Memory Management

CMPT 115/117 lecture slides

Notes written by Mark Eramian, Ian McQuillan, Michael Horsch, Lingling Jin, and Dmytro Dyachuk

Objectives

By the end of this lecture topic, you are expected to be able to

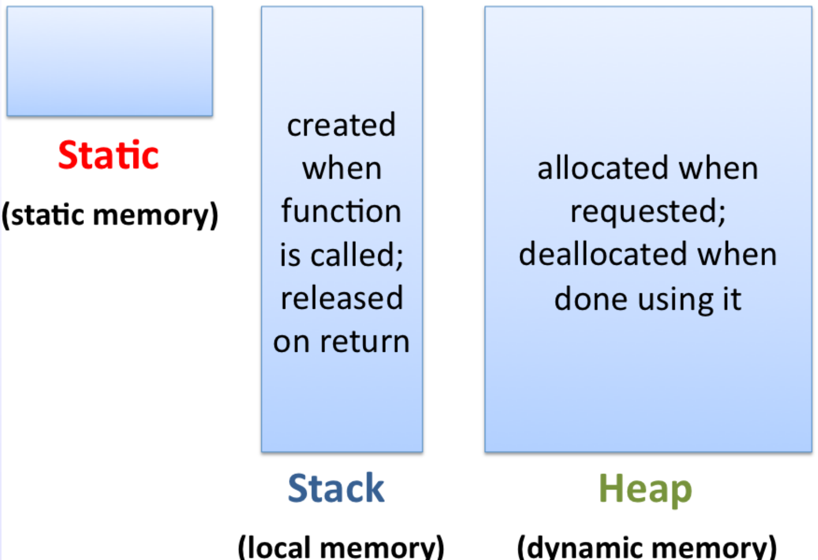
1. describe the three kinds of memory in your own words
2. justify the benefits of using heap based on the scenario discussed in class
3. write pseudocode and C++ code to dynamically allocate memory for different types
4. write pseudocode and C++ code to deallocate memory
5. apply the steps of dynamic memory pattern in programming without mistakes

Part I

Memory: Static, Automatic, and Dynamic

Outline

Three kinds of memory



Dialog 1: Static memory

Context: Program starting

OS: “Here’s the memory the compiler told me the program needed. I’ll collect it when the program terminates.”

Dialog 2: Local memory

Context: Function starting

Runtime system: "Here's the memory the function needs for variables declared in the function. I'll collect it when the function returns."

Dialog 3: Dynamic memory

Context: anywhere, anytime

Program: "Hey, I need memory to store a thing."

Memory manager: "Okay, here's a reference to the memory you asked for. Let me know when you're done with it."

Time passes …

Program: "Hey, I'm done with it."

Memory manager: "Thanks! I'll mark it as reusable in case you ask for more later."

This conversation can occur many times, in any order. That's what makes memory management tricky.

Static Memory stores Global Variables

In this course, we will use global/static memory for constants only.

- Global variables are allocated in *static memory*.
- Global variables are visible throughout the entire program file.
- Static memory is allocated before the program begins running, and which can never be deallocated while the program is running.
- Static memory is convenience at the expense of adaptability, correctness, and reusability.

Warning

Everything we have learned about programming over 50 years software tells us: Do not use global variables, except for constants.

Local memory

- Local memory is all memory declared inside a function.
- When the function is called, memory is reserved for all local variable declarations, including arrays.
- When the function returns, local memory is released.

```
void bar(){
  int myNum = 7;
  cout << "Contents of myNum: " << myNum;

  myNum++;
  doSomething(myNum);
  cout << "Contents of myNum: " << myNum;
}
```

The runtime stack

address	contents	
⋮	⋮	
4683953	7	myNum
4683954		myNum
4683955		myNum
4683956		myNum
⋮	⋮	

Local (automatic) memory

```
void foo(){
  int myNum;
  int *myNumPtr;

  myNum = 7;
  myNumPtr = &myNum;

  (*myNumPtr)++;
  cout << "Contents of myNum: " << myNum;

  doSomething(myNumPtr);
  cout << "Contents of myNum: " << myNum;
}
```

The runtime stack

address	contents	
⋮	⋮	
4683953	7	myNum
4683954		myNum
4683955		myNum
4683956		myNum
4683957	4683953	myNumPtr
4683958		myNumPtr
4683959		myNumPtr
4683960		myNumPtr

- Above, space is allocated for `myNum` and `myNumPtr`.
- When the function returns, these variables are released.
- Because this is done automatically, local memory is also called "automatic."

