CS103 Unit 2b – Dynamic Memory Allocation

CSCI 103L Teaching Team

Memory that keeps on living!

DYNAMIC MEMORY ALLOCATION

A Motivating task

```
// return an integer array of size n with values 1 to n
//   stored in it
_____ ordered_array(int n) {
    // Your code here




    return        ;
}

int main() {
   // ... Call ordered_array
   // ... Use ordered_array
   return 0;
}
```

Dynamic Memory Motivation (1)

- We want to allocate an array for student scores, but I don't know how many students exist until the user inputs it.
- What size should I use to declare my array?
  - int scores[??]
- Doing the following is not supported by all C/C++ compilers and considered **bad practice** (you may NOT do it in CS103/104):

```
int num;
cin >> num;
int scores[num];  // Instead, many compilers require the array size
                  // to be statically known
```

- Also, recall local variables die when a function returns
  - What if we need that memory to KEEP LIVING even when our function ends?
- Both problems are solved with ***dynamically-allocated*** (i.e. at run-time) memory

Dynamic Memory Motivation (2)

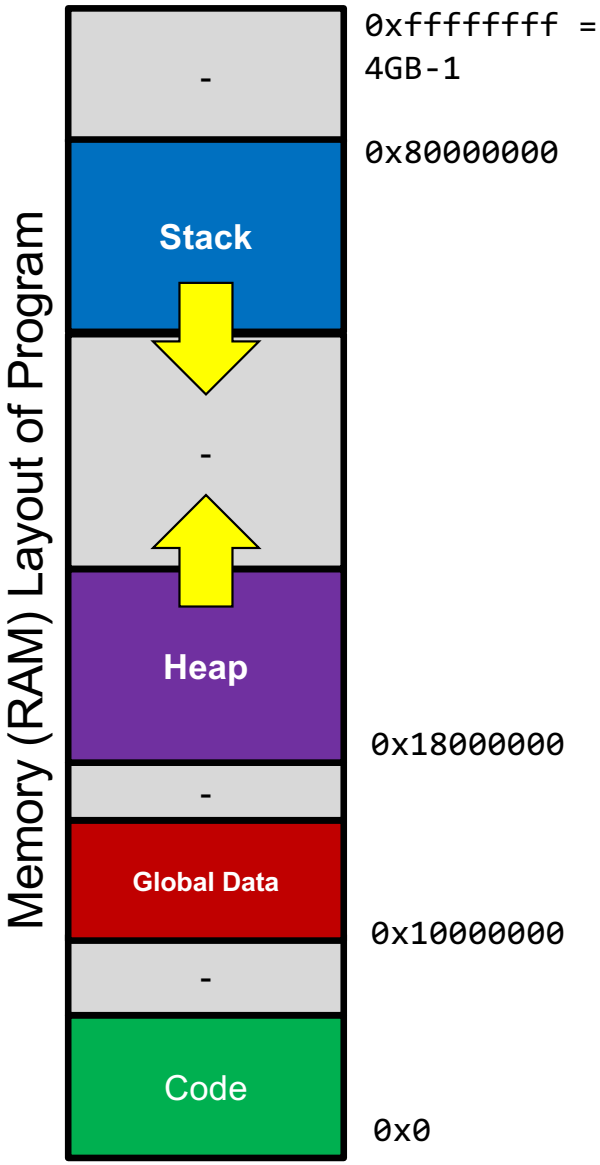
- There is ONE primary reason to use dynamic memory allocation and ONE secondary reason
- Primary reason:
  - **If we want to allocate memory in a function and have it STAY ALIVE even AFTER that function ends** (i.e. we want to manually control when memory is allocated and DEALLOCATED)
- Secondary reason
  - **If we don't know how much memory we'll need until run-time (i.e. a variable size array)**

Dynamic Memory Analogy

- Dynamic Memory is *"On-Demand Memory"*
- Analogy: Public storage rentals
  - Need extra space, just ask for some storage (using a '**new**' statement) and indicate how much you need
  - The system will allocate that memory (if it is available) from the **heap** and return the storage room number (i.e. address of / pointer to the memory) it allocated so you can access it
  - Use the pointer to access the storage/memory until you are done with it
  - Need to return it when done (using a '**delete**' statement) or else no one else will ever be able to re-use it

Dynamic Memory & the Heap

- Code usually sits at low addresses
- Global variables somewhere after code
- System stack (memory for each function instance that is alive)
  - Local variables
  - Return link (where to return)
  - etc.
- **Heap**: Area of memory that can be allocated and de-allocated in chunks during program execution (i.e. dynamically at run-time) based on the needs of the program
- Heap and stack grow toward each other…
  - In rare cases of large memory usage, they could collide and cause your program to fail or generate an exception/error



