PROGRAMMING IN C: POINTERS AND MEMORY MANAGEMENT (1)

COMP1206 - PROGRAMMING II

Enrico Marchioni
e.marchioni@soton.ac.uk
Building 32 - Room 4019

MEMORY

- Memory is where the computer stores information.
- When a variable is declared, it is assigned some memory space, depending on the type of variable.
- The variable gets whatever value is stored at that memory location.
- The compiler doesn't initialize variables in C.
- If you want a variable to have the desired value you must initialize it!
- Try running the following program and see what happens.

```c
#include <stdio.h>

int main()
{
    // Declare variables

    char c;
    short int i;
    long x;
    float f;
    double d;

    // Print values of variables without initializing them

    printf("char variable c = %c\n",c);
    printf("int variable i = %d\n",i);
    printf("long variable x = %l\n",x);
    printf("float variable f = %f\n",f);
    printf("double variable d = %f\n",d);

    return(0);
}
```

```
char variable c =
int variable i = 25042
long variable x =
float variable f = 0.000000
double variable d = 0.000000
```

- When you declare a variable, it is assigned some space depending on its type.
- The `sizeof` operator tells you how much space is taken by a variable.

```
float var;
int i;
i = sizeof( var );
```

- `var` can be a traditional variable, a structure, an array, etc.
- `sizeof` returns the number of bytes of `var` and can be stored in an integer variable.
- `sizeof` can be used to determine the memory size of an array. The number of bytes required to store an array equals the number of bytes for the array type times the number of its elements.

```
int array[10];
int i;
i = sizeof( array );
```

- In the above case `sizeof(array)` = (4 bytes) $\times$ 10 = 40 bytes.

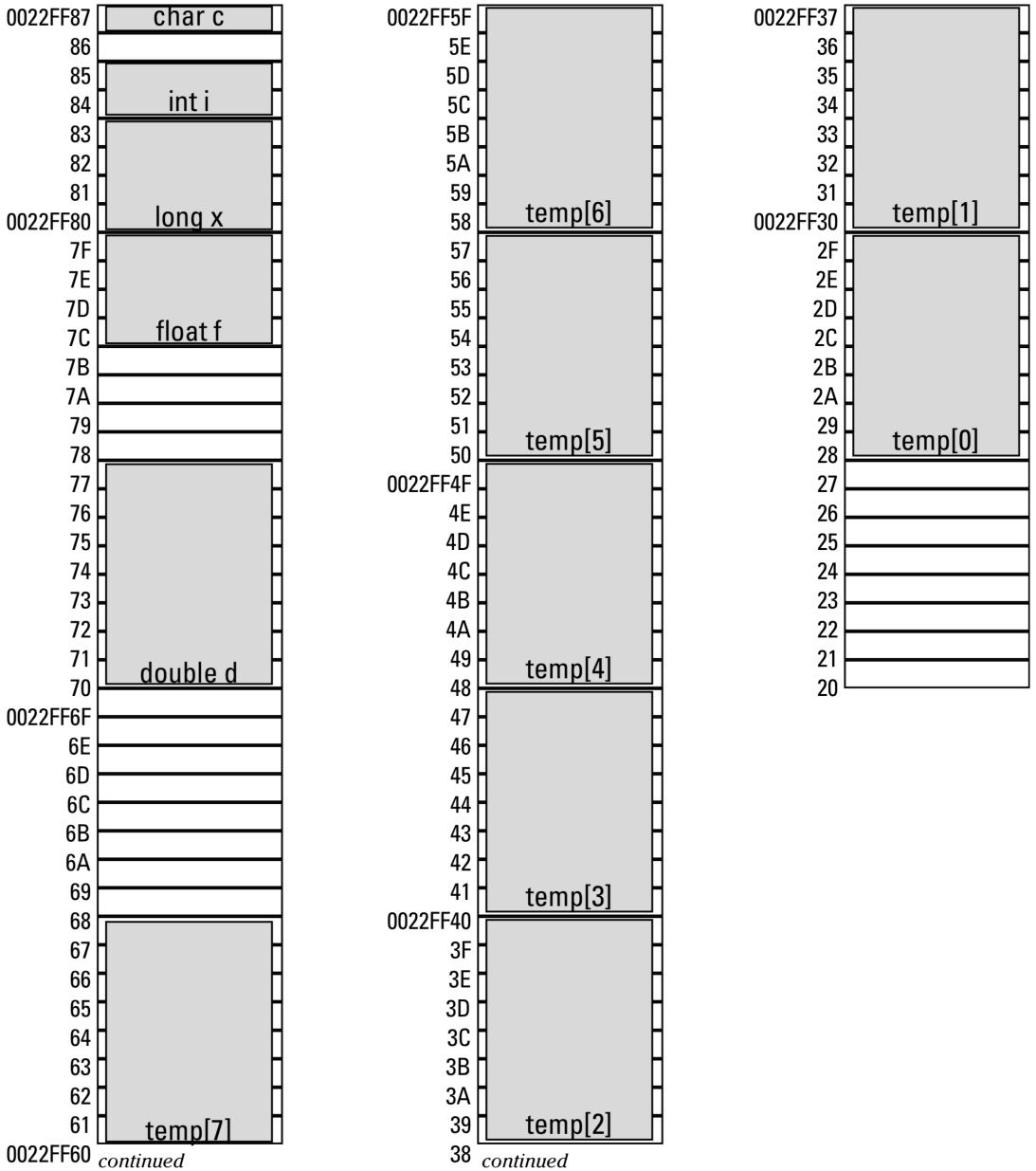
```c
#include <stdio.h>
int main()
{
    char c;
    short int i;
    long x;
    float f;
    double d;
    int array1[10];
    char array2[15];

    printf("Variable sizes:\n");

    // Obtain and print size of variables with sizeof()
    printf("Size of char variable c = %d\n",sizeof(c));
    printf("Size of short int variable i = %d\n",sizeof(i));
    printf("Size of long variable x = %d\n",sizeof(x));
    printf("Size of float variable f = %d\n",sizeof(f));
    printf("Size of double variable d = %d\n",sizeof(d));
    printf("Size of integer array array1 = %d\n",sizeof(array1));
    printf("Size of character array array2 = %d\n",sizeof(array2));
    return(0);
}
```

```
Variable sizes:
Size of char variable c = 1
Size of short int variable i = 2
Size of long variable x = 8
Size of float variable f = 4
Size of double variable d = 8
Size of integer array array1 = 40
Size of character array array2 = 15
```

