C *and* C++

Source Code Modification *using* CodeFix

by

Patrick Conley.

Codefix™ is a product of Abraxas Software, Inc.

Codefix was designed & written by Patrick Conley.

For more information, contact:
Abraxas Software, Inc.
Post Office Box 19586
Portland, OR 97280, USA
Phone: 503-232-0540
Fax: 503-232-0543
Email: support@abxsoft.com
Internet: www.abraxas-software.com

Table of Contents

Preface

Maintaining programs in C or C++ is a difficult task. Even experienced programmers need tools to aid in the program development process, but all too few tools exist to detect bugs in C and C++ source code and help the programmer to avoid problems.
Codefix is a powerful tool for modifying C and C++ source code. Unlike other tools, Codefix is itself fully programmable. It performs its primary task — analyzing and modifying C and C++ source code — entirely under the direction of a user-written control program.

Codefix is a powerful tool for modifying C and C++ source code. Standards and measures can be specified by the user for a tremendous number of features of C++ code that have an impact on *awareness, assessment, renovation, validation, and implementation.* Codefix is designed to enhance dramatically the effectiveness and efficiency of project management in commercial and industrial programming efforts.
A custom Codefix program specifying code standards and measures can be written by a project leader using the Codefix language (actually a restricted subset of C itself). Codefix can be programmed to:

- Analyze source code for date programming problems, includes rules for date type-encoded identifiers, proper use of date related macros and typedef's, prototypes, *etc.* Year-2000 is not the only calendar related date problem. There will be many problems in 2038 and coming leap years. CodeFix can be in the future to find and fix date problems.

- Modify code layout to improve readability. Most standards are supported for indentation and source program formatting. Generate 'pretty' C++.

- Obfuscation or shrouding of code for your distribution, yet still maintain proprietary trade secrets.

- Dynamic Testing is available by inserting assertions at locations in the target code where possible conflicts are found. Add code for run-time testing & debugging.

- Generate HTML Documentation from C/C++ programs.

- Database Generation is provided where Oracle & Microsoft databases can be generated so management can analyze computer programs from their favorite database in the form of graphics. Generate databases from C++/C source code.

- Commenting - Validating that C/C++ is correctly commented, and generating comment stubs for cases of missing comments.

I. Introduction to Source Code Modification

Since 1982 Abraxas Software has been providing language solutions for all programming languages. We first started out 'CodeCheck' development in 1986 and released the product in 1990. Since that time many people have asked us, "Why not 'fix' the problems instead of just logging them?."

It had been our feeling that the automatic modification of source code is a dangerous proposition, e.g. taking intelligent people out of the loop.

Today we have identified five main areas where CodeFix can be used to support C/C++ programmers in source code modification.

1.) Date/Calendar (Y2K) symbol identification, commenting, and correction. Both foreign and domestic calendar problems can be found.

2.) Program Layout Modification [Pretty Printing.] C/C++ programs can either be made more readable or completely modified to Hungarian naming standards.

3.) Obfuscation and/or shrouding of C++ programs for public distribution.

4.) Insertion of code for runtime testing. Dynamic testing is possible by selectively inserting check points in programs.

5.) Database generation from C/C++ source code. Automatically generate component library of source code resources in native database formats.

6.) HTML generation: Embed HTML and/or XML into source code for browser compliance and self-documentation.

7.) Automatic commenting of source code. Have comments automatically 'self describe' the source.

C++ parsing complexity

The parsing of C++ is extremely complex and we believe that given our sixteen years experience in this area we can help professional programmers solve extremely difficult problems. Today with the use of templates, namespace and other abstractions it is impossible to identify YYMMDD symbol related problems using conventional tools that are simply based on searching for the explicit symbols. For instance is the following example:

```
Class Date
{
public:
```

```
   Date ( int mon, int day, int year );     // constructor
   Int getYear()  const;
Private:
   Int month, day, year;                  // private data
}
inline int Date::getYear() const
{
   return year;
}

int retire;
Retire = Date.getYear();      //flag ‘retire’ as Date
```

In the above example most tools would not be aware that ‘retire’ is a Date type. Since CodeFix is capable of following the use of ‘Retire’ it is capable of finding even the most complex date usage problems related to C++.

