C 프로그래밍 및 실습

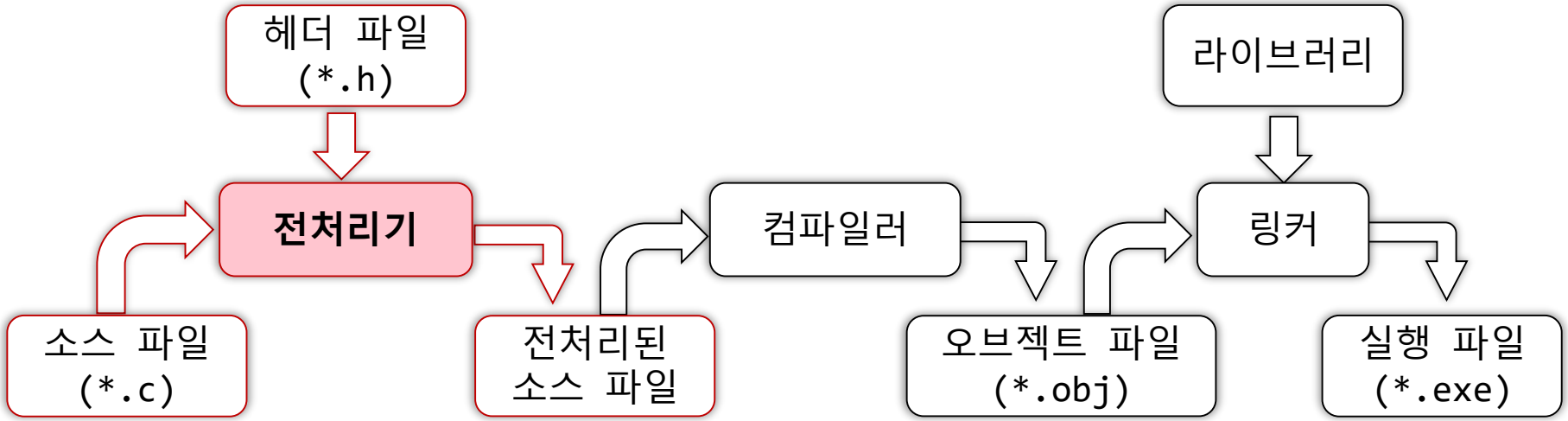
15. 전처리기와 분할 컴파일

세종대학교

목차

1) 전처리기

- **전처리란?**
  - 컴파일러가 소스 파일(*.c)을 컴파일하기 이전 과정
  - 전처리기 지시자를 사용 (예) #include
- **사용하는 이유**
  - 프로그램을 간단히 확장하고, 가독성을 높여 유지보수에 도움



1) 전처리기

- **전처리 지시자**
  - 프로그램 선두에 위치
  - #으로 시작하고 문장 끝에 세미콜론(;)을 사용하지 않음

- **자주 사용되는 전처리 지시자**

전처리 지시자	기능
#include	프로그램 외부의 파일을 불러옴
#define	매크로(macro) 상수/함수를 정의
#undef	정의한 매크로를 취소
#if ~ (#elif ~ #else ~) #endif	조건부 컴파일
#ifdef ~ (#elif ~ #else ~) #endif	
#ifndef ~ (#elif ~ #else ~) #endif	

1) 전처리기

- **#include**
  - 프로그램 외부에 존재하는 파일을 소스에 포함시킴
  - 지정된 특정 파일의 내용을 해당 지시사가 있는 위치에 삽입

  - < > : 컴파일러가 제공하는 헤더 파일을 포함시킬 때 주로 사용

  - " " : 프로그래머가 직접 만든 파일을 포함시킬 때 주로 사용
    - ✓ 절대 경로 혹은 상대 경로 형식으로 사용
    - ✓ fopen()에서 파일이름을 작성할 때와는 달리, \ 만으로 경로 표현

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```

```
#include "myheader.h"
#include "C:\user\mylib\lib.h"
#include "..\mylib\lib.h"
```

