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

Revisit: array, pointer and loop

```
int main()
{
    int a[] = {2, 3, 5, 7};
    int n = sizeof(a) / sizeof(a[0]);
    int* ptr = a;

    for (int i = 0; i < n; i++)
        cout << a[i] << endl;

    for (int i = 0; i < n; i++)
        cout << *(ptr + i) << endl;

    int* end = a + n;
    for (ptr = a; ptr != end; ptr++)
        cout << *ptr << endl;
}
```

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 problematic when passing large objects
  - Changes to the variable within the callee (i.e., invoked function) do not occur outside of function (caller)
  - Easier to understand, and less error prone in most cases
- 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
  - More efficient when passing large objects

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”

One more thing: pointer to pointer

```
int var = 42;
int* ptr = &var;
int** pptr = &ptr;
cout << "  var:  " <<   var  << endl;
cout << " &var:  " <<  &var  << endl;
cout << " *ptr:  " <<  *ptr  << endl;
cout << "  ptr:  " <<   ptr  << endl;
cout << " &ptr:  " <<  &ptr  << endl;
cout << "**pptr: " << **pptr << endl;
cout << " *pptr: " <<  *pptr << endl;
cout << "  pptr: " <<   pptr << endl;
cout << " &pptr: " <<  &pptr << endl;
```

```
 var:  42
 &var:  0x7fff59e9c7dc
 *ptr:  42
  ptr:  0x7fff59e9c7dc
 &ptr:  0x7fff59e9c7d0
**pptr: 42
 *pptr: 0x7fff59e9c7dc
  pptr: 0x7fff59e9c7d0
 &pptr: 0x7fff59e9c7c8
```

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 *p; 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?