Introduction to Programming (CS 101)

Spring 2024

Lecture 17:

Pointers again (dynamic allocation, arrays of pointers)

Instructor: Preethi Jyothi

Based on material developed by Prof. Abhiram Ranade and Prof. Manoj Prabhakaran

Recap (IA): Pointers vs. References

What is the output of the last `cout` statement?

```
#include <iostream>
using namespace std;

int main() {
  int i = 1, j = 2;
  int* p = &i, * q = &j;
  int& a = i, &b = j;
  a = b; i++; j++;
  cout << i << " " << j << endl;
}
```

- A: 2 3
- B: 3 3
- C: 1 3
- D: 3 2

References ***cannot be*** reinitialized. `a = b` would behave the same as `i = j`

Recap (IB): Pointers vs. References

What is the output of the last `cout` statement?

```
#include <iostream>
using namespace std;

int main() {
  int i = 1, j = 2;
  int* p = &i, * q = &j;
  int& a = i, &b = j;
  a = b; i++; j++;
  cout << i << " " << j << endl;
  p = q; (*p)++; (*q)++;
  cout << i << " " << j << endl;
}
```

- A 3 3
- B 4 4
- C 3 5
- D 2 3

Pointers ***can be*** reinitialized. `p = q` would assign the address of `j` (`&j`) to `p` (replacing `&i`)

Recap (II): Peril of Pointer Reassignment

What is the output of the following program?

```
#include <iostream>
using namespace std;

1  int main() {
2    int j = 1, i = 3;
3    int* a = &j;
4    int* b = a;
5    a = &i;
6    cout << *a << " " << *b << endl;
7  }
```

A 3 3

B 3 1

Say you have two pointers pointing to the same variable (line 4). Reassigning one of these pointers (a in line 5) will not reflect in the other (b).

Recap (IIIA): Pointer Arithmetic

What do the `cout` statements produce as output?

```
#include <iostream>
using namespace std;

int main() {
  int A[] = {1,2,3,4};
  int* p = A;
  p++; cout << *p << endl;
  cout << *(p++) << endl;
  cout << *(++p) << endl;
}
```

OUTPUT: 2

OUTPUT: 2

OUTPUT: 4

`*(p++)` and `*(++p)` deference from different addresses. **p++** (or **++p**) is done first, followed by dereferencing.

Recap (IIIB): Pointer Arithmetic

What is the output of this program?

```
#include <iostream>
using namespace std;

void sumslice(int* p, int* q) {
  int sum = 0;
  while(p < q) { sum += *p; p++; }
  cout << sum;
}

int main() {
  int A[] = {1,2,3,4};
  sumslice(A+1, A+4);
}
```

- A: 5
- B: 9
- C: UB (undefined behaviour)
- D: 10

`p < q` would evaluate to `true` if `p + i < q + j` where `i` is strictly less than `j`

Recap (IIIC): Pointer Arithmetic

What is the output of this program?

```
#include <iostream>
using namespace std;

int main() {
  int A[] = {0,1,2,3};
  int* p = A;
  char* c = (char*)p;
  for(int i = 0; i < 4; i++)
    cout << int(*(c+i)) << " ";
  cout << endl;
}
```

OUTPUT: **0 0 0 0**

Can cast an `int` pointer `p` to a `char` pointer `c`. Then, `c+i` would increment the pointer by 1 byte (i.e. size of a `char`). This casting allows us to print the integer byte-by-byte.

