[Computer Architecture]

Programming in C

Pointers

Agenda

- Motivations
- Declaration of pointer
- Pointers as parameters
- Pointer and array
- Pointer arithmetic

Call-by-value may not work as expected.

```c
int swap (int x, int y) {
    int temp;

    temp = x;
    x = y;
    y = temp;
}

int main() {
    int a=0, b=1;

    printf("a=%d and b=%d", a, b);
    swap(a, b);
    printf("a=%d and b=%d", a, b);
    return 0;
}
```

Question:

- Will the values of `a` and `b` be swapped, and why?

Answer:

- It won't be able to swap a and b outside of the function `swap` because swap works on a copy of `a` and `b`.

Motivations: Variables vs. physical addresses

```c
int swap(int a[], int b[]){
    int temp;
    temp=a[0];
    a[0]=b[0];
    b[0]=temp;
    return 0;
}

int main() {
    int a[]={0}, b[]={1};
    printf("a=%d and b=%d\n", a[0], b[0]);
    swap(a, b);
    printf("a=%d and b=%d\n", a[0], b[0]);
    return 0;
}
```

Using arrays as parameters, we can implement the swap. However, it is not handy to use array alway. We need some syntax for obtaining the address of a variable.

A basic principle already mentioned several times:

- Variables are locations in the computer's memory which can be accessed by their identifier (i.e., name).

In this way, the program does *not need* to care about the physical address of the data in memory: it simply uses the identifier whenever it needs to access the variable.

Hence, whenever a variable is declared, the memory needed to store its value is assigned a specific location in the memory (i.e., its memory address).

To direct access and control over memory

An important point to remember:

- In general, the C program **does not** decide the exact memory addresses where its variables are stored ⇒ this is the task of the OS.

But, it may be useful however for the program to obtain the address of a variable at run-time in order to access cells that are at a certain position relative to it in the memory. (as we've seen in the code of swap)

Address-of-operator & and pointers

```
int main(){
    int myvar=38;
    int * ptr;

    ptr = &myvar;

    printf("%p \n", ptr);
}
```

The above code will print the address of `myvar` in hexadecimal .

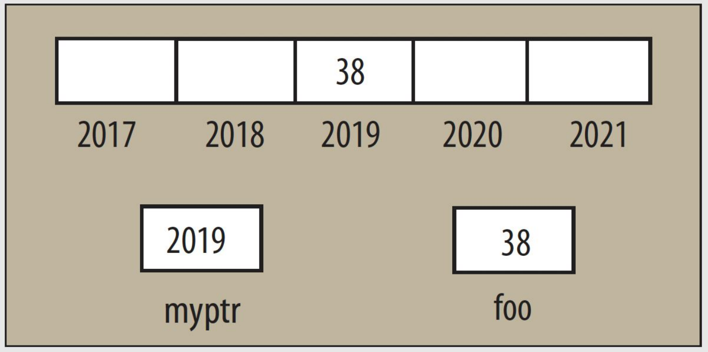
- So, the address of a variable can be obtained by preceding the name of the variable with an ampersand sign `&` known as address-of-operator: `ptr = &myvar` (**Note**: the exact address is unknown to the program before the runtime)
- A variable like `ptr` which stores the address of another variable is called a **pointer**.
- A pointer is said to be “point at” the variable whose address it stores.
- To declare a pointer variable, one should use a * in front of the variable name.
  - `int *myptrint;`
  - `char *myptrchar;`

The relation between pointers and variables

If we compile and run the C program below:

```
myvar = 38;
myptr = &myvar;
foo   = myvar;
```

We will reach the following state of the memory: (assume the address of myvar is 2019)



The address of an integer variable

By convention, the address of a variable is the address of its first memory cell:
(recall the size of different variable types)

```c
int n;
printf("%zu %zu %p", sizeof(n), sizeof(&n), &n);
```

```
4 8 0x7fff527b6b6c
```

