ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Informatik I - D-ITET

Wintersemester 2003/2004

Prof. Markus Gross, Bruno Heidelberger, Richard Keiser, Nicky Kern, Edouard Lamboray, Christoph Niederberger, Tim Weyrich, Felix Eberhard, Manuel Graber, Nathalie Kellenberger, Marcel Kessler, Lior Wehrli

Uebung 7 - Pointers II

Ausgabe: 15. Dezember 2003
Abgabe: 5. Januar 2004
Autor: Richard Keiser

1. Pointer-Arithmetik — 3 Punkte

Gegeben seien die folgenden Variablen und Anweisungen:

```
float *aPtr;
int *zPtr;
int number, i;
float a[3] = {10.0, 20.0, 30.0};
int z[5] = {1, 2, 3, 4, 5};
zPtr = z;
```

Finde den Fehler in jedem der folgenden Programmsegmente und gib die Korrektur an:

```
a)    ++aPtr;
b)    // use pointer to get first value of array
      number = zPtr;
c)    // assign array element 2 (the value 3) to number
      number = *zPtr[2];
d)    // print entire array z
      for (i=0; i<=5; i++)
            cout << zPtr[i] << endl;
e)    ++z;
f)    char s[10];
      cout << strncpy(s, "hello", 5) << endl;
g)    char s[12];
      strcpy(s, "Welcome Home");
```

Hinweis: `strncpy(s1, s2, 3)` kopiert maximal 3 Zeichen von `s2` nach `s1`.

2. Fifo Queue

Einführung

In dieser Uebung soll eine First In First Out (FIFO) queue, eine der grundlegenden Datenstrukturen, implementiert werden. Im Allgemeinen wird eine solche Datenstruktur nicht durch die Art und Weise charakterisiert, in der die Daten im Speicher des Computers abgelegt sind, sondern durch die Funktionen, die sie anbietet: Elemente können hinten an die Queue angefügt werden (*enqueue*), vorne von der Queue entfernt werden (*dequeue*), das erste Element der Queue kann ausgelesen werden (*front*) und es kann geprüft werden, ob die Queue überhaupt Daten enthält (*isEmpty)*.

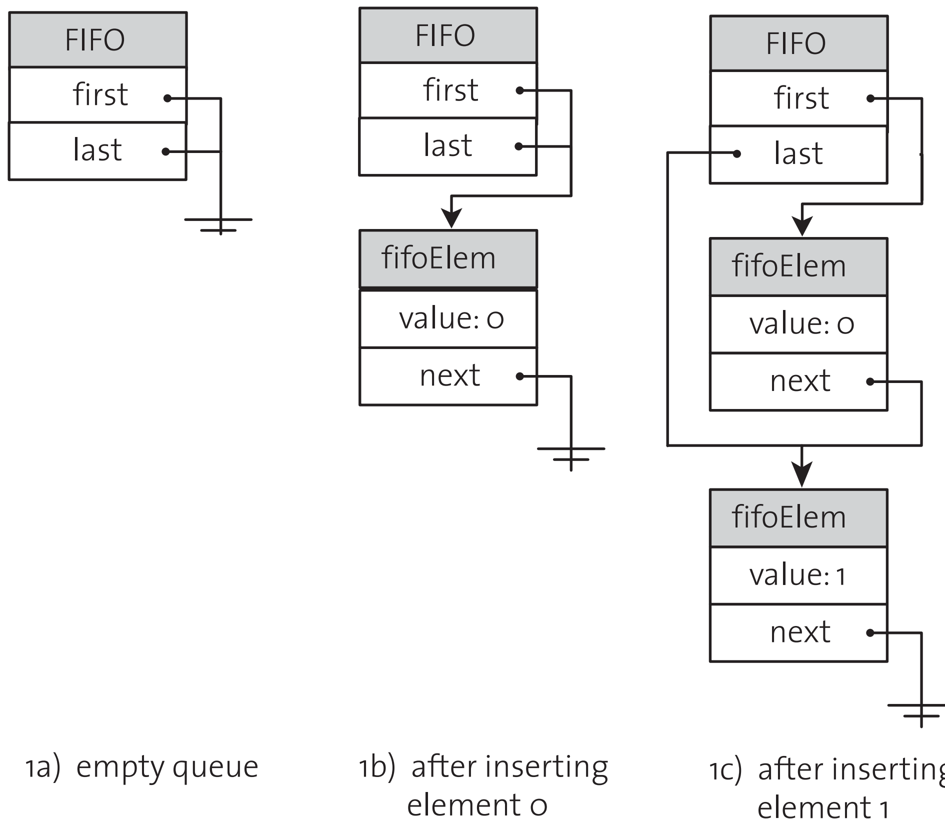
Datenstruktur

In dieser Uebung wollen wir eine Fifo Queue mit Hilfe von Pointern und dynamischem Speicher implementieren, d.h., jedes Element, das in der Queue gespeichert wird, wird in einem separaten Struct abgelegt. Neben dem eigentlichen Element, in unserem Beispiel ein `int`, enthält der Struct auch noch einen Zeiger auf das nächste Element der Queue. Abbildung 1a zeigt eine leere Queue. Abbildung 1b zeigt die Queue nach einem Aufruf von `enqueue(0)`. Abbildung 1c zeigt die Queue nach einem Aufruf von `enqueue(1)`. Das Erdungssymbol bezeichnet den NULL-Zeiger.

