A Tool for C++ Header Generation

An Extension of the C++ Programming Language

Patrick Hock[1], Koichi Nakayama[2], Kohei Arai[3]
Faculty of Science and Engineering, Saga University, Saga, Japan

***Abstract*—This paper presents a novel approach in the field of C++ development for increasing performance by reducing cognitive overhead and complexity, which results in lower costs. C++ code is split into header and cpp files. This split induces code redundancy. In addition, there are (commonly used) features for classes in C++ that are not supported by recent compilers. The developer must maintain two different files for one single content and implements unsupported features by hand. This leads to the unnecessary cognitive overhead and complex sources. The result is low development performance and high development cost. Our approach utilizes an enhanced syntax inside cpp files. It allows header file generation and therefore obsoletes the need to maintain a header file. It also enables the generation of features/methods for classes. It aims to decrease cognitive overhead and complexity, so developers can focus on more sophisticated tasks. This will lead to increased performance and lower costs.**

***Keywords*—Development; C++; header file generation; feature generation**

I. Introduction

C++ is a rather old programming language with a low convenience level. Nevertheless, it is still used in schools and the industry. Updates of the C++ standard denote, the language is not dead. Further, Microsoft promotes the use of C++ through the regular renewal of its C++ IDE *Microsoft Visual Studio* [1]. Over time, advanced IDEs and updates of the C++ standard provided a better developing experience in C++. However, development with C++ is still complex and costly. One reason for that might be the split of declaration and definition into two files: header and cpp file. This induces a code redundancy that must be kept in sync. This induces a cognitive and maintenance overhead; e.g., the change of a method name must be done inside the header file *and* the cpp file. After changing one, it is necessary to remember (cognitive overhead) to also change the other (maintenance overhead). If either one is forgotten, the compiler returns an error and the code needs to be recompiled after correcting. This decreases performance and increases development cost. The question arises whether the split in two files is necessary. Comparison to other programming languages (e.g. D [2]) reveals that this split does not seem vital.

Furthermore, there are (commonly used) features for classes that are not supported by recent compilers. E.g. generation of get/set methods (implemented in C#[3]) or the ability to initialize a member variable at declaration time (implemented in Java[4]). Henceforth called *coding inconveniences*. The developer has to work around these missing features. This increases cognitive overhead and code complexity, which leads to lower performance. This leads to higher development cost.

This paper presents the idea of a text-based inline code generator, that utilizes an enhanced C++ syntax inside cpp files to generate header files and features that are not yet supported by compilers. It aims to decrease cognitive overhead for development and reduce code complexity, which leads to higher performance and lower development cost. Further, this paper introduces the tool *cppHeaderGen* which implements the presented idea.

II. Goals and Constraints as well as Related Research

A. Goals and Constraints

The overall goal is to develop a tool to improve developing experience for C++ through lowering cognitive overhead for development and complexity of source files. To achieve that the following concrete goals should be fulfilled.

1) Obsolete the need to maintain header files; header files are being generated.

2) Improve coding inconveniences; e.g. variable definition and initialization can be done in the same place.

These concrete goals should be realized while living up to the following constraints:

1) Environment independence: The tool is on the same availability level as C++ compilers. As long as C++ compilers run on a machine, it is possible to utilize the tool. This implies the following sub constraints:

a) Independence of IDE

b) Independence of build chain

c) (Source code) Independence of operating system

2) Gradual integration into existing projects possible: The tool does not enforce its project-wide usage. It can be used for specific files only. This enables a gradual integration process for existing projects.

3) Integrable into microsoft visual studio: From the authors view, Microsoft Visual Studio is an important IDE for C++ development under Windows. Therefore, the possibility to integrate the tool into Microsoft Visual Studio is mandatory.

4) Short working distance: Code changes are done *in place*. It is not necessary to open a different software or file to change currently viewed code. Otherwise slight changes, such as a variable name change, might be refrained from, because it's perceived as "too much effort for a slight change".

