Xilinx Zynq FPGA, TI DSP, MCU기반의 프로그래밍 및 회로 설계 전문가 과정

강사 - Innov (이상훈)
gcccompil3r@gmail.com
학생 - 이유성
dbtjd1102@naver.com

문제 -단 한 번의 연산으로 대소문자 전환을 할 수 있는 연산에 대해 기술하시오

```
#include <stdio.h>
int main(void)
{
 int chr1 = 'A';

 printf("소문자는 = %c\n", chr1^32);

 return 0;
}
```

문제 -임의의 값 x가 있는데, 이를 4096 단위로 정렬하고 싶다면 어떻게 해야할까 ?

문제 -int p[7] 와 int (*p)[7] 가 있다.
내용
이 둘의 차이점에 대해 기술하시오.

int형 데이터 7개를 저장할 수 있는 배열 p

int(*p)[7] = int*[7]p 이며 int형 7개 28byte저장 할 수 있는 배열 포인터p
==>배열을 가르킬 수 있다.

문제 -다음을 C 언어로 구현하시오.
내용
이번 문제의 힌트를 제공하자면 함수 포인터와 관련된 문제다.

아래와 같은 행렬을 생각해보자!

1 2 3

1 2 3

sapply(arr, func) 이라는 함수를 만들어서 위의 행렬을 아래와 같이 바꿔보자!

sapply 에 func 함수가 연산을 수행하는 함수로 만들어야 한다.

1 2 3

1 4 9

문제- 이것이 없으면 C 언어의 함수를 호출할 수 없다.

문제 -아래 질문에 대해 답하시오.
내용
void (* signal(int signum, void (* handler)(int)))(int)라는 signal 함수의 프로토타입을 기술하시오.

프로토타입을 기술하라는 의미는 반환형(리턴 타입)과 함수의 이름, 그리고 인자(파라메터)가 무엇인지 기술하라는 뜻임.

리턴 : void (*) (int) <ㅡ함수포인터
함수명 signal
인자 : int signum 과 void(*handler)(int)

문제 -기본 for 문 활용 문제다.
내용
1 ~ 100 까지의 숫자중 홀수만 더해서 출력해보시오

```
#include <stdio.h>
int odd_num(int i){
 int res = 0;
 for(i=1; i <=100; i++){
```

```
  if(i%2)
  res = i + res;

 }
 return res;

}


int main(void) {
 int x ;
 x = odd_num();
 printf("1~100까지 숫자중 홀수만 더해서 출력한 값 = %d",x);
 return 0;
}
```

문제 -기본 배열 문제다.
내용
1 ~ 100 까지 숫자를 모두 더해서 첫 번째 배열에 저장하고

1 ~ 100 까지 숫자중 홀수만 더해서 두 번째 배열에 저장하고

1 ~ 100 까지 숫자중 짝수만 더해서 세 번째 배열에 저장한다.

다음으로 1 ~ 100 까지 숫자중 3 의 배수만 더해서 네 번째 배열에 저장한다.

각 배열의 원소를 모두 더해서 결과값을 출력하시오.

```
#include <stdio.h>
int sum(){

        int res1 = 0 ;

        for(int i =1; i <=100 ; i++){

                res1 = res1 + i;
        }

        return res1;
}

int sum_odd(){

        int res2 = 0 ;

        for(int j =1 ;j<=100; j++){
                if(j%2)

                 res2 = res2 + j;
                }
                return res2;
                }

int sum_even(){

        int res3 = 0;
        for(int k = 1; k <=100; k++){
                if(!(k%2))

                res3 = res3 +k;
                }
```

```
        return res3;
        }

int sum_mult(){

        int res4 = 0 ;
        for(int l=1;l <=100; l++){

                if(!(l%3))

                res4 = res4 + l;
        }
        return res4;
}



int main(void) {

        int res = 0;
        int arr[4];
        arr[0] = sum();
        arr[1] = sum_odd();
        arr[2] = sum_even();
        arr[3] = sum_mult();


        for(int a = 0 ; a < 4 ; a++){

                res = res + arr[a];
         }

        printf("%d", res );

        return 0;
```

}

문제 -질문에 대해 기술하시오.
내용
C 언어에서 중요시하는 메모리 구조에 대해 기술하시오.

