

# Operators and Control Flow

CS/COE 449 Spring 2016

# Review

Data Type	C Standard	32-bit	Windows 64-bit	Unix/Linux 64-bit
char	8 bits	8 bits	8 bits	8 bits
short	at least 16 bits	16 bits	16 bits	16 bits
int	at least 16 bits	32 bits	32 bits	32 bits
long	at least 32 bits	32 bits	32 bits	64 bits
long long	at least 64 bits	64 bits	64 bits	64 bits

- `sizeof(char)?`
  - Always 1
- `sizeof(int)?`
  - Depends on the **compiler/platform/OS** (but at least 2)
- Range of char vs. unsigned char?
  - char: -128 ~ 127, unsigned char: 0 ~ 255
- `sizeof(char*)?` (pointer to a char)
  - 4 (32 bits) on 32-bit systems, 8 (64 bits) on 64-bit systems

# Running Example

```
#include <stdio.h>                /* header file */

int main()
{
    int grade, count, total, average; /* declarations */
    count = 0;                       /* initialization */
    total = 0;                       /* initialization */
    while(1) {
        printf("Enter grade: ");     /* prompt */
        scanf("%d", &grade);         /* read input */
        if(grade < 0)
            break;                   /* break out of loop */
        else
            total = total + grade;
        count = count + 1;
    }
    average = total / count;
    printf("Average score is %d\n", average);
    return 0;
}
```

```
>> ./a.out
Enter grade: 100
Enter grade: 90
Enter grade: -1
Average score is 95
```

# Comments

```
#include <stdio.h>                                /* header file */

int main()
{
    int grade, count, total, average; /* declarations */
    count = 0;                        /* initialization */
    total = 0;                        /* initialization */
    while(1) {
        printf("Enter grade: ");      /* prompt */
        scanf("%d", &grade);          /* read input */
        if(grade < 0)
            break;                    /* break out of loop */
        else
            total = total + grade;
        count = count + 1;
    }
    average = total / count;
    printf("Average score is %d\n", average);
    return 0;
}
```

- Annotates code for better readability
- Ignored by compiler (not part of program)
- Syntax:
  - /\* some string \*/
  - // some string

# Variable Declarations

```
#include <stdio.h>                /* header file */

int main()
{
    int grade, count, total, average; /* declarations */
    count = 0;                      /* initialization */
    total = 0;                      /* initialization */
    while(1) {
        printf("Enter grade: ");    /* prompt */
        scanf("%d", &grade);        /* read input */
        if(grade < 0)
            break;                  /* break out of loop */
        else
            total = total + grade;
            count = count + 1;
    }
    average = total / count;
    printf("Average score is %d\n", average);
    return 0;
}
```

- Declares the types of variables that will be used in program
- Variables are the 'data' of a program
- Tells compiler the amount of memory to allocate for each variable
- Must come before use of variable (good practice: at beginning of function)
- Syntax: <type> <variable name>
- Variable names: consist of letters, digits (cannot begin with a digit), and underscores, case sensitive

# Constants

```
#include <stdio.h>          /* header file */

int main()
{
    int grade, count, total, average; /* declarations */
    count = 0;                   /* initialization */
    total = 0;                   /* initialization */
    while(1) {
        printf("Enter grade: "); /* prompt */
        scanf("%d", &grade);    /* read input */
        if(grade < 0)
            break;              /* break out of loop */
        else
            total = total + grade;
        count = count + 1;
    }
    average = total / count;
    printf("Average score is %d\n", average);
    return 0;
}
```

- Values that stay constant
- Not part of data (part of code)
- Numeric constants
  - Decimal: 0, 1, 2 ...
  - Octal: 012, 01776 (prefixed with 0)
  - Hexadecimal: 0xf5, 0xdeadbeef (prefixed with 0x)
- Character constants
- Single character: 'a', 'b', '1'
  - (single quotes)
  - Character string: "abc", "123"
  - (double quotes)

# Operators

```
#include <stdio.h>           /* header file */

int main()
{
    int grade, count, total, average; /* declarations */
    count = 0;                     /* initialization */
    total = 0;                     /* initialization */
    while(1) {
        printf("Enter grade: ");    /* prompt */
        scanf("%d", &grade);        /* read input */
        if(grade < 0)
            break;                  /* break out of loop */
        else
            total = total + grade;
            count = count + 1;
    }
    average = total / count;
    printf("Average score is %d\n", average);
    return 0;
}
```

