School of Computer Science

COMP SCI 1103/2103 Algorithm Design & Data Structure

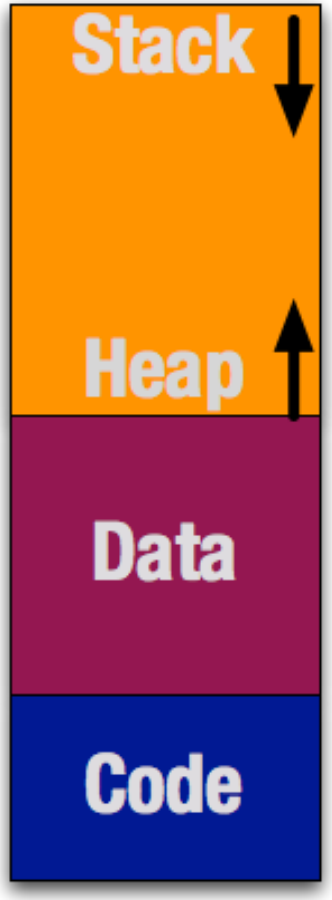
Stack & Heap

Previously on ADDS

- What are pointers?
- How can we create a pointer?
- How can we make use of a pointer?
- Pointer Arithmetic
- Arrays
- C-strings

Stack and Heap

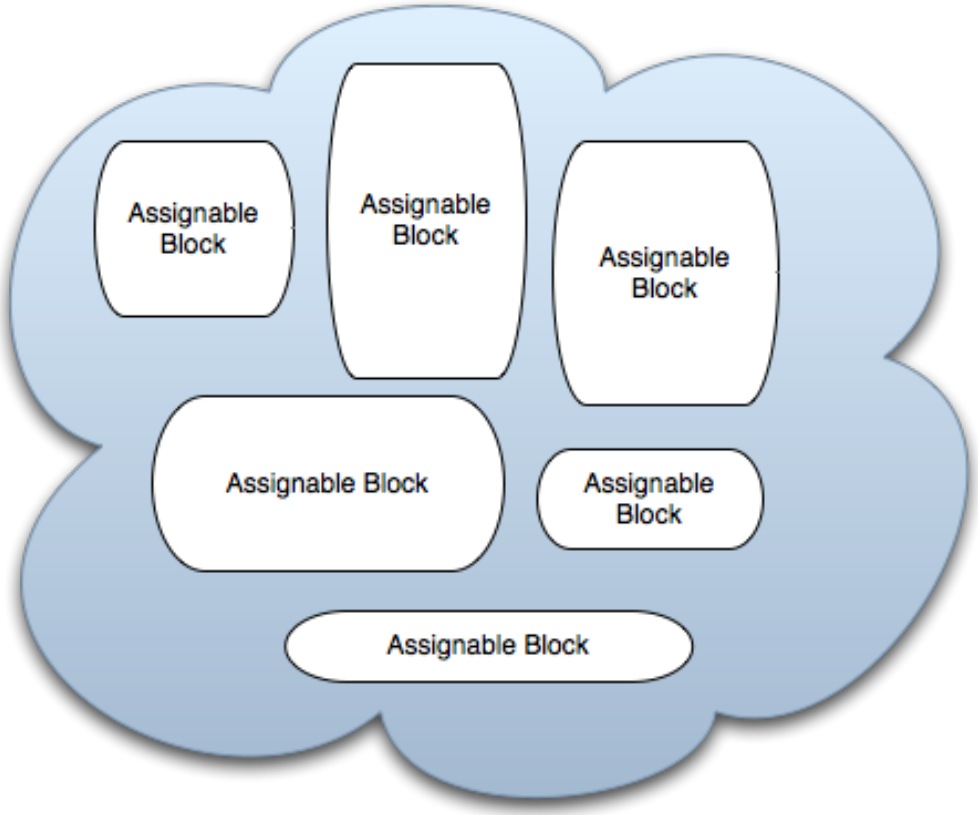
- What is stack?
- What is heap?
- What is the difference between them?



Example

- A student record management system
  - How can we store the student record?
  - How much memory do we need for a course?
  - What if some students enroll or drop out from the course?
  - The number of students is not known at the beginning
  - We need to add and remove student records dynamically

The Heap



An area reserved for dynamic variables. It is also called the freestore.

