2020.08.01
수업 복습
Function

임베디드스쿨1기
Lv1과정
2020. 08. 01
권 영근

함수

서브루틴, 메서도, 프로시저와 비슷한 개념을 지닌 코드 집합체로서 소프트웨어에서 특정한 목적의 작업을 수행하는 역활을 가지고 있습니다.

하나의 큰 프로그램을 여러 부분으로 분리함으로서 모듈화로 인하여 소스 코드의 가독성이 좋아집니다. 그리고 같은 소스 코드를 계속 사용하지 않아 프로그램의 용량을 줄일 수 있고 다른 프로그램에서 그 소스 코드를 또한 사용할 수 있습니다. 이러한 것들이 함수의 장점입니다.

함수의 구조를 간략하게 나타내 보겠습니다. 그리고 구조의 각 부분들을 간략하게 설명을 해보겠습니다.

```
int Sub(int a, int b)
{
   return (a – b);
}
```

첫 번째, 반환자료형(return type)입니다. 위의 함수 ‘Sub’ 의 자료형은 ‘int’ 입니다. 이는 Sub 함수가 반환하는 값의 자료형이 역시 ‘int’ 라는 것을 의미합니다.

다음의 소스 코드 부분은 잘못된 함수 프로그래밍의 예시입니다.

```
double Sub(double a, double b)
{
    unsigned char c;
    double diff;
    diff = a – b;

    return c;
}
```

함수의 자료형과 반환 자료형이 다르기 때문에 위의 함수 소스 코드는 잘못된 소스 코드입니다.

두 번째, 함수의 이름입니다. 함수 호출을 위한 이름을 나타냅니다.

세 번째, 매개 변수입니다. 매개 변수를 좀 더 자세히 설명을 하기 위해서 다음의 프로그래밍 소스 코드를 작성하겠습니다.

```
#include <stdio.h>

unsigned int MUL(unsigned int x, unsigned int y);
```

```
int main(void)
{
    unsigned int a = 2, b = 5, c;
    c = MUL(a, b);

    printf("%u\n", c);

    return 0;
}

unsigned int MUL(unsigned int x, unsigned int y)
{
    return x * y;
}
```

위의 소스 코드에서 헤더 파일 바로 아래 및 main 함수 바로 이전에 있는 부분을 어느 함수의 'prototype' 이라고 합니다. 이 prototype의 소괄호 안에 2개의 변수들, 즉 x, y 들이 있습니다. 이들을 그 함수의 '매개변수(parameter)' 라고 부릅니다. 이 매개변수들은 이들을 포함하고 있는 함수가 호출될 시, 전달될 값을 저장하는 변수들을 가리킵니다.

참고로 호출되는 함수에 전달되는 값을 ‘인자(argument)’ 라고 합니다. 이전 소스 코드의 ‘main’ 함수 내에 ‘c = MUL(a, b);’ 부분이 있습니다. 여기에서 a, b에 들어가는 값을 인자라고 할 수 있습니다.

네 번째, 함수 몸체입니다. 아래의 소스 코드에서 중괄호로 둘러싸인 부분이 함수의 몸체입니다.

```
void PrintHalloWelt(void)
{
   printf(“Hallo Welt!!!\n”);
}
```

앞에서도 언급하였지만 다시 한번 언급하고 싶은 것이 있습니다. 매개 변수에는 ‘값’ 만 들어갑니다. 이 점을 염두에 둬야 합니다. 이에 대해 예를 들면서 좀 더 자세히 설명을 해보고자 합니다.

정수형 변수 a와 b가 있다고 가정을 해보겠습니다. a에는 정수 ‘2’ 가 들어있고 b에는 정수 ‘3’ 이 들어있습니다. 이와 관련된 소스 코드를 간추려서 작성해보겠습니다.

```
#include <stdio.h>

int main(void)
{
   int a = 2, b = 3;

   return 0;
}
```

a에 들어있는 값과 b에 들어있는 값을 바꿔보겠습니다. a에 3이 들어있게 되고 b에 2가 들어있게 되는 것입니다. 이를 위해 함수 ‘swap’ 을 만들었습니다. 반환형이 ‘void’ 이고 매개변수는 2개의 정수형 변수들입니다.

```
void swap(int x, int y)
{
   int temp;

   temp = x;
   x = y;
   y = temp;
```

```
}
```

main 함수 안에서 소스 코드 'swap(a, b);' 다음 줄에 'printf' 함수를 사용하여 a, b 값들을 출력하는 소스 코드를 작성합니다. 그런 후 컴파일을 하고 실행을 해보겠습니다. 그런데 결과는 예상했던 것과는 다릅니다. 'a = 3, b = 2' 라는 결과가 나타나지 않습니다. 여전히 'a = 2, b = 3' 입니다. 변화가 없습니다.

왜 이러한 일이 발생하는 것일까요? swap 함수가 a, b 변수들 안에 있는 값들만을 받아들였기 때문입니다. 좀 더 자세히 말하기 위해서 swap 함수 안에서 일어나는 일들을 이야기하겠습니다. 또한 '스택' 에 대해서도 언급을 해야 할 것 같습니다.

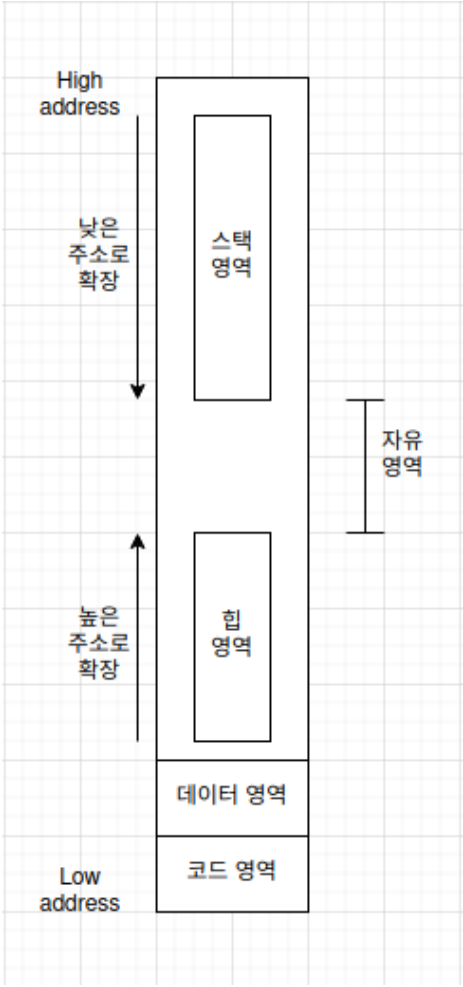
스택이란 먼저 받아들인 데이터가 나중에 나갈 수 있고 마지막에 들어온 데이터가 제일 먼저 나갈 수 있는 자료형입니다. 즉, 마지막에 들어온 데이터가 처음 들어온 데이터보다 늦게 나갈 수 없습니다. 반대로 처음 들어온 데이터가 마지막에 들어온 데이터보다 먼저 나갈 수 없습니다. 이를 영어 표현으로 "Last in, first out." 이라고 하는데, 줄여서 'LIFO' 라고도 합니다.

스택에 대한 이해를 좀더 쉽게 하기 위해 한가지 예를 들어보겠습니다. 다음의 사진과 같이 쌓여져 있는 접시들을 가정해보겠습니다.

사진에는 나타나 있지 않았지만 저 수많은 접시들이 어떤 좁은 상자 안에 있다고 또 한번 가정을 해주실 것을 부탁드립니다. 맨 아래의 접시를 안전하게 꺼내기 위해서 어떻게 하면 좋을까요? 위의 접시들을 차례로 하나씩 꺼내야 맨 아래의 접시를 꺼낼 수 있을 것입니다. 비록 지루하겠지만 좁은 상자 안에 접시들이 차례대로 빽빽이 놓여있기 때문입니다.

대신 마지막에 놓여진 접시는 제일 먼저 나갈 수 있을 것입니다. 스택의 데이터 저장 및 데이터를 빼는 방식을 이와 같은 방식으로 이해를 하시면 될 것입니다.

아래를 보면 스택이 메모리에서 차지하는 그림이 있습니다.

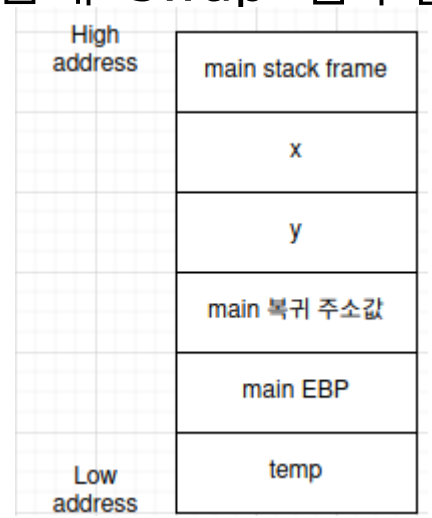


스택은 메모리의 높은 주소에서 낮은 주소로 확장을 해나가면서 메모리를 사용합니다.

어느 스택의 꼭대기 위치가 100번지라고 할 때, 그 스택에 새로운 정수형 데이터를 저장을 합니다. 정수형 데이터의 크기는 4 바이트이므로 스택의 새로운 꼭대기는 96 번지를 가리킵니다. 스택에 데이터를 저장하는 것을 'push' 라고 합니다.

반대로 이번에는 스택에서 데이터를 꺼내겠습니다. 96번지에 저장되어 있는 정수형 변수가 스택에서 사라집니다. 그렇게 되면 스택의 꼭대기는 다시 100번지를 가리키게 됩니다. 스택에서 데이터를 꺼내는 것을 'pop' 라고 합니다.

일단 main 함수에서 프로그램이 진행될 것입니다. main 함수 내의 지역 변수 a, b들을 메모리에 저장하고 나아가다가 마침내 'swap' 함수를 마주치게 될 것입니다.



참고로 'BP' 레지스터는 어느 함수의 지역 변수 저장을 위해 사용됩니다.

swap 함수 호출이 된 후, 그 함수의 파라미터에는 변수 a, b들 안에 있는 값들만이 전달이 됩니다. a, b 그 자체들이 전달되는 것이 아닙니다. swap 함수는 전달받은 값들을 temp 변수를 이용하여 바꿀 것입니다. 그리고 복귀 주소를 이용하여 main의 프로시져로 복귀합니다. 하지만 값들만을 가지고 swap함수가 작업을 한 것이라서 main 함수 내의 a, b 변수들의 값들이 서로 바뀌지게 만들지는 못합니다. swap 함수가 복귀 후 사라지면서 바뀌졌던 값들을 담은 변수들도 사라졌기 때문입니다.

