Klassen – Einleitung

Ziel von C++ ist der Support objektorientierter Programmierung (OOP). Dabei handelt es sich um ein Konzept zur Strukturierung von Programmen bei dem Programmlogik (Funktionalität) und Programmzustand (Datenstrukturen) vereinigt werden. Die Vereinigung wird Objekt genannt.

- Objekte werden im Programm erzeugt und vernichtet.
- Objekte haben zur Laufzeit des Programms definierte Zustände
- OOP erlaubt die Kommunikation zwischen Objekten

OOP wird in C++ mit Hilfe von Klassen realisiert. Sie vereinigen Datenstrukturen und Funktionen zur Veränderung der Daten.

- Anschaulich: Klassen sind Vorlagen und Konstruktionspläne aus denen zur Laufzeit Objekte erzeugt werden (Instanzen).
- Algorithmen die auf den Datenstrukturen von Klassen operieren heißen Methoden
- Klassen kapseln zusammengehörige Daten und Funktionen und unterstützen Datenzugriff nur über spezielle Funktionen. Methoden die Objekte erzeugen oder zerstören werden als Konstruktoren oder Destruktoren bezeichnet.

Klassen – Einleitung

- Klassen sind „user definierte Typen“ oder „C structs on steroids“ mit
  - Vererbung
  - Zugangskontrolle
  - Methoden
- Objekte sind Instanzen dieser user definierten Typen
- Die Klassen kapseln den Zustand und das Verhalten von „etwas“
  - Haben ein Interface
  - Stellen die Implementierung zur Verfügung, wie
    - Status und Eigenschaften
    - mögliche Wechselwirkungen
    - Erzeugung und Vernichtung
- Ein `struct` ist eine vereinfachte Klasse, Beispiel:

```
struct MyFirstClass {
  int a;
  void squareA() { a *= a; }
 };
…...
MyFirstClass myObj;
myObj.a = 2;
myObj.squareA();  // square a
```

Klassen – ein Beispiel

Ziel von C++ ist der Support von OOP. Dabei spielen Klassen eine wichtige Rolle. Sie vereinigen Datenstrukturen und Methoden zur Änderung der Daten

- Betrachte ein Beispiel: Klasse von Vierervektoren zur Beschreibung von Teilchenreaktionen

Vierervektor:

$$(E, \vec{P}) = (E, p_x, p_y, p_z)$$

```
class FourVector
{
   public:
   Definiere Funktionen, die die Elemente des Vektors
   füllen
   Definiere Funktionen, die den Impuls, Energie und
   invariante Masse zurückgeben.

   private:
   definiere Daten, die zum Vektor gehören
} vect ;
```

Klassenname, identifiziert die Klasse

Klassenmitglieder
- Datenstrukturen
- Funktionen/Methoden

Optional Objektnamen

```
int main(){
FourVector Elektron, Proton;
…..
```

Erzeuge Instanzen von FourVector
Alle Instanzen haben verschiedene Zustände.

Klassen – ein Beispiel

Ziel von C++ ist der Support von OOP. Dabei spielen Klassen eine wichtige Rolle. Sie vereinigen Datenstrukturen und Methoden zur Änderung der Daten

- Betrachte ein Beispiel: Klasse von Vierervektoren zur Beschreibung von Teilchenreaktionen

Zugriffsrechte werden explizit mit Hilfe von Schlüsselwörtern gewährt

```
class FourVector
{
   Auf alle Klassenmitglieder die hier stehen
   gibt es nur von Mitgliedern Zugriff
   public:
   Diese Klassenmitglieder können vom gesamten
   Programm benutzt werden
   private:
   Nur Mitgliedern ist der Zugriff erlaubt
};

int main(){
FourVector Elektron, Proton;
…..
```

Klassen – ein Beispiel

- Betrachte ein Beispiel: Klasse von Vierervektoren zur Beschreibung von Teilchenreaktionen

```
class FourVector{
   public:
   void setE(double e) {E = e;}
   void setP(double x, double y, double z){
             px = x; py = y ; pz = z;}
   double getE(void)const {return E;}
   double getP(void)const {
      return sqrt(px*px+py*py+pz*pz);}
   private:
   double E, px, py, pz;
} ;

int main(){
  FourVector Elektron, Proton;
  Elektron.setE(100.);
  Proton.setP(10., 20., 50.);
  Proton.getP();
  return 0 ;}
```

Methoden erlauben Zugriff auf die Datenstruktur.

