* **리눅스 프로그램**
  + 리눅스 프로그램 파일 형식
    - execuable(실행가능) 파일 : 컴퓨터가 직접 실행할 수 있는 프로그램
    - script(스크립트) : interpreter가 실행할 명령들의 집합
  + 사용자는 shell과 상호작용 하면서 프로그램을 실행(shell 프로그램은 사용자가 요청한 프로그램의 이름과 동일한 이름을 가진 파일을 PATH에 지정된 디렉터리들에서 찾는다.)
    - /bin : Binaries. 시스템을 부팅 하는데 쓰이는 프로그램들이 있다.
    - /usr/bin : User Binaries. 사용자가 실행할 수 있는 표준 프로그램들이 있다.
    - /usr/local/bin : Local Binaries. 해당 설치본에 고유한 프로그램들이 있다.
* **C 컴파일러**
  + POSIX 호환 시스템에서 C컴파일러의 이름은 **c89**이다.(역사적으로 C컴파일러는 **cc**로 불려왔다.)
  + 리눅스 시스템의 경우 **c89**, **cc**, **gcc** 명령 모두 단일한 시스템 C컴파일러를 가리킨다.
  + **gcc**
    - **-o** : 컴파일러에게 실행 파일의 경로와 이름을 알려준다.(이 옵션을 생략하면 컴파일러는 **a.out(assembler output)**이라는 파일을 생성)
    - **-I** : 헤더 파일의 위치를 명시할 때 사용하는 플래그(**gcc -I/usr/openwin/include fred.c**). 컴파일러는 fred.c가 포함하는 헤더 파일들을 찾을 때 다른 표준 디렉터리들과 함께 /usr/openwin/include 에서도 같이 찾는다.
    - **-l** : 라이브러리의 이름을 지정할 때 쓰는 플래그
    - **-L** : 라이브러리 검색 경로를 지정할 때 쓰는 플래그
* **개발 시스템**
  + 응용프로그램
    - 일반적으로 응용프로그램들은 각자 자신만의 디렉터리에 저장된다.
    - **/usr/bin** : 프로그램 개발을 비롯한 일반적인 용도를 위해 시스템이 제공하는 기본 응용프로그램들의 일반적인 위치
    - **/usr/local/bin** 또는 **/opt** : 호스트 컴퓨터나 지역 네트워크를 위해 시스템 관리자가 추가한 응용프로그램들의 일반적인 위치(**/opt** 또는 **/usr/local**을 사용하면 서드파티 제작사가 제공했거나 나중에 추가한 파일들과 시스템이 제공한 파일들을 구분할 수 있다.)
  + 헤더파일
    - 상수 정의나 시스템 및 라이브러리 함수 호출 선언들을 담은 파일
    - C의 경우 거의 항상 **/usr/include와 그 하위 디렉터리들**에 들어 있다.
    - 리눅스 설치본에 의존적인 헤더 파일들은 보통 **/usr/include/sys**와 **/usr/include/linux**에 있다.(비표준 장소에 있는 헤더 파일들을 명시적으로 찾아야할 때는 **-I**플래그를 사용 → **gcc -I/usr/openwin/include fred.c**)
  + 라이브러리 파일
    - 재사용을 염두에 두고 작성한 함수들을 미리 컴파일한 경과를 모아둔 파일
    - 공통의 과제를 수행하기 위한 연관된 함수들로 구성
    - 일반적으로 표준 시스템 라이브러리들은 **/lib**와 **/usr/lib**에 들어 있다.
    - **Linker**는 기본적으로 표준 C라이브러리만 검색하므로, 그 외의 라이브러리를 사요하는 경우에는 라이브러리의 위치와 이름을 linker에게 알려주어야 한다.
    - 라이브러리 파일의 이름을 특정 규약에 맞게 지어야 하며, 명령줄에서 라이브러리 파일의 위치와 이름을 명시적으로 지정해 주어야 한다.
      * 라이브러리 파일 이름은 항상 **lib**로 시작해야한다.
      * lib 다음에 라이브러리 이름이 붙는다.(C라이브러리의 경우 c, 수학 라이브러리의 경우 m)
      * 라이브러리의 종류를 의미하는 확장자가 붙는다.
        + **.a** : 정적 라이브러리
        + **.so** : 공유 라이브러리
      * 라이브러리 파일 이름을 지정할 때는 전체경로 이름을 지정할 수 있고, **-l**플래그를 이용할 수도 있다.(**gcc -o fred fred.c /usr/lib/libm.a** 또는 **gcc -o fred fred.c -lm**)
      * 라이브러리가 표준 장소에 없는 경우에는 **-L**옵션을 이용해서 라이브러리 검색 경로를 지정할 수 있다.(**gcc -o x11fred -L/usr/openwin/lib x11fred.c -lX11**)
  + 정적 라이브러리(static library)
    - **목적 파일(object file)**들을 즉시 사용할 수 있는 형태로 모아둔 것
    - 정적 라이브러리를 **아카이브(achive)**라고도 한다.
