Creative Software Programming

3 – Review of C Pointer and Const, Difference Between C and C++

Yoonsang Lee
Fall 2019

Today's Topics

- C Pointer & Const Review
  - Pointer to Constant & Constant Pointer
  - Two ways of declaring C Strings

- Introduction to C++
- Difference between C and C++
  - Namespace
  - String
  - Input/Output
  - Boolean
  - Function Overloading
  - Brief Intro to Class, Reference, Template, Exception Handling, STL(Standard Template Library)
- Introduction to C++ Standard Versions

C Pointer & Const Review

Constants & Literals

- Expressions with "fixed" values.
  - Cannot modify the value after initialization.

- (Symbolic) Constant
  - A "named" fixed value.

```
const double PI = 3.14;
PI = 1.1; // error!
```

- Literal (constant)
  - A fixed value in source code.
  - Stored in the executable and loaded to memory.

```
int n1 = 3 + 4;
int n2 = 7 + n1;
double n3 = 2.12 + 7.49;
printf("%d %d %f\n", n1, n2, n3);
```

Declaring a Pointer as Const - 1 (Pointer to Constant)

```
int num = 20;
const int* ptr = &num;
```

Cannot change the value of a variable **through the pointer.**

```
*ptr = 30;  // Compile error!
```

However, it does not make the `num` variable itself a constant

```
num = 30;   // Ok
```

Declaring a Pointer as Const - 2 (Constant Pointer)

```
int num1 = 20;
int num2 = 30;
int* const ptr = &num1;
```

Make the pointer **`ptr` a constant.**

→ **Cannot change the value** of `ptr`.

→ **Cannot** change `ptr` to **point to another variable.**

```
ptr = &num2;  // Compile error!
```

However, you can change the value of a variable through the pointer.

```
*ptr = 30;    // Ok
```

Summary

```
const int* ptr = &num;
```

Does **not make the pointer a constant..**

Does **not allow to change the value** of a variable **through the pointer.**

```
int* const ptr = &num;
```

Make the pointer a constant.

Does not allow to change ptr to **point to another variable.**

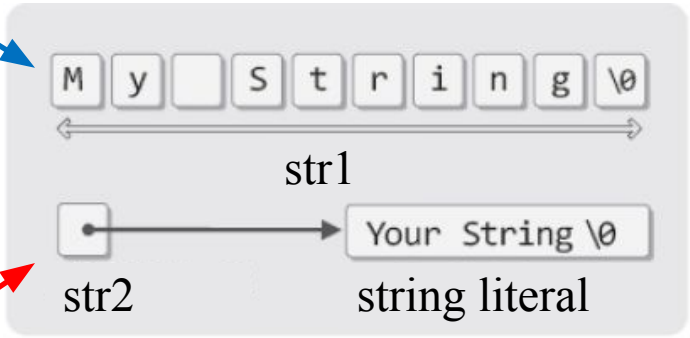
It's easy to get confused, but you have to understand it!

Two ways of declaring C Strings

char str1[] = “My String”;

Declare a string as a char array

str1: An **array** containing the entire string



const char* str2 = “Your String”;

Declare a string as a const char*

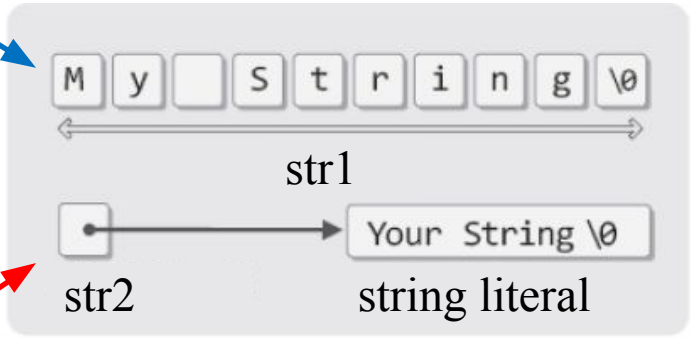
str2: A **pointer** storing the starting position of the string literal (automatically stored somewhere in memory)

Two ways of declaring C Strings

char str1[] = “My String”;

"String in **variable** form"

Can modify the string contents by accessing each element of the array



const char* str2 = “Your String”;

"String in **constant** form"

Cannot modify the string contents as it's just a pointer to a string literal & it's a pointer to constant

