Lecture #7 | A few more C/C++ syntax

SE271 Object-oriented Programming (2017)

Prof. Min-gyu Cho

Today's topic

- Constant values
  - `const`
  - `constexpr`
  - MACROs
- Standard input/output
  - C++-style standard input/output: `cin, cout`
  - C string and C-style standard input/output

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

```c
#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);
```

Standard input and output with C++

- Most operating systems, and thus, most programming languages provide methods to access standard (console) input (typically keyboard) and output (typically monitor with text output)
- C++ also provides ways to access standard input/outputs
  - Standard output: `std::cout`
  - Standard input: `std::cin`
  - Standard error: `std::cerr`

Example: standard input/output in C++

```cpp
int i, j;
double d;
char buffer[1024];
cin >> i >> j >> d; // items separated by space
cout << "i = " << i << endl;
cout << "j = " << j << endl;
cout << "d = " << d << endl;
// sting input is also separated by space
cin >> buffer;
cout << "buffer = " << buffer << endl;
// handle input with spaces using getline()
cin.getline(buffer, sizeof(buffer));
cout << "buffer = " << buffer << endl;
```

```
$ ./stdio
42 23 3.14
i = 42
j = 23
d = 3.14
Hello, world!
buffer = Hello,
buffer = world!
$ ./stdio
42 34 3.14
i = 42
j = 34
d = 3.14
Hi!
buffer = Hi!
Hello, world!
buffer = Hello, world!
```

Input/output in C (also available in C++)

- Generate formatted output to standard output, file or string

```
int printf(const char* format, ...);
int fprintf(std::FILE* stream, const char* format, ...);
int sprintf(char* buffer, const char* format, ...);
int snprintf(char* buffer, std::size_t buf_size, const char* format, ...);
```

- Read data from standard input, file or string for given data types

```
int scanf(const char* format, ...);
int fscanf(std::FILE* stream, const char* format, ...);
int sscanf(const char* buffer, const char* format, ...);
```

- Reference
  - http://en.cppreference.com/w/cpp/io/c/fprintf
  - http://en.cppreference.com/w/cpp/io/c/fscanf

Example: standard input/output in C

```c
#include <stdio.h>
// #include <cstdio> // C++ in std namespace
int main()
{
    int i = 42;
    double d = 1.618;
    double d2 = 1.0;
    printf("Using printf(): %d %f %f %g\n",
           i, d, d2, d2);
    printf("%10d %5.3f %5.3g\n", i, d, d);
    scanf("%d", &i);
    scanf("%lf", &d);
    printf("Scanned data: %d %g\n", i, d);
}
```

```
$ ./stdio
Using printf(): 42 1.618000 1.000000 1
        42 1.618  1.62
3
3
Scanned data: 3 3
$ ./stdio
Using printf(): 42 1.618000 1.000000 1
        42 1.618  1.62
3.1
Scanned data: 3 0.1
```

Escape sequence

Name	Symbol	Meaning	Name	Symbol	Meaning
Alert	`\a`	Makes an alert, such as a beep	**Single quote**	**`\'`**	**Prints a single quote**
Backspace	`\b`	Moves the cursor back one space	**Double quote**	**`\"`**	**Prints a double quote**
Formfeed	`\f`	Moves the cursor to next logical page	**Backslash**	**`\\`**	**Prints a backslash**
Newline	**`\n`**	**Moves cursor to next line**	Question mark	`\?`	Prints a question mark
Carriage return	`\r`	Moves cursor to beginning of line	Octal number	`\(number)`	Translates into char represented by octal
Horizontal tab	**`\t`**	**Prints a horizontal tab**	Hex number	`\x(number)`	Translates into char represented by hex number
Vertical tab	`\v`	Prints a vertical tab			

Structures (`struct`)

- C++: the same with class, but default access is public
- C: no member function, only (data) elements
- Example:

```
struct vector {
    int n;
    int *elem;
};
int main() {
    struct vector v; // C
    //vector v; // C++
    v.n = 10;
    printf("%d\n", v.n);
}
```

C-style dynamic memory management

- Header file
  - C: `<stdlib.h>`
  - C++: `<cstdlib>`
- Allocation

```
void* malloc(size_t size);
void* calloc(size_t num, size_t size);
```