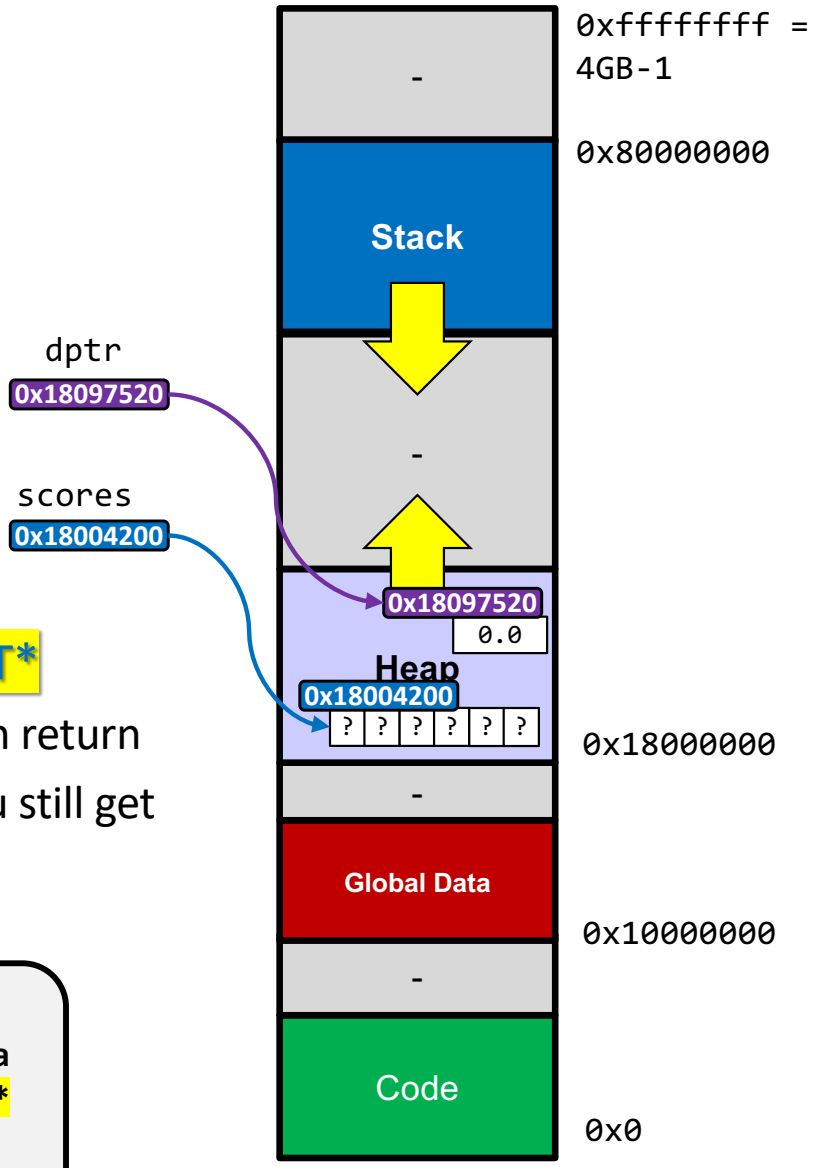
C (Pre-C++) Dynamic Allocation

- `void* malloc(int num_bytes)` function in `stdlib.h`
  - Allocates the number of bytes requested and returns a pointer to the block of memory
- `free(void * ptr)` function
  - Given the pointer to the (starting location of the) block of memory, free returns it to the system for re-use by subsequent malloc calls

This slide is for completeness. We will only use the C++ methods on the next slide in this class!

C++ new operator

- new allocates memory from heap
  - Replaced C's malloc() function
  - new should be followed with the type of the variable (and, if allocating an array, the size)
    - double *dptr = new double;
      // allocates 1 double
    - int *scores = new int[n];
      // allocates n-integer array
  - new T or new T[n] returns a pointer of type T*
    - if you ask for a new int, you get an int * in return
    - if you ask for a new array (new int[n]), you still get an int * in return]



