International Journal of Multimedia and Ubiquitous Engineering
Vol. 7, No. 4, October, 2012

A Study on Verification and Analysis of Symbol Tables for Development of the C++ Compiler

YangSun Lee[1] and YunSik Son[2]*

[1]*Dept. of Computer Engineering, Seokyeong University*
16-1 Jungneung-Dong, Sungbuk-Ku, Seoul 136-704, Korea

[2]*Dept. of Computer Engineering, Dongguk University*
26 3-Ga Phil-Dong, Jung-Gu, Seoul 100-715, Korea

*yslee@skuniv.ac.kr, *Corresponding Author: sonbug@dongguk.edu*

Abstract

The existing C++ compilers are designed to translate C++ source programs into target codes and then execute them. This translation method is that a compiler that translates C++ source programs to target codes has to be available for each platform. Reusability and portability of codes will also decrease because target codes have to be different for each platform they are run under. For that reason, much research is taking place in various fields in an effort to develop a retargetable compiler and a virtual machine that execute application programs without recompiling or modifying them though processor or operating systems are changed.

We have developed the C++ compiler for the SVM(Smart Virtual Machine) of the smart platform on smart systems. As a part of the C++ compiler development, we designed the symbol table that can support object-oriented languages, C++ and java. The symbol table is a data structure to keep track of scope and binding information about names (or identifiers). Various information is entered into the symbol table after visiting and analyzing the abstract syntax tree generated by syntax-directed translation, and then is used to check whether the use of names is consistent with their definition during the semantic analysis phase and generate a valid code during the code generation phase.

In this paper, we describe the reconstruction technique for verifying and analyzing the symbol table designed for the C++ compiler. This system reconstructs inputted C++ declarations by using information of the symbol table entered in the declaration process phase of the C++ compiler, and therefore we can verify completeness of symbol table design and correctness of information entered in the symbol table. In addition, this system also produces debug information, and so is effectively utilized for the development of the C++ compiler.

Keywords: *C++ Compiler, SVM(Smart Virtual Machine), Symbol Table, Reverse Translator*

1. Introduction

The existing C++ compilers are designed to translate C++ source programs into target codes and then execute them. This translation method is that a compiler that translates C++ source programs to target codes has to be available for each platform. Reusability and portability of codes will also decrease because target codes have to be different for each platform they are run under. For that reason, much research is taking place in various fields in an effort to develop a retargetable compiler and a virtual machine that execute application programs without recompiling or modifying them though processor or operating systems are changed [1-5].

A virtual machine is a conceptual computer with a logical system configuration, made of software unlike physical systems made of hardware. In particular, virtual machine technology for smart systems, as a core technology enabling virtual machines to be mounted on smart devices such as smart phones and smart tablets, is a requisite software technology for download solutions [6-9]. We developed the SVM (Smart Virtual Machine) for the smart platform, which is a virtual machine for smart systems in which the contents can be run without correction under different platforms, and a C++ compiler for the SVM. In this paper, we present the reverse translator that restores the attributes inputted into the symbol table to the original program in order to verify and analyze the symbol table designed in the C++ compiler development phase [5, 10-12].

2. Related Studies

2.1. SVM

The SVM(Smart Virtual Machine) is a platform that is loaded on smart phones. It is a stack-based virtual machine solution that can independently download and run application programs. The SVM consists of three main parts: compilers, an assembler and a virtual machine. Figure 1 shows the structure of the SVM system for the smart platform [15-17].

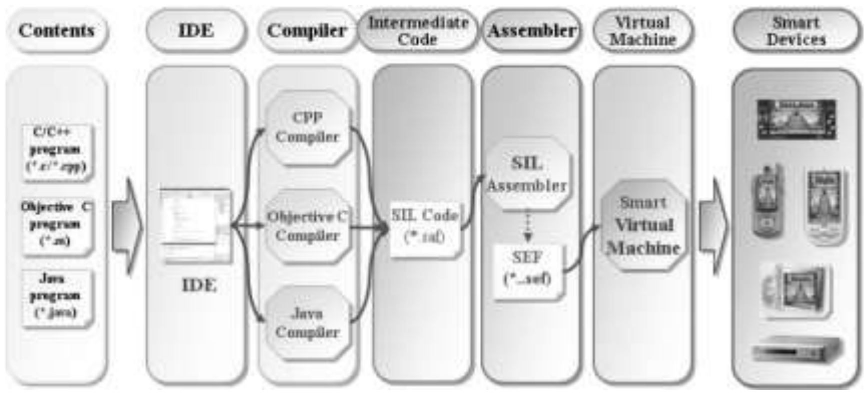


Figure 1. SVM System Configuration for Smart Systems

The SVM system can be distinguished into a compiler part, a translator part, an assembler part, and a virtual machine part. In the compiler part, a program coded in a high-level language, such as C/C++, Objective-C or Java, can be translated to a stack-based intermediate language (*.sil) for virtual machines. The assembler part takes the SIL (Smart Intermediate Language) code as input and translates it to *.sef (sil executable format) file, which can then be executed in virtual machines. Lastly, the virtual machine part, which is attached to different hardware, executes *.sef files.

