5-4-2018

Comparing and Analyzing GNU Compilers

Brandon C. Powell

NORTHERN ILLINOIS UNIVERSITY

Comparing and Analyzing GNU Compilers

A Thesis Submitted to the

University Honors Program

In Partial Fulfillment of the

Requirements of the Baccalaureate Degree

With Upper Division Honors

Department Of

Computer Science

By

Brandon Powell

DeKalb, Illinois

May 12th, 2018

University Honors Program

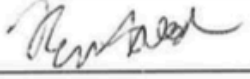
Capstone Approval Page

Capstone Title (print or type)

Comparing and Analyzing GNU Compilers

Student Name (print or type) Brandon Powell

Faculty Supervisor (print or type) Dr. Reva Freedman

Faculty Approval Signature 

Department of (print or type) Computer Science

Date of Approval (print or type) 5/3/2018

HONORS THESIS ABSTRACT

As programmers begin their careers, many do not have a keen understanding of the differences and discrepancies between differing versions of popular programming languages and their compilers. In addition, many programmers simply do not know how compilers operate when they convert code to machine code, thus making it executable. This report seeks to further my knowledge and understanding of the C++ programming language and the compiler known as GCC. In addition, there is a chance to discover if there are any forgotten features that are no longer implemented within the language that may be worth salvaging and bringing to the latest version of C++ or GCC. This report therefore will focus on researching differing aspects of the three most popular versions of C++, as well as discovering the inner machinations of a compiler.

HONORS THESIS ABSTRACT
THESIS SUBMISSION FORM

AUTHOR: Brandon Powell

THESIS TITLE: Comparing and Analyzing GNU Compilers

ADVISOR: Dr. Reva Freedman

ADVISOR'S DEPARTMENT: Computer Science

DISCIPLINE: CSCI YEAR: 2018

PAGE LENGTH: 15

BIBLIOGRAPHY: Yes

ILLUSTRATED: No

PUBLISHED (YES OR NO): No

LIST PUBLICATION: N/A

COPIES AVAILABLE: Electronic Submission

ABSTRACT (100-200 WORDS): See previous page

Code cannot run in traditional C++ without first being compiled. Compilers are an integral aspect of the computer programming world and should be placed in high regard. In rather simple terms, compilers are a special type of program that processes statements written in a specific programming language, and are translated into machine code in order for the machine's processor to finally execute the written program (Rouse). There exist multiple compilers, all having their own merits and shortcomings, but one of the most used and popular compilers is provided by the GNU family of compilers. GCC compilers are included in this family, which focus on compiling C and C++ code. Different versions of the GCC compilers have existed since C began, and the compilers continue to exist for C++. The compilers used for C++ are called G++, which are especially used in Linux operating systems (*GNU project C/C compiler - Linux man page*).

The GCC compilers and C++ as a whole are closely entwined. As improvements and changes are made to C++, the compiler must evolve and grow with it. Thus, in order to fully understand the discrepancies between GCC compilers, the discrepancies between C++ versions must be analyzed. Currently, GCC is on its 7th edition, with the newest standard version of C++ being named C++17 (GCC team(a)). Since the computer science department at Northern Illinois University has only recently upgraded to C++14, this report will analyze only the iterations of C++98, C++11, and C++14. With each new update comes new ways to muddle code, so improvements and discrepancies are sure to be found between each version.

The programming language of C++ was created in the late 70's and early 80's, and evolved from C, a less sophisticated programming language utilizing a type structure (Ritchie). The founder of C++, Bjarne Stroustrup, nicknamed the new language "C with classes," as the original language of C did not have classes, or a user defined data type that holds its own data

members and functions (Albatross). Created in 1998, C++98 marks the first international standard release of the language. This version of C++ was widely utilized and underwent an error, or bug fix in 2003. It was not until the next major release in 2011, called C++11, that many new features were added. Again, after its release in 2014, C++14 brought some new notable features.

