C Transition from Python

Kurt Schmidt
Skipjack Solutions
March 14, 2021
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
Introduction to C

- C is the language of kernels
  - Other languages, such as Java, Python, Bash, C#, C itself, and many others, are written in C
  - Applications from super computers to embedded microcontrollers
- C is an imperative, procedural language (like Python)
  - C is not object-oriented (unlike Python)
- C is compiled to native executable (unlike Python)
- C is statically-typed (also unlike Python), and statically-bound (like Python)
- C supports recursion (like Python)
Notes

• Code examples might have an accompanying link
  • Follow link to step through example at pythontutor.com
  • Does a nice job of graphically showing variables in memory, the heap, and the stack
Basics
```c
#include <stdio.h>

int main( void ) {
    int i = 13;
    double d = 1.618034;
    char name[20] = "The Lorax";

    printf( "I am %s, age %d, at %f\n", name, i, d );

    return( 0 );
}
```

- **main** is the entry point
- All code is in functions
  - Other than declarations and definitions
- Statements and type definitions are terminated with a semicolon (;)

```
$ gcc hello.c -o hello
$ ./hello
I am The Lorax, age 13, at 1.618034
```
Types, Decls
Basic Types

- Basic types are *integer* and *float*
- Characters are simply small (1-byte) integers
  - Delimited with single quotes
- C also understands strings
  - Delimited with double quotes
  - Stored as arrays of characters
- `_Bool` added (thinly) in C99
  - Don’t use it
  - `<stdbool.h>` defines the `bool` type, and literals `false` (0), and `true` (1)
C Built-in Types

<table>
<thead>
<tr>
<th>Integer</th>
<th>Float</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>float</td>
</tr>
<tr>
<td>short</td>
<td>double</td>
</tr>
<tr>
<td>int</td>
<td>long double</td>
</tr>
<tr>
<td>long</td>
<td></td>
</tr>
</tbody>
</table>

- The integer types have signed and unsigned flavors
- By default, `short`, `int` & `long` are signed
- `char` is implementation-dependent (all printing characters are positive)
Declaring Variables

type identifier [ = value ] ;

- Types held by variables known at compile time
  - This allows us to check operations and assignments at compile time
- Globals are declared outside of functions
- Locals are declared inside functions
- For scoping and extent, formal parameters behave as locals
struct
C Structs – Aggregate Types

- User-defined type
  - Type name is `struct foo`
  - This is a common idiom
    ```c
    typedef struct foo foo;
    ```
- Collection of fields
  - Field can be any defined type
- Fields are accessed with a dot (`.`)
```c
#include <string.h>
#include <stdio.h>

struct person {
    int age ;
    char[10] id ; /* string */
} ;

int main( void )
{
    struct person p ;
    p.age = 28 ;
    strcpy( p.id, "abc123" ) ;

    printf( "%s has %d years.\n", p.id, p.age ) ;

    return( 0 ) ;
}
```

$ gcc structs.c -o structs
$ ./structs
abc123 has 28 years.

- Type definitions (e.g., structs) end with a semicolon
- Functions do not
- Do not assign (=) strings
- Use `strcpy`
typedef

typedef <existing type> <new name>

- E.g.:
  
  typedef struct sNode* Stack ;
Functions
Functions

- Can *not* be defined inside another function (must be global)
- Return type part of function declaration
  - Can be `void`
  - `void` is a place-holder type
  - As a return type, it indicates that the function doesn’t return a value
- Arguments are passed *by value*
- Must be declared before use
  - A *prototype* serves
- May not be overloaded
- C supports recursion
Function Example

```c
#include <string.h>
#include <stdio.h>

/* prototype */
int foo( int k ) ;

int main( void ) {
    int j = 12, k = 13, r ;
    r = foo( j ) ;
    printf( "foo returned %d\n", r ) ;
    if( j != 12 )
        printf( "j changed\n" ) ;
    if( k != 13 )
        printf( "k changed\n" ) ;
    return( 0 ) ;
}

int foo( int k )
{
    /* j is local, and
     k behaves so */
    int j = 23 ;
    int rv = k * j ;
    k = 928357 ;
    return( rv ) ;
}
```

