# C 애플리케이션 구현
# 중간고사 포트폴리오

컴퓨터정보공학과 E반

20190706 김종휘

# 목차

# 머리말

C언어는 제가 처음 배운 프로그래밍 언어입니다. 학교에 입학하기 전 독학으로 프로그래밍 언어를 배워보자 다짐하여 공부를 시작하였는데 처음에는 C언어를 공부하면서 재미있었고 흥미도 느꼈지만, 나중엔 점점 어려워지면서 흥미가 떨어져 공부를 소홀히 하게 되었습니다. 수업 중 모르는 내용을 다시 복습하지 않고 그냥 넘어갔던 것을 반성하고 지금 제작하는 중간고사 포트폴리오를 통해 다시금 새롭게 시작함으로 초심을 되찾을 수 있는 발판이 되기를 바라며 이 포트폴리오를 제작합니다. 제 다짐을 보여주는 이 포트폴리오는 초심으로 돌아가 어려운 내용과 잊어버린 내용들을 복습하기 위해 작성되었음을 알립니다.

# C언어 소개

# C 언어의 탄생

1960년대 개발 되었던 운영체제들은 하드웨어 종속적인 언어를 사용하여 개발되었습니다. 따라서 하드웨어가 바뀌면 운영체제의 많은 부분을 다시 개발해야 했습니다. 벨 연구소의 데니스 리치와 켄 톰슨은 이런 불편함을 없애고자 하드웨어가 변경되어도 프로그램을 다시 작성하지 않아도 되는 운영체제를 만들기 위해 노력합니다. 1970년에 켄 톰슨이 B 언어를 만들었지만, 이 언어도 하드웨어로부터 독립된 운영체제를 만드는 데 적합하지 않았습니다. 이에 1972년, 켄 톰슨은 데니스 리치와 함께 새로운 언어를 개발하는데 이것이 바로 C언어 입니다.

# C 언어의 특징

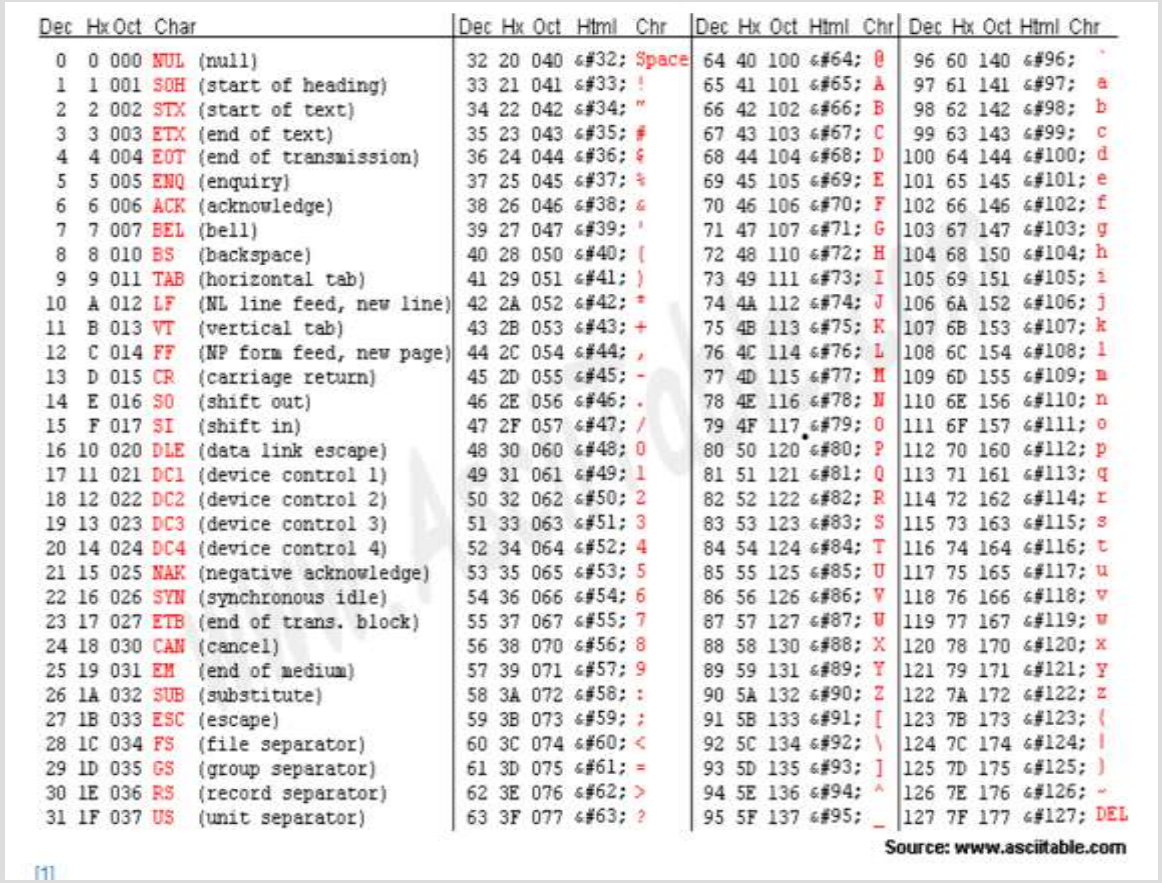
**C언어의 특징 4가지**

1. C언어로 작성된 프로그램은 다양한 하드웨어로의 이식성이 좋습니다.
2. C언어는 절차 지향 프로그래밍 언어로, 코드가 복잡하지 않아 상대적으로 유지보수가 쉽습니다.
3. C언어는 저급 언어의 특징을 가지고 있으므로, 어셈블리어 수준으로 하드웨어를 제어할 수 있습니다.
4. C언어는 코드가 간결하여, 완성된 프로그램의 크기가 작고 실행 속도가 빠릅니다.

C언어가 가지는 **단점**은 다음과 같습니다.

1. C언어는 저급 언어의 특징을 가지고 있으므로, 자바와 같은 다른 고급 언어보다 배우기가 쉽지 않습니다.
2. C언어는 다른 언어와는 달리 시스템 자원을 직접 제어할 수 있으므로, 프로그래밍하는데 세심한 주의를 기울여야 합니다.