Datentypen (2 Punkte)

Definiere einen Struct, der erlaubt Ganzzahlen in einer FIFO Queue zu speichern. Neben dem Wert muss der Struct einen Zeiger `next` auf das nächste Element der Queue enthalten.

Definiere einen Struct, welcher die FIFO Queue als Ganzes repräsentiert. Er enthält ein einen Zeiger `first` auf das erste Element in der Queue und einen Zeiger `last` auf das letzte Element in der Queue.



1a) empty queue 1b) after inserting element 0 1c) after inserting element 1

Initialisierung (2 Punkte)

Schreibe eine Funktion, welche eine FIFO Queue als Parameter übergeben bekommt und auf "leer" initialisiert. Die Bedingung für eine leere Queue ist, dass der `first`-Zeiger NULL ist. *Achtung*: Wie muss der Parameter der Funktion übergeben werden?

Schreibe eine Funktion, welche ein neues Element für die FIFO Queue zur Verfügung

stellt. Sie bekommt den Wert der im Element gespeichert werden soll als Parameter übergeben. Zuerst muss sie mit `new` Speicher für das Element allozieren, dann den `next`-Pointer auf `NULL` initialisieren und schliesslich den Wert des Parameters zuweisen. Als Rückgabewert liefert sie das neue Element. *Achtung*: Wie muss das neue Element zurückgegeben werden?

Queue-Funktionen **(2 Punkte)**

isEmpty

Schreibe eine Funktion, die testet, ob eine Queue leer ist. Die Queue soll der Funktion als Parameter übergeben werden. Der Rückgabewert muss `true` sein, wenn die Queue leer ist, sonst `false`.

enqueue

Schreibe eine Funktion, die ein neues Element in eine Queue einfügt. Die Queue und der Integer-Wert soll als Parameter übergeben werden. Der Integer-Wert muss als neues Element an letzter Stelle der Queue eingefügt werden. Verwende dafür die Initialisierungsfunktion, die du vorhin geschrieben hast. Vergiss nicht beim Einfügen eine Fallunterscheidung zu machen, ob die Queue bereits Elemente hat oder nicht.

dequeue

Schreibe eine Funktion, die das erste Element aus einer Queue entfernt. Wieder soll die Queue als Parameter übergeben werden. Vergiss nicht, den allozierten Speicher für das Element wieder freizugeben. *Achtung*: Was passiert, wenn die Queue leer ist?

front

Schreibe eine Funktion, die den Wert des ersten Elementes einer Queue zurückgibt (ohne das Element zu entfernen!). Die Queue soll als Parameter übergeben werden.

Beispielanwendung **(1 Punkt)**

Schreibe eine kleine Beispielanwendung, die die Zahlen 0 bis 9 der Reihe nach in die FIFO Queue einfügt und anschliessend jede zweite Zahl, angefangen mit 0, aus der Queue ausgibt.

3. Makefiles (fakultativ, aber wärmstens empfohlen ;-)

In dem Masse, in dem deine Programme umfangreicher werden und aus mehreren Modulen bestehen, wird es immer mühsamer, die Programme durch die direkte Eingabe der `g++`-Aufrufe zu übersetzen. Ausserdem wird es schwierig, den Ueberblick zu behalten, was alles neu übersetzt werden muss, wenn eine oder mehrere Quelldateien geändert wurden. In unserem Beispiel binden die zwei Quelldateien `karl.cpp` und `heinz.cpp` jeweils beide die Headerdatei `karl.h` mit Hilfe der `#include`-Anweisung ein. Wenn nun eine Aenderung an `heinz.cpp` gemacht wird, muss auch nur diese .cpp-Datei neu kompiliert werden. Wurde hingehen `karl.h` geändert, müssen sowohl `karl.cpp` als auch `heinz.cpp` neu übersetzt werden, da beide `karl.h` einbinden und von dessen Aenderung betroffen sein könnten.
Solche Ueberlegungen im Einzelfall anzustellen und die benötigten Dateien „von Hand“ zu übersetzen, ist zum Glück nicht notwendig, denn genau für diesen Zweck gibt es das Programm `make`[1].
`make` ist ganz allgemein dafür gedacht, Abhängigkeiten zwischen Dateien zu analysieren und bei Aenderung einer Datei andere Dateien neu zu erstellen, die von dieser ab-

hängen. Will man C++-Programme übersetzen, hängen die .o-Dateien jeweils von der zugehörigen .cpp-Datei und allen eingebundenen .h-Dateien ab.
Damit `make` die Abhängigkeiten zwischen den Dateien kennt und weiss, wie abhängige Dateien neu erstellt werden, muss man eine Datei namens `Makefile` zur Verfügung stellen. Wird make aufgerufen, sucht es im aktuellen Verzeichnis nach dieser Datei und versucht entsprechend dem Makefile, alle benötigten Dateien auf den neuesten Stand zu bringen.
Das Format eines Makefiles ist sehr einfach. Im Makefile stehen *Regeln* und *Variablendefinitionen*. Eine Regel hat immer die folgende Form[1]:

```
zielDatei: dateien von denen die Datei abhaengt
           Shell-Kommando 1
           Shell-Kommando 2
                .
                .
                .
```