FourVectorClass_0.cc

Datenstruktur ist nicht zugänglich

Zwei Instanzen von FourVector werden erzeugt.

Mit dem Punkt Operator werden die Methoden erreicht.

Klassen – ein Beispiel

- Eine Klasse wird instanziert, indem die Klassenbezeichnung und ein Name angegeben wird. Der Zugriff auf Member Funktionen und Variablen erfolgt über den Operator „ . “ Für Pointer auf Klassen wird der Operator „ → “ verwendet. Er besorgt die Dereferenzierung und den Member Zugriff.

```
class FourVector
{
   ….
}

int main() {
FourVector Kaon;
FourVector *Pion;
…..
Kaon.setE(20.);
…..
Pion->GetMomentum();
……
}
```

Memberfunktion Zugriff

Memberfunktion Zugriff bei Benutzung von Pointern

Wir versuchen in der Übungsklasse `FourVector` direkt auf die Daten zu zugreifen.

Arbeitsvorschlag:

Können Sie im Programm FourVectorClass_0.cc analog zur `struct` Synthax direkt auf die Daten Variablen zugreifen? Welche Modifikationen müssen Sie vornehmen, damit es funktioniert?

FourVectorClass_0a.cc

Klassen – ein Beispiel

- Betrachte ein Beispiel: Klasse von Vierervektoren zur Beschreibung von Teilchenreaktionen

```
class FourVector{
   public:
   void setE(double e);
   void setP(double x, double y, double z);
   double getE(void)const;
   double getP(void)const;
   private:
   double E, px, py, pz;
} ;
FourVector::setE(double e){
   E = e;
}
double FourVector::getE(void)const{
   return E;
}
```

Scope operator

Klassendefinition werden im Header File untergebracht.

Die Methoden können auch ausserhalb der Klassendefinition definiert werden. Auf die Klasse wird mit Klassenname:: Bezug genommen.

…….

FourVectorClass_I.cc

Klassen – ein Beispiel

- Konstruktor und Destruktor einer Klasse

```
class FourVector{
   public:
   FourVector(void){ E=0.;px=0.;py=0.;pz=0.}
   ~FourVector(void) {  }
   void setE(double e) {E = e;}
   void setP(double x, double y, double z){
              px = x; py = y ; pz = z;}
   double getE(void) {return E;}
   double getP(void) {return sqrt(px*px+py*py+pz*pz);}
   private:
   double E, px, py, pz;
} ;
int main(){
  FourVector Elektron, Proton;
```

Konstruktor, code wird bei der Instanzierung gerufen

Destruktor

Zwei Instanzen von FourVector werden erzeugt.

Beide sind durch den Konstruktor initialisiert.

Klassen – ein Beispiel

- Konstruktor einer Klasse

  - Kein Rückgabetyp, Argumente können void oder eine Variablenliste sein
  - Konstruktor hat den Namen der Klasse und dient der Initialisierung des Objektes

```
class FourVector{
   Public:
      FourVector(double e,double x,double y, double z);
      FourVector();
      ….
   Private:
      double E, Px, Py, Pz;
};
FourVector::FourVector(double e,double x,double y,
double z): E(e),Px(x),Py(y),Pz(z){}
FourVector::FourVector(){ E=0.; Px=0.; Py=0.; Pz=0.}

……..
FourVector Pion;
FourVector Kaon(5.,1.0,1.5,19.);
```

Initialisierung der Variablen

Synthax

Zuweisung der Variablen, um zu initialisieren

Initialisierung von array geht nur über Zuweisung !

