HÖGSKOLAN VÄST

2010-05-22

Memory management in a large project developed in C/C++

Peter Suopanki

BACHELOR'S THESIS
Computer engineering and system development, 180 hp

EXAMENSARBETE

Minneshantering i ett större projekt utvecklat i C/C++

Sammanfattning

Detta examensarbete har varit en studie i minneshantering i C/C++, och med praktiskt arbete för att hitta och lösa minnesläckor i ett större projekt. En bakgrundsteori om C++ och dess minneshantering är presenterad med tillhörande minnesproblem som kan uppstå under utveckling i C++. Flera ämnen om hur man minimerar minnesproblem med C/C++ inbyggda standard bibliotek och Boost biblioteket har också presenterats. En översikt över PCL-Opt 2.2 Beta programvaran med dess beståndsdelar har presenterats. Metoden var att lära sig allt om minneshantering och lösa minnesläckorna i PLC-Opt programvaran. Resultaten visade att det i början fanns ungefär 28,000 instanser av minnesläckor och slutresultatet visade att cirka 700 instanser av läckor lämnades olösta. En lista med ledtrådar till de olösta läckorna beskrevs i kapitlet om fortsatt arbete.

Datum:	2010-05-22
Författare:	Peter Suopanki
Examinator:	Dr. Thomas Lundqvist
Handledare:	Dr. Fredrik Danielsson, PTC
Program:	Datateknisk Systemutveckling
Huvudområde:	Datateknik
Utbildningsnivå:	grundnivå
Poäng:	180 högskolepoäng
Kurskod:	EDT501
Nyckelord:	*Minnesläcka, Minneshantering, C++, Borland, CodeGuard, Boost*
Utgivare:	Högskolan Väst, Institutionen ingenjörsvetenskap, 461 86 Trollhättan Tel: 0520-22 30 00 Fax: 0520-22 32 99 Web: www.hv.se

BACHELOR'S THESIS

Memory management in a large project developed in C/C++

Summary

This thesis has been a study in memory management in C/C++, with practical work in finding and solving memory leaks in a large project. A background theory of C++ and memory management has been presented along with the memory problems that can arise in development in C/C++. Several topics in minimizing memory problems with using inbuilt C++ libraries and the Boost libraries have also been presented. There is also an overview of the PLC-Opt 2.2 Beta application and its inner workings. The method was to learn all about memory management and solve the memory leaks in PLC-Opt. The results showed that there were approximately 28.000 instances of memory leaks and the end result showed that approximately 700 instances of leaks remained unresolved. A list with clues of the remaining memory leaks was provided in the Future work chapter.

Date:	May 22, 2012
Author:	Peter Suopanki
Examiner:	Dr. Thomas Lundqvist
Advisor:	Dr. Fredrik Danielsson, PTC
Programme:	Computer engineering and system development
Main field of study:	Computer Engineering
Education level:	Undergraduate level
Credits:	180 HE credits (see the course syllabus)
Course code:	EDT501
Keywords	*Memory leak, Memory Management, C++, Borland, CodeGuard, Boost*
Publisher:	University West, Department of Engineering Science SE-461 86 Trollhättan, SWEDEN Phone: + 46 520 22 30 00 Fax: + 46 520 22 32 99 Web: www.hv.se

Preface

This thesis project was done at the Production Technology Centre (PTC) in Trollhättan with weekly meetings with supervisor Fredrik Danielsson and Emile Glorieux. I would like to thank Fredrik Danielsson and Emile Glorieux for their guidance in understanding the PLC-Opt application and the help in finding the memory leaks.

Table of contents

1 Introduction

A research group at the Production Technical Centre (PTC) in Trollhättan has been working on a virtual manufacturing project that started in 2006. A simulation model for a sheet metal press line for the automotive industry has been developed where an actual robot sheet metal press line at Volvo Car Manufacturing is replicated in a virtual computer simulation that makes it possible to simulate the industrial process [11].

The actual physical press line at Volvo is manually fine tuned today by the operators every time a new component is to be manufactured. When a new component is going through the press line, all the settings have to be set from scratch by the operators and are depending on the experience of the operators of how well it operates. To save money and time and find the optimum settings and not rely on the operator experience, an optimization application called PLC-Opt has been developed by Dr. Fredrik Danielsson and Emile Glorieux at PTC with some additional student contributions. The purpose of the application is to calculate the optimal settings for such cases as the press line.

The PLC-Opt 2.2 Beta application has been under development since October 2011 and is developed with Borland C++Builder 6. The source code consists of approximately 33.500 lines of code (LOC) as counted with LocMetrics (Source code line counting tool) [7]. The application is programmed mainly in C++ language but has some C classes too, and the code has had contributions of several programmers with the consequences that many resource leaks and bugs have been introduced. The aim of this project is to have an application that can run for longer periods of time and be able to run as fast as possible because of the time intensive optimization algorithms.

1.1 Goal

The purpose is to do a study in memory management in general and techniques about how to avoid memory management problems. The goal is to get an understanding of the memory leaks that exists in the PLC-Opt application and finally solve them with the benefit of having an application that can be used for longer periods of time without errors caused by memory depletion.