Quiz #1

- Write down ALL line numbers where compile errors occur.

```
#include <stdio.h>
int main()
{
    int num1 = 20;
    int num2 = 30;
    const int* ptr1 = &num1;
    int* const ptr2 = &num1;
    const int* const ptr3 = &num1;
    char str1[] = "string1";
    const char* str2 = "string2";

    *ptr1 = 30;  // 1
    num1 = 30;   // 2

    ptr2 = &num2;   // 3
    *ptr2 = 30;     // 4

    ptr3 = &num2;   // 5
    *ptr3 = 40;     // 6

    str1 = "string11"; //7
    str2 = "string22"; //8

    str1[0] = 'X';  // 9
    str2[0] = 'X';  // 10
    return 0;
}
```

String in Constant Form

const char* str2 = “Your String”;

→ The start address of the literal “Your String” is stored in str2.

Ex) If “Your String” is stored at address 10266, it works as follows.

const char* str2 = 10266;

Since str2 is a pointer, you can later change it to the start address of another string.

str2 = “string2”;

This is not possible for str1 in the previous slide.

Passing a String Literal to a Function

printf("Age: %d\n", num);

Similarly, the start address of the literal "Age: %d\n" is passed to the printf() as an argument.

> Ex) If "Age: %d\n" is stored at address 10633, it works as follows.
> printf(10633, num);

That's why the type of the parameter is const char*.

int printf (const char * format, ...);

Meaning of Pointer to Constant as Function Parameter

int printf (**const** char * `format`, ...)

→ Means that printf() **will not change** the contents of the string passed as `format`.

void swap (**int*** p1, **int*** p2)

→ (Implicitly) means that swap() **will change** the value of the variable pointed to by p1 and p2.

Difference between C and C++

Introduction to C++

- Developed by Bjarne Stroustrup at Bell Labs since 1979, as an extension of the C language
- Provides both low-level functionality & efficient abstraction
  - Low-level hardware access, performance & memory efficiency
  - High-level abstraction using object-oriented, generic programming paradigm
- But, high complexity!

C++ Structure of Program

```
// Preprocessor processes #-directives.
#include <iostream>

using namespace std;  /* Use std namespace */

int main() {
  cout << "hello_world\n";  // Print hello_world.
  return 0;
}
```

- Overall structure:
  - Comments.
  - Preprocessor-related parts : #-directives.
  - C/C++ part : statements, declarations or definitions of functions and classes.
- A few notes:
  - A statement ends with a semicolon (;).
  - Blanks (spaces, tabs, newlines) do not affect the meaning, at least in C/C++ parts.

C++ Variables and Data Types

- Fundamental data types
  - Integer : `int` (4), `char` (1), `short` (2), `long` (4), `long long` (8) + `unsigned`,
  - **Boolean : `bool` (1).**
  - Floating point numbers : `float` (4), `double` (8), `long double` (8).
- Variables
  - Variables : specific memory locations
  - Declaration : `int a; double b = 1.0; char c, d = 'a';`
  - Scope : whether the variable is visible (= usable).

```
void MyFunc() {
  int a = 0, b = 1;
  { int a = 2, c = 3;
    cout << "a = " << a << ", b = " << b << ", c = " << c <<
endl;}
  cout << "a = " << a << ", b = " << b << endl;
}
```

C++ Constants

- Integer : `123` (123), `0123` (83), `0x123` (291) / `123u`, `123l`, `123ul`.
- Floating-points : `0.1` (d), `0.1f` (f). / `1e3`, `0.3e-9`.
- Character and string literal : `'c'`, `"a string\n"`.
- **Boolean : `true`, `false`.**

- Defined constants vs. declared constants.
  - Defined constant :`#define MY_NUMBER 1.234`
  - Declared constant : `const double MY_NUMBER = 1.234;`

C++ Operators

- C++ operators
  - Increment/decrement : `++a, a++, --a, a--`.
  - Arithmetic : `a + b, a - b, a * b, a / b, a % b, +a, -a`.
  - Relational : `a == b, a != b, a < b, a <= b, a > b, a >= b`.
  - Bitwise : `a & b, a | b, a ^ b, ~a, a >> b, a << b`.
  - Logical : `a && b, a || b, !a`.
  - Conditional : `a ? b : c`
  - (Compound) assignment : `a = b, a += b, a &= b`, …
  - Comma : `a, b` (e.g. `a = (b = 3, b + 2);`)
  - Other : type casting, `sizeof()`, …
- Operator precedence.
  - Enclose with () when not sure.

