

Programming Design

C Strings

Ling-Chieh Kung

Department of Information Management
National Taiwan University

Outline

- **Characters**
- C strings
- C string processing functions

char

- **char** means a character.
 - Use **one byte** (−128 to 127) to store English letters, numbers, symbols, and special characters (e.g, the newline character).
 - Cannot store, e.g, Chinese characters.
- It is also an **integer**!
 - These characters are encoded with the **ASCII code** in most PCs.

char

- **Character literals** should be placed in between a pair of **single quotation marks**.
- A char variable can also be assigned and compared with integer values.

```
int main()
{
    char c = '0';
    cout << static_cast<int>(c) << " ";
    c = 'A';
    cout << static_cast<int>(c) << " ";
    c = '\n';
    cout << static_cast<int>(c) << " ";
    return 0;
}
```

```
int main()
{
    char c = 48;
    cout << c << " ";
    c += 10;
    cout << c << " ";
    if(c > 50)
        cout << c << " ";
    return 0;
}
```

Example 1: confirming an operation

- We may let a user enter a character to confirm the execution of some operations.
 - If 'Y' or 'y', execute.
 - Otherwise, do not execute.

```
int main()
{
    int a = 0, b = 0;
    char c = 0;

    do
    {
        cout << "Enter two integers: ";
        cin >> a >> b;
        cout << "Add? ";
        cin >> c;
    } while(c != 'Y' && c != 'y');
    cout << a + b << "\n";

    return 0;
}
```

Example 2: detecting target characters

- Sometimes we want to activate some actions upon receiving a set of target characters.
 - For example, we may want to turn all capital letters into its lowercase.
 - 26 **if-else** works.
 - Is there a better way?
- Knowing the ASCII code helps (a lot):

```
#include <iostream>
using namespace std;

int main()
{
    char c = 0;
    while(cin >> c)
    {
        if(65 <= c && c <= 90)
            cout << static_cast<char>(c + 32);
        else
            cout << c;
        cout << "\n";
    }
    return 0;
}
```

Example 2: detecting target characters

- Example 2 works, but is not very good.
 - One needs to know the ASCII code to write or understand the program.
 - What if the ASCII code is changed some day?
- The C++ standard library `<cctype>` contains some useful functions for processing characters.

Function header	Result
<code>int islower(int c);</code>	Returning 0 if <code>c</code> is not lowercase; nonzero otherwise
<code>int isupper(int c);</code>	Returning 0 if <code>c</code> is not uppercase; nonzero otherwise
<code>int tolower(int c);</code>	Returning the ASCII code of the lowercase of <code>c</code>
<code>int toupper(int c);</code>	Returning the ASCII code of the uppercase of <code>c</code>

Example 2: detecting target characters

- By using `tolower()`:

```
#include <iostream>
#include <cctype>
using namespace std;

int main()
{
    char c = 0;
    while(cin >> c)
        cout << static_cast<char>(tolower(c)) << "\n";
    return 0;
}
```


Character processing functions

- There are many other useful functions.

Function header	Return value
<code>int isalpha(int c);</code>	0 if <code>c</code> is not a letter; nonzero otherwise
<code>int isdigit(int c);</code>	0 if <code>c</code> is not a digit; nonzero otherwise
<code>int isalnum(int c);</code>	0 if <code>c</code> is neither a letter nor a digit; nonzero otherwise
<code>int isprint(int c);</code>	0 if <code>c</code> is not printable; nonzero otherwise
<code>int isspace(int c);</code>	0 if <code>c</code> is not a space; nonzero otherwise
<code>int ispunct(int c);</code>	0 if <code>c</code> is not a punctuation mark; nonzero otherwise

- Question: Why not return `bool`? Why `int`?

Example 3: ASCII code table

- Let's print out the ASCII code table for printable characters:

```
#include <iostream>
#include <iomanip>
#include <cctype>
using namespace std;
```

```
int main()
{
    cout << "  0123456789\n";
    for(int i = 30; i <= 126; i++)
    {
        if(i % 10 == 0)
            cout << setw(2) << i / 10 << " ";
        if(isprint(i))
            cout << static_cast<char>(i);
        else
            cout << " ";
        if(i % 10 == 9)
            cout << "\n";
    }
    return 0;
}
```

Outline

- Characters
- **C strings**
- C String processing functions

String literals

- In many applications, we need some ways to handle **strings**.
- E.g., in an address book application, if we do not have strings:
 - We cannot store names.
 - We cannot store phone numbers.
 - We cannot store addresses.
- Strings can be implemented in two ways:
 - C strings as **character arrays**.
 - C++ strings as **objects**.
- Let's introduce C strings today.

