C Programming I: Lecture V

Andreas Heinz
FYD400 - HT 2019

xkcd.com

Recap questions

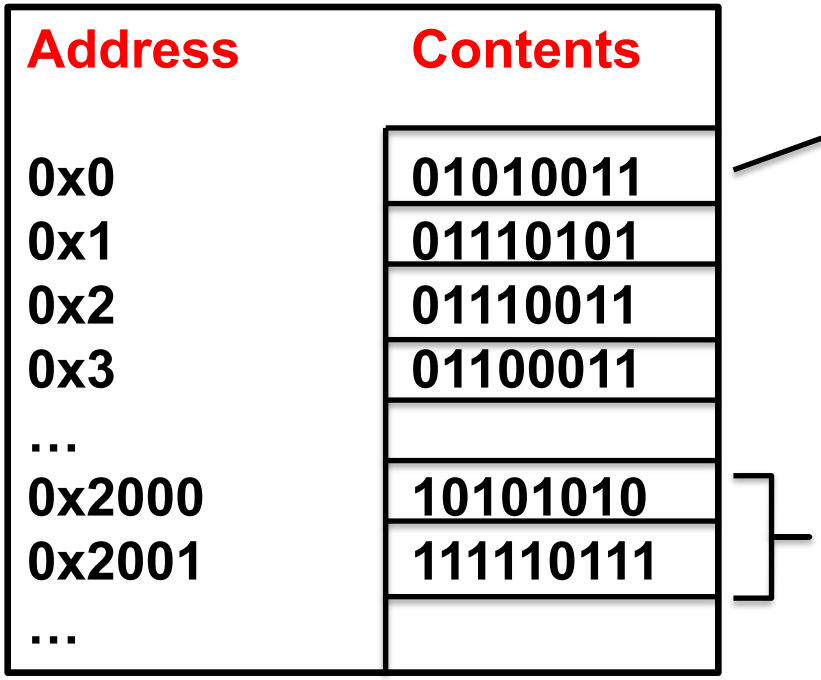
- **What is a compound statement?**
- **How does an *if*-statement work?**
- **How does a *switch*-statement work?**
- **What types of loops do you know and how do they differ?**
- **What is the difference between local and global variables?**
- **How can I write a program that consists of several files?**

Lecture 5 Contents

- **Pointers**
- **Arrays (and pointers)**
- **Dynamic memory allocation**
- **Structs (and unions)**

(VtC, Chapter 7 and 8)

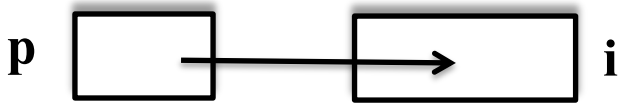
Memory and Addresses: Pointers



Data are stored as one or more bytes with 8 bits each (for example).

Each variable has a memory address given by the address of the first byte.

variable i of type that needs 16 bits; address: 0x2000



"Pointer" p points to the address of i.

Pointers

- A pointer is a variable which contains the (memory) address of a variable (= pointer value)!
- A pointer value (the address of the variable) can be assigned to a pointer variable.

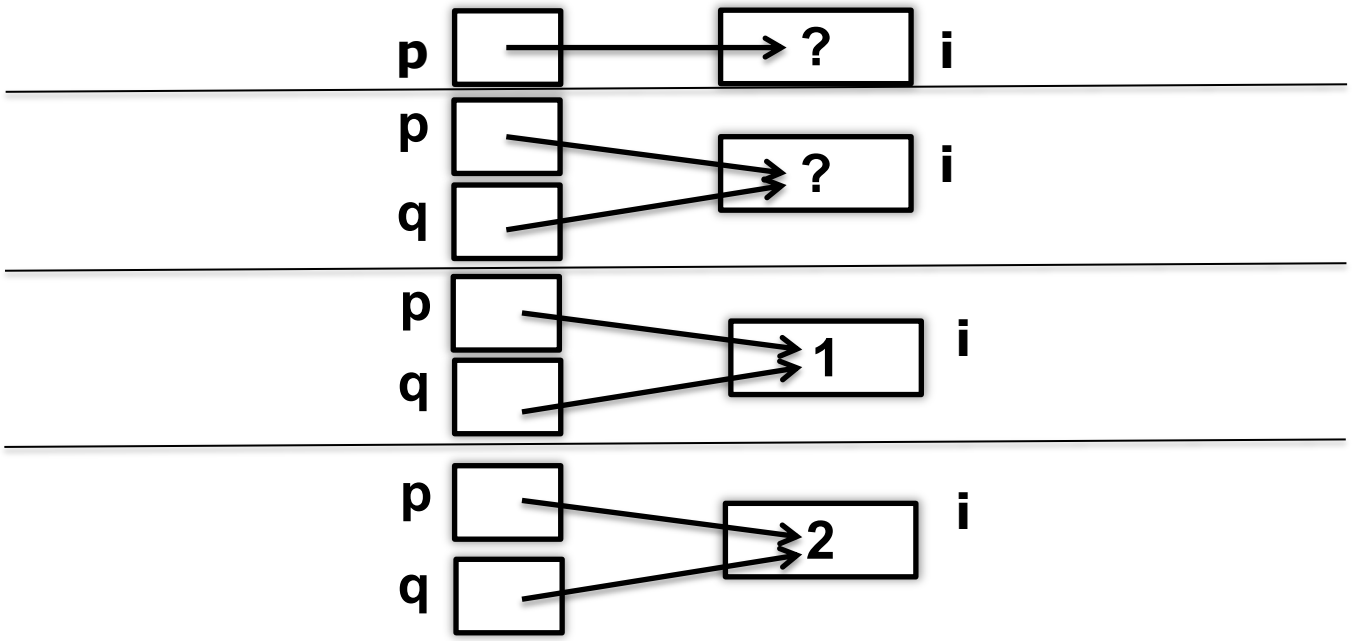
```
float z, x = 13.5;          // declaration of x, "normal"  float variable
float *floatp, *floatq;     // declaration of pointer variables
```