1) 전처리기

- **#define**
  - 특정 상수를 프로그래머가 정의한 문자열로 대체
    - ✓ 프로그램의 가독성을 높여 유지보수 용이
  - 원칙적으로는 한 라인에 작성
    - ✓ 긴 경우에는 \(역 슬래시) 기호를 사용하여 다음 줄에 작성되어 있는 내용과 연결
  - **매크로 상수**
    - ✓ 반복적으로 사용되는 상수를 새로운 이름으로 정의하여 사용
  - **매크로 함수**
    - ✓ 반복적으로 사용되는 프로그램의 모듈을 함수로 정의하여 사용
    - ✓ 프로그램의 해독과 수정이 용이

1) 전처리기

- **매크로 상수**
  - 형식: **#define 매크로이름 상수**
    - ✓ 매크로이름: 주로 대문자로 표시, 공백 허용하지 않고 숫자로 시작하면 안됨
    - ✓ 상수: 숫자, 문자, 문자열, 시스템 이름, 자료형 이름 등

```
#define PI 3.14
#define MAX 100
#define SUM MAX+1   // 중첩 매크로: SUM 정의 시
                    // 앞서 정의된 매크로 상수 MAX를 사용
```

1) 전처리기

- 매크로 상수의 필요성 및 사용법
  - ✓ 아래에서 PI를 정의하지 않고, 3.14로 프로그램을 작성했다면?
    - ✓ 3.14를 3.1415로 변경하려면, 일일이 3.14를 찾아서 수정해야 함

```
#include <stdio.h>
#define PI 3.14             // 원주율을 매크로 상수로 정의

int main() {
    int r = 3;              // 원의 반지름
    double cir, area;       // 원의 둘레, 원의 면적

    cir = 2 * PI * r;       // 매크로 상수 사용
    area = PI * r * r;
    printf("cir = %f, area = %f\n", cir, area);
    return 0;
}
```

1) 전처리기

- **매크로 함수**
  - 형식: **#define 매크로함수이름(함수인자) 함수정의**
    - ✓ 매크로 함수이름: 주로 대문자로 표시

```
#define ADD(x, y) ((x) + (y))
#define MAX(x, y) ((x) > (y) ? (x) : (y))
```

  - 주의사항
    - ✓ 매크로 함수이름과 (함수인자) 사이에 공백이 없어야 함
    - ✓ 정의 부분에서 각 함수 인자를 반드시 괄호 ( ) 로 묶어 주어야 함
    - ✓ 정의 부분 전체를 반드시 괄호 ( ) 로 묶어 주어야 함

1) 전처리기

- 매크로 함수 사용 예제

```
#include <stdio.h>

#define PI 3.14                     // 매크로 상수
#define SQUARE(x) ((x) * (x))       // 매크로 함수

int main() {
    double area;                    // 원의 면적

    area = PI * SQUARE(2 + 2);      // area = 3.14 * (4 * 4)
    printf("area = %f\n", area);
    return 0;
}
```

1) 전처리기

- 매크로 함수 정의에서 괄호 사용의 필요성

```
// 의도한 결과

#define SQUARE(x) ((x) * (x))

PI * SQUARE(2 + 2)
➜ 3.14 * ((2 + 2) * (2 + 2))
➜ 3.14 * (4 * 4)
➜ 50.24
```

```
// 인자의 괄호를 생략하면?

#define SQUARE(x) (x * x)

PI * SQUARE(2 + 2)
➜ 3.14 * (2 + 2 * 2 + 2)
➜ 3.14 * (2 + 4 + 2)
➜ 25.12
```

```
// 의도한 결과

#define ADD(x, y) ((x) + (y))

5 * ADD(2, 3)
➜ 5 * ((2) + (3))
➜ 5 * (5)
➜ 25
```

```
// 정의부 전체의 괄호를 생략하면?

#define ADD(x, y) (x) + (y)

5 * ADD(2, 3)
➜ 5 * (2) + (3)
➜ 10 + 3
➜ 13
```

1) 전처리기

- **#undef**
  - 정의된 매크로를 해제하는 지시자
  - 해제된 매크로는 재정의 가능

  - 사용법: **#undef** 매크로이름

```
#define MAX 100

#undef MAX

#define MAX 5000
```

1) 전처리기

- **조건부 컴파일 전처리 지시자**
  - 전처리문에서 주어진 조건의 만족 여부에 따라 **코드를 선택적으로 컴파일**하도록 하는 기능
  - 변수, 함수, 매크로가 중복되지 않도록 하거나, 이식성(호환성) 높은 코드를 개발할 때 유용
  - 사용법은 조건문의 if 문과 유사

