CSCI 2021: C Basics

Chris Kauffman

Last Updated:
Mon Feb 4 17:21:28 CST 2019

Logistics

Reading

- Bryant/O'Hallaron: Ch 1
- C references: whole language
  - types, pointers, addresses, arrays, conditionals, loops, structs, strings, malloc/free, preprocessor, compilation etc.

Goals

- Gain working knowledge of C
- Understand correspondence to lower levels of memory

Assignment 1

- Covers basics of C programming
- Lab02 material will be helpful for last problem

Every Programming Language

Look for the following as it should almost always be there

- ☒ Comments
- ⊟ Statements/Expressions
- ⊟ Variable Types
- ⊟ Assignment
- ⊟ Basic Input/Output
- ☐ Function Declarations
- ☐ Conditionals (if-else)
- ☐ Iteration (loops)
- ☐ Aggregate data (arrays, structs, objects, etc)
- ☐ Library System

Exercise: Traditional C Data Types

These are the traditional data types in C

Bytes*	Name	Range
	INTEGRAL	
1	`char`	-128 to 127
2	`short`	-32,768 to 32,767
4	`int`	-2,147,483,648 to 2,147,483,647
8	`long`	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
	FLOATING	
4	`float`	$\pm$3.40282347E+38F (6-7 significant decimal digits)
8	`double`	$\pm$1.79769313486231570E+308 (15 significant decimal digits)
	POINTER	
4/8	pointer	Pointer to another memory location, 32 or 64bit
		`double *d` or `int **ip` or `char *s` or `void *p` (!?)
	array	Pointer to a fixed location
		`double []` or `int [][]` or `char []`

*Number of bytes for each type is NOT standard but sizes shown are common. Portable code should NOT assume any particular size which is a huge pain in the `@$$`.

Inspect types closely

Ranges of integral types?

Missing types you expected?

`void` what now?

How do you say `char`?

Answers: Traditional C Data Types

Ranges of signed integral types

Asymmetric: slightly more negative than positive

```
char   | -128 to 127
```

Due to use of two's complement representation, many details and alternatives later in the course.

Missing: Boolean, String

- Every piece of data in C is either truthy or falsey:

  ```
  int x; scanf("%d", &x);
  if(x){ printf("Truthy"); }  // very common
  else { printf("Falsey"); }
  ```

  Typically `0` is the only thing that is falsey
- No `String` type: arrays of `char` like `char str[]` or `char *s`

`char` pronounced **CAR** like "character" (debatable)

Exercise: Void Pointers

`void *ptr; // void pointer`

- Declares a pointer to something/anything
- Useful in some contexts: "I just need memory"
- Removes compiler's ability to type check so avoid when possible

Ex: `void_pointer.c`

- Predict output
- What looks screwy
- Anything look wrong?

File `void_pointer.c`:

```c
#include <stdio.h>
int main(){
  int a = 5;
  double x = 1.2345;
  void *ptr;

  ptr = &a;
  int b = *((int *) ptr);
  printf("%d\n",b);

  ptr = &x;
  double y = *((double *) ptr);
  printf("%f\n",y);

  int c = *((int *) ptr);
  printf("%d\n",c);

  return 0;
}
```

Answers: Void Pointers

```
> cat -n void_pointer.c
     1 // Demonstrate void pointer dereferencing and the associated
     2 // shenanigans.  Compiler needs to be convinced to dereference in most
     3 // cases and circumventing the type system (compiler's ability to
     4 // check correctness) is fraught with errors.
     5 #include <stdio.h>
     6 int main(){
     7   int a = 5;                      // int
     8   double x = 1.2345;              // double
     9   void *ptr;                      // pointer to anything
    10
    11   ptr = &a;
    12   int b = *((int *) ptr);         // caste to convince compiler to deref
    13   printf("%d\n",b);
    14
    15   ptr = &x;
    16   double y = *((double *) ptr); // caste to convince compiler to deref
    17   printf("%f\n",y);
    18
    19   int c = *((int *) ptr);         // kids: this is why types are useful
    20   printf("%d\n",c);
    21
    22   return 0;
    23 }
> gcc void_pointer.c
> a.out
5
1.234500
309237645    # interpreting floating point bits as an integer
```

But wait, there're more types...