General: '*floatp' points to a float variable; accesses variable value
&floatp: address of floatp, pointer value ('&' = address operator)

```
floatp = &x;// this is the way to have floatp point to address of  x !
z = *floatp +5; // z == 18.5 ; *floatp returns the value stored in the
                // variable it points to ('*' = indirection operator )
                // identical statement: z = x + 5;
floatq = floatp;        // floatp and floatq point to the same address
*floatq = *floatp;      // copies the value floatp points to into floatq
floatp = NULL;          // null pointer – points to "nowhere"
```

Pointers

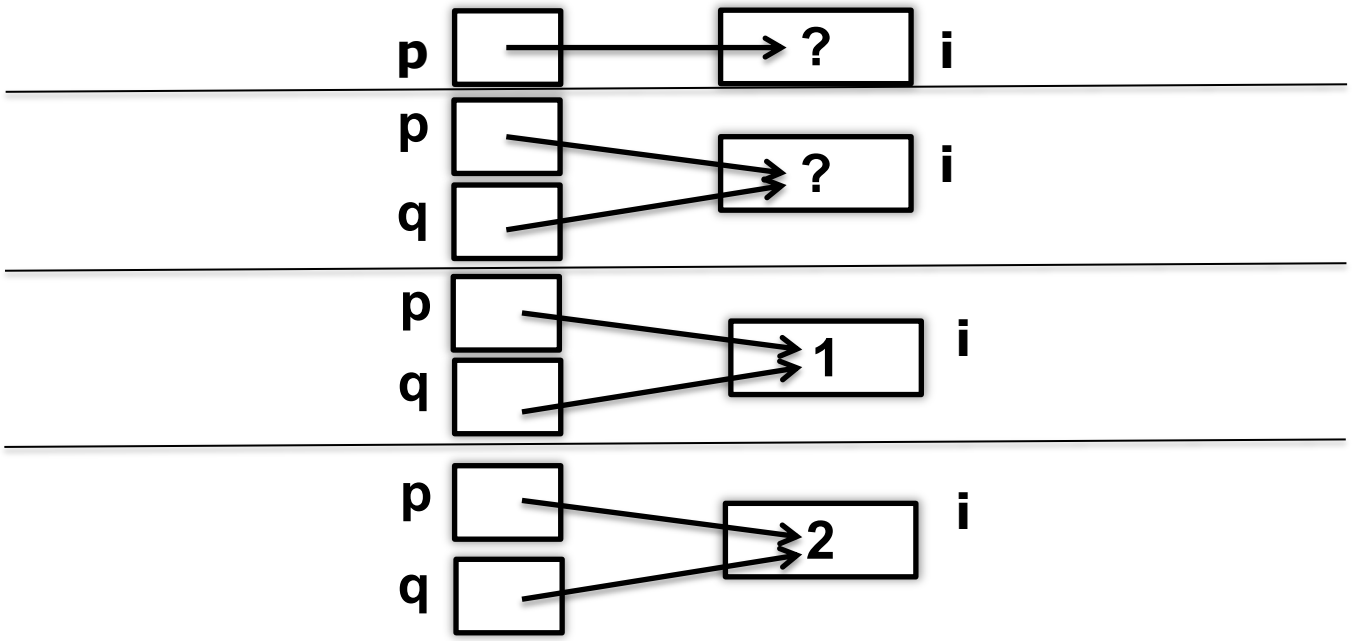
```
int i, j, *p, *q;
...
p = &i;

q = p;

*p = 1;

*q = 2;
```




```
printf("%d", *p);      // prints "2", indirection operator: "*"
Think of "*" as the "inverse" of "&".
printf("%p",p);        // prints the address p points to.
```