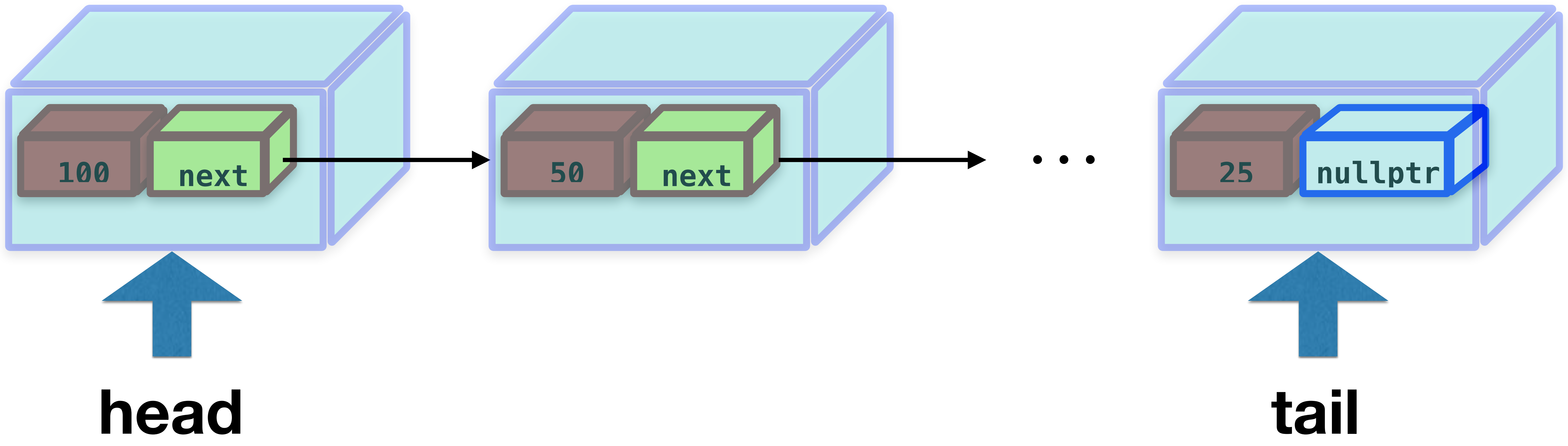
Pointers in `struct` (continued)

CS 101, 2025

Linked list

- A linked list is a data structure consisting of nodes, where each node contains one or more data fields and a pointer to the next node in the list.

```
struct node {
  int val;
  node* next;  // nullptr, if end of list
};
```



- The first node of a linked list is typically referred to as the **head** node
- The last node of a linked list is the **tail** node, and its next node is a nullptr (indicating that the list has terminated)