2 Methodology

Weekly meetings will be held with Supervisor Dr. Fredrik Danielsson and Emile Glorieux at PTC. The project is to be carried out in the following phases:

2.1 Source code- and development environment study

Analyze the PLC-Opt 2.2 Beta application and in the same time learn to use the Borland C++Builder 6 development environment where there is no experience or knowledge of on personal basis. Another goal is to learn to use the PLC-Opt application with running optimization algorithms and other functions.

2.2 Perform a research in C/C++ memory management

Perform a study in C/C++ dynamic memory management in how allocation and deallocation is done and theories of how to avoid memory problems.

2.3 Choose runtime error tools

Borland C++Builder 6 has a built in runtime error tool called CodeGuard but to confirm the results at least two other tools has to be searched for, if there are any free tools or trial versions that are suitable.

2.4 Solve memory leaks

Solving of the memory leaks are done in an iterating manner with the following steps:

1. Analyze memory leaks with runtime error tools.
2. Choose a reported leak to work on.
3. Try to solve the leak.
4. Analyze with the aid of the runtime error tools to see if solving of a leak was successful.
5. Manual testing of the application to confirm functionality.
6. If the test was successful start over from step number 1.

3 Background theory

In this chapter several topics are explained to give a brief understanding about the tools and methods for memory management in C/C++ languages including information about Borland C++Builder.

3.1 C

C was evolved from the B and the BCPL languages by Dennis Ritchie at Bell Laboratories in 1972 [3]. Programming in C is function based and it lacks object-oriented programming. C can be hardware independent if the programs are carefully designed. The C language started first as a development language of the UNIX operating system. Almost all of the operating systems of today are written in C and/or C++ [3].

3.2 C++

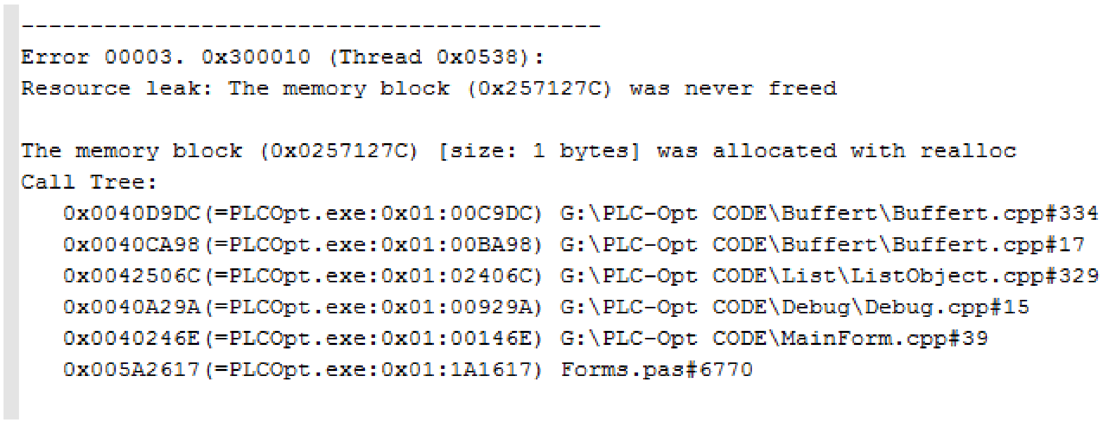
C++ was developed by Bjarne Stroustrup at Bell Laboratories and it is an enhanced version of C with the biggest improvement in added object-oriented programming capabilities [5]. C++ is one of the most popular languages and has powerful capabilities with a combination of high-level and low-level language features. Many other popular languages have been influenced by C++, most notable are C# and Java [3]. C++ is suitable for large applications where good performance is important such as in device drivers, video games and embedded software [3].

3.3 Borland C++Builder 6

C++Builder 6 released in 2005 is a RAD (Rapid Application Development) environment that was one of the leading development tools around at the time of release. As of today Borland has been acquired by Embarcadero Technologies and the latest environment released in 2011 is called Embarcadero C++Builder XE2.
C++Builder 6 integrates the Delphi Visual Component Library that consists of many components that are used in the creation of C++Builder applications [4].

