HOMEWORK

이름	문지희
날짜	2018/2/26
수업일수	4일차
담당교수	이상훈

목차

1. 디버깅을 하는 이유
2. scope란
3. static
4. do while
5. #define
6. for문
7. 재귀호출
8. goto와 파이프라인
9. 재귀함수
10. 문제 은행
11. fib함수 동작 분석

1. 디버깅을 하는이유

디버깅은 컴파일은 성공적이지만(문법 오류x) 논리적 오류가 존재하는 경우에 수행한다. 이 이외에도 예측치 못한 다양한 문제가 존재할 수 있어 이를 파악하기 위해 디버깅을 하는것이다. 간단히 말해서 프로그램이 동작은 하지만 본인이 원하는 동작과 다른 동작을 하고자 할때 알아보기 위한 것이 디버깅이다.

-디버깅 하는 방법
1. Gcc -g ~.c 를 입력하여 디버그 파일 생성
2. gdb a.out 디버그 파일을 오픈
3. (gdb) b main 메인함수로 브레이크를 걸어준다.
4. r 을 눌러 실행
5. s 는 함수가 있다면 함수 내부로 진입하고 없다면 1줄을 실행하는 것이고, n 은 함수의 존재 유무와 상관 없이 1줄을 진행한다. c는 브레이크 포인트를 만날 때 까지 계속 작업한다.

Ex)
-잘못된 코드

```
#include<stdio.h>

int main(void)
{
        int number=1;
        while(1)
        {
                printf("%d₩n",number);
                number+=number;
                if(number==100)
                break;
        }
        return 0;
}
```

~ 결과
0만이 계속나오게 되는데 이를 원래 원하던 결과 값을 출력하기 위해서는 주황색 소스 두 줄 중 한가지를 바꿔야한다.

2. scope란

{ 로 시작해서 } 로 끝나는 영역이다. 자기 자신의 영역만의 변수를 할당할 수 있다.

예제1)

```
#include<stdio.h>

//int  local_area2=3;
//int  local_area1=2;

int  main(void)
{
        //int  local_area1=2;
        //int  local_area2=3;
        int  global_area=1;

        {
                //int  local_area1=2;

                printf("global_area=%d₩n",global_area);
                printf("local_area1=%d₩n",local_area1);
        }
        {

                //int  local_area2=3;

                printf("global_area=%d₩n",global_area);
                printf("local_area2=%d₩n",local_area2);
        }


        printf("global_area=%d₩n",global_area);
        printf("local_area1=%d₩n",local_area1);
        printf("local_area2=%d₩n",local_area2);

        return  0;
}
```

~

색있는 글씨의 존재에 따라 결과값이 달라짐

연두색의 문장은 main함수 이전에 변수선언된 전역변수 이므로 이 파일의 코드 어디서든 사용이 가능하다. 따라서 결과 값이 잘 출력된다.

주황색의 문장은 main함수 내의 지역변수인데 main함수 내에서는 이 변수들을 사용 가능하기 때문에 결과 값이 잘 출력된다.

하지만 노란색의 두 문장은 스코프 내의 지역변수이기 때문에 스코프 밖을 벗어나게 되면 변수를 사용할 수 없어지게 되고 결과 값이 출력되지 못하고 오류가 발생한다.

예제2)

```
#include<stdio.h>
int glob_val=7;

void add(void)
{
        glob_val +=3;
}
void mult(void)
{
        glob_val *=3;
}
int main(void)
{
        add();
        mult();
        printf("glob_val = %d₩n", glob_val);

        return 0;
}
```

~ 결과
glob_val = 30
glob_val는 전역변수이기 때문에 소스코드 내에서 어디든 사용 가능하다.

3.static

static이란 전체 프로그램의 시작부터 종료까지를 생존 기간으로 하며 동일 기억 장소를 보유한다. 따라서 **전역 변수**들은 모두 정적 변수다
static은 정적 변수의 역할을 하게 되는데 전역 변수와 마찬가지로 데이터 영역에 로드되는데, 지역 변수를 static으로 선언했다면 이 변수를 선언한 함수내에서만 접근 가능하다.
static을 지역변수 앞에 사용하면 강제로 전역변수를 만들게 되는데 변수를 선언한 함수 내에서만 접근이 가능하다.

