An Introduction to Programming

through C++

with

Manoj Prabhakaran

Lecture 18

Pointers

Venturing outside the Stack

Based on material developed by Prof. Abhiram G. Ranade

Recap

```
int a = 2;
int* p;
p = &a;
(*p)++;   // now a==3
```

- Every variable has a unique address
  - Given by the address-of operator &
- Pointer variables can be used to store addresses
- The indirection operator * can be used to access a variable through a pointer
- So, there are two ways to access a "box" in memory: through a variable, or through an address
- Today: Creating and accessing boxes in memory which don't have any variables associated with them!

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)

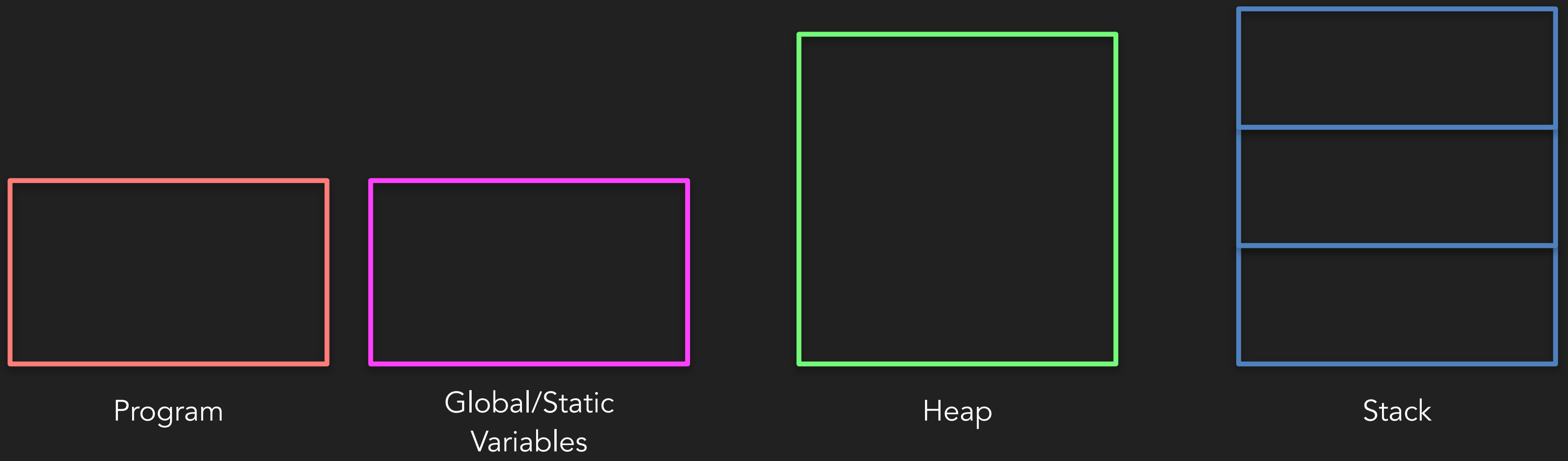
A Bit about Memory

- 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)