즉, 이런 식의 swap 함수로는 main 함수 내의 변수를 바꿀 수 없습니다. 그 대신 값 대신에 변수의 주소값을 전달함으로서 이러한 문제를 해결할 수 있습니다. swap 함수를 고쳐보겠습니다.

```
void swap(int* x, int* y)
{
    int temp;
    temp = *x;
    *x = *y;
    *y = temp;
}
```

그리고 원래 main 함수 내에 ‘swap(a, b);’ 소스 코드 부분이 있는데, 이를 ‘swap(&a, &b)’ 라고 변경을 합니다. 그리고 나서 실행을 하면 a의 값이 3이 되고 b의 값이 2가 되어 원하던 목표가 이루어졌음을 보게 될 것입니다.

swap 함수의 매개변수에 a, b 변수의 주소 값들을 인자로서 넘겨줍니다. 변경된 swap 함수는 주소 값들이 가리키는 메모리 공간의 값들을 temp 변수를 사용하여 변경합니다. swap 함수가 할 일을 마치고 복귀 주소를 사용하여 main 함수로 돌아갈 때, swap 함수 내의 지역 변수와 인자들은 사라지게 됩니다. 하지만 주소 값들을 이용하여 바꿔치기한 값들은 사라지지 않고 남아있게 됩니다. 그래서 a의 값은 2에서 3으로, b의 값은 3에서 2로 바뀌질 수 있는 것입니다.

※ 일부 사람들 혹은 책에서 매개 변수에 전달하는 인수가 정수 값들인 경우 ‘call by value’ 라는 개념으로 설명하고 인수가 주소값일 경우 ‘call by reference’ 라는 개념으로 설명하는 경우가 있습니다. 그런데 이 개념들은 C Programming에는 없는 개념들입니다. 따라서 C 언어를 가르치는 과정에서 주소값 매개변수나 함수에 입력되는 인수를 call by reference 등의 용어로 설명하는 일을 해서는 안됩니다.

```c
#include <stdio.h>

int max(int num1, int num2);

int main(void)
{
    int a = 100;
    int b = 200;
    int ret;

    ret = max(a, b);

    printf("Max value is %d\n", ret);

    return 0;
}

int max(int num1, int num2)
{
    int result;

    if(num1 > num2)
    {
        result = num1;
    }
    else
    {
        result = num2;
    }

    return result;
}
```

```
bolthakziy@bolthakziy-340XAA-350XAA-550XAA:~/C$ gcc -o main test.c
bolthakziy@bolthakziy-340XAA-350XAA-550XAA:~/C$ ./main
Max value is 200
bolthakziy@bolthakziy-340XAA-350XAA-550XAA:~/C$ vim test.c
bolthakziy@bolthakziy-340XAA-350XAA-550XAA:~/C$
```

절차 지향 프로그래밍 언어 C

다음의 소스 코드를 한번 가정해보겠습니다.

```
#include <stdio.h>

void embeddedSchool();

int main()
{
   ...
}

Void embeddedSchool()
{
   ...
}
```

위의 소스 코드를 컴파일한 후 실행을 하면 문제가 없을 것입니다. 물론 이러한 글의 전제는 main 함수와 embeddedSchool 함수에 오류가 없다는 것입니다.

그런데 위의 소스 코드에 변화가 생겼습니다.

```
#include <stdio.h>

//void embeddedSchool();

int main()
{
...
    embeddedSchool();
}

void embeddedSchool()
{
...
}
```

embeddedSchool 함수의 prototype을 주석처리함으로서 제거하였습니다. 그런 후 컴파일을 해봤습니다.

그 결과, 대개 오류가 나타날 것입니다. 절차 지향적으로 설계된 C 프로그래밍 소스 코드는 한줄 한줄 순서대로 소스 코드를 처리하면서 진행을 합니다. 그런데 main 함수 내의

embeddedSchool 함수를 마주친다면, 컴파일러는 이 함수를 인식할 수 없습니다. 왜냐하면 embeddedSchool 함수를 마주치기 전에 embeddedSchool과 관련된 어떤 정보를 얻지 못했기 때문입니다. prototype도 embeddedSchool 함수 소스 코드도 없습니다. 그래서 컴파일이 실패할 확률이 높습니다.

그런데 최근의 컴파일러는 이러한 프로그래밍 소스 코드를 큰 문제없이 컴파일을 성공시킬 것입니다. 물론 경고 문서가 나타날 확률이 높겠지만 말입니다. 자세한 이유는 모르겠지만 좋은 성능을 가진 컴파일러가 만들어지거나 공학자나 기술자가 기존의 컴파일러를 업그레이드를 했을 것이라고 짐작을 하고 있습니다.

지역 변수와 전역 변수

⊙ 지역 변수

어떤 하나의 함수 안에서만 사용되는 변수입니다. 지역 변수는 특정 함수 내에서만 사용이 되기 때문에 함수가 사용될 때만 메모리의 스택 영역에 존재하면 됩니다. 지역 변수는 그 지역 변수를 가지고 있는 함수가 실행되는 동안 인수, 복귀 주소와 함께 스택에 저장되어 있다가 함수가 종료할 때 스택에서 사라지게 됩니다.

⊙ 전역 변수

여러 개의 함수들이 공통으로 사용하는 변수입니다. 메모리의 데이터 영역에 놓이게 되며 초기값 설정이 된 전역 변수는 data segment에 그리고 초기값 설정이 안된 전역 변수는 bss segment에 위치하게 됩니다. 어떤 함수의 시작이나 종료에 상관없이 프로그램이 실행되는 동안 메모리에 항상 존재합니다.

지역 변수처럼 특정 함수 내에만 존재를 하면 지역적으로 사용될 뿐, 여러 군데에서 사용할 수 없기 때문에 전역 변수가 사용이 됩니다.

그리고 어떤 제품의 성능을 검사하는 Testing 용도로서 잘 사용이 됩니다.

그런데 이 전역 변수를 보안에 민감한 영역에 사용될 시, 많은 사람들에게 큰 피해를 끼칠 수 있습니다. 예를 들어서 은행 계좌에서 돈과 관련된 데이터를 저장하는 어떤 변수를 전역 변수로 설정한다면 어느 누구라도 손쉽게 그 변수에 접근할 가능성이 높아집니다. 그 누구가 도둑이나 악랄한 해커라면 많은 사람들이 재산을 잃어버리고 은행은 신뢰성을 잃어버려서 도퇴될 것입니다.

그렇기 때문에 보안에 덜 민감한 영역에서 전역 변수를 사용해야 합니다. 전역 변수를 사용하는 일을 매우 신중하게 고려해야 합니다.

2020.08.01
수업 복습
Array

임베디드스쿨1기
Lv1과정
2020. 08. 01
권 영근

배열

어떤 한가지 자료형을 연속적으로 나열하는 자료구조입니다. 어느 한 배열 안에는 같은 종류의 데이터들이 순차적으로 저장이 되어 배열의 한 원소의 번호는 배열의 시작점 주소로부터 값이 저장되어 있는 상대적인 위치를 의미합니다. 배열의 첫 번째 원소의 메모리 주소를 기본 주소라고 하기도 합니다.

예를 들어서 100명의 이름을 저장할 때, 100개의 변수를 선언해서 사용하는 대신, 원소가 100개인 배열 1개를 사용하면 쉽게 100명의 이름을 저장할 수 있습니다.

또한 복잡한 자료 구조를 표현할 때나 행렬, 벡터 등을 컴퓨터에서 표현하기 위해서 사용되기도 합니다.

배열을 선언할 때, 먼저 '자료형' 을 적고 뛰어써서 변수명을 작성하고 안에 원소의 개수가 적힌 대괄호를 작성하면 됩니다.

예를 들어서 원소의 개수가 10개인 8바이트 실수형 배열을 선언해보겠습니다.

```
double balance[10];
```

다음에는 원소의 개수가 7개인 정수형 배열을 선언해보겠습니다.

int Number[7];

대괄호 개수가 1개인 배열은 1차원 배열이라고 합니다. 1차원 배열을 초기화함에 있어서 규칙이 있습니다. 만일 대괄호 안에 원소의 개수를 의미하는 숫자가 없으면 컴파일러는 오류를 알리게 됩니다.

double balance[] = {4.5, 3.1, 2.7}; (X)

중괄호 안에 숫자를 적어야 오류가 나타나지 않습니다.

double balance[3] = {4.5, 3.1, 2.7}; (O)

만일 중괄호 안의 숫자보다 더 많은 개수의 원소들이 배열 안에 있으면 오류가 발생합니다.

double balance[3] = {4.5, 3.1, 2.7, 1.2, 3.5};   (X)

중괄호 안의 숫자보다 더 적은 원소들이 배열 안에 있을 때는 오류가 발생하지 않습니다. 배열 안의 빈 공간은 0으로 채워질 것입니다.

double balance[5] = {4.5, 3.1, 1.2};   (O)

=> double balance[5] = {4.5, 3.1, 1.2, 0.0, 0.0}; (O)

다차원 배열

배열의 원소의 개수를 나타내기 위해 대괄호 안에 숫자를 작성하였는데 이 숫자는 전체 배열의 주소를 알려주는 역활도 하기 때문에 'index' 라고도 합니다. 이러한 인덱스를 포함하는 대괄호의 개수가 2개 이상인 배열을 다차원 배열이라고 부릅니다.

선언하는 것 역시 1차원 배열과 비슷합니다. 먼저 자료형을 쓰고 뛰어써서 변수명과 원소의 개수가 적힌 대괄호를 적으면 됩니다.

