

소프트웨어의 동작 방식 이해  
(하드웨어 원리, 컴파일 및 실행까지...)

Sunny Kwak

(sunnykwak@daum.net)

# 목 차

- **폰 노이만 아키텍처**
  - 계산기 혹은 컴퓨터
  - 계산기의 시초
  - 기계적 계산기
  - 최초의 컴퓨터
  - 폰 노이만 아키텍처 개요
  - 폰 노이만 아키텍처 구성도
  - 폰 노이만 아키텍처의 구성 요소
  - 폰 노이만 머신의 동작방식
  - 클럭(clock)과 컴퓨터 성능
  - Register vs. Memory
  - 폰 노이만 아키텍처를 비유하자면...
- **소프트웨어의 동작 원리**
  - 최초의 프로그래머 "Ada"
  - 프로그래밍과 수학의 상관관계
  - 절차지향 프로그래밍
  - 명령문 그리고 함수 기반의 구조적 프로그래밍
  - C 프로그램의 동작 방식 이해
  - 하드웨어, 운영체제, 그리고 언어의 결합
  - main 함수의 argc 그리고 argv
- **COMPILE, LINK, LOAD**
  - 프로그램을 실행하기까지의 절차
  - 컴파일 작업 개요
  - 소스, 어셈블리, 기계어 코드 비교
  - 컴파일러 (Compiler)
  - 어셈블러 (Assembler)
  - 링커 (linker)
  - 로더 (loader)

소프트웨어 개발자가 꼭! 알아두어야 하는

# 폰 노이만 아키텍처

# 계산기 혹은 컴퓨터

- 영어로 '계산한다'는 의미의 단어는?
  - "calculate", 그리고 "compute" 가 있습니다.
  - 계산하는 기계라는 명사를 만드려면, "~er"을 붙이죠.
- 그럼, "Calculator"와 "Computer"는 같은 건가요?
  - Calculator는 '계산기', Computer는 '컴퓨터'라고 번역합니다.
  - 영어 단어 자체로도 다른 종류의 기계를 말합니다.
  - 어원(origin)이 같다는 것은 말은 달리 말해서 뿌리(기원, origin)가 같다는 말입니다.  
하나의 뿌리에서 갈라져 나왔다는 말입니다.
- 그 뿌리에서부터 이야기 해 보겠습니다.

# 계산기의 시초

- 다들 아시다시피 인간은 “도구” 활용할 줄 압니다.



손가락 셈으로 계산하다가,  
주판을 발명합니다.

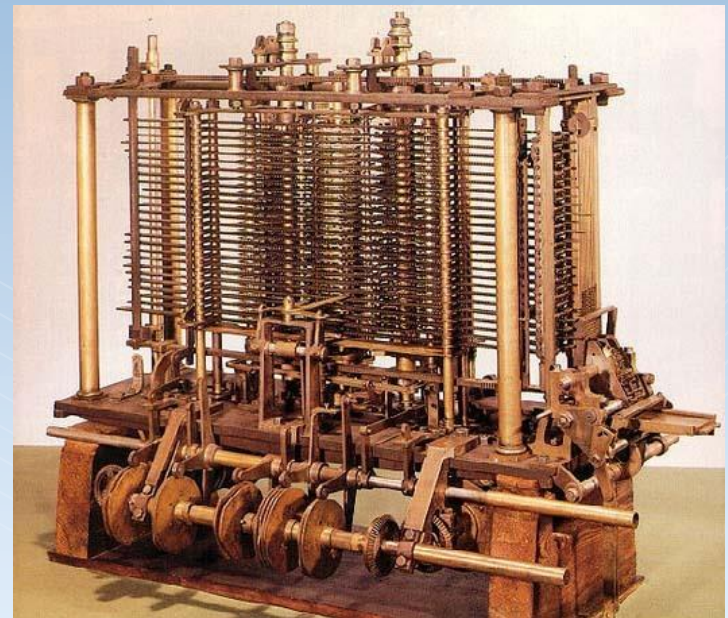


# 기계적 계산기

- 17세기, 철학자 겸 수학자 파스칼, '기계식 계산기' 발명
- 19세기, 찰스 베비지, 원시적인 프로그래밍 기계 설계



파스칼의 계산기 혹은 파스칼린(Pascaline)



베비지의 해석기관

# 최초의 컴퓨터

- '계산기'와 '컴퓨터'의 차이
  - '계산기'는 장치에 내장(각인)된 계산 기능만 수행할 수 있다.  
따라서 계산기는 공장에서 만들어질 때 주어진 기능만을 수행한다.
  - '컴퓨터'는 외부에서 프로그램을 입력해 '새로운 기능'을 부여하는 것이 가능하다.
  - 즉, 근본 원리(계산하는 기계)는 같지만,  
정해진 기능만 수행하는 것이냐, 아니면 기능을 확장할 수 있는냐 차이.
- 최초의 컴퓨터와 인물들
  - 콘라드 추제 (Conrad Zuse, 1910~1995), Z3 및 Z4 설계
  - 하워드 에이킨(Howard Aiken), 1944년 Mark-1 제작
  - 그레이스 머레이 호퍼(Grace Murray Hopper), 최초의 디버깅!
  - 프레스퍼 에커트와 존 모클리, 1947년 ENIAC 제작

## 디버깅의 유래 :

1947년 여름, 당시 그레이스 호퍼는 하버드 대학에서 마크 II 컴퓨터를 이용해 연구 중이었는데 컴퓨터가 자주 고장을 일으켰다고 한다. 호퍼와 동료들은 고장의 원인을 찾기 위해 컴퓨터 내부를 들여다보며 조사하던 중, 릴레이(relay)의 접점 사이에 끼어 들어붙어 죽어있는 나방(moth) 한 마리를 발견하게 된다.