String

- We have already used a string with **cout**:

```
cout << "Hello world";
```

- "Hello world" is a string.
- A **string literal** is contained in a pair of **double** quotation marks.
- A C string (variable) is **declared** as a character array.
 - `char s[10];`
- A character array can be **initialized** as a usual array.
 - `char s[10] = {0};`
 - `char s[10] = {'a', 'b', 'c'};`

Example: entering a string until

- Consider the following program:
 - Let a user enter a string ending with a # symbol.
 - Then print out the string (excluding #).
- The character array is simply used as a usual array.

```
#include <iostream>
using namespace std;

const int LEN = 10;
int main()
{
    char s[LEN] = {0};
    int n = 0;
    do
    {
        cin >> s[n];
        n++;
    } while(s[n - 1] != '#' && n < LEN);
    for(int i = 0; i < n - 1; i++)
        cout << s[i];
    return 0;
}
```

A special way to input a string

- Because they are used for string, character arrays are **special**.
 - Many operations are **overloaded** for character arrays in a special way.
 - In particular, **cin >>** and **cout <<**.
- For an array **A**, if we do **cin >> A**:
 - If **A** is of other types, this is not allowed.
 - But for a character array, this allows us to input the string.

```
char str[10];  
cin >> str; // if we type "abcde"  
cout << str[0]; // a  
cout << str[2]; // c
```

A special way to output a string

- For an array **A**, if we do `cout << A`:
 - If **A** is of other types, this will print out its memory address.
 - But for a character array, this prints out the whole string (some exceptions will be discussed later).

```
int values[5] = {0};  
cout << values; // an address  
char str[10];  
cin >> str; // if we type "abcde"  
cout << str; // abcde
```

- But wait... How does the `cout <<` operation know **when to stop**?
 - Why not printing out ten characters?
 - Or does it print out ten characters but we do not see it?

The null character

- When we use `cin >>` to input a string, a **null character** `\0` will be appended at the end **automatically**.
 - `\0` is an escape sequence. It marks the **end of a string**.
 - Its ASCII code is 0.
 - It is `\0`, not `\o` or `\O`.
- When you declare a character array of length n , you can store a string of length **at most $n - 1$** .
- A C string may be **initialized** with a double quotation.
 - `char s[100] = "abc";`
 - The assignment operator is **overloaded** for character arrays.
 - A null character will also be appended if a C string is initialized in this way.

Understanding the null character

- From the system's perspective, a null character marks the end of a string.
 - In particular, << is implemented to print out characters up to \0.

```
char a[100] = "abcde FGH";  
cout << a << "\n"; // abcde FGH  
char b[100] = "abcde\0 FGH";  
cout << b << "\n"; // abcde
```

- Recall that one may also initialize a C string by assigning multiple characters.
 - `char s[100] = {'a', 'b', 'c'};`
 - **No** null character will be appended (though uninitialized values will be initialized to 0, which is the null character).
 - `=` is overloaded for “a C string” and “some characters” in different ways.

Comparisons

- We have two ways to initialize a C string:

Example	Will a null character be appended?
<code>char s[10] = "abc";</code>	Yes
<code>char s[100] = {'a', 'b', 'c'};</code>	No

- We have two ways to input a C string:

Example	Will a null character be appended?
<code>cin >> s;</code>	Yes
<code>cin >> s[0];</code>	No

Example: entering a string until `\n`

- Consider the following program:
 - Let a user enter a string ending with a `\n` symbol.
 - Then print out the string (excluding `\n`).
- The character array is used with overloaded `cin >>` and `cout <<`.

```
#include <iostream>
using namespace std;

const int LEN = 10;
int main()
{
    char s[LEN] = {0};
    cin >> s;
    cout << s << "\n";
    return 0;
}
```

String assignments

- Assignments with double quotations are allowed only for initialization.

```
char s[100];  
s = "this is a string"; // compilation error!
```

- After all, **s** stores a memory address.
- One may assign values to a string by assigning values to individual characters.

```
s[0] = 'A';  
s[1] = 'B';  
s[2] = 'C';
```

String assignments

- How to explain the outputs of this program?

```
char c[100] = {0};  
cin >> c; // "123456789"  
cin >> c; // "abcde";  
cout << c << "\n"; // "abcde"  
c[5] = '*';  
cout << c << "\n"; // "abcde*789"
```

Array boundary

```
char a[5] = {0};  
cin >> a; // "123456789"  
cout << a; // "123456789" or an error
```

- C++ does not check **array boundary**!
- We may or may not touch those memory spaces used by other programs/variables.
 - If a protected space is touched, an error occurs and our program is shutdown.
 - If not, **cout <<** prints out **the whole string** until the **end of a string**, which is marked by a `\0`.