전처리 지시자	기능
#if ~ #endif	조건이 참이면 컴파일에 포함
#ifdef ~ #endif	매크로가 정의되어 있으면 컴파일에 포함
#ifndef ~ #endif	매크로가 정의되어 있지 않으면 컴파일에 포함

1) 전처리기

- **#if ~ (#elif) ~ (#else) ~ #endif**
  - 조건식이 참이면 컴파일에 포함

```
#if 조건식1
   컴파일 할 문장1;
#elif 조건식2
   컴파일 할 문장2;
…
#else
   컴파일 할 문장5;
#endif
```

```
#if   OS == 1          // Linux
   컴파일 할 문장1;
#elif  OS  == 2        // Windows
   컴파일 할 문장2;
#else                  // 그 외의 OS
      컴파일 할 문장3;
#endif
```

  - 주의) 조건식에서
    - ✓ 실수 상수, 문자열 상수, 변수 등을 사용할 수 없음
    - ✓ 관계연산자, 논리연산자, 산술연산자는 사용 가능

1) 전처리기

- **#ifdef ~ (#elif) ~ (#else) ~ #endif**
  - 매크로가 정의되어 있으면 컴파일에 포함

```
#ifdef 매크로이름1
   컴파일 할 문장1;
#elif 매크로이름2
   컴파일 할 문장2;
…
#else
   컴파일 할 문장5;
#endif
```

```
#ifdef  UNIX
   #define DDIR "/usr/data"
#else
   #define DDIR "\usr\data"
#endif
```

- **#ifndef ~ (#elif) ~ (#else) ~ #endif**
  - 매크로가 정의되어 있지 않으면 컴파일에 포함

목차

2) 분할 컴파일

- **분할 컴파일이란?**
  - 하나의 프로젝트를 모듈 별로 **여러 소스 파일에 나누어 작성**하고, 소스 파일 별로 컴파일하는 것
    - ✓ 모듈(module): 논리적으로 특정 기능 구현을 위한 함수 그룹
  - 필요성
    - ✓ 대형 프로그램 작성 시, 작업 분담 목적
    - ✓ 모듈의 재사용성을 높임
    - ✓ 프로그램 관리의 효율성을 향상시킴
    - ✓ 프로그램의 일부를 변경/확장하기 위해 프로그램 전체를 컴파일해야 하는 번거로움을 덜 수 있음

2) 분할 컴파일

- **분할 컴파일 따라 해보기**
  - 하나의 소스 프로그램인 original.c를 3개의 소스파일로 분할
    - ✓ main.c
    - ✓ myfunc1.c
    - ✓ myfunc2.c
  - 각 파일을 따로 컴파일 한 후, 링커를 통해 하나의 실행 파일을 생성

    → Visual studio에서는 "빌드"를 통해 해당 작업을 수행

2) 분할 컴파일

original.c

```c
#include <stdio.h>

int myfunc1(int x, int y);
void myfunc2(int n);

int main() {
  int a, b, cnt = 0;
  scanf("%d %d", &a, &b);
  cnt = myfunc1(a, b);
  myfunc2(cnt);
  return 0;
}

int myfunc1(int x, int y) {
  return (x * y - x);
}

void myfunc2(int n) {
  int i;
  for (i = 0; i < n; i++)
    printf("count - %d\n",
i+1);
}
```

main.c

```c
#include <stdio.h>

int myfunc1(int x, int y);
void myfunc2(int n);

int main() {
  int a, b, cnt = 0;
  scanf("%d %d", &a, &b);
  cnt = myfunc1(a, b);
  myfunc2(cnt);
  return 0;
}
```

