Functions and Program Structure

백 윤 철

Contents

- Basics of Function
- Functions Returning Non-Integers
- External Variables
- Scope Rules
- Header Flies
- Static Variables
- Register Variables
- Block Structure
- Initialization
- Recursion
- The C Preprocessor

Basics of Function

- Functions break large computing tasks into smaller ones
- and enable people to build on what others have done instead of starting over from scratch
- C programs generally consist of many small functions rather than a few big ones.
- Source files may be compiled separately and loaded together, along with previously compiled functions from libraries

Basics of Function

- Function definition

```
return-type  function-name(argument declarations)
{
        declarations and statements
}
```

- Return

```
return expression;
```

- Program searching for the pattern of letters "ould“ in the set of lines can be designed into three pieces

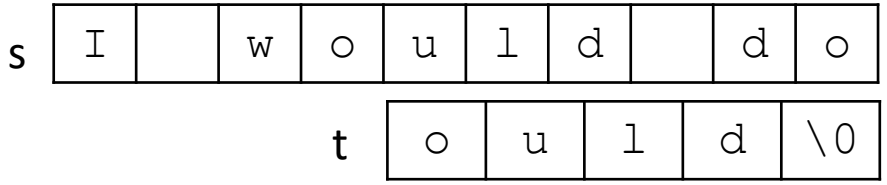
```
while (there's another line)
    if (the line contains the pattern)
        print it
```

Basics of Function

- Find all lines matching pattern

```
#include <stdio.h>
#define MAXLINE 1000    /* maximum input line length */

int getline(char line[], int max);
int strindex(char source[], char searchfor[]);

char pattern[] = "ould";    /* pattern to search for */

/* find all lines matching pattern */
main()
{
    char line[MAXLINE];
    int found = 0;

    while (getline(line, MAXLINE) > 0)
        if (strindex(line, pattern) >= 0) {
            printf("%s", line);
            found++;
        }
    return found;
}
```

Basics of Function



- getline() and strindex()

```
/* getline:  get line into s, return length */
int getline(char s[], int lim)
{
    int c, i;

    i = 0;
    while (--lim > 0 && (c=getchar()) != EOF && c != '\n')
        s[i++] = c;
    if (c == '\n')
        s[i++] = c;
    s[i] = '\0';
    return i;
}

/* strindex:  return index of t in s, -1 if none */
int strindex(char s[], char t[])
{
    int i, j, k;

    for (i = 0; s[i] != '\0'; i++) {
        for (j=i, k=0; t[k]!='\0' && s[j]==t[k]; j++, k++)
            ;
        if (k > 0 && t[k] == '\0')
            return i;
    }
    return -1;
}
```

Basics of Function

- Separate Compilation

```
$gcc main.c getline.c strindex.c
```

```
$gcc -c getline.c
$gcc -c strindex.c
$gcc -c main.c
$gcc main.o getline.o strindex.o
```

Functions Returning Non-Integers

1	2	.	3	4	\0

- atof()

```
#include <ctype.h>

/* atof:  convert string s to double */
double atof(char s[])
{
    double val, power;
    int i, sign;

    for (i = 0; isspace(s[i]); i++)  /* skip white space */
        ;
    sign = (s[i] == '-') ? -1 : 1;
    if (s[i] == '+' || s[i] == '-')
        i++;
    for (val = 0.0; isdigit(s[i]); i++)
        val = 10.0 * val + (s[i] - '0');
    if (s[i] == '.')
        i++;
    for (power = 1.0; isdigit(s[i]); i++) {
        val = 10.0 * val + (s[i] - '0');
        power *= 10.0;
    }
    return sign * val / power;
}
```

Functions Returning Non-Integers

- main calls atof()

```
#include <stdio.h>

#define MAXLINE 100

/* rudimentary calculator */
main()
{
    double sum, atof(char []);
    char line[MAXLINE];
    int getline(char line[], int max);

    sum = 0;
    while (getline(line, MAXLINE) > 0)
        printf("\t%g\n", sum += atof(line));
    return 0;
}
```

