4: Control Flow

A. Orlitsky and A. Vardy, based in part on slides by S. Arzi, G. Ruckenstein, E. Avior, S. Asmir, and M. Elad
Outline

❖ Conditionals
  ➢ if
  ➢ if-else
  ➢ switch-case
  ➢ conditional

❖ Loops
  ➢ while
  ➢ do-while
  ➢ for
  ➢ Nested loops

❖ Jumps
  ➢ break
  ➢ continue
  ➢ goto

❖ Macros

So far:
Straight-line calculation
Always do same thing
Almost calculator
No programming
About to get interesting
If expression is:

- True \((\neq 0)\) \(\rightarrow\) statement executed
- False \((=0)\) \(\rightarrow\) statement skipped, next statement executed

### Examples

```
if (x<0) {  
  x=-x;  
  printf("|x|=\%d", x);  
}
```

```
if (balance < 0)  
  printf("Chapter 11");
```

```
if (response == 'y')  
  printf("Account emptied");
```

Often, **compound statement**

```
if (n%4 != 0) {  
  printf("Not divisible by 4\n");  
  printf("Remainder is \%d\n", n%4);  
}
```

```
if (n%4) {  
  printf("Not divisible by 4\n");  
  printf("Remainder is \%d\n", n%4);  
}
```
**if-else**

\[
\text{if (expression) statement1;}
\text{else statement2;}
\]

* if `expression` is True → `statement1` executed, `statement2` skipped
* if `expression` is False → `statement2` executed, `statement1` skipped

**Examples**

```c
printf("|x|=");  
if (x>0)  
    printf("%d", x);  
else  
    printf("%d", -x);

if (n%2==1)  
    printf("Odd");  
else  
    printf("Even");

if (n%2)  
    printf("%d>\%d",x,y);  
else  
    printf("%d\%d",x,y);

if (rainy == 'y')  
    printf("Umbrella");  
else  
    printf("Shorts");
```

```c
ifelse.c
```
Roundoff Revisited

- Saw rounding:
  - towards 0 (*truncation*)
  - to nearest for *positive* numbers

```
r = (int) x;
r = (int) (x+.5);
```

- What about *negative* numbers?

```
if (x>0)
  r = (int) (x+.5);
else
  r = (int) (x-.5);
```
if-else-if Chains

\[
sign(x) = \begin{cases} 
1, & x > 0 \\
0, & x = 0 \\
-1, & x < 0 
\end{cases}
\]

```
sign.c
if (x < 0)
    printf("-1");
else
    if (x == 0)
        printf("0");
    else
        printf("1");
```

Flowchart:
- **False**
  - **False**
    - **False**
      - 1
    - **True**
      - **False**
        - x == 0
          - **False**
            - 0
          - **True**
            - -1
      - 0
  - **True**
    - 1
Indentation

Helps clarify structure of if-else-if chains

Two common conventions

```
if (a < 0)
    printf("-1");
else
    if (a == 0)
        printf("0");
    else
        if (a < 11)
            printf("1");
        else
            printf("2");
```

```
if (a < 0)
    printf("-1");
else if (a == 0)
    printf("0");
else if (a < 11)
    printf("1");
else
    printf("2");
```

Both well-established

Second more compact
Interpreting Nested if-else

```
if (a==1)
if (b==2)
printf("***");
else
printf("###");
```

Which if does else belong to?

**else** belongs to last else-less if

Can modify using {...}:

```
if (a==1) {
  if (b==2) printf("***");
}
else printf("###");
```
int i=0;
if (i++==0)
    printf("First check");
else
    printf("First increment");

int i=0;
if (++i==0)
    printf("First check");
else
    printf("First increment");
switch

❖ Executes one or more of several options

```c
switch (expression) {
    case value1: statements
             break;    // optional
    case value2: statements
             break;    // optional
    ...
    default: statements    // optional
}
```

❖ expression compared to value1, value2,... till a match

❖ If match: execution starts at case, continues till break, or closing brace }

❖ If no match: default executed, if no default, nothing executed

❖ Cases can only be integers (incl. char), not double, float
int i;

printf("Dame un nombre positivo: ");
scanf("%d", &i);

switch (i) {
    case 3:
        printf("es tres\n");
        break;
    case 1:
        printf("es uno\n");
        break;
    case 5:
        printf("es cinco\n");
        break;
    case 2:
        printf("es dos\n");
        break;
    case 4:
        printf("es quatro\n");
        break;
    default:
        printf("es muy grande!\n");
}