실제로 컴퓨터 시스템 안에 벌레가 있었고 이로 인한 누전으로 컴퓨터가 오작동한 것이다. 그녀와 동료들이 문제가 된 나방을 조심스럽게 제거(De-bugged)하자 컴퓨터는 다시 정상적으로 작동하게 되었다.

인용 : [http://www.xeschool.com/xs/documents\\_for\\_debug](http://www.xeschool.com/xs/documents_for_debug)





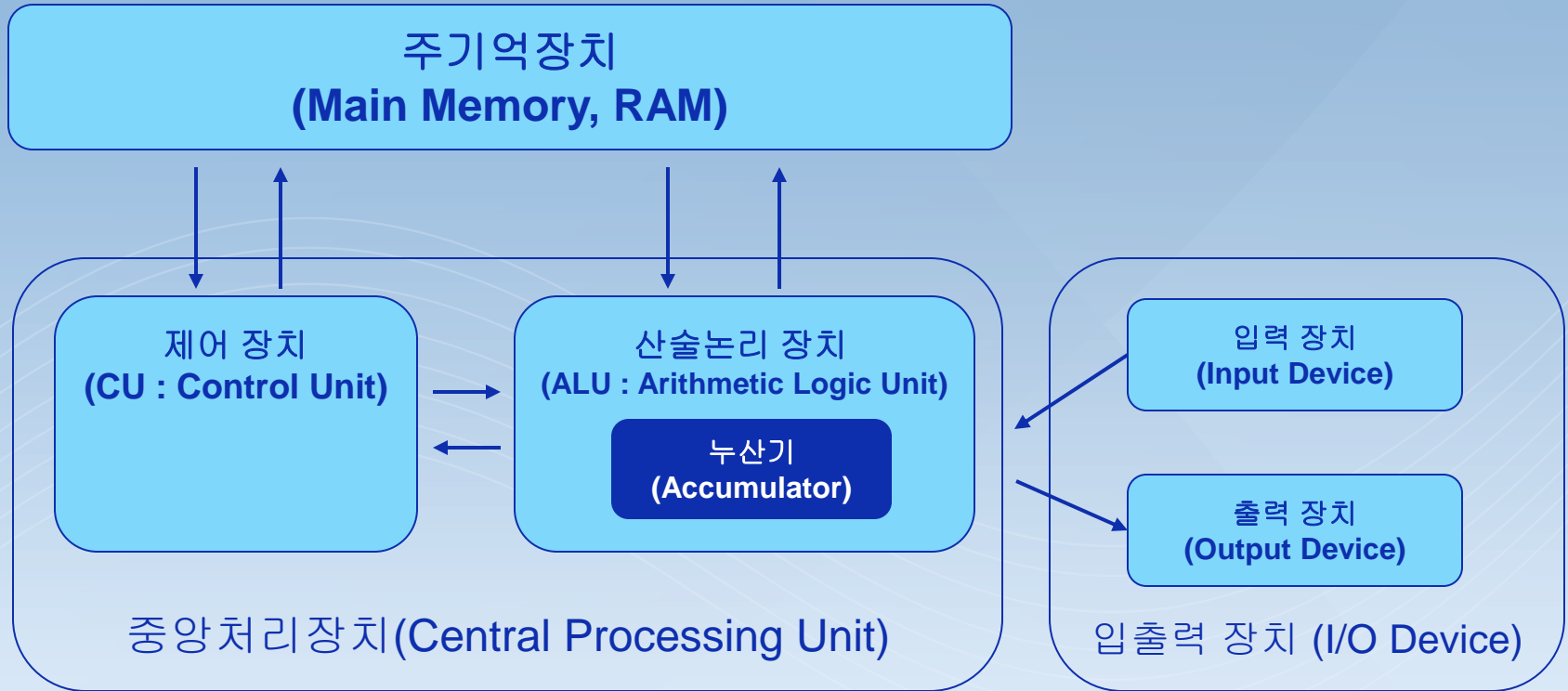
# 폰 노이만 아키텍처 개요



- 폰 노이만 박사 (John von Neumann)
  - EDVAC 설계에 참여하고, 이진수(binary) 도입.
  - 프로그램 내장 방식이라 불리는 '폰 노이만 아키텍처' 고안
  - 현대의 거의 모든 컴퓨터는 폰 노이만 박사가 제안한 구조 채택.
- 폰 노이만 아키텍처
  - 중앙처리장치(CPU), 메모리(memory), 입출력 장치(I/O device) 등 기본적으로 3가지 하드웨어를 결합하여 컴퓨터를 만드는 방식을 말한다.
- 폰 노이만 아키텍처 동작 방식
  - 컴퓨터가 수행해야 하는 소프트웨어(실행 코드)와 데이터는 영구적인 저장 매체 (하드 디스크, CD 등) 에 보관한다.
  - 소프트웨어를 실행하고자 할 때는 입력 장치(input device)가 소프트웨어 실행 코드와 데이터들을 메인 메모리로 복사한다.
  - 중앙처리장치(CPU)는 메인 메모리 상에 적재(load)된 명령어와 데이터를 고속으로 연산(processing)하고, 최종 결과를 출력 장치(output device)로 보내내 영구 보관한다.



