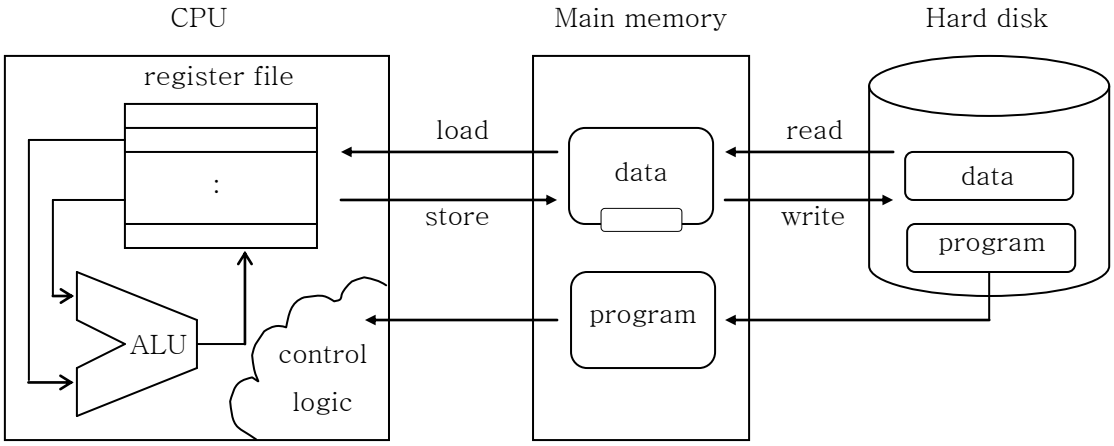
1. 컴퓨터의 구조 및 프로그램 실행 과정

컴퓨터 시스템의 구성을 간단히 살펴보면, 두뇌에 해당하는 중앙처리장치(CPU)와 중앙처리장치가 처리할 프로그램과 데이터를 임시로 저장하는 주기억장치(Main Memory), 프로그램과 데이터 등을 반영구적으로 저장할 수 있는 하드디스크를 비롯해서, 키보드, 모니터, 프린터 등의 주변장치로 구성되어 있다.



우선 가장 중요한 CPU의 구조를 좀더 자세히 살펴보면, CPU는 덧셈, 뺄셈과 같은 산술연산과 논리합, 논리곱과 같은 논리연산을 수행하는 ALU(Arithmetic and Logic Unit)가 있으며 ALU가 연산을 수행할 때 그 대상이 되는 데이터를 임시로 저장하는 CPU내의 저장 공간인 레지스터(register)가 있다. ALU는 항상 이 레지스터로부터 데이터를 가져와서 연산하고 그 결과 값 역시 다시 레지스터에 저장하게 된다. 보통 CPU내에는 이런 레지스터들이 다수 존재하게 되는데 CPU가 명령을 수행하는 데 필요한 시스템 레지스터와 프로그래머에 의해서 사용 가능한 범용 레지스터들이 있다. 보통 범용 레지스터들의 묶음을 레지스터 파일이라고 하고 레지스터 파일에 있는 각각의 레지스터들은 번호나 이름으로 구별하게 된다. CPU내에는 ALU와 레지스터 파일 외에 명령을 해석한 후에 실행을 위해 ALU와 레지스터 파일을 제어하는 회로가 존재하게 되는데 이를 컨트롤 로직(control logic)이라고 한다.

보통 프로그래머가 작성한 프로그램은 CPU가 바로 해석할 수 있는 기계어의 형태로 번역되어 하드디스크에 저장되었다가 운영체제에 의해서 실행될 때 주기억장치에 올려지고 CPU의 컨트롤 로직에 의해서 하나씩 실행되게 된다. 프로그램이 실행될 때 필요한 데이터 역시 하드디스크에 저장되었다가 메모리를 통하여 레지스터에 저장된 후 연산에 사용된다. 연산의 결과 값은 반대로 레지스터에 저장된 후에 메모리를 거쳐 하드디스크에 저장하게 된다. 물론 입력데이터는 하드디스크가 아닌 키보드나 다른 입력장치를 통해 입력 될 수도 있으며 출력 역시 모니터나 프린터 등의 장치가 될 수 있다.

