ESC101N
Fundamentals of Computing

Arnab Bhattacharya
arnabb@iitk.ac.in

Indian Institute of Technology, Kanpur
http://www.iitk.ac.in/esc101/

1st semester, 2010-11

Tue, Wed, Fri 0800-0900 at L7

Sizes of variables

- The size of a variable can be found out using the `sizeof` function

```
printf(``%d\n'', sizeof(int));
```

- In this machine
  - `char`: 1 byte
  - `int`: 4 bytes
  - `float`: 4 bytes
  - `double`: 8 bytes
- Why are these required?

Sizes of variables

- The size of a variable can be found out using the `sizeof` function

```
printf(``%d\n'', sizeof(int));
```

- In this machine
  - `char`: 1 byte
  - `int`: 4 bytes
  - `float`: 4 bytes
  - `double`: 8 bytes
- Why are these required?
- To know the limits

Type	Minimum limit	Maximum limit
`char`	-128	127
`int`	-2147483648	2147483647

Type	Maximum precision	Minimum limit	Maximum limit
`float`	1.175494E-38	-3.403823E38	3.403823E38
`double`	2.225074E-308	-1.798693E308	1.798693E308

Sizes of variables

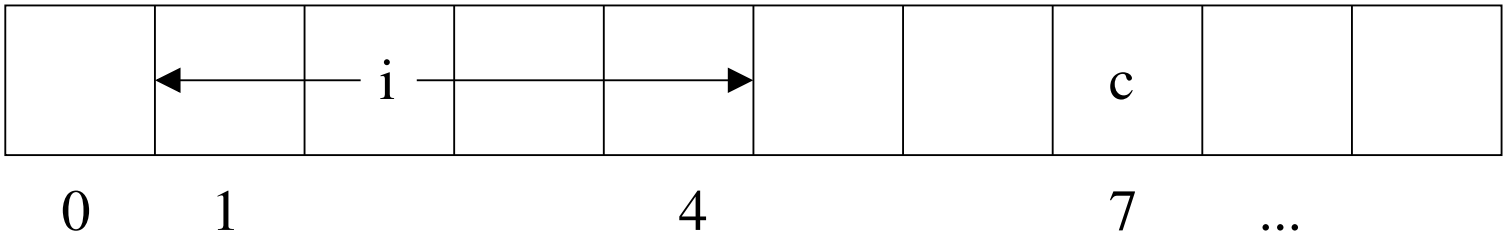
```c
#include <stdio.h>

int main()
{
    char c;
    printf("Size of character = %d bytes\n", sizeof(c));

    int i;
    printf("Size of int = %d bytes\n", sizeof(i));

    float f;
    printf("Size of float = %d bytes\n", sizeof(f));

    double d;
    printf("Size of double = %d bytes\n", sizeof(d));
}
```

Limits of variables

```c
#include <stdio.h>
#include <limits.h> // for limits on integer types
#include <float.h>  // for limits on floating-point types

void main()
{
    printf("char stores values from %d to %d\n", CHAR_MIN, CHAR_MAX);
    printf("\n");
    printf("int stores values from %d to %d\n", INT_MIN, INT_MAX);
    printf("\n");
    printf("Smallest non-zero value of type float is %e\n", FLT_MIN);
    printf("Largest value of type float is %e\n", FLT_MAX);
    printf("Smallest value of type float is %e\n", -FLT_MAX);
    printf("Smallest non-zero addition value of type float is %e\n", FLT_EPSILON);
    printf("\n");
    printf("Smallest non-zero value of type double is %e\n", DBL_MIN);
    printf("Largest value of type double is %e\n", DBL_MAX);
    printf("Smallest value of type double is %e\n", -DBL_MAX);
    printf("Smallest non-zero addition value of type double is %e\n", DBL_EPSILON);
    printf("\n");
}
```