Namespace

lib1.h

```
void func() {

}
```

lib2.h

```
void func() {

}
```

```
#include <lib1.h>
#include <lib2.h>
int main(void){
    func();
    return 0;
}
```

???

Namespace

- A method for preventing name conflicts (of variables, functions, ...) in large projects

- All identifiers (variable names, function names, ...) declared in *code* belong to namespace *ns*

```
namespace ns {
    code
}
```

```
#include <iostream>

// first name space
namespace first_space {
   void func() {
      std::cout << "Inside first_space" <<
std::endl;
   }
}

// second name space
namespace second_space {
   void func() {
      std::cout << "Inside second_space" <<
std::endl;
   }
}

int main () {
   // Calls function from first name space.
   first_space::func();

   // Calls function from second name space.
   second_space::func();
   return 0;
}
```

Namespace std

- All the classes, objects, and functions of the *C++ standard library* are defined within "standard" namespace named **std**

- For example, std::cout, std::cin, std::endl for input/output

Namespace

- **using namespace *ns*;**
  - indicates that the subsequent code will use identifiers in the namespace *ns* **as if they were in current namespace**

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

// first name space
namespace first space {
   void func() {
      cout << "Inside first_space" << endl;
   }
}
// second name space
namespace second_space {
   void func() {
      cout << "Inside second_space" << endl;
   }
}
int main () {
   using namespace first_space;
   // This calls function from first name space.
   func();
   return 0;
}
```

Input / Output

- C: printf(), scanf()
  - `#include <stdio.h>`
  - `scanf("%d", &num);`
  - `printf("hello %d\n", num);`

- C++: std::cin, std::cout, stream operators (>>, <<)
  - `#include <iostream>`
  - `std::cin >> num;`
  - `std::cout << "hello " << num << std::endl;`
  - This is the C++ way of input / output, but you can still use C-style input / output in your C++ code.

C++ Stream IO

- Stream: sequence of bytes flowing in and out of the programs
- >> - stream extraction operator. [stream] >> [variable]
- << - stream insertion operator. [stream] << [variable or value]
- std::cout - standard output stream, normally the screen
- std::cin - standard input stream, normally the keyboard
- std::endl - inserts a newline character (‘\n’)

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

int main()
{
    string s2; int i; double d;
    cin >> s2 >> i >> d; // s2, i, d are separated by a space, tab,
enter.
                         // blocked until all variables values are
finally
                         // entered by pressing Enter.
    cout << "s2:" << s2 << ", i:" << i << ", d:" << d << endl;
    return 0;
}
```

String

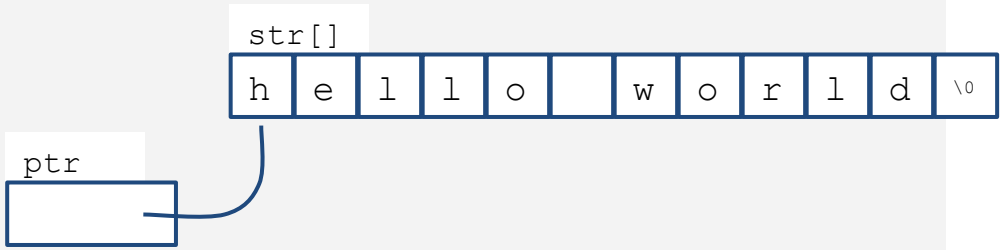
- C: C-style null-terminated string (using C-style array)
  - `char str1[] = "My String";`
  - `const char* str2 = "Your String";`
  - Just an array of characters terminated with a null character ('\0')

- C++: std::string
  - `#include <string>`
  - `std::string str1 = "abc";`
  - `std::string str2("def");`
  - Many convenient operations are available such as:
    `str1 += "123" + str2.substr(0, 2);`
  - Much more powerful and convenient.
  - Use this way in C++. But you still need to understand C-style string because of the legacy code.

