

# Introduction to C

Pointers: basics

1

- <https://m.xkcd.com/138/>



2

# Pointers

- Access to values of variables in memory
- Method of passing parameters from/to functions.
  - Simulate call-by-reference
- Strong connection between arrays and pointers
- Create and manipulate dynamic data structures.
  - Creation of dynamic structures (Linked lists)

# Pointer

- A pointer is a variable that contains the address of another variable.
- When a variable is defined, it is allocated a portion of memory. The variable has a value and an address.

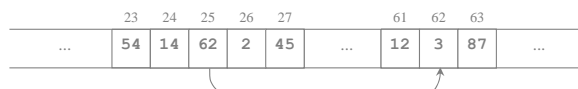
```
char x = 3;
```

- Assume **x** is stored at address 62.

- A pointer is a variable that holds the address of another variable.

```
char *px = &x;
```

- Assume **px** is stored at address 25. This variable points to **x**.



## Pointer Syntax

- Declaration with `*` means “is a pointer to”.
- The `&` operator means “address of”.

```
int i;  
int *j = &i;
```

- Spacing has no effect.

```
int* i, j, * k;  
int val = 45;  
k = &val;
```

- (Better?) style:

```
int * i, * k, j;
```

## Pointer Syntax

- Dereferencing or indirection operator `*`

```
int i = 2, x;  
int * j = &i;  
*j = 5; /* i now equals 5 */  
x = *j; /* x now equals i */
```

- Pointer dereferencing not to be confused with pointer declaration syntax. Difference is apparent from context.
- Tip style: when dereferencing, use no space

```

1 /*
2 pointer_1.c
3 http://gribblelab.org/cbootcamp/8_Pointers.html
4 */
5 #include <stdio.h>
6
7 int main ()
8 {
9     int age = 30;
10    int * p;
11
12    p = &age;
13
14    printf("age=%d\n", age);
15    printf("sizeof(age)=%ld\n", sizeof(age));
16
17    printf("p=%p\n", p);
18    printf("**p=%d\n", *p);
19    printf("sizeof(p)=%ld\n", sizeof(p));
20
21    *p = 40;
22    printf("**p=%d\n", *p);
23    printf("age=%d\n", age);
24
25    age = 50;
26    printf("**p=%d\n", *p);
27    printf("age=%d\n", age);
28
29
30    return 0;
31 }

```

```

frankvp@CRD-L-08004:~/Pointer$ gcc pointer_1.c -o pointer_1
frankvp@CRD-L-08004:~/Pointer$ ./pointer_1
age=30
sizeof(age)=4
p=0x7fff8203928c
*p=30
sizeof(p)=8
*p=40
age=40
*p=50
age=50
frankvp@CRD-L-08004:~/Pointer$

```

KU LEUVEN

7

```

1 #include <stdio.h>
2 // pointer_4.c
3 int main()
4 {
5
6     int m=3, n=100, * p;
7
8     p=&m;
9     printf("m is %d\n", *p);
10    m++;
11    printf("m is now %d\n", *p);
12    p=&n;
13    printf("n is %d\n", *p);
14    *p=500;
15    printf("n is now %d\n", n);
16
17 }

```

```

frankvp@CRD-L-08004:~/Pointer$ gcc pointer_4.c -o pointer_4
frankvp@CRD-L-08004:~/Pointer$ ./pointer_4
m is 3
m is now 4
n is 100
n is now 500
frankvp@CRD-L-08004:~/Pointer$

```

KU LEUVEN

8

## Pointer Syntax: NULL

- Bad practice to declare a pointer and not assign it to a valid object.
- Have a pointer that points to “nowhere”, make this explicit by assigning it to **NULL**.

It indicates that it does not point to a meaningful position.

```
double * pval1 = NULL;
```

```
double * pval2 = 0;
```

- The integer constants 0 and 0L are valid alternatives to **NULL**, but the symbolic constant is more readable.
- **NULL** is defined in `stdio.h`

## Pointers and Functions: Pass by Reference (simulation)

- Variables are *a/ways* passed to functions “by value”: a copy is passed.
- The following example won’t work. (`swap_by_value.c`)

```
1 /*
2 swap_by_value.c
3 Swap two variables - no pointers*/
4
5 #include <stdio.h>
6
7 void swap(double x, double y);
8
9
10 int main(){
11     double a = 5, b = 10;
12     printf("a = %f, b = %f\n", a, b);
13     swap(a, b);
14     printf("a = %f, b = %f\n", a, b);
15
16     return 0;
17 }
18
19
20
21
22 void swap(double x, double y)
23 {
24     double t = x;
25     x = y;
26     y = t;
27 }
```

```
frankvp@CRD-L-08004:~/Pointer$ gcc swap_by_value.c -o swap_by_value
frankvp@CRD-L-08004:~/Pointer$ ./swap_by_value
a = 5.000000, b = 10.000000
a = 5.000000, b = 10.000000
frankvp@CRD-L-08004:~/Pointer$
```