# 폰 노이만 아키텍처 구성도




- 폰 노이만 방식의 핵심은 '프로그램을 메모리에 내장하는 방식'이라는 점.
- 외부의 입출력 장치는 데이터를 영구 저장할 수 있지만 매우 느리고, 주기억장치인 RAM은 아주 빠르지만 전원이 꺼지면 가지고 있는 데이터가 지워지는 단점이 있다.
- 폰 노이만은 소프트웨어를 메모리에 복사한 후 실행해서 성능을 비약적으로 높일 수 있다고 제안한 것이다.
- 누산기(accumulator)는 CPU 내에 적은 양의 데이터를 임시 보관하는 메모리이며, 일반적으로 레지스터(register)라고 불리우고 주기억장치 보다 더욱 더 빨리 읽고 쓸 수 있다.

# 폰 노이만 아키텍처의 구성 요소

- 중앙처리장치 (Central Processing Unit)
  - 제어 장치(Control Unit)과 산술논리장치(Arithmetic Logic Unit)로 구성된다.
  - 제어 장치는 주기억장치에서 실행할 명령(코드)과 데이터를 읽어 들이는 작업을 수행하며, 읽어들이는 명령을 ALU에 전달해 실행 시키는 역할을 수행한다.
  - 산술논리장치는 제어 장치로부터 실행 명령을 전달 받아 실질적인 계산(혹은 연산)을 수행한다.
- 주기억 장치 (Main memory)
  - 외부 입출력 장치에서 읽어들이는 프로그램과 데이터들을 보관하는 장소.
  - 이진수(binary) 형식의 데이터를 읽고 쓰는 속도가 매우 빠르나, 전원이 꺼지면 메모리 상의 모든 데이터는 즉시 사라진다.
- 입출력 장치 (Input/Output Device)
  - 중앙처리장치와 주기억장치에 프로그램과 데이터를 제공하거나 프로그램의 수행 결과를 외부로 내보내는 장치.
  - 키보드, 모니터, 마우스, 프린터, 하드 디스크, 네트워크 장치 등 컴퓨터 본체에 연결되어 데이터 및 프로그램을 입력하고 출력할 수 있는 모든 장치들이다.

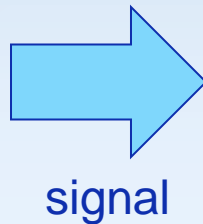
# 폰 노이만 머신의 동작방식

- Fetch
  - 메인 메모리(main memory)에서 다음에 수행할 명령을 Control Unit으로 가져옴.
- Decode
  - Control Unit이 명령어를 해석.
- Execute
  - 해석된 명령에 따라 Control Unit 이 memory에서 계산할 데이터를 꺼내온 후, ALU에 전달하고 연산을 수행한다.
  - 명령에 따라서는 연산을 수행하는 것이 아니라, 외부 입출력 장치에서 데이터를 읽어 오거나 쓰는 작업을 수행한다.
- Store
  - 필요하면 연산 결과를 다시 메인 메모리에 저장한다.

 컴퓨터에서 '연산(arithmetic operation)'이라는 표현은 정수 및 실수 값을 계산하는 산술적인 연산과 논리적인 연산(bool 대수)을 함께 일컫는 표현이다.

# 클럭(clock)과 컴퓨터 성능

- Clock in CPU
  - CPU 에 내장된 클럭(clock)에서 일정 주기마다 작업 신호 발생
  - 작업 신호가 발생할 때 마다 한 번의 명령을 수행하며, 클럭은 메트로놈, 명령을 처리하는 ALU는 주판에 비유할 수 있다.
- 컴퓨터의 연산 속도는 클럭 수에 비례
  - 1 Khz = 초당 1천회, 1 Mhz = 초당 1 백만, 1 GHz = 10억
- 산술/논리 연산
  - 32 bit CPU는 한번에 32 bit, 64 bit CPU는 한번에 64 bit 연산
  - 32 bit CPU는 클럭 한 번에 4 byte 크기의 변수(혹은 데이터) 값을 비교하거나, 사칙연산으로 계산할 수 있다.



# Register vs. Memory

- 레지스터(register)
  - CPU 내에 존재하는 데이터 임시 저장 공간.
  - 레지스터는 CPU 의 bit 수에 따라 크기가 결정된다.  
(32 bit type CPU는 32 bit, 64 bit CPU는 64 bit 크기)
  - CPU 내에 다양한 용도에 따라 수십개 이상의 레지스터가 존재.
- 메인 메모리(main memory)
  - 실행할 프로그램과 연산 전후의 데이터를 담고 있는 공간.
  - 메인 메모리에 보관된 데이터는 직접 변경될 수 없고, CPU로 이동한 후 CPU의 레지스터에 데이터가 보관된 상태에서 연산(계산)된 후 변경된 값을 다시 메인 메모리로 전송.

