Lecture #6 | C/C++: array & pointer (cont.)

SE271 Object-oriented Programming (2017)

Prof. Min-gyu Cho

Today's topic

- More on arrays and pointers
  - Multidimensional array
  - Call by value v.s. by reference
  - Function pointers
- Other topics
  - `const, constexpr,` macro
  - Null pointer
  - Simple code with array, pointer, C string manipulation

Multidimensional array

- Represented as arrays of arrays; it is linearly allocated in memory

```
int main(void) {
    int ma[3][5]; // takes 15 int's in memory; 3 arrays of 5 ints
    for (int i = 0; i < 3; i++)
        for (int j = 0; j < 5; j++)
            ma[i][j] = 10 * i + j;
}
```

Call (pass) by value v.s. call by reference

- Call by value: pass the values as function arguments
  - Values of arguments are copied; can be slow when passing complex types
  - Changes to the variable within the callee (i.e., invoked function) do not occur outside of function (caller)
- Call by reference: pass the addresses as function arguments
  - Pass (i.e., make copy of) the address of a variable
  - Changes to the variable within the callee occur outside of function

Example: call by value v.s. call by reference

```
void call_by_value(int x) {
    x++;
    cout << "call_by_value: x=" << x << endl;
}
void call_by_reference(int* ptr) {
    (*ptr)++;
    cout << "call_by_reference: *ptr=" << *ptr << endl;
}
int main(void) {
    int x = 23;
    call_by_value(x);       cout << "x=" << x << endl;
    call_by_reference(&x); cout << "x=" << x << endl;
}
```

Example: call by reference (C v.s. C++)

```
void call_by_reference(int* ptr) {
    (*ptr)++;
    cout << "call_by_reference: *ptr=" << *ptr << endl;
}
void call_by_reference_cpp(int& x) { // C++ only, will be discussed later
    x++;
    cout << "call_by_reference_cpp: x=" << x << endl;
}
int main(void) {
    int x = 23;
    call_by_reference(&x);    cout << "x=" << x << endl;
    call_by_reference_cpp(x); cout << "x=" << x << endl;
}
```

(optional) Function pointer

- Function pointer: a pointer that can designate a function
- Declaration: to declare a variable with a function pointer type, change the name of declaration change the function name in the declaration to (*
- Example:

```
// function declaration
return_type func(parameters) {}
// function pointer declaration
return_type (*func_ptr)(parameters);
// function pointer assignment
func_ptr = func;
// c.f., function returning pointer to return_type
return_type *func_ptr(parameters);
```

(optional) Example: function pointer

```
void func1(void) {
    cout << "func1 is invoked." << endl;
}
void func2(void) {
    cout << "func2 is invoked." << endl;
}
int main(void) {
    void (*func_ptr)(void);
    func_ptr = func1;
    func_ptr(); // invoke func1()
    func_ptr = func2;
    func_ptr(); // invoke func2()
}
```

How to interpret complex pointer declaration

- Read right to left, and () first

```
char **argv                 argv: pointer to pointer to char
int (*daytab)[13]           daytab: pointer to array[13] of int
int *daytab[13]             daytab: array[13] of pointer to int
void *comp()                comp: function returning pointer to void
void (*comp)()              comp: pointer to function returning void
char (*(*x())[])()          x: function returning pointer to array[] of
                            pointer to function returning char
char (*(*x[3])())[5]        x: array[3] of pointer to function returning
                            pointer to array[5] of char
```

Source: “The C Programming Languages”

How to define constant variable in C++: const

- Variable that cannot be modified in this scope, and should be initialized
- Example: `const double pi = 3.1415926535;`
- Example with pointers

```
void foo(char* p) {
    char s[] = "DGIST";
    const char* pc = s;          // pointer to constant char
    pc[1] = 'g';                 // error: pc points to constant
    pc = p;                      // OK
    char* const cp = s;          // constant pointer
    cp[1] = 'g';                 // OK
    cp = p;                      // error: cp is constant
    const char *const cpc = s;   // const pointer to const
}
```

Casting of const may result in obscure codes

```
int main()
{
    const char s[] = "DGIST";
    char* pc = (char*)s;
    pc[1] = 'g'; // no error at compile time

    cout << s << endl;
}
```

How to define constant values in C++: `constexpr`

- Values should be able to be evaluated at compile-time (C++11)
- `Example: constexpr double pi = 3.1415926535;`
- Pros*

1. Named constants makes the code easier to understand and maintain
2. A variable might be changed(so we have to be more careful in our reasoning that for a constant)
3. The language requires constant expression for array sizes, case labels, and template value arguments
4. Embedded systems programmers like to put immutable data into read-only memory because read-only memory is cheaper than dynamic memory (in terms of cost and energy consumption), and often more plentiful. Also, data in read-only memory is immune to most system crashes
5. If initialization is done at compile time, there can be no data races on that object in a multi-threaded system
6. Sometimes, evaluating something once (at compile time) gives significantly better performance than doing so a million times at run time

* Adopted from “The C++ Programming Language”

Macro

- Macro: frequently used in C to define constants or function-like expression
  - Part of C pre-processor (starts with #)
- Examples

```
#define BUFFER_SIZE 1024
#define KILO 1000
#define MIN(x, y) ((x) > (y) ? (x) : (y))

int v1 = MIN(BUFFER_SIZE, KILO);
int v2 = MIN(3 + 4, 2 * 3) + MIN(2 * 3, 3 + 4);
```

Null pointer

- A pointer that does not point to any object
- Examples

```
void printFirstValue(int* ptr) {
    if (ptr) cout << ptr[0] << endl;
}
int main(void) {
    int arr[] = {1, 2, 3};
    int* p = nullptr;
    printFirstValue(p);
    cout << "After assignment\n";
    p = arr;
    printFirstValue(p);
}
```

Null pointer in C

- The following values are typically used as a null pointer in C, which lacks proper type checking
  - `0`
  - `0L`
  - `(void*)0`
- Note: `void*` is used to represent a pointer to an address which is NOT associated with specific data type
  - In C, `void*` type can be assigned to any type of pointers
  - In C++, type conversion (with casting) is required to assign an address with `void*` type to a pointer

Obscure variable definitions

- Declaring multiple names

```
int* p, y;          // int *x; int y;
int x, *q;          // int x; int *q;
int v[10], *pv; // int v[10]; int* pv;
```

- Rule of thumb: DO NOT declare multiple names in one line

Reading list

- Learn C++
  - C-string: Ch. 6.8b
  - Call by value v.s. reference: Ch. 7.1-4
  - const, constexpr: Chapter 2.9
- Why should casting be avoided?
  - https://stackoverflow.com/questions/4167304/why-should-casting-be-avoided

ANY QUESTIONS?