CodeGuard

The CodeGuard runtime error tool is incorporated in the C++Builder 6 environment. It outputs a list of runtime errors and a list of called functions like the amount of new and delete that are being called. The leaks and other memory problems are logged with their line numbers and class names ordered in backward order according to the class tree. The log file from CodeGuard is saved with the same name as the project with the file extension “cfg”. The log file is a pure text file and it can be opened in notepad. The enabling of CodeGuard is done in the C++Builder CodeGuard tab that can be found in the project options. To get an error log output in a text file the application should be compiled first and then be executed with the compiled exe file. The error log is then created after the application is closed. An example output from CodeGuard can be seen in figure 1:



```
------------------------------------------
Error 00003. 0x300010 (Thread 0x0538):
Resource leak: The memory block (0x257127C) was never freed

The memory block (0x0257127C) [size: 1 bytes] was allocated with realloc
Call Tree:
   0x0040D9DC(=PLCOpt.exe:0x01:00C9DC) G:\PLC-Opt CODE\Buffert\Buffert.cpp#334
   0x0040CA98(=PLCOpt.exe:0x01:00BA98) G:\PLC-Opt CODE\Buffert\Buffert.cpp#17
   0x0042506C(=PLCOpt.exe:0x01:02406C) G:\PLC-Opt CODE\List\ListObject.cpp#329
   0x0040A29A(=PLCOpt.exe:0x01:00929A) G:\PLC-Opt CODE\Debug\Debug.cpp#15
   0x0040246E(=PLCOpt.exe:0x01:00146E) G:\PLC-Opt CODE\MainForm.cpp#39
   0x005A2617(=PLCOpt.exe:0x01:1A1617) Forms.pas#6770
```

Figure 1. CodeGuard error report example

Borland VCL

The Visual Component Library (VCL) is a framework developed by Borland for use in Delphi and C++Builder RAD (Rapid application development) tools. The components are written in Delphi but they can be used in C++ in the C++Builder environment. The library consists of Windows controls like forms, buttons, labels etc. and others like the *TList* collection class. Focus is put on explaining the classes which are appearing in the memory leaks: *TList* and *TStringList*.

TList

Creates a list of objects of any type and the objects are ordered in an array. When objects are added *TList* adds them in an expanding list.

To initiate a list, an instance of a *TList* must be declared in a function or in a class instance if the list is to be accessed outside a function or event. As with all VCL objects the *TList* must be declared as a pointer [6].

```
TList *ListOfObjects;
```

The *TList* pointer must then be initialized in the constructor of the class with using the new operator. To add objects to a *TList* the *Add* method is used. The first object receives an index of 0 and the *Count* member variable in *TList* will increment with 1. Each object is stored in an array of *Items[ ]* [6] .

To clear and delete a *TList* the method Clear is used *ListOfObjects->Clear();* and then delete the cleared list with delete *ListOfObjects*. The advantage with *TList* is with deleting the list contents there is no need to traverse through the list to delete every object, but instead the *Clear* method of *TList* does this task for us. But if the *Clear* method is not run there would be memory leaks because the list pointer would be deleted and the objects would remain in memory without any reference left to access them [6].

TStringList

It behaves much like *TList* but is used for storing strings in a list and adds some other functions that make it better suited for strings than using strings with *TList*. The dynamic allocating and deallocating of memory is handled the same way as with *TList*.

3.4 Memory management

There are three different ways to use memory in C++, static memory, automatic memory (stack) and free store (heap) [10].

Static memory
When an object is allocated in static memory it is constructed once and is kept in scope under the duration of the program. Static memory can consist of global variables, namespace variables, static variables and static class members [10].

Automatic memory
Function arguments and local variables are allocated in automatic memory and they are automatically created and destroyed. Every function gets its own copy [10].

Free store
The free store is where programs requests memory and where it can free memory again when it is done using it. When more of the free store is required the program can request it from the operating system [10].

The C/C++ languages do not have any garbage collector for automatic memory management so it is left for the programmer to deal with memory allocation and deallocation. If an application is programmed to allocate memory during runtime because the size cannot be determined in the source code, the memory has to be allocated dynamically during runtime.

3.4.1 Dynamic memory management in C/C++

This part is an overview in dynamic memory allocation in the free store for both C and C++.

In the C language memory allocation is done with the use of *malloc, calloc* or *realloc*. Dynamic arrays are created with *calloc* and the function *realloc* modifies the size of objects previously allocated with *malloc*, *calloc* or *realloc* [3].

With C++ memory allocation of objects or fundamental types is performed with the operator new. When allocating arrays the new[ ] operator is used. There is no similar operator to *realloc* in C++ but Bjarne Stroustrup recommends the use of the standard library containers of which vector would be more suitable [12].

To release the memory allocations in C the free function is used by including the pointer to the memory address. The pointer should also be set to NULL to avoid dangling pointers (see fig 3 and 4). To deallocate with C++ the delete operator is used together with the pointer to the memory address. For arrays the operator *new[ ]* is used for allocation and *delete[ ]* for deallocation [3].

The C style of memory allocation can be used in C++ but it is advisable to use the new and delete operators as much as possible. It is not possible to mix between the use free on objects or types that are allocated with new and vice versa and the same goes for allocating with *new[ ]* and deallocating with delete without brackets [3].

If the resources are not freed there will be memory leaks and the computer can eventually run out of memory when the application is run under longer periods of time. A graphical representation of a memory leak can be seen in figure 2.