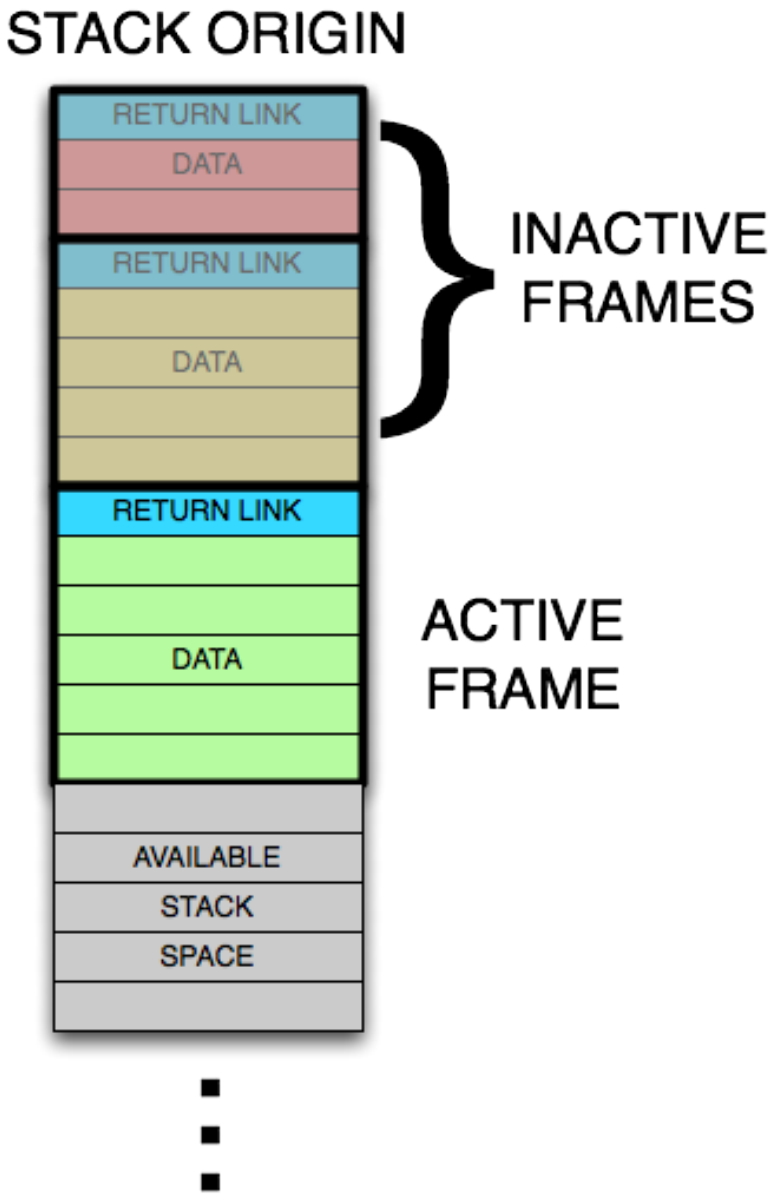
Two drawbacks:

1. Searching
2. Heap fragmentation

The Stack

Last In First Out

For allocating memory to some new variable, we just need to keep track of where the free block starts.



Stack and Heap

- The stack keeps track of all of the variables and parameters that you are currently using - also called an activation record.
  - Call too many functions - run out of stack! (stack overflow)
- The heap, or freestore, allows you to create dynamic variables and refer to them with pointers, with the *new* command. Use *delete* to hand it back.
  - Use too much of it and calls to *new* will fail!

Example

```
 2  #include<iostream>
 3  using namespace std;
 4
 5  int square(int n)
 6  {
 7  n*=n;
 8  return n;
 9  }
10
11  // compute 1^2+2^2+...+n^2
12  int squaresum(int n)
13  {
14  int result=0;
15  while(n>=1)
16      {result+=square(n);
17      n--;
18      }
19  return result;
20  }
21
22  main()
23  {
24  int n=3;
25  cout << squaresum(n);
26  return 0;
27  }
28
```

Stack

- Variables created on the stack will go out of scope and automatically deallocate when a function returns.
- Much faster to allocate in comparison to variables on the heap.
- Implemented with an actual stack data structure.
- Stores local data, return addresses, used for parameter passing.
- Can have a stack overflow when too much of the stack is used.
- Data created on the stack can be used without pointers.
- You would use the stack if you know exactly how much data you need to allocate before compile time and it is not too big.
- Usually has a maximum size already determined when your program starts.