# 아스키(ASCII)코드



| Dec | Hx | Oct | Char | Dec | Hx | Oct | Html | Chr | Dec | Hx | Oct | Html | Chr | Dec | Hx | Oct | Html | Chr |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 000 | NUL (null) | 32 | 20 | 040 |   | Space | 64 | 40 | 100 | @ | @ | 96 | 60 | 140 | ` | ` |
| 1 | 1 | 001 | SOH (start of heading) | 33 | 21 | 041 | ! | ! | 65 | 41 | 101 | A | A | 97 | 61 | 141 | a | a |
| 2 | 2 | 002 | STX (start of text) | 34 | 22 | 042 | " | " | 66 | 42 | 102 | B | B | 98 | 62 | 142 | b | b |
| 3 | 3 | 003 | ETX (end of text) | 35 | 23 | 043 | # | # | 67 | 43 | 103 | C | C | 99 | 63 | 143 | c | c |
| 4 | 4 | 004 | EOT (end of transmission) | 36 | 24 | 044 | $ | $ | 68 | 44 | 104 | D | D | 100 | 64 | 144 | d | d |
| 5 | 5 | 005 | ENQ (enquiry) | 37 | 25 | 045 | % | % | 69 | 45 | 105 | E | E | 101 | 65 | 145 | e | e |
| 6 | 6 | 006 | ACK (acknowledge) | 38 | 26 | 046 | & | & | 70 | 46 | 106 | F | F | 102 | 66 | 146 | f | f |
| 7 | 7 | 007 | BEL (bell) | 39 | 27 | 047 | ' | ' | 71 | 47 | 107 | G | G | 103 | 67 | 147 | g | g |
| 8 | 8 | 010 | BS (backspace) | 40 | 28 | 050 | ( | ( | 72 | 48 | 110 | H | H | 104 | 68 | 150 | h | h |
| 9 | 9 | 011 | TAB (horizontal tab) | 41 | 29 | 051 | ) | ) | 73 | 49 | 111 | I | I | 105 | 69 | 151 | i | i |
| 10 | A | 012 | LF (NL line feed, new line) | 42 | 2A | 052 | * | * | 74 | 4A | 112 | J | J | 106 | 6A | 152 | j | j |
| 11 | B | 013 | VT (vertical tab) | 43 | 2B | 053 | + | + | 75 | 4B | 113 | K | K | 107 | 6B | 153 | k | k |
| 12 | C | 014 | FF (NP form feed, new page) | 44 | 2C | 054 | , | , | 76 | 4C | 114 | L | L | 108 | 6C | 154 | l | l |
| 13 | D | 015 | CR (carriage return) | 45 | 2D | 055 | - | - | 77 | 4D | 115 | M | M | 109 | 6D | 155 | m | m |
| 14 | E | 016 | SO (shift out) | 46 | 2E | 056 | . | . | 78 | 4E | 116 | N | N | 110 | 6E | 156 | n | n |
| 15 | F | 017 | SI (shift in) | 47 | 2F | 057 | / | / | 79 | 4F | 117 | O | O | 111 | 6F | 157 | o | o |
| 16 | 10 | 020 | DLE (data link escape) | 48 | 30 | 060 | 0 | 0 | 80 | 50 | 120 | P | P | 112 | 70 | 160 | p | p |
| 17 | 11 | 021 | DC1 (device control 1) | 49 | 31 | 061 | 1 | 1 | 81 | 51 | 121 | Q | Q | 113 | 71 | 161 | q | q |
| 18 | 12 | 022 | DC2 (device control 2) | 50 | 32 | 062 | 2 | 2 | 82 | 52 | 122 | R | R | 114 | 72 | 162 | r | r |
| 19 | 13 | 023 | DC3 (device control 3) | 51 | 33 | 063 | 3 | 3 | 83 | 53 | 123 | S | S | 115 | 73 | 163 | s | s |
| 20 | 14 | 024 | DC4 (device control 4) | 52 | 34 | 064 | 4 | 4 | 84 | 54 | 124 | T | T | 116 | 74 | 164 | t | t |
| 21 | 15 | 025 | NAK (negative acknowledge) | 53 | 35 | 065 | 5 | 5 | 85 | 55 | 125 | U | U | 117 | 75 | 165 | u | u |
| 22 | 16 | 026 | SYN (synchronous idle) | 54 | 36 | 066 | 6 | 6 | 86 | 56 | 126 | V | V | 118 | 76 | 166 | v | v |
| 23 | 17 | 027 | ETB (end of trans. block) | 55 | 37 | 067 | 7 | 7 | 87 | 57 | 127 | W | W | 119 | 77 | 167 | w | w |
| 24 | 18 | 030 | CAN (cancel) | 56 | 38 | 070 | 8 | 8 | 88 | 58 | 130 | X | X | 120 | 78 | 170 | x | x |
| 25 | 19 | 031 | EM (end of medium) | 57 | 39 | 071 | 9 | 9 | 89 | 59 | 131 | Y | Y | 121 | 79 | 171 | y | y |
| 26 | 1A | 032 | SUB (substitute) | 58 | 3A | 072 | : | : | 90 | 5A | 132 | Z | Z | 122 | 7A | 172 | z | z |
| 27 | 1B | 033 | ESC (escape) | 59 | 3B | 073 | ; | ; | 91 | 5B | 133 | [ | [ | 123 | 7B | 173 | { | { |
| 28 | 1C | 034 | FS (file separator) | 60 | 3C | 074 | < | < | 92 | 5C | 134 | \ | \ | 124 | 7C | 174 | | | \| |
| 29 | 1D | 035 | GS (group separator) | 61 | 3D | 075 | = | = | 93 | 5D | 135 | ] | ] | 125 | 7D | 175 | } | } |
| 30 | 1E | 036 | RS (record separator) | 62 | 3E | 076 | > | > | 94 | 5E | 136 | ^ | ^ | 126 | 7E | 176 | ~ | ~ |
| 31 | 1F | 037 | US (unit separator) | 63 | 3F | 077 | ? | ? | 95 | 5F | 137 | _ | _ | 127 | 7F | 177 | &#127; | DEL |

Source: www.asciitable.com

[1]

문자를 숫자로 표현하기위한 약속으로 아스키는 1967년에 표준으로 제정되어 1986년에 마지막으로 개정되었습니다. 아스키는 초창기에 7비트 방식으로 인코딩 되었지만 다양한 표현이 필요해짐에 따라 8비트 인코딩을 사용하도록 확장 되었습니다. 총 256개의 숫자로 문자를 표현하기 때문에 0~255 범위를 가지며 이 범위는 부호를 고려하지 않는 1바이트 메모리에 저장할 수 있습니다. 따라서 1바이트 메모리에 저장하는 것이 가장 효율적입니다.

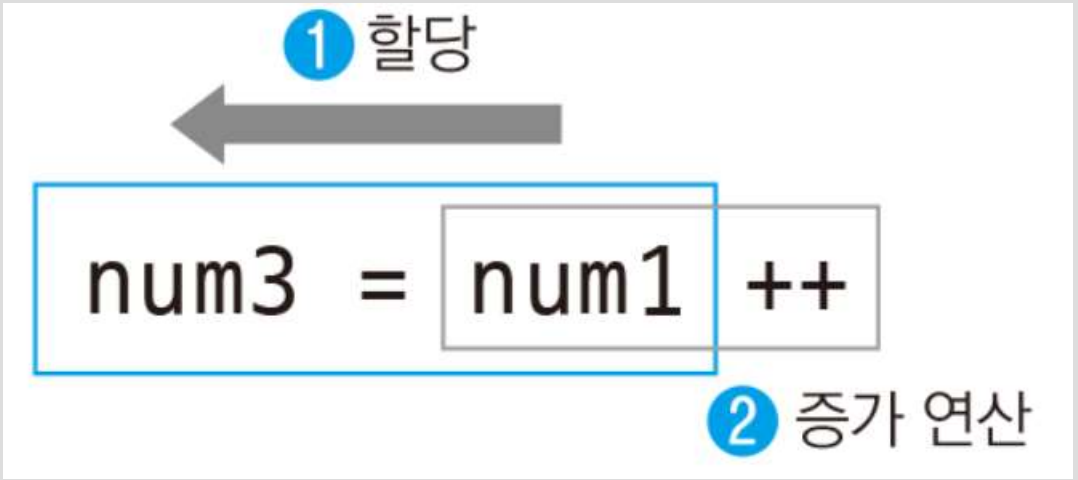
# 복습

# 증감 연산자 ++, --

```c
#include <stdio.h>

int main()
{
    int num1 = 2;
    int num2 = 2;
    int num3;
    int num4;

    num3 = num1++;    // num1의 값을 num3에 할당한 뒤 num1의 값을 1 증가시킴
    num4 = num2--;    // num2의 값을 num4에 할당한 뒤 num2의 값을 1 감소시킴

    printf("%d %d\n", num3, num4);    // 2 2

    return 0;
}
```