Functions Returning Non-Integers

- atoi() using atof()

```
/* atoi:  convert string s to integer using atof */
int atoi(char s[])
{
    double atof(char s[]);

    return (int) atof(s);
}
```

External Variables

- External variables are defined outside of any function, and are thus potentially available to many functions
- Functions themselves are always external, because C does not allow functions to be defined inside other functions.
- If a large number of variables must be shared among functions, external variables are more convenient and efficient than long argument lists.

External Variables

- In reverse Polish notation, each operator follows its operand
  - Parentheses are not needed
  - the notation is unambiguous as long as we know how many operands each operator expects.
  - The implementation is simple.

```
1 2 - 4 5 + *
```

- infix notation

```
(1 - 2) * (4 + 5)
```

External Variables

- postfix calculator implementation
  - Each operand is pushed onto a stack
  - when an operator arrives, the proper number of operands (two for binary operators) is popped
  - the operator is applied to them
  - and the result is pushed back onto the stack.

```
while (next operator or operand is not end-of-file indicator)
    if (number)
        push it
    else if (operator)
        pop operands
        do operation
        push result
    else if (newline)
        pop and print top of stack
    else
        error
```

External Variables

- main of calculator

```
#include <stdio.h>
#include <stdlib.h>   /* for atof() */

#define MAXOP   100   /* max size of operand or operator */
#define NUMBER  '0'   /* signal that a number was found */

int getop(char []);
void push(double);
double pop(void);

/* reverse Polish calculator */
main()
{
    int type;
    double op2;
    char s[MAXOP];

    while ((type = getop(s)) != EOF) {
        switch (type) {
        case NUMBER:
            push(atof(s));
            break;
        case '+':
            push(pop() + pop());
            break;
        case '*':
            push(pop() * pop());
            break;
        case '-':
            op2 = pop();
            push(pop() - op2);
            break;
        case '/':
            op2 = pop();
            if (op2 != 0.0)
                push(pop() / op2);
            else
                printf("error: zero divisor\n");
            break;
        case '\n':
            printf("\t%.8g\n", pop());
            break;
        default:
            printf("error: unknown command %s\n", s);
            break;
        }
    }
    return 0;
}
```

External Variables

- push and pop

```
#define MAXVAL   100    /* maximum depth of val stack */

int sp = 0;             /* next free stack position */
double val[MAXVAL];     /* value stack */

/* push:  push f onto value stack */
void push(double f)
{
    if (sp < MAXVAL)
        val[sp++] = f;
    else
        printf("error: stack full, can't push %g\n", f);
}



/* pop:  pop and return top value from stack */
double pop(void)
{
    if (sp > 0)
        return val[--sp];
    else {
        printf("error: stack empty\n");
        return 0.0;
    }
}
```

External Variables

- getop

```
#include <ctype.h>

int getch(void);
void ungetch(int);

/* getop:  get next operator or numeric operand */
int getop(char s[])
{
    int i, c;

    while ((s[0] = c = getch()) == ' ' || c == '\t')
        ;
    s[1] = '\0';
    if (!isdigit(c) && c != '.')
        return c;     /* not a number */
    i = 0;
    if (isdigit(c))   /* collect integer part */
        while (isdigit(s[++i] = c = getch()))
            ;
    if (c == '.')     /* collect fraction part */
        while (isdigit(s[++i] = c = getch()))
            ;
    s[i] = '\0';
    if (c != EOF)
        ungetch(c);
    return NUMBER;
}
```

External Variables

- getch and ungetch

```
#define BUFSIZE 100

char buf[BUFSIZE];  /* buffer for ungetch */
int  bufp = 0;      /* next free position in buf */

int getch(void) /* get a (possibly pushed back) character */
{
    return (bufp > 0) ? buf[--bufp] : getchar();
}

void ungetch(int c) /* push character back on input */
{
    if (bufp >= BUFSIZE)
        printf("ungetch: too many characters\n");
    else
        buf[bufp++] = c;
}
```

Scope Rule

- an automatic variable declared at the beginning of a function
- the scope is the function in which the name is declared.
- The scope of an external variable or a function lasts from the point at which it is declared to the end of the file being compiled
- A declaration announces the properties of a variable (primarily its type)
- a definition also causes storage to be set aside.