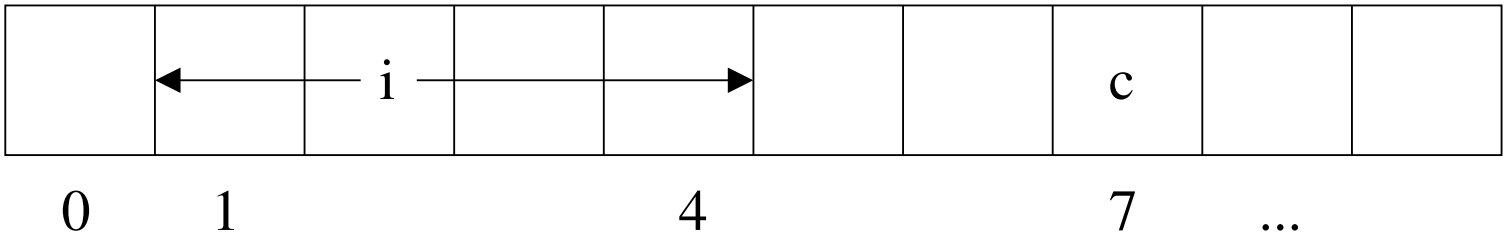
Memory model

- Memory is a list of bytes, each having a particular index called address



Memory model

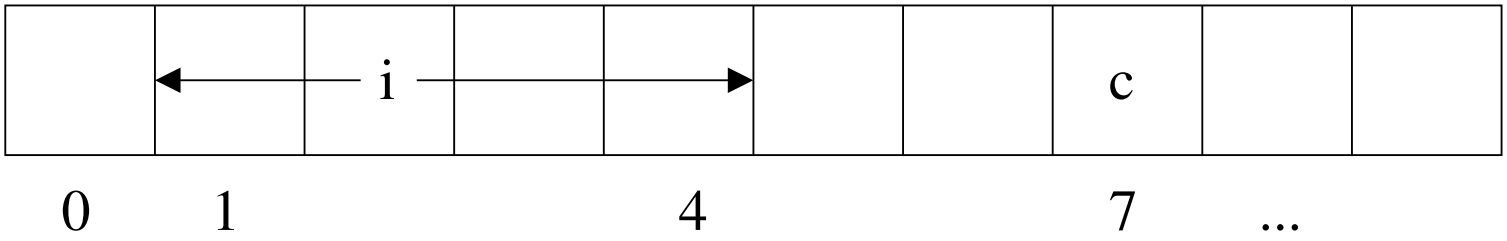
- Memory is a list of bytes, each having a particular index called address



- All variables are stored in memory
- Depending on the size, contiguous bytes are occupied
- The address of a variable is the first byte where it is stored

Memory model

- Memory is a list of bytes, each having a particular index called address



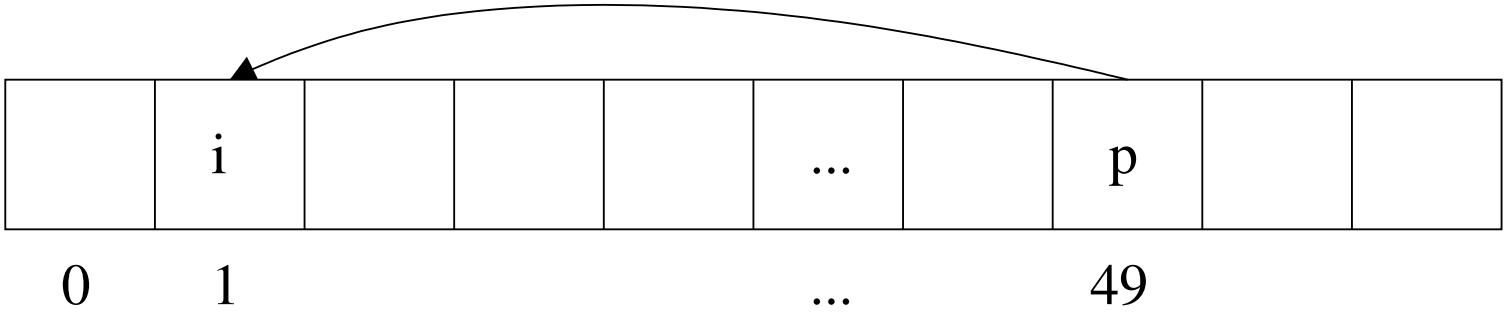
- All variables are stored in memory
- Depending on the size, contiguous bytes are occupied
- The address of a variable is the first byte where it is stored

```
int i;
char c;
```

- Address of `i` is 1
- It occupies bytes 1 through 4
- Address of `c` is 7
- It only occupies byte 7