-예제

```
#include<stdio.h>

void  count_static_value(void)
{
static  int  count=1;
printf("count  =  %d₩n",count);
count++;
}

int  main(void)
{

        int  i;
        for(i=0;i<7;i++)
        count_static_value();
}
```

~결과

```
count  =  1
count  =  2
count  =  3
count  =  4
count  =  5
count  =  6
count  =  7
```

4. continue

해당 케이스는 실행하지 않고 반복을 계속하고 싶을 때 사용한다. NaN과 Inf와 관련이 있다.

예제)

```
#include<stdio.h>

int main(void)
{

        int num=0;

        while(1)
        {
                num++;
                if(num==5)
                        continue;
                printf("%d₩n",num);
                if (num==10)
                        break;
        }
        return 0;
}
```

~결과

```
1
2
3
4
6
7
8
9
10
```

5를 제외한 나머지 숫자가 출력되었다.

//5. do while

do while은 일단 조건이 만족하지 않더라도 1번은 수행하고 싶을 때 사용된다. 궁극적으로 do while을 사용하는 이유는 매크로 확장 때문이다.

예제)

```
#include<stdio.h>

int main(void)

{

	int n=0;

	do
	{
		if(n==5)
		continue;

		printf("%d₩n",n);
		n++;
	}while(n<10);

	return 0;
}
```

~결과

```
0
1
2
3
4
```

6. #define

#define A B : B를 A로 대체한다.

- #define 을 사용하는 이유
반복되는 코드에 고쳐야 하는 값을 손 쉽게 바꾸기 위해 사용된다. #define 을 사용하지 않는다면 많은 값들을 일일히 수정해야하는 번거로움을 방지할 수 있다.

```
#include<stdio.h>

#define test 1000

int main(void)
{
	printf("%d₩n",test);
	return 0;
}
```

~결과
1000

8. for문

초기화, 조건식, 증감식을 한번에 볼수있어 간결하다.
for(초기화;조건부;증감식)
의 형태로 이루어져 있으며 ( )안의 초기화, 조건식, 증감식들은 여러개의 식을 입력할 수 있다.
단 리눅스에서는 초기값 앞에 자료형을 사용하면 오류가 날 수 있다. C에서는 가능하다.
for(int i=0; ~)

예제)

```
#include<stdio.h>

int main(void)

{

        int i,result;
        for(i=0,result='A';i<10;i++,result++)
        {
        printf("%c₩n",result);
        }

return 0;

}
```

~ 결과

```
A
B
C
D
E
F
G
H
I
J
```

*Infinite Loop with for
조건을 막론하고 for문의 조건부가 비어 있으면 언제든지 무한루프가 된다.

예제)

```
#include<stdio.h>

int main(void)

{
        int i, result='A';
        for(;;)
        {
                printf("%c₩n",result);
        }
        return 0;
}
```

~ 결과

```
A
A
A
A
...(무한반복)
```

*Nested Loop
중첩된 루프로서 지금까지 배웠던 반복문들을 중첩시킨 형태이다.

10. goto문

goto를 사용하면 특정한 상황을 아주 간결하게 해결할 수 있다. 주로Kernel에서 많이 사용하며 buffer를 사용하는 곳에서 주로 사용한다.