(ex) int a[3][4];

다차원 배열들 중 2차원 배열을 초기화를 할 때, 역시 주의해야 할 점이 있습니다. 다음과 같이 선언하는 것은 괜찮습니다.

int a[3][4] = {{0, 1, 2, 3}, {4, 5, 6, 7}, {8, 9, 10, 11}}; (O)

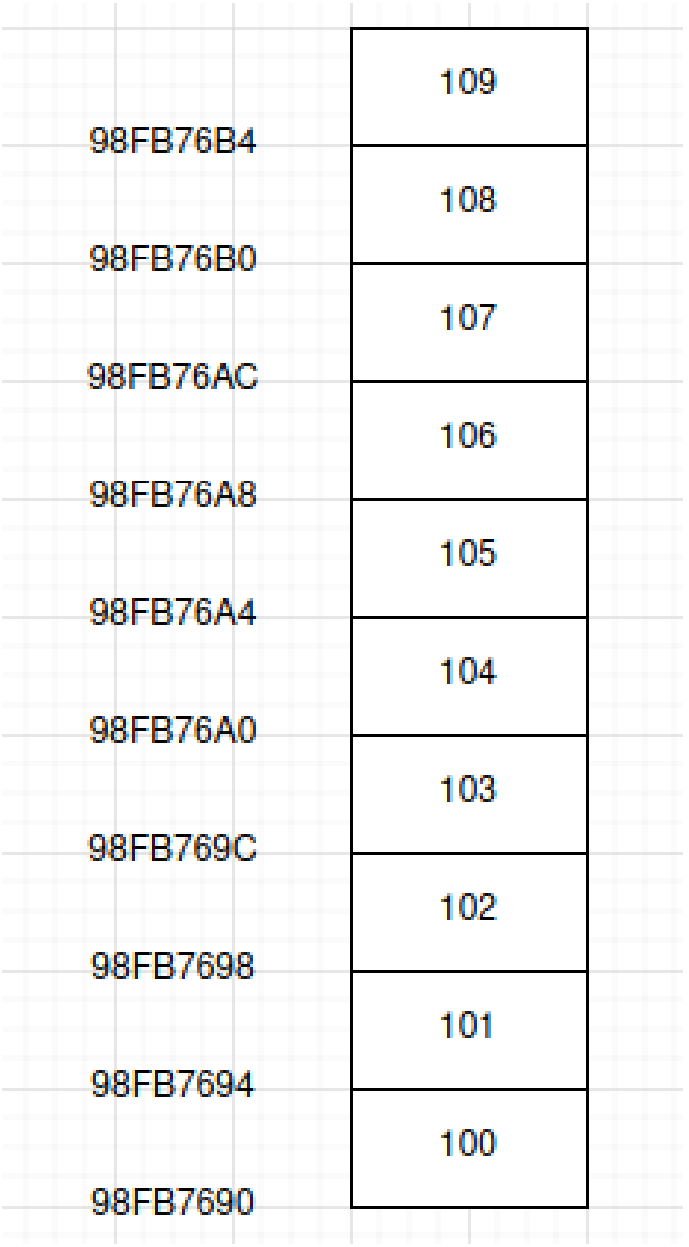
아래처럼 첫 번째 중괄호 안에 숫자가 없어도 대입되는 원소들이 있다면 괜찮습니다. 그런데 대입되는 원소가 없으면 오류가 발생합니다.

int a[][4] = {{0, 1, 2, 3}, {4, 5, 6, 7}, {8, 9, 10, 11}}; (O)
int a[][4]; (X)

```c
#include <stdio.h>

int main(void)
{
    int n[10];
    int i, j;

    for(i = 0; i < 10; i++)
    {
        n[i] = i + 100;
    }

    for(j = 0; j < 10; j++)
    {
        printf("Element[%d] = %d\n", j, n[j]);
    }

    return 0;
}
```

```
Element[0] = 100
Element[1] = 101
Element[2] = 102
Element[3] = 103
Element[4] = 104
Element[5] = 105
Element[6] = 106
Element[7] = 107
Element[8] = 108
Element[9] = 109
```

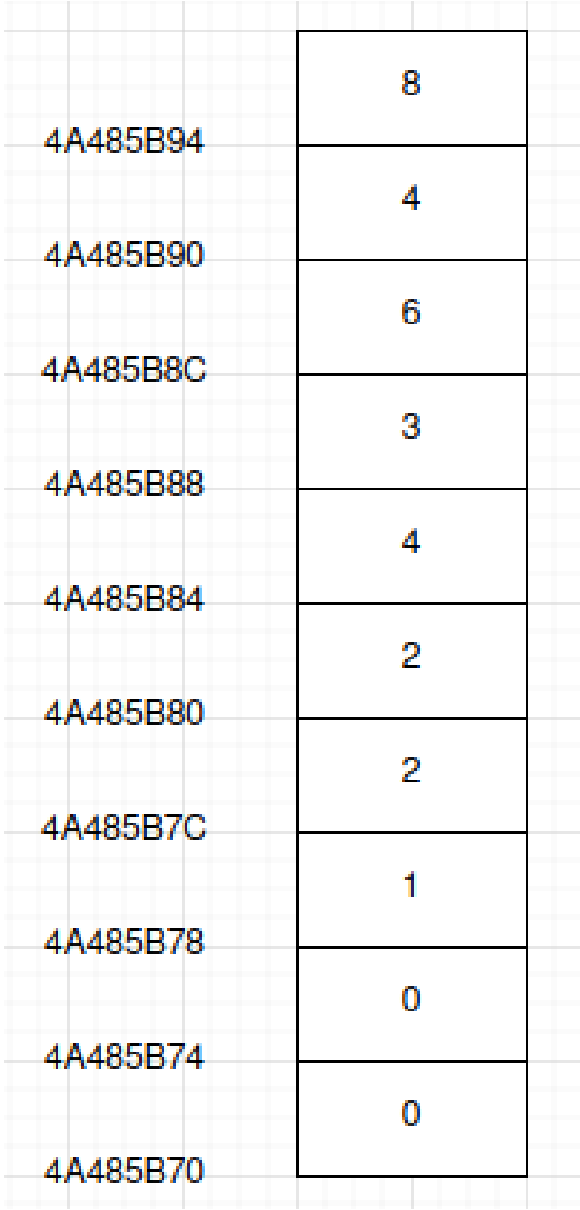
```
Address[0] = 98fb7690
Address[1] = 98fb7694
Address[2] = 98fb7698
Address[3] = 98fb769c
Address[4] = 98fb76a0
Address[5] = 98fb76a4
Address[6] = 98fb76a8
Address[7] = 98fb76ac
Address[8] = 98fb76b0
Address[9] = 98fb76b4
```



```c
#include <stdio.h>

int main(void)
{
    int a[5][2] = {{0, 0}, {1, 2}, {2, 4}, {3, 6}, {4, 8}};
    int i, j;

    for(i = 0; i < 5; i++)
    {
        for(j = 0; j < 2; j++)
        {
            printf("a[%d][%d] = %d\n", i, j, a[i][j]);
        }
    }

    return 0;
}
```

```
a[0][0] = 0
a[0][1] = 0
a[1][0] = 1
a[1][1] = 2
a[2][0] = 2
a[2][1] = 4
a[3][0] = 3
a[3][1] = 6
a[4][0] = 4
a[4][1] = 8
```

```
adress[0][0] = 4a485b70
adress[0][1] = 4a485b74
adress[1][0] = 4a485b78
adress[1][1] = 4a485b7c
adress[2][0] = 4a485b80
adress[2][1] = 4a485b84
adress[3][0] = 4a485b88
adress[3][1] = 4a485b8c
adress[4][0] = 4a485b90
adress[4][1] = 4a485b94
```



```c
#include <stdio.h>

double getAverage(int arr[], int size);

int main(void)
{
    int balance[5] = {1000, 2, 3, 17, 50};
    double avg;
    avg = getAverage(balance, 5);

    printf("Average value is %f\n", avg);

    return 0;
}

double getAverage(int arr[], int size)
{
    int i;
    double avg, sum = 0;

    for(i = 0; i < size; ++i)
    {
        sum += arr[i];
    }

    avg = sum / size;

    return avg;
}
```

```
bolthakziy@bolthakziy-340XAA-350XAA-55
bolthakziy@bolthakziy-340XAA-350XAA-55
Average value is 214.400000
bolthakziy@bolthakziy-340XAA-350XAA-55
```

2020.08.01
HW

임베디드스쿨1기

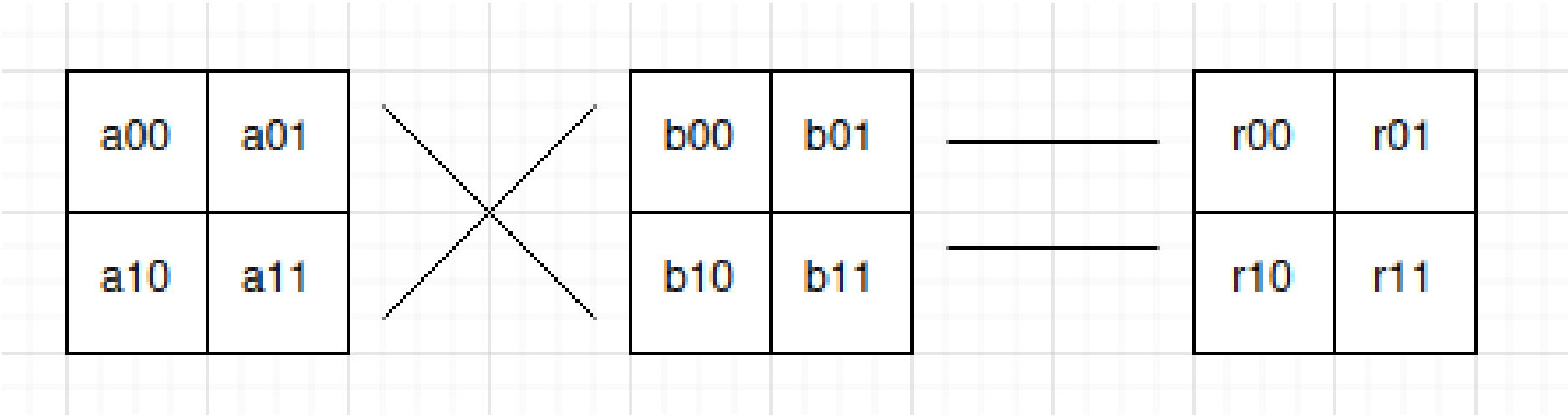
Lv1과정

2020. 08. 01

권 영근

```c
#include <stdio.h>

int main(void)
{
    char str1[] = "angel", str2[] = "devil";
    char temp;
    int i;

    printf("Before operation : str1 = %s, str2 = %s\n", str1, str2);

    for(i = 0; str1[i] != '\0'; i++)
    {
        temp = str1[i];
        str1[i] = str2[i];
        str2[i] = temp;
    }

    printf("After operation : str1 = %s, str2 = %s\n", str1, str2);

    return 0;
}
```

```
bolthakziy@bolthakziy-340XAA-350XAA-550XAA:~/C$ ./main
Before operation : str1 = angel, str2 = devil
After operation : str1 = devil, str2 = angel
bolthakziy@bolthakziy-340XAA-350XAA-550XAA:~/C$ vim test.
```


