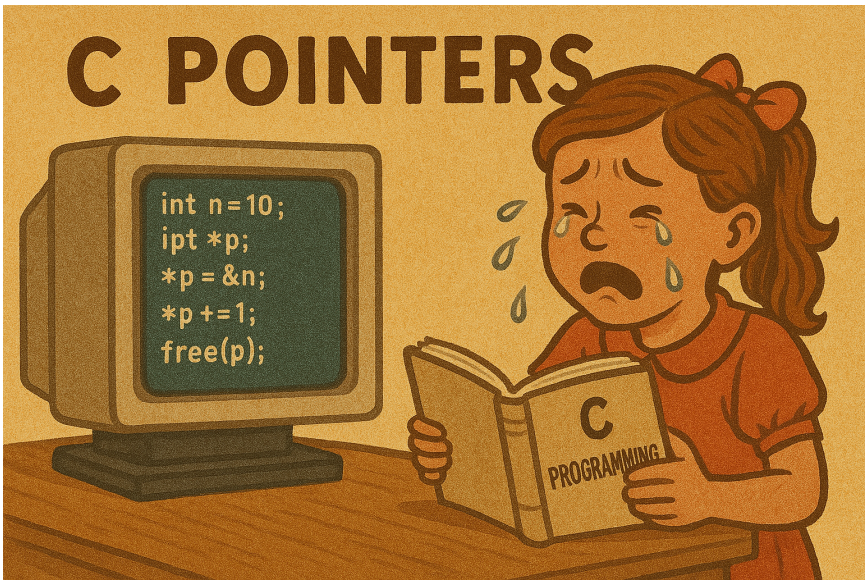
Pointers

CSC100 / Introduction to programming in C/C++

Marcus Birkenkrahe

April 25, 2025

README



- This script introduces C pointers in theory and practice.
- This section, including some sample code, is based on: chapter 11 in King (2008), and chapter 7 in Davenport/Vine.

Overview

- Remember the fundamental architectural problem of the "von Neumann" architecture of digital computers.

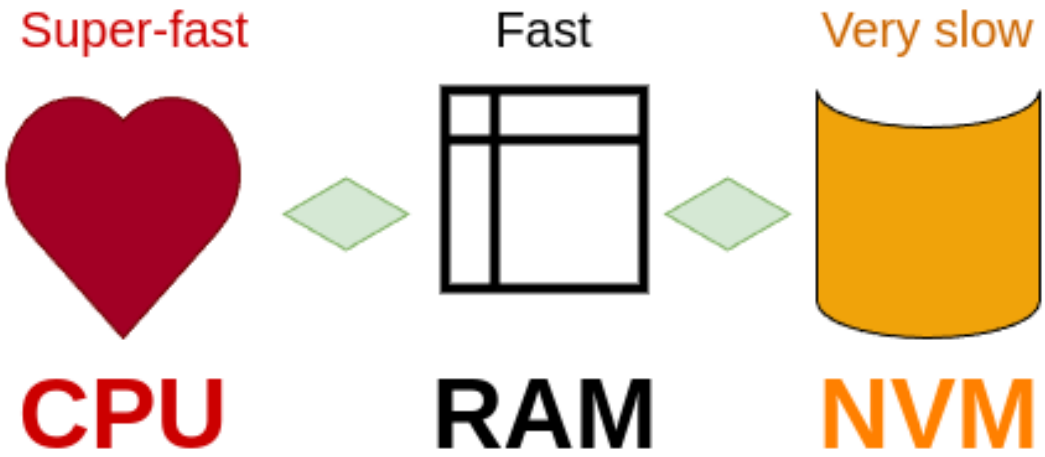


Figure 1: Computer architecture (simplified)