예제1)

- 오류가 나는 소스코드

```
#include<stdio.h>

int  main(void)
{
        int  i,j,k;
        for(i=0;i<5;i++)
        {

                for(j=0;j<5;j++)
                {
                        for(k=0;k<5;k++)
                        {
                                if((i==2)&&(j==2)&&(k==2))
                                {
                                        printf("Error₩n");
                                }
                                else
                                {
                                        printf("data₩n");
                                }
                        }
                }
        }
return  0;

}
```

~결과

에러가 나도 계속 실행

예제2)

-break를 사용하여 반복문 벗어남

```
#include<stdio.h>

int  main(void)
{
	int  i,j,k,flag;
	for(i=0;i<5;i++)
	{
		for(j=0;j<5;j++)
		{
			for(k=0;k<5;k++)
			{
				if((i==2)&&(j==2)&&(k==2))
				{
					printf("Error₩n");
					flag=1;
				}
				else
				{
					printf("data₩n");
				}
				if(flag)
				{
					break;
				}
			}
			if(flag)
			{
				break;
			}
		}
		if(flag)
		{
			break;
		}
	}
	return  0;

}
```

~결과

error가 난 후 실행종료

예제3)

-goto를 사용하여 반복문 벗어남

```
#include<stdio.h>

int main(void)
{
	int i,j,k;
	for(i=0;i<5;i++)
	{

		for(j=0;j<5;j++)
		{
			for(k=0;k<5;k++)
			{
				if((i==2)&&(j==2)&&(k==2))
				{
					printf("Error₩n");
					goto err_handler;
				}
				else
				{
					printf("data₩n");
				}
			}
		}
	}
	return 0;
	err_handler:
	printf("Goto Zzang!₩n");
return -1;
}
```

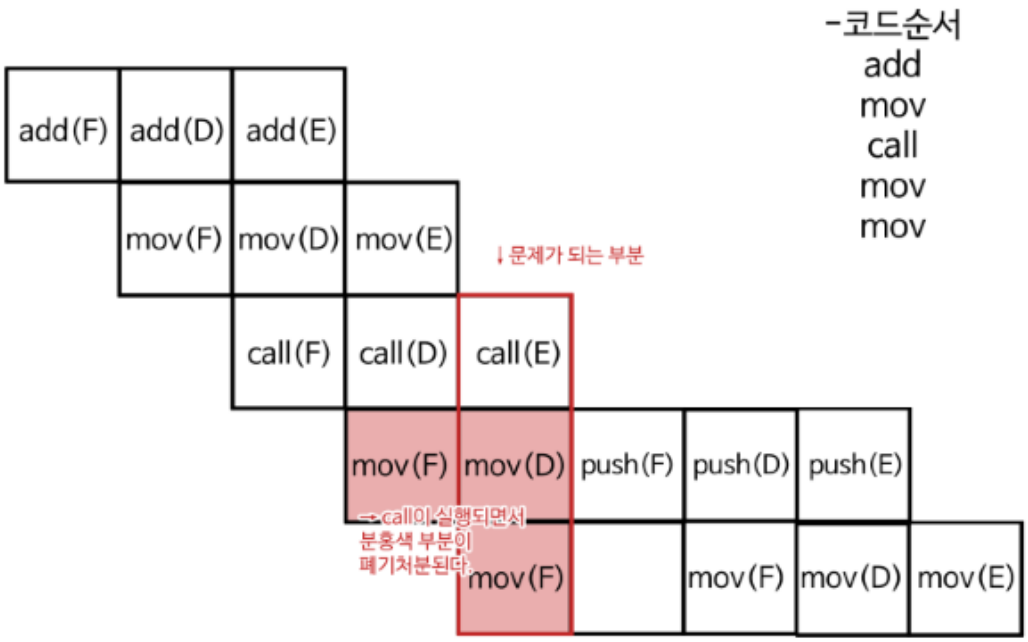
~결과
error나오면 바로 종료된다.

- goto 의 이점과 CPU 파이프라인
if문은 mov, cmp, jmp 로 구성되지만 goto문은 jmp하나로 끝이다. for문이 여러개 생길 때 if와 break의 조합은 for문의 갯수만큼 mov, cmp, jmp 로 구성되는데 jmp명령어에 문제가 생기게 된다.
분기 명령어라고 하는 call이나 jmp는 CPU Instruction(명령어) 레벨에서 파이프 라인에 치명적인 손실을 가져다 준다.