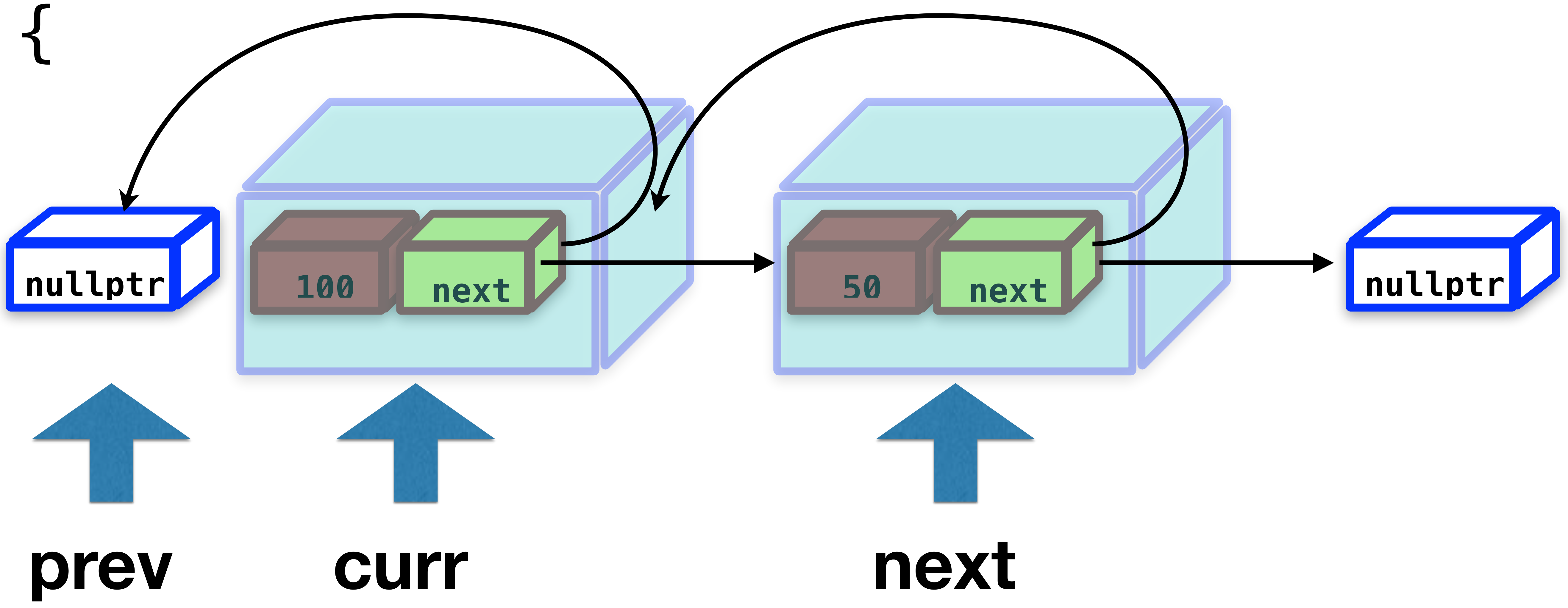
Reversing a linked list

Given the head of a list, reverse the list by changing the links between the nodes (without creating a new list)

```
node* reverseList(node* head) {
  node* prev = nullptr;
  node* curr = head;
  node* next = nullptr;

  while(curr != nullptr) {
    next = curr->next;
    curr->next = prev;
    prev = curr;
    curr = next;
  }
  return prev;
};
```

```
struct node {
  int val;
  node* next;
};
```



Dynamic allocation of memory (using pointers)

CS 101, 2025

Dynamically Allocated Memory

- Suppose we want to create a queue that can grow without limit (other than the limits set by the system policies/resources)
  – Create a queue that is as big as the maximum allowed?
  – But what if we want multiple such a priori unbounded queues?
- Ideally, the memory used for the queue should grow/shrink as the queue grows/shrinks
- More generally, we would like to create "boxes" in memory dynamically (decided at the time of program execution)

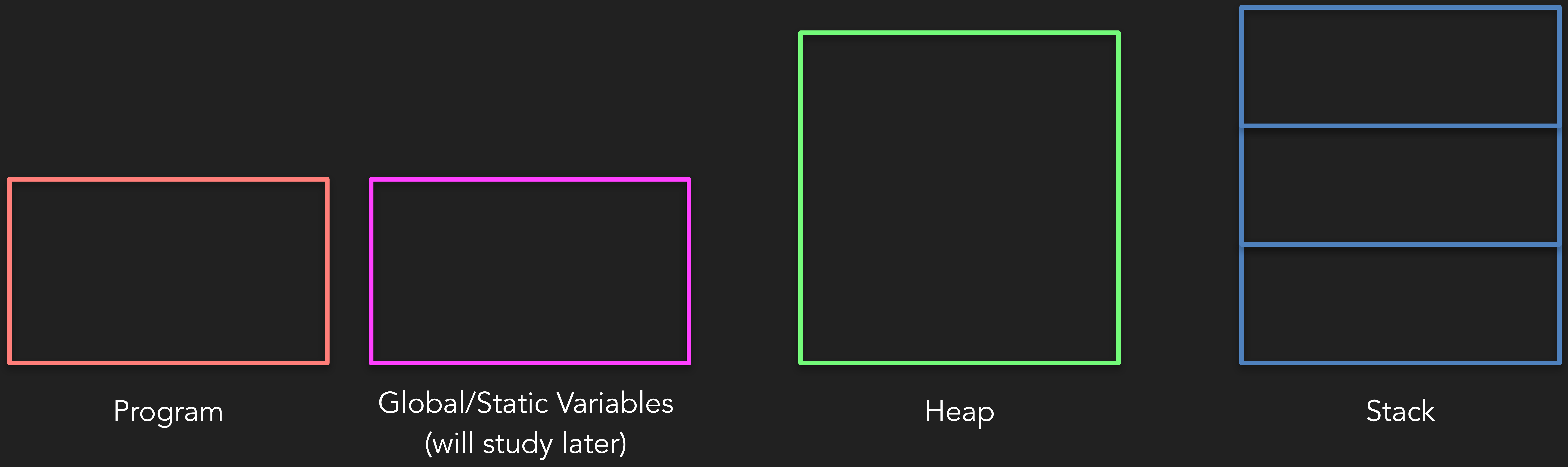
A Bit about Memory

- Each program gets its own memory space
  - Memory isolation
- Not all of the addressable space will be used by a process
  - Physical memory will not be allocated until the process needs it
  - Virtual memory
- Mapping virtual memory to physical memory is quite complex and is handled by the operating system and the hardware
  - The program only works with the virtual memory