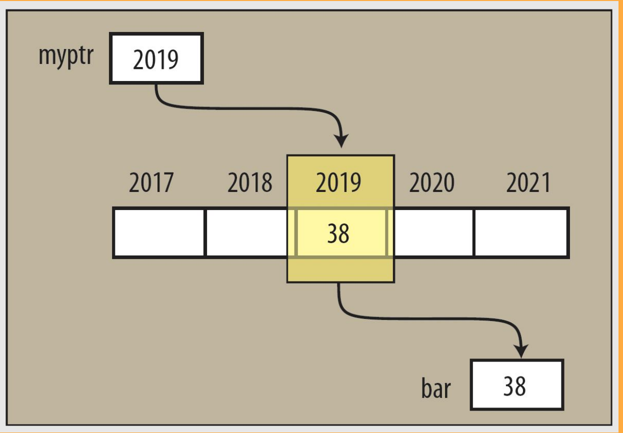
- `sizeof(n)` ⇒ the size of the integer variable n
- `sizeof(&n)` ⇒ the size of the address of the variable n
- `&n` ⇒ the address of its first memory cell printed here in hexadecimal

The dereference operator *

```c
int main(){
    int myvar=38, foo, bar;
    int * myptr;

    myptr = &myvar;
    bar = *myptr;
    printf("%d\n", bar);
}
```

The above code will print 38 because `bar` stores the value in the memory cells where `myptr` pointed at.

- `*myptr`: get the value of the address that the pointer `myptr` pointed at.
- Assume `myptr` pointed at `2019`; the relation between `myptr`, the memory cells storing the value, and `bar` is as the following figure.



Size of pointers

```c
int main() {
    int *myptrint;
    char *myptrchar;
    double *myptrdouble;

    printf("size of myptrint is %zu\n", sizeof(myptrint));
    printf("size of myptrchar is %zu\n", sizeof(myptrchar));
    printf("size of myptrdouble is %zu\n", sizeof(myptrdouble));
}
```

Question:
If a laptop is a 64-bit machine, what will the size of the pointers?

Although pointers point to variables of different data types, they occupy the **same** amount of space in memory, corresponding to the size of a memory address in the machine.

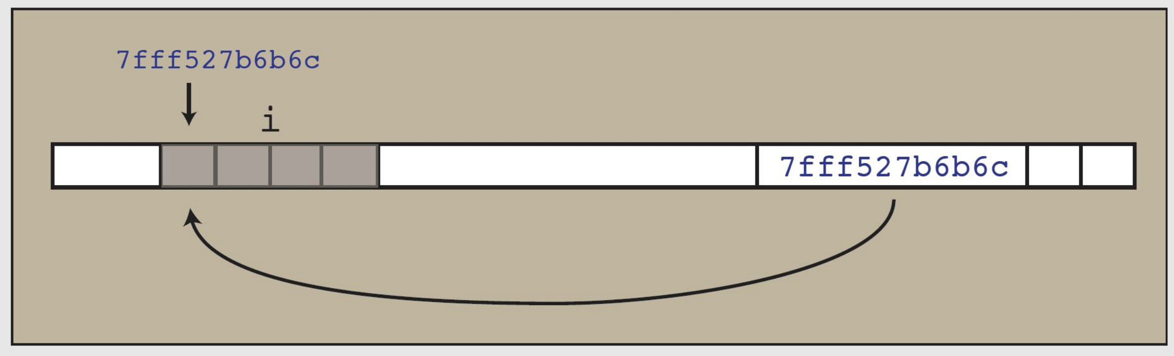
However, these variables they pointed at with different types and thus which do not occupy the same amount of space in memory.

Example: pointer syntax

```c
#include <stdio.h>

int main(){
    int i=42;
    int *p;

    p = &i; //the address of i is stored in p
    printf("value of i       : %d\n", i); //print 42
    printf("value of i, via p: %d\n", *p);//print 42

    *p = 17; //store a value in the address of i
    printf("value of i       : %d\n", i);//print 17
    printf("value of i, via p: %d\n", *p);//print 17
}
```

Basic principle

- As long as the pointer `p` points to the variable `i`, the notations `*p` and `i` are equivalent from an optional point of view (they have the same effect).



