

C Programming

Day08.B

2017.10.16

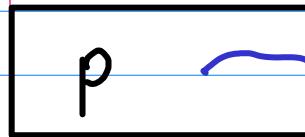
for loop
functions

storage class,
scope,
linkage

Copyright (c) 2015 - 2017 Young W. Lim.

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

$\&P$



P

$*P$

$P+1$

$*P+1$

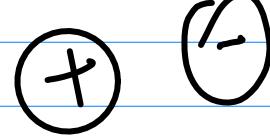
$P+2$

$*P+2$

$P+3$

$*P+3$

$p : \text{pointer var}$



$$*(P) \Leftrightarrow P[0]$$

$$*(P+1) \Leftrightarrow P[1]$$

$$*(P+2) \Leftrightarrow P[2]$$

$$*(P+3) \Leftrightarrow P[3]$$

array

```
#include <stdio.h>

int main( void ) {
    int i;
    char *p = "Hello!";
    printf("p = %s \n", p);
    printf("*p = %c \n", *p);
    printf(" p = %p \n", p);
    printf("&p = %p \n", &p);

    printf("-----\n");
    printf("*(p+0) = %c \n", *(p+0));
    printf("*(p+1) = %c \n", *(p+1));
    printf("*(p+2) = %c \n", *(p+2));
    printf("*(p+3) = %c \n", *(p+3));
    printf("*(p+4) = %c \n", *(p+4));
    printf("*(p+5) = %c \n", *(p+5));
    printf("*(p+6) = %c \n", *(p+6));

    printf("-----\n");
    for (i=0; i<7; ++i)
        printf("*(p+%d) = %c \n", i, *(p+i));

    printf("-----\n");
    printf("p[0] = %c \n", p[0]);
    printf("p[1] = %c \n", p[1]);
    printf("p[2] = %c \n", p[2]);
    printf("p[3] = %c \n", p[3]);
    printf("p[4] = %c \n", p[4]);
    printf("p[5] = %c \n", p[5]);
    printf("p[6] = %c \n", p[6]);

    printf("-----\n");
    for (i=0; i<7; ++i)
        printf("p[%d] = %c \n", i, p[i]);
}
```

```
p = Hello!
*p = H
p = 0x400944
&p = 0x7ffc7e144d90
-----#include <stdio.h>
*(p+0) = H
*(p+1) = e
*(p+2) = l
*(p+3) = l
*(p+4) = o
*(p+5) = !
*(p+6) =
-----
*(p+0) = H"p = %s \n", p
*(p+1) = e
*(p+2) = l
*(p+3) = l
*(p+4) = o" p = %p \n",
*(p+5) = !"&p = %p \n"
*(p+6) =
-----
p[0] = H
p[1] = e
p[2] = l
p[3] = l
p[4] = o
p[5] = !
p[6] = int("*(p+4) = %c
p[0] = H
p[1] = e
p[2] = l
p[3] = l
p[4] = o
p[5] = !
p[6] = print("-----\n", p[0], p[1], p[2], p[3], p[4], p[5], p[6])
```

local variable i,j,k

The screenshot shows a terminal window with the following C code:

```
#include <stdio.h>

void func() {
    int i = 11;
    printf("1.i= %d\n", i);

    {
        int j = 22;
        printf("2.i= %d\n", i);
        printf("2.j= %d\n", j);

        {
            int k = 33;
            printf("3.i= %d\n", i);
            printf("3.j= %d\n", j);
            printf("3.k= %d\n", k);
        }
    }

    printf("2.i= %d\n", i);
    printf("2.j= %d\n", j);
}
```

Annotations highlight specific variables:

- The variable `i` is circled in blue at its first declaration.
- The variable `j` is circled in green at its first declaration.
- The variable `k` is circled in yellow at its first declaration.

A large blue rectangle surrounds the entire code block, and a green rectangle surrounds the innermost block where `k` is declared.

local variable i's scope

```
#include <stdio.h>

void func() {
    int i = 111;
    printf("1.i= %d\n", i);

    {
        int j = 222;
        printf("2.i= %d\n", i);
        printf("2.j= %d\n", j);

        {
            int k = 333;
            printf("3.i= %d\n", i);
            printf("3.j= %d\n", j);
            printf("3.k= %d\n", k);
        }
        printf("2.i= %d\n", i);
        printf("2.j= %d\n", j);
    }
}
```

local variable's scope

The screenshot shows a terminal window with the following C code:

```
#include <stdio.h>
void func() {
    int i = 111;
    printf("1.i= %d\n", i);

    {
        int j = 222;
        printf("2.i= %d\n", i);
        printf("2.j= %d\n", j);

        {
            int k = 333;
            printf("3.i= %d\n", i);
            printf("3.j= %d\n", j);
            printf("3.k= %d\n", k);
        }

        printf("2.i= %d\n", i);
        printf("2.j= %d\n", j);
    }
}
```