– Imagine: CPU ≡ brain, Memory ≡ house, Harddisk ≡ moon

1. CPU and the Brain: The CPU (Central Processing Unit) is like the brain because it is responsible for processing all instructions and making decisions. It performs the core computations and controls other parts of the computer based on the input it receives, similar to how the brain processes sensory information and dictates responses.
2. RAM and a House: RAM (Random Access Memory) is akin to a house because it serves as the active, working area where tasks are carried out and information is temporarily held. Just as a house contains rooms where daily activities occur (e.g., sleeping, eating, working), RAM holds active applications and data for quick access by the CPU. The size of the RAM affects how much activity (or how many applications) can be handled at once, much like the size of a house determines how many activities can comfortably occur simultaneously.
3. Hard Disk and the Moon: A hard disk is comparable to the Moon in that it serves as a distant but integral storage location. Just as the Moon orbits Earth

and is a more remote body holding impacts of meteorites and historical footprints, a hard disk stores data that is not immediately needed for current processing tasks but is crucial for long-term storage. Its contents are not as quickly accessible as those in RAM, reflecting the Moon's greater distance compared to our immediate environment.

- Computer memory is like a list of *locations*
- Each chunk of memory has an *address* to a location.
- *Pointers* point to these addresses:
- The *address* is not the house, it's a *reference* to the house:

Pointers in other languages

Most languages besides C/C++, like C#, Java, Python... do not offer pointers (easily) - most of the direct memory management is "abstracted" away (i.e. the memory is hidden and a simpler user interface is offered).

1. **C#:** C# does support pointers, but their use is limited to unsafe code blocks. You must explicitly mark these areas with the `unsafe` keyword. This feature allows you to perform operations that involve direct memory access, similar to C or C++. However, using pointers is generally discouraged in favor of safer constructs provided by the language, such as objects and reference types.
2. **Java:** Java does not support pointers, at least not in the traditional sense as seen in C or C++. Instead, Java handles memory through references to objects, and these references are abstracted away from direct memory address manipulation. This abstraction is part of Java's design to ensure security and simplicity in memory management, reducing the likelihood of errors such as memory leaks or pointer arithmetic.
3. **Python**: Python also does not use pointers. It handles everything by object references. The language is designed to abstract away most direct memory management tasks from

Main Memory

Address	Data
0000	Hello
0001	World
0010	This
0011	are
0100	Some
0101	Sample
0110	Data
0111	Stored
1000	Into
1001	An
1010	Imaginary
1011	Main
1100	Memory
1101	Of
1110	A
1111	computer

Figure 2: Sample data stored in an imaginary memory stack

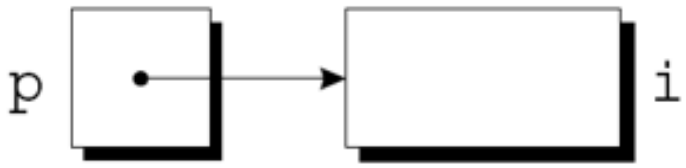


Figure 3: A pointer p points to address of i

Figure 4: envelope = pointer to an address

the developer, automatically handling object creation and destruction through a built-in garbage collector. Python's approach simplifies the development process but at the cost of direct control over memory and performance optimizations that pointers can offer.

4. **R:** R does not support pointers in the traditional sense used in lower-level languages like C. Instead, R uses a system of pass-by-value semantics with copy-on-modify behavior, meaning that objects are generally copied when modified, though under the hood this is optimized using references to avoid unnecessary duplication. Memory management in R is handled automatically through garbage collection, and direct manipulation of memory addresses is not exposed to the user. While advanced users can interface with C code via the `.C` or `.Call` interfaces for performance or memory control, everyday R programming is intentionally abstracted from low-level memory details to promote safety and ease of use.

What's the big deal?

- C and C++ offer pointer variables and operators naturally.
- This gives you a lot more control over the computer (because every operation, every process involves memory management).
- Examples:
  - **String manipulation**: Strings are arrays (working with text - e.g. when creating fast-performing chat bots or AI agents).

```
char *name = "Marcus"; // pointer to 'M' ('\0' is added at end)

printf("Hello, %s\n", name); // the whole string = array

Hello, Marcus
```

  - **Dynamic memory allocation** - the process of assigning memory during the execution time (when a program typically competes with thousands of other processes).

```
int *p = malloc(sizeof(int)); // allocate memory for 1 integer
(*p) = 42;  // de-reference the pointer p
printf("%d\n", *p); // print the value
free(p); // de-allocate the memory


42
```