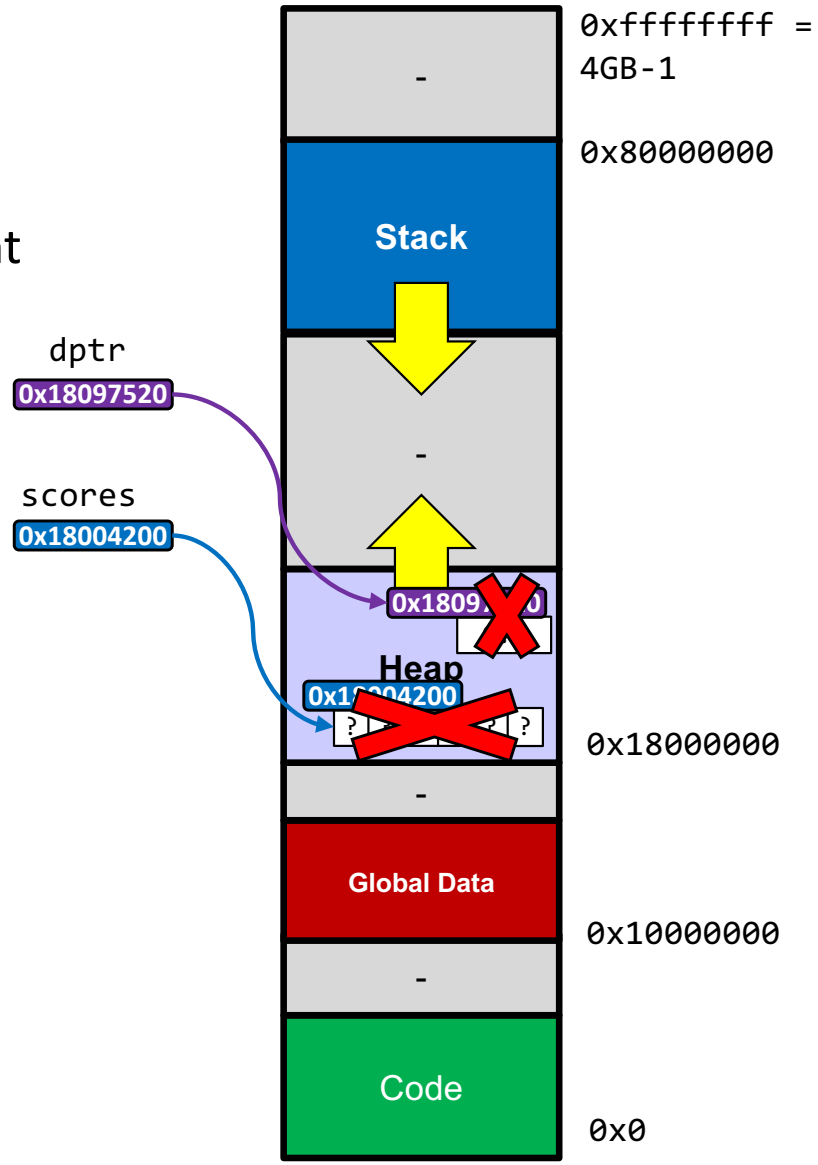
Hard and Fast Rule of Dynamic Allocation:

If you dynamically allocate a single type T or even a T array you will ALWAYS get back a pointer of type T*

```
new T;       // yields a type T*
new T[n];    // yields a type T*
```

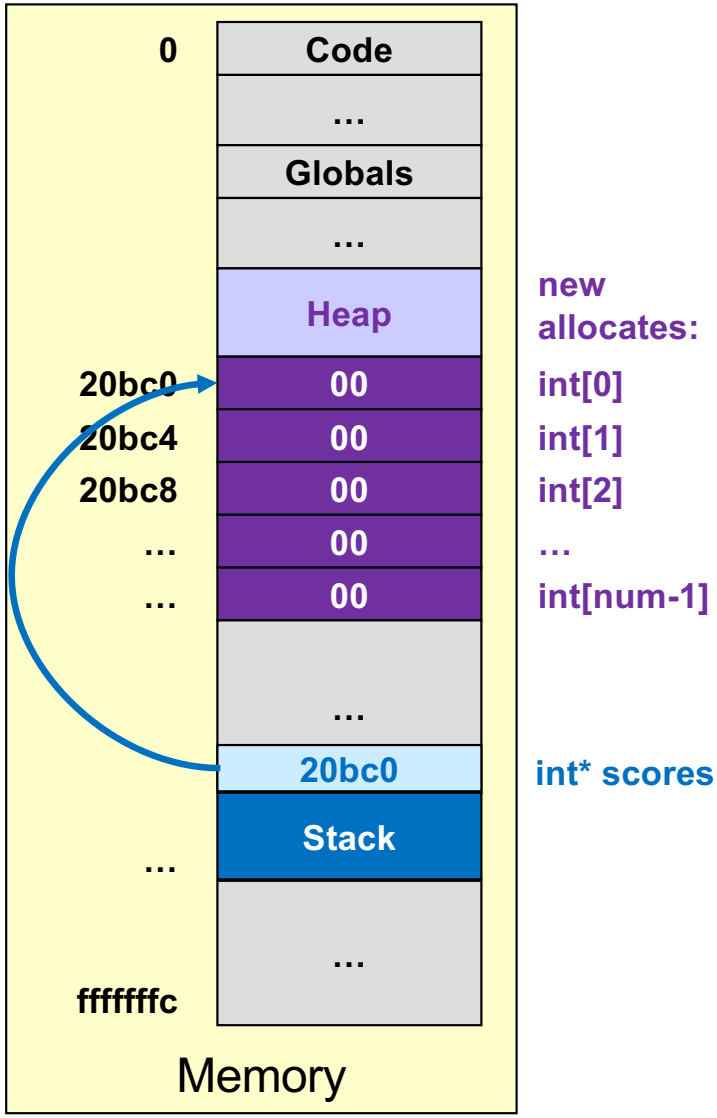
C++ `delete` operator

- **delete** returns memory to heap
  - Replaces C's `free()` function
  - Followed by the **pointer** to the data you want to de-allocate
    - `delete dptr;`
  - use `delete []` for arrays
    - `delete [] scores;`