Annotations with green lines and boxes highlight the scope of variables:

- A blue box encloses the entire function body, indicating the scope of the global variable `i`.
- A yellow box encloses the innermost block, indicating the scope of the local variable `k`.
- Green lines connect the `printf` statements to their corresponding variable declarations, showing the visible range of each variable.

local variable k's scope

The screenshot shows a terminal window titled "guest-tb5s9t@CLASS ~". The window contains a C program with several annotations:

```
#include <stdio.h>
void func() {
    int i = 111;
    printf("1.i= %d\n", i);

    {
        int j = 222;
        printf("2.i= %d\n", i);
        printf("2.j= %d\n", j);

        {
            int k = 333;
            printf("3.i= %d\n", i);
            printf("3.j= %d\n", j);
            printf("3.k= %d\n", k);
        }
    }
    printf("2.i= %d\n", i);
    printf("2.j= %d\n", j);
}
```

- A yellow circle highlights the identifier "k" in the innermost block.
- A green rectangular box encloses the entire function body, labeled "1. Function Body".
- A blue rectangular box encloses the inner block containing "j", labeled "2. Block containing j".
- A yellow rectangular box encloses the innermost block containing "k", labeled "3. Block containing k".
- Yellow lines connect the highlighted "k" to its declaration in the innermost block.

Local Variable hides other variable with the same name

The screenshot shows a terminal window titled "guest-tb5s9t@CLASS ~". The window displays the following C code:

```
#include <stdio.h>

void func() {
    int i = 111; (i)
    printf("1.i= %d\n", i);

    { (i)
        int i = 222; (i)
        printf("2.i= %d\n", i);

        { (i)
            int i = 333; (i)
            printf("3.i= %d\n", i);
        }

        printf("2.i= %d\n", i);
    }

    printf("1.i= %d\n", i);
}

printf("1.i= %d\n", i);
```

The code illustrates variable shadowing. It defines a global variable *i* with value 111. Inside the `func()` function, three local variables *i* are defined with values 222, 333, and 222 respectively. The final `printf` statement outside the function prints the value of the global *i*, which is 111. Handwritten annotations include circled *i*'s at various points in the code and a red note "the same named *i*" pointing to the innermost *i*.

1st i's scope

```
guest-tb5s9t@CLASS ~
File Edit View Search Terminal Help
#include <stdio.h>
void func() {
    int i = 111;
    printf("1.i= %d\n", i);
    {
        int i = 222;
        printf("2.i= %d\n", i);
        {
            int i = 333;
            printf("3.i= %d\n", i);
        }
        printf("2.i= %d\n", i);
    }
    printf("1.i= %d\n", i);
}
t.c " 51 lines, 506 characters
```

1st i's scope 2nd i's scope

The screenshot shows a terminal window with the following C code:

```
#include <stdio.h>
void func() {
    int i = 111;
    printf("1.i= %d\n", i);
    {
        int i = 222;
        printf("2.i= %d\n", i);
        {
            int i = 333;
            printf("3.i= %d\n", i);
        }
        printf("2.i= %d\n", i);
    }
    printf("1.i= %d\n", i);
}
```

The code demonstrates three levels of variable scoping for the variable `i`:

- Level 1 (Outermost): `i = 111` (blue box)
- Level 2 (Middle): `i = 222` (green box)
- Level 3 (Innermost): `i = 333` (yellow box)

Each `printf` statement outputs the value of the `i` variable from its respective scope. The output will be:

```
1.i= 111
2.i= 222
3.i= 333
2.i= 222
1.i= 111
```

1st i's scope
2nd i's scope
3rd i's scope

```
#include <stdio.h>

void func() {
    int i = 111;
    printf("1.i= %d\n", i);

    {
        int i = 222;
        printf("2.i= %d\n", i);

        {
            int i = 333;
            printf("3.i= %d\n", i);
        }
        printf("2.i= %d\n", i);
    }
    printf("1.i= %d\n", i);
}
```

separate compilation --- multiple-source program

t1.c

```
#include <stdio.h>

int a;

void func1(void) {
    puts("func1 is called");
}

void func3(void) ;

int main(void) {
    printf("a= %d\n", a);

    func1();
    // func2();
    func3();

}
```