Hey. Wrong language! Besides, nombre means name!
int x, y;
char operation;
...
scanf("%d %c %d", &x,&operation, &y);
switch (operation) {
  case '+':
    printf("Result is: %d", x + y);
    break;    // what if we remove it?
  case '-':
    printf("Result is: %d", x - y);
    break;
  ...
  // code for '*' and '/' and '%'
  default:
    printf("Invalid operation!");
}
What does the following program do?

```c
int i;
printf("Start at: ");
scanf("%d", &i);
switch(i) {
case 1:
    printf("1 ");
case 2:
    printf("2 ");
case 3:
    printf("3 ");
case 4:
    printf("4 ");
case 5:
    printf("5\n");
    break;
default:
    printf("Too big or too small :-)\n");
}
```

- Asks for first integer `i`
- Prints `i i+1 ... 5` or too big/small

What if remove `break`?

Can be done with `if`

Can be done with `if-else`

Much easier with `loops`!
Switch vs. if-else

- **Switch** with all breaks can be implemented using **if, else if**

```c
switch(i) {
    case 7:
        printf("Lucky!\n");
        break;
    case 13:
        printf("Unlucky!\n");
        break;
    case ....
}
```

```c
if (i==7)
    printf("Lucky!\n");
else if (i==13)
    printf("Unlucky!\n");
else if ....
```

```c
switch(i) {
    case 7:
        printf("Lucky!\n");
        break;
    case 13:
        printf("Unlucky!\n");
        break;
    default:
        printf("Blah..\n");
}
```

```c
if (i==7)
    printf("Lucky!\n");
else if (i==13)
    printf("Unlucky!\n");
else
    printf("Blah..\n");
```

Without breaks (fall-through) more complicated.
Conditional (?:)

expression1 ? expression2 : expression3

if expression1 is
True → expression2 evaluated
False → expression3 evaluated

absoluteValue = (x >= 0) ? x : -x;
min = (x < y) ? x : y;
difference = (a > b) ? (a - b) : (b - a);
printf(n%2 ? "Odd" : "Even");
printf("You have %d item%s.", n, (n == 1) ? "" : "s");

❖ Precedence lower than regular operators (+,-,*,/,%,...)

❖ x nonzero. Want: when n>1, divide x by 2; when n<=1, divide x by 4

x/(n > 1) ? 2.0:4.0;

x/ ( (n > 1) ? 2.0:4.0);

Operates on 3 arguments - even lower priority
Multiple Conditionals

❖ Multiple conditionals associated from right to left

\[ c_1 \ ? \ v_1 \ : \ c_2 \ ? \ v_2 \ : \ v_3; \]

❖ Natural interpretation: If \( c_1 \) then \( v_1 \), else if \( c_2 \) then \( v_2 \), else \( v_3 \)

❖ Example: Print ordinal form of nonnegative integer (1st, 2nd, 24th, etc.)

```c
#include <stdio.h>

int main() {
    int x, y, z;

    printf("Nonnegative integer: ");
    scanf("%d", &x);
    y = x % 100;
    z = (y >= 11 && y <= 13) ? 0 : y % 10;
    printf("Ordinal: %d%s\n", x, z == 1 ? "st" : z == 2 ? "nd" : z == 3 ? "rd" : "th");
    return 0;
}
```

Example: Print ordinal form of nonnegative integer (1st, 2nd, 24th, etc.)
Conditional v. if-else

❖ Often interchangeable

❖ ?: more concise

```c
printf("|x|=");  // Reference code
if (x>0)
  printf("%d", x);
else
  printf("%d", -x);
```

```c
printf("|x|=");  // Using ?:
printf("%d", x>0 ? x : -x);
```

❖ if else often more versatile

```c
if (n > 1) {
  c = 2*a;
} else {
  c = 2*b;
}
```

```c
c = 2*(n>1 ? a : b);
```
Roundoff Once Again

❖ We used **if-else** to round to *nearest int, half away from 0*

```c
if (x>0)
    r = (int) (x+.5);
else
    r = (int) (x-.5);
```

❖ Using **conditional**

```c
r = x>0 ? (int) (x+.5) : (int) (x-.5);
```

- **Examples:**
  - 2.4 ➔ (int)(2.4+.5) ➔ (int)(2.9) ➔ 2
  - -2.5 ➔ (int)(-2.5-.5) ➔ (int)(-3.0) ➔ -3
Conditionals
- if
- if-else
- switch-case
- conditional