    - 함수들을 gcc로 컴파일해서 얻은 목적 파일들을 **ar**프로그램으로 합치면 정적 라이브러리를 만들 수 있다.
    - 정적 라이브러리의 단점
      * 동일한 라이브러리의 함수들을 사용하는 여러 응용프로그램들을 동시에 실행하는 경우, 메모리 안에 동일한 함수의 복사본이 여러 개 존재하게 된다.(프로그램 파일들 자체에도 수많은 복사본들이 중복해서 존재하는데, 그러면 메모리와 디스크 공간이 크게 낭비될 수 있다.)
  + 공유 라이브러리(shared library)
    - **정적 라이브러리의 단점을 극복**
    - 프로그램이 공유 라이브러리를 사용하는 경우, 컴파일러는 함수 코드 자체 대신 **실행 시점**에서 쓰일 **공유 코드에 대한 참조 정보를 프로그램에 포함**시킨다.
    - 실행을 위해 프로그램을 메모리에 적재하는 과정에서 시스템은 함수 참조들을 분석해 공유 라이브러리의 해당 코드에 연결 시킨다.(아직 메모리에 적재되지 않았다면 시스템이 적재)
    - 시스템은 공유 라이브러리의 복사본 하나를 여러 응용프로그램들이 동시에 사용할 수 있도록 메모리 안에 공유 라이브러리를 배치(디스크에도 공유 라이브러리 복사본이 하나만 있으면 된다.)
    - **동적 로더** : 공유 라이브러리를 적재하고 클라이언트 프로그램 함수 참조를 분석, 해소하는 프로그램(리눅스의 경우 **ld.so**)
    - 공유 라이브러리들을 검색하는 추가적인 장소들은 **/etc/ld.so.conf** 파일에서 설정 할 수 있다.(변경 후 반드시 **ldconfig**를 실행해서 변경을 시스템에 반영해 주어야 한다.)
* **쉘 프로그래밍**
  + 쉘은 빠르고 간단하게 프로그래밍 할 수 있다.
  + 쉘은 비교적 간단하고 반복 적인 과제를 수행하되 효율성보다는 쉬운 설정, 유지 보수, 이식성이 더 중요한 작은 유틸리티를 만들 때에도 이상적이다.
  + 쉘을 프로세스 제어에 사용할 수 있으므로, 여러 명령들을 미리 결정된 순서에 따라 실행하되 이전 단계의 성공 여부에 따라 실행을 제어하는 등의 작업이 가능하다.
  + 스크립트(script 또는 shell program)은 실행 시점에 해석되어서 실행된다.
    - 스크립트를 고칠 때마다 매번 스크립트를 다시 컴파일 하는데 시간을 낭비할 필요가 없다.
    - 스크립트를 한 줄씩 실행할 수 있기 때문에, 일반적으로 디버깅이 쉽다.
    - 매번 해석이 필요하므로 속도가 중요하거나 프로세서를 과도하게 사용하는 경우 부적합하다.
  + 쉘 프로그래밍 방법
    - 대화식
      * 키보드로 일련의 명령들을 입력해서 쉘이 실행하게 하는 방법
    - 프로그램
      * 명령들을 파일에 작성하고 파일을 프로그램으로 실행하는 방법(**쉘 스크립트**)
* **쉘 스크립트**
  + **#** : comment
  + **#!** : 시스템이 이 파일을 실행할 때 사용할 프로그램 이름
    - 절대 경로(**#!/bin/sh**)
    - 경로는 32자 미만으로 주는 것이 관례이나 리눅스는 일반적으로 제한이 없다.(글자 수에 제한을 두는 오래된 UNIX 버전들이 존재)
  + 스크립트는 본질적으로 쉘에 대한 표준 입력으로 취급되므로, **PATH** 환경 변수를 통해 접근할 수 있는 리눅스 명령이라면 스크립트에서 사용할 수 있다.
  + **exit** : 스크립트가 의미 있는 종료 코드를 돌려주게 한다.
    - 한 스크립트를 다른 스크립트에서 실행해서 해당 스크립트가 성공적으로 수행 되었는지를 점검해야 하는 경우에는 스크립트가 적절한 종료 코드를 돌려주게 하는 것이 중요하다.
  + 변수
    - 쉘의 변수는 최초로 쓰이는 지점에서 자동으로 생성된다.
