CMSC216: C Basics

Chris Kauffman

Last Updated:
Mon Feb 9 06:15:16 PM EST 2026

CMSC216 Logistics

Reading

- C references (any / all), whole language:

  types, pointers, addresses, arrays, conditionals, loops, structs, strings, malloc/free, preprocessor, compilation etc.
- C References are any of...
  - "The C Programming Language" book by Kernighan / Ritchie
  - Free refs linked at bottom of ELMS/Canvas Frontpage

Assignments

- Lab03 due Sun
- P1 Due Mon
- Exam 1 Next Week

Past students to Present concerning Project 1:
Start EARLY, end Happy (or at least less stressed)

Goals

Wrap up overview of basic C / machine semantics

Announcements

None

Every Programming Language

Look for the following as it should almost always be there

- ☒ Comments
- ☒ Statements/Expressions
- ☒ Variable Types
- ☒ Assignment
- ☐ Basic Input/Output (`printf()` and `scanf()` from HW1)
- ☒ 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`	±3.40282347E±38 (6-7 significant decimal digits)
8	`double`	±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 and discuss the following:

1. Ranges of integral types?
2. Missing types you expected?
3. `void` what now?
4. How do you say `char`?

Answers: Traditional C Data Types

Ranges of signed integral types

Asymmetric: slightly more negative than positive

`char is -128 to 127`

Due to use of **Two's Complement** representation, many details and alternatives later in the course.

Missing: Boolean

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

Missing: String

- `char` holds a single character like `'A'` or `'5'`
- No `String` type: arrays of `char` like `char str[]` or `char *s`
- `char` pronounced **CAR / CARE** like "character" (debatable)

Recall: Pointers, Addresses, Dereferences

`type *ptr;`	**Declares** a pointer variable
`type* ptr;`	**Declares** a pointer variable[1]
`*ptr = val;`	**Dereferences** pointer to set value pointed at
`other = *ptr;`	**Dereferences** pointer to get value pointed at

```c
int *iptr;                   // Declare a pointer
int x = 7;                   // Declare/set an int
iptr = &x;                   // Set pointer
int y = *iptr;               // Deref-ptr, gets x
*iptr = 9;                   // Deref-set ptr, changes x

double z = 1.23;             // Declare/set double
double *dptr = &z;           // Declare/set double ptr
*dptr = 4.56;                // Deref-set ptr, changes z

printf("x: %d z: %f\n",      // print via derefs
        *iptr, *dptr);
```

Declaring pointer variables to specific types is the *normal and safest* way to write C code but can be circumvented

[1] While `int *p;` and `int* p;` do the same thing, placing the `*` next to the variable name is the more common style in C for cases like `int a, *p, b;`

Normal Pointers are Typed

Compiler enforces that `int*` pointers point at integers and nothing else. Code violating this will generate **Compiler-Time Errors** in the general category of a **Type Error**

```c
// pointer_type_error.c: compiler will detect and
// error when assigning a pointer to refer to the
// wrong type of data. This code has an
// intentional error and WILL NOT COMPILE.

#include <stdio.h>
int main(){
  int a = 10;
  int *aptr = &a;     // int pointer to int
  double b = 4.56;
  double *bptr = &b; // double pointer to double
  aptr = &b;         // ERROR: int pointer to double
  printf("*aptr is %d\n",*aptr);
  return 0;
}
// >> gcc pointer_type_error.c
// pointer_type_error.c: In function main:
// pointer_type_error.c:12:8:: error: assignment to
// int * from incompatible pointer type double *
// [-Wincompatible-pointer-types]
```

Exercise: Legacy of the Void (Pointer)

`void *ptr; // void pointer`

- Declares a pointer to something/anything
- Useful to store an arbitrary memory address
- Removes compiler's ability to **Type Check** so introduces risks managed by the programmer

Example: `void_pointer.c`

- Predict output
- What looks screwy?

```c
// void_pointer.c: pluses and perils
#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: Legacy of the Void (Pointer)

```
>> 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);         // typecast to convince compiler to deref
    13    printf("%d\n",b);
    14
    15    ptr = &x;
    16    double y = *((double *) ptr); // typecast 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 half of a double as an integer
```

Byte-level Picture of Memory at `main()` line 20

