Lecture #9 | Class (cont.)

SE271 Object-oriented Programming (2017)

Prof. Min-gyu Cho

We have covered so fat…

- What it is class
- How to declare or define class
  - Declaration/definition
  - Member function definition within class declaration
- Constructors
- Destructors
- Access control

Today’s topic

- Dynamic memory allocation
  - `new`
  - `delete`

Dynamic memory allocation

- You can allocate & deallocate memory as needed with dynamic memory allocation
- Why do you need dynamic memory allocation?
  - When the maximum amount of memory to use is NOT known at compile time, e.g.,

```
int variable_length_array(int n) {
    int array[n]; // not allowed
    ...
}
```

  - When you need to allocate a very large object
  - You need to allocate memory that can span beyond the scope of a function or a code block

Example: variable beyond the scope of a function

- Example

```
int* square(int n) {
    int squared_value = n * n;
    return &squared_value;
}
void test() {
    int *value = square(10);
    cout << "10^2=" << *value << endl;
}
```

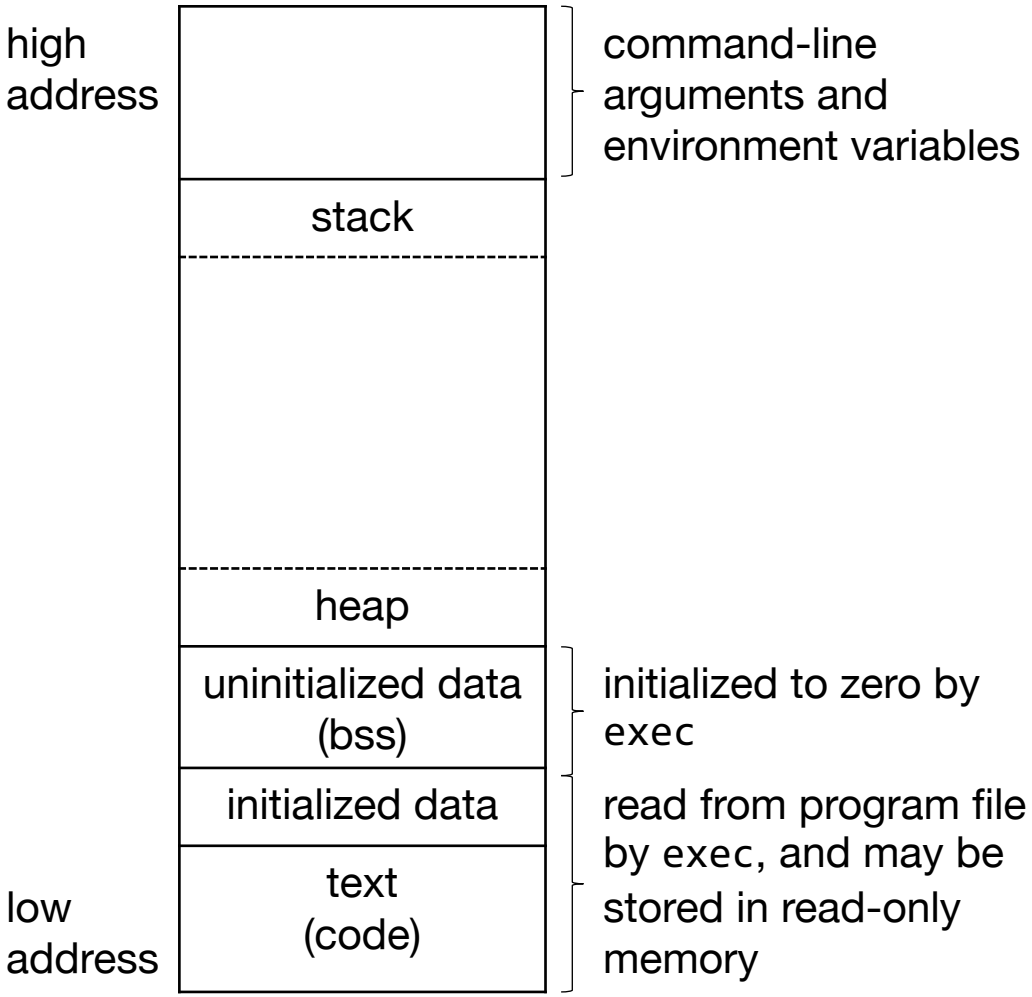
- Warning message at compile time

```
dynmem.cpp:23:13: warning: address of stack memory associated with local
variable 'squared_value' returned [-Wreturn-stack-address]
    return &squared_value;
```

Dynamic Memory allocation & deallocation*

- new operator: allocate memory and return a pointer to the allocated memory
  - A pointer value is the address of the first byte of the memory
  - A pointer to an object of a specified type
  - A pointer does NOT know how many elements it points to
  - Invokes a constructor for user-defined type (a.k.a., class)
- `delete` operator: free (i.e., deallocate) memory
  - `delete` frees the memory for an individual object allocated by `new`
  - `delete[]` frees the memory for an array of objects allocated by `new`
  - Invokes a destructor for user-defined type
- To avoid *memory leaks*, you have to *free* memory after you *get* one
- Do not delete twice, or use memory after deallocation

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



- 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

Example

```
int main() {
    int n = 10;
    int* p1 = new int;     // allocate 1 int
    int* p2 = new int[4]; // allocate 4 ints (an array of 4 ints)
    int* p3 = new int[n]; // allocation n ints
    ...
```

Example (cont.)

```
    // with initializations
    int* p4 = new int {42}; // allocate 1 int, initialized with 42
    int* p5 = new int[4] {2, 3, 5, 7}; // allocate 4 ints
    int* v = new Vehicle("Kia") ; // allocate memory for Vehicle type
    ...

    // release memory
    delete[] p5;
    delete p4;
    delete[] p3;
    delete[] p2;
    delete p1;
}
```

Example: fix of the previous example

```
int* square2(int n) {
    int* ptr_squared = new int {n * n};
    return ptr_squared;
}
void test2() {
    // where to put 'delete value’?
    int *value;
    for (int i = 0; i < 10; i++){
        value = square2(i);
        cout << "i^2=" << *value << endl;
    }
}
```

Now let's fix `IntList` class

```
// intlist.h
class IntList
{
private:
    int n;
    int* elem[max_list];
public:
    IntList(int n_ = 0);
    IntList(int n_, int* a);
    ~IntList();
    ...
};
```

```
// intlist_dynamic.cpp
IntList::IntList(int n_)
{
    n = n_;
    elem = new int[n];
    for (int i = 0; i < n; i++)
        elem[i] = 0;
}
IntList::IntList(int n_, int* a)
{
    elem = new int[n];
    for (n = 0; n < n_; n++)
        elem[n] = a[n];
}
IntList::~IntList()
{
    delete[] elem;
}
```

Reading list

- Learn C++
  - Memory management: Ch. 6.9-9a

ANY QUESTIONS?