**Программа** – это данные, предназначенные для управления конкретными компонентами системы обработки информации (СОИ) в целях реализации определенного алгоритма

* **Для программиста** программа кажется активной — она выполняет действия, управляет вычислениями.
* **Для процессора** же команды программы — это просто набор данных, которые он считывает, интерпретирует и выполняет шаг за шагом.

**Программное обеспечение** = программа + документация. В отличие от разовых программ, написанных, например, для одного эксперимента, ПО разрабатывается с расчётом на долговременное и многократное применение разными пользователями.

Ряд необходимых свойств ПО:

- **Необходимость документирования.** Программы становятся ПО только при наличии документации. Конечный пользователь не может работать, не имея документации.

- **Эффективность** (Программное обеспечение, рассчитанное на многократное использование пишется и отлаживается один раз, а выполняется многократно. Выгодно переносить затраты на этап производства ПО)

- **Надежность** (Тестирование программы при всех допустимых спецификациях входных данных. Защита от неправильных действий пользователя. Защита от взлома.

Появление ошибок любого уровня не должно приводить к краху системы. Ошибки должны вылавливаться диагностироваться и (если их невозможно исправить) превращаться в корректные отказы

1. **Системные** **структуры** данных должны сохраняться **безусловно**
2. Сохранение целостности **пользовательских** данных **желательно**)

- **Возможность сопровождения** (адаптация ПО к конкретным  
условиям применения, устранение ошибок, модификация)

**Системная программа** – программа, предназначенная для поддержания работы СОИ или повышения эффективности её использования

Например: операционные системы, файловые системы, драйверы, утилиты, системы программирования

**Прикладная программа** – программа, предназначенная для решения задачи в определенной области применения СОИ

Например: Текстовые редакторы, табличные редакторы, графические редакторы, браузеры, игры

**Система** – единое целое, состоящее из множества компонентов и множества связей между ними.

**Системное программирование** – это процесс разработки системных программ (в том числе, управляющих и обслуживающих). Также это разработка программ сложной структуры.

Подразделение ПО на системное и прикладное является до некоторой степени устаревшим. Сегодняшнее деление предусматривает по меньшей мере три градации ПО:

Системное

Промежуточное (связующее)

Прикладное

**Промежуточное (связующее) ПО** – совокупность программ, осуществляющих управление вторичными (**виртуальные** ресурсы, которые создаются **самими программами, а не физические**) ресурсами, ориентированными на решение определенного класса задач

Например: СУБД, модули управления языком интерфейса ИС, программы сбора и предварительной обработки информации

**Промежуточное (связующее) ПО** – комплекс технологического ПО для обеспечения взаимодействия между различными приложениями, системами, компонентами (совокупность программ, осуществляющих управление вторичными ресурсами, ориентированными на решение определенного класса задач)

Например: Веб-сервер, сервер приложений, сервисная шина, система управления контентом (СУБД, модули управления языком интерфейса ИС, программы сбора и предварительной обработки информации)

С точки зрения инструментальных средств разработки промежуточное ПО ближе к прикладному, так как не работает на прямую с первичными ресурсами, а использует для этого сервисы, предоставляемые системным ПО

С точки зрения алгоритмов и технологий разработки промежуточное ПО ближе к системному, так как всегда является сложным программным изделием многократного и многоцелевого использования и в нем применяются те же или сходные алгоритмы, что и в системном ПО

Современные тенденции развития ПО состоит в снижении объема как системного, так и прикладного программирования. Основная часть работы программистов выполняется в промежуточном ПО

К функциям системного ПО принято относить:

* Системное ПО создаёт среду, в которой могут работать другие программы.
* Автоматизация разработки новых программ
* Обеспечение надежной и эффективной работы компьютера и компьютерной сети
* Проведение диагностики и профилактики аппаратуры компьютера и компьютерных сетей
* Выполнение вспомогательных технологических процессов (копирование, архивирование, восстановление после сбоев и т.д.)

