“Functions”

Using Bloodshed Dev-C++

Heejin Park

Hanyang University

Introduction

- Reviewing Functions
- ANSI C Function Prototyping
- Recursion
- Compiling Programs with Two or More Source Code Files
- Finding Addresses: The & Operator
- Altering Variables in the Calling Function
- Pointers: A First Look

Reviewing Functions

What is a function?

- A function is a self-contained unit of program code designed to accomplish a particular task.
- Why should you use functions?
  - For one, they save you from repetitious programming.

- **Using a function** is worthwhile
  - It makes a program more modular.
  - Easier to read and easier to change or fix.

Reviewing Functions

What is a function?

- Suppose
- You want to write a program that does the following:
  - Read in a list of numbers.
  - Sort the numbers.
  - Find their average.
  - Print a bar graph.

Reviewing Functions

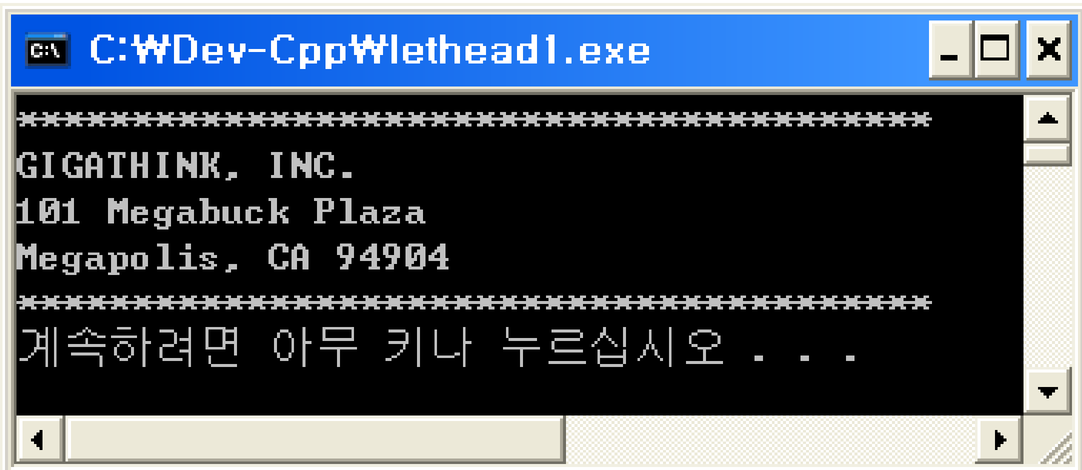
What is a function?

- You could use this program:

```
#include <stdio.h>
#define SIZE 50

int main(void)
{
  float list[SIZE];

  readlist(list, SIZE);
  sort(list, SIZE);
  average(list, SIZE);
  bargraph(list, SIZE);
  return 0;
}
```

Reviewing Functions

The lethead1.c Program



Reviewing Functions

The lethead1.c Program

```
#include <stdio.h>
#define NAME "GIGATHINK, INC."
#define ADDRESS "101 Megabuck Plaza"
#define PLACE "Megapolis, CA 94904"
#define WIDTH 40

void starbar(void);   /* prototype the function */

int main(void)
{
    starbar();
    printf("%s\n", NAME);
    printf("%s\n", ADDRESS);
    printf("%s\n", PLACE);
    starbar();        /* use the function       */
    return 0;
}
void starbar(void)    /* define the function    */
{
    int count;
    for (count = 1; count <= WIDTH; count++)
        putchar('*');
    putchar('\n');
}
```

Reviewing Functions

The lethead1.c Program

- Several major points
- **It uses the `starbar` identifier in three separate contexts:**
  - a ***function prototype*** that tells the compiler what sort of function `starbar()` is,
  - a ***function call*** that causes the function to be executed,
  - a ***function definition*** that specifies exactly what the function does.

Reviewing Functions

The lethead1.c Program

- Several major points
- **Like variables, functions have types.**
  - Any program that uses a function should declare the type for that function before it is used.
  - Consequently, this ANSI C prototype precedes the `main()` function definition:

```
void starbar(void);
```

Reviewing Functions

The lethead1.c Program

- For compilers that don't recognize ANSI C prototyping, just declare the type, as follows:

```
void starbar();
```

Reviewing Functions

The lethead1.c Program

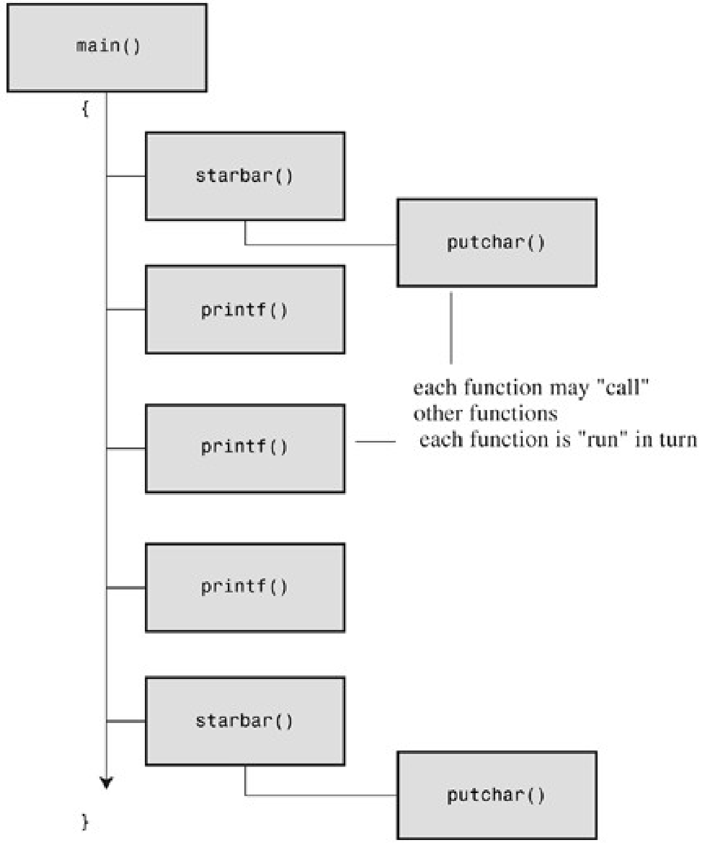
- The program calls the function `starbar()` from `main()` by using its name followed by parentheses and a semicolon.
- thus creating the statement

```
starbar();
```