64 bit address space has 16 Exabytes (16 billion GB)

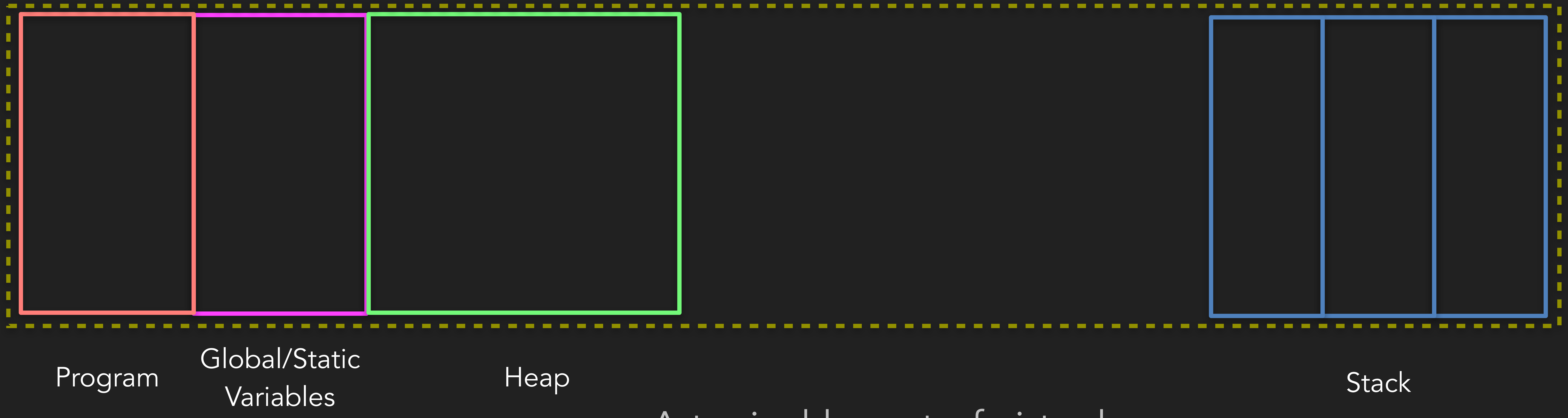
A Bit about Memory

- The virtual memory space is divided into different <u>segments</u> to hold various things needed by the program
- Dynamically allocated memory comes from one such segment called the <u>*heap*</u> which can grow/shrink as needed



A Bit about Memory

- The virtual memory space is divided into different <u>segments</u> to hold various things needed by the program
- Dynamically allocated memory comes from one such segment called the *<u>heap</u>* which can grow/shrink as needed



A typical layout of virtual memory

A Box in the Heap

- A `new` expression can be used to create "boxes" in the heap memory dynamically (i.e., decided at the time of program execution)
- But how will we access this box without a variable name?
- `new` returns a *pointer to the box*
  - It is the programmer's responsibility to save/use that pointer appropriately (and not lose it)

```
int* p = new int; // creates a new int "box" without a variable name!
*p = 7;           // we can access the new box only through its address
p = new int;      // oops! the previous box has become inaccessible now!
```

Memory Leak!

`new` and `delete` operators

- Whenever `new` is used to dynamically allocate a region of memory in the heap, you must use `delete` to deallocate the storage (i.e. return the storage to the heap)

- `new` and `delete` should be used together

```
int* p = new int; // creates a new int "box" without a variable name
      ⋮
delete p;         // p's use is over. release the memory used for it!
```

- `delete` does not actually delete the pointer variable. It returns the allocated heap memory back to the operating system.