Specified on the GNU website is a list of versions of GCC that support different versions and features of C++. Under the C++98 category, it states that all GCC versions fully support it, excluding one feature which was later removed following a later update to C++. This feature unsupported is called the export feature, which allowed a template, or a special kind of parameter that can be used to pass a data type as an argument, to be undefined but still declared. This means the template could be declared in a header file, or file that is used generally for function declarations or macro definitions for the main source file, and the template would thus be implemented in the source file. The export feature is used only in a few notable compilers other than GCC, known as Borland and Comeau (Sutter). However, since implementing the feature might have required a redesign of the compiler due to the link stage of compiling potentially failing, GCC decided against integrating it. To this day, it is widely unused due to a lack of necessity in more popular compilers.

Within the C++98 library, there is a feature once widely known as the `auto_ptr`, which was a type of pointer. A pointer is a data object that refers to another value stored in memory using its memory address, also known as "pointing to" that address. The now deprecated auto pointer allowed programmers to refer to an element in memory, with the added feature of obtaining ownership of that element ("Std::auto_ptr"). In other words, the auto pointer has the responsibility of deleting the element it points to once itself is deleted automatically. As stated

before, this feature of C++98 is now deprecated as of C++11, meaning it is no longer supported. This is due to the `auto_ptr` having issues with security of the data, which means this feature should be left in the past. Instead, a similar feature does exist in later versions called the `unique_ptr`, which contains better security and manipulation of data. In addition, the word `auto` is repurposed in C++11 and will be discussed later in this report.

The GNU website states that the first feature complete implementation for C++11 is GCC 4.8.1. With each version of GCC starting with GCC 4.3, new features were added to accommodate for the current version of C++ at the time (GCC team(a)). Therefore, it is most appropriate to discuss some important new features that C++11 brought forth, in addition to the `unique_ptr` already mentioned. One of these new features is called the array class, which allows for better memory management. An array consists of a series of elements sharing the same data type that can be referenced individually by index (Singh 2017a). This is competing with the original C-style arrays, and the vector class, which was implemented in C++98. The vector class consists of containers which represents arrays that can change size dynamically. Therefore, when the user creates a vector, that data is stored on the heap, or the pre-reserved computer memory that a process can use to store data that will not be known until runtime. Alternatively, the area where the data is stored for the array class is the stack, or the memory set aside for a thread of execution used for static memory allocation (Gribble). The reason the array class uses the stack is due to the fixed number of elements in the container, unlike the vector class. Arrays have always existed within C++ and its predecessor, C, but the array class brought with it methods similar to the vector class along with the more efficient memory handling. This allows the array class to be potentially faster than the vector class (Gove). An example of the array class follows:

```
// C++ code to demonstrate working of
// front() and back()
```

```
#include<iostream>
#include<array> // for front() and back()

using namespace std;

int main()
{
    // Initializing the array elements
    array<int,6> ar = {1, 2, 3, 4, 5, 6};

    // Printing first element of array
    cout << "First element of array is: ";
    cout << ar.front() << endl;

    // Printing last element of array
    cout << "Last element of array is: ";
    cout << ar.back() << endl;

    return 0;
}
```

In order to use the array class, users must use the `#include <array>` line to include the correct library. In addition, comments presented within this code example, and any hereafter, are presented with two forward slashes before them: `//`. To compile this file called `array.cpp` through GCC, if C++11 is not set as the default, users type this command in the terminal assuming G++ is being used:

```
g++ -O -std=c++11 array.cpp
```

Once it runs, the output is as follows:

```
First element of array is: 1
Last element of array is: 6
```

This output demonstrates two methods for the array class, with `ar.front()` and `ar.back()`, with the methods providing the first and last elements of the array respectively. These methods provide users with new tools for better data management control as well as clever manipulation of the container. For example, there is a method that can swap each element of one array with another array, simply called `swap()`, whereas a loop is needed for C-type array swapping (Singh 2017a).