Unsigned Variants

Trade sign for larger positives

Name	Range
`unsigned char`	0 to 255
`unsigned short`	0 to 65,535
`unsigned int`	0 to 4,294,967,295
`unsigned long`	0 to... big, okay?

After our C crash course, we will discuss representation of integers with bits and relationship between signed / unsigned integer types

Fixed Width Variants since C99

Specify size / properties

`int8_t`
`int16_t`
`int32_t`
`int64_t`
	signed integer type with width of
	exactly 8, 16, 32 and 64 bits respectively

`int_fast8_t`
`int_fast16_t`
`int_fast32_t`
`int_fast64_t`
	fastest signed integer type with width of
	at least 8, 16, 32 and 64 bits respectively

`int_least8_t`
`int_least16_t`
`int_least32_t`
`int_least64_t`
	smallest signed integer type with width of
	at least 8, 16, 32 and 64 bits respectively

`intmax_t`	maximum width integer type
`intptr_t`	integer type capable of holding a pointer

`uint8_t`
`uint16_t`
`uint32_t`
`uint64_t`
	unsigned integer type with width of
	exactly 8, 16, 32 and 64 bits respectively

`uint_fast8_t`
`uint_fast16_t`
`uint_fast32_t`
`uint_fast64_t`
	fastest unsigned integer type with width of
	at least 8, 16, 32 and 64 bits respectively

`uint_least8_t`
`uint_least16_t`
`uint_least32_t`
`uint_least64_t`
	smallest unsigned integer type with width of
	at least 8, 16, 32 and 64 bits respectively

`uintmax_t`	maximum width unsigned integer type
`uintptr_t`	unsigned int capable of holding pointer

Relationship of Arrays and Pointers: Subtle differences

Property	Pointer	Array
Declare like...	`int *p; // rand val`	`int a[5]; // rand vals`
	`int *p = &x;`	`int a[] = {1, 2, 3};`
	`int *p = q;`	`int a[2] = {2, 4};`
Refers to a	Memory location	Memory location
Which could be	Anywhere	Fixed location
Location ref is	Changeable	Not changeable
Location...	Assigned by coder	Determined by compiler
Has at it..	One or more thing	One or more thing
Brace index?	Yep: `int z = p[0];`	Yep: `int z = a[0];`
Dereference?	Yep: `int y = *p;`	Nope
Arithmetic?	Yep: `p++;`	Nope
Assign to array?	Yep: `int *p = a;`	Nope
Interchangeable	`doit_a(int a[]);`	`doit_p(int *p);`
	`int *p = ...`	`int a[] = {1,2,3};`
	`doit_a(p);`	`doit_p(a);`
Tracks num elems	NOPE	NOPE
	Nada, nothin, nope	No `a.length`

Example: pointer_v_array.c

```c
// Demonstrate equivalence of pointers and arrays
#include <stdio.h>
void print0_arr(int a[]){                    // print 0th element of a
  printf("%ld: %d\n",(long) a, a[0]); // address and 0th elem
}
void print0_ptr(int *p){                   // print int pointed at by p
  printf("%ld: %d\n",(long) p, *p); // address and 0th elem
}
int main(){
  int *p = NULL;                     // declare a pointer, points nowhere
  printf("%ld: %ld\n",               // print address/contents of p
         (long) &p, (long)p);        // by casting to 64 bit long
  int x = 21;                        // declare an integer
  p = &x;                            // point p at x
  print0_arr(p);                     // pointer as array
  int a[] = {5,10,15};               // declare array, auto size
  print0_ptr(a);                     // array as pointer
  //a = p;                           // can't change where array points
  p = a;                             // point p at a
  print0_ptr(p);
  return 0;
}
```

Execution of Code/Memory 1

```
    #include <stdio.h>
    void print0_arr(int a[]){
      printf("%ld: %d\n",(long) a, a[0]);
    }
    void print0_ptr(int *p){
      printf("%ld: %d\n",(long) p, *p);
    }
    int main(){
      int *p = NULL;
      printf("%ld: %ld\n",
             (long) &p, (long)p);
<1>   int x = 21;
<2>   p = &x;
<3>   print0_arr(p);
      int a[] = {5,10,15};
      print0_ptr(a);
      //a = p;
<4>   p = a;
<5>   print0_ptr(p);
      return 0;
    }
```