- About the `sizeof` operator: It returns the size of a memory cell (10 elements of length 4 for an integer each):

```
int a[10];

int size_a0 = sizeof(a[0]); // size of array element
int size_a  = sizeof(a)/size_a0; // size of array

printf("Size of array element = %d\n", size_a0);
printf("Size of array         = %d\n", size_a);


Size of array element = 4
Size of array         = 10
```

- This is **mind control**: You can essentially decide what the computer should think with which part of its "brain" (great potential to mess up, too), e.g. when you mis-allocate resources.

Indirection (concept)

- Imagine you have a *variable* `iResult` that contains the *value* `75`.
- The variable is *located* at a memory address, e.g. `0061FEC8`. To find this out, you need to print its address, `&iResult`, with `%p`.

```
int iResult = 75.;
printf("%d is referenced by %p\n", iResult, &iResult);


75 is referenced by 0x7fff2969be84
```

- Repeated executions of this code creates different addresses. A pointer is a variable that holds an address.
- Imagine you have a *pointer variable* `myPointer` that contains the address of the variable `iResult`:

```
int iResult = 75;
printf("%d <- %p\n", iResult, &iResult);

int *myPointer = &iResult;
printf("%p -> %d\n",myPointer, iResult);


75 <- 0x7ffeadef575c
0x7ffeadef575c -> 75
```

- This means that `myPointer` *indirectly* points to the value `75`.
- You already worked with addresses: An *array name* `a` is a pointer to the start of the array, the address of `a[0]`:

```
int a[3] = {100,200,300}; // define an array
printf("a = %d", *a);  // printing ‘a‘ means printing ‘*a‘


a = 100
```

- The conversion specifier `%p` lets us access the addresses that correspond to elements of the array `a`, and even the address for the whole array.
- We declare an array `a[2]` and then we print its addresses:

```
int a[2] = {100,1000}; // define a 1-dim array of 2 elements
printf("a[0] = %p\na[1] = %p\n&a   = %p\n", &a[0], &a[1], &a);


a[0] = 0x7fff7d7e3130
a[1] = 0x7fff7d7e3134
&a   = 0x7fff7d7e3130
```

- You can see that the address for `a` points to `a[0]`.

- You already worked with pointers: arguments in the call of `scanf` are *pointers*: without the `&`, the function would be supplied with the *value* of `i`, not the *address*. But `scanf`'s job is to assign a memory location (an address) to the input variable.

```
int i; // reserve memory for an integer called ‘i‘
scanf("%d", &i); // put the keyboard input into that memory
```

- The relationship between variable value and memory address is called **indirection**: A *pointer* provides *indirect* access to the value via the address where the value is stored.

Indirection (code)

- There are two *unary* pointer operators:
  - the *address* (or referencing) operator `&`
  - the *indirection* (or dereferencing) operator `*`
- The unary *address* operator `&` returns a computer memory address, e.g. `&i = 0x7ffc7600b79c` - it *references* the memory location.
- What if `i` has not been initialized yet? Will the address change? Print the reference to `i` before initializing it, and after.

```
int i;  // reserve memory for an integer i
printf("%p\n",&i); // print the memory address
i = 100;  // assign a value to i
printf("%p\n",&i); // print the memory address


0x7ffeca08d214
0x7ffeca08d214
```

- The unary *indirection* operator `*` returns a value, e.g. `*p = 75` if `p` points at a variable that contains the value `75`.
- Let's do it all in one code block (**Open your C editor now**):
  1. declare an integer variable `i`
  2. assign the value `1` to `i`

3. declare an integer pointer `*p`
4. assign the address of `i` to `p`
5. print `i` and `&i` (`"is located at address"`)
6. print `p` and `*p` (`"points to value"`)

```
// variables
int i; // declare integer variable
i = 1; // assign value to variable

// pointers
int *p; // declare integer pointer
p = &i; // assign address to pointer

// print variable value
// PRINT "i is located at address &i"
printf("%d is located at address %p\n",i,&i);

// print pointer memory address
// PRINT "p points to the value i"
printf("%p points to value %d\n",p,i);
```

```
1 is located at address 0x7ffe61bcd93c
0x7ffe61bcd93c points to value 1
```