- https://goo.gl/v4rZgR

Kurt Schmidt (Skipjack Solutions)
C Transition from Python
March 14, 2021 18/81
main.c

```c
#include <string.h>
#include <stdio.h>
#include "foo.h"

int main( void ) {
    int j = 12,
    k = 13,
    r ;

    r = foo( j ) ;

    printf( "foo returned %d\n", r ) ;

    return( 0 ) ;
}
```

foo.h

```c
#ifndef __MY_FOO_H_
#define __MY_FOO_H_
int foo( int k ) ;
#endif /* __MY_FOO_H_ */
```

foo.c

```c
#include "foo.h"

int foo( int k )
{
    int j = 23 ;
    int rv = k * j ;
    k = 928357 ;

    return( rv ) ;
}
```

$ gcc main.c foo.c -o foo # $
Scoping in C

- C uses *lexical scoping* (or *static binding*)
- A symbol (variable name) is bound to a location at compile time
- A variable is either
  - Local to the function it appears in (if it’s declared there, or is a formal parameter), or
  - Global (declared outside of all functions, possibly in another file)
- Pretty straightforward in C, since we can’t define functions inside of other functions
Pointers
• Allow us to access memory locations, variables, indirectly
• Pointer is just an integer which stores an address (memory location)
• “Pointer” and “address” are synonymous.
• Can point to local, global, or heap memory
  • Or another pointer
  • Even to a function (not this set of slides)

**NOTE:** Run the examples in the following slides in http://www.pythontutor.com/c.html. It allows you to step through the code, provides visualisations
• Pointers are typed
• Declared with a *
• To declare a pointer to type T:
  \[ T \ *p = \text{NULL} ; \]
  • NULL is a safe, default pointer
  • It is 0, defined in \texttt{stdlib.h}
• Always point to \texttt{NULL (0)} when pointer not in use
• \texttt{void*} is a generic pointer
  • Can \textit{not} be de-referenced
& – Reference Operator

- Returns address of (pointer to) variable
- Prefix operator

```c
int i = 42;
int *p = NULL; /* initialise p */
p = &i;
```
* – Dereference Operator

- Used to get at the contents of memory pointed to
- Can be used to assign into memory

```c
int i = 42;
int *p = NULL; /* initialise p */
p = &i; /* point p at i */
*p = 5; /* use p to modify i */
```
```c
int main( void )
{
    int i = 5 ,
        j = 12 ;

    int *p = NULL ;

    p = &i ; /* modifying pointer */
    *p = 23 ; /* modifying thing pointed to */

    printf( "*p is %d\n", *p ) ;
    printf( "i is now %d\n", i ) ;

    return( 0 ) ;
}
```

/Kurt Schmidt (Skipjack Solutions)  C Transition from Python  March 14, 2021  26/81/
• We can pass pointers to values into functions
• Functions can then modify caller’s data
  • Indirectly, through the local pointer (formal parameter)
• Remember, *addresses* of variables must be passed in
Example – Pass by Reference

```c
void add3( int *p )
{ *p += 3 ; }

int main( void )
{
    int i = 28 ;

    add3( &i ) ; /* pass in ptr to i */
    printf( "i is now %d\n", i ) ;

    return( 0 ) ;
}
```

