Yrkes Akademin
Vi hjälper dig att lyckas!

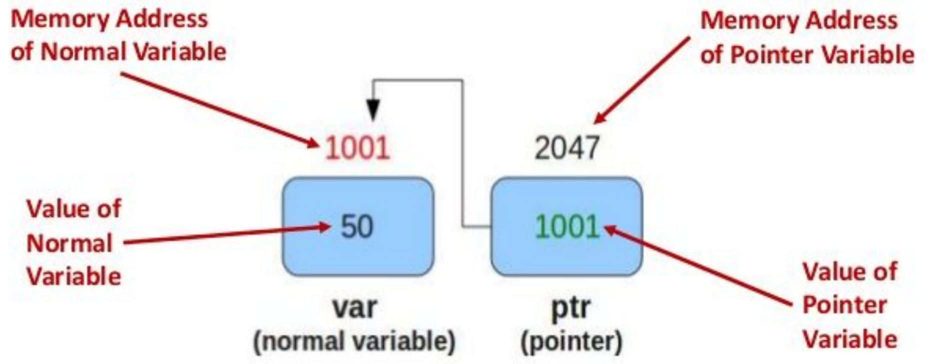
C Programming

Pointers

Pointers in C

- ❖ A pointer is a reference to a variable or a function.
  - ➢ A pointer to a variable can be declared using the * operator as

    type *pointer_name [= initializer];
  - ➢ E.g. int var = 50; int ***iptr** = &var; // iptr points to var
- ❖ A pointer basically ***holds an address*** and even ***has an address*** like a variable
  - ➢ E.g. printf("Address of var: %#p\nValue of iptr: %#p\nAddress of iptr: %#p\n", &var, iptr, &iptr),
  - ➢ So sizeof a pointer to any type is the same and depends on the architecture of the system
- ❖ Pointers are generally used to get indirectly access to variables and functions in order to improve performance and memory usage


Memory Address of Normal Variable
Memory Address of Pointer Variable
1001
2047
Value of Normal Variable
50
1001
Value of Pointer Variable
var (normal variable)
ptr (pointer)

Pointers in C

- Pointers can be dereferenced using the * operator
  - To get access to the variables which are the target of pointers.
  - E.g. `int var = 50; int *iptr = &var; *iptr = 10; printf("Value of var: %d\n",*iptr); // Now the value of var is 10`
- Pointers are the most powerful feature of C and they can be used for example
  - Implementation of **call by reference** functions
    - Without having pointers, variables are passed by values.
      - When we call functions, the value of variables are copied to the arguments of functions and inside functions we have copies of variables and we can not change the original variable.
  - Implementation of dynamic data structures like linked lists
  - Instead of moving around big data in the memory, address of data can be moved or copied

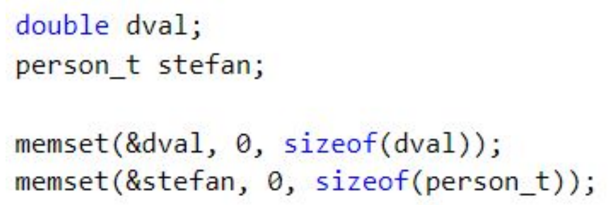
```c
#include <stdio.h>

static void func1(int x) { x = 5; }
static void func2(int *x) { *x = 20; }

int main(void) {
    int var = 2;

    func1(var); // A copy of var is passed to func1
    (void)printf("Value of var is %d\n", var); // var is 2

    func2(&var); // Address of var is passed to func2
    (void)printf("Value of var is %d\n", var); // var is 20

    return 0;
}
```

Pointers in C

- Pointers are the main source of hard to find bugs in C programs
- In C a null pointer constant macro has been defined as **#define NULL ((void *)0)**
- A null pointer points to nowhere and it cannot be dereferenced.
  - E.g. int *iptr = NULL; *iptr = 20; // Error. We can not dereference a null pointer
- A null pointer is always unequal to any valid pointer to an object or function
  - Functions that return a pointer type usually use NULL to indicate a failure condition
  - E.g. if(fgets(string, LENGTH, stdin) != NULL) { ... } // If fgets fails, it returns NULL
- void pointers in C (pointers to void)
  - A void pointer can be declared as void * and it is used as a general-purpose pointer
  - Means that a void pointer can point to any type except void. E.g. NULL is a void pointer
    - We know that we can not declare variables of type void