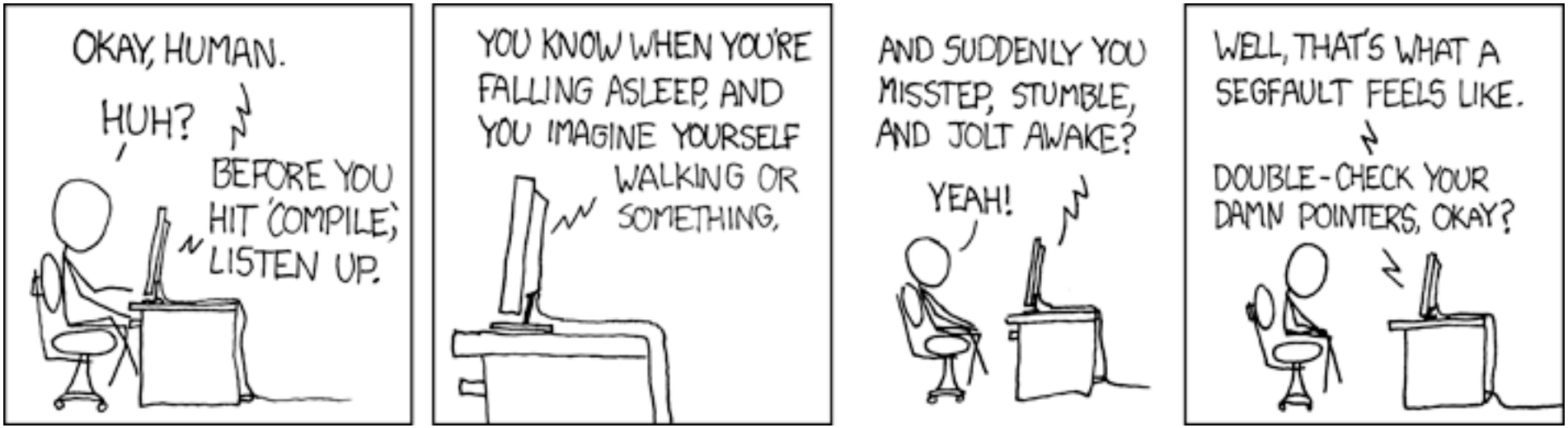
Pointers

```
int i, j, *p, *q;
...
p = &i;

q = p;

*p = 1;

*q = 2;
```




```
printf("%d", *p);      // prints "2"; indirection operator: "*"
Think of "*" as the "inverse" of "&".
printf("%p",p);        // prints the address p points to.
```

Pointers



xkcd.com

Exercise

Explain the ideas about pointers we just discussed to your neighbor.

Arrays

- important for manipulating text strings but also of data sets
- up to now we used only "scalar" variables
- aggregate values can store collections of data of the same type (elements)
- simple figure of a 1-dimensional array a[8]:

Arrays

- **declaration** of an array: *int array [100]; // array with 100 // entries*
  - the length of an array must be known and constant
  - all elements must be of the same (declared) type
  - **index** runs from **0** – 99! It **starts at 0**!!!!
- **initialization** of an array: *int array [100] = { 3, 4, 11, 2 };*
  - the first four elements have values, 96 elements are zero.
- **indexing** of an array:

  Beware of using an index which exceeds the declaration!!!!

  array[100] = 7; // tries to assign the 101nd element…

  'array' is a pointer to the first element of the array!

  *int *intp = array; // pointer intp points to first element of // array… watch for the type!*

Pointers and Arrays

- ***intp == array [1];*** **// checks if value intp points to is**
  // the same as that of the 2nd element of array
- **operations with pointers:**
  - **assignment, addition, subtraction, address (&), indirection (*)**
    Example: *intp[9] == *(intp + 9);* // VtC, chapter 7.3
- **Pointer stepping in arrays (very common!)**

Example from chapter 7.4 in VtC:

```
int i;
float sum = 0.0, fflt [1000], *fp;     // ???
fp = fflt;                             // ???
for (i=0; i < 1000, i++)               // ???
{
        sum += *fp++;                  // ???
}
```

Guess the comments!

Pointers and Arrays

- ***intp == array [1]; // checks if value intp points to is***
  // the same as that of the 2nd element of array
- **operations with pointers:**
  - **assignment, addition, subtraction, address (&), indirection (*)**
    Example: *intp[9] == *(intp + 9); // VtC, chapter 7.3*
- **Pointer stepping in arrays (very common!)**

Example from chapter 7.4 in VtC: Summing up an array

```
int i;
float sum = 0.0, fflt [1000], *fp;     // works because type is the same
fp = fflt;              // fp points to the 1st element of fflt (with index = 0)
for (i=0; i < 1000, i++)      // loop to sum up all elements of array fflt
{
        sum += *fp++;     // takes the value fp points to, and increments
}
```

Arrays of Characters

- *char text [50];* // array with 50 elements (actually 49 plus a '\0' at the end)
- *char lecture [] = "Physics";* // length of 8 elements (P, h, y, s, i, c, s, \0)
  // lecture is a pointer to the first element of lecture
  *char *pcharac;* // pointer to a character
  pcharac = lecture; // pcharc points to the same location as lecture

Example:

```
char flt [20], *ptext;  // flt[20] contains a string literal (i.e. array)
int nchar = 0;
ptext = flt;            // ptext is a pointer that points to element 0 of flt
while (*ptext != '\0')
{
        nchar++;        // calculates the number of characters in flt
        ptext++;        // moves the pointer one step up
}
```

- *"\0"* // null character
- *"I am a string literal."*

Array of Arrays, Array of Pointers

- ***int matrix [3] [5];*** **// three "rows", each with five "columns"**
- ***int *p = matrix [1];*** **// pointer to the 2nd row, 1st element**

Fill a matrix
int matrix [3] [5] = { {1,2,3,4,5},{6,7,8,9,10},{11,12,13,14,15}};
int *p = matrix [1]; **// p points to the 2nd row and 1st element (== 6)**

char *lines [20]; **// array of pointers with 20 elements**
// Used in a (multi-dimensional) array, this gives you
// many possibilities to manipulate text.

