**Глава 2. Этапы обработки кода программы на языке С++**

**2.1 Директивы препроцессора**

Препроцессор – это текстовый процессор, управляющий текстом файла исходного кода в ходе первого этапа трансляции. Препроцессор не анализирует исходный текст, а разбивает его на маркеры для обнаружения вызовов макросов. Хотя компилятор обычно вызывает препроцессор при первом проходе, препроцессор можно также вызвать отдельно для обработки текста без его компиляции.

Директива (указание) по тому, как ее используют, похожа на команду, так как также используется для описания некоторых конструкций языка программирования, то есть указаний компилятору особенностей обработки при компиляции. В языке программирования С++ встроена поддержка препроцессора. Строки в исходном коде, которые должны быть обработаны препроцессором в виде #define и #include, называются препроцессорными директивами.

В языке программирования C++ препроцессор – это часть компилятора, управляющая формированием исходного кода в объектный. Препроцессор имеет набор команд, называемых директивами препроцессора, а также маркеры, где находятся "команды" – макросы. Маркер разделяет код на части, а также служит флажком для нахождения макросов. Директива же является командой, которая образует сценарий компилятора [7].

Директивы препроцессора, такие как #define определяет идентификатор и последовательность символов, которые будут заменять этот идентификатор в программе, а #ifdef позволит вставлять содержимое других файлов в файл исходного кода или отключать компиляцию части файла путем удаления разделов текста. Директивы в исходном файле указывают препроцессору на необходимость выполнения определенных действий. Например, препроцессор может заменять токены в тексте, вставлять содержимое других файлов в файл исходного кода или отключать компиляцию части файла путем удаления разделов текста. Строки препроцессора распознаются и выполняются до расширения макросов. Таким образом, если макрос разворачивается в нечто, похожее на команду препроцессора, он не распознается препроцессором. Инструкции препроцессора используют тот же набор символов, что и инструкции исходного файла, за исключением того, что не поддерживаются. Набор символов в операторах препроцессора совпадает с кодировкой выполнения. Препроцессор также распознает отрицательные значения символов [15].

Препроцессор распознает следующие директивы: #define, #endif, #import, #include, #pragma, #undef, #ifndef.

Знак решетки ( # ) должен быть первым символом, не пробелом, в строке, содержащей директиву. символы пробела могут находиться между знаком номера и первой буквой директивы. Некоторые директивы содержат аргументы или значения. Любой текст, следующий за директивой (кроме аргумента или значения, который является частью директивы), должен предшествовать разделителю однострочных комментариев ( // ) или быть заключенным в разделители комментариев ( / \* \* / ). Строки, содержащие директивы препроцессора, могут быть продолжены непосредственно перед маркером конца строки с обратной косой чертой ( \ ). Директивы препроцессора могут находиться в любом месте исходного файла, но они применяются только к остальной части исходного файла после их появления.

Текст программы C++ состоит из маркеров и пробелов. Токен – это наименьший элемент на С++, который имеет значение для компилятора.

Средство синтаксического анализа C++ распознает следующие типы маркеров:

1. Ключевые слова

2. Идентификаторы

3. Числовые, логические и указатели литералы

4. Строковые и символьные литералы

5. Определяемые пользователем литералы

6. Инструкции

7. Символы пунктуации

Маркеры обычно разделяются пробелами, которые могут быть одним или несколькими:

1. Пустые значения

2. Символы горизонтальной и вертикальной табуляции

3. Символы перевода строки

4. Веб-каналы форм

5. Комментарии

Стандарт С++ задает базовый исходный набор символов, который может использоваться в исходных файлах. Для представления символов вне этого набора можно указывать дополнительные символы, используя универсальные имена символов. Реализация MSVC позволяет использовать дополнительные символы. Базовый набор исходных символов состоит из 96 символов, которые могут использоваться в исходных файлах. Этот набор включает символ пробела, горизонтальной и вертикальной табуляции, управляющие символы перевода страницы и новой строки, а также следующий набор графических символов: a b c d e f g h i j k l m n o p q r s t u v w x y z A B C D E F G H I J K L M N O P Q R S T U V W X Y Z 0 1 2 3 4 5 6 7 8 9 \_ { } [ ] # ( ) <> % : ; . ? \* + - / ^ & | ~ ! = , \ " ‘

