C Programming

1) Embedded System Programming [2.1.2]

- language prevalence

	2008	2013	2018
assembly languages	45%	20%	~0%
C	90%	60%	70%
C++	40%	~~50%~~	25%
Java	10%	30%	~0%
C#	5%	25%	~0%
	* text		Barr Group Survey

- why C?

properties	benefits
no garbage collection	deterministic execution time
compiled (not interpretted)	deterministic execution time, fast, no vm overhead
pointers	needed for memory-mapped I/O
small executable	less memory, fast

- Keil uVision 5 uses C -c90 standard is default (--c99)

2) Structs [2.2.1]

- only aggregate data types in C are structs and arrays (no classes)
- structs are similar to classes except:
  1. no methods (no constructors)
  2. all data members are public

```
#include <stdio.h>          standard I/O header

struct pair {               ← tag
    int first;
    int second;
};

int main(void) {            ← C idiom uses void

    struct pair p1;

    p1.first = 1;
    p2.second = 2;

    printf("%d, %d\n", p1.first, p1.second);

}
```

format string

conversion specifiers (d = decimal, x = hexadecimal, c char, s string, f, e, g float/double, p pointer)

⇒ 1, 2

"variadic function"

e.g. same example but using typedef

```
#include <stdio.h>

typedef struct {
    int first, second;
} pair_t;

int main(void) {

    pair_t p2 = { 3, 4 };

    printf("%d, %d\n", p2.first, p2.second);

    return 0;
}
```

textbook convention

initializer

= {.first=3, .second=4};

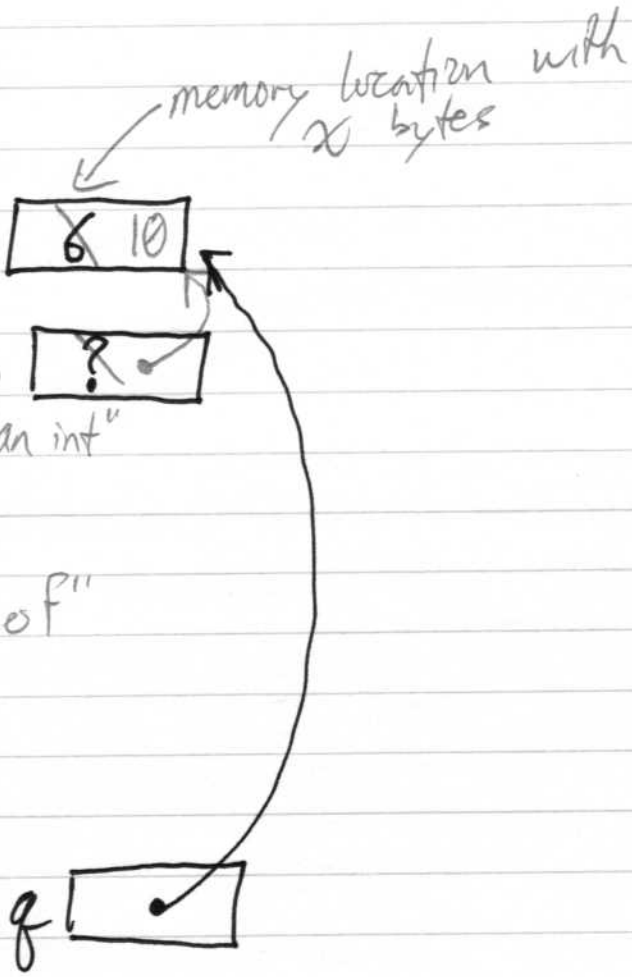
⇒ 3, 4

return 0; is optional for main

3) Pointers

```
#include <stdio.h>
int main(void) {
    int i = 6;
    int *p;
    p = &i;
    *p = 10;
    int *q = p;
    printf("%d\n", *q);
}
```

⇒ 10



"pointer to" "an int"

"address of"

"dereference"

- syntax note

```
int*  p, q;
```

pointer to int

int

associates with the variable

what was probably wanted

```
int *p, *q;
```

- NULL pointer

0 or (void *) 0

type cast

generic pointer type
- it automatically converts to any pointer type

```
int *p = 0;
int *q = NULL;
```