**Существуют следующие группы системного ПО:**

* Операционные системы - базовое
* Интерфейсные оболочки (ОС) – позволяют пользователю взаимодействовать с операционной системой, текстовые, графические - базовое
* Системы управления файлами - базовое
* Системы программирования - сервисное
* Утилиты - сервисное
* Драйверы – программы, которые позволяют операционной системе взаимодействовать с аппаратными устройствами - базовое/сервисное
* Средства сетевого доступа - базовое

Классификация системного ПО

**Управляющие программы** — это системные программы, которые выполняют важные функции для правильного функционирования всей системы. Они управляют ресурсами компьютера и взаимодействуют с внешней средой (например, другими устройствами, пользователями или сетью). + Восстановление после неисправностей

**Обслуживающие программы (или утилиты)** — это программы, которые предназначены для выполнения рутинных задач, облегчающих работу пользователей и обслуживающего персонала.

**Базовое системное ПО** — это **минимальный набор программ**, который необходим для того, чтобы **компьютер мог функционировать**. Оно включает в себя все самые важные компоненты, без которых компьютер не сможет выполнять основные задачи.

**Сервисное системное ПО** — это **программы и комплексы программ**, которые дополняют базовое ПО и создают **удобную среду для работы** других программ и пользователей. Это ПО предоставляет дополнительные функции и удобства, которые делают работу с компьютером и сетью более эффективной.

**Система программирования** – система, образуемая языком программирования, компилятором или интерпретатором программ, представленных на этом языке, соответствующей документацией, а также вспомогательными средствами для подготовки программ к форме, пригодной для выполнения

Системы программирования включают в себя следующие средства:

* Редактор текста
* Транслятор
* Компоновщик
* Отладчик
* Библиотеки подпрограмм

В лабораторных работах ваша система программирования будет основана на Clang, LLVM и CMake

**Программный модуль** –программа или фрагмент программы, предназначенный для хранения, трансляции, объединения с другими программными модулями и загрузки в оперативную память

Требования к программным модулям:

**- Функциональность** – модуль должен выполнять законченную функцию

**- Несвязность** – модуль должен иметь минимум связей с другими модулями, связь через глобальные переменные и области памяти нежелательна

**- Специфицируемость** – входные и выходные параметры модуля должны четко формулироваться

**Исходный модуль** – программный модуль на исходном языке, обрабатываемый транслятором и представляемый для него как целое, достаточное для проведения трансляции

**Транслятор** – системная программа, преобразующая исходную программу на одном языке программирования в программу на другом языке

**Виды трансляторов:**

**- Ассемблер** — это программа, которая переводит код из языка ассемблера в машинный код (или в промежуточный код).

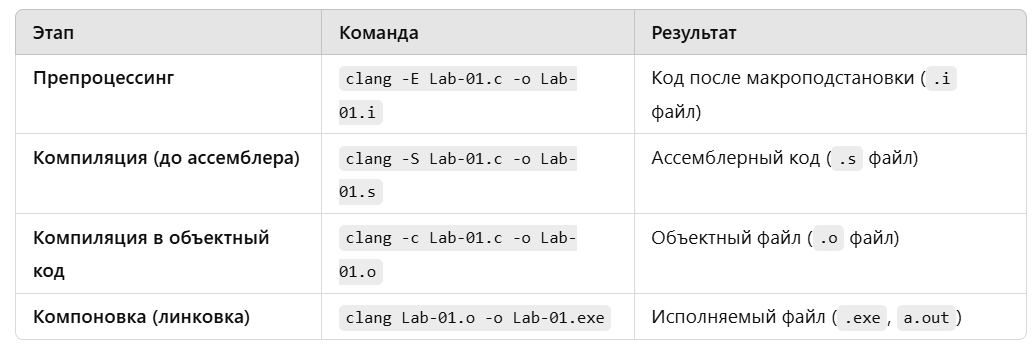
- **Компилятор** — это программа, которая переводит весь исходный код программы, написанный на высокоуровневом языке программирования, в машинный код или в промежуточный код. Компилятор делает это за один раз.

- **Интерпретатор** — это программа, которая построчно выполняет исходный код программы, написанный на высокоуровневом языке программирования.