- Performs the actual computation of a program
- Assignment
  - Stores expression to a variable
  - Expression: constant, variable, computation
- Arithmetic
  - +, -, \*, /, %
- Logical
  - &&, ||, !
- Comparison
  - ==, !=, <, >, <=, >=
- Bitwise
  - &, |, ^, ~, <<, >>
- Reference / Dereference
  - &, \* (will worry about these later)

# Bitwise Operators

- What are they good for?
- System programs often store multiple values (typically flags) in a variable.
  - Why? To save memory.
  - How? By shifting data in and out.
- Examples
  - Get value of 5<sup>th</sup> bit:  $(x \gg 5) \& 1$
  - Set 5<sup>th</sup> bit to 0:  $x \& \sim(1 \ll 5)$



# Shortcut Operators (Syntactic Sugar)

- Programmers are lazy. Give them some sugar.
- Shortcut assignment operators
  - Shorthand for a computation and an assignment
  - Ex)  $x += 10$  is equivalent to  $x = x + 10$ ;
  - $+=$ ,  $-=$ ,  $*=$ ,  $/=$ ,  $\%=$ ,  $\&=$ ,  $|=$ ,  $\wedge=$ ,  $<<=$ ,  $>>=$
- Increment and decrement operators
  - Adds or subtracts 1 from a variable
  - Ex)  $x++$  is equivalent to  $x = x + 1$ ;
  - $++$ ,  $--$
  - Post-increment vs. Pre-increment

# Operator Precedence

<u>Type</u>	<u>Operators</u>	<u>Associativity</u>
<u>highest</u>	<u>()</u>	<u>left to right</u>
<u>unary</u>	<u>+ - ++ -- ! *</u>	<u>right to left</u>
<u>multiplicative</u>	<u>* / %</u>	<u>left to right</u>
<u>additive</u>	<u>+ -</u>	<u>left to right</u>
<u>relational</u>	<u>&lt; &lt;= &gt; &gt;=</u>	<u>left to right</u>
<u>equality</u>	<u>== !=</u>	<u>left to right</u>
<u>logical and</u>	<u>&amp;&amp;</u>	<u>left to right</u>
<u>logical or</u>	<u>  </u>	<u>left to right</u>
<u>assignment</u>	<u>= += -= *= /=</u>	<u>right to left</u>

- When not sure, use parenthesis!

# Control Statements

```
#include <stdio.h>           /* header file */

int main()
{
    int grade, count, total, average; /* declarations */
    count = 0;                     /* initialization */
    total = 0;                     /* initialization */
    while(1) {
        printf("Enter grade: ");    /* prompt */
        scanf("%d", &grade);        /* read input */
        if(grade < 0)
            break;                  /* break out of loop */
        else
            total = total + grade;
            count = count + 1;
    }
    average = total / count;
    printf("Average score is %d\n", average);
    return 0;
}
```

- Changes the control flow of the program (what code to execute next)
- Iteration statements
  - Loops over statement while condition is satisfied
  - Statement: line ending with semicolon or compound statement (a.k.a block) inside braces
  - while, do/while, for
- Selection statements
  - Selects one among multiple statements to execute
  - if/else, switch/case
- Jump statements
  - Jumps to specific location in code
  - break, continue, return, goto

# Iteration Statements

## <while loop>

```
i = 0;
while (i < 10) {
    printf("%d ", i);
    i = i + 1;
}
```

## <do while loop>

```
i = 0;
do {
    printf("%d ", i);
    i = i + 1;
} while (i < 10);
```

## <for loop>

```
for(i = 0; i < 10; i = i + 1) {
    printf("%d ", i);
}
```

```
>> ./a.out
0 1 2 3 4 5 6 7 8 9
```

# Selection Statements

## <if statement>

```
score = 75;  
if (score >= 90) printf("A");  
else if (score >= 80) printf("B");  
else if (score >= 70) printf("C");  
else if (score >= 60) printf("D");  
else printf("F");
```

```
>> ./a.out  
C
```

