**Вопросы. Архитектура и программирование.**

---

**(SE)** Архитектура компьютера: архитектура фон Неймана, гарвардская архитектура.

---

В 1940-х годах в процессе работы над первыми электронными вычислительными машинами Джон фон Нейман и его коллеги определили ряд принципов построения вычислительных машин.

**Принципы фон Неймана:**

1. *Принцип однородности памяти*

   Команды и данные хранятся в одной и той же памяти и внешне в памяти неразличимы. Распознать их можно только по способу использования; то есть одно и то же значение в ячейке памяти может использоваться и как данные, и как команда, и как адрес в зависимости лишь от способа обращения к нему. Это позволяет производить над командами те же операции, что и над числами, и, соответственно, открывает ряд возможностей. Так, например, команды одной программы могут быть получены как результат исполнения другой программы. Эта возможность лежит в основе трансляции — перевода текста программы с языка высокого уровня на язык конкретной вычислительной машины.

2. *Принцип адресности*

   Структурно основная память состоит из пронумерованных ячеек, причём процессору в произвольный момент доступна любая ячейка. Двоичные коды команд и данных разделяются на единицы информации, называемые словами, и хранятся в ячейках памяти, а для доступа к ним используются номера соответствующих ячеек — адреса.

3. *Принцип программного управления*

   Все вычисления, предусмотренные алгоритмом решения задачи, должны быть представлены в виде программы, состоящей из последовательности управляющих слов — команд. Каждая команда предписывает некоторую операцию из набора операций, реализуемых вычислительной машиной. Команды программы хранятся в

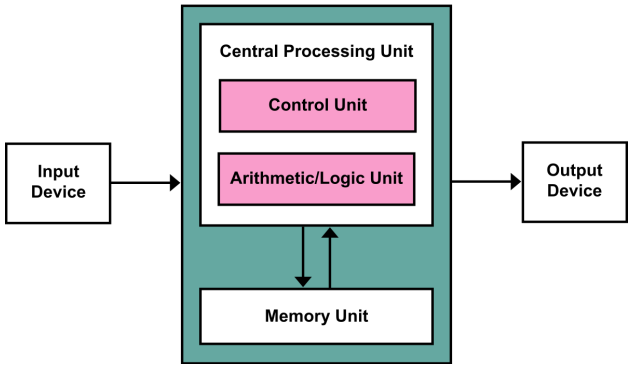


Рис. 1: Архитектура фон Неймана

последовательных ячейках памяти вычислительной машины и выполняются в естественной последовательности, то есть в порядке их положения в программе. При необходимости, с помощью специальных команд, эта последовательность может быть изменена. Решение об изменении порядка выполнения команд программы принимается либо на основании анализа результатов предшествующих вычислений, либо безусловно.

**Узкое место архитектуры фон Неймана**

Совместное использование шины для памяти программ и памяти данных приводит к узкому месту архитектуры фон Неймана, а именно ограничению пропускной способности между процессором и памятью по сравнению с объёмом памяти. Из-за того, что память программ и память данных не могут быть доступны в одно и то же время, пропускная способность канала «процессор-память» и скорость работы памяти существенно ограничивают скорость работы процессора — гораздо сильнее, чем если бы программы и данные хранились в разных местах.

Данная проблема решается совершенствованием систем кэширования, что в свою очередь усложняет архитектуру систем и увеличивает риск возникновения побочных ошибок (например, проблема когерентности памяти).

**Гарвардская архитектура**

Гарвардская архитектура — архитектура ЭВМ, разработанная в конце 1930-х годов в Гарвардском университете. Отличительными признаками данной архитектуры являются:

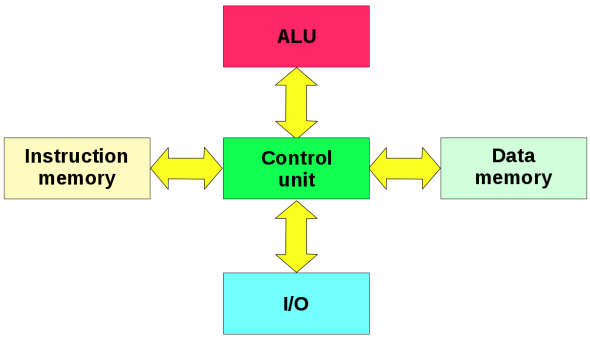


Рис. 2: Гарвардская архитектура

- хранилище инструкций и хранилище данных представляют собой разные физические устройства;
- канал инструкций и канал данных также физически разделены.