2.2. Symbol Table

The symbol table is a data structure to be used to keep and manage track of scope and biding information about names (or identifiers). Lexical and syntax analysis takes place in the compiler, then the abstract syntax tree generated by the SDT (Syntax-directed Translation) is analyzed to collect and reference the attributes of the recognized names. These attributes are inserted in a symbol table where the names are defined and the validity of the use of these names and attributes is verified in the semantic analysis phase. After this, the code generation phase produces the correct code using the attributes [18-22].

The symbol table designed in the C/C++ compiler divided into window table and sub-tables. Window tables consist of symbol table, type table. And sub-tables consist of a concrete table, an abstract table, an aggregate table, an aggregate table, a member table, a link table, and a template table. Figure 2 shows the relationships between the composed symbol tables and each table.

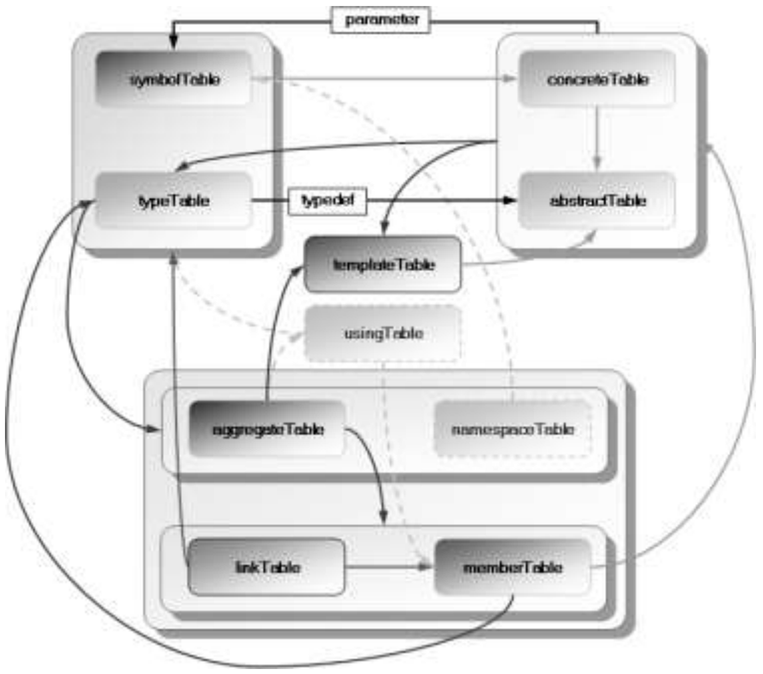


Figure 2. Relationships between Symbol Tables

The symbol table stores the offset and scope of the identifiers of variables and functions declared in the declaration part. The type table stores the size, the type, and the identifiers of user-defined types such as class, struct, and union. The storage table is divided into a concrete table and an abstract table. These are commonly used for storing the variable type, the function return type, and the parameter information. Unlike the abstract table, the concrete table is allocated additional memory space for storing the initial value.

The user-defined type table is composed of an aggregate table, a member table, and a link table. These tables hold the information on inheritance, the number of members, the name of member variables, the offset, and access control.

The template table, a space for storing template parameter variables used in function templates and class templates, is a table for storing information that is required in the specialization process, the initial value, and the information of each parameter variable.

2.3. Declaration Processor

The declaration processor is responsible for semantically analyzing the declaration part of an inputted program, for inserting the information on the recognized symbols (or identifiers) into the symbol table if the statements are valid, and for generating an error message if the statements are invalid. The overall organization of the declaration processor is divided into a preprocessor, a lexical analyzer, a syntax analyzer, and a declaration processor [10, 11, 20, 22]. Figure 3 shows the organization of the declaration processor of the C++ compiler.

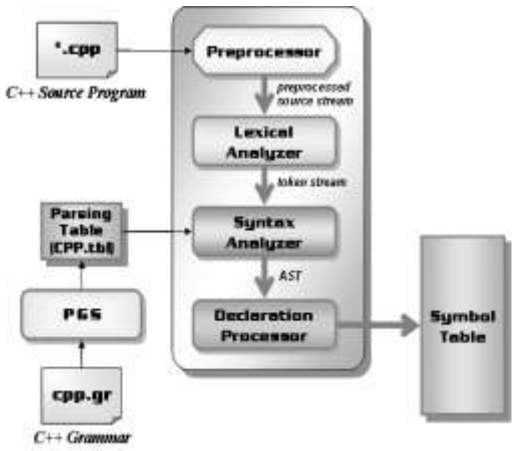


Figure 3. System Structure of the Declaration Processor

3. The Translator for Verification and Analysis of Symbol Tables

3.1. An Outline of the Reverse Translator System

The reverse translator (detranslator) inserts the declaration part extracted from C++ (*.cpp) files into the symbol table and uses the information to restore them to the C++ declaration part. First, the C++ declaration part, which contain declaration processor, are semantically analyzed and the analyzed properties of the symbols are stored in the memory space, or the symbol table. The reverse translator then restores the stored information back to C++ declaration statements. Thus, a result similar to declaration statements composed for primary input can be obtained [10, 11]. Figure 4 shows the reverse translation for verification and analysis of symbol tables.

