Programming Fundamentals

- Lecturer: Dr. Arnold Meijster
- e-mail: a.meijster@rug.nl
- room: 5161.343 (Bernoulliborg)

```
void printArr(int length, int a[]) {
   printf("%d", a[0]);
   for (int i=1; i < length; i++) {
     printf(", %d", a[i]);
   }
   printf("\n");
 }

 void increment1(int a) {
   a++;
 }

void increment2(int a[]) {
   a[0]++;
 }

void increment3(int length, int a[]) {
   for (int i=0; i < length; i++) {
     a[i]++;
   }
 }
```

```
int a[] = {0,1,2,3,4,5,6};
int x=1;

printf("%d\n", x);
increment1(x);
printf("%d\n", x);
printArr(7, a);
increment1(a[0]);
printArr(7, a);
increment2(a);
printArr(7, a);
increment3(7, a);
printArr(7, a);
```

```
1
1
0 1 2 3 4 5 6
0 1 2 3 4 5 6
1 1 2 3 4 5 6
2 2 3 4 5 6 7
```

Swapping rows in a matrix

```
void swapArrays(int length, int a[], int b[]) {
  for (int i=0; i < length; i++) {
    int h = a[i];
    a[i] = b[i];
    b[i] = h;
  }
}

void swapRows(int a, int b, int matrix[3][4]) {
  swapArrays(4, matrix[a], matrix[b]);
}

int a[3][4] = {
   { 1,  0, 12, -1},
   { 7, -3,  2,  5},
   {-5, -2,  2,  9}
};
```

We can swap *the contents* of rows 0 and 2 using: `swapRows(0, 2, a);`

Note that we can not swap columns this way! This is the result of the storage scheme that is used in C: 2D-arrays are stored row-wise in memory.

Sudoku checker

Check whether a matrix of 9x9 numbers is a completed sudoku.

```
int sudoku[9][9] = {
   {7, 5, 1, 8, 4, 3, 9, 2, 6},
   {8, 9, 3, 6, 2, 5, 1, 7, 4},
   {6, 4, 2, 1, 7, 9, 5, 8, 3},
   {4, 2, 5, 3, 1, 6, 7, 9, 8},
   {1, 7, 6, 9, 8, 2, 3, 4, 5},
   {9, 3, 8, 7, 5, 4, 6, 1, 2},
   {3, 6, 4, 2, 9, 7, 8, 5, 1},
   {2, 8, 9, 5, 3, 1, 4, 6, 7},
   {5, 1, 7, 4, 6, 8, 2, 3, 9}
};
```

Sudoku checker

```
int isCorrectSudoku(int sudoku[9][9]) {
  for (int r = 0; r < 9; r++) {
    for (int c = 0; c < 9; c++) {
      int digit = sudoku[r][c];
      if ((digit < 1) || (digit > 9) ||
          (cntRow(r, digit, sudoku) != 1) ||
          (cntColumn(c, digit, sudoku) != 1) ||
          (cntBlock(r, c, digit, sudoku) != 1)) {
         return 0;  /* FALSE */
      }
    }
  }
  return 1;  /* TRUE */
}
```

Sudoku checker

```
int cntRow(int row, int digit, int sudoku[9][9]) {
  int cnt = 0;
  for (int column = 0; column < 9; column++) {
    cnt += (sudoku[row][column] == digit);
  }
  return cnt;
}

int cntColumn(int column, int digit, int sudoku[9][9]) {
  int cnt = 0;
  for (int row = 0; row < 9; row++) {
    cnt += (sudoku[row][column] == digit);
  }
  return cnt;
}
```

