Agenda

- Array
- C-style String
    - Escaping characters
    - String functions
- C++ features and containers
    - `vector`
    - `string`

Array

Array is a group of continuous memory. Array must have a unique name, and store values of the same type.

The way to define an array:

```cpp
int a[5];
```

Here we define an array, its type is `int`, and it has 5 spaces to store.

Each space is called an *element* of the array. All elements share the same type (technically, when we say *the type of array*, we actually mean the type of every single element in the array). We could access the elements by using **indexes**. Indexes start from **0**. `a[0], a[1], a[2], a[3], a[4]`

Note that we declared `int a[5]`, but the largest element we could access has the index of `[4]`.

```cpp
a[0] = 10;
a[4] = 16;
a[5] = 20; // This is out of the boundary of array, causing a FATAL ERROR called "Segmentation Fault". When this occurs, it usually means you accessed some illegal memory that you never declared.
```

In addition, once the array size is declared, it cannot be resized.

I actually forgot to tell the naming rules of variables / arrays in C++. All names in C++ must match the **naming rule of identifiers**. In other words, if it is an identifier, it can be used as a name. The rule is simple:

- An identifier is a sequence of letters and digits.
- The first character must be a letter.
- Character `_` (underscore) counts as a letter.
- Uppercase and Lowercase letters are different (i.e. *case sensitive*)
- C++ keywords are reserved for special uses thus cannot be used an identifier. (such as `for`, `if`, etc.)

As said, arrays are blocks of memory that we could use index to access. Let's see the powerfulness of array by looking at two simple examples: count occurrences, and sorting.

Example: Count occurrences of each letter

You have a piece of English paragraph made with only English characters, whitespaces and punctuation marks. Your task is to count for each English character, how many times it occurred in this paragraph. **Note that uppercase letters are considered the same as the lowercase letter**.

```cpp
string s; // We havn't introduced about the character array, i.e. string yet, but let's just assume this is an array of "char"s.
cin>>s;

int occurrence[26]; // We care about 26 different English characters
for (int i = 0; i < 26; i++) occurrence[i] = 0; // We flush the whole array to 0
// We store the occurrence of 'A'/'a' in o[0], 'B'/'b' in o[1], etc.

for (int j = 0; j < s.length(); j++) {
    // s.length() is the length of string s
    // This for loop goes over all numbers from 0 to s.length()-1, such that we could access each character by using s[j]

    if (s[j] >= 'A' && s[j] <= 'Z') {
        // s[j] is an uppercase letter
        // Its index to store is (s[j] - 'A').
        occurrence[ s[j] - 'A' ]++; // occurence[s[j]-'A'] = occurrence[s[j]-'A'] + 1;
    } else if (s[j] >= 'a' && s[j] <= 'z') {
        // s[j] is a lowercase letter
        // Similarly, index to store is (s[j] - 'a').
        occurrence[ s[j] - 'a' ]++;
    }
}

for (int i = 0; i < 26; i++) {
    cout << (char)(i+'a') << " " ; // Print out the character first
    cout << occurrence[i] << endl; // And then print out its occurrence
}
```

Example: Read in a sequence of numbers and Sort them

Here comes another challenge about array usage. We'll read a group of numbers, store them in arrays and *sort* them. Sorting an array means to change the order of the array elements such that all elements in the array satisfy some specific conditions (such as $a[i] \leq a[i+1]$). Here we make an assumption that there are N numbers in total (all integers and may repeat), and we want the array in **non-decreasing** order. ($N \leq 10^3$)

```
int a[1000 + 10]; // We usually add some extra spaces at the end. Just for safety purposes.
                // It is also very common to just do the addition and write a[1010];

int N; cin>>N; // N = how many numbers to sort
for (int i = 0; i < N; i++) {
    // We go over all indexes. For each index, we read a number and store it to the corresponding element.
    cin>>a[i];
}
```

The following operation we use to sort an array is called *Selection Sort*.