Another new addition to C++11 is a container called a tuple, which is defined as an object able to hold a collection of elements, with each element able to be a different data type (Gove). Comparing this to the array class newly introduced in C++11, the two container objects are very similar. They both offer methods within their classes for container manipulation, such as swapping the elements with another array or tuple and finding the maximum size of either. They differ in that tuples can store differing types within one container, while arrays can store only one type per container. In addition, tuple possesses a method called `tie()`, which allows the elements of the tuple to be placed in separate variables with the same data types as in the tuple. An example of this method is as follows:

```
// C++ code to demonstrate working of tie()
#include<iostream>  // for input and output
#include<tuple>     // for tie() and tuple

using namespace std;

int main()
{
    // Initializing variables for data to be read into
    int i_val;
    char ch_val;
    float f_val;

    // Initializing tuple. Listing which types the data will be in order.
    tuple <int,char,float> tup1(20,'g',17.5);

    //Calling the function tie() to place tuple elements into variables.
    tie(i_val,ch_val,f_val) = tup1;

    // Displaying unpacked tuple elements
    cout << "The unpacked tuple values are: ";
    cout << i_val << " " << ch_val << " " << f_val;
    cout << endl;

    return 0;

}
```

This code, after again using the same compiling command at the terminal as before and running it, will produce the output:

```
The unpacked tuple values are: 20 g 17.5
```

This output shows that tuples are highly versatile, as even when a tuple cannot be used, it can still be converted to a traditional data type. Since the tuple class is also a fixed size, the stack is used, thus allowing programmers the same amount of memory management as the array class (Singh 2017b).

Yet another new feature for C++11 introduced is called the for-each loop, or more commonly, the range-based for loop. As the name implies, it allows easy access "for each" element of a container object, such as a vector, array, or structure. However, it proves difficult when working with tuples possessing differing data types since it usually requires all elements to have the same data type. An important keyword often utilized when creating a range-based for loop is the aforementioned `auto` keyword. First implemented in C++11, this keyword is utilized as a means to offer a data type when the data type is not yet known. Once it is determined however, GCC then replaces `auto` with the specified data type.

```
// Illustration of range-for loop
// using C++ code
#include <iostream>
#include <vector>

using namespace std;

int main()
{
    // Creating the vector:
    vector<int> v = {0, 1, 2, 3, 4, 5};
    //range-based for loop:
    for (auto i: v)
        cout << i << ' ';
}
```

After compiling and running this code, the output is:

```
0 1 2 3 4 5
```

The above code demonstrates a range-based for loop utilizing the auto keyword, which is common practice. The for loop creates a variable, the `auto i`, which must be the same data type as the elements within the following container since they are temporarily stored there, in this case from the vector `v`. This specific loop then commences by printing the value of that element (Thapiyal). This intuitive addition to C++ saves programmers time and energy, as iterating through containers without it requires more lines of code. In addition, the auto keyword in C++11 solves the issue of variable types being unknown for a period of time, as well as again saving time and resources by being able to reuse code segments for differing data types. Unlike the other newly added features of the array class and tuple class, which were stored in libraries that had to be included in the file, the auto keyword and for-each loop had to be recognized by the compiler. The GCC compiler supports the auto keyword starting with version 4.4, and the range-based for loop starting with version 4.6 (GCC team(a)).

In addition, C++11 brought with it the ability to use functional programming, which is defined as programming with functions that always return the same data type. It "relies on expressions and declarations rather than statements." Functional programming depends only on the arguments passed to the function. (Arsenault). This helps software to behave more predictably, and therefore programmers are drawn to it. C++ was not designed the same way as other more prominent functional programming languages, but the update in C++11 makes a promising case for functional programming within the language. In normal functional programming languages, functions are treated like objects, while in C++, they are not. To address this in C++11, function objects, or lambdas as they are commonly called, were added. These lambdas are simply a class with an operate method. Instead of having to create a class for each function, programmers can use the following code to create a class for a function:

```
auto println = [](const char  *message){ std::cout << message << std::endl;};
```

This code creates a function object called `println`, which takes a single parameter and returns nothing. The `[]` signifies the closure for the function, or the environment being specified. This means the body of the lambda expression accesses no variables in the enclosing scope. The auto keyword is used as a means of easier code reuse. The function object created above will print a message to standard output of whatever message was passed in. The lambdas within C++11 were first supported by GCC version 4.5 (GCC team(a)). Functional programming does not stop at lambdas, however, and the for-each loop as well as the auto keyword are important additions to help this style of programming (Phatak).

