Introduction to Programming and Computational Physics

Lesson n.7

Pointers
Dynamic allocation

Values and addresses

Declaring a variable means associating a purely symbolic name called variable to a memory location . The type of the variable specifies how much space has to be allocated for the variable.

The memory is a sequence of contiguous cells, each cell has an own address.

The address of a variable is the address of its first cell.

```
int myvar = 643437; //assigning the value
```

00000000	00001001	11010001	01101101
0x2020E687	0x2020E688	0x2020E689	0x2020E68A
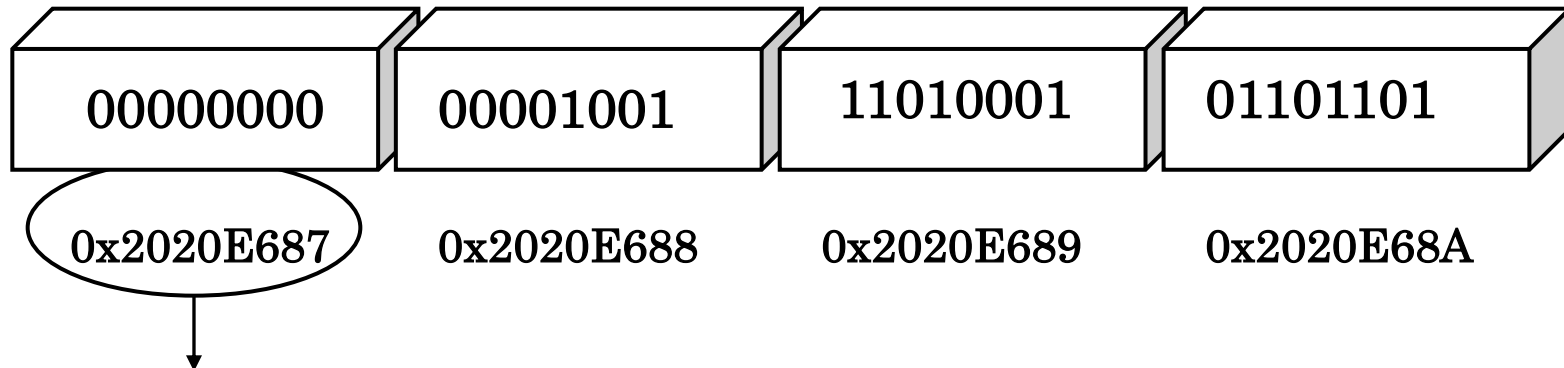
Address of `myvar`: `&myvar`

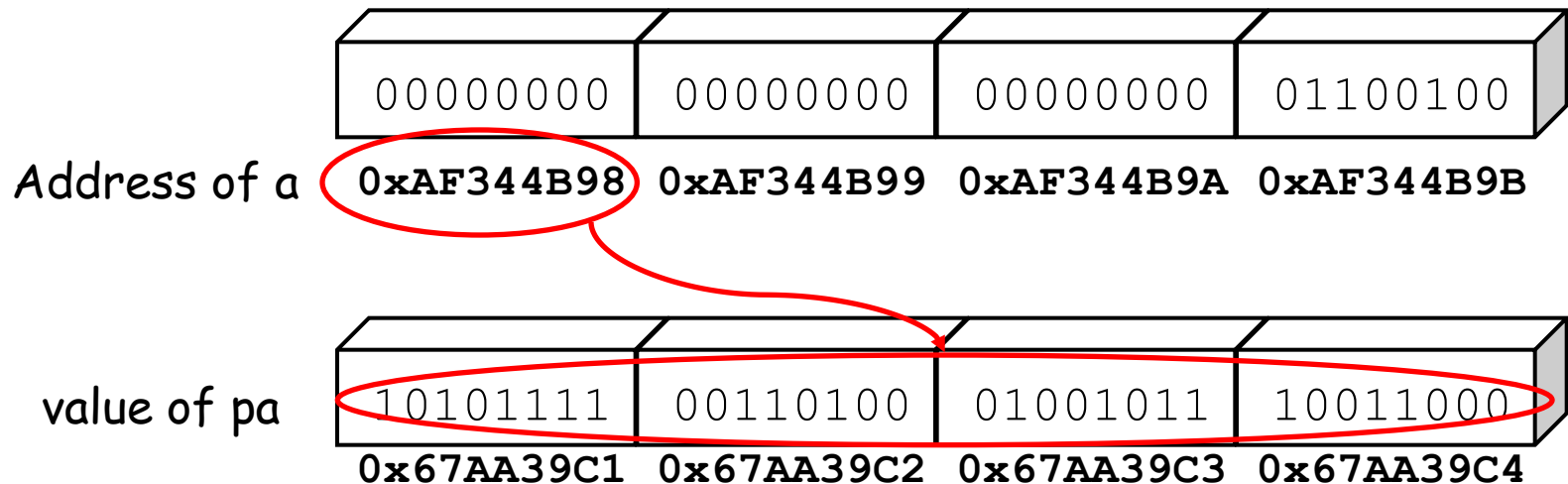& is an operator returning the address of the input variable.