Klassen – ein Beispiel

- Konstruktor einer Klasse

  - Kein Rückgabetyp, Argumente können void oder eine Variableliste sein
  - Konstruktor hat den Namen der Klasse und dient der Initialisierung des Objektes

```
class FourVector{
    Public:
        FourVector(double e,double x,double y, double z);
        FourVector();
        FourVector(double e);
    Private:
        double E, Px, Py, Pz;
};
```

Überladen der Konstruktoren

Initialisierung der Variablen

```
FourVector::FourVector(double e,double x,double y,
double z): E(e),Px(x),Py(y),Pz(z){}
```

Synthax

```
FourVector::FourVector(){ E=0.; Px=0.; Py=0.; Pz=0.}

FourVector::FourVector(double e): E(0)
                                  {Px=0.; Py=0.; Pz=0.}
```

Setzt E=0

Klassen – ein Beispiel

- Konstruktor einer Klasse
  - Im Header können Default Parameter angegeben werden.
  - Konstruktor kann überladen werden
  - Kopierkonstruktur wird vom System erzeugt, erstellt ein Objekt der Klasse anhand eines vorhandenen Objektes. Der Parameter ist immer eine Referenz auf ein konstantes Objekt der selben Klasse.

```
class FourVector{
    Public:
    ~FourVector(){}; Destruktor
    FourVector(double e=0.,double x,double y, double z);
    FourVector(FourVector const& FourVectorObjekt);
        ….
    Private:
        double E, px, py, pz;
};
```

Copy Konstruktor: Muss nicht explizit definiert werden

- Destruktor einer Klasse
  - Einer pro Klasse
  - Ist für Aufräumarbeiten zuständig

Wir ergänzen die Übungsklasse `FourVector` um einen Destruktor und mehrere Konstruktoren.

Arbeitsvorschlag:

Schreiben Sie ein Programm ausgehend von FourVectorClass_I.cc, dass 3 Konstruktoren zur Verfügung stellt; den Konstruktor ohne Argumente, mit einer Variablen soll nur die Energie übergeben werden, mit 3 Variablen sollen alle 3 Impulskomponenten gegeben werden. Benutzen Sie dabei Initialisierungen.

Die Konstruktoren und auch der Destruktor sollen ausgeben, wenn Sie benutzt wurden. Geben Sie die Initialisierungswerte aus.

Einem neuen `FourVector` Objekt ein existierendes zuweisen. Welcher Konstruktor wird benutzt?
Können wir eine Swap Funktion schreiben. Wird unser Konstruktor benutzt oder ein Kopierkonstruktor des Systems. Was wird benötigt?

Erzeugen Sie in einer Funktion mit `new` ein `FourVector` Objekt? Wird das Objekt zerstört?

FourVectorClass_II.cc

this Zeiger

- this Pointer einer Klasse

Jede nicht statische Methode hat eine versteckte Zeigervariable, in der die Adresse des Objektes gespeichert wird, die die Methode aufgerufen hat. `this` ist ein C++ Schlüsselwort. Die “interne“ Deklaration kann man sich so vorstellen:

```
myClass * const this = &object;
```

Der `this` Zeiger wird benutzt, wenn eine Adresse der eigenen Klasse zurückgegeben wird.
In vielen Fällen wird `this` automatisch ergänzt. In nicht eindeutigen Fällen, z.B. lokale Variablen heissen wie die Klassen Variablen, sollte `this` hinzugefügt werden.

```
class FourVector {
…….
FourVector & FourVector::compare (FourVector & v){
  If ( condition )
       return v ;    // v ist Argument
  else
       return *this ;  // aufrufendes Objekt
}
```

Operatoren

- Operatoren in Klassen

  Klassen definieren neue Typen, die in C++ genutzt werden. Typ Definitionen werden aber nicht nur in Form von Konstrukten und Zuweisungen verwendet, sondern müssen auch Operatoren (z.B + - ..) verstehen.

```
class FourVector{
   double E, px, py, pz;
   public:
   FourVector(){ E=0.;px=0.;py=0.;pz=0.}
   void setE(double e) {this->E = e;}
   …..
   double getP(void){return sqrt(Px*Px+Py*Py+Pz*Pz);}
   FourVector operator+(const FourVector& v) {
     FourVector vec;
     vec.E=this→E+v.E;  vec.Px=this→Px + v.Px;
     vec.Py=this→Py+v.Py;
     vec.Pz=this->Pz+v.Pz;
     return vec;
   }
   …….
```

Erlaubt das Summieren von zwei Instanzen von FourVector

Das Schlüsselwort `this` ist ein pointer auf das aktuell benutzte Objekt.

Wir ergänzen die Übungsklasse `FourVector` um Operatoren

Arbeitsvorschlag:

Schreiben Sie ein Programm ausgehend von FourVectorClass_II.cc , dass die Operatoren + und += für die Klasse `FourVector` zur Verfügung stellt.
Geben Sie Energie und Impuls der Summen zweier `FourVector` Objekte mit den get Methoden aus.
In ähnlicher weise kann die skalare Multiplikation implementiert werden.

