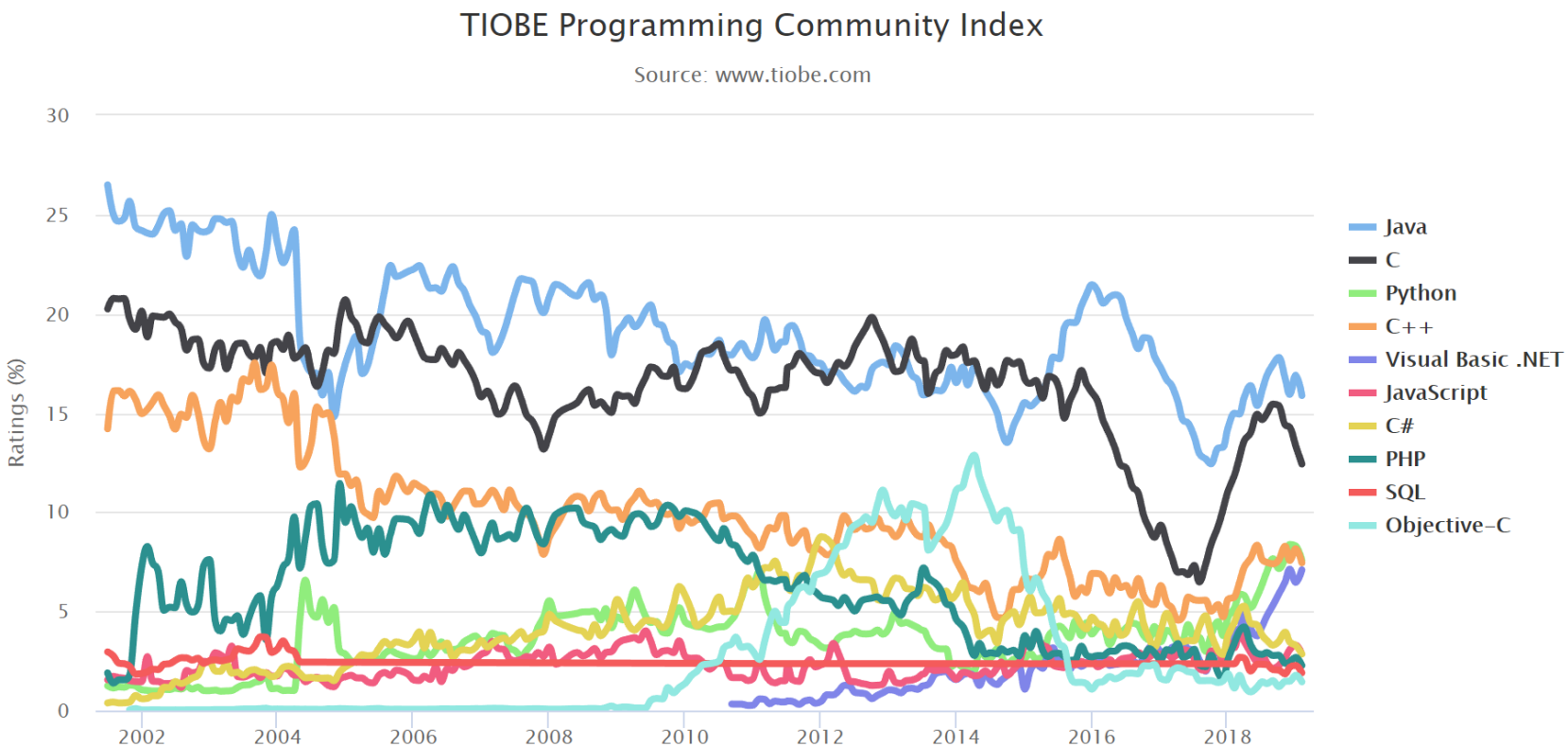
Introduction to the C Programming Language

Evan Russenberger-Rosica

March 11, 2020

History

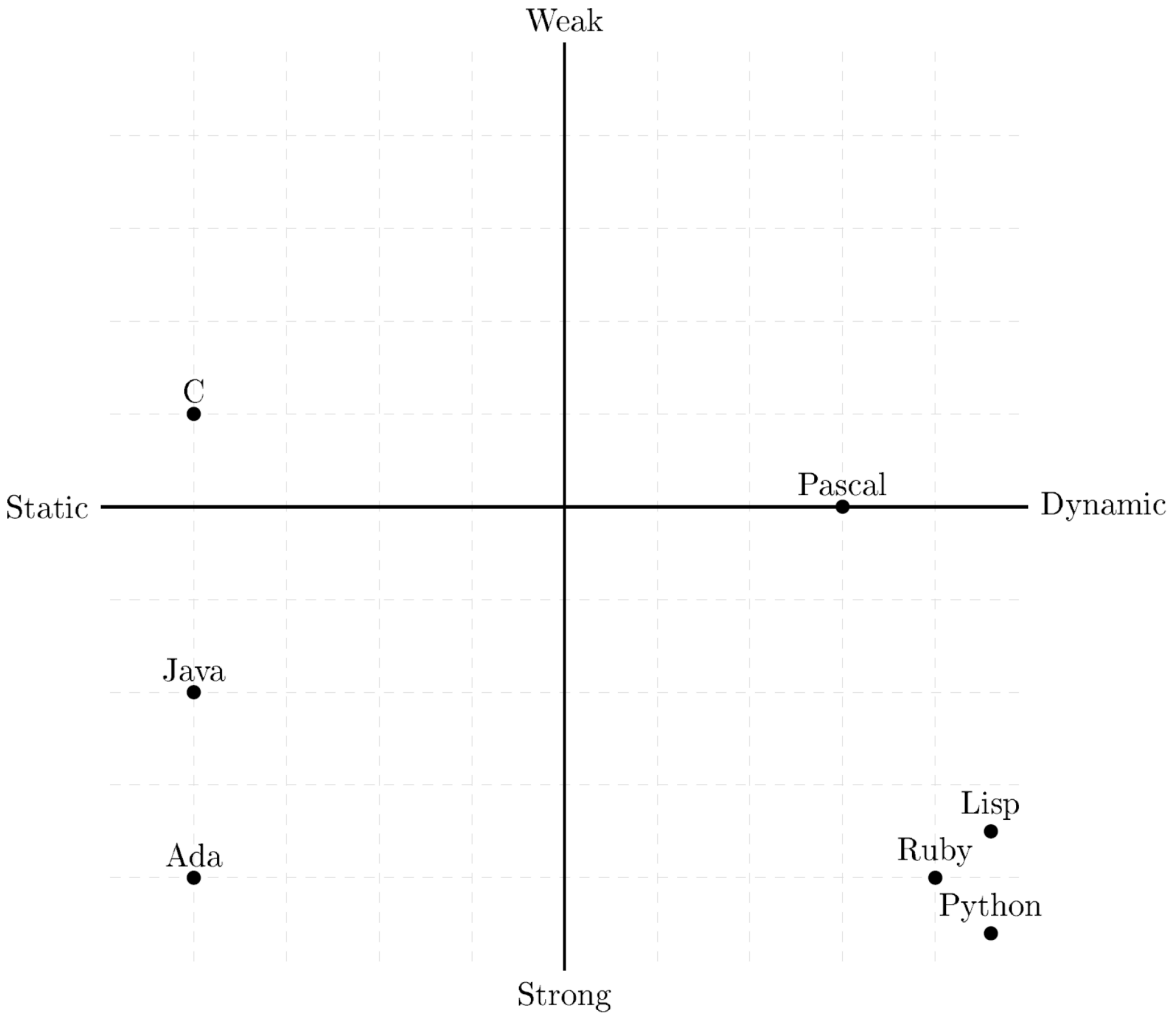
- Developed by Dennis Ritchie at Bell Labs between 1972-1973
- Influential on Java, Python, C++, C#, Objective-C, etc...



Typing

- **Static/dynamic typing** refers to the time when type checking occurs:
  - compile time for static typing, and run time for dynamic languages.
- **Strong/weak typing** refers to how aggressive a language is in enforcing its type system
- A language in which the programmer must explicitly identify the type of a variable at declaration is said to be **manifestly** or **explicitly typed**.
- C is statically, weakly, and manifestly typed. See [4, 291]

Static/Dynamic - Strong/Weak Typing Plane



Compilation & Execution

- C code must be **compiled** in order to be run.
- Typically, C is compiled to **assembly** code.
- The assembly code is then **assembled** to **machine code** by a program known as an **assembler**
- The machine code can then be run directly

We'll use the C compiler from the GNU compiler collection (GCC).
Syntax: `gcc myfile.c -o execfile`

Basic Data Types

There are only a few basic data types in C:

- `char` - a single byte, capable of holding one character in the local character set
- `int` - an integer, typically reflecting the natural size of integers on the host machine
- `float` - single-precision floating point double double-precision floating point
- `double` - double-precision floating point

See [2, 35].

Qualifying Types

We can apply several qualifiers to the basic data types.