- What if you assign a number `1` to `p` instead of an address?

```
int *p; // declare integer pointer
p = 1; // warning: missing 'cast' (C knows p is an address)

// print pointer value
printf("%p\n",p); // '0x1' is a reserved memory address
```

```
0x1
```

- Compiler message (`-Wint-conversion` is a compiler flag):

```
warning: assignment to ‘int *’ from ‘int’ makes pointer from integer
         without a cast [-Wint-conversion]
15 | p = 1; // warning: missing ’cast’
   |   ^
```

- Here is more documentation on compiler warnings. You can add them to your code block with the header argument `:flags`, e.g. `:flags -Wall`
- The figure illustrates these concepts. Can you describe what goes on from line to line?

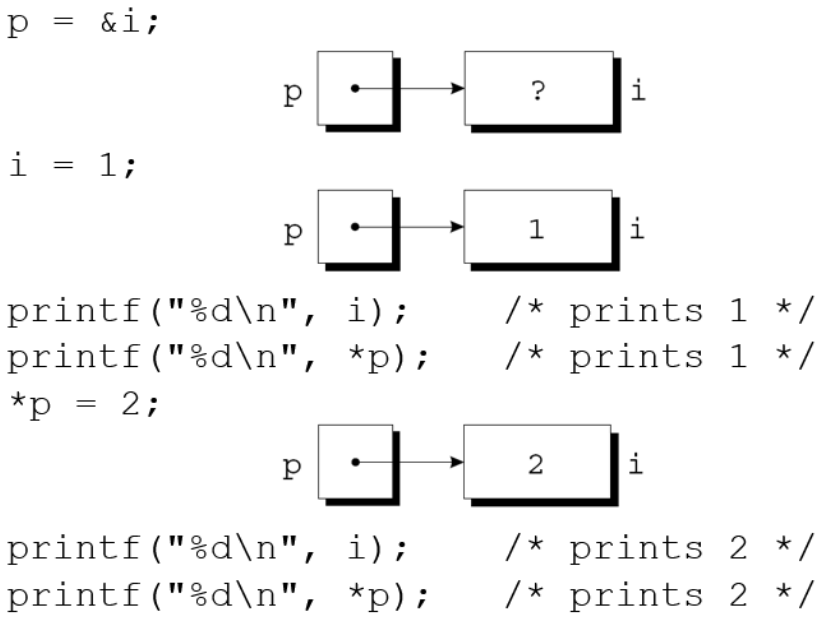


Figure 5: Graphical illustration of the indirection operator (Source: King)

1. The pointer `p` points to the address `&i` of the variable `i`.
2. `i` is initialized with the value `1`. `p` still points at it.
3. To change the value of `i` indirectly using the pointer `p`, we assign `*p = 2`. The indirection operator `*` designates a pointer.
4. To check that `i` indeed has been changed, we print it.
5. `*p` also prints the value of `i`.

Practice: Indirection operator

- Code the example that you just saw.
- The value for the **NULL** pointer is just that: `NULL`.
- Use the starter code: onecompiler.com/c/43fg9nvyj
- Sample output:

```
Variable: 1 - Pointer: 1
Variable: 2 - Pointer: 2
```

- Solution:

```
/* Change value of integer variable using pointer to variable
   Input: initialized integer variable, initialied NULL pointer
   Output: print original value of variable and pointer to variable
   before and after the change of value */

// include standard input/output library header file
#include <stdio.h>

int main()
{
  // declare integer variable
  int i;
  // declare and initialize NULL pointer
  int *p = NULL;
  // store address-to variable in pointer
  p = &i;
  // assign 1 to variable
  i = 1;
  // print values of variable and of pointer to variable
  printf("Variable: %d - Pointer: %d\n",i,*p);
  // change value of variable to 2 using pointer to variable
  *p = 2;
  // print values of variable and of pointer to variable
  printf("Variable: %d - Pointer: %d\n",i,*p);
  return 0;
}
```

```
Variable: 1 - Pointer: 1
Variable: 2 - Pointer: 2
```