Loops
- while
- do-while
- for
- Nested loops

Jumps
- break
- continue
- goto

Macros
Outline

❖ Conditionals
   » if
   » if-else
   » switch-case
   » conditional

❖ Loops
   » while
   » do-while
   » for
   » Nested loops

❖ Jumps
   » break
   » continue
   » goto

❖ Macros
Loops

❖ Code segment executed repeatedly:

\[
\text{while } (n>1) \\
\quad \text{factorial } *= n--; \\
\text{for } (i = 1; i <= n; i++) \\
\quad \text{factorial } *= i;
\]

❖ Each loop execution is an \textit{iteration}
❖ Iterations continue as long as \textit{loop condition} is satisfied

❖ Three types of loops
  ▶ \textbf{while}
  ▶ \textbf{do-while}
  ▶ \textbf{for}
while Loop

- `while (expression) statement;`
- expression evaluated
- If True (nonzero) statement executed
- Repeats till expression becomes False
- Then exists loop and proceeds as normal

Statement can be compound
{statment1; statement2; ...}

```c
int n, i=1;
printf("Number of lines: ");
scanf("%d", &n);
while (i<=n) {
    printf("%d\n", i);
    i++;
}
```

```c
int n, i=1;
printf("Number of lines: ");
scanf("%d", &n);
while (i<=n) {
    printf("%d\n", i);
}
```

whilePrint1.c
whilePrint2.c
More Examples

❖ 1+2+3+...+n

```c
int n, sum = 0, i = 1;
scanf("%d", &n);
while(i <= n)
{
    sum = sum+i;
    i++;
}
printf("1+..+n = %d", sum);
```

❖ a^n

```c
int a, n, power=1, i=1;
scanf("%d %d", &a, &n);
while(i <= n) {
    power *= a;
    i++;
}
printf("%d^%d=%d", a,n,power);
```
Increment!

❖ Recall:

```c
int n, i=1;
printf("Number of lines: ");
scanf("%d", &n);
while (i<=n) {
    printf("%d\n", i);
    i++;
}
```

❖ What if we don’t increment?

```c
int n, i=1;
printf("Number of lines: ");
scanf("%d", &n);
while (i<=n) {
    printf("%d\n", i);
    // i++;
}
```

❖ Computer ‘hangs’
❖ To stop an infinite loop, type **Control-C**
Task: Prompt user for a positive integer, repeat while user complies. When input is non-positive, output average of all previous (positive) #'s.

Data structure: count - # positive integers, sum - their sum

```c
#include <stdio.h>

int main() {
    int count = 0, sum = 0, next = 1;
    while (next > 0) {
        printf("Positive integer: ");
        scanf("%d", &next);
        if(next>0) {
            sum += next;
            count++;
        }
    }
    if (count)
        printf("Average is: %.2f\n", (double)sum/count);
    else
        printf("You're so negative!\n");
}
```
**do-while Loops**

```c
do { 
    printf("Positive integer: ");
    scanf("%d", &value);
} while (value > 0);
```

- First, `statement` is executed, then, `expression` evaluated
- If `True` (nonzero), process repeats till `expression` becomes `False`
- Then execution continues to next statement

![Flowchart showing do-while loop execution](chart.png)

> The statement of a do-while loop is executed at least once!
Average

Calculating average using do while

```c
#include <stdio.h>

int main() {
    int count = 0, sum = 0, next;
    do {
        printf("Positive integer: ");
        scanf("%d", &next);
        if (next>0) {
            sum += next;
            count++;
        }
    } while (next > 0);
    if (count)
        printf("\nAverage is: %.2f\n", (double)sum/count);
    else
        printf("You're so negative!\n");
    return 0;
}
```

No artificial initialization
While v. Do-While

- When to use `while`, and when `do-while`?
- Generally speaking (there are always exceptions..)
  - If iteration always performed at least once: `do-while`
  - If even the first iteration may depend on condition: `while`
for Loops

```
for (expression1; expression2; expression3) {
    statement
}
```

❖ First, `expression1` is executed and initializes the loop

❖ Next, `expression2` is evaluated. If it is True (nonzero), `statement` and then `expression3` are executed, and keep getting executed, repeatedly, as long as `expression2` remains True

❖ When `expression2` evaluates to False, loop ends, program moves to next statement