gcc -c t1.c

=> t1.o

t2.c

```
#include <stdio.h>

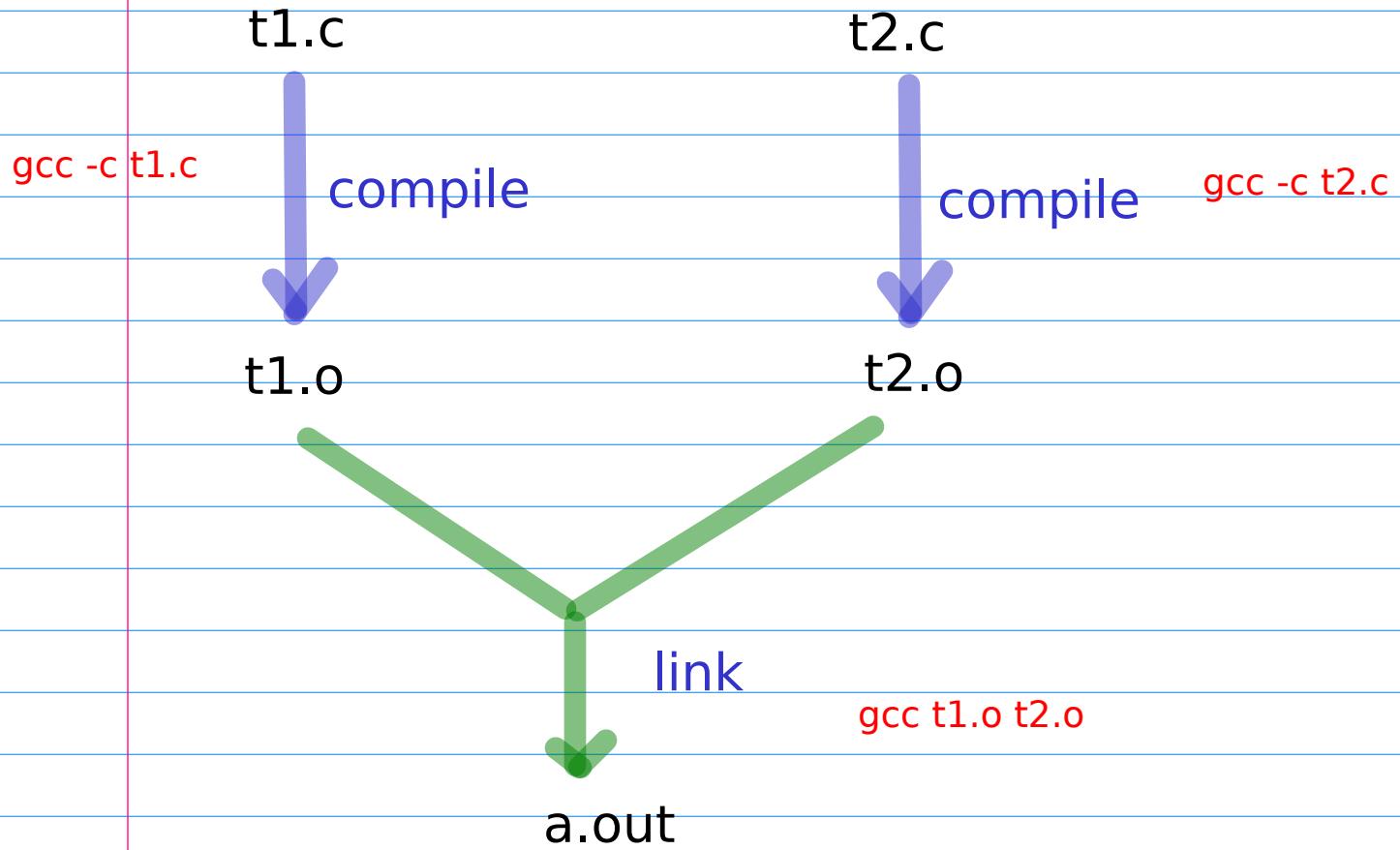
int a = 333;

static void func2(void) {
    puts("func2 is called");
}

void func3(void) {
    printf("func3: ");
    func2();
}
```

gcc -c t2.c

=> t2.o



t2.c

```
#include <stdio.h>

int a = 333;

static void func2(void) {
    puts("func2 is called");
}

void func3(void) {
    printf("func3: ");
    func2();
}
```

func2 can be used only within t2.c



useful to hide implementation details.

func2 has an internal linkage

thus, it cannot be called in main()
which is defined in t1.c

but func3 has an external linkage

and it is called in main()

func2 cannot be called by main() in t1.c
but can be called by func3() in t2.c

func2 is known only in t2.c
because of static keyword

static storage class example

```
#include <stdio.h>

void func() {
    static int i = 0;
    printf("i= %d\n", i);
    i++;
}

int main(void) {
    func();
    func();
    func();
    func();
}
```

local variable --> automatic storage

static local variable --> static storage