실행 결과

```
2 2
```



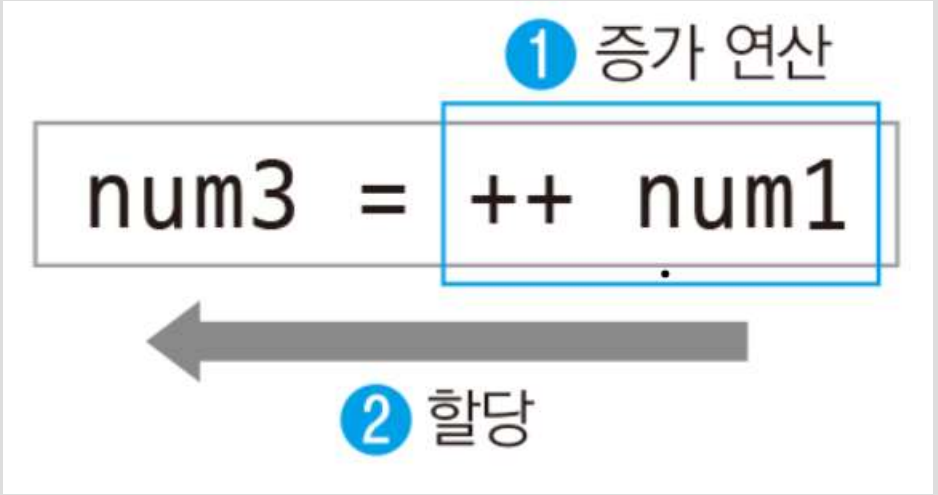

후위 연산자는 할당 이후에 연산 하기 때문에 결과값이 변하지 않을 것을 알 수 있습니다.

# 증감 연산자 ++, --

```c
#include <stdio.h>

int main()
{
    int num1 = 2;
    int num2 = 2;
    int num3;
    int num4;

    num3 = ++num1;    // num1의 값을 1 증가시킨 뒤 num3에 할당
    num4 = --num2;    // num2의 값을 1 감소시킨 뒤 num4에 할당

    printf("%d %d\n", num3, num4);    // 3 1

    return 0;
}
```

실행 결과

```
3 1
```




전위연산자는 할당 전에 연산하기 때문에 결과값이 변하는 것을 알 수 있습니다.

# 증감 연산자 (연습문제)

```c
#include <stdio.h>

int main()
{
    int num1 = ①______;
    int num2 = ②______;
    int num3;
    int num4;

    num1++;
    num3 = --num1;

    --num2;
    num4 = num2++;

    printf("%d\n", num3);
    printf("%d\n", num4);

    return 0;
}
```

실행 결과

```
2
7
```

정답

① 2
② 8

코드를 역순으로 살펴보면 값을 유추해볼 수 있습니다.

num3 = --num1;
의 값이 2면 전위 연산을 했기 때문에
num1의 값은 3이고
num1++; 으로 값이 1늘었기 때문에
num1의 초기값은 2 입니다.

num4 = num2++;
의 값이 7이면 후위연산으로 증가하기 전 값이며
--num2;로 값이 1줄었기 때문에
num2의 초기값은 8입니다.

# 다중 for문

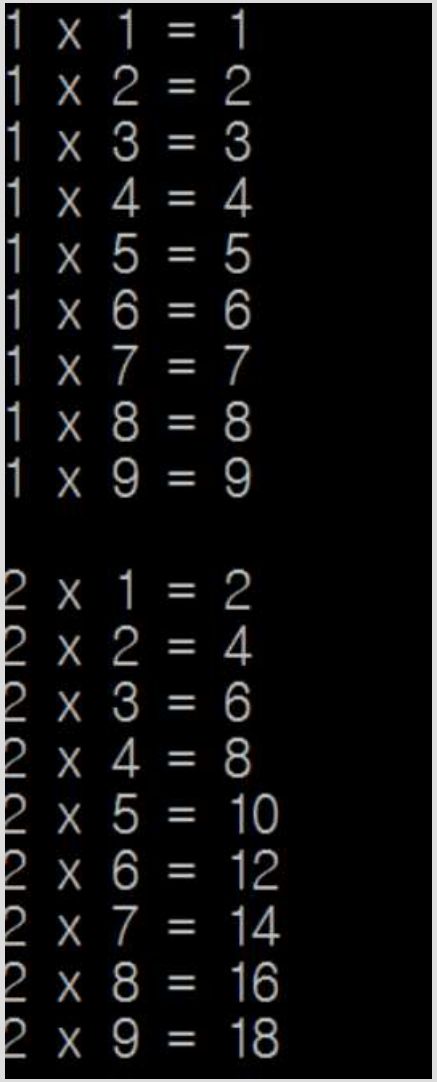
```c
#include <stdio.h>

int main()
{
    for (int i = 0; i < 5; i++)    // 5번 반복. 바깥쪽 루프는 세로 방향
    {
        for (int j = 0; j < 5; j++)    // 5번 반복. 안쪽 루프는 가로 방향
        {
            printf("j:%d ", j);        // j값 출력
        }

        printf("i:%d\\n", i);      // i값 출력, 개행 문자 모양도 출력
        printf("\n");              // 가로 방향으로 숫자를 모두 출력한 뒤 다음 줄로 넘어감
    }

    return 0;
}
```

결과


첫번째 for문은 열을 의미하며 두번째 for문은 행을 의미합니다.
i 값이 0부터 4까지 반복하기 때문에 5줄이 출력되고
j 값도 0부터 4까지 반복하기 때문에 5칸이 출력됩니다.

# 다중 for문 (구구단)

```c
#include <stdio.h>

int main()
{
    //기본적인 구구단 세로 출력
    for (int i = 1; i <= 9; i++)
    {
        for (int j = 1; j <= 9; j++)
        {
            printf("%d x %d = %d₩n", i, j, (i*j));
        }
        printf("₩n");
    }

    return 0;
}
```




다중 for문으로 구구단을 출력

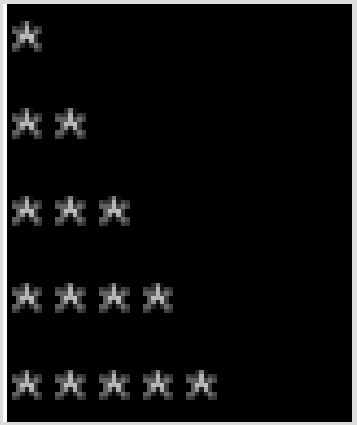
1단부터 9단까지 반복하기 때문에
바깥쪽 for문의 i의 값은 1부터 9까지
반복합니다.

또한 구구단은 1부터 9를 곱하기 때문에
안쪽 for문의 j의 값도 1부터 9까지
반복합니다.

