common feeling
after (my) exams:

1. more people handed in
before end of class than last year
2. things will turn out OK;
you can still get the grade you want

uC Programming w/C I

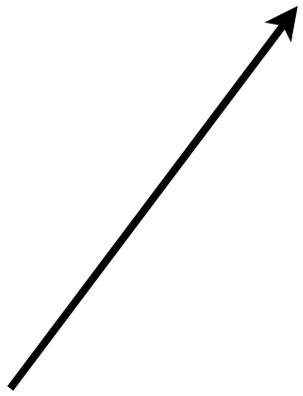
ECE 3710

I'm addicted to placebos... I'd quit but it wouldn't matter.

- Steven Wright

why C?

YOU: ASM is lame



ME: please elaborate

why C?

1. it's natural: we think in it already
2. maintenance
3. portability
4. functions (please, be lazy)

C doesn't lead to understanding...

...it is not your saviour, you do not rule over uC

why not C?

1. debugging hardware
2. software/hardware interface
3. speed
4. you get lazy

/etc/master.passwd:

```
rgerdes:8ef121cb8b1beeaf88801b210367b8ee:1001:1001:Ryan Gerdes:/home/rgerdes:/bin/tcsh
```

md5 of my password
(fuzzybunny)

hacker:

1. get dictionary
2. create md5 of entries
3. compare
4. profit

Implementation	Throughput (MB/s)
md5-amd64 64-bit (AMD64 asm language)	356
OpenSSL MD5 32-bit (i386 asm language)	312
OpenSSL MD5 64-bit (C language)	204

MD5 throughput on an Opteron 244 (1.8 GHz) processor

75% faster than C

14% faster than C

note:

expect you to be familiar with basic syntax, arrays, pointers, etc.

you need to have your own C resources

basic data types

Type	Size in bits	Natural alignment in bytes
char	8	1 (byte-aligned)
short	16	2 (halfword-aligned)
int	32	4 (word-aligned)
long	32	4 (word-aligned)
long long	64	8 (doubleword-aligned)
float	32	4 (word-aligned)
double	64	8 (doubleword-aligned)
long double	64	8 (doubleword-aligned)
All pointers	32	4 (word-aligned)
bool (C++ only)	8	1 (byte-aligned)
_Bool (C only[a])	8	1 (byte-aligned)
wchar_t (C++ only)	16	2 (halfword-aligned)

support via libraries

(if no FPU onboard or compiler doesn't use FPU)

don't forget: uC has limited memory for code/data

(choose smallest data type that gets the job done...)

does not refer to type but size

```
unsigned char z;
```

use all bits for numbers

(don't reserve for sign)

add_uchar.c

your first C program

```
int main(void)
{
   unsigned char z;
   for(z=0;z<=255;z++);
}
```

i want me a number

note: no native `printf`

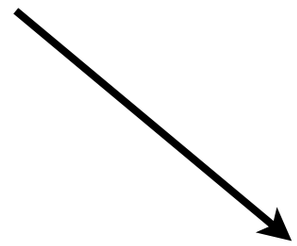
(need to use a debugger)

can set one up to output on UART

your ASM vs. the compiler's

```
int main(void)
{
  unsigned char z;

  for(z=0;z<=255;z++);
}
```



```
mov R0,#0
loop
  add R0,#1
  cmp R0,#255
  bls loop
```

this is actually sloppy code, on our part (why?)

compiler ASM

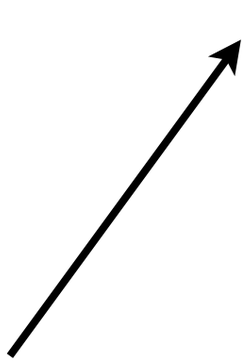
```
    25:             for(z=0;z<=255;z++);
    26:
0x000001C8 2100         MOVS    r1,#0x00
0x000001CA E001         B       0x000001D0
0x000001CC 1C48         ADDS    r0,r1,#1
0x000001CE B2C1         UXTB    r1,r0
0x000001D0 29FF         CMP     r1,#0xFF
0x000001D2 DDFB         BLE     0x000001CC
```