II. Calendar related Symbol Identification

The core concept of date/calendar assessment using CodeFix is that of symbol identification. Symbol Identification involves several passes to acquire the needed information. The passes can be considered as - collection, documentation, analysis, and correction.

The scope of symbols can include.

1. Simple 'C' data types, like

```
int year=98.
```

Here we have the simple use of year with being initialized with a two digit date.

2. Date/Time service routines, like

```
set_this_year( (int) 98 )
```

In this case we have a time handler setting the current year to a two digit year.

3.) Sort routines, like

```
merge_table( emp_list, result_list, START_DATE, 98 )
```

Here we have a sort routine where the employee list is being merged by start_date from a two digit date.

4.) C++ complex data template types like,

```
template <class date> class employee{};
employee<date> de;
```

In this typical C++ template problem "de" for 'employee date' has been instantiated as a date type.

Collection of foreign/domestic Date related Symbols in a C++ program

Collection involves the building of a calendar-name symbol tables that the expert system 'codefix' will use in the identification process.

A simple symbol table may be thought as the following.

Begin	beg	bgn	mdy	mmddyy	mmyy
Month	mon	mo	mmm	ccyy	cyyddd
Cyyddm	cyymmdd	Curr	Current	date	Day

Figure 1.

As shown in figure 1, we have a collection of symbols, which in effect are just strings of common Year-2000 related names. These are string known to represent time/date information and experience has shown that these are the typical names that programmers have used historically for time/date data types.

The problem of course is that not all information representing time/date information uses these name combinations. History has shown that not only do programmers not use meaningful names in their programs, they may even use the name of their cat to represent the day of year!

The collection phase of CodeFix is to build the symbol table, e.g. build a list of strings that by context represents date and/or highly likely may represent dates, by the context of the program using expert system technology and advanced parsing techniques.

Provided Symbol Tables

Codefix comes with several pre-built symbol tables for checking source code, they include -

1.) Simple symbols

> Simple symbols containing the classic symbols usually provided by most YEAR2000 documentation, it includes about two dozen of the most common Symbols used in programming for dates.

2.) Advanced symbols

This example is from a large suite of 'date' related public code samples, this Symbol table provides hundreds of symbols used for representing dates in the industry.

3.) Foreign symbols

This example provides a large set of strings used in providing computation for Worldwide calendar sets outside of the USA.

Collection Phase

This section will discuss how to build your own CodeFix symbol table.

The generation of symbol tables requires the extraction of symbols a large set of C/C++ which is know to contain 'date' related computation in your organization. Most likely after the generation of the initial symbol table some pruning will be required to reject base types that are not considered date related.

For example,

```
Check -rcollect.cc datecode.c
```

In the above example, expert system rule script symbol.cc contains the 'rules' for building the symbol table from the known date code in the example datecode.c.

The results will be written to the file symbol.tmp, by default. If a name other than symbol.tmp is desired then the file symbol.cc must be modified.

If there is more than one file to be included, the wild card option (*) may be used before the dot-c suffix.

A symbol collection example

What follows is a simple 'C' example of symbol table generation.

```
1: typedef struct DATE_INFO
  2: {
  3:    int year;
  4:    int month;
  5:    int day;
  6: } DI;
  7: DI dtglo;                  // global
  8: enum DATE { year=1900 };
  9: // simple 'date' example
 10: struct DATE_INFO bridge_2to4 (struct DATE_INFO *date)
 11: {
 12:    int y2, y4;
```

```
13:    struct DATE_INFO ywd;   // local
14:
15:    y2 = date->year;
16:
17:    if ( y2 > 49 )
18:    y4 = y2 + 1900;
19:    else
20:            y4 = y2 + 2000;
21:
22:    ywd.year = y4; // local usage
23:
24:    dtglo.year = y4; // global usage
25:
26:    return (ywd);
27: }
```

For the above case the initial generated symbol table would appear as follows.

```
DM year 2 DATE_INFO      // data member 'year' from line 3
DM month 2 DATE_INFO
DM day 2 DATE_INFO
GT DATE_INFO 3           // GT - Global Tag 'DATE_INFO'
GD DI 26 26
GD dtglo 28 26           // GD - Global Definition
ED year 6                // Enum
GT DATE 1
LD date  26 26           // Local Definition
FD bridge_2to4 26 0      // function definition
LD y2 bridge_2to4 6 6
LD y4 bridge_2to4 6 6
LD ywd bridge_2to4 26 26
```

- note file name info and line number, author, other information is kept internally.