a00
a01
a10
a11
b00
b01
b10
b11
r00
r01
r10
r11

단위 행렬을 곱한 결과를 출력해봤습니다.

```c
#include <stdio.h>

int main(void)
{
    int a[2][2], b[2][2], r[2][2];
    int i, j;

    for(i = 0; i < 2; i++)
    {
        for(j = 0; j < 2; j++)
        {
            scanf("%d", &a[i][j]);
        }
    }

    for(i = 0; i < 2; i++)
    {
        for(j = 0; j < 2; j++)
        {
            scanf("%d", &b[i][j]);
        }
    }

    r[0][0] = a[0][0] * b[0][0] + a[0][1] * b[1][0];
    r[0][1] = a[0][0] * b[0][1] + a[0][1] * b[1][1];
    r[1][0] = a[1][0] * b[0][0] + a[1][1] * b[1][0];
    r[1][1] = a[1][0] * b[0][1] + a[1][1] * b[1][1];

    for(i = 0; i < 2; i++)
    {
        for(j = 0; j < 2; j++)
        {
            printf("r[%d][%d] = %d\n", i, j, r[i][j]);
        }
    }

    return 0;
}
```

```
4
5
8
9
1
0
0
1
r[0][0] = 4
r[0][1] = 5
r[1][0] = 8
r[1][1] = 9
```

```c
#include <stdio.h>

int main(void)
{
    int arr[3][3] = {{1, 15, 4}, {8, 10, 16}, {2, 7, 20}};
    int max, i, j;
    max = arr[0][0];

    for(i = 0; i < 3; i++)
    {
        for(j = 0; j < 3; j++)
        {
            if(max >= arr[i][j])
            {
                continue;
            }
            else
            {
                max = arr[i][j];
            }
        }
    }

    printf("The biggest number in the array is %d!!!\n", max);

    return 0;
}
```

```
bolthakziy@bolthakziy-340XAA-350XAA-550XAA:~/C$ vim te
bolthakziy@bolthakziy-340XAA-350XAA-550XAA:~/C$ gcc -o
bolthakziy@bolthakziy-340XAA-350XAA-550XAA:~/C$ ./main
The biggest number in the array is 20!!!
bolthakziy@bolthakziy-340XAA-350XAA-550XAA:~/C$ vim te
bolthakziy@bolthakziy-340XAA-350XAA-550XAA:~/C$
```

4

```c
#include <stdio.h>

int main(void)
{
    int balance[] = {1000, 2, 3, 17, 50};
    int i, j, temp;

    for(i = 0; i < 4; i++)
    {
        for(j = 0; j < 4 - i; j++)
        {
            if(balance[j] <= balance[j + 1])
            {
                continue;
            }
            else
            {
                temp = balance[j];
                balance[j] = balance[j + 1];
                balance[j + 1] = temp;
            }
        }
    }

    for(i = 0; i < 5; i++)
    {
        printf("%4d\t", balance[i]);
    }
    printf("\n");

    return 0;
}
```

```
bolthakziy@bolthakziy-340XAA-350XAA-550XAA:~/C$ vi
bolthakziy@bolthakziy-340XAA-350XAA-550XAA:~/C$ gc
bolthakziy@bolthakziy-340XAA-350XAA-550XAA:~/C$ ./
   2       3       17      50      1000
bolthakziy@bolthakziy-340XAA-350XAA-550XAA:~/C$ vi
bolthakziy@bolthakziy-340XAA-350XAA-550XAA:~/C$
```

2020.08.01
수업 복습
Pointer

임베디드스쿨1기
Lv1과정
2020. 08. 01
권 영근

포인터

포인터는 그냥 간단하게 생각하면 됩니다. 메모리의 '주소값' 을 담는 변수입니다.

다만 프로그래밍을 함에 있어서 약간의 혼란스러운 부분때문에 적지않은 사람들이 어려움을 겪는 것 같습니다.

먼저 정수형 포인터 변수를 선언을 해보겠습니다. 정수형 변수도 1개 선언하겠습니다.

```
int* ip;
int x;
```

정수형 자료형을 먼저 쓰고 그 다음에 '*(Asterisk)' 를 작성한 후, 변수명을 작성하면 됩니다.

그런데 다음 소스 코드 부분이 일부 프로그래머들에게 혼란을 가져다 줍니다.

```
*ip = 1024;
&x = ip;
```

위에서 포인터 변수를 선언하였는데 그 다음 줄에서 또 포인터 변수를 중복하여 선언하는

작업을 하는 것같이 몇몇 프로그래머들에게 느껴집니다.

하지만 원리만 알면 어렵지 않습니다. Asterisk가 자료형과 밀접하게 있을 시, 그 때 Asterisk의 오른쪽에 있는 변수명은 포인터 변수를 의미하게 됩니다.

반대로 Asterisk 왼쪽에 자료형이 없고 오른쪽에만 변수명이 있을 경우, 그 변수는 포인터 변수이며 더 나아가서 왼쪽에 Asterisk가 밀접하게 있기 때문에 그 포인터 변수에 담겨진 주소를 가진 메모리 안에 있는 값을 의미하게 되는 것입니다.

그리고 변수명 왼쪽에 '&' 가 있게 되면 그것은 그 변수의 주소값을 담겠다는 것을 의미하며 &(변수 이름) 안에는 주소값이 들어가야 합니다. 당연히 일반 자료형 안의 값을 넣기보다는 포인터 변수에 담겨진 값을 대입하는데에 많이 사용이 됩니다.

이제 이전의 소스 코드들을 분석해보겠습니다.

```
int* ip;
int x;

*ip = 1024;
&x = ip;
```

ip는 포인터 변수로서 주소값을 담습니다. 그리고 x는 정수형 변수로서 정수를 답게 됩니다. ip안의 주소값이 가리키는 메모리 안에 '1024' 라는 정수 값이 들어가 있습니다. 그리고 정수형 변수 x의 주소는 ip에 담겨진 주소값이라고 합니다.

그렇다면 x의 안에는 어떤 정수 값이 들어있을까요? 당연히 1024가 들어있습니다.

```c
#include <stdio.h>

const int MAX = 3;

int main(void)
{
    int var[] = {10, 100, 200};
    int i, *ptr;
    ptr = var;

    for(i = 0; i < MAX; i++)
    {
        printf("Address of var[%d] = %x\n", i, ptr);
        printf("Value of var[%d] = %d\n", i, *ptr);
        printf("\n");

        ptr++;
    }

    return 0;
}
```

```
Address of var[0] = 9ddb3b3c
Value of var[0] = 10

Address of var[1] = 9ddb3b40
Value of var[1] = 100

Address of var[2] = 9ddb3b44
Value of var[2] = 200
```

```c
#include <stdio.h>

int main(void)
{
    double balance[5] = {1000.0, 2.0, 3.4, 17.0, 50.0};
    double* p;
    int i;

    p = balance;

    printf("Array values using pointer\n");

    for(i = 0; i < 5; i++)
    {
        printf("*(p + %d) : %f\n", i, *(p + i));
    }

    printf("Array values using balance as address\n");

    for(i = 0; i < 5; i++)
    {
        printf("*(balance + %d) : %f\n", i, *(balance + i));
    }

    return 0;
}
```

```
Array values using pointer
*(p + 0) : 1000.000000
*(p + 1) : 2.000000
*(p + 2) : 3.400000
*(p + 3) : 17.000000
*(p + 4) : 50.000000
Array values using balance as address
*(balance + 0) : 1000.000000
*(balance + 1) : 2.000000
*(balance + 2) : 3.400000
*(balance + 3) : 17.000000
*(balance + 4) : 50.000000
```

배열의 각 원소들의 주소들을 포인터 변수를 이용하여 표현될 수 있음을 보게 됩니다.

```c
#include <stdio.h>
#include <time.h>

int* getRandom(void)
{
    static int r[10];
    int i;

    srand((unsigned)time(NULL));

    for(i = 0; i < 10; ++i)
    {
        r[i] = rand();
        printf("%d\n", r[i]);
    }

    return r;
}

int main(void)
{
    int* p;
    int i;
    p = getRandom();

    for(i = 0; i < 10; i++)
    {
        printf("*(p + [%d]) : %d\n", i, *(p + i));
    }

    return 0;
}
```

```
30098674
1322016719
186638512
1725270126
713713022
123709050
1907351177
1023173140
1450088221
132560659
*(p + [0]) : 30098674
*(p + [1]) : 1322016719
*(p + [2]) : 186638512
*(p + [3]) : 1725270126
*(p + [4]) : 713713022
*(p + [5]) : 123709050
*(p + [6]) : 1907351177
*(p + [7]) : 1023173140
*(p + [8]) : 1450088221
*(p + [9]) : 132560659
```

여기에서도 각 배열의 원소들 주소가 포인터 변수를 활용하여 표현될 수 있음을 봅니다.

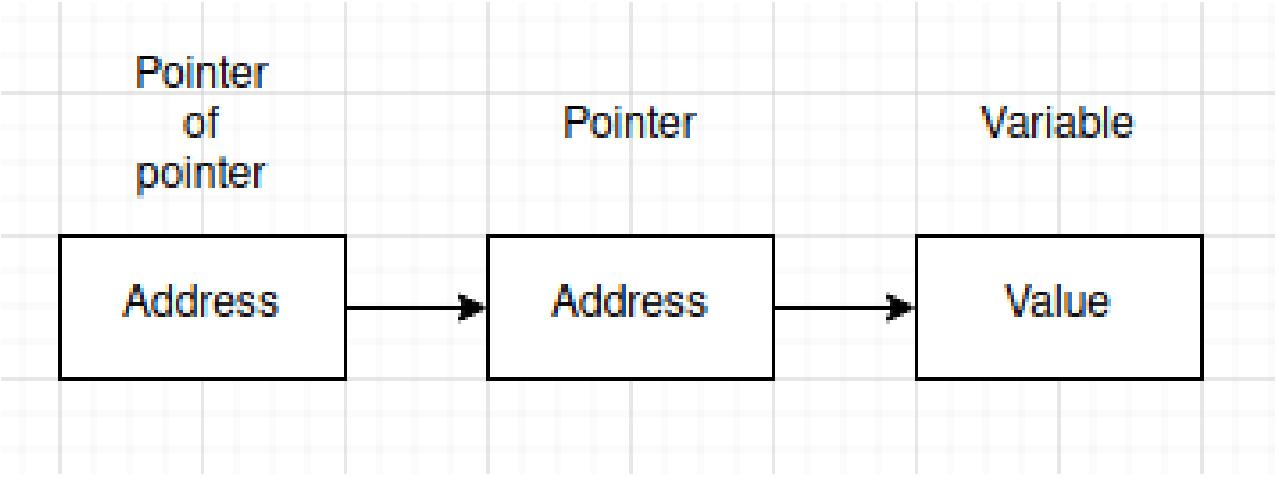
```c
#include <stdio.h>

double getAverage(int* arr, int size);

int main(void)
{
    int balance[5] = {1000, 2, 3, 17, 50};
    double avg;
    avg = getAverage(balance, 5);

    printf("Average value is %f\n", avg);

    return 0;
}

double getAverage(int* arr, int size)
{
    int i, sum = 0;
    double avg;

    for(i = 0; i < size; ++i)
    {
        sum += arr[i];
    }
    avg = (double)sum / size;

    return avg;
}
```

```
bolthakziy@bolthakziy-340XAA
Average value is 214.400000
bolthakziy@bolthakziy-340XAA
```