Memory at indicated <POS>

```
<1>
| Addr  | Type | Sym  | Val |
|-------+------+------+-----|
| #4924 | int  | x    | ?   |
| #4928 | int* | p    | NULL|
| #4936 | int  | a[0] | ?   |
| #4940 | int  | a[1] | ?   |
| #4944 | int  | a[2] | ?   |
| #4948 | ?    | ?    | ?   |
<3>
| Addr  | Type | Sym  | Val   |
|-------+------+------+-------|
| #4924 | int  | x    | 21    |
| #4928 | int* | p    | #4924 |*
| #4936 | int  | a[0] | ?     |
| #4940 | int  | a[1] | ?     |
| #4944 | int  | a[2] | ?     |
| #4948 | ?    | ?    | ?     |
```

Execution of Code/Memory 2

```
    #include <stdio.h>
    void print0_arr(int a[]){
      printf("%ld: %d\n",(long) a, a[0]);
    }
    void print0_ptr(int *p){
      printf("%ld: %d\n",(long) p, *p);
    }
    int main(){
      int *p = NULL;
      printf("%ld: %ld\n",
             (long) &p, (long)p);
<1>   int x = 21;
<2>   p = &x;
<3>   print0_arr(p);
      int a[] = {5,10,15};
      print0_ptr(a);
      //a = p;
<4>   p = a;
<5>   print0_ptr(p);
      return 0;
    }
```

Memory at indicated <POS>

```
<4>
| Addr  | Type | Sym  | Val   |
|-------+------+------+-------|
| #4924 | int  | x    | 21    |
| #4928 | int* | p    | #4924 |
| #4936 | int  | a[0] | 5     |*
| #4940 | int  | a[1] | 10    |*
| #4944 | int  | a[2] | 15    |*
| #4948 | ?    | ?    | ?     |
<5>
| Addr  | Type | Sym  | Val   |
|-------+------+------+-------|
| #4924 | int  | x    | 21    |
| #4928 | int* | p    | #4936 |*
| #4936 | int  | a[0] | 5     |
| #4940 | int  | a[1] | 10    |
| #4944 | int  | a[2] | 15    |
| #4948 | ?    | ?    | ?     |
```

Exercise: Pointer Arithmetic

"Adding" to a pointer increases the position at which it points:

- Add 1 to an `int*`: point to the next `int`, add 4 bytes
- Add 1 to a `double*`: point to next `double`, add 8 bytes

Examine `pointer_arithmetic.c` below. Show memory contents and what's printed on the screen

```
    #include <stdio.h>
    void print0_ptr(int *p){
      printf("%ld: %d\n",(long) p, *p);
    }
    int main(){
      int x = 21;
      int *p;
      int a[] = {5,10,15};
<1>   p = a;
      print0_ptr(p);
      p = a+1;
<2>   print0_ptr(p);
      p++;
<3>   print0_ptr(p);
      p+=2;
<4>   print0_ptr(p);
      return 0;
    }
```

<1>

Addr	Type	Sym	Val
#4924	int	x	21
#4928	int*	p	?
#4936	int	a[0]	5
#4940	int	a[1]	10
#4944	int	a[2]	15
#4948	?	?	?

<2> ???

<3> ???

<4> ???

Answers: Pointer Arithmetic

```
    int main(){
      int x = 21;
      int *p;
      int a[] = {5,10,15};
<1>   p = a;
      print0_ptr(p);
      p = a+1;
<2>   print0_ptr(p);
      p++;
<3>   print0_ptr(p);
      p+=2;
<4>   print0_ptr(p);
      return 0;
    }
```

```
<2>
| Addr  | Type | Sym  |   Val |  SCREEN:
|-------+------+------+-------|  5
| #4924 | int  | x    |    21 |
| #4928 | int* | p    | #4940 |
| #4936 | int  | a[0] |     5 |
| #4940 | int  | a[1] |    10 |
| #4944 | int  | a[2] |    15 |
| #4948 | ?    | ?    |     ? |
```

```
<3>
| Addr  | Type | Sym  |   Val |  SCREEN:
|-------+------+------+-------|  5
| #4924 | int  | x    |    21 |  10
| #4928 | int* | p    | #4944 |
| #4936 | int  | a[0] |     5 |
| #4940 | int  | a[1] |    10 |
| #4944 | int  | a[2] |    15 |
| #4948 | ?    | ?    |     ? |
```

```
<4>
| Addr  | Type | Sym  |   Val |  SCREEN:
|-------+------+------+-------|  5
| #4924 | int  | x    |    21 |  10
| #4928 | int* | p    | #4952 |  15
| #4936 | int  | a[0] |     5 |
| #4940 | int  | a[1] |    10 |
| #4944 | int  | a[2] |    15 |
| #4948 | ?    | ?    |     ? |
| #4952 | ?    | ?    |     ? |
```