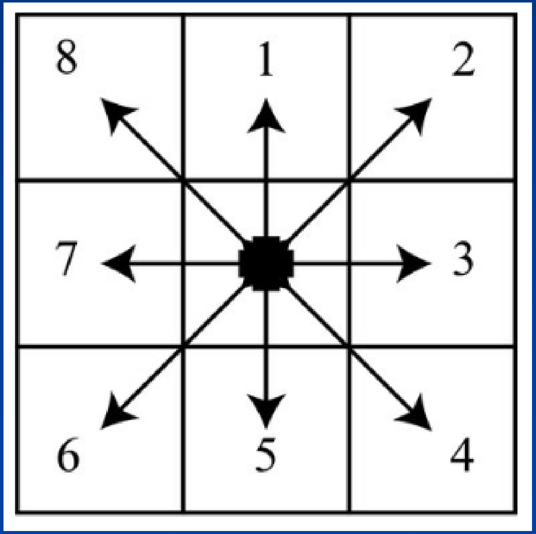
Sudoku checker

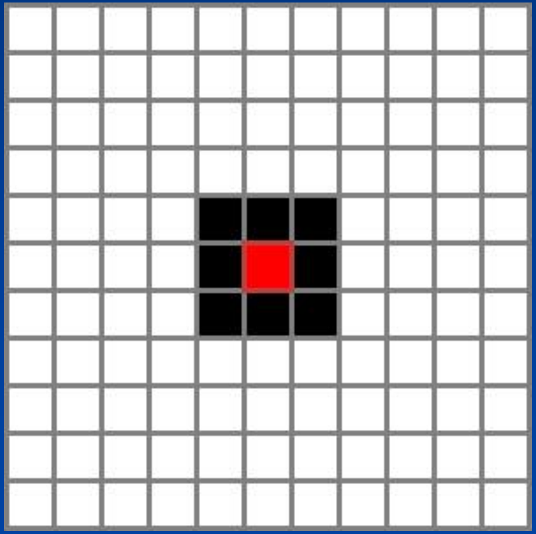
```
int cntBlock(int row, int column, int digit, int sudoku[9][9]) {
  int cnt = 0;
  int r0, c0; /* upper left corner of block */
  r0 = row - row%3;         /* or r0 = 3*(row/3);    */
  c0 = column - column%3;   /* or c0 = 3*(column/3); */
  for (row = r0; row < r0 + 3; row++) {
    for (column = c0; column < c0 + 3; column++) {
      cnt += (sudoku[row][column] == digit);
    }
  }
  return cnt;
}
```

John Conway's Game of Life

Each cell has 8 neighbors

- 4 adjacent orthogonally
- 4 adjacent diagonally.
- This is called 8-connectivity / chesboard neighborhood.





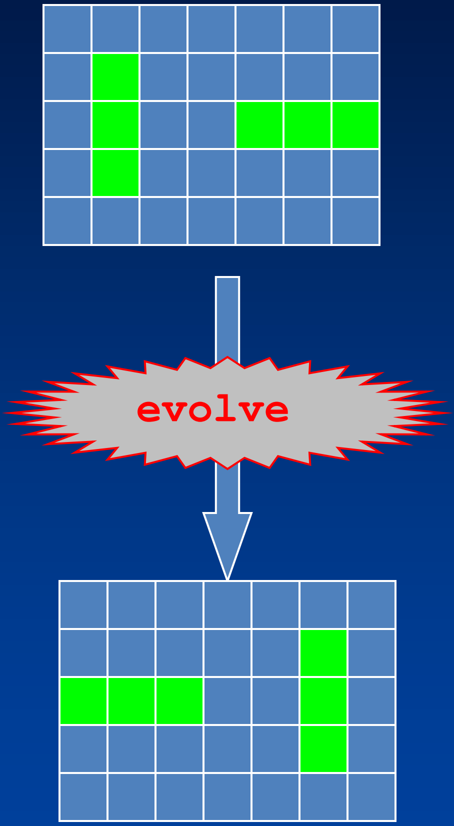
Game of Life: Simple rules, executed at each time step:

A living cell with 2 or 3 alive neighbors survives to the next round.