‘gerAverage(balance, 5);’ 부분에서 어느 배열의 이름은 주소 그 자체라는 것을 알 수 있었습니다.

```c
#include <stdio.h>

int main(void)
{
    int var;
    int* ptr;
    int** pptr;
    var = 3000;
    ptr = &var;
    pptr = &ptr;

    printf("Value of var = %d\n", var);
    printf("Value available at *ptr = %d\n", *ptr);
    printf("Value available at **pptr = %d\n", **pptr);

    return 0;
}
```

```
Value of var = 3000
Value available at *ptr = 3000
Value available at **pptr = 3000
```



HW1~HW3

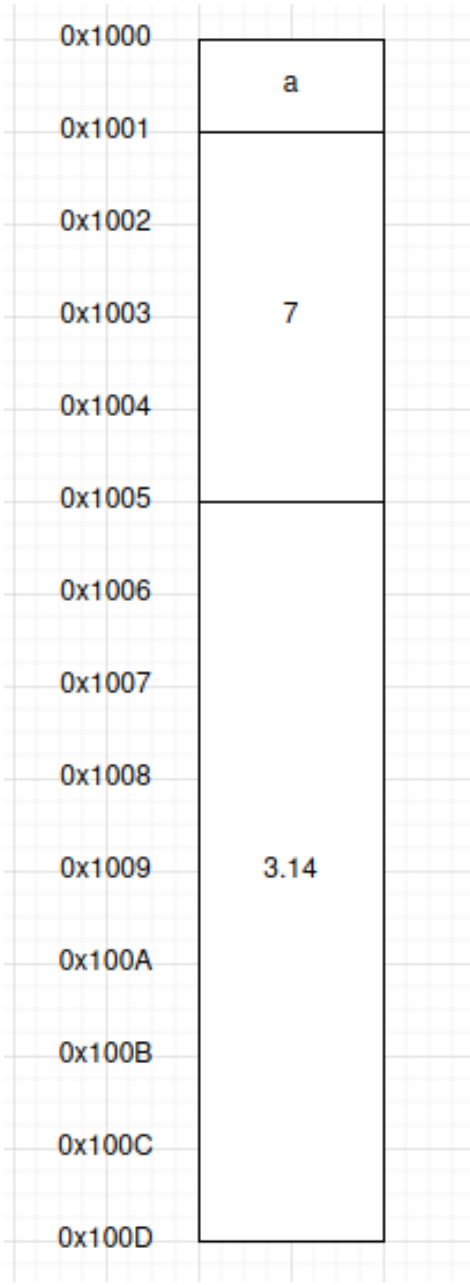
임베디드스쿨1기
Lv1과정
2020. 08. 01
권 영근

HW1

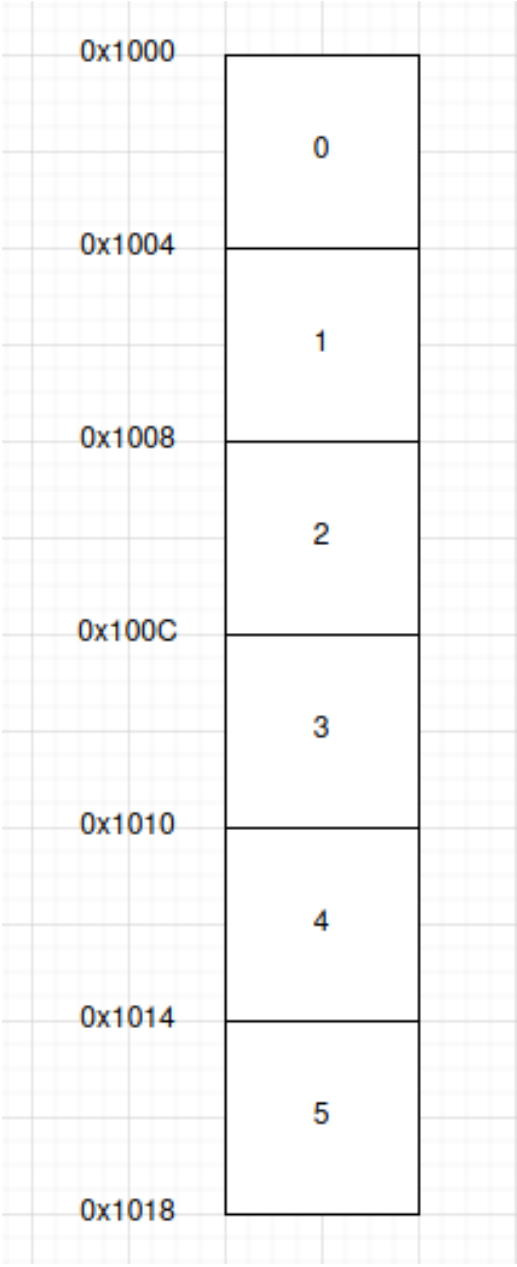
※ 변수의 주소값은 변수의 시작 주소값을 의미합니다.

※ 포인터 변수의 크기는 CPU마다 다르지만 대개 4 바이트입니다.

※ 메모리의 인접한 두 주소들의 차이는 ‘1 바이트’ 입니다.



HW2



HW3

1) &a = 0x1000

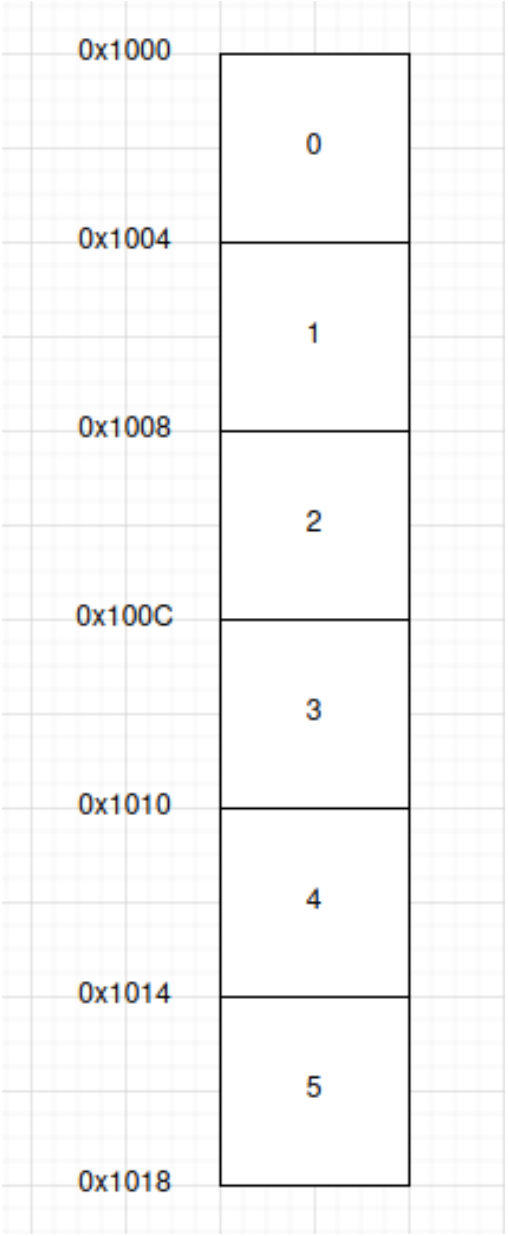
2) &a + 1 = 0x1018

3) a + 1 = 0x100C

4) *a = 0x1000

5) *a + 1 = 0x1004

6) **a = 0



좀 더 보충을 해보겠습니다.

‘int a[2][3] = {{0, 1, 2}, {3, 4, 5}};’ 에서

&a = 0x1000으로서 배열 a의 시작 주소를 가리킵니다.

(&a + 1) = 0x1018로서 배열 a의 마지막 부분의 주소를 가리킵니다.

a + 1 = 0x100C로서 배열 a의 두 번째 행의 시작 주소를 가리킵니다.

*a = 0x1000으로서 역시 &a와 같이 배열 a의 시작 주소를 가리킵니다.

(*a + 1) = 0x1004로서 배열 a의 첫 번째 행의 두 번째 열의 시작 주소를 가리킵니다.

**a = 0으로서 배열 a의 첫 번째 행의 첫 번째 열, 즉 최초의 원소의 값을 가리킵니다.

Pointer
&
Memory(심화)

임베디드스쿨1기
Lv1과정
2020. 08. 01
권 영근

Data Segment

데이터 세그먼트는 객체 파일의 일부 혹은 초기화된 정적 변수들인 전역 변수들과 정적 지역 변수들을 포함하는 프로그램 공간의 해당 주소입니다. 이 세크먼트의 크기는 프로그램 소스에 있는 값들의 크기에 따라 결정이 되고 실행 시간에는 바꿔지지 않습니다.

변수값은 실행 시간 중에 바꿔질 수 있기 때문에 데이터 세그먼트는 read/write 용입니다. 이는 변수보다는 정적 상수를 포함하는 읽기 전용 데이터 세그먼트(rodata 세그먼트 혹은 .rodata)와는 대조적입니다. 또한 많은 컴퓨터 구조들에서 읽기 전용인 텍스트 세그먼트로 알려진 코드 세그먼트와도 대조적입니다. 초기화되지 않은 변수들과 상수들 모두 BSS 세그먼트에 있습니다.

역사적으로 내부 주소 레지스터가 허용하는 원래 크기보다 큰 메모리 주소 공간들을 지원하는 것이 가능하도록 초기 CPU들은 분할 시스템을 구현하여 특정 지역에 오프셋으로 사용하기 위해 작은 인덱스 세트를 저장하였습니다. 인텔 8086 제품군들은 4개의 세그먼트들을 제공하였습니다 : 코드 세그먼트, 데이터 세그먼트, 스택 세그먼트와 부가적인 세그먼트들입니다. 각 세그먼트들은 실행 중인 소프트웨어에 의하여 메모리의 특정 위치에 배치되었고 저들 세그먼트들 안에 있는 데이터에서 작동되는 모든 지시어들은 세그먼트 시작점들과 연관되면서 수행이 됩니다. 이는 보통 64KB 메모리 공간에 접근이 가능한 16비트 주소 레지스터가 1MB 메모리 공간에 접근하는 것을 허용합니다.

오늘날의 프로그래밍 언어들과 개념으로 이어지는 특정 작업들을 지닌 개별적인 블록들로

나누는 것은 현대 프로그래밍 언어들에서 여전히 널리 사용이 됩니다.