# 폰 노이만 아키텍처를 비유하자면...

중앙처리장치 : CPU



ALU + CU



Register

메인 메모리 (Main memory)



- CPU는 타이머와 도마를 가지고 뷔페 요리를 준비하는 요리사에 비유할 수 있다.
- '요리사', '타이머' 그리고 '도마'는 각기 ALU, clock, register와 같다.
- 요리사가 요리 재료를 냉장고에서 조금씩 가져와 도마 위에서 요리한 후, 연회에 내놓듯이 CPU는 처리할 명령과 데이터를 조금씩 가져와 레지스터 내에서 연산한다.

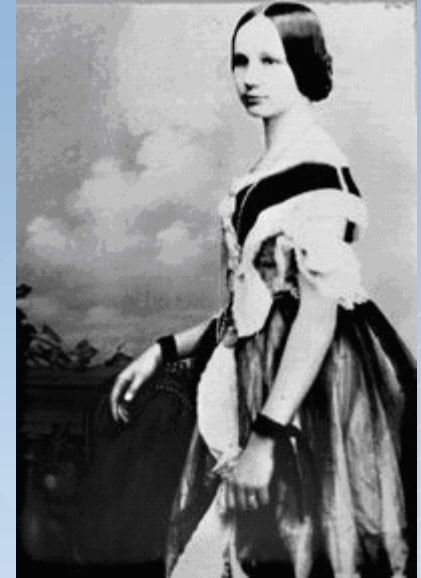
절차 지향 패러다임 중심으로...

# 소프트웨어의 동작 원리



# 최초의 프로그래머 “Ada”

- 에이다 오거스타 킹, 러브레이스 백작부인  
(Ada Augusta King, Lady Lovelace. 1815-1851.)
  - 세계 최초의 컴퓨터 프로그래머이자 수학자.
  - 서유럽 사상 최고의 문호 중 하나로 알려진 바이런 경의 유일한 법적 자식.
- "배비지의 해석기관에 대한 분석  
(Observations on Mr. Babbage's Analytical Engine)" 출간, 1842년
  - 오늘날 컴퓨터의 원형이 된 "해석기관"에 관한 책.
  - 서브루틴(subroutine), 루프(loop), 점프(jump) 개념 뿐만 아니라, "if ~ then" 구문까지 고안.
  - 프로그래밍에서 "If" 구문이 중요한 이유는, 이러한 발명으로 기계가 단순히 계산만 하는 것을 뛰어넘어, 주어진 조건에 따라 "결정"을 내리고 "논리"를 수행하게 됐기 때문이다.



출처 : "세계 최초의 프로그래머는 미모의 귀족부인이었다."  
<http://shunman.tistory.com/99>

# 프로그래밍과 수학의 상관관계

- 프로그래밍의 선구자들은 모두 수학자
  - 에이다, 앨런 튜링, 폰 노이만 등 수학자들이 소프트웨어 이론과 개념을 창안
  - 수학의 논리와 기호(symbol)를 바탕으로 소프트웨어 관련 논문을 작성.
- 소프트웨어의 각종 용어, 기호, 원리는 수학에서 유래한 것이 많다.
  - 함수, 변수와 같은 용어나 알고리즘을 증명하는 방식 등 수학에서 빌려옴.
  - 최초의 소프트웨어들은 실행 명령을 수학 공식을 풀이하듯 절차적(순서대로)으로 작성.
  - 수학자들에게는 너무나 당연하고 익숙한 방식.
  - 그래서인지 모르지만, **폰 노이만 아키텍처의 하드웨어도 '절차적'으로 동작.** 달리 말해, 한 번에 하나씩 작업을 처리한다. 동시에 2개 이상의 명령을 수행하지는 않는다.

