6장 함수

김명호

내용

- 함수정의
- 함수원형
- 값에 의한 함수 호출
- 지역변수와 전역변수
- 기억영역 클래스
- 수학함수
- Inline 함수
- 재귀함수
- 시간함수

함수

- 단순하지 않는 프로그램을 기능별로 나누어 프로그래밍 하는 기법을 위한 도구
- 함수를 사용한 프로그래밍 이점
  - 하향식 프로그래밍
  - 코드 재사용
  - 디버깅
  - 유지보수
- 프로그램은 하나 이상의 함수로 구성됨
- 프로시저 / 함수

함수 정의

- 함수가 수행할 일을 기술하는 코드
- 함수 정의의 일반적인 형식

```
형 함수_이름(매개변수_목록)        // 함수 헤더
{                                  // 함수 몸체
  선언문
  문장
}
```

함수 정의

- **함수 헤더**

  형 함수_이름(매개변수_목록)

  – 함수_이름 : 함수를 구분함
  – 매개변수_목록 : 함수가 입력 받아 처리하는 데이터 목록
  – 형 : 함수가 리턴하는 값의 형, *함수의 형*이라고 함

- **함수 몸체**
  – 함수가 호출되면 실행될 문장들

함수 정의

프로그램 6.1

```
#include <stdio.h>
int main(void)                    // main() 함수 정의
{
   printf("재미있는 함수\n");
   return 0;
}
```

함수 정의

프로그램 6.1

```
#include <stdio.h>
int main(void)                    // 함수 헤더
{                                 // 함수 몸체
    printf("재미있는 함수\n");
    return 0;
}
```

함수 정의

프로그램 6.1

```
#include <stdio.h>
int main(void)                          // 함수 헤더
```

매개변수 목록

함수이름

함수 형

```
{                                       // 함수 몸체
    printf("재미있는 함수\n");
    return 0;
}
```

함수 정의

함수 6.1

```
int print_it(void)
{
   printf("재미있는 함수\n");
   return 0;
}
```

함수 정의

- 프로그램 6.1의 main() 함수와 print_it() 함수는 이름만 다르고 다 같음
  - 즉 print_it() 함수는 호출되면 프로그램 5.1의 main() 함수와 똑같은 일을 수행함
- main() 함수는 프로그램이 실행되면 자동적으로 호출되어 실행되는 반면 print_it() 함수는 프로그램에서 명시적으로 호출되어야지 실행됨
  - main은 키워드는 아니지만 매우 특별한 식별자임

함수 호출

- 정의되어 있는 함수는 필요할 때 마다 호출하여 사용할 수 있음
- 함수를 호출할 때에는 함수 매개변수에 대응되는 인자를 명시해 줘야 함
- 일반적인 함수 호출 형태

변수 = 함수_이름(인자_목록);

- 변수에는 함수_이름 함수가 리턴 한 값이 배정됨
- 변수의 형은 함수_이름 함수의 형과 같아야 됨
  - 다르면 리턴 값의 형 변환이 일어남

함수 호출

- 함수 매개변수 목록이 void 라는 것은 매개변수가 없음을 뜻함
- 함수 매개변수 목록이 void 일 때의 함수 호출 형태

```
변수 = 함수_이름( );
```

함수 호출

- 함수 형이 void 라는 것은 함수가 리턴 하는 값이 없음을 뜻함
- 함수 형이 void 일 때의 함수 호출 형태

```
함수_이름(인자_목록);
```

  - 함수 형이 void가 아닐 때 이렇게 호출하면 리턴되는 값을 사용하지 않겠다는 의미임
  - 예를 들어 printf()의 함수 형은 int 이지만 그 리턴 값을 보통 사용하지 않기 때문에 위와 같이 호출함

함수 호출

- 함수 형과 매개변수가 모두 void 일 때의 함수 호출 형태

```
함수_이름();
```

함수 호출 예제

- **헤더 : void fun1(void)**

```
fun1();
fun1("hello");      // 오류
a = fun1();         // 오류
```

- **헤더 : double fun2(double x)**

```
fun2(-3.4);
y = fun2(z);
y = fun2(x, z);      // 오류
```

함수 호출

프로그램 6.2

```
#include <stdio.h>
int print_it(void)
{
   printf("재미있는 함수\n");
   return 0;
}
int main(void)
{
   print_it();       // print_it() 함수 호출, 리턴 값 사용 안함
   return 0;
}
```

함수 호출

프로그램 6.3

```
#include <stdio.h>
void print_it(void)          // 리턴 값 없음
{
   printf("재미있는 함수\n");
   return;
}
int main(void)
{
   print_it();
   return 0;
}
```

함수 호출

프로그램 6.3

```
#include <stdio.h>
void print_it(void)
{
    printf("재미있는 함수\n");
    return;
}
int main(void)
{
    print_it();
    print_it();
    print_it();                // 필요한 만큼 호출하면 됨
    return 0;
}
```

매개 변수가 있는 함수

함수 6.2

```
long long power(int n, int m)        // n^m
{
   int   i;
   long long pow = 1;
   for (i = 0; i < m;  ++i)
      pow *= n;
   return pow;
}
```

