Guide to Good Practice in using Open Source Compilers with the AGCC Lexical Analyzer

Rocsana BUCEA-MANEA-ȚONIȘ
Academy of Economic Studies, Bucharest, Romania
imm.online@yahoo.com

Quality software always demands a compromise between users' needs and hardware resources. To be faster means expensive devices like powerful processors and virtually unlimited amounts of RAM memory. Or you just need reengineering of the code in terms of adapting that piece of software to the client's hardware architecture. This is the purpose of optimizing code in order to get the utmost software performance from a program in certain given conditions. There are tools for designing and writing the code but the ultimate tool for optimizing remains the modest compiler, this often neglected software jewel the result of hundreds working hours by the best specialists in the world. Even though, only two compilers fulfill the needs of professional developers, a proprietary solution from a giant in the IT industry, and the Open source GNU compiler, for which we develop the AGCC lexical analyzer that helps producing even more efficient software applications. It relies on the most popular hacks and tricks used by professionals and discovered by the author who are proud to present them further below.
***Keywords**: registers, dynamic linkage, cache, null pointers, tweaking.*

1 Rules on the organization and naming files and variables

Many standards provide rules in this regard and a good example is the suffix *.h* of the file name that contains the header or the definition of a class project in C/C++. Not even a compiler directive or any rule of syntax does mention this type of file so it is only a convention matter. The files must structure together similar features. It will use a *.h* for files that will contain the function declaration or a class and extensions *.c*, *.cs* or *.cpp* for different implementations. These files will contain the default *#include* directive followed by the file name and the appropriate header and the following:

- comment which specifies the code author and rights related to copyright;
- commentary indicates that the functionality and destination file;
- definitions of local variables;
- interface prototypes for the class methods.

After [7], Hungarian notation is a rule in the name of variables invented by Charles Simonyi at Microsoft. Hungarian notation major advantage is that allows the name of variables depending on the type of variable. It is obviously that Hungarian notation does not make sense in pure object oriented languages since they will not use primitive but variable object. In C++, a handler to a window can be noted with *hWnd*, and a pointer to a numeric value can be abbreviated *pData*, achieving an economy of time and effort and keeping the code as clean as possible. Often the interest related to a variable is not the type, but the scope (local, static, global or member) so that MFC prefixes member variables with *m_* and with *c_* the static ones. This technique could be misleading when the variable type is changed but the variable name remains the same throughout the source code.

A well-documented software project may help to understand the overall logic, the construction and implementation of complex applications. We present below some relevant documentation, after [6]:

d_1. **system specification** - specify the major objectives of the application requirements for basic functionality and minimal configuration. This information describes the general and technical conditions to be met for optimum operation of the application;

d_2. **software specification** - describes in detail the software and overall architecture of the host system. They mentioned the type of

processing, type of interfaces, logical schemes of databases, etc. This specification may also include possible changes to the source code due to the evolution of technology; we may use standard software specification for performance measurement and benchmarking other applications from the same class;

d_3. **design specification** - provides the general architecture of the application, variable types and data structures, the interaction of various modules; in the case of object oriented software will be a description of the main classes and methods; we may use this specification to understand the code and other elements of its functionality;

d_4. the **results of testing** the application - containing information about the testing, the technical conditions under which testing was conducted and its outcome; based on these data we can evaluate at least part of the application functionality;

d_5. the **user's manual** - includes a detailed description of the functions of the application, installation instructions and an overview of the application. In most cases the manual is the only type of documentation which will access a potential customer, and it is very useful especially in beta applications.

2. The degree of complexity of the comment lines of code

Comments delimit distinct parts of the source code from the function or purpose to be achieved. To distinguish lines of code, some may use special characters (-, *), either proceed to use different font color and style. The detailed procedures shall be given in blocks, providing information on:

- the algorithm used;
- the meaning of the variables;
- the sequences;
- the last update;
- the bibliographic source base;
- restrictions on the variables domains.