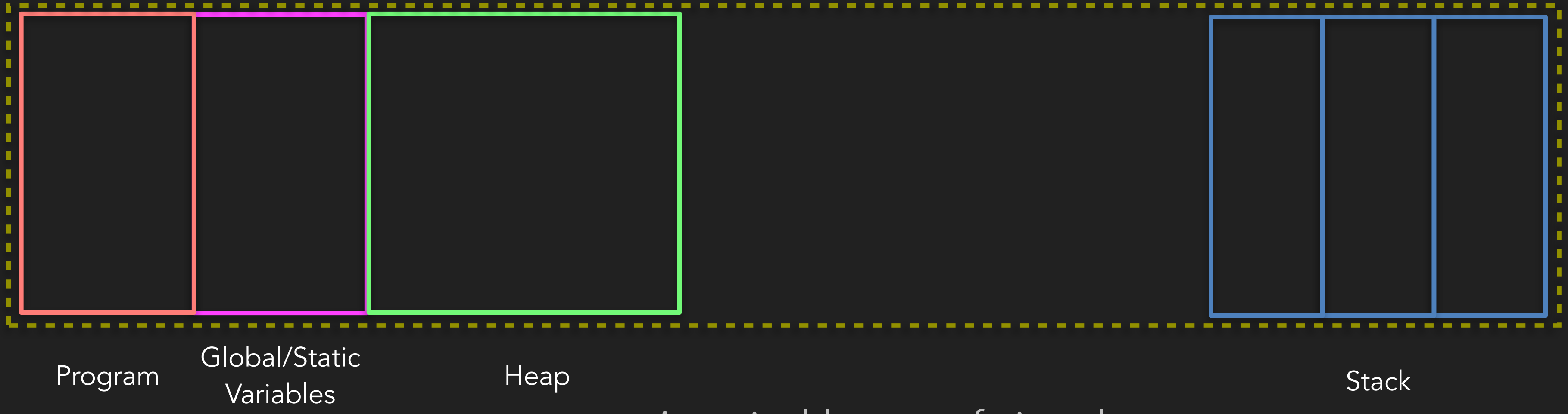
A Bit about Memory

- 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 segments to hold various things needed by the program
- Dynamically allocated memory comes from one such segment called the *heap* 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 memory dynamically (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!

Dynamic Queue

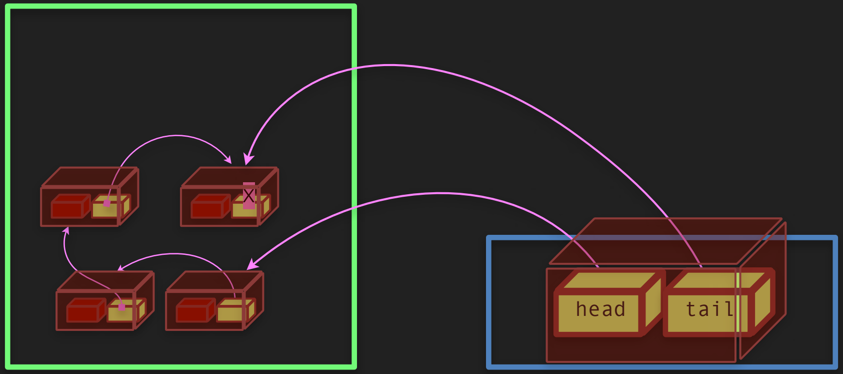
- Today's example: A queue that grows without limit (other than the limits set by the system policies/resources)
- Idea: Each queue element will be dynamically allocated in the heap
  - We will need to access the head and the tail of the queue (for dequeuing and enqueuing): two pointers
  - In fact, the address of every element in the queue should be saved (so that it doesn't become inaccessible)
    - So for n elements in the queue, we'll need n pointers too. But where will we keep them?

Dynamic Queue

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

- Solution: Use a struct which contains a pointer to another such struct, as well as a queue element (say an int)
- For convenience, we will define another struct for the whole queue

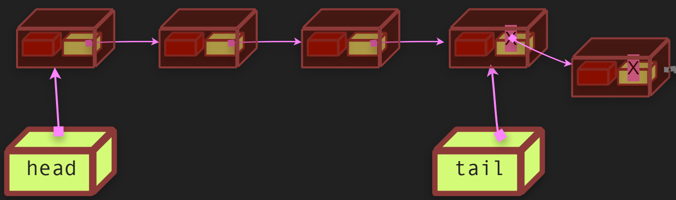
```
struct queue {
  node* head = nullptr;
  node* tail = nullptr;
  void enqueue(int v);
  bool dequeue(int& v);
};
```



Enqueuing

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

```
struct queue {
  node* head = nullptr;
  node* tail = nullptr;
  void enqueue(int v);
  bool dequeue(int& v);
};
```



new gives `next` initialised to `nullptr` as specified in the definition of the struct