다음은 이를 응용한 문제가 있겠습니다.

# 다중 for문 (연습 문제)

다중 for문을 이용하여 다음과 같은 피라미드를 *로 출력하세요 (빈칸 채우기)



```
for(int i=0;    ;i++){
    for(int j=0;    ;j++){
        printf("*");
    }
    printf("\n");
}
```

피라미드가 5줄이기 때문에 첫번째 for문은 i<5를 이용하여 5번 반복합니다.

두번째 for문은 별의 개수가 1개씩 점점 늘어나야 합니다.
그것을 i와 연결하여 j<=i로 하여
첫번째 줄 출력은 i가 0 1회 출력
i가 1 이면 2회 출력
i가 2 이면 3회 출력
i가 3 이면 4회 출력
i가 4 이면 5회 출력
하여 다음과같은 피라미드를 출력할 수 있습니다.

답 5, j<=i

# 포인터

```c
#include <stdio.h>

int main() {
    int *p = NULL;  // int* p == int * p 모두 같음
    int num = 15;

    p = &num;

    printf("int 변수 num의 주소 : %d \n", &num);
    printf("포인터 p의 값 : %d \n", p);
    printf("포인터 p가 가리키는 값 : %d \n", *p);

    return 0;
}
```

```
> int 변수 num의 주소 : -1592303284
포인터 p의 값 : -1592303284
포인터 p가 가리키는 값 : 15
```

포인터는 “주소”를 가리키며, 포인터 변수라고 부르기도 합니다.
포인터 변수를 선언할 때는 자료형에 *(참조 연산자)를 붙여서 선언하고 주소연산자 &을 이용하여 주소를 대입합니다.

동일한 운영체제 시스템일 경우 주소 값이 동일한 크기를 갖기 때문에 포인트 변수의 크기는 모두 동일합니다.

옆에 코드를 보면
int *p = NULL; 포인트를 선언과 동시에 초기화
p = # 포인터 변수에 주소 값 대입 하여
&num 과 p의 값은 같고
*p 는 15의 값을 가리키게 됩니다.

# 포인터

```c
#include <stdio.h>

int main() {
  int *p = NULL;
  int num = 15;

  p = &num;
  printf("포인터 p가 가리키는 값 : %d\n", *p);
  printf("num의 값 : %d\n\n", num);

  *p += 5;
  printf("포인터 p가 가리키는 값 : %d\n", *p);
  printf("num 값 : %d\n\n", num);

  (*p)++;
  printf("포인터 p가 가리키는 값 : %d\n", *p);
  printf("num 값 : %d\n\n", num);

  *p++;
  printf("포인터 p가 가리키는 값 : %d\n", *p);
  printf("num 값 : %d\n", num);

  return 0;
}
```

*p는 포인터가 가리키는 값인 15가 출력
*p += 5;
포인터가 가리키는 값에 5증가 이므로
num값도 5증가
*p++;
증감연산자가 참조연산자보다 우선순위가 높기
때문에 p++이 먼저 수행되어 p에 저장된 주소 값이
1증가하기 때문에 쓰레기 값이 출력
(*p)++;
()로 우선순위를 설정해 num값이 1증가한 21 출력

```
> 포인터 p가 가리키는 값 : 15
num의 값 : 15

포인터 p가 가리키는 값 : 20
num 값 : 20

포인터 p가 가리키는 값 : 21
num 값 : 21

포인터 p가 가리키는 값 : 926994960
num 값 : 21
```

# 포인터

```c
#include <stdio.h>

void pointerPlus(int *num){
  *num += 5;

}

void numPlus(int num){
  num += 5;

}

int main() {
  int num = 15;
  printf("num 값 : %d\n", num);

  numPlus(num);
  printf("numPlus 사용 후 : %d\n", num);

  pointerPlus(&num);
  printf("pointerPlus 사용 후 : %d\n", num);

  return 0;
}
```

포인터는 함수를 사용할 때 진가를 발휘합니다.

num의 값이 numPlus()함수를 사용할 때는
값의 변화가 없지만,
pointerPlus()함수에서는 값이 5증가함을 볼 수
있습니다.

즉 함수에서 수정할 수 없는 변수와는 달리
포인터는 포인터로 메모리의 주소를 넘겨 함수에서
메모리에 직접적으로 참조하여 변수의 값을 바로
수정하는 것이 가능한 것입니다.

```
> num 값 : 15
numPlus 사용 후 : 15
pointerPlus 사용 후 : 20
```

# 포인터

```c
#include <stdio.h>

int main()
{
    int arr[5] = { 10, 20, 30, 40, 50 };
    int *arrPtr = arr;

    printf("%d\n", *arrPtr);
    printf("%d\n", arr[0]);

    return 0;
}
```

```
> 10
10
```

배열의 주소는 연속 되어있습니다. 또한 배열의 이름은 포인터 변수와 같은 기능을 하며, 첫번째 요소의 주소 값을 나타냅니다.이런 배열의 특성을 포인터 연산에 이용하면 유용하게 사용 가능합니다.

&연산자를 사용하지않고 arr 이름 자체가 주소 값이기 때문에 바로 포인터에 대입 가능합니다. scanf로 입력 받을 때 문자열을 &연산자를 붙여주지 않아도 되는 이유와 같습니다.

# 포인터

```c
#include <stdio.h>

int main()
{
  int arr[5] = { 10, 20, 30, 40, 50 };
  double arr2[5] = { 10.1, 20.2, 30.3, 40.4, 50.5 };
  int *arrPtr = arr;
  double *arrPtr2 = arr2;

  printf("포인터 주소 : %d %d\n", arrPtr++, arrPtr2++);
  printf("증가 연산 후 : %d %d\n", arrPtr, arrPtr2);
  printf("변수 값 : %d %.2f\n", *arrPtr, *arrPtr2);

  arrPtr += 2;
  arrPtr2 += 2;

  printf("증가 연산 후 : %d %d\n", arrPtr, arrPtr2);
  printf("변수 값 : %d %.2f\n", *arrPtr, *arrPtr2);

  return 0;
}
```

포인터 변수도 일반 변수처럼 증감 연산을 할 수 있습니다. 옆 코드를 보면 int 포인터인 arrPtr은 1더하니 4증가하고 double포인터인 arrPtr2는 8씩 증가합니다. 즉 포인터 변수가 n만큼 더하거나 뺄 때 자료형의 크기 x n만큼 증가한다는 것을 알 수 있습니다. 포인터를 배열처럼 사용하여 *(arr+i) == arr[i] 라는 것을 알 수 있습니다.

```
> 포인터 주소 : 402235312 402235344
증가 연산 후 : 402235316 402235352
변수 값 : 20 20.20
증가 연산 후 : 402235324 402235368
```

# 포인터

```c
#include <stdio.h>

int main(){
  int num = 10;
  int *ptr;
  int **pptr;

  ptr = &num;
  pptr = &ptr;

  printf("num : %d *ptr : %d **ptr : %d\n", num, *ptr, **pptr);
  printf("num 주소 : %d *ptr 값 : %d **ptr 값 : %d\n", &num, ptr, *pptr);
  printf("ptr 주소 : %d pptr 값 : %d", &ptr, pptr);

  return 0;
}
```

```
> num : 10 *ptr : 10 **ptr : 10
num 주소 : 1346832580 *ptr 값 : 1346832580 **ptr 값 : 1346832580
ptr 주소 : 1346832584 pptr 값 : 1346832584
```

이중 포인터는 포인터의 주소 값을 담는 변수로, 포인터의 포인터라고 할 수 있습니다.

