Computer Systems Fundamentals

add_memory.c

```
#include <stdio.h>

int x, y, z;
int main(void) {
    x = 17;
    y = 25;
    z = x + y;
    printf("%d", z);
    printf("\n");
    return 0;
}
```

add_memory.s

add 17 and 25 use variables stored in memory and print result

```
main:                   #  x, y, z in $t0, $t1, $t2,
    li   $t0, 17        # x = 17;
    sw   $t0, x

    li   $t0, 25        # y = 25;
    sw   $t0, y

    lw   $t0, x
    lw   $t1, y
    add  $t2, $t1, $t0 # z = x + y
    sw   $t2, z

    lw   $a0, z        # printf("%d", a0);
    li   $v0, 1
    syscall

    li   $a0, '\n'      # printf("%c", '\n');
    li   $v0, 11
    syscall

    li   $v0, 0        # return 0
    jr   $ra

.data
x:  .word 0
y:  .word 0
z:  .word 0
```

array_element_address.c

```
#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>


int main(void) {
    double array[10];

    for (int i = 0; i < 10; i++) {
        printf("&array[%d]=%p\n", i, &array[i]);
    }

    printf("\nexample computation for address of array element \\n\n");

    uint64_t a = (uint64_t)&array[0];
    printf("&array[0] + 7 * sizeof (double) = 0x%lx\n",     a + 7 * sizeof (double));
    printf("&array[0] + 7 * %lx               = 0x%lx\n", sizeof (double), a + 7 * sizeof (double));
    printf("0x%lx + 7 * %lx             = 0x%lx\n", a, sizeof (double), a + 7 * sizeof (double));
    printf("&array[7]                       = %p\n", &array[7]);
}
```

emulating_array_indexing.c

non-portable code illustrating array indexing this relies on pointers being implemented by memory addresses which most compiled C implementations do

```
#include <stdio.h>
#include <stdint.h>

uint32_t array[10] = { 10, 11, 12, 13, 14, 15, 16, 17, 18, 19 };

int main(void) {
    // use a typecast to assign array address to integer variable i
    uint64_t i = (uint64_t)&array;

    i += 7 * sizeof array[0]; // add 28 to i

    // use a typecast to assign  i to a pointer vaiable
    uint32_t *y = (uint32_t *)i;

    printf("*y = %d\n", *y); // prints 17

    // compare to pointer arithmetic where adding 1
    // moves to the next array element
    uint32_t *z = array;
    z += 7;
    printf("*z = %d\n", *z); // prints 17
}
```

store_array_element.c

simple example of accessing an array element

```
#include <stdio.h>

int x[10];

int main(void) {
    x[3] = 17;
}
```

store_array_element.s

```
main:
    li   $t0, 3
    mul  $t0, $t0, 4
    la   $t1, x
    add  $t2, $t1, $t0
    li   $t3, 17
    sw   $t3, ($t2)
    # ...
.data
x:  .space 40
```

print5.c

print 5 numbers

```
#include <stdio.h>

int numbers[5] = { 3, 9, 27, 81, 243};

int main(void) {
    int i = 0;
    while (i < 5) {
        printf("%d\n", numbers[i]);
        i++;
    }
    return 0;
}
```

print5.simple.c

print 5 numbers

```
#include <stdio.h>

int numbers[5] = { 3, 9, 27, 81, 243};

int main(void) {
    int i = 0;
loop:
    if (i >= 5) goto end;
        int j = numbers[i];
        printf("%d", j);
        printf("%c", '\n');
        i++;
    goto loop;
end:
    return 0;
}
```

print5.s

print 5 numbers i in $s0 j in $s1

```
main:
    li   $s0, 0          # int i = 0;
loop:
    bge  $s0, 5, end     # if (i >= 5) goto end;
    la   $t0, numbers    #    int j = numbers[i];
    mul  $t1, $s0, 4
    add  $t2, $t1, $t0
    lw   $s1, ($t2)
    move $a0, $s1        # printf("%d", j);
    li   $v0, 1
    syscall
    li   $a0, '\n'       #   printf("%c", '\n');
    li   $v0, 11
    syscall

    add  $s0, $s0, 1     #   i++
    b loop               # goto loop
end:

    li   $v0, 0          # return 0
    jr   $ra

.data

numbers:                 # int numbers[10] = { 3, 9, 27, 81, 243};
     .word 3, 9, 27, 81, 243
```