Another addition of C++11 is known as a smart pointer. A certain type of smart pointer in C++11 is known as the `unique_ptr`, previously mentioned during the `auto_ptr` discussion. Many features of the original `auto_ptr` in C++98 persist within smart pointers, as they also automatically delete the object pointed to, with the object pointing to being in the heap. However, smart pointers have far more control and security compared to their predecessor, such as when they delete what they point to. There are various types of smart pointers in C++11, with each specializing in differing data objects and ownerships. For example, the `scoped_ptr` points to an object with a sole ownership and is unable to be copied, compared to the `shared_ptr`, which has object ownership shared between multiple pointers (Colvin). In addition, the aforementioned `unique_ptr`, although highly similar to the `scoped_ptr` in that they both point to objects and are non-copyable, does support transfer of ownership, whereas `scoped_ptr` does not. These various smart pointers thus provide significant control over computer memory and the data being pointed to, as well as ownerships between the data.

C++ evolved yet again in the year of 2014, when C++14 was implemented. The GCC compiler fully supported this version of C++ with 6.1 (GCC Team(a)). This particular version of C++ has also been recently added to hopper and turing systems for the computer science department at NIU and provides even more new features. One important new feature concerns concurrency, which is best described as multiple processes running at once. Concurrency in C++ improved significantly with the addition of the `shared_time_mutex`, which is "a multiple reader, single-writer mutex" (Williams). First introduced in C++11, the mutex class is designed to signal when critical sections of code need exclusive access, preventing other threads, or smaller parts of processes, from executing at the same time and causing errors in data. It accomplishes this through the use of lockable objects, which prevent data from being accessed for a time. Before C++14, there was not a way to lock in shared ownership mode. Once this version was released however, multiple threads could then hold a shared ownership lock at the same time. This means data structures which are read frequently, but modified rarely, could use this method for better code execution. Any number of threads could potentially read from the data concurrently, which actually would produce no errors in the data, with at most only one thread able to write to the data structure. This particular new feature must be used in correct situations for maximum effectiveness however, as being used in the wrong situation such as for data structures that require constant data modification would not be ideal since it will cause the data to be much slower because it is constantly being accessed.

Another new feature implemented in C++14 is the further use of user-defined literals. Again, this feature was first created in C++11, but there were no new types of literals from the standard library. In C++14, this has changed. There is a new header called the "chrono" header in C++14, and with it comes vital ways to handle concurrency dealing with time, ranging from

hours to nanoseconds (Williams). The user-defined literals save programmers unnecessarily long keystroke time by allowing users to simple type '`30s`' representing thirty seconds of time, rather than the actual code for it: `std::chrono::seconds(30)`. Similarly, hours, minutes, and even nanoseconds can be abbreviated using this technique, so long as the programmer types in the following code into the program to access the abbreviations:

```
using namespace std::literals::chrono_literals;.
```

In addition, C++14 implemented a deprecated keyword to use in conjunction with the attribute specifier sequence, first created in C++11. The attribute specifier sequence allowed programmers to give their own code attributes for specific types, objects, and segments of code. The deprecated attribute specifically allowed users to mark their own code as obsolete. In other words, the code can still be used, but it is discouraged to use it. The following is a segment of code implementing this:

```
[[deprecated]]
void foo() {}

int main() {
        foo(); // using a deprecated function
```

In order for this code to run properly within a program file, again assuming C++14 is not set as the default for the compiler, the following must be entered when compiling at the terminal, as long as GCC 5.2 or later is used:

```
g++ -O -std=c++14 file.cpp
```

The deprecated attribute is an excellent technique for programmers to use who might work on a large project together. They may realize there is a function that should be discouraged to be used, but removing it entirely may break the code (Mansfield). The GCC compiler first implemented this attribute in version 4.9 (GCC Team(a)).