**2.2 Основные директивы препроцессора**

Макросы в C++ представляют собой инструкции, которые препроцессор обрабатывает перед компиляцией кода. Они позволяют осуществлять текстовую замену в исходном коде, что может быть полезно для автоматизации и упрощения некоторых задач (рисунок 2.1).

|  |
| --- |
|  |
| Рисунок 2.1 – Пример использования макроса |

Токен – это последовательность символов, представляющая то, что считается единицей измерения, например, число или оператор (рисунок 2.2).

|  |
| --- |
|  |
| Рисунок 2.2 – Пример токенов C++ |

Директива #define создает макрос, который представляет собой ассоциацию идентификатора или параметризованного идентификатора со строкой токена. После определения макроса компилятор может подставить строку токена для каждого обнаруженного идентификатора в исходном файле [11].

Синтаксис: #define токен идентификатора значение (если имеется).

А #define без строки токена удаляет вхождения идентификатора из исходного файла. Идентификатор остается определенным и может быть проверен с помощью #if defined директивы и #ifdef. на рисунках 2.3, 2.4 представлены примеры использования.

|  |
| --- |
|  |
| Рисунок 2.3 – Пример использования директивы #define |
|  | |
| Рисунок 2.4 – Пример использования директивы #undef | |

Директива #if с директивами #elif, #else и #endif управляют компиляцией частей исходного файла. Если выражение после #if имеет ненулевое значение, группа строк сразу после директивы #if сохраняется в записи преобразования [14]. Каждая директива #if в исходном файле должна соответствовать закрывающей директиве #endif [16]. Между директивами #if и #endif может использоваться любое число директив #elif, но допускается не более одной директивы #else. Директива #else должна быть последней директивой перед #endif.

Данные директивы нужны для различных отладочных действий, которые не стоит реализовывать через прописывание кода. На рисунке 2.5, представлен пример использования.

|  |
| --- |
|  |
| Рисунок 2.5 – Пример использования директив #if, #else, #endif |

Директива #error выдает указанное пользователем сообщение об ошибке во время компиляции, а затем завершает компиляцию: #error token-string.

Эта директива наиболее полезна во время предварительной обработки для уведомления разработчика несогласованности программы или нарушения ограничений. Следующий пример демонстрирует ошибки при обработке во время предварительной обработки [17].

Директивы #ifdef, #ifndef так же, как и #if используются для проверки, но не с условием, а переменной, которую занесли как макрос. На рисунке 2.6 представлен пример конструкции.

|  |
| --- |
|  |
| Рисунок 2.6 – Пример использования директив #ifdef, #ifndef |

Если переменная DEBUG распознана, то в консоль выводится информация, если же переменной не было найдено, но данный блок не активен.

Директива #include указывает препроцессору включить содержимое указанного файла в точку, где отображается директива.

Синтаксис:

#include "путь – спецификация"

#include <путь – спецификация>

Если имя файла, заключенное в двойные кавычки, является неполным указанием пути, препроцессор сначала выполняет поиск в каталоге родительского файла. Родительский файл – это файл, содержащий #include директиву. Например, если включить файл с именем file2 в файл с именем file1, то файл file1 будет родительским. #include отвечает за подключения различного рода библиотек, написанных другими разработчиками, которые позволяют реализовывать в коде новые блоки и функционал [22].

Директива #import используется для включения сведений из библиотеки типов. Содержимое библиотеки типов преобразовано в классы C++, в основном описывающие интерфейсы модели COM.

Синтаксис:

#import "filename" [атрибуты]

#import <имя файла> [атрибуты] filename

Директива #line указывает препроцессору установить сообщаемые значения компилятора для номера строки и имени файла в заданный номер строки и имя файла.

Синтаксис: #line digit-Sequence ["имя\_файла"].

