```
//*******************************************************************
// File:        semaphore.cc
// Author:  M. Thaler    15.01.2003
//
// Semaphor operations
//*******************************************************************

#include "semaphore.h"

//*******************************************************************
//* Static class variables

int    Semaphore::numOfSems = 0;
int    Semaphore::semID = 0;
char*  Semaphore::keyFilename = NULL;
int    Semaphore::projectID = 0;

//*******************************************************************
//* Constructor & Destructor
//* if number > 0 then create array with num semaphore
//*      else get existing semaphore

Semaphore::Semaphore(int num) {
  numOfSems = num;
  createSemaphorArray();
}

Semaphore::Semaphore(int num, const char* keyFile, int projID) {
  numOfSems = num;
  keyFilename = (char *)keyFile;
  projectID = projID;
  createSemaphorArray();
}

Semaphore::~Semaphore() {}    // do nothing

//*******************************************************************
//* release semaphore

int
Semaphore::up(int semaphor) {
  struct sembuf buf;
  buf.sem_num = semaphor;   // semaphor number
  buf.sem_op = 1;     // add 1 to value
  buf.sem_flg = 0;    // SEM_UNDO ist explicitly not
          // set, since otherwise a full
          // reset is made on exit, which
          // complicates termination
  return semop(semID, &buf, 1); // do it
}

//*******************************************************************
//* wait for sempahore to be release (if closed)

int
Semaphore::down(int semaphor) {
  struct sembuf buf;    // semaphor number
  buf.sem_num = semaphor;   // semaphor number
  buf.sem_op = -1;    // sub 1 from value
  buf.sem_flg = 0;    // SEM_UNDO ist explicitly not
          // set, since otherwise a full
          // reset is made on exit, which
          // complicates termination
  return semop(semID, &buf, 1); // do it
}

//*********************************************************************
//* get value of semaphor

int
Semaphore::getValue(int semaphor) {
  semun sem_union;
  return semctl(semID, semaphor, GETVAL, sem_union);
}

//*********************************************************************
//* set value of semaphor

int
Semaphore::setValue(int semaphor, int value) {
  semun sem_union;
  sem_union.val = value;
  return semctl(semID, semaphor, SETVAL, sem_union);
}

//*********************************************************************
//* cleanup: destroy semaphor array and delete key file

void
Semaphore::removeSemaphore(void) {
        semctl(semID, 0, IPC_RMID);
  if (keyFilename != NULL)
    unlink(keyFilename);
}

//*********************************************************************
//* Local procedures
//* obtain an array of Semaphores

int
Semaphore::createSemaphoreArray(void) {
  key_t key = IPC_PRIVATE;
  semun sem_union;
  int   flags, semErr;

  if (numOfSems > 0)
    flags = 0664 | IPC_CREAT;
  else
    flags = 0;

  if (keyFilename == NULL) {
    cout << "sem: not implemented feature\n";
    exit(-1);
  }
  // create key file, if not available
  int fd = open(keyFilename, O_RDWR | O_CREAT, 0770);
  close(fd);

  // get key by key file and ID
  key = ftok(keyFilename, projectID);

  // obtain semaphor array and initilaize to 0
  semID = semget(key, numOfSems, flags);
  semErr = semID;
  if (semErr > -1) {
    sem_union.val = 0;
    for (int j = 0; j < numOfSems; j++) {
      if ((semErr = semctl(semID, j, SETVAL, sem_union)) < 0)
        break;
    }
  }
  if (semErr < 0) {
    cout << "failed to allocate semaphore array\n";
    exit(-1);
  }
  return semID;
}

//***********************************************************************
```