Each of the three expressions can be empty; empty `expression2` is True!
Examples

❖ Sum of first n integers:

\[ \text{sum} = 0; \]
\[ \text{for } (i = 1; i \leq n; i++) \]
\[ \text{sum} += i; \]

❖ Product of first n integers:

\[ \text{factorial} = 1; \]
\[ \text{for } (i = 2; i \leq n; i++) \]
\[ \text{factorial} *= i; \]

**for and while loops are equivalent**

\[ \text{for (expression1; expression2; expression3) } \]
\[ \text{statement; } \]

expression1;
while(expression2) {
    \[ \text{statement} \]
    expression3;
}

\[ \text{for } (i = 2; i \leq n; i++) \]
\[ \text{factorial} *= i; \]

\[ i = 2; \]
while(i <= n) {
    \[ \text{factorial} *= i; \]
    i++;
}

Value of i after loop? \( n+1 \)
When *for* and When *while*

- *for* natural when iterations indexed (e.g., by i)
- All iteration book-keeping (initialization, update, test) in one line

**Add 10 numbers**

```c
for (i = 1; i <= 10; i++) {
    scanf("%d", &value);
    sum += value;
}
```

**Add till negative**

```c
next = 1;
for( ; next > 0; ) {
    printf("Pos integer: ");
    scanf("%d", &next);
    if (next>0) {
        count++;
        sum += next;
    }
}
```

```c
i = 1;
while (i <= 10) {
    scanf("%d", &value);
    sum += value;
    i++;
}
```

```c
next = 1;
while(next > 0) {
    printf("Pos integer: ");
    scanf("%d", &next);
    if (next>0) {
        count++;
        sum += next;
    }
}
```
Printing the ASCII Table

**Recall:** char is an 8-bit integer from -128 to 127
printf uses format %c, %d to interpret meaning

- Too many lines
- Show page at a time: a.out|more
- Or print 10 symbols per line

```c
#include <stdio.h>

int main() {
    char i;
    for (i=32; i<127; i++)
        printf("%3d -> %c\n", i, i);
    printf("\n");
    return 0;
}
```

Value of i after loop? 127
Comma Operator

- Expression like \( \text{exp1, exp2, exp3} \) is evaluated sequentially
- The type of a \textit{comma expression} is that of the last \textit{exp} evaluated
- Can perform multiple tasks in the initialization/increment of \textit{for} loop

Example

\[
\begin{align*}
\text{sum} &= 0; \\
\text{for } (i = 1; i \leq n; i++) \\
& \quad \text{sum } += i*i;
\end{align*}
\]

Can also write

\[
\begin{align*}
\text{for } (\text{sum} = 0, i = 1; i \leq n; i++) \\
& \quad \text{sum } += i*i;
\end{align*}
\]

Or even...

\[
\begin{align*}
\text{for } (\text{sum} = 0, i = 1; i \leq n; \text{sum } += i*i, i++);
\end{align*}
\]

Or Steven...

\[
\begin{align*}
\text{for } (\text{sum} = 0, i = 1; i \leq n; \text{sum } += i*i++);
\end{align*}
\]

\textit{Don’t abuse!}
In C99, you can define variables inside the loop control line, which makes the code cleaner and less cluttered.

```c
#include <stdio.h>
int main() {
    int i;
    for (i=0; i<3; i++)
        printf("%d\n", i);
    return 0;
}
```

- `i` is defined where it is used, making the code cleaner and less cluttered.
- `i` is recognized only inside the `for` loop.

Compilation may require `gcc -std=c99`.
Most program are *interactive* - use input and output to communicate with user.

Better be *user friendly* - describe expected input and explain meaning of output.

Check validity of user’s input.
#include <stdio.h>

int main() {
    int sum = 0, number_values;

    printf("Number of values: ");
    scanf("%d", &number_values);

    for (int value, i=0; i < number_values; i++){
        printf("Value: ");
        scanf("%d", &value);
        sum = sum + value;
    }

    printf("Sum is %d\n", sum);
    return 0;
}
#include <stdio.h>

int main() {
    int sum = 0, number_values;
    printf("Number of values: ");
    if (scanf("%d", &number_values) == 0
        || number_values < 0 ) {
        printf("Invalid # values\n");
        return 1;
    }
    for(int val, i=0; i<number_values; i++){
        printf("Value: ");
        if (scanf("%d", &val) == 0) {
            printf("Invalid value\n");
            return 1;
        }
        sum = sum + val;
    }
    printf("Sum is %d\n", sum);
    return 0;
}
Nested Loops