pointer5.c

print 5 numbers

```
#include <stdio.h>

int numbers[5] = { 3, 9, 27, 81, 243};

int main(void) {
    int *p = &numbers[0];
    int *q = &numbers[4];
    while (p <= q) {
        printf("%d\n", *p);
        p++;
    }
    return 0;
}
```

pointer5.simple.c

print 5 numbers

```
#include <stdio.h>

int numbers[5] = { 3, 9, 27, 81, 243};

int main(void) {
    int *p = &numbers[0];
    int *q = &numbers[4];
loop:
    if (p > q) goto end;
        int j = *p;
        printf("%d", j);
        printf("%c", '\n');
        p++;
    goto loop;
end:
    return 0;
}
```

pointer5.s

print 5 numbers p in $s0 q in $s1 j in $s2

```
main:
    la   $s0, numbers     # int *p = &numbers[0];
    la   $t0, numbers     # int *q = &numbers[4];
    add  $s1, $t0, 16     #
loop:
    bgt  $s0, $s1, end    # if (p > q) goto end;
    lw   $s2, ($s0)       # int j = *p;
    move $a0, $s2         # printf("%d", j);
    li   $v0, 1
    syscall
    li   $a0, '\n'        #   printf("%c", '\n');
    li   $v0, 11
    syscall

    add  $s0, $s0, 4      #   p++
    b loop                # goto loop
end:

    li   $v0, 0           # return 0
    jr   $ra

.data

numbers:                  # int numbers[10] = { 3, 9, 27, 81, 243};
     .word 3, 9, 27, 81, 243
```

pointer5.faster.s

print 5 numbers - this is closer to the code a compiler might produce p in $s0 q in $s1

```
main:
    la   $s0, numbers     # int *p = &numbers[0];
    add  $s1, $s0, 16     # int *q = &numbers[4];
loop:
    lw   $a0, ($s0)       # printf("%d", *p);
    li   $v0, 1
    syscall
    li   $a0, '\n'        #   printf("%c", '\n');
    li   $v0, 11
    syscall
    add  $s0, $s0, 4      #   p++
    ble  $s0, $s1, loop   # if (p <= q) goto loop;

    li   $v0, 0           # return 0
    jr   $ra

.data

numbers:                  # int numbers[10] = { 3, 9, 27, 81, 243};
     .word 3, 9, 27, 81, 243
```

read10.c

read 10 numbers into an array then print the 10 numbers

```c
#include <stdio.h>

int numbers[10] = { 0 };

int main(void) {
    int i;

    i = 0;
    while (i < 10) {
        printf("Enter a number: ");
        scanf("%d", &numbers[i]);
        i++;
    }
    i = 0;
    while (i < 10) {
        printf("%d\n", numbers[i]);
        i++;
    }
    return 0;
}
```

read10.s

read 10 numbers into an array then print the 10 numbers

i in register $s0 registers $t1, $t2 & $t3 used to hold temporary results

```
main:

    li $s0, 0             # i = 0
loop0:
    bge $s0, 10, end0     # while (i < 10) {

    la $a0, string0       #   printf("Enter a number: ");
    li $v0, 4
    syscall

    li $v0, 5             #   scanf("%d", &numbers[i]);
    syscall               #

    mul $t1, $s0, 4       #   calculate &numbers[i]
    la $t2, numbers       #
    add $t3, $t1, $t2     #
    sw $v0, ($t3)         #   store entered number in array

    add $s0, $s0, 1       #   i++;
    b loop0               # }
end0:

    li   $s0, 0            # i = 0
loop1:
    bge  $s0, 10, end1     # while (i < 10) {

    mul  $t1, $s0, 4       #   calculate &numbers[i]
    la   $t2, numbers      #
    add  $t3, $t1, $t2     #
    lw   $a0, ($t3)        #   load numbers[i] into $a0
    li   $v0, 1            #   printf("%d", numbers[i])
    syscall

    li   $a0, '\n'         #   printf("%c", '\n');
    li   $v0, 11
    syscall

    add  $s0, $s0, 1       #   i++
    b loop1                # }
end1:

    li   $v0, 0            # return 0
    jr   $ra

.data

numbers:                  # int numbers[10];
     .word 0 0 0 0 0 0 0 0 0 0

string0:
    .asciiz "Enter a number: "
```