CPU 의 파이프라인을 간단히 설명하면 아래와 같은 3 단계로 구성되는데

1. Fetch - 실행해야할 명령어를 물어옴
2. Decode - 어떤 명령어인지 해석함
3. Execute - 실제 명령어를 실행시킴

아래 그림과 같이 분기가 발생하면 call의 execute부분의 아래라인의 실행중이던 부분은 폐기처분이 되어 CPU clock을 낭비하게 된다. 분기가 여러개 일 수록 더 많은 clock을 낭비하므로 goto문 사용하는 것이 성능면에서 아주 좋다.



11.재귀함수

사용한 함수를 다시 호출하는 방식이다. 프로그램 구현 상 반드시 필요한 경우가 있는데 편의에 따라 성등 등으 잘 고려해서 구현해야한다.

예제)

```
#include<stdio.h>

int fib(int n)
{
	if(n==1||n==2)
		return 1;
	else
		return fib(n-1)+fib(n-2);
}
int main(void)
{
	int r,v;
	printf("피보나치 수열의 항의 개수를 입력하시오 :");
	scanf("%d",&v);
	r=fib(v);
	printf("%d번째 항의 수는 = %d₩n",v,r);
	return 0;

}
```

12. 문재은행

1. 1 ~ 1000사이에 3의 배수의 합을 구하시오.

```
#include<stdio.h>

void f1(int s,int e)
{
	double r=0;

	for (s=1;s<=e;s++)
	{
		if((s%3==0))
		{
			r+=s;
		}
	}

	printf("%lf₩n",r);
}

int main(void)
{
	f1(1,1000);
	return 0;
}
```

~결과

166833.000000

2. 000사이에 4나 6으로 나눠도 나머지가 1인 수의 합을 출력하라.

```
#include<stdio.h>

void f1(int s,int e)
{
	double r=0;

	for (s=1;s<=e;s++)
	{
		if((s%4==1)||(s%6==1))
		{
			r+=s;
		}
	}

	printf("%lf₩n",r);
}

int main(void)
{
	f1(1,1000);
	return 0;
}
```

~결과

166167.000000

3. 1 ~ 110. 구구단을 만들어보시오.

```
#include<stdio.h>

void f1()
{
int r,i,r2=0;

        for(i=0;i<10;i++)
        {

                for(r=0;r<10;r++)
                {
                        r2=r*i;
                printf("%d * %d = %d₩n", r,i,r2);

                }



        }
}

int main(void)
{
        f1(1,9);
        return 0;
}
```

-결과

0 * 0 = 0
1 * 0 = 0
2 * 0 = 0
3 * 0 = 0
4 * 0 = 0
5 * 0 = 0
6 * 0 = 0
7 * 0 = 0
8 * 0 = 0
9 * 0 = 0
0 * 1 = 0
1 * 1 = 1
2 * 1 = 2
3 * 1 = 3
4 * 1 = 4
5 * 1 = 5
6 * 1 = 6
7 * 1 = 7
8 * 1 = 8
9 * 1 = 9
0 * 2 = 0
1 * 2 = 2
2 * 2 = 4
3 * 2 = 6
4 * 2 = 8
5 * 2 = 10
6 * 2 = 12
7 * 2 = 14
8 * 2 = 16
9 * 2 = 18
0 * 3 = 0
1 * 3 = 3
2 * 3 = 6
3 * 3 = 9
4 * 3 = 12
5 * 3 = 15
6 * 3 = 18
7 * 3 = 21
8 * 3 = 24
9 * 3 = 27
0 * 4 = 0
1 * 4 = 4
2 * 4 = 8
3 * 4 = 12
4 * 4 = 16
5 * 4 = 20
6 * 4 = 24
7 * 4 = 28
8 * 4 = 32
9 * 4 = 36
0 * 5 = 0
1 * 5 = 5
2 * 5 = 10
3 * 5 = 15
4 * 5 = 20
5 * 5 = 25
6 * 5 = 30
7 * 5 = 35
8 * 5 = 40
9 * 5 = 45
0 * 6 = 0
1 * 6 = 6
2 * 6 = 12
3 * 6 = 18
4 * 6 = 24
5 * 6 = 30
6 * 6 = 36
7 * 6 = 42
8 * 6 = 48
9 * 6 = 54
0 * 7 = 0
1 * 7 = 7
2 * 7 = 14
3 * 7 = 21
4 * 7 = 28
5 * 7 = 35
6 * 7 = 42
7 * 7 = 49
8 * 7 = 56
9 * 7 = 63
0 * 8 = 0
1 * 8 = 8
2 * 8 = 16
3 * 8 = 24
4 * 8 = 32
5 * 8 = 40
6 * 8 = 48
7 * 8 = 56
8 * 8 = 64
9 * 8 = 72
0 * 9 = 0
1 * 9 = 9
2 * 9 = 18
3 * 9 = 27
4 * 9 = 36
5 * 9 = 45
6 * 9 = 54
7 * 9 = 63
8 * 9 = 72
9 * 9 = 81