3.2. Organization of the Reverse Translator System

This reverse translator system is implemented using Visual C++ 6.0 in a Windows XP environment and it translates to the original C++ program by using information on the variables inserted into the symbol table, or the user-defined type.

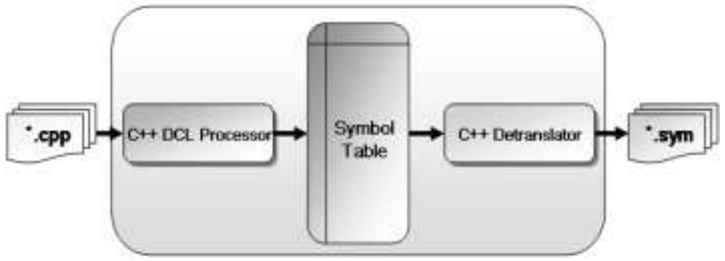


Figure 4. Reverse Translation Process of the DCL Part

The reverse translator system is composed of a symbol table, a symbol process that uses the symbol table to process a reverse translation, and a type process. Figure 5 is the system organization of the reverse translator.

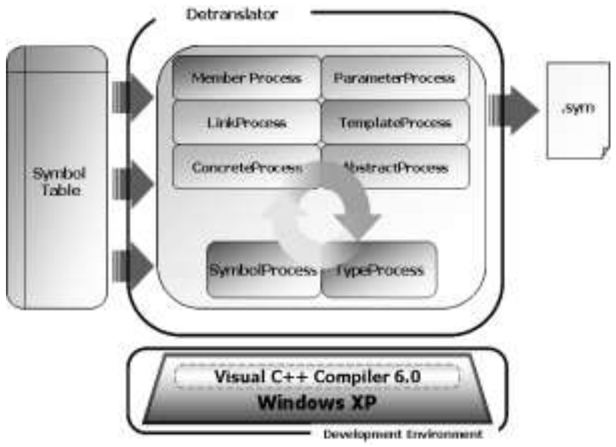
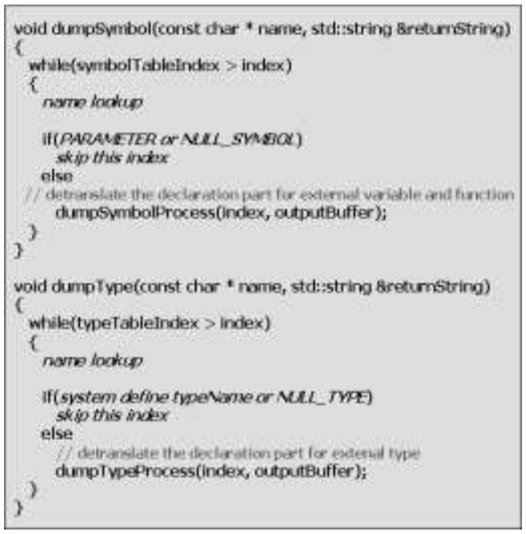


Figure 5. Organization of the Reverse Translator

The symbol process and the type process, according to the declared variable or type, cycles through member process, which is a sub process, parameter process, link process, template process, type specifier process(TSP), and symbol kind process(SKP) to perform a reverse translation.

3.3. Implementation of the Reverse Translator System

The reverse translation system is mainly subcategorized into a symbol process and type process. It distinguishes the tables that need to be translated by sequentially searching the symbol table and the type table. Moreover, the sub table is referenced by the table index and the information is used to translate to a C++ program. Figure 6 is an outline of the algorithm that forms the reverse translation system.



```
void dumpSymbol(const char * name, std::string &returnString)
{
  while(symbolTableIndex > index)
  {
    name lookup

    if(PARAMETER or NULL_SYMBOL)
      skip this index
    else
  // detranslate the declaration part for external variable and function
      dumpSymbolProcess(index, outputBuffer);
  }
}

void dumpType(const char * name, std::string &returnString)
{
  while(typeTableIndex > index)
  {
    name lookup

    if(system define typeName or NULL_TYPE)
      skip this index
    else
      // detranslate the declaration part for external type
      dumpTypeProcess(index, outputBuffer);
  }
}
```

Figure 6. Outline of a Reverse Translation Algorithm

The symbol process is the part that translates reversely all the information corresponding to the external variables in the symbol table. The extracted information details the variable type through the TSP and determines, through the SKP, if it is an array, a function, or a regular variable, and then outputs it after translating it to a C++ program. A parameter process is added for functions with parameters, and the reversely translated parameters are outputted. Figure 7 shows the flow of the symbol process.