# Pass by Reference

- Desired effect achieved by passing pointers to the variables.
- Example `swap.c`
  - Pointers are passed “by value”, but these copies still hold addresses of original variables.

```
1 /*
2 swap.c
3 Swap two variables */
4
5 #include <stdio.h>
6
7 void swap(double * px, double * py);
8
9
10 int main (void)
11 {
12     double a = 5, b = 10;
13
14     printf("a = %f, b = %f\n", a, b);
15     printf("address a = %p, address b = %p\n", &a, &b);
16     swap(&a, &b);
17     printf("address a = %p, address b = %p\n", &a, &b);
18     printf("a = %f, b = %f\n", a, b);
19
20     return 0;
21 }
22
23 void swap(double * px, double * py)
24 {
25     double t = *px;
26     *px = *py;
27     *py = t;
28 }
```

```
frankvp@CRD-L-08004:../Pointer$ gcc swap.c -o swap
frankvp@CRD-L-08004:../Pointer$ ./swap
a = 5.000000, b = 10.000000
address a = 0x7ffca03f5338, address b = 0x7ffca03f5340
address a = 0x7ffca03f5338, address b = 0x7ffca03f5340
a = 10.000000, b = 5.000000
frankvp@CRD-L-08004:../Pointer$
```

KU LEUVEN

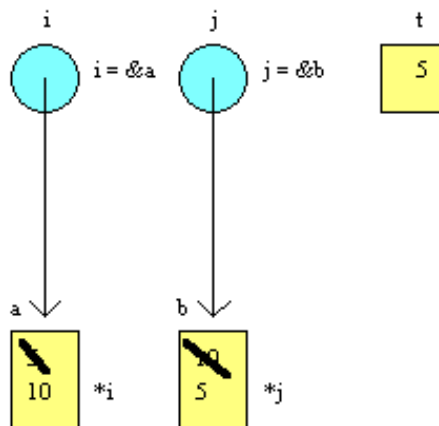
12

## pointers and function arguments

```
#include <stdio.h>

/* Swap two variables */
void swap(double *px, double *py)
{
    double t = *px;
    *px = *py;
    *py = t;
}

int main (void)
{
    double a = 5, b = 10;
    printf("a = %f, b = %f\n", a, b);
    swap(&a, &b);
    printf("a = %f, b = %f\n", a, b);
    return 0;
}
```



KU LEUVEN

13

## Pass by Reference

- Pointers provide *indirect* access to variables. This is why the `*` operator is called the *indirection* operator.
- Useful for:
  - Modifying the values of function arguments. Effectively enabling multiple return values. Example `scanf.c`
  - Avoiding copying of large objects (e.g., arrays, structures).

## Pointer Syntax: `const` Pointers

- A pointer may be declared `const` in two different ways.
- **Constant pointer**
  - `<type of pointer> * const <name of pointer>`
  - This type of pointer is a pointer that cannot change the address it is holding.
- **Pointer to a constant**
  - `const <type of pointer> * <name of pointer>`
  - This type of pointer can change the address it is pointing to but cannot change the value kept at those addresses.

# Constant pointer

- `<type of pointer> * const <name of pointer>`
- Can change value of pointed-to object, but pointer must always refer to the same address.

```
1 #include<stdio.h>
2 /*
3  const_pointer.c
4  constant pointer: A constant pointer is a pointer that cannot change the address its holding.
5  Once a constant pointer points to a variable then it cannot point to any other variable.
6  */
7
8 int main(void)
9 {
10     int var1 = 0, var2 = 0;
11     int * const ptr = &var1;
12
13     printf("var1 pointed by ptr = %d\n", *ptr);
14     *ptr = 10;
15     printf("var1 pointed by ptr = %d\n", *ptr);
16
17     // ptr = &var2; // try compiling
18
19     return 0;
20 }
```

```
frankvp@CRD-L-08004:~/Pointer$ gcc const_pointer.c -o const_pointer
frankvp@CRD-L-08004:~/Pointer$ ./const_pointer
var1 pointed by ptr = 0
var1 pointed by ptr = 10
frankvp@CRD-L-08004:~/Pointer$
```

