# C++ Basics and Elementary Memory Management

임종우 (Jongwoo Lim)

# 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 (l-value vs. r-value)
  - 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.

# C++ String, Basic Input/Output

- C++ strings
  - `#include <string>`
  - `std::string empty_str, my_str = "abc", str("def");`
  - Many operations are possible including
    `my_str += "123" + str.substr(0, 2);`

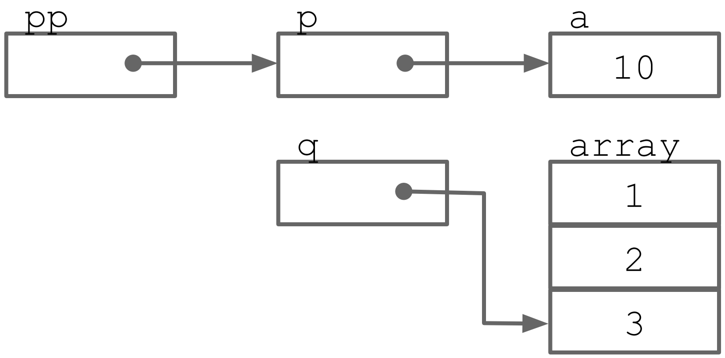
- C++ iostream
  - `#include <iostream>`
  - `std::cin`, operator >>.
  - `std::cout`, `std::cerr`, operator <<.

# Pointer

- Pointer : a variable that contains the address of a memory block.
  - Point to a variable, array, struct (class) or function.

```
int a = 10;
int* p = &a;
cout << "*p = " << *p << endl;      // Outputs 10.
*p = 20;
cout << "a = " << a << endl;  // Outputs 20.

int array[3] = { 1, 2, 3 };
p = array;
int* q = &array[2];
int** pp = &p;
cout << "**pp = " << **pp << endl; // Outputs 1.
pp = &q;
cout << "**pp = " << **pp << endl; // Outputs 3
```



# C malloc / free

- Allocate and deallocate memory block.
  - Example: C arrays are with fixed sizes.
  - How can we use variable size array?

```
void TestFunction(int n) {
  int fixed_size_array[20];
  int variable_size_array[n];  // Compile error.

  for (int i = 0; i < n; ++i) {
    cout << fixed_size_array[i] << ", "  // SEGFAULT if n > 20.
         << variable_size_array[i];
  }
}
```

# C malloc / free

- Allocate and deallocate memory block.
  - ○ Example: C arrays are with fixed sizes.
  - ○ Use `malloc/free` to manage memory allocation.

```
#include <stdlib.h>

void TestFunction(int n) {
  int* variable_size_array = (int*) malloc(sizeof(int) * n);
  for (int i = 0; i < n; ++i) {
    cout << variable_size_array[i] << endl;
  }
  free(variable_size_array);
}
```

  - ○ `malloc(n)` : allocates n bytes of memory block and return the pointer to the block.
  - ○ `free(ptr)` : deallocates the allocated memory block.

# C malloc / free

- What happens if allocated blocks are not freed?
- Memory leak : an allocated but unused memory is not returned to OS.
    - Usually happens when the pointer to it gets lost.

```
#include <stdlib.h>

void TestFunction(int n) {
  double* another_array = (double*) malloc(sizeof(double) * n);

  for (int i = 0; i < n; ++i) {
    int* variable_size_array = (int*) malloc(sizeof(int) * n);
    cin >> another_array[i]
        >> variable_size_array[i];
    // free(variable_size_array);
  }
  another_array = (double*) malloc(sizeof(double) * n);
  free(another_array);
}
```

# C++ new / delete

- C++ has `new` and `delete` operators built-in.
  - `new` : creates an instance of the class(type).
  - `delete` : destructs an instance created by `new`.
  - `new []` : creates an array of instances of the class.
  - `delete[]` : destructs an object array created by `new[]`.

| | One instance | Array |
|---|---|---|
| Allocate | `new` | `new []` |
| Deallocate | `delete` | `delete[]` |

# C++ new / delete

- C- and C++-version of the previous example.

```
#include <stdlib.h>

void TestFunction(int n) {
  int* int_instance = (int*) malloc(sizeof(int));
  int* variable_size_array = (int*) malloc(sizeof(int) * n);

  *int_instance = 10;
  for (int i = 0; i < n; ++i) cin >> variable_size_array[i];

  free(int_instance);
  free(variable_size_array);
}
```

```
void TestFunction(int n) {
  int* int_instance = new int;
  int* variable_size_array = new int[n];

  *int_instance = 10;
  for (int i = 0; i < n; ++i) cin >> variable_size_array[i];

  delete int_instance;
  delete[] variable_size_array;
}
```