myfunc1.c

```c
int myfunc1(int x, int y) {
  return (x * y - x);
}
```

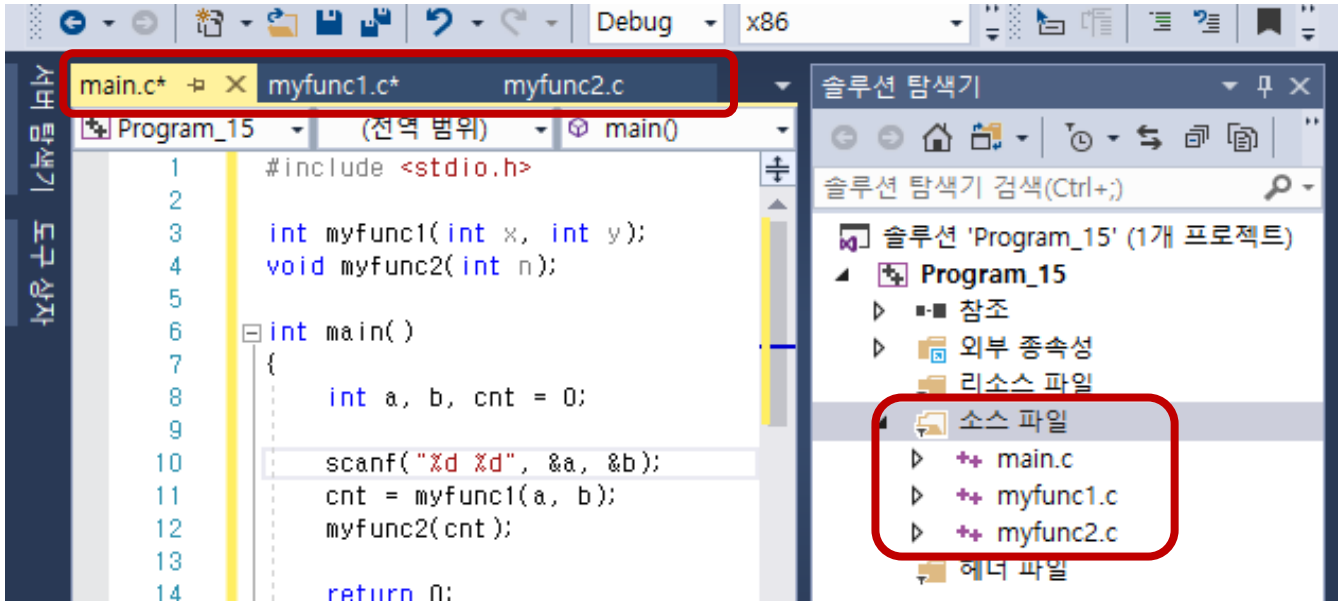
myfunc2.c

```c
#include <stdio.h>

void myfunc2(int n) {
  int i;
  for (i = 0; i < n; i++)
    printf("%d\n", i+1);
}
```

2) 분할 컴파일

- **Visual studio 환경에서 앞의 예제 실행하는 방법**
  - ① Project_15라는 이름으로 프로젝트 하나를 생성
  - ② 소스 파일 폴더 내에 main.c, myfunc1.c, myfunc2.c를 생성 후, 각 파일에 앞의 소스 작성
  - ③ Project1_15 빌드



2) 분할 컴파일

- **분할 컴파일시 주의사항**
  - 각 소스 파일은 독립적으로 오류 없이 컴파일 되어야 함
    - ✓ 각 파일 내에서 사용하는 변수나 함수 등은 해당 파일 내에서 선언되어야 함
  - 다른 파일에서 선언된 변수나 정의된 함수 등을 사용하고 싶다면?
    - ✓ 해당 컴파일러에게 어떤 변수와 함수가 외부 파일에서 선언 및 정의된 변수나 함수인지를 알려주어야 함

    → extern 선언

2) 분할 컴파일

- **extern 키워드**
  - 다른 소스 파일에 선언된 변수 및 정의된 함수를 현재의 소스 파일에 사용하고자 할 때 사용
    - ✓ 변수 또는 함수가 외부에 선언 및 정의되어 있다는 사실만 컴파일러에게 알려줌
    - ✓ 구체적으로 어느 파일에 선언 및 정의되어 있는지는 몰라도 됨
    - ✓ extern 키워드가 붙은 변수를 외부 변수라고도 함
  - 형식
    - ✓ **extern 변수선언/함수선언;**
      - ✓ 함수의 경우 extern을 생략할 수 있음

2) 분할 컴파일

- **extern 선언 사용 예**

메모리가 할당된 실제 변수

main.c

```
#include <stdio.h>
void count();          // 외부함수 선언 (extern 생략)
int num = 1;           // 전역 변수 선언
int main() {
   printf("before - main.c: num = %d\n", num);
   count();                    // 외부함수 호출
   printf("after - main.c: num = %d\n", num);
   return 0;
}
```