A strange case

```
char a1[100] = {0};  
cin >> a1; // "this is a string"  
cout << a1; // "this"
```

- Is it because that a white space is treated as an end of C strings?
- No!

```
char a2[100] = {'a', 'b', ' ', 'c', '\\0', 'e'};  
cout << a2; // ab c
```

- Then why?

cin >> vs. cin.getline()

- **cin >>** **splits** the input stream into pieces according to **white spaces**.
 - The same thing happens for **the newline character** and **tab**.
- To input a string with white spaces, use **cin.getline()**.
 - A instance function of the object **cin** (to be introduced later in this semester).
 - It splits the input stream according to **newline characters** only.
 - A null character is appended.

```
char a[100] = {0};  
char b[100] = {0};  
cin >> a >> b; // this is  
cout << a << "\n"; // this  
cout << b << "\n"; // is
```

```
char a[100];  
cin.getline(a, 100); // Hi, it's me  
cout << a << "\n"; // Hi, it's me
```

Example: counting the number of spaces

- Several strings are contained in a text file.
 - Each string is put in one line.
 - Each string has at most 100 characters.
- For each input string, we want to count the **number of spaces** in it (which is the number of words minus one in some applications).
- Challenges:
 - If consecutive spaces are considered as only one space, how to count the number?
 - What if `cin.getline(a, 100)` is replaced by `cin >> a`?

```
char a[100] = {0};
while(cin.getline(a, 100))
{
    int i = 0;
    int spaceCount = 0;

    while(a[i] != '\0')
    {
        if(a[i] == ' ')
            spaceCount++;
        i++;
    }
    cout << spaceCount << "\n";
}
```

String arrays

- A character array represents a string.
- A **two-dimensional character array** represents a set of strings.
 - This may also be called a **string array**.
 - Each one-dimensional array is a C string (with a `\0` at the end).

```
char name[4][10] = {"John", "Mikasa", "Eren", "Armin"};
cout << name << "\n"; // an address
cout << name[1] << "\n";
cin >> name[2];
cout << name[2][0] << "\n";
```

C strings as character pointers

- Recall that a pointer may point to an array.
- Therefore, a **character pointer** may also represent a C string.

```
char s[100] = "12345";  
char* p = s;  
cout << p << "\n";  
cin >> p; // or s  
cout << s; // or p
```

- More interestingly:

```
char s[100] = "12345";  
char* p = s;  
cout << p + 2 << "\n";
```

A pointer is not an array

- Still, a pointer is not an array.
 - In particular, **no space** has been allocated to store values.
- When we use a character pointer as a C string:
 - We may write values into an array through the pointer.
 - We cannot write values if there is no space allocated.
- Just like usual arrays and pointers:

```
char* p;  
cin >> p; // run-time error!
```

```
cout << sizeof(a); // 100  
cout << sizeof(p); // 8
```

String literals and character pointers

- A character pointer may also be **initialized** as a string literal.

```
char* p = "12345";  
cout << p + 2 << "\n"; // 345
```

- When we do so, the system allocates space storing “12345”.
- That space is **read-only**.
- **p** stores the address of that space.

String literals and character pointers

- In fact, one may assign a string literal to a character pointer **at any time**.

```
char a[100] = {0};  
a = "123"; // compilation error
```

```
char* p;  
p = "abc"; // okay
```

- What will happen is the same as assigning the string literal at initialization.
- Note that the pointer now points to a different (read-only) place:

```
char a[100] = "12345";  
char* p = a;  
p = "abc"; // does not affect a  
cout << p << "\n"; // abc  
cout << a << "\n"; // 12345
```

```
char a[100] = "12345";  
char* p = a;  
p = "abc";  
cout << p << "\n";  
cin >> p; // run-time error
```