Erzeugen Sie ein `FourVector` Objekt mit new und weisen Sie die Summe von 2 `FourVector` Objekten zu. Geben Sie wieder Energie und Impuls der Summe mit den `get` Methoden aus.

FourVectorClass_III.cc

Schlüsselwort static

- Methoden/Funktionen einer Klasse als static

Methoden/Funktionen die als `static` deklariert sind werden von allen Instanzen gemeinsam verwendet. Insbesondere müssen Funktionen, die ohne Instanzierung verwendet werden sollen (sich also als freie Funktionen verhalten sollen) als static deklariert werden.

```
class FourVector{
    double E, Px, Py, Pz;
    Public:
    …….
    static double ScalPro(FourVector &u,FourVector &v);
    …..
    };
double FourVector::ScalPro(FourVector &u,FourVector &v){
  double s = 0.;
  s = u.E * v.E - u.Px*v.Px+ u.Py*v.Py + u.Pz*v.Pz ;
  return s ;
}
FourVector kaon,pion;
double m = FourVector::ScalPro(kaon,pion);
```

Es gibt keine Instanzierung und daher auch keinen `this` Zeiger

Aufruf ohne Instanzierung, die Klasse wird explizit angegeben

Schlüsselwort const

- const steuert die Modifizierbarkeit von Variablen

  Mit Hilfe von `const` können Variable, Pointer und Referenzen im Programm als nicht veränderbar gesetzt werden.

  - Variable

    ```
    const int myConstant = 7;
    ```

  - Pointer

    es ist entscheidend an welcher Position `const` steht, `const` wird immer auf das nächste links stehende Element angewendet.

    `const int *myPointer;` Daten auf die der Pointer zeigt können nicht verändert werden

    `int const *myPointer;` gleiches Resultat, Pointer Variable auf konstante Integer Variable

    `int *const myPointer;` konstante Pointer Variable auf Integer Variable

    `int const *const myPointer;` konstante Pointer Variable auf konstante Integer Variable

Schlüsselwort const

- const steuert die Modifizierbarkeit von Variablen
  - Referenzen

```
int val; int * const myPointer=&val;
```

Die Adresse des Pointers ist nicht veränderbar (`val` kann nicht geändert werden).

Wenn `val` an eine Funktion übergeben wird, läßt sich der Wert von der Funktion nicht verändern!

```
void func (big_structe_type const &myStructRef);
```

übergeben einer Struktur an eine Funktion ohne zu kopieren und ohne dass die Struktur geändert werden darf.

  - const Methoden (Funktionen)

```
double myClass::myMethod(myVar) const;
```

Die Methode der Klasse (Funktion) darf die Objekte nicht ändern.

Die Methode kann aber überall verwendet werden. Referenzen oder Pointer auf Klassenmitglieder müssen als `const` zurückgegeben werden.

const Methoden und mutable

`const` Methoden können Klassenmitgliedsvariable ändern, wenn sie mit dem Schlüsselwort `mutable` versehen werden.

```
class FourVector {
  private:
  double E; double Px,Py,Pz;
  mutable double mass ;
  public:
  …….
  double getMass(void) const {
    mass = sqrt(this→E*this→E - ( this→Px*this→Px + \
                this→Py*this→Py + this→Pz*this→Pz);
    return mass; }
  …….
};
```

`mutable` untergräbt oft den Sinn einer konstanten Memberfunktion. Überdenken der Datenstruktur macht den Einsatz oft überflüssig!