Example of Variable Size Array with Dynamic Allocation

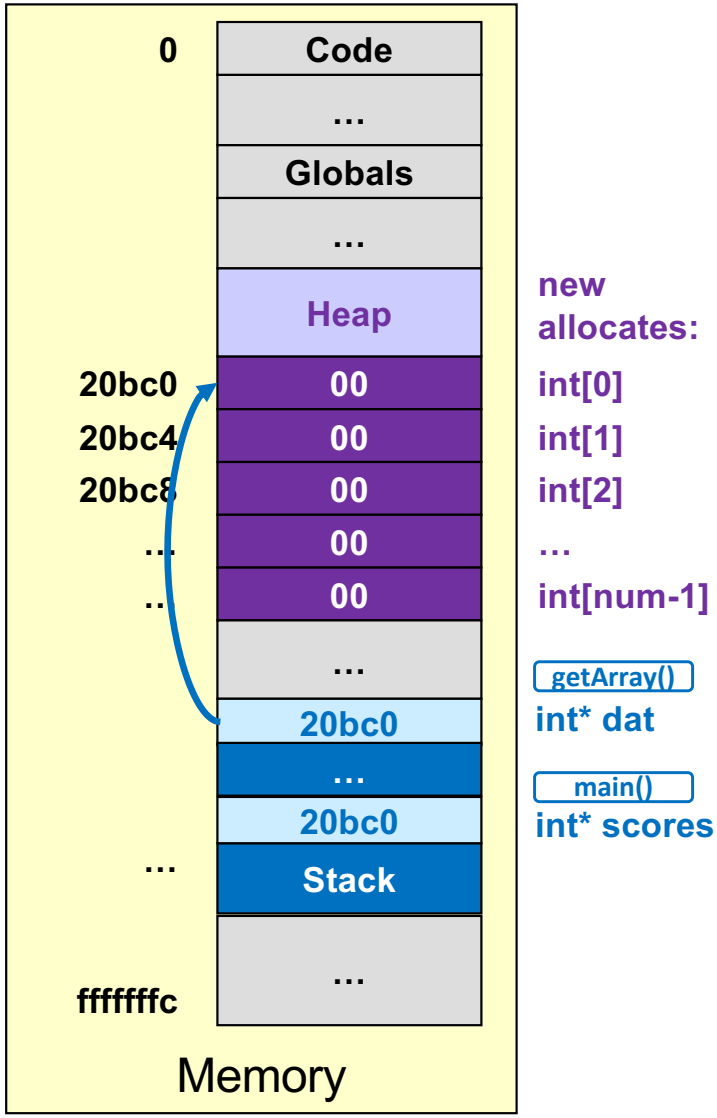
```
int main(int argc, char *argv[])
{
  int num;
  cout << "How many students?" << endl;
  cin >> num;
  int *scores = new int[num];

  // can now access scores[0..num-1]
  for(int i=0; i < num; i++){
    cin >> scores[i];
  }
  // Do more with scores

  // free up scores
  delete [] scores;
  return 0;
}
```



- Recall the two reasons to use dynamic allocation:
  - Secondary reason: Variable size array [seen in this example]
  - Primary reason: Allocate memory that should not go out of scope at the end of the function [seen in next example]

Example of Variable Size Array with Dynamic Allocation

```
int* getArray(int n)
{
  int *dat = new int[n]; // or int dat[n];
  // can now access scores[0..n-1]
  for(int i=0; i < n; i++){
    cin >> dat[i];
  }
  return dat;
}
int main(int argc, char *argv[])
{
  int num, sum = 0;
  cout << "How many students?" << endl;
  cin >> num;
  int* scores = getArray(num);

  for(int i=0; i < num; i++)  // use the array
   {  sum += scores[i]; }
  delete [] scores; // free up scores
  return 0;
}
```



Fill in the Blanks

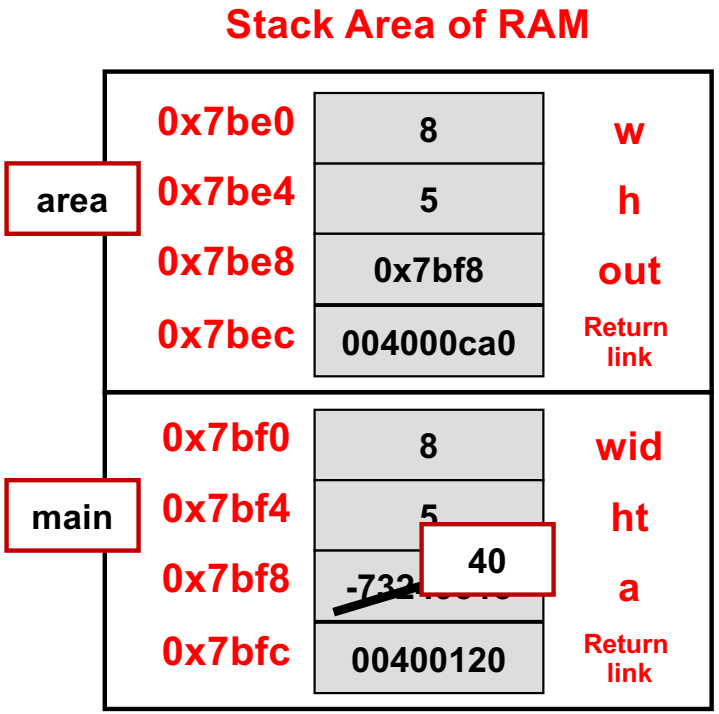
- _____ `data = new int;`
- _____ `data = new char;`
- _____ `data = new char[100];`
- _____ `data = new double[20];`
- _____ `data = new string;`
- _____ `data = new char*[10];`