Wird `zielDatei` benötigt, so versucht `make` die Datei neu zu erstellen wenn sie a) noch nicht vorhanden ist oder b) eine der Dateien hinter dem Doppelpunkt neuer als `zielDatei` ist. Ob eine Datei neuer ist, wird anhand des Dateidatums ermittelt.
`zielDatei` wird neu erstellt, indem die Shell-Kommandos in den darauffolgenden Zeilen ausgeführt werden.
Variablendefinitionen weisen make-internen Variablen Texte zu. Sie bestehen aus einem Variablennamen, einem Gleichheitszeichen und dem Text, der der Variablen zugewiesen wird. Eine Definition kann zum Beispiel so aussehen:

```
MYEDITOR = xemacs
```

Den Inhalt einer Variablen kann man dann an beliebigen Stellen des Makefiles wieder (als Text) einfügen, indem man sie in `$()` einschliesst. Im Beispiel steht `$(MYEDITOR)` für den Text `xemacs`.
Schliesslich kann ein Makefile auch Kommentare enthalten. Kommentare beginnen mit einem '#' und reichen bis an das Ende einer Zeile.

Beispiel: Zum oben geschilderten Szenario könnte folgendes Makefile gehören:

```
# Makefile for the karl-heinz example (1)
#
#   builds a programm `karlheinz' from the source files
#   karl.cpp, karl.h, and heinz.cpp
#

karlheinz:      karl.o heinz.o
                g++ -o karlheinz karl.o heinz.o

karl.o:         karl.cpp karl.h
                g++ -o karl.o -c karl.cpp

heinz.o:        heinz.cpp karl.h
                g++ -o heinz.o -c heinz.cpp
```

Wenn `make` aufgerufen wird, versucht es, die Zieldatei der obersten Regel, in diesem Fall `karlheinz`, auf den neuesten Stand zu bringen. Dazu überprüft es zuerst, ob die Date-

1. Manche Makefiles verwenden Erweiterungen einer bestimmten `make`-Version. So benötigt z.B. das Makefile des Miniprojekts GNU-make, das auf den Pool-Suns als `gmake` aufgerufen werden kann.
1. Bei vielen `make`-Versionen ist es unbedingt notwendig, dass `zielDatei` in der ersten Spalte der Zeile anfängt und die Zeilen mit den Shell-Kommandos mit einem Tabulator beginnen.

ien rechts vom Doppelpunkt aktualisiert sind, wozu es wiederum weitere Regeln im Makefile untersucht. Möchte man nicht die Zieldatei der obersten Regel bauen, kann man make die zu erstellende Datei als Argument geben. Zum Beispiel 'make karl.o'.
Üblicherweise nutzt man Definitionen, um das Makefile flexibler zu halten. So verwendet man die Variable CC, um den Namen des verwendeten C-Compilers festzulegen. Die Variable CFLAGS nimmt weitere Optionen auf, die dem Compiler bei der Uebersetzung mitgegeben werden. Mit der Verwendung von Definitionen sieht das obenstehende Makefile so aus:

```
# Makefile for the karl-heinz example (2)
#
#   builds a programm `karlheinz' from the source files
#   karl.cpp, karl.h, and heinz.cpp
#
CC = g++
CFLAGS = -g -Wall -pedantic
OBJS = karl.o heinz.o

karlheinz:       $(OBJS)
                 $(CC) $(CFLAGS) -o karlheinz $(OBJS)

karl.o:          karl.cpp karl.h
                 $(CC) $(CFLAGS) -o karl.o -c karl.cpp

heinz.o:         heinz.cpp karl.h
                 $(CC) $(CFLAGS) -o heinz.o -c heinz.cpp

# say `make clean' to tidy up:
#
clean:
                 rm -f karlheinz $(OBJS)
```

Im Beispiel werden dem Compiler übrigens noch drei Optionen übergeben: -g benötigst du, wenn du dein Programm debuggen willst. -Wall aktiviert alle möglichen Warnmeldungen des Compilers. Nicht selten machen einen diese Warnungen auf Programmierfehler aufmerksam. Die Option -pedantic schliesslich lässt g++ alles als Fehler werten, was gegen den ANSI-C++-Standard verstösst.

Wir wünschen Euch noch ein

Frohes Fest

und einen

Guten Rutsch ins Neue Jahr!