Commenting source code for programs written in evolved programming languages, use the pair / * ... * /, and for brief comments from each of the programming instructions, it is used digraph //... *Grep* command is used to identify commented lines of code expression by providing standard character as argument.

```
% grep "^[ \t]*\* "
% grep "^[ \t]*\/\/"
```

Comments in Visual Basic code are employed introducing the '(apostrophe) in front of sequence we want to comment. The apostrophe takes effect only in the line of code that appears. An apostrophe inserted in a line of code has the effect of commenting everything following it until the end of the line. Commented text will appear in the green label [1]:

```
'member definition of Domiciliu Class
Option Explicit
Private m_Localitate, m_Judet, m_Strada
As String
'*****************                    city
********************
Public Property Get Localitate() As
String
Localitate = m_Localitate
End Property

Public     Property     Let     Locali-
tate(n_Localitate As String)
m_Localitate = n_Localitate
End   Property'*******************   to
continue
```

A number of lines too small or too large may be an indication that the program is more difficult to maintain or understand. It is preferable that a line of code does not end with a comment, or to have so many lines as needed to fit the comment in question. Exceptions are expressions like *#if/ #endif*. For more detailed comments is preferable to delimit blocks of comments situation encountered especially in functions. Bolded comments will delimit separate parts of the source code from the function or purpose to be achieved [2]:

```
/*
|////////////////////////////////////
//////////////////|
|///////////////| inLimite Class Tem-
plate |////////////////|
|////////////////////////////////////
//////////////////|
*/
template <class T>
class inLimite{
      const T& min;
```

```
        const T& max;
public:
        inLimite(const T& m, const T&
M):min(m), max(M){}
        bool operator() (const T& x){
                return x>min && x<max;
        }
};
```

Also, the comments at the end of line are acceptable when accompanying a variable declaration and explaining its purpose. It is preferable that the majority of comments to explain *Why* and not *How* the code works, and they are not written after drafting the code.

3. Standards and conventions used in editing the source code

A good practice in writing and debugging code for optimal performance of the functions includes after [3]:

- check the type of input parameters and returned values of the functions;
- use a template to eliminate unwanted conversions between data types;
- constant definition with the *const* type followed by the date type agreed;
- declaration of variables prior to use and release of memory space previously allocated immediately after being used;
- use of parentheses whenever logic calculation is not very transparent;
- assert use when handle exceptions to prevent errors;
- using an output parameter for the result and *return* for error when a function must return a calculated value and an error at the same time;
- checking error values returned by functions of the library system for those functions that provide access to system resources, such as *malloc()* or *open();*
- systematization of possible errors, based on the critic level of them and their different influence on the final results of the program.

It is recommended to avoid:

- conversion between different types of data and especially the conversion of a pointer data type to *void *;*
- definition of data types derived from pointers (ie: `typedef char * Sir_de_caractere`) and preprocessing constants with #define;
- introduce more in line instructions especially in alternative and repetitive structures;
- creating structures of more than 3 levels;
- invoking *exit()* inside a system library function;
- the use of *goto*, *break* or *continue* to exit from a structure and repetitive post-conditioned structures like *do {...} while ();*
- unsustainable use of global variables when processing concurrent flows and macro language.

It is recommended that radical changes to occur in the first phase of the development cycle so that problems that may arise to be solved, and adding the patches will be produced following a cost-benefit analysis. It aims to maintain complementarities with the standard C++ in the next version.

4. Optimizing code

The requirements of structured or object oriented programming conflicts with the optimization software to increase speed of execution. Even the modern hardware manages hard to keep pace with the complex software that requires grater space in memory and higher processing speed. Open source projects usually successful achieve the compromise between the limited hardware resources and programs tailored to user needs in respect of size and speed of execution.

The choice of programming language is a compromise between efficiency, portability and the development cycle. Preferred programming language is C/C++ for the following reasons:

- it is supported by modern compilers and software libraries of optimized functions;
- C++ is a high level language with multiple facilities;
- C allows access to system resources;
- most of the available C compilers outputs the generated assembler code for making new optimizations;
- accept inline assembler directives for higher optimizations;
- it is portable on most hardware platforms.

Disadvantages of using C language are:

- the need for separation of the graphical interface from the main functionalities;
- the development cycle for a C/C++ is considerably high;
- it has no tools for additional memory allocation for the arrays, nor for the avoidance of null pointers.