- After deleting, the pointer variable can be assigned a new value. E.g., `p = nullptr;`

Variable Length Arrays

- Can create variable length arrays in the heap
- `new` *type*`[`*n*`]` expression returns a pointer to the first element in an array of *n* elements of the requested type
- Should free it using the operator `delete[]` (not using `delete` which will result in undefined behaviour)

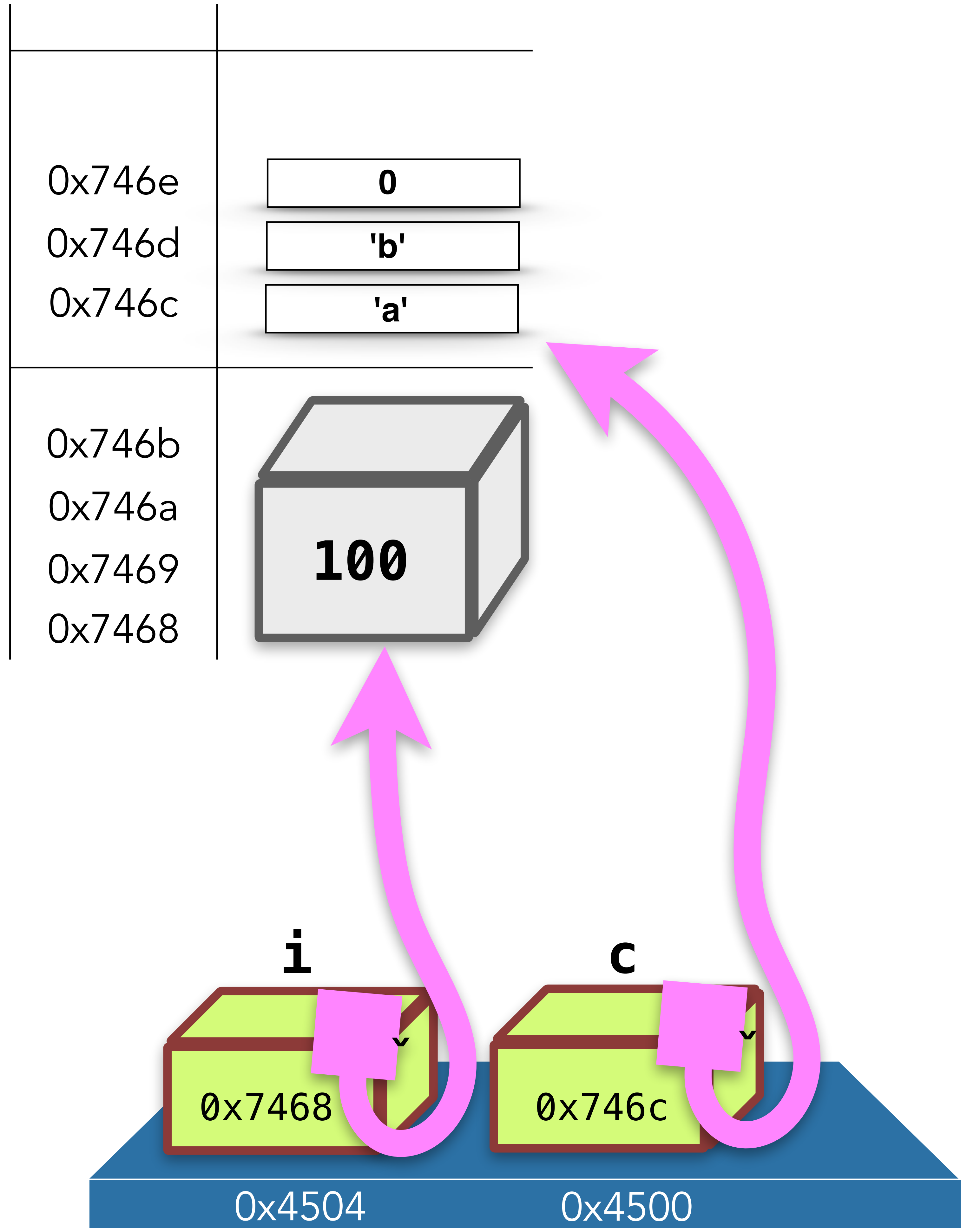
```
unsigned n; cin >> n;
int* p = new int[n]; // p[0], ..., p[n-1] are allocated now
...                  // use the array
delete[] p;          // release the memory for the entire array
p = nullptr;         // now we can overwrite the address
```

Illustration of heap memory allocation using `new` and `delete`

```
int main() {
  int* i = new int;
  char* c = new char[3];
  *i = 100;
  c[0] = 'a';
  c[1] = 'b';
  c[2] = '\0';
  delete i;
  delete[] c;
}
```

Heap memory

Stack memory



Pointer Bugs

Point out any bugs in this function. Need not be compiler errors.

