The if statement;
Loops in C ;
the switch statement

references

- Kernighan & Ritchie –
  - Chapter 3.2, 3.3, *If-Else, Else-If*

- Kernighan & Ritchie –
  - Chapter 3.5, *Loops – While and For*
  - Chapter 3.6, *Loops – Do-while*
  - Chapter 3.7, *Break and Continue*

- Kernighan & Ritchie –
  - Chapter 3.4, *Switch*

Binary Decisions and the If Statement

- Interesting programs make decisions based on two-way ("binary") choices
- If-else blocks implement choices
- Requires *Boolean* value of **True** or **False** to make the choice
- Boolean expressions evaluate numeric, text, and Boolean values to produce a Boolean result

Examples of Boolean Expressions

- (7 > -5)
  - **True** (*a.k.a.* 1)
- (x == 0)
  - **True** if x is zero
  - *Not the same as* (x = 0) – be careful!
- a = 99.5;
  b = 1e4;
  c = (b < a);
  - c gets **False** (*a.k.a.* **0**)
- (x >= 97 && x <= 122)
  - **True** if x is in the range 97..122
    - These are ASCII lowercase…
- !(a == 0 || b == 0)
  - **True** if a and b are both non-zero
- z = (c && !d);
  - z gets **True** if c is **True** and d is **False**

Remark

- No Boolean type
  –Integer **0** is interpreted as **False**
- *Any* non-zero integer is interpreted as **True**
- So, *any* expression that evaluates to an integer, and *any* function that returns an integer, can be used as a Boolean expression!

- Among other things, this means that
  - "`if (x = 7) …`" is *legal* (and always True)
  - Very different from "`if (x == 7) …`"
  - Causes many bugs in C programs; today the compiler warns when this going to happen

If-else Patterns

- A typical high-level if-else block looks like:
  - *if (**Boolean-expression-is-true**)*
    do-this-block
    else
    do-that-block
  - Each block can be multiple statements (or empty)
- A multi-way if – else-if – else block:
  - *if (**Boolean-expression-is-true**)*
    do-block-A
    *else if (**other-Boolean-expression-is-true**)*
    do-block-B
    ⋮
    else
    do-default-block

Three kinds of code loops

- for loops
- while loops
- do loops (do-while loops)

- Loops can be affected by additional statements:
  - continue
  - break

General-purpose Loops

- For loops
  - Usually used for a fixed number of iterations, but not necessarily
- While loops
  - Usually used when it's uncertain or indeterminate how many iterations may occur

- Choose the statement that best reflects the programmer's intent
  - A for loop and a while loop can always be used interchangeably

for loops

```
for (start-statement(s) ;
      condition ;
      increment-statement(s)
     ) {
   loop-body statements ;
}
```

- start, increment, loop-body can be empty if desired
- If condition is empty , the loop never terminates
  - but see "break"

while loops

```
while (condition) {
   loop-body statements ;
}
```

- A for loop is equivalent to this:

```
start-statement(s) ;
while (condition) {
   loop-body statements ;   // but see "continue"
   increment-statement(s) ;
}
```

do loops (do-while loops)

- more specialized than for loops and while loops

```
do {
   loop-body statements ;
} while (condition) ;
```

- A do-while loop always executes at least once
  - for loops and while loops may execute zero times

Infinite loops

- These loops will execute forever, unless...

```
for ( ; ; ) {
   loop-body statements ;
}

while (1) {
   loop-body statements ;
}

do {
   loop-body statements ;
} while (1) ;
```

This is the preferred style for infinite loops, because it's obvious that the infinite loop is intentional.

continue and break

- loops can be interrupted by continue and break statements

- `continue;`
  - this statement stops a loop body
  - the loop continues with the next iteration

- `break;`
  - this statement terminates a loop early
  - execution porceeds to the next statement after the loop

example of continue

- This use of `continue` skips the value 5 after printing "boo!"
- Execution continues with the next iteration of the loop.

```c
#include<stdio.h>

int main(void)
{
        int i;
        for (i = 0; i < 9; i++) {
                if (i == 5 {
                        printf("boo!\n");
                        continue;
                }
                printf("%4i\n", i);
        }
        return 0;
}
```

example of break

```c
/* example of a break statement */
#include<stdio.h>
#include<math.h>

int main(int argc, char **argv)
{
   int r;
   double count, entry, product = 1.0, gmean;

   printf("Enter values. 0 ends entry.\n");
   for (count = 0 ; ; count++ ) {
      r = scanf(" %lf", &entry);
      if (EOF == r || 0.0 == entry)
         break;

      product *= entry;
   }
   gmean = pow( product, 1.0/count );
   printf("%lf terms; %lf geometric mean\n", count, gmean);
   return 0;
}
```

This routine accepts non-zero numbers and computes their geometric mean.

It could also be done without the break, but it would be uglier.

(no-break version of previous example)

```c
/* example of a break statement */
#include<stdio.h>
#include<math.h>

int main(int argc, char **argv)
{
   double count, entry, product = 1.0, gmean;

   printf("Enter values. 0 ends entry.\n");
   for (count = 0 ; EOF != scanf(" %lf", &entry) && 0 != entry ; count++ ) {
      product *= entry;
   }
   gmean = pow( product, 1.0/count );
   printf("%lf terms; %lf geometric mean\n", count, gmean);
   return 0;
}
```

This version shoves almost everything into the for statement itself.

Which version is easier to understand?

switch() statement

- Multi-way choices

```
switch (expression) {
        case const-expr :   statements ;
        case const-expr :   statements ;
          ⋮
    default :   statements ;
}
```

  - ***expression*** must evaluate to an integer-like value
  - ***const-expr*** must be an integer or constant (integer) expression
  - `default` is optional; ***statements*** can be empty

switch example – find vowels

- The first five cases all "fall through" to the last one.
- The `break` is required, or else everything would "fall through" into the default.

```
#include<stdio.h>

int main(void)
{
    int c;
    c = getchar();
    switch ( c ) {
        case 'a':
        case 'e':
        case 'i':
        case 'o':
        case 'u':
        case 'y':
            printf("vowel\n");
            break;
        default:
            printf("not a vowel\n");
    }
    return 0;
}
```