

deKs

Richard Bartholemew
Ryan Butler
Steven Kurtz
Andrew Sheehan

Abstract

We decided to write an application named deKs for the Catholic Worker House in South Bend. This organization receives donations from multiple donors, but currently does not have an efficient way of keeping track of these donations. The purpose of our project is to provide the Catholic Worker House with a simple and easy to use application that will allow users to create and save files and add and remove donors and donations. The application will also be able to sort donors and donations to make them easier to navigate.

Key Terms

Donor, Donation, Address, Date, Donorlist, Donationlist

General Description

Steven Kurtz's firsthand involvement with the Catholic Worker House led to our correspondence with them. Steven realized they needed a better way to monitor the donations they received and the donors that made those donations. We felt this would work well with our assignment.

Members of our group made a failed attempt to implement the final project for CSE212 using Qt. After meeting with the client, it was decided that another effort at implementing a GUI using Qt would not only give us a slick looking user interface, but would also enable us to design in the environment of our choice, Linux.

Although our client runs Windows, the Windows version of Qt is for trial purposes only and programs created with it will only work for a month. Obviously, this is not acceptable, so we turned to the Linux version 3.3, which has as much usability as the Enterprise version of Qt which costs \$2,490 per developer. Also, given several members had Qt installed on their Linux machines, it made the design more convenient. Our current course of action is to get a working version of deKs running on a Linux box. Then, using Qt's qmake features, make a Windows executable. We then install Linux on a partition at the Catholic Worker House and run deKs from there, or make our own LiveCD that can boot a computer into a Linux environment and have deKs run directly from the LiveCD. We are

confident that one of these methods will enable us to get deKs operating at the Catholic Worker House.

Qt has excellent online documentation which makes programming with it fairly straightforward. Qt provides numerous classes we used to implement our application. By searching Trolltech's online documentation, we were able to find a member function that would perform the task at hand. The most difficult challenge involved getting the different dialogs to interact with to each other. Most of the information we found concerning Qt usage was found on TrollTech's documentation website. (<http://doc.trolltech.com/3.3/>) When problems arose, we searched the Qt-interest mailing list to see if somebody else had the same difficulties, and had found working solutions. (<http://lists.trolltech.com/qt-interest/>)

Functionality

The program works with an intuitive GUI that is structured in a manner similar to a standard database application. There is a File menu at the top of the screen that allows the user to create, open, and save files. There is a Donor menu that allows the user to add, edit, and remove donors. There is also a Donations menu that allows the user to add, edit, and remove donations for the selected user. Also found along the menu bar is the Edit menu which is primarily used to find a user. There are several icons below the menu bar that allow for quick access to the functions New, Open, Save, Save As, and Find.

UML Based Overview

The main focus of deKs is to provide an efficient way to handle large amounts of data. In addition to storing the data, we need to provide ways to access the data quickly and sort it by a number of criteria. We wrote four main classes to help us fulfill these desires effectively. We made donor and donation classes that hold all of the pertinent information for those entities. The donor class kept track of not only the donor's personal info, but also a list of all of that donor's donations. To manage all of the donor and donation objects, we created donorlist and donationlist classes. They had a "has-a" inheritance of their respective data classes. These classes were in charge of managing the donors and donations and allowing the main program to access them as needed.

Key Data Structures Used and Subsequent Rationale

We chose several different data structures and sorting algorithms to

accomplish the tasks required by our classes. The underlying data structure for the donor list is a hash table. The hash table consists of a vector of lists of donors that stores all of the donors in a file in alphabetical order. This turned out to be a very efficient method. Each of the indices of the vector corresponds to a letter of the alphabet, and the lists inside the vector hold the donors whose last name begins with the letter of that vector index. This allows us to maintain a constant alphabetical order among the donors and the use of lists provide easy insertion and removal anywhere in the list. Finding donors is also greatly simplified because only one of the twenty- six lists needs to be checked. The underlying data structure for the donation list is a list of donations allowing easy addition and subtraction of donors.