# 절차지향 프로그래밍

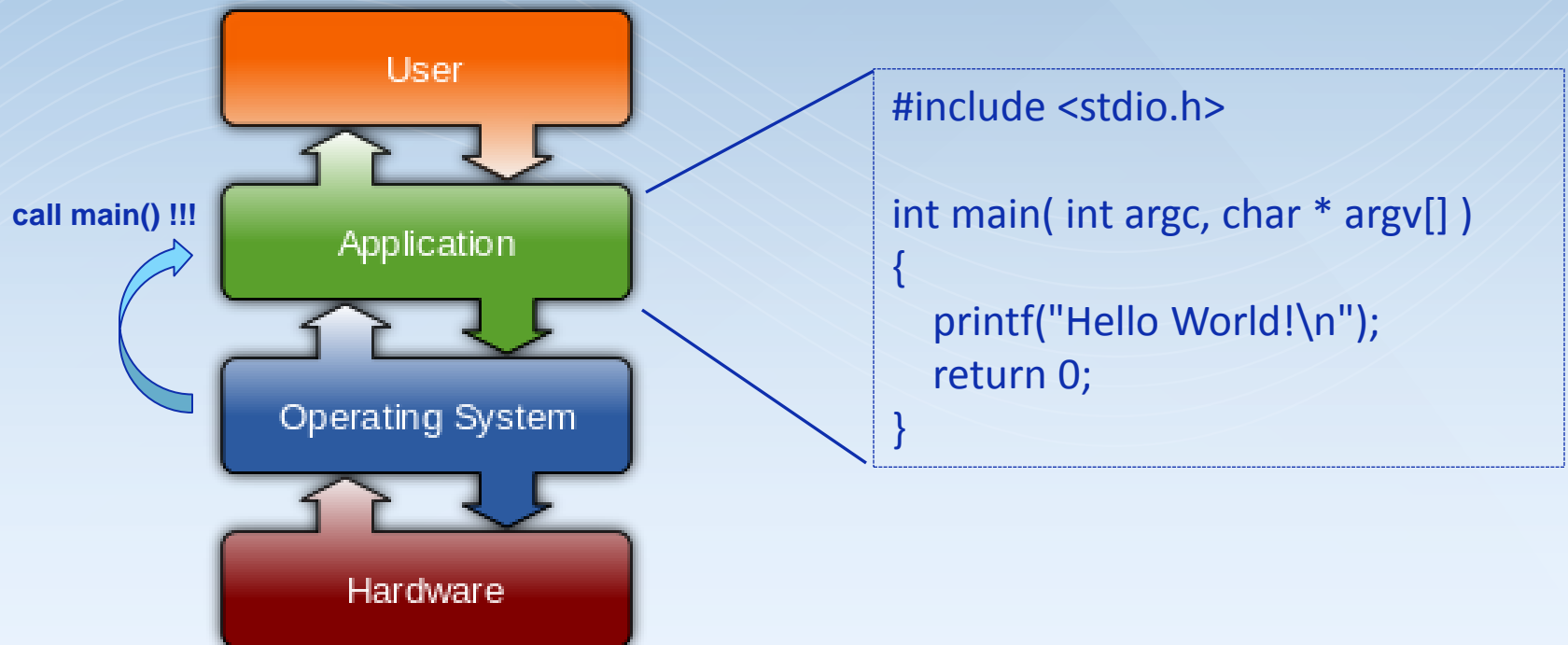
- A.K.A 튜링 머신 (turing machine)
  - 절차지향 프로그래밍은 오늘날 가장 널리 쓰이는 프로그래밍 패러다임으로, 튜링머신으로 압축하여 이야기할 수 있다.
  - 오늘날 컴퓨터 '하드웨어'에 가장 가깝게 대응되기 때문에, 컴퓨터라는 기기를 조작한다는 측면에서 볼때 커다란 이점을 갖고 있다. 어셈블리어, C, C++, Java, Python 등이 모두 여기에 속한다.
- 상태 (state) 중심
  - 절차지향형 언어의 특이점은, 모든것이 '상태'를 중심으로 돌아간다는점
  - 절차지향에서 말하는 '상태'란 메모리 등 저장 공간의 상태를 말한다. 예를 들면, 특정 메모리 주소에 저장된 데이터 혹은 특정 변수의 값을 상태라 할 수 있다.
  - 즉, 절차형 프로그램은 특정 변수나 메모리 주소의 상태를 변경(update)하고, 특정 변수나 메모리 주소의 상태를 읽어서 참조하는 방식으로 만들어진다. 그래서, 절차지향형 프로그램은 상태를 읽고 변경하는 명령문(statement)의 집합으로 만들어진다.
- 절차지향의 장점
  - 하드웨어에 가까운 프로그래밍이 가능하다는 것이 커다란 장점
  - 하드웨어의 성능을 최대한 끌어낼 수 있고, 하드웨어의 상태를 참조하고 조작한다는 가장 직관적인 프로그래밍 패러다임이기때문에, 언어 습득의 진입장벽이 크게 낮다.

# 명령문 그리고 함수 기반의 구조적 프로그래밍

- 명령문(statement)이란?
  - 전처리기(preprocessor)를 제외한 C 언어의 구조를 아주 간략하게 분해해서 보자면, types, statements가 된다.
  - 타입(type)이란 쉽게 말해 숫자나 문자 등 직관적인 데이터의 집합이다.
  - 명령문(statement)은 특정 위치의 상태를 참조하고 업데이트하는 기계가 이해하는 명령이다.
- 명령문의 주요 기능
  - 명령문의 주된 기능은 상태를 참조/조작함과 동시에 실행 순서를 결정하는 것이다. 여러가지 내장 연산자(built-in operator)와 더불어 if ~ else 같은 제어 구조(control structure) 및 for, while 로 대변되는 loop 등이 여기에 속한다.
  - 명령문을 여러개 묶은것을 block 이라 한다. 일반적으로 프로그램은 매우 거대하기 때문에, 이리저리 묶어서 정리하는것이 매우 중요하다. 단순한 statement 의 나열은 해독 불가능한 코드를 만들기때문에 유지보수에 큰 어려움을 겪게 되는게 당연하다. 따라서 이것들을 block 단위로 묶어 사용하는 경우가 많습니다. block 은 control statement, loop 등에 일반적으로 사용된다.
- 그리고, 함수
  - block 보다 좀더 크고 독립적인 단위로 묶는것이 '함수'이다.
  - block 과의 차이점은, 독립적인 실행 환경(environment)를 구성할 수 있다는 점이다.
- 함수 기반의 절차적 프로그래밍 기법 중에서 가급적 goto 명령을 최대한 억제하는 방식을 '구조적 프로그래밍(structured programming) 이라고 한다.

# C 프로그램의 동작 방식 이해

- C 프로그램은 main() 이라는 함수가 운영체제(O/S) 에서 호출되면서 시작되고, 그 main 이라는 함수가 호출되면 main 이름으로 정의된 block 으로 진입하여 block 내의 명령문들을 순서대로 실행하는 것이다. 물론, main 내에서 다른 여러가지 함수를 호출하면서 점차 복잡해진다.