Practice: Reassigning Pointers and Changing Values

- The following program starts with two variables a, b pointed at by two pointers p and q. We then manipulate the variable values by reassigning the pointers.
- Starter Pseudocode: onecompiler.com/c/43fhagatk

```
/* pointers.c: Manipulate variables using pointers */
/* Author: Marcus Birkenkrahe (pledged) GPLv3 */
/* Date: 04/20/2025 */

// declare two integer variables a and b

// declare two integer pointers p and q

// assign values: 5 to a and 10 to b

// point p to a, point q to b

// print values pointed to by p and q

// assign value pointed to by p to value pointed to by q

// print the values of a and b directly

// reassign p to point to b

// change value of b to 20 using the pointer p

// print a and b directly

// print values pointed to by p and q
```

- Sample output:

```
Values a = 5, b = 10
Values a = 5, b = 5
Values a = 5, b = 20
Values a = 20, b = 20
```

- Solution:

```
// declare two integer variables a and b
int a, b;
// declare two integer pointers p and q
int *p, *q;
// assign values: 5 to a and 10 to b
a = 5, b = 10;
// point p to a, point q to b
p = &a, q = &b;
// print values pointed to by p and q
printf("p points to %d, q points to %d\n", *p, *q);
// assign value pointed to by p to value pointed to by q
(*q) = (*p);
// print the values of a and b directly
printf("Values a = %d, b = %d\n", a, b);
// reassign p to point to b
p = &b;
// change value of b to 20 using the pointer p
(*p) = 20;
// print a and b directly
printf("Values a = %d, b = %d\n", a, b);
// print value pointed to by p and q
printf("Values a = %d, b = %d\n", *p, *q);
```

```
p points to 5, q points to 10
Values a = 5, b = 5
Values a = 5, b = 20
Values a = 20, b = 20
```

- Analysis:

CODE	EXPLANATION
`a = =5, b = =10`	Assign values to a, b directly
`p = &a, q = =&b`	Pointers point a address-of a, b
`*p, *q`	Dereferenced pointers p,q print a,b
`*q = *p=`	Move value *p = a into *q = b:
	Now a, b have the same values
	but p still points at a, q at b
`p = &b=`	Redirect p to address-of b
	Now both q and p point at b
`*p = 20=`	Change value of b to 20
	Both dereference pointers print b
	The value of a is not changed.

* and & are inverse to one another

- Address and indirection operator are *inverse* to one another (i.e. they reverse each other's operation - applying both amounts to doing nothing).
- Applying indirection `*` to an address *dereferences* it.
- Applying referencing `&` to a pointer extracts its address.

```
// declaring and initializing
int val = 75, *ptr = &val;

// print variable and dereferenced pointer
printf("value = %d => *&(value) = %d\n",
       val, *&val);

// print pointer and address of pointer
printf("pointer = %p => &*(ptr) = %p\n",
       ptr, &*ptr);


value = 75 => *&(value) = 75
pointer = 0x7fffc27cb37c => &*(ptr) = 0x7fffc27cb37c
```

- Applying `*` to the pointer takes us back to the original variable (dereferences the pointer)

```
j = *&i  // same as j = i
```

Practice: Referencing and De-referencing

- Practice how `*` (indirection) and `&` (address-of) are inverse operations in C.
- Write a short C program that:
  1. Declares an integer variable and assigns it a value.
  2. Declares a pointer to that variable using `&`.
  3. Uses `*` to get the original value from the pointer.
  4. Prints all of the following:
     - The value of the variable.
     - The value of `*&variable`.
     - The value of `*pointer`.
     - The value of `&*pointer`.
- Starter code: onecompiler.com/c/43fhb4ft3

```
// BEGIN
  // DECLARE integer variable x
  // SET x to some value

  // DECLARE pointer p to integer
  // SET p to address of x

  // PRINT value of x
  // PRINT value of *&x
  // PRINT value of *p
  // PRINT value of &*p
// END
```

