**ВВЕДЕНИЕ**

С++ - это мощный объектно-ориентированный язык программирования, который был разработан как расширение языка C с целью улучшить его удобство использования в крупных проектах и добавить возможности для создания сложных систем. С++ поддерживает многие ключевые концепции объектно-ориентированного программирования (ООП), такие как наследование, полиморфизм, инкапсуляция и связывание.

Метапрограммирование — это процесс написания программ или алгоритмов, которые генерируют или обрабатывают другие программы. В контексте языка программирования C++, метапрограммирование используется для выполнения вычислений и преобразований на уровне компилятора. Эти операции выполняются во время компиляции, что позволяет улучшить производительность результирующего кода, так как часть работы происходит перед фактическим запуском программы. В С++ метапрограммирование – мощное средство генерации кода, однако имеет ряд существенных недостатков, для решения которых предлагается разработка дополнительной системы генерации исходного кода поверх имеющихся механизмов метапрограммирования С++.

Производственная практика была пройдена в ФГБОУ ВО КубГУ с 02.02 по 15.02.2025 г. по адресу Краснодарский край г. Краснодар, ул. Ставропольская, 149 .

Цель практики – разработка средства генерации исходного кода.

Для достижения поставленной цели были решены следующие задачи:

1) Проведён анализ встроенных средств языка программирования С++ генерации исходного кода;

2) На основе анализа выявлены проблемы средств генерации кода;

3) Разработана спецификация дополнительного средства генерации исходного кода, решающего эти проблемы.

**1 Краткий обзор средств генерации кода в С++.**

Язык программирования С++ предоставляет широкий спектр инструментов для создания эффективного, производительного программного обеспечения. Так как С++ является компилируемым языком, то одним из самых полезных инструментов являются встроенные в язык средства генерации кода. С++ обладает широкими возможности в области порождающего программирования и метапрограммирования.

Метапрограммирование в C++ реализуется шаблонами и препроцессором. Препроцессор достался языку С++ в наследство от языка С, в то время как шаблоны – исключительно средство языка С++. Основное различие между шаблонами и препроцессором заключается в том, что препроцессор намного примитивнее, его цель – работать исключительно с исходным текстом, например, запрещать компиляцию каких-то участков кода или, наоборот, разрешать, он занимается подстановкой макросов и так далее, в то время как шаблоны намного более сложны и занимаются не столько работой с исходным текстом, сколько работой с кодом по написанному программистом шаблону, оптимизацией вычислений и даже исполнением целых участков написанного кода на этапе его компиляции (если это возможно), шаблоны неотрывны от языка С++. Несмотря на то, что С++ имеет столько средств порождающего программирования, все равно существуют задачи, которые встроенные средства порождающего программирования не способны решить. Проведём краткий обзор встроенных средств генерации кода.

**1.1 Препроцессор.**

Препроцессор C++ выполняет ряд операций по предварительной обработке текстового файла с программой: подстановка макросов, определение условий и управление инклюзиями (включение заголовочных файлов). Препроцессор С++ реализуется набором директив – коротких команд с аргументами, по своему синтаксису имеющим много общего с классической командной строкой.

Основные возможности препроцессора и директивы:

1. Включение заголовочных файлов ("#include") – эта директива используется для включения содержимого других текстовых файлов, обычно заголовочных файлов (.h или .hpp), прямо в текущий файл;

2. Определение макросов ("#define") - макросы позволяют заменять определенные последовательности символов другим набором символов. Также возможно определение макросов со списком аргументов ("#define" с параметрами) - это более продвинутое использование макросов, которое позволяет создавать параметризованные макросы, которые могут принимать переменное количество аргументов;

3. Управление условными компиляциями ("#ifdef", "#ifndef", "#else", "#endif") - эти директивы позволяют включать или исключать определенные части кода на основе значений макросов, определяемых другими директивами препроцессора или переменными окружения;

4. Удаление строк ("#line" и "#pragma") и другие специфические директивы - эти директивы позволяют настраивать синтаксический анализатор компилятора, указывая номер строки или изменяя поведение определенных функций препроцессора.

#include <iostream>

#define PI 3.14159265

int main() {

double radius = 5;

double area = PI \* radius \* radius;

std::cout << "Area: " << area << std::endl;

}

Листинг 1 – программный код, который показывает подключение заголовочного файла "<iostream>" и использование макроса "PI".

