Spiele programmieren mit C++

Zwischen dem Kennen des OOP-Syntax und der vorteilhaften Anwenden dieser Techniken ist ein großer Schritt. Schüler, die eine Hochsprache erlernen (z.B. C oder PASCAL) neigen dazu, ihr Programm in eine Klassenhülle zu verpacken, die Programmstruktur bleibt wie gewohnt C und PASCAL.

Um dem abzuhelfen erhalten Schüler, die die C++-Syntax bereits kennen im Rahmen einer 12-stündigen Laborübung die Gelegenheit, ein kleines Projekt in reinem C++ auszuführen. Sie können dabei im Rahmen von 1-2-Schüler-Teams allmählich ihre Vorstellung vom rein prozeduralen Ablauf zum arbeitsteiligen Zusammenspiel der Objekte erweitern.

Leser der PCNEWS können diesen Übergang von Strukturen und Funktionen zu Klassen mit Methoden nachvollziehen.

Bitte erwarten Sie aber kein perfektes Bildschirm-Spiel à la Nintendo. Es geht hier nur um die prinzipielle Darstellung einer OOP-Organisationsform, die man dann selbst beliebig ausfeilen und verbessern kann. Die Bildschirmdarstellung beschränkt sich auf den Textmodus; das nachfolgende Programm Kompass zeigt ein ähnliches Problem aber im Grafikmodus.

Die Einführung erfolgt in 3 Teilen: Wiederholung der Grundbegriffe der Sprache C++ soweit für die Übung erforderlich (friends, Mehrfachvererbung, Überladen von Operatoren, Fehlerbehandlung werden übergangen), ein Glossar als Wiederholungshilfe sowei ein vollständiges Programmbeispiel. Die Schüler können je nach Wissensstand den einführenden Teil mit eigenen Vorstellungen nachvollziehen.

Franz Fiala

Klassen und Objekte

Programm ohne Variablen

Ein Text am Bildschirm entsteht als Erstlingswerk etwa so:

```
gotoxy(10,4);
printf("Erster Text");
```

In dieser Version weiß nur der Programmierer, wo sich der Text befindet. Niemand sonst wäre in der Lage, den Text zu verändern oder zu löschen.

In weiteren Schritten werden wir versuchen, die Eigenschaften des Textes zu definieren und ihm auch Fähigkeiten zuzuschreiben. Den Text mit Eigenschaften und Fähigkeiten werden wir **Objekt** bezeichnen, dessen Beschreibung eine **Klasse**.

Programm ohne Strukturen

Wir beschreiben die Kenngrößen, die einen Text am Bildschirm ausmachen mit Variablen. Mann nennt diese Variablen auch Eigenschaften (properties).

```
unsigned char s[80];
unsigned char x;
unsigned char y;
```

Es gibt einige Anweisungen, die eine Initialisierung der Eigenschaften vornehmen:

```
x=10; y=4;
strcpy(s,"Zweiter Text");
```

sowie Anweisungen, die den Text darstellen:

```
gotoxy(x,y);
printf("%s",s);
```

Strukturen statt Variablenhaufen

Da die Variablen zusammen gehören und bei einem Text immer gemeinsam auftreten, kann und soll man sie in Strukturen zusammenfassen.

```
struct TEXT             //Bauvorschrift
{                       //kein Code
  unsigned char s[80];  //entspricht Klasse
  unsigned char x;
  unsigned char y;
};
struct TEXT t;          //Durchführung
t.x=10; t.y=4;          //entspricht Objekt
strcpy(t.s,"Zweiter Text");
gotoxy(t.x,t.y);
printf("%s",t.s);
```

Mit Einführung der Struktur fällt auf, daß das Objekt, beziehungsweise seine Eigenschaften über den Strukturnamen eine Bezeichnung, hier `TEXT`, erhalten haben. Im Programm kommt er an zwei Stellen vor: einmal bei der Deklaration der Struktur (wir werden die zusammengefaßten Eigenschaften des Objekts auch Klasse nennen), dann bei der Definition des Objekts.