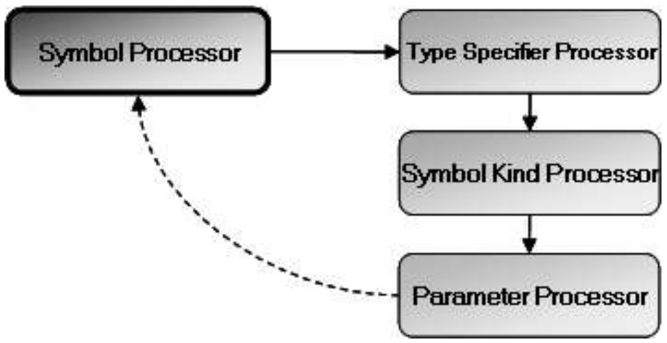


Figure 7. Flow of the Symbol Process

The type process is composed of a member process, a link process, a TSP, and a SKP. It searches the user-defined type information in the type table to classify the information into class, struct, union, enum, namespace, and typedef for the reverse translation. The member process is performed when the type is class, struct, union, and namespace, and is similar to the symbol process in that it outputs after translating the member variables and member functions included in the type through the TSP, the SKP, and the parameter process. Moreover, the user-defined type can have a nested class as its member; in order to do this, the member process: self-references the type process to translate it reversely. The link process then translates reversely and outputs the inherited and friends if the type is class and struct. Figure 8 is the flow of the type process.

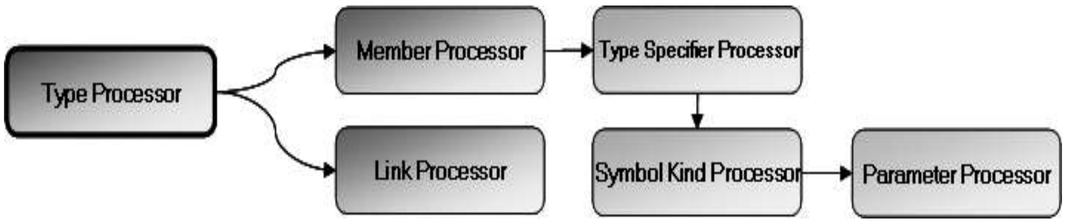


Figure 8. Flow of the Type Process

The template process is responsible for translating templates reversely, one of the C++ features, and the template function calls the template process in the symbol process and the template class calls the template process in the type process and translates it reversely. Figure 9 is the flow of the template process.

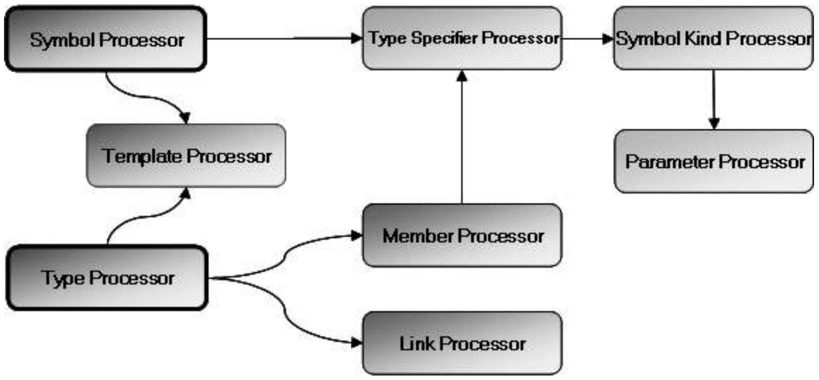


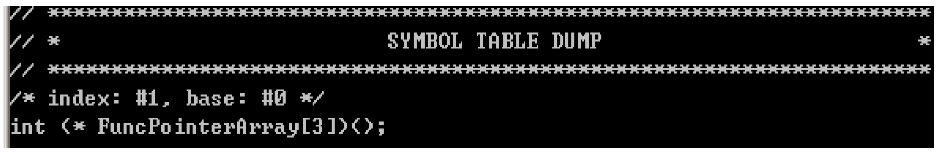
Figure 9. Flow of the Template Process

4. Experimental Results and Analysis

The following Figure 10, 11, and 12 is a function pointer array example of a program that inserts the attributes of the declarations into a symbol table and translates reversely in order to restore it back to a C++ program.

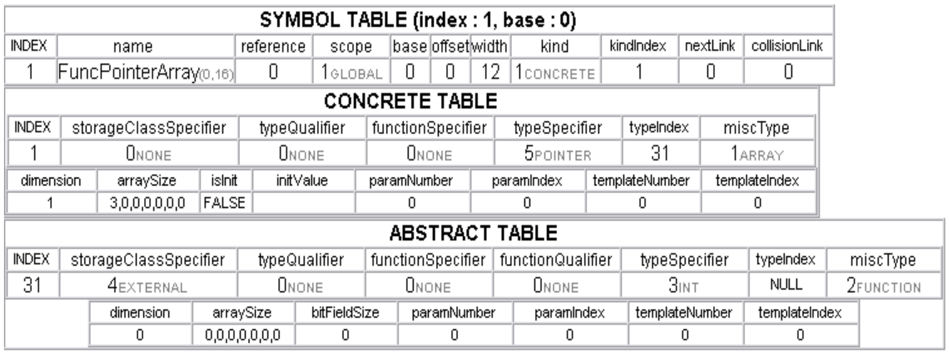
```
int (*FuncPointerArray[3])();
```

Figure 10. Declaration Part of a Function Pointer Array



```
// ***************************************************************************
// *                          SYMBOL TABLE DUMP                              *
// ***************************************************************************
/* index: #1, base: #0 */
int (* FuncPointerArray[3])();
```