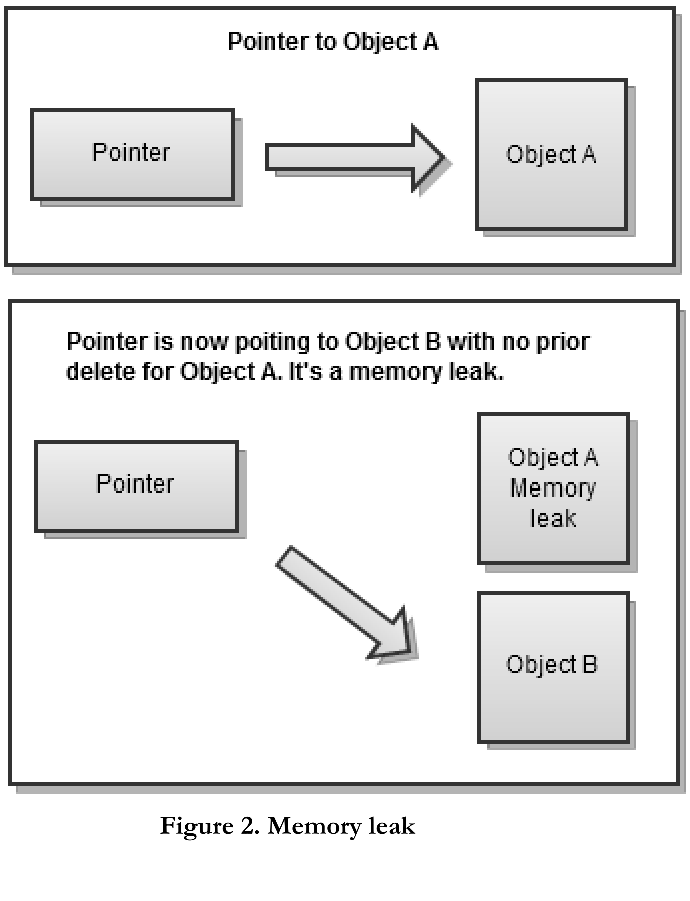

Pointer to Object A
Pointer
Object A
Pointer is now poiting to Object B with no prior delete for Object A. It's a memory leak.
Pointer
Object A Memory leak
Object B

Figure 2. Memory leak

Dangling pointers

A dangling pointer is a common programming mistake that happens when a pointer that points to an object where the memory address already has been freed or the objects lifetime is over and is out of scope. A graphical presentation of a dangling pointer caused by a deleted object can be seen in figure 3. Another presentation of a dangling pointer when a shared object is deleted with one owner object can be seen in figure 4.

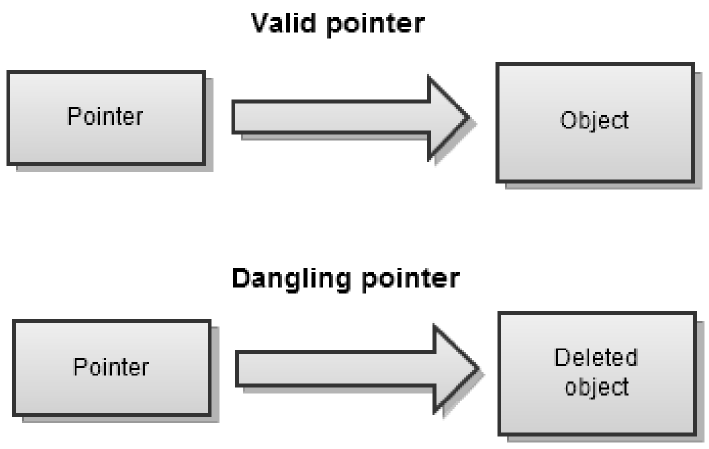


Figure 3. Dangling pointer example 1

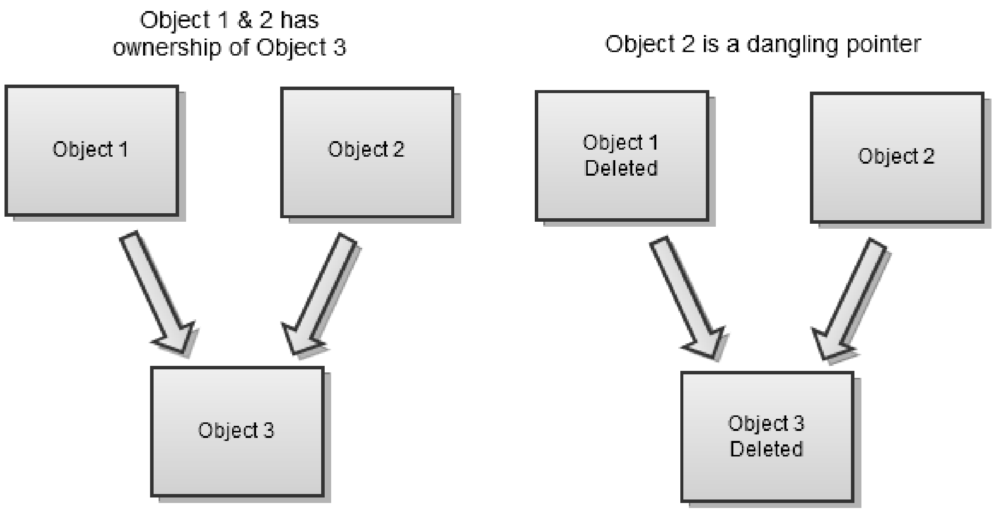

Object 1 & 2 has ownership of Object 3
Object 1
Object 2
Object 3
Object 2 is a dangling pointer
Object 1 Deleted
Object 2
Object 3 Deleted