## <switch statement>

```
char grade = 'A';  
switch (grade) {  
    case 'A':  
    case 'B':  
    case 'C':  
    case 'D':  
        printf("Pass");  
        break;  
    case 'F':  
        printf("Fail");  
        break;  
    default:  
        printf("Unknown");  
        break;  
}
```

```
>> ./a.out  
Pass
```

# Jump Statements

## <break>

```
i = 0;
while (i < 10) {
    if (i > 5) break;
    printf("%d ", i);
    i = i + 1;
}
```

```
>> ./a.out
0 1 2 3 4 5
```

## <continue>

```
i = 0;
while (i < 10) {
    if (i % 2) {
        i = i + 1;
        continue;
    }
    printf("%d ", i);
    i = i + 1;
}
```

```
>> ./a.out
0 2 4 6 8
```

## <goto>

```
i = 0;
while (i < 10) {
    if (i > 5) goto label;
    printf("%d ", i);
    i = i + 1;
}
label:
```

```
>> ./a.out
0 1 2 3 4 5
```

# Printf and Scanf

```
#include <stdio.h>                /* header file */

int main()
{
    int grade, count, total, average; /* declarations */
    count = 0;                       /* initialization */
    total = 0;                       /* initialization */
    while(1) {
        printf("Enter grade: ");    /* prompt */
        scanf("%d", &grade);        /* read input */
        if(grade < 0)
            break;                  /* break out of loop */
        else
            total = total + grade;
            count = count + 1;
    }
    average = total / count;
    printf("Average score is %d\n", average);
    return 0;
}
```

- Declared in stdio.h header file
- Part of standard C library (gets linked in during execution)
- Printf outputs text to the stdout stream, which is typically connected to the text console (e.g. your monitor)
- Scanf gets input from the stdin stream, which is typically connected to your keyboard

# Printf

- `int printf(const char* format, ...)`
- Format: a string (e.g. `"sum=%d\n"`) that can contain escape characters and format specifiers (e.g. `%d`)
- Escape characters:
  - `\n` – newline ( go to the next line)
  - `\r` – return ( go to the beginning of the line )
  - `\t` – tab character
  - `\'` – single quote (character ' )
  - `\"` – double quote (character " )



# Printf (cont'd)

- Format specifier: %[flags][width][.precision][length]specifier
- Flags:
  - 0: Pads numbers with 0s instead of spaces
- Width: minimum number of characters printed (if value is shorter, it is padded with spaces)
- Precision: for real numbers, number of digits to be printed after the decimal point
- Length: length of data type to be printed
  - (none): int
  - hh: char
  - h: short
  - l: long
  - ll: long long

# Printf (cont'd)

- Specifiers
  - d or i: Signed decimal integer
  - u: Unsigned decimal integer
  - o: Unsigned octal
  - x, X: Unsigned hexadecimal (lowercase, uppercase)
  - c: Character
  - s: Character String
  - p: Pointer address
  - f, F: Decimal floating point (lowercase, uppercase)
  - e, E: Scientific notation (mantissa + exponent)
  - g, G: Use shortest representation between f and e

# Printf (cont'd)

```
#include <stdio.h>
int main()
{
    printf ("Characters: %c %c \n", 'a', 65);
    printf ("Preceding with blanks: %10d \n", 1977);
    printf ("Preceding with zeros: %010d \n", 1977);
    printf ("Some different radices: %d %x %o \n", 100, 100, 100);
    printf ("floats: %4.2f %+0e %E \n", 3.1416, 3.1416, 3.1416);
    printf ("%s \n", "A string");
    return 0;
}
```

```
>> ./a.out
Characters: a A
Preceding with blanks:    1977
Preceding with zeros: 0000001977
Some different radices: 100 64 144
floats: 3.14 +3e+000 3.141600E+000
A string
```

# Scanf

- `int scanf(const char* format, ...)`
- Format: identical to `printf` except that now it specifies the format of the input stream.
  - If input does not match format (e.g. a string is given in place of a number specifier), an error is returned and the input is not consumed
- Example
  - `scanf("%d", &x);` <= input "abcd" : Failure!
  - `scanf("%x", &x);` <= input "abcd" : Success!