2. 프로그램과 프로그래밍 언어

기계어(machine language)

프로그램이라는 것은 최종적으로는 CPU의 컨트롤 로직이 명령을 분석해서 어떤 작업을 수행할 수 있는 0과 1의 비트조합을 말한다. 이 비트조합은 CPU의 설계에 따라서 일정한 길이와 형태로 작성되어야 하며 이렇게 만들어진 프로그램을 기계어 프로그램이라고 한다. 따라서 기계어로 작성된 프로그램은 CPU가 바로 해석해서 실행할 수 있는 프로그램이다.

어셈블리어(assembly language)

컴퓨터를 사용한 초기에는 이런 기계어로 프로그램을 작성했지만 점차 컴퓨터로 처리하는 일이 많아지고 프로그램의 크기가 커지면서 기계어로 프로그램을 작성하는 데는 한계에 부딪치게 된다. 기계어로 프로그램을 작성한다는 것은 해당 CPU가 해석할 수 있는 명령어의 형태를 일일이 외워야 하며 프로그램을 작성한 후에도 작성한 사람조차 이해하기 힘들어서 에러수정이 힘들고 다른 사람이 이해하기에는 더욱 어렵기 때문이다. 이런 프로그래밍의 어려움을 조금이나마 해소하고자 나온 것이 어셈블리어이다. 어셈블리어는 0과 1로 되어있는 기계어 명령코드를 인간이 이해하기 쉬운 기호의 형태로 바꾸어서 프로그래밍을 하는 것이다. 어셈블리어로 작성된 코드는 CPU가 바로 해석할 수 없으며 기계어로 바꾸는 프로그램인 어셈블러에 의해서 기계어로 바꾸는 과정이 필요하다. 어셈블리어도 기계어 명령을 단지 기호로서 읽기 쉽게 한 것 뿐이고 본질은 여전히 CPU의 설계에 의존적이다. 따라서 서로 다른 설계에 의해 만들어진 CPU는 서로 다른 어셈블리어를 갖는다. 우리는 통상 기계어와 어셈블리어를 저급어(love level language)라고 부른다.

고급언어(high level language)

프로그래밍 언어에 대한 연구가 진행되면서 CPU의 특성을 타지 않으며 인간이 작성하고 이해하기 쉬운 프로그래밍 언어를 개발하게 되었는데 FORTRAN, COBOL, C 등 무수히 많은 언어가 나오게 되었고 이와 같은 언어를 고급언어라고 부른다. 고급언어 역시 CPU에 의해서 실행되기 위해서는 기계어 형태로의 변환과정을 거쳐야 하는데 이 과정을 컴파일이라고 한다. 컴파일 해주는 프로그램을 컴파일러라고 하며 컴파일러는 번역하고 실행하는 방법에 있어서 두 가지 형태로 나누어 볼 수 있다. 하나는 소스코드 전체를 기계어 코드로 번역

해서 실행파일의 형태로 저장해 놓았다가 실행 시에 바로 실행시킬 수 있는 형태인 컴파일러형이 있고, 또 하나는 프로그램 실행 시에 프로그램의 코드를 한 줄씩 번역해서 번역과 동시에 실행시키는 인터프리터형이 있다. 컴파일러형은 한번 번역해 놓으면 바로 실행시킬 수 있으므로 실행속도가 빠르며 FORTRAN, C 등 대부분의 컴파일러가 컴파일러형에 속한다. 인터프리터형은 번역하면서 실행시키므로 실행속도가 느리지만 소스코드를 작성하고 수정하면서 바로 실행시켜 볼 수 있는 장점이 있다. 교육용으로 개발되었던 초기의 BASIC과 인공지능언어인 LISP 등이 대표적인 인터프리터형 언어이다.

3. 저급어와 고급어로 프로그램 작성

주소(address)

프로그램을 작성해서 컴퓨터로 어떤 작업을 수행하는 것은 컴퓨터에 처리할 데이터를 넣어주고 프로그램에 따라 데이터를 처리한 후에 그 결과를 얻는 과정이라고 할 수 있다. 일단 컴퓨터가 처리할 데이터는 파일의 형태로 하드디스크에 저장되어 있든지, 키보드나 다른 입력장치를 통해서 컴퓨터에 전해지게 되는데 이러한 데이터 들은 CPU에 의해서 처리되기 전에 반드시 일단 메모리에 저장된다. 그리고 메모리에 저장된 데이터들은 그 위치에 따라서 서로 구별하게 되며 이렇게 데이터가 저장된 메모리의 위치를 주소라고 한다. 메모리의 주소는 바이트 단위로 매겨져 있으며 예를 들어 메모리의 용량이 64K 라면 64K 바이트의 저장공간이 있는 것이고, 메모리의 주소는 0번지부터 65535번지까지 존재한다.