- **Эмулятор** — это программа, которая имитирует работу одного компьютера на другом. Эмулятор позволяет запускать программы, написанные для другой архитектуры (например, программы для старых компьютеров или разных операционных систем), на современной системе.

- **Перекодировщик** — это программа, которая переводит код с одного языка программирования на другой, сохраняя его функциональность. Он может преобразовывать код с одного высокоуровневого языка в другой.

- **Макропроцессор** — это программа, которая занимается обработкой макросов в исходном коде. Макрос — это некая предварительно заданная инструкция или шаблон кода, который автоматически заменяется на более длинный фрагмент кода до того, как программа будет скомпилирована или интерпретирована.

****

**Шаг первый – Предварительная обработка кода**:

* Присоединение исходных файлов
* Работа макропроцессоров



**Шаг второй – Анализ**:

* Лексический анализ



### -fsyntax-only

Этот флаг заставляет Clang выполнять только проверку синтаксиса исходного кода

**XClang**: Когда нужно использовать специфичные для Clang опции или плагины, которые требуют передачи параметров непосредственно Clang. XClang сообщает компилятору Clang, что следующее — это параметр, который будет обработан внутри Clang, а не внешним компилятором.

Флаг **-dump-tokens** используется в **Clang** для вывода токенов исходного кода во время процесса препроцессинга или компиляции. Токены — это минимальные единицы исходного кода, которые компилятор разбивает и понимает. Токенами могут быть ключевые слова, операторы, идентификаторы, литералы, скобки и другие элементы языка программирования.

* Синтаксический анализ
* Семантический анализ



**AST** (Abstract Syntax Tree, абстрактного синтаксического дерева) программы. Абстрактное синтаксическое дерево — это структура данных, которая представляет синтаксис исходного кода в виде дерева, где каждый узел соответствует конструкции языка (например, выражению, оператору или типу).

**Шаг третий – Синтез**:

* Генерация машинно-независимого кода



**LLVM (Low-Level Virtual Machine) IR** — это промежуточный формат, который используется в компиляторах LLVM.

**LLVM** (Low-Level Virtual Machine) — это набор технологий, предназначенных для разработки компиляторов и других инструментов, которые работают с промежуточным представлением (IR) кода.

Оптимизация машинно-независимого кода



Результатом работы **компилятора** является объектный модуль

**Объектный модуль** – программный модуль, получаемый в результате трансляции исходного модуля

Содержимое объектного модуля не содержит признаков на каком языке был написан исходный модуль

Поскольку транслятор обрабатывает только один конкретный модуль, он не может должным образом обработать те части этого модуля, в которых запрограммированы обращения к данным или процедурам, определенным в другом модуле

Такие обращения называются внешними ссылками. Те места в объектном модуле, где содержатся внешние ссылки, транслируются в некоторую промежуточную форму, подлежащую дальнейшей обработке. Говорят, что объектный модуль представляет собой программу на машинном языке с неразрешенными внешними ссылками

Разрешение внешних ссылок выполняется на следующем этапе подготовки, который обеспечивается Редактором Связей (**Компоновщиком**)

Он соединяет вместе все объектные модули, входящие в программу. Результатом работы Редактора Связей является загрузочный модуль

**Загрузочный модуль** – программный модуль, представленный в форме, пригодной для загрузки в оперативную память для выполнения

**Архитектурная модель ПО** – принципиальная организация ПО, воплощенная в его элементах, их взаимоотношениях друг с другом и со средой, а также принципы, направляющие проектирование и эволюцию ПО

**Классификация ПО по количеству звеньев (уровней):**

- Одноуровневая

- Двухуровневая

- Трёхуровневая

**Виды организации межпрограммного взаимодействия:**

- На уровне библиотек

- На уровне сервисов, осуществляющих обмен данными по различным протоколам

- На уровне данных

- Через различные технологии интеграции (ESB, CORBA, COM, DCOM, ActiveX и др.)