Figure 4. Dangling pointer example 2

4 Avoiding memory problems

How can we minimize the risks in adding memory management problems in our code? One way is to use the C++ Standard Template Library (STL) functions. There are also other downloadable libraries like the smart pointers in the Boost library that helps developing safer code [1]. This part will present these techniques;

String

The *String* class is included in the STL and it is a safe alternative to using *char** array as a string. With the use *char** array, a programmer have to take care of the dynamic memory allocation and that can increase the risk of memory leaks. *String* takes automatically care of memory management and therefore takes away the burden of memory allocation from the programmer [10].

Vector

Vector is a container class in the STL and it is an alternative for using a regular array with the benefit of it taking care of memory management automatically. It can be used for creation of dynamic arrays of variables, struct or objects [2].
Compared to a regular array, *vector* consumes more memory because it reserves extra storage space in case it grows. The point with reserving extra space is to save computing time with not having to reallocate storage each time it grows unless the size is bigger than the previous allocation. Using vector helps avoiding memory leaks because there is no need to free vectors and there is no memory leakage when an exception occurs.

Dynamic two dimensional arrays

The old way of doing dynamic two dimensional arrays consisted of using a pointer pointing to another pointer. In this example one pointer is pointing to a row and the other is pointing to a column. This technique can lead to risk with memory leakage when the deletion is not handled the right order. Declaration of a two dimensional array of integers would then be:

```
int **int2dArray;
int2dArray = new*[numRows];
for(int i = 0; i < numRows; i++)
    int2dArray[i] = new int[numCols];
```

Using a double vector instead would prevent memory resource problems and declaration of a double vector would then be:

```
vector<vector<int> > int2dArray;
int2dArray.resize(numRows);
for ( int i = 0; i < numRows; i++)
    int2dArray[i].resize(numCols);
```

Boost C++ libraries

The Boost libraries extend the C++ language functionality with approximately 80 individual libraries and many of them are incorporated in the new C++11 standard [1]. The smart pointers are the most important functions in Boost for memory management purposes. The libraries are not integrated in C++Builder 6 and therefore needs to be downloaded and installed on the environment. The Boost smart pointers are header files and they don't have to be installed and can therefore easily be integrated into C++Builder by extracting the Boost folder to the C++Builder library folder and then a simple include in the program is all it takes [1].

Smart pointers behave like automatic garbage collectors that automatically deallocate resources when they get out of scope because the pointers keep track of the objects they point to. Another advantage with smart pointers versus regular pointers is that the deallocation is handled if an exception is thrown, and that is not the case with regular pointers that adds memory leaks if the deallocation is not taken care of in the exception handling. There are six different smart pointers, scoped_ptr, scoped_array, shared_ptr, shared_array, weak_ptr and intrusive_ptr [1].

scoped_ptr

With *scoped_ptr* dynamically allocated objects are automatically deleted without the need of manually deleting. It cannot be copied because it assumes ownership of the resource in the current scope. It is as fast as raw pointer and does not add any overhead. It's not suitable for arrays or Standard Library containers, but there are other smart pointers for those needs [1].

scoped_array

It has the same features as *scoped_ptr* with the difference that it is used in arrays. It is equivalent with the new[ ] operator for raw pointers. As with other smart pointers dynamically allocated arrays of objects are ensured of proper deletion without manually doing delete[ ][1].

shared_ptr

A *shared_ptr* is a pointer to an object that is shared among multiple pointers. It can be used in the Standard Library containers as for example the *vector* container. The object is automatically deleted by the last owner of the pointer. Disadvantages are that it is easy to cause circular references because of reference counted ownership of a pointer. Another disadvantage is that it has bigger memory footprint than raw pointers [1].

shared_array

It has the same features as *shared_ptr* with the difference that it shares the ownership of arrays [1].

weak_ptr

Observes *shared_ptr* owned objects and can therefore avoid the circular reference problem in *shared_ptr*. It avoids dangling pointers that can be the case with *shared_ptr*. It can be used in the Standard Library containers [1].

intrusive_ptr
Rarely used, often a *shared_ptr* is enough. Used when you need the "this" pointer to be treated as a *shared_ptr*. It is impossible to create a *weak_ptr* from an *intrusive_ptr*. A dynamically allocated object can have several *intrusive_ptr* which makes it a better choice to *shared_ptr* if that is required. Memory footprint is the same as for raw pointers [1].

5 PLC-Opt

The PLC-Opt application is developed to optimize parameter settings for a given problem. In this text a robot sheet-metal press at Volvo Cars is used as an example. It is now at a level where it can optimize one robot press station at a time in an isolated manner. The main objective in the future is to develop it to be able to optimize all the press stations together in one shot and get the best parameters for all the press stations [11].