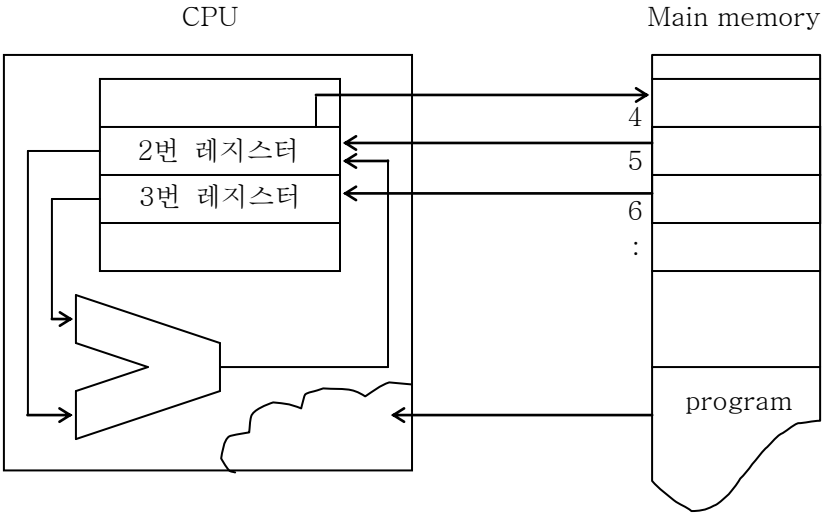
명령(instruction)

그럼 메모리의 특정 위치 5번지와 6번지에 있는 값을 더해서 4번지에 저장하는 프로그램을 작성해 보자. 일단 극히 제한된 세가지 명령만을 수행할 수 있는 가상의 CPU를 설정하고 이 CPU로 기계어와 어셈블리어 그리고 고급언어로 어떻게 프로그램이 작성되는지 간단히 살펴본다. 우리가 만든 가상의 CPU는 다음의 세가지 명령을 수행한다.

첫째, 더하는 명령 => 기계어 코드 001
둘째, 메모리에서 레지스터 파일로 load하는 명령 => 기계어 코드 010
셋째, 레지스터 파일에서 메모리로 store하는 명령 => 기계어 코드 100

기계어 프로그램

위의 세가지 명령을 이용해서 기계어 프로그램을 짜보면, 우선 메모리의 5번지와 6번지의 값을 두 개의 레지스터로 옮기는 명령과 이 레지스터의 값을 더하는 명령과 더해진 결과를 다시 레지스터에 저장하고, 최종적으로 이 더해진 레지스터의 값을 메모리의 4번지로 저장하는 순서로 진행된다.



010 010 101 => load, 레지스터 번호, 메모리 위치
=> 메모리 5번지의 값을 2번 레지스터로 load해라
010 011 110 => load, 레지스터 번호, 메모리 위치
=> 메모리 6번지의 값을 3번 레지스터로 load해라
001 010 011 => 덧셈, 레지스터 번호, 레지스터 번호
=> 2번, 3번 레지스터의 값을 더해서 2번 레지스터에 저장해라
100 010 100 => store, 레지스터 번호, 메모리 위치
=> 2번 레지스터의 값을 메모리 주소 4번지에 저장해라

어셈블리어 프로그램

위와 같이 0과 1로 된 기계어 프로그램을 작성하면 CPU에 의해서 바로 실행 할 수 있으나 코드 자체를 쉽게 이해할 수 없다. 따라서 위 기계어 코드를 기호화 해서 어셈블리어로 다시 작성할 수 있는데 덧셈명령은 AR(Add Register)로, load명령은 L, store명령은 ST로 각각 바꾸고 레지스터 번호는 십진 숫자로, 그리고 메모리의 주소는 알파벳을 이용한 이름으

로 각각 바꾸어 볼 수 있다.

