*x;
}
```

```
int main(){
    double x1 = 1.7725;
    long x2 = 13;
    cout << square<double>(x1) << endl;
    cout << square(x2) << endl;
}
```

Klassen-Templates

```
template<class KEY, class VAL>
class SomeClass{
     std::vector<VAL> _values;
     std::vector<KEY>  _keys;
     VAL default;
public:
     SomeClass(VAL d):default(d){};
     VAL& operator[](KEY key);
};
```

```
VAL& SomeClass::operator[](KEY key){
     auto idx = bin_search(_keys, key);
     if(idx == _keys.end())
     {
          _keys.push_back(key);
          _values.push_back(default);
          return *(_values.end() - 1);
     }
     return *idx;
}
```

Template-Klassen können nicht impliziert angelegt werden!

std::vector

- dynamisches Array
- indizierbar über op[], wie Arrays
- passt automatisch Größe an
- einfaches Anhängen

```
#include<vector>
[...]
std::vector<int> vec;
vec.push_back(23);
vec.push_back(42);
cout << vec[0] << endl;
cout << vec.get(1) << endl;
cout << vec.size() << endl;
cout << vec.empty() << endl;
```

Iteratoren

- Vereinfachung von Pointern,
- nur auf std-Containern gültig (Verallgemeinerung von Containern)

`container.begin()` liefert Iterator auf erstes Element

`container.end()` liefert Iterator hinter letztes Element

```
for(auto i = cont.begin(); i != cont.end(); i++) [...]
```

- Pointer-Arithmetik (begrenzt) auf Iteratoren anwendbar

```
i = i + 1;    // Ein Element weiter        x = *i;  //Elem. auf das i zeigt
dist = i - j; // Abstand zw. i und j
```

std::list

- doppelt verkettete Liste
- zugriff primär über Iteratoren
- es existiert kein “random access”
  - Zugriff durch Iteration

```
#include<list>
[...]
std::list<double> l;
l.push_back(2.3);
l.push_front(4.2);
cout << l.front() << endl;
cout << *(++l.begin()) << endl;
cout << l.size() << endl;
cout << l.empty() << endl;
```

std::map

- assoziatives Array
- speichert Werte zu beliebigen Indizes

```
#include<map>
[...]
std::map<double, char> m;
m[2.3] = ‘i’;
m[4.2] = ‘u’;
cout << l[4.2] << endl;
cout << l.begin()->first << endl;
cout << l.size() << endl;
cout << l.empty() << endl;
```