Local (automatic) memory

```
void foo(){
  int myNum;
  int *myNumPtr;

  myNum = 7;
  myNumPtr = &myNum;

  (*myNumPtr)++;
  cout << "Contents of myNum: " << myNum;

  doSomething(myNumPtr);
  cout << "Contents of myNum: " << myNum;
}
```

The runtime stack

address	contents	
⋮	⋮	
4683953	7	myNum
4683954		myNum
4683955		myNum
4683956		myNum
4683957	4683953	myNumPtr
4683958		myNumPtr
4683959		myNumPtr
4683960		myNumPtr

- Local memory is allocated in a region called "the stack".
- The stack builds up with each function call, and is reduced with each return.
- So: `myNum` remains "alive" throughout the call to `doSomething()`, but is released at the end of `foo()`

Local (automatic) memory

```
void foo(){
   int myNum;
   int *myNumPtr;

   myNum = 7;
   myNumPtr = &myNum;

   (*myNumPtr)++;
   cout << "Contents of myNum: " << myNum;

   doSomething(myNumPtr);
   cout << "Contents of myNum: " << myNum;
}
```

The runtime stack

address	contents	
⋮	⋮	
4683953	7	myNum
4683954		myNum
4683955		myNum
4683956		myNum
4683957	4683953	myNumPtr
4683958		myNumPtr
4683959		myNumPtr
4683960		myNumPtr

- Note: local variables declared in the function `doSomething` are allocated and deallocated while `foo()` waits.
- It would be a serious error for `doSomething` to return a reference to one of its local variables!

Dynamic memory

- Static and local memory cannot solve all problems, because:
    - Local memory is released when the function returns, limiting its uses.
    - Static memory is pre-allocated, therefore wasteful; and leads to gross violations of C.E.R.A.R. principles.
- Dynamic memory
    - Is allocated from a separate region called the "heap."
    - Is persistent
    - Can be requested at any time.

Dynamic memory

- The heap is allocated to a program when it starts.
- The program can ask for memory from the heap at any time, and the memory persists until the program releases it.
- In C++, requesting and releasing memory is not automatic; the programmer makes it part of the algorithm!
- The whole heap is reclaimed by the operating system when the program terminates.

Advantages of Dynamic memory

- You can allocate exactly what you need.
- You can release it when you're done with it.
- It can "live" longer than any function (unlike local memory).
- Memory is semi-private: a function can only access some memory if the reference is known.
- The basis for efficient and effective data storage (we will see!)

Disadvantages of Dynamic memory

- References create more room for human error.
- An extra step to remember when allocating memory.
- It can be difficult to know when to release the memory.
- If memory is frequently allocated but none of it released, the program may eventually crash.
- Note: We're teaching C's 45-year old memory management model. C++ gives us a slightly nicer interface to it. Most modern programming languages use the heap slightly differently.

C++, and everything else

- In C/C++ you can do things to references that we do not advise in CMPT 115/117.
- In modern languages, most of these things are prevented by the language itself.
- Wait until you're a master before you start experimenting with those things.

Requesting Memory from the Heap: pseudocode

- The operator `allocate new` requests an allocation of memory from the heap to your program.
- The operator `allocate new` returns a reference to the memory. It must be stored in a pointer.
- The operator `deallocate` releases the memory, allowing the Memory Manager (MM) to reuse the memory.
- The operator `deallocate` tells the MM the address of the memory to be released.

Dynamic memory pattern

1. **Create** a reference to a thing.
2. **Ask** for memory for the thing itself.
3. **Use** memory the thing.
4. **Deallocate** memory for the thing.

refToThing myThingPtr

myThingPtr ← allocate new Thing

*myThingPtr ← thingValue

deallocate myThingPtr

- Only step 3 is optional! Step 2 is easy to forget! It can be difficult to know when to do step 4.
- The pointer `myThingPtr` is not deallocated; the memory its value points to is deallocated.