Figure 11. Reconstructed C++ Declaration Part



SYMBOL TABLE (index : 1, base : 0)

INDEX	name	reference	scope	base	offset	width	kind	kindIndex	nextLink	collisionLink
1	FuncPointerArray(0,16)	0	1GLOBAL	0	0	12	1CONCRETE	1	0	0

CONCRETE TABLE

INDEX	storageClassSpecifier	typeQualifier	functionSpecifier	typeSpecifier	typeIndex	miscType
1	0NONE	0NONE	0NONE	5POINTER	31	1ARRAY

dimension	arraySize	isInit	initValue	paramNumber	paramIndex	templateNumber	templateIndex
1	3,0,0,0,0,0,0	FALSE		0	0	0	0

ABSTRACT TABLE

INDEX	storageClassSpecifier	typeQualifier	functionSpecifier	functionQualifier	typeSpecifier	typeIndex	miscType
31	4EXTERNAL	0NONE	0NONE	0NONE	3INT	NULL	2FUNCTION

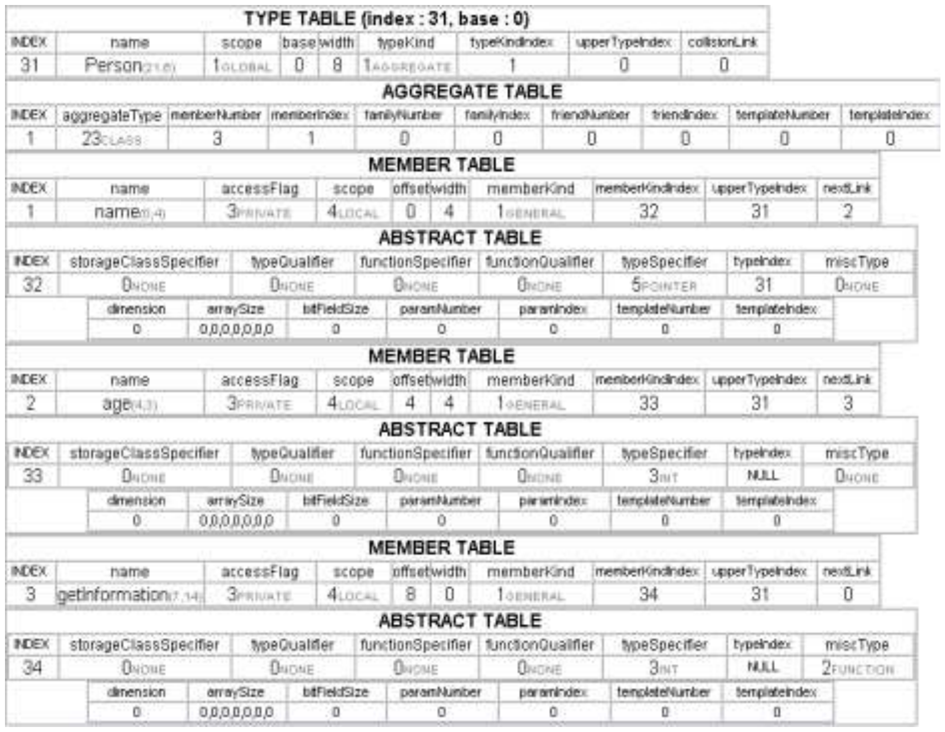
dimension	arraySize	bitFieldSize	paramNumber	paramIndex	templateNumber	templateIndex
0	0,0,0,0,0,0,0	0	0	0	0	0

Figure 12. Information of Symbol Tables

The following Figure 13, 14, and 15 is a class example.

```
class Person {
        char * name;
        int age;
        int getInformation();
 };
```

Figure 13. Declaration Part of a Class



TYPE TABLE (index : 31, base : 0)

INDEX	name	scope	base	width	typeKind	typeKindIndex	upperTypeIndex	collisionLink
31	Person$_{(21,6)}$	1$_{GLOBAL}$	0	8	1$_{AGGREGATE}$	1	0	0

AGGREGATE TABLE

INDEX	aggregateType	memberNumber	memberIndex	familyNumber	familyIndex	friendNumber	friendIndex	templateNumber	templateIndex
1	23$_{CLASS}$	3	1	0	0	0	0	0	0

MEMBER TABLE

INDEX	name	accessFlag	scope	offset	width	memberKind	memberKindIndex	upperTypeIndex	nextLink
1	name$_{(0,4)}$	3$_{PRIVATE}$	4$_{LOCAL}$	0	4	1$_{GENERAL}$	32	31	2

ABSTRACT TABLE

INDEX	storageClassSpecifier	typeQualifier	functionSpecifier	functionQualifier	typeSpecifier	typeIndex	miscType
32	0$_{NONE}$	0$_{NONE}$	0$_{NONE}$	0$_{NONE}$	5$_{POINTER}$	31	0$_{NONE}$

dimension	arraySize	bitFieldSize	paramNumber	paramIndex	templateNumber	templateIndex
0	0,0,0,0,0,0,0	0	0	0	0	0