The initial pass of CodeFix for Y2K collection is the generation of this intermediate symbol table containing the base information on all symbols defined. The constants shown can be found in the appendix and contain type information both current and base for complex types.

The next step is determination of whether a date usage is found in the source example.

Identifying Code Using Date Name Criteria

Given input source code in the form of a singled file or complete project including many files a symbol table is generated as shown in the previous section.

The basic concept of identification is finding all use of symbols that meet the criteria of the 'Y2K' keyword list, and then generating a subset symbol table of the original definitions meeting those criteria.

What follows is an intermediate form of the source example in this section. Where the first character identifies source origination.

Where first character in record means:

* The file name
- Source Header File (This data is not emitted to the final output)
+ Source from file

Note that before all usage of date symbols there is inserted code identified by the control string '$DATE$', all symbols defined that meet the criteria of Y2K keywords are marked for there usage, prior to use. This intermediate step helps identify potential Y2K usage of all symbols.

```
*//$DATE$ MN b.c
+
+#include "b.h"
+
+DI dtglo; // global
+
+// simple 'date' example
+struct DATE_INFO bridge_2to4 (struct DATE_INFO *date)
+{
+   int y2, y4;
+   struct DATE_INFO ywd;       // local
+
-//$DATE$ IL y2 int 6
-//$DATE$ IL date DATE_INFO 26
+   y2 = date->year;
+
-//$DATE$ IL y2 int 6
+   if ( y2 > 49 )
-//$DATE$ IL y4 int 6
-//$DATE$ IL y2 int 6
+       y4 = y2 + 1900;
+   else
-//$DATE$ IL y4 int 6
-//$DATE$ IL y2 int 6
+       y4 = y2 + 2000;
+
-//$DATE$ IL ywd DATE_INFO 26
-//$DATE$ IL y4 int 6
+   ywd.year = y4; // local usage
+
-//$DATE$ IG dtglo DI 28
-//$DATE$ IL y4 int 6
+   dtglo.year = y4; // global usage
+
-//$DATE$ IL ywd DATE_INFO 26
```

```
+   return (ywd);
+}
```

Automatic Detection of Date Symbols in C/C++

Using our original example, from the previous section we now have collected the symbols, and reduced them to the subset that are candidates for Y2K.

In this section we have emitted the original source with candidates documented.

```
1: typedef struct DATE_INFO
 2: {
 3:    int year;
 4:    int month;
 5:    int day;
 6: } DI;
 7: DI dtglo;                    // global
 8: enum DATE { year=1900 };
 9: // simple 'date' example
10: struct DATE_INFO bridge_2to4 (struct DATE_INFO *date)
11: {
12:    int y2, y4;
13:    struct DATE_INFO ywd;  // local
14:
15: //$DATE$ IL date DATE_INFO 26        // 26 - means 'struct' base type
16:    y2 = date->year;
17:
18:    if ( y2 > 49 )
19:   y4 = y2 + 1900;
20:    else
21:          y4 = y2 + 2000;
22:
23:    ywd.year = y4; // local usage
24:
25: //$DATE$ IG dtglo DI 28                // 28 means defined type by 'typedef'
26:    dtglo.year = y4; // global usage
27:
28:    return (ywd);
29: }
```

Note on lines 15 & 25 we have marked our commented control string with "$DATE$. Following the control string we have the name of the symbol, followed the base name of the object that defined the symbol, followed by the base type. The constants are defined in the appendix.