The development of PLC-Opt started in October 2011 and has now been under development for approximately 7 months. The PLC-Opt version was in 2.2 Beta when this thesis work was done. Under this time there has not been any documentation done. An overview of the functioning of the application and also information about the source code can be read in Appendix A:1 and A:2.

The application has several optimization algorithms with different focus points. These algorithms can be chosen to be used in project files in the application to calculate minimum function values. There is also a programming language called SBasic incorporated which can be used to program macros and gives possibility to alter the behavior of the optimizations.

6 Result

The results are presented in the order of the different methodology phases.

6.1 Weekly meetings

Weekly meetings were held with Dr. Fredrik Danielsson and Emile Glorieux with the topics discussed on what accomplishments had been done and eventual problems that had arisen. The meetings ended with discussions on what problem the focus would be put on for the coming week.

6.2 Memory management research

A research in memory management was performed for both C and C++ with the focus on how dynamic memory allocation was performed and also theories of how to avoid introducing memory leaks in C++. The theories on how to avoid introducing memory leaks was to use the inbuilt Standard Template Library (STL) functions *String* instead of *char** arrays [10] and *Vector* instead of regular arrays [2]. A safer alternative to using two dimensional arrays with double pointers was to use double *Vectors* instead. Another advice was to use smart pointers from the Boost libraries, which are libraries that can be incorporated in the source code by including smart pointer header files alone.

6.3 Chosen runtime error tools

Two other runtime error tools were chosen to assist the output from CodeGuard a runtime error tool that comes with Borland C++Builder 6. The tools were AQTime [8] and Memory Validator [9]. Why AQTime was chosen was that it could produce an error list that showed how many leak instances there were in each object in compared to CodeGuard that outputs the leaks in no particular order and only outputs the total amount of errors and information of where the not freed objects were allocated. The choice to Memory Validator was because the tools capability to output a sum of the error to be able to compare the amount of the CodeGuard log.

6.4 Memory leaks solved

In PLC-Opt 2.2 Beta there were approximately 28.000 memory leak instances in the code as reported by the CodeGuard log. The leaks were tested with CodeGuard, with additional use of Memory Validator [9] and AQtime [8] to compare and confirm the CodeGuard error reports. All three tools reported the same amount of memory leaks.

The result of 28.000 leak instances was a combined result of running the main application with also running the optimization algorithms in the application itself. A decision was made to divide the problem to smaller parts and the first thing was to execute the application and close it directly afterwards to avoid adding any more leak instances. This way the focus was put on solving the leak instances that were created in the startup of the application. There were a total of 22.816 leak instances created in the starting of the main application.

By analyzing these leak instances it was easy to see that there were a lot of leak instances pointed to the same problem area. Codeguard reported that the majority of memory leak instances were in the functions *BuffertObject::Allocate* and *ListObject::ReadListFile*. The *ReadListFile* function was using the *Allocate* function in the *BuffertObject* class to allocate memory. *BuffertObject* class is written in C and the memory allocation is done with *realloc*. The first thing was to try to isolate the *BuffertObject* and *ListObject* classes in a new project and try to recreate the problem to see if the problems was in that class or elsewhere. There were no signs of memory leaks in those classes when proper deletion was taken care of.

With a lot more reading and analyzing of the code the source of the problem was found and it was some deallocations that were not performed in the *FormClose* function of *MainForm*. Another fix was to move down the deleting of *pApplicationController* out from an enclosed if statement and move it outside of that block so that it could be deleted every time a *pApplicationController* object goes out of scope. Another fix was to clear and delete *configlist* for the debug. These three fixes got rid of all the *BuffertObject* leaks and some more so the total amount of leaks decreased from the previous 22.816 leak instances to only 561.

In a meeting with Dr. Fredrik Danielsson the question came up why there were thses approximately 20.000 *BufferObject* instances created. The conclusion was that it had to be a bug because that amount was not realistic because the *BuffertObject* class is only handling outputting *char* arrays and he knew that there was nothing in the application that should create so many char strings instances in the applications startup. The problem was found by tracing where the objects were created and the problem was found in the *API* class.

Continued work lead to finding a missing delete in the destructor of *SBasic*, an instance of *Debug* was not deleted, that fix lead to a decrease of the leaks to two instances.

Another missing delete was found in *FormClose* where *WindowObjectList* was not deleted and the total amount of leaks decreased with two instances.

Deleted *GraphLogAxis* array that is used in the printing of graphs in the application and by deleting *FrameOPT->GraphLogAxis* in *MainForm*, decreased the leaks with one instance.

Deleting of *ModuleList* in the destructor of *Debug* was not done properly. The leaks were decreased with 8 instances.

The report from Codeguard was now running out of clear reports of where in the code the remaining leak instances were. The list had over 500 leak instances that were pointed to the Borland built in VCL components that are programmed in Delphi and CodeGuard is not capable to detect these leaks because VCL classes are not allocated with *new* but *SysGetMem* which is the Delphi way of doing it. It was then decided to force more error reports by running the different algorithms and other functions in the program. There were 151 leak instances added by actually using the application with its built in capabilities.