Reviewing Functions

The lethead1.c Program

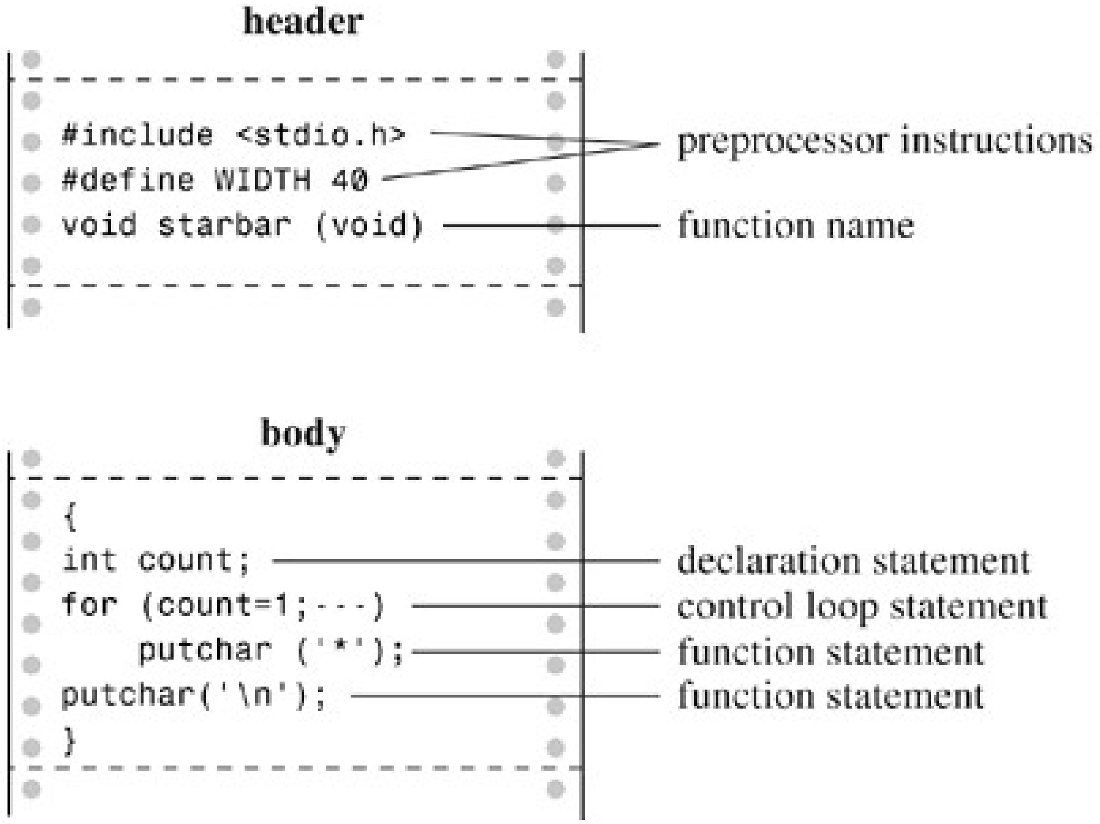
- **Control flow for lethead1.c**



Reviewing Functions

The lethead1.c Program

- **Structure of a simple function**



Reviewing Functions

The lethead2.c Program(1/2)

- **Function Arguments**

```
#include <stdio.h>
#include <string.h>            /* for strlen() */
#define NAME "GIGATHINK, INC."
#define ADDRESS "101 Megabuck Plaza"
#define PLACE "Megapolis, CA 94904"
#define WIDTH 40
#define SPACE ' '

void show_n_char(char ch, int num);

int main(void){
    int spaces;

    show_n_char('*', WIDTH);   /* using constants as arguments */
    putchar('\n');
    show_n_char(SPACE, 12);    /* using constants as arguments */
    printf("%s\n", NAME);
    spaces = (WIDTH - strlen(ADDRESS)) / 2;
    show_n_char(SPACE, spaces);/* use a variable as argument   */
    printf("%s\n", ADDRESS);
    show_n_char(SPACE, (WIDTH - strlen(PLACE)) / 2);
    printf("%s\n", PLACE);
    show_n_char('*', WIDTH);
    putchar('\n');

    return 0;
}
```

Reviewing Functions

The lethead2.c Program(2/2)

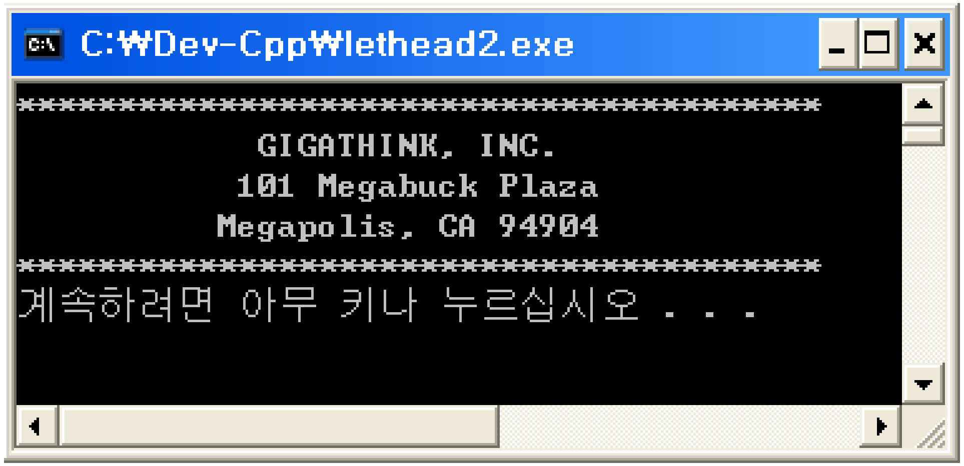
- **Function Arguments**

```
void show_n_char(char ch, int num)
{
    int count;

    for (count = 1; count <= num; count++)
        putchar(ch);
}
```

Reviewing Functions

The lethead2.c Program

- **Function Arguments**



Reviewing Functions

Defining a Function with an Argument: Formal Parameters

- The function definition begins with the following ANSI C function header:

```
void show_n_char(char ch, int num);
```

Reviewing Functions

Defining a Function with an Argument: Formal Parameters

- Note that the ANSI C form requires that each variable be preceded by its type.

```
void dibs(int x, y, z)          /* invalid function header */

void dubs(int x, int y, int z)  /* valid function header   */
```

Reviewing Functions

Defining a Function with an Argument: Formal Parameters

- ANSI C also recognizes the pre-ANSI form but characterizes it as obsolescent:

```
void show_n_char(ch, num)

char ch;

int num;
```

Reviewing Functions

Defining a Function with an Argument: Formal Parameters

- Here, the parentheses contain the list of argument names, but the types are declared afterward.
- This form does enable you to use comma-separated lists of variable names if the variables are of the same type, as shown here:

```
void dibs(x, y, z)

int x, y, z;                /* valid */
```

Reviewing Functions

Prototyping a Function with Arguments

- We used an ANSI prototype to declare the function before it is used:

```
void show_n_char(char ch, int num);
```

