Objectives

Intended audience: Student who has working knowledge of Python

Target compiler: I’ll try to center the discussion on C99 using gcc 7.4

Objectives:

- To gain some experience with a statically-typed language
- To gain some experience with a compiled language
- To get practice in using pointers
  - To get practice with dynamically-allocated memory
  - Play around with linked lists
Intro
Loops

- Computers are handy for their speed
- Good at repetitive tasks
- We need a decision, or branch, and a jump (goto) to earlier code
- A loop is a slightly more structured way to repeat code
- A `while` loop is the simplest concept
While Loop
while Loop

```c
while( test )
    body
```

1. The *test* is evaluated
2. If true:
   1. Evaluate the statements in the *loop body*
   2. Return to the *test*
3. If false:
   1. Jump to statement following the *body*

- Again, C doesn’t care about newlines, nor indenting
- The body is:
  - A single C statement, OR
  - A sequence of statements, inside curly braces
Example **while** to Read Lines

- Loops are waiting for some condition to change, to become false
- To read lines until end of file:

```c
printf( "Enter a line, use ctrl-d to quit => " ) ;
read = getline( &buff, &len, stdin ) ;
while( ! feof( stdin ) )
{
    // overwrite the newline
    buff[read-1] = '\0' ;
    printf( "%3d: |%s|\n", ++i, buff ) ;

    printf( "Enter a line, use ctrl-d to quit => " ) ;
    read = getline( &buff, &len, stdin ) ;
}
```

- Note, we “prime the pump”
  - We prompt before the loop, and again near the end
Example **while** to Copy File by Char

- To read lines until end of file:

```c
FILE *fin = fopen( src, "rb" ) ;
FILE *fout = fopen( targ, "wb" ) ;
int c ; // EOF is -1, NOT a char

while( (c = getc( fin )) != -1 )
   putc( (char)c, fout ) ;

fclose( fin ) ;
fclose( fout ) ;
```

- `getc` is `fgetc`
  - Returns -1 at EOF
Example `while` to Read Words

- If the newline has no meaning
- Words are safe to parse\(^1\)

```c
FILE* fin = fopen( in_file_name );
char word[100];

fscanf( fin, "%99s", word );
while( ! feof( stdin ) )
{
    count( word );

    fscanf( fin, "%99s", buff );
}
```

- "\%99s" limits the size of the read word
  - Avoid buffer overflow
  - Reserve space for '\0'

\(^1\)I'm pretty sure

Kurt Schmidt (Skipjack Solutions)  C Loops  March 23, 2021  9/26
char resp = 'y' ;

while( resp==’y’ || resp==’Y’ )
{
    tellJoke() ;

    puts( "Would you like another joke? (y/[n]) => " )
    read = getline( &buff, &len, stdin ) ;
    sscanf( buff, " %c", &resp ) ;
}

- This example defaults to “no”
  - Any character other than y or Y drops out of the loop
  - Choose the least destructive default response
- Space in " %c" skips leading whitespace characters
Example `while` – Prompt to Continue, Default “Yes”

```c
char resp = 'y' ;

while( resp!='n' && resp!='N' )
{
    tellJoke() ;

    puts( "\nWould you like another joke? (y/[n]) => " )
    read = getline( &buff, &len, stdin ) ;
    ssprintf( buff, " %c", &resp ) ;
}
```

- This example defaults to “yes”
  - Only `n` or `N` drops out of the loop
- Space in " `%c`" skips leading whitespace characters
Example while – Counting

```c
int i;

i = 1;
while( i<=12 )
{
    printf( "%4d %8lu\n", i, (unsigned long)i * i ) ;

    ++i ;
}
```
**Style – the Dangling Semicolon**

- It’s possible to have a loop with an empty body:

```c
// discard input until 'X'
while( (c=getc(stdin)) != 'X' ) ;
```

- The semicolon after a loop intro is a common error
- Place it on the next line, to show that it is intentional

```c
// discard input until 'X'
while( (c=getc(stdin)) != 'X' )
;
```
Do-While
do-while Loop

do {
    stmt-list
} while( test ) ;