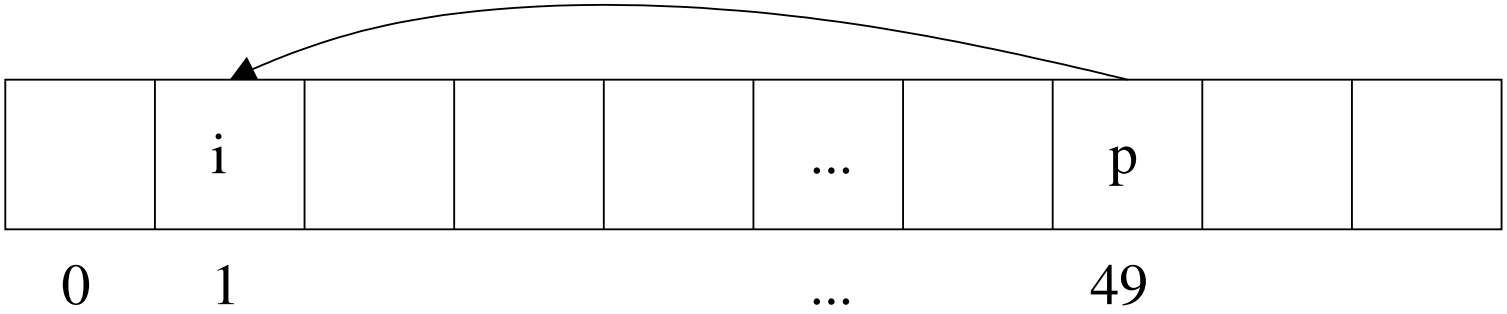
Pointer

- A pointer is a *variable* that stores the address of another variable



Pointer

- A pointer is a *variable* that stores the address of another variable



- The type of a pointer depends on the type of the variable it points to
- A pointer is denoted by the symbol *

```
int i; // i is a variable of type int
int *p; // p is a pointer to int
```

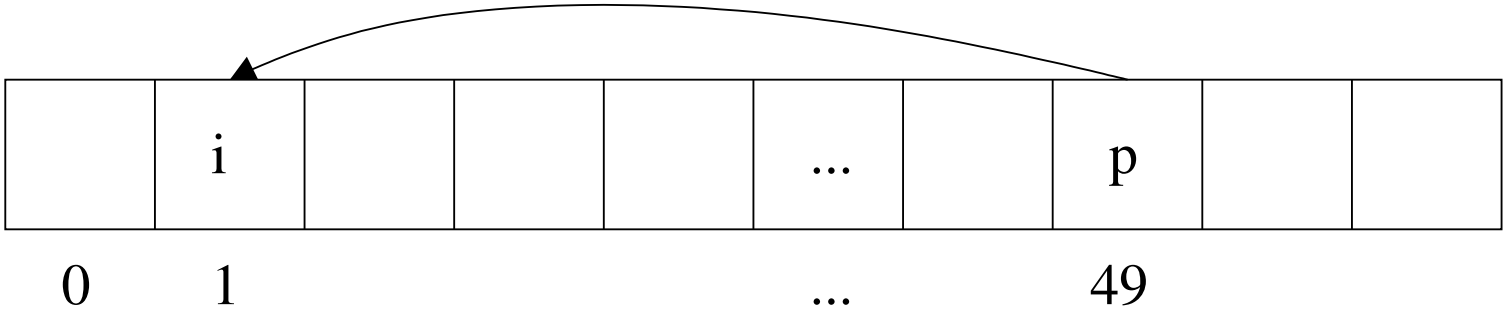
- The address of a variable is denoted by the symbol &

```
p = &i; // p is now 1 (in figure)
```

- p stores the address of i, i.e., &i
- *p denotes the content pointed by p, i.e., i

Pointer

- A pointer is a *variable* that stores the address of another variable



- The type of a pointer depends on the type of the variable it points to
- A pointer is denoted by the symbol *

```
int i; // i is a variable of type int
int *p; // p is a pointer to int
```

- The address of a variable is denoted by the symbol &

```
p = &i; // p is now 1 (in figure)
```

- p stores the address of i, i.e., &i
- *p denotes the content pointed by p, i.e., i
- The value of a pointer can be printed using %u
- The size of a pointer can be found using sizeof

Pointers

```c
#include <stdio.h>

int main()
{
    int i;
    int *p;

    printf("Size of pointer p is %d\n", sizeof(p));

    i = 5;

    p = &i;
    printf("%d\t%d\t%u\t%u\n", i, (*p), &i, p);

    *p = 7;
    printf("%d\t%d\t%u\t%u\n", i, (*p), &i, p);
}
```