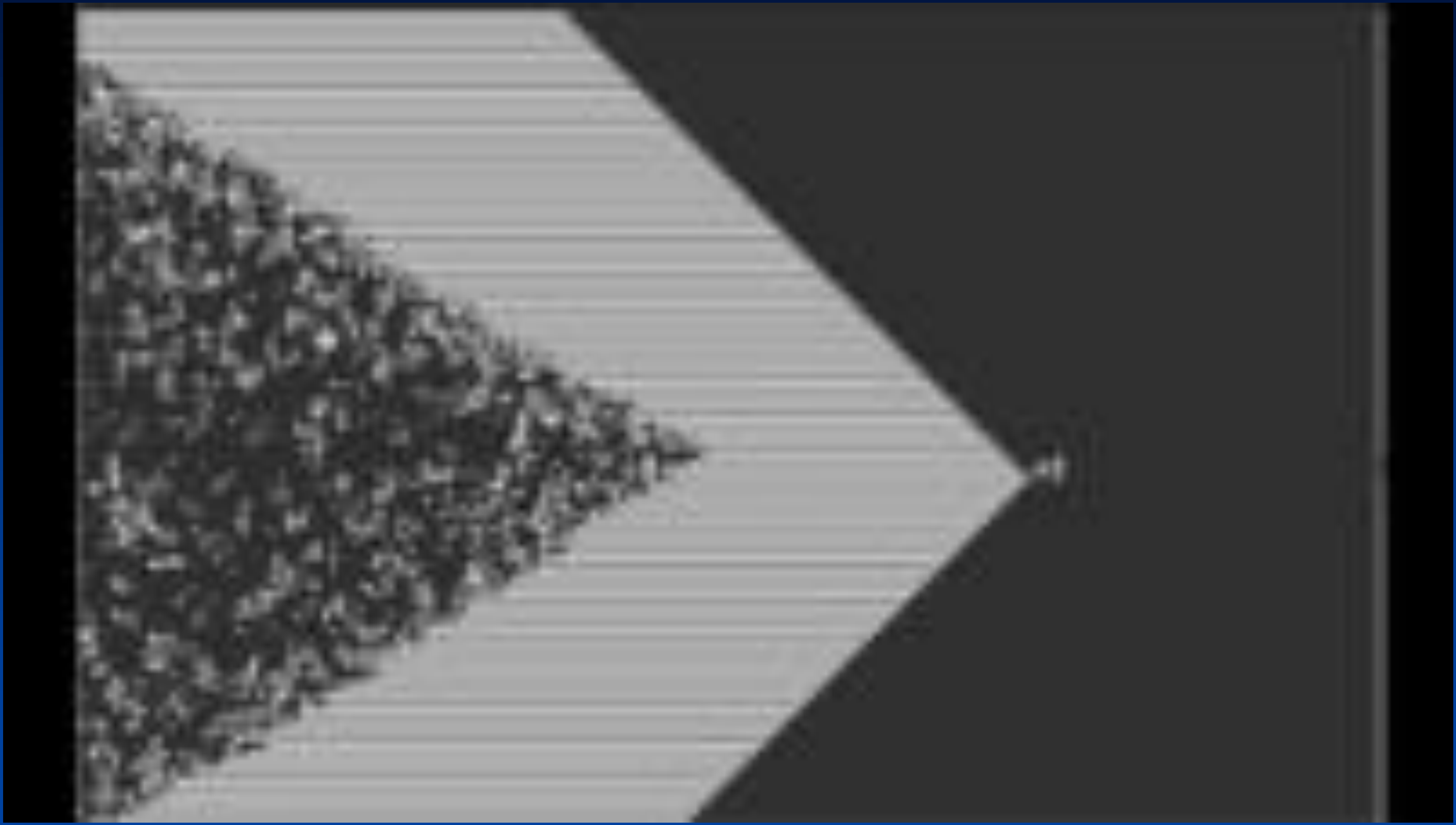
A living cell with 4 or more neighbors dies of overpopulation.

A living cell with 1 or 0 neighbors dies of isolation.

An empty cell with exactly 3 neighbors becomes a living cell in the next round.



Game of Life: video



John Conway's Game of Life

```
for (int row=0; row < n; row++) {
  for (int column=0; column < n; column++) {
    int neighbors = 0;
    for (int r=-1; r <= 1; r++) {
      for (int c=-1; c <= 1; c++) {
        if ((0 <= row + r) && (row + r < n) &&
             (0 <= column + c) && (column + c < n)) {
          neighbors += grid[row+r][column+c];
        }
      }
    }
    / * correction */
    neighbors -= grid[row][column];  /* subtract 1 if alive */
    /* compute next[row][column] */
    if (grid[row][column] == 0) {
      next[row][column] = (neighbors == 3);
    } else {
      next[row][column] = ((neighbors == 2) || (neighbors == 3));
    }
  }
}
```

Grades

Assignment: write a program fragment that reads the grades of a number of students from the input and then prints a list with the grade of every student and the deviation relative to the mean grade.