Before some work with the tweaking over the source code it is necessary to analyze the code with AGCC. The analysis may cover:
- how many times a routine is called;
- introduce break points to identify potential errors;
- generate temporary interruption to test the functionality of various components of the program;
- how the interruption manager of CPU affects the cache allocation by the program, errors in the affected control structures or exceptions in floating point calculations.

Inefficiency of a program C/C++ may be due to:
- loading resources in memory may further trigger the procedure for swapping to dynamic allocation of memory on the hard disk;
- use dynamic linked libraries over static loading leads to additional unnecessary functions;
- prolonged spend in reading and writing files;
- accessing a database in Windows can get a few seconds;
- writing a graphic is done by allocating small blocks of memory;
- the program bases on the resources accessible through a network;
- access to the RAM that exceeds the cache level 2 can be 100 times more slowly;
- access to multithreading CPU is mutual conditioned by the flows from different stages of processing.

In [4] are detailed the main techniques of optimization programs in C/C++:

A. Methods of allocating space in memory for variables
- using stacks by additional allocations in the same area of memory, all variables must be declared inside the function that is used;
- declare global variables involve using static memory which is divided into three parts one of which dedicated to constants, this memory stays busy until the end of execution of the program;
- using the CPU registers to allow access to data for 1-2 cycles amounts of time;
- dynamic allocation of memory using the heap memory can become fragmented in the allocations and reallocations of objects with different sizes.

B. Using the pointer and references
- pointers can be used in arithmetic operations and can change the value of the object they points to;
- the references cannot indicate an invalid address;
- it is required to allocate an additional register when using a pointer or reference.

C. Repetitive structures
- we recommend unrolling loops by a factor greater or equal 2 when the disadvantage of the cache congestion is justified by the elimination of additional instructions or loops;
- a loop is more efficient with a control structure that tests an integer and the test value does not result from internal structure of repetitive calculations;
- *memset()* and *memcpy()* can successfully replace loops for initialization or copying a row.

D. Calls to functions may slow down implementation of a program
- we recommend grouping instructions with the same scope in the same function regardless of the number of lines;
- is preferably a reference to an object than a function that returns the same object;
- a call to an inline function is replaced by the compiler with the body of the function;
- definitions may substitute a macro function but the parameters are evaluated every time;
- the integer function parameters can be called with *__fastcall* so that the first two parameters are transferred in the register and not stack;
- a static access modifier restricts functionality to the module that is part of.

Modern compilers operating a series of optimizations independently by the developer

which:
- replace expressions with their outcome;
- eliminating pointers by replacing them with the pointed values;
- sending variables used more often in the registry;
- analysis the lifetime of variables so that they can occupy the same register;
- merging identical instructions on the branches of a control structure;
- the jump instructions can be replaced by the instructions indicated by that jump;
- unrolling repetitive structures;
- remove outside loop instructions independent from the counter;
- reordering instructions so as to facilitate parallel processing;
- mathematical reductions;
- the virtualization of member functions through intelligent management of the code without consulting the virtual table.

5. Using assembly language programs for optimizing C/C++

Optimizing speed execution involves modifications on the intermediate file generated by assembler compiler C++ accordingly to [5]. In general, compilers performance cannot equal human developer writing code in optimized assembler. Furthermore, one application written in C appeals API routines to generate Windows objects:

```
#include <windows.h>
#define BOPEN1 1
#define BSTAT1 2
void InitApp(HINSTANCE);
LRESULT APIENTRY MainProc(HWND,UINT,WPARAM,LPARAM);
HWND hwnd;
HWND bopen;
HWND bstat;
HINSTANCE g_hInst;
char *text;
int APIENTRY WinMain(HINSTANCE hInst,HINSTANCE hPrev,LPSTR line,int CmdShow)
{      MSG msg;
       g_hInst = hInst;
       InitApp(hInst);
       while(GetMessage(&msg,0,0,0))
       {      TranslateMessage(&msg);
              DispatchMessage(&msg);
       }
       return msg.wParam;
}
void InitApp(HINSTANCE hInst)
{      WNDCLASS wc;
       wc.cbClsExtra = 0;
       wc.cbWndExtra = 0;
       wc.hbrBackground = (HBRUSH) GetStockObject(LTGRAY_BRUSH);
       wc.hInstance = hInst;
       wc.hCursor = LoadCursor(NULL,IDC_ARROW);
       wc.hIcon = LoadIcon(NULL,IDI_APPLICATION);
       wc.lpfnWndProc = (WNDPROC) MainProc;
       wc.lpszClassName = "Main";
       wc.lpszMenuName = NULL;
       wc.style = CS_HREDRAW | CS_VREDRAW;
       RegisterClass(&wc);
       hwnd = CreateWindow("Main","Simple Dialog", WS_OVERLAPPEDWINDOW, 50,
50, 150, 80,0,0,hInst,0);
       ShowWindow(hwnd,SW_SHOW);
       UpdateWindow(hwnd);
}
```

