

# CME 212: C programming II

## Lecture 4

January 20, 2012

# Memory management in C

Our programs normally don't know how much memory they are going to need before they start. Thus, we need to be able to dynamically allocate memory. C provides 2 mechanisms for this:

- variable length arrays
- memory management functions in `<stdlib.h>`

## Variable length arrays

```
1 int do_something(int n) {  
2     double a[n];  
3     ...  
4 }
```

- Introduced in C99. Controversial feature. Not allowed in standard C++.
- If  $n$  is too large, will cause a segmentation fault with `gcc`, cannot check for error during runtime
- What happens if  $n$  is negative?
- My recommendation: avoid this in real code

## Dynamic memory allocation

We are asking the operating system for memory. The OS can say “no”, but is usually generous.

```
1 int *array; int i;
2
3 // ask for memory
4 array = (int *)malloc(20*sizeof(int));
5
6 // check allocation
7 if ( !array ) {
8     // memory not allocated, do something smart
9 }
10
11 for ( i = 0; i < 20; i++ ) {
12     array[i] = i; // or *(array+i) = i
13 }
14
15 free(array); // release the memory when done
```

## malloc() & free()

- Remember to include `<stdlib.h>`

```
void *malloc(size_t size)
```

- allocates a block of `size` bytes
- returns pointer to first byte if successful
- returns `NULL` if unsuccessful
- data is not initialized

```
void free(void *ptr)
```

- `ptr` must have been allocated by `malloc` or associated functions

## Other memory functions

```
void *calloc(size_t count, size_t eltsize)
```

- note that function has 2 arguments
- sets all data to 0

```
void *realloc (void *ptr, size_t newsize)
```

- can be used to expand or contract the size of a block
- if `ptr == NULL`, behaves like `malloc`
- might need to copy data to another part of memory
- indicates failure by returning `NULL`, data associated with `ptr` is not affected

```
void *memalign(size_t boundary, size_t size);
```

- returned address is an even multiple of `boundary`
- useful if you want data in a particular part of memory

## Remember

The programmer is responsible for freeing all memory. This is a common source of errors. The term “memory leak” refers to memory that a program allocates, but does not free. If you do bad things with memory, the program might not crash until you attempt to free the memory.

A tool called Valgrind can help find these sorts of bugs.

## Relationship between pointers and arrays

- array indexing and pointer arithmetic can be used on both
- pointers are mutable, arrays are not
- `sizeof` on an array returns the size of the array in bytes
- `sizeof` on a pointer returns the size of an address

## Pointers vs. arrays example

```
1 int array[4];
2 array[0] = 0; array[1] = 1;
3 *(array+2) = 2; *(array+3) = 3;
4 int *ptr = malloc(4*sizeof(int));
5 ptr[0] = 0; ptr[1] = 1;
6 *(ptr+2) = 2; *(ptr+3) = 3;
7
8 ptr = array;      // YES
9 array = ptr;     // NO!
10 ptr = array[i]; // NO!
11 ptr++;          // YES
12 array++;        // NO!
13
14 size_t as = sizeof(array); // as = 16
15 size_t ps = sizeof(ptr);   // ps = 8
```

## Passing data to functions

There are two ways to pass data to a function:

- pass-by-value
  - Data is copied in to variables in the scope of the function. The variable in the function call is not changed.
- pass-by-reference
  - A reference to data in the calling scope is passed to function. Function may change data in the calling scope.
- C is pass-by-value, but arrays are cast to pointers
- use pointers for pass-by-reference

```
1 int f(int a, int *b) {  
2     *b = *b + 1;  
3     // *b is now changed in scope of the call to f  
4  
5     a = a + 1; // does not change first argument to f  
6     return a;  
7 }
```

# The const keyword

Sometimes we don't want a function to change the data.

```
1 int f(const int a, const int *b,  
2     int *const c, const int *const d) {  
3     // cannot change a  
4     // cannot change data b points to  
5     // cannot change address stored in c  
6     // cannot change data d points to  
7     // cannot change address stored in d  
8     return 0;  
9 }
```

# The C preprocessor

- C code is “pre-processed” before being sent to the compiler
- Carries out fairly simple, but tedious operations
  - inserting code (conditionally)
  - macros
  - setting constants
- Most common preprocessor directives are
  - `#include`: include a file
  - `#define`: define constant, or macro
- One very important application: the header guard...

## The header guard

code.h

```
1 #ifndef CODE_H_  
2 #define CODE_H_  
3  
4 // declare functions  
5  
6 #endif
```

code.c

```
1 #include "other_code.h"  
2 #include "code.h"  
3  
4 // define functions
```

If `other_code.h` includes `code.h`, the `code.h` header will not be included twice in `code.c`. Do this in all of your headers.

## Part of the C standard library

header	description
assert.h	Diagnostics
ctype.h	Character Class Tests
errno.h	Error Codes Reported by (Some) Library Functions
float.h	Implementation-defined Floating-Point Limits
limits.h	Implementation-defined Limits
locale.h	Locale-specific Information
math.h	Mathematical Functions
setjmp.h	Non-local Jumps
signal.h	Signals
stdarg.h	Variable Argument Lists
stddef.h	Definitions of General Use
stdio.h	Input and Output
stdlib.h	Utility functions
string.h	String functions
time.h	Time and Date functions

(list is extended by newer C standards.)