- `short`,`long`
- `signed`,`unsigned`

The actual size of the types depends on the compiler. See [2, 35-36].

```
short int sh;
long int counter;
```

Declaration, Initialization, and Assignment

- All variables must be declared before use. A **declaration** specifies a type, and contains a list of one or more variables of that type.
- Once a variable has been assigned a value, it is said to be **initialized**. This process is called **initialization**.
- C allows us to declare and initialize variables at the same time.
- In the absence of explicit initialization, global and static variables are guaranteed to be initialized to zero; local and register variables have undefined (i.e., garbage) initial values.[2, 72].
- Once a variable has been declared, we do not need to specify the type at (re)assignment. [2, 35-36].

Declaration, Initialization, and Assignment: Example

```
1  #include <stdio.h>
2  int g;                    //declare global var g
3  void main()               //function declaration
4  {
5      int l;                //declare local var l
6      int k=0;              //declare and initalize local var k
7      printf("%d", l)       //uninitialized local prints garbage
8      printf("%d", g)       //uninitialized global prints 0
9      g=2;                  //(re)assignment: no type decl.
10     //since there is no local g, compiler sets global g=2
11 }
```

Arrays

- Arrays are **homogeneous**: they can only contain one data type (e.g. `int`)
- This makes it easy to allocate memory:

$$\text{array size} = |\text{elements}| \cdot \texttt{sizeof(element)}$$

- Declaration: `type name[size];`
- Indexes: start at 0

```
// Example: Iterating over an array
int ndigit[10];          //declare array of 10 ints
for (int i = 0; i < 10; ++i)  //set each element to 0
        ndigit[i] = 0;
```

Arrays Continued

- An array may be initialized by following its declaration with a list of initializers enclosed in braces and separated by commas.
- When the size of the array is omitted, the compiler will compute the length by counting the initializers, of which there are 4 in this case.

  ```
  int myarray[] = {1,2,3,4}
  ```
- If there are fewer initializers for an array than the specified size, the others will be zero. See [2, 73]

Character Arrays

- In C, a **string** is just an array of characters.
- Character arrays are a special case of initialization; a string may be used instead of the braces and commas notation. See [2, 73]
- The following are equivalent

```
char pattern = "hi";
char pattern[] = {'h','i','\0'};
```

Functions

- A **function definition** specifies how the function works (the "**implementation**")
- A **function declaration** (aka a **function prototype**) specifies a function's **interface**, i.e. its arguments, their types, and its return type. Functions must be declared before they are used. See [2, 27]
- Declaration Syntax:
  `return_type name(type1 param1, type2 param2, ...);`

Function Definition & Declaration: Example

We must declare a function before using it. In the example below, if we did not declare `power` before calling it in `main`, we would get an error. Alternativley, we could define `main` after `power`, which eliminates the need for the `power` function declaration.

```
1  int power(int base, int n);      //fn decl.
2  int main(){; /*assume main uses power*/}
3  int power(int base, int n)       //fn defn.
4  {
5      int p=1;
6      for (int i = 1; i <= n; ++i)
7          p = p * base;
8      return p;
9  }
```

Function Arguments & Call by Value

- In C, all function arguments are **passed by value**. This means that the called function is given the values of its arguments in temporary variables rather than the originals.
  - Basically, parameters can be treated as conveniently local variables in the called routine.
  - Exception: Arrays - with array arguments, the value passed to the function is the location or address of the beginning of the array - there is no copying of array elements.
- Contrast this will **call by reference** languages like FORTRAN, in which the called routine has access to the original argument, not a local copy.

Reading Complex Function Declarations

How do we make sense of the following function declaration?

```
int (*my_func())[4][4]
```

The trick is: starting with the identifier (i.e. the name `my_func`), we "read right when we can and left when we must". Thus we see:

$$\texttt{my\_func} \longrightarrow \texttt{my\_func...}$$
$$\texttt{my\_func()} \longrightarrow \ldots \text{ is a function...}$$
$$\texttt{my\_func(void)} \longrightarrow \ldots \text{ with no args...}$$
$$\texttt{(*my\_func(void))} \longrightarrow \ldots \text{ which points to...}$$
$$\texttt{(*my\_func(void))[4][4]} \longrightarrow \ldots \text{ a 2D array...}$$
$$\texttt{int(*my\_func(void))[4][4]} \longrightarrow \ldots \text{ of ints}$$

Resource: this website allows you to translate C declarations to and from English.

Reading Complex Type Declarations: Exercise

Intepret this function declaration:

```
int (*my_func(int (*arr)[4][4]))[4][10];
```

Remember: starting with the identifier (i.e. the name `my_func`), we "read right when we can and left when we must".

Reading Complex Type Declarations: Solution

```
int (*my_func(int (*arr)[4][4]))[4][10];
```

`my_func` ⟶ `my_func`...

`my_func()` ⟶ ... is a function...

`my_func(arr)` ⟶ ... with 1 arg `arr`...

`my_func((*arr))` ⟶ ... which points to...

`my_func((*arr)[4][4])` ⟶ ... a 2D array...

`my_funcy(int(*arr)[4][4])` ⟶ ... of `int`s...

`(my_func(int(*arr)[4][4]))` ⟶ ... and returns...

`(my_func(int(*arr)[SIZE][SIZE]))[4][10]` ⟶ ... a 2D array...

`int(my_func(int(*arr)[4][4]))[4][10]` ⟶ ... of `int`s.

Global Variables and Scope

- **local variables** in a function are created when the function is called, and disappear when the function is exited.
  - they do not retain their values from one call to the next
- **global variables** exist outside all functions, and can be accessed by name by any function
  - can be used instead of argument lists to communicate data between functions. [2, 31-32]
  - must be declared in each function that wants to access it using `extern`.
  - `extern` may be omitted if global variable occurs in the source file before the function [2, 33]
  - WARNING: overuse of global variables can lead to bad code! [2, 34]

Global Variables and Scope: Example

```
1  #include <stdio.h>
2  int a;                  //global declaration
3  void helper()
4  {
5      printf("%d",a); //no local a, so print global a
6  }
7  void main()             //execution starts here
8  {
9      extern int a;    //extern redundant since a def'd above
10     a=2;
11     helper();
12 }
```

```
1  >>> 2                    //2 is sent to helper via global
```

Introduction to Pointers

A computer has a set of memory addresses which label the bytes of its physical memory. Variables are an abstraction of this physical memory. In certain situations, it is advantageous to directly manipulate the relationship between variables and their address. This is the function of pointers.

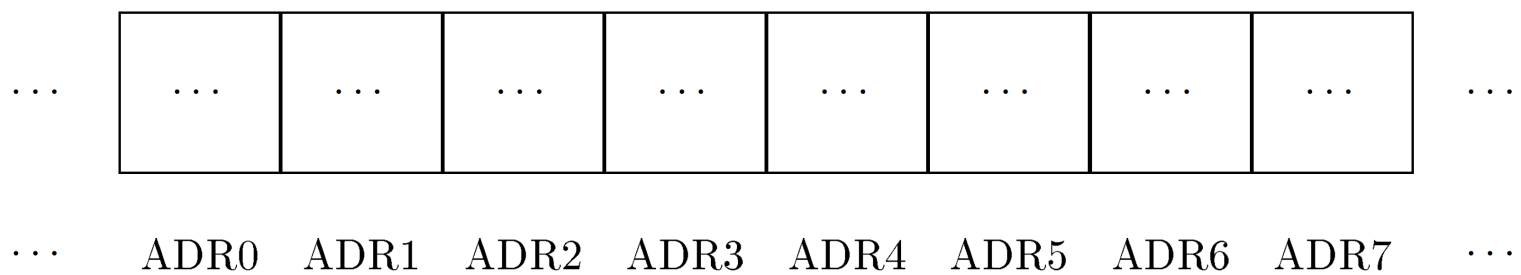


Figure 5.1: Memory Addresses Label the Bytes of the Physical Memory

The Referencing Operator: &

- A **pointer** `p` is a variable that contains the address of a variable `c`.
- The unary **referencing operator** `&` is used to access the address of variable`c`.
- E.g: `p=&c;` assigns the address of `c` to the variable `p`.
- If `p` is a pointer to `c`, we say `p` "points to" or is a "reference to" `c`. We may also refer to `c` as the "pointee" of `p`.

The Dereferencing Operator: *

- The unary operator * is the indirection or **dereferencing operator**; when applied to a pointer, it accesses the object the pointer points to.
- We can declare pointers similarly to how we declare variables.
  - `int = *ip;` declares that `*ip` is an integer, and therefore that `ip` is a pointer to an integer.
- This notation is indented to suggest that `*ip` can occur anywhere an `int` could.
- Pointers are constrained to point to a particular kind of object: every pointer points to a specific data type.[2, 78].