Fill in the Blanks

- _____ `data = new int;`
  - `int*`
- _____ `data = new char;`
  - `char*`
- _____ `data = new char[100];`
  - `char*`
- _____ `data = new double[20];`
  - `double*`
- _____ `data = new string;`
  - `string*`
- _____ `data = new char*[10];`
  - `char**`

Correct Usage of Pointers

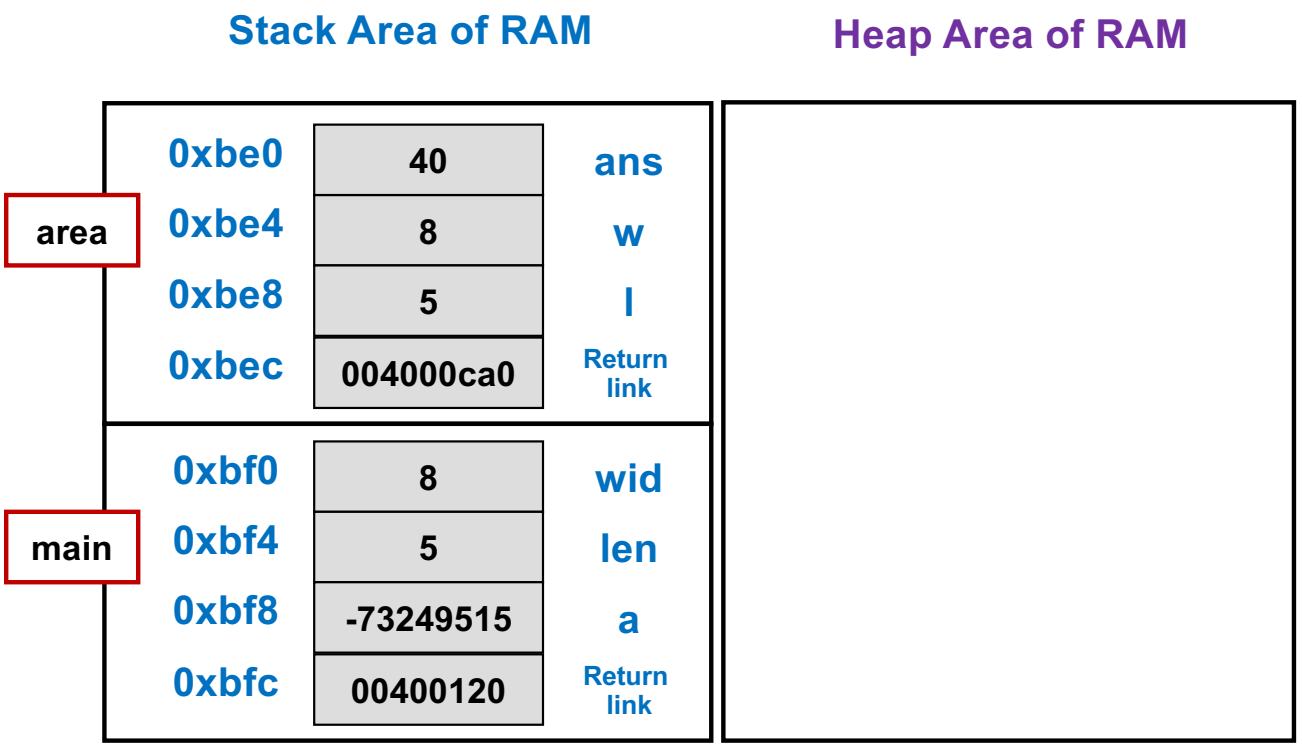
- Commonly functions will take some inputs and produce some outputs
  - We'll use a simple area' function for now even though we can easily compute this without a function
  - We could use the return value but let's practice with pointers and say mul() must return void
- Can use a pointer to have a function modify the variable of another




```
// Computes the area of a rectangle
int area1(int w, int h);
void area2(int w, int h, int* a);


int main()
{
  int wid = 8, ht = 5, a;
  area2(wid,len,&a);
  cout << "Ans. is " << a << endl;
  return 0;
}

int area1(int w, int h)
{
  return w * h;
}

void area2(int w, int h, int* a)
{
  *a = w * h;
}
```