MEMBER TABLE

INDEX	name	accessFlag	scope	offset	width	memberKind	memberKindIndex	upperTypeIndex	nextLink
2	age$_{(4,3)}$	3$_{PRIVATE}$	4$_{LOCAL}$	4	4	1$_{GENERAL}$	33	31	3

ABSTRACT TABLE

INDEX	storageClassSpecifier	typeQualifier	functionSpecifier	functionQualifier	typeSpecifier	typeIndex	miscType
33	0$_{NONE}$	0$_{NONE}$	0$_{NONE}$	0$_{NONE}$	3$_{INT}$	NULL	0$_{NONE}$

dimension	arraySize	bitFieldSize	paramNumber	paramIndex	templateNumber	templateIndex
0	0,0,0,0,0,0,0	0	0	0	0	0

MEMBER TABLE

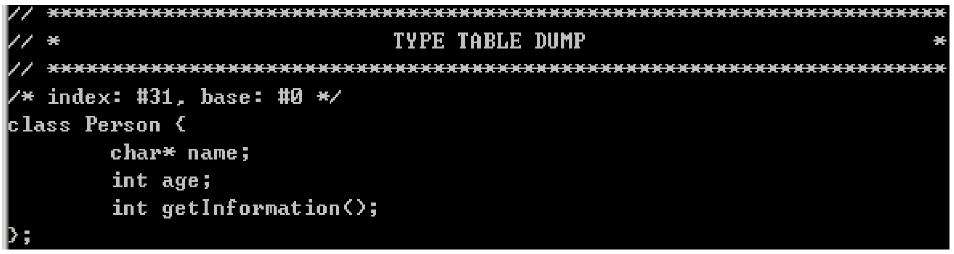
INDEX	name	accessFlag	scope	offset	width	memberKind	memberKindIndex	upperTypeIndex	nextLink
3	getInformation$_{(7,14)}$	3$_{PRIVATE}$	4$_{LOCAL}$	8	0	1$_{GENERAL}$	34	31	0

ABSTRACT TABLE

INDEX	storageClassSpecifier	typeQualifier	functionSpecifier	functionQualifier	typeSpecifier	typeIndex	miscType
34	0$_{NONE}$	0$_{NONE}$	0$_{NONE}$	0$_{NONE}$	3$_{INT}$	NULL	2$_{FUNCTION}$

dimension	arraySize	bitFieldSize	paramNumber	paramIndex	templateNumber	templateIndex
0	0,0,0,0,0,0,0	0	0	0	0	0

Figure 14. Information of Symbol Tables



```
// ****************************************************************
// *                      TYPE TABLE DUMP                         *
// ****************************************************************
/* index: #31, base: #0 */
class Person {
        char* name;
        int age;
        int getInformation();
};
```

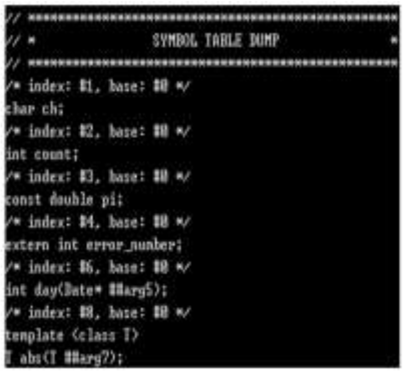
Figure 15. Reconstructed C++ Declaration Part

The following Figure 16 is an example of a program that inserts the attributes of the declarations into a symbol table and translates reversely in order to restore it back to a C++ program.

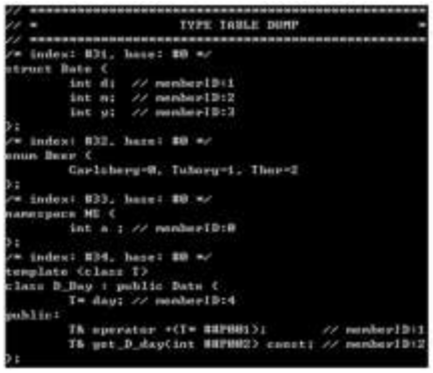
char ch;   int count;   const double pi;
extern int error_number;
struct Date { int d,m,y; };
int day(Date*p);
template<class T> T abs(T a);
enum Beer { Carlsberg, Tuborg, Thor };
namespace NS { int a; }
template<class T>
	class D_Day : public Date
	{
	    T* day;
	  public:
	    T& operator+(T*);
	    T& get_D_day(int) const;
	}

Figure 16. Declaration Part in the C++ Program

Figure 17 shows the output of the debug information, along with the restored C++ program's declaration part, using the symbol table attributes for a reverse translation.



```
// ************************************************
// *              SYMBOL TABLE DUMP               *
// ************************************************
/* index: #1, base: #0 */
char ch;
/* index: #2, base: #0 */
int count;
/* index: #3, base: #0 */
const double pi;
/* index: #4, base: #0 */
extern int error_number;
/* index: #6, base: #0 */
int day(Date* ##arg5);
/* index: #8, base: #0 */
template <class T>
T abs(T ##arg7);
```



