Exam 2 Review

★ UNICODE Q in separate video ★ → at 9:10 for 0x31, I say 2 & 4 spaces, I meant to say 1 & 2 spaces in memory

① Assembly

- Arrays – how do we create arrays? ✓
  how do we traverse them? }
  how do we load them? }
- Registers & STACK
  - AAPCS
- Useful instructions
  - see your addendum ✓
  - LDR R1, = 40000 ✓
  - MOV can only do imm16
  - BL
  - BX LR
- Passing values between C & Assembly

```
func (int a, int b, int c)
```

↑

```
func2    a → R3

         MOV  R0, R3
         MOV  R1, b
         MOV  R2, c

         BL func
```

② C

Pointers
- creating pointers
- dereferencing pointers
- passing in as an input parameter

Arrays
- creating arrays
- dereferencing array
- passing in as an input parameter

How are these two related to each other?

Structs
- creating a struct
- accessing values in a struct from object
- accessing values in struct from pointers

Loops
- for
- while

If/else

return

Pitfalls when taking the exam

★ → be aware of the size & sign of your value / use correct loads
→ Don't confuse pointers & dereferences when passing values
→ Know what is in your registers, don't accidentally overwrite

How to Load and types of loads & stores

```
LDR R0, =var  →
LDR R1, [R0]
```

```
STR  → 32
STRB → 8
STRH → 16
```

```
LDR  → 32
LDRB → 8
LDRH → 16
LDRSB → 8
LDRSH → 16
LDRS → 32
```

unsigned & signed

Creating arrays in Assembly

In Assembly

- DCB → 8 bits
- DCW → 16 bits
- DCD → 32 bits

In C

```
uint8_t  arr[]
16       arr[]
32       arr[]
```

```
AREA CODE
THUMB

Array   DCD   1,2,3,4
Array2D DCB   6,4,9
```

put in ROM

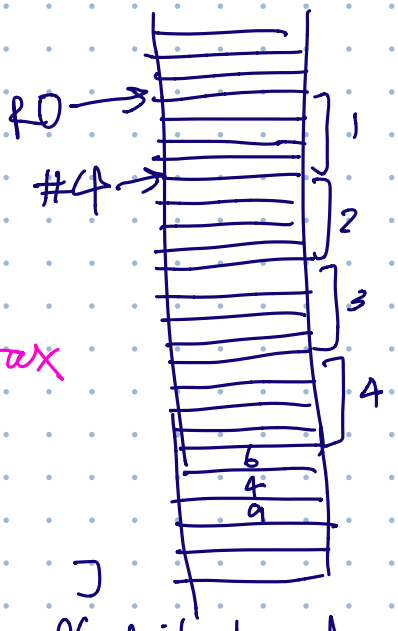
1x4 (1)

offset

syntax

offset

1x1



Accessing a value in the array?

Get the address of the array and offset it based on the amount of space in memory each value before and use the correct *load based on data type*

LDR R1, [R0, #offset]

the index you want

- Creating an array in RAM → when you want to write to an array

100 element array
16 bits → how many bytes?

```
AREA DATA
Array SPACE 200
```
↙ RAM

↳ 16 bit array with 100 elements

Load Address of Array
Store to address using correct size of store
Increment pointer based on the number of bytes each value occupies — STRH

- loops in Assembly

```
loop
    SUBS
    BNE , 0
```
↳ counting down

```
loop
    CMP  0
    BNE  0
```
→ waiting for sentinel

0 → 0 is the sentinel (null terminator) you are looking for

```
loop  CMP
      BEQ

      B loop
```
depends on your edge cases or how you implement

- Creating a pointer:

↑ make the distinction, pointers are created as *p, but *p used after is a dereferenced value and p is the pointer

pointer data type should match obj it's pointing to

```
uint16_t var = 16;
uint16_t *p = &var
```

→ get address

↳ var++

- Dereferencing pointers:

*p = 16

p = address

*p++ p++ have same effect both will increment variable

After initializing a pointer:

*p = value at address, p = address

= 16 = &var

Incrementing the pointer:

If you have a pointer to an array and want to get to the next element two methods:

① p++;

*p; → now gives value of next element

same thing

*(p+1) → increment point temporarily then dereferences

gives value of next element

arr[i+1]

The parenthesis are important, see struct section

- Passing pointer as an input parameter

① Function being called expects pointer input

Correct

```
Func2(int *p){
    :
}
Func1(int *p){
    Func2(p);
}
```

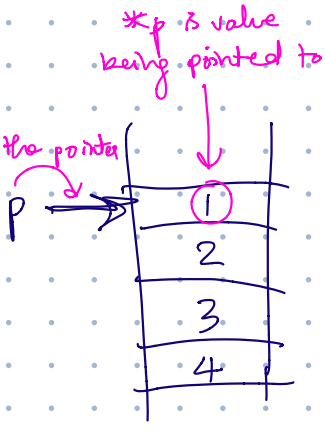
correct p is the pointer/address

Incorrect

```
Func2(int *p){
    :
}
Func1(int *p){
    Func2(*p);
}
```

↳ incorrect, passes in dereferenced integer value pointed to by p

*p is value being pointed to



② Function being called expects integer input

Correct

```
func2 ( int p){
    :
}
Func1 (int *p){
   Func2(*p);
}
```

↳ correct since *p is a dereferenced integer value

Incorrect

```
func2 ( int p){
    :
}
Func1 (int *p){
   Func2(p);
}
```

↳ incorrect, since p is a pointer

Pointer type questions may end with a sentinel or pass in a size

Creating Arrays

Array [5] = {1, 2, 3, 4, 5}

0 1 2 3 4

Make the distinction Arr[5] when creating array but when used after Arr[5] is a dereference not the whole array

size must match, cannot dynamically change size of array

Dereferencing Arrays

Array [1] has the value 2. This is also a type of dereferencing.