Pointers

A pointer is a 4 byte (or in 64bit operation systems: 8 byte) data type whose value is the address of another variable. When we define a pointer we have to say what is the *pointed* data type.

`int a;` //integer

`int* pa;` //pointer to an integer (same as: `int *pa;`)

`a = 100;`

`pa = &a;` //the pointer pa points to the integer a

00000000	00000000	00000000	01100100

Address of a **0xAF344B98 0xAF344B99 0xAF344B9A 0xAF344B9B**

10101111	00110100	01001011	10011000

value of pa

0x67AA39C1 0x67AA39C2 0x67AA39C3 0x67AA39C4

`printf("If I print %d and %d, the result is the same`

`address\n",pa,&a);`

Pointers dereferencing

The pointer provides an indirect way to access the value of the pointed variable.

```
int a = 100;  (1)
int* pa;  // (2) pa is a pointer to an int
pa = &a;  // (3) let pa point to a
int b = *pa;   //pointer dereferencing (4)

(take the value to which pointer pa points and
copy it to variable b)
```

The value of b is set to 100. If now we write:

```
int c = 200;  (5)
*pa = c;  (6 write the value of c to the value
          with the address pa)
```

The value of a (the variable pointed to by pa) is now 200. This means that we can modify the value of a variable using a pointer to it.

a and *pa are fully equivalent, we can use both to access the value of the variable (as long as pa points to a)

„newborn" pointers always point to the invalid address 0x0

Address	Value	Var
..1200	(1) 100	a
..1204	(4)	
..1208	100	b
..9840	(2) 0	pa
	(3) ..1200	

Address	Value	Var
..1200	(6) 200	a
..1204		
..1208	100	b
..1212	(5) 200	c
..9840	..1200	pa

4

Summarizing...

```
int a = 100;
int* pa;
pa = &a;
int b = *pa;
```

& is the *referencing* operator (the address of). When applied to a variable, it returns the address of the variable. (If applied to a pointer it returns the address of the pointer)

```
printf("If I print %d and %d, the result is the
same address\n",pa,&a);
```

* is the *dereferencing* operator (the content of). When applied to a pointer, it returns the value of the pointed variable. (It can be applied only to a pointer)

```
printf("If I print %d and %d, the result is the
same value\n",*pa,a);
```

Pointers and arrays

If we define an array(1) and set some values(2)

`int v[5];` `v[0]=193;` `v[1]=13;` `v[2]=19;` `v[3]=23;`

each variable of the array has its own address:

&v[0], &v[1] …

The name of the array contains the address of the first element of the array (in other words, it is a pointer to the first location of the memory assigned to the array)

v ⟷ &v[0]

We can then define another pointer to the array by writing:

`int* pv = v;` (3)

    (the same as: `int* pv = &v[0]`)

Don't use `&v`, it is different from `&v[0]` and `v` !

Address	Value	Var
..2210	(2) 193	
..2214	(2) 13	
..2218	(2) 19	
..2222	(2) 23	
..2226		
..8888	..2210	(1) v
	(3) ..2210	pv

Pointers and arrays

int v[5] = {3,4,7,9,1}

int* pv = v;

When we define a pointer to the array, we can easily access all the element of the array:

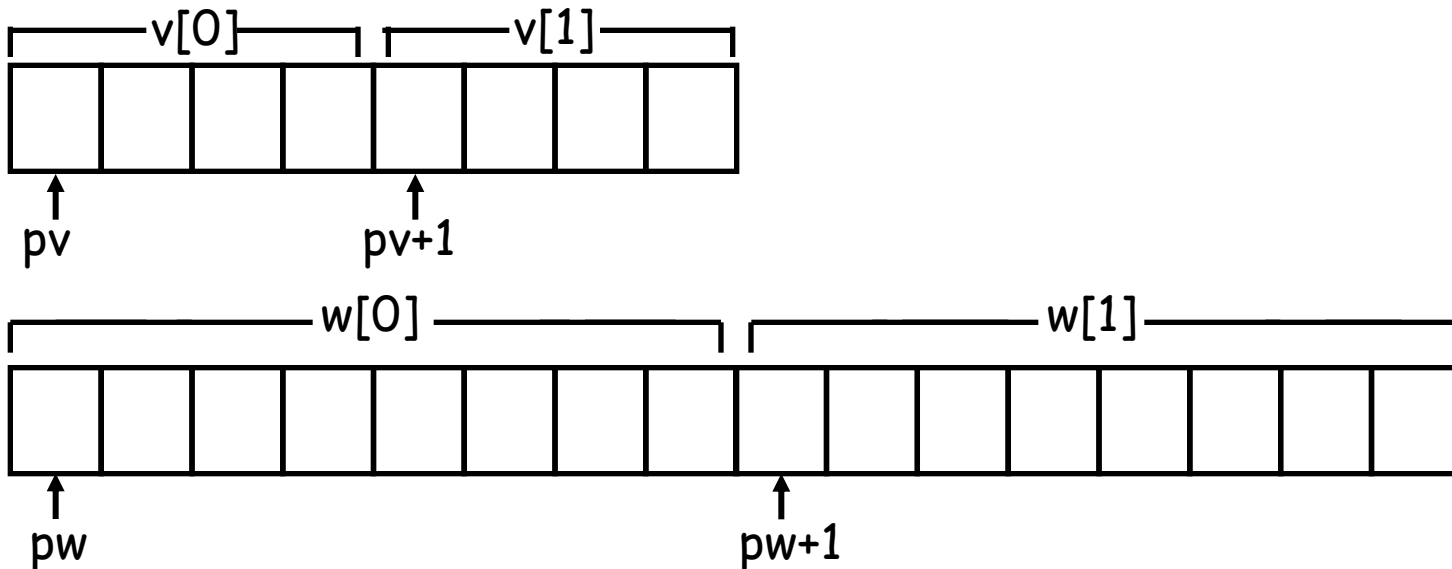| value	v[0]	v[1]	v[2]	v[3]	v[4]
address	&v[0]	&v[1]	&v[2]	&v[3]	&v[4]
value	*v	*(v+1)	*(v+2)	*(v+3)	*(v+4)
address	v	v+1	v+2	v+3	v+4
value	*pv	*(pv+1)	*(pv+2)	*(pv+3)	*(pv+4)
address	pv	pv+1	pv+2	pv+3	pv+4

Pointers arithmetic

A pointer contains an address, so the only meaningful operations involving pointers are increment and decrement, by using the operators:

+   ++   -   --

If we have:

```
float v[2]; float* pv = &v[0];

double w[2]; double* pw = &w[0];
```

Passing by address

A function can modify the value of a variable defined elsewhere if it knows the address. There is indeed a second way to pass parameters to function, it is called "by address".

```
typeReturned nameFun(typePar1*,...); //declaration

typeReturned nameFun(typePar1* var,...)
{
        //implementation
}
```

Example: A function which calculates the two solutions of a quadratic equation:

```
void square_equation(float a, float b, float c, float* r1, float* r2)

{

        ...

}
```

When we call the function passing a pointer, the value of the pointer (the address of the pointed variable) is copied to a local pointer. This pointer can therefore access (and modify) the value of the pointed variable, even if the variable itself is not visible by the function.

Now we know why scanf needs &

int a;
printf("Please enter a number\n");
scanf("%f",a);   // if scanf were like this, it would not
                                be able to actually change variable a

scanf("%f",&a); // like this, we give the address of "a"
                                - so scanf can happily change it

Somewhere inside scanf:
scanf(char* format, void *a)
{
    ...
    *a = myvar;
    ...
}

Compound interest

```c
#include <stdio.h>

void UpdateCap(float*,float);

int main()
{
    int years = 5; //number of years
    float in = 0.03; //interest rate
    float cap = 10000.; //initial capital
    float *pcap = &cap;
    int i;

    for (i=1; i<=years; i++)
        UpdateCap(pcap,in);

    printf("The capital after %d years will be: %.2f \n", years,cap);
    return 0;
}

void UpdateCap(float *pcap, float in)
{
    *pcap = (*pcap)*(1+in);
}
```