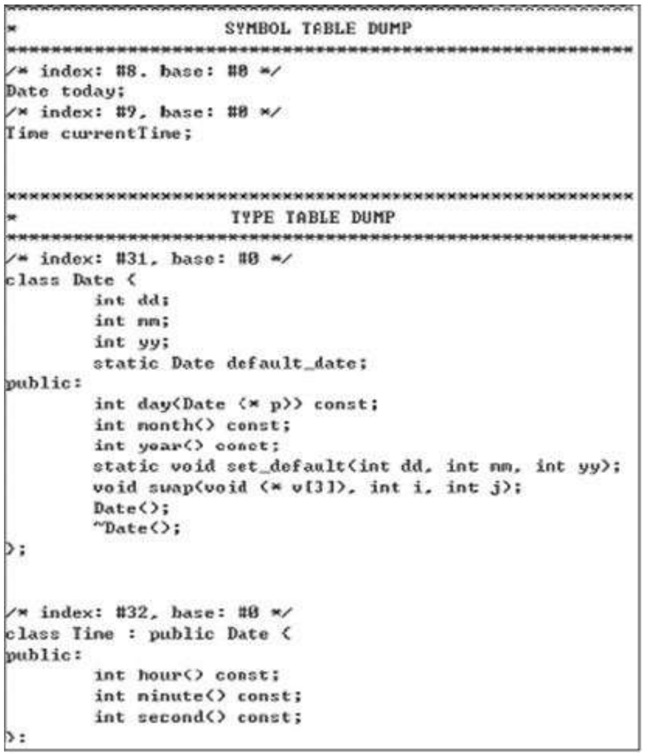
```
// ************************************************
// *               TYPE TABLE DUMP                *
// ************************************************
/* index: #31, base: #0 */
struct Date {
        int d;   // memberID:1
        int m;   // memberID:2
        int y;   // memberID:3
};
/* index: #32, base: #0 */
enum Beer {
        Carlsberg=0, Tuborg=1, Thor=2
};
/* index: #33, base: #0 */
namespace NS {
        int a ; // memberID:0
};
/* index: #34, base: #0 */
template <class T>
class D_Day : public Date {
        T* day; // memberID:4
public:
        T& operator +(T* ##PH01);        // memberID:1
        T& get_D_day(int ##PH02) const; // memberID:2
};
```

Figure 17. Reconstructed C++ Declaration Part

Figure 18 is a class example, and Figure 19 shows the output of the restored C++ class's declaration part.

```
class Date {
private:
    int dd, mm, yy;
    static Date default_date;
public:
    int day(Date * p) const;
    int month() const;
    int year() const;
    static void set_default(int dd, int mm, int yy);
    void swap(void *v[3], int i, int j);
    Date();
     ~Date();
 };
class Time : public Date{
 public:
     int hour() const;
     int minute() const;
     int second() const;
 };
 Date today;
 Time currentTime;
```

Figure 18. C++ Class Declaration Part



```
*                       SYMBOL TABLE DUMP
****************************************************************
/* index: #8. base: #0 */
Date today;
/* index: #9. base: #0 */
Time currentTime;


****************************************************************
*                        TYPE TABLE DUMP
****************************************************************
/* index: #31. base: #0 */
class Date {
        int dd;
        int mm;
        int yy;
        static Date default_date;
public:
        int day(Date (* p)) const;
        int month() const;
        int year() const;
        static void set_default(int dd, int mm, int yy);
        void swap(void (* v[3]), int i, int j);
        Date();
        ~Date();
};


/* index: #32. base: #0 */
class Time : public Date {
public:
        int hour() const;
        int minute() const;
        int second() const;
};
```

Figure 19. Reconstructed C++ Class Part

5. Conclusions

This paper covers the design and implementation of a reverse translator that verifies and analyzes the symbol table designed during the development stage of the C++ compiler. The reverse translator's role is to restore, using only the information in the symbol table, back to the original program. Therefore, it is possible to analyze and verify the completeness of the designed symbol table and the information on the identifiers stored in the symbol table.

Moreover, based on the verified symbol table, a correct code can be generated by examining the use of the referenced identifiers and the attributes of the stored identifiers in the code generation stage. Furthermore, in addition to translating programs reversely, the correcting of the C++ compiler became easy by outputting the debug information. The reverse translator is being expanded to accommodate java, another object-oriented language, and it is anticipated that it will output even more debugging information using the symbol table information.

Acknowlegements

This research was supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education, Science and Technology(No. 20110006884).