Pointer Mistake

- **Never** return a pointer to a local variable

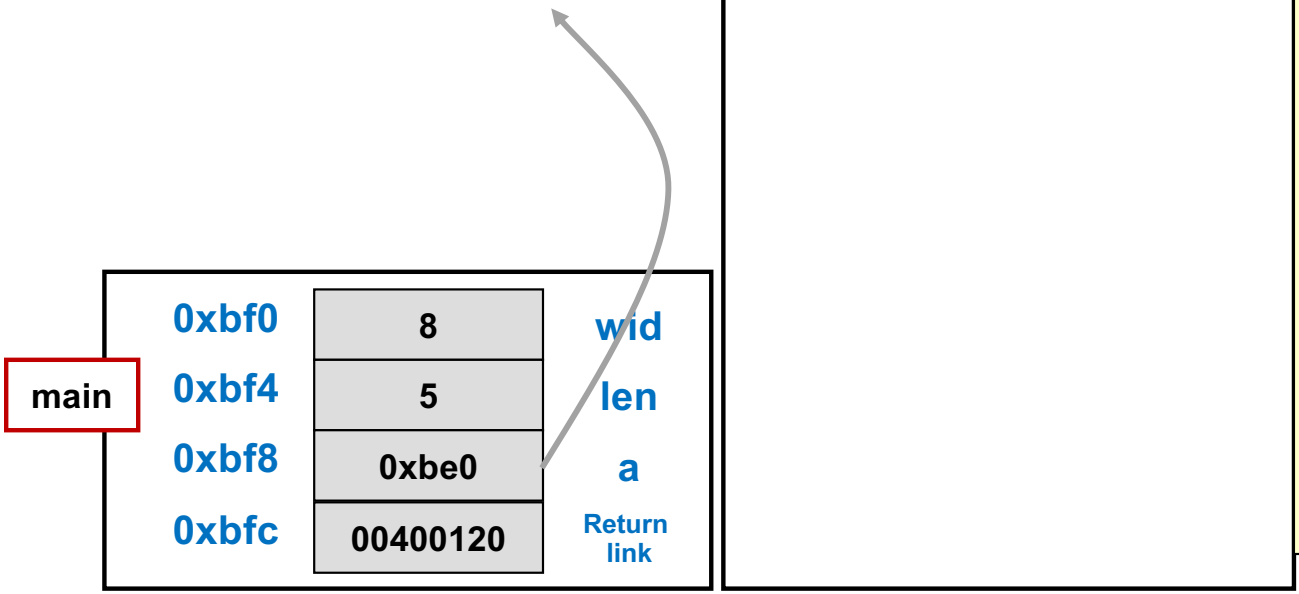



```
// Computes rectangle area,
//  prints it, & returns it
int* area3(int, int);

int main()
{
  int wid = 8, len = 5, *a;
  a = area3(wid,len);
  cout << *a << endl;
  return 0;
}

int* area3(int w, int l)
{
  int ans;
  ans = w * l;
  return &ans;
}
```

Pointer Mistake

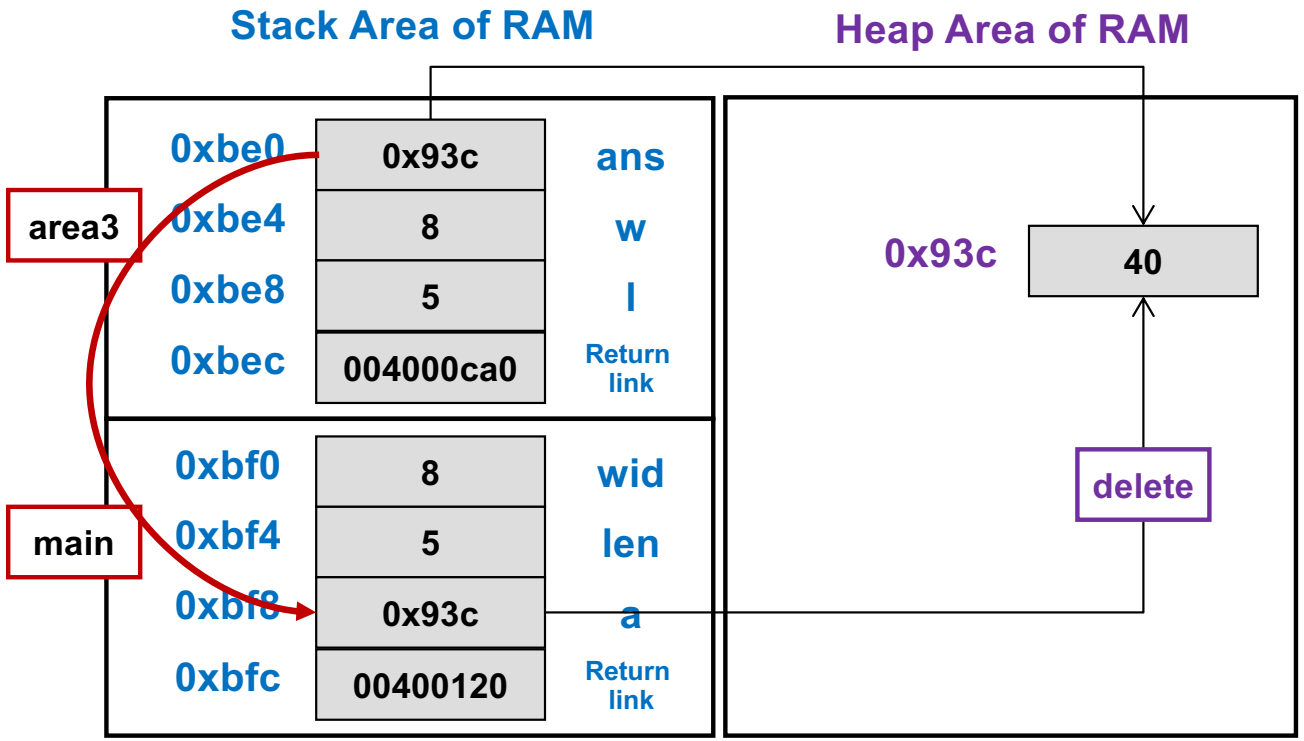
- **Never** return a pointer to a local variable
- Pointer will now point to dead memory and the value it was pointing at will be soon corrupted/overwritten
- We call this a dangling pointer (i.e. a pointer to bad or dead memory)

```
// Computes rectangle area,
//  prints it, & returns it
int* area(int, int);
void print(int);

int main()
{
  int wid = 8, len = 5, *a;
  a = area(wid,len);
  cout << *a << endl;
}


int* area(int w, int l)
{
  int ans;
  ans = w * l;
  return &ans;
}
```