- Sample output:

```
x = 123
*&x = 123
*p = 123
&*p = 0x7fffc147e19c
```

- Solution:

```
#include <stdio.h>

int main(void) { // BEGIN
  int x; // DECLARE integer variable x
  x = 123; // SET x to some value

  int *p;  // DECLARE pointer p to integer
  p = &x;  // SET p to address of x

  printf("x = %d\n", x); // PRINT value of x
  printf("*&x = %d\n", *&x); // PRINT value of *&x
  printf("*p = %d\n", *p); // PRINT value of *p
  printf("&*p = %p\n", (void *)&*p); // PRINT value of &*p

  return 0;
}  // END
```

```
x = 123
*&x = 123
*p = 123
&*p = 0x7fff9af9d54c
```

Pointers must be initialized

- Non-initialized pointers lead to invalid data or expressions.
- Pointer variables should always be initialized with:
  - another variable's memory address (e.g. `&i`), OR
  - with 0, OR
  - with the keyword `NULL`.
- Here are some *valid* pointer initializations - `printf` uses the conversion specifier `%p` for pointers.

```
double *ptr1;  // pointer declarations
int *ptr2;
int *ptr3;
double x = 3.14; // initialize variable
```

```
ptr1 = &x; // initialize with address
ptr2 = 0;  // initialize with 0
ptr3 = NULL; // initialize with NULL

printf("%p %p %p\n", ptr1, ptr2, ptr3);


0x7ffc08eeb8d8 (nil) (nil)
```

- Let's print these last values: how do you have to change the `printf` statement? (Add the flag `-w` to disable all warnings)

```
double *ptr1;  // pointer declarations
int *ptr2;
int *ptr3;
double x = 3.14; // initialize variable

ptr1 = &x; // initialize with address
ptr2 = 0;  // initialize with 0
ptr3 = NULL; // initialize with NULL

// different conversion specifiers
printf("%.2f %d %d\n", *ptr1, ptr2, ptr3);


3.14 0 0
```

- Here are a few non-valid initializations: we want to change the value of a variable using the pointer to its memory address.
  - can you tell why?
  - can you right the wrong?
  - print `iPtr`, `&i` and `i`

```
int i=5; // declare integer i
int *iPtr; // declare pointer iPtr

iPtr = &i; // initialize pointer
iPtr = 7;  // change value of variable
```

- Solution:

```
int i;
int *iPtr;

iPtr = &i;   // pointer initialized with memory address
*iPtr = 7;   // value of i indirectly changed

printf("%p %p %d\n", iPtr, &i, i);


0x7ffc28463cac 0x7ffc28463cac 7
```

Practice: Initializing pointers

- A few things went wrong in the code block below.
- Fix the problems, and then print the value that `ptr` points to, the address that `ptr` references, and the value of `var`.
- Open this in the editor on: onecompiler.com/c/43fhtmp7k

```
// declare double precision variable
double var = 3.14159;
// declare pointer to double precision variable
int ptr;

// initialize pointer
ptr = var;     // ptr is assigned the address of x
ptr = 2.71828; // value of x is indirectly changed to e

// print dereferenced pointer, address and value of var
printf("Deref'd ptr %d\nAddress-of var = %d\nValue of var = %p\n",
       ptr, &ptr, &var);


Deref'd ptr 2
Address-of var = 1924530732
Value of var = 0x7ffd72b60230
```

Solution

```
// declare double precision variable
double var = 3.14159;
// declare pointer to double precision variable
double *ptr;

// initialize pointer
ptr = &var;      // ptr is assigned the address of var
(*ptr) = 2.71828; // value of var is indirectly changed to the Euler number

// print pointer, address and value of i
  printf("Deref'd ptr %g\nAddress-of var = %p\nValue of var = %g\n",
         *ptr, ptr, var);


Deref'd ptr 2.71828
Address-of var = 0x7ffddba52b68
Value of var = 2.71828
```

