Agenda

- Review
- Memory
  - Address
- Pointers
- Pass by value/reference
- Pointers and Arrays
- Some notes on function parameters

Review

First, Let's review what we've learnt in the past few lectures.

- Arithmetics
- Variables and Types (Type Conversions)
- Conditionals and Logicals, if-else
- For/While loop
- Functions
- Arrays / `vector`
- String (C-style and `string`)

Memory

Memory, sometimes called *storage*, is the system component that remembers data values for use in computation.

Names of Memory: Addresses

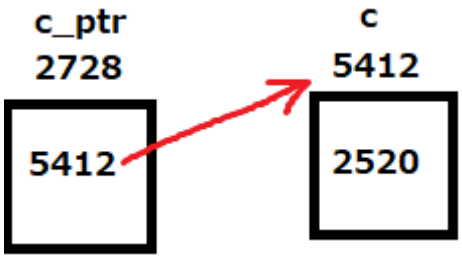
> Physical implementations of memory devices nearly always name a memory cell by the geometric coordinates of its physical storage location. It is easy to design hardware that maps geometric coordinates to and from sets of names consisting of consecutive integers (0, 1, 2, etc.). These consecutive integer names are called *addresses*, and they form the *address space* of the memory device.
>
> A memory system that uses names that are sets of consecutive integers is called a *location-addressed memory*.

In C++, every piece of data is stored in the memory. Each piece of memory has a name, which we called **address**. This address is a **number**.

All variables have unique addresses. You could find the address of an variable by using the operator `&` before it. Example:

```
1 int c = 2520;
2 int *c_ptr = &c;
```

Here, `&c` returns the address where `c` stores. Assume this address is `5412`. If you print the value of `&c` in this case, it would print out `5412`.



We introduced another variable called `c_ptr`, its type is `int *`(we'll introduce this in a minute), and it stores the address of `c`, i.e. `&c`.

Once we introduced the address of variables, we could actually do some fancy stuff. I'm going to introduce one of the most famous (~~notorious~~) feature of C++: pointers.

Pointers

A pointer is a variable that contains the address of a variable. The variable `c_ptr` we created in the previous example is exactly a pointer. The type `int *` indicates that `c_ptr` *points to* an integer value. Here, we say `c_ptr` points to the integer variable `c`.

To find the object of a pointer points, we use the unary operator `*`, called the *dereferencing* operator. For example, `*c_ptr` has the return value of `2520`, which is the content of `c`.

Let's review the usage of `&` and `*` by a snippet example:

```
int x = 1, y = 2;
int *p;                 // p is a pointer to int

p = &x;                 // p now points to x;
y = *p;                 /* *p is the value of p points, which is value of x
                           thus *p = 1. Here y is assigned to 1. */
*p = 0;                 // *p is x. Now x gets assigned to 0.
```

Note that the dereferencing operator returns the object, not a value. This is why `*p = 0` in the example above is legal: A variable can be assigned to a new value, but a constant number can't. Thus, if `int *ip = &x` where `x` is a integer variable, we could have `(*ip) += 10` indicating `x += 10`.

Note that the unary operator `&` and `*` binds more tightly than other arithmetic operators such as `*`, `+`, `++` etc. One note is that when using `++` and `--` with dereferencing `*`:

```
++*p;
(*p)++; // Same as ++*p
*p++; // Different as ++*p
```

Unary operators are associate right to left, so the third line `*p++` indicates I increase the value of `p` , instead of what `p` points to.

A pointer is constrained to a particular data type (with one exception of `void *`).

Finally, since pointers are variables, they could also be used without dereferencing. If `q` is another pointer to same type as `p`, then we can have `q = p` that copies the contents of `p` into `q`, making `p` and `q` pointing to the same object.

Pass by value/reference

The terms *pass by value* and *pass by reference* are related to functions. Recall what we learnt in functions:

```
1  void swap(int a, int b) {
2      int temp;
3      temp = a;
4      a = b;
5      b = temp;
6      return;
7  }
8  int main() {
9      int X = 10, Y = 20;
10     swap(X, Y);
11     // Here X = 10, Y = 20.
12 }
```

You could notice that the `swap` function doesn't work. Or actually, it executes, but it doesn't produce the result that we want. This is because in C++, function calls **copy** the parameters into new variables.

To make it clear, when we call `swap(X, Y)` in line 10, the interpreter copies the value 10 and 20, and assigned them to two new variables `a` and `b`. Inside the function we swapped the value of `a` and `b`, but the value didn't copy back to `X` and `Y` at the end. What's inside the `swap()` function could not interfere with the `X` and `Y` variables in `main()`. (Side note, even if the name of parameters and original variables are the same, the values won't copy back at the end.)

Does that mean we have no way to swap the two values in a new function? Not exactly. Instead of passing the values, we could try passing the *reference*, i.e. the pointers.