**Компилятор clang**

Стоит отметить, что clang не является компилятором в прямом смысле этого слова, а является «фронтендом» для языка программирования С (для языка C++ таковой будет называться clang++)

Под «фронтендом» в инфраструктуре LLVM понимается транслятор из некоторого языка программирования в промежуточный язык (LLVM IR), т.е. непосредственная генерация объектного кода перекладывается на LLVM

Инструмент сборки CMake

**Система сборки** – это инструмент или набор инструментов, используемых для автоматизации процесса преобразования исходного кода программы в исполняемый файл или библиотеку

**Основная задача системы сборки** – управлять различными этапами этого процесса, такими как компиляция, компоновка, препроцессинг и другие задачи, связанные с подготовкой программного обеспечения к выполнению

CMake не является традиционной системой сборки

СMake – это система мета-сборки

**Система мета-сборки** – это инструмент, который генерирует файлы конфигурации для других систем сборки. В отличие от традиционных систем сборки, которые напрямую управляют процессом компиляции и компоновки, мета-сборки создают промежуточные файлы, которые затем используются другими инструментами для выполнения фактической сборки

**Основные особенности систем мета-сборки:**

**Генерация конфигураций:** Мета-сборки создают файлы конфигурации (например, Makefile, Ninja файлы), которые затем используются другими инструментами для сборки проекта

**Кроссплатформенность:** Они обеспечивают возможность генерации конфигураций для различных платформ и компиляторов, что делает их особенно полезными для кроссплатформенных проектов

**Абстракция**: Мета-сборки предоставляют более высокий уровень абстракции, позволяя разработчикам описывать процесс сборки в более общем виде, не привязываясь к конкретным инструментам сборки

**Гибкость:** Поддерживают различные сценарии сборки и позволяют легко изменять конфигурацию сборки в зависимости от требований проекта

**cmake\_minimum\_required(VERSION 3.30)** – данная команда обозначает минимальную необходимую версию CMake для работы с проектом

**project(HelloWorld C)** – устанавливает имя нашего проекта

**add\_executable(Hello src/Source.c)** - добавление исполняемого файла в проект, используя указанные исходные файлы

Перед тем, как генерировать файлы сборки для проекта стоит упомянуть два следующих момента:

1. Понятие Генератор – по сути, это та система сборки для которой будут сгенерированы файлы сборки

2. Сборки бывают двух основных видов:

**In-source** – вид сборки при котором выходные файлы проекта будут располагаться в каталоге с исходными «source» файлами

**Out-of-source –** вид сборки при котором выходные файлы проекта будут располагаться в отдельном от каталога с исходными «source» файлами месте

Генератор будет указываться при первичной инициализации проекта CMake:



Генератор указывается через флаг –G

Через **флаг –B** был указан каталог в котором будут размещены файлы сборки проекта (что по сути делает сборку «out-of-source»)

**Флаг –T** указывает на используемые инструменты для сборки (данный параметр в данном случае необходим для указания Clang в качестве компилятора, т.к. стандартный способ задания не работает с VS)

Для указания используемого компилятора для языка C необходимо использовать параметр   
**-DCMAKE\_C\_COMPILER=<название компилятора>**

Как уже упоминалось ранее, данный параметр не работает с генератором Visual Studio

После создания файлов сборки необходимо собственно собрать проект. Для этого необходимо вызвать тот сборщик, генератор которого был указан в параметре –G

Но чтобы не задумываться о всём многообразии сборщиков и как их вызывать, CMake позволяет вызывать их в общем вид



В зависимости от выбранного генератора, полученный exe-файл будет расположен в том или ином подкаталоге каталога build

Некоторые генераторы также поддерживают мультиконфигурацию, т.е. позволяют настроить и генерировать файлы под несколько конфигурации (например, Debug и Release). Таких в настоящий момент только 2: Visual Studio и XCode

В данном случае добавляются такие команды как:

**add\_subdirectory()** – добавляет в сборку другой каталог

**include()** – команда которая подключает модули CMake к текущему проекту

**check\_language()** – команда которая проверяет доступность компиляторов для того или иного языка

**if(), else(), endif()** – команды ветвления

**enable\_language(**) – включает поддержку указанного языка в проекте

**message()** – позволяет вывести сообщение на консоль

**set() –** устанавливает значение какой-либо переменной

**1. Препроцессинг** (предварительная обработка)

clang -E Lab-01C.c -o Lab-01C.pre.c



-E — выполнить только предпроцессинг (развернуть макросы, обработать #include, #define, #ifdef и т. д.), но не компилировать код.