Pointer Arithmetic Alternatives

Pointer arithmetic often has more readable alternatives

```
printf("enter 5 doubles\n");
double arr[5];
for(int i=0; i<5; i++){
  // POINTER: ick                    // PREFERRED
  scanf("%lf", arr+i);      OR     scanf("%lf", &arr[i]);
}
printf("you entered:\n");
for(int i=0; i<5; i++){
  // POINTER: ick                    // PREFERRED
  printf("%f ", *(arr+i)); OR     printf("%f ",arr[i]);
}
```

But not always: following uses pointer arithmetic to append strings

```
char name[128];                      // up to 128 chars
printf("first name: ");
scanf(" %s", name);                  // read into name
int len = strlen(name);              // compute length of string
name[len] = ' ';                     // replace \0 with space
printf("last name: ");
scanf(" %s",name+len+1);             // read last name at offset
printf("full name: %s\n",name);
```

Allocating Memory with malloc() and free()

Dynamic Memory

- Most C data has a fixed size: single vars or arrays with sizes specified at compile time
- `malloc()` is used to dynamically allocate memory
    - single arg: number of bytes of memory
    - frequently used with `sizeof()` operator
    - returns a `void*` to bytes found or `NULL` if not enough space could be allocated
- `free()` is used to release memory

malloc_demo.c

```c
#include <stdio.h>
#include <stdlib.h> // malloc / free
int main(){
  printf("how many ints: ");
  int len;
  scanf(" %d",&len);

  int *nums = malloc(sizeof(int)*len);

  printf("initializing to 0\n");
  for(int i=0; i<len; i++){
    nums[i] = 0;
  }
  printf("enter %d ints: ",len);
  for(int i=0; i<len; i++){
    scanf(" %d",&nums[i]);
  }
  printf("nums are:\n");
  for(int i=0; i<len; i++){
    printf("[%d]: %d\n",i,nums[i]);
  }
  free(nums);
  return 0;
}
```

Exercise: Allocation Sizes

How Big

How many bytes allocated?
How many elements in the array?

```
char    *a = malloc(16);
char    *b = malloc(16*sizeof(char));
int     *c = malloc(16);
int     *d = malloc(16*sizeof(int));
double  *e = malloc(16);
double  *f = malloc(16*sizeof(double));
int    **g = malloc(16);
int    **h = malloc(16*sizeof(int*));
```

Allocate

- Want an array of `ints` called `ages`, quantity 32
- Want an array of `doubles` called `dps`, quantity is in variable `int size`

Deallocate

Code to deallocate `ages` / `dps`

How many bytes CAN be allocated?

- Examine `malloc_all_memory.c`

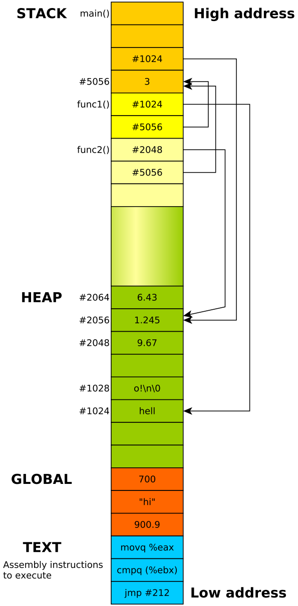
Answers: Allocation Sizes

```
char    *a = malloc(16);
char    *b = malloc(16*sizeof(char));
int     *c = malloc(16);
int     *d = malloc(16*sizeof(int));
double  *e = malloc(16);
double  *f = malloc(16*sizeof(double));
int    **g = malloc(16);
int    **h = malloc(16*sizeof(int*));

int *ages = malloc(sizeof(int)*32);
int size = ...;
double *dps = malloc(sizeof(double)*size);

free(ages);
free(dps);
```

Where does `malloc()'d` memory come from anyway...