컴퓨터 프로그램 메모리는 크게 2개의 부분으로 분류될 수 있습니다 : Read 전용과 Read/Write들입니다. 이 구별은 초기 Mask ROM, PROM 혹은 EEPROM같은 Read 전용 메모리에 메인 프로그램을 넣는 초기 시스템으로부터 자라났습니다. 시스템이 점점 복잡해지고 프로그램들이 ROM에서 실행되는 대신 다른 매체에서 RAM으로 로딩됨에 따라 프로그램 메모리의 일부분은 수정되지 말아야 한다는 생각이 유지되었습니다. 이 부분들은 프로그램의 '.text' 세그먼트와 '.rodata' 세그먼트로 되었고 나머지들은 특정 작업들을 위한 많은 여러 세그먼트들로 나뉘어지도록 Write될 수 있습니다.

◉ Text

텍스트 세그먼트 혹은 단순히 텍스트라고 알려진 코드 세그먼트는 객체 파일의 일부분 혹은 실행가능한 지시어들을 포함하는 프로그램 주소 공간의 해당 구역들이 저장되는 곳이고 일반적으로 Read 전용이고 고정된 크기를 가지고 있습니다.

◉ Data

데이터 세그먼트는 미리 정의된 값을 가지고 있으며 수정될 수 있는 전역 변수들 혹은 정적 변수들을 포함하고 있습니다. 곧 함수 내에서 정의되지 않거나(그리고 어디에서나 접근할 수

있는) 함수 내에서 정의되었지만 정적으로 정의되어서 연속 호출에 걸쳐져서 주소값을 유지하는 변수들입니다. C 프로그래밍의 예를 살펴보겠습니다:

```
int val = 3;
char string[] = “Hello World”;
```

처음에 이들 변수들의 값들은 Read 전용 메모리(전형적으로 .text 안에) 안에 저장되고 프로그램의 스타트업 루틴 동안에 .data 세그먼트로 복사가 됩니다.

위의 예에서 만일 이들 변수들이 함수 내에서 선언이 된다면, 지역 스택 프레임에 기본값으로 저장된다는 점을 주의하십시오.

◉ BSS

초기화되지 않은 데이터로 알려진 BSS 세그먼트는 대개 데이터 세그먼트와 인접해 있습니다. BSS 세그먼트는 0으로 초기화되거나 소스 코드에서 명시적으로 초기화되지 않은 모든 전역 변수들과 정적 변수들을 포함합니다. 예를 들어서 ‘static int i;’ 로 정의된 변수는 BSS 세그먼트에 포함이 됩니다.

◉ Heap

힙 영역은 대개 .bss 세그먼트와 .data 세그먼트의 끝 부분에서 시작을 하여 높은 주소들로 확장을 해나갑니다. 힙 영역은 ‘brk’ 시스템 콜과 ‘sbrk’ 시스템 콜들을 사용하여 그것의 크기를 조정하는 ‘malloc’, ‘calloc’, ‘realloc’ 그리고 ‘free’ 함수들에 의해 관리됩니다(brk/sbrk와 단일 힙 영역을 사용하는 것은 malloc/calloc/realloc/free와의 계약을 충족하는데에 있어서 필요하지는 않습니다; 그들은 ‘mmap’/’munmap’ 들을 사용하여 잠재적으로 프로세스의 가장 주소 공간으로 비연속적인 가상 메모리 영역을 예약/해약을 하면서 구현될지도 모릅니다.). 힙 영역은 모든 쓰레드들, 공유된 라이브러리들 그리고 프로세스에서 동적으로 로드가 된 모듈들에 의해 공유가 됩니다.

◉ Stack

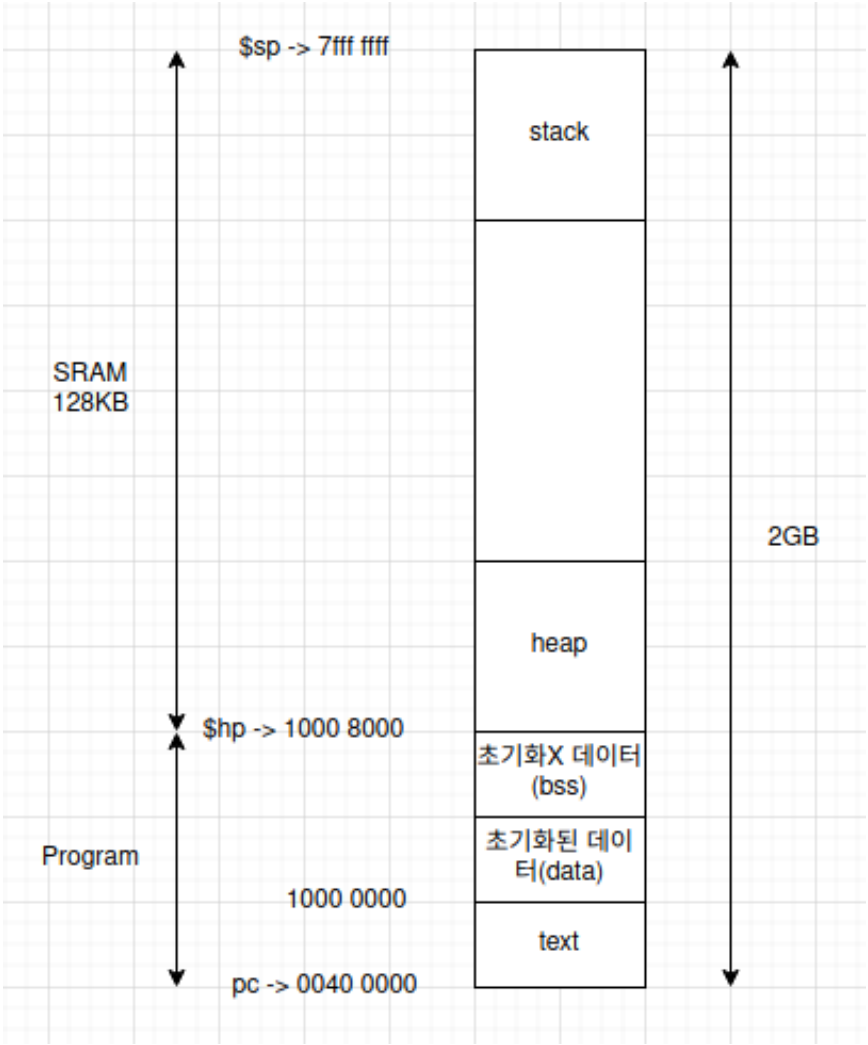
스택 영역은 전형적으로 메모리의 상위 부분에 위치한 LIFO 구조인 프로그램 스택을 포함합니다. “스택 포인터” 레지스터는 스택의 꼭대기를 추적을 합니다; 스택에 값이 “push” 될 때마다 값은 조정이 됩니다. 함수 호출을 위해 push되는 값들 집합을 ‘스택 프레임’ 이라고 합니다. 자동 변수들 또한 스택에 할당이 됩니다.

스택 영역은 전통적으로 힙 영역과 인접하였고 그들은 서로를 향해 확장해 나갔습니다; 스택 포인터가 힙 포인터와 만났을 때, 자유 메모리는 소진되었습니다. 넓은 주소 공간과 가상 메모리 기술들로 스택 영역과 힙 영역들은 좀 더 자유롭게 배치되는 경향이 있지만 여전히 전형적으로 한 곳으로 집중되는 방향으로 확장합니다. 표준 PC x86 구조에서 스택은 주소

0을 향해 증가를 하며 이는 수적으로 낮은 주소들에서 최신 항목들이 더 많아지고 콜 체인에서 더 깊어지며 힙에 더 가까워지는 것을 의미합니다. 다른 구조에서는 반대 방향으로 확장을 합니다.

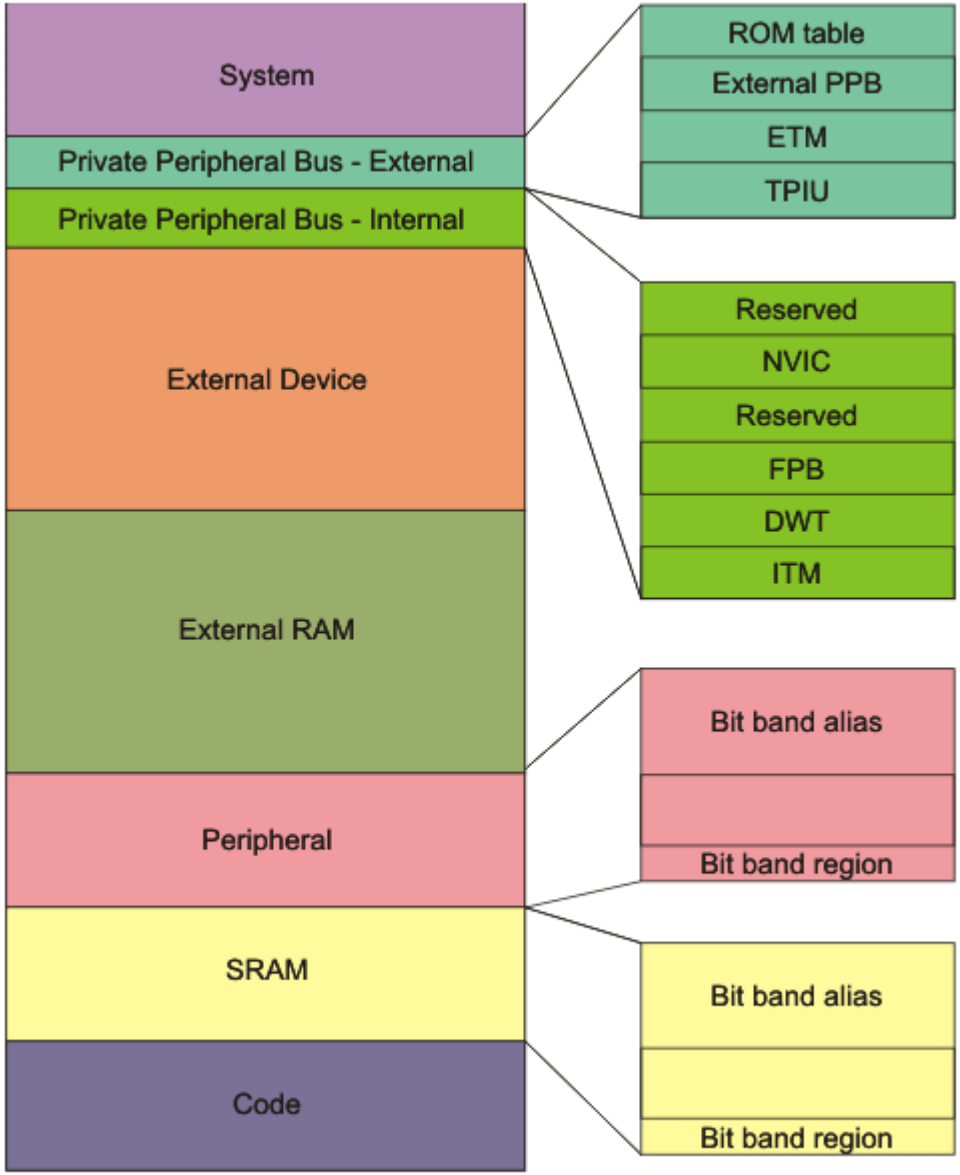
Convention Program Memory

우리가 프로그래밍하는 소스 코드가 할당되는 영역입니다. SRAM 영역이며 이 영역에 대해 이해를 잘 해야 프로그램의 크기까지 고려하는 개발자가 될 수 있습니다. MCU의 경우 메모리가 작아 시스템이 커질수록 메모리 관리가 중요하기 때문입니다.



