Frivillig laboration LEU480 HT 2011

Prioritetskö

Avsikt

I de objektorienterade programspråken Java och C++ spelar klasserna en central roll för att bygga upp program som består av väl avgränsade moduler med tydliga gränssnitt mot övriga delar av programmet. Det finns i dessa språk också möjlighet att använda sig av generiska enheter (templates) för att bygga generella programmoduler. Detta är speciellt användbart när man konstruerar s.k. containerklasser (behållare som innehåller andra objekt). I programspråket C finns det varken klasser eller generiska enheter, men trots detta kan man, om man programmerar på ett disciplinerat sätt, konstruera hyggligt återanvändbara programmoduler.

I denna laboration får du lära dig hur en sådan programmodul kan konstrueras i C. Det gäller att implementera en prioritetskö. Prioritetsköer används internt i realtidsoperativsystem för att hålla reda på de olika processer som står i tur att exekveras. Du kommer dessutom att få en god träning i att använda pekare, dynamiskt allokerade objekt och länkade datastrukturer.

Programmeringsmiljö

Även i denna laboration går det att använda en vanlig enkel texteditor för att redigera programmen. Kommandon för att kompilera och köra programmen skall i så fall ges i ett vanligt kommandofönster (*Kommandotolken* i Windows). Men programmeringsarbetet underlättas om du använder en lite mer avancerad programutvecklingsmiljö som t.ex. *Visual Studio* eller *Dev-C++*.

Uppgiften

Uppgiften är att konstruera en programmodul som kan användas för att skapa prioritetsköer. En modul i C skall som bekant alltid byggas upp med hjälp av två filer, en .h-fil som innehåller deklarationer av funktioner och typer och en .c-fil som innehåller funktionsdefinitionerna, dvs. implementeringen av funktionerna. I denna uppgift skall modulen bestå av de två filerna queue.h och queue.c. Filen queue.h är redan färdigskriven och finns på kursens webbsida. På kursens webbsida finns också ett färdigskrivet testprogram i filen qtest.c. Detta skall du använda för att provköra din kömodul. *Din uppgift är att skriva filen queue.c*. I denna fil skall du implementera alla de funktioner som deklareras i filen queue.h. Obs! Du får inte ändra något i filen queue.h. Du måste också i filen queue.c använda dig av de typdefinitioner som ges i avsnittet **Implementering** i detta PM. De skall användas precis som de är.

Gränssnittet

I filen queue.h specificeras kömodulens gränssnitt mot andra programdelar:

```
// Filen queue.h
 // Makrot DATA skall ange typen för sådana data som skall läggas i kön.
 // Det bör ha definierats av användaren, annars sätts det här till void.

 #ifndef DATA
 #define DATA void
 #endif

 #ifndef QUEUE_H
 #define QUEUE_H

 struct qstruct;                          // anger att qstruct och qiteratorstruct
 struct qiteratorstruct;                  // definieras på annat ställe

 typedef struct qstruct *Queue;           // typerna Queue och Iterator
 typedef struct qiteratorstruct *Iterator;  // skall utnyttjas av användaren

 Queue new_queue();                       // allokerar minnesutrymme för en ny kö
 void delete_queue(Queue q);              // tar bort kön helt och hållet
 void clear(Queue q);                     // tar bort köelementen men behåller kön
 int  size(Queue q);                      // ger köns aktuella längd
 void add(Queue q, int priority, DATA *d);    // lägger in d på rätt plats
 DATA *get_first(Queue q);                // avläser första dataelementet
 void remove_first(Queue q);              // tar bort det första elementet

 Iterator new_iterator(Queue q);          // allokerar utrymme för en ny iterator
 void delete_iterator(Iterator it);       // tar bort iteratorn
 void go_to_first(Iterator it);           // går till köns första element
 void go_to_last(Iterator it);            // går till köns sista element
 void go_to_next(Iterator it);            // går till till nästa element
 void go_to_previous(Iterator it);        // går till föregående element
 DATA *get_current(Iterator it);          // ger pekare till aktuellt dataelementet
 int is_valid(Iterator it);               // returnerar 0 om iteratorn inte är giltlig,
                                          // dvs inte refererar något element, annars 1.
 void change_current(Iterator it, DATA *d); // ändrar aktuellt dataelementet
 void remove_current(Iterator it);        // tar bort aktuellt dataelement
 void find(Iterator it, DATA *d);         // söker d, iteratorn kommer att
                                          // referera till *d eller vara ogiltlig.
 #endif
```

Typerna `Queue` och `Iterator` definieras som pekare till typerna **`struct`** `qstruct` resp. **`struct`** `qiteratorstruct`. De fullständiga definitionerna av dessa två typer göms inne i filen `queue.c` och blir därför privata för kömodulen.