0https://goo.gl/sSwyh4
int* foo(int i)
{
    i += 3;
    return(&i);
}

int main(void)
{
    int j = 28;
    int*p = NULL;

    p = foo(j); /* NO! */

    return(0);
}
Pointers to Structs

- Given a person:
  ```c
  struct person {
    int age;
    char id[10]; /* string */
  } baker;
  ```
- And a pointer, `p`, to a person:
  ```c
  struct person *p = &baker;
  ```
- If `p` points to a person, then `*p` is that person
- To access the age field:
  ```c
  (*p).age = 28
  ```
- C gives us gentler syntax:
  ```c
  p->age = 28;
  strcpy(p->id, "s983742");
  ```
#include <stdio.h>
#include <string.h>

struct person
{
    int age ;
    char id[10] ;
} ;

int main( int argc, char *argv[] )
{
    struct person baker ;
    struct person *p = &baker ;

    p->age = 32 ;
    strcpy( p->id, "s9283" ) ;

    return( 0 ) ;
}
Arrays
Declaration of Static Arrays

\[ T \ name[size] \]

- Arrays are typed
- Size of \textit{statically-allocated} arrays (global or local) must be known at compile time
- Indexing starts at 0 (zero)
- Name of array is really a pointer to the first element
  - So, generally, \( a[i] \) is equivalent to \( *(a+i) \)
- Can be initialised \textbf{at declaration}

\[
\text{int } a[] = \{ 121, 17, 2, 88, -273 \} ;
\]
# Example – Arrays

```c
int main( void )
{
    int i ;
    int a[5] ;
    int *p = NULL ;

    for( i=0; i<5; ++i )
        a[i] = (i+2)*3 ;

    for( i=0; i<5; ++i )
        printf( "*(a+%d) = %d\n", i, *(a+i) ) ;

    for( p=a, i=0; i<5; ++i, ++p )
        *p += 2 ; /* add 2 to each element */

    return( 0 ) ;
}
```

[https://goo.gl/5guXKm](https://goo.gl/5guXKm)
Passing Arrays to Functions

- Arrays are passed by reference
  - Remember, name of an array is a pointer
    - A pointer is a reference to a memory location,
    - The *pointer* is copied into the function
  - Array can be modified by the function
- Size (number of elements) must also be passed to function
  - Unless a suitable sentinel value exists (e.g., '\0')
- The following 2 prototypes are equivalent
  
  ```c
  void foo( float a[], int n ) ;
  void foo( float* a, int n ) ;
  ```
```c
#include <stdio.h>

void print( float a[], int n, FILE *ofile )
{
    for( int i=0; i<n; ++i )
        fprintf( ofile, "%.2f ", a[i] ) ; /* access by index */
}

void addPhi( float *a, int n )
{
    for( int i=0; i<n; ++i, ++a )
        *a += 1.61803398 ; /* access through ptr */
}
```

- First parameter same in both functions\(^1\)
- `FILE*` is how we pass pointers to files

\(^1\)Not *quite*, in C++
```c
int main( int argc, char **argv )
{
    float fs[] = { 3.1415926535, 2.718281828, 22.4, 1.414213562 };
    addPhi( fs, 4 );
    printf( "The array now: " );
    print( fs, 4, stdout );
    printf( "\n" );
    return( 0 );
}
```

- Name of array is pointer to first element

¹https://goo.gl/g3AoeD
Pointers to Arrays

TODO

- Array name is pointer
- Pointer arithmetic
Strings
Introduction

- Stored in a character array
- String (in the array) is null-terminated
  - Special sentinel, the *null-terminator character*, (\0), marks the end of the string

```c
int main( int argc, char *argv[] )
{
    char name[10] ;
    strcpy( name, "Kurt" ) ;

    printf( "My name is %10s\n", name )

    return( 0 ) ;
}
```
### String Initialisation

Only use the assignment operator ( = ) on strings at initialisation.

```c
int main( int argc, char *argv[] )
{
    char last[20] = "Schmidt" ;
    char first[] = { 'K', 'u', 'r', 't', '\0' } ;

    printf( "My first name is %12s, last is %20s\n", first, last ) ;

    return( 0 ) ;
}
```

- `last` array has size 20 (can hold 20 characters)
  - Can only store a string of length 19
- `first` is of just sufficient size
  - 5
Basic Library Functions

All available in `<string.h>`

- `int strlen( char* s )` – Returns length of `s`
- `char* strcpy( char* d, char* s )` – How we assign. Copies contents from `s` to `d`
- `char* strcat( char* d, char* s )` – Appends contents of `s` to `d`
- `int strcmp( char* s1, char* s2 )` – Compare 2 strings. Return -1 if `s1 < s2`, 0 if equal, and 1 if `s1 > s2`

String Assignment

Do not use the assignment operator (`=`) for strings.