13. fib함수 동작 분석

-소스코드

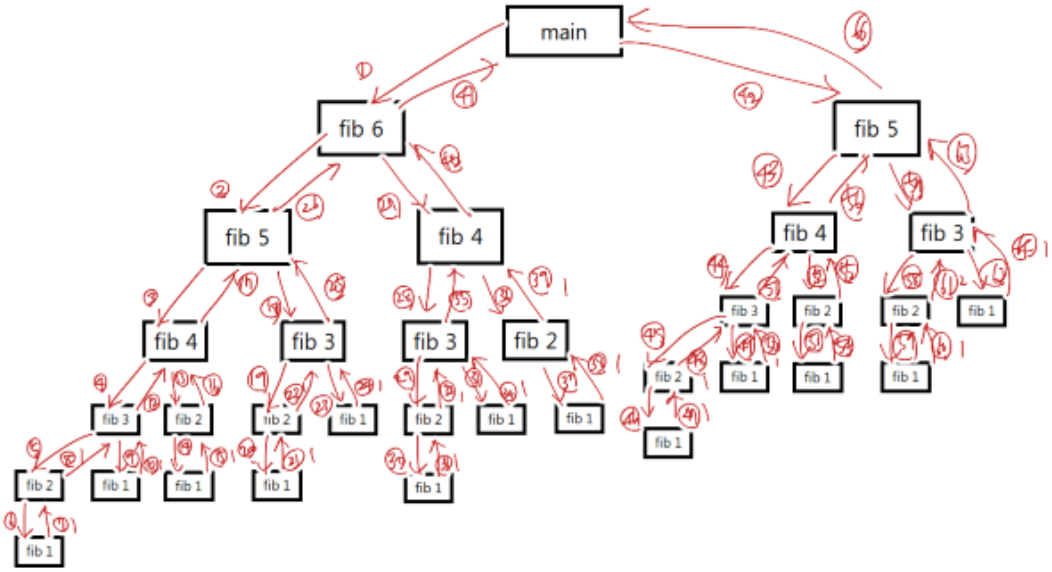
```
#include<stdio.h>
int fib(int n)

{

        if(n==1||n==2)
                return 1;
        else
                return fib(n-1)+fib(n-2);
}
int main(void)
{
        int r,v;
        printf("피보나치 수열의 항의 개수를 입력하시오 :");
        scanf("%d",&v);
        r=fib(v);
        printf("%d번째 항의 수는 = %d₩n",v,r);
        return 0;

}
```

~결과

피보나치 수열의 항의 개수를 입력하시오 : 6

8

디버그의 값은 너무 길어 복사하지 못했다.



위의 사진은 재귀함수가 실행되는 순서를 그림으로 표현한 것이다. fib1과 fib2는 리턴값 1을 반환하고 fib3은2, fib4는3, fib5는 5, fib6은 8을 반환한다.
위의 코드에 따른 피보나치 수열을 나열하면 1 1 2 3 5 8이므로 6번째에 오는 수열은 8이 됨을 알 수 있다.