- When a function takes arguments, the prototype indicates their number and type by using a comma-separated list of the types.
- If you like, you can omit variable names in the prototype:

```
void show_n_char(char, int);
```

Reviewing Functions

Prototyping a Function with Arguments

- Again, ANSI C also recognizes the older form of declaring a function, which is without an argument list:

```
void show_n_char();
```

Reviewing Functions

Calling a Function with an Argument: Actual Arguments

- You give `ch` and `num` values by using ***actual arguments*** in the function call.
- Consider the first use of `show_n_char()`:

```
show_n_char(SPACE, 12);
```

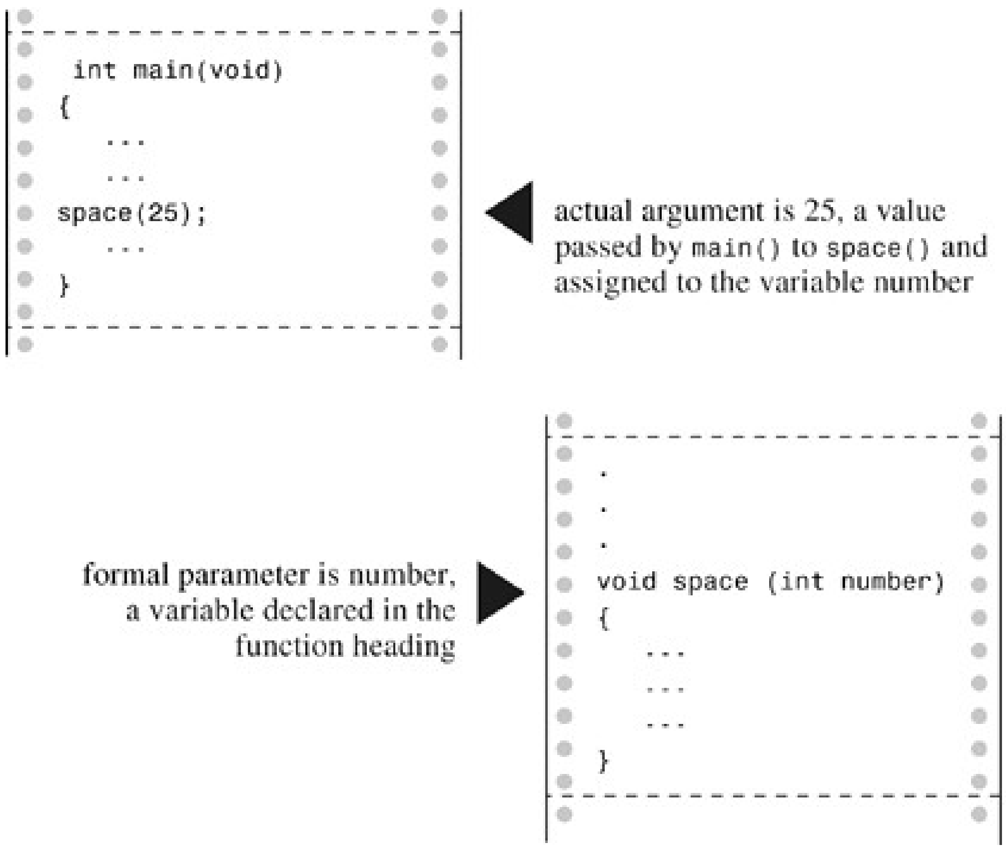
- consider the final use of `show_n_char()`:

```
show_n_char(SPACE, (WIDTH - strlen(PLACE)) / 2);
```

Reviewing Functions

Calling a Function with an Argument: Actual Arguments

- **Formal parameters and actual arguments**



Reviewing Functions

The Black-Box Viewpoint

- Taking a black-box viewpoint of **`show_n_char()`**,
- the input is the character to be displayed and the number of spaces to be skipped.
- The resulting action is printing the character the specified number of times.

- The fact that **`ch`**, **`num`**, and **`count`**
- local variables private to the **`show_n_char()`** function is an essential aspect of the black box approach.

Reviewing Functions

Listing 9.3. The lesser.c Program(1/2)

- **Returning a Value from a Function with return**

```
#include <stdio.h>

int imin(int, int);
int main(void)
{
    int evil1, evil2;

    printf("Enter a pair of integers (q to quit):\n");
    while (scanf("%d %d", &evil1, &evil2) == 2)
    {
        printf("The lesser of %d and %d is %d.\n",
            evil1, evil2, imin(evil1,evil2));
        printf("Enter a pair of integers (q to quit):\n");
    }
    printf("Bye.\n");
    return 0;
}
```

Reviewing Functions

Listing 9.3. The lesser.c Program(2/2)

- **Returning a Value from a Function with return**

```
int imin(int n,int m)
{
    int min;

    if (n < m)
        min = n;
    else
        min = m;
    return min;
}
```

Reviewing Functions

Listing 9.3. The lesser.c Program

- **Returning a Value from a Function with return**

```
C:₩Dev-Cpp₩lesser.exe
Enter a pair of integers (q to quit):
509 333
The lesser of 509 and 333 is 333.
Enter a pair of integers (q to quit):
-9393 6
The lesser of -9393 and 6 is -9393.
Enter a pair of integers (q to quit):
q
Bye.
계속하려면 아무 키나 누르십시오 . . .
```

Reviewing Functions

Listing 9.3. The lesser.c Program

- The variable `min` is private to `imin()`, but the value of `min` is communicated back to the calling function with `return`.
- The effect of a statement such as the next one is to assign the value of `min` to `lesser`:

```
lesser = imin(n,m);
```

Reviewing Functions

Listing 9.3. The lesser.c Program

- Could you say the following instead?

```
imin(n,m);

lesser = min;
```

Reviewing Functions

Listing 9.3. The lesser.c Program

- No, because the calling function doesn't even know that `min` exists.
- Remember that `imin()`'s variables are local to `imin()`.

- Not only can the returned value be assigned to a variable.
- it can also be used as part of an expression.

```
answer = 2 * imin(z, zstar) + 25;

printf("%d\n", imin(-32 + answer, LIMIT));
```

Reviewing Functions

Listing 9.3. The lesser.c Program

- The return value can be supplied by any expression, not just a variable.
- For example, you can shorten the program to the following:

```
/* minimum value function, second version */

imin(int n,int m)
{
      return (n < m) ? n : m;
}
```

Reviewing Functions

Listing 9.3. The lesser.c Program

- What if the function returns a type different from the declared type?

```
int what_if(int n)
{
    double z = 100.0 / (double) n;
    return z;  // what happens?
}
```

Reviewing Functions

Listing 9.3. The lesser.c Program

- Ex) suppose we have the following function call:

```
result = what_if(64);
```

- Then `z` is assigned `1.5625`.
- The return statement, however, returns the `int` value `1`.