It is necessary to declare two types of handlers, one of which addressed *HWND* window itself and the other type *HINSTANCE*, refers to the *Win* type application. The main routine is called *WinMain* and receives the main argument handler application. In this main function is declared a variable of type *MSG* which keeps the user's answers in the form of messages and calls initialization procedure with the application handler. This

procedure contains a reference type *WNDCLASS* for its members initializations and *RegisterClass()* method to register window. The window is displayed with the *ShowWindow()* method and updated with *UpdateWindow()* method, both with the handler type argument to the window previously obtained with *CreateWindow()* method. Window behavior is dictated by the *LRESULT* function with the main window handler argument. The user's messages are handled in an alternative structure with multiple options (*switch*).

```
LRESULT APIENTRY MainProc(HWND hwnd,UINT msg,WPARAM wParam,LPARAM lParam)
{      switch(msg)
       {
       case WM_DESTROY: PostQuitMessage(0); break;
       case WM_CREATE:
              {      SendMessage(hwnd,WM_SETICON,1,(LPARAM)LoadImage(NULL,
"eye.ico", IMAGE_ICON, 16, 16, LR_LOADFROMFILE));
                     bopen = CreateWindow("BUTTON","OK",WS_CHILD | WS_VISIBLE |
BS_PUSHBUTTON,48,40,50,15,hwnd,(HMENU)BOPEN1,g_hInst,0);
                     bstat = CreateWindow("STATIC","Simple Dialog Written In
C++",WS_CHILD | WS_VISIBLE,2,20,140,18,hwnd,(HMENU)BSTAT1,g_hInst,0);
              } break;
       case WM_COMMAND:
              {      switch(HIWORD(wParam))
                     {
                     case BN_CLICKED:
                            switch(LOWORD(wParam))
                            {
                            case BOPEN1:
                                   { ShowWindow(hwnd,SW_HIDE);
                                     UpdateWindow(hwnd);
                                   } break;
                            } break;
                     }
              } break;
       default: return DefWindowProc(hwnd,msg,wParam,lParam);
       }
       return 0;
}
```

The source code is compiled by the free Borland C++ 5.5 utility with -*S* option to generate the assembler code properly. The result is a modal window (Figure 1) with a *Close* button, a welcome text and an *.ICO* formatted image in the title bar.

Generated assembler file contains 369 lines. Some may decide to use the free assembler editor MASM32 and compares simple.asm of the template folder dialogs:

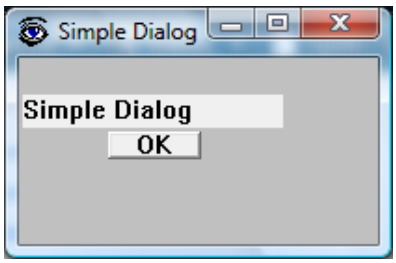

Fig. 1. Window generated by the *simple.c* program

```
      .486                      ; create 32 bit code
      .model flat, stdcall      ; 32 bit memory model
      option casemap :none      ; case sensitive
      include \masm32\include\dialogs.inc
      include simple.inc
      dlgproc PROTO :DWORD,:DWORD,:DWORD,:DWORD
    .code
start:
```

```
      mov hInstance, FUNC(GetModuleHandle,NULL)
      call main
      invoke ExitProcess,eax
main proc
    Dialog "Simple Dialog","MS Sans Serif",10, \           ; caption,font,pointsize
           WS_OVERLAPPED or WS_SYSMENU or DS_CENTER, \      ; style
           2, \                                             ; control count
           50,50,150,80, \                                  ; x y co-ordinates
           1024                                             ; memory buffer size
    DlgButton "&OK",WS_TABSTOP,48,40,50,15,IDCANCEL
    DlgStatic "Simple Dialog Written In MASM32",SS_CENTER,2,20,140,9,100
    CallModalDialog hInstance,0,dlgproc,NULL
    ret
main endp
dlgproc proc hWin:DWORD,uMsg:DWORD,wParam:DWORD,lParam:DWORD
    .if uMsg == WM_INITDIALOG
      invoke SendMessage,hWin,WM_SETICON,1,FUNC(LoadIcon,NULL,IDI_ASTERISK)
    .elseif uMsg == WM_COMMAND
      .if wParam == IDCANCEL
        jmp quit_dialog
      .endif
    .elseif uMsg == WM_CLOSE
      quit_dialog:
      invoke EndDialog,hWin,0
    .endif
    xor eax, eax
    ret
dlgproc endp
end start
```

Together with *simple.inc* file there are 227 lines of assembler code optimized, so a difference of 142 lines, with 38.5% fewer than in the version of compiler code generated automatically. We conclude that optimizing the code generated by compiler brings significantly more efficient execution of programs written in C/C++. We treated the previous example by moving the lines of common code in various functions outside their body, get a global functionality of the instructions and variables so the total length of the source code is reducing from 240 lines of code to 188 lines of code, so an improved efficiency in lines of code by over 21% .

In this way we avoid allocating space in memory several times for the same object and improve the overall readability of the code. To compare the original code with the optimized one we used the open source *Winmerge* that ensuring data integrity by creating back-ups for the optimized files.

6. AGCC, Graphviz and Cygwin compiler case study

Graphviz package was designed to rely on the UNIX like program-as-filter software paradigm, in which distinct graph operations or transformations are embodied as programs. Graph drawing and manipulation are achieved by using the output of one filter as the input of another, with each filter understanding a common Sugiyama-style hierarchical layout [8].

AGCC analyzer uses the Graphvitz software as a library with bindings in DOT language to describe the graphs and attributes attached as name-value pairs [9]. AGCC invokes the Graphviz renderers generating a drawing of a graph in a graphic format such as png:

```
gvRender (GVC_t *gvc, Agraph_t* g, char
*format, FILE *out);
gvRenderFilename (GVC_t *gvc, Agraph_t*
g, char *format, char *filename);
```

The first and second arguments are a Graphviz context handle and a pointer to the graph to be rendered. The final argument gives a file stream open for writing or the name of a file to which the graph should be written. The third argument names the renderer to be used, such as "ps", "png" or "dot".

Libcgraph supports graph programming by maintaining graphs in memory and reading and writing graph files. Graphs are composed of nodes, edges, and nested sub graphs.

These graph objects may be attributed with string name-value pairs and programmer-defined records.