Now when there were new fresh leak instances with a clear report on where they could be located a problem in the *SBasic* class where *TList *VarList* had objects allocated with *malloc* and not freed. The list was looped through and every item was freed up by using *free(VarList->Items[i]);* with calling the *clear* function afterwards. The leaks were decreased with 19 instances.

By reading up on how *TList* works there were some classes found where the lists were not properly deleted. Before they were deleted by casting the list to a new list and then looped through the list and deleting every item one by one. That was unnecessary because *TList* takes care of deleting of all the items with one call to it´s *Clear* function.

The Nelder Mead algorithm added over one hundred leaks per iteration. Nelder Mead includes a matrix class that constructs the different matrixes and the leak instances were pointing to the matrix class. The problem was found in the destructor of the matrix class where a double pointer array was looped through in reverse order from end to beginning. The leaks were solved by looping from the beginning to end.

The thought of including the Boost library for smart pointers was not applicable because Boost would add some overhead which is not desirable in this project. With using the STL containers like *vector* and *string* there were no effort to change the existing code that was already working with using *char* arrays, dynamic two dimensional arrays and regular arrays.

The work that had been done was incorporated in an updated version of PLC-Opt that had been improved by Fredrik Danielsson and Emile Glorieux under these weeks the project was undergoing. By adding all the memory deallocation code to the updated version the memory leaks grew with 5 instances, so they had caused more leaks to the application.

There was a problem in the *Direct* algorithm where a delete was deliberately commented out because it produced an error in the application. The error was created in the very last iteration of the algorithm where an object tried to access some part of it and it was then already deleted. The *Direct* algorithm added leak instances for the total sum of approximately 700 leak instances in the whole application. By now the deadline of this project was coming closer and when the majority of the remaining leak instances being unclear VCL leaks, there was no more time to spend on searching for leaks.

The last part of the project was to gather a list of the remaining leak instances and where they could be located. A thorough testing of all the algorithms was done by running them one by one and counting the amount of leaks they added. The results from AQTime [8] were also gone through to get a list of where the remaining leaks could be located. All these results were done to give a good starting point to carry on for future work.

7 Conclusion

In this thesis, a study has been done about memory management in C/C++ and recommendations how to program with less risk of generating memory leaks. Some good practices on how that is done with the C++ standard libraries and Boost libraries have been presented. But these practices have not been implemented into PLC-Opt because of the added overhead it could introduce.

The main point is that it is very easy to make programming mistakes and cause memory leaks. The amount of leaks can quickly grow to big numbers if they are not being taken care of at regular basis.

The goal was to solve memory leaks but with the large amount of leak instances and unclear VCL leak instances there was unfortunately not enough time to solve them all. Still the amount was decreased considerably and should be easier to tackle for future work. The hardest part has been to understand where the allocated objects should be deleted, because it was easy to find where the allocations were done but harder to know when and where the deallocation could be done.

8 Future Work

There are still a lot of memory leaks left, and some analyzing has been done with the use of CodeGuard, AQTime [8] and Memory Validator [9]. Testing of the different algorithms was done by choosing a single evaluation plan to run the algorithms for a single iteration to prevent the errors duplicating.

The objective was to pinpoint memory leaks for every algorithm compared to the leak instances that already were reported in main program. AQtime [8] was used because it could present a list that made it easy to find out if the algorithm did add any leaks.

There were problems found in the *Direct* algorithm and the problem existed also in the *CoLiS* algorithm because it mixes the *Direct* with the *Neldermead* algorithm. The memory problems with the *Direct* algorithm is because the Direct algorithm is divided into three classes, *directclass.cpp*, *ClassDirect.cpp* and *point.cpp*. *ClassDirect* is instantiating a *directclass* object and the *directclass* uses the point class. The problem is when trying to delete the *directclass* object in the destructor of *ClassDirect*. If deleted the program crashes right before the end of the last iteration and CodeGuard says that there is an attempt to use already deleted references to the point class. At the moment the deletion is commented out and that is why there are leaks in that algorithm.

There were also problems when running the *DifferentialEvolution* but there is no reports in CodeGuard that there are any faults in the *DifferentialEvolution* class but the faults are related to the other known memory leaks that exist in *tVarList* and so on. A list with the other problems is listed below:

- **Debug.cpp**
  There are four not freed allocations leading to line 30 in the source code and the four problems are reported to be in *tProtocol*, *tSimulate*, *ParentAlg* and *tPlan*.

- **variable.cpp**
  There are ten not freed allocations leading to line 15, and they are divided in 3 errors in *simlist* and 7 errors in *sim.cpp*.

- **avarlist.cpp**
  There are eight not freed allocations with four of them pointing at lines 11, 22, 24, 44 and 45.The rest points to sim.cpp at lines 16, 279 to 282.

- **simulate.cpp**
  There is one error report in line 4.

- **SourceRosenbrock.cpp**
  An array named *Values* is somehow not deleted even if it a delete is implemented in the destructor.

