2 Grundlagen von C++ und Objektorientierung

- Ansätze für die Strukturierung von Programmen
- Modellierung mit abstrakten Datentypen
- C++ als besseres C
- Abstrakte Datentypen in C++ und LEDA

2.1 Ansätze für den Programmentwurf

Aufeinander aufbauende Konzepte:

2.1.1 Prozedural

Welche Prozeduren werden gebraucht

Finde besten Algorithmus

Fortran, Algol, Pascal, C

2.1.2 Modular

Modul = Daten + Operationen, die zusammengehören

Welche Module werden gebraucht?

Verberge Daten in Modulen

Modula-2, (C)

2.1.3 Abstrakte Datentypen

Welche Datentypen (einer oder wenige verwandte pro Modul)?

Biete vollständigen Satz von Operationen

Ada

Theoretische Sichtweise

Definition eine Datentypen über sein „Verhalten“ = Signatur + Spezifikation

Beispiele aus InfoII: Treap, Matrix

Vorteile:

- Mächtiges Spezifikationswerkzeug
- Einfachere („black-box“) Wiederverwendung
- Programm läßt sich leichter ändern ,weil verschiedene Implementierungen das Interface nicht ändern

2.1.4 Objektorientiert

Modelliere zusätzlich Verwandtschaftsbeziehungen zwischen Datentypen.

Simula, Smalltalk, C++, Objective C, Java, Eiffel, Sather, CLOS

Sprachverwirrung

- Abstrakte Datentypen sind *Klassen*
- Die konkreten Daten sind *Objekte*
- Das ganze nennt sich auch *Objektbasierte Programmierung* (Teilmenge von *Objektorientierter Programmierung*)
- Objekte sind *Instanzen* einer Klasse

2.2 Wie Findet man Klassen

(Am Beispiel des Projekts)

2.2.1 Modellierung des Problems selbst

- Sender (Position, Reichweite, Name, zugeordnete Frequenzen, ...)
- Frequenzbereich
- Kanal? Störungsbeziehung? (zu einfach?)
- Programm (SR2, ...) brauchen wir das?
- Planungsproblem? Zu umfassend?

2.2.2 Existierende Klassen

Fenster, Knöpfe, Menüs, Listen, Graphen, Punkte, Punktmengen

→ Bottom-Up-Entwurf

2.2.3 Weitere Regeln

- Finde Substantive im Pflichtenheft (Prozedural: Verben) aber ...
- Es sollte nützliche, autonome Operationen geben
- Untersuche/Verwende die (Top-Level)-Struktur existierender ähnlicher Anwendungen

2.3 C++ als besseres C

2.3.1 Kompatibilität

"Saubere" ANSI C Programme sind (meistens) äquivalente C++ Programme.

2.3.2 Kommentare

```
learned++; // wieder etwas gelernt
```

- Weniger (ver)tippen
- Mit `/* ... */` Code und Kommentar ausblenden

2.3.3 Referenzparameter

```
void increment(int *i) { (*i)++; }
void increment(int& i) { i++; }

int getSomeTinyPart(const HugeObject& huge)
{ return huge.tinyPart; }
```

- Lesbarer, weniger fehleranfällig
- Ohne weitere Codeänderung zwischen Wert- und Referenzparameter umschalten (Tuning)

Für Pascal-Kenner: Wie `VAR` Parameter

2.3.4 `#define` Considered Harmful

- Bedingt Typunsicherheit
- Macht fehleranfällige Klammerungen nötig
- Produziert Seiteneffekte

Konstanten

```
#define PI 3.14
const double pi = 3.14;
```

Inline-Funktionen

```
#define Max(x,y)\
( (x) > (y) ? (x) : (y))

inline double max(double x, double y)
{ return x > y ? x : y; }
```

`Max(i++,j++)`?

2.3.5 Überladen von Funktionsnamen (Overloading)

```
void print(int    i) { fprintf("%d", i); }
void print(double i) { fprintf("%f", i); }
```

Auch mit unterschiedlicher Zahl Parameter

```
double logarithm(double x, double base)
{ return log(x)/log(base); }

double logarithm(double x            )
{ return log(x)/log(2.0 ); }
```

Default-Parameter

```
double logarithm(double x, double base=2.0)
{ return log(x)/log(base); }
```


2.3.6 Operatoren überladen

Sei $\otimes$ ein beliebiger binärer Operator ($+$, $*$, $\ldots$, $||$, $\ldots$).

Dann ist `x` $\otimes$ `y` nur eine Abkürzung für `operator`$\otimes$`(x, y)`.

Folglich kann man diese Funktion (und damit auch den Operator) mit einer Deklaration der Form

```
ResultType
operator⊗(LeftType x, RightType y)
```

überladen.

Analog für **unäre** Operatoren.

Beispieanwendungen:

- Arithmetik mit komplexen Zahlen, Matrizen, ...
- Komfortablere Arrays (operator `[]`); Zugriffsgrenzen testen, dynamische Vergrößerung, ...

2.3.7 Das Streamkonzept für Ein- und Ausgabe

```
ostream operator<<(ostream& s, int i)
{
  fprintf(getFilePointer(s), "%d", i);
  return s;
}
ostream operator<<(ostream& s, char *str)
{
  fprintf(getFilePointer(s), "%s", str);
  return s;
}
...
```

Benutzung:

```
#include<stdio.h>
#include<iostream.h>
...
printf("(%d, %d)", x, y);
cout << "(" << x << ", " << y << ")";
```

Beispiel: Typ von x, y ändern ...