**1.2 Шаблоны.**

Шаблоны позволяют создавать функции или классы, которые могут работать с различными типами данных. Компилятор подставляет конкретные типы в шаблонах во время компиляции, что позволяет выполнять вычисления и преобразования на этапе компиляции.

Варьируемые шаблоны позволяют создавать функции и классы с переменным числом аргументов. Это особенно полезно для создания генераторов последовательностей или обработки неизвестного количества входных данных. Ниже представлен пример варьируемого шаблона.

template <typename T, typename... Args>

T add(T first, Args... args) {

if constexpr(sizeof...(args) == 0) return first;

else return first + add(args...);

}

int main() {

std::cout << add(1, 2, 3, 4) << std::endl;

}

Листинг 2 – пример использования варьируемых шаблонов.

Шаблоны позволяют выполнять различные вычисления во время компиляции, что может повысить эффективность кода.

template<int N>

struct factorial {

static constexpr int value = N \* factorial<N-1>::value;

};

template<>

struct factorial<0> { // базовый случай

static constexpr int value = 1;

};

int main() {

constexpr int result = factorial<5>::value;

}

Листинг 3 – пример использования шаблонного метапрограммирования для подсчета размера последовательности.

Метапрограммирование позволяет проверять и анализировать свойства типов, что полезно для создания гибких шаблонных структур. В примере ниже показано, как можно использовать метафункцию "std::enable\_if<bool B, typename T>" заголовочного файла "type\_traits", которая позволяет сделать конкретную специализацию шаблона доступной или недоступной, в зависимости от того, выполняется ли некоторое условие во время компиляции.

template <typename T>

typename std::enable\_if<std::is\_integral<T>::value, void>::type

integral\_type\_func(T val) {

// функция определена только для целочисленных типов

}

int main() {

integral\_type\_func(42); // работает

// integral\_type\_func(3.14); // не компилируется

}

Листинг 4 – пример шаблонной функции.

**1.3 Проблемы.**

Язык программирования С++ известен тем, что в его состав включено очень много различных конструкций, с выпуском каждого нового стандарта язык планомерно усложняется. Это следствие одного из принципов этого языка программирования: код, написанный под ранний стандарт, должен без ошибок компилироваться и работать на новом стандарте. Этот принцип одновременно и благо, и зло, которое как привело к морю очень полезных технологий и "синтаксического сахара" в языке, так и заставило обрасти язык таким же морем откровенно устаревших и лишних конструкций, повышающих порог вхождения в язык и ухудшающих читаемость кода. Действие этого принципа затронуло и шаблоны: первый приходящий на ум пример подобного рода – это все метафункции заголовочного файла "type\_traits" (который специализируется на метафункциях и метаобъектах специально для шаблонов), добавленные в С++11, а позже в С++14 получили своих продолжателей с приставками "\_t" и "\_v". На листинге ниже показана суть проблемы на примере "std::enable\_if":

template<bool B, class T = void>

struct enable\_if {};

template<class T>

struct enable\_if<true, T> { typedef T type; };

template<bool B, class T = void>

using enable\_if\_t = typename enable\_if<B,T>::type;

Листинг 5 – определение "std::enable\_if" и "std::enable\_if\_t" в "type\_traits".

Разница между двумя конструкциями в том, что для того, чтобы получить тип "std::enable\_if", нужно обратиться к его типу-члену type, тогда как "std::enable\_if\_t" – это и есть этот тип. И существование абсолютно эквивалентных конструкций, одна из которых создана для того, чтобы использоваться вместо другой, в составе языка для С++ – обыденность. В этом языке много старых ненужных конструкций, мусорящих код, которые язык продолжает тянуть за собой.

Однако подобные проблемы шаблонов как средства генерации кода – следствие общих фундаментальных проблем языка С++, решение которых в рамках данного языка без его серьезной перестройки невозможно, тогда как препроцессор, доставшийся языку от С, не так тесно взаимодействует с языком и не является его неотрывной частью. Препроцессор – очень мощный инструмент, настолько, что, пользуясь исключительно препроцессором, можно вполне успешно симулировать немалую часть функционала шаблонов (хотя итоговый код будет совершенно нечитаем и абсолютно неудобен). С момента внедрения препроцессора в стандарт языка он фактически не претерпел больших изменений. Большинство задач, которые не может решить препроцессор, в целом, получается решать у шаблонов, однако у этого "тандема" встроенных средств генерации кода все равно остаются нерешаемые задачи и недостатки.