Passing a string to a function

- To pass a string to a function, let the parameter type be a character pointer or a character array. Then pass **an address** to it.

```
void reverse(char p[])
{
    int n = strlen(p);
    char* temp = new char[n];
    for(int i = 0; i < n; i++)
        temp[i] = p[n - i - 1];
    for(int i = 0; i < n; i++)
        p[i] = temp[i];
    delete [] temp;
}
void print(char* p)
{
    cout << p << "\n";
}
```

```
#include <iostream>
#include <cstring>
using namespace std;

void reverse(char p[]);
void print(char* p);
int main()
{
    char s[100] = "12345";
    reverse(&s[1]);
    print(s);
    return 0;
}
```


Main function arguments

- In fact, we may pass arguments to the main function.

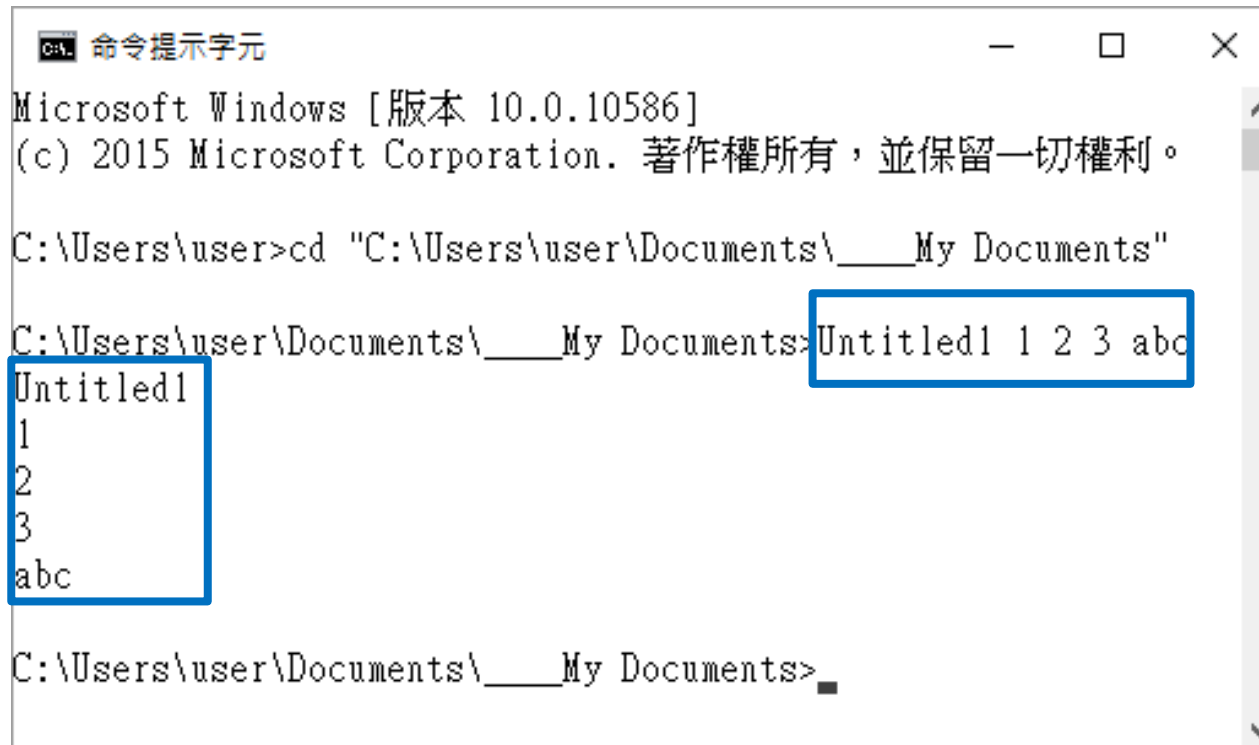
```
#include <iostream>
using namespace std;

int main(int argc, char* argv[])
{
    for(int i = 0; i < argc; i++)
        cout << argv[i] << "\n";
    return 0;
}
```

- **argv[0]** is the name of the executable file.
- **argv[i]** is the *i*th string passed into **main**.

