Grundlagen // Modern C++ in a minute

Programmrahmen

> in Texteditor schreiben, speichern
> `hallo.cpp` übersetzen
> `hallo` ausführen

```
make hallo
./hallo
Hallo, Welt!
```

```
#include <iostream>
// ... Deklarationen,
// Funktionsdefinitionen

int main()
{
  // ... Anweisungen
  std::cout << "Hallo, Welt!\n";
}
```

Datentypen und Werte

	`auto`	aus zugewiesenem Wert erschlossen
Wahrheitswerte	`bool`	`true`, `false`
Ganzzahlen	`int`, `long`	`-1` (dez), `0b10` (bin), `0123` (oktal), `0x1ABC` (hex), `2017L` (long)
Gleitkommazahlen	`float`, `double`	`-0.5f`, `31.f`, `3.1415`, `-1.602e-19`
Einzelzeichen	`char`	`'A'`, `'7'`, `'*'`, `'\n'` (new line), `'\t'` (tab), `'\\'`, `'\''`, `'\"'`, `'\x4A'` (ASCII)
Zeichenketten	`std::string`	`"Hallo"`
Platzhalter	`void`	"leer", wo kein Typ angegeben werden kann

Konstanten

```
auto const richtigeAntwort = 42;
double const LIGHTYEAR  = 9.46e12; // m
double const MASS_EARTH = 5.98e24; // kg
```

Variablen

```
int x, y = 3;
x = y + 5;
++x; y--;
```

Operationen

arithmetisch	`x+y x-y x*y x/y x%y` (Rest)
um 1 ändern	`++x --x` (vor Auswertung) `x++ x--` (nach Auswertung)
Zuweisung Kurzschrift	`x = wert` `x += wert` d.h. `x = x + wert`
Vergleiche	`x<y x<=y x>=y x>y` `x==y` (gleich), `x!=y` (ungleich)

Logik

`a`	`b`	NICHT `!b`	UND `a && b`	ODER `a || b`
false	false	true	false	false
false	true	false	false	true
true	false		false	true
true	true		true	true

Standard-Ausgabe auf Konsole

```
std::cout << x << '\n';
```

Standard-Eingabe von Tastatur

```
std::cin >> x;
```

Funktionsdefinition

```
Ergebnistyp funktionsname(Parameterliste)
{ // ... Anweisungen
  return ergebnis;
}
```

Parameterliste, durch Komma getrennt:
`Typ parameter` oder `Typ& parameter` (Referenz)

Funktionsdeklaration und -aufruf

```
Ergebnistyp
funktionsname(Parameterliste);

ergebnis = funktionsname(argumentliste);
```

Argumentliste: Übereinstimmung in Anzahl und Typ der Werte mit Parameterliste

Steueranweisungen (Wiederholung/Entscheidung)

```
for (int i = 0; i < 5; ++i)
{
  std::cout << i << "  ";
}
```

```
for (auto n : {1,4,9,16})
{
  std::cout << n << "  ";
}
```

```
while (x > 0)   // abweisend
{  --x;
}
```

```
if (x < 5)
{  ++x;
}
else   // kann wegfallen
{      // oder etwas anderes tun
}
```

Bibliotheken (Auswahl)

<string> Zeichenketten

statt `char`-Felder

```
std::string s = "Hallo";
s += ", Welt!";
```

`s.size()`	Anzahl Zeichen
`s[index]`	Zugriff auf Einzelzeichen, `index < s.size()`
`s == s2`	gleicher Inhalt?
`s < s2`	`s` vor `s2`?
`s + s2`	Verkettung

<vector> Feldcontainer

mit beliebigem Typ und variabler Elementanzahl

```
std::vector<int> v = { 1, 2, 3 };
for(auto& e : v) e += 10;
```

`v.size()`	Anzahl Elemente
`v[index]`	Zugriff auf Element, `index < v.size()`
`v == v2`	gleicher Inhalt?
`v < v2`	`v` vor `v2`?
`v.push_back(wert)`	Wert hinten anhängen

<cmath> mathematische Funktionen

`fabs(x)`	Absolutbetrag $\lvert x\rvert$
`sqrt(x)`	$\sqrt{x}$
`pow(x,y)`	x^y
`exp(x)`	e^x
`log(x)`	$\ln x$
`sin(x)`	Winkelfunktionen
`cos(x)`	Umkehr `asin(x)`, `acos(x)`
`tan(x)`	Umkehr `atan(x)`, `atan2(y,x)`

<algorithm> Algorithmen

auf halboffenen Bereichen `[begin, end)`

```
std::sort(v.begin(), v.end());
std::sort(s.begin(), s.end());

std::for_each(v.begin(), v.end(),
    [](int e) { std::cout << e << ' '; }
  );
```

<fstream> Datei-I/O-Datenströme

```
std::ofstream output("dateiname");
if (!output) /* Fehler beim Oeffnen */;
output << "Hallo 123\n";
output.close();

std::ifstream input("dateiname");
if (!input) /* Fehler beim Oeffnen */;
input >> s >> x;
```

Offene Datei schließt automatisch am Blockende.