Typically, when `i` is located at the address `7fff527b6b6c`

- `i = 38;` ⇒ write 38 in the variable i at address `7fff527b6b6c`
- `*p = 38;` ⇒ write 38 at the address store in p (at `7fff527b6b6c`)

The value of a pointer can be altered

```c
int main(){
    int i=38, j;
    int *p;

    p=&i; //p pointed to i
    *p = *p+1; //it is equivalent to i=i+1
    printf("value of i : %d\n", i); //print 39
    p=&j; // p pointed to j
    *p = i+1; //it is equivalent to j = i+1;
    printf(" i : %d, j : %d\n", i, j); //print 39, 40
}
```

Two pointers can point to the same variable

```c
int main(){
    int i=17;
    int *p, *q;

    p = &i; //p pointed to i
    q = p; //the address stored in p is copied to q

    *p = *p + 1; //it is equivalent to i = i+1;
    *q = *q + 1; //it is equivalent to i = i+1;
    printf(" i : %d\n", i); //print 19
}
```

`i` incremented twice because `p` and `q` are both pointed at `i`.

Pointers as parameters of a function

```c
void increment(int *p){
    //access to the content of the variable
    //whose address is stored in the printer p
    //add 1 and store it in the same variable.
    *p = *p+1;
}

int main(){
    int i=17;

    increment(&i);

    printf("i : %d\n", i); //print 18
    return 0;
}
```

- The parameter `*p` of the function `increment` is a type of “pointer of int”.

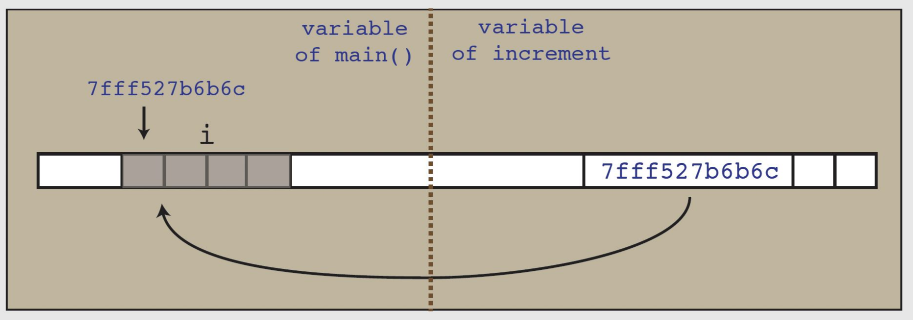
Swapping using pointer

```c
void swap(int * xptr, int* yptr){
    int temp;

    temp = *xptr;
    *xptr = *yptr;
    *yptr = temp;
}

int main() {
    int a=0, b=1;
    printf("a=%d b=%d\n", a, b);
    swap(&a, &b);
    printf("a=%d b=%d\n", a, b);
    return 0;
}
```

Swapping two values can be implemented by “call-by-reference” using pointers.

Describing the call by reference

```c
void increment(int* p){
    *p = *p+1;
}

int main(){
    int i;
    i = 0;
    printf("i = %d\n", i);
    increment(&i);
    printf("now, i is incremented by 1: i = %d\n", i);
    return 0;
}
```



In the function call of `increment`:

- the address `&i` is copied and allocated in a **new** variable `p` in `increment`.
- after the copy, the pointer `p` points on the variable `i` of main()
- the instruction `*p=*p+1` is equivalent to `i=i+1`

Pointers of pointers of pointer of …

```c
int main(){
    int i;       //int i
    int *p;      //pointer of int
    int **q;     //pointer of pointer of int
    int ***r;    //pointer of pointer of pointer of int

    p = &i;     //p points on i
    q = &p;     //q points on p
    r = &q;     //r points on q
    ***r = 17; // it is equivalant to i=17
    printf("i: %d\n", i); //print 17
    printf("p, the address of i: %p\n", p); //print the address of i
    printf("q, the address of p: %p\n", q); //print the address of p
    printf("r, the address of q: %p\n", q); //print the address of q
    printf("&r,the addres of r: %p\n", &r);//print the address of r
}
```

A pointer is a variable and is thus located at an address.

So, you can define a pointer to point at a pointer……

Example: p2p

When manipulating a 2D array, we often use the pointer of pointer to refer the matrix.