References

[1] A. V. Aho, R. Sethi and J. D. Ullman, “Compilers: Principles, Techniques and Tools”, Addison Wesley, **(1986)**.
[2] D. Galles, “Modern Compiler Design”, Addison-Wesley, **(2004)**.
[3] C. Fraser and D. Hanson, “A Retargetable C Compiler: Design and Implementation”, Addison Wesley, **(1995)**.
[4] B. Stroustrup, “The C++ Programming Language”, Addison-Wesley, **(2000)**.
[5] International Standard ISO/IEC 14882:1998(E) Programming Language - C++, ISO/IEC, **(1998)**.
[6] B. Venners, “Inside the JAVA Virtual Machine”, 2nd ed., McGraw-Hill, **(2000)**.
[7] J. E. Smith and R. Nair, “Virtual Machines”, Morgan Kaufmann, **(2005)**.
[8] T. Lindholm and F. Yellin, “The Javatm Virtual Machine Specification”, 2nd ed., Addison Wesley, **(1999)**.
[9] S. M. Oh, Y. S. Lee and K. M. Ko, "Design and Implementation of a Virtual Machine for Embedded Systems", Journal of Korea Multimedia Society, vol. 8, no. 1282, **(2004)**.
[10] H. J. Kwon, Y. K. Kim, J. K. Park and Y. S. Lee, "Development of C Program Detranslator from Symbol Table for ANSI C Compiler", Proceedings of Korea Multimedia Society, vol. 8, no. 69, **(2005)**.
[11] M. S. Son, H. J. Kwon, Y. K. Kim and Y. S. Lee, "The Declarations Reconstruction Technique for the Symbol Table Verification of the Object-oriented Compiler", Proceedings of Korea Multimedia Society, vol. 13, no. 669, **(2006)**.
[12] Y. S. Lee, Y. K. Kim and H. J. Kwon, “Design and Implementation of the Decompiler for Virtual Machine Code of the C++ Compiler in the Ubiquitous Game Platform”, LNCS 4413, Springer, **(2007)**, pp. 511.
[13] Y. S. Lee and S. W. Na, "Java Bytecode-to-.NET MSIL Translator for Construction of Platform Independent Information Systems", Springer, LNAI 3215, **(2004)**, pp. 726.
[14] Y. S. Lee, S. W. Na and D. H. Whang, "Intermediate Language Translator for Execution of Java Programs in .NET Platform", Journal of Korea Multimedia Society, vol. 7, no. 6, **(2004)**, pp. 824-831.
[15] Y. S. Son and Y. S. Lee, “Design and Implementation of an Objective-C Compiler for the Virtual Machine on Smart Phone, Multimedia”, Computer Graphics and Broadcasting, CCIS, vol. 262, Springer, (2011), pp. 52.
[16] Y. S. Son and Y. S. Lee, "The Semantic Analysis Using Tree Transformation on the Objective-C Compiler", Multimedia, Computer Graphics and Broadcasting, CCIS, Springer, vol. 262, (2011), pp. 60-68.
[17] Y. S. Lee and Y. S. Son, "A Study on the WIPI-to-Windows Mobile Game Contents Converter using a Resource Converter and a Platform Mapping Engine", Advanced Science Letters, vol. 5, no. 3, to be published, Amer Scientific Publishers, **(2012)**.
[18] R. P. Cook and T. J. Leblanc, "Symbol Table Abstraction to Implement Languages with Explicit Scope Control", IEEE Transactions on Software Engineering, vol. 9, no. 8, **(1983)**.
[19] S. C. Dewhurst, "Flexible Symbol Table Structures for Compiling C plus plus", Software-Practice and Experience, vol. 17, **(1987)**, pp. 503.
[20] M. Gallego-Carrillo, F. Gortázar-Bellas, J. Urquiza-Fuentes and J. Á. Velázquez-Iturbide, "SOTA: a Visualization Tool for Symbol Tables", ACM SIGCSE Bulletin, vol. 37, **(2005)**, pp. 385.
[21] R. Meyers, "The New C: Declarations and Initializations", C/C++ Users Journal, vol. 19, **(2001)**, pp. 56.
[22] J. F. Power and B. A. Malloy, "Symbol Table Construction and Name Lookup in ISO C++", the Conference on Technology of Object-Oriented Languages and Systems, TOOLS, TOOLS-PACIFIC, **(2000)**, pp. 57.
[23] W. Alouini, O. Guedhami, S. Hammoudi, M. Gammoudi, and D. Lopes, "Semi-Automatic Generation of Transformation Rules in Model Driven Engineering: The Challenge and First Steps", International Journal of Software Engineering and Its Applications, vol. 5, SERSC, **(2011)**, pp. 77.
[24] P. G. Vijayrajan, "Analysis of Performance in the Virtual Machines Environment", International Journal of Software Engineering and Its Applications, vol. 32, SERSC, **(2011)**, pp. 53.

Authors

YangSun Lee

He received the B.S. degree from the Dept. of Computer Science, Dongguk University, Seoul, Korea, in 1985, and M.S. and Ph.D. degrees from Dept. of Computer Engineering, Dongguk University, Seoul, Korea in 1987 and 2003, respectively. He was a Manager of the Computer Center, Seokyeong University from 1996-2000, a Director of Korea Multimedia Society from 2004-2005, a General Director of Korea Multimedia Society from 2005-2006 and a Vice President of Korea Multimedia Society in 2009. Also, he was a Director of Korea

Information Processing Society from 2006-2010 and a President of a Society for the Study of Game at Korea Information Processing Society from 2006-2010. And, he was a Director of Smart Developer Association from 2011-2012. Currently, he is a Professor of Dept. of Computer Engineering, Seokyeong University, Seoul, Korea. His research areas include smart system solutions, programming languages, and embedded systems.