Main function arguments

- To pass a string, add it behind the execution statement.



```
命令提示字元
Microsoft Windows [版本 10.0.10586]
(c) 2015 Microsoft Corporation. 著作權所有，並保留一切權利。
C:\Users\user>cd "C:\Users\user\Documents\__My Documents"
C:\Users\user\Documents\__My Documents>Untitled1 1 2 3 abc
Untitled1
1
2
3
abc
C:\Users\user\Documents\__My Documents>
```

A dynamic character arrays as a C string

- A dynamic character array (pointed by a pointer) can store a C string.

```
char* p = new char[100];  
cin >> p;  
cout << p;  
delete p;
```

- Be careful when you use dynamic arrays:

```
char* p = new char[100];  
cin >> p;  
cout << p << "\n";  
p = "123"; // memory leak  
cout << p;  
delete [] p; // run-time error
```

Outline

- Characters
- C strings
- **C string processing functions**

C String processing functions

- The C++ standard library `<cstring>` contains many useful **pointer-based** string processing functions.
 - Query: **strlen**.
 - Searching: **strchr**, **strstr**.
 - Concatenation: **strcat**, **strncat**.
 - Copying: **strcpy**, **strncpy**.
 - Comparison: **strcmp**, **strncmp**.
 - Splitting: **strtok**.
- The C++ standard library `<cstdlib>` contains some more.
 - String-number conversion: **atoi**, **atof**.

String length query

- The function `strlen` returns the **string length** of a given C string.

```
unsigned int strlen(const char* str);
```

- It returns the number of characters between `str` and the first `'\0'`.
- It accepts a string literal, a character pointer, and a static character array.

```
char* p = new char[100];  
cin >> p;  
cout << strlen(p) ;  
p[3] = '\0';  
cout << strlen(p + 1) ;  
delete [] p;
```

```
char* p = "12345";  
cout << strlen(p) << "\n";  
char a[100] = "1234567";  
cout << strlen(a) << "\n";
```

String length query

- Note the difference between **strlen** and **sizeof**!

```
char* p = "12345";  
cout << strlen(p) << "\n";  
char a[100] = "1234567890";  
cout << strlen(a) << "\n";  
cout << sizeof(a) << "\n";  
cout << sizeof(a + 2) << "\n";
```

- **strlen(p)** returns 5 because the length of the string pointed by **p** is 5.
- **strlen(a)** returns 10 because the length of the string contained in **a** is 10.
- **sizeof(a)** return 100 because that is the size of **a**.
- **sizeof(a + 2)** returns 8. why?

Example: counting the number of spaces

- Recall that we want to count the **number of spaces** in a given string.
- With **strlen**, we may do this better.

```
char a[100] = {0};
while(cin.getline(a, 100))
{
    int spaceCount = 0;
    for(int i = 0; i < strlen(a); i++)
    {
        if(a[i] == ' ')
            spaceCount++;
    }
    cout << spaceCount << "\n";
}
```


Searching in a string

- To find **the location of a character** in a string (or conclude that it does not exist in the string), use **strchr**.

```
char* strchr(char* str, int character);
```

- It returns the **address** of the first occurrence of the character.
 - If the character does not exist, it returns **nullptr**.

```
char a[100] = "1234567890";  
char* p = strchr(a, '8');  
if(strchr(a, 'a') == nullptr)  
    cout << "!!!\n";  
cout << strchr(a, '4') << "\n";  
cout << strchr(a, '4') - a;
```

Searching for the next occurrence

- Two advanced techniques:
 - The returned address may be used to **modify** the given string.
 - The returned address may be used as the starting location of a “new string” to search for **the next occurrence** of the character.
- The following program replaces all white spaces to underlines.

```
char a[100] = "this is a book";
char* p = strchr(a, ' ');
while(p != nullptr)
{
    *p = '_';
    p = strchr(p, ' '); // why p?
}
cout << a;
```

Searching for a substring

- If we want to search for a **substring**, we use **strstr**.

```
char* strstr(char* str1, const char* str2);
```

- This returns the **address** of the first occurrence of **str2** in **str1**.

```
char a[100] = "this is a book";  
char* p = strstr(a, "is");  
while(p != nullptr)  
{  
    *p = 'I'; // not p = "IS"!  
    *(p + 1) = 'S';  
    p = strstr(p, "is");  
}  
cout << a;
```

String copying

- In the previous example, replacing a substring in a string requires multiple character modifications.
- We may do it at once with `strcpy`.

```
char* strcpy(char* dest, const char* source);
```

- It copies the string at **source** into the array at **dest**, including the terminating null character in **source**. It returns **dest**.

```
char a[100] = "watermelon";  
char b[100] = "orange";  
cout << a << "\n";  
strcpy(a, b);  
cout << a << "\n";
```

```
char a[100] = "watermelon";  
char b[100] = "orange";  
cout << a << "\n";  
strcpy(a, b);  
cout << a + 7 << "\n"; // ?
```

String copying

- Let's modify the previous program:

```
char a[100] = "this is a book";
char* p = strstr(a, "is");
while(p != nullptr)
{
    strcpy(p, "IS");
    p = strstr(p, "is");
}
cout << a;
```

- It does not work! Why? How to fix it?