Pointers in C

- A void pointer can be converted to a pointer to any type and vice versa.
  - The compiler performs the conversion implicitly. E.g. float *fptr = NULL;
  - A void pointer points to data as a typeless data.
    - It can be used to declare a general type function parameter and return type
      - E.g. void *memset(void *s, int c, size_t n) which
      - Is used to set all bytes in a block of data to a value(**c**) and
      - It return a pointer to the block of data regardless of its type

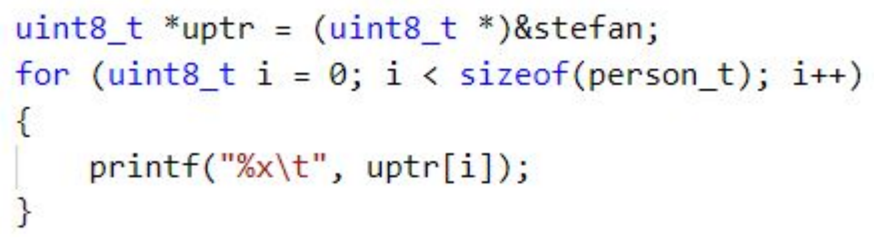


```
double dval;
person_t stefan;

memset(&dval, 0, sizeof(dval));
memset(&stefan, 0, sizeof(person_t));
```

- An uninitialized pointer(**wild pointer**) points to a random location in the memory
- A pointer shall be initialized under the general initialization rules in C
- We can use NULL to initialize a pointer to any type. E.g. float *fptr = NULL; int *iptr = NULL;
  - Instead of **0** as the initializer of a pointer, we shall use NULL as a null pointer.

Pointers in C

- ❖ A pointer can also be initialized using a pointer to the same type.
- ❖ A cast shall not be performed between object pointers of different types.
  - ➢ Exception: It is permitted to convert a pointer to an object type into a pointer to char, signed char or unsigned char.
    - ■ Pointers to these types can be used to access the individual bytes of objects.

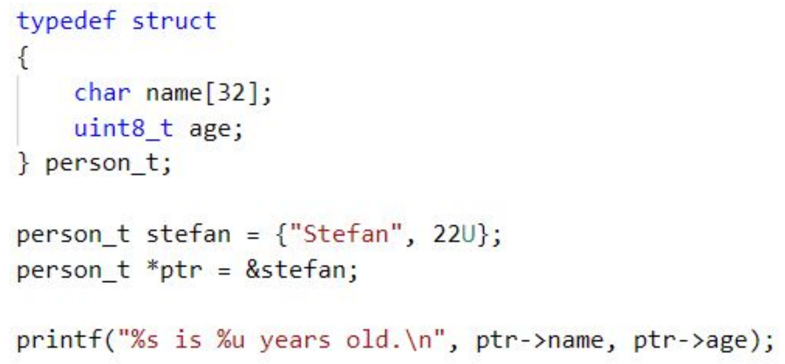


```c
uint8_t *uptr = (uint8_t *)&stefan;
for (uint8_t i = 0; i < sizeof(person_t); i++)
{
    printf("%x\t", uptr[i]);
}
```

- ❖ A pointer is a variable and has an address and it is possible to make a pointer to it
  - ➢ To declare a pointer to a pointer we use double asterisks (**)
  - ➢ E.g. char c = 'A'; char *cptr = &c; char **dcptr = &cptr; **dcptr = 'B'; // dcptr is a double pointer.
  - ➢ printf("c = %c, *cptr = %c, **dcptr = %c and &c = %p, cptr = %p, *dcptr = %p\n", c, *cptr, **dcptr, &c, cptr, *dcptr);
  - ➢ It is possible to have more than 2 levels of pointer nesting. But we shall avoid more than 2 levels