KU LEUVEN

16

# Pointer to a constant

- `const <type of pointer>* <name of pointer>`
- Can change *what* it points to but cannot change the *value* of the object it points to.

```
1 #include<stdio.h>
2 /*
3  pointer_to_const.c
4  This type of pointer can change the address it is pointing to
5  but cannot change the value kept at those addresses.
6  */
7
8 int main(void)
9 {
10     int var1 = 0;
11     int var2 = 100;
12     const int *ptr = &var1;
13
14     var1 = 6;
15     printf("%d\n", *ptr);
16
17     // *ptr = 1; // try compiling this line
18     var1 = 1;
19     printf("%d\n", *ptr);
20
21     ptr = &var2;
22     printf("%d\n", *ptr);
23
24     return 0;
25 }
```

```
frankvp@CRD-L-08004:~/Pointer$ gcc pointer_to_const.c -o pointer_to_const
frankvp@CRD-L-08004:~/Pointer$ ./pointer_to_const
6
1
100
frankvp@CRD-L-08004:~/Pointer$ gcc pointer_to_const.c -o pointer_to_const
pointer_to_const.c: In function 'main':
pointer_to_const.c:17:10: error: assignment of read-only location '*ptr'
   17 |     *ptr = 1; // try compiling this line
      |     ^
frankvp@CRD-L-08004:~/Pointer$
```

KU LEUVEN

17



# Pointer Arithmetic

- Each variable type has a corresponding pointer type.
  - This allows the compiler to scale pointer-offset expressions appropriately. That is, it automatically calculates byte-offset.
- Pointer arithmetic is uniform for all types, and type-size details are hidden from the programmer.

18

```
1 #include <stdio.h>
2 /*
3 pointer_arithmetic_1.c
4 https://www.geeksforgeeks.org/pointer-arithmetics-in-c-with-examples/
5 */
6 #include <stdio.h>
7
8 // Driver Code
9 int main()
10 {
11     // Integer variable
12     int N = 4;
13
14     // Pointer to an integer
15     int *ptr1, *ptr2;
16
17     // Pointer stores
18     // the address of N
19     ptr1 = &N;
20     ptr2 = &N;
21
22     printf("sizeof(int): %ld\n", sizeof(int));
23     printf("Pointer ptr1 before Increment: ");
24     printf("%p \n", ptr1);
25     printf("content of memory ptr1 pointing at: %d \n", *ptr1);
26     ptr1++; // Incrementing pointer ptr1;
27     printf("Pointer ptr1 after Increment: ");
28     printf("%p \n", ptr1);
29     printf("content of memory ptr1 pointing at: %d \n\n", *ptr1);
30
31     printf("Pointer ptr2 before Decrement: ");
32     printf("%p \n", ptr2);
33     printf("content of memory ptr2 pointing at: %d \n", *ptr2);
34     ptr2--; // Decrementing pointer ptr2
35     printf("Pointer ptr2 after Decrement: ");
36     printf("%p \n", ptr2);
37     printf("content of memory ptr2 pointing at: %d \n\n", *ptr2);
38
39     return 0;
40 }
```

```
frankvp@CRD-L-08004:~/Pointer$ ./pointer_arithmetic_1
sizeof(int): 4
Pointer ptr1 before Increment: 0x7ffc40bdc444
content of memory ptr1 pointing at: 4
Pointer ptr1 after Increment: 0x7ffc40bdc448
content of memory ptr1 pointing at: 1086180680

Pointer ptr2 before Decrement: 0x7ffc40bdc444
content of memory ptr2 pointing at: 4
Pointer ptr2 after Decrement: 0x7ffc40bdc440
content of memory ptr2 pointing at: 0

frankvp@CRD-L-08004:~/Pointer$
```

19

## Pointer Arithmetic Operations

```
float fval, array[10];
float *p1, *p2, *p3 = &array[5];
int i=2, j;
p1 = NULL; /* Assignment to NULL (or to 0 or 0L). */
p2 = &fval; /* Assignment to an address. */
p1 = p2; /* Assignment to another pointer - same type).*/
p2 = p3 - 4; /* Integer operation */
p2 += i; /* Integer operation */
j = p3 - p2; /* Pointer subtraction: number of elements */
i = p2 < p3; /* Relational operations <, >, ==, !=, <=, >= */
```

- At end of this sequence:
  - p1 points to fval
  - p2 points to array[3]
  - p3 points to array[5]
  - i equals 1 and j equals 2

## Invalid Pointer Operations

- Addition or subtraction by a floating-point value.
- Multiplication or division by value of any type.
- Assignment to a non-pointer type.