UXTB:

```
    R1[31:0] = [0...0 R0[7:0]]
```

force number to be 8-bits...
this is not good

```
    25:             for(z=0;z<=255;z++);
    26:
0x000001C8 2100         MOVS    r1,#0x00
0x000001CA E001         B       0x000001D0
0x000001CC 1C48         ADDS    r0,r1,#1
0x000001CE B2C1         UXTB    r1,r0
0x000001D0 29FF         CMP     r1,#0xFF
0x000001D2 DDFB         BLE     0x000001CC
```



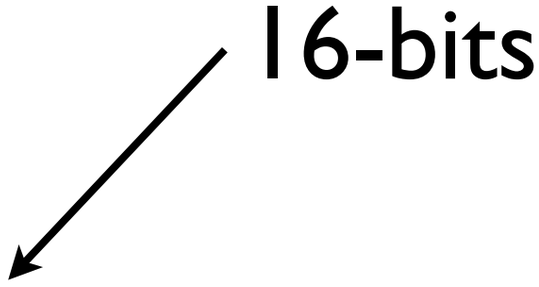
can never be > 255

because max(8-bit number) = 255

your second C program

```
int main(void)
{
   unsigned short z;

   for(z=0;z<=255;z++);
}
```



Q: when does it make sense to tailor type to data?

e.g. know data will only need byte; use `int` or `char`?

e.g. know data will only need byte;
use `int` or `char`?

1. registers 32-bits
(we operate on 32-bit data natively; no truncating of data)
2. more space in memory

1. compiler will enforce
(extra instruction)
2. less space used in memory

A: a tradeoff

Isn't it a bit unnerving that doctors call what they do "practice"?

- George Carlin

what do we need vars for?

1. computation
2. configuration

writing to the var sets/clears bits at a certain place in memory

e.g.

Run-Mode Clock Configuration (RCC)

Base 0x400F.E000
Offset 0x060
Type R/W, reset 0x078E.3AD1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
reserved				ACG	SYSDIV				USESYSDIV	reserved	USEPWMDIV	PWMDIV			reserved
Type	RO	RO	RO	RO	R/W	R/W	R/W	R/W	R/W	R/W	RO	R/W	R/W	R/W	R/W	RO
Reset	0	0	0	0	0	1	1	1	1	0	0	0	1	1	1	0

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
reserved		PWRDN	reserved	BYPASS	reserved	XTAL				OSCSRC		reserved		IOSCDIS	MOSCDIS
Type	RO	RO	R/W	RO	R/W	RO	R/W	R/W	R/W	R/W	R/W	R/W	RO	RO	R/W	R/W
Reset	0	0	1	1	1	0	1	0	1	1	0	1	0	0	0	1

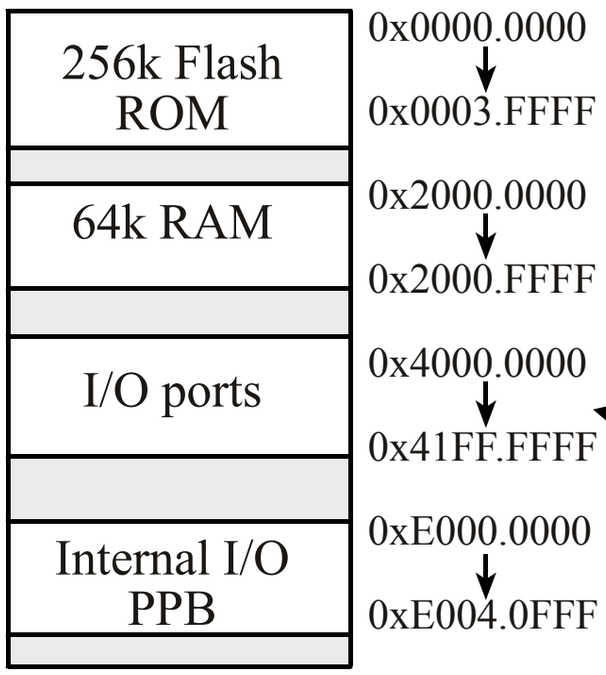
1. registers
2. stack
3. non-specific RO/RW addr.
(i.e. code or data memory)

where are vars located?