Grades: plan

We number the students, starting with 0.

We ask the user to first type the number of students.

We create an array that has the right size. ???

Read the grades from the input.

Compute the mean.

Print the list with grades and deviations.

WARNING !!!! NO VLAs!

A **variable-length array** (**VLA**) is an array data structure whose length is determined at runtime, instead of at compile time.

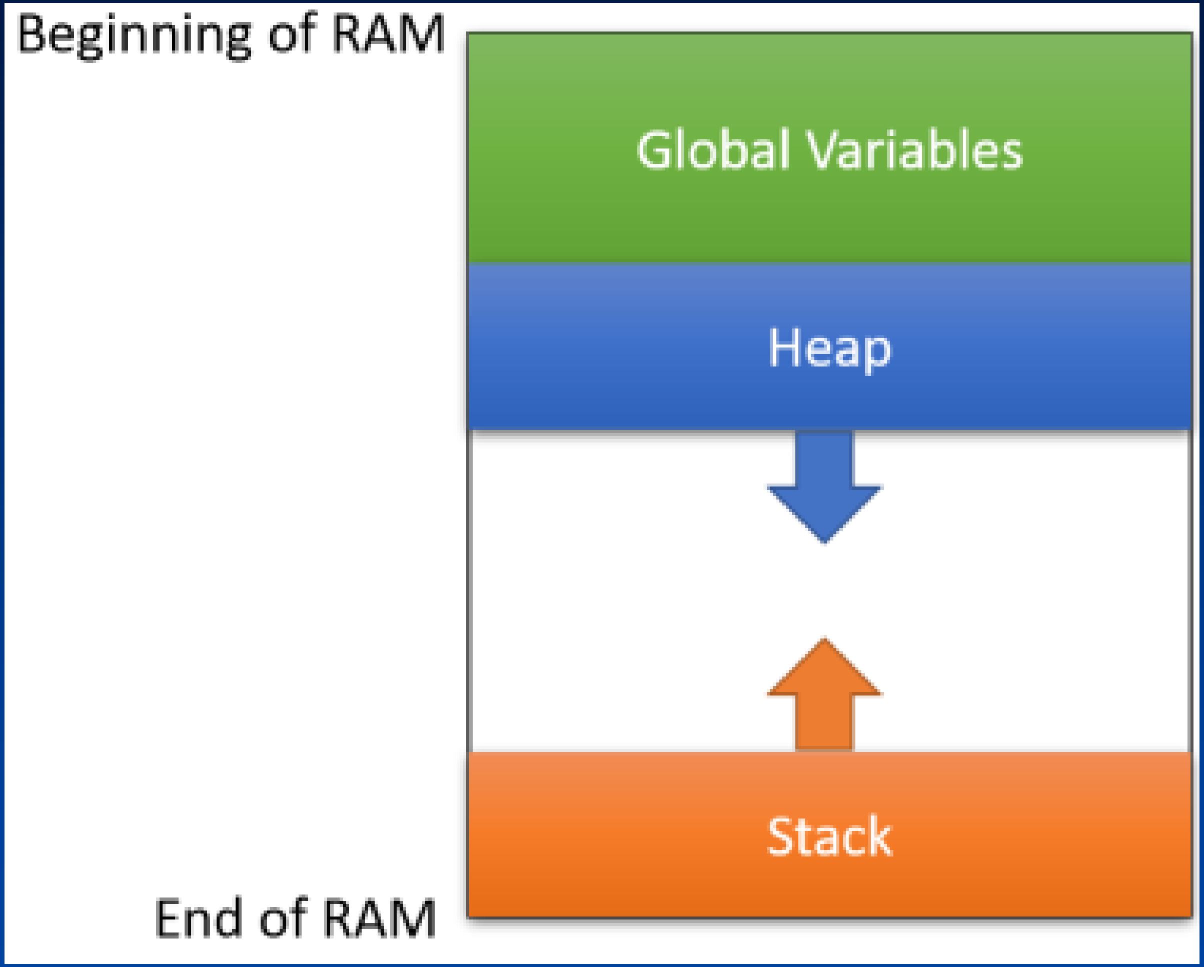
```
int n;
scanf("%d", &n);
int vla[n]; // DO NOT DO THIS!
```

C implementations are not required to support VLAs.