Pointer Example: Breakdown

We discuss the example from [2, 78] line by line.

```
1  int x=1, y, *ip; //declare ip points to int
2  ip = &x;         //assign ip the address of x
3  y = *ip;         //y is now 1
```

1. We declare two integers `x`,`y`, initialize `x` to 1, and a declare `ip` is a pointer to an integer (`ip` short for **i**nteger **p**ointer).
2. `ip = &x;` assigns `ip` the address of `x`, so that `ip` points to or references `x`.
3. The assignment statement `y = *ip;` sets `y` equal to the value stored at the address contained in `ip`. Since line 2 told `ip` to store the address of `x`, we have that `y` is equal to the value stored at the address of `x`, which is 1.

Pointer Example: Visualization

The code `ip=&x` tells `ip` to store the address of `x`. We say `ip` points to or references `x`. Dereferencing the pointer`ip` via `y = *ip` means assigning `y` the value pointed to by the arrow.

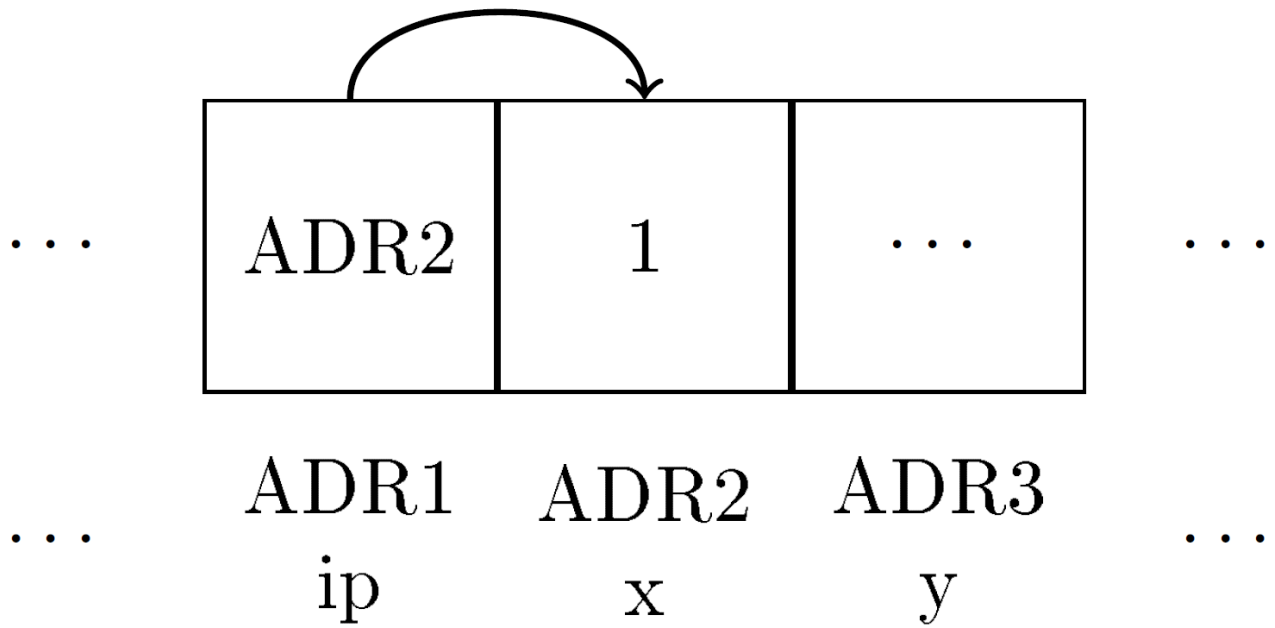


Figure 5.2: ip=&x

Shared References

- **Pointer assignment** between two pointers makes them point to the same pointee. So the assignment `y = x;` makes `y` point to the same pointee as `x`.
- Pointer assignment does not touch the pointees. It just changes one pointer to have the same reference as another pointer.
- After pointer assignment, the two pointers are said to be "sharing" the pointee, or that they have a **shared reference**.

Shared References Example: Code Breakdown

Consider the following code:

```
1  int x=1, y=2, *p1, *p2;
2  p1=&x;
3  p2=p1;          //create shared reference
4  printf("%p\t%p\n",*p1,*p2);
```

```
>>> 1   1
```

1. We declare two pointers `p1`, `p2` and initialize two int vars `x`, `y`.
2. We set `p1` to be address of `x` via the referencing operator `&`.
3. We use pointer assignment so that `p1=p2=&x`.
4. We dereference `p1` and `p2` and print the results

Shared References Example: Visualization

After running the code of the example, both `p1` and `p2` point to `x`, as illustrated by the arrows. Dereferencing `p1` and `p2` tells us to follow the arrows to find their value.

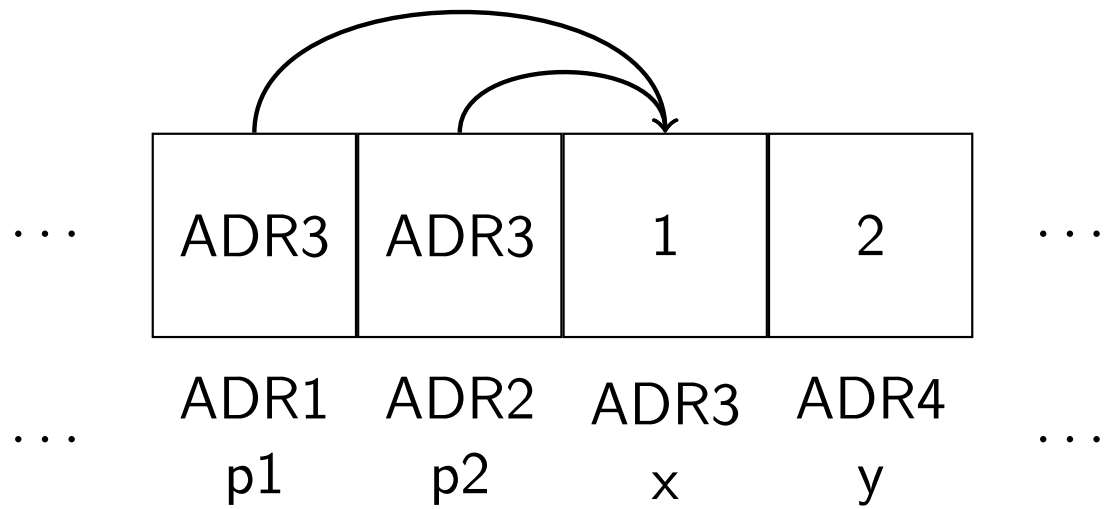


Figure 5.3: Output of Algorithim **??**

Exercise: Changing a Shared Reference

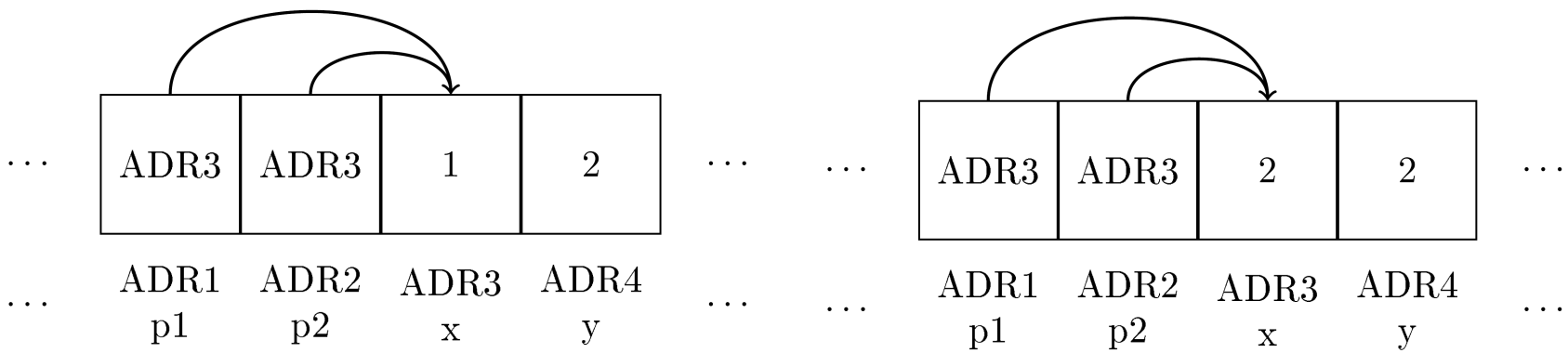
Exercise 1

Let `p1` and `p2` be pointers to `x=1`. What happens to `*p1`, `*p2` if we change the value of `x` to 2? Draw a diagram representing the program before and after the assignment `x=2`. Write a C program to confirm your answer.