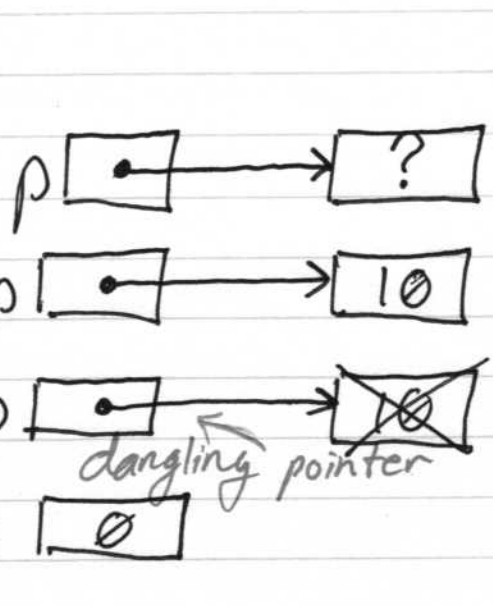
macro (stdlib.h)

- dynamic memory allocation [2.2.4]

```
#include <stdlib.h>

int main(void) {

    int *p;
    p = (int *) malloc(sizeof(int));
    *p = 10;
    free(p);
    p = NULL;
}
```



good practice

- malloc = memory allocate: allocates a block of n bytes and returns a pointer of type void * to it (returns NULL if unsuccessful)

- sizeof - operator: returns the number of bytes of storage for the parameter (a type or a variable)

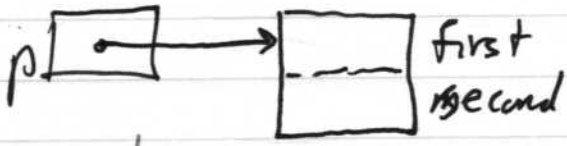
- free - reclaims the storage indicated by the pointer

- pointers to structs

e.g.
```
#include <stdio.h>
#include <stdlib.h>

typedef struct {
    int first, second;
} pair_t;

int main(void) {
    pair_t *p = malloc(sizeof(pair_t));
```




```
    (*p).first = 1;      okay
    p->second = 2;       better
    ...
    free(p);
}
```

e.g.
```
int *x = malloc(10 * sizeof(int));
```
allocate array of 10 ints

```
x[0] = 0;
x[1] = 1;
...
free(x);
```

```
int y[5] = {};
```
- not standard
- supported by gcc

4) Integer Type Sizes [2.2.4]

- integer type sizes are compiler dependent
- rules:

char >= 8 bits

short (int) >= 16 bits

int >= 16 bits

long (int) >= 32 bits

long long (int) >= 64 bits

char <= short <= int <= long <= long long

- "int" could be 16 bits, 32 bits, 64 bits
- sized types are better (safer) for real-time

in <stdint.h>

-signed types: int8_t, int16_t, int32_t, int64_t

-unsigned types: uint8_t, uint16_t, uint32_t, uint64_t

bits

5) Pass By Value only [2.2.5]

- arguments and return values are passed by value only (i.e. copied) - no pass by reference

C++

```
void incr(int &n){
    n++;
}
```

C

```
void incr(int *n){
    (*n)++;
}
```

- passing large structs to functions is faster with a pointer

6) No Mixed Declarations and Code

- C90 doesn't allow - C99 does

in µVision 5 add --c99

⇒ all variable declarations must occur at the top of a block

```
{
    int j = 10;
    for(int i=0; i<5; i++)
        ...
}
```

only okay in C99

```
{
    int i, j = 10;
    for(i = 0; i<5; i++)
        ...
}
```

okay in C90/C99

7) Boolean Variables

```
#include <stdbool.h>

bool x = true || false;
```

8) Pre-processor [2.2.13]

- transforms the source code before compiling
  - includes header files
  - expands macros
  - does conditional compilation

e.g. from Retarget.c (from lab)

```
...
#include <rt_misc.h>          // system headers

...
#ifdef __RTGT_UART            // conditional compilation
  #include "uart.h"           // local headers
  #define PORT_NUM 0          // macro definition
  #define BAUD_RATE 9600
#endif
```