---

1 Assumes `d` has sufficient space
More Library Functions

All available in `<string.h>`

- `char* strchr( char* s, char c )` – Return pointer to first occurrence of `c` in `s`
- `char* strstr( char* haystack, char* needle )` – Substrings. Return pointer to first occurrence of `needle` in `haystack`
- `char* strdup( char* s )` – Duplicates `s` to heap memory, returns pointer to new memory
  - We’ve not discussed heap memory yet

---

\(^1\)Actually passed as an int, cast to a char
Command-Line Arguments

- `argv` is an array of `char*` (strings)
- There are `argc` of them
- `argv[0]` is the program name (invocation)

```
int main( int argc, char *argv[] )
{
    int i ;

    for( i=0; i<argc; ++i )
        printf( "%3d %s\n", i, argv[i] )

    return( 0 ) ;
}
```
Input/Output
#include <stdio.h>

FILE* fopen( const char* pathname, const char* mode );
int fflush( FILE* stream );
int fclose( FILE* stream );

Modes include:

- r  Open for reading (the default)
- r+ Open for reading and writing. Pointer positioned at beginning of file
- w  Truncate to zero length, open for writing
- a  Open for appending (writing at end of file). File pointer to end of file
- a+ Open for reading and appending. File pointer to end of file. Created, if needed
Binary vs. ASCII mode

- By default, files opened in ASCII (text) mode
  - Conversion for END-OF-LINE representations
  - E.g., write a LF (\n) in C, and a CR-LF (\r\n) will be written to the file in Windows, Vax.
  - On Mac pre-OS X, stored as a CR (\r)
  - On Unix, ASCII and binary modes the same

- Add a b to any mode string to open in binary mode
```c
#include <stdio.h>
#include <stdlib.h>

int main( int argc, char **argv )
{
    FILE *fp = NULL;
    fp = fopen( argv[1], "r" );
    if( fp == NULL )
    {
        fprintf( stderr, "Can not open %s for reading.\n", argv[1] );
        exit( 1 );
    }
    /* ... */
    fclose( fp );
    return( 0 );
}
```
Writing Text – printf, fprintf

#include <stdio.h>

int printf( const char *format, ... ) ;
int fprintf( FILE *fp, const char *format, ... ) ;

• ... is how we indicate variable arguments in C
• printf ( foo ) is just sugar for
  fprintf( stdout, foo )
• format is the string to be printed.
  • might contain format specifiers
  • values that follow correspond to the specifiers
%[flags][width][.precision][length]specifier

flags Justification, padding, leading +/-
width The *minimum* total field width (sig. info not truncated)
precision The number of decimals (might be truncated)
length Modifier for the type specifier
specifier Specifies the data type, and presentation
**fprintf Specifiers**

`%[flags][width][.precision][length]specifier`

<table>
<thead>
<tr>
<th>Character</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
<td>Character</td>
</tr>
<tr>
<td>f</td>
<td>Float</td>
</tr>
<tr>
<td>s</td>
<td>String</td>
</tr>
<tr>
<td>p</td>
<td>Pointer</td>
</tr>
<tr>
<td>%</td>
<td>%</td>
</tr>
<tr>
<td>d, i</td>
<td>Signed decimal int</td>
</tr>
<tr>
<td>e</td>
<td>Scientific notation</td>
</tr>
<tr>
<td>g</td>
<td>Shorter of %e or %f</td>
</tr>
<tr>
<td>o</td>
<td>Unsigned octal</td>
</tr>
<tr>
<td>u</td>
<td>Unsigned decimal int</td>
</tr>
<tr>
<td>x</td>
<td>Unsigned hex</td>
</tr>
</tbody>
</table>