- When a compiler runs, for each variable, it knows its name, its type, its size in bytes and its memory location.
- When you write a program, you know the variable name and its type, since you declared it.
- You can know its size by using `sizeof`.
- To know where the variable is located in the memory you can use the `&` operator.
- When `&` is prefixed to a variable name, it outputs its memory address.
- So, for a variable `var1`, we can know its memory address by using `&var1`.

```c
#include <stdio.h>
int main() {
    char c;
    short int i;
    long x;
    float f;
    double d;
    double temp[8];
    // Obtain and print memory address of variables with &
    printf("Memory location of char variable c = %p\n",&c);
    printf("Memory location of short int variable i = %p\n",&i);
    printf("Memory location of long variable x = %p\n",&x);
    printf("Memory location of float variable f = %p\n",&f);
    printf("Memory location of double variable d = %p\n",&d);
    for(i=0;i<8;i++)
        printf("Memory location of temp[%d] = %p\n", i, &temp[i]);
    return(0);
}
```

```
Memory location of char variable c = 0022FF87
Memory location of short int variable i = 0022FF84
Memory location of long variable x = 0022FF80
Memory location of float variable f = 0022FF7C
Memory location of double variable d = 0022FF70
Memory location of temp[0] = 0022FF28
Memory location of temp[1] = 0022FF30
Memory location of temp[2] = 0022FF38
Memory location of temp[3] = 0022FF40
Memory location of temp[4] = 0022FF48
Memory location of temp[5] = 0022FF50
Memory location of temp[6] = 0022FF58
Memory location of temp[7] = 0022FF60
```


0022FF87
char c
86
85
int i
84
83
82
81
0022FF80
long x
7F
7E
7D
7C
float f
7B
7A
79
78
77
76
75
74
73
72
71
double d
70
0022FF6F
6E
6D
6C
6B
6A
69
68
67
66
65
64
63
62
61
temp[7]
0022FF60
continued
next column
0022FF5F
5E
5D
5C
5B
5A
59
temp[6]
58
57
56
55
54
53
52
51
temp[5]
50
0022FF4F
4E
4D
4C
4B
4A
49
temp[4]
48
47
46
45
44
43
42
41
temp[3]
0022FF40
3F
3E
3D
3C
3B
3A
39
temp[2]
38
continued
next column
0022FF37
36
35
34
33
32
31
temp[1]
0022FF30
2F
2E
2D
2C
2B
2A
29
temp[0]
28
27
26
25
24
23
22
21
20

POINTERS

- A pointer is a variable that holds a memory address.
- Pointer variables are declared like other variables.
- You need to specify a type, but most importantly, you need to prefix `*` to the variable name.

```
int (float ...) *pointervar;
```