# 하드웨어, 운영체제, 그리고 언어의 결합

- **폰 노이만 아키텍처, UNIX 운영체제 그리고 C 언어**

- 폰 노이만 머신을 잘 살펴보면, CPU가 동작하는 원리가 외부의 입력을 받아 연산을 처리한 후에 그 결과를 출력하는 방식으로 동작한다는 것을 알 수 있다. 소프트웨어의 동작 방식 및 설계도 하드웨어의 동작 원리를 응용한 것이다.
- C 언어와 UNIX 운영체제를 함께 개발한 데이스 리치와 브라이언 커니건이 C 언어를 만들게 된 계기는 UNIX라는 운영체제를 만들게 되고 다양한 하드웨어에 이식하기 쉽도록 당시에 널리 쓰이는 어셈블리(Assembly) 보다는 좀 더 편리한 언어를 만들고자 한 것이다.
- 이들은 하드웨어와 운영체제에 대한 달인이었고, 프로그래밍 언어도 하드웨어와 운영체제의 동작원리를 벗어나지 않는 프로그래밍 언어를 고안한 것이다. 진정한 위대함은 어려운 문제를 쉽게 푸는 능력이다. (Simple is Best!)

- **프로그램은 함수다.**

- 프로그램은 '함수의 집합체' 이다. 그렇다면 프로그램 자체도 함수 아닌가? main 함수는 프로그램이라는 거대한 함수들의 집합체에서 첫 진입점(entry point)이자 마지막 출구(exit point)이다.
- 프로그램도 함수라면 그 함수를 호출하는 주체는 운영체제(Operating System) 이며, 운영체제도 결국 거대한 함수이다. 운영체제가 프로그램을 실행한다는 것은 특정 실행 파일에 존재하는 'main() 함수'를 호출하는 것이다. 함수는 입력과 출력을 가질 수 있다. 유닉스 운영체제는 프로그램이라는 거대 함수가 반환하는 값이 0 (zero) 이면 정상적으로 종료한 것으로 판단한다. 그 외의 값 - 예를 들어 -1 혹은 1, 그외에 모든 0이 아닌 값 -은 비정상적으로 종료한 것을 판단한다.

# main 함수의 argc 그리고 argv

- 왜 main 함수의 인자(parameter)는 2개인가?

- main() 함수는 운영체제가 호출하는 프로그램을 실행하기 위해 호출하는 첫번째 함수이다. 따라서, 운영체제는 프로그램을 실행할 때 사용자가 입력하는 부가적인 옵션(option 혹은 parameter)를 main() 함수에 전달할 수 있어야 한다.
- 프로그램의 인자는 명령행(command line)에서 사용자가 키보드를 이용해 타이핑 하는 것이 일반적이고 그 형태는 복수의 문자열을 공백(space)으로 구분해서 입력한다. 달리 말해서, 0 (zero)개 이상의 문자열 배열 형태라고 표현할 수 있다. 인자의 갯수가 일정하지 않기 때문에 인자의 갯수와 인자 값(문자열)의 목록(혹은 배열)을 함께 전달하며, 인자의 갯수와 인자 배열을 순서대로 전달하기 위해 정수형의 argc 와 문자열 포인터 배열 형태의 argv 변수를 사용한다.