Obviously this example is very simple were only matching those types that are explicitly declared as having the date keyword in the symbol name. However we could have included scope to that of the parent type, or even in the case of the assignment we could consider the type on the left-value (lvalue), e.g. the type to the left of the equal sign.

Automatic Code Modification for Date Programming Problems

Finally were at the goal of our problem. Lets use a more simple case here.

III. Program Layout Modification.

There are many standards for readability. Codefix provides templates for the three most common formats to automatically be applied to you C/C++ software.

Since the script sources are provided for the layout modification any type of formatting can be applied to your source code.

IV. Obfuscation and/or shrouding of C++ programs

In selling or distributing software in today's marketplace it is essential to support all computer platforms. Given the large number of computer and operating system combinations its is not possible for even the largest corporation to ship binary software for all platforms. Given the portability of C/C++ the shipment of source code is sometime the only solution. CodeFix will apply the highest levels of code Obfuscation to your source code so that you can deliver to your customer with no loss of trade secrets. The generated source code while compilable is not meaningful to a recipient.

V. Dynamic Testing: Insertion of code for runtime testing.

While the principal use of CodeFix is on that of static analysis, it is possible to apply the notion of runtime dynamic analysis.

A simple example will be provided in the case of the Year-2000 problem.

```
Int date;

Int year_4, year_2;

year_4 = year_2 + 1900;

the bridge patch replaces the line with code such as:

   if (year_2 > 49)
      year_4 = year_2 + 1900;
   else
      year_4 = year_2 + 2000;
```

In the above case it would be desirable for instance to find all symbols that use that are determined to represent a date and then assert that the dates are always in the YYYY format.

In this case CodeFix would insert the following assertions prior to use in all cases of use the of the symbols 'year_2' and 'year_4'

Where 'ASSERT_YYYY' is defined as -

```
#define ASSERT_YYYY(x) ( if ( !x )   fprintf( testfp, "Invalid YYYY
usage in file=%s at line=%d\n", FILE, LINE )
```

CodeFix can automatically add the the assertion as above in all cases where a symbol is determined to be and illegal date.

From the above case the included assertion into the source code would appear as follows.

```
ASSERT_YYYY( year_4 > 1900 );
ASSERT_YYYY( year_2 > 1900 );
year_4 = year_2 + 1900;
```

As shown the assertions are inserted prior to use in the generated source code. The code is then compilied and linked and at runtime if there is case where the assertion fails then the message is written to the file channel testfp (FILE * testfp),where this may be a database for future analysis.

Codefix can add such assertions as detailed above for both simple (int) and complex (class, template) data types.

VI. Database generation from C/C++ source code.

Quite often it is simply impossible to analyze the results from source code analysis products because of huge quantity of information. In this case it is essential to take a set of source code and generate a database that is compatible with Oracle or Microsoft Access so that management and/or programmers can analyze the results of source code analysis.

```
year_4 = year_2 + 1900;
```

In this case whenever Y2K symbols are found the information is written to a file in 80 byte card image format.

Year_4	int	file.c	l	simple

In this case the generated record would appear as above, where the data would contain the symbol name, the type, the file name, the line number, and the scope of the symbol. When this analysis is done on large body of code it is possible for a generic database to provide graphics and even keep track of all relevant information from the source including extracting information from the comments.

VII. HTML Generation from C/C++

In this section we discuss automatically generating web documentation from C/C++, e.g. generating HTML from C/C++ to be used by an internet browser.

VIII. Comment Analysis & Generation

This section will discuss the requirements of source code commenting, validating, and generating correct comment blocks for documentation.

IX. Generating Pre-Compiled Source Program

This Section will discuss generating pre-compiled translation units. A translation unit is one major source file with all #include<>'s and macros expanded. Often for performance reason and/or security individual system header files must be collapsed to one file rather than hundreds. While it is common for compilers to offer 'pre-processing' on source modules [unix cc –E, dos cl –E] system compilers to not macro expand and collapse individual header files. For instance #include<windows.h> in Microsoft systems references over 300,000 lines of code and #include's 100's of source files, so pre-compiling this header file can drastically improve compilation performance.

X. References

Index