C-Style String

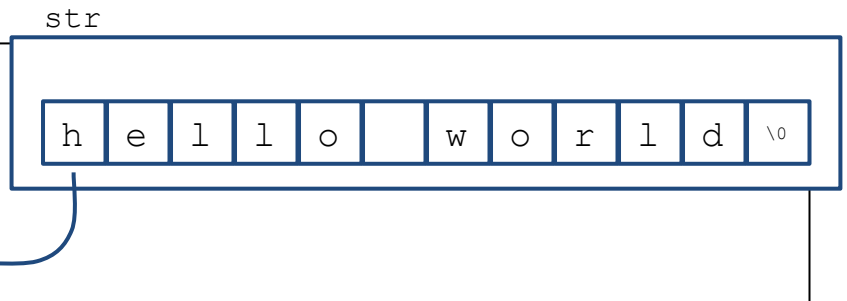
- A string is basically an array of characters (char []).
- C standard requires a string must be terminated with 0 ('\0').

```
#include <stdio.h>

int main() {
  char str[] = "hello world";

  char* ptr = str;
  while (*ptr != '\0') {
    printf("%c", *ptr++);
  }
  return 0;
}
```



C++ std::string

- C++ provides a powerful string class.




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

int main()
{
    string str = "hello world";
    cout << str << endl;    // C++ way of printing to stdout

    string str1 = str + " - bye world";
    cout << str1 << endl;           // hello world - bye world
    cout << str1.length() << endl;  // 23
    cout << str1[0] << endl;        // h

    str[0] = 'j';
    str.resize(5);
    cout << str << endl;  // jello

    const char* ptr = str.c_str();
    printf("%s\n", ptr); // use c_str() for printf(), c++ string -> const char*

    return 0;
}
// check out http://www.cplusplus.com/reference/string/string/
// resize(), substr(), find(), etc.
```

More C++ Input Functions

```
std::string str;
std::cin >> str; // read a word (separated by a space, tab, enter)
```

```
#include <iostream>

using namespace std;

int main(){
  string line;
  cout <<"write a line " << endl;
  while (cin >> line && line != "q")
      cout << line << "---" << endl;
  return 0;
}
```

```
write a line
I like HY⏎
I---
like---
HY---
I love my son⏎
I---
love---
my---
son---
q⏎
```

More C++ Input Functions

```
std::string str;
std::cin >> str; // read a word (separated by a space, tab, enter)

std::getline(cin, str); // read characters until the default
                        // delimiter '\n' is found
```

```
#include <iostream>

using namespace std;

int main(){
  string line;
  cout <<"write a line " << endl;
  while (getline(cin, line)){
    cout << line << "---" << endl;
  }
  return 0;
}
```

```
write a line
I like HY ⏎
I like HY---
I love my son ⏎
I love my son---
```

More C++ Input Functions

```
std::string str;
std::cin >> str; // read a word (separated by a space, tab, enter)

std::getline(cin, str, ':'); // read characters until the delimiter
                             // ':' is found
```

```
#include <iostream>

using namespace std;

int main(){
  string line;
  cout <<"write a line " << endl;
  while (getline(cin, line, ':')){
    cout << line << "---" << endl;
  }
  return 0;
}
```

```
write a line
I:like:HY ⏎
I---
like---
I:love:my:son ⏎
HY
I---
love---
my---
: ⏎
son
---
```

More C++ Input Functions

- Note that `std::string` automatically resize to the length of target string.

```
char fname[10];
string lname;
cin >> fname;       // could be a problem if input size > 9 characters
cin >> lname;       // can read a very, very long word
cin.getline(fname, 10);  // may truncate input
getline(cin, lname);     // no truncation
```

Boolean

- To express Boolean values (true or false),

- C:
  - `int var1 = 1; // true`
  - `int var2 = 0; // false`
  - Non-zero values are regarded as ‘true’
  - (C99 standard support ‘bool’ type with <stdbool.h> header)

- C++:
  - `bool var1 = true;  // true`
  - `bool var2 = false;  // false`
  - More intuitive. Use this way in C++.

Quiz #2

- Write down the expected result of running the following program with the standard input in the right box. (⏎ means pressing Enter key)

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

int main () {
    string str;
    getline(cin, str, ',');
    getline(cin, str, ',');
    getline(cin, str, ',');
    cout << str << endl;

    return 0;
}
```

```
hello:world,good,bye,⏎
```

Function Overloading

- Use multiple functions sharing the same name
  - A family of functions that do the same thing but using different argument lists

```
void print(const char * str, int width);  // #1
void print(double d, int width);          // #2
void print(long l, int width);            // #3
void print(int i, int width);             // #4
void print(const char *str);              // #5


print("Pancakes", 15);          // use #1
print("Syrup");                 // use #5
print(1999.0, 10);              // use #2
print(1999, 12);                // use #4
print(1999L, 15);               // use #3
```