Exercise: Changing a Shared Reference: Solution

```
1  int x=1, p1*, p2*;
2  printf("%d\t%d\n",*p1,*p2);
3  x=2
4  printf("%d\t%d\n",*p1,*p2)
```

```
>>> 1   1
>>> 2   2
```



(a) Before x=2

(b) After x=2;

Shared References Exercise

Recall algorithm 2:

```
1  int x=1, y=2, *p1, *p2;
2  p1=&x;
3  p2=p1;          //create shared reference
4  printf("%p\t%p\n",*p1,*p2);
```

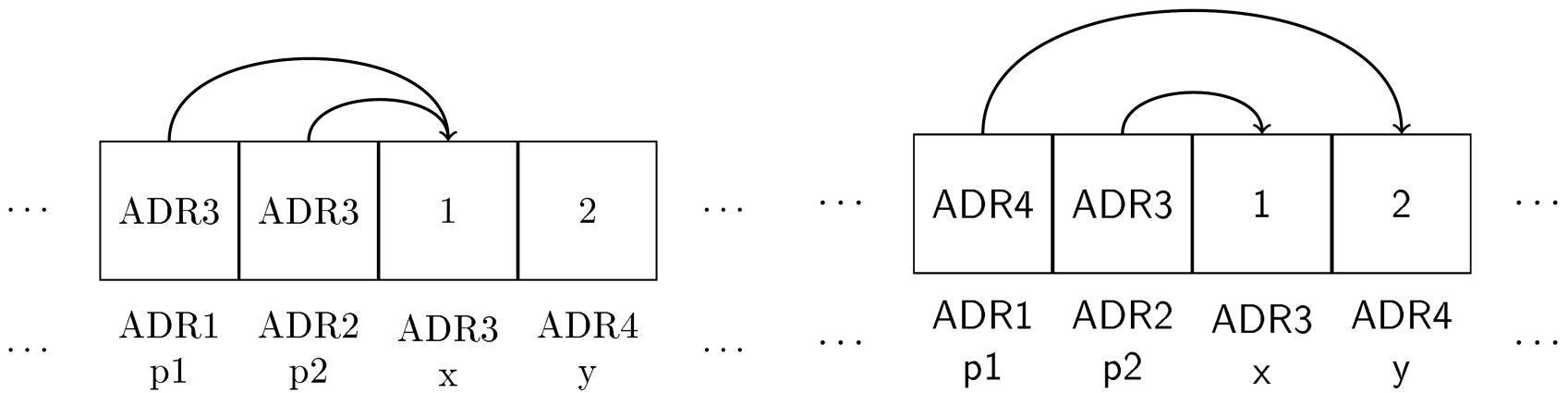
```
>>> 1   1
```

Exercise 2

Suppose that after running the above, we assign `p1=&y`. What are the values of `*p1` and `*p2` after this assignment? Draw diagrams representing the situation before and after assigning `p1=&y`. Run the code in C to confirm your answer.

Shared References Exercise: Solution

The statement `p1=&y` saves the address of `y` in `p1`. Nothing else changes. Thus `p2` still points to `x`, and thus `*p2` equals 1, while `p1` now points to `y`, and thus `*p1` equals 2.



(c) Before `p1=&y`

(d) After `p1=&y;`

Figure 5.4: Reassigning a Shared Reference

Takeaway: Shared References Do Not Form Pointer Chains!

- In the previous exercise, we assigned `p2=p1`, and then later assigned `p1=&y`. Why didn't this affect the value of `*p2`?
- The pointer assignment `p2=p1` does not say that `p2` is a pointer to `p1`. `p2=p1` and `p1=&y` are assignments not equalities!
- Do not apply the transitive property and say that `p2=p1` and `p1=&y` implies `p2=&y`.
- It only matters what `p1` pointed to when we assigned `p2=p1`, as this is the address that is stored in `p2`. Our subsequent reassignment of `p1` has no bearing on `p2`.
- Put succinctly, pointer assignment does not create a chain of pointers. Do not read the above code as saying `p2` points to `p1` and `p1` points to `y`, therefore `p2` points to `y`.

Creating a Chain of Pointers

To create a chain of pointers, we declare p2 to be a pointer to a pointer and dereference twice.

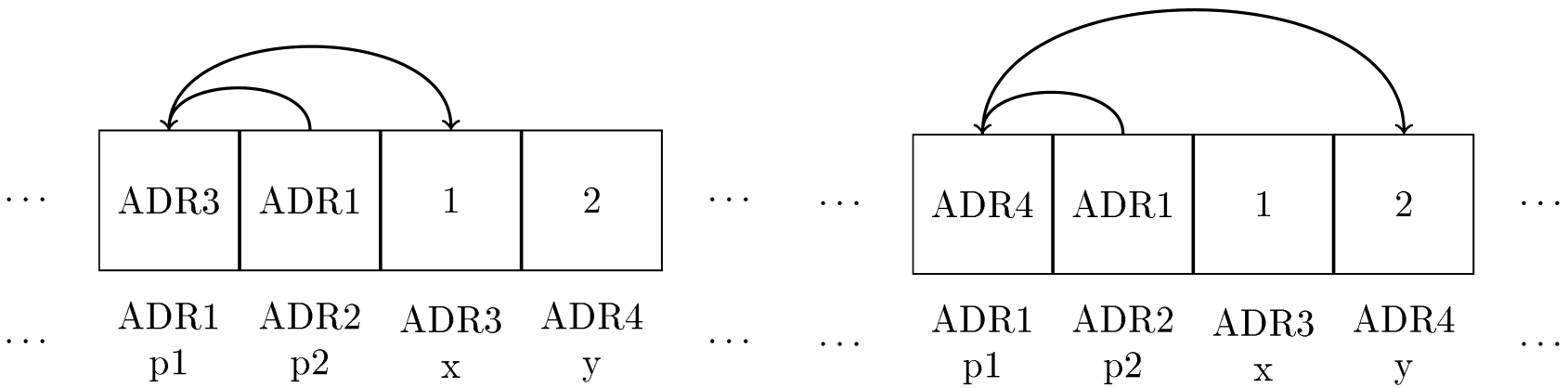
```
1  int x=1, y=2, *p1, **p2;
2  p1=&x;
3  p2=&p1;
4  printf("%d\t%d\n",*p1,**p2);
5  p1=&y;
6  printf("%d\t%d\n",*p1,**p2);
```

```
>>> 1   1
>>> 2   2
```

Thus changing p1 to point to y changes both p1 and p2. Contrast this behavior with that of shared references, where changing one pointer does not affect the other.

Creating a Chain of Pointers

Pictorially, dereferencing p2 twice via **p2 means we must follow two sets of arrows to get the value of p2. We show the situation before and after the assignment p1=&y; in figured

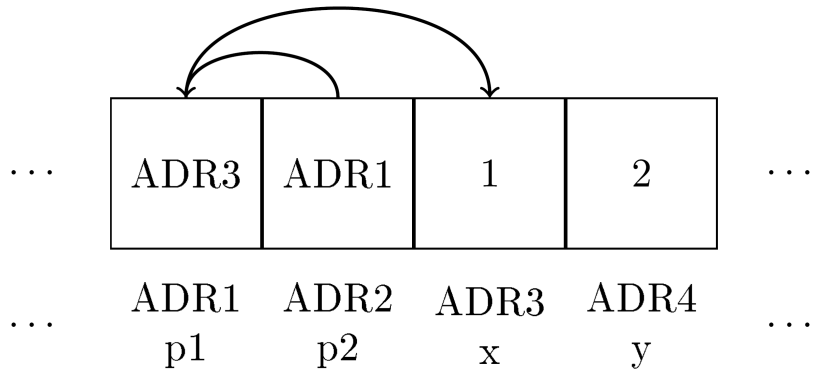


(a) Before p1=&y;

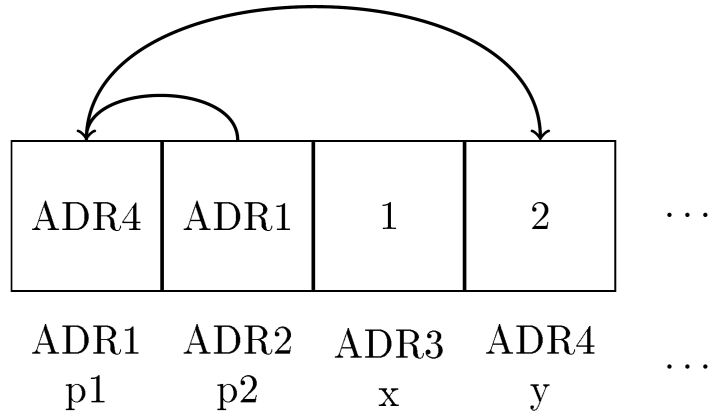
(b) After p1=&y;

Figure 5.5: Pointer Chain

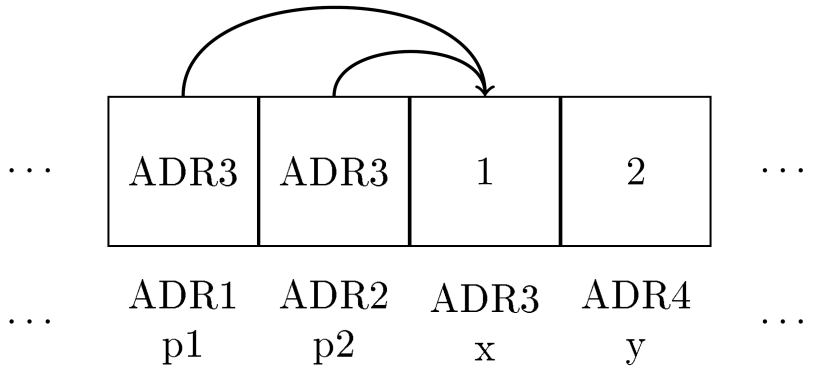
Pointer Chains (Top) vs Shared References (Bottom)