옆 코드를 보시면 ptr에는 num의 주소 값을 대입, pptr에는 ptr의 주소값을 대입했습니다.

이중 포인터는 그 포인터가 가리키고 있는 곳에서 한번 더 그 포인터가 가리키는 주소로 찾아가 그 변수 값을 사용합니다.

그래서 첫번째 출력문에 num, ptr, pptr 세 값이 모두 같게 되는 것입니다.

# 포인터 (문제풀이)

```c
#include <stdio.h>

int main()
{
    int arr[5] = { 1, 3, 5, 7, 9 };
    double arr2[5] = { 1.1, 3.2, 5.3, 7.4, 9.5 };
    int *arrPtr = arr;
    double *arrPtr2 = arr2;

    (*arrPtr)++;
    *arrPtr2++;

    printf("%d %lf %d %d\n", *arrPtr, *arrPtr2, arrPtr, arrPtr2);


    return 0;
}
```

- 코드를 실행했을 때 출력 값은?

(arr의 주소 값은 1000, arr2 2000이라고 가정)

풀이

(*arrPtr)++;은 가리키는 값을 1증가

*arrPtr2++;은 주소 값을 1증가

*arrPtr = 2

*arrPtr2 = 3.200000  //lf로 출력

arrPtr = 1000

arrPtr2 = 2008

답 : 2 3.200000 1000 2008

# 함수

```c
#include <stdio.h>

int arrayPlus(int arr[], int length) {
    int i;
    int result = 0;

    for(i = 0; i < length; i++) {
        result += arr[i];
    }

    return result;
}

int main() {

    int arr[5] = { 1, 2, 3, 4, 5 };
    int result;
    int length = sizeof(arr) / sizeof(int);

    result = arrayPlus(arr, length);

    printf("모든 배열의 합 : %d", result);

    return 0;
}
```

```
> 모든 배열의 합 : 15
```

함수란 특정한 기능을 따로 분리 해 놓은 것으로 수학에서 사용하는 함수의 개념과 비슷합니다. 함수를 사용하면 유지보수와 가독성을 높일 수 있으며 코드를 재활용 할 수 있다는 장점이 있습니다.

arrayPlus라는 함수를 호출하고 (arr, length)를 인자로 넘겨 주었습니다. 인자(매개변수)란, 함수를 호출할 때 넘겨주는 값입니다. 인자를 사용하는 이유는 전역변수와 지역변수가 존재하여 다른 함수의 변수는 사용할 수 없기 때문입니다. 이때 주의할 점은 넘겨받은 인자는 매개변수에 복사된다는 것입니다. 즉 함수에서 넘겨받은 값을 변경해도 원래의 그 값은 변경되지 않습니다.

# 함수

```
[반환형] [함수명] (인자 목록){

   [호출 시 작동될 함수 내부 코드]

}
```

```c
int arrayPlus(int arr[], int length) {
  int i;
  int result = 0;

  for(i = 0; i < length; i++) {
    result += arr[i];
  }

  return result;
}
```

함수의 형태는 예시와 같습니다. 예제 함수 경우 반환형은 int, 함수명은 arrayPlus, 넘길 인자는 배열(arr), 배열의 길이(length)입니다. 호출시 작동된 내부 코드는 중괄호{}로 둘러쌓인 모든 코드를 의미합니다.

ArrayPlus의 경우 반환형이 int입니다. 정수를 반환한다는 뜻입니다. 반환형이 있는 함수는 return이 필수적이며 return은 함수의 종료와 값을 반환하는 역할을 합니다.

반환형이 없는 함수는 void라는 반환형을 사용하고 return은 사용하지 않습니다.

또한 C언어는 절차지향언어임으로 함수의 선언이 main함수 아래에 있다면 인식하지 못합니다.

# 함수

```c
#include <stdio.h>

int func1(void);

int main() {

    func1();

    printf("함수 실행 완료\n");

    return 0;
}

int func1(void) {
    printf("예시 함수입니다.\n");
    return 0;
}
```

```
> 예시 함수입니다.
함수 실행 완료
```

main 함수 아래에 함수를 작성하려면 함수 원형을 작성하면 됩니다. 예시와 같이 함수의 코드는 빼고, 원형만 위에 선언해 주면 됩니다. main 함수 위에 전부 정의해도 되지만 이렇게 두번이나 정의하는 이유는 코드가 길어지고 함수가 증가하게 되면 프로그램의 전체적인 형태를 볼 수 있는 main 함수가 밑으로 내려가고 코드를 보는 것이 불편함을 느끼기 때문입니다.

코드가 짧다면 main 함수 위에 선언해도 상관없지만, 코드가 길어진다면 함수원형을 필요로 할 것입니다.

# 함수 (문제)

```c
#include <stdio.h>

double average( 인수 작성 ) {
  /* 평균을 구하는 코드를 작성해주세요 */
}

int main() {
  double my_average;
  코드작성

  printf("평균 점수는 다음과 같습니다 : %.1f\n",  my_average);

  return 0;
}
```

평균을 구하는 함수를 만들어서 사용한 후,
출력하는 프로그램을 작성하세요

- scanf를 이용하여 사용자에게 입력 받는다. 입력 받는 값은 정수이며 총 4개이다.
- 평균을 구하는 average 함수를 구현한다. 평균 점수는 소수점 이하까지 구한다.
- 입력 받은 값들은 함수로 넘기고 평균을 구해 반환한다. 함수의 반환 값은 my_average 변수에 저장한다.
- 반환 받은 값은 출력한다. 이때 나머지는 소수점 이하 한 자리까지만 출력한다.

# 함수 (문제풀이)

```c
#include <stdio.h>

double average(int n[]) {
 double avr = 0;
  for(int i = 0; i<4; i++)
  {
    avr += n[i];
  }
  avr = (double)avr/4;
  return avr;
}

int main() {
  double my_average;
  int n[4];
  for(int i = 0; i < 4; i++)
  {
    scanf("%d", &n[i]);

  }
  my_average = average(n);

  printf("평균 점수는 다음과 같습니다 : %.1f\n",  my_average);

  return 0;
}

```

- 4자리 배열을 생성한다.
- main함수에서 for문을 이용해 4번 반복하여 scanf로 배열에 숫자 4개를 입력 받는다.
- my_averag로 넘김과 동시에 average()함수를 n을 인자로 하여 호출한다.
- 호출 받은 average함수는 받은 인자 값을 for 문을 통해 avr변수에 저장한다.
- for문 종료 후 avr을 4로 나눠 doubl형으로 변환하여 반환한다.
- printf로 평균점수 출력한다.