(힌트: Stack, Heap, Data, Text 에 대해 기술하시오.)

메모리 구조의 기초

메모리공간 이름/ 들어오는 데이터 / 관리방식

Stack 지역 변수가 위치하는 영역 동적

Heap 동적 할당된 메모리들이 위치하는 영역 동적

Data 전역 변수 및 static으로 정적
선언된 것들이 위치하는 영역
초기화 되지 않은 모든것은 0으로 저장됨

코드 Machine Code가 위치하는 영역 정적

변수는 공통적으로 자료가 저장되는 공간

동적:프로그램 실행 중 바뀌는 공간(메모리 할당 및 해제)

정적:프로그램 실행 중 바뀌지 않는 공간

문제 -다음 질문에 대해 답하시오.
내용
goto 를 사용하는 이유에 대해 기술하시오.

goto 문법의 이점

예를 들어 for문을 중복해서 사용하고 if와 break로 for문에서 빠져 나올 때, 상당히 많은 불필요한 명령어들이 사용된다.

이러한 단점을 보오나하기 위해 goto 문법을 사용하면 불필요한 코드를 사용하지 않고 한 번에 for문을 빠져나올 수 있다.

if(swtich/goto,continue포함,)와 break를 사용하면 기본적으로 mov,cmp,jmp를 해야한다.

하지만 goto는 jmp하나로 끝난다.

또한, for문과 if,breack를 여러 개 조합을 할수록 mov,cmp,jmp가 늘어난다. 여기서 문제는 jmp임

call 이나 jmp를 cpu instruction레벨에서 분기 명령어라고 하고 이들은 cpu 파이프라인에 치명적인 손실을 가져다 준다.

기본적으로 분기 명령어는 파이프라인을 부순다.

이뜻은 위의 가장 단순한 cpu가 실행까지 3clock을 소요하는데 기존의 파이프라인이 깨지니

쓸데없이 또 다시 3clock을 버려야 함을 의미한다.

만약 이러한 파이프라인의 단계가 수십 단계라면

여기서 낭비되는 cpu clock이 수없이 많음.

즉, 성능면으로만 보아도 goto가 월등히 좋다는 것을 알 수 있다.

문제 -Stack 및 Queue 외에 Tree 라는 자료구조가 있다.
내용
이 중에서 Tree 는 Stack 이나 Queue 와는 다르게 어떠한 이점이 있는가 ?

stack이나 queue는 선형구조이기 때문에 처음부터 일일이 찾아봐야 해서 시간이 오래걸림

Tree는 비선형구조라 데이터를 찾는 시간이 적게 걸림

문제 -아래 자료 구조를 C 언어로 구현해보시오.
내용
Stack 자료구조를 아래와 같은 포맷에 맞춰 구현해보시오.

(힌트: 이중 포인터)

ex)

```
int main(void)

{

    stack *top = NULL;

    push(&top, 1);

    push(&top, 2);

    printf("data = %d\n", pop(&top));
```

```
#include <stdio.h>
#include <malloc.h>
#define EMPTY 0


struct node{

 int data;
 struct node *link;
};
typedef struct node stack;




stack *get_node(){

 stack *tmp;
 tmp = (stack *)malloc(sizeof(stack));
 tmp->link = EMPTY;
 return tmp;
}
```

```
void push(stack **top , int data){

 stack *tmp = *top;
 *top = get_node();
 (*top)->data =data;
 (*top)->link = tmp;
}
```

```
int pop(stack **top){

 stack *tmp = *top;
 int num ;

 if(*top ==EMPTY){

 printf("stack is empty!!!\n");
 return ;

 }
  num = (*top)->data;
  *top = (*top)->link;
 free(tmp);
 return num;
}
```

```
int main(void){
```

```
 stack *top =NULL;
 push(&top,1);
 push(&top,2);

 printf("data = %d\n",pop(&top));
 printf("data = %d\n",pop(&top));
 pop(&top);


 return 0;
}
```

문제-다음 질문에 답하시오.
내용
Binary Tree 나 AVL Tree, Red-Black Tree 와 같이 Tree 계열의
자료구조를 재귀 호출 없이 구현하고자 한다.