- Much like the while loop
- But, test is evaluated after the body has been evaluated
- So, the body is always evaluated at least once
- Not needed
  - I rarely use them
- They are, occasionally, the right fit
  - Not as often as young programmers think they are, though
Example **do-while**

define

typedef enum { TAILS, HEADS } eCoinSide ;
int cnt = -1 ; // adjust for off-by-one error, below

eCoinSide guess, result ;

do{
    cnt++ ;
    guess = getUserGuess() ;
    result = flipCoin() ;
}
while(guess == result ) ;

printf("You guessed %d straight tosses.\n", cnt ) ;
Example do-while

- Do-loops are okay for validating user input, maybe
  - Say, on a menu:

```c
char resp;

do
{
    showMenu();
    resp = getChoice();
}
while( ! strchr("WwDdBbTtQq", resp ) );
```
For Loop
The `for` Loop

- Syntactic sugar for While loop
  - Slightly more organised

```c
inits
while( test ) {
    body
    ...
    incr. stmts
}
```

```
for( inits ; test ; incr. stmts )
    body
```

1. `inits` happen ! once, upon reaching the loop
2. `test` is evaluated
   - If true:
     1. `body` is evaluated
     2. `incr. stmts` are evaluated
     3. Execution returns to the `test`
   - If false:
     1. Execution resumes below the `body`
Example for Loop Over Array

```c
int a[CAP] ;
int size = 0 ;
...

for( size_t i=0; i<size; ++i )
{
    printf( "%3zu: %-d\n", i, a[i] ) ;
}
```

- Remember, arrays in C are zero-based
- If a (dense) array holds \( n \) things, then they occupy indices \([0, n - 1]\)
- “For \( i = 0, i < len \)” is a standard idiom for processing arrays in C
Example: Counting for Loop

Arguments to Main

- First 2 arguments to main are:
  1. The number of command-line args (including calling the program)
  2. An array of strings (char*), storing each argument
- Names are arbitrary, but argc and argv by convention

```c
int main( int argc, char* argv[] )
{
    for( size_t i=0; i<argc; ++i )
    {
        printf( "Arg[%2zu]: %-s\n", i, argv[i] ) ;
    }
    ...
    return 0 ;
}
```
sNode *p ;
...
for( ; p!=NULL; p=p->next )
{
    printf( "%d\n", p->data ) ;
}

• Remember, arrays in C are zero-based
• If a (dense) array holds \( n \) things, then they occupy indices \([0, n - 1]\)
• “For \( i = 0, i < len \)” is a standard idiom for processing arrays in C
Standard Idioms Help Write Better Code

Use Standard Idioms When Coding

- Consider these loops to visit each element of array \( a \) and size \( n \):

```c
for( i=0; i<n; ++i )
    process( a[i] ) ;
for( i=1; i<=n; ++i )
    process( a[i-1] ) ;
for( i=n-1; i>=0; ++i )
    process( a[i] ) ;
```

- The first is the *standard idiom* we use in C
  - Our eyes recognise it right way, understand what it’s doing
  - If the loop looks different, it catches our eye
    - Presumably, it’s doing something different
    - It might need a second look
We can initialise and update more than 1 variable:

```c
int n = getN() ;
unsigned short i ;
unsigned long r ;
...
for( i=0, r=1; i<n; i+=1, r*=2 )
{
    printf( "%4hu %10lu\n", i, r ) ;
}
```
The Comma Operator

We talked about 2 operators who promise to evaluate operands left-to-right, `&&` and `||`

The only other one (I think) is the comma

Commas separate expressions

The value of the entire comma-expression is the right-most expression:
  - Has the lowest precedence of all C operators
  - Associates left-to-right
    - Remember, this is *not* the same as how operands are evaluated

The commas in argument and parameter lists, and in declarations, is different, just a delimiter
The Comma Operator – example

Comma Operator Eg.

```c
int i, j, k ;
i = 2, 4, 6, 8 ;  // i is 2 ; = has higher precedence
j = (928382, 287, 3) ;  // j is 3
k = (i+=3, i+j) ;
    // i is incremented to 5, k is assigned 5+3, or 8
```

• Mostly useful in:
  • For-loops
  • The ternary operator
  • Don’t, please. Use an if

• Subtle errors

```c
int a[20] ;
a[2, 5] = 7 ;
```

is not assignment into a 2D matrix, just a[5]