In selection sort, we go over every index i in the array. For each index i, we do the following, find the smallest element in $a[i] \ldots a[N-1]$ (i.e. from current to the tail of an array) and swap that element with the current element $a[i]$. After this loop, $a[0] \ldots a[i]$ must be the smallest $i+1$ numbers in the array, and in non-decreasing order.

I would like you to double-read the text upon to understand how we operate.

```
for (int i = 0; i < N; i++) { // For every index i
    int smallest_num = a[i], smallest_index = i; // Assume the smallest element is the current one
    for (int j = i+1; j < N; j++) { // For the remaining array
        if (smallest_num > a[j]) { // If the current a[j] is smaller
            smallest_num = a[j];
            smallest_index = j; // We replace the smallest_num with a[j] and record its index j
        }
    }
    // We swap the smallest element with the current element;
    int temp = a[i];
    a[i] = a[smallest_index];
    a[smallest_index] = temp;
}
```

And we can now do the printing.

```
for (int i = 0; i < N; i++) {
    cout<<a[i]<<" ";
}
cout<<endl;
/*
    Note that there'll be an extra space at the end of the line. If you don't want that, try the    following instead.
    for (int i = 0; i < N-1; i++) {
        cout<<a[i]<<" ";
    }
    cout<<a[N-1]<<endl;
*/
```

C-style string

Now we could talk a little about the C-style string. It's called C-style because it is used in C, and later included in C++.

A C-style string is quoted with double quotes `""`. For example, `"Hello World"` you used in the first class. This is called a string constant, meaning it is a *constant*, or a value that cannot change.

And of course we have the C-style string *variable*. You might already guessed out: it is an **array of** `char`.

```
char s[12] = "Hello World";
cout<<s<<endl; // This will print out "Hello World";
s[0] = 'h';
cout<<s<<endl; // This will print out "hello World";
s[6] = 'w';
cout<<s<<endl; // This will print out "hello world";
```

A very important point to remember is that, in C, strings are *terminated* with a **null** character. In other words, there is a character indicating the string is end, and this character is called null. It has the ASCII value of 0, usually typed as `'\0'`. This is exactly why I defined the char array *s* with size of 12, where only 11 characters are quoted - that termination character also need to take a place. In other words, `s[11] == '\0'`.

Escaping Characters

You might notice the strange character `'\0'` I just introduced. As I said earlier, probably a few lectures ago, that single quotes `''` are used to represent single characters. However `\0` has two characters.

This is because the backslash `\` in C language have a special usage. As you might notice from the ASCII chart, there are a few characters that are not visible, such as *null*, *line feed*, *backspace*, *beep*, etc. To represent these non-visible characters in C, one way is to use this backslash to **escape** the character following it, such that it express another character. There are two ways to represent any character in ASCII if known its ASCII code.

\ooo, \xhh

The three o's are octal numbers (i.e. in base 8, from 0 ~ 7). It is the octal representation of that ASCII code. For example, 'A' is 65, $(65)_{10} = (101)_8$, therefore `'\101'` is exactly `'A'`. Another example, say the *line feed* (the character indicating that the end of a line) is 10, $(10)_{10} = (12)_8$, therefore `'\12'` could represent this line feed. If you use octal representation, you could have either 1, 2, or 3 numbers after it. Side note, the using of `\0` is exactly this representation.

Another way is to use hexadecimal numbers (i.e. in base 16, 0~9 and a~f). The representation starts with an `'\x'`, and then following two hexadecimal numbers. For example, $(65)_{10} = (41)_{16}$, therefore `'\x41'` indicates `'A'`. $(10)_{10} = (A)_{16}$, therefore `'\x0a'` indicates line feed. If you use hexadecimal representation, `\x` is required.