See 7.10 in VtC for more, also for pointers to pointers.

2-D Character Array (Array of Arrays)

```
char planets[][8] = {“Mercury”, “Venus”, ”Earth”, “Mars”, “Jupiter”};
```

This is how the array is stored

	0	1	2	3	4	5	6	7
0	M	e	r	c	u	r	y	\0
1	V	e	n	u	s	\0	\0	\0
2	E	a	r	t	h	\0	\0	\0
3	M	a	r	s	\0	\0	\0	\0
4	J	u	p	i	t	e	r	\0

2-D Character Array (Array of Arrays)

```
char planets[][8] = {“Mercury”, “Venus”, ”Earth”, “Mars”, “Jupiter”};
```

This is how the array is stored

	0	1	2	3	4	5	6	7
0	M	e	r	c	u	r	y	\0
1	V	e	n	u	s	\0	\0	\0
2	E	a	r	t	h	\0	\0	\0
3	M	a	r	s	\0	\0	\0	\0
4	J	u	p	i	t	e	r	\0

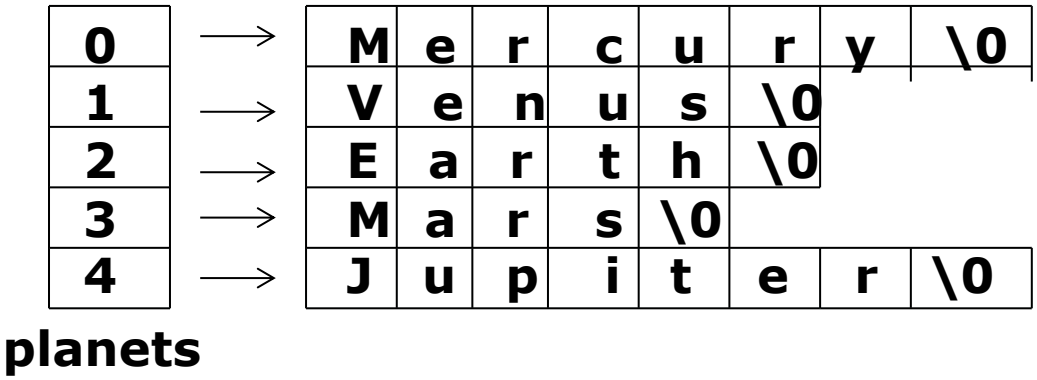
“null-character padding”: a waste of memory

A better way: array of pointers!

Array of Pointers

```
char *planets[] = {“Mercury”, “Venus”, ”Earth”, “Mars”, “Jupiter”};
```

This is how the array of pointers works



Without a lot of change this saves a lot of storage space and is widely used.

Variable Length Arrays in C99

```
# include <stdio.h>          // use a variable length array to reverse numbers
int main(void){
        int i, n;
        printf("How many numbers do you want to reverse?");
        scanf("%d", &n);
        int a[n];              // array length determined at runtime
                               // => works only in C99 this way!
        printf("Enter %d numbers: ", n);
        for (i=0; i  < n; i++)
                scanf("%d", &a[i]);
        printf("In reverse order:");
        for (i = n -1; i >= 0; i--)
                printf("%d", a[i]);
        printf("\n");
        return 0;              // once the array length has been determined
}                              //  it remains fixed for the rest of its "lifetime".
```

Copy of a Text String

Example:

```
int text1 [20], text2 [20];
text1 = text2;              // ERROR – does not compile
*text1 = *text2;            // copies only the 1st element of text2
                            // into the 1st element of text1 => a loop over
                            // text1 AND text2 gives a copy

while (*text1++ = *text2++) // The way to do it! See 7.8 in VtC.
          ;
```