För att skapa en ny kö anropar man funktionen `new_queue`. Man kan sedan lägga in element i kön med hjälp av funktionen `add`. När man anropar funktionen `add` styr prioriteten var det nya elementet läggs in. Ett element med hög prioritet placeras före ett med lägre prioritet och om flera element har samma prioritet hamnar dessa i s.k. FIFO-ordning (first in first out). Funktionen `add` har tre parametrar: kön, prioriteten och en pekare till det element som skall läggas in i kön. (Det är egentligen inte elementen som hamnar i kön, utan pekare till dem.) Den sista parametern har typen `DATA *`, där `DATA` är ett makro. Meningen är att den som använder kö-modulen själv skall definiera makrot `DATA`. Om man t.ex. vill skapa prioritesköer där elementen är poster av typen **`struct`** `person`, så skall man i sitt program lägga in raderna

```
#define DATA struct person
#include "queue.h"
```

Man kan säga att makrot `DATA` fungerar som en generisk parameter (template parameter), eftersom det används som parameter till modulen. Det finns emellertid ett problem i C: Det är inte tillåtet att ha överlagrade funktioner. Man får därför inte ha flera versioner av de funktioner som deklareras i filen `queue.h`. Filen får alltså inte inkluderas flera gånger med olika värden på makrot `DATA`. För att lösa detta kan man låta `DATA` ha typen **`void`**. `DATA` definieras därför villkorligt till detta i början av filen `queue.h`. (Då kommer kön att innehålla pekare av typen **`void`** `*` som kan peka till dataelement av vilken typ som helst.) Säkerheten minskar förstås i detta fall eftersom man förlorar typkontrollerna.

Funktionen `get_first` avläser det första elementet i kön, utan att ta bort det, och funktionen `remove_first` tar bort det första elementet. Funktionen `size` ger köns längd. Funktionen `clear` tar bort alla element ur kön, dvs. egentligen alla pekarna till elementen. Kön blir då tom, men kan användas igen. Funktionen `delete_queue` tar bort kön helt och hållet.

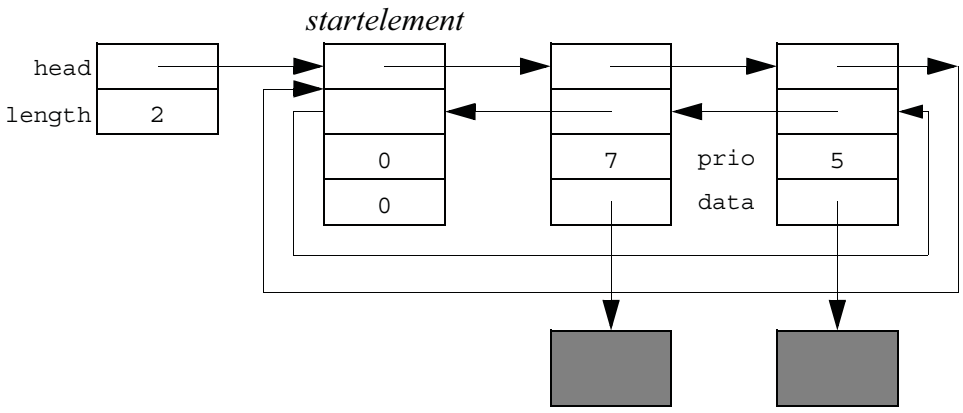
Man kan använda sig av en s.k. *iterator* för att löpa igenom en kö. Iteratorer beskrivs med typen `Iterator`. För att få en iterator som är kopplad till en viss kö anropar man funktionen `new_iterator`. En ny iterator refererar alltid till köns första element. Funktionerna `go_to_first` och `go_to_last` sätter iteratorn att referera till köns första resp. sista element. Funktionerna `go_to_next` och `go_to_previous` flyttar iteratorn ett steg i kön. För att avläsa det element iteratorn för tillfället refererar till anropas funktionen `get_current`. Om iteratorn inte refererar till något element returneras värdet 0. Om iteratorn inte refererar till något element returnerar dessutom funktionen `is_valid` värdet 0. Detta inträffar om kön är tom eller om man har stegat den för långt åt något håll. Med hjälp av en iterator kan man också ändra eller ta bort det aktuella elementet. Detta görs med funktionerna `change_current` och `remove_current`. Notera att `remove_current` skall se till att iteratorn refererar till efterföljande element. `change_current` får inte ändra på några data när den får en ogiltig iterator. Med hjälp av funktionen `find` kan man söka efter ett visst element. Om det sökta elementet finns sätts iteratorn att refererar till detta, annars sätts iteratorn så att den inte refererar till något element.