В архитектуре фон Неймана процессор в каждый момент времени может либо читать инструкцию, либо читать/записывать единицу данных из/в памяти. Оба действия одновременно происходить не могут, поскольку инструкции и данные используют один и тот же поток (шину).

В компьютере с использованием гарвардской архитектуры процессор может считывать очередную команду и оперировать памятью данных одновременно и без использования кэш-памяти.

Исходя из физического разделения шин команд и данных, разрядности этих шин могут различаться и физически не могут пересекаться.

---

**(SE)** Объектно-ориентированное программирование. Основные принципы.

---

**Объектно-ориентированное программирование** — методология программирования, основанная на представлении программы в виде совокупности взаимодействующих *объектов*, каждый из которых является экземпляром определённого *класса*, а классы образуют *иерархию наследования*.

**Объект** — это сущность, которой можно посылать сообщения и которая может на них реагировать, используя свои данные. Объект — это экземпляр класса. Данные объекта скрыты от остальной программы.

**Класс** — в объектно-ориентированном программировании, представляет собой шаблон для создания объектов, обеспечивающий начальные значения состояний: инициализация полей-переменных и реализация поведения функций или методов.

Объекты внутри программы взаимодействуют с помощью **сообщений**. Во многих языках программирования используется концепция «*отправка сообщений как вызов метода*» — объекты имеют доступные извне методы, вызовами которых и обеспечивается взаимодействие объектов.

**Основные принципы ООП.**

1. **Абстракция**

   Абстракция в объектно-ориентированном программировании — это использование только тех характеристик объекта, которые с достаточной точностью представляют его в данной системе. Основная идея состоит в том, чтобы представить объект минимальным набором полей и методов и при этом с достаточной точностью для решаемой задачи.

2. **Инкапсуляция. Две трактовки**

   Инкапсуляция — свойство системы, позволяющее объединить данные и методы, работающие с ними, в классе. В общем случае в разных языках программирования термин *инкапсуляция* относится к одной или обеим одновременно следующим нотациям:

   - механизм языка, позволяющий ограничить доступ одних компонентов программы к другим;
   - языковая конструкция, позволяющая связать данные с методами, предназначенными для обработки этих данных.

3. **Наследование**

   Наследование — свойство системы, позволяющее описать новый класс на основе уже существующего с частично или полностью заимствованной функциональностью. Класс, от которого производится наследование, называется базовым, родительским или суперк-

лассом. Новый класс — потомком, наследником, дочерним или производным классом.

4. **Полиморфизм**

   Полиморфизмом называется возможность единообразно обрабатывать разные типы данных. В языке C++ можно выделить следующие механизмы полиморфизма:

   - Перегрузка функций (не обязательно ООП?). Выбор функции происходит в момент компиляции на основе типов аргументов функции, *статический полиморфизм.*
   - Виртуальные методы. Выбор метода происходит в момент выполнения на основе типа объекта, у которого вызывается виртуальный метод, *динамический полиморфизм.*

---

**(SE)** Компиляция программ. Как устроен компилятор? Зачем нужен компилятор. Интерпретация программ.

---

**Определение.** *Компилятор* — это программа, которая считывает текст программы, написанной на одном языке — исходном, и транслирует (переводит) его в эквивалентный текст на другом языке — целевом.

Одна из важных ролей компилятора состоит в сообщении об ошибках в исходной программе, обнаруженных в процессе трансляции.