5) Debugging and coding in the same file: It is possible to debug and code in the same file. This reduces working distance (constraint 4)) and cognitive overhead for working with multiple files. It eliminates a possible corruption of breakpoint settings after a line number change within the code file. This is important for debugging, where *step execution and code updating* are repeated several times.

B. Related Research

There are already tools available, that aim to improve development experience for C++. The following sections introduce some of the currently available tools and illustrate their major drawback(s). The sections illustrate that currently available tools do not implement all aforementioned goals while living up to all constraints stated in Section II.A.

1) IDEs: Some IDEs (e.g. Microsoft Visual Studio [5], Eclipse [6], JetBrains CLion [7], etc.) offer great support for a better development experience in C++. E.g. classes or methods can be conveniently created or changed via the GUI. Their major drawback is their dependency on themselves and the operating system (violation of constraint 1)). Changing the IDE disables their features. Changing the operating system might enforce an IDE change.

2) Plug-Ins for IDEs: Some plug-ins for IDEs (e.g. JetBrains ReSharper [8] for Microsoft Visual Studio [5]) offer great enhanced functionality for a better development experience in C++, such as method generation. Their major drawback is their dependency on the IDE and operating system (violation of constraint 1)). Changing the IDE disables their features. Changing the operating system might enforce an IDE change.

3) Graphical code generators: Graphical code generators offer a great functionality for generating cpp and header files. They make it possible for a single change to be effective in both files. Their major drawback is the long working distance between coding and generation (violation of constraint 4)). E.g. changing the name of a member variable requires the overhead of opening the code generator software, navigating to the corresponding class and searching for the member variable declaration. This overhead might be perceived as "too much effort for a slight change". As a result, such minor changes (that might improve readability) might not be done and less readable code remains.

4) Domain specific language to C++ (text-based code generation): There is a methodology that focuses on translating a domain specific language [9][10][11] (henceforth *DSL*) to C++. This can be regarded as text-based code generation. Code generator instructions and source code are merged to one entity. Therefore, this methodology is not subject to the *working distance* drawback of graphical code generators. Its major drawback is the inability to debug and code in the same file (violation of constraint 5)). This leads to the following subsequent problems:

- During a debugging session *step execution and code updating* might be repeated several times. The DSL makes it necessary to update and debug in two different files: the DSL source file for updating code and the cpp file for debugging code. This induces a maintenance and cognitive overhead on the developer.
- While breakpoints for debugging are set inside the cpp file, coding is done inside the DSL source file. If a code change results in a line number change, the breakpoint settings inside the IDE might become obsolete. It might be necessary to re-set all breakpoints by hand.

5) Lzz–the lazy C++ programmer's tool: Lzz[12] is a text-based code generator focused on making C++ development more convenient. It can be regarded as a DSL within the ease-of-use domain. The focus of Lzz is making C++ development more convenient. Its major drawback is the inability to debug and code in the same file (violation of constraint 5)).

III. Proposed Method

To fulfill all goals and constraints from Section II.A, this paper proposes the use of a *text-based inline code generator* that utilizes an enhanced C++ syntax to generate a header file from a cpp file. It also introduces the tool *cppHeaderGen* (short for *C++ Header Generator*) as an implementation of the proposal.

A. Text-Based Inline Code Generator

The text-based inline code generator (henceforth abbreviated as *code generator*) receives a cpp file with an enhanced C++ syntax as input (see TABLE. Ifor an example list of new keywords). It generates the corresponding header file with all necessary declarations as output. Therefore, it obsoletes the maintenance of the header file (goal 1)). It can also generate (commonly used) methods such as get/set methods and provide convenient features like initialization and declaration at the same time. This improves coding convenience (goal 2)). Through the respective keywords within the cpp file the code generator knows which declarations, methods or features it needs to be generated.

B. Method Generation

Regarding method generation there are two possible solutions. As either one has its benefits or drawbacks both should be provided.

1) Generate code in a separate cpp file and add it to the list of files to compile within a project. This solution has the drawback of adding a new file to the project, which will make its structure more complex. The advantage is, that content of generated methods is not exposed to the public.