-o Lab-01C.pre.c — записать результат в Lab-01C.pre.c

Обрабатывает директивы #include, #define, заменяет макросы. Удаляет комментарии. Не выполняет компиляцию.

**Результат**: Lab-01C.pre.c - код после обработки препроцессором.

**2. Анализ кода**

**2.1. Лексический анализ** (токены)

clang -fsyntax-only -Xclang -dump-tokens Lab-01C.c



-fsyntax-only Проверяет только синтаксис кода, не компилируя и не создавая объектный файл. Используется для проверки кода на ошибки.

-Xclang — передает опции непосредственно фронтенду Clang (поскольку -dump-tokens не поддерживается напрямую через обычные флаги).

-dump-tokens — выводит список токенов после лексического анализа (лексер разбирает код на токены: ключевые слова, идентификаторы, операторы и т. д.).

Разбирает код на отдельные элементы (токены). Показывает, как Clang видит переменные, операторы, ключевые слова.

**Результат:** список токенов в консоли.

**2.2. Синтаксический анализ** (AST)

clang -fsyntax-only -Xclang -ast-dump Lab-01C.c



Строит абстрактное синтаксическое дерево (AST). Показывает, как Clang интерпретирует код до генерации машинных команд.

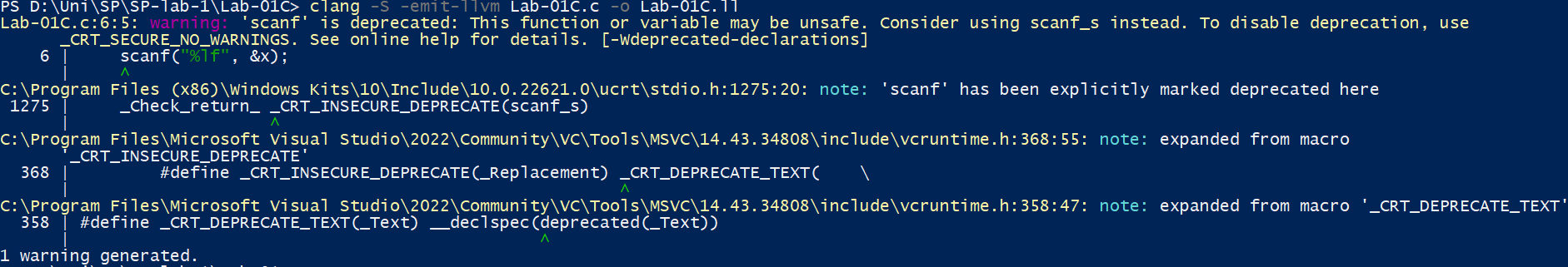
**Результат:** дерево AST в консоли.

**3. Генерация промежуточного кода** (LLVM IR)

**Без оптимизации**

clang -S -emit-llvm Lab-01C.c -o Lab-01C.ll





Генерирует LLVM IR (байткод) — промежуточное представление перед компиляцией в машинный код.

**Результат:** файл Lab-01C.ll с байткодом.

**С оптимизацией (-O3)**

clang -S -O3 -emit-llvm Lab-01C.c -o Lab-01C\_opt.ll



Выполняет агрессивные оптимизации. Упрощает выражения, удаляет лишние переменные.

**Результат:** Lab-01C\_opt.ll — оптимизированный LLVM IR.

LLVM IR (Lab-01C.ll) — это не C-код, а промежуточное представление.  
Его нельзя передавать напрямую компилятору, который ждёт C-код. Если мы хотим использовать main.ll вместо main.c, то нам нужно не clang -c, а llc или clang с опцией -c.

-S Генерирует ассемблерный код, но не создает объектный файл (.o).

