Further introductory topics of C

Xiaoqiang Wang

What is the memory

- Memory is like a big table of numbered slots where bytes can be stored.
- The number of a slot is its Address. One byte Value can be stored in each slot.
- Array data values span more than one slot, like the character string “Hello\n”

Addr	Value
0	
1	
2	
3	
4	‘H’ (72)
5	‘e’ (101)
6	‘l’ (108)
7	‘l’ (108)
8	‘o’ (111)
9	‘\n’ (10)
10	‘\0’ (0)
11	
12	

72?

Data type

- A Type names a logical meaning to a span of memory. Some simple types are:

`char`	a single character (1 slot)
`char [10]`	an array of 10 characters
`int`	signed 4 byte integer
`float`	4 byte floating point
`double`	signed 8 byte floating point

- Use sizeof() to calculate the size of the Type:

```
double x[10];
printf("x spans %d bytes memory\n", sizeof(x));
```

What is a variable?

A Variable names a place in memory where you store a Value of a certain Type.

symbol table?

Symbol	Addr	Value
	0	
	1	
	2	
	3	
x	4	?
y	5	'e' (101)
	6	
	7	
	8	
	9	
	10	
	11	
	12	

You first Define a variable by giving it a name and specifying the type, and optionally an initial value

declare vs define?

```
char x;
char y='e';
```

Initial value of x is undefined

Initial value

Name

What names are legal?

Type is single character (char)

extern? static? const?

The compiler puts them somewhere in memory.

Multi-byte variables

- Different types consume different amounts of memory. No necessary to be adjacent immediately.

```
char x;
char y='e';
int z = 0x01020304;
```

0x means the constant is written in hex

padding

An int consumes 4 bytes

Symbol	Addr	Value
	0	
	1	
	2	
	3	
x	4	?
y	5	'e' (101)
	6	
	7	
z	8	4
	9	3
	10	2
	11	1
	12	

What is the Stack?

- It's a special region of your computer's memory that stores temporary variables created by each function (including the main() function).
- The stack is a "FILO" (first in, last out) data structure
- A key to understanding the stack is the notion that **when a function exits**, all of its variables are popped off of the stack (and hence lost forever). Thus stack variables are **local** in nature. This is related to a concept we saw earlier known as **variable scope**, or local vs global variables.

Stack

- The stack grows and shrinks as functions push and pop local variables
- There is no need to manage the memory yourself, variables are allocated and freed automatically
- The stack has size limits
- Stack variables only exist while the function that created them, is running

Stack

Recall lexical scoping. If a variable is valid "within the scope of a function", what happens when you call that function recursively? Is there more than one "exp"?

Yes. Each function call allocates a "stack frame" where Variables within that function's scope will reside.

```
#include <stdio.h>
#include <inttypes.h>

float pow(float x, int exp)
{
  /* base case */
  if (exp == 0) {
    return 1.0;
  }

  /* "recursive" case */
  return x*pow(x, exp – 1);
}

int main(int argc, char **argv)
{
  float p;
  p = pow(5.0, 1);
  printf("p = %f\n", p);
  return 0;
}
```

float x	5.0	
int exp	0	Return 1.0
float x	5.0	
int exp	1	Return 5.0
int argc	1	
char **argv	0x2342	
float p	5.0	

Grows

Challenge Problem

- Write pow() so it requires log(n) iterations.

Scoping summary

- The scope of Function Arguments is the complete body of the function.
- The scope of Variables defined inside a function starts at the definition and ends at the closing brace of the containing block
- The scope of Variables defined outside a function starts at the definition and ends at the end of the file. Called "Global" Vars.

(Returns nothing)

```
void p(char x)
{
                /* p,x */
  char y;
                /* p,x,y */
  char z;
                /* p,x,y,z */
}
                /* p */
char z;
                /* p,z */

void q(char a)
{
  char b;
                /* p,z,q,a,b */

  {
    char c;
                /* p,z,q,a,b,c */
  }

  char d;
  /* p,z,q,a,b,d (not c) */
}

/* p,z,q */
```

char b?

legal?