The Parts of Memory

- Running program typically has 4 regions of memory
  1. Stack: automatic, push/pop with function calls
  2. Heap: `malloc()` and `free()`
  3. Global: variables outside functions, `static` vars
  4. Text: Assembly instructions
- Stack grows into Heap, hitting the boundary results in **stack overflow**
- Will study ELF file format for storing executables
- Heap uses **memory manager**, will do an assignment on this



Memory Tools on Linux/Mac

Valgrind[1]: Suite of tools including Memcheck

- Catches most memory errors[2]
  - Use of uninitialized memory
  - Reading/writing memory after it has been free'd
  - Reading/writing off the end of malloc'd blocks
  - Memory leaks
- Source line of problem happened (but not cause)
- Super easy to use
- Slows execution of program *way* down

[1] http://valgrind.org/
[2] http://en.wikipedia.org/wiki/Valgrind

Valgrind in Action

See some common problems in badmemory.c

```
# Compile with debugging enabled: -g
> gcc -g badmemory.c

# run program through valgrind
> valgrind a.out
==12676== Memcheck, a memory error detector
==12676== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==12676== Using Valgrind-3.10.1 and LibVEX; rerun with -h for copyright info
==12676== Command: a.out
==12676==
Uninitialized memory
==12676== Conditional jump or move depends on uninitialised value(s)
==12676==    at 0x4005C1: main (badmemory.c:7)
==12676==
==12676== Conditional jump or move depends on uninitialised value(s)
==12676==    at 0x4E7D3DC: vfprintf (in /usr/lib/libc-2.21.so)
==12676==    by 0x4E84E38: printf (in /usr/lib/libc-2.21.so)
==12676==    by 0x4005D6: main (badmemory.c:8)
...
```

Link: Description of common Valgrind Error Messages

Exercise: Memory Review

Last lecture is on Youtube and answered the following questions

1. How do you allocate memory on the Stack? How do de-allocate it?
2. How do you allocate memory on the Heap? How do de-allocate it?
3. What other parts of memory are there in programs?
4. How do you declare an array of 8 integers in C? How big is it and what part of memory is it in?
5. Describe several ways arrays and pointers are similar.
6. Describe several ways arrays and pointers are different.
7. Describe how the following to arithmetic expressions differ.

```
int  x=9, y=20;
int *p = &x;
x = x+1;
p = p+1;
```

Answers: Memory Review

1. How do you allocate memory on the Stack? How do de-allocate it?

   Declare local variables in a function and call it. Stack frame has memory for all locals and is de-allocated when the function returns.

2. How do you allocate memory on the Heap? How do de-allocate it?

   Make a call to ptr = malloc(nbytes) which returns a pointer to the requested number of bytes. Call free(ptr) to de-allocate that memory.

3. What other parts of memory are there in programs?

   Global area of memory has constants and global variables. Text area has binary assembly code for CPU instructions.

4. How do you declare an array of 8 integers in C? How big is it and what part of memory is it in?

   On the stack: int arr[8]; De-allocated when function returns.

   *On the heap: int *arr = malloc(sizeof(int) * 8); Deallocated with free(arr);*

Answers: Memory Review

5. Describe several ways arrays and pointers are similar.

   Both usually encoded as an address, can contain 1 or more items, may use square brace indexing like arr[3] = 17; Interchangeable as arguments to functions. Neither tracks size of memory area referenced.

6. Describe several ways arrays and pointers are different.

   *Pointers may be deref'd with *ptr; can't do it with arrays. Can change where pointers point, not arrays. Arrays will be on the Stack or in Global Memory, pointers may also refer to the Heap.*

7. Describe how the following to arithmetic expressions differ.

```
int  x=9, y=20;    // x at #1024
int *p = &x;
x = x+1;           // x is now 10:    normal arithmetic
p = p+1;           // p is now #1028: pointer arithmetic
                   // may or may not point at y
```

Exercise: `free()`'ing in the Wrong Spot

Program to the right is buggy, produces following output on one system

```
> gcc free_twice.c
> ./a.out
ones[0] is 0
ones[1] is 0
ones[2] is 1
ones[3] is 1
ones[4] is 1
```