=> see also strcopy in VtC 10.8.1 – to copy and to append

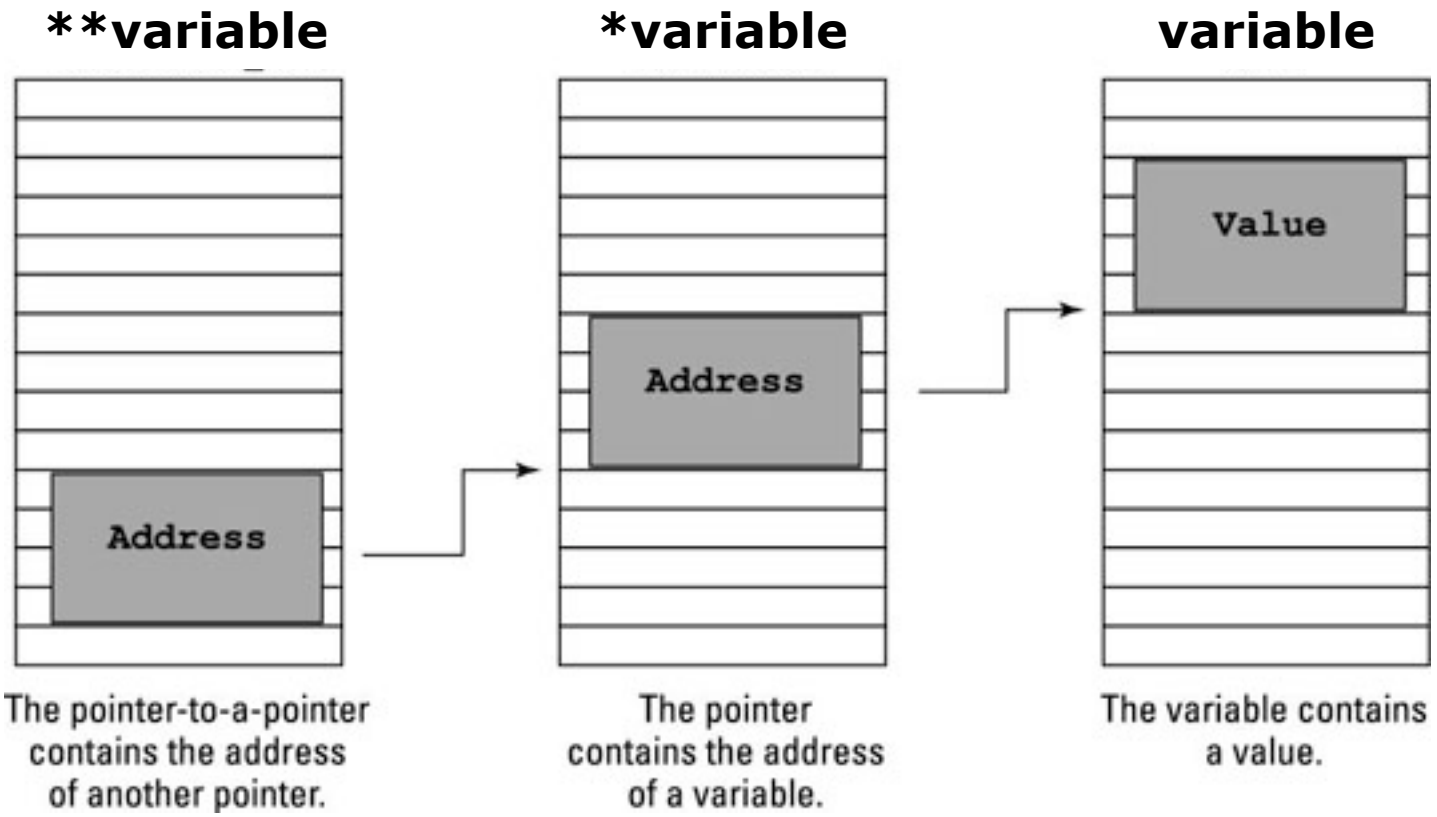
Call by reference

Pointer as arguments:
A function provides only one return value (arrays can also not be returned); pointers allow a way to circumvent this problem!
In C: "call by value" => function arguments are not changed

```
int i=8;
plus3(&i);        // function call with address of 'i' => "call by reference"
…
void plus3 (int *iref) // pointer *iref is an alias for i; allows to modify i
{
        *iref = *iref +3; // value iref and thereby also of i is increased
                          // this function does not need to return 11
}
```

The scanf() function works this way – by calling the variable address with "&" and modifying it.

Pointers to Pointers



An address is stored in memory => it needs memory space and has itself an address, to which a pointer can point.

Exercise: fix this program

```
#include <stdio.h>

void go_south_east (int lat, int lon)
{
        lat = lat -1;
        lon = lon +1;
}

int main(void)
{       int latitude = 32;
        int longitude = -64;
        go_south_east(latitude, longitude);
        printf("Now at: [%i, %i]\n", latitude, longitude);
        return 0;
}
```

Exercise: fix this progam

```
#include <stdio.h>

void go_south_east (int *lat, int *lon) // arguments need to be pointers
{
        *lat = *lat -1;                 // modify values at the given
        *lon = *lon +1;                 // addresses
}

int main(void)
{       int latitude = 32;
        int longitude = -64;
        go_south_east(&latitude, &longitude);     // pass addresses
        printf("Now at: [%i, %i]\n", latitude, longitude);
        return 0;
}
```

More about Pointers

Pointers and constants (see chapter 7.6 in VtC)

```
const float *pf; // pf points to a constant float value

float *const pt;  // pt is a const pointer; address cannot be changed

const float *const ptr; // pointer and variable value are constant
```

What is the use of it?