reverse10.c

read 10 integers then print them in reverse order

```c
#include <stdio.h>

int numbers[10];

int main() {
    int count;

    count = 0;
    while (count < 10) {
        printf("Enter a number: ");
        scanf("%d", &numbers[count]);
        count++;
    }

    printf("Reverse order:\n");
    count = 9;
    while (count >= 0) {
        printf("%d\n", numbers[count]);
        count--;
    }

    return 0;
}
```

reverse10.s

read 10 integers then print them in reverse order

count in register $s0 registers $t1 and $t2 used to hold temporary results

```
main:
    li   $s0, 0           # count = 0

read:
    bge  $s0, 10, print   # while (count < 10) {
    la   $a0, string0     # printf("Enter a number: ");
    li   $v0, 4
    syscall

    li   $v0, 5           #   scanf("%d", &numbers[count]);
    syscall               #
    mul  $t1, $s0, 4      #   calculate &numbers[count]
    la   $t2, numbers     #
    add  $t1, $t1, $t2    #
    sw   $v0, ($t1)       #   store entered number in array

    add  $s0, $s0, 1      #   count++;
    b read                # }

print:
    la   $a0, string1     # printf("Reverse order:\n");
    li   $v0, 4
    syscall
    li   $s0, 9           # count = 9;
next:
    blt  $s0, 0, end1     # while (count >= 0) {

    mul  $t1, $s0, 4      #   printf("%d", numbers[count])
    la   $t2, numbers     #   calculate &numbers[count]
    add  $t1, $t1, $t2    #
    lw   $a0, ($t1)       #   load numbers[count] into $a0
    li   $v0, 1
    syscall

    li   $a0, '\n'        #   printf("%c", '\n');
    li   $v0, 11
    syscall

    sub  $s0, $s0,1       #   count--;
    b next                # }
end1:

    li   $v0, 0           # return 0
    jr   $ra

.data

numbers:                 # int numbers[10];
     .word 0 0 0 0 0 0 0 0 0 0

string0:
    .asciiz "Enter a number: "
string1:
    .asciiz "Reverse order:\n"
```

scale10.c

```
#include <stdio.h>

int
main() {
    int i;
    int numbers[10];

    i = 0;
    while (i < 10) {
        printf("Enter a number: ");
        scanf("%d", &numbers[i]);
        i++;
    }
    i = 0;
    while (i < 10) {
        numbers[i] *= 42;
        i++;
    }
    i = 0;
    while (i < 10) {
        printf("%d\n", numbers[i]);
        i++;
    }
    return 0;
}
```

scale10.s

i in register $s0 registers $s1 and $s2 used to hold temporary results

```
main:
    li $s0, 0           # i = 0

loop0:
    bge $s0, 10, end0   # while (i < 10) {
    la $a0, string0     # printf("Enter a number: ");
    li $v0, 4
    syscall

    li $v0, 5           # scanf("%d", &numbers[i]);
    syscall             #
    mul $s1, $s0, 4     # calculate &numbers[i]
    la $s2, numbers     #
    add $s1, $s1, $s2   #
    sw $v0, ($s1)       # store entered number in array

    add $s0, $s0, 1     # i++;
    b loop0
end0:
    li $s0, 0           # i = 0

loop1:
    bge $s0, 10, done   # while (i < 10) {

    mul $s1, $s0, 4     # printf("%d", numbers[i])
    la $s2, numbers     # calculate &numbers[i]
    add $s1, $s1, $s2   #
    lw $a0, ($s1)       # load numbers[i] into $a0
    li $v0, 1
    syscall

    li    $a0, '\n'      # printf("%c", '\n');
    li    $v0, 11
    syscall

    add $s0, $s0, 1     # i++
    b loop1

done:
    jr $31

.data

numbers:
    .space 40           # int numbers[10];

string0:
    .asciiz "Enter a number: "
string1:
    .asciiz "Reverse order:\n"
```

endian.c

```
#include <stdio.h>
#include <stdint.h>

int main(void) {
    uint8_t b;
    uint32_t u;

    u = 0x03040506;
    b = *(uint8_t *)&u;
    printf("%d\n", b); // prints 6 on a little-endian machine
}
```