- Why does this bug happen?
- How can it be fixed?
- Answers in `free_twice.c`

```c

#include <stdlib.h>
#include <stdio.h>

int *ones_array(int len){
  int *arr = malloc(sizeof(int)*len);
  for(int i=0; i<len; i++){
    arr[i] = 1;
  }
  free(arr);
  return arr;
}

int main(){
  int *ones = ones_array(5);
  for(int i=0; i<5; i++){
    printf("ones[%d] is %d\n",i,ones[i]);
  }

  free(ones);
  return 0;
}

```

structs: Heterogeneous Groupings of Data

- Arrays are homogenous: all elements the same type
- structs are C's way of defining heterogenous data
- Each **field** can be a different kind
- One instance of a struct has all fields
- Access elements with 'dot' notation
- Several syntaxes to declare, we'll favor modern approach
- Convention: types have _t at the end of their name to help identify them (not a rule but a good idea)

```
typedef struct{  // declare type
  int    field1;
  double field2;
  char   field3;
  int    field4[6];
} thing_t;

thing_t a_thing;    // variable
a_thing.field1    = 5;
a_thing.field2    = 9.2;
a_thing.field3    = 'c';
a_thing.field4[2] = 7;
int i = a_thing.field1;


thing_t b_thing = {  // variable
  .field1 = 15,      // initialize
  .field2 = 19.2,    // all fields
  .field3 = 'D',
  .field4 = {17, 27, 37,
             47, 57, 67}
};
```

`struct` Ins/Outs

Recursive Types

- `struct`s can have pointers to their same kind
- Syntax is a little wonky

```
typedef struct node_struct {
  char data[128];
  struct node_struct *next;
} node_t;
```

Arrow Operator

- Pointer to struct, want to work with a field
- Use 'arrow' operator `->` for this (dash/greater than)

Dynamically Allocated Structs

- Dynamic Allocation of `struct`s requires size calculation
- Use `sizeof()` operator

```
node_t *one_node =
  malloc(sizeof(node_t));
int length = 5;
node_t *node_arr =
  malloc(sizeof(node_t) * length);
```

```
node_t *node = ...;
if(node->next == NULL){ ... }

list_t *list = ...;
list->size = 5;
list->size++;
```

Exercise: Structs in Memory

- Structs allocated in memory are laid out compactly
- Compiler may *pad* fields to place them at nice alignments (even addresses or word boundaries)

```
typedef struct {
  double x;
  int y;
  char nm[4];
} small_t;

int main(){
  small_t a =
    {.x=1.23, .y=4, .nm="hi"};
  small_t b =
    {.x=5.67, .y=8, .nm="bye"};
}
```

Memory layout of `main()`

```
| Addr  | Type   | Sym     | Val  |
|-------+--------+---------+------|
| #1000 | double | a.x     | 1.23 |
| #1008 | int    | a.y     | 4    |
| #1012 | char   | a.nm[0] | h    |
| #1013 | char   | a.nm[1] | i    |
| #1014 | char   | a.nm[2] | \0   |
| #1015 | char   | a.nm[3] | ?    |
| #1016 | double | b.x     | 5.67 |
| #1024 | int    | b.y     | 8    |
| #1028 | char   | b.nm[0] | b    |
| #1029 | char   | b.nm[1] | y    |
| #1030 | char   | b.nm[2] | e    |
| #1031 | char   | b.nm[3] | \0   |
```

Result of?

```
scanf("%d",  &a.y); // input 7
scanf("%lf", &b.x); // input 9.4
scanf("%s",  b.nm); // input yo
```

Answers: Structs in Memory

```
scanf("%d",  &a.y); // input 7
scanf("%lf", &b.x); // input 9.4
scanf("%s",  b.nm); // input yo
```