⇒ ***const*** **signals both programmer and compiler: "Don't modify!"**

⇒ **security and speed**

Null Pointers and Function Pointers

- **Null pointer**
  pointers point to "nowhere"; declared like this:
  char *nullpt=NULL;* // e.g. if you cannot allocate memory
  ***char *textstr=" ";* // != Null pointer but pointer to empty string!**

The NULL pointer is the return value if memory cannot be allocated!

- **Pointers to functions – an example for more advanced pointer applications:**
  call argument is a pointer to a (mathematical) function (used in GUIs in "callback functions" – functions which expect input of the user, e.g. a mouse click)

Function Pointers

Example for a pointer to a function:

```
int x;                          // declare an integer
double lut;                     // declare a double
double (*funcpt) (int);  /* pointer to function with int argument and
                              double return value.  */
double maxderiv (int);   // function declaration
funcpt = maxderiv; // our pointer points to function
                   // maxderiv: without "()" == pointer to maxderiv
lut = funcpt(x);        // call maxderiv via pointer, type has to match
```

See VtC, 7.13 for possible combinations of arrays, pointers and functions.

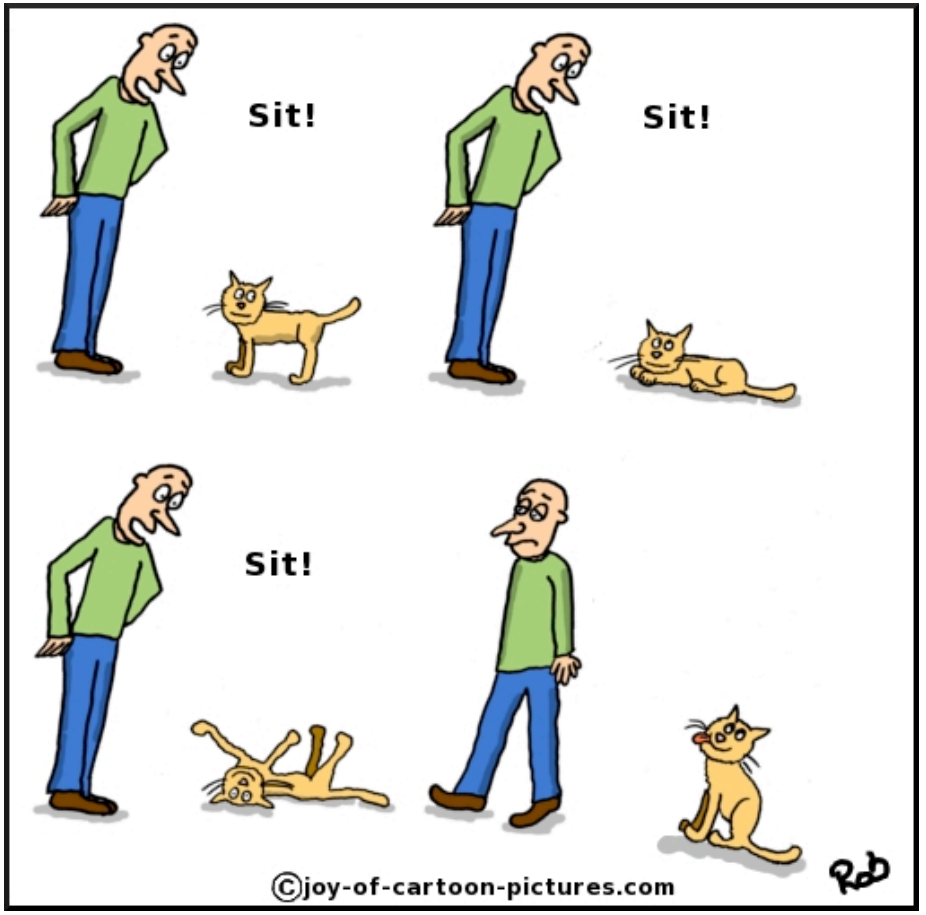
Exercise

Explain to your neighbor:

What is the connection between pointers and arrays?

What are function pointers used for?

Why do we make such a big fuss about pointers?



Dynamic Memory Allocation (and Pointers)

- Pointers are needed for dynamic memory allocation (see VtC, chapter 7.11)
  - we want to allocate memory without prior knowledge on how much is needed
- Standard functions *malloc*, *calloc*, *realloc* and *free*

```
#include <stdlib.h> // includes functions for memory allocation

double *dp;                                          // pointer is of type double
     dp = (double*) calloc(100,sizeof(double)); //points to the start of
                                                     // allocated memory
     ...
     free (dp);          // frees block of allocated memory
```