# Pointer return from function

- Return the memory address of the value is stored instead of the value itself
- Be careful not to return an address to a temporary variable in a function
- File: *pointer\_return.c*

```
1#include <stdio.h>
2#include <stdlib.h>
3
4/*
5taken from Computer programming in C for beginners
6*/
7
8int * test(int); // the prototype
9int main()
10{
11    int var = -20;
12    int * ptr = NULL;
13    ptr = test(var);
14    printf("This is what we got back in main()-pointer: %p \n", ptr);
15    printf("This is what we got back in main()-value: %d \n", *ptr);
16    return 0;
17}
18
19int * test(int k)
20{
21    int y = abs(k);
22    int * ptr1 = &y;
23    printf("The value of y in test() directly is %d \n", y);
24    printf("The value of y in test() indirectly is %d \n", *ptr1);
25    return ptr1;
26}
```

KU LEUVEN

24

# Pointer return from function

```
1#include <stdio.h>
2// return_pointer_max.c
3// https://cs.brynmawr.edu/Courses/cs246/spring2014/Slides/10_Pointer_Array.pdf
4
5int * max(int * x, int * y);
6
7int
8main ()
9{
10    int a = 1, b = 2;
11    int *ptr;
12
13    ptr = max(&a, &b);
14
15    printf (" pointer returned %p pointing to max value %d \n", ptr, *ptr);
16
17    return 0;
18}
19
20
21int * max(int * x, int * y)
22{
23    if (*x > *y)
24        return x;
25    return y;
26}
```

```
1#include <stdio.h>
2// return_pointer_max_bis.c
3// https://cs.brynmawr.edu/Courses/cs246/spring2014/Slides/10_Pointer_Array.pdf
4
5int * max(int x, int y);
6
7int
8main ()
9{
10    int a = 1, b = 2;
11    int *ptr;
12
13    ptr = max(a, b);
14
15    printf (" pointer returned %p pointing to max value %d \n", ptr, *ptr);
16
17    return 0;
18}
19
20
21
22int * max(int x, int y)
23{
24    if (x > y)
25        return &x;
26    return &y;
27}
```

KU LEUVEN

25

# Pointer return from function

- A function may return a pointer value.

```
int* func_returns_pointer(void);
```

- However, this is often a mistake – if points to a local variable in the function
  - *File: return\_pointer\_danger\_2.c*
  - Especially dangerous because it sometimes works

26

```
1 #include<stdio.h>
2 // return_pointer_danger_2.c
3
4
5 int *abc(); // this function returns a pointer of type int
6 int fabc(int a);
7
8 int main()
9 {
10     int b;
11     int *ptr;
12     ptr = abc(10);
13     printf("pointer pointing to %p - value %d \n", ptr, *ptr);
14     b = fabc(23);
15     printf("pointer pointing to %p - value %d \n", ptr, *ptr);
16     printf("b - value %d \n", b);
17     return 0;
18 }
19
20 int *abc(int a)
21 {
22     int x, *p;
23     x = a;
24     p = &x;
25     return p;
26 }
27
28 int fabc(int a)
29 {
30     int x;
31     x = a;
32     return x + x;
33 }
```

```
(base) frankvp@CRD-L-08004:/mnt/c/Temp/Develop/CDev/Pointer$ gcc -Wall return_pointer_danger_2.c -o return_pointer_danger_2
(base) frankvp@CRD-L-08004:/mnt/c/Temp/Develop/CDev/Pointer$ ./return_pointer_danger_2
pointer pointing to 0x7ffc542acd4c - value 10
pointer pointing to 0x7ffc542acd4c - value 23
b - value 46
(base) frankvp@CRD-L-08004:/mnt/c/Temp/Develop/CDev/Pointer$
```

I

27

# Pointer return from function

- Pointer return values are OK if the object it points to remains in existence.
- Static or external variables: static extent.

```
double* geometric_growth(void) {  
    static double grows = 1.1;  
    grows *= grows;  
    return &grows;}  

```

- Variables with dynamic extent. (Dynamic memory )
- Variables passed as input arguments to the function.

```
char* find_first(char* str, char c)  
/* Return pointer to first occurrence of c in str.  
   Return NULL if not found. */  
{  
    while(*str++ != '\0')  
        if (*str == c) return str;  
    return NULL;  
}
```

29

KU LEUVEN

29

- <https://m.xkcd.com/371/>



KU LEUVEN

33