## Some important functions & constants

functions	library	note
atof(),atoi(),atol()	<stdlib.h>	ASCII to float, integer..
rand(),srand()	<stdlib.h>	Random numbers
abort(),exit()	<stdlib.h>	Aborts or exits
assert()	<assert.h>	Error checking
strcpy(),strlen(). . .	<string.h>	Manipulating strings
INT_MIN, INT_MAX	<limits.h>	Limits, bounds of ints
DBL_EPSILON. . .	<float.h>	Limits, bounds of FP
sin(),floor(). . .	<math.h>	Math functions, rounding
printf,fscanf. . .	<stdio.h>	Input and Output

## I/O: `<stdio.h>`

- The standard library uses the abstraction of a stream
- Can be connected to a file or the terminal
- There are 3 predefined streams
  - `stderr`
  - `stdin`
  - `stdout`
- These are connected to the terminal

## Using Streams

- `fputs(char *s, FILE *stream)` writes the string `s` to stream, which is of type `FILE *`
- `puts(char *s)` is predefined to print to `stdout`
- `int fgetc(FILE *stream)` reads a character from stream and returns it as an integer
- `int getchar(void)` reads from `stdin`

## Formatting Output

- Using only `fputc()`, etc. will force you to manually convert the binary representation of your type to ASCII
- The standard library can help you with this conversion using so-called format strings
- These strings contains a code for how you want your conversion to be performed

## Conversion Letters

d	signed integer
u	unsigned integer
f,g	floating point numbers
e,E	FP in exponential format
c	character
s	null terminated string
p	pointer

## Format specification

Apart from the conversion letter you can also control the output in many other ways. This is done by specifying formatting strings:

"%d"	prints an integer
"%12e"	prints FP in 12 characters
"%8.4f"	prints FP, width 8, 4 decimals
"%-p"	prints pointer, left justified
"%+le"	Always print the "+" sign

Keep a reference handy!

## printf()

- Prints the conversion of a variable using the format specification to `stdout`

```
1 | int a = 5;  
2 | printf("a = %d\n", a);
```

## Formatted Input

- The `scanf()` family of function scans a stream for data using the same formatting strings
- Remember to use pointers for the variables

```
1 | int a;  
2 | int num_inputs;  
3 | num_inputs = scanf("%d", &a);
```

## Working with files

- You can connect files to streams using `fopen()`, `fclose()`
- There are functions that control your position in the file
  - `fseek()`: goto a position
  - `ftell()`: get the position
  - `rewind()`: goto beginning of file
  - `fgetpos()`: more portable `ftell`
  - `fsetpos()`: more portable `fseek`

## Binary Files

- You can also dump the contents of the memory to a file, called a binary file
- Saves space
- Makes it hard for someone to read the file
- You must remember the representation of your data when reading
- Use: `fread()`, `fwrite()`

## Binary files, example

Write binary data:

```
1 size_t n = 100;
2 double a[n];
3 for (size_t i = 0; i != n; ++i)
4     a[i] = (double) i;
5 FILE *fp = fopen("data.bin", "wb");
6 size_t w1 = fwrite(&n, sizeof(size_t), 1, fp);
7 size_t w2 = fwrite(a, sizeof(double), n, fp);
8 fclose(fp);
```

Read binary data:

```
1 FILE *fp = fopen("data.bin", "rb");
2 size_t n;
3 size_t w1 = fread(&n, sizeof(size_t), 1, fp);
4 double a[n];
5 size_t w2 = fread(a, sizeof(double), n, fp);
6 fclose(fp);
```

## Using `errno`

- If a call returned that an error occurred, the global variable `errno` is set to a negative integer to indicate what went wrong
- The `errno` variable is set to a predefined error code, see `man errno`
- With every `errno` code there is a corresponding error message which can be retrieved using the functions `perror()` and `strerror()`

## errno, Example

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main(int argc, char *argv[]) {
4     FILE *f;
5     if ((f = fopen(*(argv+1), "r")) == NULL)
6         perror("fopen_failed"); exit(1);
7
8     fclose(f);
9     return 0;
10 }
```

```
$ ./errno bogus-file
```

```
fopen failed: No such file or directory
```

```
$ ./errno
```

```
fopen failed: Bad address
```

## Assertion

The standard library includes a macro that can catch errors

```
void assert (expression);
```

If the expression is false, assert prints out the line and file of the assert statement that triggered

```
`myprogram.c':234: function: Assertion  
'expression' failed.
```

If `NDEBUG` is defined before `assert.h` is included, all assertions will map to an empty macro

## Assert, Example

```
1 int main(void) {  
2     int *a;  
3     a = (int *)malloc(100*sizeof(int));  
4     assert( a != NULL );  
5 }
```

## primitive types

keyword	note
void	
char	
int	
float	
double	
signed	
unsigned	
short	
long	
_Bool	C99
_Complex	C99
_Imaginary	C99

## defined types

keyword	note
typedef	define a type name
struct	define aggregate data type
enum	enumeration
union	multiple type for same data

for types

Table: type qualifiers

keyword	note
const	hold constant
volatile	do not optimize

Table: storage classes

keyword	note
auto	
static	also function qualifier
extern	
register	hint to compiler

## flow control

keyword
if
else
do
while
for
switch
case
default
break
continue
goto
return

other

keyword	note
restrict	for optimization
inline	inline a function
sizeof	get size of type