010 010 101 => L 2, A (메모리 주소 5번지)
=> A라는 이름의 메모리 위치의 값을 2번 레지스터로 load해라
010 011 110 => L 3, B (메모리 주소 6번지)
=> B라는 이름의 메모리 위치의 값을 3번 레지스터로 load해라
001 010 011 => AR 2, 3
=> 2번, 3번 레지스터의 값을 더해서 2번 레지스터에 저장해라
100 010 100 => ST 2, C (메모리 주소 4번지)
=> 2번 레지스터의 값을 C라는 이름의 메모리 위치에 저장해라

고급어 프로그램

이렇게 어셈블리 코드로 작성된 프로그램은 기계어 보다 훨씬 이해하기 쉬운 형태의 프로그램이지만 어셈블리 코드를 실행시키기 위해서는 다시 기계어로 바꾸는 어셈블 과정이 필요하다. 결론적으로 기계어와 어셈블리어로 프로그램을 작성하는 것은 해당 CPU가 처리할 수 있는 명령의 형태를 알아야 하며 프로그램이 실행되는 과정에서 메모리와 레지스터, ALU 등의 하드웨어 장치들이 어떻게 작동하는지에 대한 이해가 선행되어야 한다. 하지만 위의 프로그램을 인간이 이해하는 수준에서 생각해 보면 결국 A와 B라는 메모리 위치에 있는 두 값을 더해서 C라는 메모리 위치에 저장하는 프로그램이다. 결국 인간이 보다 이해하기 쉬운 고급언어로 다시 작성하면 다음과 같은 형태의 프로그램이 된다.

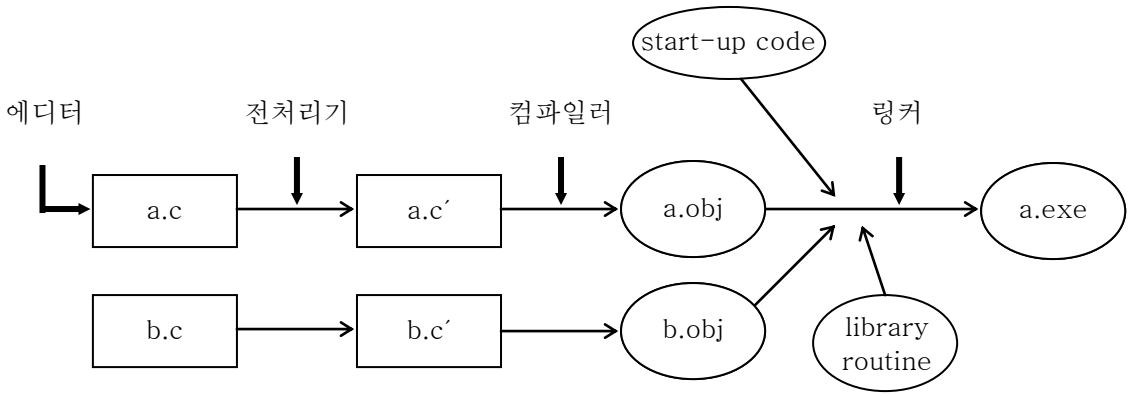
C = A + B

이렇게 고급언어로 작성된 프로그램 역시 실행을 위해서는 컴파일러에 의해 해당 시스템의 어셈블리어로 번역되고 다시 어셈블러에 의해 기계어로 번역되어야 한다. 여기서 한가지 생각해 볼 점은 고급언어에서 A, B, C와 같이 메모리의 주소에 이름을 붙인 것을 변수라고 하는데 결국 변수는 메모리의 특정 기억공간 또는 그 공간의 값을 의미하며 최종적으로는 기계어로 번역되었을 때 주소로 바뀌게 됨을 알 수 있다. 그리고 A, B, C 모두 메모리의 주소를 의미하는 변수이지만 '=' 기호의 어느 쪽에 위치하느냐에 따라서 각각 다른 기계어 코드로 번역됨을 알 수 있다. 즉, '=' 기호의 오른쪽에 있는 변수들은 기계어의 load 명령의 주소로 번역되어 그 기억공간에 저장된 값을 사용하고 '=' 기호의 왼쪽에 있는 변수는 기계어의 store 명령의 주소 값으로 번역되어 데이터가 저장될 위치를 의미한다. 프로그램에서 '='기호의 왼쪽에 있는 것을 L-value(Left value), 오른쪽에 있는 것을 R-value(Right value)라 하는데, 결국 L-value는 값을 저장하는 기억공간을 의미하고 R-value는 저장되는

값을 의미함을 알 수 있다.

4. C 프로그램의 컴파일 과정

C언어는 고급언어이며 따라서 C로 작성된 프로그램은 컴파일러에 의해서 기계어로 번역해야 실행할 수 있다.



