Runtime memory model

Prof. Y C Cheng
2009/10/7

國立台北科技大學 資訊工程系
National Taipei University of Technology
Dept of Comp Science and Information Engineering
Background

• Your program *shares* the main memory of your computer with other programs, e.g., Word, Firefox, and so on.
• The OS manages the memory (enabling the sharing).
• The OS gives your program a small chunk of (a few pages of) memory.
• To your program, the memory looks like…
C/C++ Runtime memory model

Data area
- Where literals (e.g., 3, ‘a’, “hello”) and global variables are stored

Text area
- Where the executable code of your program is stored

Stack area
- Where the frames of functions are stored LIFO. Inside a frame as: parameters, local variables, etc. defined in the function. Managed by the C/C++ runtime.

Heap area
- Memory area you can freely use using `new`, `malloc`, `calloc` etc. Your program is responsible for managing the memory.
Where to put things?

```c++
void printVector(double v[], int dim)
{
    cout << "(";
    for (int i=0; i< dim-1; ++i)
        cout << v[i] << ",";
    cout << v[dim-1] << ")" << endl;
}

int results[65536];

int main(int argc, char *argv[])
{
    double v1[2] = {1.0, 0.0};
    double v2[2] = {1.0, 1.0};
    double v3[3] = {1,1,0};
    double v4[3] = {0,1,1};

    int * scores = new double [75];
    printVector(v2,2);
    printVector(v3,3);
    printVector(v4,3);
}
```
Looking back: Is our program really correct?

- What if we call printVector(a) many times?
  - Parameter passing of printVector(Vector v) is call-by-value
  - a is copied by the C++ runtime into v
  - But we did not specify how copy is done!
  - So C++ gives you one “copy constructor” automatically, with “shallow copy” semantics
  - Thus creating two pointers (component in a and component in v) pointing to the same area in heap
  - When v goes out of scope at the end of outputVector, memory in heap is deleted by the destructor ~Vector
After Vector a(10) (1/4)
First call to outputVector(a) (2/4)

Output: (0, 0, 0, 0, 0, 0, 0, 0, 0, 0)

- outputVector
- v
  - component
  - dim
- main
  - a
    - component
    - dim
- stack
- heap
  - 0, 0, ..., 0
after return from the first call of `outputVector(a)` (3/4)
second call of outputVector(a) (4/4)

Output: (1,0,0,0,2.122e-314,4.94066e-324,0,9.9271e-315,0,7.84041e+268)
Concluding remarks

• Avoid huge global variables
• Always initialize local variables
• If you borrow memory from heap, you must always return it.
• If you run out of memory, OS kicks out your program (abnormal termination)
  – Declaring huge globals (depleting data area) and locals (depleting stack area)
  – Deep recursive function call that depletes the stack area
  – Keep borrowing and without returning (depleting heap area), a.k.a. memory leak.