❖ Often need a loop inside another
❖ Two-dimensional objects
❖ Print a rectangle

![Rectangle example]

```c
#include <stdio.h>

int main() {
    int rows, columns;
    int i, j;

    printf("# rows: ");
    scanf("%d", &rows);
    printf("# columns: ");
    scanf("%d", &columns);

    for (i=0; i<rows; i++) {
        for (j=0; j<columns; j++)
            printf("(%d,%d) ", i,j);
        printf("\n");
    }
    return 0;
}
```

rectangle1.c

* * * * * * * * *
* * * * * * * * *
* * * * * * * * *
* * * * * * * * *
* * * * * * * * *
* * * * * * * * *
* * * * * * * * *
* * * * * * * * *
* * * * * * * * *
* * * * * * * * *
* * * * * * * * *
* * * * * * * * *
Suppose use same index for both loops

What will happen?

- **columns+1 >= rows**
  - Print one row

- **columns+1 < rows**
  - Infinite loop

```
#include <stdio.h>
int main() {
    int rows, columns;
    int i;
    printf("# rows: ");
    scanf("%d", &rows);
    printf("# columns: ");
    scanf("%d", &columns);
    for (i=0; i<rows; i++) {
        for (i=0; i<columns; i++)
            printf("%d ", i);
        printf("\n");
    }
    return 0;
}
```
Multiplication Table

❖ Print the multiplication table

```
#include <stdio.h>
int main() {
    int i, j;
    for (i=1; i<=9; i++) {
        for (j=1; j<=9; j++)
            printf("%2d ", i*j);
        printf("\n\n");
    }
    return 0;
}
```

```
1  2  3  4  5  6  7  8  9
2  4  6  8 10 12 14 16 18
3  6  9 12 15 18 21 24 27
4  8 12 16 20 24 28 32 36
5 10 15 20 25 30 35 40 45
6 12 18 24 30 36 42 48 54
7 14 21 28 35 42 49 56 63
8 16 24 32 40 48 56 64 72
9 18 27 36 45 54 63 72 81
```
#include <stdio.h>

int main() {
    int i, j;

    for(j=1; j<=9; j++)
        printf(j==1 ? "%1d|" : j==2 ? "%2d" : "%3d", j);
    printf("\n");

    for(j=1; j<=9; j++)
        printf((j==1 || j==9) ? "--" : "---");
    printf("\n");

    for(i=2; i<=9; i++) {
        printf("%1d|", i);
        for (j=2; j<=9; j++)
            printf("%2d ", i*j);
        printf(i!=9 ? "\n |\n" : "\n");
    }
    return 0;
}
Largest Non-Decreasing Substring

- Input: sequence of positive integers, followed by a single \( \leq 0 \)
- Output: length of longest non-decreasing contiguous substring

Examples:

- \( 1 \ 2 \ 1 \ 4 \ 4 \ 2 \ -1 : \ 3 \)
- \( 7 \ 5 \ 2 \ 4 \ 4 \ 6 \ 5 \ -3 : \ 4 \)
- \( 7 \ 6 \ 5 : \ 1 \)

Variables

- \textit{current\_value}: current value in sequence
- \textit{previous\_value}: previous sequence value
- \textit{current\_length}: current length of non-decreasing sequence
- \textit{longest\_length}: longest length of non-decreasing sequence seen so far

Algorithm

- If \( \text{current\_value} \geq \text{previous\_value} \), increment \textit{current\_length}.
- Else: if \( \text{current\_length} > \text{longest\_length} \), update latter.
- Reset \textit{current\_length} to 1.
Run Through

```c
if (current_value >= previous_value)
    current_length++; // Inside increasing sequence
else { // An increasing sequence just ended
    if (current_length > longest_length)
        longest_length = current_length;
    current_length = 1;
}
previous_value = current_value;
```
```c
#include <stdio.h>

int main() {
    int current_value, previous_value = 1;
    int current_length = 0, longest_length = 0;

    printf("Integers >0 followed by one <=0 to terminate: ");
    do {
        scanf("%d", &current_value);
        if (current_value >= previous_value)
            current_length++;
        else {
            int current_length > longest_length)
                longest_length = current_length;
                current_length = 1;
            }
        }
    previous_value = current_value;
    } while (current_value > 0);

    printf("Longest non-decreasing: %d\n", longest_length);
    return 0;
}
```

At first, only syntax matters.