프로그램의 작성과정은 메모장이나, 한글, MS 워드와 같은 텍스트 파일의 편집이 가능한 프로그램을 이용하여 약속된 문법으로 작성하고 작성된 프로그램은 일단 아스키코드 형태의 텍스트 파일로 저장한다. 저장할 때 파일 이름은 .c 확장자를 주어 C프로그램 파일임을 표시한다. 이렇게 만든 파일을 소스파일(source file) 이라고 한다. 이 소스파일은 일단 전처리기(preprocessor)가 좀더 번역하기 쉬운 형태의 텍스트 파일로 바꾸고, 그 후에 컴파일러가 CPU가 해석할 수 있는 형태의 기계어로 번역한다. 이 상태의 파일을 개체파일(object file) 이라고 한다. 개체파일은 비록 기계어 형태의 파일이지만 그 자체로는 실행할 수 없고 프로그램을 시작하는 코드인 스타트업(start-up) 코드를 붙여야 비로소 실행파일(execute file)이 된다. 이 과정을 링크(link)라고 하며 링커(linker)가 수행한다. 링커는 프로그램에서 사용한 라이브러리 함수나 프로그래머가 따로 작성하여 미리 컴파일 한 개체파일을 서로 연결하여 하나의 실행파일을 만든다. 보통 컴파일이라면 전처리와 컴파일, 링크 과정을 통틀어 말하지만 컴파일과 링크를 따로 분리하여 이해할 필요가 있다. 큰 프로젝트를 여러 명이 나누어 작성할 경우에 각자 만든 프로그램은 개별적으로 전처리와 컴파일을 거쳐 개체파일을 만들고 이렇게 만든 개체파일을 링크해서 하나의 실행파일을 만든다. 큰 프로그램이 아니더라도 대부분의 프로그램은 입출력과 같은 작업을 하게 되는데 입출력은 운영체제의 지원을 받아야 하고 프로그래밍이 까다로우므로 컴파일러 제조사에서 미리 함수형태로 프로그래밍

하여 개체파일로 제공하는데 이것을 라이브러리 함수(Library function)라고 한다. 라이브러리 함수는 입출력 외에도 자주 쓰거나 유용한 처리과정을 미리 프로그래밍하여 제공한다. 보통 프로그램은 이런 함수를 호출하여 작성하고 라이브러리 함수 자체는 링크과정에서 연결되어 실행파일이 된다.

실행파일은 결국 분할컴파일 된 개체파일이나 라이브러리 함수들이 링크되어 만들어졌다고 할 수 있으나 단순한 기계어 코드와 실행파일을 결정짓는 것은 바로 start-up 코드다. 링크 단계에서 최종적으로 프로그램 선두에 start-up 코드를 붙이므로 실행파일을 만들게 되는데, 이 start-up 코드는 프로그램이 실행될 때 운영체제로부터 제어권을 넘겨 받아서 프로그램의 시작 부분인 메인함수를 호출하는 코드와 프로그램 종료시 메인함수로부터 제어를 넘겨받아 운영체제에 돌려주는 코드 등이 있다. 이 외에도 start-up 코드는 프로그램을 실행할 때 운영체제로부터 프로그램에 전달되는 인수를 받아서 관리하고 메인함수에 넘겨주는 역할을 하며, 프로그램에서 기본적으로 사용되는 운영체제의 환경변수도 운영체제로부터 받아서 유지하는 역할을 수행한다.

5. C 언어의 특징

절차적 프로그래밍(procedural programming)

C언어의 특징을 살펴보기 전에 프로그램을 작성하는 기법을 간단히 살펴보면, 우선 작업의 흐름에 따라 프로그램을 차례대로 써내려 가는 방법이 있을 수 있는데 이것을 절차적 프로그래밍이라고 한다. 이런 방법으로 프로그래밍을 할 경우에는 프로그램을 수행하는 제어의 흐름이 차례로 진행되므로 실행 속도가 빠르고 어떻게 보면 가장 이상적인 프로그래밍이라고 볼 수 있다. 하지만 프로그램 중에는 타이틀을 반복적으로 출력하거나 특정 작업을 필요에 따라 반복 수행해야 할 경우가 많은데, 이 경우에 같은 코드를 중복해서 사용 함으로서 프로그램의 크기가 커지고 반복 횟수에도 한계가 있을 수 있다. 따라서 같은 코드를 한번만 작성하고 재활용 하기 위해서는 필연적으로 제어의 흐름을 바꿔야 하는데 프로그램의 크기가 커질수록 이런 제어의 흐름은 복잡하고 결국 프로그램 자체를 이해하기 어려운 상태로 만들게 된다. 이렇게 제어의 흐름이 복잡한 프로그램은 에러의 수정이나 유지보수할 때에도 수정한 부분과는 상관 없는 다른 부분에서 또 다른 오류가 생기는 문제를 발생시킨다.