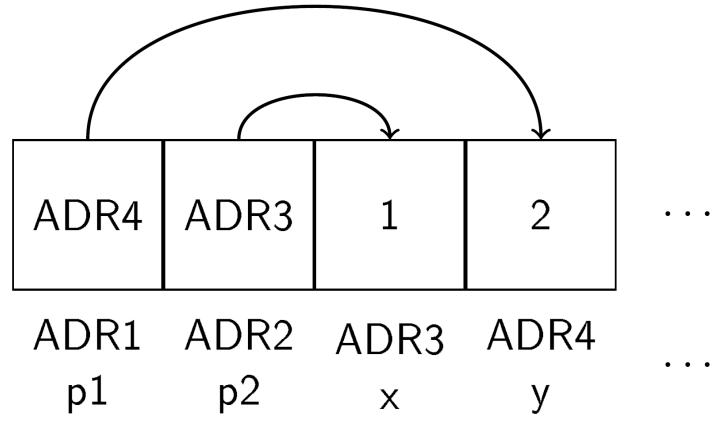
(a) **p2==1



(b) After p1=&y, **p2==2



(c) *p2==1



(d) After p1=&y, *p2==1

Pointers and Arrays

- The declaration `int a[10];` defines an array of 10 **consecutive** objects named `a[0], a[1], ...,a[9]`.
- Since the objects in an array are consecutive, if we know the address of the 0th element we can calculate the addresses of the *i*th object.
- By definition, the value of an array variable `a` is the address of its 0th element.
- Thus `pa = &a[0];` and `pa=a;` are equivalent ways to access the 0th element of `a`.[2, 83]

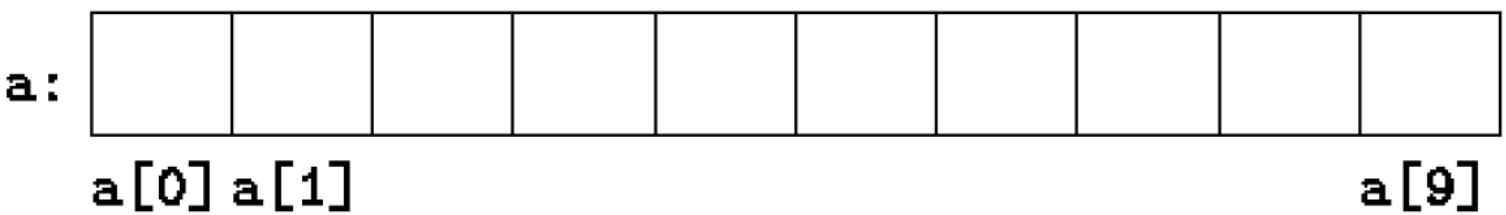


Figure 5.6: Arrays are Consecutive in Memory

Pointer Arithmetic

- Given the information from the previous slide, we see that any operation that can be achieved by array subscripting can also be done with pointers.
- This pointer manipulation is known as **pointer arithmetic**.

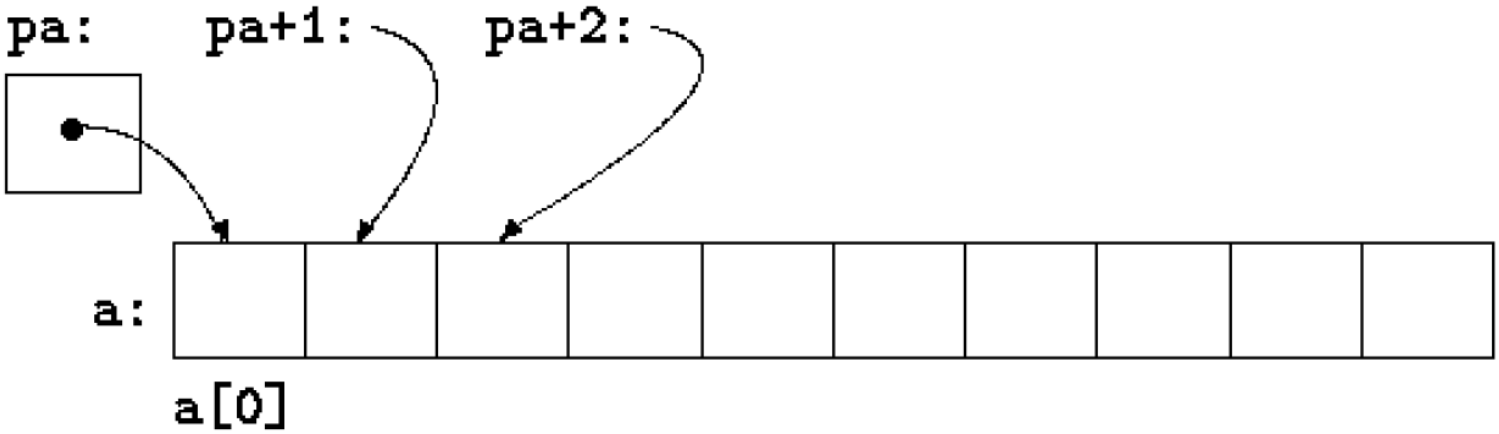


Figure 5.7: Arrays are Consecutive in Memory

Pointer Arithmetic: Easy Exercises

Exercise 3

If `a==&a[0]==31415`, what is the address of `a[5]`, i.e. what is `&a[5]` (aka `a+5`)?

Exercise 4

How do we get the value stored in address `a+5`?

Pointer Arithmetic: Easy Solutions

Solution

Ex 3: 31420

Solution

Ex 4: To get the value the value stored in address `a+5`, we simply dereference the pointer `a+5` via `*(a+5)`.

Pointer Arithmetic: Hard Exercise

Exercise 5

Write a C program which declares an character array `a` containing the word "`hello\0`. Declare `b` as a pointer to `a`. Use the pointer `b` to index array `a` via a for-loop, which terminates upon seeing `\0`. Print both the address and value stored there for each element in this list.

Pointer Arithmetic: Hard Solution

```c
#include <stdio.h>
void main()
{
  char a[] = "hello\0";
  //try setting the type to int by mistake
  for (char *b = a;*b != '\0'; b++)
    printf("%s%c%s%p\n", "Value: ", *b, " is at ", b);
}
```

```
>>> The value: h is at 0x7ffc950e06e0
>>> The value: e is at 0x7ffc950e06e1
>>> ...
```

Function Arguments: Exercise

Exercise 6

Goal: change the value of `main`'s local variable `a` to `4` in `helper` by manipulating pointers. Fill in the ???? below.

```
1  #include <stdio.h>
2  void helper(????)
3  {
4      ????=4;
5  }
6  void main()
7  {
8      int a=3;
9      helper(????);
10     printf("%d\n", a);
11 }
```

Function Arguments: Solution

```
1  void helper(int *a)  //accepts a pointer to int
2  {
3      *a=4;                    //change the value stored at address a
4  }
5  void main()
6  {
7      int a=3;
8      helper(&a);            //call with the address of a
9      printf("%d\n", a);
10 }
```

```
>>> 4
```

Function Pointers

- Just as we can have pointers to data, we can have pointers to functions. These are called **function pointers**.
- Interestingly, dereferencing function pointers does nothing.
- The syntax for declaring a function pointer p is below

```
1  #include <stdio.h>
2  int my_func(int x, int y) {return (x+y);}
3  void main()
4  {
5      //p is a pointer to a fn of two ints returning int
6      int (*p) (int, int) = &my_func;
7      printf("%d\n", p(1,2));
8  }
```

Function Pointers: Example

Suppose we have several functions which we want to successivley apply to an argument `x`. Because functions may have different sizes, we cannot create an array of them. However, we can create an array of pointers to functions

```
1  #include <stdio.h>
2  int A (int i) {return i;}
3  int B (int i) {return i;}
4  int C (int i) {return i;}
5  void main()
6  {
7      int (*fn_list[3]) (int i) = {A,B,C};
8      for(int i=0; i<3; i++) {
9          printf("%d ", (*fn_list[i])(i));
10     }
11 }
```

```
>>> 0 1 2
```

Pointers to Void

- Recall that like variables, every pointer has a type; with one exception: void pointers
- A **void pointer** is a pointer that has no associated data type with it. We must cast to tell C the type.
- A void pointer can hold an address of any type and can be typcasted to any type.[2, 161]

```
1  #include <stdio.h>
2  void main()
3  {
4      int x=257;
5      void *A = &x;
6      //printf("%d", *A); //error "dereferencing 'void *'"
7      printf("%d\n", *(int *)A);//cast to int * and dereference
8  }
```

Void Pointers: Exercise

What does this code print and why? Hint: consider 257 in binary.

```
1  #include <stdio.h>
2  void main()
3  {
4      unsigned x=257;
5      void *A = &x;
6      printf("%d\n", *((unsigned int *)A));
7      printf("%d\n", *((unsigned char *)A));
8  }
```