매개 변수가 있는 함수

- 외부의 값을 입력 받아 일을 해야 할 때 매개 변수가 있는 함수를 만듦
  - power() 함수는 n^m을 계산하는 함수이기 때문에 n과 m을 입력 받아야 함
- 매개 변수의 형이 같다고 다음과 같이 하면 안됨

```
long long power(int n, m)
```

  - 각 매개 변수 별로 형을 지정해야 함

매개 변수가 있는 함수 호출

- 매개 변수가 있는 함수를 호출 할 때에는 매개 변수의 개수와 인자의 개수를 일치시켜야 함
  - 그 수가 같지 않으면 컴파일 오류 발생
- 또한 인자의 형은 대응되는 매개 변수의 형과 일치시켜야 함
  - 형이 일치하지 않으면 컴파일러가 인자의 값의 형을 변환함

매개 변수가 있는 함수 호출

- power() 호출 방법

```
power(a, b);
```

  - power()의 리턴 값 사용하지 않음, 유용하지 않은 호출
  - 만일 a와 b의 형이 int가 아니면 컴파일러는 a와 b의 값을 int 형으로 변환할 것임

```
x = power(3, 5);
```

  - power()의 리턴 값을 x에 배정함

```
printf("%d^%d = %lld\n", a, b, power(a, b));
```

  - power()의 리턴 값을 바로 출력함

묵시적 int

함수 6.3

```
print_it(void)                    // 함수의 형이 정의되어 있지 않음
{
   printf("재미있는 함수\n");
   return 0;
}
```

묵시적 int

- C90에서는 함수의 형이 생략되면 그 함수의 형을 자동적으로 int로 함
  - 함수 6.3의 print_it()은 함수 형이 지정되지 않았지만 컴파일러는 int로 가정함
- C99에서는 묵시적 int가 제거되었음
  - 함수의 형을 반드시 명시해야 함
- 보통의 컴파일러는 묵시적 int를 처리하지만 새 표준은 안임

함수 정의 예제

- 함수 정의
  1. 함수 범위 결정
  2. 함수 헤더
     1. 함수 이름 선택
     2. 매개변수 목록
     3. 함수 형
  3. 함수 몸체

함수 정의 예제

1. 함수 범위 선택

```
#include <stdio.h>
int main(void){
    unsigned long long f;
    int n, i;
1)  printf("계승을 구할 수를 입력 하세요 : ");
    scanf("%d", &n);
    if (n >= 0){
        for (f = i = 1; i <= n; i++)        2)
            f *= i;
        printf("%d! = %llu\n", n, f);
    }
    else
        printf("음수를 입력했습니다.\n");
    return 0;
}
```

함수 정의 예제

2. 함수 헤더 만들기

- 함수 이름, 매개변수 목록, 함수 형 결정
- 함수 이름
  - 함수 기능에 맞는 이름 선택
  - 여러 단어로 이루어지는 함수 이름을 선택한다면 _ 사용할 수 있음
  - 함수 이름 : factorial

함수 정의 예제

2. 함수 헤더 만들기

- 매개변수 목록
  - 함수 안에서 초기화해도 되는 변수와 아닌 변수 구분하여 초기화하면 안 되는 변수를 매개변수로 만듦
  - f, i, n중 n은 초기화하면 안됨
  - 매개변수 목록 : int n

함수 정의 예제

2. 함수 헤더 만들기

- 함수 형
  - 결과 값의 형
  - f가 결과 값을 가지고 있음
  - 함수 형 : unsigned long long
- 함수 헤더

```
unsigned long long factorial(int n)
```

함수 정의 예제

3. 함수 몸체 만들기

- 코드 삽입, 내부 변수 선언, return 문 삽입

```
unsigned long long f;
int   i;
for (f = i = 1; i <= n;  i++)
    f *= i;
return f;
```

함수 정의 예제

함수 6.4

```
unsigned long long factorial(int n)
{
    unsigned long long f;
    int   i;
    for (f = i = 1; i <= n;  i++)
        f *= i;
    return f;
}
```

함수 정의 예제

- **함수 호출**

```
unsingned long long factorial(int n)
C = factorial(10);
factorial(20);
factorial();
X = factorial(n);
X = factorial(x, y);
```

함수 정의 예제

프로그램 6.4

```
#include <stdio.h>
unsigned long long factorial(int n){
   unsigned long long f;
   int   i;
   for (f = i = 1; i <= n;  i++)
      f *= i;
   return f;
}
int main(void){
   int i;
   printf(" i | factorial(i)\n");
   printf("-----------------\n");
   for (i = 1; i <= 10; i++)
      printf("%2d | %9lld\n", i, factorial(i));
   return 0;
}
```

