Real Power of C

“A C program is like a fast dance on a newly waxed dance floor by people carrying razors.”

Waldi Ravens.
Advanced Data Types in C

- pointer – addressing arbitrary object in memory
- array – ordered set of elements of same type
- structure – multiple items in single data type
- custom datatypes – typedef command
- enumerators, unions
Pointers

- variable that store address of another variable, structure, function, ....
- typed: "pointer to the type XXX"
- but may be type-casted
- size is machine-dependent

```c
char c = 'A';
char *cptr;
cptr = &c;
```
Pointers II.

- reference op &: &var denotes address of variable var
- dereference op *: *ptr denotes value of variable placed on address ptr

```c
int i = 1, j;  // integer
int *pi;       // pointer to integer
p_i = &i;      // store address of i to p_i
*p_i = 5;      // same as i = 5;
```

Note: asterisk in pointer definition – do not mix with dereferene.
Pointers and Functions

- both arguments of function or return value can be a pointer
- useful for *call by reference*
- function may modify the argument

```c
int *max(int *a, int *b){
    //returns pointer, not the value
    return (*a > *b ? a : b);
}
```
Pointers TO Functions

- address of memory where function is located
- no parameter checking!

```c
// functions
int print1(int);  
int print2(int);

// declaration
int (*pif)();  
int a;

// assignment
pif = print1;  
(*pif)(1);  
pif(a);

pif = print2;  
pif(a);  
pif(a,1,2); // extra arguments ommited
pif(); // undefined value in parametes
```
**Arrays**

- homogenous datatype, set of variables of same kind
- ordered, indexed from zero
- array bounds not checked by compiler
- array elements stored linearly in memory

```c
int a[10]; // define array of 10 elements
a[1] = 6; // assign value to second element
    // (first is a[0])
a[10] = -1; // bad bad bad bad bad bad bad
```

Note: array overrun is usual cause of errors ... and freezes ... and resets ... and security compromises ... etc.
Arrays II.

Initialization:

```c
int prime[3] = {5, 7, 11, 13}; // error -- more elements
char letters[5] = {'a', 'h', 'o', 'j', '!'};
```

Note: array is NOT a primary datatype, following assignment is not correct:

```c
int tmp[10];
tmp = happy; // bad
```
Arrays and Pointers

- arrays and pointers are closely related
- for int arr[10];, arr is constant pointer to beginning of memory segment

```c
int i, happy[3] = {7, 13, 25};
int *ptr;

ptr = happy;

for (i = 0; i < 3; i++){
  cool_fn(ptr[i]);
  cool_fn(happy[i]);
}
```
Arrays as Function Arguments

- always passed by reference

```c
// equivalent prototypes
int max (int n, const int *arr);
int max (int n, const int arr[]);

// definition
int max(int n, int ...){
    int lmax = 0, i;
    for (i = 0; i < n; i++)
        if(arr[i] > lmax) lmax = arr[i];
    return lmax;
}

// programmer is responsible for boundary checking!!!
int pole[3] = {1, 3, 4};

// following statement lead to undefined result
printf("Max: %d", max(5, pole));
```
Pointer arithmetics

Pointers are de-facto numbers, following operation may have sense:

- comparison (==, !=, <, >): if (p1 == p2) {
- sum of pointer and integer: *(p + n)
- increments (++, --)
- difference (p1 - p2)
Comparing Pointers

- comparison of memory addresses
- have sense only for pointers to same memory area – e.g. single array
- only pointers of same type, except NULL
Assume that:

```c
int *pc, data[10] = ...  
pc = data;
```

Then it holds:

```c
*pc = pc[0] = data[0]  
*(pc + 1) = data[1]    
*(pc + n) = data[n]
```

- *(pc + n) points to n-th element (of type pc) after pc
Pointer arithmetics – addressing

Assume, that sizeof(char) == 1, sizeof(int) == 2, sizeof(float) == 4

char *pc = 10; int *pi = 10; float *pf = 10;

Then it holds:

p_c + 1 == 11
p_i + 1 == 12
p_f + 1 == 14
(char *) p_i + 1 == 11
(char *) p_f + 1 == 11
```c
#define ARRAY_SIZE 3
int i;
unsigned int arr1[ARRAY_SIZE] = {0xfeed, 0xdead, 0xbeef};
unsigned int *ptr_i;
unsigned char *ptr_c;

ptr_i = arr1; // ... = &(arr[0]) also possible
ptr_i = (char *)arr1;

// print integer values
for (i = 0; i < ARRAY_SIZE; i++) printf("0x%x ",ptr_i[i]);

// or
for (i = 0; i < ARRAY_SIZE; i++,ptr_i++) printf("0x%x ",*ptr_i);

// print individual characters
for (i = 0; i < ARRAY_SIZE * sizeof(int); i++) printf("0x%x ",ptr_c[i]);
```
Strings

- basically array of char
- terminated with null character (’\0’)
- literals enclosed in double-quotes: "string";

```c
char s1[10] = "Hello!"; //
char s2[] = "Howdy!";    //
char *ptrs;

s1 = "Bye!";            // not possible!
ptrs = s2;              // as w/ any other array
```
#include<ctype.h>

void str_to_upper(char *string){
    int i;
    char *ptr = string;

    // loop until '\0' character occurs
    while(*(ptr++)) // or while(*ptr != 0)
        *ptr = toupper (*ptr);
}

char retezec[] = "PokusnyRetezec c. #1";
str_to_upper(retezec);

printf("Converted string  %s\n",retezec);
Dynamic Allocation

```c
void *malloc (size_t size);
void free(void *ptr);

#include <stdlib.h>

// create array with powers for n = 1..10
unsigned char *p_c;
p_c = (unsigned char *) malloc (sizeof(char) * 10);

for (i = 0; i < 10; i++) p_c[i] = i + 1 * i + 1;
...
free ((void *) p_c);
```
Dynamic Allocation II.

```c
#include <stdlib.h>

// create array of structures
fellow_t *my_friends;
p_c = (fellow_t *) malloc (sizeof(fellow_t) * 10);
```
2D Arrays

Static definition:
int arrA[3][4];

- allocated in continuous memory block
- equivalent to  *(*(arrA + i) + j)
- fixed number of columns and row

Dynamic – pointer to array (rarely used):
int (*arrB)[4]; // pointer to array

- dynamically allocated in single block
  arrB[i] = (int *[4])malloc ( 4 * 2 * sizeof(int));

- fixed number of columns, arbitrary number of rows
2D Arrays II.

Dynamic – array of pointers:
int *arrC[3]; // pointer array

- dynamically allocated per line
  
  for (i = 0; i < 3; i++) arrC[i] = (int *)malloc ( 4 * sizeof(int));

- fixed number of rows of varying size

Dynamic – pointer to pointer:
int **arrD; // pointer to pointer

- dynamically allocated array for pointer to lines:
  
  arrD = (int **)malloc ( 4 * sizeof(int));

- then allocation per line:
  
  arrD[i] = (int *) malloc ( 3 * sizeof(int));