이 선언으로 인해 메모리 공간이 할당 되지는 않음

counter.c

```
#include <stdio.h>
extern int num;              // 외부 변수 선언
void count() {               // count() 함수 정의 부분
   num++;                    // 외부 변수 사용
   printf("count() - count.c: num = %d\n", num);}
```

2) 분할 컴파일

- **extern vs. static 변수와 함수**
  - 앞 예제에서 변수 num은 전역 변수이자 외부변수
    - ✓ 즉, 다른 파일에서도 extern 변수로 선언하면 자유로이 사용 가능
  - 변수가 선언된 파일 안에서는 전역 변수처럼 쓰지만,
    외부(다른 파일)로부터의 접근을 제한하려면?

    → 해당 전역 변수 선언 시 앞에 **static 키워드**를 붙이면 됨
    (정적 전역 변수)
    - ✓ 함수도 동일하게 적용
  - 참고) 정적 지역 변수 (8장에서 학습)
    - ✓ 함수 내에서 선언된 정적 변수로, 사용 범위는 해당 함수 내부로 제한
    - ✓ 지속 시간은 프로그램 실행 기간 전체

2) 분할 컴파일

- **extern vs. static 함수 사용 예**
  - 함수 eval()은 main.c에서 호출 가능
    - ✓ 이를 위해 main.c에서 extern 으로 선언 (extern 생략 가능)
  - 함수 f()는 static 이므로 func.c 내에서만 호출 가능
    - ✓ main.c에서 호출 불가능

main.c

```
#include <stdio.h>
extern int eval(int n);

int main()
{
    printf("%d\n", eval(5));
    return 0;
}
```

func.c

```
static int f(int x)
{
    return x*4+3;
}

int eval(int n)
{
    return 3*f(n);
}
```

2) 분할 컴파일

- **헤더 파일 (.h)**
  - 여러 소스 파일에서 사용하는 외부 변수나 함수 등을 모아 놓은 파일로, 필요한 경우 간단히 헤더파일만 include 시키면 됨
    - ✓ 예) 문자열 관련 함수는 <string.h>에 선언
  - 헤더파일(.h)에 포함되는 내용
    - ✓ 상수, 자료형 정의 (구조체 정의 등), 함수 원형 선언, 외부 변수 선언(extern), typedef 정의, 매크로 명령문
  - 비교) 소스파일(.c)에 포함되는 내용
    - ✓ 전역 변수 선언, 함수 본체 정의
  - Visual Studio에서 사용자 정의 헤더 파일을 만드는 방법
    - ✓ 소스 파일 추가와 유사

2) 분할 컴파일

- **헤더파일 중복삽입 문제의 예**
  - struct student가 두 번 정의됨

myheader1.h

```
#include <stdio.h>
#define SIZE 3

struct student{
   int id;
   double score;
};
extern struct student st[SIZE];
```

main.c

```
#include "myheader1.h"
#include "myheader2.h"
struct student st[SIZE];

int main() {
  int i;
  for (i=0; i<SIZE; i++) {
    scanf("%d %lf", &st[i].id, &st[i].score);
    grade(&st[i]);
  }
  return 0;
}
```

myheader2.h

```
#include "myheader1.h"

void grade(struct student *st1);
```

grade.c

```
#include "myheader.h"
void grade(struct student *st1) {
   ...
}
```

2) 분할 컴파일

- **조건부 컴파일을 사용하여 헤더파일 중복삽입 문제 해결법**
  - #ifndef ~ #endif 이용

```
#ifndef 헤더파일명
#define 헤더파일명
  (헤더파일 내용)
#endif
```

```
#ifndef __MYHEADER_H__
#define __MYHEADER_H__
  ...
#endif
```

✓ 추후에 이 헤더 파일이 다시 포함되더라도,
이미 상수 __MYHEADER_H__가 정의되어 있으므로,
헤더 파일 내용은 중복으로 포함되지 않음

2) 분할 컴파일

- 앞서 살펴 본 프로그램에서의 헤더파일 중복삽입 문제 해결 (3/3)

myheader1.h

```
#ifndef __MYHEADER1_H__
#define __MYHEADER1_H__

#include <stdio.h>
#define SIZE 3
struct student{
   int id;
   double score;
};
extern struct student st[SIZE];

#endif
```