프로그램 결과

```
 i | factorial(i)
----------------
 1 |          1
 2 |          2
 3 |          6
 4 |         24
 5 |        120
 6 |        720
 7 |       5040
 8 |      40320
 9 |     362880
10 |    3628800
```

return

- return 문은 함수를 종료하고 제어를 호출한 환경으로 전달하는 역할을 함
- return 문은 함수의 형이 void이면 수식을 포함하지 않고, void가 아니면 수식을 포함함
- 제어가 넘어갈 때 return 문에 명시된 수식이 평가되어 전달됨
  - 필요하다면, 그 수식의 값은 함수 정의에 명시된 함수의 형으로 변환됨
- 필요에 따라 두 개 이상 있을 수도 있음

return

- **일반적인 형식**

```
return;
return 수식;
```

- **예**

```
return;
return ++a;
return (a * b);
if (error)
    return 1;
else
    return 0;
```

return

함수 6.5

```
unsigned long long factorial(int n)
{
   unsigned long long f;
   int   i;
   if (n < 0)
      return -1;      // n이 음수일 때 -1 리턴
   for (f = i = 1; i <= n;  i++)
      f *= i;         // n이 양수일 때 계승 리턴
   return f;
}
```

return

- C90에서는 함수의 형이 void가 아니더라도 수식이 없는 return 문이 가능함
- C99에서는 함수의 형이 void가 아니면 수식을 반드시 가져야 함

exit()

- 프로그램을 종료시킴
  - 어떤 함수에서 호출되더라도 프로그램은 종료함
- <stdlib.h>를 포함시켜야 함

exit()

함수 6.6

```
unsigned long long factorial(int n)
{
   unsigned long long f;
   int   i;
   if (n < 0) {
      printf("factorial() 함수 호출 오류\n");
      exit(-1);                    // 프로그램 종료
   }
   for (f = i = 1; i <= n;  i++)
      f *= i;
   return f;                       // factorial() 함수 종료
}
```

함수 원형

- 컴파일러는 함수 헤더의 모양과 함수 호출 모양을 비교하여 올바른 함수 호출이 일어나도록 함
  - 매개변수의 수와 인자의 수를 비교하여 다르면 오류 메시지 출력
  - 인자의 형을 매개변수의 형과 일치시킴
  - 함수 형과 함수 값이 배정되는 변수의 형이 다르면 리턴되는 값의 형을 변수 형으로 변환

함수 원형

프로그램 6.5

```
#include <stdio.h>
double power_r(double n, int m){
   int   i;
   double pow = 1.0;
   for (i = 0; i < m;  ++i)
      pow *= n;
   return pow;
}
int main(void){
   int m, n;
   printf("n^m 계산 프로그램\n");
   printf("n : ");
   scanf("%d", &n);
   printf("m : ");
   scanf("%d", &m);
   printf("%d^%d = %.3lf\n", n, m, power_r(n, m));
   return 0;
}
```

프로그램 결과

```
n^m 계산 프로그램
n : 3
m : 4
3^4 = 81.000
```

함수 정의 순서

- main() 함수는 가장 먼저 실행되는 함수이기 때문에 프로그램의 제일 앞이나 제일 뒤에 둠
- main() 함수를 제일 앞에 두는 경우가 많음
  - 가장 먼저 실행되므로 눈에 잘 띄게 하기 위해

함수 원형

프로그램 6.6

```
#include <stdio.h>
int main(void){
   int m, n;
   printf("n^m 계산 프로그램\n");
   printf("n : ");
   scanf("%d", &n);
   printf("m : ");
   scanf("%d", &m);
   printf("%d^%d = %.3lf\n", n, m, power_r(n, m));
   return 0;
}
double power_r(double n, int m){
   int   i;
   double pow = 1.0;
   for (i = 0; i < m;  ++i)
      pow *= n;
   return pow;
}
```

프로그램 결과

```
n^m 계산 프로그램
n : 3
m : 4
3^4 = 0.000
```

함수 원형

- 함수 정의가 함수 호출 문장보다 뒤에 있을 때 사용
- 컴파일러에게 함수로 전달되는 인자의 수와 형 그리고 함수의 리턴 값의 형을 알려줌
- 일반적인 형식

  형 함수_이름(매개변수_형_목록);
- 예제

```
double power_r(double, int);
double power_r(double n, int m);
double power_r(double x, int y);
```

  - 모두 같은 함수 원형임

함수 원형

프로그램 6.7

```
#include <stdio.h>
double power_r(double, int);
int main(void){
   int m, n;
   printf("n^m 계산 프로그램\n");
   printf("n : ");
   scanf("%d", &n);
   printf("m : ");
   scanf("%d", &m);
   printf("%d^%d = %.3lf\n", n, m, power_r(n, m));
   return 0;
}
double power_r(double n, int m){
   int   i;
   double pow = 1.0;
   for (i = 0; i < m;  ++i)
      pow *= n;
   return pow;
}
```

프로그램 결과

```
n^m 계산 프로그램
n : 3
m : 4
3^4 = 81.000
```

함수 매개변수 전달 방법

- 값에 의한 호출
- 참조에 의한 호출

값에 의한 호출

- C에서는 값에 의한 호출로 함수를 호출함
  - 함수 호출에 명시된 인자는 그 값만 함수에 전달됨
  - 호출된 곳에서 값이 변해도 호출한 곳의 인자에는 영향을 안줌