Other frequently used non-visible characters

The two ways introduced above are usable for all characters, but in reality, some non-visible characters are more frequently used than others, such as *null* and *line feed*. In order to save us from remembering their ASCII code (which is less frequently used), these characters have some special representations, or you could understand them as privileges.

- `'\n'` - Line feed, or simply just called new line.

- `'\t'` - A horizontal *tab*
- `'\b'` - Backspace
- `'\r'` - Carriage Return (the new line character used in Windows)

Also, since single quotes `''` , double quotes `""` , and backslash `\` themselves have special meanings, if you want to use their original text, you need to escape themselves using `\`, i.e. `\'`, `\"`, `\\`.

There's actually another character that is defined to be escaped, which is the question mark `?`. If you want to know more (but not required), you may search for the keyword *trigraph*.

C string functions (optional)

There are a few functions dedicated for string operations. In C, they are in `<string.h>`, and it turns to `<cstring>` in C++. But don't worry, `<cstring>` is part of the `bits/stdc++.h`, so you don't have to include `<cstring>` separately.

Several functions involve a special type called `char *`. This is exactly what `char []` means, but it might not make much sense right now. Don't worry. They will make sense shortly once you learned pointers, which is after your midterm examination.

strlen()

```
size_t strlen(char *cs);
```

`strlen` is a function to find the length of a string. In other words, it finds from the current character, the distance between here and the following first `\0` null character.

The return type `size_t` is just a predefined function return type convention, and you may consider it as `int`.

```
cout<<strlen("Happys")<<endl; // 6
```

strcpy()

```
char *strcpy(char *s, char *ct);
char *strncpy(char *s, char *ct, size_t n);
```

Copy string `ct` to string `s`, and then returns string `s`. If `strncpy()` is used, only the first `n` characters of `ct` will be copied.

strcat()

```
char *strcat(char *s, char *ct);
char *strncat(char *s, char *ct, size_t n);
```

Concatenate `ct` after `s`, and returns string `s`. Similarly, if the `strncat()` function is used, only the first `n` characters of `ct` will be concatenated.

strcmp()

```
int strcmp(char *cs, char *ct);
int strncmp(char *cs, char *ct, size_t n);
```

Comparing two strings (variable). When comparing two strings (not in `""` which are constants), you should use this function. Return `<0` if `cs < ct`; return 0 if `cs == ct`; return `>0` if `cs > ct`.

strstr()

```
char *strstr(char *cs, char *ct);
```

Return the pointer of the first occurrence of `ct` in `cs`, or `NULL` if not present.

C++ features and containers

You may notice that everything upon are inherited from C. In fact, almost everything I introduced since the first class can be used in C programming (except `cin`, `cout`). It is true that C itself is a very convenient programming language. But the reason more programmers prefer C++ is because C++ includes what we called the *Standard Template Library*, or **STL** for short.

STL included a large number of **containers** and their following methods. These containers are powerful lifesavers. Unfortunately most of them are useful enough that they may even have bad effects on you if you use them frequently. As a result, I'll only introduce the least information about containers / functions you should know. You may get in touch with more of them once you finished the C++ syntax and a few frequently used data structures / algorithms.

vector

Vector is a container in STL. It is usually called the *C++ version of array*. A big difference from the old-fashioned array is that `vector` is able to change its length.

```
vector<int> a(50);
```

`vector` provides a few functions like `size()`, `push_back()`, `insert()`, `erase()`, etc.

The full list of `vector` usage can be found here: cplusplus.com/vector or cppreference.com/vector.

---

string

Similar to `vector`, `string` is another C++ feature. `string` is not a container, but an individual library in C++. `string` is much more convenient than the char arrays and char functions introduced in the section above.

```
string hi = "Hello World";
```

The full list of `string` usage can be found here: cplusplus.com/string or cppreference.com/string.

I would strongly suggest you to look at the references to the two C++ features upon if you have time. Only using the C array is totally fine, but I believe they are better.

Homework

复习，下周考试。