Reviewing Functions

Listing 9.3. The lesser.c Program

- Using `return` has one other effect.
- It terminates the function and returns control to the next statement in the calling function.

```
/* minimum value function, third version */

imin(int n,int m)
{
    if (n < m)
        return n;
    else
        return m;
}
```

Reviewing Functions

Listing 9.3. The lesser.c Program

- Even this version works the same:

```
/* minimum value function, fourth version */

imin(int n, int m)
{
    if (n < m)
        return n;
    else
        return m;
    printf("Professor Fleppard is like totally a fopdoodle.\n");
}
```

Reviewing Functions

Listing 9.3. The lesser.c Program

- The `return` statements prevent the `printf()` statement from ever being reached.
- You can also use a statement like this:

```
return;
```

Reviewing Functions

Function Types

- Functions should be declared by type.
- The type declaration is part of the function definition.
- Keep in mind that it refers to the return value, not to the function arguments.

```
double klink(int a, int b)
```

Reviewing Functions

Function Types

- You generally inform the compiler about functions by declaring them in advance
- Ex) the `main()` function in The lesser.c program contains these lines:

```
#include <stdio.h>

int imin(int, int);
int main(void)
{
    int evil1, evil2, lesser;
```

Reviewing Functions

Function Types

- Can also be placed inside the function.
- Ex) you can rewrite the beginning of lesser.c as follows:

```
#include <stdio.h>

int main(void)
{
    int imin(int, int);        /* imin() declaration */
    int evil1, evil2, lesser;
```

Reviewing Functions

The math.h header

- contains function declarations for a variety of mathematical functions.
- Ex)

```
double sqrt(double);
```

  – it contains to tell the compiler that the `sqrt()` function returns a type double value.

ANSI C Function Prototyping

pre-ANSI C scheme

- The following pre-ANSI declaration informs the compiler that `imin()` returns a type `int` value:

```
int imin();
```

- However, it says nothing about the number or type of `imin()`'s arguments.

ANSI C Function Prototyping

The misuse.c Program

```
#include <stdio.h>

int imax();       /* old-style declaration */

int main(void)
{
    printf("The maximum of %d and %d is %d.\n",
            3, 5, imax(3));
    printf("The maximum of %d and %d is %d.\n",
            3, 5, imax(3.0, 5.0));
    return 0;
}

int imax(n, m)
int n, m;
{
    int max;
    if (n > m)
        max = n;
    else
        max = m;
    return max;
}
```

ANSI C Function Prototyping

The misuse.c Program

- The first call to `printf()` omits an argument to `imax()`.
- The second call uses floating-point arguments instead of integers.
- Despite these errors, the program compiles and runs.

ANSI C Function Prototyping

The misuse.c Program

- Here's the output using Metrowerks Codewarrior Development Studio 9:

```
The maximum of 3 and 5 is 1245120.
The maximum of 3 and 5 is 1074266112.
```

- Digital Mars 8.4 produced values of 4202837 and 1074266112.
- The two compilers work fine.
- They are merely victims of the program's failure to use function prototypes.

ANSI C Function Prototyping

The ANSI Solution

- **Function prototype**
- a declaration that states the return type, the number of arguments, and the types of those arguments.
  - To indicate that `imax()` requires two `int` arguments, you can declare it with either of the following prototypes:

```
int imax(int, int);

int imax(int a, int b);
```

ANSI C Function Prototyping

The proto.c Program

```
#include <stdio.h>

int imax(int, int);          /* prototype */
int main(void)
{
    printf("The maximum of %d and %d is %d.\n",
            3, 5, imax(3));
    printf("The maximum of %d and %d is %d.\n",
            3, 5, imax(3.0, 5.0));
    return 0;
}

int imax(int n, int m)
{
    int max;
    if (n > m)
        max = n;
    else
        max = m;
    return max;
}
```

ANSI C Function Prototyping

The proto.c Program

- What about the type errors?

```
int imax(int, int);          /* prototype */
int main(void)
{
    printf("The maximum of %d and %d is %d.\n",
            3, 5, imax(3));
    printf("The maximum of %d and %d is %d.\n",
            3, 5, imax(3.0, 5.0));

    system("pause");
    return 0;
```

Too few arguments to function ‘imax’

ANSI C Function Prototyping

The proto.c Program

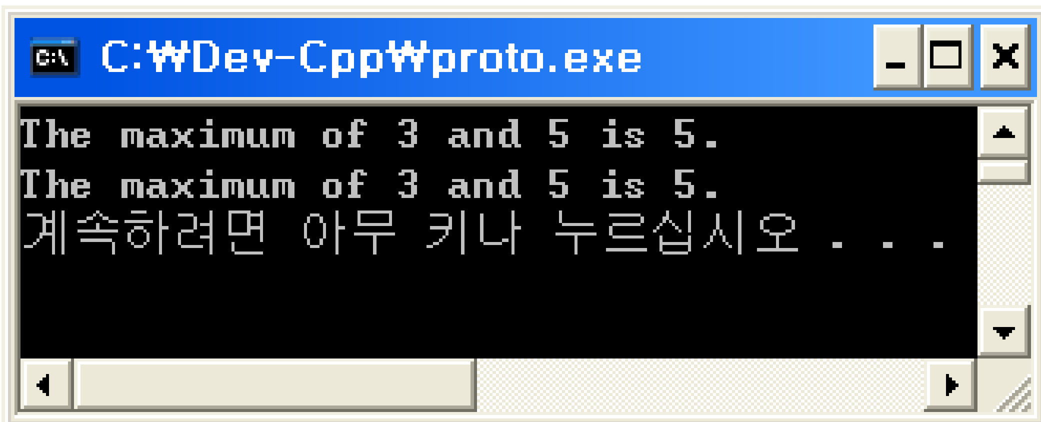
- To investigate those, we replaced `imax(3)` with `imax(3,5)`and tried compilation again.

```
int imax(int, int);          /* prototype */
int main(void)
{
    printf("The maximum of %d and %d is %d.\n",
           3, 5, imax(3, 5));
    printf("The maximum of %d and %d is %d.\n",
           3, 5, imax(3.0, 5.0));

    system("pause");
    return 0;
```

ANSI C Function Prototyping

The proto.c Program

- This time there were no error messages, and we ran the program.