- `pointervar` is the name of the variable. The compiler knows it's a pointer because `pointervar` is preceded by the asterisk `*`.
- Pointer variables need a type, because the type specifies the kind of data that lives at the memory location.
- How do you initialize a pointer variable?

```c
#include <stdio.h>

int main()
{
    float euros, pounds;
    float *p_euros; // Declare pointer

    printf("Enter a value in pounds:");
    scanf("%f",&pounds);

    euros = pounds*0.8763;
    printf("%f pounds work out to %f euros.\n",pounds, euros);

    p_euros = &euros;                /* Initialize pointer */

    printf("Variable 'euros' is %i bytes long at %p address\n",sizeof(euros),p_euros)
        ;

    return(0);
}
```

```
Enter a value in pounds:245
245.000000 pounds work out to 214.693497 euros.
Variable 'euros' is 4 bytes long at 0x7ffeefbff588 address
```

- `p_euros` is declared as a pointer variable of type `float`

```
float *p_euros;
```

- The asterisk `*` means that `p_euros` is a pointer variable and contains a memory address, i.e. the address of a `float`.
- Like all variables, pointers must be initialized.
- The statement

```
p_euros = &euros;
```

  initializes the pointer and assigns to it the address of `euros`
- `&` is used to obtain the address of `euros`
- The asterisk `*` is NOT used when a pointer variable is initialized.
- THIS IS WRONG!

```
*p_euros = &euros;
```

```c
#include <stdio.h>
int main()
{
    int array[5] = { 2, 4, 6, 8, 10};
    int *a; // Declare pointer

    a = array; // Initialize pointer with array

    printf("Array 'array' is %i bytes long and lives at %p address.\n",sizeof(array),
          a);

    return(0);
}
```

- Arrays are declared similar to other variables, but they behave differently.
- Pointers can be assigned the memory location of an array.

```c
a = array;
```

- To do that you do NOT need the operator `&` in front of the array.
- THIS IS WRONG!

```c
a = &array;
```

- The compiler knows the array is an address, so `&` is unnecessary.

```c
#include <stdio.h>
int main()
{
    int array[5], x;
    int *a,*e; // Declare pointers

    for (x = 0; x < 5; ++x)
    {
        array[x] = x;
    }

    a = array; // Initialize pointer with array
    e = &array[3]; // Initialize pointer with array element

    printf("Array 'array' is %i bytes long and lives at %p address \n",sizeof(array),
          a);

    printf("%i is the fourth element of 'array', is %i bytes long and lives at %p
          address.\n",array[3],sizeof(array[3]),e);

    return(0);
}
```

```
Array 'array' is 20 bytes long and lives at 0x7ffeefbff620 address
3 is the fourth element of 'array', is 4 bytes long and lives at 0x7ffeefbff62c
     address.
```

- When you assign the address of an array element to a pointer you still need `&`.

- We have seen that pointers are variables that hold a memory address.
- They need a type and must be declared.
- They are initialized by assigning them an address.
- We will see later that they make it possible to work with memory addresses.
- Pointers can also be used to work with whatever lives at the address they hold, i.e. they can be used to work with the specific data that is contained at a memory location.

```c
#include <stdio.h>

int main()
{
    int var1, *point1;

    var1 = 1;
    point1 = &var1;

    printf("The value of var1 is %i\n", var1);
    printf("The value of point1 is %p\n", point1);

    // Print the value of *print1
    printf("The value of *point1 is %i\n", *point1);
    return 0;
}
```

```
The value of var1 is 1
The value of point1 is 0x7ffeefbff638
The value of *point1 is 1
```

- To work with the value living at an address assigned to a pointer you must use the asterisk `*` in front of the pointer variable name.
- `*point1` tells us what is at the location `0x7ffeefbff638`.
- This makes it possible to manipulate the value of variable with pointers.
- Since

```
var1 = 1;
point1 = &var1;
```

  we have that

```
*point1 = 1;
```

- In fact, after initializing the pointer, `*point1` and `var1` are essentially the same.
- `*` is called the *dereference operator*

```c
#include <stdio.h>
int main()
{
    int group1,group2;
    int *outcome;

    outcome = &group1;
    *outcome = 63;
    printf("The average mark for group 1 is %i.\n", group1);

    outcome = &group2;
    *outcome = 67;
    printf("The average mark for group 2 is %i.\n", group2);

    return(0);
}
```

```
The average mark for group 1 is 63.
The average mark for group 1 is 67.
```

- A pointer variable can be shared, i.e. `outcome` is used for storing the address of `group1` first and, later, the address of `group2`.
- Both `group1` and `group2` are initialized indirectly, through the pointer.
- We can manipulate what lives at a memory address by using a pointer.