값에 의한 호출

프로그램 6.8

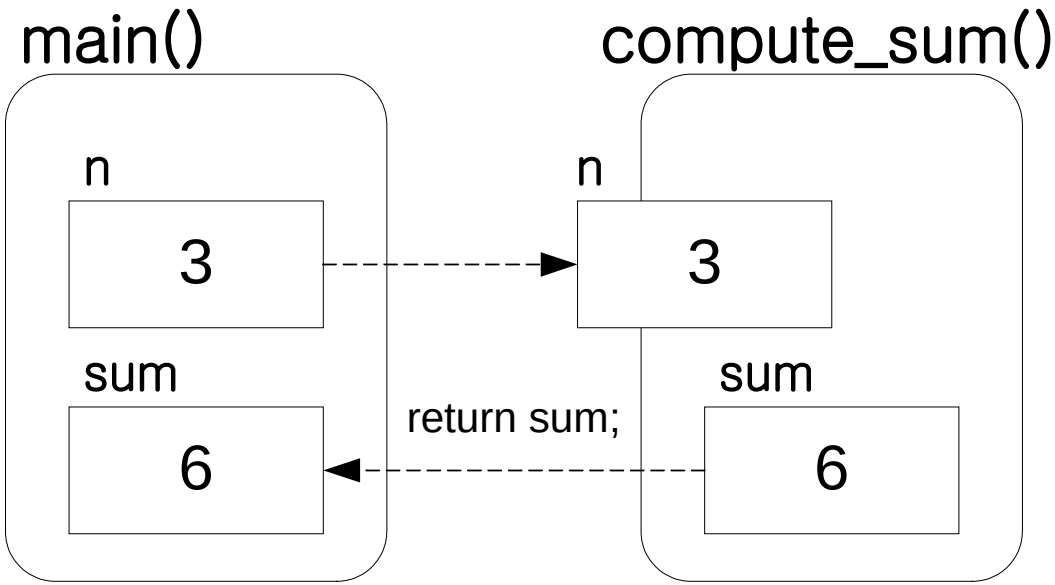
```
#include <stdio.h>
int   compute_sum(int);
int main(void)
{
   int   n = 3, sum;
   printf("함수 호출 전 : n = %d\n", n);
   sum = compute_sum(n);
   printf("함수 호출 후 : n = %d\n", n);
   printf("합 = %d\n", sum);
   return 0;
}
int compute_sum(int n)
{
   int sum = 0;
   for ( ; n > 0; --n)     // n의 값이 변함
      sum += n;
   return sum;             // n == 0
}
```

프로그램 결과

```
함수 호출 전 : n = 3
함수 호출 후 : n = 3
합 = 6
```

값에 의한 호출

- main() 함수와 compute_sum() 함수를 위한 메모리 내용
  - main()의 n과 compute_sum()의 n은 독립적



값에 의한 호출

- 매개변수의 이름을 바꿔도 똑같은 함수임
  - 프로그램 5.8의 compute_sum()을 다음과 같이 수정해도 됨

함수 6.7

```
int compute_sum(int x)
{
    int s = 0;
    for ( ; x > 0; --x)
        s += x;
    return s;
}
```

유효범위 규칙

- 변수의 사용 범위를 정의함
- 기본적인 유효범위 규칙
  - 식별자의 유효범위는 그 식별자가 선언된 곳부터 시작하여 식별자가 선언된 블록의 마지막 부분까지
  - 그 블록을 벗어나면 사용될 수 없음
- 외부 블록에 선언된 변수는 내부 블록에서 같은 이름으로 다시 선언되지 않는 한, 내부 블록에서도 유효함
  - 만일 같은 이름으로 내부 블록에 선언된다면, 외부 블록 변수는 내부 블록에서는 사용할 수 없음

유효범위 규칙

- 외부 블록 이름은 내부 블록이 그것을 다시 정의하지 않는 한, 내부 블록에서도 유효함
- 만일 다시 정의된다면, 외부 블록 이름은 내부 블록으로부터 숨겨짐

유효범위 규칙

```
{     // 블록 A
      int a = 2, b = 4;
      printf("A: a = %d, b = %d\n", a, b);
      { // 블록 B
         int a;
         a = 5;
         b++;
         printf("B: a = %d, b = %d\n", a, b);
      }
      printf("A: a = %d, b = %d\n", a, b);
}
```

블록 A의 a와 b의 유효범위

블록 B의 a의 유효범위

지역 변수와 전역 변수

- **지역 변수**(Local variable)
  - 함수 몸체 안에서 선언된 변수
  - 함수 내에서만 사용이 가능한 변수
- **전역 변수**(Global variable)
  - 함수 외부에서 선언된 변수
  - 모든 함수에서 사용이 가능한 변수

지역 변수와 전역 변수

프로그램 6.9

```
#include <stdio.h>
int   a = 0, b = 0, c = 0;// 전역 변수
int   f(void);
int main(void){
   a = 1;
   b = 2;
   printf("a + b = %d\n", f());
   printf("a = %d, b = %d, c = %d\n", a, b, c);
   return 0;
}
int f(void){
   int   c;                // 지역 변수
   c = a + b;
   return c;
}
```