ANSI C Function Prototyping

The proto.c Program

- Compiler did give a warning to the effect that a `double` was converted to `int` and that there was a possible loss of data.

```
imax(3.9, 5.4)
```

- The call becomes equivalent to the following:

```
imax(3, 5)
```

ANSI C Function Prototyping

No Arguments and Unspecified Arguments

- Suppose you give a prototype like this:

```
void print_name();
```

- To indicate that a function really has no arguments, use the void keyword within the parentheses:

```
void print_name(void);
```

ANSI C Function Prototyping

No Arguments and Unspecified Arguments

- ANSI C allows partial prototyping for such cases.
- You could, for example, use this prototype for `printf()`:

```
int printf(char *, ...);
```

ANSI C Function Prototyping

Hooray for Prototypes

- You can accomplish the same end by placing the entire function definition before the first use.
- Then the definition acts as its own prototype.

```
// the following is a definition and a prototype

int imax(int a, int b) { return a > b ? a : b; }
int main()
{
...
    z = imax(x, 50);
...
}
```

Recursion

Recursion

- C permits a function to call itself.
- This process is termed ***recursion***.
- Recursion is a sometimes tricky, sometimes convenient tool.
- Recursion often can be used where loops can be used.

Recursion

The recur.c Program

```c
#include <stdio.h>

void up_and_down(int);

int main(void)
{
    up_and_down(1);
    return 0;
}

void up_and_down(int n)
{
    printf("Level %d: n location %p\n", n, &n); /* 1 */
    if (n < 4)
         up_and_down(n+1);
    printf("LEVEL %d: n location %p\n", n, &n); /* 2 */
}
```

Recursion

The recur.c Program

```
C:₩Dev-Cpp₩recur.exe
```

```
Level 1: n location 0022FF60
Level 2: n location 0022FF40
Level 3: n location 0022FF20
Level 4: n location 0022FF00
LEVEL 4: n location 0022FF00
LEVEL 3: n location 0022FF20
LEVEL 2: n location 0022FF40
LEVEL 1: n location 0022FF60
계속하려면 아무 키나 누르십시오 . . .
```

Recursion

Recursion Fundamentals

- **Recursion variables**

variables:	n	n	n	n
after level 1 call	1			
after level 2 call	1	2		
after level 3 call	1	2	3	
after level 4 call	1	2	3	4
after return from level 4	1	2	3	
after return from level 3	1	2		
after return from level 2	1			
after return from level 1				

(all gone)

Recursion

Tail Recursion

- In the simplest form of recursion, the recursive call is at the end of the function, just before the `return` statement.
- This is called ***tail recursion*** or ***end recursion.***
  - because the recursive call comes at the end.

Recursion

The factor.c Program(1/2)

```
#include <stdio.h>

long fact(int n);
long rfact(int n);
int main(void)
{
    int num;

    printf("This program calculates factorials.\n");
    printf("Enter a value in the range 0-12 (q to quit):\n");
    while (scanf("%d", &num) == 1)
    {
        if (num < 0)
            printf("No negative numbers, please.\n");
        else if (num > 12)
            printf("Keep input under 13.\n");
        else
        {
            printf("loop: %d factorial = %ld\n",
                    num, fact(num));
            printf("recursion: %d factorial = %ld\n",
                    num, rfact(num));
        }
        printf("Enter a value in the range 0-12 (q to quit):\n");
    }
    printf("Bye.\n");
    return 0;
}
```

Recursion

The factor.c Program(2/2)

```c
long fact(int n)       // loop-based function
{
    long ans;
    for (ans = 1; n > 1; n--)
        ans *= n;
    return ans;
}

long rfact(int n)      // recursive version
{
    long ans;
    if (n > 0)
        ans= n * rfact(n-1);
    else
        ans = 1;
    return ans;
}
```

Recursion

The factor.c Program

```
C:\Dev-Cpp\factor.exe
This program calculates factorials.
Enter a value in the range 0-12 (q to quit):
5
loop: 5 factorial = 120
recursion: 5 factorial = 120
Enter a value in the range 0-12 (q to quit):
10
loop: 10 factorial = 3628800
recursion: 10 factorial = 3628800
Enter a value in the range 0-12 (q to quit):
q
Bye.
계속하려면 아무 키나 누르십시오 . . .
```

Recursion

Recursion and Reversal

- The problem is this:
- Write a function that prints the binary equivalent of an integer.
- Binary notation represents numbers in terms of powers of 2.
- Just as 234 in decimal means **2 x 102 + 3 x 101 + 4 x 100**.
  - so 101 in binary means **1 x 22 + 0 x 21 + 1 x 20**.
- Binary numbers use only the digits 0 and 1.

Recursion

The binary.c Program(1/2)

```
#include <stdio.h>

void to_binary(unsigned long n);

int main(void)
{
    unsigned long number;
    printf("Enter an integer (q to quit):\n");

    while (scanf("%ul", &number) == 1)
    {
        printf("Binary equivalent: ");
        to_binary(number);
        putchar('\n');
        printf("Enter an integer (q to quit):\n");
    }
    printf("Done.\n");

   return 0;
}
```

Recursion

The binary.c Program(2/2)

```
void to_binary(unsigned long n)    /* recursive function */
{
    int r;

    r = n % 2;
    if (n >= 2)
       to_binary(n / 2);
    putchar('0' + r);

    return;
}
```

Recursion

The binary.c Program

- The expression `'0' + r` evaluates to the character `'0'`, if `r` is `0`, and to the character `'1'`, if `r` is `1`.
- This assumes that the numeric code for the `'1'` character is one greater than the code for the `'0'` character.
- More generally, you could use the following approach:

```
putchar( r ? '1' : '0');
```

Recursion

The binary.c Program

```
C:\Dev-Cpp\binary.exe
Enter an integer (q to quit):
9
Binary equivalent: 1001
Enter an integer (q to quit):
255
Binary equivalent: 11111111
Enter an integer (q to quit):
1024
Binary equivalent: 10000000000
Enter an integer (q to quit):
q
Done.
계속하려면 아무 키나 누르십시오 . . .
```

Recursion

Recursion Pros and Cons

- **Fibonacci numbers** can be defined as follows:
- The first Fibonacci number is 1
- The second Fibonacci number is 1
- Each subsequent Fibonacci number is the sum of the preceding two.
  - Therefore, the first few numbers in the sequence are **1, 1, 2, 3, 5, 8, 13**.