Function Overloading

- The function signature, not the function type, enables function overloading
  - A function signature consists of function name, parameter order & types
  - Function signatures **do not include return type**

```
int test1(int n, float m);         // different signatures,
double test1(float n, float m);    // hence allowed

int test2(int n, float m);       // the same signatures
double test2(int n, float m);    // compile error
```

Function Overloading

```
void dribble(char * bits);          // overloaded
void dribble (const char *cbits);   // overloaded
void dabble(char * bits);           // not overloaded
void drivel(const char * bits);     // not overloaded

const char p1[20] = "How's the weather?";
char p2[20] = "How's business?";
dribble(p1);      // dribble(const char *);
dribble(p2);      // dribble(char *);
dabble(p1);       // no match
dabble(p2);       // dabble(char *);
drivel(p1);       // drivel(const char *);
drivel(p2);       // drivel(const char *);
```

Quiz #3

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

double square(double a)
{
    cout << "(double version)";
    return a*a;
}

double square(float a)
{
    cout << "(float version)";
    return a*a;
}

int main()
{
    cout << square(5.0f) << endl;

    return 0;
}
```

- What is the expected result of running the following program ?
  - 1) (double version)25
  - 2) (float version)25
  - 3) A compile error occurs

References

- References can be used similar to pointers (Think of it as a "referenced pointer")
  - Less powerful but safer than the pointer type.

```
int b = 10;
int& rb = b; // rb can be regarded as an "alias" of b
rb = 20;
cout << b << " " << rb << endl;    // 20 20
```

- **Will be covered in the next class**

STL (Standard Template Library)

- Powerful, template-based, reusable components
- STL extensively uses templates

- Divided into three components:
  - Containers: data structures that store objects of any type
  - Iterators: used to manipulate container elements
  - Algorithms: searching, sorting and many others

- **Will be covered in later classes**

Template

- Generalizes function or class by delaying type specification until compile-time.

```
// We also want to sort a double array.
void SelectionSort(double* array, int size) {
  for (int i = 0; i < size; ++i) {
    int min_idx = i;
    for (int j = i + 1; j < size; ++j) {
      if (array[min_idx] > array[j])
        min_idx = j;
    }
    double tmp = array[i];
    array[i] = array[min_idx];
    array[min_idx] = tmp;
  }
}
// And also a string array.
void SelectionSort(string* array, int
size) {
  for (int i = 0; i < size; ++i) {
    int min_idx = i;
    for (int j = i + 1; j < size; ++j) {
      if (array[min_idx] > array[j])
        min_idx = j;
    }
    string tmp = array[i];
    array[i] = array[min_idx];
    array[min_idx] = tmp;
  }
}
```

```
// Suppose we want to sort an array of type T.
template <typename T>
void SelectionSort(T* array, int size) {
  for (int i = 0; i < size; ++i) {
    int min_idx = i;
    for (int j = i + 1; j < size; ++j) {
      if (array[min_idx] > array[j])
        min_idx = j;
    }
    // Swap array[i] and array[min_idx].
    T tmp = array[i];
    array[i] = array[min_idx];
    array[min_idx] = tmp;
  }
}
```

- **Will be covered in later classes**

Exception Handling

- Examples of exceptions:
  - Memory allocation error - out of memory space.
  - Divide by zero.
  - File IO error.
  - …

- C++ provides a systematic way of handling exceptions

- **Will be covered in later classes**

```
try {
   // protected code
} catch( ExceptionName e1 ) {
   // catch block
} catch( ExceptionName e2 ) {
   // catch block
} catch( ExceptionName eN ) {
   // catch block
}
```

Introduction to C++ Standard Versions

- C++98 (the first standard) / C++03 (its minor revision)
  - Called “traditional C++”
- C++11 / C++14 / C++17
  - Many cool & useful features such as smart pointer, auto keyword, lambda function, etc
  - Called “modern C++”
- This class is based on C++98 / C++03
  - The large majority of C++ is still same to C++98 / C++03
  - A large number of codebases are written in C++98 / C++03
- References to modern C++:
  - https://github.com/AnthonyCalandra/modern-cpp-features
  - https://en.cppreference.com/w/cpp/compiler_support

Next Lecture

- Next lecture:
  - 4 - Dynamic Memory Allocation, References