**Table:** Common Specifiers
### fprintf Flags

\%
[flags][width][.precision][length]specifier

- **Left-justify**
- **+** Forces leading sign, even for positive numbers
- **#** With o, x, or X specifier, output is preceded by 0, 0x, or 0X
- **0** Pads with 0, rather than space

**Table: Flags**
#include <stdio.h>

int main( int argc, char *argv[] ) {
    int c;

    for( c='S' ; c<='k'; ++c ) {
        fprintf( stdout, "ASCII value = %3d, \\
                    "Character = %c\n", c , (char)c );
    }

    return(0);
}

• Note, the 2 strings are a single string, a single argument, passed to fprintf
ASCII value = 83, Character = S
ASCII value = 84, Character = T
...
ASCII value = 89, Character = Y
ASCII value = 90, Character = Z
ASCII value = 91, Character = [
ASCII value = 92, Character = \nASCII value = 93, Character = ]
ASCII value = 94, Character = ^
ASCII value = 95, Character = _
ASCII value = 96, Character = '
ASCII value = 97, Character = a
ASCII value = 98, Character = b
...
ASCII value = 106, Character = j
ASCII value = 107, Character = k
```
ssize_t getline( char** line, size_t* n, FILE* stream );
ssize_t getdelim( char** line, size_t* n, int delim, FILE* stream );
```

- Reads a line from the (ASCII) stream
- Allocates memory, modifies the pointer `line`, and the size `n`
- Extracts and stores the newline
- `getdelim` the same, but you specify the record separator, `delim`

---

¹Only in Unixland – see `fgets`
getline (e.g.)

```c
FILE *fp = fopen( "someTextFile" ) ;
char *buff = NULL ;
size_t len = 0;

while( getline( &buff, &len, fp ) != -1 )
{
    /* overwrite newline */
    buff[ strlen(buff)-1 ] = '\0' ;
    printf( "%z chars: %s\n", len, buff ) ;
}
```
char* fgets( char* s, int size, FILE* stream )
char* gets( char* s, int size )

- Reads a line from the (ASCII) stream (maybe)
  - Reads at most \(size - 1\) characters
- Extracts and stores the newline (maybe)
  - If length of line (w/NL) < \(size\)
- \texttt{gets} is just \texttt{fgets}, reading \texttt{stdin}
#include <stdio.h>
#include <string.h>

int main( int argc, char **argv )
{
    FILE *fp = ... ;
    char buff[100] ;

    while( fgets( buff, 100, fp ))
    {
        len = strlen( buff )-1
        if( buff[len] == '\n' ;
            buff[len] = '\0' ;
        printf( "%zu chars at %p: %s\n", strlen(buff), buff, buff ) ;
    }

    return( 0 ) ;
}
int fscanf( FILE *stream, const char *format, ... );
int scanf( const char *format, ... );

- Formatted input
  - Read an int, or a float, double, whitespace-separated word as a char*, etc.
  - Whitespace is not stored
- Uses specifiers similar to printf
- args are pointers to variables (storage)
- scanf is just fscanf, given stdin
```c
#include <stdio.h>
#include <string.h>

int main( int argc, char **argv )
{
    int theInt;
    double theDouble;
    char buff[20]; /* hold each word */

    fscanf( stdin, "%x %lf", &theInt, &theDouble );
    printf( "got int: %d and double: %.2f\n", theInt, theDouble );

    while( fscanf( stdin, "%19s", buff ) > 0 ) {
        printf( "word: %s\n", buff );
    }

    return( 0 );
}
```
fscanf – Eg. (cont.)