프로그램 결과

```
a + b = 3
a = 1, b = 2, c = 0
```

지역 변수와 전역 변수

- 함수는 하나의 값만 호출한 곳으로 리턴 할 수 있음
- 여러 개의 값을 호출한 곳으로 리턴 해야 할 때 전역 변수를 사용하면 쉬움

지역 변수와 전역 변수

프로그램 6.10 일부

```
int   quotient, rem;          // 전역 변수
int main(void){
   int a = 10, b = 3;
   if (divide(a, b))
      printf("0으로 나눌 수 없습니다.\n");
   else
      printf("%d / %d : 몫은 %d이고 나머지는 %d입니다.\n",
             a, b, quotient, rem);
   return 0;
}
int divide(int dividend, int divisor){
   if (is_zero(divisor))
      return -1;
   quotient = dividend / divisor;
   rem = dividend % divisor;
   return 0;
}
```

지역 변수와 전역 변수

- 프로그램 6.10의 divide()는 나누기 결과인 몫과 나머지를 호출한 곳으로 전달하기 위해 전역 변수 사용
- 전역 변수는 다른 함수에서도 사용이 가능하기 때문에 부작용이 발생할 수 있고 모듈성에도 방해가 됨
- 포인터를 사용하여 여러 개의 값을 리턴 할 수 있음
  - 전역 변수를 사용하는 것보다 포인터를 사용하는 것이 더 좋음

기억영역 클래스

- C의 모든 변수와 함수는 두 가지 속성을 가짐
  - 형, 기억영역 클래스
- 기억영역 클래스는 객체의 유효범위와 라이프타임을 지정함
- 기본 기억영역 클래스
  - 자동, 외부, 레지스터, 정적
- 대응되는 키워드
  - auto, extern, register, static

auto

- 함수의 몸체에서 선언된 변수는 디폴트로 자동
- 블록 안에서 선언된 변수는 묵시적으로 자동 기억영역 클래스임
- auto를 사용하여 기억영역 클래스를 명시할 수도 있지만, 보통은 생략함
- 예, 지역 변수 i, j, k 선언

```
auto int   i, j, k;      // int   i, j, k;
```

auto

- 블록을 들어갈 때, 자동 변수들을 위해 메모리가 할당되고,
- 블록을 빠져나갈 때, 자동 변수가 할당 받은 메모리는 회수됨

register

- 기억영역 클래스 register는 컴파일러에게 변수를 가능하다면 고속 메모리 레지스터에 저장되도록 함
- 한정된 자원으로 인해 할당하지 못하면, 이 기억영역 클래스는 디폴트로 자동 기억영역 클래스가 됨
  - 전역 변수는 register로 지정할 수 없음
- 자동 변수에는 주소 연산자를 사용할 수 없음
- 이러한 변수는 사용되기 바로 직전에 선언하는 것이 좋음

register

- 묵시적 int
  - C90에서는 다음과 같이 선언한 i는 int 형임

    ```
    register i;
    ```

  - C99에서는 int를 생략하면 안됨

    ```
    register int i;
    ```

- for 문에서 선언된 변수도 register를 지정할 수 있음

extern

- **변수의 선언과 정의**
  - 변수 선언 : 컴파일러에게 변수 이름과 형을 알려줌
  - 변수 정의 : 컴파일러에게 변수에 메모리를 할당하게 함
  - 지금까지 변수 선언은 변수 정의를 포함하였지만, extern 변수에서는 구분해야 함

extern

- 함수 밖에서 선언된 변수의 기억영역 클래스는 기본적으로 extern
- 앞에서 다룬 전역 변수는 외부 변수임
- 전역 변수(외부 변수)를 정의할 때 extern을 생략할 수 있음
  - 함수 밖에서 extern 없이 선언된 변수는 외부 변수를 정의하는 것임

extern

- 외부 변수를 정의할 때 extern을 명시하고자 한다면 그 외부 변수를 반드시 초기화해야 함
  - extern이 붙은 선언문에서 변수가 초기화되지 않는 다면, 이는 외부 변수를 선언하는 것임
    - 외부 변수 정의

```
extern int quotient = 0, rem = 0;
```

    - 외부 변수 선언

```
extern int quotient, rem;
// 다른 곳에 quotient와 rem이 외부 변수로
// 정의되어 있고, 이를 사용할 것임을 나타냄
```

extern

- 외부 변수는 프로그램이 종료될 때까지 메모리에 계속 남아 있게 됨
- 함수들 간에 정보 전달을 위해 종종 사용 됨
  - 사용하기는 편하지만 부작용이 발생할 수 있기 때문에 좋은 방법은 아님
  - 모듈성을 방해 함
- 외부 변수는 자동적으로 0으로 초기화 됨
- 외부 변수들은 자동이나 레지스터 기억영역 클래스를 가질 수 없음

extern

프로그램 6.11 일부

```
int main(void){
   int a = 10, b = 3;
   extern int   quotient, rem;
   if (divide(a, b))
      printf("0으로 나눌 수 없습니다.\n");
   else
      printf("%d / %d : 몫은 %d이고 나머지는 %d입니다.\n",
             a, b, quotient, rem);
   return 0;
}
int divide(int dividend, int divisor){
   extern int   quotient, rem;
   if (is_zero(divisor))
      return -1;
   quotient = dividend / divisor;
   rem = dividend % divisor;
   return 0;
}
int   quotient, rem;     // 외부 변수 정의
//extern int quotient = 0, rem = 0;
```