Scope Rule

- extern declaration

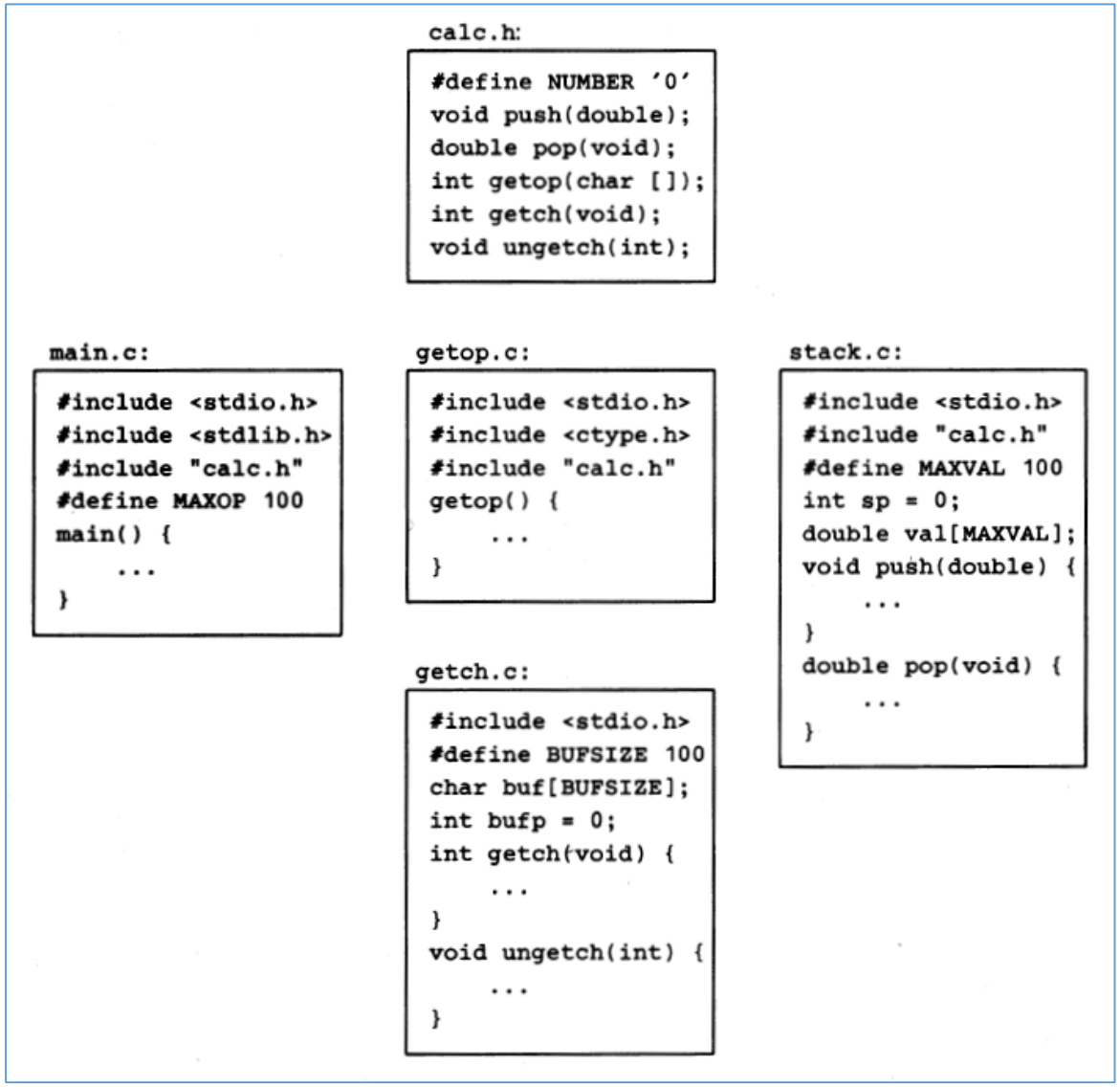
In file1:

```
extern int sp;
extern double val[];

void push(double f) { ... }

double pop(void) { ... }
```