We make a few modifications to our `swap()`, and change the corresponding in `main()`:

```
1  void swap(int *a, int *b) {
2      int temp;
3      temp = *a;
4      *a = *b;
5      *b = temp;
6      return;
7  }
8  int main() {
9      int X = 10, Y = 20;
10     swap(&X, &Y);
11     // Here X = 20, Y = 10.
12 }
```

Now this `swap()` works perfectly. By passing the reference (a pointer to the variable), we copy the pointer's value (i.e. the address to the two variables), thus we could still refer to the original variable.

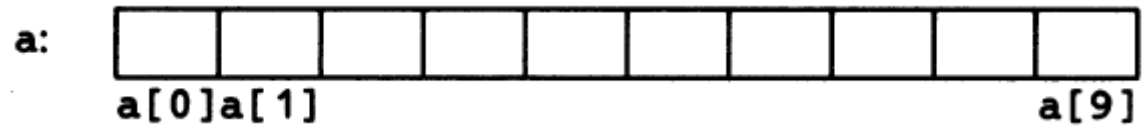
If you don't want to modify the value in the original function, you should use pass by value (i.e. the original object, not pointers); if you need to modify the value in the original function, you must use pass by reference (i.e. pointers). If it doesn't matter, you could use either.

Note that for each function call, some extra memory is needed for the *state* of the call and the new parameters. If your parameter contains some large objects (you could self-define some very large stuff), a lot of extra memory is needed, resulting your program would be "memory inefficient". So when large objects are involved, pointers are more suggested to use.

Pointers and Arrays

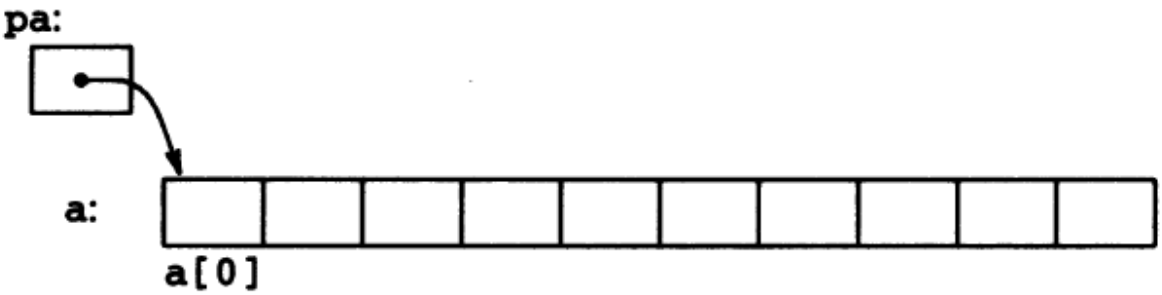
The first time you heard about the word *pointer* in this course is not here, but a few lectures earlier: when I introduced *arrays*. Here I would introduce something crazier: **An array variable is actually a pointer**. Any operation that can be achieved by array subscription (the `[]` operator) can also be done by pointers. Actually, the pointer version will in general be faster, but somewhat harder to understand.

The declaration of an array `int a[10]` defines an array `a` of size 10, from `a[0]` to `a[9]`, where the notation `a[i]` refers to the `i`-th element of the array.



Assuming we have a `int *` pointer called `pa`, and we assign `pa` to the address of `a[0]`, that is:

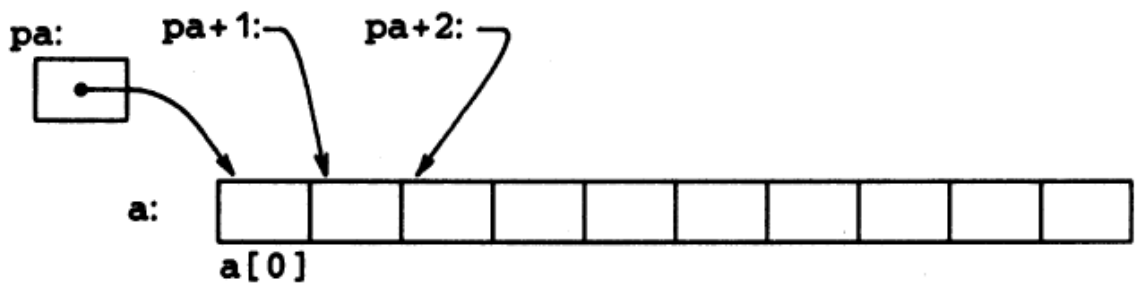
```
1 int *pa = &a[0];
```



If `pa` points to the first element of array `a`, it is very normal to define that `pa+1` points to the next element, `a[1]`. Thus, `pa + i` points to the `i`-th element after `pa`, while `pa - i` points to the `i`-th element before.

We should also be able to access the value in `a[1]` by dereferencing the pointer points to `a[1]`, in other words `*(pa+1)`. Note that `*pa+1` $\neq$ `*(pa+1)`, as the previous one means that the value of `pa` points to, and add numerical 1, while the latter one means the value of "the next element after `pa`".

Nevertheless, we could refer that `pa+i` is the address of `a[i]`, while `*(pa+i)` is the content of `a[i]`.