Während in C der Bezeichner `struct` bei der Definition zu wiederholen ist, kann diese Wiederholung in C++ entfallen.

```
struct TEXT t;  /* C */
TEXT t;         // C++
```

Schließlich muß der Text immer auch dargestellt werden, daher kann man dafür auch eine eigene Funktion vorsehen.

```
void drawTEXT(struct TEXT t)
{
  gotoxy(t.x,t.y);
  printf("%s",t.s);
```

```
}

struct TEXT makeTEXT(unsigned char x,
unsigned char y, unsigned char *text)
{
  struct TEXT t;
  t.x=x; t.y=y;
  strcpy(t.s,text);
  return t;
}

void main(void)
{
  struct TEXT t;
  t=makeTEXT(10,4,"Zweiter Text");
  drawTEXT(t);
}
```

Um die Zusammengehörigkeit der Funktion mit einem bestimmten Datentyp zu kennzeichnen, ist es zweckmäßig, diese Zusammengehörigkeit semantisch durch einen gemeinsamen Namensteil (hier `TEXT`) anzudeuten.

Klassen: Strukturen mit Funktion

Bis hierher gehen die Möglichkeiten der Sprache C. Jetzt fällt folgendes auf: zu jeder Datenstruktur, die ein Objekt beschreibt, gibt es auch einen Funktionensatz, der mit diesen Eigenschaften arbeitet. Es ist naheliegend, Eigenschaften und Funktionen unter dem Mantel der Struktur zu vereinigen. Dazu muß man aber der Datei die Endung `CPP` geben, denn das ist die erste Konvention, die nur unter C++ gilt: eine Struktur, die auch Funktionen enthält, ist eine Klasse.

```
struct TEXT
{
  unsigned char text[80];
  unsigned char x;
  unsigned char y;
  void draw()
  {
    gotoxy(x,y);
    printf(„%s“,text);
  }
  void make(unsigned char x,
            unsigned char y,
            unsigned char *text);
};

void TEXT::make(unsigned char x, unsigned
char y, unsigned char *text)
{
  TEXT::x=x; TEXT::y=y;
  strcpy(TEXT::text,text);
}

void main(void)
{
  TEXT text;
  text.make(10,4,"Zweiter Text");
  text.draw();
}
```

Gegenüber der C-Version fällt auf, daß die Zusammengehörigkeit der Funktionen mit den Variablen durch den gemeinsamen Klassennamen automatisch erfolgt.

```
makeTEXT
text.make
```

Weiters ist Parameterübergabe bei den Funktionen entbehrlich, da in einem Objekt die zu bearbeitenden Variablen globale Größen innerhalb des Objekts sind.

```
drawTEXT(f)
text.draw()
```

Objekte: konkretisierte Klassen

Es gibt zwei Möglichkeiten, eine Funktion als zur Struktur `TEXT` gehörig zu bezeichnen: 1.Definition innerhalb `TEXT` (gezeigt bei `draw()`, diese Funktion wird vom Compiler als schnelle Inline-Funktion angelegt), 2. Definition außerhalb von `TEXT` und nur ein Prototyp innerhalb von `TEXT` (gezeigt bei `make()`). Ist die Funktionsdefinition außerhalb, braucht man ein Hilfsmittel, um auszudrücken, daß diese Zugehörigkeit besteht. Das ist der Operator `::`,

[illegible]

[illegible]	[illegible]
[illegible]	[illegible]
Variable	Objekt
Eigenschaft	Variable
Elementfunktion (Methode)	Funktion

Kapselung (data hiding)