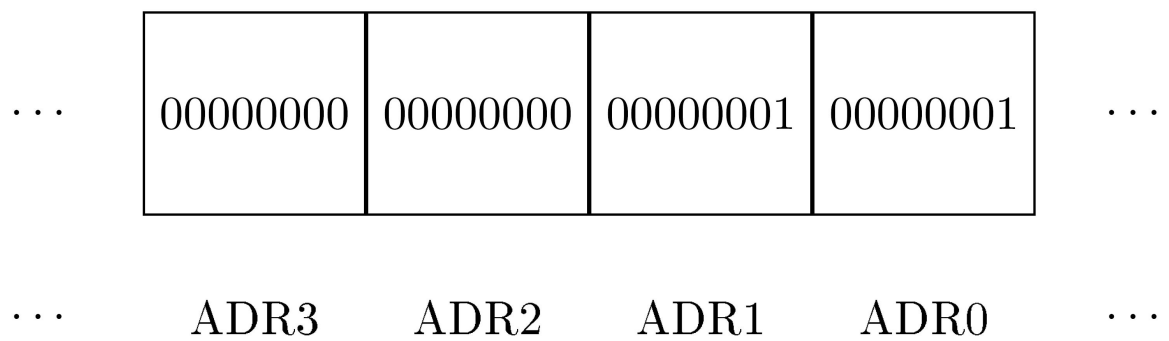


Figure 5.8: 257 in Memory as Binary

Void Pointers: Solution

Because unsigned ints are 4 bytes, the statement `x=257` writes the following into memory (assuming the computer reads memory from right to left starting at address 0).

$$\underbrace{00000000}_{\text{ADR3}}, \underbrace{00000000}_{\text{ADR2}}, \underbrace{00000001}_{\text{ADR1}}, \underbrace{00000001}_{\text{ADR0}}$$

In C, `chars` are 1 byte, and `ints` are 4 bytes.
When you cast `A` to an `int*`, you are telling the compiler that `A` is a pointer to an `int`, and `ints` are 4 bytes. Thus the compiler reads the next 4 bytes after `ADR0`, and sees the number $00000000, 00000000, 00000001, 00000001 = 257$.
When you cast `A` to a `char*`, you are telling the compiler that `A` is a pointer to a `char`, and `chars` are 1 byte. Thus the compiler reads the next 1 byte after `ADR0`, and sees the number $00000001 = 1$.

Defining Structures

- A **structure** or **struct** is a "collection of one or more variables, possibly of different types, grouped together under a single name for convenient handling."[2, 105]
- Structs are basically C's version of classes.
  - They are used to group related information together
  - Unlike Java classes, structs do not have methods.[1]
- The elements of a C structure are known as its **members** or **fields**. The members of a structure are analogous to the instance variables in Java.

[1] although a struct can have a variable which holds a function pointer

Defining Structures: Example

Below we define a Bank_Account struct. Its members represent the value of the account, the account number, and the owner (whose name must be less than 100 chars).

```
1  struct Bank_account {
2      int value;
3      int account_no;
4      char owner[100] ;
5  };
```

Initializing Structures

- Once we define a structure, we must create "instances" of it to do useful work.
- We do this by declaring a struct object using the `struct` keyword (line 1), and initializing its members (lines 2-5).
- We access the members of a struct using the familiar `object.variable` notation. E.g., `bank_account.value=100;` sets the `value` member of the `bank_account` struct to 100.
- Structs can be initialized similarly to lists as below:

```
1  struct Bank_account bank_account = {.value = 100,
2  .account_no = 8675309, .owner = "Bill"};
3  printf("value is %d\n", bank_account.value);
```

Typedef

- Every time we refer to the `Bank_account` "class", we must say `struct Bank_account`. This gets annoying quickly.
- We can avoid this by defining our structure using `typedef`

```
1  typedef struct{
2      int value;
3      int account_no;
4          char owner[100];
5  } Bank_account ;
6  Bank_account bank_account = {.value = 100,
7                               .account_no = 8675309,
8                               .owner = "Bill"};
9  printf("value is %d\n", bank_account.value);
```

Introduction: Linux System Calls

- The Linux operating system provides its services through a set of **system calls**, which are in effect functions within the operating system that may be called by user programs.[2, 138].
- System calls can also be thought of as "requests to the operating system (kernel) to do a hardware/system-specific or privileged operation."[3, 13]
- As of Linux kernel 3.7, there are 393 system calls.[1, 7]
- This section describes how to use some of the most important system calls from within C programs.

Files in Linux

- All input and output is done by reading or writing files, because all peripheral devices, even keyboard and screen, are files in the file system.
- This means that a single homogeneous interface handles all communication between a program and peripheral devices.
- Before you read and write a file, you must inform the system of your intent to do so, a process called opening the file. The system checks your right to do so (Does the file exist? Do you have permission to access it?) and if all is well, returns to the program a small non-negative integer called a **file descriptor**. [2, 138].

File Descriptors

- Whenever input or output is to be done on the file, the file descriptor is used instead of the name to identify the file. All information about an open file is maintained by the system; the user program refers to the file only by the file descriptor.
- Every running program starts with the following three files already opened:[2, 138].

File Name	Short Name	File #	Description
Standard In	stdin	0	Input from the keyboard
Standard Out	stdout	1	Output to the console
Standard Error	stderr	2	Error output to the console

Introduction: Fork, Exec, and Wait

- In Linux, a new process is created by the `fork()` system call.
- The new process consists of a copy of the address space of the original process.
- This mechanism allows the parent process to communicate easily with its child process.
- Both processes (the parent and the child) continue execution at the instruction after the `fork()`, with one difference: the return code for the `fork()` is zero for the new (child) process, whereas the (nonzero) process identifier of the child is returned to the parent.[5, 117]

- After a `fork()` system call, one of the two processes typically uses the `exec()` system call to replace the process's memory space with a new program.
- The `exec()` system call loads a binary file into memory (destroying the memory image of the program containing the `exec()` system call) and starts its execution.
- In this manner, the two processes are able to communicate and then go their separate ways.
- The parent can then create more children; or, if it has nothing else to do while the child runs, it can issue a `wait()` system call to move itself off the ready queue until the termination of the child.[5, 117]

Example: Fork, Exec, and Wait

```c
//must include stdio.h, unistd.h, sys/types.h
int main(){
  /* fork a child process */
  pid_t pid = fork();
  if (pid < 0) { /* error occurred */
    fprintf(stderr, "Fork Failed");
    return 1;
  }else if (pid == 0) {
    /* child process is the `ls` program*/
    execlp("/bin/ls","ls",NULL);
  }else { /* parent process */
    /* parent will wait for the child to complete */
    wait(NULL);
    printf("Child Complete");
  }
  return 0;
}
```

Introduction: Interprocess Communications (IPC)

- Linux **Interprocess Communications (IPC)** facilities provide a method for multiple processes to communicate with one another. There are several methods of IPC available to Linux C programmers:
  - Half-duplex pipes
  - FIFOs (named pipes)
  - Networking sockets (Berkeley style)
  - and more: see [3, 17]

Introduction to (Half-duplex) Pipes

- A half-duplex **pipe** is a method of connecting the standard output of one process to the standard input of another.
- Pipes are the eldest of the IPC tools, having been around since the earliest incarnations of the UNIX operating system.
- They provide a method of one-way communications (hence the term half-duplex) between processes.

- When a process creates a pipe, the kernel sets up two file descriptors for use by the pipe. We'll call these `fd0` and `fd1` for now.[2]
- One descriptor (`fd0`) is used to obtain data from the pipe (read), while the other (`fd1`) is used to allow a path of input into the pipe (write).
- If the process sends data through the pipe (`fd1`), it has the ability to obtain (read) that information from `fd0`.[3]
- In Linux, pipes are actually represented as a file within the kernel. This particular point will open up some pretty handy I/O doors for us, as we will see a bit later on.[3, 18]

[2] As we'll see shortly, `fd0` and `fd1` are just nicknames for `fd[0]`, and `fd[1]`. Do not confuse these with file descriptors 0 and 1!

[3] This is corrected from [3, 18], where there is an error; they have `fd0` and `fd1` backwards!

Intuition

- If you have ever used an interpreted language like Python, then you have given a program (the Python interpreter) input on `stdin` and read the results from `stdout`.
- From the interpreter's perspective, input is recieved (i.e. read) from `stdin`, and output is sent (i.e. written) to `stdout`.
- Thus `stdin` (file descriptor 0) is the "read", and `stdout` (file descriptor 1) is the "write".
- Analogously, it was decided that `fd0` would be the "read end" of the pipe and `fd1` would be the "write end"
- Some authors also call the "write end" the 'input' and the "read end" the 'output', since we must first input data into the "write end" in order to read it later as output.