Referencing Data from Other Scopes

- So far, all of our examples all of the data values we have used have been defined in our lexical scope

```
float pow(float x, uint exp)
{
  float result=1.0;
  int i;
  for (i=0; (i < exp); i++) {
    result = result * x;
  }
  return result;
}

int main(int argc, char **argv)
{
  float p;
  p = pow(10.0, 5);
  printf("p = %f\n", p);
  return 0;
}
```

Nothing in this scope

Uses any of these variables

Can a function modify its arguments?

What if we wanted to implement a function pow_assign() that *modified* its argument, so that these are equivalent:

```
float p = 2.0;
/* p is 2.0 here */
p = pow(p, 5);
/* p is 32.0 here */
```

```
float p = 2.0;
/* p is 2.0 here */
pow_assign(p, 5);
/* p is 32.0 here */
```

Would this work?

```
void pow_assign(float x, uint exp)
{
  float result=1.0;
  int i;
  for (i=0; (i < exp); i++) {
    result = result * x;
  }
  x = result;
}
```

NO!

Remember the stack!

```
void pow_assign(float x, uint exp)
{
  float result=1.0;
  int i;
  for (i=0; (i < exp); i++) {
    result = result * x;
  }
  x = result;
}

{
  float p=2.0;
  pow_assign(p, 5);
}
```

float x	32.0
uint exp	5
float result	32.0
float p	2.0

Grows

In C, all arguments are passed as values

But, what if the argument is the *address* of a variable?

Passing Addresses

- Recall our model for variables stored in memory
- What if we had a way to find out the address of a symbol, and a way to reference that memory location by address?

```
address_of(y) == 5
memory_at[5] == 101
```

```
void f(address_of_char p)
{
  memory_at[p] = memory_at[p] - 32;
}
```

```
char y = 101;       /* y is 101 */
f(address_of(y));   /* i.e. f(5) */
/* y is now 101-32 = 69 */
```

Symbol	Addr	Value
	0	
	1	
	2	
	3	
char x	4	‘H’ (72)
char y	5	‘e’ (101)
	6	
	7	
	8	
	9	
	10	
	11	
	12	

"Pointers"

This is exactly how "pointers" work.

```
"address of" or reference operator:  &
"memory_at" or dereference operator: *
```

```
void f(address_of_char p)
{
  memory_at[p] = memory_at[p] - 32;
}
```

```
char y = 101;       /* y is 101 */
f(address_of(y));   /* i.e. f(5) */
/* y is now 101-32 = 69 */
```

A "pointer type": pointer to char

```
void f(char * p)
{
  *p = *p - 32;
}
```

```
char y = 101;       /* y is 101 */
f(&y);              /* i.e. f(5) */
/* y is now 101-32 = 69 */
```

Pointers are used in C for many other purposes:

- Passing large objects without copying them
- Accessing dynamically allocated memory
- Referring to functions

Pointer Validity

- A Valid pointer is one that points to memory that your program controls.
- Using invalid pointers will cause non-deterministic behavior, and will often cause Linux to kill your process (SEGV or Segmentation Fault).
- There are two general causes for these errors:
  - Program errors that set the pointer value to a strange number
  - Use of a pointer that was at one time valid, but later became invalid
- Will ptr be valid or invalid?

```
char * get_pointer()
{
  char x=0;
  return &x;
}

{
  char * ptr = get_pointer();
  *ptr = 12;  /* valid? */
}
```

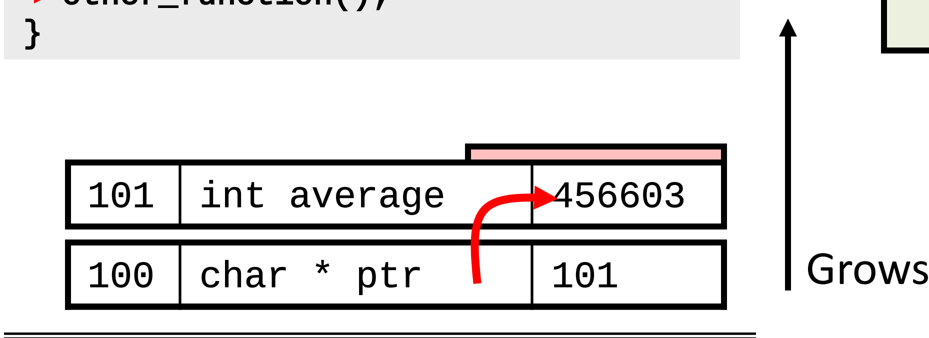
Answer: Invalid!