Bytes	Hex	Address range
16	10	0 - F
256	100	0 - FF
1K	400	0 - 3FF
4K	1000	0 - FFF
64K	10000	0 - FFFF
1M	100000	0 - FFFFF
16M	1000000	0 - FFFFFF

ARM Memory Map



ARM Memory Map

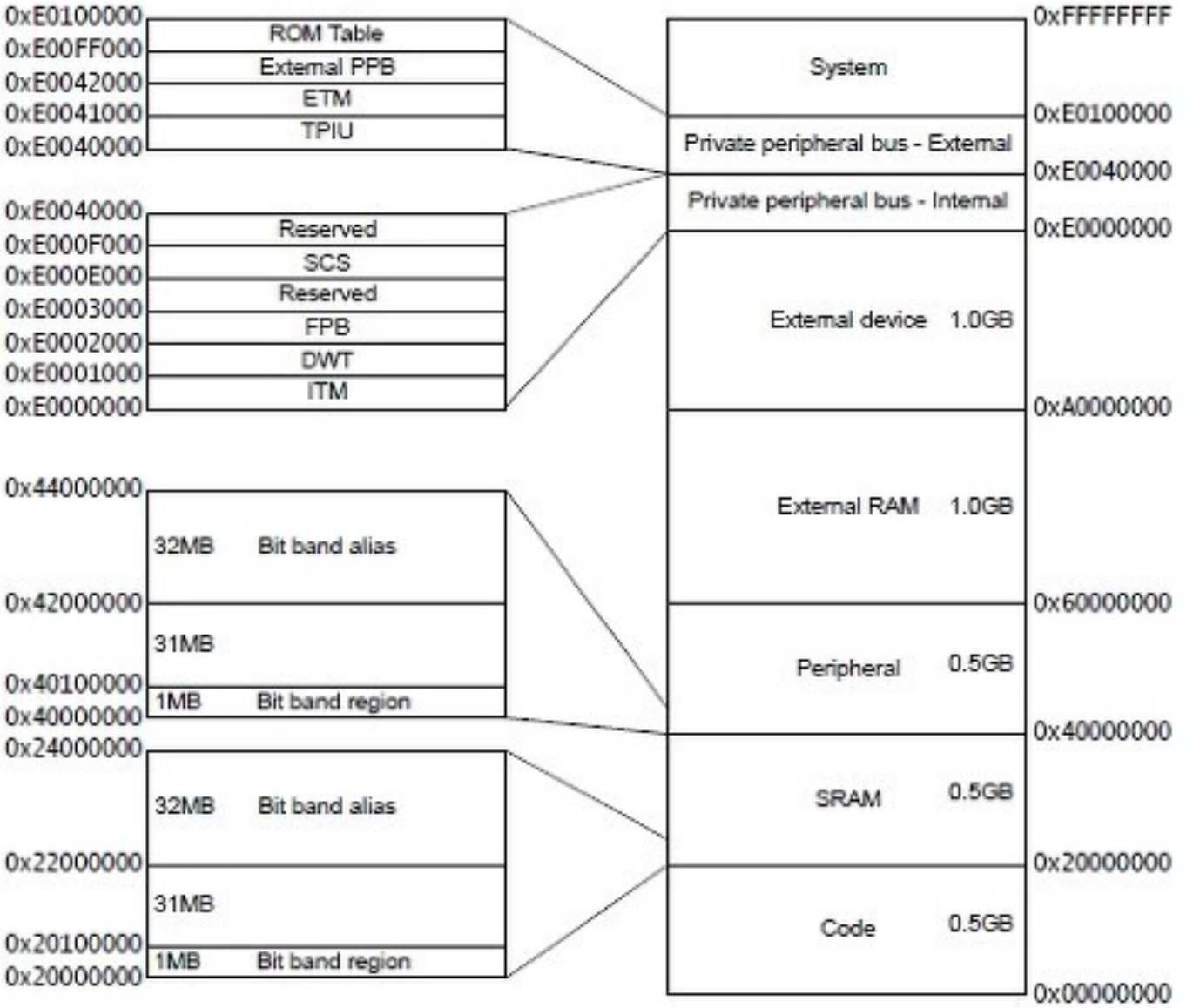
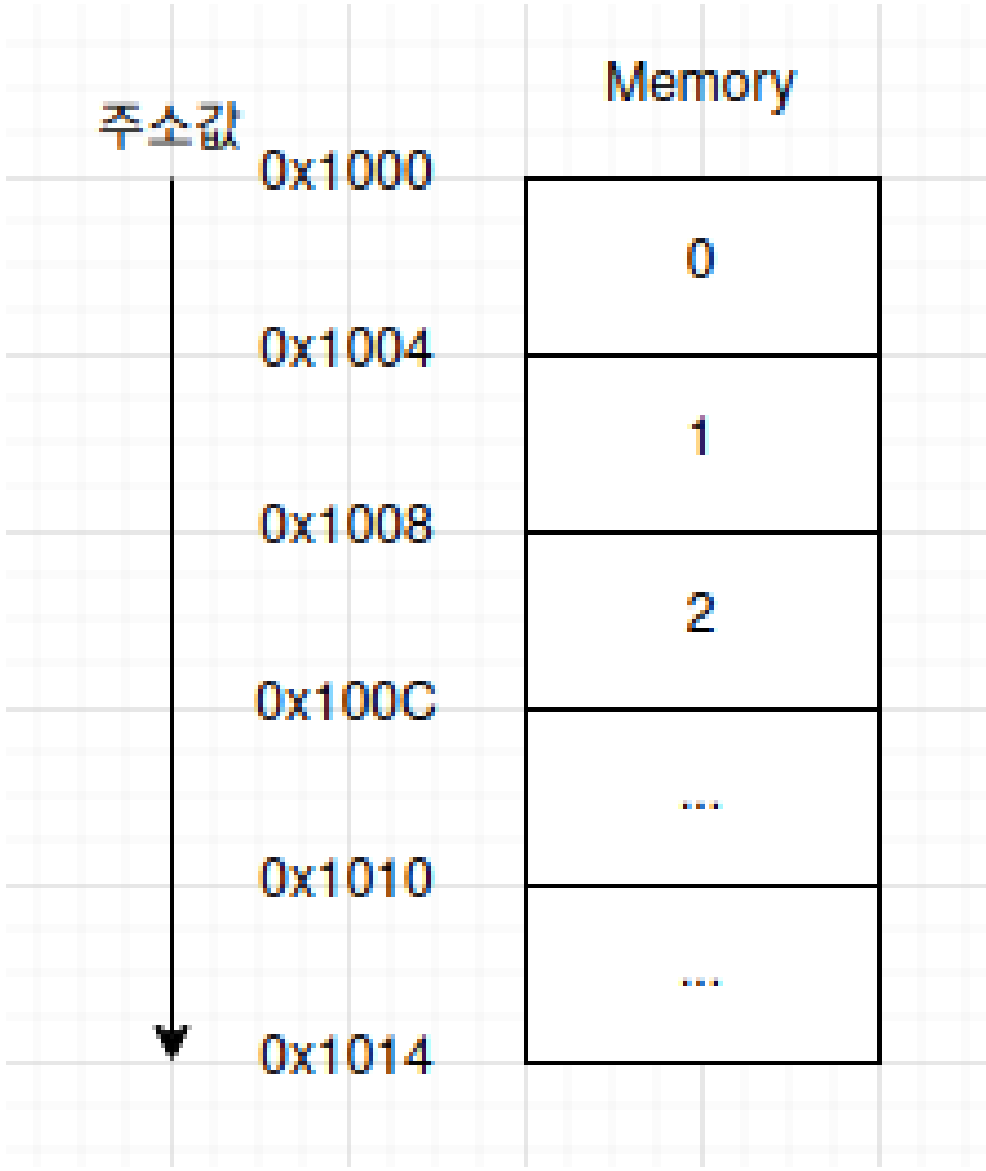


Figure 3-1 System address map

배열의 메모리 이미지

```
int main(void)
{
   int a[3] = {0, 1, 2};
}
```



포인터의 정의

포인터는 시작 주소를 값으로 가지고 있고 해당 값의 정체를 알려주는 변수입니다. 곧 주소를 값으로 담고 있는 변수입니다.

1. 포인터를 사용할 때, *(Asterisk)를 사용합니다.