__MYHEADER_H__가 매크로 상수로 정의됨

①

main.c

```
#include "myheader1.h"
#include "myheader2.h"
struct student st[SIZE];

int main() {
  int i;
  for (i=0; i<SIZE; i++) {
    scanf("%d %lf", &st[i].id, &st[i].score);
    grade(&st[i]);
  }
  return 0;
}
```

②

myheader2.h

```
#ifndef __MYHEADER2_H__
#define __MYHEADER2_H_

#include "myheader1.h"
void grade(struct student *st1);

#endif
```

__MYHEADER_H__가 이미 정의되어 있으므로
#include "myheader.h"는 중복으로 삽입되지 않음!

목차

3) 변수의 사용범위와 지속기간

- **다양한 변수의 종류**
  - 전역변수, 지역변수, 정적 변수

- **변수의 종류를 구분 짓는 특징**
  - 변수의 사용범위
    - ✓ 프로그램에서 변수가 사용될 수 있는 장소의 범위
    - ✓ 변수의 공간적 특성을 나타냄
  - 변수의 지속기간
    - ✓ 변수에 할당된 기억장소가 존재하는 기간
    - ✓ 변수의 시간적 특성을 나타냄

3) 변수의 사용범위와 지속기간

- **전역 변수 (사용범위와 지속기간 모두 전역적)**
  - 사용범위
    - ✓ 프로그램 전역에서 사용 가능
    - ✓ extern 키워드를 이용하면 다른 파일에서도 사용 가능
  - 지속기간
    - ✓ 프로그램 시작 시 할당된 메모리 공간이 프로그램 종료 시까지 유지

- **지역 변수 (사용범위와 지속기간 모두 지역적)**
  - 사용범위
    - ✓ 선언된 함수/블록 내부에서만 사용 가능
  - 지속기간
    - ✓ 프로그램 수행 도중, 함수/블록 시작 시 메모리 할당되고, 해당 함수/블록이 끝나면 해제

3) 변수의 사용범위와 지속기간

- **정적변수**
  - 분류
    - ✓ 정적 지역 변수: 지역 변수 앞에 static 키워드가 붙은 변수
    - ✓ 정적 전역 변수: 전역 변수 앞에 static 키워드가 붙은 변수
  - 변수의 지속기간 (전역적)
    - ✓ 둘 다 전역 변수처럼 프로그램 실행 기간 전체 동안 지속
  - 변수의 사용범위 (제한됨)
    - ✓ 정적 지역 변수: 선언된 함수/블록 내부에서만 사용
    - ✓ 정적 전역 변수: 선언된 파일 내부에서만 사용

3) 변수의 사용범위와 지속기간

▪ 각 변수 별 사용범위 : **전역 변수**

main.c

```
. . . . . .
int global1;
extern func2();

int main()
{
    int local1;
    func2();
    . . . . .
}
```

func1.c

```
. . . . . .
extern int global1;

static int func1()
{
    int local2;
    . . . . .
}
```

func2.c

```
. . . . . .
static int global2;

int func2()
{
    static int local3;
    . . . . .
}
```

3) 변수의 사용범위와 지속기간

- 각 변수 별 사용범위 : 지역변수

main.c

```
. . . . . .
int global1;
extern func2();

int main()
{
    int local1;
    func2();
    . . . . .
}
```

func1.c

```
. . . . . .
extern int global1;

static int func1()
{
    int local2;
    . . . . .
}
```