In file2:

```
int sp = 0;
double val[MAXVAL];
```

Header Files



calc.h:

```
#define NUMBER '0'
void push(double);
double pop(void);
int getop(char []);
int getch(void);
void ungetch(int);
```

main.c:

```
#include <stdio.h>
#include <stdlib.h>
#include "calc.h"
#define MAXOP 100
main() {
    ...
}
```

getop.c:

```
#include <stdio.h>
#include <ctype.h>
#include "calc.h"
getop() {
    ...
}
```

stack.c:

```
#include <stdio.h>
#include "calc.h"
#define MAXVAL 100
int sp = 0;
double val[MAXVAL];
void push(double) {
    ...
}
double pop(void) {
    ...
}
```

getch.c:

```
#include <stdio.h>
#define BUFSIZE 100
char buf[BUFSIZE];
int bufp = 0;
int getch(void) {
    ...
}
void ungetch(int) {
    ...
}
```

Static Variables

정적변수 (기억클)
함수내외 사용 가능

- The static declaration, applied to an external variable or function, limits the scope of that object to the rest of the source file being compiled.
- External static thus provides a way to hide names
- Internal static variables provide private, permanent storage within a single function.

```
static char buf[BUFSIZE];  /* buffer for ungetch */
static int  bufp = 0;      /* next free position in buf */

int getch(void) { ... }

void ungetch(int c) { ... }
```

Register Variables

- A register declaration advises the compiler that the variable in question will be heavily used.
- The idea is that register variables are to be placed in machine registers
- which may result in smaller and faster programs.
- But compilers are free to ignore the advice.

```
f(register unsigned m, register long n)
{
    register int i;
    ...
}
```

Block Structure

- C is not a block-structured language in the sense of Pascal or similar languages, because functions may not be defined within other functions.
- On the other hand, variables can be defined in a block-structured fashion within a function

```
if (n > 0) {
    int i;  /* declare a new i */

    for (i = 0; i < n; i++)
        ...
}
```

```
int x;
int y;

f(double x)
{
    double y;
    ...
}
```

Initialization

- external and static variables are guaranteed to be initialized to zero
- automatic and register variables have undefined (i.e., garbage) initial values.
- For external and static variables, the initializer must be a constant expression;
- the initialization is done once, conceptually before the program begins execution.
- For automatic and register variables, it is done each time the function or block is entered.
- For automatic and register variables, the initializer is not restricted to being a constant:

Initialization

```
int binsearch(int x, int v[], int n)
{
    int low = 0;
    int high = n - 1;
    int mid;
    ...
}
```

```
char pattern[] = "ould";
```

is a shorthand for the longer but equivalent

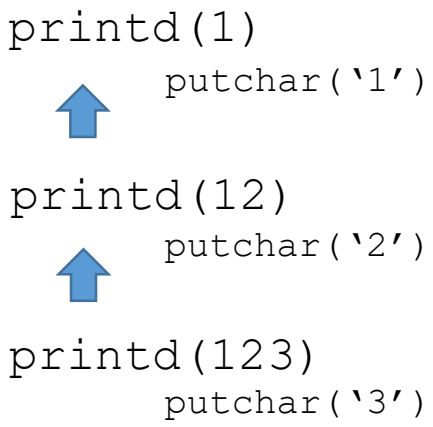
```
char pattern[] = { 'o', 'u', 'l', 'd', '\0' };
```