In the example, we define a pointer `ptr` that points at an array of three elements. So, it is essentially a pointer of pointer.

```c
// Function to modify the 2D array
void modifyArray(int **arr) {
    // Modify the first element of the first row
    arr[0][0] = 99;
}
int main() {
    // Declare a 2D array
    int matrix[3][3] = {
        {1, 2, 3},
        {4, 5, 6},
        {7, 8, 9}
    };
    // Declare a pointer to a pointer
    int *ptr[3];
    // Assign the address of each row to the pointer array
    for (int i = 0; i < 3; i++) {
        ptr[i] = matrix[i];
    }
    // Call the function and pass the 2D array via pointer to pointer
    modifyArray(ptr);
    // Print the modified array
    printf("Modified 2D Array:\n");
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 3; j++) {
            printf("%d ", matrix[i][j]);
        }
        printf("\n");
    }
```

Immediate initialization of pointer

The instruction:

```
int i, *p = &i, *q = p;
```

is equivalent to the sequence of declarations and assignments:

```
int i, *p, *q;

p = &i;
q = p;
```

The sizes of pointers are the same

The size of a pointer to an integer:

```
int i, *p = &i;
printf("%zu\n",sizeof(p)); // print 8
```

is the same as the size of a pointer to a double:

```
double i, *p = &i;
printf("%zu\n",sizeof(p));  // print 8
```

and the same as the size of a pointer to a char:

```
double i, *p = &i;
printf("%zu\n",sizeof(p));  // print 8
```

Correspondence between arrays and pointers

```c
int main(){

    int arr[4], *p;

    p = arr;
    p[0] = 38;
    p[1] = 0;
    p[2] = 17;
    p[3] = 12;
//now, the arr's elemeents
//are {38, 0, 17, 12}
}
```

An array variable is treated as the address of the **first byte** of the **first element** of the array.

So, `p = arr` stores the address of array to `p`.

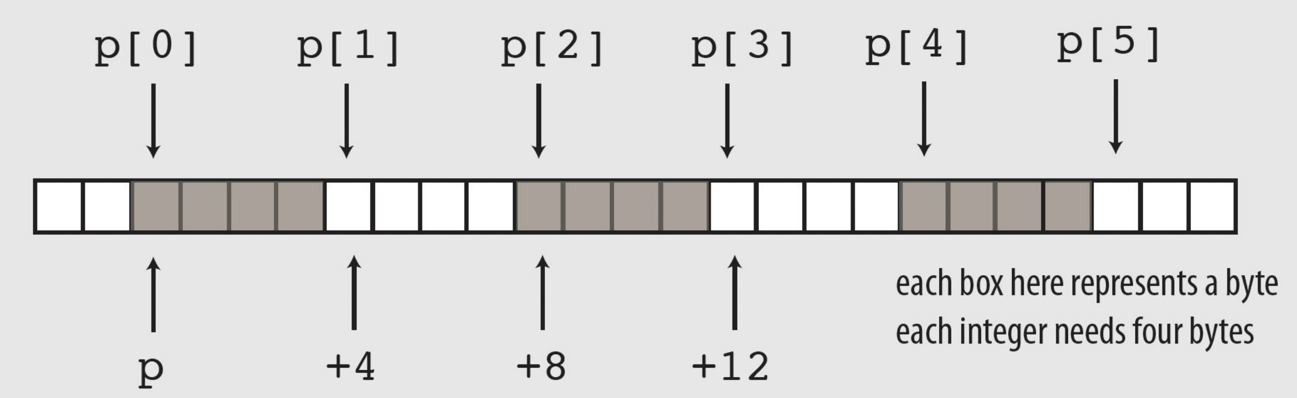
The syntax `p[i]` is for accessing the `i`-th element in the array.

Indexed notation for pointers

When a pointer `p` of type “pointer to x” contains an address, the notation `p[i]` enables one to access the address `p` shifted by `i` times the **size** of the elements of type x.

- size means the **number of bytes** necessary to allocate an element of type x.