Dynamic Allocation

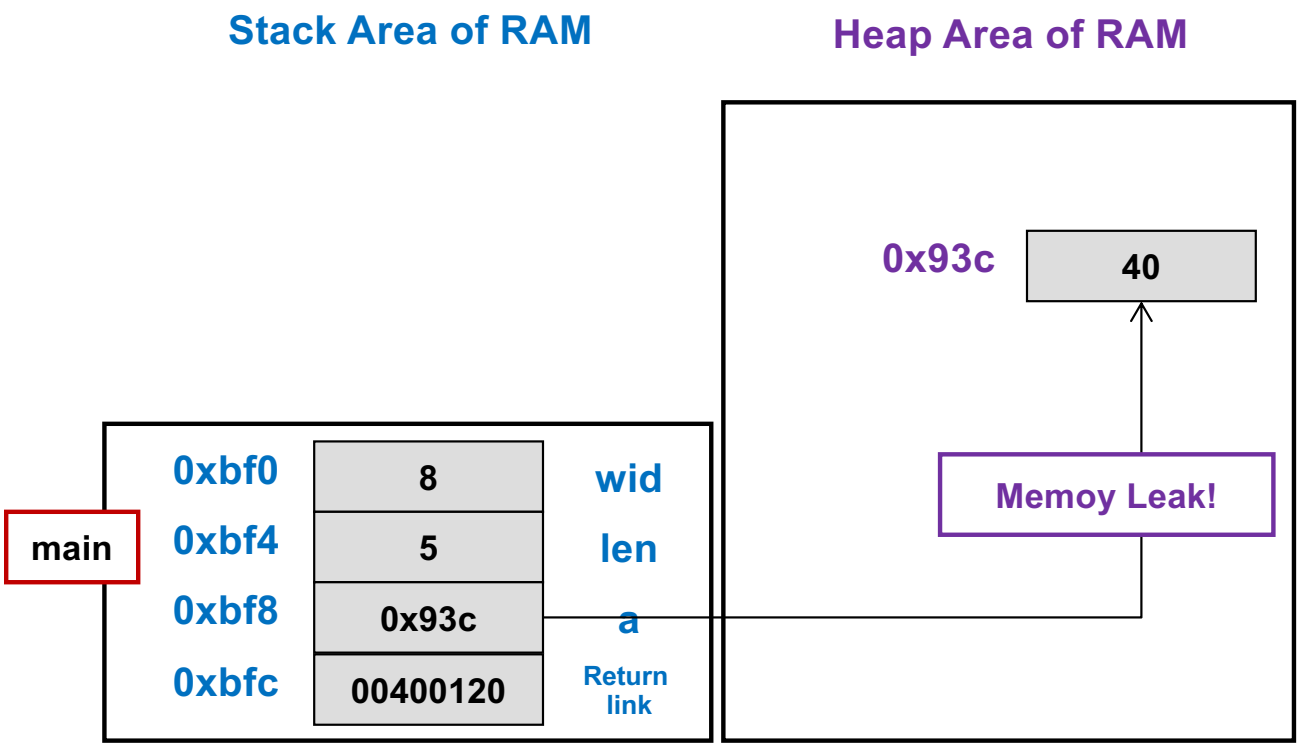
- Dynamic Allocation
  - Lives on the heap
    - Doesn't have a name, only pointer/address to it
  - Lives until you 'delete' it
    - Doesn't die at end of function (though pointer to it may)
- You must keep at least 1 pointer to a dynamic memory allocation at all times until it is deleted




```
// Computes rectangle area,
//  prints it, & returns it
int* area3(int, int);

int main()
{
  int wid = 8, len = 5, *a;
  a = area3(wid,len);
  cout << *a << endl; // 40
  delete a;
  return 0;
}

int* area3(int w, int l)
{
  int* ans = new int;
  *ans = w * l;
  return ans;
}
```