Solange wir uns nur mit einer Klasse und einem Objekt beschäftigen, ist alles ganz einfach. Programmieren im Team oder Linken mit fremden oder früher programmierten Programmteilen enthält ganz besondere Probleme, die zum Beispiel in der unbeabsichtigten Beeinflussung von Variablen durch andere Programme bestehen. (Stichwort: globale Variablen oder einfach unkontrollierbarer Zugriff durch globale Funktionen oder Datenstrukturen). In unserem Textbeispiel ist diese Beeinflussung ebenfalls möglich, z.B. durch folgenden Kode:

```
TEXT text(3,4,"text");
text.x=5; //??
```

Jeder Programmteil kann auf die Eigenschaften des Objekts zugreifen und diese verändern. Man hat deshalb weitere Bezeichner eingeführt, die diesen Zugriff von außen unmöglich machen beziehungsweise stark einschränken. Wir schreiben unsere Klassendefinition jetzt so:

```
// Klasse als struct
struct TEXT
{
private:
  unsigned char s[80];
  unsigned char x;
  unsigned char y;
public:
  void draw();
  TEXT(unsigned char x
       unsigned char y,
       char *text);
};
// Klasse als class
struct TEXT
{
  unsigned char s[80];
  unsigned char x;
  unsigned char y;
public:
  void draw();
  TEXT(unsigned char x
       unsigned char y,
       char *text);
};
```

Die beiden Bereichsbezeichner `private` und `public` erlauben die Unterscheidung zwischen öffentlichen, benutzbaren Teilen (`public`) und jenen Teilen, die man von außen nur über die Funktionen erreichen kann (`private`). Diese Eigenschaft, Daten abschirmen zu können, nennt man Kapselung oder **data hiding**.

Kapselung ist eine gewünschte Eigenschaft, die man durch das neue Schlüsselwort `class` als default bekommt. Eine Struktur `struct` hat als Anfangswert das `public`-Verhalten (kompatibel zu C), eine Klasse `class`, das `private`-Verhalten.

Konstruktor

Auffällig ist, daß man bei praktisch allen denkbaren Objekten eine Funktion benötigen wird, die die Variablen des Objekts, also seine Eigenschaften initialisiert. Diese Funktion haben wir im Beispiel mit `make()` bezeichnet. Da jedes Objekt ein solche Funktion brauchen kann, wurde dafür ein besonderer Name reserviert: der Konstruktor. Er hat auch einen unverwechselbaren Namen, nämlich denselben Namen wie auch das Objekt selbst, hier

TEXT. (Strukturen in C leiden unter dem Mangel, daß man sie nicht mit einem Anfangswert ausstatten kann, sondern eigene Anweisungen anwenden muß.)

Durch neue Eigenschaften (z.B. Farbe) muß auch der Konstruktor neue Aufgaben übernehmen. Ein besonderes Feature ist es, daß man optionalen Parametern einen Anfangswert zuweisen kann, und diese optionalen Parameter nur bei Bedarf initialisieren muß. Im Prototyp schreibt man:

```
TEXT(unsigned char x, unsigned char y, char
*text, unsigned char farbe=15, unsigned char
unterstrichen=0);
```

Aurufen kann man TEXT sowohl als

```
TEXT t(15,4,"text");    // als auch als
TEXT t(15,4,"text",13); //oder
TEXT t(15,4,"text",13,1);
```

Das Vorhandensein eines Konstruktor wirft aber auch ein neues Licht auf die Definition von Variablen. Während in C die Definition von Variablen am Beginn eines Blocks stehen muß und ausschließlich dazu dient, Speicherplatz festzulegen, bedeutet in C++ die Definition eines Objekts den Aufruf einer Funktion, des Konstruktors. Daher kann in C++ eine Variable oder ein Objekt überall im Kode definiert werden. Die eingebauten Variablen char, int, float... bekommen daher den sogenannten Default-Konstruktor mit auf den Weg, ebenso wie alle Klassen, die über keinen expliziten Konstruktor verfügen.