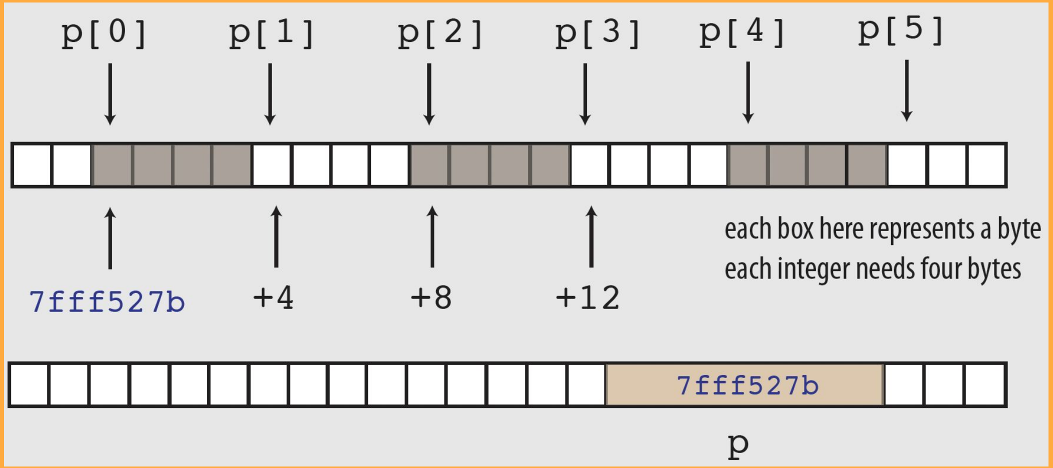
  Typical the size of an integer is 4 bytes. Hence:



Indexed notation for pointers

For an array a starting at address `7ff527b`, and the point p containing the same address:

- `p[0]` ⇒ `7fff527b`
- `p[1]` ⇒ `7fff527b + 4`
- `P[2]` ⇒ `7fff527b + 8`



Compare two arrays: always be false

What does the code below print?

```
int t[] = {38, 10, 12};
int u[] = {38, 10, 12};

if (t == u)
  printf("equal!\n");
else
  printf("different!\n");
```

```
compare_arrays.c:7:10: warning: array comparison always evaluates to false [-Wtautological-compare]
    if (t==u) {
         ^
1 warning generated.
different!
```

Compare two arrays: always be false

Evaluation of `(t==u)`:

- `t` and `u` are converted into their addresses and the equality test compares the addresses of two arrays
- The two arrays `t` and `u` have the same content but they are allocated at different addresses in the memory ⇒ `t==u` is **always be false** (i.e., 0)

So, in order to compare the content of two arrays, there is no other choice than test the elements one after the other **using a loop**.

Assignment of a variable of array: impossible!

An important exception to the conversion rule is:

- On the left of an **assignment**, a name of array is not converted to a pointer. ⇒ It thus remains of type “an array with … elements of type…”

Guess what happens when one tries to compile the following code?

```
int main(){
  int t[4] = {38, 10, 12}, u[4];
  u = t;  //  ?????
  return 0;
```

Assignment of a variable of array: impossible!

The message of clang:

```
array_assignment.c:6:7: error: array type 'int[4]' is not assignable
    u = t;
    ~ ^
1 error generated.
```

In other words, it is a error of typing…

- `u` conserves its types of an array with 4 integer elements;
- `t` is converted to is address of type “pointer of int”;

So, it is impossible to assign to the array u an expression of the appropriate type!

Conversion during compilation

During compilation, the **arrays** given as parameters to functions are **treated as pointers** and may be thus written as pointers:

```
void erase (int a[], int size){ ... }
```

is in fact equivalent to

```
void erase (int *a, int size){ ... }
```

Actually, the second notation is more usual than the first one.

- This means that the array `a` is treated as a pointer inside the function erase. Accordingly, the notation `a[i]` used for arrays in erase is in fact an indexed notation on a pointer to `int`.