2) Generate code inside the header file and enable it only within the controlling cpp file. This solution has the drawback of revealing class internal details to the interface. The advantage is, that the number of project files does not increase.

If methods are defined within header files, it's necessary to prevent *multiple definition* errors. The mechanism to generate code inside header files without raising *multiple definition* errors is demonstrated in Fig. 1.

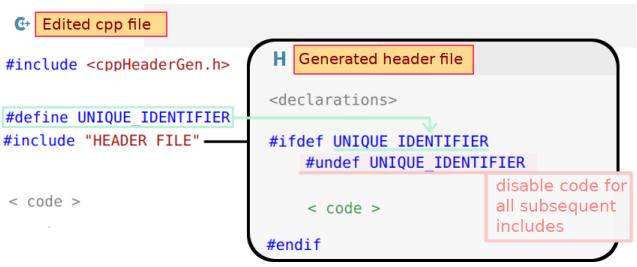

Fig. 1. Mechanism to Generate Code Inside the Header File without Raising Multiple Definition Errors.

C. Complying with Constraints

A text-based inline code generator would be the first entity within the build chain and therefore independent of any other entity (e.g. IDE) (constraint (1.1), (1.2)). To ensure independence of the operating system (constraint (1.3)), the implementation must be open source. As it is used *per cpp file*, it does not persist on project wide usage and is therefore gradually integrable into existing projects (constraint 2)). Providing a command line interface will ensure the possibility to integrate it into Microsoft Visual Studio (constraint 3)). As generated code is controlled directly via the cpp file, it remedies the working distance drawback of graphical code generators (constraint 4)) and allows coding and debugging in the same file (constraint 5)). As the content of generated methods is trivial, it's not rated a violation of constraint 5).

D. Implement Enhanced C++ Syntax

The enhanced syntax must only be visible by the code generator. It must not be visible to C++ compiler. If it would be visible to a C++ compiler, it would return compile errors. The syntax can be implemented utilizing the preprocessor. `define`s and macro definitions within a separate header file (henceforth *syntax header file*) can remove all enhanced syntax prior to compiling. To comply with constraint (1.2) it's must be ensured, that preprocessor commands are backwards compatible.

Content of the syntax header file must be contained in every cpp file that uses the enhanced syntax. This could be accomplished via a direct `include` within the cpp file or a generated `#include <cppHeaderGen.h>` inside the generated header file.

E. Limitations

1) Use of macros is inevitable: Content that is not supposed to be inside a cpp file must be removed by the preprocessor. This constraint makes the use of macros for these cases inevitable. Macros are the only possibility to remove arbitrary content through the preprocessor. Therefore, syntax as shown in Fig. 2 is not possible. Instead, syntax like in Fig. 3 needs to be used.

```
class MyClass  // class declaration start
int foo = 5;   // member variable declaration
```

Fig. 2. Impossible Syntax within the cpp File.

```
Class (MyClass)      // class declaration start
Var (int foo = 5);   // member variable declaration
```

Fig. 3. Use of Macros for Content that is not Supposed to be Inside a cpp File.

```
void MyClass::foo() { … }
```

Fig. 4. Method Definition: Method `Foo` of Class `MyClass`.

```
void foo() { … }
```

Fig. 5. Desirable Method Definition: Method `Foo` of class `MyClass`.

2) Class name before method name at definition: When defining methods in C++, it's necessary to write the class name in front of method names (see Fig. 4). However, a more convenient way as in Fig. 5 might be desirable.

Technically it is possible to remove the burden of writing the class name before the method name. However, it is suggested not to implement such a solution, because it would render currently available C++ code outliners useless.

F. Further Details

Further details about the proposed code generator are implementation dependent and are therefore described along with its example implementation *cppHeaderGen*.

IV. Example Implementation: cppHeaderGen

This chapter presents the features and implementation details of cppHeaderGen.

A. Outline

cppHeaderGen is the example implementation of the proposed text-based inline code generator for C++. It uses the cpp file with an enhanced C++ syntax as input. It basically outputs a header file containing necessary declarations. Method generation can be outputted within a separated cpp file or directly within the header file. An example input is shown in Fig. 6 and the respective output in Fig. 7.