# Pitfall 1: The equality operator

- What's wrong with the following code?

```
if (x = 10) {
```

```
    ...
```

```
}
```

- The equality operator `==` should be used instead of the assignment operator `=`
- Will not even compile in Java but will actually run in C!
  - `x` will be assigned the value 10
  - assignment expression itself has the value 10 and code inside `if` will be executed

# Pitfall 2: The increment operator

- The following code wants to print 1 – 9.  
What's wrong with it?

```
int i = 0;  
while(i++ < 10) {  
    printf("%d\n", i);  
}
```

- Should have used pre-increment rather than post-increment

# Pitfall 3: Initialization

- What's wrong with the following code?

```
int sum, num, i;  
for(i = 0; i < 10; ++i) {  
    scanf("%d", &num);  
    sum += num;  
}
```

- Sum was not initialized to 0. Java does this automatically for you. C does not.

# Pitfall 4: Malformed if

- The following code wants to print a and b if both are larger than 0. What's wrong?

```
if (a > 0 && b > 0)
    printf("a=%d\n", a);
    printf("b=%d\n", b);
```

- Use proper indentation! Better yet, use curly braces even for single statements.



# Pitfall 5: Malformed switch/case

- What's wrong with the following code?

```
switch(x) {  
case 0:  
    printf("x = 0\n");  
case 1:  
    printf("x = 1\n");  
}
```

- Always remember to put breaks after each case (unless you want multiple cases to execute the same code).
- Always make a habit of putting in a default clause.

# Pitfall 6: Type conversion

- C standard says: When operations happen between two different types, the less precise type is converted to the more precise type
- What's wrong with the following code?

```
int x = 1, y = 2;
```

```
float z = x / y;
```

- Result of integer division is an integer and cannot hold a fractional result.
- Solution: `float z = (float)x / y`

- How about this code?

```
int x = -1;
```

```
unsigned int y = 1;
```

```
if(x > y) print("x is larger than y.\n");
```

- x is implicitly converted to an unsigned by the compiler
- Solution: `if(x > (int)y) print("x is larger than y.\n");`
- Do not leave things up to conversion rules. Always cast explicitly.

# Pitfall 7: Disappearing printf

- What would happen? Will Hello print?

```
printf("Hello.");
```

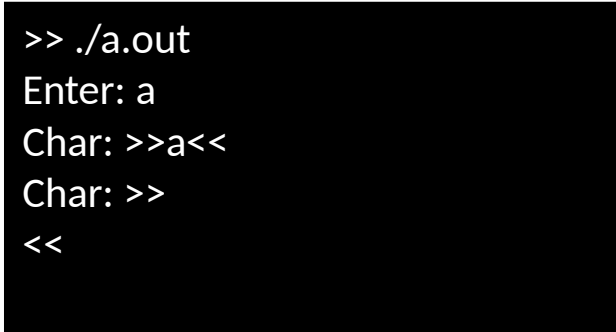
```
while(1);
```

- If you don't do anything, no. Because the stdout stream is by default buffered at line granularity (if to console) or buffer size granularity (if to file) in the C library
  - For efficiency reasons when performing I/O
  - Same applies to stdin
- Solutions
  - `fflush(stdout)` immediately after `printf` flushes the buffer
  - `setbuf(stdout, NULL)` to remove all buffering
  - `setlinebuf(stdout)` to buffer at line granularity

# Pitfall 8: Scanf and newline

- What's wrong with the following code?

```
while(1) {  
    printf("Enter: ");  
    scanf("%c", &c);  
    printf("Char: >>%c<<", c);  
}
```



```
>> ./a.out  
Enter: a  
Char: >>a<<  
Char: >>  
<<
```

- In scanf, %c consumes exactly one character. Solution:

```
while(1) {  
    printf("Enter: ");  
    scanf(" %c", &c);  
    printf("Char: >>%c<<", c);  
}
```

- White space in " %c" consumes previous newline.

# Review

- What is the value of “4 && 2”? And “4 & 2”?

>> 1 and 0 respectively

- What would the following print?

```
printf(“Num=%06.2f\n”, 3.1);
```

>> Num=003.10