- A pointer to a variable allocated on the stack becomes invalid when that variable goes out of scope and the stack frame is “popped”. The pointer will point to an area of the memory that may later get reused and rewritten.

```
char * get_pointer()
{
  char x=0;
  return &x;
}

{
  char * ptr = get_pointer();
  *ptr = 12;  /* valid? */
  other_function();
}
```

But now, `ptr` points to a location that’s no longer in use, and will be reused the next time a function is called!



Arrays

- Arrays in C are composed of a particular type, laid out in memory in a repeating pattern. Array elements are accessed by stepping forward in memory from the base of the array by a multiple of the element size.

```
/* define an array of 10 chars */
char x[5] = {'t','e','s','t','\0'};

/* accessing element 0 */
x[0] = 'T';

/* pointer arithmetic to get elt 3 */
char elt3 = *(x+3);  /* x[3] */

/* x[0] evaluates to the first element;
 * x evaluates to the address of the
 * first element, or &(x[0]) */

/* 0-indexed for loop idiom */
#define COUNT 10
char y[COUNT];
int i;
for (i=0; i<COUNT; i++) {
  /* process y[i] */
  printf("%c\n", y[i]);
}
```

Brackets specify the count of elements. Initial values optionally set in braces.

Arrays in C are 0-indexed (here, 0..9)

x[3] == *(x+3) == 't'   (*NOT* 's'!)

What's the difference between char x[] and char *x?

For loop that iterates from 0 to COUNT-1. Memorize it!

Symbol	Addr	Value
char x [0]	100	't'
char x [1]	101	'e'
char x [2]	102	's'
char x [3]	103	't'
char x [4]	104	'\0'

Dynamic Memory Allocation

- So far all of our examples have allocated variables statically by defining them in our program. This allocates them in the stack.
- But, what if we want to allocate variables based on user input or other dynamic inputs, at run-time? This requires dynamic allocation.

sizeof() reports the size of a type in bytes

```
int * alloc_ints(size_t requested_count)
{
  int * big_array;
  big_array = (int *)calloc(requested_count, sizeof(int));
  if (big_array == NULL) {
    printf("can't allocate %d ints: %m\n", requested_count);
    return NULL;
  }

  /* now big_array[0] .. big_array[requested_count-1] are
   * valid and zeroed. */
  return big_array;
}
```

calloc() allocates memory for N elements of size k

Returns NULL if can't alloc

%m ?

It's OK to return this pointer. It will remain valid until it is freed with free()

Dynamic Memory Functions

Function	Description
`malloc`	allocates the specified number of bytes
`realloc`	increases or decreases the size of the specified block of memory. Reallocates it if needed
`calloc`	allocates the specified number of bytes and initializes them to zero
`free`	releases the specified block of memory back to the system

malloc returns a void pointer (void *), malloc allocates based on byte count but not on type. One may "cast" this pointer to a specific type:

```
int *ptr;
ptr = malloc(10 * sizeof (*ptr)); /* without a cast */
ptr = (int *)malloc(10 * sizeof (*ptr)); /* with a cast */
```

Do not forget to free your dynamic allocated memory!

Caveats with Dynamic Memory

- Dynamic memory is useful. But it has several caveats:
  - Whereas the stack is automatically reclaimed, dynamic allocations must be tracked and free()'d when they are no longer needed. With every allocation, be sure to plan how that memory will get freed. Losing track of memory is called a "memory leak".
  - Whereas the compiler enforces that reclaimed stack space can no longer be reached, it is easy to accidentally keep a pointer to dynamic memory that has been freed. Whenever you free memory you must be certain that you will not try to use it again. It is safest to erase any pointers to it.
  - Because dynamic memory always uses pointers, there is generally no way for the compiler to statically verify usage of dynamic memory. This means that errors that are detectable with static allocation are not with dynamic