Dynamic Allocation

- Dynamic Allocation
  - Lives on the heap
    - Doesn't have a name, only pointer/address to it
  - Lives until you 'delete' it
    - Doesn't die at end of function (though pointer to it may)
- You should remember to delete the memory you allocated

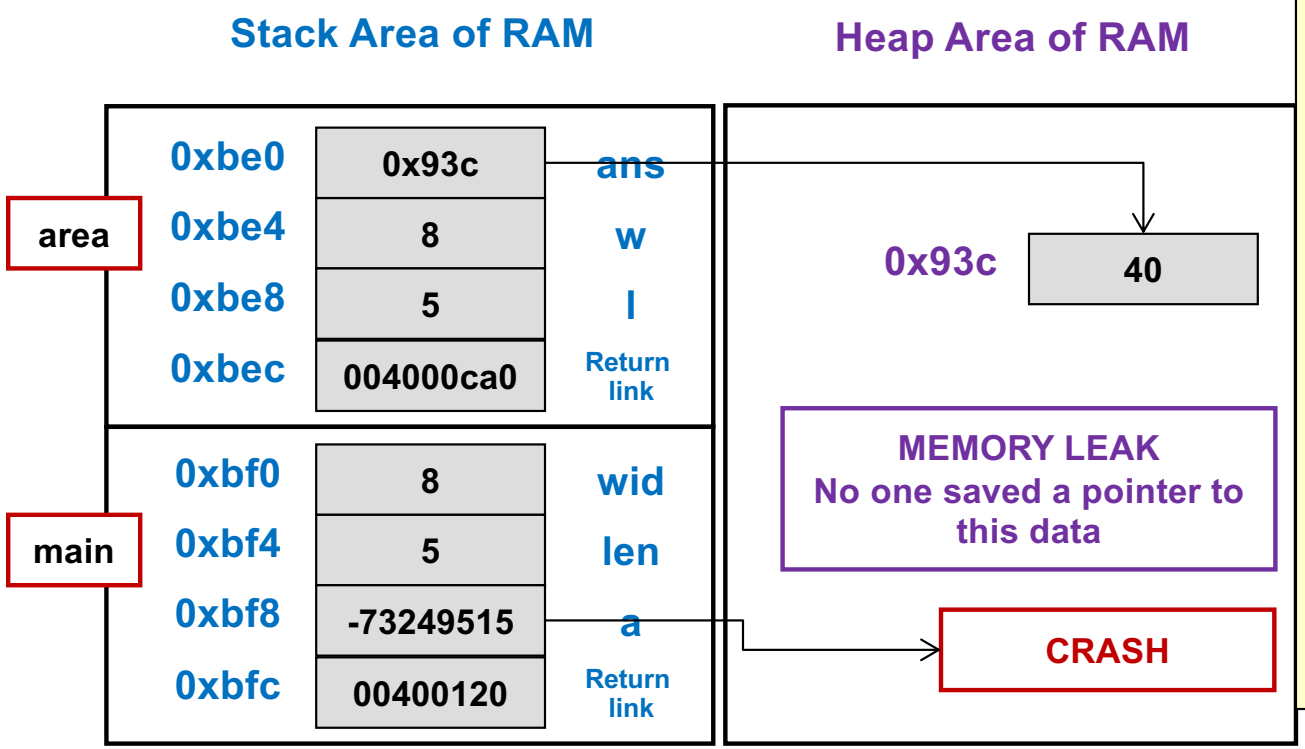



```
// Computes rectangle area,
//  prints it, & returns it
int* area3(int, int);

int main()
{
  int wid = 8, len = 5, *a;
  a = area3(wid,len);
  cout << *a << endl; // 40
  // delete a;
  return 0;
}

int* area3(int w, int l)
{
  int* ans = new int;
  *ans = w * l;
  return ans;
}
```

Dynamic Allocation

- Dynamic Allocation
  - Lives on the heap
    - Doesn't have a name, only pointer/address to it
  - Lives until you 'delete' it
    - Doesn't die at end of function (though pointer to it may)
- This code fails to save a pointer to the new int once area() finishes

```
// Computes rectangle area,
//  prints it, & returns it
int* area3(int, int);

int main()
{
  int wid = 8, len = 5, *a;
  area3(wid,len);
  cout << *a << endl; // crash
  return 0;
}

int* area3(int w, int l)
{
  int* ans = new int;
  *ans = w * l;
  return ans;
}
```



Exercises