extern

- 함수는 기본적으로 extern 임
  - 보통 함수를 정의할 때 extern 생략함
  - 프로그램 6.11의 divide() 함수 헤더는 다음과 같음

```
extern int divide(int dividend, int divisor)
```

static

- static은 지역 변수와 전역 변수에 적용할 수 있음
  - 전역 변수에 적용하는 것은 13장에서 다룸
- 지역 변수에 static이 적용되면 정적 변수가 어떤 프로그램이 끝날 때까지 그 값을 계속 유지하게 됨
  - 즉 제어가 정적 변수가 선언된 블록을 빠져나가도 다시 들어오면 이전 값을 가지고 있음
- 정적 변수는 디폴트로 0으로 초기화 됨

static

프로그램 6.12

```
unsigned next_fibonacci(void);
int main(void){
   int i;
   printf(" i | fibonacci\n");
   printf("--------------\n");
   for (i = 1; i <= 10; i++)
      printf("%2d | %6u\n", i, next_fibonacci());
   return 0;
}
unsigned next_fibonacci(void){
   static unsigned f = 0, pre_f = 1;      // 정적 변수
   unsigned re;
   re = f;
   f += pre_f;
   pre_f = re;
   return re;
}
```

프로그램 결과

```
 i | fibonacci
--------------
 1 |         0
 2 |         1
 3 |         1
 4 |         2
 5 |         3
 6 |         5
 7 |         8
 8 |        13
 9 |        21
10 |        34
```

수학 함수

- C는 다양한 수학 함수를 라이브러리 제공

```
#include <math.h>
```

- 구형 C 시스템에서는 수학 함수가 표준 라이브러리와 분리되어 있음
  - 컴파일할 때 수학 라이브러리를 지정해 줘야 함

```
gcc pgm.c -lm
```

수학 함수

프로그램 6.13

```
#include <math.h>
#include <stdio.h>
int main(void)
{
   int i;
   printf(" i   sqrt(i)      e^i            i^i\n");
   for (i = 1; i <= 10; i++)
      printf("%2d %8.2f %10.2f %15.2f\n", i,
                        sqrt(i), exp(i), pow(i, i));
   return 0;
}
```

프로그램 결과

```
 i     sqrt(i)       e^i              i^i
 1      1.00        2.72             1.00
 2      1.41        7.39             4.00
 3      1.73       20.09            27.00
 4      2.00       54.60           256.00
 5      2.24      148.41          3125.00
 6      2.45      403.43         46656.00
 7      2.65     1096.63        823543.00
 8      2.83     2980.96      16777216.00
 9      3.00     8103.08     387420489.00
10      3.16    22026.47   10000000000.00
```

inline 함수

- 함수의 단점은 함수 호출과 리턴 시에 추가적인 연산으로 인해 실행시간이 길어진다는 것임
- 함수의 장점을 살리면서 단점을 보완하는 inline 함수가 C99에서 추가 됨
- inline 함수는 함수 헤더에 inline이 명시된 것만 제외하면 일반 함수와 같음
- inline은 컴파일러에게 inline 함수를 호출하는 문장을 inline 함수의 몸체로 대치하도록 하기 위함
  - 대치 여부는 컴파일러가 결정함

inline 함수

- inline 함수 호출 문장이 inline 함수의 몸체로 대치되기 때문에 긴 함수를 inline 함수로 만들면 실행 파일이 길어진다는 단점이 있음
- inline 함수는 길이가 짧은 함수에 보통 적용함

inline 함수

프로그램 6.14

```
#include <stdio.h>
inline int square(int x)
{
    return x * x;
}
int main(void)
{
    int a, b = 5;
    a = square(b);
    return 0;
}
```

inline 함수

프로그램 6.15

```
#include <stdio.h>
int main(void)
{
    int a, b = 5;
    a = b * b;
    return 0;
}
```

- 프로그램 6.14는 프로그램 6.15와 같은 모양으로 바뀜
- 대치 여부와 형태는 컴파일러에 달려있음
- 프로그램 6.15는 단지 예임

재귀 함수

- 어떤 함수가 직접이든 간접이든 자기 자신을 호출하는 함수를 재귀 함수라 함

프로그램 6.16

```
#include <stdio.h>
int main(void)
{
    printf("끝나지 않는 재귀 함수...\n");
    main();
    return 0;
}
```

프로그램 결과

```
끝나지 않는 재귀 함수...
끝나지 않는 재귀 함수...
끝나지 않는 재귀 함수...
끝나지 않는 재귀 함수...
끝나지 않는 재귀 함수...
끝나지 않는 재귀 함수...
              .
              .
              .
```

재귀 함수

- 많은 문제들이 재귀적으로 정의됨
- 재귀적으로 정의되는 문제들을 재귀 함수로 구현하면 매우 쉽게 프로그래밍이 가능함
- 재귀 함수를 작성할 때에는 무한 루프에 빠지지 않게 작성해야 함
  - 재귀적으로 호출하면 언젠가는 프로그램이 종료되도록 해야 함