    - 모든 변수들에 저장되는 값은 문자열로 간주된다.(숫자도 문자열로 간주)
      * 쉘과 일부 유틸리티들은 필요에 따라 수치 값 형태의 문자열을 적절히 실제 수치 값으로 변환한다.
    - 리눅스는 영문 대소문자를 구분한다.
    - 변수에 접근할 때는 **$**를 변수 이름에 붙인다($PATH)
    - 스크립트의 매개 변수들은 일반적으로 공백 문자(빈 칸, 탭, 새 줄)로 구분한다.(매개 변수 자체에 공백 문자가 포함된 경우 매개 변수를 따옴표로 감싸야 한다.)
    - 따옴표 안에 ‘$’ 변수 표현식이 있는 경우
      * 큰 따옴표(“”) : 해당 줄이 실행될 때 변수 표현식이 해당 변수의 내용으로 치환
      * 작은 따옴표(‘’) : 치환되지 않음(‘$’를 문자로 인식)
    - ‘$’를 문자로 인식하고 싶을 때 ‘**\**’를 붙인다.(**\$**)
    - 환경 변수
      * 환경에 설정된 값들로 초기화되는 변수
      * 스크립트 안의 사용자 정의 변수들과 구분 할 수 있도록, 환경 변수들의 이름은 일반적으로 대문자로만 구성된다.
      * **$#** : 전달된 매개 변수
      * **$0** : 쉘 스크립트의 이름
      * **$IFS** : 입력 필드 구분자. 쉘이 입력을 읽을 때 단어들을 구분하는 용도로 쓰이는 문자들의 목록(일반적으로 반칸, 탭, 새 줄 문자)
      * **$$** : 쉘 스크립트의 PID. 스크립트에서 고유한 임시 파일 이름을 만들 때 흔히 쓰인다.
    - 매개변수 참조 변수
      * **$1, $2, .…** : 스크립트에 주어진 매개 변수들(10번째부터는 중괄호를 써야한다 → **${10}, ${11}, …**)
      * **$\*** : 매개 변수들 모두를 환경 변수 IFS의 첫 문자로 구분한 형태의 문자열을 담은 하나의 변수.
      * **$@** : $\*와 비슷하되 IFS 환경 변수를 사용하지 않는다.
  + **read** : 사용자 입력을 받아서 변수에 저장(입력 내용을 받을 변수명을 매개변수로 하나 받는다. 터미널에서 텍스트를 입력 할 때는 빈 칸이 포함된 경우라도 “”로 감쌀 필요가 없다.)
  + 조건문
    - **test 또는 [** : 쉘의 부울 판정 구문(‘[’를 쓰는 경우 관례상 가독성을 위해 ‘]’를 덧붙인다.)
      * test (‘[’) 명령에 사용할 수 있는 조건들 종류
        + 문자열 비교
        + 산술 비교
        + 파일 조건
      * $ **help test** 참조
  + 제어 구조
    - **if** : if 조건; then 명령문들 elif 조건; then 명령문들 else 명령문들 fi
    - **for** : for 변수 in값들; do 명령문들 done
    - **while** : while 조건; do 명령문들 done
    - **until** : until 조건; do 명령문들 done (조건이 참이 될 때까지 반복)
    - **case** : case 변수 in 패턴 [ | 패턴 ] …) 명령문들**;;** … esac
    - **AND(&&)** : 명령문1 && 명령문2 && …
    - **OR(||)** : 명령문1 || 명령문2 || … (하나라도 참인 것이 있으면 수행을 멈춘다)
    - [ -f file\_one ] && 명령문1 || 명령문2 → file\_one이 존재하면 명령문1이, 존재하지 않으면 명령문2가 실행(괄호를 이용해서 조건들의 평가 순서를 명확하게 지정하는 것이 바람직)
    - **{}** : 명령문들을 묶음
  + 함수
    - **스크립트 구조화**
      * 스크립트 구조화의 다른 방식은 여러 개의 간단한 과제를 수행하는 스크립트들로 분할.
        + 단점

한 스크립트에서 다른 스크립트를 실행하는 것은 스크립트에서 함수를 실행하는 것보다 느리다.

수행 결과를 돌려주기가 더 힘들다

작은 스크립트들이 아주 많아져서 관리하기가 힘들어질 수 있다.

* + - **함수명 () {명령문들}**
    - 함수가 호출되는 곳 이전에 함수가 정의되어있어야 한다.
    - 함수는 함수 호출에 주어진 인수들을 위치 매개변수들($\*, $@, $#, $1 … )을 이용해서 읽을 수 있다.