Therefore, never use them!

We subtract grade points in case you do!

Intermezzo: Stack and Heap Memory



Intermezzo: Stack and Heap Memory

Stack (memory): *implicit* memory usage taken care of by the compiler.

Used for function calls, parameters and return values of functions.

Local variables are also stored on the stack.

The compiler is responsible for memory management.

Deallocation (‘clean up’) is automatic!

Intermezzo: Stack and Heap Memory

Heap memory: *explicit* memory usage by the programmer.

The programmer is responsible for heap memory management.

Sloppy memory management leads to ***memory leakage.***

Deallocation (‘clean up’) is not automatic!

`void` and `NULL`-pointer

`void *p;` declares a *generic*, or *typeless* pointer `p`.

Commonly called a void-pointer.

Before usage, a void-pointer is cast to a typed pointer: `(T *)p`

For example: `int x = *((int *)p);`

A void-pointer can be assigned to a typed pointer without casting, and vice versa.

For example: `int *iptr = p;`

The NULL-pointer is a pointer to “any” data type.

It is called a *generic* pointer.

Its type is `void *`.

It can be assigned to any pointer, regardless its type.

`NULL` is actually defined as `(void *)0`.

Generic Pointers

```
int i = 2;
int *ip = &i;
void *p;


p = ip;
printf("%d", *p);  /* wrong */


printf("%d", *((int*)p));  /* ok */
```

Heap Memory Management

There are 2 functions that can be used for ***dynamic memory allocation***:

```
void *malloc(int requestedSize);
void *calloc(int requestedCount, int requestedSize);
```

```
T *p;

p = malloc(sizeof(T));     /* or */
p = calloc(1, sizeof(T));
```

Always check whether allocation was successful or failed. If the allocation failed, then the returned value is NULL!

Safe memory allocation

```
void *safeMalloc(int n) {
  void *p = malloc(n);
  if (p == NULL) {
    printf("Error: malloc(%d) failed. Out of memory?\n", n);
    exit(EXIT_FAILURE);
  }
  return p;
}
```

```
int *p = safeMalloc(sizeof(int));
*p = 42;
```

Memory Deallocation: ‘Clean Up’

Dynamically allocated memory must be returned (i.e. deallocated) when it is no longer needed.

```
free(p);
/* Note: p has not changed! (call-by-value, remember?)
 * But, you cannot use *p any longer!
 */
```

Pointer Arithmetic

You can perform (simple) arithmetic with pointers:

- *sum* of a pointer and an integer
- *difference* between a pointer and and integer
- pointer *comparison*

Sum of a pointer and an integer

```
int *p = safeMalloc(100*sizeof(int));
```

The sum `p + n` (where `n` is an integer) has the same type as `p`.

So, an int-pointer

We can produce a pointer to the `n`th element of a memory block by simply adding `n` to the pointer `p`, i.e. `p + n`.

This pointer points `n` elements (not bytes!) further than `p`.

So, do not write `p + n*sizeof(int)`

The operation (+) is not defined for void-pointers!

Sum of a pointer and an integer

So, we can address the `n`th element of a memory block using

```
    *(p + n)
```

This is the same as (preferred notation):

```
        p[n]
```

Hence, `p[0]` and `*(p+0)` and `*p` are equivalent.

Hence, `p[i]` and `*(p+i)` and `i[p]` are all equivalent.

Hence, `&p[i]` and `p+i` are equivalent.