Номер строки и (необязательно) имя файла используется компилятором для указания на ошибки, которые он обнаруживает во время компиляции. Номер линии обычно указывает на текущую строку входных данных, а имя файла – на текущий входной файл. Номер строки увеличивается на единицу после обработки каждой строки. Значением последовательности цифр может быть любая целочисленная константа в диапазоне от 0 до 2147483647 включительно. для маркеров предварительной обработки можно использовать замену макросов, но результат должен иметь правильный синтаксис. Имя файла может быть любым сочетанием символов и должно быть заключено в двойные кавычки (""). Если параметр filename опущен, то предыдущее имя файла остается неизменным. На рисунке 2.7 представлен пример использования [20].

|  |
| --- |
|  |
| Рисунок 2.7 – Пример использования директивы #line |

Директива #pragma исключает проблему повторного включения. При вызове директив #include в конкретных двух файлах можно вызвать два файла друг у друга, что приведет к ошибке, глубиной 1024 вызова и произойдет исключение. Директива #pragma решает эту проблему, проверяя, не вызывался ли файл до этого. Если данный #include уже был, то далее повторного подключения происходить уже не будет. Так же обойти данную проблему можно через условную компиляцию (#if, #elif, #else и #endif).

Синтаксис: #pragma строка\_токена.

Ключевое слово \_\_pragma.

Компилятор также поддерживает ключевое слово, определенное \_\_pragma корпорацией Майкрософт, которое имеет те же функции, что и директива #pragma. Разница заключается в том, что ключевое \_\_pragma слово является пригодным для использования встроенным в определении макроса. Пример представлен на рисунке 2.8.

|  |
| --- |
|  |
| Рисунок 2.8 – Пример использования директивы #pragma |

**2.3 Компиляция программы**

Состав компилятора g++:

срр – препроцессор

as – ассемблер

g++ – компилятор

ld – линкер

Препроцессор – это программа, которая преобразует исходный код в код понятный компилятору.

Ассемблер – транслятор программы из текста на языке ассемблера в программу на машинном языке.

Компилятор – это программа, которая переводит исходный код на языке программирования в машинный код.

Линкер – программа, которая производит компоновку: принимает на вход один или несколько объектных модулей и собирает по ним исполняемый модуль.

Для чего нужно компилировать исходные файлы?

Исходный С++ файл – это код, но его невозможно запустить как программу или использовать как библиотеку. Поэтому каждый исходный файл требуется скомпилировать в исполняемый файл, динамическую или статическую библиотеки.

Исходный файл для работы – driver.cpp, представлен на рисунке 2.9.

|  |
| --- |
|  |
| Рисунок 2.9 – Исходный файл driver.cpp |

Препроцессинг – самая первая стадия компиляции программы. Препроцессор (данный термин упоминался ранее)– это макропроцессор, который преобразовывает программу для дальнейшего компилирования.

Хэдер (header – заголовочный файл) – файл, содержимое которого автоматически добавляется препроцессором в исходный текст в том месте, где располагается некоторая директива.

Хэдеры, включенные в программу с помощью директивы #include, рекурсивно проходят стадию препроцессинга и включаются в выпускаемый файл. Каждый хэдер может быть подключен несколько раз, поэтому обычно используются специальные директивы препроцессора, которые предотвращают циклические зависимости.

Получим препроцессированный код в выходной файл driver.ii (прошедшие через стадию препроцессинга файлы имеют расширение .ii), используя флаг -Е, который сообщает компилятору, что компилировать (об этом далее) файл не нужно, а только провести его препроцессинг: g++ -Е driver.cpp -o driver.ii. Взглянув на тело функции main в новом сгенерированном файле, можно заметить, что макрос RETURN был заменен (рисунок 2.10).

|  |
| --- |
|  |
| Рисунок 2.10 – Сгенерированный файл driver.ii |

В новом сгенерированном файле также можно увидеть огромное количество новых строк, это различные библиотеки и хэдер #iostream.

На данном шаге g++ выполняет свою главную задачу – компилирует, то есть преобразует полученный на прошлом шаге код без директив в ассемблерный код. Это промежуточный шаг между высокоуровневым языком и машинным (бинарным) кодом.

Ассемблерный код – это доступное для понимания человеком представление машинного кода.