Swap

```c
#include <stdio.h>

void swap(int*,int*);

int main()
{
    int x = 2,y = 5;
    int *px = &x, *py = &y;
    printf("values before swap: %d %d\n",x,y);
    swap(px,py); //or swap(&x,&y);
    printf("values after swap: %d %d\n",x,y);
    return 0;
}

void swap(int *a, int *b)
{
    int temp;
    temp = *a;
    *a = *b;
    *b = temp;
}
```

Passing by address (array)

```c
#include <stdio.h>
#define dim 10
void sort(int*,int);
void swap(int*,int*);

int main()
{
    int v[dim], i;
    for (i=0; i<dim; i++)
    {
        printf("Enter no.%d\n",i+1);
        scanf("%d",&v[i]);
    }
    sort(v,dim);
    for (i=0; i<dim; i++)
        printf("no.%d is %d\n",i+1,v[i]);
    return 0;
}
```
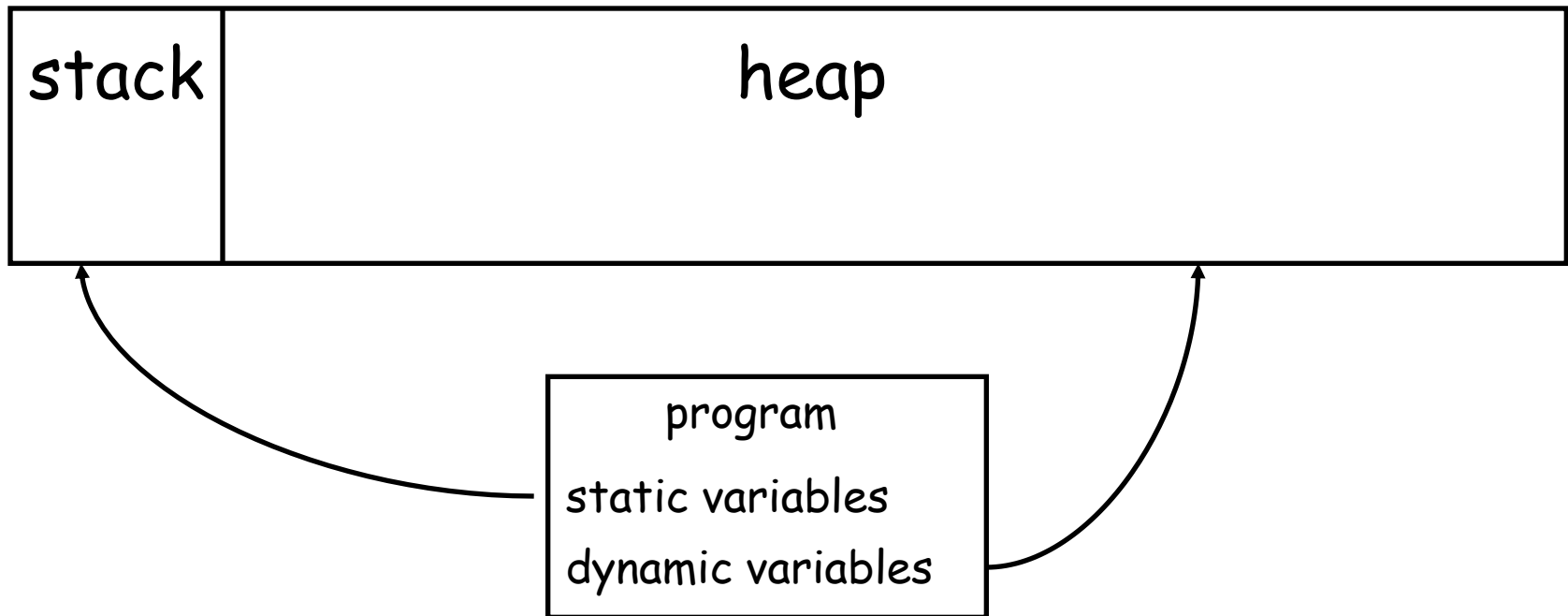
In C language an array cannot be passed or returned. The only way to use arrays with external functions is passing their address.

The program shows how we can sort an array

```c
void sort(int *pv,int d)
{
    int i,j;
    for(i=0; i<d; i++)
        for(j=i+1; j<d; j++)
        {
            if(*(pv+i)>*(pv+j))
                swap(pv+i,pv+j);
        }
}
```

```c
void swap(int *pc, int *pd)
{
    int temp;
    temp = *pc;
    *pc = *pd;
    *pd = temp;
}
```

Memory handling

stack	heap

```
       program
   static variables
   dynamic variables
```

The static variables (the only ones that we used until now) are allocated in a small memory space (stack) associated to the program when we start the program execution.