# 문자열 함수

표 11-1 문자입력 함수 scanf(),getchar(), getche(), getch()의 비교

| 함수 | scanf("%c", &ch) | getchar() | getche()<br>_getche() | getch()<br>_getch() |
|---|---|---|---|---|
| 헤더파일 | stdio.h | | conio.h | |
| 버퍼 이용 | 버퍼 이용함 | | 버퍼 이용 안함 | |
| 반응 | [enter] 키를 눌러야 작동 | | 문자 입력마다 반응 | |
| 입력 문자의 표시(echo) | 누르면 바로 표시 | | 누르면 바로 표시 | 표시 안됨 |
| 입력문자 수정 | 가능 | | 불가능 | |

- scanf()와 getchar()함수는 라인 버퍼링 방식을 사용 하여 문자 하나를 입력해도 반응이 없다가 [enter] 키를 누르면 그제서야 이전에 입력한 문자마다 입력이 실행됩니다.
- getche()와 getch()함수는 버퍼를 하용 하지 않고 문자 하나를 바로 입력할 수 있는 함수입니다. 함수를 이용하려면 헤더파일 conio.h를 삽입해야 합니다.

# 문자열 관련 함수

**strlen() :** NULL문자를 제외한 문자열 길이를 출력해주는 함수입니다.

**strcpy(), strncpy() :** 문자열을 복사하는 함수이다. strcpy()함수는 앞 인자 문자열에 뒤 인자 문자열을 복사합니다. 문자열은 항상 마지막 NULL문자까지 포함 한다.

**strcat(), strncat() :** 앞 문자열에 뒤 문자열의 NULL 문자까지 연결하여, 앞의 문자열 주소를 반환하는 함수입니다. strncat()의 경우 덧붙일 문자열의 크기를 지정할 수 있습니다.

**strtok() :** 문자열에서 구문자(delimiter)인 문자를 여러 개 지정하여 토큰을 추출하는 함수입니다.

**strcmp, strncmp :** 문자열 비교와 복사, 그리고 문자열 연결 등과 같은 다양한 문자열 처리는 헤더파일 string.h에 함수원형으로 선언된 라이브러리 함수로 제공됩니다.

# 전역변수와 지역변수

```c
#include <stdio.h>

int functionTest() {
  int temp = 5;
  temp += result;
  return temp;
}

int main() {
  int result = 10;

  printf("result 결과 : %d", functionTest());

  return 0;
}
```

에러 발생

```
Main.c:5:10: error: ‘result’ undeclared (first use in this function)
    5 |  temp += result;
      |          ^~~~~~
Main.c:5:10: note: each undeclared identifier is reported only once for
 it appears in
```

지역변수란 한 지역에서만 사용할 수 있는 변수 입니다. main 함수내의 변수들은 다른 함수내에서 사용할 수 없습니다. 다른 함수에서 사용하기 위해선 인자 값으로 넘겨주거나, 전역변수로 선언한 후 사용해야 합니다. 옆 예시를 보면 result가 선언되지 않았다며 에러가 났습니다. 위와 같이 main 함수 내에서 선언된 변수는 functionTest같은 다른 함수에서는 사용할 수 없습니다. 물론 반대의 경우로 temp변수도 main 함수 내에서는 사용할 수 없습니다.
지역변수는 초기화를 하지 않으면 쓰레기 값이 저장됨을 주의 해야 합니다. 또한 지역변수는 스택 메모리 영역에 할당되며 함수나 블록이 종료되는 순간 메모리에서 자동 제거 됩니다. 지역변수 선언에서 자료형 앞에 키워드 auto가 사용되며 생략 가능합니다.

# 전역변수와 지역변수

```c
#include <stdio.h>

int global = 10;

void globalTest() {
  global += 5;
  printf("함수에서 전역 변수 : %d\n", global);

}

int main() {
  int result = 10;
  printf("전역변수 : %d\n", global);
  printf("지역변수 : %d\n", result);

  globalTest();

  return 0;
}
```

```
지역변수 : 10
함수에서 전역 변수 : 15
```

전역변수는 어느 지역에서나 사용할 수 있는 변수 입니다. 괄호 안에 쓴 변수가 지역변수 였다면 괄호 밖에서 쓴 변수는 전역변수 입니다. 전역변수로 선언하면 main 함수이든, functionTest 함수이든 변수 사용이 가능합니다. 또한 이 전역변수는 프로그램의 시작과 동시에 메모리 공간에 할당되어 프로그램이 종료 될 때 까지 존재합니다. 예시를 보면 전역변수 global이 두 함수에 접근 하는 것을 볼 수 있습니다. 다만 전역변수는 코드가 길어지고 복잡해지면 어떤 함수에서 값을 바꾸는 것을 알기 어렵기 때문에 전역변수가 꼭 필요한 것이 아니라면 지역변수를 사용하는 것을 권장합니다. 전역변수를 다른 파일에서 참조하려면 키워드 extern을 사용하여 전역변수임을 선언해야 합니다.

# 정적변수

```c
#include <stdio.h>

void increaseNumber()
{
    static int num1 = 0;     // 정적 변수 선언 및 값 초기화

    printf("%d\n", num1);    // 정적 변수 num1의 값을 출력

    num1++;    // 정적 변수 num1의 값을 1 증가시킴
}

int main()
{
    increaseNumber();    // 0
    increaseNumber();    // 1
    increaseNumber();    // 2
    increaseNumber();    // 3: 정적 변수가 사라지지 않고 유지되므로 값이 계속 증가함

    return 0;
}
```