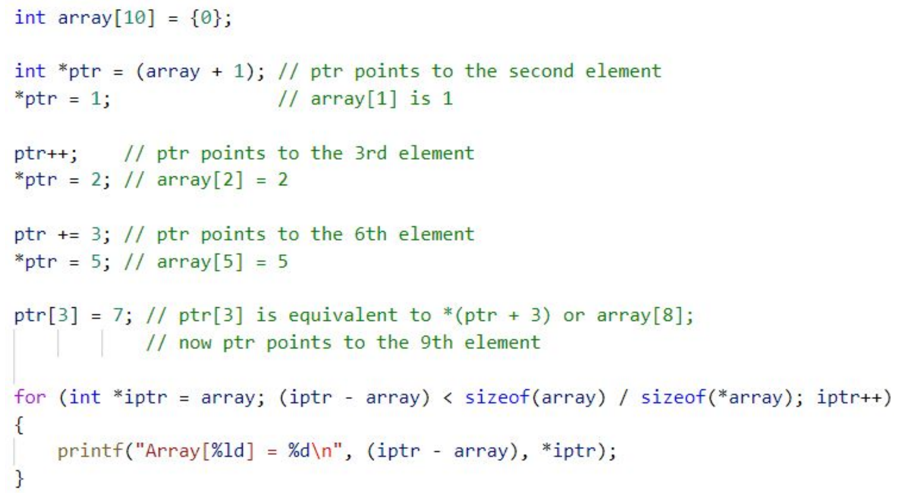
Pointers in C

- A pointer to a structure or a union
  - In two ways we can get access to the members
    - Using the dot(.) operator.
      - E.g. **`(*ptr).name`** // Parentheses are required
    - Using the arrow operator. E.g. **`ptr->name`**
    - They are equivalent.



```
typedef struct
{
    char name[32];
    uint8_t age;
} person_t;

person_t stefan = {"Stefan", 22U};
person_t *ptr = &stefan;

printf("%s is %u years old.\n", ptr->name, ptr->age);
```

- Pointer Comparing and Arithmetic Operations
  - Addition and subtraction of an integer.
  - Subtracting one pointer from another.
  - Comparing two pointers
  - **Pointers know size of the type.**
    - *In the case of addition and subtraction they are moved according to size of the data type*



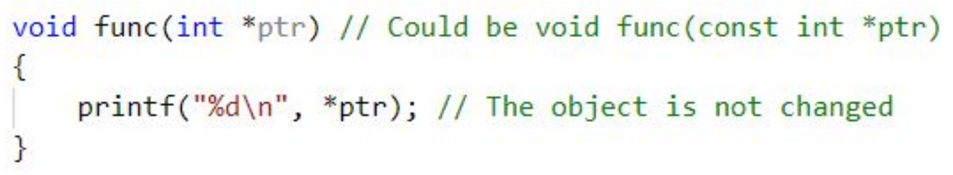
```
int array[10] = {0};

int *ptr = (array + 1); // ptr points to the second element
*ptr = 1;               // array[1] is 1

ptr++;    // ptr points to the 3rd element
*ptr = 2; // array[2] = 2

ptr += 3; // ptr points to the 6th element
*ptr = 5; // array[5] = 5

ptr[3] = 7; // ptr[3] is equivalent to *(ptr + 3) or array[8];
            // now ptr points to the 9th element

for (int *iptr = array; (iptr - array) < sizeof(array) / sizeof(*array); iptr++)
{
    printf("Array[%ld] = %d\n", (iptr - array), *iptr);
}
```

Pointers in C

- It is possible to compare two pointers using the ==, !=, <,>,<= and >= operators
- A pointer resulting from arithmetic operation shall address an element of the same array as that pointer operand. E.g. int arr[5] = {0}; int *ptr = &arr[0]; ptr--; // Not OK
- Subtraction between pointers shall only be applied to pointers that address elements of the same array. E.g. int a1[5] = {0}; int a2[8] = {0}; int *p1 = a1; int *p2 = a1; diff = p1 - p2; // Not OK
- The operators >, >=, < and <= shall not be applied to pointers that don't point the same object. E.g. int a1[5] = {0}; int a2[8] = {0}; int *p1 = a1; int *p2 = a1; if(p1 > p2) {...} // Not OK
- The +, -, += and -= operators should not be applied to an expression of pointer type
  - It is ok to use prefix and postfix ++ and --. E.g. int a1[5] = {0}; int *p = a1; p++; *p = 2;
  - Better to use array indexing. E.g. int a1[5] = {0}; int *p = a1; p[1] = 2;