2.3.8 Speicherverwaltung

```
char *inputBuffer =
  new char[maxZeilenLaenge() + 1];
assert(inputBuffer != 0);
BigObject *temp = new BigObject;
assert(temp != 0); ...
delete temp;
delete [] inputBuffer;

char *cInputBuffer;
cInputBuffer = malloc(cInputBuffer,
  sizeof(char) * maxZeilenLaenge());
assert(cInputBuffer != 0); ...
free(cInputBuffer);
```

Warum das Rad neu erfinden?

- Etwas weniger Notation
- Weniger Rumrechnen mit Größen
- Automatischer Aufruf von →Konstruktoren und Destruktoren
- Advanced features (hier nicht)

2.4 Abstrakte Datentypen in C++

- Spezifikation meist nur als Kommentar
- Die Daten eines Objekts sind in *Members* gespeichert. (Diese Information kann veröffentlicht werden.)
- Verhalten wird durch *Member-Funktionen* definiert. (Signatur + (verborgene) Implementierung)

2.4.1 Ein Beispiel: Die Klasse `point`

In C

```
typedef struct pointS {
  double x;
  double y;
} point;

double distanceFromOrigin(point a)
{ return sqrt(a.x * a.x + a.y * a.y); }
```

In schlechtem C++ gleichbedeutend mit

```
class point {
public:
  double x;
  double y;
}
```

Autistische points

```
class Point {
// hier implizit "private:"
  double x, y;
public:
  double xCoord() {return x;}
  double yCoord() {return y;}
  double distanceFromOrigin(point a);
}
double Point::distanceFromOrigin(point a);
{ return sqrt(x*x + y*y);
}
```


Konstruktoren

Automatisch generierter Default-Konstruktor:

```
Point garbage;
```

Selbstgebauter Default-Konstruktor:

```
class Point {
...
public:
  Point() { x = 0.0;  y = 0.0; }
}
...
Point zero;
```

Oder

```
  Point(xc = 0.0, yc = 0.0)
  { x = xc;  y = yc; }
...
Point zero;
Point here(hereX, hereY); // not default
```


Schreibzugriff

Schreibzugriff

```
  void setXCoord(double xc) { x = xc; }
```


Weiteres Getier

Kopier-Konstruktor

```
  // superfluous reproduction
  Point(const Point& other)
  { x = other.xCoord();
    y = other.yCoord();
  }
```

Zuweisungsoperator

```
  // also superfluous
  Point& operator=(const Point& other)
  { x = other.xCoord();
    y = other.yCoord();
    return *this; // why that?
  }
```

Destruktor

```
  ~Point() { }
```

Das alles braucht man z.B. wenn Members auf dynamisch allozierten Speicher zeigen.

2.4.2 Diskussion

- Reale Programme machen oft einige members `public`
- Tools könnten helfen Schreibaufwand zu mindern
- Andere OO-Programmiersprachen unterscheiden nicht zwischen Members und Member-Funktionen.
  (Default-Zugriffsfunktionen)

Und wozu das alles?

Beispiel: Verschiedene Repräsentationen (Polarkoordinaten).

Bei anderem Satz von Operationen: Umsteigen auf Geometrie auf der Kugel. (Kundenwunsch für Planung von Kurzwellensendern)

2.5 Ein LEDA Beispiel

```
// file: stack.cc
#include <LEDA/stack.h>

main()
{
  stack<int> S;

  S.push(5);
  S.push(3);
  S.push(1);

  cout << "size of stack: " << S.size()
       << endl;

  cout << "elements:" << endl;

  while (!S.empty())
    cout << S.pop() << endl;

  cout << "stack is empty now" << endl;
}
```



```
# file: Makefile

XLIBS      = -lX11
LEDALIBS   = -lW -lP -lG -lL
STDLIBS    = -lm

C++ = g++

LEDA_ROOT = /usr/local/LEDA-3.6
LEDA_INCL = $(LEDA_ROOT)/incl

LEDA_LIB = $(LEDA_ROOT)
X11_LIB  = /usr/X11R6/lib

IN = -I$(LEDA_INCL)
LI = -L$(LEDA_LIB) -L$(X11_LIB)
LIBS =  $(LEDALIBS) $(XLIBS) $(STDLIBS)

CFLAGS = -O

stack: stack.o
$(C++) $(CFLAGS) -o $@ stack.o $(LIBS) $(LI)

%.o: %.cc
$(C++) $(CFLAGS) -c $(IN) $*.cc

clean:
rm -f *.o
```


- Environment-Variablen in .cshrc

```
/usr/local/LEDA-3.6/Manual/cmd in "path" aufnehmen
setenv LEDAROOT /usr/local/LEDA-3.6
setenv XDVIFONTS .:/usr/local/TeX/fonts/pk/dpi300pk/cm
```

- LEDA manual pages
  1. Latex-output

     `Lman stack`

  2. ASCII-output

     `Fman stack`

- Beispiel (stack.cc und Makefile) steht in

  `~cagoe/pp/slides/`