    - 스크립트 자체의 위치 매개 변수들을 함수 호출의 인수들로 사용하는 것도 가능하다.(**foo $1**)
      * 이 경우 해당 매개변수들은 함수 내부의 매개변수들로 치환되며, 함수의 수행이 끝나면 다시 원래의 값으로 복원된다.
      * 구식 쉘들 중에는 함수 수행 이후 위치 매개변수들의 값을 복원하지 않는 것들도 있다.
    - 함수는 **return** 명령으로 수치 값을 호출한 곳에 돌려 줄 수 있다.
      * 문자열을 직접 돌려 줄 수는 없지만, 우회책으로 함수 안에서 문자열을 하나의 변수에 저장하고 함수를 호출한 곳에서 그 문자열을 사용 하거나 echo로 출력하되 그것을 하나의 변수에 저장하는 방법이 있다.
      * return 문이 없는 경우, 함수는 마지막으로 수행된 명령의 종료 코드를 돌려 준다.
    - 쉘 변수들은 본질적으로 전역 변수이지만 **local** 키워드를 이용해 지역 변수를 정의할 수 있다.
      * 같은 이름의 전역 변수가 존재한다면, 함수 안에서는 전역 변수는 무시되고, 지역 변수가 쓰인다.
  + 명령
    - 종류
      * 외부 명령(external command)
        + 스크립트 뿐만 아니라, 명령 프로프트에서도 사용할 수 있는 명령
      * 내부 명령(internal command 또는 내장 명령(built-in command))
        + 쉘 자체에 구현되어 있기 때문에 외부 프로그램으로써 실행할 수는 없다.
    - 일반적인 경우에는 어떤 명령이 내부 명령인지 외부 명령인지를 구분할 필요가 없다.(내부 명령이 더 효율적으로 수행된다.)
    - **break** : for, while, until 루프를 종료 조건이 만족되기 전에 벗어나고자 할 때 사용
    - **:** : 아무 일도 하지 않는 무연산 명령(null operation). 이 명령의 결과는 항상 참이다. ‘:’은 **내장 명령**이므로 외부 true 명령 보다 빠르지만 가독성을 위해 true가 자주 쓰인다.
    - **continue** : for, while, until 루프에서 실행이 루프의 끝에 도달하기 전에 루프를 다음 번 반복으로 넘어가게 만들고 싶을 때 사용.
    - **.** : 현재 쉘에서 명령 수행. 일반적으로 스크립트에서 외부 명령이나 외부 스크립트를 수행하는 경우 새로운 환경(하위 쉘)이 만들어지고, 외부 명령은 그 새 환경 안에서 수행된다.(외부 명령 **source** 나 ‘**.**’명령을 사용하면 해당 스크립트에 있는 명령들이 원래의 스크립트와 동일한 쉘 안에서 수행된다.)
    - **eval** : 주어진 인수들을 평가한다. eval은 내장 명령이며 외부 명령으로는 존재하지 않는다. 실행 시점에서 코드를 생성하고 실행할 수 있다.
    - **exec** : 현재 쉘을 다른 프로그램으로 대체한다.(exec wall “Thanks for all the fish“ → 현재 쉘이 wall명령으로 대체된다. 이 경우 스크립트에서 exec명령 이후에 있는 중들은 전혀 실행되지 않는다(스크립트를 실행하던 쉘 자체가 사라짐)). 현재 파일 서술자를 수정한다.(exec 3< afile → 파일 서술자 3이 afile이라는 파일을 읽기 위해 열린다.)
    - **exit n** : 인수 n을 종료 코드로 해서 스크립트의 수행을 끝낸다. 스크립트에 exit명령이 없는 경우, 스크립트가 돌려주는 종료 코드는 스크립트 안에서 마지막으로 수행된 명령의 결과와 같지만 항상 명시적으로 종료 코드를 돌려주는 것이 바람직하다.
      * 0 : 성공
      * 1 ~ 125 : 사용자 정의 오류 코드
      * 126 : 파일이 실행 가능한 파일이 아니었음
      * 127 : 명령을 찾지 못했음
      * 128 이상 : signal이 발생했음
    - **export** : 변수 이름을 하위 쉘의 매개 변수들로 사용할 수 있게 만든다. 기본저긍로 한 쉘 안에서 생성된 변수는 그 쉘에서 호출한 다른 쉘들에서 사용하지 못한다. export 명령은 매개 변수를 환경 변수로 만들어서 현재 프로그램이 호출한 다른 스크립트나 프로그램이 그 매개 변수를 볼 수 있게 만든다. Export 명령으로 내보낸 변수는 자식 프로세스들에서 **환경 변수**로 존재한다.
      * **set -a** 명령이나 **set -allexport** 명령은 이후의 모든 변수를 내보낸다
    - **expr** : 주어진 인수들을 하나의 표현식으로 평가한다.