구조적 프로그래밍(structured programming)

이런 절차적 프로그래밍 기법의 단점을 보완하고자 나온 것이 구조적 프로그래밍이다. 구조적 프로그래밍은 처리해야 할 작업 전체를 기능별로 나누고 기능별로 프로그래밍 한 후에 필요에 따라 해당 작업을 수행하는 부분을 사용하여 전체적인 프로그램을 완성하는 방법이다. 이렇게 나누어서 구현된 부분을 함수라고 하며 전체 프로그램은 이러한 함수들을 호출하여 작성한다. 함수는 단순히 같은 코드를 반복하는 작업에 사용할 수도 있지만, 대부분의 경우는 특별한 입력을 주고 처리된 결과를 돌려 받는 방식으로 함수를 사용한다. 프로그램의 작성기법은 프로그래밍 언어를 개발할 때 반영하는데, C언어는 바로 구조적 프로그래밍 기법을 효율적으로 구현할 수 있도록 문법체계가 만들어졌다. 따라서 C프로그램은 메인 함수를 비롯한 함수들의 집합으로 이루어진다.

시스템 프로그래밍(system programming)

C언어는 유닉스 운영체제를 개발하기 위해 만들어졌다. 운영체제는 컴퓨터를 구성하는 CPU, 메모리, 주변장치 등의 시스템 자원을 관리하고 프로그램을 실행시키는 등의 기능을 하는데, 이러한 기능들을 구현하기 위해서 C언어는 메모리를 직접 관리할 수 있도록 포인터를 사용하고, 비트 단위의 연산을 수행할 수 있는 비트연산자가 있는 등 운영체제 같은 시스템 소프트웨어를 쉽게 작성할 수 있도록 만들어 졌다. 실제로 많은 운영체제가 C로 작성되었고 이러한 운영체제가 가지고 있는 시스템 자원을 관리할 수 있는 기능들은 C의 함수 형태로 제공되어 프로그래머가 하드웨어를 컨트롤 할 수 있는 강력한 프로그래밍을 가능하게 한다. 이렇게 운영체제가 가지고 있는 기능을 호출하여 사용하는 것을 일반 함수 호출(function call)과 구분하여 시스템 호출(system call)이라고 한다.

시스템 라이브러리(system library)

C언어가 가지고 있는 비교적 간단한 문법과, 많은 연산자, 포인터 등의 개념을 가지고는 하드웨어를 컨트롤 해서 입출력을 한다든지, 또는 문자열을 처리하는 간단한 기능의 프로그램을 작성할지라도 실제적으로 구현할 때는 어려운 프로그래밍이 될 수 있다. 예를 들어 입출력의 경우에는 최종적으로는 운영체제가 제공하는 시스템 함수를 호출하여 입출력을 수행할 수 밖에 없는데 운영체제가 제공하는 시스템 함수는 기능은 제한적이고 원시적이므로 프로그래머가 원하는 편리하고 다양한 형태의 입출력을 수행하는 일은 쉬운 일이 아니다. 입출력 뿐만이 아니라 두 문자열을 붙이는 간단한 처리도 두 문자열의 주소를 이용해서 한 문자씩 일일이 복사해서 붙이는 방법으로 프로그래밍을 해야 한다. 따라서 복잡하고 큰 프로그램을 C가 제공하는 문법과 연산자 만으로 프로그래밍 한다는 것은 어려운 일이다. 그러므로

C 컴파일러 제조사는 입출력을 비롯해서 프로그램을 작성하는데 필요한 기본적인 기능들을 미리 함수로 구현하여 개체파일의 형태로 같이 제공하고 있는데 이것을 시스템 라이브러리 함수라고 한다. C언어는 이러한 라이브러리 함수가 풍부하게 제공 되므로 프로그래머는 그 기능을 익혀서 호출 함으로서 편리하고 빠르게 프로그램을 작성할 수 있다.