Если целевая программа представляет собой программу на машинном коде (языке), то она может быть вызвана пользователем для обработки некоторых входных данных и получения некоторых выходных данных.

**Определение.** *Интерпретация* — процесс построчного анализа, обработки и выполнения исходного кода программы, в отличие от компиляции, где весь текст программы перед запуском анализируется и транслируется в машинный или байт-код без ее выполнения.

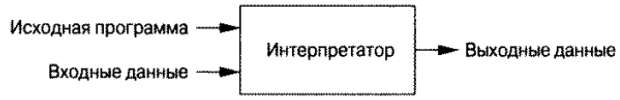


Рис. 3: Схематичное изображение принципа работы интерпретатора


Рис. 4: Схематичное изображение принципа работы компилятора и целевой программы

**Определение.** *Интерпретатор* — программа, выполняющая интерпретацию.

Целевая программа на машинном языке, производимая компилятором, обычно гораздо быстрее, чем интерпретатор, получает **выходные** данные на основе входных. Однако интерпретатор обычно обладает лучшими способностями к диагностике, поскольку он выполняет исходную программу инструкция за инструкцией. При этом компилятор требует больше времени для анализа и обработки языка высокого уровня.

**Пример.** Языковой процессор Java объединяет в себе и компиляцию и интерпретацию. Исходная программа на Java может сначала компилироваться в байт-код, а затем байт-код интерпретируется виртуальной машиной. Преимущество такого подхода в том, что скомпилированный на одной машине байт-код может быть выполнен на другой.

Перевод исходной программы в целевой машинный код происходит в несколько этапов (см. Рис. 5).

**Этап №1. Препроцессор**

Препроцессор работает с исходным текстом. Обычно он выполняет замену макросов (C/C++), удаляет комментарии, выполняет директивы, начинающиеся с символа # (#include, #define, #pragma ...).

**Этап №2. Компилятор (трансляция)**

На данном этапе происходит перевод исходного кода (программы на исходном языке) в целевой код (программу на целевом языке). Целевым языком может быть один из низкоуровневых языков: машинный код, байт-код, язык ассемблера.

**Этап №2,5. Ассемблер** Если компилятор возвращает программу на языке ассемблера, то необходим данный этап, на котором происходит об-


Рис. 5: Система обработки языка

работка языка ассемблера программой *ассемблер*. Результатом данного этапа является машинный код.

**Этап №3. Компоновщик/Загрузчик (сборка, линковка)**

Большие программы компилируются по частям, поэтому после получения нескольких объектных файлов их нужно скомпоновать. Компоновщик выполняет разрешение адресов памяти, по которым код из одного файла сможет обращаться к информации из другого файла. *Загрузчик* затем помещает все выполнимые объектные файлы в память для выполнения.

**(SE)** C++: как происходит компиляция, интерпретация, выполнение.

C++ — компилируемый язык. Для запуска программы на C++ необходимо транслировать ее из текстовой формы, понятной для человека, в форму, понятную для машины. Этим занимается компилятор.

Особенности компиляции:

1. Нет накладных расходов при исполнении программы.
2. При компиляции можно отловить некоторые ошибки (?).
3. Программу требуется компилировать для каждой платформы отдельно.

Подробнее о этапах компиляции программ на C++.

**Этап №1. Препроцессор**

Препроцессор работает с кодом на C++ как с текстом. Команды языка препроцессора называют *директивами*, все директивы начинаются со знака #. Директива #**include** позволяет подключать заголовочные файлы к файлам кода. Препроцессор просто заменяет директиву #**include "func.h"** на содержимое файла **func.h**. Препроцессор может выдать ошибку, если указанного файла нет или директива написана с ошибкой. При этом он ничего не знает о синтаксисе C++, то есть не будет выдавать ошибку на **intt main().** Для того, чтобы отследить работу препроцессора компилятора **g++** необходимо вызвать команду:
**g++ -E main.cpp**.