```
|-------+-----+--------+-----------+------+------------|
| BYTE  |     | VALUE  | VALUE     |  VAL |  int VALUE |
| ADDR  | SYM | TYPED  | BINARY    |  HEX | in DECIMAL |
|-------+-----+--------+-----------+------+------------|
| #2043 | ptr | v      | 0000 0000 | 0x00 |            | void *ptr occupies
| #2042 | ptr | v      | 0000 0000 | 0x00 |            | 8 contiguous bytes
| #2041 | ptr | v      | 0000 0000 | 0x00 |            | from #2036-#2043
| #2040 | ptr | v      | 0000 0000 | 0x00 |          0 | and currently points
| #2039 | ptr | v      | 0000 0000 | 0x00 |            | at #2028; the bits/bytes
| #2038 | ptr | v      | 0000 0000 | 0x00 |            | there must be typecast
| #2037 | ptr | v      | 0000 0111 | 0x07 |            | in order to dereference
| #2036 | ptr | #2028  | 1110 1100 | 0xec |       2028 |
| #2035 | x   | v      | 0011 1111 | 0x3f |            | double x occupies
| #2034 | x   | v      | 1111 0011 | 0xf3 |            | 8 contiguous bytes
| #2033 | x   | v      | 1100 0000 | 0xc0 |            | from #2028-#2035
| #2032 | x   | v      | 1000 0011 | 0x83 | 1072939139 | but ptr points to
| #2031 | x   | v      | 0001 0010 | 0x12 |            | #2028 and prints bytes
| #2030 | x   | v      | 0110 1110 | 0x6e |            | #2028-2031 as a 4-byte
| #2029 | x   | v      | 1001 0111 | 0x97 |            | integer
| #2028 | x   | 1.2345 | 1000 1101 | 0x8d |  309237645 |
| #2027 | a   | v      | 0000 0000 | 0x00 |            | int a occupies
| #2026 | a   | v      | 0000 0000 | 0x00 |            | 4 contiguous bytes
| #2025 | a   | v      | 0000 0000 | 0x00 |            | from #2024-#2027
| #2024 | a   | 5      | 0000 0101 | 0x05 |          5 |
|-------+-----+--------+-----------+------+------------|
```

Answers: Legacy of the Void (Pointer)

- The big weird integer `309237645` printed at the end is because...
  - `ptr` points at a memory location with a `double`
  - The compiler is "tricked" into treating this location as storing `int` data via the `(int *)` typecast
  - Integer vs Floating bit layout is **very different**; we'll study this difference (briefly) later
  - Compiler generates low level instructions to move 4 bytes of the `double` data to an integer location
  - Both size and bit layout don't match
- Since this is possible to do on a Von Neumann machine C makes it possible
- This does not mean it is a good idea: `void_pointer.c` illustrates **weird code** that is atypical and error-prone
- Avoid `void *` pointers when possible, take care when you must use them (there are *many times* you must use them in C)

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

Arrays in C

- Array: a continuous block of homogeneous data
- Automatically allocated by the compiler/runtime with a **fixed size** [1]
- Support the familiar `[ ]` syntax
- Refer to a single element via `arr[3]`
- **Bare name** `arr` **is the memory address where array starts**

```
{
  int x     = 42;
  int *p    = &x;
  int a[3] = {10,20,30};
  int *ap   = a;
}
| Addr  | Type | Sym  |   Val |
|-------+------+------+-------|
| #4948 | int* | ap   | #4936 |
| #4944 | int  | a[2] |    30 |
| #4940 | int  | a[1] |    20 |
| #4936 | int  | a[0] |    10 |
| #4928 | int* | p    | #4924 |
| #4924 | int  | x    |    42 |
```

[1] Modern C supports variable sized arrays in the stack but we will not use them until much later in the class. The are NOT a general substitute for heap-allocation with `malloc()`

Arrays and Pointers are Related with 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` or `length(a)`

Example: pointer_v_array.c

```c
// pointer_v_array.c: Demonstrate equivalence of pointers and
// arrays. An array is represented by its starting address so can be
// passed to a function taking a pointer as such. Similarly, a pointer
// value is an address so can be passed to a function taking an array
// argument. printf("%p") prints pointer values in hexadecimal format.