Over time, algorithms are most interesting.
Recap: which Loop?

❖ Can always use for, while, do-while

❖ Rules of thumb when to use which

❖ for
  ‣ Natural iteration count (do something 10 times)
  ‣ Esp. if simple initialization before loop, or after each iteration (i++)

❖ while or do-while
  ‣ No natural iteration count, loop condition not iteration #
  ‣ First iteration always performed: do-while
  ‣ First iteration may not be required: while

❖ Just rules, do the more natural in each case

❖ Loop types interchangeable, don’t fret, just do it
Two Indentation Styles

```c
for (i=0; i<9; i++) {
    something;
    for (i=0; j<9; j++) {
        something;
        for (k=0; k<9; k++) {
            something;
            something_else;
        }
    }
}
```

```c
for (i=0; i<9; i++)
{
    something;
    for (i=0; j<9; j++)
    {
        something;
        for (k=0; k<9; k++)
        {
            something;
            something_else;
        }
    }
}
```

Up to you
Outline

❖ Conditionals
  ‣ if
  ‣ if-else
  ‣ switch-case
  ‣ conditional

❖ Loops
  ‣ while
  ‣ do-while
  ‣ for
  ‣ Nested loops

❖ Jumps
  ‣ break
  ‣ continue
  ‣ goto

❖ Macros
Outline

❖ Conditionals
  ‣ if
  ‣ if-else
  ‣ switch-case
  ‣ conditional

❖ Loops
  ‣ while
  ‣ do-while
  ‣ for
  ‣ Nested loops

❖ Jumps
  ‣ break
  ‣ continue
  ‣ goto

❖ Macros
**break and continue**

*Jump* statements that change the loop’s default behavior

**break**  
*Immediate loop exit*. Rest of loop skipped  
Execution proceeds to *statement after end of loop*

Note: **break** works as in **switch** statement

**continue**  
*Rest of iteration skipped*  
Execution proceeds to *next loop iteration*

Note: **continue** works **only** for loops, not for **switch**
Examples of continue

❖ Following reads 100 integers and sums the non-negatives

```c
for (i = 0; i < 100; i++) {
    scanf("%d", &num);
    if (num < 0)
        continue;
    sum += num;
}
```

❖ Does it differ from the right?

```c
i = 0;
while (i < 100) {
    scanf("%d", &num);
    if (num < 0)
        continue;
    sum += num;
    i++;
}
```

❖ The for loop will always read exactly 100 integers. The while loop will read as many integers as needed (perhaps 1,000,000) to get exactly 100 nonnegative integers

**What should we change to make the two loops equivalent?**
Infinite Loops and break

- Infinite loops often arise by mistake
- Other times, by design

```
while(1) {...}  for( ;1; ) {...}  for( ; ; ) {...}
```

- Such loops can be exited via: `break`, `return`, or `goto`

**Example:** Write a program that reads nonnegative integers and computes their sum, exiting when the user inputs the first negative integer

```
sum = 0;
do {
    scanf("%d",&num);
    sum += num;
} while (num >= 0)
```

OK? Adds first negative

```
sum = 0;
while(1) {
    scanf("%d",&num);
    if (num < 0)
        break;
    sum += num;
}
```
**Problem:** Compute the logical OR of $n$ input values

**Solution:** Two different approaches, both use `break`

```c
int i, n, value;
bool or;

or = false;
for (i=0; i<n; i++) {
    scanf("%d", &value);
    if (value) {
        or = true;
        break;
    }
}
```

What if we delete `break`?

```c
for (i=0; i<n; i++) {
    scanf("%d", &value);
    if (value)
        break;
    or = (i < n);
}
```

What if we delete `break`?