- 변수 명칭의 유래

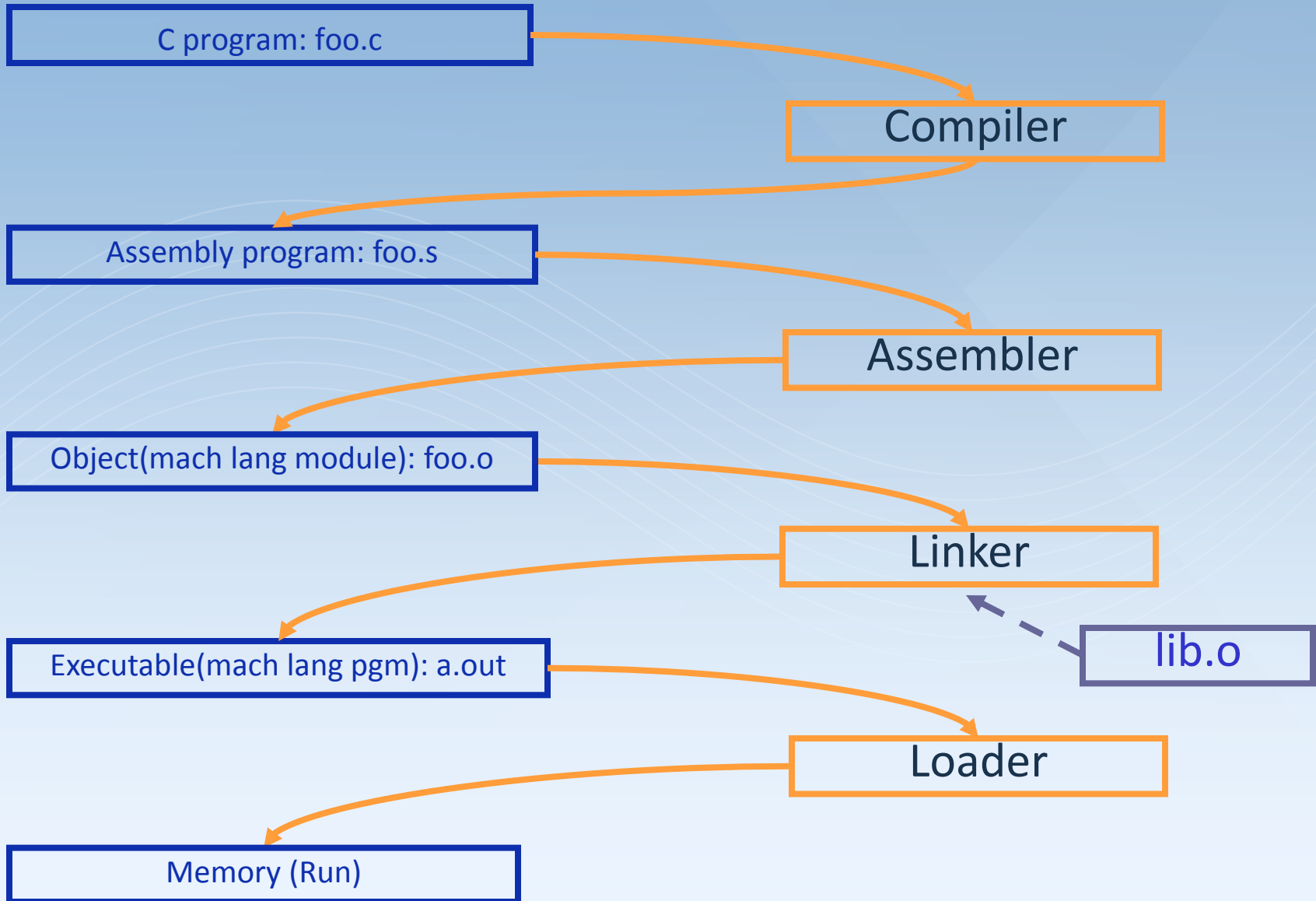
- argc 변수의 공식적인 명칭(full name)은 'argument counter'이고, argv 변수는 argument variable 혹은 argument vector이다.
- ac, av 라던가 argumentcounter, argumentvariable 이라는 극단적으로 길거나 짧지 않은 변수 명칭을 사용하는 이유는 너무 짧아서 모호한 변수 명칭도 배제하고, 너무 길어서 코딩하는데 걸리는 시간을 허비하거나 오타 때문에 고생하지 않도록 실용적인 접근을 시도한 것이다. 의미를 담을 수 있고 최대한 유니크한 짧은 약어(abbreviation)를 이용해 변수 명칭을 부여하는 것이다.



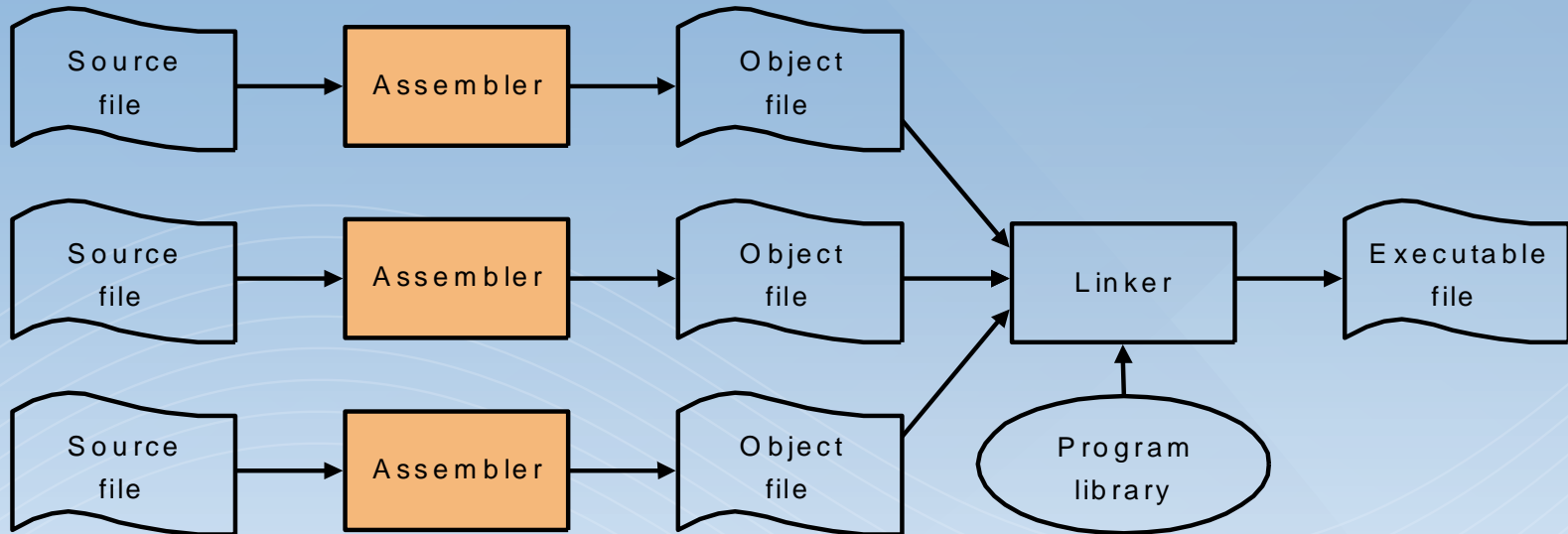
프로그램 소스는 어떻게 컴파일하고 또 실행하는가?

