Operators and Control Flow

CS449 Fall 2017
```c
#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
```
#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;
}
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;
}
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 associated with a storage location
- 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

- Performs the actual computation
- Assignment : Stores value into location
  - =
- Arithmetic : Operates on numbers
  - +, -, *, /, %
- Logical : Operates on boolean values
  - &&, ||, !
- Comparison : Compares two numbers
  - ==, !=, <, >, <=, >=
- Index : Operates on arrays
  - []
- Bitwise : Bitwise operations on numbers
  - &, |, ^, ~, <<, >>
- Reference / Dereference
  - &, *
Bitwise Operators

- Same semantics as Java 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\(^{th}\) bit: \((x >> 5) \& 1\)
  - Set 5\(^{th}\) bit to 0: \(x \& \sim(1 << 5)\)
  - Set 5\(^{th}\) bit to 1: How would you do it?
Reference / Dereference Operators

• Pointer: Variable (or storage location) that stores the address of another location.

• Reference operator: e.g. “p = &x;”
  – address of storage location for “x”
  – “Store the address of x to pointer p”

• Dereference operator: e.g. “*p = 0;”
  – storage location pointed to by “p”
  – “Store value 0 to storage location pointed by p”

• What’s the value of “x” at the end?
  int x = 0; int *p = 0;
  p = &x;
  *p = 100;

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1000</td>
<td>0</td>
</tr>
<tr>
<td>0x1004</td>
<td>0</td>
</tr>
<tr>
<td>0x1000</td>
<td>100</td>
</tr>
<tr>
<td>0x1004</td>
<td>0x1000</td>
</tr>
</tbody>
</table>
Rules for Reference / Dereference

• Rules for Reference Operator &
  – Operand **must be** a storage location
    • E.g. “&x” is legal, but “&(x + y)” is illegal
  – Result **is not** a storage location (it’s an address)
    • E.g. “y = &x” is legal, but “&x = 100” is illegal
  – Is the following expression legal “&(x)”?

• Rules for Dereference Operator *
  – Operand **need not be** a storage location (as long as it’s an address)
    • E.g. “*x” is legal, and so is “*(x + y)”
  – Result **is always** a storage location
    • E.g. “y = *x” is legal, and so is “*x = 100”
  – Is the following expression legal “*(x)”?
Shortcut Operators (Syntactic Sugar)

• Shortcut operators exist strictly for convenience
• Shortcut assignment operators
  – Shorthand for a computation and an assignment
  – Ex) x += 10 is equivalent to x = x + 10;
  – +=, -=, *=, /=, %=, &=, |=, ^=, <<=, >>=
• Increment (++) and decrement (--) operators
  – Adds or subtracts 1 from a variable
  – Post-increment (x++) vs. Pre-increment (++x)
    • Both increment x by 1
    • Difference in value of expression:
      If x was originally 10, x++ returns 10, ++x returns 11
# Operator Precedence

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

• When not sure, use parenthesis!
#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>
```c
i = 0;
while (i < 10) {
    printf("%d ", i);
    i = i + 1;
}
```

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

<for loop>
```c
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");

<switch statement>

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

>> ./a.out
C

>> ./a.out
Excellent
 Pass
Jump Statements

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

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

<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
#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;
}
Printf

• `int printf(const char* format, ...)`
• Return value: number of characters printed
• `...` : List of values you want to print, separated by commas
• Format: a string that specifies how to format values
  – Can contain escape characters and format specifiers
  – E.g. “sum=%d\n”
• Escape characters: Used to denote characters that would confuse the compiler parser if used directly
  \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 Format Specifiers

- Format specifier: \%[optional flags]specifier
- 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 Format Optional Flags

- Format specifier: %[padding][width][.precision][length]specifier
- Padding:
  - 0: Pads numbers with 0s instead of spaces
- Width: minimum number of characters printed (if value is shorter, it is padded with spaces or 0s depending on padding)
- 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
#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: %.2f %08.4f %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: 3.14 003.1416 3.141600E+000
A string
Scanf

• int scanf(const char* format, ...)  
• Return value: number of input items successfully matched and consumed  
• ... : List of values you want to scan, separated by commas  
• 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), not consumed  
• Example  
  – scanf(“%d”, &x); <= input “abcd” : Failure!  
  – scanf(“%x”, &x); <= input “abcd” : Success!  
    (x will contain hexadecimal value 0xabcd)
• 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

• What would be the value of x for each of the following inputs, after executing scanf("Num %d", &x)?
  User input: “5”?  
  >> Scan failure. Whatever the old value of x was
  User input: “Num 5”?  
  >> 5
Pitfall 1: Initialization

- What’s wrong with the following code?
  ```java
  int sum, i;
  for(i = 0; i < 10; ++i) {
    sum += i;
  }
  ```
- Sum was not initialized to 0. Java does this automatically for you. C does not.
Pitfall 2: The equality operator

• What’s wrong with the following code?
  
  ```java
  if (x = 10) {
    ...
  }
  ```

• The equality operator `==` should be used instead of the assignment operator `=`

• Will not even compile in Java but is legal in C!
  – ‘=‘ operator returns the assigned value
  – In example, assignment expression has value 10
Pitfall 3: The increment operator

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

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

- Should have used pre-increment (`++i`) rather than post-increment
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?

```c
switch(x) {
case 0:
    printf("x == 0\n");
break;

case 1:
    printf("x == 1\n");
break;
}
```

- Always remember to put breaks appropriately
- Always make a habit of putting in a default clause
Pitfall 6: Type conversion

• C standard: 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;
• Or this code?
  int x = -1;
  unsigned int y = 1;
  if(x > y) print(“x is larger than y.\n”);
• First case: result of integer division is an integer and cannot hold a fractional result
• Second case: x is implicitly converted to an unsigned by the compiler
• Solutions:
  – float z = (float)x / y
  – if(x > (int)y) print(“x is larger than y.\n”);
• Always cast explicitly when not certain.
Pitfall 7: Disappearing printf

• What would happen? Will Hello print?
  printf(“Hello.”);
  /* program crashes */
• If you don’t do anything, no. Because the stdout stream is
  by default buffered at line granularity (if connected to
  monitor console)
  – For efficiency reasons when performing I/O
  – Same applies to stdin (only consumes input on a newline)
• Solutions
  – fflush(stdout) immediately after printf will flush the buffer
  – setbuf(stdout, NULL) to remove all buffering
  – attach newline to the end of string
Pitfall 8: Scanf and &

• What’s wrong with the following code?

```c
printf("Enter: ");
scanf("%d", num);
printf("You typed: %d", num);
```

• Scanf always takes addresses as arguments, which it dereferences to get their storage locations

• Solution:

```c
printf("Enter: ");
scanf("%d", &num);
printf("You typed: %d", num);
```

```
>> ./a.out
Enter: 10
Segmentation fault (core dumped)
```

```
>> ./a.out
Enter: 10
You typed: 10
```
Pitfall 9: Scanf and newline

• What’s wrong with the following code?
while(1) {
    scanf(“Please print %d”, &num);
    printf(“%d”, num);
}
• No way to consume newline! Solution:
while(1) {
    scanf(“Please print %d”, &num);
    do { scanf(“%c”, &c); } while(c != ‘\n’);
    printf(“You typed: %d”, num);
}