Memory Functions

- Include <mem.h>
- void *memchr(const void *ptr, int ch, size_t len)
  – memchr finds the first occurence of ch in ptr and returns a pointer to it (or a null pointer if ch was not found in the first len bytes
- int memcmp(const void *ptr1, const void *ptr2, size_t len)
  – memcmp compares two memory byte by byte. Return 0 if equal.
- void *memcpy(void *dst, const void *src, size_t len)
  – memcpy copies len characters from src to dst and returns the original value of dst
  – The result of memcpy is undefined if src and dst point to overlapping areas of memory
- void *memmove(void *dst, const void *src, size_t len)
  – memmove is just like memcpy except that memmove is guaranteed to work even if the memory areas overlap
- void *memset(void *ptr, int byteval, size_t len)
  – memset sets the first len bytes of the memory area pointed to by ptr to the value specified by byteval

String functions

- Include <string.h>
- size_t strlen(const char *str)
  - strlen returns the number of characters in str that preceed the terminating null ('\0') character
- char *strcpy(char *dst, const char *src)
  - Copy characters from src to dst (up to and including the terminating null character ('\0') of src)
- char *strncpy(char *dst, const char *src, size_t len)
  - Copy up to len characters from src to dst
  - If src is shorter than len, dst is filled to len characters with null characters
  - If src is longer than len characters, the string in dst is **not** terminated with a null character
- strcpy and strncpy return a pointer to dst
- If src and dst are overlapping string, the results of strcpy and strncpy are undefined

String functions

- char *strcat(char *dst, const char *src)
  - Append (catenate) the string src onto the end of dst, beginning by overwriting the terminating null in dst and continuing until the terminating null character of src is copied to dst
- int strcmp(const char *str1, const char *str2)
  - Compare each character in str1 to the corresponding character in str2 until either a dissimilar character or a null terminator ('\0' character) is found
  - If both strings are identical up to and including the terminating '\0', 0 is returned
  - If str1[*n*] < str2[*n*], -1 is returned
  - If str1[*n*] > str2[*n*], 1 is returned
  - If str1 is shorter than str2 (for example, str1 points to "abc" and str2 points to "abcdefg"), -1 is returned
  - Likewise, if str1 is longer than str2, 1 is returned
- char *strstr(const char *str, const char *substr);
  - strstr finds the first occurence of substr in str and returns a pointer to the first character of the substring in str
  - If substr is not found in str, a null pointer is returned.
  - If substr is the empty string (that is, if the first character in substr is a null character) a pointer to the first character in str is returned.
- Conversion Functions: atoi, atol and atof
  - int atoi(const char *str), convert string to integer

File I/O

- #include <stdio.h>
- For files you want to read and write, you need a file pointer, e.g.:
  FILE *fp;
- Opening a file:
  fp = fopen(filename, mode);

  mode: "r" - Read only
  "w" – Create a file to write
  "a" – Append
  "b" – binary mode
- Always close a file when you no need to use it:
  fclose(fp);
- Writing to a file:
  - Text mode: fprintf(…), fscanf(…), fgetc(fp), fgets(buf, n, fp);
  - Binary mode: fwrite(ptr, size_of_elements, number_of elements, fp);
    fread(ptr, size_of_elements, number_of_elements, fp);
- you can set the position where to do the write and read:
  feek(fp, offset, whence);
  whence can be SEEK_SET – beginning of file,
  SEEK_CUR - Current position, SEEK_END - End of file.
  ftell(fp): get the current position of the file.

File I/O Example

- You can treat a text file as a binary file, and read the entire file into memory:

```
/* example: read an entire file */
#include <stdio.h>
#include <stdlib.h>

int main ()
{
 FILE * pFile;
 long lSize;
 char * buffer;
 size_t result;

 pFile = fopen ( "myfile.bin" , "rb" );
 if (pFile==NULL) {fputs ("File error",stderr); exit (1);}

 // obtain file size:
 fseek (pFile , 0 , SEEK_END);
 lSize = ftell (pFile);
 fseek (pFile , 0 , SEEK_SET);

// allocate memory to contain the whole file:
 buffer = (char*) malloc (sizeof(char)*lSize);
 if (buffer == NULL) {fputs ("Memory error",stderr); exit (2);}

 // copy the file into the buffer:
 result = fread (buffer,1,lSize,pFile);
 if (result != lSize) {fputs ("Reading error",stderr); exit (3);}

 /* the whole file is now loaded in the memory buffer. */

 // terminate
 fclose (pFile);
 free (buffer);
 return 0;
}
```