Pointer operations

- Integers can be added to or subtracted from a pointer
  - Different from simple integer arithmetic
  - Adds (subtracts) in units of the size of the variable pointed to

```
int *p; // suppose p is now 1234
p++; // p is now 1234 + sizeof(int) = 1238 and not 1235
```

Pointer operations

- Integers can be added to or subtracted from a pointer
  - Different from simple integer arithmetic
  - Adds (subtracts) in units of the size of the variable pointed to

```
int *p; // suppose p is now 1234
p++; // p is now 1234 + sizeof(int) = 1238 and not 1235
```

- A pointer cannot be added to another pointer

Pointer operations

- Integers can be added to or subtracted from a pointer
  - Different from simple integer arithmetic
  - Adds (subtracts) in units of the size of the variable pointed to

```
int *p; // suppose p is now 1234
p++; // p is now 1234 + sizeof(int) = 1238 and not 1235
```

- A pointer cannot be added to another pointer
- Pointers cannot be multiplied or divided with integers or other pointers

Pointer operations

- Integers can be added to or subtracted from a pointer
  - Different from simple integer arithmetic
  - Adds (subtracts) in units of the size of the variable pointed to

```
int *p; // suppose p is now 1234
p++; // p is now 1234 + sizeof(int) = 1238 and not 1235
```

- A pointer cannot be added to another pointer
- Pointers cannot be multiplied or divided with integers or other pointers
- A pointer can be subtracted from another pointer
  - Useful in arrays

Pointer operations

- Integers can be added to or subtracted from a pointer
  - Different from simple integer arithmetic
  - Adds (subtracts) in units of the size of the variable pointed to

```
int *p; // suppose p is now 1234
p++; // p is now 1234 + sizeof(int) = 1238 and not 1235
```

- A pointer cannot be added to another pointer
- Pointers cannot be multiplied or divided with integers or other pointers
- A pointer can be subtracted from another pointer
  - Useful in arrays
- Pointers can be compared
- A pointer can be assigned the address of a variable or another pointer

Pointer operations

- Integers can be added to or subtracted from a pointer
  - Different from simple integer arithmetic
  - Adds (subtracts) in units of the size of the variable pointed to

```
int *p; // suppose p is now 1234
p++; // p is now 1234 + sizeof(int) = 1238 and not 1235
```

- A pointer cannot be added to another pointer
- Pointers cannot be multiplied or divided with integers or other pointers
- A pointer can be subtracted from another pointer
  - Useful in arrays
- Pointers can be compared
- A pointer can be assigned the address of a variable or another pointer
- A pointer can be assigned a constant integer
  - It is best to avoid such assignments

Pointer operations

- Integers can be added to or subtracted from a pointer
  - Different from simple integer arithmetic
  - Adds (subtracts) in units of the size of the variable pointed to

```
int *p; // suppose p is now 1234
p++; // p is now 1234 + sizeof(int) = 1238 and not 1235
```

- A pointer cannot be added to another pointer
- Pointers cannot be multiplied or divided with integers or other pointers
- A pointer can be subtracted from another pointer
  - Useful in arrays
- Pointers can be compared
- A pointer can be assigned the address of a variable or another pointer
- A pointer can be assigned a constant integer
  - It is best to avoid such assignments
- A special value `NULL` can be assigned to a pointer
  - It signifies that the pointer points to nothing
  - It is equivalent to 0

Pointer operations I

```c
#include <stdio.h>

int main()
{
    int i, j, *p, *q;

    printf("Assigning constant to a pointer\n");
    p = 30564378;   // warning, may result in segmentation fault
    printf("%u\n", p);
    //printf("%d\n", (*p)); // may result in segmentation fault

    printf("Assigning NULL to a pointer\n");
    p = NULL;
    printf("%u\n", p);
    //printf("%d\n", (*p)); // error

    printf("Printing a pointer\n");
    p = &i;
    printf("%u\n", p);

    printf("Incrementing a pointer\n");
    p++;
    printf("%u\t%u\n", p, (p - 3));

    printf("Printing a pointer\n");
    q = &j;
    printf("%u\n", q);

    //p = p * 2;     // error
    //q = q / p;     // error
```