`p1,p3,p4` are not deallocated (deleted) ⚠

```
void f() {
  int* p1, *p2, *p3, *p4;
  p1 = new int;
  p3 = new int;
  p4 = new int;
  p3 = p1;
  *p2 = 5;
}
```

Memory leak! `p3 = p1`, means that we are overwriting `p3` that used to contain the address of a variable in the heap. The latter can now no longer be addressed 😟 ⚠

`p4` is allocated but never used. ⚠

Dereferencing an uninitialized pointer ⚠

Example: Reading Inputs of Given Length

```
int* sort(int p[], int n); // returns an array allocated on the heap

int main() {
  int n; cin >> n;
  int* p = new int[n];
  for(int i=0; i<n; i++) cin >> p[i];
  int* q = sort(p,n);
  for(int i=0; i<n; i++) cout << q[i] << " ";
  cout << endl;
  delete[] p;           // release the memory allocated
  delete[] q;           // was allocated within sort as an array
}
```

Example: Reading Inputs of Given Length

```
void sort(int in[], int out[], int n); // cleans up all new memory

int main() {
  int n; cin >> n;
  int* p = new int[n]; int* q = new int[n]; // two arrays created here
  for(int i=0; i<n; i++) cin >> p[i];
  sort(p,q,n);
  for(int i=0; i<n; i++) cout << q[i] << " ";
  cout << endl;
  delete[] p;           // and two arrays deleted here
  delete[] q;           // easier to prevent memory leaks
}
```

Some Tips

- Whenever you use `new` or `new[]` in your program, make sure there is a matching `delete` or `delete[]`
  - Even if it may look like it doesn't matter (small program, will anyway exit right after this,...), before exiting, a good C++ program should `delete` all the heap memory allocated via `new`
    - Because `new` may do more than allocate memory and `delete` may do more than free it. (Maybe files or sockets are involved via a `new` invocation

Some Tips

- Whenever you use `new` or `new[]` in your program, make sure there is a matching `delete` or `delete[]`

Note that `delete p` doesn't change the address stored in `p`

- Accessing a deleted pointer is an error (undefined behaviour)
- Deleting an already deleted pointer is an error (crashes, typically)
  – Beware when multiple pointers may hold the same address
- C++ has several mechanisms to help with correctly using memory
  – Pre-implemented data structures in the standard library (e.g., vectors)
  – Constructor and destructor functions (coming up in a later class)

Arrays of Pointers

CS 101, 2025

Arrays of pointers

- We can define an array of pointers, that is, each element of an array will be a pointer. Example:

```
int main() {
  int* A[5];
  int B[5] = {};
  for(int i = 0; i < 5; i++) {
    A[i] = &B[i];
    cout << *A[i] << " ";
  }
}
```

OUTPUT: **0 0 0 0 0**

Arrays of pointers: Command line arguments to `main`

- So far, you run a C++ program on the command line using `./a.out`

- You can pass *additional arguments* to the main function using additional text after ./a.out. E.g.:

```
./a.out hello world
```

- This requires using an alternative overloaded definition of `main`, namely:

```
int main(int argc, char* argv[]);
```

- Here, main takes two arguments:

1. `argc`: `int` argument giving the number of space-delimited words typed on the command line
2. `argv`: Array of pointers to C-style char arrays, with `argv[i]` containing the address of the `i`th word on the command line

Print command line arguments

```
int main(int argc, char* argv[]) {
  for(int i = 0; i < argc; i++)
    cout << argv[i] << endl;
}
```

- If you use the following command: `./a.out hello world`

  then, `argc = 3` (`./a.out`, `hello`, `world`) and `argv` would have three elements of type `char*` (pointing to null-terminated C-style `char` arrays) and it would print as output:

```
./a.out
hello
world
```