Pointers in C

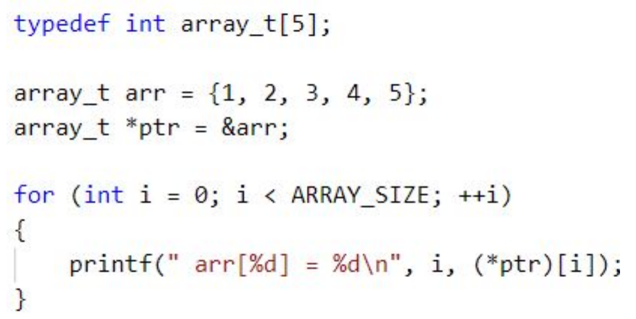
- The declaration of a pointer may contain the type qualifiers. E.g. int const volatile * ptr;
- The const and volatile qualifiers may qualify
  - Either the pointer type itself, or type of the object it points to.
    - E.g. A pointer to a const variable or a const pointer to a variable.
  - If type qualifiers occur between asterisk and the pointer name, they qualify the pointer itself.
    - E.g. int var = 20; int *const ptr = &var; // ptr is a const pointer to var
    - E.g. const int var = 20; const int *ptr = &var; // ptr is a pointer to a const
    - E.g. const int var = 20; const int *const ptr = &var; // ptr is a const pointer to a const
    - E.g. int *const ptr; int * const *p2cp = &ptr; // ptr is a pointer to a const pointer
- A pointer should point to a const-qualified type whenever possible
  - The same rule also shall be applied to arrays
    - An array is a pointer



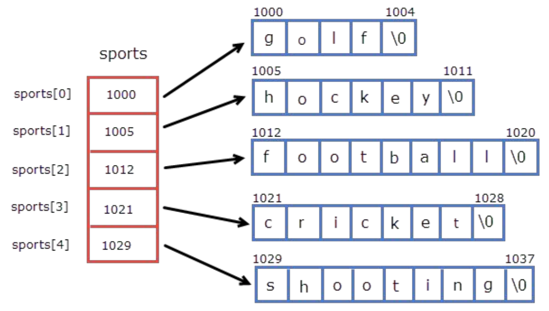
```
void func(int *ptr) // Could be void func(const int *ptr)
{
    printf("%d\n", *ptr); // The object is not changed
}
```

Pointers in C

- Name of an array is a pointer to the first element in the array.
  - E.g. int arr[5] = {0}; int *ptr = arr; ptr[0] == arr[0]; const char *str = "Hello World!";
- We can have a pointer to an array as a whole (array pointer)
  - E.g. int (*ptr)[5] = &arr; // ptr is a pointer to an array of five int elements. parentheses are required
  - Even it is possible to define types of an array pointer.
    - E.g. typedef int array_t[5]; array_t arr = {0}; array_t *ptr = &arr;
  - Array pointers are useful when we deal with multidimensional arrays.
    - E.g. Name of a two multidimensional array is a double pointer.
    - E.g. int matrix[2][3] = {0}; int (*ptr)[3] = matrix;
      - ***matrix** is a pointer which points to the first row.
      - **E.g. printf("maxtrix[0][0] = %d = %d = %d = %d\n",**matrix,**mptr,matrix[0][0],(*mptr)[0]);**



```
typedef int array_t[5];

array_t arr = {1, 2, 3, 4, 5};
array_t *ptr = &arr;

for (int i = 0; i < ARRAY_SIZE; ++i)
{
    printf(" arr[%d] = %d\n", i, (*ptr)[i]);
}
```



- We can also have an array of pointers (pointer array)
  - E.g. char *parr[5] = {0}; // parr is an array of 5 char pointers (strings)