POINTER ARITHMETIC

```c
#include <stdio.h>

int main()
{
    short int array[5];
    short int *pa, x;

    for(x=0;x<5;x++)
    {
        array[x] = x;
        pa = &array[x];
        printf("array[%i] at %p = %i\n",x,pa,array[x]);
    }

    return(0);
}
```

```
array[0] at 0022FF78 = 0
array[1] at 0022FF7A = 1
array[2] at 0022FF7C = 2
array[3] at 0022FF7E = 3
array[4] at 0022FF80 = 4
```

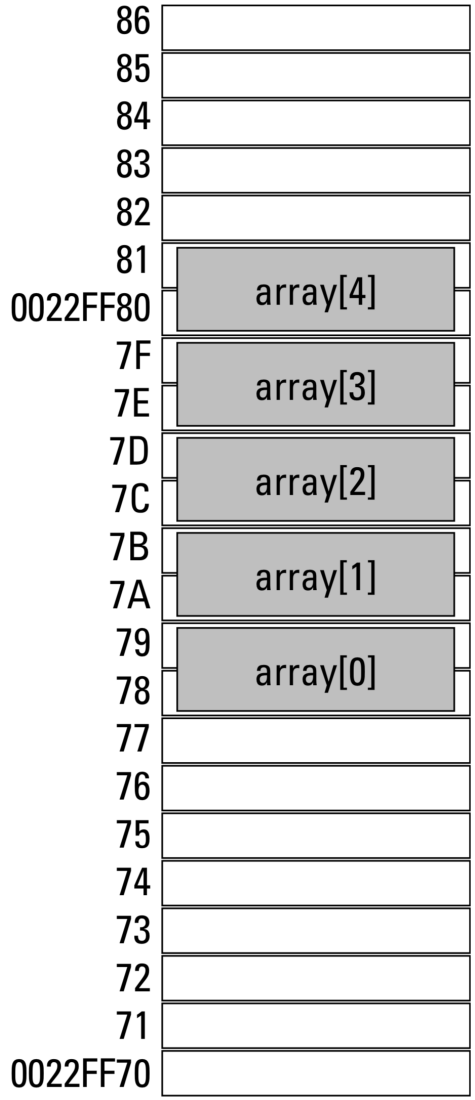
```c
#include <stdio.h>

int main()
{
    short int array[5];
    short int *pa, x;
    pa = array;
    for(x=0;x<5;x++)
    {
        array[x] = x;
        printf("array[%i] at %p = %i\n",x,pa,array[x]);
        pa++;
    }

    return(0);
}
```

```
array[0] at 0022FF78 = 0
array[1] at 0022FF7A = 1
array[2] at 0022FF7C = 2
array[3] at 0022FF7E = 3
array[4] at 0022FF80 = 4
```

- The output of the above programs is the same, but the code is different.
- In the first program we have:

```
for(x=0;x<5;x++)
    {
        array[x] = x;
        pa = &array[x];
        printf("array[%i] at %p = %i
            \n",x,pa,array[x]);
    }
```