An interesting feature of C++14 was placed on the working paper for the language, and even made it into the compiler G++ at the time, but never went in the standard release version of C++14. This feature would allow array sizes to be determined at runtime with automatic storage duration. It did have some restrictions as well, such as not allowing multidimensional arrays and not allowing the user to use the function `sizeof(array)` to find the size of the array at runtime. However, since the feature was not added into the published standard version of C++14, it has been removed from the G++ compiler entirely. The feature had major drawbacks, as memory management would become a huge issue in C++. The GNU compiler as a whole still supports variable length arrays, or VLAs, since they are a feature in C99. These VLAs are therefore the same between C++98, C++11, and now C++14 (GCC Team(b)).

The C++14 language did not see many new features when comparing to the number of new features found within C++11. This is mostly due to the fact that the window of time between the two major releases of C++11 and C++14 only consisted of three years. The time gap between C++98 and C++11 consists of thirteen years, and even considering the 2003 update to C++ consists of an eight-year time gap. The language needed to evolve to stay relevant, and it managed to accomplish this with the many new additions. The GCC compiler over the years also updated regularly as each new version came out, adding new ways to improve the language as much as possible, including slight improvements to speed and quality of error messages. GCC and in turn G++ provided the support that the language needed, and still does.

Analyzing how compilers operate is essential to understanding why C++ and GCC change over time and provides a means of further understanding how code becomes executable. The process of analysis and synthesis, which is the process of creating an intermediate representation of source code, then taking that and converting it into an equivalent target of

machine code, allows GCC to run C++ programs and execute them (Kumar). GCC accomplishes this by utilizing six steps of conversion. The first is known as the Lexical Analyzer, which converts the program written by the user into tokens, which are the smallest element of a program meaningful to the compiler (Rouse). In this stage, comments and whitespace are also removed. The following stage is known as the Syntax Analyzer, which is also called a parser. Within this stage, a parse tree is constructed. The compiler takes in the previously created tokens one by one and utilizes context free grammar to construct the tree. Context Free grammars are a set of recursive rewriting rules used to generate patterns of strings (Rochester). It is here that syntax error is detected, if the input does not align with the grammar. The third conversion is known as the Semantic Analyzer, which verifies the parse tree. If this stage is successful, the parse tree becomes flagged as verified. The next stage is known as the Intermediate Code Generator. This is the main stage where code becomes the aforementioned intermediate code. This code can be readily executed by the machine and becomes converted within the final stages. It is important to note that up until this stage, most compilers follow these first four main steps. It is not until the final two steps where other compilers differ.

Within the fifth stage, known as the Code Optimizer, the compiler optimizes the intermediate code so that it consumes fewer resources and produces better speeds. The actual meaning of the code is not altered in any way in this stage. In addition, this stage allows optimization to be more focused on machine dependency or machine independency. Machine dependency allows the use of CPU registers, while machine independency does not involve any CPU registers. An advantage of machine dependency is optimizing memory hierarchy, while machine independency saves CPU cycles it could be used on any processor (Tutorials Point). The final step is known as the Target Point Generator, which writes code the machine

understands. The output produced depends on the type of assembler used by the compiler (Kumar). In the case of GCC, it utilizes the GNU Assembler, or GAS, which was developed for GNU. It was announced in 1986 and features cross-platform compatibility (Elsner 15). Understanding the mechanics of a compiler deepen the understanding of programming as a whole, as compiling is an integral aspect of any programming language.

It is important to look back on previous versions of C++ and GCC to find which features may have been upgraded, which features may have been added, and even which features that may have been left behind. By examining these discrepancies and taking into account how intertwined the GCC compiler is with C++ as a whole, programmers can discover if any features lost should be implemented once again or left in the past. In addition, understanding the fundamentals of compilers such as GCC and how they operate expands the understanding of any programmer. The compiler grows with the language, and as users give feedback and find breaches in security or memory leakage, the programming community handles the problems and helps to fix them. Whether it be the `auto_ptr` of C++98, or the never implemented C++14 standard variable size array, the community decided these features had major drawbacks and that they should be left in the past. New versions of C++ and G++ may cause code written in a previous version to break in some aspect as well, which exemplifies why having a keen understanding and sharp knowledge of the discrepancies between these versions are quintessential for maintaining code as C++ and GCC evolve.