Implementeringen

Du skall implementera prioritetskön med hjälp av en *dubbellänkad lista*. En sådan består av ett antal sammanlänkade poster, s.k. *köelement*. Varje köelement innehåller två pekare. Dessa pekar på nästa resp. föregående köelement. I denna laboration skall varje köelement dessutom innehålla ett heltal som anger köelementets prioritet samt en pekare till ett dataelement. Köelementen beskrivs av typen **`struct`** `qelemstruct`, vilken skall definieras på följande sätt i filen `queue.c`:

```
struct qelemstruct {                    // typen för ett köelement
  struct qelemstruct *next, *prev;      // pekare till nästa och föregående
  int prio;                             // anger prioritet
  DATA *data;                           // pekare till dataelement
};
```

Figuren visar hur en prioritetskö, som för ögonblicket innehåller två dataelement, byggs upp. De två dataelementen (vilka kan ha vilken typ som helst) har markerats med skuggade rektanglar.



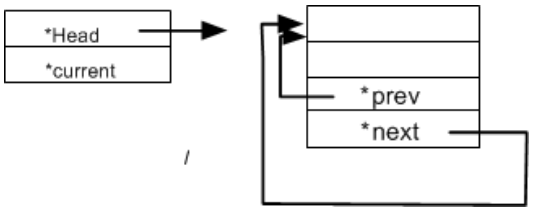
Rektangeln längst till vänster är ett objekt av typen **`struct`** `qstruct`. Detta objekt innehåller dels en pekare till det första köelementet och dels ett heltal som anger hur många dataelement som för tillfället ingår i kön. Typen **`struct`** `qstruct` skall definieras på följande sätt:

```
struct qstruct {                        // typen för en prioritetskö
  struct qelemstruct *head;             // pekare till startelementet i listan
  int length;                           // antal dataelement i kön
};
```

De tre övriga rektanglarna i figuren är köelement. Lägg märke till att dessa har länkats ihop cirkulärt så att framåtpekaren i det sista köelementet pekar till det första köelementet och bakåtpekaren i det första köelementet pekar till det sista köelementet.

När man arbetar med länkade listor visar det sig att fallen att en lista är tom eller att man skall sätta in eller ta ut ett element först eller sist ofta måste specialbehandlas. Detta gör att funktionerna som hanterar listor kan blir ganska komplicerade. För att slippa ifrån dessa problem är det praktiskt att låta varje dubbellänkad lista ha ett speciellt startelement som sitter först i listan. Då blir funktionerna mycket enklare.Vi skall utnyttja denna teknik i denna laboration. Det är därför det finns tre köelement i figuren ovan, trots att bara två dataelement har lagts in i kön. Observera

att startelementet inte pekar till något dataelement. Hur en tom kö skall se ut illustreras i följande figur. Notera att startelementet måste peka till sig själv, både framåt och bakåt.
När man lägger in ett nytt dataelement i en kö skall man skapa ett nytt köelement (allokera det dynamiskt) och låta det peka på det nya dataelementet. Därefter skall man länka in det nya köelementet på rätt ställe i den dubbellänkade listan. Prioriteten avgör placeringen. När man tar bort ett dataelement från kön skall man länka ur motsvarande köelement ur listan och därefter frisläppa det allokerade minnesutrymmet.



En iterator implementeras enkelt med hjälp av typen **struct** qiteratorstruct som skall definieras på följande sätt i filen queue.c:

```
struct qiteratorstruct {                // typen för en iterator
struct qstruct        *q;               // pekare till kön
struct qelemstruct *curr;               // pekare till aktuellt element
};
```

En iterator innehåller två pekare: en pekare till den kö som iteratorn är kopplad till och en pekare till det köelement i listan som iteratorn för ögonblicket refererar till. Om pekaren curr pekar på det speciella startelementet tolkar man det som att iteratorn inte refererar till något dataelement.
Observera att en iterator inte får uppträda cirkulärt. Om pekaren curr pekar på startelementet skall man inte kunna förflytta iteratorn med hjälp av funktionerna go_to_next och go_to_prev.

Godkännande
Din kömodul skall provköras med programmet i filen qtest.c. När programmet fungerar skall det visas upp för kursansvarig i kursen för godkännande. För att laborationen skall bli godkänd räcker det inte med att programmet fungerar. Dina funktioner måste också vara skrivna på ett snyggt och begripligt sätt. Programraderna skall t.ex. indenteras (dras in) på det sätt som lärs ut i kursen.
Redovisning av denna uppgift görs under LV 1 2012. Anmälan görs till Peter Lundin via mail (peter.lundin at chalmers se) varefter tid föreslås för redovisningen.