Recursion 재귀

- a function may call itself either directly or indirectly.

```
#include <stdio.h>

/* printd:  print n in decimal */
void printd(int n)
{
    if (n < 0) {
        putchar('-');
        n = -n;
    }
    if (n / 10)
        printd(n / 10);
    putchar(n % 10 + '0');
}
```

재귀



Recursion

- quick sort

```
/* qsort:  sort v[left]...v[right] into increasing order */
void qsort(int v[], int left, int right)
{
    int i, last;
    void swap(int v[], int i, int j);

    if (left >= right)    /* do nothing if array contains */
        return;           /* fewer than two elements */
    swap(v, left, (left + right)/2); /* move partition elem */
    last = left;                     /* to v[0] */
    for (i = left+1; i <= right; i++)    /* partition */
        if (v[i] < v[left])
            swap(v, ++last, i);
    swap(v, left, last);          /* restore partition elem */
    qsort(v, left, last-1);
    qsort(v, last+1, right);
}
```

Recursion

- swap

```
/* swap:  interchange v[i] and v[j] */
void swap(int v[], int i, int j)
{
     int temp;

     temp = v[i];
     v[i] = v[j];
     v[j] = temp;
}
```

Recursion

- Recursion may provide no saving in storage, since somewhere a stack of the values being processed must be maintained.
- Nor will it be faster.
- But recursive code is more compact, and often much easier to write and understand than the non-recursive equivalent.

The C Preprocessor

- conceptually a separate first step in compilation
- File Inclusion
  - to include the contents of a file during compilation

```
#include <filename>
#include “filename”
```

  - If the *filename* is quoted, searching for the file typically begins where the source program was found
  - if it is not found there, or if the name is enclosed in < and >, searching follows an implementation-defined rule to find the file.
  - It guarantees that all the source files will be supplied with the same definitions and variable declarations

The C Preprocessor

- macro substitution

```
#define   name   replacement text
```

  - The scope of a name defined with #define is from its point of definition to the end of the source file being compiled

```
#define max(A, B) ((A) > (B) ? (A) : (B))

x = max(p+q, r+s);
x = ((p+q) > (r+s) ? (p+q) : (r+s));
```

```
#max(i++, j++)   /* WRONG */
#define square(x) x * x   /* WRONG */
```

The C Preprocessor

- macro substitution
  - One practical example comes from <stdio. h>, in which getchar and putchar are often defined as macros to avoid the run-time overhead of a function call per character processed.
  - The functions in <ctype. h> are also usually implemented as macros.
- Names may be undefined with #undef, usually to ensure that a routine is really a function, not a macro

```
#undef getchar

int getchar(void) { ... }
```

The C Preprocessor

- conditional inclusion
  - #if and defined()

```
#if !defined(HDR)
#define HDR

/* contents of hdr.h go here */

#endif
```

  - #elif, #endif

```
#if SYSTEM == SYSV
    #define HDR "sysv.h"
#elif SYSTEM == BSD
    #define HDR "bsd.h"
#elif SYSTEM == MSDOS
    #define HDR "msdos.h"
#else
    #define HDR "default.h"
#endif
#include HDR
```

The C Preprocessor

- conditional inclusion
  - #ifdef, #ifndef

값이 X면은

그냥넘어가기 처음

```
#ifndef HDR
#define HDR

/* contents of hdr.h go here */

#endif
```

정리

- Basics of Function
- Functions Returning Non-Integers
- External Variables
- Scope Rules
- Header Flies
- Static Variables
- Register Variables
- Block Structure
- Initialization
- Recursion
- The C Preprocessor

끝