endian.s

```
main:
    li   $t0, 0x03040506

    sw   $t0, u

    lb   $a0, u

    li   $v0, 1          # printf("%d", a0);

    syscall

    li   $a0, '\n'       # printf("%c", '\n');
    li   $v0, 11
    syscall


    li   $v0, 0          # return 0
    jr   $ra

    .data
u:
    .word 0
```

unalign.c

```
#include <stdio.h>
#include <stdint.h>

int main(void) {
    uint8_t bytes[32];
    uint32_t *i = (int *)bytes[1];
    *i = 0x03040506;   // store will not be aligned on a 4-byte boundary
    printf("%d\n", bytes[1]);
}
```

unalign.s

```
main:
    li $t0, 1

    sb $t0, v1            # will succeed because no alignment needed
    sh $t0, v1            # will fail because v1 is not aligned on 2-byte boundary
    sw $t0, v1            # will fail because v1 is not aligned on 4-byte boundary

    sh $t0, v2            # will succeeed because v2 is aligned on 2-byte boundary
    sw $t0, v2            # will fail because v2 is not aligned on a 4-byte boundary

    sh $t0, v3            # will succeeed because v3 is aligned on 2-byte boundary
    sw $t0, v3            # will fail because v3 is not aligned on a 4-byte boundary

    sh $t0, v4            # will succeeed because v4 is aligned on 2-byte boundary
    sw $t0, v4            # will succeeed because v4 is  aligned on a 4-byte boundary

    sw $t0, v5            # will succeeed because v5 is aligned on a 4-byte boundary

    sw $t0, v6            # will succeeed because v6 is aligned on a 4-byte boundary

    jr   $ra             # return

    .data     # data will be aligned on a 4-byte boundary
              # most likely on at least a 128-byte boundary
              # but safer to just add a .align directive
    .align 4
    .space 1
v1:
    .space 1
v2:
    .space 4
v3:
    .space 2
v4:
    .space 4
    .space 1
    .align 2 # ensure e is on a 4 (2**2) byte boundary
v5:
    .space 4
    .space 1
v6:
    .word 0  # word directive automaticatically aligns on 4 byte boundary
```

2d_array_element_address.c

```c
#include <stdio.h>

#define X 3
#define Y 4

int main(void) {
    int array[X][Y];

    for (int x = 0; x < X; x++) {
        for (int y = 0; y < Y; y++) {
            array[x][y] = x + y;
        }
    }

    for (int x = 0; x < X; x++) {
        for (int y = 0; y < Y; y++) {
            printf("%d ", array[x][y]);
        }
        printf("\n");
    }

    printf("sizeof array[2][3] = %lu\n", sizeof array[2][3]);
    printf("sizeof array[1] = %lu\n", sizeof array[1]);
    printf("sizeof array = %lu\n", sizeof array);

    printf("&array=%p\n", &array);
    for (int x = 0; x < X; x++) {
        printf("&array[%d]=%p\n", x, &array[x]);
        for (int y = 0; y < Y; y++) {
            printf("&array[%d][%d]=%p\n", x, y, &array[x][y]);
        }
    }
}
```

emulating_2d_array_indexing.c

non-portable code illustrating 2d-array indexing this relies on pointers being implemented by memory addresses which most compiled C implementations do

```c
#include <stdio.h>
#include <stdint.h>


uint32_t array[3][4] = {{10, 11, 12, 13}, {14, 15, 16, 17}, {18, 19, 20, 21}};

int main(void) {
    // use a typecast to assign array address to integer variable i
    uint64_t i = (uint64_t)&array;

    // i += (2 * 16) + 2 * 4
    i += (2 * sizeof array[0]) + 2 * sizeof array[0][0];

    // use a typecast to assign  i to a pointer vaiable
    uint32_t *y = (uint32_t *)i;

    printf("*y = %d\n", *y); // prints 20
}
```

print2d.c

print a 2d array

```
#include <stdio.h>

int numbers[3][5] = {{3,9,27,81,243},{4,16,64,256,1024},{5,25,125,625,3125}};

int main(void) {
    int i = 0;
    while (i < 3) {
        int j = 0;
        while (j < 5) {
            printf("%d", numbers[i][j]);
            printf("%c", ' ');
            j++;
        }
        printf("%c", '\n');
        i++;
    }
    return 0;
}
```