- Enqueuing affects only the tail
- But **beware of corner cases!**
- If queue empty, `head`, `tail` are `nullptr`

```
if(!tail)
  tail = head = new node;
else {
  tail->next = new node;
  tail = tail->next;
}
  tail->val = v;
```

Dynamic Queue

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

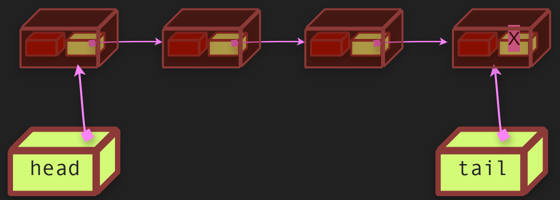
```
struct queue {
  node* head = nullptr;
  node* tail = nullptr;
  void enqueue(int v);
  bool dequeue(int& v);
};
```

```
void queue::enqueue(int v) {
  tail = (tail? tail->next : head) = new node;
  tail->val = v;
}
```

Dequeuing

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

```
struct queue {
  node* head = nullptr;
  node* tail = nullptr;
  void enqueue(int v);
  bool dequeue(int& v);
};
```




```
if(!head)
  return false;
v = head->val;
head = head->next;
if(!head) tail=nullptr;
return true;
```

Memory Leak!

- If the queue is empty, return false
- **Corner case:** Queue becomes empty after dequeuing

Freeing a Box in the Heap

- `new` allocates a box in the heap and returns a pointer to it
- If we no longer need that box, before letting go of the pointer to it, we should "release" the memory allocated for the box
- Using operator `delete`

Especially important for programs that run for a long time, and/or allocate large amounts of memory

```
int* p = new int; // creates a new int "box" without a variable name!
...               // use *p to work with the box
delete p;         // box's use over. release the memory used for it.
p = nullptr;      // now we can overwrite the address
```

Dynamic Queue

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

```
struct queue {
  node* head = nullptr;
  node* tail = nullptr;
  void enqueue(int v);
  bool dequeue(int& v);
};
```

```
void queue::enqueue(int v) {
  tail = (tail? tail->next : head) = new node;
  tail->val = v;
}
```

```
bool queue::dequeue(int &v) {
  if(!head) return false;
  v = head->val;
  node* old_head = head;
  head = head->next;
  delete old_head;
  if(!head) tail=nullptr;
  return true;
}
```

Dynamic Queue

Demo

```
void queue_demo() {
  queue Q;
  for(int i=0; i < 10000000; i++) // 10 Million times
    Q.enqueue(1);
}
```

Memory Leak! When Q goes out of scope, its "contents" allocated using new are not deleted!

```
int main() {
   for(int i=0; i < 100; i++)
       queue_demo();
}
```

Dynamic Queue

```
void queue_demo() {
  queue Q;
  for(int i=0; i < 10000000; i++) // 10 Million times
    Q.enqueue(1);
  Q.clear();
}
```

Later: Calling such a "clean up" function when an object goes out of scope can be automated

```
int main() {
   for(int i=0; i < 100; i++)
      queue_demo();
}
```

```
struct queue {
  node* head = nullptr;
  node* tail = nullptr;
  void enqueue(int v);
  bool dequeue(int& v);
  void clear(){
    int v;
    while(dequeue(v)) {}
  }
};
```

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
```

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]; // instead of relying on VLA support: int p[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.
    - Tools which analyse a program for bugs may detect such errors "that don't matter", and the real bugs will remain hard to find

Some Tips

- Whenever you use `new` or `new[]` in your program, make sure there is a matching `delete` or `delete[]`
  - May be hidden inside other functions (e.g., `enqueue`, and `dequeue` or `clear`)

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
  - Constructor and destructor functions (Coming up)
  - Pre-implemented data structures in the standard library (Later)
  - Smart Pointers (not covered)