Vererbung

Klassen können die Implementierung (Methoden und Variablen) von einer anderen Klasse übernehmen (Vererbung). Altes Verhalten kann umgeändert und neues hinzugefügt werden, die grundlegenden Schnittstellen bleiben jedoch gleich (Polymorphie). Das Ableiten einer neuen Klasse (Sub- oder Unterklasse) von einer bereits bestehenden (Ober- oder Basis-) Klasse wird folgendermaßen erreicht:

```
class BaseClass {        private:
 …..                     protected:
};                       public:
class SubClass: ZugriffsOperator BaseClass {
 …..
};
```

} nicht nach aussen sichtbar

public	In SubClass stehen fast alle member von BaseClass zur Verfügung
private	Alle member von BaseClass sind privat
protected	Alle member von BaseClass in `protected` sind nur in abgeleiteten Klassen sichtbar

- Zugriffskontrolle auf die BaseClass Variablen von SubClass aus

  Variable im `private` Bereich können von SubClass aus nicht erreicht werden. Lösung: verschiebe `private` BaseClass Variable in `protected`

Vererbung

Beispiel mit unserer `FourVector` Klasse, die als Basis Klasse die `ThreeVector` Klasse hat :

```
class ThreeVector {
  double Px,Py,Pz;
  public:
  ThreeVector(double x, double y, double z):
                                    Px(x),Py(y),Pz(z){}
  …….
};
class FourVector: public ThreeVector {
  double E ;
  public:
  FourVector(double e, double x, double y, double z):
                              E(e), ThreeVector(x,y,z) {}
  …….
};
```

inherit.cc

Beide Klassen verfügen über `Px`, `Py`, `Pz` und die Methoden von `ThreeVector`

Vererbung

Dies erlaubt implizit folgende Benutzung

inherit.cc

```
class ThreeVector {
   ……...
   static double SP(const ThreeVector &u,const ThreeVector &v);
   ……...
};
double ThreeVector::SP(const ThreeVector &u,const ThreeVector &v)
{return (u.Px*v.Px+u.Py*v.Py+u.Pz*v.Pz); }

class FourVector: public ThreeVector {
   ……...
   static double DotP(const FourVector &u,const FourVector &v);
   ……...
};
double FourVector::DotP(const FourVector &u,const FourVector &v)
{double s = u.E*v.E - SP(u,v);
 return s;}
```

Die Methoden von `ThreeVector::` stehen in `FourVector::` zur Verfügung!

Virtual Funktionen

- Zu verwendende Methoden werden zur Laufzeit festgelegt (late binding)

Das Schlüsselwort `virtual` erlaubt in abgeleiteten Klassen die geerbten (gleichnamigen) Mitgliedsmethoden anzupassen. Dieses Verhalten wird auch als Polymorphie bezeichnet.

- Beispiel:

```
class ThreeVector {
  ……...
  virtual double ScalarProd(const ThreeVector &u) {
    return (this→Px*u.Px+this→Py*u.Py+this→Pz*u.Pz); }
  ……...
};
class FourVector: public ThreeVector {
  ……...
  double ScalarProd(const FourVector &u) {
  s=this->E*u.E-(this→Px*u.Px+this→Py*u.Py+this→Pz*u.Pz);
  return s;}
  ……...
};
```

Die Methode `ScalarProd` hat in `ThreeVector::` und in `FourVector::` eine unterschiedliche Bedeutung

Arbeitsvorschlag:

Schreiben Sie unsere Übungsklasse `FourVector` mit Hilfe einer Klasse `ThreeVector` und implementieren Sie ein Skalarprodukt $E_1 \cdot E_2 - \vec{P}_1 \cdot \vec{P}_2$

Lesen Sie die Datei pi0_photons.txt, in der die gemessenen Vierervektoren der beiden Photonen eines pi0 Zerfalls stehen (E1 px1 py1 pz1 E2 px2 py2 pz2). Speichern Sie die Photonen als `FourVector` Objekte in einem vector array. Berechnen Sie die Invariante Masse der pi0s mit dem Skalarprodukt der Klasse `FourVector`. Bestimmen Sie die mittlere pi0 Masse und die Varianz.

readPi0.cc

Funktionen - Overloading

Funktionen können gleichzeitig mit den selben Namen definiert werden, wenn Sie eine unterschiedliche Anzahl von Parametern besitzen oder die Parameter von unterschiedlichem Typ sind.

- Overloading

overloading.cc

```
…....
class stdOutputData
{
public:
  void print(int j){cout << "Write integer: " << j << endl;}
  void print(double x){cout << "Write double: " << x << endl;}
  void print(char* c){cout << "Write character: " << c << endl;}
};
…...
stdOutputData p;
int k = 5 ;    double f = 2.3 ;     char c[256] = "Hi there" ;
// Write print to print integer ,  float or character
p.print(k);
p.print(f);
p.print(c);
…..
```