The C language offers the possibility to allocate space *dynamically* for variables in a much bigger *shared* place (heap). The dynamic allocation can be done at any time during the program execution.

Dynamic allocation

There are 2 functions for the memory dynamic allocation:

```
void *malloc (int dim);
```

```
void *calloc (int ELnum, int ELdim);
```

where dim is the quantity of memory to be allocated.

They are defined in the malloc.h library

We can use them in the following way:

```
int *pa;
pa = (int*) malloc(10*sizeof(int));
```

or

```
pa = (int*) calloc(10,sizeof(int));
```

And then we can access the memory using the pointers:

```
*pa =  5;   *(pa+1) = -1;    ...   *(pa+9) = 0;
```
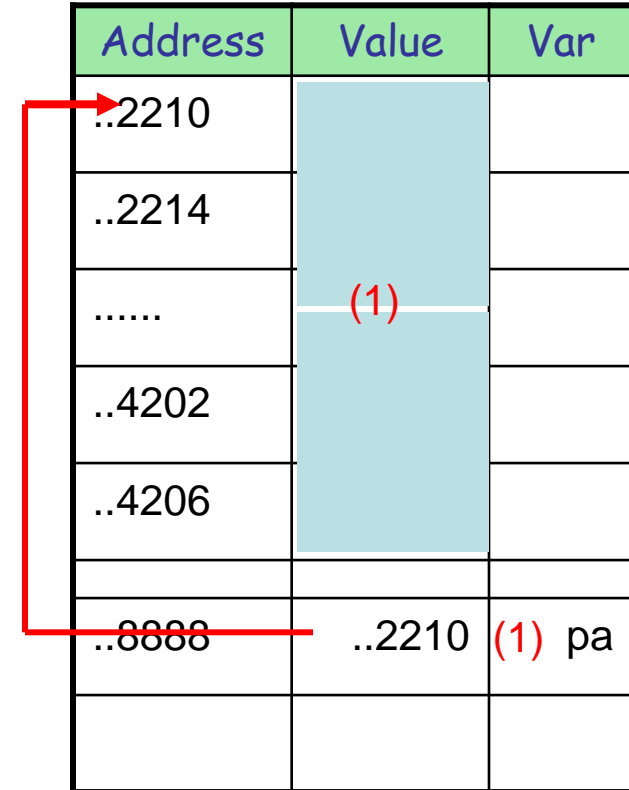   *OR array-like*
```
pa[0] = 5;  pa[1] = -1;      ...   pa[9] = 0;
```

Dynamic allocation

Variables dynamically allocated need to be explicitly *deallocated*

This is done with the function "free":

int *pa = (int*) malloc (1000*sizeof(int));

… // (1) we create a memory block of 1000 ints

…

free(pa); //and at the end,
          we need to delete it.

Address	Value	Var
..2210		
..2214		
......	(1)	
..4202		
..4206		
..8888	..2210	(1) pa

If we don't free the memory, it will be lost

And cannot be reclaimed (memory leak)

16

Average and standard deviation for a data set

```c
#include <stdio.h>
#include <malloc.h>
#include <math.h>

int main()
{
    int n,i;
    float sum = 0, ave = 0, sum2 = 0, stdev = 0;
    printf("number of data? \n");
    scanf("%d",&n);
    float *pv = (float*)malloc(n*sizeof(float));
    for (i=0; i<n; i++)
    {
        printf("Enter a value: \n");
        scanf("%f", pv+i);
        sum += *(pv+i);
    }
    ave = sum/n;
    for (i=0; i<n;i++)
        sum2 += pow(*(pv+i)-ave,2);
    stdev = sqrt(sum2/(n-1));
    free(pv);
    printf("ave = %.2f st.dev = %.2f \n",ave,stdev);
    return 0;
}
```

Segmentation fault - why?

code

int *a;
if (b > c)
{
    int q = 100;

    a = &q;
}

printf("%d",*a);

What happens

Pointer a is created.

Starting scope with {
q is defined inside this scope

Pointer a gets the address of q

At the end of scope (}), q is deleted implicitely, but pointer a still points to the location where q was.

If we try to access the value which a points to, we are accessing unallowed memory
-> Segmentation violation

This is also what happens if we forget the & in scanf. We are giving some arbitrary value as address to scanf, and as soon as scanf tries to write there, it accesses unallowed memory and the program crashes.