2. 변수와 마찬가지로 자료형을 적은 후 '*' 를 붙이고 포인터 변수로 선언을 합니다.

ex) int* ptr;

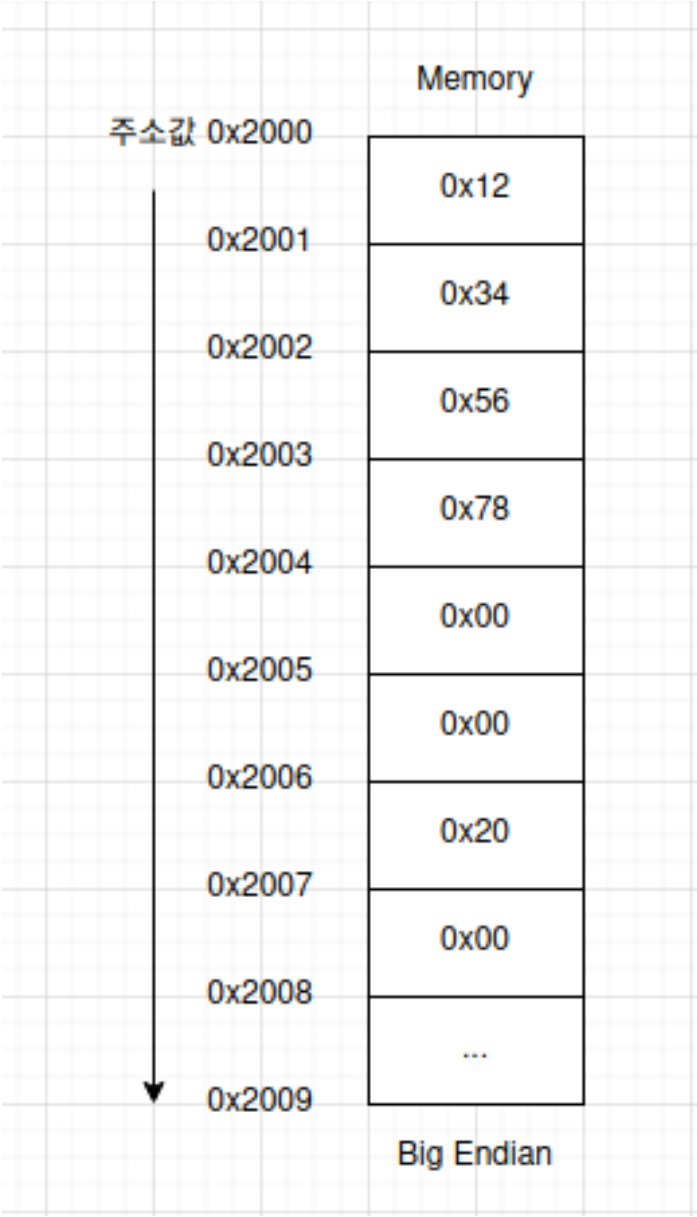
3. 포인터 변수 선언 후, 자료형 부분을 제외하고 포인터 변수 앞에 '*' 를 붙이면(*ptr) 포인터가 가리키는 메모리 값을 반환하고 '&' 를 포인터 변수 앞에 붙이면 포인터 변수의 주소 값을 반환합니다.

배열의 메모리 이미지

```
int main(void)
{
    int a = 0x12345678;
    int* ptr;

    ptr = &a;
}
```

&ptr => 이중 포인터
&&ptr => 없음, 오류

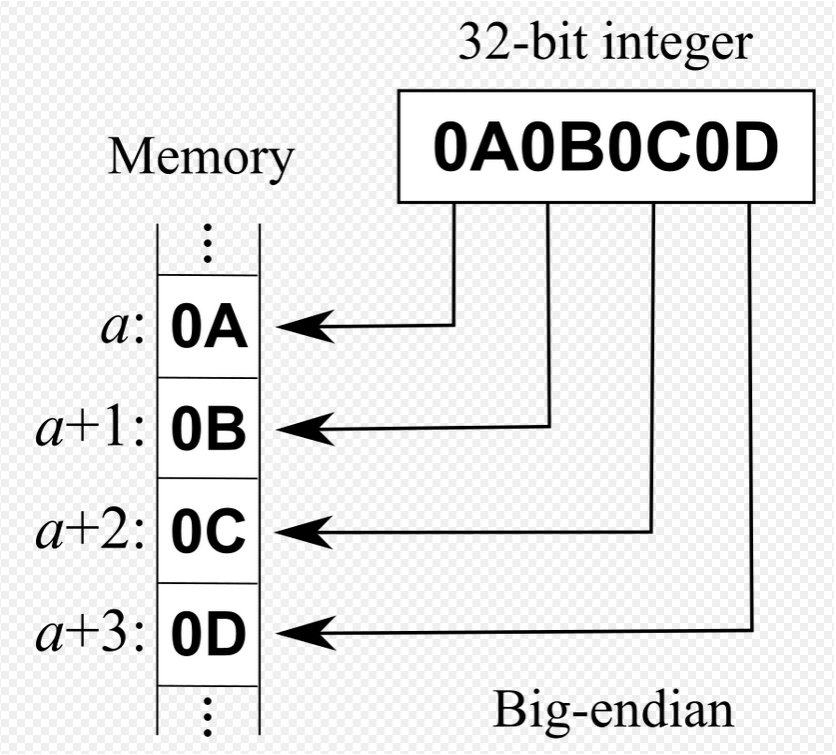
(short*)ptr => 형변환
(short)ptr => 2000번지, 2바이트만 읽음

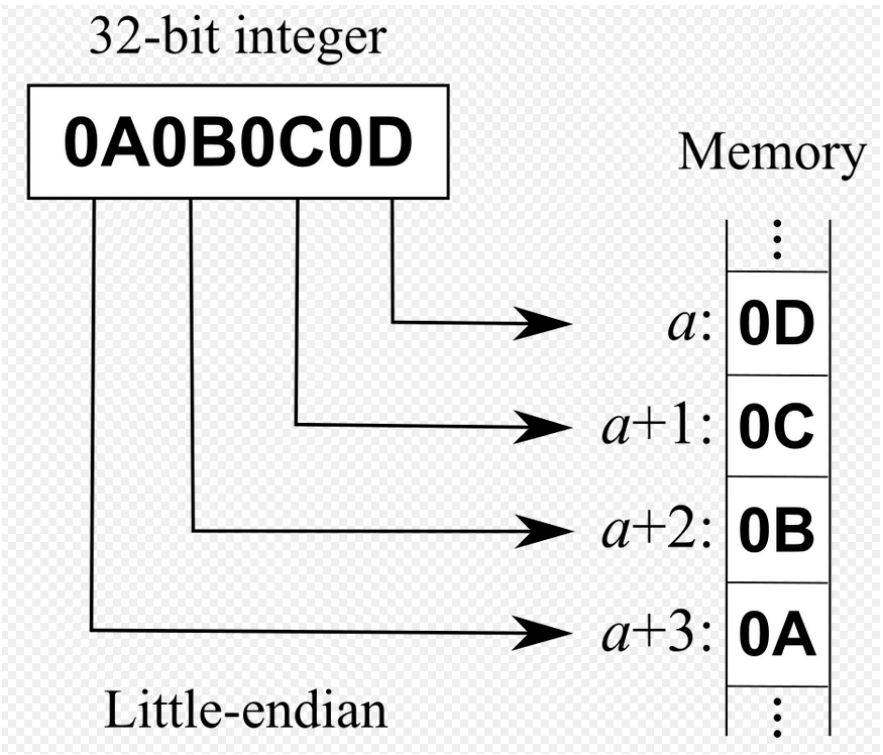


◉ Big Endian & Little Endian

먼저 엔디언(Endian)의 개념에 대해 작성해보겠습니다.

엔디언이란 캄퓨터의 메모리와 같이 1차원의 공간에 여러 개의 연속된 대상을 배열하는 방법을 의미합니다. 보통 2가지 경우로 나뉠 수 있는데 보통 큰 단위가 앞에 나오는 방식을 ‘빅 엔디언’ 이라고 하고 작은 단위가 앞에 나오는 방식을 ‘리틀 엔디언’ 이라고 합니다.





인텔 펜티엄 CPU 4415U의 랩탑에서 실습을 하였습니다. 그 결과 이 랩탑은 리틀 엔디언 방식으로 데이터를 메모리에 저장한다는 사실을 확인하였습니다.

```c
#include <stdio.h>

int main(void)
{
    int a = 0x12345678;
    int* ptr;
    short* ptr2;
    ptr = &a;

    printf("Before type conversion : %x\n", ptr);
    printf("The number in a is %x\n", *ptr);

    ptr2 = (short*)ptr;

    printf("After type conversion  : %x\n", ptr2);
    printf("The number in a is %x\n", *ptr2);

    return 0;
}
```

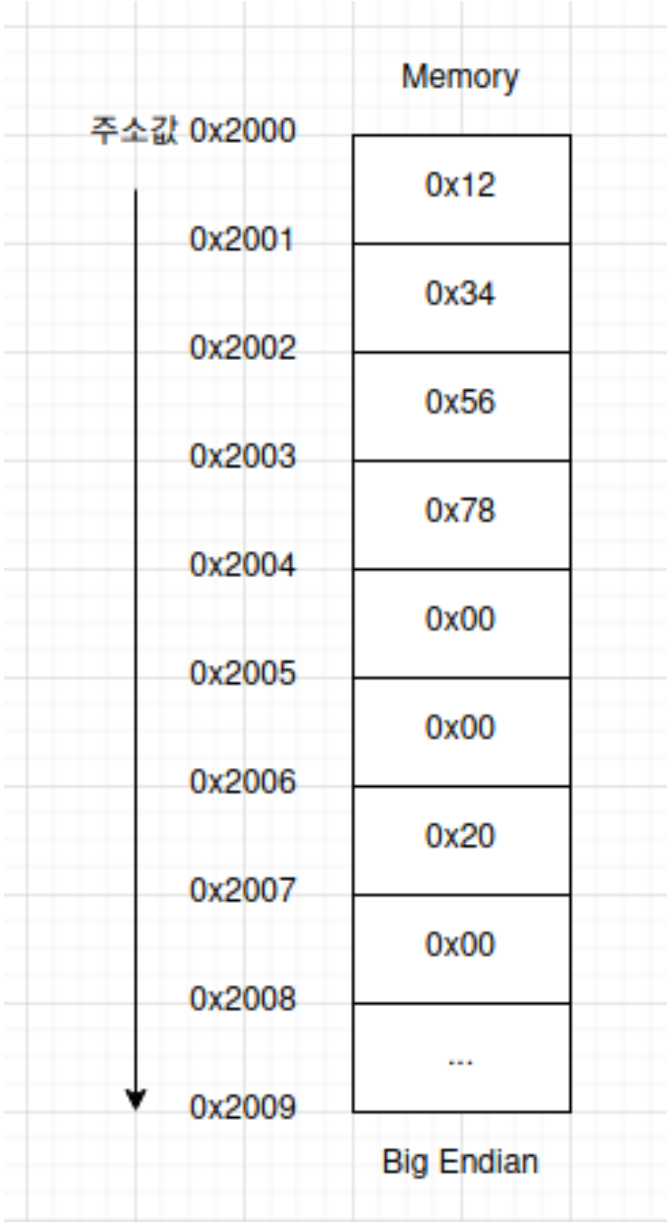
```
Before type conversion : c6064294
The number in a is 12345678
After type conversion  : c6064294
The number in a is 5678
```

&ptr => 0x2006

&&ptr => X(오류 혹은 알아야 할 가치가 없습니다.)

(short*)ptr => 0x2000

(short)ptr => 0x1234



참고 자료

1. 위키피디아

2. http://www.soen.kr/book/

3. draw.io

4. NAVER 지식 백과

5. https://www.mikroe.com/

6. https://embed-avr.tistory.com/112