In Fig. 6, cppHeaderGen utilizes the keywords `Class`, `ClassEnd` and `Public` to determine how the header should look like. `cppHeaderGen.h` contains code to implement these keywords. Its inclusion is mandatory to prevent compile errors (for details see Section IV.B.1)).

B. Development Environment

cppHeaderGen is written in C++ using generated files from flex (lexer)[13], GNU Bison[14] and cppHeaderGen itself.

```
#include <cppHeaderGen.h>
#include <iostream>
Class (MyClass)
  Public void MyClass::foo(int param1) {
    std::cout << "hello world";
  }
ClassEnd
```

Fig. 6. Code Example for a Class with a Method Written for cppHeaderGen.

```
//File Generated by cppHeaderGen
#ifndef _test_H_DOUBLE_INC_PREVENTION
#define _test_H_DOUBLE_INC_PREVENTION
class MyClass
{
        public:  void foo(int param1);
};
#endif
```

Fig. 7. Example Output: Code Generated from Fig. 6.

1) Implement enhanced C++ syntax: The enhanced syntax is implemented within `cppHeaderGen.h` using `define` directives and macro definitions. The content of `cppHeaderGen.h` is show in Fig. 8.

To prevent compilation errors, the inclusion of `cppHeaderGen.h` is mandatory. It's possible to generate an `#include "cppHeaderGen.h"` inside the header file using command line options.

2) New features and method generation: For features like member initialization at declaration time or method generation, there are two options as destination location for the code.

- Inside a dedicated `gen.cpp` file (default)
- Inside the generated header file

The examples in the chapters below use the `Var` keyword to trigger the generation of a standard constructor for initialization (see 5) for more details)

a) Generate methods inside a dedicated gen.cpp file (default): Generating methods inside a dedicated gen.cpp file is the default setting. It's necessary to add the generated file to the *list of files to compile* (e.g. the project). Fig. 9 shows an example input for generating a constructor for a class. Fig. 10 shows the generated output.

```
#ifndef CPPHEADERGEN_H
#define CPPHEADERGEN_H
#define Class(…)
#define ClassEnd
#define HF( ... )
#define Include( ... )
#define Def
#define GlobalVar
#define ExternVar
#define Static
#define Virtual
#define Public
#define Private
#define Protected
#define PublicVar( ... )
#define PrivateVar( ... )
#define ProtectedVar( ... )
#define Var( ... )
#define GENERATE_copyNonPointerMember
#define CTOR __init
#endif
```

Fig. 8. Content of `cppHeaderGen.h`.

```
#include <cppHeaderGen.h>
#include "myclass.h"
Class ( MyClass )
  Var(public; int; foo; 8); // define a new variable
ClassEnd
```

Fig. 9. Example Input: Class with a Member Variable Definition. Setting a Default Value ("8") Triggers the Creation of a Standard Constructor.

```
#include "myclass.h"
MyClass::MyClass() : foo(8) {}
```

Fig. 10. Example Output: Content of File `Myclass.gen.cpp` from Fig. 9.

b) Generate methods inside header files: If it's desirable to generate only one file, methods can be generated directly into the header file. This might expose class-private data through the header file. To activate this option a specific `define` is set inside the cpp file before the inclusion of the corresponding header include. The `define` complies with the following pattern: `#define genInHeader_`[unique identifier]. Fig. 11 shows an example input and Fig. 12 shows the generated output.

3) Support for older compilers: The invalidation of the enhanced syntax uses macros with variable parameter count (henceforth: *variadic macros*). Some older compilers [15] do not support variadic macros. For older compilers there is a different header file to include: `cppHeaderGenNoVar.h`. Macros inside this header are not defined *variadic*. An extract of the file is shown in Fig. 13.