У препроцессора имеются следующие недостатки:

- Отсутствие возможности шаблонизации директив препроцессора – нельзя создать функции из директив препроцессора, в которых аргументы директив зависели бы от аргументов функции;

- Отсутствие возможности сравнения значений макросов лексикографически;

- Отсутствие циклов – циклы бы пригодились в инициализации константных и глобальных объектов, в генерации множества похожих объектов, функций, структур, без необходимости заниматься этим вручную.

Шаблоны не покрывают данные недостатки, так как предназначены для работы с исходным кодом, а не с набором лексем.

**2 Спецификация средства генерации исходного кода.**

На основе изученного материала разработана спецификация дополнительного средства генерации кода. Данное средство генерации кода будет использоваться совместно с уже включенными в состав С++, позволяя решать те задачи, которые встроенные средства не могут решить. Новое средство генерации кода является препроцессором, который обрабатывает исходный текст перед стандартным препроцессором С++, и позволяет генерировать не только стандартный код С++, но и код препроцессора С++.

Препроцессор получил название MacroFunc, так как его основа и фундамент - "макрофункции". Макрофункция – это программный код генерации кода, в который в разных ситуациях могут передаваться различные аргументы, являющиеся макросами препроцессора С++, литералами, лексемами и так далее. Может как определяться непосредственно в обрабатываемом файле, так и в отдельном файле.

Препроцессор MacroFunc, как и стандартный препроцессор С++, реализуется посредством директив, при помощи которых он обрабатывает лексемы, создаёт и изменяет переменные (о них речь пойдёт ниже) и так далее. MacroFunc определяет следующие директивы:

- macrofunc, endmacrofunc, integrate - директивы создания макрофункций;

- if, elif, else, endif - условные директивы;

- error, warning - директивы прдеупреждения и ошибок;

- var - директива создания переменной;

- for, endfor, foreach, endforeach - директивы циклов.

Директивы MacroFunc синтаксически подобны директивам препроцессора С++, однако перед именем каждой директивы помещается не один, а два знака #, чтобы отличать директивы. Рассмотрим их подробнее.

**2.1 Директивы определения макрофункции.**

Макрофункция - фундаментальное средство генерации кода. Препроцессор обрабатывает только содержимое макрофункций, все за пределами макрофункций он игнорирует. Внутри макрофункции разрешены классические комментарии из С++: "//", "/\* \*//'.

##macrofunc <название макрофункции>(<список аргументов>)

// действия макрофункции

##endmacrofunc

// ...

// при вставке следующей строки в необходимом месте отрабатывается макрофункция

##integrate <название макрофункции>(<список аргументов>)

Листинг 6 – шаблон создания макрофункции.

Весь текст внутри тела макрофункции при ее обработке будет помещен на место вызова директивы "##integrate", в том числе и директивы препроцессора. При этом будет осуществлена подстановка аргументов макрофункции. Примеры макрофункций будут представлены ниже при рассмотрении других директив.

**2.2 Директива создания переменных (var).**

Переменная - это некоторое значение, которое видно лишь внутри макрофункции (то есть, при обработке макрофункции данной переменной в сгенерированном коде не будет).

##var(<имя переменной>, <содержимое>)

Листинг 10 – шаблон "##var".

В MacroFunc переменная может иметь следующие типы данных:

- Пустой - ничего не содержит;

- Одно слово - набор букв и цифр в одно слово:

- Число - дробное или целое;

- С-строка - вся строка будет считаться одним словом; под С-строками понимаются все строки символов, которые оговорены по стандарту (например, "wchar\_t");

- Лексема - просто какое-то слово;

- Имя другого макроса;

- Список - набор букв и цифр в несколько слов.

Над переменной можно производить следующие операции:

- "+","-","/","\*","%" - арифметические, можно производить, только если переменная является числом;

- "#" - превратить в С-строку;

- "##" - объединить лексемы.

##macrofunc tmp()

##var(tmp, 0)

#define MACRO1 tmp

##var(tmp, tmp + 3)

#define MACRO2 tmp

##var(tmp, tmp + MACRO2)

#define MACRO3 tmp

##var(cstr, #MACRO3)

#define MACRO4 cstr

##var(cstr, L ## #MACRO3)

#define MACRO5 cstr

##endmacrofunc

// ...

##integrate tmp()

/\*

#define MACRO1 0

#define MACRO2 3

#define MACRO3 6

#define MACRO4 "6"

#define MACRO5 L"6"

\*/

Листинг 7 – Пример использования "##var".