What gets printed?

```c
for (k = i = 0; i < 8; i++)
    for (j = 0; j < 8; j++) {
        if (j == 4)
            break;
        k++;
    }
printf("%d,%d,%d", i,j,k);
```

```c
for (k = i = 0; i < 8; i++)
    for (j = 0; j < 8; j++) {
        if (j == 4)
            continue;
        k++;
    }
printf("%d,%d,%d", i,j,k);
```

8, 4, 32

8, 8, 56
goto

Jumps to a **labeled statement**

```c
sum: x = a+b;
myloop: while (1) x++;
error: return 1;
```

Any identifier can be a **label**. Can go there from anywhere in program

```c
scanf("%d", &value);
if (value%2)
  goto odd;
printf("Even\n");
return 0;
odd:
  printf("Odd\n");
  return 0;
```

```c
for (i=0; i<n; i++) {
  // do something
  if (disaster) goto error;
  // no disaster here
}
return 0;
error: printf("Fixing it\n");
  // do something
return 1;
```

Frowned upon! **Avoid if possible**

Two return statements, first encountered terminates program
Outline

❖ Conditionals
  ‣ if
  ‣ if-else
  ‣ switch-case
  ‣ conditional

❖ Loops
  ‣ while
  ‣ do-while
  ‣ for
  ‣ Nested loops

❖ Jumps
  ‣ break
  ‣ continue
  ‣ goto

❖ Macros
Outline

❖ Conditionals
  ▸ if
  ▸ if-else
  ▸ switch-case
  ▸ conditional

❖ Loops
  ▸ while
  ▸ do-while
  ▸ for
  ▸ Nested loops

❖ Jumps
  ▸ break
  ▸ continue
  ▸ goto

❖ Macros
Macros

❖ What if want to multiply up to 5?
❖ Change 9 to 5 in \textit{6 locations}
❖ Time consuming, tedious, error prone
❖ Can use a variable \texttt{max=9}
❖ Value won’t change
❖ Define \texttt{MAX} to represent 9
❖ All \texttt{MAX} substituted by 9
❖ \textbf{Macro definition}
❖ Advantages:
  ‣ Single change
  ‣ Fewer errors
  ‣ Computationally efficient
  ‣ Clear
❖ Disadvantage: value can’t change
❖ Convention: CAPITAL letters

\begin{verbatim}
#include <stdio.h>

int main() {
    int i, j;
    for (j=1; j<=9; j++)
        printf(j==1 ? "%1d|" : j==2 ? ... : ...);
    printf("\n");
    for (j=1; j<=9; j++)
        printf((j==1 || j==9) ? "--" : ...);
    printf("\n");
    for (i=2; i<=9; i++)
        printf("%1d|", i);
        for (j=2; j<=9; j++)
            printf("%2d ", i*j);
        printf(i!=9 ? "\n |\n" : "\n");
    return 0;
}
\end{verbatim}

Helps identify as constant
#include <stdio.h>

#define MAX 9

int main() {
    int i, j;
    for(j=1; j<=MAX; j++)
        printf(j==1 ? "%1d|" : j==2 ? "%2d" : "%3d", j);
    printf("\n");
    for(j=1; j<=MAX; j++)
        printf((j==1 || j==MAX) ? "--" : "---");
    printf("\n");
    for(i=2; i<=MAX; i++) {
        printf("%1d|", i);
        for (j=2; j<=MAX; j++)
            printf("%2d ", i*j);
        printf(i!=MAX ? "\n |\n" : "\n");
    }
    return 0;
}
#define CLASS_SIZE 100

- Tells compiler to replace all CLASS_SIZE by 100
  - Except if CLASS_SIZE appears inside a string constant
  - Done during **pre-processing**, before anything else
  - CLASS_SIZE is **constant**, not **variable**, cannot change during runtime

```c
#include <stdio.h>
#define CLASS_SIZE 100

int main() {
    int grade, sum=0, i;
    for(i=0; i<CLASS_SIZE; i++) {
        printf("Grade: ");
        scanf("%d", &grade);
        sum += grade;
    }
    printf("Average=%.2lf\n", (double) sum/CLASS_SIZE);
    return 0;
}
```

No = or ;

Convention: **UPPER_CASE**

class_size.c
When to Use Macros

❖ **Macro v. constant**

- Simplifies changes: If `CLASS_SIZE` changes, update **only one line**
- Clarifies program: `CLASS_SIZE` clearer than `100`
- Eliminates retyping long constants

```
#define PI 3.141592653589793238462643383279502884
```

❖ **Macro v. variable**

- Faster access (no need to check variable value)
- Clarifies this is a constant whose value won’t change

```
#define CLASS_SIZE 100
```

❖ **Conclusion**

- Use macros for *meaningful or repeated constants*
Strings

```c
#define STRING "I am Sam\n"
#define PRINTME printf("Hello!\n");
...
printf(STRING);
printf("%s", STRING);
PRINTME
```