Heap

- Variables on the heap must be destroyed manually after use and there is no scope.
- Slower to allocate.
- Used on demand to allocate a block of memory for use by the program.
- Can have fragmentation after a lot of allocations and deallocations.
- To access heap variables, you need pointers.
- Can have allocation failures.
- You would use the heap if you don't know exactly how much data you will need at runtime or if you need to allocate a lot of data.
- May lead to memory leaks.

Review (not just a review) of Pointers

- Are these sentences true? Explain your answer.
  - We use *new* to get a chunk of new memory from the stack.
    - **No! From the heap.**
  - This allows us to make changes to our memory allocation while the program is running, without using pointers.
    - **No! we do need pointers for this.**

- Where is the integer variable stored? How about "ptr"?
  - int a;
  - int * ptr= &a;
  - int * ptr= new int;
- Is it possible to store a pointer in Heap? **Double pointers**
  - int ** ptr2= new int *; //which pointer is in heap?
- Is it possible/meaningful to do these:
  - *ptr2= &ptr; *ptr=&a;
  - ptr2= &ptr;

Variables in C++

- There are three basic descriptions for how C++ handles the memory management of variables:
  - Global
  - Automatic
  - Dynamic
- What do each of these words mean to you?

Global Variables

- Global variables are declared outside of any function definition.
  - When `main` starts executing, these variables are already defined!
- They exist as long as the program is running.

Automatic Variables

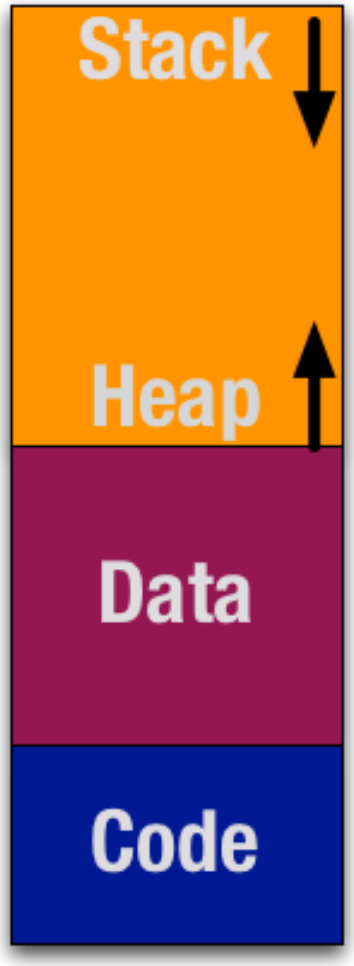
- Automatic variables are created for you to use, automatically, whenever a function is called.
  - Local to a function or the main part of the program
  - Once the function is returned, the variables are destroyed

Dynamic Variables

- Dynamic variables are created and destroyed as the program is running.
  - These variables are created with *new* and destroyed with *delete*.
  - No automatic deletion during runtime!

Where are they stored?

- The stack and heap are in a shared area.
- Where do we store each of these types of variables?
- Pre-process data area
  - Each process has a special data area set aside for variables that are global.
  - Other information for the process is also stored here.



More about stack

- All automatic variables are stored in here.
- As we call a function, another frame is put onto the stack to hold the space for the variables that we are currently using.
- When the function ends, it is removed. How?
- What if the function calls itself, without stopping?
  - recursion

What if we don't delete?

- *new* without *delete* will continue to use more and more memory.
- Losing track of memory is referred to as a memory leak - eventually, something's going to happen.
- In order to prevent memory leak, delete the memory of a pointer whenever you want to set it to another part of the memory by new.
- Returning memory to the freestore doesn't solve all of our problems. What do I mean?

Segmentation Faults

- Your program will crash if you try to:
  - Use memory that isn't allocated to you
    - Use a deleted variable
    - Use uninitialized objects;
    - Go outside arrays
  - Try to write to memory that's read only
  - Use up all the memory
- In some cases, this crash will also leave a copy of the computer's memory state – referred to as a core dump

Debugging Segmentation Fault

- Try to avoid this by taking care of the memory management by following a good programming style.
- Compile your code with –g option and use gdb to get stacktrace of the segmentation fault.
- “cout” what you expect for Debugging

Summary

- There are global, automatic (ordinary) and dynamic variables in C++.
- You need to understand how these work to make the best use of them.
- Runtime problems cause segmentation faults and crash. Core dumps help you with finding the problem.
- If you mismanage the stack, or the heap, your program will fail.
- Space is finite - management is important.
- Using memory without deallocating it will compile but will generally crash at runtime. Nasty!
- *gdb* can help you understand what has happened. (But a good code walkthrough will, too)


THE UNIVERSITY
of ADELAIDE