1. computation
2. configuration

specific addr in RW memory goes here (0x400FE060)

var for this



Run-Mode Clock Configuration (RCC)

Base 0x400F.E000
Offset 0x060
Type R/W, reset 0x078E.3AD1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
reserved				ACG	SYSDIV				USESYSDIV	reserved	USEPWMDIV	PWMDIV			reserved
Type	RO	RO	RO	RO	R/W	R/W	R/W	R/W	R/W	R/W	RO	R/W	R/W	R/W	R/W	RO

local vars

‘c’ will be allocated to a register
(maybe stack later)

```
unsigned char c;
unsigned char *d = (unsigned char *) 0x20000043;
```

how to set initial addr of what ptr points to:

create byte ptr that points to addr given by num*,
d is then assigned addr of what ptr points to
(*d =0x123 writes 0x123 to 0x20000043)

*or: cast num as a byte ptr that points to addr given by num,

global vars/compiler definitions

put 'a' at given addr

```
unsigned char a __attribute__((at(0x20000040)));
#define b (*((unsigned char *) 0x20000041))
```

assignment to b is like
dereferncing a pointer that points to addr

configuring the uC in C

create global vars situated at registers for GPIO Port D:

```
volatile int PD_DATA_R __attribute__((at(0x400073FC)));
volatile int PD_DIR_R __attribute__((at(0x40007400)));
volatile int PD_AF_R __attribute__((at(0x40007420)));
volatile int PD_DEN_R __attribute__((at(0x4000751C)));
volatile int RCGC2_R __attribute__((at(0x400FE108)));
//value for RCGC2 to enable clock for port D
#define RCGC2_PD 0x00000008
```

`volatile`: use when uC might change contents of addr on you; e.g. when doing memory mapped I/O

(otherwise compiler might 'optimise' away functionality [like polling])

ex: write 0--15 continuously on PD

```
int main(void)
{
   unsigned char z;

  // initialise port: RMW-cycle so much better in C...
  //activate port D: RCGC2 = RCGC2 | RCGC2_PD
  RCGC2_R |= RCGC2_PD;
  //make PD3-0 output
   PD_DIR_R |= 0x0F;
   //disable alt. func.
   PD_AF_R &= 0x00;
  //enable digital I/O on PD3-0
   PD_DEN_R |= 0x0F;
        ←──────────────────────────────── while(1)
  return 1;                                 {
}                                             for(z=0;z<16;z++)
                                                PD_DATA_R = z;
                                            }
```

Q: C program to copy input port data to output port

(read from one port and write to another)

configuration using offsets

how to set initial addr of
what ptr points to:

```
unsigned char *d = (unsigned char *) 0x20000000;
```

use like C-array:

```
d[0x12]=0xAB;
```

points to
0x20000000+0x12
=0x20000012

memory(0x20000012)
=0xAB

configuration using offsets

tedious to write word

slight downside:

1. byte addressable (char for ptr)
2. addressing words (int for ptr)

indexing is off

example: configuration using offsets

final value of word at byte offset
0x60 should be 0x078E3B82:

```
//1. unsigned char *SYSCTL = (unsigned char *) 0x400FE000;
SYSCTL[0x60] = 0x82;
SYSCTL[0x61] = 0x3B;
SYSCTL[0x62] = 0x8E;
SYSCTL[0x63] = 0x07;
//2. unsigned int *SYSCTL = (unsigned int *) 0x400FE000;
SYSCTL[0x60/4] = 0x078E3B82;
```