Operators

```c
#define AND &&
#define OR ||
...
if ( age<21 OR drink==1 ) .... // no car
if ( !(n%2) AND n>2 ) ....     // composite
```

Expressions

```c
#define TWO_PI 2 * 3.14159
...
circumference = TWO_PI * r;
```
Recursive Macros

❖ Macros can use other macros

```
#define TWO_PI 2 * 3.14159
...
circumference = TWO_PI * r;
```

```
#define PI 3.14159
#define TWO_PI 2 * PI
...
circumference = TWO_PI * r;
```

❖ Can appear anywhere

❖ Need to define all terms before using them

```
#define TWO_PI 2 * PI
#define PI 3.14159
...
circumference = TWO_PI * r;
```

```
#define TWO_PI 2 * PI
#define PI 3.14159
...
circumference = TWO_PI * r;
```
Quiz: What does the following program do?

```c
#define POSTEND POST return 0; }
#define POSTPRIN POST PRIN
#define POST
#define PRIN printf("%d\n",
#define BEGIN int main() {
#include <stdio.h>
BEGIN
PRIN 3
POSTPRIN 5
POSTPRIN 7
POSTEND
```

Expands to:

```c
#include <stdio.h>
int main() {
    printf("%d\n", 3);
    printf("%d\n", 5);
    printf("%d\n", 7);
    return 0;
}
```

That is:

Use cautiously.
Variables

❖ Since just replacement, macros can include variables

❖ Square

```c
#include <stdio.h>
#define SQUARE a*a
int main() {
    int a;
    printf("a: ");
    scanf("%d", &a);
    printf("a*a=%d", SQUARE);
}
```

❖ Min of two numbers

```c
#include <stdio.h>
#define MIN a<b ? a:b
int main() {
    int a, b;
    printf("a b: ");
    scanf("%d%d", &a, &b);
    printf("min=%d\n", MIN);
}
```
Macros with Arguments

What if want to apply macros to different variables?

```c
#include <stdio.h>
define SQUARE a*a
int main() {
    int a;
    printf("a: ");
    scanf("%d", &a);
    printf("a*a=%d", SQUARE);
}

#include <stdio.h>
define SQUARE(a) a*a
int main() {
    int a, b;
    printf("a b: ");
    scanf("%d%d", &a, &b);
    printf("a*a=%d", SQUARE(a));
    printf("b*b=%d", SQUARE(b));
}
```

Arguments

Almost..
Caution

❖ What if use expressions?

❖ Use parentheses

```c
#define SQUARE(a) a*a
int main() {
    int a;
    printf("a: ");
    scanf("%d", &a);
    printf("(a+1)*(a+1)=%d", SQUARE(a+1));
}
```

```c
#include <stdio.h>
#define SQUARE(a) a*a
int main() {
    int a, b;
    printf("a b: ");
    scanf("%d%d", &a, &b);
    printf("a=a=%d, SQUARE(a));
    printf("b=b=%d, SQUARE(b));
}
```

```c
#define SQUARE(a) a*a
int main() {
    int a;
    printf("a: ");
    scanf("%d", &a);
    printf("(a+1)*(a+1)=%d", SQUARE(a+1));
}
```

```c
#define SQUARE(a) (a)*(a)
int main() {
    int a;
    printf("a: ");
    scanf("%d", &a);
    printf("(a+1)*(a+1)=%d", SQUARE(a+1));
}
```

```c
#define SQUARE(a) (a+1)*(a+1)
int main() {
    int a;
    printf("a: ");
    scanf("%d", &a);
    printf("(a+1)*(a+1)=2*a+1", SQUARE(a+1));
}
```
More

❖ Complex macros with arguments

```c
#define MIN(u,v) (u)<(v) ? (u):(v)

#include <stdio.h>

int main() {
    int b;
    printf("b: ");
    scanf("%d%d", &b);
    printf("min=%d\n",MIN(b+1,3*7+2));
}
```

❖ Multiple arguments

```c
#define LEAP_YEAR(y) (y)%4 == 0 && (y)%100 != 0 || (y)%400 == 0

#include <stdio.h>

int main() {
    int b;
    printf("b: ");
    scanf("%d%d", &b);
    printf("min=%d\n",MIN(b+1,3*7+2));
}
```
Outline

❖ Conditionals
  › if
  › if-else
  › switch-case
  › conditional

❖ Loops
  › while
  › do-while
  › for
  › Nested loops

❖ Jumps
  › break
  › continue
  › goto

❖ Macros