재귀 함수

- **계승의 정의**

$$n! = \begin{cases} 1, & \text{if } n = 0 \\ n \times (n-1)!, & \text{if } n > 0 \end{cases}$$

  - 재귀적으로 정의되어 있음
    - n!를 정의하기 위해 (n-1)!가 사용됨
    - 이 재귀는 (n-1)이 0이 되면 종료함

재귀 함수

프로그램 6.17

```
#include <stdio.h>
int factorial_r(int n);
int main(void){
   int n = 5;
   printf("%2d! = %9d\n", n, factorial_r(n));
   return 0;
}
int factorial_r(int n){
   if (n == 0)
      return 1;
   else
      return (n * factorial_r(n - 1));
}
```

재귀 함수

- factorial_r(5)를 호출했을 때, 재귀호출 되는 과정

```
factorial_r(5) : return (5 * factorial_r(4))
```

재귀 함수

- factorial_r(5)를 호출했을 때, 재귀호출 되는 과정

```
factorial_r(5) : return (5 * factorial_r(4))

factorial_r(4) : return (4 * factorial_r(3))
```

재귀 함수

- factorial_r(5)를 호출했을 때, 재귀호출 되는 과정

```
factorial_r(5) : return (5 * factorial_r(4))

factorial_r(4) : return (4 * factorial_r(3))

factorial_r(3) : return (3 * factorial_r(2))
```

재귀 함수

- factorial_r(5)를 호출했을 때, 재귀호출 되는 과정

```
factorial_r(5) : return (5 * factorial_r(4))

factorial_r(4) : return (4 * factorial_r(3))

factorial_r(3) : return (3 * factorial_r(2))

factorial_r(2) : return (2 * factorial_r(1))
```

재귀 함수

- factorial_r(5)를 호출했을 때, 재귀호출 되는 과정

```
factorial_r(5) : return (5 * factorial_r(4))

factorial_r(4) : return (4 * factorial_r(3))

factorial_r(3) : return (3 * factorial_r(2))

factorial_r(2) : return (2 * factorial_r(1))

factorial_r(1) : return (1 * factorial_r(0))
```

재귀 함수

- factorial_r(5)를 호출했을 때, 재귀호출 되는 과정

```
factorial_r(5) : return (5 * factorial_r(4))

factorial_r(4) : return (4 * factorial_r(3))

factorial_r(3) : return (3 * factorial_r(2))

factorial_r(2) : return (2 * factorial_r(1))

factorial_r(1) : return (1 * factorial_r(0))

factorial_r(0) : return 1
```

재귀 함수

- factorial_r(5)를 호출했을 때, 재귀호출 되는 과정

```
factorial_r(5) : return (5 * factorial_r(4)) → 5*24=120

factorial_r(4) : return (4 * factorial_r(3)) → 4*6=24

factorial_r(3) : return (3 * factorial_r(2)) → 3*2=6

factorial_r(2) : return (2 * factorial_r(1)) → 2*1=2

factorial_r(1) : return (1 * factorial_r(0)) → 1*1=1

factorial_r(0) : return 1
```

재귀 함수

- factorial_r(5)를 호출했을 때, 재귀호출 되는 과정

```
factorial_r(5)
      = 5 * factorial_r(4)
      = 5 * (4 * factorial_r(3))
      = 5 * (4 * (3 * factorial_r(2)))
      = 5 * (4 * (3 * (2 * factorial_r(1))))
      = 5 * (4 * (3 * (2 * (1 * factorial_r(0)))))
      = 5 * (4 * (3 * (2 * (1 * (1)))))
      = 120
```

재귀 함수

- **피보나치 수열의 정의**

$$f_i = \begin{cases} 0, & \text{if } i = 0 \\ 1, & \text{if } i = 1 \\ f_{i-1} + f_{i-2}, & \text{if } i > 1 \end{cases}$$