#else, #elif, #undef } other compiler directives

- including a header file multiple times could lead to multiple definitions of variables or types or functions
  - to avoid this, header files are bracketed by #ifndef ... #endif (include guards)

eg. from uart.h

```
#ifndef __UART_H
#define __UART_H

... file contents

#endif
```

- "#pragma once" is a possibly better but non-standard alternative

- brackets around macros are important

eg. `#define SQR(x) x * x`

`int y = SQR(1+2);` y [5]

⇓

`1+2 * 1+2`

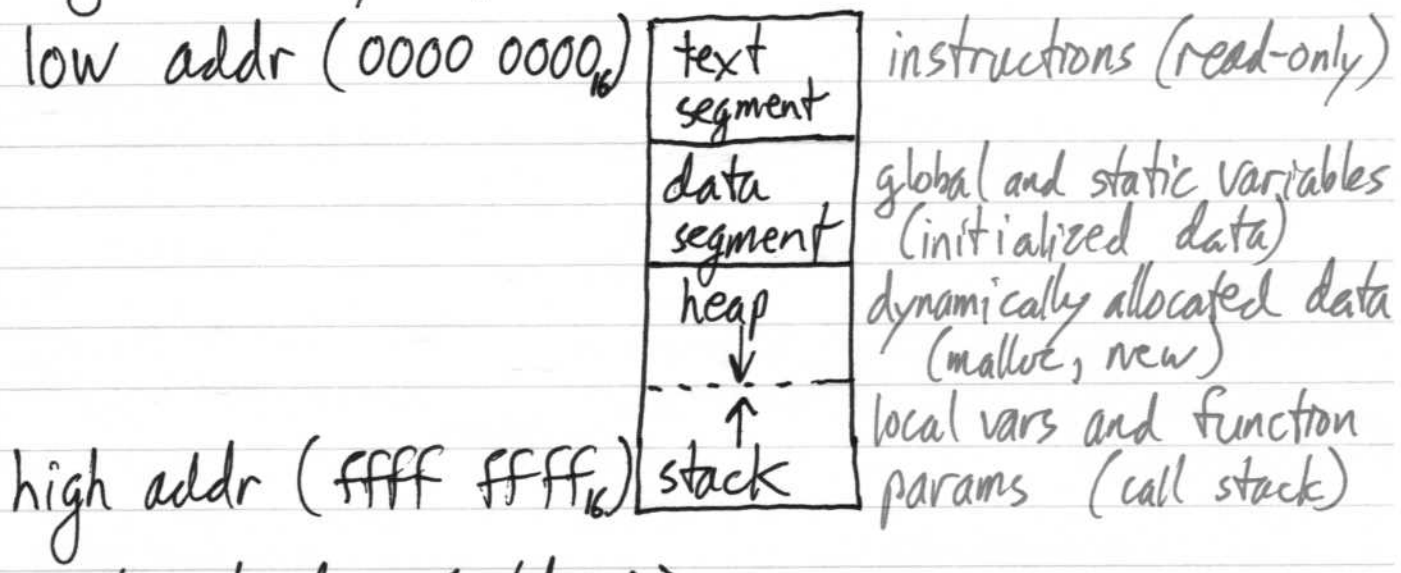
`#define SQR(x) ((x) * (x))`

`int z = SQR(1+2);` z [9]

⇓

`((1+2) * (1+2))`

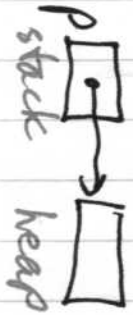
9) Program Memory Layout



e.g.
```
#include <stdio.h>
#include <stdlib.h>

char s[] = "abc";          // program literal (constant)

int main(void) {
    int m = 4;
    int *p = malloc(sizeof(int));
    printf("%14p\n%14p\n%14p\n%14p\n",   // print pointer 14 chars wide
        s,    &m,    &p,    p);
    free(p);
}
```