typecast: returns now pointer of type double

allocates/initializes 100 elements

each element has the size of a double

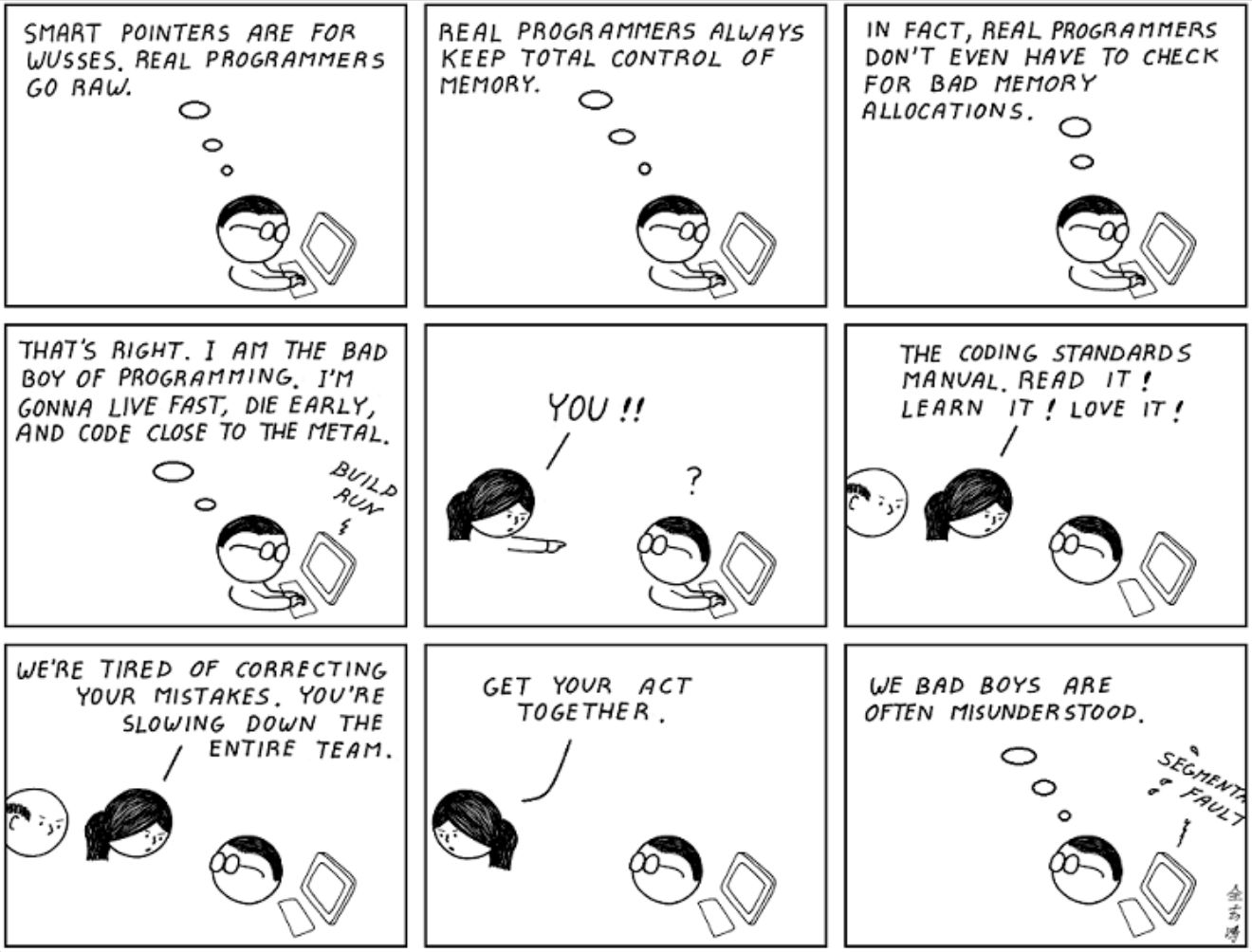
Memory Allocation: Example

```
/* dynamic_array.c -- dynamically allocated array */
#include <stdio.h>
#include <stdlib.h> /* for malloc(), free() */

int main(void){
  double *ptd;
  int max, number, i=0;
   printf("What is the maximum number of type double entries?");
  if (scanf("%d", &max) != 1)
  {
       printf("Number not correctly entered -- bye.");
       exit(EXIT_FAILURE);          // proper error handling, see next lecture
   }
  ptd = (double *) malloc(max * sizeof (double));
  if (ptd == NULL)
  {
         printf("Memory allocation failed. Goodbye.");
       exit(EXIT_FAILURE);          // proper error handling, see next lecture
   }                                // main is still open, cont. on next slide
```

Memory Allocation: Example cont.

```
  /* ptd now points to an array with max number of elements */
   printf("Enter the values (q to quit):");
   while (i < max && scanf("%lf", &ptd[i]) == 1)
       ++i;
   printf("Here are your %d entries:\n", number = i);
   for (i = 0; i < number; i++)
   {
          printf("%7.2f", ptd[i]);
          if (i%7 == 6)                  // nice output
                 putchar('\n');          // new line
    }
    if (i%7 != 0)
          putchar('\n');                 //new line
  printf("Done.");
  free(ptd);                             // free allocated memory
  return 0;
}                                        //end of main(void) function
```

Programmers responsibility



Structured Variables: *struct*

- **array: values of the same type**
- **struct: values of different type inside a new type**
- **flexible type that can be adjusted to the problem ("primitive class")Example:**
- ***struct car{***
  char owner[40], regnumber[6], brand [15];
  int age, price;
  } pb1, pb2;
- **pb1 and pb2 are of type struct car (compare with enum), and have 5 elements each**