#include <stdio.h>

void print0_arr(int a[]){        // print 0th element of a
  printf("%p: %d\n", a, a[0]);   // address and 0th elem
}
void print0_ptr(int *p){         // print int pointed at by p
  printf("%p: %d\n", p, *p);     // address and 0th elem
}
int main(){
  int *p = NULL;                 // declare a pointer, points nowhere
  printf("%p: %p\n", &p, p);     // print address/contents of p
  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

```
     void print0_arr(int a[]){
       printf("%p: %d\n", a, a[0])
     }
     void print0_ptr(int *p){
       printf("%p: %d\n", p, *p);
     }
     int main(){
       int *p = NULL;
       printf("%p: %p\n", &p, 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 |
|-------+------+------+-----|
| #4948 | ?    | ?    | ?   |
| #4944 | int  | a[2] | ?   |
| #4940 | int  | a[1] | ?   |
| #4936 | int  | a[0] | ?   |
| #4928 | int* | p    | NULL|
| #4924 | int  | x    | ?   |
<3>
| Addr  | Type | Sym  | Val   |
|-------+------+------+-------|
| #4948 | ?    | ?    | ?     |
| #4944 | int  | a[2] | ?     |
| #4940 | int  | a[1] | ?     |
| #4936 | int  | a[0] | ?     |
| #4928 | int* | p    | #4924 |*
| #4924 | int  | x    | 21    |
```

Execution of Code/Memory 2

```c
    void print0_arr(int a[]){
      printf("%p: %d\n", a, a[0])
    }
    void print0_ptr(int *p){
      printf("%p: %d\n", p, *p);
    }
    int main(){
      int *p = NULL;
      printf("%p: %p\n", &p, 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   |
|-------+------+------+-------|
| #4948 | ?    | ?    | ?     |
| #4944 | int  | a[2] | 15    |*
| #4940 | int  | a[1] | 10    |*
| #4936 | int  | a[0] | 5     |*
| #4928 | int* | p    | #4924 |
| #4924 | int  | x    | 21    |
<5>
| Addr  | Type | Sym  | Val   |
|-------+------+------+-------|
| #4948 | ?    | ?    |     ? |
| #4944 | int  | a[2] | 15    |
| #4940 | int  | a[1] | 10    |
| #4936 | int  | a[0] | 5     |
| #4928 | int* | p    | #4936 |*
| #4924 | int  | x    | 21    |
```

Summary of Pointer / Array Relationship

Arrays

- Arrays are allocated by the Compiler at a **fixed location**
- **Bare name** `a` references is the starting address of the array
- Must use square braces `a[i]` to index into them

Pointers

- Pointers can point to anything, can change, must be manually directed
- Can use square braces `p[i]` or deref `*p` to index into them

Interchangeability

- In most cases, functions that require an array can be passed a pointer, functions that that require a pointer can be passed an array
- Works BECAUSE array variables are passed as their starting memory address, a pointer value

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

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

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

SCREEN:
4936: 5

<2> ???
<3> ???
<4> ???
```

Answers: Pointer Arithmetic

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

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

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

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

```
Out of bounds deref of #4952 is
undefined behavior; may print
random garbage values or may
Segfault and killing the program.
```

Pointer Arithmetic Alternatives

Alternatives to pointer arithmetic exist that improve readability

```
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]);
}
```

However, some situations benefit from pointer manipulations, often in string processing like the following:

```
// read_name.c : string processing example
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);
```

read_name.c : String Functions + Pointer Arithmetic