<sstream> Zeichenkettenströme

```
std::ostringstream outbuffer;
outbuffer << "Hallo 123";
s = outbuffer.str();

std::istringstream inbuffer(s);
while (inbuffer >> s >> x)
{ // ... verarbeiten
}
```

Stromvariable als Bedingung: „erfolgreich gelesen“

Klassen

benutzerdefinierte Datentypen

```
class Particle      // oder struct ...
{
  // ... Attribute
public:
  // ... Konstruktoren, Destruktor
  // ... Methoden deklarieren
};
```

Attribute

```
private: // kein Zugriff von aussen
  double x, y;
protected: // fuer Erben zugaenglich
  double mass;
```

Konstruktoren, Destruktor

```
  Particle (double x, double y, double m)
  : x(x), y(y), mass(m)
  {  // ... Attribute setzen
  }

  // wenn notwendig, Destruktor:
  ~Particle () { /* aufraeumen */ }
```

Methoden

```
void Particle::move(double dx, double dy)
{
  x += dx; // this->x, wenn verdeckt
}
```

außerhalb Klasse mit Klassenname qualifiziert
`const`-Methoden garantieren nur-lesenden Zugriff:

```
  double whereX() const // in Klasse
  {
    return x;
  }
```

Objekte

- Instanzen eines Klassentyps,
- erzeugt durch Konstruktoraufruf,
- ansprechbar über Variable,
- reagieren auf Methodenaufruf (Botschaft).

```
  Particle sun(0, 0, 3.32e5*MASS_EARTH);
  sun.move(10000 * LIGHTYEAR, 0);
```

Vererbung

Ableitung von Klassen aus Basisklasse(n)

```
class BlackHole : public Particle
{
  // ... Zusatzattribute
public:
  Blackhole(double x, double y, double m)
  : Particle(x, y, m)
    // Aufruf Basiskonstruktor
  { // ...
  }
  // ... abgeleitete / neue Methoden
}
```

abgeleitete Methoden (in Basis `virtual` deklariert)

```
  void move(double dx, double dy)
  { // Basis-Methode aufrufen (optional):
    Particle::move(dx, dy);
    // ...
  }
```

Zeiger und dynamische Polymorphie

Destruktor `virtual ~Particle()` in Basisklasse notwendig!

```
  Particle* ufo = new Blackhole(0, 0, 1);
  ufo->move(0, 1);
  delete ufo;
```

Besser: Verwaltung mit `std::shared_ptr<T>` oder `std::unique_ptr<T>` aus `<memory>` ohne `delete`

```
  std::shared_ptr<Particle> ufo =
    std::make_shared<Particle>(0, 0, 1);
```

Downcast

```
  if (auto hole =
        std::dynamic_pointer_cast
                  <Blackhole>(ufo))
  {  hole->collapse();
  }
```

Sonstiges

Ausnahmen behandeln

```
try
{ // Abschnitt kann Ausnahme werfen
}
catch (Particle& e) { /* behandeln */ }
```

Ausnahme werfen, wenn

- Abbruch der Aktion erforderlich ist und
- Fehler nicht vor Ort behoben werden kann.

```
if (sun.whereX() == 0) throw sun;
```

Operatoren überladen

```
bool operator<(Particle a, Particle b)
{ return a.mass < b.mass;
}
```

`private`-Zugriff für `friend` erlauben:

```
  // in class Particle:
  friend
  bool operator<(Particle a, Particle b);
```

Ausgabe

```
std::ostream&
operator<<(std::ostream& os, Particle p)
{
  return os << p.x << ' '
            << p.y << ' '
            << p.mass;
}
```

Eingabe (Überschreiben nur bei Erfolg)

```
std::istream&
operator>>(std::istream& is, Particle& p)
{ double x, y, mass;
  if (is >> x >> y >> mass)
    p = Particle(x,y, mass);
  return is;
}
```

Felder

```
int const m = 3, n = 4;  // feste Anzahl
Typ feld[n];      // feld[0]...feld[n-1]
Typ matrix[m][n]; // m Zeilen, n Spalten
```

erfordern Konstruktor `Typ()` ohne Parameter

`matrix[0][0]`	...	`matrix[0][n-1]`
...		...
`matrix[m-1][0]`	...	`matrix[m-1][n-1]`

Funktionsschablonen

```
template<typename T>
void tausche(T& a, T& b)
{
  T t(a); a = b; b = t;
}
```

```
tausche(x, y);
```

Klassenschablonen

```
template <typename T, int N>
struct Array
{
  T elements[N];
};
```

```
Array<Particle,9> solarsystem;
solarsystem.elements[0] = sun;
```

Namensräume

```
namespace Physics
{
  double const e = 1.602e-19; // As
}
namespace Math
{
  double const e = 2.71828;
}
```

```
 std::cout << Physics::e << ' '
           << Math::e << '\n';

 using Math::e;
 std::cout << Physics::e << ' '
           << e << '\n';
```