String concatenation

- A similar task is to **concatenate** two strings. We may use **strcat** to do this.

```
char* strcat(char* dest, const char* source);
```

- This copies **source** to **the end of dest**. The `\0` of **dest** is replaced by the first character of **source**. The `\0` of **source** is also copied. It returns **dest**.

```
char a[100] = "watermelon";  
char b[100] = "orange";  
cout << a << "\n";  
strcat(a, b);  
cout << a << "\n";
```

Preparing enough space

- **strcpy** and **strcat** modifies the destination array.
- A programmer must make sure that there is **enough space** of the modification.

```
char a[15] = "watermelon";  
char b[100] = "orange";  
cout << a << "\n";  
strcat(a, b); // dangerous!  
cout << a << "\n";
```

- The destination must be an array (static or dynamic), not **just a pointer**.

```
char* a;  
char b[100] = "orange";  
strcat(a, b); // dangerous!
```

Preparing enough space

- To help prevent run-time error, two additional versions are provided:

```
char* strncpy(char* dest, const char* source, unsigned int num);  
char* strncat(char* dest, const char* source, unsigned int num);
```

- If **source** has at most **num** characters, only **the first num characters** in source are copied. **No** `\0` is copied into **dest**.
- If **source** has fewer than **num** characters, **\0 will be padded** so that in total **num** characters are copied.

```
char a[15] = "watermelon";  
char b[100] = "orange";  
strncat(a, b, sizeof(a) - strlen(a) - 1);  
cout << a << "\n";
```


String comparisons

- Strings may also be **compared alphabetically** (consider your dictionaries!).
- We may use **strcmp** and **strncmp** to compare two strings.

```
int strcmp (const char* str1, const char* str2);  
int strncmp(const char* str1, const char* str2, unsigned int num);
```

- They returns 0 if the two strings are identical, a negative number if **str1** is in front of **str2**, and a positive number if **str2** is in front of **str1**.
- **strcmp** compares the whole **str1** and **str2**.
- **strncmp** compares up to **num** characters.

Example: sorting names alphabetically

- Given a set of names, let's sort them alphabetically.
- For example:
 - Before: (John, Mikasa, Eren, Armin).
 - After: (Armin, Eren, John, Mikasa).
- Strategy:
 - We may implement, e.g., bubble sort.
 - To compare two names, we use **strcmp**.
 - When we want to swap two names, we use **strcpy**.

Example: sorting names alphabetically

```
#include <iostream>
#include <cstring>
using namespace std;

const int CNT = 4;
const int LEN = 10;

void swapName(char* n1, char* n2)
{
    char temp[LEN] = {0};
    strcpy(temp, n1);
    strcpy(n1, n2);
    strcpy(n2, temp);
}
```

```
int main()
{
    char name[CNT][LEN]
        = {"John", "Mikasa", "Eren", "Armin"};

    for(int i = 0; i < CNT; i++)
        for(int j = 0; j < CNT - i - 1; j++)
            if(strcmp(name[j], name[j + 1]) > 0)
                swapName(name[j], name[j + 1]);

    for(int i = 0; i < CNT; i++)
        cout << name[i] << " ";

    return 0;
}
```

Example: sorting names alphabetically

- That implementation works, but may be improved.
 - A lot of **strcpy** makes the implementation inefficient.
 - Let's **swap pointers** instead.
- Strategy:
 - Create pointers pointing to names.
 - When we find that two names should be swapped, we swap the corresponding pointers, i.e., **exchanging the addresses** contained in them.