```
INITIAL MEMORY       STEP 1                 STEP 2                 STEP 3
char name[128]       scanf(" %s", name);
// space for a 128   // Enters 'Chris'                             scanf(" %s", name+len+1);
// chars (a string)  len = strlen(name);    name[len] = ' ';       // Enter 'Kauffman'
     | ...   |   |        | ...   |      |       | ...   |      |       | ...   |      |
     | #1038 | ? |        | #1038 | ?    |       | #1038 | ?    |       | #1038 | '\0' |
     | #1037 | ? |        | #1037 | ?    |       | #1037 | ?    |       | #1037 | 'n'  |
     | #1036 | ? |        | #1036 | ?    |       | #1036 | ?    |       | #1036 | 'a'  |
     | #1035 | ? |        | #1035 | ?    |       | #1035 | ?    |       | #1035 | 'm'  |
     | #1034 | ? |        | #1034 | ?    |       | #1034 | ?    |       | #1034 | 'f'  |
     | #1033 | ? |        | #1033 | ?    |       | #1033 | ?    |       | #1033 | 'f'  |
     | #1032 | ? |        | #1032 | ?    |       | #1032 | ?    |       | #1032 | 'u'  |
     | #1031 | ? |        | #1031 | ?    |       | #1031 | ?    |       | #1031 | 'a'  |
     | #1030 | ? |        | #1030 | ?    |       | #1030 | ?    |       | #1030 | 'K'  |
     | #1029 | ? |        | #1029 | '\0' |       | #1029 | ' '  |       | #1029 | ' '  |
     | #1028 | ? |        | #1028 | 's'  |       | #1028 | 's'  |       | #1028 | 's'  |
     | #1027 | ? |        | #1027 | 'i'  |       | #1027 | 'i'  |       | #1027 | 'i'  |
     | #1026 | ? |        | #1026 | 'r'  |       | #1026 | 'r'  |       | #1026 | 'r'  |
     | #1025 | ? |        | #1025 | 'h'  |       | #1025 | 'h'  |       | #1025 | 'h'  |
name | #1024 | ? |   name | #1024 | 'C'  |  name | #1024 | 'C'  |  name | #1024 | 'C'  |
len  | #1020 | ? |   len  | #1020 | 5    |  len  | #1020 | 5    |  len  | #1020 | 5    |

                     Initial scanf() +      Overwrite null char    Read in after space
                     strlen()               with a space           using scanf()
```

Note the null character \0 terminates “standard” strings in C, honored by standard string functions like `printf()`, `strlen()`, `strcpy()`, etc.

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

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

- Null termination done by compiler for string constants

```c
char str2[6] = "Chris";
// is null terminated
```

- Null termination done by most standard library functions like `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

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(nbytes)` is used to manually 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

```c
// malloc_demo.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;
}
```

Optional 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*));
```

How many bytes CAN be allocated?

- Examine `malloc_all_memory.c`

Allocate / Deallocate

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

Answers: Allocation Sizes

```c
char    *a = malloc(16);                 // 16
char    *b = malloc(16*sizeof(char));    // 16
int     *c = malloc(16);                 // 16
int     *d = malloc(16*sizeof(int));     // 64
double  *e = malloc(16);                 // 16
double  *f = malloc(16*sizeof(double));  // 128
int    **g = malloc(16);                 // 16
int    **h = malloc(16*sizeof(int*));    // 128

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

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

Compile and Runtime vs Memory Management

Compile Time Sizes

- Some sizes are known at **Compile Time**
- Compiler can calculate, sizes of fixed variables, arrays, sizes of stack frames for function calls and **automatically allocate** them
- Most of these are automatically managed on the **function call stack** and don't require use of `malloc() / free()`

Run Time Sizes

- Compiler can't predict the future, at **Run Time** programs must react to
  - Typed user input like names, Size of a file that is to be read
  - Elements to be added to a data structure
  - Memory allocated in one function and returned to another
- As these things are determined, `malloc()` is used to allocate memory in the **heap**, when it is finished `free()` it

Common Misconception: `sizeof(thing)`

- `sizeof(thing)` determines the **Compile Time Size** of `thing` from C source, encodes size in executable program
- Useful when `malloc()`'ing stuff as in

```
int *arr = malloc(count * sizeof(int));
```

- **NOT USEFUL** for size of arrays/strings

```
int *arr = ...;
int nelems = sizeof(arr);  // always 8 on 64-bit systems
// REASON: arr is an (int *) and pointers are 8 bytes big
double darr[4] = {};
int len = sizeof(darr);    // always 32 as in 32 bytes
```

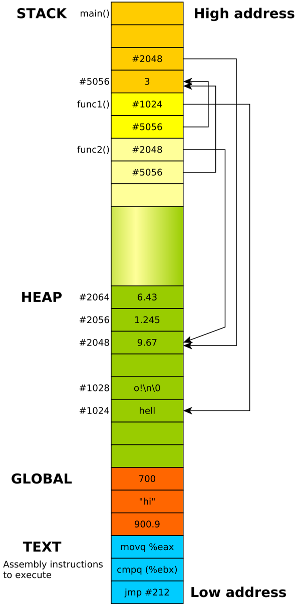
- To determine the size of arrays, must be given size OR have an ending sentinel value
- Strings commonly use `strlen()` to determine length:

```
char *str = "Hello world!\n";
int len = strlen(str);  // 13
```

See `sizeof_arrays.c` for some modest examples

The 4 Logical Regions of Program Memory

- Running program typically has 4 regions of memory
  1. **Stack**: automatic, push/pop with function calls
  2. **Heap**: `malloc() / free()`
  3. **Global**: variables outside functions, `static` vars
  4. **Text**: Program instructions in Binary
- Stack grows toward Heap, a collision results in *stack overflow*
- Global and Text regions usually fixed in size
- "Logical Regions" for Humans to organize their programs; no physical differences for regions



Memory Tools on Linux

The Valgrind Logo alluding to the Quixotic battle against the Dragon of memory errors

Valgrind[2]: Suite of tools including Memcheck

- Catches most memory errors[3]
  - 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 where problem arose (but not its cause)
- Super easy to use: `valgrind ./program ...`
- Slows execution of program *way* down

[2] http://valgrind.org/

[3] http://en.wikipedia.org/wiki/Valgrind

Examine: Valgrind in Action

Analyze Valgrind's errors for common mistakes in `badmemory.c`

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

# run program under 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: free()'ing in the Wrong Spot

Common use for `malloc()` is for one function to allocate memory and return its location to another function (such as in P1). Question becomes when to `free()` such memory.

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
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;
}
```

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

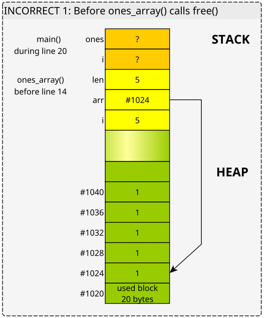
- Once a `malloc()`'d area is `free()`'d, it is no longer valid
- Don't `free()` data that is the `return` value of a function
- Never `free()` twice

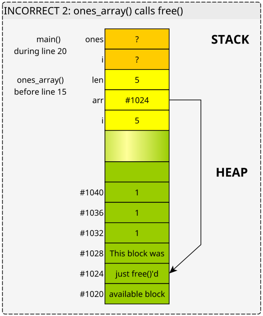
```
> gcc -g free_twice.c
> a.out
ones[0] is 0
ones[1] is 0
ones[2] is -1890717680
ones[3] is 22008
ones[4] is 1
free(): double free detected in tcache 2
Aborted (core dumped)

> valgrind a.out
==10125== Memcheck, a memory error detector
...
==10125== Invalid free()
==10125== at 0x48399AB: free
==10125== by 0x10921A: main (free_twice.c:24)
```

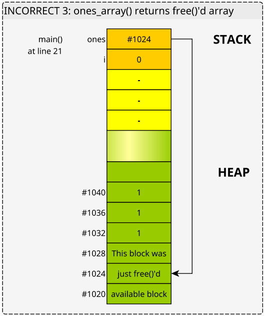
```c
 9 int *ones_array(int len){
10   int *arr = malloc(sizeof(int)*len);
11   for(int i=0; i<len; i++){
12     arr[i] = 1;
13   }
14   //free(arr); // should not free an array
15   return arr;  // being returned
16 }
17
18 int main(){
19   int *ones = ones_array(5);
20   for(int i=0; i<5; i++){
21     printf("ones[%d] is %d\n",i,ones[i]);
22   }
23
24   free(ones);  // later free makes
25   return 0;    // more sense
26 }
```

Note that the Valgrind output gives an **exact line number** where the problem occurs but this is **not the line to change** to fix the problem.

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



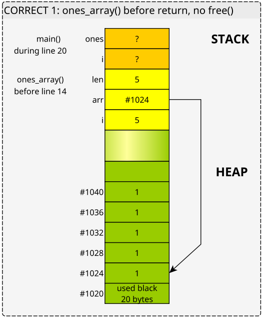


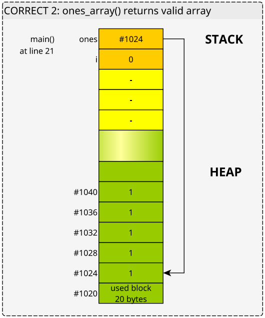


free_twice.c Program

ABOVE: Incorrect free version free()'s array before returning leading to main() getting a memory area that has no longer valid and has been marked for re-use by free().