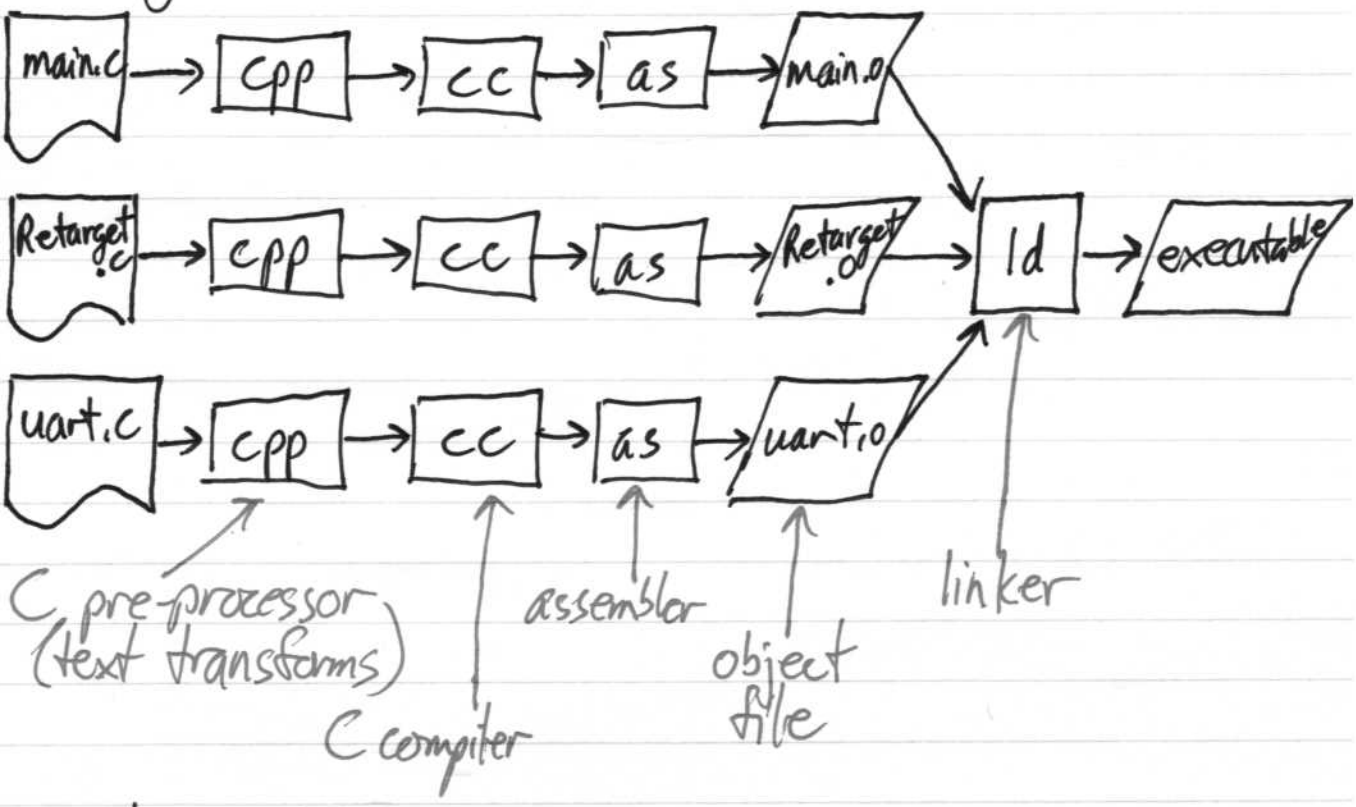


global — local — heap

⇒

	0x601050	s
0x7fff	dcd8afa4	&m
0x7fff	dcd8afa8	&p
0	x252e010	p

- compiling



- execution
  - steps:
    1) OS allocates memory (typically virtual memory) for the process (for code and data)
    2) loader copies the program code into the text segment, initializes the data segment
    3) execution starts at the entry point
       Linux: exec → _start → main()
       _start: put command-line arguments on stack

Keil RTX: Reset_Handler → SystemInit
↳ __main → main()
-no command-line arguments