**2.3 Директивы условий (if, elif, else).**

Используются для сравнения аргументов макрофункции с константами и переменными. В примере ниже исполняются два условия: "macro\_name" является макросом "RAND\_MACRO", но и также равняется 1; будет исполнено то условие, которое было объявлено первым.

##macrofunc creater(macro\_name)

##if macro\_name == IO\_MACRO

#define MACRO 0

##elif macro\_name == RAND\_MACRO

#define MACRO 1

##elif macro\_name == 100

#define MACRO 2

##else

#define MACRO 3

##endif

##endmacrofunc

//...

#define RAND\_MACRO 100

##integrate creater(RAND\_MACRO)

// #define MACRO 1

Листинг 8 – пример использования условных директив.

Для того, чтобы определить, какой тип данных у переменной, в условиях ##if, ##elif можно использовать спецфункции:

- "\_\_IS\_VOID\_\_" - возвращает истину, если переменная пустой;

- "\_\_IS\_INT\_\_" - возвращает истину, если переменная - целое число;

- "\_\_IS\_FLOAT\_\_" - возвращает истину, если переменная - дробное число;

- "\_\_IS\_CSTR\_\_" - вовзращает истину, если переменная - С-строка;

- "\_\_IS\_MACRO\_\_" - возвращает истину, если переменная содержит другую переменную;

- "\_\_IS\_WORD\_\_" - возвращает истину, если переменная - одно слово;

- "\_\_IS\_LIST\_\_" - возвращает истину, если переменная - список.

Каждая спецфункция принимает один аргумент - имя переменной.

**2.4 Директивы предупреждения и ошибки (error, warning).**

MacroFunc также поддерживает директивы предупреждения и ошибки для удобства генерации кода, который не должен компилироваться или должен компилироваться с предупреждениями из-за значений каких-либо макросов.

##error <условие, которое в случае истины вызывает ошибку>

##warning <условие, которое в случае истины вызывает предупреждение>

Листинг 9 – шаблон "##error" и ##warning".

В случае, если не поддерживается директива препроцессора "#warning" (либо нет никакой замены), "##warning" будет недоступен.

##macrofunc int\_is\_4\_bytes(sizeof)

##error sizeof == 4

##endmacrofunc

// ...

##integrate int\_is\_4\_bytes(sizeof(int))

/\*

#if sizeof(int) == 4

#error "sizeof(int) == 4"

#endif

\*/

Листинг 10 – пример использования "##error".

**2.5 Директивы циклов (for).**

"##for" - классический цикл, работающий с числами.

##for(<имя переменной>, <начальное значение>, <конечное значение>, <что будет прибавляться>)

// действия

##endfor

Листинг 11 – шаблон "##for"

Переменная должна быть числом (может быть и целым, и дробным). На каждой итерации к переменной всегда прибавляется какое-то константное число; поэтому для вычитания перед числом нужно поставить минус. Константа может быть как литералом, так и другой переменной.

Цикл будет продолжать работать, пока переменная не будет строго равна конечному значению. Для дробных чисел работает исключение: у переменной и конечного значения должны быть равны все разряды конечного значения, например, если переменная равна 0.3599 или 0.35001, а конечное значение 0.35, то то все разряды после цифры 5 отбрасываются, поэтому считается, что переменная и конечное значение равны.

##macrofunc create\_functions(count)

int global[count];

##for(VAR, 0, count, 1)

void function ## VAR (void) { global[VAR]++; }

##endfor

void (\*global\_functions[count])(void) = {

##for(VAR, 0, count, 1)

function ## VAR

##if VAR != count - 1

,

##endif

##endfor

};

#endmacrofunction

// ...

#integrate create\_functions(3)

/\*

int global[3];

void function0(void) { global[0]++; }

void function1(void) { global[1]++; }

void function2(void) { global[2]++; }

void (\*global\_functions[count])(void) = {

function0,

function1,

function2

};

\*/

Листинг 12 – пример создания множества функций и массива, хранящего указатели на функции.

**2.6 Директивы циклов (foreach).**

Если for работает с численными переменными и диапазонами, то for\_each - со списками лексем и макросов.

##foreach(<имя переменной>, <список>)

// действия

##endforeach

Листинг 13 – шаблон "foreach".

Список - это переменная, содержащая какие-либо лексемы через пробельные символы. Таким образом, любую переменную можно понимать как список с произвольным количеством элементов.