Recursion

Recursion Pros and Cons

- If we name the function `Fibonacci()`,
- `Fibonacci(n)` should return `1` if `n` is `1` or `2`.
- It should return the sum `Fibonacci(n-1)` + `Fibonacci(n-2)` otherwise:

```
long Fibonacci(int n)
{
    if (n > 2)
        return Fibonacci(n-1) + Fibonacci(n-2);
    else
        return 1;
}
```

Compiling Programs with Two or More Source Code Files

Unix

- This assumes the Unix system has the standard Unix C compiler `cc` installed.
- Suppose
- `file1.c` and `file2.c` are two files containing C functions.
- Then the following command will compile both files and produce an executable file called `a.out`:

```
cc file1.c file2.c
```

Compiling Programs with Two or More Source Code Files

Unix

- In addition, two object files called `file1.o` and `file2.o` are produced.
- If you later change `file1.c` but not `file2.c`,
- you can compile the first and combine it with the object code version of the second file by using this command:

```
cc file1.c file2.o
```

Compiling Programs with Two or More Source Code Files

Linux

- This assumes the Linux system has the GNU C compiler gcc installed.
- Suppose
- `file1.c` and `file2.c` are two files containing C functions.
- Then the following command will compile both files and produce an executable file called `a.out`:

```
gcc file1.c file2.c
```

Compiling Programs with Two or More Source Code Files

Linux

- In addition, two object files called `file1.o` and `file2.o` are produced.
- If you later change `file1.c` but not `file2.c`,
- you can compile the first and combine it with the object code version of the second file by using this command:

```
gcc file1.c file2.o
```

Compiling Programs with Two or More Source Code Files

DOS Command-Line Compilers

- Most DOS command-line compilers work similarly to the Unix `cc` command.
- One difference is that object files wind up with an **`.obj`** extension instead of an `.o` extension.

Compiling Programs with Two or More Source Code Files

Windows and Macintosh Compilers

- Windows and Macintosh compilers are project oriented.
- A project describes the resources a particular program uses.
- The resources include your source code files.

Compiling Programs with Two or More Source Code Files

Using Header Files

- If you put `main()` in one file and your function definitions in a second file,
- The first file still needs the function prototypes.
- Rather than type them in each time you use the function file,
  - You can store the function prototypes in a header file.
- Place the `#define` directives in a header file and then use the `#include` directive in each source code file.

Compiling Programs with Two or More Source Code Files

The usehotel.c Control Module(1/2)

```
#include <stdio.h>
#include "hotel.h" /* defines constants, declares functions */

int main(void)
{
    int nights;
    double hotel_rate;
    int code;
```

Compiling Programs with Two or More Source Code Files

The usehotel.c Control Module(2/2)

```
   while ((code = menu()) != QUIT)
   {
      switch(code)
      {
      case 1 : hotel_rate = HOTEL1;
               break;
      case 2 : hotel_rate = HOTEL2;
               break;
      case 3 : hotel_rate = HOTEL3;
               break;
      case 4 : hotel_rate = HOTEL4;
               break;

      default: hotel_rate = 0.0;
               printf("Oops!\n");
               break;
      }
      nights = getnights();
      showprice(hotel_rate, nights);
   }
   printf("Thank you and goodbye.");
   return 0;
}
```

Compiling Programs with Two or More Source Code Files

The hotel.c Function Support Module(1/2)

```
#include <stdio.h>
#include "hotel.h"

int menu(void)
{
    int code, status;

    printf("\n%s%s\n", STARS, STARS);
    printf("Enter the number of the desired hotel:\n");
    printf("1) Fairfield Arms           2) Hotel Olympic\n");
    printf("3) Chertworthy Plaza        4) The Stockton\n");
    printf("5) quit\n");
    printf("%s%s\n", STARS, STARS);

    while ((status = scanf("%d", &code)) != 1  ||
             (code < 1 || code > 5))
   {
        if (status != 1)
            scanf("%*s");
        printf("Enter an integer from 1 to 5, please.\n");
    }

    return code;
}
```

Compiling Programs with Two or More Source Code Files

The hotel.c Function Support Module(2/2)

```
int getnights(void)
{
    int nights;

    printf("How many nights are needed? ");
    while (scanf("%d", &nights) != 1)
    {
        scanf("%*s");
        printf("Please enter an integer, such as 2.\n");
    }

    return nights;
}

void showprice(double rate, int nights)
{
    int n;
    double total = 0.0;
    double factor = 1.0;

    for (n = 1; n <= nights; n++, factor *= DISCOUNT)
        total += rate * factor;
    printf("The total cost will be $%0.2f.\n", total);
}
```

Compiling Programs with Two or More Source Code Files

The hotel.h Header File

```
#define QUIT        5
#define HOTEL1    80.00
#define HOTEL2   125.00
#define HOTEL3   155.00
#define HOTEL4   200.00
#define DISCOUNT   0.95
#define STARS "**********************************"

// shows list of choices
int menu(void);
// returns number of nights desired
in
```

Finding Addresses: The & Operator

The & Operator

- One of the most important C concepts is the pointer.
- which is a variable used to store an address.
- The unary & operator gives you the address where a variable is stored.
  - If `pooh` is the name of a variable, `&pooh` is the address of the variable.

Finding Addresses: The & Operator

The & Operator

- You can think of the address as a location in memory.
- Suppose you have the following statement:

```
pooh = 24;
```

- Suppose that the address where pooh is stored is 0B76. Then the statement

```
printf("%d %p\n", pooh, &pooh);
```

- would produce this (%p is the specifier for addresses):

```
24 0B76
```

Finding Addresses: The & Operator

The loccheck.c Program

```
#include <stdio.h>

void mikado(int);                      /* declare function  */
int main(void)
{
    int pooh = 2, bah = 5;             /* local to main()   */

    printf("In main(), pooh = %d and &pooh = %p\n",
            pooh, &pooh);
    printf("In main(), bah = %d and &bah = %p\n",
            bah, &bah);
    mikado(pooh);

    return 0;
}

void mikado(int bah)                   /* define function   */
{
    int pooh = 10;                     /* local to mikado() */
    printf("In mikado(), pooh = %d and &pooh = %p\n",
            pooh, &pooh);
    printf("In mikado(), bah = %d and &bah = %p\n",
            bah, &bah);
}
```

Finding Addresses: The & Operator

- The loccheck.c Program

```
C:₩Dev-Cpp₩loccheck.exe
In main(), pooh = 2 and &pooh = 0x0012ff48
In main(), bah = 5 and &bah = 0x0012ff44
In mikado(), pooh = 10 and &pooh = 0x0012ff34
In mikado(), bah = 2 and &bah = 0x0012ff40
계속하려면 아무 키나 누르십시오 . . .
```

Altering Variables in the Calling Function

Altering Variables in the Calling Function

- Suppose
- You have two variables called x and y and you want to swap their values.

```
x = y;
y = x;
```

Altering Variables in the Calling Function

Altering Variables in the Calling Function

- Suppose
- You have two variables called `x` and `y` and you want to swap their values.

```
x = y;
y = x;
```

- Does not work.
  - the original value of `x` has already been **replaced** by the original `y` value.