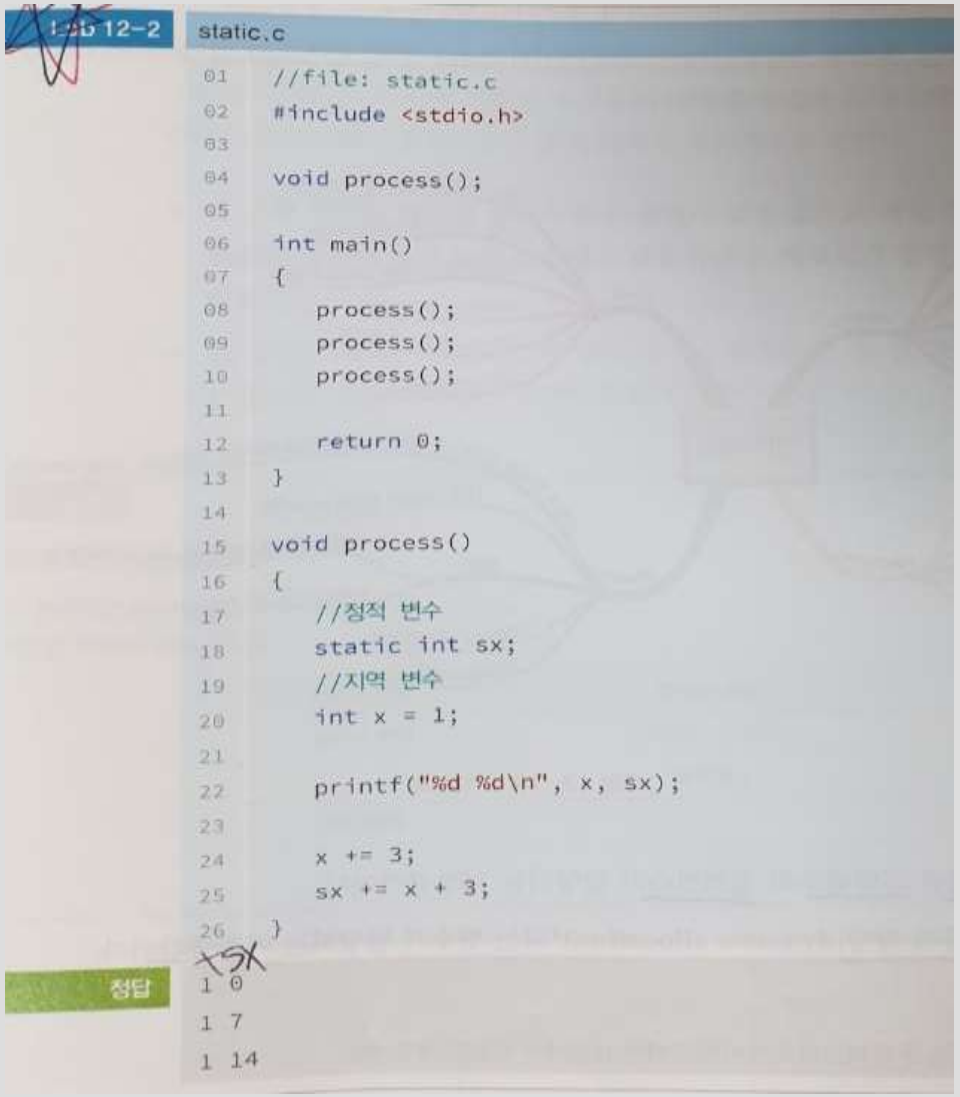
실행 결과

```
0
1
2
3
```

## 키워드 static

변수 선언에서 자료형 앞에 키워드 static을 넣어 정적변수(static variable)를 선언할 수 있다.
정적변수는 메모리에서 제거되지 않으므로 지속적으로 저장 값을 유지하고 수정할 수 있다.
프로그램이 종료하면 메모리에서 제거가 됩니다.
정적변수는 초기값을 지정하지 않으면 자료형으로 0이나 '\0'또는 NULL값이 저장됩니다. 초기화는 단 한번만 수행할  수 있습니다. 예시 코드를 보면 함수를 여러 번 호출했음에도 값이 초기화 되지 않고 static으로 선언한 num1값에 값이 계속적으로 증가함을 볼 수 있습니다.

# 정적변수 (문제풀이)



예제 12-2 static.c

```
01  //file: static.c
02  #include <stdio.h>
03
04  void process();
05
06  int main()
07  {
08      process();
09      process();
10      process();
11
12      return 0;
13  }
14
15  void process()
16  {
17      //정적 변수
18      static int sx;
19      //지역 변수
20      int x = 1;
21
22      printf("%d %d\n", x, sx);
23
24      x += 3;
25      sx += x + 3;
26  }
```

정답

```
1 0
1 7
1 14
```

sx는 정적변수, x는 지역변수로 선언

process() 1회

sx = 0　　x=1로 초기화

printf()로 x sx 출력

x +=3; sx += x+3;

process() 2회

sx 정적변수로 초기화 안됨

x 지역변수로 1로 초기화

x +=3; sx += x+3;

printf()로 x sx 출력

process() 3회

위와 같이 x만 초기화됨

| x | sx |
|---|---|
| 1 | 0 |
| 4 | 7 |
| 1 | 7 |
| 4 | 14 |
| 1 | 14 |

# 구조체

```c
#include <stdio.h>

struct student {
  char name[15];
  int s_id;
  int age;
  char phone_number[14];
};

int main(){
  struct student goorm;

  printf("이름 : ");
  scanf("%s", goorm.name);
  printf("학번 : ");
  scanf("%d", &goorm.s_id);
  printf("나이 : ");
  scanf("%d", &goorm.age);
  printf("번호 : ");
  scanf("%s", goorm.phone_number);

  printf("이름 : %s 학번 : %d 나이 : %d 번호 : %s\n", goorm.name, goorm.s_id, goorm.age, goorm.phone_number);

  return 0;
}
```

구조체란, 하나 이상의 변수를 묶어서 좀더 편리하게 사용할 수 있도록 도와주는 도구 입니다. 즉 연관성 있는 서로 다른 개별적의 자료형의 변수들을 하나의 단위로 묶은 새로운 자료형을 구조체라 합니다. 구조체는 대표적인 유도 자료입니다. 기존 자료형으로 새로이 만들어진 자료형을 유도 자료형이라 합니다.

구조체를 사용하려면 먼저 구조체 틀을 정의해야 합니다. 키워드 struct 다음에 구조체 태그 이름을 기술하고 중괄호를 이용해 원하는 멤버를 여러 개의 변수로 선언합니다. 구조체 정의는 변수의 선언과는 다르며 새로운 구조체 자료형을 정의하는 구문입니다. 문장은 세미콜론으로 종료하며 초기값을 대입할 수 없습니다. 또한 멤버의 이름은 모두 유일해야 합니다.

# 구조체

```c

#include <stdio.h>

struct student {
  int age;
  char phone_number[14];
  int s_id;
};

int main(){
  struct student goorm = { .age = 20, .phone_number = "010-1234-5678" };
  struct student codigm = { 20, "010-1234-5678", 1001 };
  
  printf("나이 : %d 번호 : %s 학번 : %d\n", goorm.age, goorm.phone_number, goorm.s_id);
  printf("나이 : %d 번호 : %s 학번 : %d\n", codigm.age, codigm.phone_number, codigm.s_id);

  return 0;
}
```

```
> 나이 : 20 번호 : 010-1234-5678 학번 : 0
나이 : 20 번호 : 010-1234-5678 학번 : 1001
```

구조체 멤버의 값을 main에서 선언할 때 대입해서 초기화 할 수 있습니다. 초기화 할 때는 멤버 연산자 . 와 중괄호를 사용합니다. 구조체는 배열처럼 멤버 전체를 초기화 할 수 있고, 원하는 변수만 초기화 할 수도 있습니다.

초기화 할 때에는 {.멤버이름=값} 과 같은 형태로 초기화 할 수도 있으며, 멤버이름을 적지않고 초기화 할 수 도 있습니다.

멤버 이름을 적지 않을 때는 구조체를 정의했던 순서대로 값이 들어갑니다. 또한 값을 따로 넣어주지 않은 멤버는 0으로 초기화 됩니다.