Trivial Example

- If we simply create a pipe, and read/write from the appropriate ends, the creating process can only use the pipe to communicate with itself. However data traveling through the pipe still moves through the kernel.[3, 18] This situation is shown below:

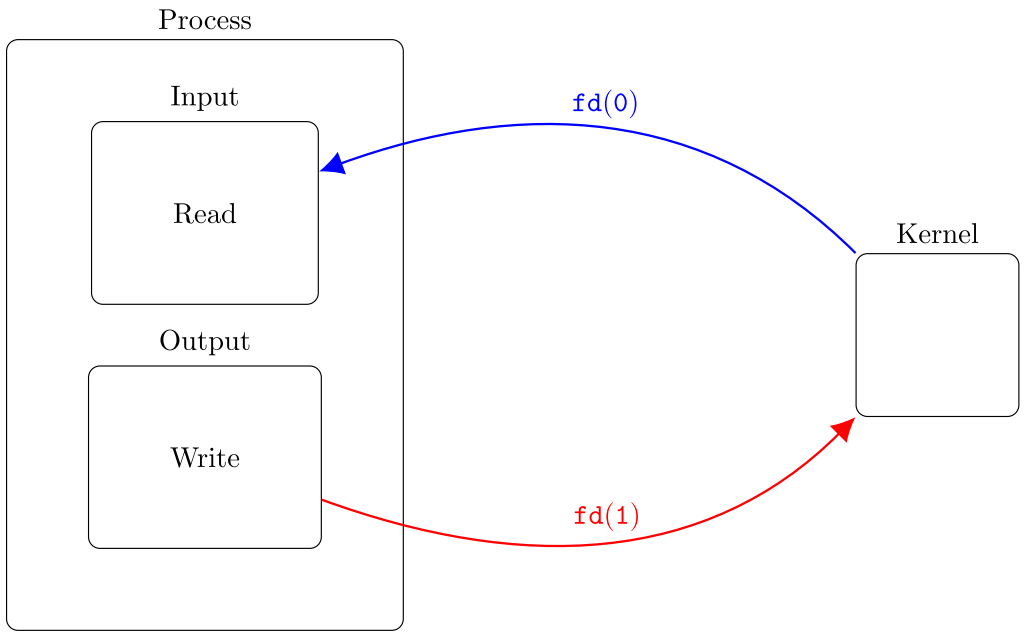


Figure 10.1: A Trivial Pipe

One Way Communication

- The pipe in figure 10.1 on the preceding page is fairly useless; why go to the trouble of creating a pipe to talk to ourself?
- The critical fact is that upon forking, the child process will inherit any open file descriptors from the parent.
- By manipulating this, we can obtain multiprocess communication (between parent and child processes).
- In the following slides, we will detail one way and two way communication betweeen parent and child processes in increasing levels of detail. We will then examine the corresponding code.

One Way Communication

Suppose we want a parent process to read input from a child process. We depict this below, where the arrow represents a pipe carrying output written by the child process. The output is then read as input by the parent process.

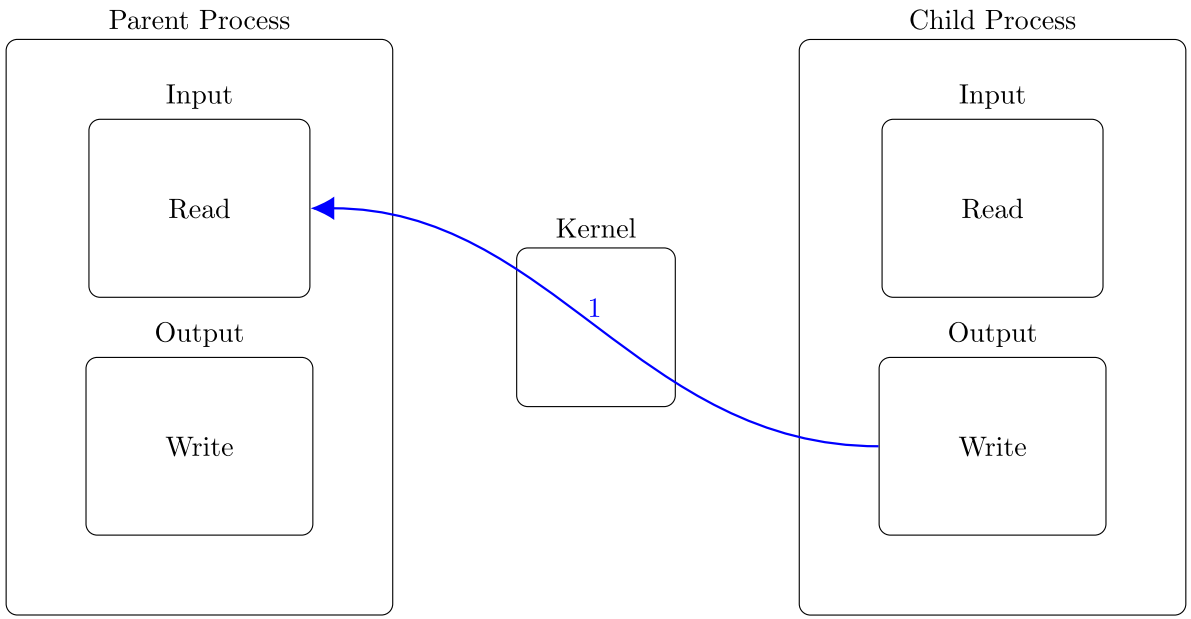


Figure 10.2: One Way Communication

One Way Communication in Detail

- Recalling that `fd1` represents the "write end" of the pipe, and `fd0` represents the "read end", we can update our diagram with more detail.
- Observe below that the child writes to `fd1` - the "write end", and the parent reads from `fd0` - the "read end".

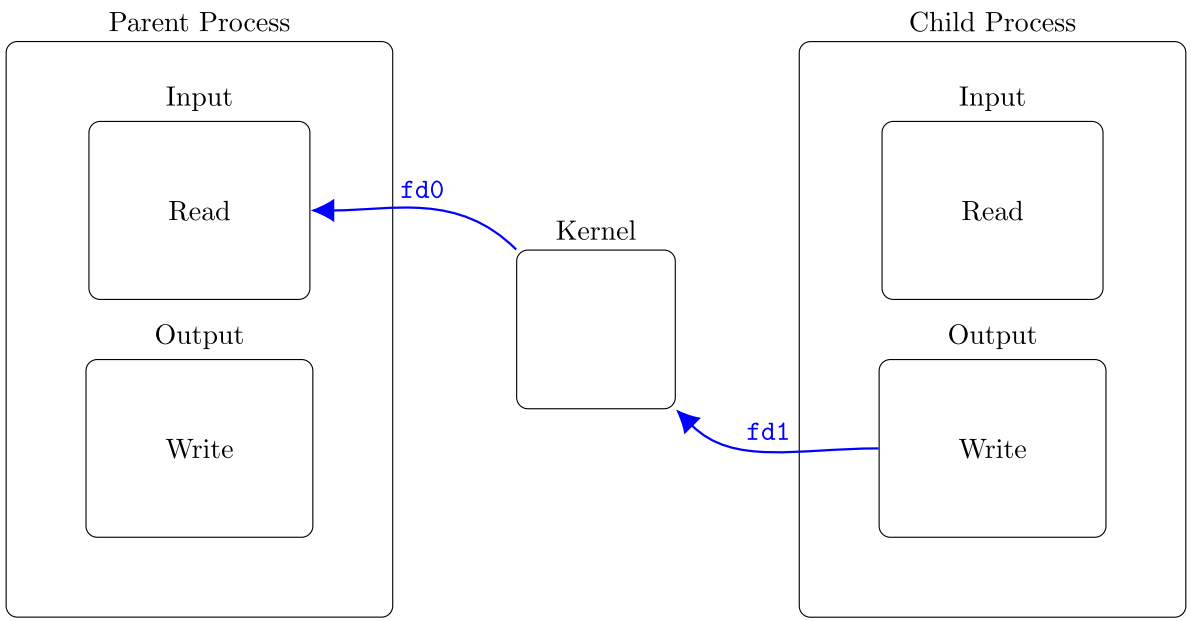


Figure 10.3: One Way Communication in Detail

Two Way Communication

- Recall that pipes are half-duplex, i.e, they only allow one way communication.
- Thus to achieve two way communication we need.... two pipes!
- The dotted green arrow indicates that the child processes the input in some way before writing back to the parent.