Using this include file changes the enhanced syntax. Instead of single brackets for macros, double brackets are used. Fig. 14 shows an example for the `Class` macro.

```
#include <cppHeaderGen.h>
#define genInHeader_MyClass
#include "myclass.h"
Class ( MyClass )
  Var(public; int; foo; 8); // define new variable
ClassEnd
```

Fig. 11. Example Input: Generate Methods Inside Header File. The Trigger for Generating Methods Inside the Header File is Marked Bold.

```
[…]
class MyClass { […] }

#ifdef genInHeader_MyClass
#undef getInHeader_MyClass
MyClass::MyClass() : foo(8) {}
#endif
[…]
```

Fig. 12. Example Output: Code Generated From Fig. 11. "[…]" is used as Abbreviation of Content.

```
#define HF( A )
#define Include( A )

#define Def
#define GlobalVar
#define ExternVar

#define Static
#define Virtual
```

Fig. 13. Extract of File `cppHeaderGenNoVar.h` that Shows a Non-Variadic Macro Definition.

```
Class(( MyClass ))
  Public void MyClass::foo(int param1) {
    std::cout << "hello world";
  }
ClassEnd
```

Fig. 14. Example Input: Double-Bracketed Enhanced Syntax for Support for Older Compilers. The Parameter List of the Method is Not Part of the Enhanced Syntax. Therefore it must not have Double Brackets.

```
Class( MyClass : public Base1, Base2 )
```

Fig. 15. Demonstration of a Macro Containing Two Parameters.

4) Variadic macros: The reason why variadic macros are necessary is because even a simple class definition with several base classes contains a comma, which is interpreted by the preprocessor as *multiple parameters* (see Fig. 15).

5) Keyword list: The following TABLE. Iintroduces all keywords and features provided by cppHeaderGen at the time being.

6) Management features: Regarding file generation cppHeaderGen provides the following features.

- No double inclusion

Double inclusion of headers is avoided through the `#ifndef` include guard directive.

- Handwritten header files do not get overwritten

Every generated header file contains a specific comment that marks the file as *generated*. A header file will only be overwritten, if it is marked as *generated*.

- Only renew on change

A header file is only renewed, if its content changed. This preserves file generation timestamps and therefore prevents unnecessary rebuilds.

7) Integration into microsoft visual studio: Integration into Microsoft Visual Studio can be accomplished through *pre-build events* within the project settings.

TABLE. I. LIST OF ALL KEYWORDS AND FEATURES PROVIDED BY THE SYNTAX OF CPPHEADERGEN (SEE 6) FOR FILE MANAGEMENT FEATURES).

Keyword	
Explanation	***Example***
HF([content]))	
Copies [content] verbatim into the header file. All hashtags within [content] must be escaped with a backslash.	Example input: HF(// copy to header file. \#ifdef FOO \#endif) Example output: // copy to header file. #ifdef FOO #endif
Include("[filename]") / Include(<[filename]>)	
Creates an include statement inside the header file.	Example input: Include(<string>) Example output: #include <string>
Class ([classname]) / Struct ([structname])	
Denotes the start of a new class. In the current version nested classes are not fully supported.	Example input: Class(MyClass) Example output: class MyClass {
ClassEnd	
Denotes the end of Class.	Example input: ClassEnd Example output: }
Public / Private / Protected	
Denotes the start of a method definition with the given visibility.	Example input: Public MyClass::foo(int param) { ... } Example output: public: foo(int param);
Static	
Keyword used to declare a method static.	Example input: Public Static void MyClass::foo() { ... } Example output: public: static void foo();
Virtual	
Keyword used to declare a method virtual.	Example input: Public Virtual void MyClass::foo() { ... } Example output: public: virtual void foo();