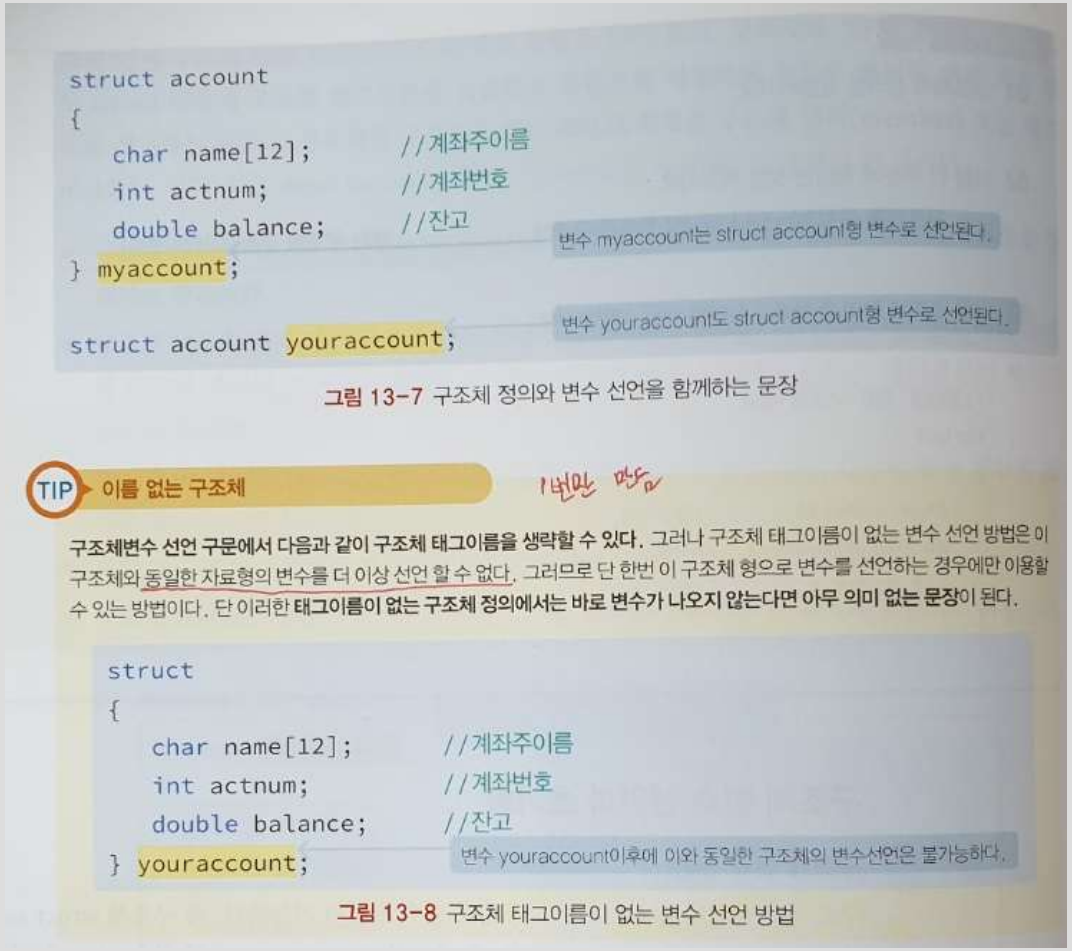
# 자료형 재정의

```c
#include <stdio.h>

typedef struct _Student {
  int age;
  char phone_number[14];
} Student;

int main(){
  Student goorm;

  printf("나이 : ");
  scanf("%d", &goorm.age);
  printf("번호 : ");
  scanf("%s", goorm.phone_number);

  printf("----\n나이 : %d\n번호 : %s\n----", goorm.age, goorm.phone_number);

  return 0;
}
```

## typedef 구문

typedef는 이미 사용되는 자료 유형을 다른 새로운 자료형 이름으로 재정의할 수 있도록 하는 키워드 입니다.
일반적으로 자료형을 재정의 하는 이유는 프로그램의 시스템 간 호환성과 편의성을 위해 필요합니다.
typedef를 이용하면 구조체를 선언할 때 매번 struct를 써줄 필요가 없습니다. 이 typedef를 사용할 때에는 구조체 별칭을 구조체 정의할 때 중괄호 뒤에 써줍니다.
예시와 같이 typedef와 별칭을 써주면 main 함수에서 struct [구조체이름]을 써줄 필요 없이 별칭만 써도 구조체 선언이 가능합니다. 구조체 별칭은 구조체 이름과 동일하게 써도 무관하지만 일반적으로 구조체 이름 앞에 _를 붙여줍니다.

# 익명 구조체



```
struct account
{
   char name[12];        //계좌주이름
   int actnum;           //계좌번호
   double balance;       //잔고
} myaccount;

struct account youraccount;
```

변수 myaccount는 struct account형 변수로 선언된다.

변수 youraccount도 struct account형 변수로 선언된다.

그림 13-7 구조체 정의와 변수 선언을 함께하는 문장

TIP 이름 없는 구조체

1번만 만듦

**구조체변수 선언 구문에서 다음과 같이 구조체 태그이름을 생략할 수 있다.** 그러나 구조체 태그이름이 없는 변수 선언 방법은 이 구조체와 동일한 자료형의 변수를 더 이상 선언 할 수 없다. 그러므로 단 한번 이 구조체 형으로 변수를 선언하는 경우에만 이용할 수 있는 방법이다. 단 이러한 **태그이름이 없는 구조체 정의에서는 바로 변수가 나오지 않는다면 아무 의미 없는 문장**이 된다.

```
struct
{
   char name[12];        //계좌주이름
   int actnum;           //계좌번호
   double balance;       //잔고
} youraccount;
```

변수 youraccount이후에 이와 동일한 구조체의 변수선언은 불가능하다.

그림 13-8 구조체 태그이름이 없는 변수 선언 방법

```c
#include <stdio.h>

typedef struct {
  int age;
  char phone_number[14];
} Student;

int main(){
  Student goorm;

  printf("나이 : ");
  scanf("%d", &goorm.age);
  printf("번호 : ");
  scanf("%s", goorm.phone_number);

  printf("----\n나이 : %d\n번호 : %s\n----", goorm.age, goorm.phone_number);

  return 0;
}
```

예시처럼 구조체 이름을 적지않고 사용하는 것이 가능합니다.

이렇게 구조체 이름을 다로 지정하지 않고 별칭만 사용하는 것을 익명 구조체라고 합니다.

# 함수와 포인터

```c
#include <stdio.h>

void swapNumber(int *first, int *second)    // 반환값 없음, int 포인터 매개변수 두 개 지정
{
    int temp;    // 임시 보관 변수

    // 역참조로 값을 가져오고, 값을 저장함
    temp = *first;
    *first = *second;
    *second = temp;
}

int main()
{
    int num1 = 10;
    int num2 = 20;

    swapNumber(&num1, &num2);        // &를 사용하여 num1과 num2의 메모리 주소를 넣어줌

    printf("%d %d\n", num1, num2);   // 20 10: swapNumber에 의해서 num1과 num2의 값이 서로 바뀜

    return 0;
}
```

실행 결과

```
20 10
```

C언어에선 함수 외부의 변수를 함수 내부에서 수정할 수 없기 때문에 포인터를 매개변수로 사용하여 함수로 전달된 실인자의 주소를 사용하여 그 변수를 참조합니다. 이와 같이 함수에서 주소의 호출을 참조에 의한 호출 (call by reference)이라 합니다.

main함수 내부에 있는 변수num1, num2를 포인터 변수 *first와 *second로 받아 swapNumber()함수에서 값을 바꾸는 것을 볼 수 있습니다.

# 소감

# 소감

저는 이 포트폴리오를 제작하면서 처음 구매했던 책을 펼쳐보게 되었습니다. 제가 전에 밑줄 치고 메모하면서 공부 했던 것들을 보면서 과거를 되돌아 보는 계기가 되었고, 이 계기로 흐트러진 현재의 다짐을 다시금 바로잡을 수 있었습니다. 또한 C언어 책을 처음부터 정독 하니 얼핏 알고 있었던 부분들과 모르는 부분들을 되짚어 볼 수 있었습니다. C언어 뿐만 아니라 다른 프로그래밍언어도 이번 기회를 통해 복습하면서 정리해본다면 앞으로 공부를 하고 취업을 하는데 있어서도 큰 도움이 될 것임을 느끼게 되었습니다. 새로이 시작하는 마음을 갖게 되어 좋은 경험이 되었음을 새기며 포트폴리오 마치겠습니다.

# 감사합니다.