print2d.simple.c

print a 2d array

```
#include <stdio.h>

int numbers[3][5] = {{3,9,27,81,243},{4,16,64,256,1024},{5,25,125,625,3125}};

int main(void) {
    int i = 0;
loop1:
    if (i >= 3) goto end1;
        int j = 0;
    loop2:
        if (j >= 5) goto end2;
            printf("%d", numbers[i][j]);
            printf("%c", ' ');
            j++;
        goto loop2;
    end2:
        printf("%c", '\n');
        i++;
    goto loop1;
end1:
    return 0;
}
```

print2d.s

print a 2d array i in $s0 j in $s1

```
main:
    li   $s0, 0          # int i = 0;
loop1:
    bge  $s0, 3, end1    # if (i >= 3) goto end1;
    li   $s1, 0          #     int j = 0;
loop2:
    bge  $s1, 5, end2    #     if (j >= 5) goto end2;
    la   $t0, numbers    #         printf("%d", numbers[i][j]);
    mul  $t1, $s0, 20
    add  $t2, $t1, $t0
    mul  $t3, $s1, 4
    add  $t4, $t3, $t2
    lw   $a0, ($t4)
    li   $v0, 1
    syscall
    li   $a0, ' '        #         printf("%c", ' ');
    li   $v0, 11
    syscall
    add  $s1, $s1, 1     #         j++;
    b loop2              #     goto loop2;
end2:
    li   $a0, '\n'       #     printf("%c", '\n');
    li   $v0, 11
    syscall

    add  $s0, $s0, 1     #   i++
    b loop1              # goto loop1
end1:

    li   $v0, 0          # return 0
    jr   $ra

.data
# int numbers[3][5] = {{3,9,27,81,243},{4,16,64,256,1024},{5,25,125,625,3125}};
numbers:
     .word  3, 9, 27, 81, 243, 4, 16, 64, 256, 1024, 5, 25, 125, 625, 3125
```

student.c

access fields of a simple struct

```c
#include <stdio.h>
#include <stdint.h>

struct details {
    uint16_t  postcode;
    char      first_name[7];
    uint32_t  zid;
};

struct details student = {2052, "Alice", 5123456};

int main(void) {
    printf("%d", student.zid);
    putchar(' ');
    printf("%s", student.first_name);
    putchar(' ');
    printf("%d", student.postcode);
    putchar('\n');
    return 0;
}
```

student.unpadded.s

struct details { uint16_t postcode; char first_name[7]; uint32_t zid; };

offset in bytes of fields of struct details

```
DETAILS_POSTCODE   = 0
DETAILS_FIRST_NAME = 2
DETAILS_ZID        = 9

main:
    la   $t0, student           # printf("%d", student.zid);
    add  $t1, $t0, DETAILS_ZID
    lw   $a0, ($t1)
    li   $v0, 1
    syscall

    li   $a0, ' '               #   putchar(' ');
    li   $v0, 11
    syscall

    la   $t0, student           # printf("%s", student.first_name);
    add  $a0, $t0, DETAILS_FIRST_NAME
    li   $v0, 4
    syscall

    li   $a0, ' '               #   putchar(' ');
    li   $v0, 11
    syscall

    la   $t0, student           # printf("%d", student.postcode);
    add  $t1, $t0, DETAILS_POSTCODE
    lhu  $a0, ($t1)
    li   $v0, 1
    syscall

    li   $a0, '\n'              #   putchar('\n');
    li   $v0, 11
    syscall

    li   $v0, 0                 # return 0
    jr   $ra

.data

student:                # struct details student = {2052, "Alice", 5123456};
     .half 2052
     .asciiz "Andrew"
     .word 5123456
```

student.s

access fields of a simple struct

struct details { uint16_t postcode; char first_name[7]; uint32_t zid; };