<table>
<tr><td colspan="2">Var([visibility]; [type] ; [variable name])</td></tr>
<tr><td>Create a member variable declaration inside the header file. This notation develops its full potential when used with GENERATE_copyNonPointerMember.</td><td>Example input:
Var(public; int; var)
Example ouput:
public: int var;</td></tr>
<tr><td colspan="2">Var([visibility]; [type]; [variable name]; [initialization value])</td></tr>
<tr><td>Create a member variable declaration inside the header file and initialize it with 7. The initialization is realized through the generation of initializer lists and constructors. If no custom constructor is defined, a standard constructor will be generated.</td><td>.Example input:
Var(public; int; var; 7)
Example ouput:
public: int var;
[...]
MyClass::MyClass() : var(7) {}</td></tr>
<tr><td colspan="2">GENERATE_copyNonPointerMember</td></tr>
<tr><td>Generate a method that copies the content of all declared non-pointer variables to another object. Only variables declared via Var() are considered.</td><td>Example input:
Class (MyClass)
Var(public; int; var)
GENERATE_copyNonPointerMember
EndClass
Example ouput:
class MyClass {
private:
void copyNonPointerMemberFrom
(const MyClass & source);
[...]
void MyClass::
copyNonPointerMemberFrom (
const MyClass &source)
{
this->var = source.var; }</td></tr>
<tr><td colspan="2">[visibility] void [classname]::CTOR([parameter]) {}</td></tr>
<tr><td>Generate a constructor for the class [classname]. It must be used in conjunction with a visibility indicator (Public / Private / Protected) and the classname.
Inside cppHeaderGen.h CTOR is changed to __init through the following define:
#define CTOR __init</td><td>Example input:
Public void MyClass::CTOR
(int param1)
{ }
Example output:
public: inline void __init
(int param1);
public: MyClass(int param1);</td></tr>
<tr><td colspan="2">Def [function definition]</td></tr>
<tr><td>Create a declaration for a (global) function. The namespace of the function will be stripped away.</td><td>Example input:
Def std::string myNamespace
::foo(int param){}
Example output:
std::string foo(int param){}</td></tr>
<tr><td colspan="2">GlobalVar</td></tr>
<tr><td>Create an extern declaration for a given variable.</td><td>Example input:
GlobalVar int gValue = 1;
Example output:
extern int gValue;</td></tr>
<tr><td colspan="2">[method generation]</td></tr>
<tr><td colspan="2">By default methods are generated inside a dedicated gen.cpp file. The generated file must be included in the list of files to compile. Methods can also be generated inside the header file.</td></tr>
<tr><td colspan="2">#define genInHeader_[unique specifier]</td></tr>
<tr><td>Instructs the generator to generate methods directly inside the header file. No seperate gen.cpp file will be generated. The define must be set before the associated header file is included.</td><td>Example input:
#define genInHeader_MyClass
#include “myclass.h”
Class (MyClass)
Var(public;int;foo;8)
EndClass
Example output:
class MyClass{
public: int foo;
public: MyClass();
}
#ifdef genInHeader_MyClass
#undef genInHeader_MyClass
MyClass::MyClass() : foo(8);
#endif</td></tr>
</table>

C. Limitations

1) Syntax for member variable declaration: The current syntax for variable declaration (`Var(` [visibility]; [type]; [name]; [initial value]`)`) is very different from the C++ standard. The reason why this syntax was chosen over a more native syntax is that it's easier to parse. In future versions the syntax shown in Fig. 16 might become supported.

```
PublicVar ( int foo = 5 )
PrivateVar( const string foo("hello") )
```

Fig. 16. Possible Future Syntax for Variable Definition.

The reason why a syntax as shown in Fig. 17 cannot be supported is that `Public` is already defined as `#define Public` (non-macro definition). Creating a macro with the same name is not allowed by the preprocessor.

```
Public( int foo = 5 )
```

Fig. 17. Possible Future Syntax for Variable Definition.

2) Class name before method name at definition: Removing the need to write a class name before a method name at definition time renders a code outliner useless. To ensure a working C++ code outlining, no measures are taken to eliminate the need to write the class name before method names at definition time.

3) CppHeaderGen can only process one file per call: It is not possible for cppHeaderGen to process multiple files or whole directories per. If such functionality is needed (e.g. as for Section III.7)), it's necessary to use an external program or script that calls cppHeaderGen multiple times.