이 경우 반드시 필요한 것은 무엇인가 ?

Stack

문제 -다음 자료 구조를 C 로 구현하시오.
내용
Binary Tree 를 구현하시오.

초기 데이터를 입력 받은 이후 다음 값이 들어갈 때 작으면 왼쪽 크면 오른쪽으로 보내는 방식으로 구현하시오.

삭제 구현이 가능하다면 삭제도 구현하시오.

```
#include<stdio.h>
#include<malloc.h>

#define EMPTY 0



struct node{

    int data;
    struct node *left;
```

```
    struct node *right;
    };
typedef struct node tree;




tree *get_node()
{
    tree *tmp;
    tmp = (tree *)malloc(sizeof(tree));
    tmp->left =EMPTY;
    tmp->right =EMPTY;
    return tmp;
}




void tree_ins(tree **root, int data)

{
    if(*root ==NULL)
    {
        *root = get_node();
        (*root)->data = data;
        return;
```

```
    }
    else if ((*root)->data > data)
        tree_ins(&(*root)->left,data);
    else if ((*root)->data < data)
        tree_ins(&(*root)->right, data);
}




void print_tree(tree *root)
{
    if(root)
    {
        printf("data = %d, " , root->data);

        if(root->left)                          //여기서 부터 8줄 빼도 상관
없음.
            printf("left = %d, " , root->left->data);
        else
            printf("left = NULL, ");

        if(root->right)
            printf("right = %d\n" , root->right->data);
        else
            printf("right = NULL\n");

        print_tree(root->left);
        print_tree(root->right);
    }
}
```

```
tree *chg_node(tree *root)
{
   tree *tmp=root;

   if(!root->right)
      root = root->left;
   else if(!root->left)
      root=root->right;

   free(tmp);

   return root;
}



tree *find_max(tree *root, int *data)
{
   if (root->right)
      root->right = find_max(root->right, data);
   else
   {
      *data = root->data;
      root = chg_node(root);
   }

   return root;
}
```

```
tree *delete_tree(tree *root, int data)
{

    int num;
    tree *tmp;
    if(root == NULL)
    {
        printf("Not Found\n");
        return NULL;
    }
    else if(root->data >data)
        root->left = delete_tree(root->left, data);
    else if(root->data <data)
        root->right = delete_tree(root->right,data);
    else if(root->left && root->right)
    {
        root->left = find_max(root->left,&num);
        root->data = num;
    }
    else
        root = chg_node(root);
    return root;
}









int main(void)
{
```

```
    int i;
    int data[14] = {50, 45, 73 , 32, 48, 46, 16, 37, 120, 47, 130, 127,
124};

    tree *root =NULL;

    for(i=0; data[i]; i++)
        tree_ins(&root,data[i]);

    print_tree(root);

    delete_tree(root,50);
    printf("After Delete\n");

    print_tree(root);

    return 0;
}
```

문제 -다음 질문에 대해 기술하시오.
내용
AVL 트리는 검색 속도가 빠르기로 유명하다.

Red-Black 트리도 검색 속도가 빠르지만 AVL 트리보다 느리다.

그런데 어째서 SNS 솔루션등에서는 AVL 트리가 아닌 Red-Black 트리를 사용할까 ?

AVL이 빠르긴 하지만 RB트리를 사용 하는 이유 AVL트리은 데이터 삽입,삭제 할 때마다 균형을 맞추기 위해 노드를 재조정 해야 한다.

RB트리는 자료를 이진트리로 구성하여 검색 구조로 삼는 이유는 단순한 구조이면서도 자료의 검색, 삽입, 삭제가 효율적이기 때문이다.

(소규모, 대규모에도 좋다)

문제 -다음 자료구조를 C 로 구현하시오.
내용
AVL 트리를 C 언어로 구현하시오.

AVL 트리는 밸런스 트리로 데이터가 1, 2, 3, 4, 5, 6, 7, 8, 9 와 같이

순서대로 쌓이는 것을 방지하기 위해 만들어진 자료구조다.