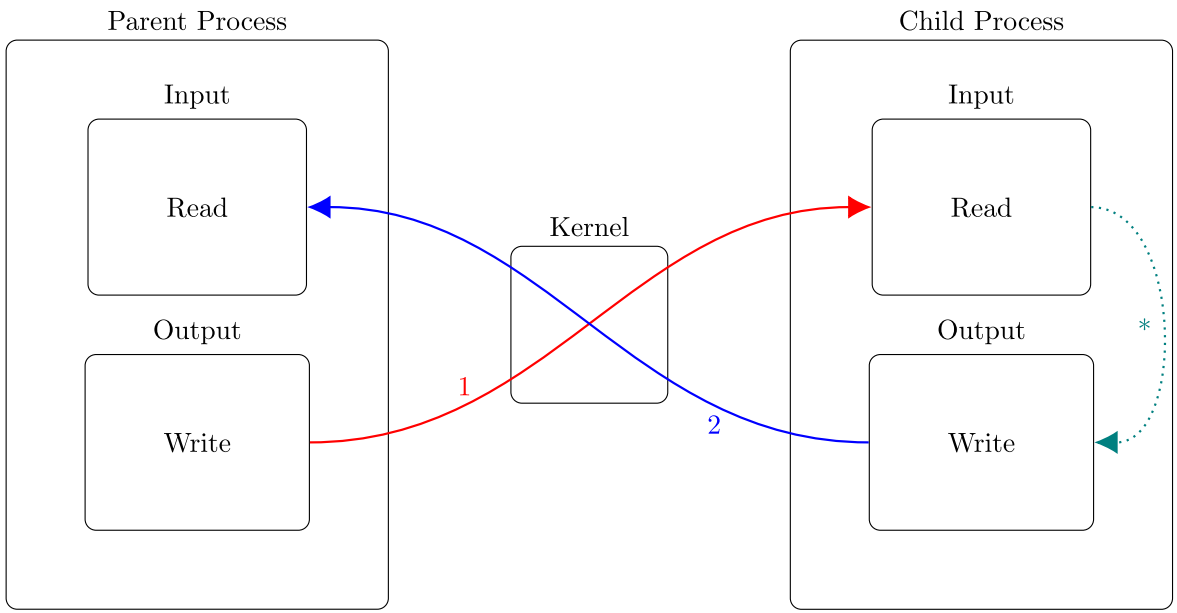


Figure 10.4: Two Way Communication

Two Way Communication in Detail

- We'll call our two pipes `fda` and `fdb` so that e.g., `fda0` is the read end of the `fda` pipe. We select `fda` to go from parent to child, and `fdb` to do the opposite.
- We now refine 10.4 on the previous page to be more detailed.

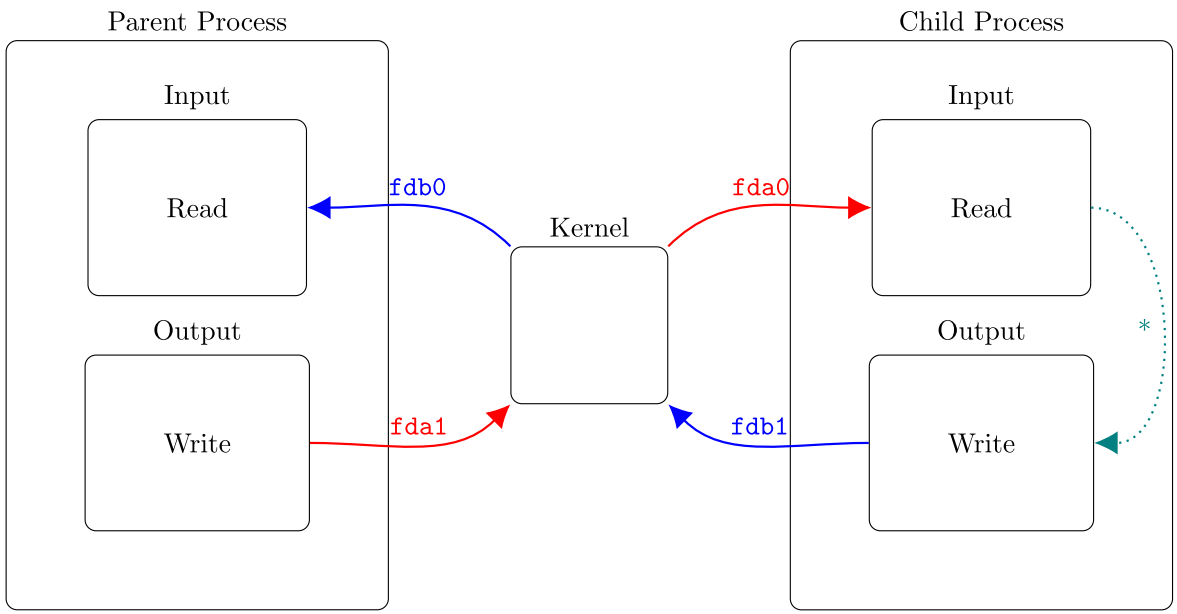


Figure 10.5: Two Way Communication in Detail

Introduction: Pipes in C

- With the theory in mind, we now discuss how to implement pipes in C.
- We will begin by having a look at the pipe entry in the the Linux Programmer's Manual.
- The manual is available on Linux Machines by typing `man` at the command line. For instance, the `pipe` documentation can be obtained via `man pipe`.
- The manual is also available online.

```
PIPE(2)                    Linux Programmer's Manual                   PIPE(2)
NAME
       pipe, pipe2 - create pipe
SYNOPSIS
       int pipe(int pipefd[2]);

       #define _GNU_SOURCE             /* See feature_test_macros(7) */
       #include <fcntl.h>              /* Obtain O_* constant definitions */
       #include <unistd.h>

       int pipe2(int pipefd[2], int flags);
DESCRIPTION
       pipe() creates a pipe, a unidirectional data channel that can be used
       for interprocess communication.  The array pipefd is used to return
       two file descriptors referring to the ends of the pipe.  pipefd[0]
       refers to the read end of the pipe.  pipefd[1] refers to the write
       end of the pipe.  Data written to the write end of the pipe is
       buffered by the kernel until it is read from the read end of the
       pipe.  For further details, see pipe(7).
```

Key Points of Pipes in C

From the manual, we see that:

- the `pipe` system call takes one argument: `pipefd`; an array of 2 integers. We'll call it `fd` for short.
- the `pipe` system call places two file descriptors (integers) into `fd` which refer to the ends of the pipe.
- Which integers are written to the `fd` array depends on what your program has done previously.
- In a program which has only created a pipe, these integers should be 3 and 4. That is to say `fd[0]==3` and `fd[1]==4`The reason is that the integers 0,1,2 are used by stdin, stdout, stderr, which are open by default in every program. Thus the next available file descriptors are 3 and 4.
- We confirm this when we implement the trivial example on on page 76.

Tips

If you forget that `0` is the read end and `1` is the write end, you can define a symbolic constant/macro to make it easier. Then you can refer to `fd[WRITE]` and `fd[READ]` instead of `fd[1]` and `fd[0]` respectivley. This approach eliminates these "magic numbers".

```
1  #define WRITE 1
2  #define READ 0
3  ...
4  int fd[2];
5  pipe(fd);
6  write(fd[WRITE])
7  read(fd[READ])
```

Implimenting the Trivial Example (10.1 on page 66)

```
1  //must include stdio.h, unistd.h, sys/types.h, string.h
2  //define 2 macros to eliminate the magic numbers 0,1
3  #define WRITE 1
4  #define READ 0
5  char mystring[] = "Hello, world";
6  // a buffer to hold the string we read from the pipe
7  char readbuffer[80];
8  void main(){
9    int fd[2];
10   pipe(fd);
11   write(fd[WRITE], mystring, (strlen(mystring)+1));
12   read(fd[READ], readbuffer, sizeof(readbuffer));
13   printf("Read: %s. New FDs: %d, %d\n", readbuffer, fd[0],fd[1])
14 }
```

```
>>>Read: Hello, world. New FDs: 3, 4
```

Implimenting One Way Communication (10.3 on page 69)

```
1   //must include stdio.h, unistd.h, sys/types.h, stdlib.h
2   void main(void) {
3       int fd[2], nbytes;
4       pid_t childpid;
5       char string[] = "Hello, world!\n";
6       char readbuffer[80];
7       pipe(fd);
8       if((childpid = fork()) == -1){
9           perror("fork");
10          exit(1);
11      } else if(childpid == 0){
12          /* Child closes read end of pipe pipe */
13          close(fd[0]);
14          /* Write "Hello, world!\n" to the write end of the pipe*/
15          write(fd[1], string, (strlen(string)+1));
16          exit(0);
17      } else {
18          /* Parent closes the write end of the pipe*/
19          close(fd[1]);
20          /* Read the string from the read end of the pipe */
21          nbytes = read(fd[0], readbuffer, sizeof(readbuffer));
22          printf("Received string: %s", readbuffer);
23      }
24  }
```

```
>>>Received string: Hello, world!
```

Exercise: Impliment Two Way Communication

As exercise


Mojtaba Bagherzadeh, Nafiseh Kahani, Cor-Paul Bezemer, Ahmed E Hassan, Juergen Dingel, and James R Cordy.
Analyzing a decade of linux system calls.
Empirical Software Engineering, 23(3):1519–1551, 2018.


Dennis Ritchie Brian W. Kernighan.
The C Programming Language.
Microsoft Press, 1988.


Sven Goldt, Sven van der Meer, Scott Burkett, and Matt Welsh.
The linux programmer's guide.


Michael L. Scott.
Programming Language Pragmatics, Third Edition.
Morgan Kaufmann, 2009.


Abraham Silberschatz, Peter B. Galvin, and Greg Gagne.
Operating System Concepts.

John WIiley & Sons INC, 2012.