Altering Variables in the Calling Function

Altering Variables in the Calling Function

- An additional line is needed to temporarily store the original value of `x`.

```
temp = x;

x = y;
y = temp;
```

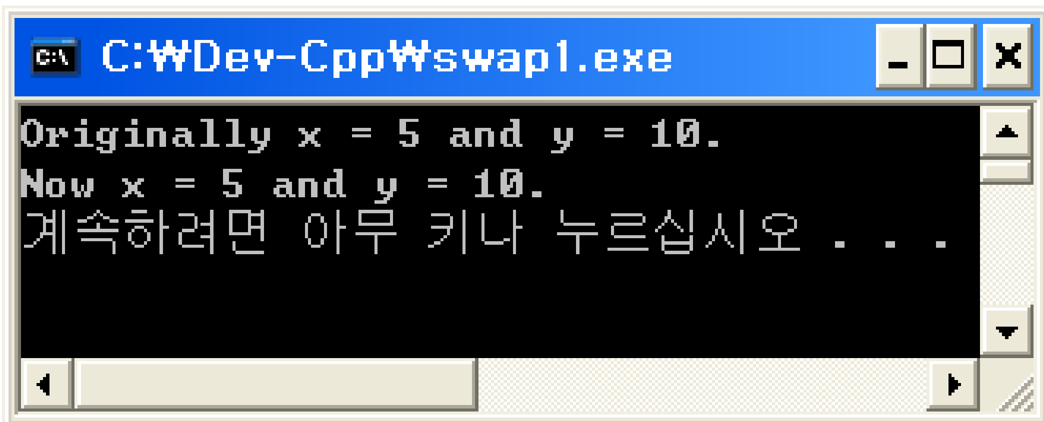
Altering Variables in the Calling Function

The swap1.c Program

```c
#include <stdio.h>

void interchange(int u, int v); /* declare function */

int main(void)
{
    int x = 5, y = 10;
    printf("Originally x = %d and y = %d.\n", x , y);
    interchange(x, y);
    printf("Now x = %d and y = %d.\n", x, y);

    return 0;
}

void interchange(int u, int v)  /* define function  */
{
    int temp;
    temp = u;
    u = v;
    v = temp;
}
```

Altering Variables in the Calling Function

- **The swap1.c Program**



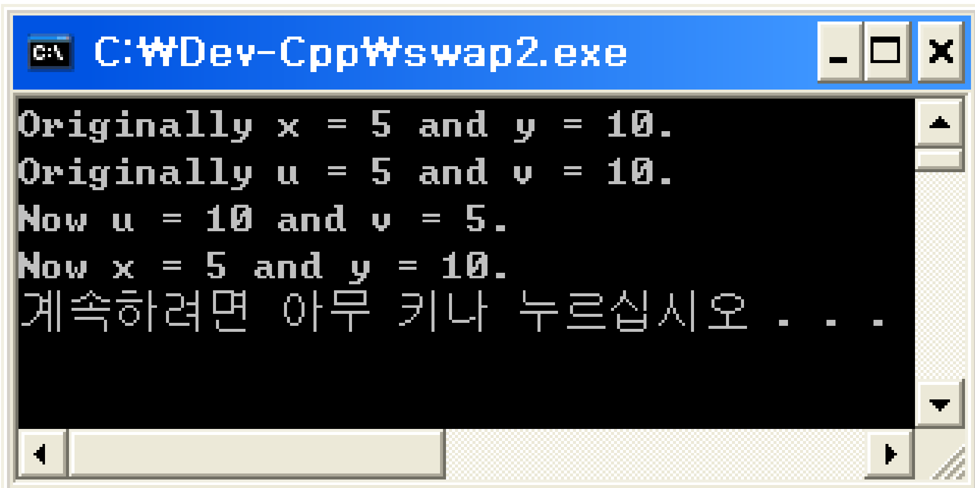
Altering Variables in the Calling Function

The swap2.c Program

```
#include <stdio.h>

void interchange(int u, int v);

int main(void)
{
    int x = 5, y = 10;
    printf("Originally x = %d and y = %d.\n", x , y);
    interchange(x, y);
    printf("Now x = %d and y = %d.\n", x, y);

    return 0;
}

void interchange(int u, int v)
{
    int temp;
    printf("Originally u = %d and v = %d.\n", u , v);
    temp = u;
    u = v;
    v = temp;
    printf("Now u = %d and v = %d.\n", u, v);
}
```

Altering Variables in the Calling Function

The swap2.c Program



Altering Variables in the Calling Function

The swap2.c Program

- Can you somehow use return?
- Well, you could finish `interchange()` with the line

```
return(u);
```

- and then change the call in `main()` to this:

```
x = interchange(x,y);
```

Pointers: A First Look

Pointer

- A variable whose value is a memory address.
- If you give a particular pointer variable the name `ptr`, you can have statements such as the following:

```
/* assigns pooh's address to ptr */
ptr = &pooh;
```

- We say that **`ptr` "points to" `pooh`**.
  - `ptr` is a variable.
  - `&pooh` is a constant.

Pointers: A First Look

Pointer

- If you want, you can make `ptr` point elsewhere:

```
/* make ptr point to bah instead of to pooh */
ptr = &bah;
```

- Now the value of `ptr` is the address of `bah`.

Pointers: A First Look

The Indirection Operator: *

- Suppose you know that `ptr` points to `bah`, as shown here:

```
ptr = &bah;
```

- Then you can use the indirection operator `*` to find the value stored in `bah`:

```
/* finding the value ptr points to */
val = *ptr;
```

Pointers: A First Look

The Indirection Operator: *

- The statements `ptr = &bah;` and `val = *ptr;` taken together amount to the following statement:

```
val = bah;
```

Pointers: A First Look

Declaring Pointers

- How do you declare a pointer variable?
- You might guess that the form is like this:

```
pointer ptr;
```

- Why not?
  - Because it is not enough to say that a variable is a pointer.