D. Example

Fig. 18 shows an example of a generated header file. On the left side, there is the manually created file *myclass.cpp*. On the right side, there is the generated file *myclass.h*. Fig. 19 demonstrates the use of class *MyClass* defined in Fig. 18. Particularly it demonstrates the use of the generated method for copying non-pointer member variables.

myclass.cpp × Edited cpp file

```
1  #include "cppHeaderGen.h"
2
3  #define genInHeader_MyClass
4  #include "myclass.h"
5  Include( <string> )
6  Include( <iostream> )
7
8  HF( using namespace std;)
9
10 Class ( MyClass )
11   Var( private;  string; _content; "Hello World")
12   Var( private;  int;    _printCounts; 3)
13
14   Public void MyClass::setTimesPrint(int count){
15     _printCounts = count;
16   }
17
18   Public void MyClass::print(){
19     for ( int i = 0; i < _printCounts; ++i){
20       cout << _content << endl;
21     }
22   }
23   GENERATE_copyNonPointerMember
24   // copy ctor does not need initializer lists
25   Public MyClass::MyClass(const MyClass & myclass){
26     copyNonPointerMemberFrom(myclass);
27   }
28
29 ClassEnd
30
31 // ---------------------------------
32
```

myclass.h × Generated header file

```
1  //File Generated by cppHeaderGen
2  #ifndef _myclass_H_DOUBLE_INC_PREVENTION
3  #define _myclass_H_DOUBLE_INC_PREVENTION
4
5  #include  <string>
6  #include  <iostream>
7   using namespace std;
8
9  class  MyClass
10 {
11     private: string  _content;
12     private: int  _printCounts;
13     public:  void setTimesPrint(int count);
14     public:  void print();
15     private: void copyNonPointerMemberFrom( const MyClass & source);
16     public:  MyClass(const MyClass & myclass);
17     public:  MyClass();
18 };
19
20 #ifdef genInHeader_MyClass
21 #undef genInHeader_MyClass
22 void MyClass::copyNonPointerMemberFrom( const MyClass & source){
23     this->_content     = source._content;
24     this->_printCounts = source._printCounts;
25 }
26
27
28 MyClass::MyClass() : _content("Hello World"), _printCounts(3){}
29
30
31 #endif
32 #endif
```

Fig. 18. Example Generation of File Myclass.H Containing Declarations and Definitions for Class Myclass. The Input File (*Myclass.Cpp*) is Shown on the Left. The Output File (*Myclass.H*) is Shown in the Right. Colored Areas Indicate Correlated Code. The Following Features are in use: Method Generation in Header File, Include, Verbatim Copy to Header File, Variable Definition and Initialization, Constructor Generation, Method Declaration, Generation of Member Variable Copy Method.

```
int main()
{
  MyClass printer1;
  cout << "printer1:\n";
  printer1.print(); //  3x "hello world"

  printer1.setTimesPrint(5);

  cout << "printer2:\n";
  MyClass printer2(printer1);

  printer2.print(); //  5x "hello world"

  return 0;
}
```

Fig. 19. Example Program: uses Myclass from Fig. 18 to demonstrate the use of the Generated Method `CopyNonPointerMemberFrom`.

V. Discussion

Using cppHeaderGen in practice smoothened C++ development. For developing cppHeaderGen itself usage of cppHeaderGen is already part of the build chain. For small-sized projects or projects without a full-featured development environment cppHeaderGen is rated worth using by the author. There is no experience regarding the usage in large projects.

VI. Conclusion

This paper worked on a concept for improving the development experience in C++. It presented the idea to utilize a text-based inline code generator controlled by a cpp file to generate and obsolete the need to manually maintain the according header file. It could improve coding inconveniences, as it was able to provide new features to the C++ language (like initialization at declaration time) and method generation. This paper introduced the tool cppHeaderGen, which implemented the idea of a text-based inline code generator. cppHeaderGen took a cpp file with an enhanced C++ syntax as input and outputted the corresponding header file. cppHeaderGen successfully obsoleted the need to maintain the header file. It allowed for a more convenient developing experience through the ability of method generation. E.g. it allowed member variable initialization at declaration time. It was independent of the underlying operating system, IDE or build chain and could gradually be integrated into existing projects. It was integrable into Microsoft Visual Studio. All coding was done in place. Therefore, it had a short working distance and coding and debugging could be done in the same file.