Pointer operations II

```
    printf("Comparing two pointers\n");
    printf("%d\t%d\n", (p < q), (p > q));

    printf("Assigning address to pointers\n");
    p = &i;
    *p = 3;
    q = &j;
    *q = 9;
    printf("%u\t%u\t%d\t%d\t%d\t%d\n", p, q, (*p), (*q), i, j);

    printf("Assigning a pointer to another pointer\n");
    p = q;
    printf("%u\t%u\t%d\t%d\n", p, q, (*p), (*q));

    *p = 4;
    printf("%u\t%u\t%d\t%d\n", p, q, (*p), (*q));

    printf("Subtracting a pointer from another pointer\n");
    p = q + 3;
    printf("%u\t%u\t%d\t%d\n", p, q, (p - q), (q - p));

    printf("Incrementing the content of a pointer\n");
    printf("%d\t%d\t%d\n", (*p), (*(p + 1)), (*p + 1));
}
```

Pointers and arrays

- Array names are essentially pointers

```
int a[8];
```

- Array elements are stored contiguously in memory
- `a` is a pointer to the first element of the array, i.e., `a[0]`
- `*a` is equivalent to `a[0]`
- `(a + i)` is a pointer to `a[i]`
- `*(a + i)` is equivalent to `a[i]`
- Remember: `(a + i)` is actually `a + i * sizeof(int)`

Pointers and arrays

- Array names are essentially pointers

```
int a[8];
```

- Array elements are stored contiguously in memory
- `a` is a pointer to the first element of the array, i.e., `a[0]`
- `*a` is equivalent to `a[0]`
- `(a + i)` is a pointer to `a[i]`
- `*(a + i)` is equivalent to `a[i]`
- Remember: `(a + i)` is actually `a + i * sizeof(int)`
- When `a[i]` is used, compiler actually computes the address of `(a + i)` and accesses the variable in that address
  - No error checking on array boundaries is done
  - Therefore, `a[-2]`, `a[12]`, etc. become legal as they are computed to possibly valid memory addresses

Pointers and arrays

- Array names are essentially pointers

```
int a[8];
```

- Array elements are stored contiguously in memory
- `a` is a pointer to the first element of the array, i.e., `a[0]`
- `*a` is equivalent to `a[0]`
- `(a + i)` is a pointer to `a[i]`
- `*(a + i)` is equivalent to `a[i]`
- Remember: `(a + i)` is actually `a + i * sizeof(int)`
- When `a[i]` is used, compiler actually computes the address of `(a + i)` and accesses the variable in that address
  - No error checking on array boundaries is done
  - Therefore, `a[-2]`, `a[12]`, etc. become legal as they are computed to possibly valid memory addresses
- Pointers can be assigned array names

```
int *p;
p = a;
```

Passing pointers to functions

- Pointers can be passed to functions just like any other variable
- Function header must indicate that the parameters are pointers

```
int swap(int *pa, int *pb)
{
  int t = *pa; *pa = *pb; *pb = t;
}
```

- Important: If content of a pointer is changed inside the function, the change is permanent, i.e., visible even outside the function
- Reason: When a pointer is passed as an argument, a copy of the pointer is available inside the function
- However, the copy still points to the same address
- Hence, changing the content of this address changes the content of the original address
- Similar to the situation when pointers p and q are same, i.e., `p = q;` and `*p` is changed; `*q` changes automatically

Passing pointers to functions

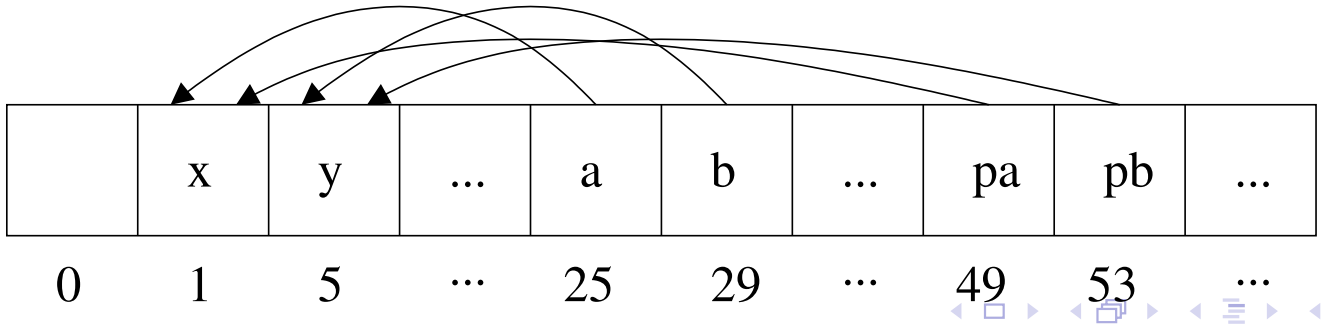
- Pointers can be passed to functions just like any other variable
- Function header must indicate that the parameters are pointers

```
int swap(int *pa, int *pb)
{
  int t = *pa; *pa = *pb; *pb = t;
}
```