- Deallocation

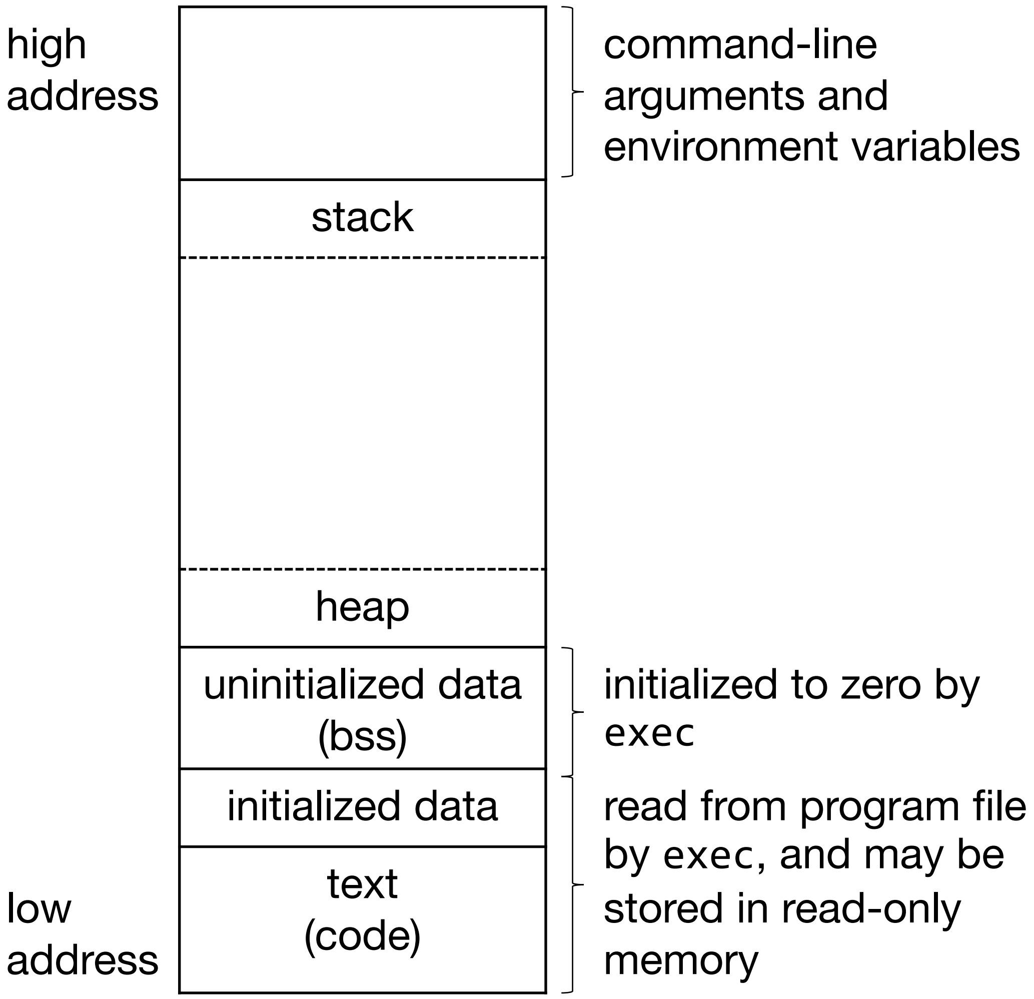
```
void free(void* ptr);
```

- Caution
  - You have to allocate memory before reading/writing
  - You have to deallocated allocated memory when you do not need it any more
  - You should NOT use any memory after deallocation

Example: structure and memory management

```c
#include <stdio.h>
#include <stdlib.h>
struct vector {
    int n;
    int *elem;
};
int main() {
    struct vector v; // C
    //vector v; // C++
    struct vector *ptr;
    v.n = 10;
    //v.elem = (int *)malloc(10 * sizeof(int)); or
    v.elem = (int *)calloc(10, sizeof(int));
    v.elem[0] = 42;
    ptr = &v;
    printf("%d\n", ptr->elem[0]);
    free(v.elem);
}
```

Reference: memory layout of C/C++ programs*


high address
stack
heap
uninitialized data (bss)
initialized data
text (code)
low address
command-line arguments and environment variables
initialized to zero by exec
read from program file by exec, and may be stored in read-only memory

- Stack area contains the program stack (a LIFO structure) and stores automatic (a.k.a., local) variables
- Heap area can be allocated to the program as requested, but memory can be leaked w/o proper memory management
- Uninitialized data area typically stores uninitialized global variables
- Initialized data area stores initialized global variables and literals
- Text area stores program codes

* May have variations by different system

Aliasing of types

- `typedef` (C/C++): put a new type name in the place of variable name in a declaration statement of the given type
- `using` (C++ only): put a new type name, following by = and old type
- Example

```
typedef long int lint;
using Lint = long int;
lint variable_name;
Lint another_variable_name;
```

Reading list

- Reference on C-style dynamic memory management
  - http://en.cppreference.com/w/c/memory/malloc
  - http://en.cppreference.com/w/c/memory/calloc
  - http://en.cppreference.com/w/c/memory/free

ANY QUESTIONS?