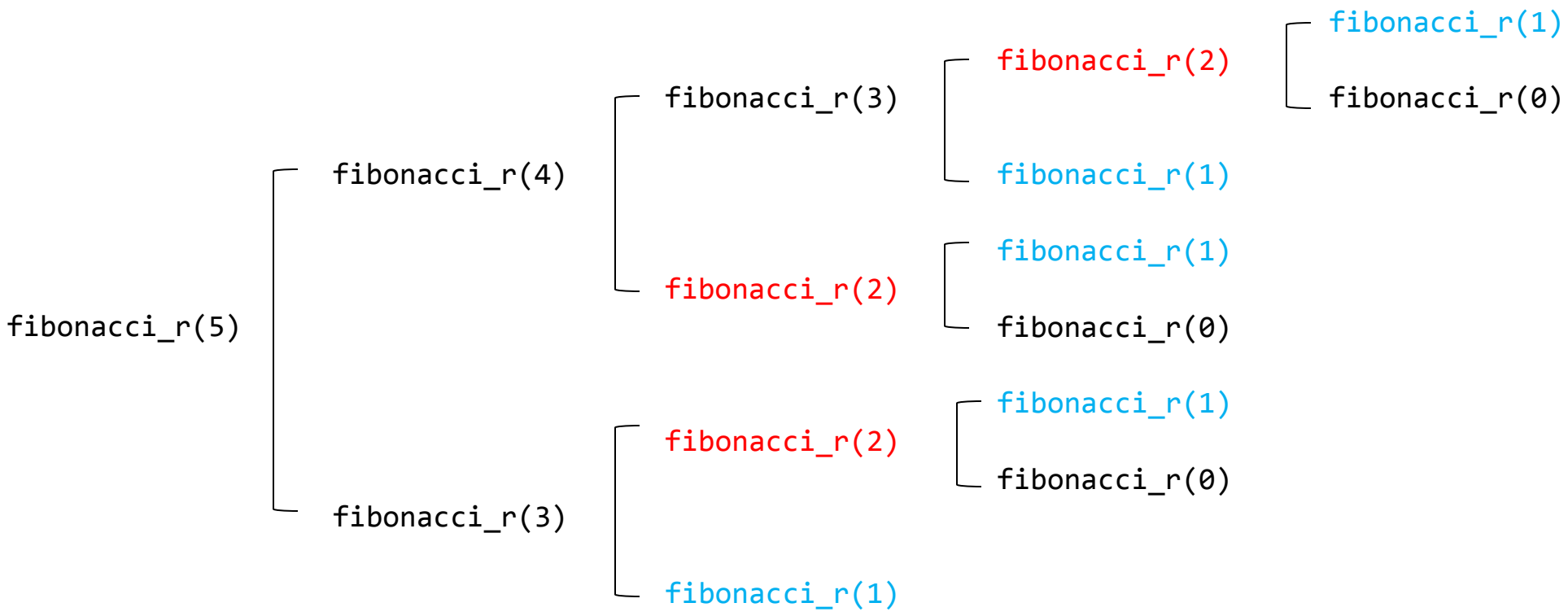
재귀 함수

함수 6.8

```
int fibonacci_r(int n)
{
   if (n <= 1)
      return n;
   else
      return (fibonacci_r(n - 1) + fibonacci_r(n - 2));
}
```

재귀 함수

- fibonacci_r(5)가 호출되었을 때의 호출되는 함수 호출 순서



재귀 함수

- n 값에 따른 fibonacci_r() 함수 호출 개수

n	fibonacci_r(n)의 값	fibonacci_r() 함수가 호출되는 횟수
0	0	1
1	1	1
2	1	3
.....		
23	28657	92735
24	46368	150049
.....		
42	267914296	866988873
43	433494437	1402817465

재귀 함수

함수 6.9

```
int fibonacci_i(int n)
{
   int i, tmp, fib = 1, fib_p = 0;
   if (n <= 1)
      return n;
   for (i = 2; i <= n; i++){
      tmp =  fib;
      fib += fib_p;
      fib_p = tmp;
   }
   return fib;
}
```

재귀의 효율성

- 많은 알고리즘은 재귀적 방식과 반복적 방식 둘 다로 표현할 수 있음
- 전형적으로, 재귀가 더 간결하고 같은 계산을 하는 데 더 적은 변수를 필요로 함
- 반면, 재귀는 각 호출을 위한 인자와 변수를 스택에 쌓아두어 관리하기 때문에 많은 시간과 공간을 요구함
- 즉, 재귀를 사용할 때에는 비효율성을 고려해야 함
- 그러나 일반적으로 재귀적 코드는 작성하기 쉽고, 이해하기 쉬우며, 유지보수하기가 쉬움

재귀 함수

```
#include <stdio.h>
int main(void)
{
    static int i = 0;
    printf("끝나지 않는 재귀 함수...:%d\n", i++);
    main();
    return 0;
}
```

시간 함수

- C는 현재시간과 프로그램의 CPU 사용 시간을 알려 주는 함수 제공

```
#include <time.h>
clock_t clock(void);
  // 프로그램의 사용 CPU 클록 시간 리턴
  // CLOCKS_PER_SEC : 1초당 클록 수
time_t time(time_t *p);
  // 1970년 1월 1일부터 경과된 시간 리턴
double difftime(time_t time1, time_t time0);
  // time1과 time0의 시간차 리턴
```

시간 함수

프로그램 6.18

```
#include <time.h>
#include <stdio.h>
int main(void){
   int i, n = 1000000000;
   clock_t start_clock, end_clock, diff_clock, ex_time;
   float a, b = 1.3, c = 100.2;
   start_clock = clock();
   for (i = 0; i < n; i++)
      a = b * c;
   end_clock = clock();
   diff_clock = end_clock - start_clock;
   printf("%d번 곱하기 실행시간 : %d 클록\n", n, diff_clock);
   ex_time = diff_clock / CLOCKS_PER_SEC;
   printf("%d번 곱하기 실행시간 : %d 초\n", n, ex_time);
   return 0;
}
```

프로그램 결과

```
1000000000번 곱하기 실행시간 : 6375 클록
1000000000번 곱하기 실행시간 : 6 초
```