Pointers: A First Look

Declaring Pointers

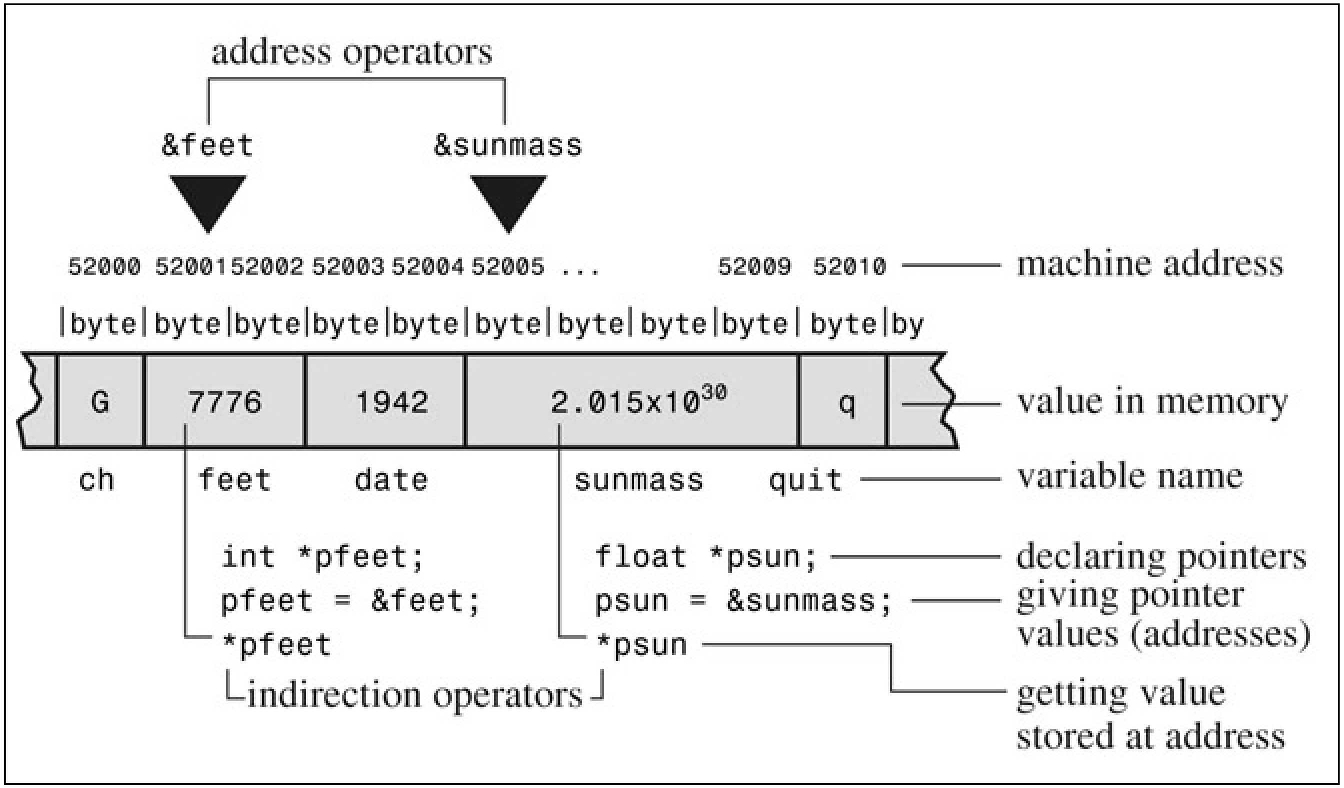
- A `long` and a `float` might use the same amount of storage, but they store numbers quite differently.
- Here's how pointers are declared:

```
int * pi;
char * pc;
float * pf, * pg;
```

Pointers: A First Look

Declaring Pointers

- **Declaring and using pointers.**



Pointers: A First Look

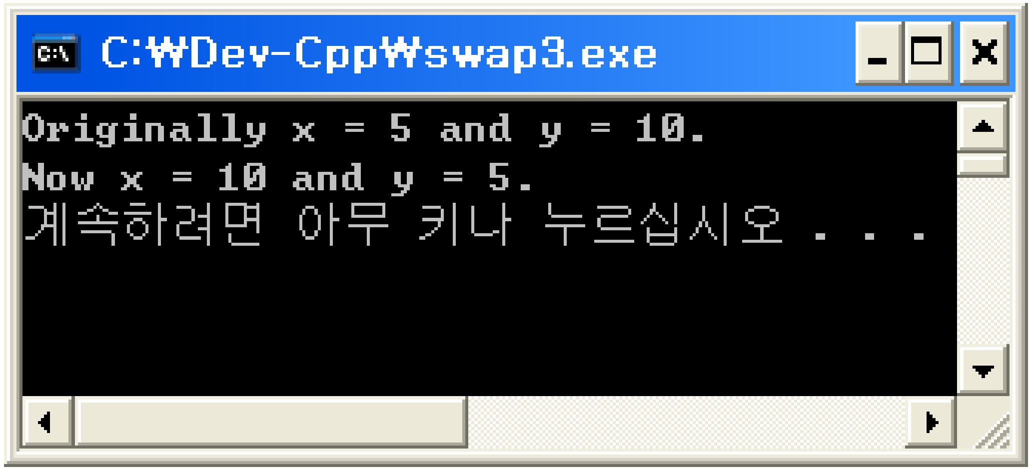
The swap3.c Program

- **Using Pointers to Communicate Between Functions**

```
#include <stdio.h>

void interchange(int * u, int * v);
int main(void)
{
    int x = 5, y = 10;

    printf("Originally x = %d and y = %d.\n", x, y);
    interchange(&x, &y);  /* send addresses to function  */
    printf("Now x = %d and y = %d.\n", x, y);
    return 0;
}

void interchange(int * u, int * v)
{
    int temp;
    temp = *u;         /* temp gets value that u points to */
    *u = *v;
    *v = temp;
}
```

Pointers: A First Look

The swap3.c Program



Pointers: A First Look

Let's see how Listing 9.15 works.

- First, the function call looks like this:

```
interchange(&x, &y);
```

- Instead of transmitting the values of `x` and `y`, the function transmits their *addresses*.

Pointers: A First Look

Let's see how Listing 9.15 works.

- x and y are integers, u and v are pointers to integers.
- so declare them as follows:

```
void interchange (int * u, int * v)
```

- Next, the body of the function declares

```
int temp;
```

  - to provide the needed temporary storage.

Pointers: A First Look

Let's see how Listing 9.15 works.

- To store the value of `x` in `temp`, use

```
temp = *u;
```

- Remember, `u` has the value `&x`, so `u` points to `x`.
- This means that `*u` gives you the value of `x`, which is what we want.
  - Don't write

```
temp = u;    /* NO */
```

Pointers: A First Look

Let's see how Listing 9.15 works.

- Similarly, to assign the value of y to x, use

```
*u = *v;
```

- which ultimately has this effect:

```
x = y;
```

Pointers: A First Look

Let's see how Listing 9.15 works.

- Using pointers and the * operator,
- The function can examine the values stored at those locations and change them.

- You can omit the variable names in the ANSI prototype.

```
void interchange(int *, int *);
```

Pointers: A First Look

Let's see how Listing 9.15 works.

- In general, you can communicate two kinds of information about a variable to a function.
- If you use a call of the form

```
function1(x);
```

  – you transmit the **value of x**.

- If you use a call of the form

```
function2(&x);
```

  – you transmit the **address of x**.

Pointers: A First Look

Let's see how Listing 9.15 works.

- The first form

```
function1(x);
```

- requires that the function definition includes a formal argument of the same type as `x`:

```
int function1(int num)
```

Pointers: A First Look

Let's see how Listing 9.15 works.

- The second form

```
function2(&x);
```

- requires the function definition to include a formal parameter that is a pointer to the right type:

```
int function2(int * ptr)
```

Pointers: A First Look

- **Names, addresses, and values in a byte-addressable system, such as the IBM PC.**