**COMPILE, LINK, LOAD**

# 프로그램을 실행하기까지의 절차



# 컴파일 작업 개요



- 컴파일(compile)이란, 사람이 이해하는 고급 프로그래밍 언어(high level programming language)를 기계어(machine language) 코드로 변환한 후, 최종적으로 운영체제에서 실행할 수 있는 형태의 파일로 만드는 작업을 말한다.
- 요약해서 컴파일(compile)이라고 표현하나, 그 내부 과정에서는 compiler, assembler, linker 등의 프로그램이 동작한다.
- 현대적인 컴파일러들은 3가지 프로그램을 모두 내장하고 있어서 한번에 모든 작업을 처리할 수 있다.

# 소스, 어셈블리, 기계어 코드 비교

```
void main()
{
    int n,s;

    printf("Enter upper limit: ");
    scanf("%d",&n);
    s = count(n);
    printf("Sum of i from 1 to %d =
    %d\n",n,s);
}
```

```
.LCFI1:
    sd    $28,32($sp)
.LCFI2:
    move  $fp,$sp
.LCFI3:
    .set   noat
    lui   $1,%hi(%neg(%gp_rel(count)))
    addiu $1,$1,%lo(%neg(%gp_rel(count)))
    daddu $gp,$1,$25
    .set   at
    sw    $4,16($fp)
    sw    $0,24($fp)
    li    $2,1        # 0x1
    sw    $2,20($fp)
.L3:
    lw    $2,20($fp)
    lw    $3,16($fp)
    slt   $2,$3,$2
    beq   $2,$0,.L6
    b     .L4
```

```
00000000 7f45 4c46 0102 0100 0000 0000 0000 0000
00000020 0002 0008 0000 0001 1000 1060 0000 0034
00000040 0000 6c94 2000 0024 0034 0020 0007 0028
00000060 0023 0022 0000 0006 0000 0034 1000 0034
00000100 1000 0034 0000 00e0 0000 00e0 0000 0004
00000120 0000 0004 0000 0003 0000 0114 1000 0114
00000140 1000 0114 0000 0015 0000 0015 0000 0004
00000160 0000 0001 7000 0002 0000 0130 1000 0130
00000200 1000 0130 0000 0080 0000 0080 0000 0004
00000220 0000 0008 7000 0000 0000 01b0 1000 01b0
00000240 1000 01b0 0000 0018 0000 0018 0000 0004
00000260 0000 0004 0000 0002 0000 01c8 1000 01c8
00000300 1000 01c8 0000 0108 0000 0108 0000 0004
00000320 0000 0004 0000 0001 0000 0000 1000 0000
00000340 1000 0000 0000 3000 0000 3000 0000 0005
.....
```

C 언어 소스

Assembly 소스

기계어 코드

Compiler

Assembler

# 컴파일러 (Compiler)

- 입력 (Input)
  - 고급 언어의 소스 코드 (High-Level language code)
  - 예를 들어 C, Java 등
- 출력 (Output)
  - 어셈블리 언어 코드 (Assembly language code)
  - 예를 들어, Intel x86, MIPS 등
- 수행 작업
  - 고급 언어의 코드를 의사 명령(pseudo instruction)으로 변환한다.
  - 의사 명령은 어셈블러(assembler)는 이해하나 기계는 이해하지 못하는 명령을 말한다. (예를 들면, `mov $s1, $s2 = or $s1, $s2, $zero`)

# 어셈블러 (Assembler)

- 입력 (Input)
  - 어셈블리 언어 코드 (Assembly language code)
- 출력 (Output)
  - 거의 완벽한 형태의 실행 코드  
(Nearly-complete image of executable code)
- 수행 작업
  - 어셈블리 명령을 16진수의 기계어 코드로 변환한다.  
(Binary encoding of each instruction)
  - 서로 다른 바이너리 파일 간의 주소 참조(reference)가 확정되지 않은 상태  
(Missing linkages between code in different files)

# 링커 (linker)

- 입력 (Input)
  - 하나 이상의 바이너리 파일
- 출력 (Output)
  - 실행 가능한 프로그램 파일
- 수행 작업
  - 서로 다른 바이너리 파일 간의 참조 주소(포인터)를 계산 및 설정  
(Resolves references between files)
  - 정적인 런타임 라이브러리를 결합  
(Combines with static run-time libraries, e.g., code for `malloc`, `printf`)
  - 일부 라이브러리는 동적으로 결합  
(Some libraries are *dynamically linked*, linking occurs when program begins execution)



# 로더 (loader)

- 로더의 역할
  - 실행 파일을 디스크에 저장되어 있다.
  - 실행 파일이 실행되면 로더는 실행 파일을 메모리로 적재(load)한 후 실행을 시작한다.
  - 실질적으로 로더는 운영체제의 기능 중 하나이다.
- 로더의 작업 순서
  - 실행 파일의 헤더(header)를 읽어 필요한 메모리 용량을 계산한다.
  - 메모리를 할당(allocate) 한 후, 프로그램을 메모리에 적재한다.
  - 메인 함수에서 사용할 수 있도록 프로그램 실행 인자를 스택(stack)에 복사한다.
  - 프로그램 시작 전에 CPU 레지스터를 초기화 한다.
  - 프로그램의 메인 함수를 시작한다.
  - 프로그램이 종료하면 메인 함수의 반환 값을 운영체제에 전달한다.