Destruktor

Die Klasse TEXT zeichnet den Text am Bildschirm. Wenn sie ihre Arbeit beendet, bleibt der Text stehen. Im allgemeinen hilft sich der Programmierer mit einem generellen clrscr(), um Textreste loszuwerden, doch bietet C++ eine elegantere Möglichkeit, den Destruktor. Er hat denselben Namen wie die Klasse, nur hat er eine vorangestellte Tilde ‚~'. Im Falle der Textklasse kann er die Aufgabe übernehmen, den Text vom Bildschirm zu löschen.

```
~TEXT() { for (int i=0; i<strlen(s); i++)
{
  gotoxy(x+i,y);
  printf („ “);}
  }
```

Der Destruktor der Klasse TEXT entfernt den Text nach Beendigung automatisch vom Bildschirm.

Getestet wird das Verhalten am besten durch ein Objekt in einer eigenen Funktion, die im Hauptprogramm gerufen wird. Ein getch() vor Beendigung des Programms überprüft die Wirkung des Destruktors.

Aufgaben

Man kann zu neuen TEXT-Klasse weitere Eigenschaften hinzufügen, die TEXT-Objekte brauchbarer machen. Z.B. Textfarbe und andere Textattribute, wie blinken, unterstreichen oder versteckt. Auch neue Methoden wie löschen, verschieben können zur Klasse TEXT hinzugefügt werden. Weiters ist es nützlich, sich vor dem Beschreiben des Hintergrunds dessen aktuellen Zustand zu merken und nach Beendigung der Objektlebensdauer diesen wiederherzustellen.

Es ist weiter keine Hexerei, eine ähnliche Klasse ZEICH zu formulieren, die im Gegensatz zu TEXT nur aus einem einzigen Zeichen besteht.

Vererbung (inheritace)

Allen Programmelementen, die bisher in Variablen und Funktionen ausgedrückt worden sind, können Klassen zugeordnet werden. Wenn man das probeweise auf bekannte Programme anwendet, stellt man fest, daß sich Dinge wiederholen, daß zwischen Klassen "verwandschaftliche" Verhältnisse bestehen. Wenn man eine Klasse benötigt, die sich von einer bestehenden Klasse nur durch zusätzliche Merkmale unterscheidet, kann man alle bestehenden Merkmale von dieser Klasse "erben".

Angenommen, man benötigt eingerahmte Texte. Die Texte selbst haben ebenso wie die Klasse TEXT einen Ursprung und einen Inhalt. Darüberhinaus haben sie aber auch einen Rahmen. Die draw()-Funktion muß daher mehr tun als nur den Text auf den Bildschirm bringen.

```
class TEXTR : public TEXT
{
  unsigned int laenge;
  unsigned char rahmenart;
public:
  void draw();
  TEXTR(unsigned char x, unsigned char y,
char *text, unsigned int rahmen);
};
```

```
TEXTR::TEXTR(unsigned char x, unsigned char
y, char *text, unsigned int rahmen): TEXT(x,
y, text)
{
  rahmenart = rahmen;
  laenge = strlen(text);
}

void TEXTR::draw()
{
  TEXT::draw();
  If (rahmen==1)
  for (int i=0; i<laenge; i++)
  {
    gotoxy(x+i,y-1); printf("%c",'-');
    gotoxy(x+i,y+1); printf("%c",'-');
  }
}
```

Aufgerufen wird die Klasse als Ojekt text1:

```
TEXTR text1(12,6,"Dritter Text",1);
text1.draw();
```

Es wird eine neue Klasse TEXTR deklariert, die von der Klasse TEXT abgeleitet ist. Die neue Klasse hat zusätzliche Eigenschaften. Die Methode draw() wiederholt sich, sie muß zusätzlich zu draw() von TEXT auch noch den Rahmen zeichnen.