Addr	Type	Sym	Val Before	Val After
#1000	double	a.x	1.23	
#1008	int	a.y	4	7
#1012	char	a.nm[0]	h	
#1013	char	a.nm[1]	i	
#1014	char	a.nm[2]	\0	
#1015	char	a.nm[3]	?	
#1016	double	b.x	5.67	9.4
#1024	int	b.y	8	
#1028	char	b.nm[0]	b	y
#1029	char	b.nm[1]	y	o
#1030	char	b.nm[2]	e	\0
#1031	char	b.nm[3]	\0	

read_structs.c: malloc() and scanf() for structs

```c
// Demonstrate use of pointers, malloc() with structs, scanning
// structs fields

#include <stdlib.h>
#include <stdio.h>

typedef struct {                // simple struct
  double x;     int y;    char nm[4];
} small_t;

int main(){
  small_t c;                                    // stack variable
  small_t *cp = &c;                             // address of stack var
  scanf("%lf %d %s", &cp->x, &cp->y, cp->nm); // read struct fields
  printf("%f %d %s\n",cp->x, cp->y, cp->nm);  // print struct fields

  small_t *sp = malloc(sizeof(small_t));      // malloc'd struct
  scanf("%lf %d %s", &sp->x, &sp->y, sp->nm); // read struct fields
  printf("%f %d %s\n",sp->x, sp->y, sp->nm);  // print struct fields

  small_t *sarr = malloc(5*sizeof(small_t));  // malloc'd struct array
  for(int i=0; i<5; i++){
    scanf("%lf %d %s", &sarr[i].x, &sarr[i].y, sarr[i].nm); // read
    printf("%f %d %s\n", sarr[i].x, sarr[i].y, sarr[i].nm); // print
  }

  free(sp);                                     // free single struct
  free(sarr);                                   // free struct array
  return 0;
}
```

File Input and Output

- Standard C I/O functions for reading/writing file data.
- Work with text data: formatted for human reading

```
FILE *fopen(char *fname, char *mode);
// open file named fname, mode is "r" for reading, "w" for writing
// returns a File Handle (FILE *) on success
// returns NULL if not able to open file

int fclose(FILE *fh);
// close file associated with fh, write any data to the

int fscanf(FILE *fh, char *format, addr1, addr2, ...);
// read data from an open file handle according to format string
// storing parsed tokens in given addresses returns EOF if end of file
// is reached

int fprintf(FILE *fh, char *format, arg1, arg2, ...);
// prints data to an open file handle according to the format string
// and provided arguments

void rewind(FILE *fh);
// return the given open file handle to the beginning of the file.
```

Example of use in `struct_text_io.c`

Binary Data I/O Functions

- Open/close files same way with `fopen()/fclose()`
- Read/write raw bytes (not formatted) with the following

```
size_t fread(void *dest, size_t byte_size, size_t quantity, FILE *fh);
// read binary data from an open file handle. Attempt to read
// byte_size*quantity bytes into the buffer pointed to by dest.
// Returns number of bytes that were actually read

size_t fwrite(void *src, size_t byte_size, size_t quantity, FILE *fh);
// write binary data to an open file handle. Attempt to write
// byte_size*quantity bytes from buffer pointed to by src.
// Returns number of bytes that were actually written
```

Example of use in `struct_binary_io.c`

- Binary files usually smaller than text but NOT easy on the eyes
- Text data more readable but more verbose, must be parsed

Strings are Character Arrays

Conventions

- Convention in C is to use character arrays as strings
- Terminate character arrays with the \0 null character to indicate their end

```
char str1[6] =
{'C','h','r','i','s','\0'};
```

- Done automatically by compiler for string constants

```
char str2[6] = "Chris";
```

- Done by most standard library functions (`scanf()`)

Be aware

- `fread()` does not append nulls when reading binary data
- Manually manipulating a character array may overwrite ending null

String Library

- Include with `<string.h>`
- Null termination expected
- `strlen(s)`: length of string
- `strcpy(dest, src)`: copy chars from `src` to `dest`
- Limited number of others

Common C operators

Arithmetic	`+ - * / %`
Comparison	`== > < <= >= !=`
Logical	`&& || !`
Memory	`&` and `*`
Compound	`+= -= *= /= ...`
Bitwise Ops	`^ | & ~`
Conditional	`? :`

Bitwise Ops

Will discuss soon

```
int x = y << 3;
int z = w & t;
long r = x | z;
```

Keep in mind...