Arrays and Pointers

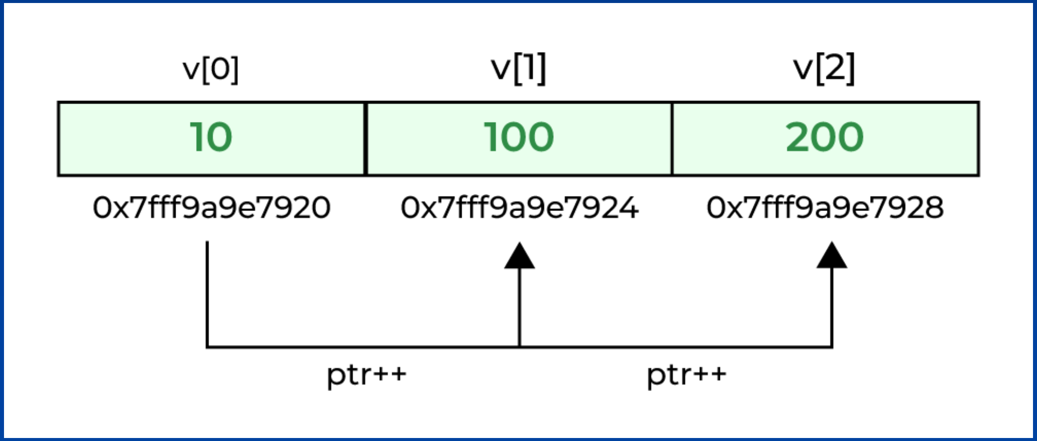
An array is actually a *typed constant* pointer that is initialised as a pointer that points to a memory block of exactly the right size.

```
int v[3];
int *ptr = &v[0];
```

v is a (constant!) **int**-pointer that points to a memory block that is reserved for 3 integers.

ptr is a (non-constant!) pointer to an **int**.

```
ptr = v;   /* ok */
v = ptr;   /* wrong */
```



Pointer Comparison

Two pointers of the same type, `p` and `q`, can be compared.

The comparison operators are: `<` , `>` , `<=` , `==` , `>=` , `!=`

BEWARE: if you compare two pointers, then you compare memory addresses (NOT the content of the addresses)!

```
int main () {
  int *p, *q, *r;
  p = safeMalloc(3*sizeof(int));
  q = safeMalloc(3*sizeof(int));

  *p = 0;
  *(p+1) = 1;
  p[2] = 2;

  printf ("p: %d %d %d\n", p[0], p[1], p[2]);

  r = p+3;
  while (p!=r) {
    *q++ = *p++;
  }
  q -= 3;
  p -= 3;

  printf ("p: %d %d %d\n", p[0], p[1], p[2]);
  printf ("q: %d %d %d\n", q[0], q[1], q[2]);
  return 0;
}
```

```
p: 0 1 2
p: 0 1 2
q: 0 1 2
```

Dangling References

Stack-data are deallocated automatically as soon as a function terminates.

So, NEVER return a pointer to a local variable!

```
int *f() {
   int i = 2;
   int *ptr = &i;
   printf("ptr points to %d\n", *ptr);  /* ok */
   return ptr;  /* DO NOT DO THIS! */
}

int main(int argc, char *argv[]) {
  int *ptr = f();
  printf("ptr points to %d\n", *ptr); /* wrong!!! */
  return 0;
}
```

Operations on Memory Blocks

Operations on blocks of memory use *generic (void)* pointers and are declared in the standard header file `string.h`.

A memory block can be resized to `len` bytes using:

```
void *realloc(void *ptr, int len);
```

A memory block of `len` bytes can be copied from `src` to `dest` (blocks should not overlap) using:

```
void *memcpy(void *dest, const void *src, int len);
```

A memory block of `len` bytes can be completely filled with a certain byte value using:

```
 void *memset(void *s, int c, size_t n);
```

For example: `memset(s, 0, 100*sizeof(int));`

End of intermezzo

Grades: plan

- Assignment: write a program fragment that reads the grades of a number of students from the input and then prints a list with the grade of every student and the deviation relative to the mean grade.

- We number the students, starting with 0.
- We ask the user to first type the number of students.
- We create an array that has the right size.
- Read the grades from the input.
- Compute the mean.
- Print the list with grades and deviations.