- Output, given the input below
- Note, the scanf family doesn’t note newlines

```
0x1B 1.618
Don’t get too lost in all I say.
```

```
got int: 27 and double: 1.62
word: Don’t
word: get
word: too
word: lost
word: in
word: all
word: I
word: say.
```
Other I/O Functions

• Some other functions which might be handy

\[\text{int } \text{fputc}( \text{int } c, \text{FILE* stream } );\]
\[\text{int } \text{fputs}( \text{const char }* s, \text{FILE* stream } );\]
\[\text{int } \text{fgetc}( \text{FILE* stream } );\]
\[\text{int } \text{ungetc}( \text{int } c, \text{FILE* stream } );\]

• EOF is \textit{not} a character
  • It is -1
• So, int, not char

\[\text{\textsuperscript{1}}\text{Use the man pages, section 3}\]
Binary I/O Functions

- In binary file, `NL`, `\0` are not special
  - Can’t be used as sentinels
  - Can’t mark the end of a string
  - Think of data as an array of bytes

```c
size_t fread( void *ptr, size_t size_of_elem, size_t number_of elems, FILE *fp );
size_t fwrite( const void *ptr, size_t size_of_elem, size_t number_of elems, FILE *fp );
```

- `fgetc` and `fputc` can be used here, too
String I/O

- We can format strings
- We can read from and write to strings just as we can files:

```c
int sscanf( const char *str,
            const char *format, ... );
int sprintf( char *str, const char *format, ... );
```
Heap Memory
Also known as the *free store*

- Programmer can request chunks of memory (using `malloc`\(^1\))
- Programmer is responsible for returning chunk to the heap (using `free`)
- Chunks are unnamed
  - *Must* be accessed through a pointer
- Chunk can be any size
  - One `int`
  - Array of `floats`
  - Person struct
- Good for resizeable arrays and strings, linked lists, . . .

\(^1\) Or `calloc`, `realloc`, or `strdup`
• Memory that is `malloc’d should be `free’d
• Cast return from `malloc
• Do `not assume memory was available
  • `malloc returns `NULL when it fails
  • **Check!**
• Use `sizeof operator to get size of types
• `free( `p ) does `nothing to `p itself
  • Simply marks memory pointed to by `p as available
• Don’t `free same chunk twice
  • Always set pointers to `NULL when not using them
Example – `malloc` for individual items

```c
int main( void )
{
    int *ip = (int*) malloc( sizeof( int ));
    int *jp = (int*) malloc( sizeof( int ));

    *ip = 52 ;
    *jp = 88 ;

    printf( "*ip is %d, and *jp is %d\n", *ip, *jp );

    free( ip ) ;
    ip = NULL ;
    jp = NULL ; /* MEMORY LEAK! */

    return( 0 ) ;
}
```

1. https://goo.gl/52seVC
```c
int* foo( int i )
{
    int *rv = (int*) malloc( sizeof( int ) ) ;
    *rv = i+3 ;
    return( rv ) ;
}

int main( void )
{
    int *p = foo( 28 ) ; /* Okay! */

    if( p != NULL )
        /* Do something w/p ... */

    free( p ) ;
    p = NULL ; /* No dangling ptrs */

    return( 0 ) ;
}
```

1 https://goo.gl/BFyX2H
$T \,*a = (T\,*) \text{malloc}( \,\text{CAP} \,\,* \,\text{sizeof}( \,\,T ) \,) \,;$

- We can maintain our own arrays, for greater flexibility
- Used just the same as statically-allocated
- Still must remember \textit{size}, the number of things of value it stores
- Must also remember \textit{capacity}, how many things it \textit{can} currently store
- A single value is simply an array of length one
```c
int main( void )
{
    int cap = 5;
    int *p = (int*) malloc( cap * sizeof( int ) );
    int *t = NULL, i;

    if( p==NULL ) return( 1 );

    for( i=0; i<cap; ++i ) p[i] = (i+2)*3;

    for( i=0; i<cap; ++i )
        printf( "*(p+%d) = %d\n", i, *(p+i) );

    for( t=p, i=0; i<cap; ++i, ++t )
        *t += 2; /* add 2 to each element */

    free( p );
    return( 0 );
}
```