- Important: If content of a pointer is changed inside the function, the change is permanent, i.e., visible even outside the function
- Reason: When a pointer is passed as an argument, a copy of the pointer is available inside the function
- However, the copy still points to the same address
- Hence, changing the content of this address changes the content of the original address
- Similar to the situation when pointers p and q are same, i.e., `p = q;` and *p is changed; *q changes automatically
- Other rules of parameter passing are followed
- Change in pointer itself is temporary

Swapping two integers

```c
#include <stdio.h>

void swap(int *pa, int *pb)
{
    int t = *pa;
    *pa = *pb;
    *pb = t;
}

int main()
{
    int x = 3, y = 4;
    int *a, *b;

    printf("x = %d, y = %d\n", x, y);

    //swap(&x, &y); // equivalent to the next three lines
    a = &x;
    b = &y;
    swap(a, b);

    printf("x = %d, y = %d\n", x, y);
}
```



scanf and printf

- scanf requires passing a pointer
- Reason: scanf requires to change the content of the variable to what has been typed using the keyboard
- Therefore, the address of the variable is passed
- This is also why scanf requires & with the variable

```
scanf(``%d'', &i);
```

scanf and printf

- scanf requires passing a pointer
- Reason: scanf requires to change the content of the variable to what has been typed using the keyboard
- Therefore, the address of the variable is passed
- This is also why scanf requires & with the variable

```
scanf(``%d'', &i);
```

- printf does not require passing a pointer
- Reason: printf does not require to change the content of the variable; it just needs to print it
- Therefore, only the variable is passed
- This is also why printf does not require & with the variable

```
printf(``%d'', i);
```

scanf and printf

- scanf requires passing a pointer
- Reason: scanf requires to change the content of the variable to what has been typed using the keyboard
- Therefore, the address of the variable is passed
- This is also why scanf requires & with the variable

```
scanf(``%d'', &i);
```

- printf does not require passing a pointer
- Reason: printf does not require to change the content of the variable; it just needs to print it
- Therefore, only the variable is passed
- This is also why printf does not require & with the variable

```
printf(``%d'', i);
```

- General rule: If a variable needs to be changed in a function, pass a pointer that holds the address of the variable and change the contents of the pointer

Passing arrays as pointers

- Since array names are pointers, arrays can be passed where pointers are required
- To access the successive elements of the array, the pointer can be incremented

Passing arrays as pointers

```c
#include <stdio.h>

void read_array(int *p, int size)
{
    int i = 0;
    while (i < size)
    {
        scanf("%d", p);
        p++;
        i++;
    }
}

void print_array(int *p, int size)
{
    int i = 0;
    while (i < size)
    {
        printf("%d\t", *p);
        p++;
        i++;
    }
    printf("\n");
}

int main()
{
    int a[] = {3, -2, 7, 19};
    print_array(a, 4);
    read_array(a, 4);
    print_array(a, 4);
}
```

Passing pointers as arrays

- Pointers can also be passed where arrays are required
- The pointer must point to the correct array address
- Pointer may point to the middle of the array, and operations may start from there

Passing pointers as arrays

```c
#include <stdio.h>

void read_pointer(int a[], int size)
{
    int i = 0;
    while (i < size)
    {
        scanf("%d", &a[i]);
        i++;
    }
}

void print_pointer(int a[], int size)
{
    int i = 0;
    while (i < size)
    {
        printf("%d\t", a[i]);
        i++;
    }
    printf("\n");
}

int main()
{
    int b[] = {3, -2, 7, 19};
    int *p = b;
    print_pointer(p, 4);
    print_pointer(p + 1, 3);
    read_pointer(p, 4);
    print_pointer(p, 4);
}
```