Works Cited

Albatross. “History of C++.” *Cplusplus.com*, www.cplusplus.com/info/history/.

Arsenault, Cody. “Functional Programming - What Is It and Why Does It Matter?” *KeyCDN Blog*, 24 Aug. 2017, www.keycdn.com/blog/functional-programming/.

Colvin, Greg, et al. “Boost.SmartPtr: The Smart Pointer Library.” *Boost C++ Libraries*, www.boost.org/doc/libs/1_67_0/libs/smart_ptr/doc/html/smart_ptr.html.

Elsner, Dean. *Using as: the Gnu Assembler*. IUniverse Com, 2000.

“g (1) - Linux man page.”*g (1): GNU project C/C compiler- Linux man page*, linux.die.net/man/1/g .

GCC team. (a) “C++ Standards Support in GCC.” *GNU*, gcc.gnu.org/projects/cxx-status.html#cxx98.

GCC team. (b) “GCC 5 Release SeriesChanges, New Features, and Fixes.” *GNU*, gcc.gnu.org/gcc-5/changes.html#cxx.

Gove, Darryl, and Steve Clamage. “Using the New C++11 Array and Tuple Containers.” *How to Use the New C++ Array and Tuple Containers*, July 2014, www.oracle.com/technetwork/articles/servers-storage-dev/c-array-containers-2252536.html.

Gribble, Paul. “Memory: Stack vs Heap.” *Gribblelab*, www.gribblelab.org/CBootCamp/7_Memory_Stack_vs_Heap.html.

Kumar, Rajesh. “Compiler Design | Phases of a Compiler.” *GeeksforGeeks*, 7 Feb. 2018, www.geeksforgeeks.org/compiler-design-phases-compiler/.

Mansfield, Joseph. “Marking as Deprecated in C++14.” *Joseph Mansfield*, 9 June 2014, josephmansfield.uk/articles/marking-deprecated-c++14.html.

Phatak, Madhukara. “Functional Programming in C++.” *Madhukaraphatak*, 16 Nov. 2014, blog.madhukaraphatak.com/functional-programming-in-c++/.

Ritchie, Dennis M. “The Development of the C Language*.” *Chistory*, 2003, www.bell-labs.com/usr/dmr/www/chist.html.

Rochester. “Context-Free Grammars.” *Rochester*, www.cs.rochester.edu/~nelson/courses/csc_173/grammars/cfg.html.

Rouse, Margaret. “What Is Token?” *WhatIs.com*, whatis.techtarget.com/definition/token.

Singh, Manjeet. (a) “Array Class in C++.” *GeeksforGeeks*, 29 May 2017, www.geeksforgeeks.org/array-class-c/.

Singh, Manjeet. (b) “Tuples in C++.” *GeeksforGeeks*, 29 May 2017, www.geeksforgeeks.org/tuples-in-c/.

“Std::auto_ptr.” *Cplusplus.com*, www.cplusplus.com/reference/memory/auto_ptr/.

Sutter, Herb. “Sutter's Mill: Herb Sutter ‘Export’ Restrictions, Part 1.” *Dr. Dobb's*, www.drdobbs.com/cpp/sutters-mill-herb-sutter-export-restrict/184401563.

Thapliyal, Rohit. “Range-Based for Loop in C++.” *GeeksforGeeks*, 1 Sept. 2017, www.geeksforgeeks.org/range-based-loop-c/.

Tutorials Point. “Compiler Design - Code Optimization.” *Www.tutorialspoint.com*, Tutorials Point, 8 Jan. 2018, www.tutorialspoint.com/compiler_design/compiler_design_code_optimization.htm.

Williams, Anthony. “New Concurrency Features in C++14.” *JustSoftwareSolutions*, www.justsoftwaresolutions.co.uk/threading/new-concurrency-features-in-c++14.html.