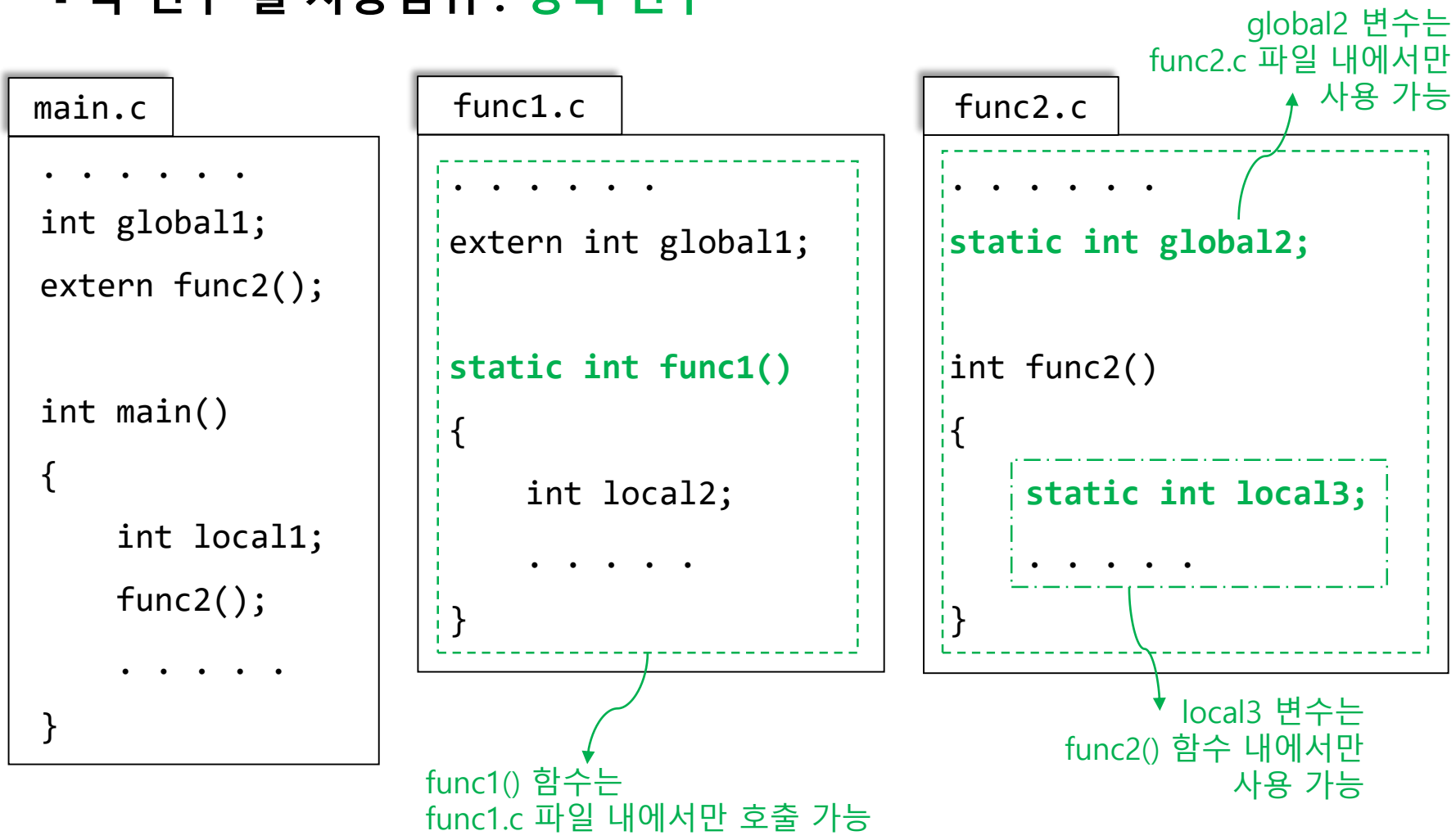
func2.c

```
. . . . . .
static int global2;

int func2()
{
    static int local3;
    . . . . .
}
```

3) 변수의 사용범위와 지속기간

▪ 각 변수 별 사용범위 : **정적 변수**



3) 변수의 사용범위와 지속기간

- **변수의 특징 정리**

<table>
<tr><th colspan="2">분류</th><th>관련
예약어</th><th>지속기간</th><th>기본
초기값</th><th>사용범위</th><th>동일 파일의
외부함수에서
사용</th><th>다른 파일의
외부함수에서
사용</th></tr>
<tr><td colspan="2">전역변수</td><td>extern</td><td rowspan="3">프로그램
실행기간
전체</td><td rowspan="3">0
'\0'
NULL
(자료형에
따라)</td><td>프로그램
전체</td><td>O</td><td>O</td></tr>
<tr><td rowspan="2">정적
변수</td><td>전역</td><td rowspan="2">static</td><td>파일 내부</td><td>O</td><td>X</td></tr>
<tr><td>지역</td><td rowspan="2">함수/
블록 내부</td><td rowspan="2">X</td><td rowspan="2">X</td></tr>
<tr><td colspan="2">지역변수</td><td>-</td><td>변수가 선언된
함수/블록의
실행 기간</td><td>쓰레기 값</td></tr>
</table>