Используя флаг -S, который сообщает компилятору остановиться после стадии компиляции, получим ассемблерный код в выходном файле driver.s: $ g++ -S driver.ii -o driver.s

Можно все также посмотреть полученный результат. Но для того, чтобы машина поняла код, требуется преобразовать его в машинный код, который получится на следующем шаге.

Так как процессоры исполняют команды на бинарном коде, необходимо перевести ассемблерный код в машинный с помощью ассемблера.

Ассемблер преобразовывает ассемблерный код в машинный код, сохраняя его в объектном файле.

Объектный файл – это созданный ассемблером промежуточный файл, хранящий кусок машинного кода. Этот кусок машинного кода, который еще не был связан вместе с другими кусками машинного кода в конечную выполняемую программу, называется объектным кодом.

Далее возможно сохранение данного объектного кода в статические библиотеки для того, чтобы не компилировать данный код снова.

Получим машинный код с помощью ассемблера (as) в выходной объектный файл driver.o: $ asdriver.s -o driver.o

Но на данном шаге нужно соединить объектные файлы в единый исполняемый файл с помощью компоновщика (линкера). Поэтому следующая стадия – компоновка.

Компоновщик (линкер) связывает все объектные файлы и статические библиотеки в единый исполняемый файл, который сможем запустить в дальнейшем. для того, чтобы понять, как происходит связка, следует рассказать о таблице символов.

Таблица символов – это структура данных, создаваемая самим компилятором и хранящаяся в самих объектных файлах. Таблица символов хранит имена переменных, функций, классов, объектов и т.д., где каждому идентификатору (символу) соотносится его тип, область видимости. Также таблица символов хранит адреса ссылок на данные и процедуры в других объектных файлах. Именно с помощью таблицы символов и хранящихся в них ссылок линкер будет способен в дальнейшем построить связи между данными среди множества других объектных файлов и создать единый исполняемый файл из них.

Получим исполняемый файл driver: $ g++ driver.o -o driver // также тут можно добавить другие объектные файлы и библиотеки.

Последний этап, который предстоит пройти программе – вызвать загрузчик для загрузки нашей программы в память. На данной стадии также возможна загрузка динамических библиотек. Итоговый результат всех проделанных шагов представлен на рисунке 2.11.

|  |
| --- |
|  |
| Рисунок 2.11 – Итоговый результат |

**2.4 Пространства имен**

Пространства имён служат для логического группирования объявлений и ограничения доступа к ним. Чем больше программа, тем более актуально использование поименованных областей. Простейшим примером применения является отделение кода, написанного одним человеком, от кода, написанного другим. При использовании единственной глобальной области видимости формировать программу из отдельных частей очень сложно из-за возможного совпадения и конфликта имен. Использование поименованных областей препятствует доступу к не нужным средствам. Объявление пространства имен имеет формат, представленный на рисунке 2.12.

|  |
| --- |
|  |
| Рисунок 2.12 – Формат объявления |

После объявления пространства имен можно обращаться к его элементам. Имя пространства имен можно не указывать, если это имя объявлено в этом же пространстве имен. То, как за пространством имен выглядит обращение, представлено на рисунке 2.13.

|  |
| --- |
|  |
| Рисунок 2.13 – Формат объявления |

Пример объявления и использования пространства имён представлен на рисунке 2.14.

|  |
| --- |
|  |
| Рисунок 2.14 – Пример объявления и использования пространства имен |

С целью повышения читабельности программы и избегания постоянного указания имени пространства имен при доступе к его элементам используется директива using. Эта директива может применяться в двух случаях, которые представлены на рисунке 2.15.

|  |
| --- |
|  |
| Рисунок 2.15 – Случаи применения директивы using |

В первом случае подключается все пространство имен. После этого можно использовать все элементы данного пространства имён без явного указания области.

Во втором случае подключается отдельный элемент пространства имен. После этого можно использовать имя без явного указания области.

Пример доступа ко всем элементам пространства имен представлен на рисунке 2.16.

|  |
| --- |
|  |
| Рисунок 2.16 – Пример доступа ко всем элементам |

Пример доступа к отдельному элементу в пространстве имен представлен на рисунке 2.17.

|  |
| --- |
|  |
| Рисунок 2.17 – Пример доступа к отдельному элементу |

Результат выполнения программы: z = 2.