**Этап №2. Компиляция**

На вход компилятору поступает код на C++, который уже прошел обработку препроцессора. Каждый файл с кодом (**file.cpp**) компилируется отдельно и независимо от других. Эта особенность позволяет, например, перекомпилировать только файл с ошибкой. На выходе компилятора из каждого файла с кодом получается "объектный файл— бинарный файл со скомпилированным кодом (с расширением **.o** или **.obj**). Файл с расширением **.o** уже нельзя прочитать (см. Рис.6). Для компиляции файлов после препроцессинга можно использовать команду
**g++ -c main_preprocessed.cpp**.

**Этап №3. Компоновка (линковка)**

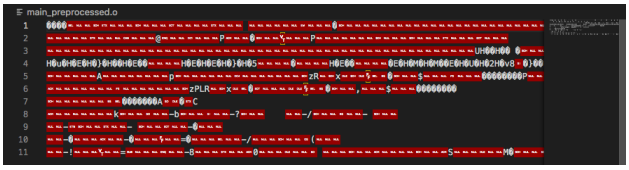


Рис. 6: Попытка открыть объктный файл в текстовом редакторе

Программа может состоять из отдельных частей. Пример: программа "Hello, World!"состоит из написанной нами части **int main() { }** и частей стандартной библиотеки языка C++. Эти отдельные части называются единицами трансляции.

**Определение.** В языках программирования *единица трансляции* — максимальный блок исходного текста, который физически можно оттранслировать (преобразовать во внутреннее машинное представление; в частности, откомпилировать).

*Примечение. Раньше размер оперативной памяти компьютера не позволял содержать одновременно компилятор, текст крупной программы и результирующий код, поэтому приходилось компилировать ее по частям и собирать из откомпилированных частей исполняемый файл.*

В языках программирования C и C++ единица трансляции — подаваемый на вход компилятора исходный текст (файл с расширением **.c** или **.cpp**) со всеми включёнными (содержимое которых вставляется препроцессором) в него файлами.

Программа, связывающая файлы с результирующим объектным кодом, называется *редактором связей* или *компоновщик* или *линковщик*.

Итак, на этапе линковки все объектные файлы объединяются в один исполняемый (или библиотечный) файл. При этом происходит подстановка адресов функций в места их вызова. По каждому объектному файлу строится таблица всех функций, которые в нем определены.

На этапе линковки важно, чтобы каждая функция имела уникальное имя. В C++ может быть две функции с одинаковыми именами, но разными параметрами. Для того, чтобы не путать функции с одинаковыми именами происходит искажение имен функций. Например, gcc преобразует функцию **void foo(int, double)** в **_Z3fooid**. Поэтому нельзя начинать имена функций и переменных с цифры.

Помимо прочего на этапе линковки происходит выставление точки

входа в программу.

**Определение.** Точка входа — это функция, вызываемая при запуске программы. По умолчанию — это функция **main()**.

Для того чтобы собрать объектные файлы в один исполняемый можно воспользоваться командой
**g++ main.o square.o -o file_name**.

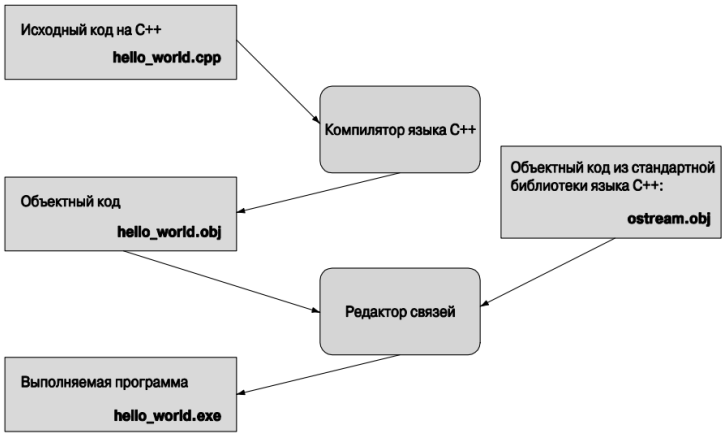


Рис. 7: Схема процесса компиляции программы на C++