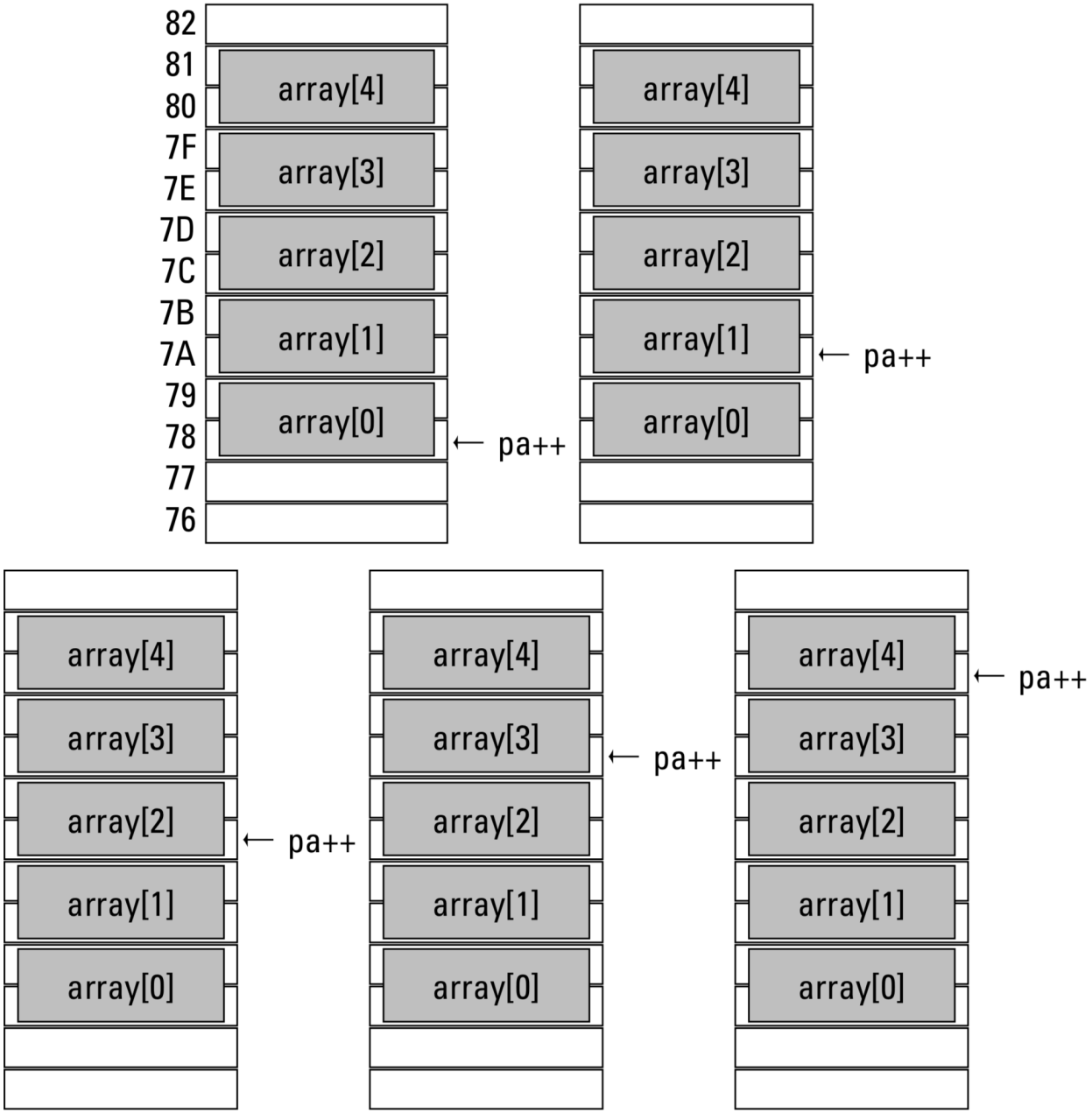
- For each `x` the pointer variable `pa` is assigned the address of the element `array[x]`.
- Whenever the value of `x` is increased by one, the the pointer variable `pa` takes as value the memory location of the following element in the array.



- In the second program we have:

```
pa = array;
for(x=0;x<5;x++)
    {
        array[x] = x;
        printf("array[%i] at %p = %i\n",x,pa,array[x]);
        pa++;
    }
```

- The pointer variable `pa` is assigned the address of the array `array` before the start of the loop.
- At each value of `x` in the loop the value of `pa` is incremented with `pa++`.
- Notice that `pa++` does not jump to the memory address that immediately follows the initial one.
- An array occupies as many memory blocks as the number of its elements and each block has the size of the data type of the array.
- In this case our array has 5 blocks of 2 bytes each and `pa++` jumps to the next memory block.
- Pointers are manipulated in blocks, each block having a size equal to the variable type.


82
81
80
7F
7E
7D
7C
7B
7A
79
78
77
76
array[4]
array[3]
array[2]
array[1]
array[0]
← pa++
array[4]
array[3]
array[2]
array[1]
array[0]
← pa++
array[4]
array[3]
array[2]
array[1]
array[0]
← pa++
array[4]
array[3]
array[2]
array[1]
array[0]
← pa++

```c
#include <stdio.h>

int main()
{
    float var1;
    float *p1;

    p1 = &var1;

    printf("Variable var1 is at location %p\n", p1);
    printf("The next three floats in memory are at location\n %p,\n %p,\n %p\n", p1
          +1, p1+2, p1+3);

    return 0;
}
```

```
Variable var1 is at location 0x7ffeefbff638
The next three floats in memory are at location
 0x7ffeefbff63c,
 0x7ffeefbff640,
 0x7ffeefbff644
```

- Pointer math uses units that correspond to blocks of memory.
- Each block has size equal to the type of the pointer.
- A float pointer will need a block of 4 bytes.
- Adding/subtracting/multiplying pointers will work with memory units, i.e. in blocks of memories determined by the variable type.