В вышеприведенном примере подключение методов осуществляется директивой using с помощью строк: usingOperations::Sub2;

После этого эти методы могут быть вызваны непосредственно без использования префикса operations::.

Если попытаться вызвать неподключенный метод Add2(), то компилятор выдаст сообщение об ошибке: identifier Add2 isundefined.

Глобальное пространство имен определяет область видимости самого высокого уровня. В данной области видимости могут быть объявлены глобальные переменные, типы, функции и т.д. для доступа к глобальному пространству имен необходимо использовать оператор расширения области видимости ::. Если в глобальном пространстве имен и в некотором локальном пространстве есть совпадающие имена, то для доступа к глобальному имени использование оператора :: обязательно, на рисунке 2.18 представлен пример использования оператора.

|  |
| --- |
|  |
| Рисунок 2.18 – Пример доступа к отдельному элементу |

Результат выполнения программы: Pi = 3.1415, MathItems::Pi = 3.1415.

Если в вышеприведенном коде попробовать реализовать вызов ShowPi(), то компилятор генерирует ошибку: ambiguouscalltooverloadedfunction. Это логично, поскольку неизвестно, к какой функции ShowPi() происходит обращение.

Пространство имен может объявляться неоднократно, причем последующие объявления рассматриваются как расширения предыдущих. Таким образом, пространство имен может объявляться и изменяться за рамками одного файла, на рисунке 2.19 представлен пример повторного объявления пространства имен.

|  |
| --- |
|  |
| Рисунок 2.19 – Пример повторного объявления пространства имен |

В объявлении поименованной области могут присутствовать как объявления, так и определения. Логично помещать в нее только объявления, а определять их позднее с помощью имени области и оператора доступа к области видимости ::, пример представлен на рисунке 2.20.

|  |
| --- |
|  |
| Рисунок 2.20 – Пример определения переменной в пространстве имен |

В классическом С++ базовым пространством имен есть пространство имен std. В этом пространстве имен реализуются средства стандартной библиотеки С++. В ней реализовано тысячи функций, которые полезны при разработке любых программ. В предыдущих версиях С++ вся стандартная библиотека была в глобальном пространстве имен. Помещение стандартной библиотеки в пространство имен std существенно снижает возможность конфликтов имен. Для подключения пространства имен std используется строка: using namespace std;

Также возможно обращение к элементам пространства имен std указанием полного имени, например: double a = std: :sqrt (9);

Пространства имен могут быть вложенными. Это означает, что одно пространство имен может включать другое пространство имен. В этом случае к вложенному пространству имен и его составляющим можно получить доступ обычным способом с помощью оператора ::. На рисунке 2.21 представлен пример вложенных пространств имен.

|  |
| --- |
|  |
| Рисунок 2.21 – Вложенные пространства имен |

**2.5 Локализация и настройка локали программы**

Локализация – это процесс адаптации продукта к конкретному региону или стране.

Локализация может влиять на:

1. Перевод

2. Изображения, иконки, цвета

3. Названия бренда

4. Числа

5. Валюты и цены

6. Время и даты

7. Величины измерения

8. Телефонные номера

9. Почтовые индексы

Пример локализации представлен на рисунке 2.22.

|  |
| --- |
|  |
| Рисунок 2.22 – Локализация программы |

Локаль – это набор параметров, определяющий региональные настройки пользовательского интерфейса.

Локаль содержит информацию о том, как интерпретировать и выполнять определенные операции ввода/вывода и преобразования с учетом географического расположения и специфики языков в определённых условиях.

Может быть определена:

1. По данным, заданным пользователем

2. По геолокации

3. Через конфигурацию системы

Итак, для задания локали в С++ существует функция setlocale, которая устанавливает или извлекает языковой стандарт времени выполнения.

Она определяется двумя параметрами: category и locale.

Параметр category указывает части информации о языковом стандарте программы, которые подвергаются влиянию.

Параметр locale ссылается на расположение (страна, регион, язык), для которого можно настраивать определенные аспекты программы.