

Control flow statements: specify order in which computations are done

terminal
code

vars
output

Statements and Blocks

an expression becomes statement when it's followed by a semicolon

semicolon is a statement terminator

Braces `{}` group declarations & statements together into a compound statement/block

If-Else

```
if (expression) {  
    statement1  
} optional  
else  
    statement2
```

→ preferred to `expression != 0`

be careful w/ braces for nested if statements

Else-If

best way for multi-way decisions

```
if (condition1)  
    statement1  
else if (condition2)  
    statement2  
:  
else  
    statementn
```

evaluates @ first true expression, terminates chain

→ "none of the above", good for error checking, still optional

Switch

switch statement - multiway decision testing whether an expression matches one of a number of constant int. values & branches accordingly

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

must be different

→ optional default

count ^① digits, ^② whitespace, ^③ others

```
int c, i, nwhite, nother, ndigits [10];
```

```
nwhite = nother = 0;
```

```
for ( i=0; i<10; i++)  
    ndigit[i] = 0;
```

```
while ( (c=getchar()) != EOF ) {
```

```
    switch ( c ) {
```

```
        case '0': case '1': case '2': case '3': case '4':  
        case '5': case '6': case '7': case '8': case '9':  
            ndigits [c - '0'] ++ ;  
            break ;
```

```
        case ' ':  
        case '\n':  
        case '\t':  
            nwhite ++ ;  
            break ;
```

```
        default :  
            nother ++ ;  
            break ;
```

```
}
```

break statements causing immediate exit from the **switch**
%c **cases** are just labels, once a **case** code is executed, execution continues "falls through"
%o explicit escape. most common exits: **break** & **return**
↓
should be used sparingly

Loops - While and For

```
while ( expression )  
    statement
```

→ if non-zero, statement is executed
continue until it is zero

```
for ( expr1 ; expr2 ; expr3 )  
    statement
```

→ **same** →

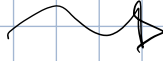
```
    expr1 ;  
    while ( expr2 ) {  
        statement  
    }  
    expr3
```

Usually..

$expr_1$ \rightarrow assignment or function calls
 $expr_2$ \rightarrow relational assignment
 $expr_3$ \rightarrow assignment or function calls

all are still optional

for is preferable when there is a simple initialization & increment

 **for** ($i=0$; $i < n$; $i++$)

can have 2 expressions in $expr_1$ & $expr_3$ separated by a comma

Loops - Do while

test @ bottom after making a pass

```
do {  
    statement  
} while (expression);
```

once expression is false, loop terminates