Der Konstruktor der neuen Klasse muß auch den Konstruktor der Mutter-Klasse initialisieren, was bereits in der ersten Zeile geschieht. draw() muß nur mehr den Rahmen zeichnen, nicht aber den Text selbst, das übernimmt bereits die Funktion TEXT::draw().

Ganz ohne Änderung der Klasse TEXT geht es aber nicht, denn anfangs haben wir gesagt, niemand könne auf die Eigenschaften der Klasse TEXT zugreifen. Im obigen Beispiel geschieht es aber doch im Rahmen der draw()-Funktion. Dort greift TEXTR::draw() auf x und y zu, obwohl diese in TEXT definiert sind. Abgeleitete Klassen müssen das sehr oft tun, daher verleiht man jetzt den Variablen, auf die abgeleitete Klassen einen Zugriff haben müssen, das Attribut protected. Zugriffe fremder Funktionen sind nach wie vor unmöglich, abgeleitete Klassen haben aber Zugriffsrechte.

Aufgaben

Ausgehend von einer der fertigen Klassen sind spezialisierte Klassen abzuleiten, z.B. solche, die einen feststehenden Text ausgeben können und solche, die man bewegen kann und solche, die den Text selbsttätig auf einen vorhandenen Platz ausdehnen.

Polymorphie

Dynamische Objekte

Bisher haben wir lediglich automatische Objekte der Klasse `TEXT` gebildet. Das sind solche, die bei der Definition am Beginn einer Funktion am Stack angelegt werden. Nach Beendigung der Funktion werden alle Variablen der Funktion, also auch unser Objekt wieder gelöscht. Diese Methode zum Anlegen von Variablen eignet sich gut für Versuche, doch weniger für Programme, die diese Objekte erst zur Laufzeit, etwa durch Benutzerinteraktion erzeugen. In diesem Fall ist es beim Verfassen des Programms gar nicht bekannt, wie viele Objekte zur Laufzeit benötigt werden. Hier sind dynamische Variablen am Heap zu verwenden. In C++ sind zwei neue Operatoren definiert, die das Arbeiten mit dynamischen Variablen erheblich erleichtern: new und delete. Mit new legt man eine Variable am Heap an und erhält einen Pointer auf diese Variable zurück. Mit delete löscht man die Variable wieder.

Um diese neuen Fähigkeiten zu testen, müssen wir an den Klassen `TEXT` und `TEXTR` nichts ändern. Lediglich die Art der Generierung im Hauptprogramm ist verschieden. Die Lebensdauer des Objekts bestimmt der Programmierer selbst.

```
void main(void)
{
  TEXT *text = new TEXT(10,4,"Zweiter
Text");
  text-draw();
  TEXTR *text1 = new TEXTR(12,6,"Dritter
Text",1);
  text1-draw();
  getch();
  delete text;
  delete text1;
  getch();
}
```

Jetzt repräsentieren die Pointer `text` und `text1` die Texte. Die Lebensdauer kann jene der Funktion übersteigen.

Der Unterschied zu den C-Funktionen `malloc()` und `free()` ist der, daß `new` einen typenrichtigen Pointer zurückliefert.

Erzeugen wir mehrere `TEXT`-Objekte, entsteht ein Verwaltungproblem, wenn diese Objekte gleichartig bearbeitet werden sollen, z.B. wenn man sie alle verschieben will. Natürlich kann man sie alle so verschieben:

```
text1-move(1,1);
text2-move(1,1);
...
```

Besser wäre es aber, wenn man alle Objekte in einem Pointer-Array zusammenfassen würde. Dem Array kann man dann mit einer for-Schleife zu Leibe rücken. Unbenutzte Positionen im Array enthalten `NULL`-Pointer. Das Array wieder kann man zum Bestandteil einer Klasse machen, die ein Behälter für Textobjekte wird. Zum Beispiel kann der Bildschirm dieses Container-Objekt sein.