Pass by value and pass by reference

- Since we've established the difference between a value and a reference (or address), we can use it in functions.
- What happens here?

```
#include <stdio.h> // input/output
// swap two integer arguments a->b, b->a
void swap(int, int);

/* main function */
int main(void)
{
  // declare and initialize two integer variables
  int x = 5, y = 10;
  // PRINT "Before swap: (x,y)"
  printf("Before swap: (%d,%d)\n", x, y);
  // swap x and y
  swap(x,y);
  // PRINT "After swap (in ‘main‘): (x,y)"
```

```
  printf("After swap (in 'main'): (%d,%d)\n", x, y);

  return 0;
}
/* function definition */
void swap(int a, int b) {
  int temp; // store value of a temp-orarily
  temp = a;
  a = b;
  b = temp;
  // PRINT "After swap (swap): (a,b)"
  printf("After swap (in 'swap'): (%d,%d)\n", a, b);
}


Before swap: (5,10)
After swap (in 'swap'): (10,5)
After swap (in 'main'): (5,10)
```

- Explanation:

  We're thwarted by the main advantage of functions: That their values are encapsulated and that their memory is returned when the function is done. The arguments x,y are said to be **passed by value**.

- How can we swap the numbers so that they remain swapped?

  Instead of passing the values, we can pass the addresses of the memory cells where x and y reside - then the swap will be permanent. This is called **pass by reference**.

- Code along: Open the editor on the URL onecompiler.com/c/43fhpg9cu

```
// input/output

/* function declaration */
// swap two integer arguments a->b, b->a as pointers

/* main function */
// BEGIN
```

```
// declare and initialize two integer variables

// PRINT "Before swap: (x,y)"

// swap x and y by reference (via their addresses)

// PRINT "After swap (in ‘main‘): (x,y)"

// END

/* function definition */
// swap two integer arguments a->b, b->a as pointers
// BEGIN
// store value of first variable temp-orarily
// assign value of second variable to first variable
// assign temporary value to second variable
// PRINT "After swap (swap): (a,b)"
// END
```

- Solution:

```
#include <stdio.h> // input/output
// swap two integer arguments a->b, b->a as pointers
void swap(int *, int*);

/* main function */
int main(void)
{
  // declare and initialize two integer variables
  int x = 5, y = 10;
  // PRINT "Before swap: (x,y)"
  printf("Before swap: (%d,%d)\n", x, y);
  // swap x and y by reference (via their addresses)
  swap(&x,&y);
  // PRINT "After swap (in ‘main‘): (x,y)"
  printf("After swap (in 'main'): (%d,%d)\n", x, y);

  return 0;
}
/* function definition */
```

```
// swap two integer arguments a->b, b->a as pointers
void swap(int *a, int *b) { // arguments are pointers!
  int temp; // store value of a temp-orarily
  temp = *a;
  (*a) = *b;
  (*b) = temp;
  // PRINT "After swap (swap): (a,b)"
  printf("After swap (in 'swap'): (%d,%d)\n", *a, *b);
}


Before swap: (5,10)
After swap (in 'swap'): (10,5)
After swap (in 'main'): (10,5)
```

Challenge: Incrementing an integer by value/reference

- The difference between modfiying a variable inside a function by:
  1. Pass by value (no lasting effect)
  2. Pass by reference (permanent effect using pointers)
- Write two functions:
  1. `void increment_val(int)` to increment an `int` passed by value
  2. `void increment_ref(int*)` to increment an `int` passed by reference
- Starter code: Open the editor on onecompiler.com/c/43fhqxrkc

```
// input / output

/* function declarations */
// increment an integer passed by value

// increment an integer passed by reference

/* main program */
// BEGIN
  // declare and initialize n as 1
  // PRINT "Before increment: x"
```

```
  // function call: increment x by value

  // PRINT "After increment_val " + increment x by value

  // function call: increment x by reference

  // PRINT "After increment_ref " + increment x by reference

// END

/* function definitions */
// increment an integer passed by value
// BEGIN
  // increment argument n
// END

// increment an integer passed by reference
// BEGIN
  // increment pointer argument

// END
```