Grades

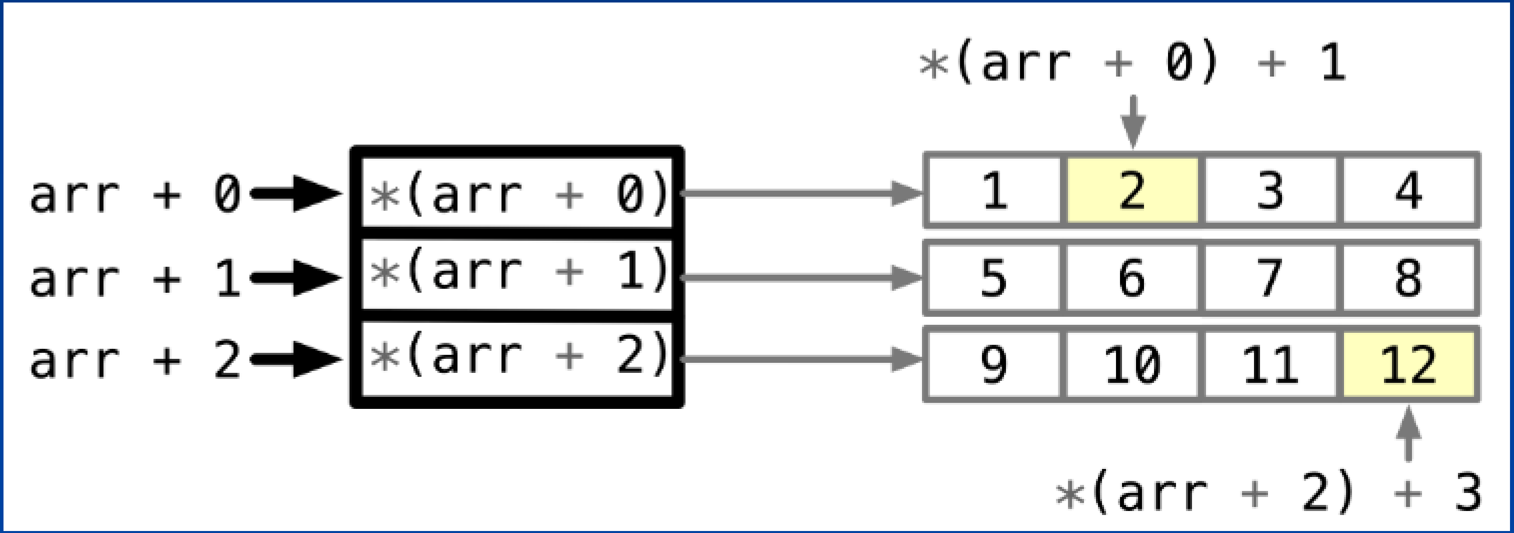
```
int numStudents;
printf("Number of students: ");
scanf("%d", &numStudents);

float *grade = safeMalloc(numStudents*sizeof(float));

for (int s=0; s < numStudents; s++) {
  printf("Grade of student %d: ", s);
  scanf("%f", &grade[s]);
}

float sum = 0.0;
for (int s=0; s < numStudents; s++) {
  sum += grade[s];
}

float mean = sum/numStudents;

print("student\tgrade\tdeviation\n");
for (int s=0; s < numStudents; s++) {
  printf("%d\t%.1f\t%.1f\n", s, grade[s], grade[s] - mean);
}
```

Grades: merge loops

```
int numStudents;
printf("Number of students: ");
scanf("%d", &numStudents);

float *grade = safeMalloc(numStudents*sizeof(float));

float sum = 0.0;
for (int s=0; s < numStudents; s++) {
  printf("Grade of student %d: ", s);
  scanf("%f", &grade[s]);
  sum += grade[s];
}

float mean = sum/numStudents;

print("student\tgrade\tdeviation\n");
for (int s=0; s < numStudents; s++) {
  printf("%d\t%.1f\t%.1f\n", s, grade[i], grade[i] - mean);
}
```

Making a dynamic 2D-array

```
int **makeIntArray2D(int width, int height) {
  int **arr = safeMalloc(height*sizeof(int *));
  for (int row=0; row < height; row++) {
    arr[row] = safeMalloc(width*sizeof(int));
  }
  return arr;
}
```



Freeing a dynamic 2D-array

```
void destroyIntArray2D(int **arr) {
  for (int row=0; row < height; row++) {
    free(arr[row]);
  }
  free(arr);
}
```

End week 5