1https://goo.gl/zTjuPy
Resizing Arrays

- If array is full:
  - Get a larger array from the heap
  - Copy contents of full array over
  - Free old array
  - Update array pointer
  - Update capacity

- See `realloc`

- Read through the Data Structures intro notes from CS265
Example – Resizing Arrays

```c
int main( void )
{
    int *t = NULL , i, cap = 5 ;
    int *p = (int*) malloc( cap * sizeof( int )) ;
    if( p==NULL ) return( 1 ) ;

    for( i=0; i<5; ++i ) p[i] = (i+2)*3 ;

    /* need more space */
    t = (int*) malloc( cap * 2 * sizeof( int )) ;
    if( t==NULL ) return( 1 ) ;

    for( i=0; i<cap; ++i )
        t[i] = p[i] ;

    free( p ) ;
    p = t ; t = NULL ; cap *= 2 ;

    return( 0 ) ;
}
```

1https://goo.gl/kXNyAG
Linked Lists – Intro

- Pointers to heap memory are well-suited to working with linked lists
- A List:
  - Linear, homogeneous container
  - *Not* indexed (no constant-time access)
  - Middle can be modified in constant time
- Made of *nodes*
  - Container for single element
  - Has references (pointers) to other nodes
typedef struct sNode sNode;
struct sNode {
    sNode *next;
    other data fields...
};

• Such a node makes up a *singly-linked* list
  • Can only be traversed in one direction
  • We could make a *doubly-linked* list
• The `typedef` allows us to refer to the type simply as `sNode`
#include <stdio.h>
#include <stdlib.h>

typedef struct sNode sNode;

struct sNode
{
    sNode *next;
    int    data;
};

void print( sNode *l, FILE *fp )
{
    fprintf( fp, "<" );
    while( l != NULL )
    {
        fprintf( fp, " %d", l->data );
        l = l->next;
    }
    fprintf( fp, " >" );
}
```c
int main( int argc, char *argv[] )
{
    sNode *l = NULL ,
        *t = NULL ;

    l = (sNode*) malloc( sizeof( sNode ) ) ;
    l->data = 13 ; l->next = NULL ;

    t = (sNode*) malloc( sizeof( sNode ) ) ;
    t->data = 12 ; t->next = l ; l = t ;

    t = (sNode*) malloc( sizeof( sNode ) ) ;
    t->data = 5 ; t->next = l ; l = t ;

    print( l, stdout ) ;
    printf( "\n" ) ;

    return( 0 ) ;  /* memory leak */
}
```

1 https://goo.gl/SHQHrK

Kurt Schmidt (Skipjack Solutions)  C Transition from Python  March 14, 2021
Freeing a Linked List

Never do this:

```c
sNode *p ;
for( p=L; p!=NULL; p=p->next )
    free( p ) ;
```

- This is the source of a subtle, but sinister, bug
- We free memory, and then we look at it

Consider this:

```c
sNode *p=L; *q ;
while( p != NULL )
{
    q = p->next ;
    free( p ) ;
    p = q ;
}
```
C99, C11
Some Changes in C99

These are *some* of the features added in C99:\(^1\)

- Inline functions
- Declare variables anywhere
- `bool` type, `true false` in `<stdbool.h>`
- Complex numbers in `<complex.h>`
- Line comments `//`
- Type-generic math functions in `<tgmath.h>`

\(^1\) Though several appeared as extensions in earlier compilers.
Some Changes in C11

These are some of the features added in C11:

- Multi-threading support in `<threads.h>`
- `gets` removed (deprecated in C99)
- Anonymous structures and unions
- Support for UTF-8, UTF-16, and UTF-32
- Bounds-checking functions