Incrementing Array pointer

Array[i]; ⇒ i++ → increment array index to access next element

0 Passing array as an input parameter

① function being called expects array input

Correct

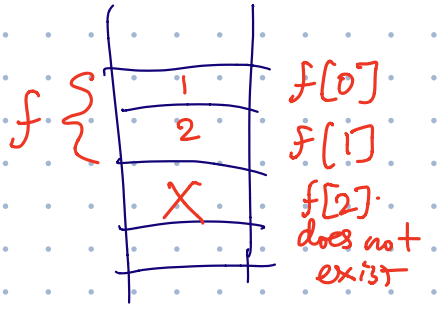
```
Func2 ( int f[2]) {
    :
}
Func1 (int f[2]) {
    Func2(f);
}
```

correct f is the entire array

Incorrect

```
Func2 ( int f[2]) {
    :
}
Func1 (int f[2]) {
    Func2(f[2])
}
```

↳ incorrect, passes in dereferenced f[2], not only is this incorrect because an array of size 2 can have only f[0] & f[1], f[2] is an attempt to dereference and is not the whole array



② Function being called expects integer input

Correct

```
func2 ( int f ) {
    :
}
Func1 (int f[]) {
    Func2(f[0]);
}
```

↳ correct since f[0] is a dereferenced integer value

Incorrect

```
func2 ( int f ) {
    :
}
Func1 (int f[]) {
    Func2(f);
}
```

↳ incorrect, since f is an array not an integer value (data types don't match)

```
func( int f[] )
func (int f)
func ( int * f)
```

- Creating a struct

Required syntax is in pink!

```
struct Person {
    int age;
    short weight;
    short height;
} typedef person_t;
```

- Creating a struct object

```
person_t Bob;
Bob.age = 50;
Bob.weight = 73 // kg
Bob.height = 180 // cm
```

- Accessing struct attributes when given struct objects

```
Func (person_t p1, person_t p2)
    (p1.age > p2.age){
        return p1.weight;
}
```

- Accessing struct attributes when given pointer / array of structs.

```
person_t f[3] = {Ant, Bob, Cat}
person_t *p = f;
```

p++ lose original address

f[i] i++ retain og address

otherwise *p+1.age ⇒ *(p+1.age)

* With pointers must use parentheses

```
Func (person_t *p){
    int Bob_age = *(p+1).age
    Bob_age = p[1].age
    p++;
    Bob_age = *p.age
    Bob_age = p[0].age
}
```

same result

(*p).age

*(f+2).age is the same as f[2].age, both can be used regardless of whether you are given an array or pointer

*(p+2).age

p[2].age also the same

```
Func (person_t f[3]){
    int i = 0;
    int Cat_height = f[i+2].height   // or f[2]
    Cat_height = *(f+2).height
    i = i+2;                         // still f[2]
    Cat_height = f[i].height;
    Cat_height = *f.height.
}
```

* Pointers and Arrays can be dereferenced both ways

- Loops

```
int i;
for(i=0; i<20; i++){
}
```

semi-colons

best to use when you know how many times you need to loop

```
while(a!=0){
}
```

boolean

don't use & | = accidentally

&& || ==

best to use when your not sure how many times you will need to loop

- if-else statements

```
if ( a>10){
}
else if ( a==10){
}
else{
}
```

- return

```
if (i==0){
    return 3;
}
if (i==1){
    return 2;
}
```

only one return will execute, after return is call function is exited and remaining code will not be run.

Calculus Spring 2013 useful Easy

Exam2_Database Average → Typecasting Easy

EKGProcess LDR R4, =4095 Medium - Easy

look at max value move can take

or no. you can cmp

Look at your edge cases

```
uint16_t var[3] = [16, 2, 3]
uint16_t *p = &var;
            &var[0];
```

```
print(var[0])      16
print(*p)          16
print(p)           x400 ... 2   address
```

var[0]

```
p++;   → address   x400 ... 4
```

var[1]

```
print(var[0])      16
print(var[1])      2
print(*p)          2
```

```
*p++;  → value at address
print(var[1])      3
print(*p)          3
print(var[0])      16
```

address

func(int *p){ 16

}

func2 (int *p){

func (p) p or *p

}

*p = &f

f[3] = {Ant, Bob, Cat}

func (person_t *p){

int Bob_age = *(p+1)

}

var = *p (16)

val = arr[0] (16)