Array variable is treated as pointer

```c
void print(int *a, int size){
    int i;
    for (i=0; i<size; i++) printf("%d ", a[i]);
    printf("\n");
}

void erase(int *a, int size){
    int i;
    for (i=0; i<size; i++) a[i]=0;
}

int main(){
    int array[] = {3, 38, 23, 17};

    print(array, 4);
    erase(array, 4);
    print(array, 4);
    return 0;
}
```

```c
void print(int a[], int size){
    int i;
    for (i=0; i<size; i++) printf("%d ", a[i]);
    printf("\n");
}

void erase(int a[], int size){
    int i;
    for (i=0; i<size; i++) a[i]=0;
}

int main(){
    int array[] = {3, 38, 23, 17};

    print(array, 4);
    erase(array, 4);
    print(array, 4);
    return 0;
}
```

The flexibility of using addresses

- Every object in a program is stored at some memory location. So we can obtain its address using the ampersand operator `&`.
- **Key principle:** the ampersand operator `&` can be applied to any expression which can be on the left side of an assignment.
  - In particular, `p=array;` and `p=&(array[0]);` are equivalent.
  - Also, it can be applied to array[0], array[1], etc…

```c
int main(){
    int array[5] = {0, 1, 2, 3, 4};
    int * p;

    p = &array[2]; //p is aligned with array[2]

    printf("*p       : %d \n", *p);
    printf("p[0]     : %d \n", p[0]);
    printf("array[2]: %d \n", array[2]);
    printf("p[1]     : %d \n", p[1]);
    printf("array[3]: %d \n", array[3]);
    printf("p[2]     : %d \n", p[2]);
    printf("array[4]: %d \n", array[4]);
}
```

once `p` has been assigned as `p = &(array[2]);`

- `*p` and `p[0]` and `array[2]` are equivalent
- `p[1]` and `array[3]` are equivalent
- `p[2]` and `array[4]` are equivalent

`&(array[i])` is equal to `array` shifted `i` times the size of an element of the array.

Size of array vs. size of pointer

Although the array variable is often treated as a pointer, they are not treated equally by the `sizeof`.

- `sizeof(an array)` returns (the size of an element) x (number of elements)
- `sizeof(a pointer)` returns the size of a pointer

```
int main(){
    int array[100];
    int *p;

    p = array;
    printf("size of array: %zu\n", sizeof(array));//print 400
    printf("size of p     : %zu\n", sizeof(p));//print 8
}
```

- `sizeof(array)` returns `sizeof(int)*100 = 400` bytes.
- `sizeof(p)` returns `8` bytes.

Pointer arithmetic

Pointers are addresses but essentially they are integers ⇒ we can add or subtract them with integers. For example,

- `p + i` means `shift` the pointer `p` by `i` times of the size of an element pointed by `p`.
- So, `*(p+i)` is equal to `p[i]`
- `p[i] = 38;` means `*(p+i) = 38;`

Similarly,

- for an array variable, `array[i]=38;` means `*(array+i)=38;`

Example: pointer arithmetic

```c
int main(){
    int array[3], *p;

    p = array; //p is equal to the address of array
    *p = 38;   //store 38 in array[0]
    p = p+1;   //explicit shift of p by one
               //the memory address increases by 4
               //because sizeof(int) = 4
    *p = 17;   //store 17 in array[1]
    p = p+1;   //a new shift
    *p = 3;    //store 3 in array[2]

    //now array contains {38, 17, 3}
}
```

For `p` of type `int *` and of value `7fff527b6b6c`, we have

- `p+0` is equal to `7fff527b6b6c`
- `p+1` is equal to `7fff527b6b6c + 4`
- `p+2` is equal to `7fff527b6b6c + 8`

because the size of int is 4 bytes.

Summary of the indexed access

The indexed notation is in fact a syntactic short-cut:

`p[i]` is an abbreviation for `*(p + i)`

- `p+i` is equal to `7fff527b6b6c + i*sizeof(int)`
- `*(p+i)` is equal to `p[i]`
- `p[i] = 38;` means `*(p+i) = 38;`

Similarly, `array[i]=38;` means `*(array+i)=38;`

Exercise

Assume `&array` is `1861579264` (represented in decimal for simplification), what will be printed by the following code, and why?

```c
int main(){
    double array[3], *p;

    p = array; //p is equal to the address of array
    printf("%d\n", p);
    p = p+1;
    printf("%d\n", p);
}
```