offset in bytes of fields of struct details

```
DETAILS_POSTCODE   = 0
DETAILS_FIRST_NAME = 2
DETAILS_ZID        = 12

main:
    la   $t0, student           # printf("%d", student.zid);
    add  $t1, $t0, DETAILS_ZID
    lw   $a0, ($t1)
    li   $v0, 1
    syscall

    li   $a0, ' '                #   putchar(' ');
    li   $v0, 11
    syscall

    la   $t0, student           # printf("%s", student.first_name);
    add  $a0, $t0, DETAILS_FIRST_NAME
    li   $v0, 4
    syscall

    li   $a0, ' '                #   putchar(' ');
    li   $v0, 11
    syscall

    la   $t0, student           # printf("%d", student.postcode);
    add  $t1, $t0, DETAILS_POSTCODE
    lhu  $a0, ($t1)
    li   $v0, 1
    syscall

    li   $a0, '\n'               #   putchar('\n');
    li   $v0, 11
    syscall

    li   $v0, 0                 # return 0
    jr   $ra

.data

student:                # struct details student = {2052, "Alice", 5123456};
     .half 2052
     .asciiz "Andrew"
     .space 3           # struct padding to ensure zid field is ona 4-byte boundary
     .word 5123456
```

struct_address.c

```c
#include <stdio.h>
#include <stdint.h>

struct s1 {
    uint32_t    i0;
    uint32_t    i1;
    uint32_t    i2;
    uint32_t    i3;
};

struct s2 {
    uint8_t     b;
    uint64_t    l;
};

int main(void) {
    struct s1 v1;

    printf("&v1       = %p\n", &v1);
    printf("&(v1.i0) = %p\n", &(v1.i0));
    printf("&(v1.i1) = %p\n", &(v1.i1));
    printf("&(v1.i2) = %p\n", &(v1.i2));
    printf("&(v1.i3) = %p\n", &(v1.i3));

    printf("\nThis shows struct padding\n");

    struct s2 v2;
    printf("&v2       = %p\n", &v2);
    printf("&(v2.b)   = %p\n", &(v2.b));
    printf("&(v2.l)   = %p\n", &(v2.l));
}
```

struct_packing.c

```
$ dcc struct_packing.c -o struct_packing
$ ./struct_packing
sizeof v1 = 32
sizeof v2 = 20
alignment rules mean struct s1 is padded
&(v1.c1) = 0x7ffdfc02f560
&(v1.l1) = 0x7ffdfc02f564
&(v1.c2) = 0x7ffdfc02f568
&(v1.l2) = 0x7ffdfc02f56c
struct s2 is not padded
&(v2.c1) = 0x7ffdfc02f5a0
&(v2.l1) = 0x7ffdfc02f5a4
$
```

```c
#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>

void print_bytes(void *v, int n);

struct s1 {
    uint8_t     c1;
    uint32_t    l1;
    uint8_t     c2;
    uint32_t    l2;
    uint8_t     c3;
    uint32_t    l3;
    uint8_t     c4;
    uint32_t    l4;
};

struct s2 {
    uint32_t    l1;
    uint32_t    l2;
    uint32_t    l3;
    uint32_t    l4;
    uint8_t     c1;
    uint8_t     c2;
    uint8_t     c3;
    uint8_t     c4;
};

int main(void) {
    struct s1 v1;
    struct s2 v2;

    printf("sizeof v1 = %lu\n", sizeof v1);
    printf("sizeof v2 = %lu\n", sizeof v2);

    printf("alignment rules mean struct s1 is padded\n");

    printf("&(v1.c1) = %p\n", &(v1.c1));
    printf("&(v1.l1) = %p\n", &(v1.l1));
    printf("&(v1.c2) = %p\n", &(v1.c2));
    printf("&(v1.l2) = %p\n", &(v1.l2));

    printf("struct s2 is not padded\n");

    printf("&(v1.l1) = %p\n", &(v1.l1));
    printf("&(v1.l2) = %p\n", &(v1.l2));
    printf("&(v1.l4) = %p\n", &(v1.l4));
    printf("&(v2.c1) = %p\n", &(v2.c1));
    printf("&(v2.c2) = %p\n", &(v2.c2));
}
```

emulating_struct_addressing.c

non-portable code illustrating access to a struct field this relies on pointers being implemented by memory addresses which most compiled C implementations do

```c
#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>

struct simple {
    char     c;
    uint32_t i;
    double   d;
};

struct simple s = { 'Z', 42, 3.14159 };

int main(void) {
    // use a typecast to assign struct address to integer variable i
    uint64_t i = (uint64_t)&s;

    // 3 bytes of padding - likely but not guaranteed
    i += (sizeof s.c) + 3;
    // use a typecast to assign  i to a pointer vaiable
    uint32_t *y = (uint32_t *)i;

    printf("*y = %d\n", *y); // prints 42
}
```