Dynamic memory example: Integers

Pseudo-code

```
refToInteger y
y ← allocate new Integer

*y ← 0

for i from 0 to 10 do
    *y ← *y + 1
done


print *y

deallocate y
```

C++

```
int *y;
y = new int;

*y = 0;

for (int i = 0; i<10; i++)
{
    *y = *y + 1;
}

cout << *y;

delete y;
```

Dynamic memory example: Floating point

Pseudo-code

```
refToFloat y
y ← allocate new Float

*y ← 0.0

for i from 0 to 10 do
    *y ← *y + 1.0
done


print *y

deallocate y
```

C++

```
float *y;
y = new float;

*y = 0.0;

for (int i = 0; i<10; i++)
{
    *y = *y + 1.0;
}

cout << *y;

delete y;
```

Dynamic memory example: Arrays

Pseudo-code

```
refToFloat y
y ← allocate new Float[10]

for i from 0 to 10 do
    y[i] ← 1.0
done


for i from 0 to 10 do
    print y[i]
done


deallocate y
```

C++

```cpp
float *y;
y = new float[10];

for (int i = 0; i<10; i++)
{
    y[i] = 1.0;
}

for (int i = 0; i<10; i++)
{
    cout << y[i];
}

delete [] y;
```

Dynamic memory example: Arrays (2)

Pseudo-code

```
refToFloat y


read size from console
y ← allocate new Float[size]

for i from 0 to size do
    y[i] ← 1.0
done


for i from 0 to size do
    print y[i]
done


deallocate y
```

C++

```
float *y;
int size;

cin >> size;
y = new float[size];

for (int i = 0; i<size; i++)
{
    y[i] = 1.0;
}

for (int i = 0; i<size; i++)
{
    cout << y[i];
}

delete [] y;
```

Dynamic memory pattern (recap)

1. **Create** a reference to a thing.
2. **Ask** for memory for the thing itself.
3. **Use** memory the thing.
4. **Deallocate** memory for the thing.

Only step 3 is optional!

refToThing myThingPtr

myThingPtr $\leftarrow$ allocate new Thing

*myThingPtr $\leftarrow$ thingValue

deallocate myThingPtr

Dynamic memory failure pattern

1. Create a reference to a thing.
2. Forget to ask for space for a new thing.
3. Use the new thing anyway.

refToThing myThingPtr

*myThingPtr $\leftarrow$ thingValue

Dynamic memory release pattern

1. Set thing free.

```
deallocate myThingPtr
```

Remember: the pointer variable is not deallocated; the memory its value points to is deallocated.

Dynamic memory release failure pattern

1. Deallocate the wrong thing.

```
deallocate *myThingPtr
```

Dynamic memory release failure pattern

1. Set thing free.
2. Use it anyway.

deallocate myThingPtr
*myThingPtr $\leftarrow$ thingValue

Dynamic memory release failure pattern

1. Don't recycle a thing when you're done with it.

```
refToThing myThingPtr

myThingPtr ← allocate new Thing
*myThingPtr ← thingValue

myThingPtr ← allocate new Thing
*myThingPtr ← thingValue
```

Dynamic memory release failure pattern

1. Make copies of the reference to a thing
2. Deallocate the thing
3. Copied references still point to the thing

```
refToThing myThingPtr
refToThing myOtherThingPtr

myThingPtr ← allocate new Thing
myOtherThingPtr ← myThingPtr

deallocate myThingPtr

*myOtherThing ← thingValue
```

Reminder: local variables

This is an example of a local reference to a local integer on the stack.

```
Integer myNum
refToInteger myNumPtr

myNum ← 7
myNumPtr ← &myNum
```

local memory (on the stack)

address	contents	
⋮	⋮	
4683953	7	myNum
4683954		myNum
4683955		myNum
4683956		myNum
4683957	4683953	myNumPtr
4683958		myNumPtr
4683959		myNumPtr
4683960		myNumPtr

Dynamic allocation details

```
refToInteger myNumPtr

myNumPtr ← allocate new Integer
*myNumPtr ← 7
```

- The space for the integer is allocated on the heap.
- The space for `myNumPtr` is on the stack.
- The integer persists until it is explicitly deallocated.
- The integer has no name.