Integer versus real division:
values for each of these?

```
int q = 10 / 4;
int r = 10 % 4;
double x = 10 / 4;
double y = ((double)10) / 4;
double z = 10.0 / 4;
double w = 10 / 4.0;
double t = q / r;
```

Conditional (ternary) Operator

```
double x = 9.95;
int y = (x < 10.0) ? 2 : 4;
```

C Control Structures

Looping/Iteration

```
// while loop
while(truthy){
  stuff;
  more stuff;
}

// for loop
for(init; truthy; update){
  stuff;
  more stuff;
}

// do-while loop
do{
  stuff;
  more stuff;
} while( truthy );
```

Conditionals

```
// simple if
if( truthy ){
  stuff;
  more stuff;
}

// chained exclusive if/elses
if( truthy ){
  stuff;
  more stuff;
}
else if(other){
  stuff;
}
else{
  stuff;
  more stuff;
}

// ternary ? : operator
int x = (truthy) ? yes : no;
```

Jumping Around in Loops

break: often useful

```
// break statement ends loop
// only valid in a loop
while(truthy){
  stuff;
  if( istrue ){
    something;
    break;-----+
  }            |
  more stuff;  |
}              |
after loop; <--+

// break ends inner loop,
// outer loop advances
for(int i=0; i<10; i++){
  for(int j=0; j<20; j++){
    printf("%d %d ",i,j);
    if(j == 7){
      break;-----+
    }            |
  }              |
  printf("\n");<-+
}
```

continue: occasionally useful

```
// continue advances loop iteration
// does update in for loops

                         +-------+
                         V       |
for(int i=0; i<10; i++){  |
  printf("i is %d\n",i);  |
  if(i % 3 == 0){         |
    continue;-------------+
  }
  printf("not div 3\n");
}

Prints
i is 0
i is 1
not div 3
i is 2
not div 3
i is 3
i is 4
not div 3
...
```

Really Jumping Around: goto

- Machine-level control involves jumping to different instructions
- C exposes this as
  - `somewhere:` label for code position
  - `goto somewhere;` jump to that location
- `goto_demo.c` demonstrates a loop with gotos
- **Avoid** `goto` unless you have a compelling motive
- Beware spaghetti code...

```c
// Demonstrate control flow with goto
// Low level assembly jumps are similar
#include <stdio.h>
int main(){
  int i=0;
 beginning:      // a label for gotos
  printf("i is %d\n",i);
  i++;
  if(i < 10){
    goto beginning; // go back
  }
  goto ending;      // go forward
  printf("print me please!\n");
 ending:            // label for goto
  printf("i ends at %d\n",i);
  return 0;
}
```

XKCD #292

`switch()/case`: the **worst** control structure

- `switch/case` allows jumps based on an integral value
- Frequent source of errors
- `switch_demo.c` shows some features
  - use of `break`
  - fall through cases
  - `default` catch-all
  - Use in a loop
- May enable some small compiler optimizations
- Almost **never** worth correctness risks: one good use in my experience
- **Favor** `if/else if/else` unless compelled otherwise

```c
// Demonstrate peculiarities of switch/case
#include <stdio.h>
int main(){
  while(1){
    printf("enter a char: ");
    char c;
    scanf(" %c",&c); // ignore preceding spaces
    switch(c){       // switch on read char
      case 'j':      // entered j
        printf("Down line\n");
        break;       // go to end of switch
      case 'a':      // entered a
        printf("little a\n");
      case 'A':      // entered A
        printf("big A\n");
        printf("append mode\n");
        break;       // go to end of switch
      case 'q':      // entered q
        printf("Quitting\n");
        return 0;    // return from main
      default:       // entered anything else
        printf("other '%c'\n",c);
        break;       // go to end of switch
    }                // end of switch
  }
  return 0;
}
```

A Program is Born: Compile, Assemble, Link, Load

- Write some C code in `program.c`
- Compile it with toolchain like GNU Compiler Collection

  `gcc -o program prog.c`
- Compilation is a multi-step process
  - Check syntax for correctness/errors
  - Perform optimizations on the code if possible
  - Translate result to **Assembly Language** for a specific target processor (Intel, ARM, Motorola)
  - **Assemble** the code into **object code**, binary format (ELF) which the target CPU understands
  - **Link** the binary code to any required libraries (e.g. printing) to make an **executable**
- Result: executable `program`, but...
- To run it requires a **loader**: program which copies executable into memory, initializes any shared library/memory references required parts, sets up memory to refer to initial instruction