When it came to sorting, we decided to switch to vectors as opposed to lists for two reasons. The container used for sorting did not need to be resized during the sorts, and the best way to sort is to swap elements. A vector is ideal because elements can be moved according to indices. Once we had decided on the appropriate container, we had to choose the best sorting algorithms. Of the sorting algorithms, we obviously expect the fast sorts to be faster than the slow sorts. However, we do not think there will be a significant difference. The Catholic Worker House expects to have around several hundred donors and maybe five hundred donations. We wrote the program with this in mind, meaning that although the fast sorts may be five or even ten times faster than the slow sort, the difference in the amount of time it takes to sort three or four hundred elements will not be noticeable on a modern computer. Another factor we needed to take into consideration is the amount of code required for each sort. The fast sorts each require two helper functions and end up taking up two to three times more code than the slow sort. In addition, sorts like merge sort require even more space since they are not in place searches. The differences in speed ended up being negligible in our case, so we decided to use mostly slow sorts (e.g. insertion).

Performance/Efficiency/Results

After implementing all of the sorts, we compared efficiency by counting the number of assignments that needed to be made for each. As we expected the fast sort completed the sort with less assignments, but for our project the difference was not very significant. For small files, the ratio is nearly one to one. For a large file of one thousand donors, the quick sort turned out to be twenty- five times faster. However, the files that will need to be sorted will be less than half the size of the largest we tested, so we expect the fast sorts to be at most 12 times faster. This turned out to be a moot point though because in either case, it takes much longer to print the larger files than it takes to sort them, so the

difference will not be noticed by the user.

Error Handling

When creating our project in QT we did not use any formal error handling devices such as exception handling. Instead we tried to make our program as error-free as possible. One thing we did to improve the safety of our program is to input all data as QStrings. This takes away most of the hazards caused by the user inputting invalid data such as spaces or numbers instead of strings. We also prevent users from modifying data if they do not have a required field filled out. In all unnecessary fields, we automatically fill in default entries if the user does not input anything. Unfortunately there are a few circumstances that can still give you errors that these things do not prevent. We tried to remove these problems by testing the code over and over again to ensure that it works. In the future, before the client gets the program, we will install an autosave feature that saves every time data is modified. By doing this, even if the program terminates, no data will be lost.

Conclusions

We are incredibly happy with the progress that has been made in our deKs endeavor. While we did not reach the highest level of completion, we feel that we came a long way and are turning out a nearly finished product. We have a fully functional console-based application which fulfills all of the basic requirements of the Catholic Worker. Our QT-based program has the majority of the basic requirements but there are a few things we still plan to add. The GUI implementation of deKs does not sort donors and donations in as many ways as the console-application. It also lacks a few key features such as autosave, encryption, and a few other random wishes of the Catholic Worker. All of these we are close to being able to implement but simply ran out of time. But the QT version looks very impressive and is easy to use. Our plan for the future is to put together all of the odds and ends of this project so that we can actually give the Catholic Worker a program that they can use easily, safely and effectively.

As far as our personal development,

References

- TrollTech's Documentation Website -
<http://doc.trolltech.com/3.3/>
- Qt- Interest Mailing List Database -
<http://lists.trolltech.com/qt-interest/>
- C/C++ Reference -
<http://www.cppreference.com/>



Chris Moretti -

Helped get the console application to properly use getline

Garrett Britten -

Helped us to create a dialog as child window to the Main_Form

Biographies

<p>Steven Kurtz</p>	<p>Richie Bartholomew</p>
 <p>Steve Kurtz is currently a sophomore Computer Engineer at the University of Notre Dame. Outside of school, he spends all of his time playing Ultimate or hanging out with friends. In the future he hopes to graduate from college and win college nationals.</p>	 <p>Richie Bartholomew studies Computer Engineering at the University of Notre Dame. Richie is a member of the Notre Dame Ultimate Team and likes baking brownies. He hopes his future contains a prime-time cooking show on ESPN2.</p>
<p>Ryan Butler</p>	<p>Andrew Sheehan</p>



Ryan Butler is a Junior just making the jump from Mathematics to Computer Science. Outside of school, he frequents himself with those crazy Ultimate hippies, and putting various Linux Distributions on his POC laptop. He hopes to design really old hardware that won't burn your hands when you touch it.



Andrew Sheehan is a Junior Computer Engineering major who spends plenty of his time playing around with Linux. He enjoys playing Ultimate, and is a member of the Notre Dame Club team.