# 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;
}
```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;
}
```

- **Comments**
  - Annotates code for better readability
  - Ignored by compiler (not part of program)
  - **Syntax:**
    - `/* some string */`
    - `// some string`
#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;
}

- Syntax: `<type> <variable name>
- Compiler allocates a `memory location` for the variable
- Also, declares the `type` of the variable
- Type tells compiler the amount of memory to allocate for each variable
- Must come before use of variable (for readability: at beginning of function)
- Variable names: consist of letters, digits, and underscores, case sensitive
#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;
}
#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;
}
Reference / Dereference Operators

- **Pointer**: Variable that stores a memory address
- **Reference operator**: “&x”
  - Value: **address** of variable “x”
  - Type: pointer to “x”
- **Dereference operator**: “*p”
  - Value: **memory location** pointed to by “p”
  - Type: base type of pointer “p”

What’s the value of “x” at the end?

```c
int x = 0;  // x is of type “int”
int *p = 0; // p is of type “int *”
p = &x;     // assign address of x to p
*p = 13;    // assign 13 to location pointed to by p
```
Reference / Dereference Quiz

<table>
<thead>
<tr>
<th>Address</th>
<th>Memory</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0001</td>
<td>5</td>
<td>x</td>
</tr>
<tr>
<td>0002</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0003</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0004</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0005</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>0006</td>
<td></td>
<td></td>
</tr>
<tr>
<td>.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- If x is declared as “char x;”
  - What is x?
    A) Value: 5, Type: char
  - What is &x?
    A) Value: 1, Type: char *
  - What is *x?
    A) Illegal! (x is not an address)

- If x is declared as “char *x;”
  - What is x?
    A) Value: 5, Type: char *
  - What is &x?
    A) Value: 1, Type: char **
  - What is *x?
    A) Value: 10, Type: char
Rules for Reference / Dereference

• Rules for Reference Operator &
  – Operand: a variable (i.e. name of a memory location)
    • “&(100)” is illegal // numerical constant does not have address
  – Result: address of memory location
    • “&x = 100” is illegal // cannot assign number to another number
  – Can the following expression be legal “&(x)”?

• Rules for Dereference Operator *
  – Operand: address of memory location
    • “*( (int *) 100 )” is legal // (int *) 100 is a legal address
  – Result: memory location
    • E.g. “*x = 100” is legal // *x is a location that can be assigned to
  – Can the following expression be legal “*(x)”?
Bitwise Operators

• Java also has them but more often used in C
• System programs often store multiple values (typically flags) in a single variable.
  – Why? To save memory.
  – How? By shifting data in and out.
• Examples
  – Get value of 5th bit: flag = (x >> 5) & 1
    /* SHIFT x right 5 times, AND result with 1 */
  – Set 5th bit to 1: x = x | (1 << 5)
    /* SHIFT 1 left 5 times, OR result with x */
  – Set 5th bit to 0: x = x & ~(1 << 5)
    /* SHIFT 1 left 5 times, NOT the result, then AND with 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 return 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!

© 2000 Prentice Hall, Inc. All rights reserved.
#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
", average);
    return 0;
}
**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);

<form 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”);

<switch statement>

char grade = ‘A’;
switch (grade) {
    case ‘A’:
        printf(“Excellent
”);
    case ‘B’:
    case ‘C’:
    case ‘D’:
        printf(“Pass
”);
        break;
    case ‘F’:
        printf(“Fail
”);
        break;
    default:
        printf(“Unknown
”);
        break;
}

>> ./a.out
C

>> ./a.out
Excellent
Pass
Jump Statements

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

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

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
Printf Examples

#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 variables you want to scan into
- 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)
Pitfall 1: Initialization

• What’s wrong with the following code?
  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?
  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 switch/case

• What’s wrong with the following code?

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

• Always remember to put breaks appropriately
• Always make a habit of putting in a default clause
Pitfall 5: 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;
  A) Problem: x / y results in an integer division
      Solution: float z = (float) x / y; /* x to float */
• Or this code?
  int x = -1;
  unsigned int y = 1;
  if(x > y) print(“x is larger than y.
”);
  A) Problem: x is implicitly converted to an unsigned int
      Solution: if(x > (int) y) …;
• Always cast explicitly when not certain.
Pitfall 6: Scanf and &

• What’s wrong with the following code?
```c
printf("Enter: ");
scanf("%d", num);
printf("You typed: %d", num);
```

• Scanf takes pointers as arguments, which it dereferences to update the memory location

• Solution:
```c
printf("Enter: ");
scanf("%d", &num);
printf("You typed: %d", num);
```
Pitfall 7: Disappearing printf

• What would happen? Will “Here” print?

```c
int *p = NULL;  // p initialized to null pointer
printf(“Here”);  // debug output
*p = 10;          // CRASH due to null pointer write
```

• Most likely, no. Stdout stream is by default buffered at line granularity (for efficiency reasons)
  – C standard library stores printf output into a memory buffer and flushes the buffer to screen only when a new line is encountered
  – What if program crashes before buffer gets chance to be flushed?

• Solutions
  – attach newline to the end of string (e.g. “Here\n”)
  – fflush(stdout); immediately after printf will flush the buffer
  – setbuf(stdout, NULL); to remove all buffering