A program example adapted from Stephen North from AT&T Research and used by AGCC may be sketched like this:

```
#include "cgraph.h"
#include <stdio.h>
typedef struct mydata_s {Agrec_t hdr; int x,y,z;} mydata;
int main(int argc, char **argv)
{
	// Graphviz inner defined types for building a graph
	Agraph_t *g;
	Agnode_t *v;
	Agedge_t *e;
	// String attributes of nodes, edges and graphs identified by name and by an
internal symbol table entry created by Libcgraph
	Agsym_t *attr;
	Dict_t *d;
	// Contor for counting the graph edges
	int cnt;
	// Developer strucure for filling the graph in
	mydata *p;
	// Source file in .dot style
	FILE *fl;
	if (argc > 1)
		fl = fopen(argv[1], "r");
	else
		fl = stdin;
	if (g = agread(fl,NIL(Agdisc_t*))) { //Agdisc_t defines callbacks to be in-
voked by libcgraph when initializing, modifying, or finalizing graph objects
		cnt = 0; attr = 0;
		while (attr = agnxtattr(g, AGNODE, attr)) cnt++; // agnxtattr permits
traversing the list of attributes of a given type
		printf("The graph %s has %d \n",agnameof(g),cnt);
		/* Make the graph have a node color attribute, default is blue */
		attr = agattr(g,AGNODE,"color","blue"); // agattr creates or looks up
attributes
		/* Counts all the edges of the graph */
		cnt = 0;
		for (v = agfstnode(g); v; v = agnxtnode(g,v))
			for (e = agfstout(g,v); e; e = agnxtout(g,e))
				cnt++;
		/* attach records to edges */
		for (v = agfstnode(g); v; v = agnxtnode(g,v))
			for (e = agfstout(g,v); e; e = agnxtout(g,e)) {
				p = (mydata*) agbindrec(e,"mydata",sizeof(mydata),TRUE);
agbindrec attach records to individual objects one at a time
			}
	}
	return 0;
}
```

In order to build the program authors used the Cygwin compiler that adapts GNU's GCC compiler to Windows environment. The two header files, cgraph.h and cdt.h must be also present in the project.

The command line in order to compile the original C file and create the object file is the following:

```
g++ -c mygraph.c
```

The command line responsible with building the binary file with two of the Graphitz Libraries is shown below:

```
g++ -o mygraph mygraph.o libcgraph.dll
libcdt.dll
```

It is recommended to create a new object and copy the contents of an old one in order to change or modify edges. A new object may be created like this:

```
new_g = agopen("tmp",g->desc);
```

7. Conclusions

There are tools for designing and writing the code but the ultimate tool for optimizing remains the modest compiler, this often neglected software jewel the result of hundreds

working hours by the best specialists in the world. Even though, only two compilers fulfill the needs of professional developers, a proprietary solution from a giant in the IT industry, and the Open source GNU compiler, for which we develop the AGCC lexical analyzer that helps producing even more efficient software applications. It relies on the most popular hacks and tricks used by professionals and discovered by the author who present them in this paper,

References

[1] R. Bucea-Manea-Ţoniş, R. Bucea-Manea-Ţoniş - *Integrarea aplicaţiilor Visual Basic cu SQLServer2000*, AGIR, 2006

[2] R. Bucea-Manea-Ţoniş. *Developing Object Oriented Applications with C++ and UML*, AGIR, 2008.

[3] R. Bucea-Manea-Ţoniş "Open source integrated lexical analyzer for commented code lines," *Revista Română de Informatică şi Automatică*, vol. 19, nr. 1, 2009

[4] A. Fog. *Optimizing software in C++. An optimization guide for Windows, Linux and Mac platforms*, Copenhagen University College of Engineering, 2008

[5] R. K. Irvine. *Assembly Language for Intel-based Computers*, Prentice Hall, 2006

[6] D. Spinellis. *Code Reading: The Open Source Perspective*, Addison-Wesley, 2003

[7] http://ootips.org/hungarian-notation.html - Hungarian Notation - The Good, The Bad and The Ugly

[8] K. Sugiyama, S. Tagawa, and M. Toda. "Methods for Visual Understanding of Hierarchical System Structures," *IEEE Trans. Systems, Man and Cybernetics*, SMC-11(2):109–125, February 1981.

[9] R. E. Gansner (2007, August). Drawing graphs with Graphviz, Available: http://www.graphviz.org/

Rocsana TONIS (BUCEA-MANEA) is PhD student at The Academy of Economic Studies, Bucharest, in the field of Accountings, specialization Administration Informatics, having the PhD thesis - "Decision Support System with Business Intelligence applications for Romanians' SMEs". The competitiveness fields are: developing web sites (http://csme.spiruharet.ro, http://immonline.ilive.ro/index.php), administering the data bases (Access, SqlServer2005, MySql), server scripting with PHP, ASP and client scripting with JavaScript, statistical data analyses in the marketing research using SPSS, Eview, Sphinx, implementing web marketing techniques on freeware and open-source platforms.