local memory (the stack)

address	contents	
⋮	⋮	
4683953	5488482	myNumPtr
4683954		myNumPtr
4683955		myNumPtr
4683956		myNumPtr

the heap

⋮	⋮
5488482	7
5488483	
5488484	
5488485	
⋮	⋮

Dynamically allocated types

- The same thing applies to record types (structs) as well.

```
Date
   Integer year
   Integer month
   Integer day
end Date

Algorithm main()
    refToDate datePtr

    datePtr ← allocate new Date
    (*datePtr).year ← 2007
    (*datePtr).month ← 1
    (*datePtr).day ← 18
end main
```

local memory

address	contents	
4683953	5488482	datePtr
4683954		datePtr
4683955		datePtr
4683956		datePtr

the heap

address	contents
5488482	2007
5488483	
5488484	
5488485	
5488486	1
5488487	
5488488	
5488489	
5488490	18
5488491	
5488492	
5488493	

FYI: In C, it's a bit messier

The C syntax to dynamically allocate space is unpleasant.

```
refToInteger myNumPtr

myNumPtr ← allocate new Integer
*myNumPtr ← 7
deallocate myNumPtr
```

⇔

```
int *myNumPtr;

myNumPtr = (int*) malloc(sizeof(int));
*myNumPtr = 7
free(myNumPtr);
```

To allocate an array of size 10:

```
refToInteger myNumPtr

myNumPtr ← allocate new Integer [10]
for i from 0 to 9 do
    myNumPtr[i] ← 0
done
deallocate myNumPtr
```

⇔

```
int *myNumPtr;

myNumPtr = (int*) malloc(10 * sizeof(int));
for (i = 0; i<10; i++){
    myNumPtr[i] = 0;
}
free(myNumPtr);
```

Releasing dynamic memory is important

- When your program terminates, the operating system takes the heap back; all data there is lost.
- It is good practice to deallocate memory on the heap as soon as you know your program is done with the memory.
- When you deallocate memory on the heap, the heap will make it available to be used later in your program.
- If you never deallocate memory on the heap, requests for more memory from the heap will eventually fail, because the heap is a finite resource.
- To save space on lecture slides, we won't always bother with de-allocating memory in our pseudocode.

Dynamically allocated records

```
Date
   Integer year
   Integer month
   Integer day
end Date

Algorithm main()
    refToDate datePtr

    datePtr ← allocate new Date

    (*datePtr).year ← 2007
    (*datePtr).month ← 1
    (*datePtr).day ← 18

    deallocate datePtr
end main
```

⇔

```
struct Date {
   int year;
   int month;
   int day;
};

int main(){
    Date *datePtr;

    datePtr = new Date;

    (*datePtr).year = 2007;
    (*datePtr).month = 1;
    (*datePtr).day = 18;

    delete datePtr;
}
```

Arrays of dynamically allocated records

- We often need arrays of variables from some data structure. Here we will make a local array that holds dynamically allocated records.

```
Date
   Integer year
   Integer month
   Integer day
end Date

Algorithm main()
    refToDate arrayOfDates[100]
    Integer i
    for i from 0 to 99 do
        arrayOfDates[i] ←
                 allocate new Date
    done
    //do stuff
    for i from 0 to 99 do
        deallocate arrayOfDates[i]
 end main
```

```
struct Date {
   int year;
   int month;
   int day;
} ;

int main(){
    Date* arrayOfDates[100];
    int i;
    for (i = 0; i<100; i++)
        arrayOfDates[i] = new Date;

    //do stuff

    for (i = 0; i<100; i++)
        delete arrayOfDates[i];
}
```

Dynamically allocated arrays... wait a minute!

- In C++, arrays are actually pointers to the first element in the array.
- A reference to a single integer looks almost the same as a reference to an array of integers.

```
refToInteger x
refToInteger y

x ← allocate new Integer
y ← allocate new Integer[10]

*x ← 7
for i from 0 to 10 do
  y[i] ← 0
done

deallocate x
deallocate [] y
```

Summary

- Running programs have access to 3 kinds of memory: static, local, and dynamic.
- Static memory is constant, and determined at compile-time.
- Local memory is automatically allocated when a function is called, and deallocated when the function returns.
- Dynamic allocation allows persistent allocation of exactly as much memory as you ask for.
- Without dynamic allocation, most of what we'll study from now on would be impossible.
- Programming errors with dynamic memory are easy to make, and very common.