BELOW: Corrected version which comments out the free() call in ones_array(); a valid memory area is returned which is printed by main() and then free()'d





`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)

```c
typedef struct{  // declare type
  int    an_int;
  double a_doub;
  char   the_car;
  int    my_arr[6];
} thing_t;

thing_t a_thing;    // variable
a_thing.an_int    = 5;
a_thing.a_doub    = 9.2;
a_thing.the_char  = 'c';
a_thing.my_arr[2] = 7;
int i = a_thing.an_int;


thing_t b_thing = {  // variable
  .an_int = 15,      // initialize
  .a_doub = 19.2,    // all fields
  .the_char = 'D',
  .my_arr = {17, 27, 37,
             47, 57, 67}
};
```

struct Ins/Outs

Recursive Types

- structs can have pointers to their same kind
- Syntax is a little wonky

```
                 vvvvvvvvvvv
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 structs 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)

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

Result of?

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

Structs: Dots vs Arrows

Newcomers wonder when to use Dots vs Arrows

- Use Dot (`s.field`) with an **Actual** struct
- Use Arrow (`p->field`) for a **Pointer** to a struct

```
small_t small;        // struct: 16 bytes
small_t *sptr;        // pointer: 8 bytes

sptr = &small;        // point at struct

small.x   = 1.23;     // actual struct
sptr->x   = 4.56;     // through pointer
(*sptr).x = 4.56;     // ICK: not preferred

small.y = 7;          // actual struct
sptr->y = 11;         // through pointer

small.nm[0] = 'A';    // through struct
sptr->nm[1] = 'B';    // through pointer
sptr->nm[2] = '\0';   // through pointer
```

Memory at end of code on left

```
| Addr  | Sym         | Value |
|-------+-------------+-------|
| #2072 | ...         | ...   |
| #2064 | sptr        | #2048 |
| #2063 | small.nm[3] | ?     |
| #2062 | small.nm[2] | \0    |
| #2061 | small.nm[1] | B     |
| #2060 | small.nm[0] | A     |
| #2056 | small.y     | 11    |
| #2048 | small.x     | 4.56  |
```

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; do not fclose(NULL)

int fclose(FILE *fh);
// close file associated with fh, writes pending data to file,
// free()'s memory associated with open file
// Do not fclose(NULL)

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

See examples of use in `struct_binary_io.c`

Tradeoffs between Binary and Textual Files

- Binary files usually smaller than text and can be directly read into memory but NOT easy on the eyes
- Text data more readable but more verbose, must be parsed and converted to binary numbers

Optional Exercise: 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;
```

Integer/Floating Division

Predict values for each variable

```
int q = 9 / 4;
int r = 9 % 4;
double x = 9 / 4;
double y = (double) 9 / 4;
double z = ((double)9) / 4;
double w = 9.0 / 4;
double t = 9 / 4.0;
int a=9, b=4;
double t = a / b;
```

Conditional (ternary) Operator

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

Answers: Integer vs Floating Division

Integer versus real division: **values** for each of these are...

```
int q = 9 / 4;                    // quotient 2
int r = 9 % 4;                    // remainder 1
double x = 9 / 4;                 // 2.0 (int quotient first)
double y = (double) 9 / 4;        // 2.25
double z = ((double)9) / 4;       // 2.25
double w = 9.0 / 4;               // 2.25
double t = 9 / 4.0;               // 2.25
int a=9, b=4;
double t = a / b;                 // 2.0 (int quotient)
```

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... and raptor attacks...

```c
// goto_demo.c: control flow with goto
// Low level assembly jumps are similar
#include <stdio.h>
int main(){
  int i=0;
 beginning:            // 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
// switch_demo.c: 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

Review Exercise: Memory Review

1. How do you allocate memory on the Stack? How do you de-allocate it?
2. How do you allocate memory dynamically (on the Heap)? How do you 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 two arithmetic expressions differ.

```c
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 you de-allocate it?

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

2. How do you allocate memory on the Heap? How do you 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?

   An array of 8 ints will be 32 bytes big (usually).

   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 two arithmetic expressions differ.

```c
int  x=9, y=20;     // x at #1024
int *p = &x;        // p hold VALUE #1024 (points at 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
```