Note that these arithmetic are true regardless of type or size of the variables in array `a`. The meaning of "adding 1 to a pointer", and could be extended to all arithmetic operations to a pointer, is that `pa+1` points to the next object, `pa+i` points to the `i`-th object beyond `pa`.

In fact, here's what more surprising: the array name `a` itself can be used as a reference! `a` itself can be used as a reference to `a[0]` element, i.e. `a === &a[0]`.

The value of `a[i]` could be written as `*(a+i)`. When evaluating `a[i]`, C++ actually turns this to `*(a+i)` immediately: the two forms are equivalent.

This also means that applying `&` operation on both sides give the same result, thus `a+i` is equal to `&a[i]`. `a+i` is the address of the `i`-th element beyond the `a`.

The only difference I would like you to keep in mind is that the name of an array is not a pointer. A pointer is a variable, so operations like `pa = a` and `pa++` are legal. However, an array name is not a variable, so operations like `a = pa` and `a++` are *strictly illegal*.

When an array name is passed to a function, what really passed is the address of the initial element.

Let's see an example of array as a parameter: `strlen()`.

The function `strlen(char *s)`, as introduced in the lecture of C-style string functions, returns the length of a string. The end of a string is, just a reminder, `\0`, or `NULL` character.

```
int strlen(char *s) {
    int result = 0;
    int i = 0;
    while ((s+i) != '\0') {
        result++;
        i++;
    }
    return result;
}
```

Side Notes

It is possible to pass a part of an array to a function. Example, if `a` is an array, `f` is a function that takes an array as a parameter:

```
f(&a[2]);
f(a+2);
```

They both pass to the function `f` the address of subarray that starts at `a[2]`.

Within the function `f`, the parameter declaration of an array can read:

```
f(int arr[]) { ... }
f(int *arr) { ... }
```

Either works.

Some notes on function parameters

A function may have a number of parameters. The order of the parameters matters.

When calling the function, each parameter must match its order. That means, if a function has the parameter list of `(char, double, int)`, then when calling this function you should also plug in values in types `(char, double, int)`.

Let's check an example of multi-parameter call.

The function `strncpy(char *s, char *ct, size_t n)` copies the first `n` characters of string `ct` to string `s`, and then returns string `s`.

Example Usage

```
char a[] = "Hello World";
    /* When a char array is define-assigned to a const string,
       and you do not want extra memory,
       you could omit the array size as C++ could help you fill that out.
     */
cout<< strncpy ( a, "Mari", 4 ) << endl;
    // Prints out "Mario World", the first 4 characters "Hell" gets
overwritten
```

Implementation

```
char *strncpy(char *s, char *t, size_t n) {
    for (int i = 0; i < n; i++) {
        *(s+i) = *(t+i); // copy, also can write s[i] = t[i]
    }
    return s;
}
```

Homework

Homework 1. Sum of an array

请你实现函数`sum_of_array()`，计算整数数组`a`的所有元素之和。你可以假设`a`数组中任意的一个子数组它的和都不超过`int`的范围。

```
int sum_of_array(...) { // 请将...替换成合适的内容
    // 在这里编写你的代码
}

// 以下代码禁止更改
int main() {
    int N; cin>>N;
    int a[5000];
    for (int i = 0; i < N; i++) cin>>a[i];
    cout << sum_of_array(a, N) << endl;
    return 0;
}
```

样例输入1：

```
5
2 1 4 3 5
```

样例输出1：

```
15
```

Homework 2. strstr()

请你实现C字符串函数`strstr()`。`strstr(char *cs, char *ct)`的作用是，在字符串`cs`里查找是否存在字符串`ct`。如果存在，则返回在cs中第一次找到的引用。如果不存在，返回`NULL`。

另外，为了防止与系统自带的`strstr()`函数冲突，你的函数叫做`my_strstr()`。

```
#include <bits/stdc++.h>
#include <cstdio>
#include <cstring>

char *my_strstr(char *cs, char *ct) {
    // 在这里编写你的代码。提示，你可以使用上面介绍过的strlen()函数。
}

// 以下代码禁止更改
int main() {
    char a[4096], b[4096];
    memset(a, 0, sizeof a); // 这两行用于给数组a和b清0。
    memset(b, 0, sizeof b);
    gets(a); // 这行表示读入一个字符串，并读到数组a中。下一行类似。
    gets(b);
    char *first_occur = my_strstr(a, b);
    if (first_occur == NULL) {
        cout << "Not exist" << endl;
    } else {
        int diff = first_occur - a; // 想想这一行表示什么？
        cout << diff << endl;
    }
    return 0;
}
```

样例输入1：

```
Fate Grand Order
Gr
```

样例输出1：

```
5
```

样例输入2：

```
This string does not contain what I want to search.
Hello
```

样例输出2：

```
Not exist
```

样例输入3：

```
This string contains multiple n's, but only the first n is found.
n
```

样例输出3：

```
9
```

样例输入4：

```
quick fun work the fox jump a tea work the lazy dog over drag red
the
```

样例输出1：

```
15
```