Example: sorting names alphabetically

```
#include <iostream>
#include <cstring>
using namespace std;

const int CNT = 4;
const int LEN = 10;

void swapPtr(char*& p1, char*& p2)
{
    char* temp = p1;
    p1 = p2;
    p2 = temp;
}
```

- Why calling by reference?
- Is **name** modified?

```
int main()
{
    char name[CNT][LEN]
        = {"John", "Mikasa", "Eren", "Armin"};
    char* ptr[CNT]
        = {name[0], name[1], name[2], name[3]};

    for(int i = 0; i < CNT; i++)
        for(int j = 0; j < CNT - i - 1; j++)
            if(strcmp(ptr[j], ptr[j + 1]) > 0)
                swapPtr(ptr[j], ptr[j + 1]);

    for(int i = 0; i < CNT; i++)
        cout << ptr[i] << " ";

    return 0;
}
```

Splitting a string into substrings

- We often want to **split a string into substrings** based on some characters.
 - E.g., to split a sentence into words based on spaces and punctuation marks.
 - E.g., to split a comma-separated row into attributes based on commas.
 - These characters are called **delimiters**. These substrings are called **tokens**.
- Let's write a program that split an URL into tokens based on '.' and '/'.
 - Input: `www.im.ntu.edu.tw/~lckung/courses/PD16`
 - Output: `www im ntu edu tw ~lckung courses PD16`
- We may implement this by using **strchr**.
 - A better way is to use **strtok**.

```
char* strtok(char* str, const char* delimiters );
```

Example: splitting an URL

```
#include <iostream>
#include <cstring>
using namespace std;

const int CNT = 100;
const int WORD_LEN = 50;
const int SEN_LEN = 1000;

int main()
{
    char url[SEN_LEN];
    char delim[] = ".\\\"";
    char word[CNT][WORD_LEN] = {0};
    int wordCnt = 0;
    cin >> url;
```

```
    char* start = strtok(url, delim);
    while(start != nullptr)
    {
        strcpy(word[wordCnt], start);
        wordCnt++;
        start = strtok(nullptr, delim);
    }

    for(int i = 0; i < wordCnt; i++)
        cout << word[i] << " ";

    return 0;
}
```

Splitting a string into substrings

- The function **strtok** is special.

```
char* strtok(char* str, const char* delimiters );
```

- At the first invocation (assuming the first character is not a delimiter):
 - **str** is the beginning of the string to be split, and **delimiters** is a character array containing delimiter characters.
 - The first delimiter will be found and replaced by `\0`.
 - The returned pointer stores the address of the first token (which is **str**).
 - The **location** of the first non-delimiter character right after the first delimiter is **recorded internally** in the function for future uses.

Splitting a string into substrings

- The function **strtok** is special.

```
char* strtok(char* str, const char* delimiters );
```

- At subsequent invocations:
 - **str should be nullptr.**
 - The **internally recorded starting location** is automatically used as the starting point.
 - The next delimiter is replaced by **\0**.
 - The returned pointer stores the address of the current token (which is that starting location internally stored before this invocation).

Visualization of string splitting

i	m	.	n	t	u	.	e	d	u	.	t	w	/	~	l	c	k	u	n	g	\0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	----

i	m	\0	n	t	u	.	e	d	u	.	t	w	/	~	l	c	k	u	n	g	\0
---	---	----	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	----

i	m	\0	n	t	u	\0	e	d	u	.	t	w	/	~	l	c	k	u	n	g	\0
---	---	----	---	---	---	----	---	---	---	---	---	---	---	---	---	---	---	---	---	---	----

i	m	\0	n	t	u	\0	e	d	u	\0	t	w	/	~	l	c	k	u	n	g	\0
---	---	----	---	---	---	----	---	---	---	----	---	---	---	---	---	---	---	---	---	---	----

i	m	\0	n	t	u	\0	e	d	u	\0	t	w	\0	~	l	c	k	u	n	g	\0
---	---	----	---	---	---	----	---	---	---	----	---	---	----	---	---	---	---	---	---	---	----

i	m	\0	n	t	u	\0	e	d	u	\0	t	w	\0	~	l	c	k	u	n	g	\0
---	---	----	---	---	---	----	---	---	---	----	---	---	----	---	---	---	---	---	---	---	----

String-number conversion

- In `<cstdlib>`, two functions converts a character array into a number:

```
int    atoi(const char* str);  
double atof(const char* str);
```

- For `atoi`, `str` should contain only digits (but the first character can be ‘-’).
- For `atof`, `str` may contain at most one ‘.’.

```
char a[100] = "1234";  
cout << atoi(a) * 2 << "\n";  
char b[100] = "-12.34";  
cout << atof(b) / 2 << "\n";
```