Ein Problem haben wir aber: Es gibt jetzt die Klassen `ZEICH`, `TEXT` und `TEXTR` und es können Objekte aller Klassen generiert werden, doch haben die Pointer auf diese Objekte verschiedenen Typ und sind daher nicht unmittelbar austauschbar.

In C++ hat man dieses Problem semantisch so gelöst: Wenn es gelingt, eine gemeinsame Basisklasse zu bilden, die diesen Objekten als gemeinsame Elternklasse dient, kann man mit Zeigern dieser Basisklasse mit allen Objekten arbeiten, obwohl sie verschiedenartige Form haben (Polymorphie).

Allen genannten Klassen gemeinsam ist ein Ort am Bildschirm, nennen wir daher diese neue Klasse `ORT` und leiten wir alle folgenden Klassen von dieser Basisklasse `ORT` ab. Sonst brauchen wir diese Klasse `ORT` nicht weiter. Dennoch übernehmen wir alle Funktionen, die wir für die anderen Klassen bereits formuliert haben in diese neue Klasse `ORT`.

Durch diese weitere Abstraktion wandern die Variablen `x` und `y` in die neue Mutterklasse `ORT`. Dadurch vereinfachen sich die vorher entworfenen Klassen `TEXT`; `TEXTR` und `ZEICH` noch weiter. Schauen wir uns die Mutterklasse `ORT` noch einmal an:

```
class ORT
{
protected:
  unsigned char x;
  unsigned char y;
public:
  virtual void draw() = 0;
  ORT(unsigned char x, unsigned char y)
  {
    ORT::x=x; ORT::y=y;
  }
  virtual ~ORT() {};
};
```

Das Schlüsselwort `protected` bewirkt, daß abgeleitete Klassen auf die Variablen zugreifen können.

`draw()` ist eine virtuelle Funktion. Das bedeutet, daß ein Basisklassenpointer, der auf abgeleitete Objekte zeigt, jeweils jene `draw()`-Funktion aufruft, die zu diesem Objekt gehört. Diese Eigenschaft nennt man Polymorphie.

Auch der Destruktor ist eine virtuelle Funktion, da bei jeder Klasse ein anderer Destruktor gerufen werden muß.

Schauen wir uns polymorphes Verhalten im Hauptprogram an:

```
void main(void)
{
  ORT *text =
  new TEXT(10,4,"Zweiter Text");
  text-draw();
  ORT *textr =
  new TEXTR(12,6,"Dritter Text",1);
  textr-draw();
  ORT *zeich = new ZEICH(1,1,'Z');
  zeich-draw();
  getch();
  delete text;
  delete textr;
  delete zeich;
  getch();
}
```

`text`, `textr` und `zeichr` sind Pointer der Basisklasse `ORT`, von der gar kein Objekt gebildet wird. Jeder Pointer zeigt auf einen anderen Typ (`TEXT`, `TEXTR` und `ZEICHR`). Jeder der drei Aufrufe der Funktion `draw()` verwendet eine andere Funktion.

Aufgaben

- In allen Klassen wird die langsame Funktion `printf()` verwendet. `printf()` ist duch das vordefinierte Objekt `cout` zu ersetzen.
- Es wäre vorteilhaft, würden sich die Funktionen auch den jeweiligen Hintergrund merken und beim Löschen wiederherstellen.
- Man sollte in der Lage sein, den Textinhalt bereits gezeichneter Objekte zu verändern.
- Mehrzeilige Texte (Container-Klasse mit mehreren Zeilen mit oder ohne Zeilenumbruch)
- Stringklasse (ohne Bildschirmdarstellung).
- Entwurf einer Klasse für grafische Objekte (`KREIS`, `RECHTECK`, `LINIE`... mit gemeinsamer Basisklasse `OBJEKT`)
- Verwaltung überlappender Textbereiche (Fenster) in einem Container-Objekt.

Anwendungen

- Beschriftungen von Bildschirmmasken.
- Einfache Spiele
- Editoren