-emit-llv Вместо обычного ассемблера генерирует LLVM IR (Intermediate Representation, промежуточное представление).

1. **Компиляция в объектный файл**

clang -c Lab-01C.c -o Lab-01C.o



Преобразует код в объектный файл (.o). Это ещё не исполняемый файл.

**Результат:** Lab-01C.o — машинный код без компоновки.

Компилирует исходный код без линковки, создавая объектный файл (.o на Linux/macOS или .obj на Windows).

Линковка — это процесс объединения объектных файлов (.o или .obj) в исполняемый файл или библиотеку.

1. **Компоновка в исполняемый файл**

clang Lab-01C.o -o Lab-01C.exe



### **Основные флаги компиляции**

* -E – Только препроцессинг (#include, #define), без компиляции.
* -c – Компилирует в объектный файл (.o), без линковки.
* -o <file> – Задает имя выходного файла.
* -S – Компиляция в **ассемблерный код** (.s).
* -emit-llvm – Генерирует **LLVM IR** (можно с -S для .ll или -c для .bc).

### **Оптимизация кода**

* -O0 – Без оптимизации (быстрая компиляция).
* -O1 – Легкая оптимизация.
* -O2 – Оптимизация скорости.
* -O3 – Максимальная оптимизация.
* -Os – Оптимизация размера.
* -Ofast – Самая агрессивная оптимизация (может нарушить стандарт).

### **Анализ кода**

* -fsyntax-only – Проверка **синтаксиса**, без компиляции.
* -Xclang -dump-tokens – Вывод **лексических токенов**.
* -Xclang -ast-dump – Вывод **AST-дерева** кода.
* -Weverything – Включает **все возможные предупреждения**.

### **Компиляция C++ (clang++)**

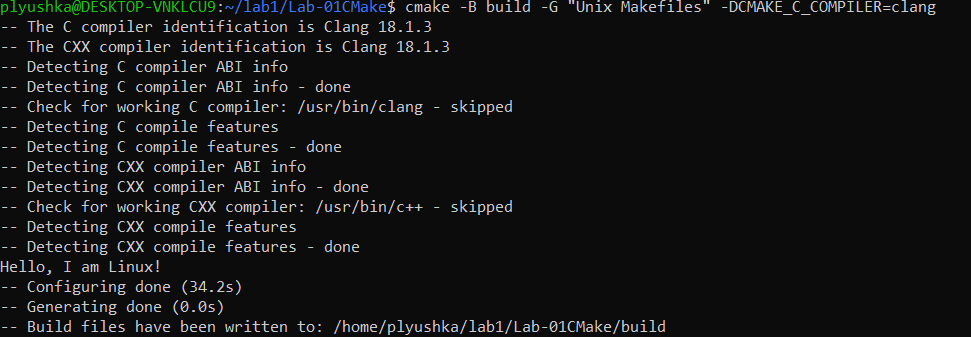
* clang++ – Компиляция C++ (аналог g++).

### **Генерация кода**

### **Отладка и диагностика**

### **Выбор стандарта языка**

### **Линковка и библиотеки**



-B build — указывает каталог для сборки.

-G "Unix Makefiles" — выбирает генератор для Unix-систем (на Linux обычно используется этот генератор).

-DCMAKE\_C\_COMPILER=clang — указывает компилятор C для CMake

### **1. Генерация файлов сборки**

* -B <dir> – Задает **каталог сборки** (out-of-source).
* -S <dir> – Задает **каталог с исходниками**.
* -G <generator> – Выбирает **генератор сборки** (например, Ninja, Unix Makefiles, Visual Studio).

### **2. Сборка проекта**

* --build <dir> – Запускает сборку в указанной папке.
* --target <name> – Собирает **конкретную цель** (по умолчанию all).
* --config <type> – Выбирает конфигурацию (Debug, Release и т. д.).

| **Генератор** | **Описание** |
| --- | --- |
| Unix Makefiles | Make (Linux/macOS) |
| Ninja | Ninja (быстрый билд) |
| Visual Studio 17 2022 | Visual Studio 2022 |
| Xcode | Xcode (macOS) |