VII. Future Work

The code generator runs before the compilation process and therefore allows for a wide spectrum of possibilities regarding code generation. Future work should focus on finding new helpful features and generatable methods. Research should also be done regarding *helpfulness of a paradigm change*, such as making `virtual` the default modifier for method declaration.

cppHeaderGen should implement further, already known features to evaluate their usefulness. At the time being, the following features are candidates for future implementations.

- Generation of `get` / `set` methods for member variables.
- Generation of virtual clone methods for classes.
- More native-like syntax for variable declaration, like `PublicVar(int foo = 5)`.
- Generation of enum classes.
- Constructor initialization through parameters.
- Generation of a method that deletes all pointers.

References

[1] List of Microsoft Visual Studio versions: https://visualstudio.microsoft.com/vs/older-downloads/, (Accessed on Jun 2019).

[2] D programming language: https://dlang.org/, (Accessed on Jun 2019).

[3] C#: Auto-Implemented Properties: https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/auto-implemented-properties, (Accessed on Jun 2019).

[4] Java: Initialize member variable at declaration time: https://docs.oracle.com/javase/tutorial/java/javaOO/initial.html, (Accessed Jun 2019)

[5] Microsoft Visual Studio: https://visualstudio.microsoft.com/, (Accessed on Jun 2019).

[6] Eclipse (for C++ developers) : https://www.eclipse.org/downloads/packages/release/2019-03/r/eclipse-ide-cc-developers, (Accessed on Jun 2019).

[7] JetBrains CLion: https://www.jetbrains.com/clion/, (Accessed on Jun 2019).

[8] JetBrains ReSharper: https://www.jetbrains.com/resharper-cpp/, (Accessed on Jun 2019).

[9] Domain Specific Language by Microsoft: https://docs.microsoft.com/en-us/visualstudio/modeling/about-domain-specific-languages?view=vs-2019, (Accessed on Jun 2019).

[10] Domain Specific Language by Martin Fowler: https://www.martinfowler.com/books/dsl.html, (Accessed on Jun 2019).

[11] Domain Specific Language by JetBrains: https://www.jetbrains.com/mps /concepts/domain-specific-languages/, (Accessed on Jun 2019).

[12] Lzz: The Lazy C++ Programmer's Tool: https://www.lazycplusplus.com/, (Accessed on Jun 2019).

[13] Flex (lexer): https://github.com/westes/flex, (Accessed on Jun 2019).

[14] GNU Bison: https://www.gnu.org/software/bison/, (Accessed on Jun 2019).

[15] "Variadic macros became a standard part of the C language with C99": https://gcc.gnu.org/onlinedocs/cpp/Variadic-Macros.html, (Accessed on Jun 2019).

Authors' Profile

Kohei Arai received BS, MS and PhD degrees in 1972, 1974 and 1982, respectively. He was with The Institute for Industrial Science and Technology of the University of Tokyo from April 1974 to December 1978 and also was with National Space Development Agency of Japan from January, 1979 to March, 1990. During from 1985 to 1987, he was with Canada Centre for Remote Sensing as a Post Doctoral Fellow of National Science and Engineering Research Council of Canada. He moved to Saga University as a Professor in Department of Information Science on April 1990. He was a Councilor for the Aeronautics and Space related to the Technology Committee of the Ministry of Science and Technology during from 1998 to 2000. He was a councilor of Saga University for 2002 and 2003. He also was an executive councilor for the Remote Sensing Society of Japan for 2003 to 2005. He is an Adjunct Professor of University of Arizona, USA since 1998. He also is Vice Chairman of the Commission-A of ICSU/COSPAR since 2008. He wrote 30 books and published 570 journal papers.

Koichi Nakayama is an Associate Professor of Faculty of Science and Engineering at Saga University. He received his Ph.D from Kyoto University in 2005. He proposed a genetic algorithm to apply multi-agent systems. He was a researcher at ATR (Advanced Telecommunications Research Institute International) and NICT (National Institute of Information and Communications Technology). His research interest is an information system optimization method using block chain technology.