Memory space:
(Note: elements need different amounts of memory)

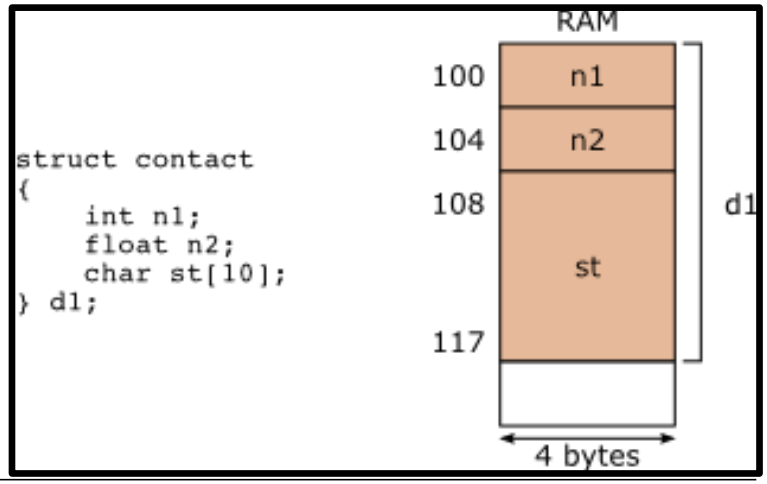
owner[40]

regnumber[6]

brand[15]

age

price



Structured Variables: *struct*

- **syntax**

```
struct car{
        char owner[40], regnumber[6], brand [15];
        int age, price;
       } pb1, pb2;
```

- **access to elements with the period (point) operator:**

```
pb1.regnumber="DRC552";
```

- ***pb1* and *pb2* are real variables:**

```
pb1 = pb2; // full copy is created by assignment
```

- ***pb1.brand* is a pointer to the first element of the array, i.e. arrays inside the struct are "normal" arrays.**

- **Note: Functions cannot return an array but they can return a struct!**

Structured Variables: *union*

```
union car{
            char name[40], regnumber [6], brand [15];
            int year, price;
           } pb1;

// pb1 can hold only one of the above elements, accessed by
// the point operator

pb1.name = “John Q”;
pb1.price = 2000; // pb1.name is corrupted by this statement!!
```

You can save one or the other member…
Application of unions?

Structured Variables: *union*

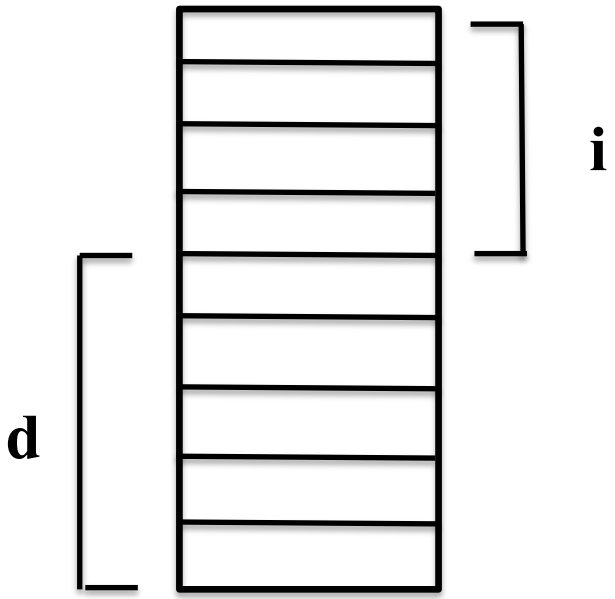
```
union car{
            char name[40], regnumber [6], brand [15];
            int year, price;
           } pb1;

// pb1 can hold only one of the above elements, accessed by
// the point operator

pb1.name = “John Q”;
pb1.price = 2000; // pb1.name is corrupted by this statement!!
```

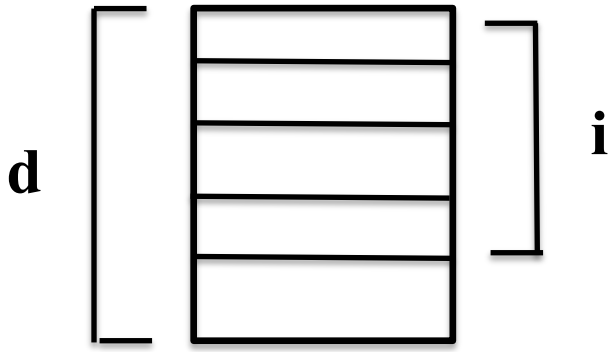
You can save one or the other member…
Application of unions? It saves memory space and works if you have one variable that needs to have different types.

Structured Variables: *union*

```
struct y{
        int i;
        double d;
        } v;
```

```
union x {
        int i;
        double d;
         } u;
```





=> i and u are stored at the same space!

Summary of Lecture 5

- **Pointers, pointers and arrays, pointers and text strings**
- **Dynamic memory allocation with malloc() or calloc()**
- **Structs and unions**