- Solution:

```
/* function declarations */
// increment an integer passed by value
void increment_val(int);

// increment an integer passed by reference
void increment_ref(int *);

/* main program */
int main(void)
{ // BEGIN
  int x = 1; // declare and initialize n as 1
  // PRINT "Before increment: x"
  printf("Before increment: x = %d\n", x);

  // function call: increment x by value
```

```
  increment_val(x);

  // PRINT "After increment_val " + increment x by value
  printf("After increment_val: x = %d\n", x);

  // function call: increment x by reference
  increment_ref(&x);

  // PRINT "After increment_ref " + increment x by reference
  printf("After increment_ref: x = %d\n", x);

  return 0;
} // END

/* function definitions */
// increment an integer passed by value
void increment_val(int n) {
  // increment argument n
  n = n + 1;
}
// increment an integer passed by reference
void increment_ref(int *n) {
  // increment pointer argument
  (*n) = (*n) + 1;
}


Before increment: x = 1
After increment_val: x = 1
After increment_ref: x = 2
```

Bonus: Arrays and pointers as function arguments

- Arrays in C decay to pointers when passed to a function:

```
int array[4]; // actual array
sumArray(array, 4); // array decays to pointer to array[0]
sumPointer(*array, 4); // pass only the reference to array[0]
```

- The code below contains two functions:

1. function `sumArray` that directly takes an array and its size as parameters.
2. function `sumPointer` that takes a pointer to the first array element only
3. The size of the array is computed with `sizeof` and passed on so that the upper bound of the `for` loop is independent (`size`).

- Code example:

```
#include <stdio.h>

/* function declarations */
// Function takes array arr[size] as argument and returns integer
int sumArray(int [], int);

// Function takes pointer to the first element of array as argument
int sumPointer(int *, int);

/*  Main function */
int main()
{ // BEGIN
  // SET array of 4 elements initialized to 100
    int array[4] = {100,100,100,100};
  // SET size of array
  int size = sizeof(array) / sizeof(array[0]);
  // PRINT sum of array elements using sumArray
  printf("Sum using array: %d\n", sumArray(array, size));
  // PRINT sum of array elements using sumPointer
  printf("Sum using pointer: %d\n", sumPointer(array, size));

  return 0;
} // END

/* function definitions */
// sumArray: Sum the elements of an array
// takes array arr[size] as argument and returns integer
int sumArray(int arr[], int size)
{ // BEGIN
  int sum = 0; // SET sum to 0
  // FOR i from 0 to size; DO
```

```
  for ( int i = 0; i < size; i++ ) {
      sum += arr[i]; // sum array elements using values
  } // END FOR
  return sum;
} // END

// sumPointer: Sum the elements of an array
// takes pointer to the first element of array as argument
int sumPointer(int *ptr, int size)
{ // BEGIN
  int sum = 0; // SET sum to 0
  // FOR i from 0 to size; DO
  for ( int i = 0; i < size; i++ ) {
      sum += *(ptr + 1); // sum array elements using pointer
  } // END FOR
  return sum;
} // END

Sum using array: 400
Sum using pointer: 400
```

- How does this work exactly?

```
int array[4] = {100,200,300,400};
int size = sizeof(array) / sizeof(array[0]);
int *ptr = array;

for (int i = 0; i < size; i++)
  printf("%d ", *(ptr + i));

100 200 300 400
```

- What's the big deal? Even though arrays decay to pointers in function calls, they are not the same type in general:

Feature	Array	Pointer
Memory allocation	Fixed at declaration	Dynamic or points anywhere
sizeof operator	Total size	Size of pointer
Pointer arithmetic	No	Yes
Can be reassigned?	No (name is fixed)	Yes (can point elsewhere)

References

- Davenport/Vine (2015) C Programming for the Absolute Beginner (3ed). Cengage Learning.
- Kernighan/Ritchie (1978). The C Programming Language (1st). Prentice Hall.
- King (2008). C Programming - A modern approach (2e). W A Norton. URL: knking.com.
- Orgmode.org (n.d.). 16 Working with Source Code [website]. URL: orgmode.org