Над списком также можно использовать некоторые спецфункции:

- \_\_SIZE\_LIST\_\_ - возвращает количество слов в списке;

- \_\_IS\_END\_LIST\_\_ - истина, если аргумент - последний элемент в списке (если список пустой - всегда ложь);

- \_\_IS\_BEGIN\_LIST\_\_ - тоже самое, но для начального элемента;

- \_\_IS\_ITEM\_LIST\_\_ - истина, если аргумент есть в списке;

- \_\_INDEX\_\_ - возвращает элемент списка по индексу (возвращает пустоту, если индекс расположен за границами списка).

Так как любая переменная является списком, то эти функции можно использовать и над не списками.

##macrofunc create\_functions(row\_of\_names)

##foreach(VAR, row\_of\_names)

void function ## VAR (void) { std::cout << VAR << std::endl; }

##endforeach

void (\*global\_functions[\_\_SIZE\_LIST\_\_])(void) = {

##foreach(VAR, row\_of\_names)

function ## VAR

##if ! \_\_IS\_END\_LIST\_\_(row\_of\_names, VAR)

,

##endif

##endforeach

};

#endmacrofunction

// ...

#define ROW 0 1 2

#integrate create\_functions(ROW)

/\*

void function0(void) { std::cout << 0 << std::endl; }

void function1(void) { std::cout << 1 << std::endl; }

void function2(void) { std::cout << 2 << std::endl; }

void (\*global\_functions[3])(void) = {

function0,

function1,

function2

};

\*/

Листинг 14 – пример использования "##foreach".

Каждый элемент списка является лексемой, к которой можно обратиться как к переменной и узнать её тип.

##macrofunc is\_macro(list)

##foreach(VAR, list)

##if ! \_\_IS\_MACRO\_\_(row\_of\_names, VAR)

#error "is not macro: " #VAR

##endif

##endforeach

#endmacrofunction

// ...

#define ROW EOF NULL stdout 400

#integrate is\_macro(list)

/\*

#error "is not macro: 400"

\*/

Листинг 15 – пример работы с типами в списке.

**2.7 Преимущества и отличия от аналогов**

Разработанное средство генерации кода обладает следующими преимуществами и отличиями от аналогов:

- MacroFunc предоставляет средства для шаблонизирования не только одного лишь исходного кода, что делают стандартные шаблоны С++, но и целых препроцессорных блоков, позволяя шаблонизировать и сократить объемные куски препроцессорного кода;

- В отличие от стандартного препроцессора С++, который в качестве переменных предоставляет макросы, не имеющие типа данных и являющиеся лишь наименованием лексем и списков лексем, MacroFunc предлагает фактически полноценно работать с переменными и как с числами, и как со стандартными лексемами, и как со списками лексем, определяет различные операции над переменными, от лексикографического сравнения наименований переменных до арифметических операций над значениями переменных. При этом MacroFunc поддерживает все возможности стандартного препроцессора, которые он предоставлял для макросов: определение, является ли лексема макросом, превращение лексемы или списка лексем в строку, объединение лексем и так далее;

- MacroFunc предоставляет возможность работать с циклами, которые позволяют создавать много подобных объектов и функций, инициализировать массивы, создавать макросы и так далее;

- MacroFunc разработан для языка программирования С++, однако он, подобно препроцессору С++, не является неотрывной частью языка и является автономным инструментом, и в каком-то смысле он даже больше связан со стандартным препроцессором, так как аргументы макрофункций, являющиеся макросами, преобразуются в переменные MacroFunc. Однако даже так MacroFunc можно без каких-либо изменений интегрировать в другие языки программирования;

- MacroFunc, подобно стандартному препроцессору С++, обладает минималистичным и простым инструментарием, который довольно легко изучить и использовать.

**СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ**

1 - Огнева М.В., Кудрина Е.В. Программирование на языке C++. Практический курс: учебное пособие для вузов. - М.: Юрайт, 2022. - 335 с.

2 - Вандевурд, Дэвид, Джосаттис, Николаи М., Грегор, Дуглас. В17. Шаблоны C++. Справочник разработчика, 2-е изд.: Пер. с англ. — СпБ.: ООО “Альфа-книга”, 2018. — 848 с.: ил. — Парал. гит. англ. ISBN 978-5-9500296-8-4 (рус.)

3 - Шилдт, Герберт. Ш57. С: полное руководство, классическое издание. : Пер. с англ. – М. : ООО «И.Д. Вильямс», 2017. – 704 с. : ил. – Парал. Тит. Англ.