- **Protocol.cpp**

Debug is allocated and deallocated in the destructor but leakage exists anyway.

- **Other unclear leaks**
  55 problems with *TList*, 3 *TStringList*, 1 *TWinHelpViewer*, 1 *tIterationPlan*, 3 *TItemProp*, 4 *AnsiString* and approximately 400 leading to VCL native memory allocations that does not give any hints about where they are. Probably related with the *TList*,*TStringList* and *TItemProp* leaks.

As listed there are many memory leak instances but that doesn't mean that there are 700 actual problem areas in the source code but they could be with a rough estimate around 50 actual problem areas based on the before mentioned leaks in the dotted list. The exact amount is hard to calculate because of the 400 anonymous leaks pointing at VCL native memory allocations.

9 References

1. Boost.org (2002) *Smart Pointers - Boost 1.49.0*. [online] Available at: http://www.boost.org/doc/libs/1_49_0/libs/smart_ptr/smart_ptr.htm [Accessed: 1 May 2012].
2. Cplusplus.com (2000) *vector - C++ Reference*. [online] Available at: http://www.cplusplus.com/reference/stl/vector/ [Accessed: 2 May 2012].
3. Deitel, P. and Deitel, H. (2010) *C How to Program*. 6th ed. New Jersey: Pearson Education, Inc.
4. En.wikipedia.org (2011) *C++Builder - Wikipedia, the free encyclopedia*. [online] Available at: http://en.wikipedia.org/wiki/C%2B%2BBuilder [Accessed: 2 May 2012].
5. En.wikipedia.org (1979) *C++ - Wikipedia, the free encyclopedia*. [online] Available at: http://en.wikipedia.org/wiki/C%2B%2B [Accessed: 5 May 2012].
6. Functionx.com (2004) *Creating and Using Lists*. [online] Available at: http://www.functionx.com/bcb/classes/tlist.htm [Accessed: 4 May 2012].
7. Locmetrics.com (2007) *LocMetrics - Source Code Line Counting Tool*. [online] Available at: http://www.locmetrics.com/index.html [Accessed: 9 May 2012].
8. Smartbear.com (2008) *Application Performance Profiling*. [online] Available at: http://smartbear.com/products/qa-tools/application-performance-profiling/free-aqtime-pro-trial [Accessed: 20 April 2012].
9. Softwareverify.com (2012) *Software tools for Windows developers, Software Verification*. [online] Available at: http://www.softwareverify.com/index.php [Accessed: 19 April 2012].
10. Stroustrup, B. (1997) The C++ Programming Language. 3rd ed. Reading, Massachusetts: Addison-Wesley, p.843-846.
11. Svensson, B. et al. (2007) Off-Line Optimisation of Complex Automated Production Lines – Applied on a Sheet-Metal Press Line. *Assembly and Manufacturing, 2007. ISAM '07. IEEE International Symposium on*, p.82-87.
12. www2.research.att.com (2005) Bjarne Stroustrup's Homepage. [online] Available at: http://www2.research.att.com/~bs/homepage.html [Accessed: 2 May 2012].

A. Appendix 1: PLC-Opt classes

This is an overview of the different classes in PLC-Opt.

ApplicationController.cpp
The main connector for all the parts in the application and also controls the saving and loading of project files.

Buffert.cpp
Handles char arrays in the application. Makes string handling in the rest of the application easier. Entire class is written in C.

Debug.cpp
Gives debugging capabilities to the application so that errors or other information be printed out on the applications log window.

ListObject.cpp
Handles lists wit adding objects to lists and loading and saving list files.

plan.cpp
Handles which algorithm is to be used in the evaluation plan

Protocol.cpp
Is handling the starting, stopping and resetting in the simulations.

SBasic.cpp
An internal programming language which enables building macros or changing evaluation functions and so on.

avarlist.cpp
Does the adding of variables to lists, and is also for loading and saving initiation files.

sim.cpp
It is using the variables from *avarlist*.

simlist.cpp
It uses both *avarlist* and *variable* for simulation lists.

Source.cpp
Is used to make a connection between the optimization algorithm to parameters and/or an evaluation function. A source might be a simulation tool, a function or an application.

SourceRosenbrock.cpp
Used as a testing function for debugging purposes.

variable.cpp
It manages variables for different input types.

A. Appendix 2: Optimization algorithms

There are several optimization algorithms implemented that have different approaches in calculating the optimal parameters by calculating minimum objective function values.

Direct
DIRECT for DIviding RECTangles. It solves difficult global optimization problems.

Neldermead
Nelder Mead is a Simplex based Direct Search Method. It uses an estimation of the objective function values for each point of the simplex

DifferentialEvolution
Differential Evolution is a kind of Evolutionary Computation and it is sometimes called greedy search or greedy selection process.

CoLis
Is a mixed algorithm between Direct and Nelder Mead.

MixDifferentialEvolutionNelderMead
Mixed algorithm between Differential Evolution and Nelder Mead

Screening
Screening Method uses Two Full Factorial Method of point selections. It is used to look at the best objective function value in the area limited by the high and the low value of each parameter.