# CS3520 Programming in C++
# Pointers and Memory

Ken Baclawski
Fall 2016

# Outline

- Pointers
  - Compared with references
  - Syntax
- Memory Organization
  - Free Store
- Memory Allocation
  - The new operator
  - The delete operator
- Memory Management Strategies
  - Container
  - Tree
  - DAG
  - Garbage Collection
- Low level operations
- Assignment #6

# Pointers

# Pointers

- A pointer is a variable whose value is the address of an object
- If an object has type `T` then the type of pointer to `T` is `T*`
- One can have pointers to pointers to any number of levels
- The address of an object is obtained by using `&`
- Dereferencing a pointer is performed by using `*`
- References are similar to pointers but the syntax is different.

```
int p = 100;
int q = p;
q = 200;
cout << p << endl;

// Output is 100

int x = 100;
int* y = &x;
*y = 200;
cout << x << endl;

// Output is 200

int a = 100;
int& b = a;
b = 200;
cout << a << endl;

// Output is 200
```

# Pointers vs References

- References are a high-level concept and are safe
- Pointers are a low-level concept and are *not* safe
- What is different about pointers?
  - Store a pointer and use it later
    - The object may not exist (dangling pointer)
  - Pointer arithmetic
  - Index operator
  - Null pointer (written `nullptr`)
- None of these are possible for a reference itself

# Pointer Syntax

- Write the `*` immediately after the type name
- This is the same style used by reference types
- Dereference by prepending an asterisk
- Invoking a method on a pointer uses the `->` abbreviation.
- Set a pointer variable to null by using `nullptr`
  - Never set to `0` or `NULL`
- Check whether a pointer variable is null by using it in an `if` conditional
  - Never compare with 0 or `nullptr`
- Avoid taking the address of a stack variable.

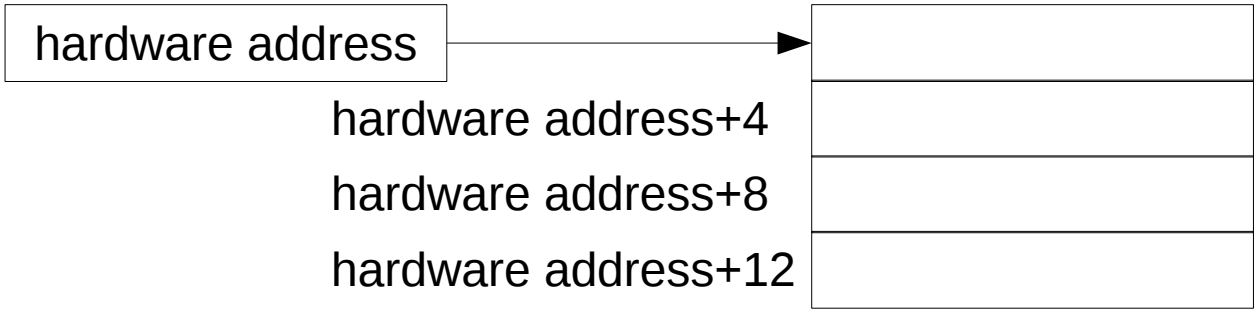
```
Node* node = nullptr;
// ...
if (node) {
  cout << “Node is okay” << endl;
} else {
  cout << “Node is null” << endl;
}

for (Node* n : nodes) {
  if (n) {
    cout << n->label() << endl;
  }
}
// n->label() is the same as
// (*n).label()
```

# Arrays

- Arrays are a low-level construct
- An array is a collection of objects that are adjacent to one another in memory
- The address of the first object is the address of the array
- The address of another object in the array is obtained by adding a multiple of the size of the object
  - Very efficient
  - Accessing beyond the end of an array is not prevented

```
int* array = new int[4];
```

```
array
```




Pointer arithmetic:

`array+1` points to the second element in the array

`array+2` points to the third element in the array

...

Index notation:

`array[0]` is the same as `*array`

`array[1]` is the same as `*(array+1)`

`array[2]` is the same as `*(array+2)`

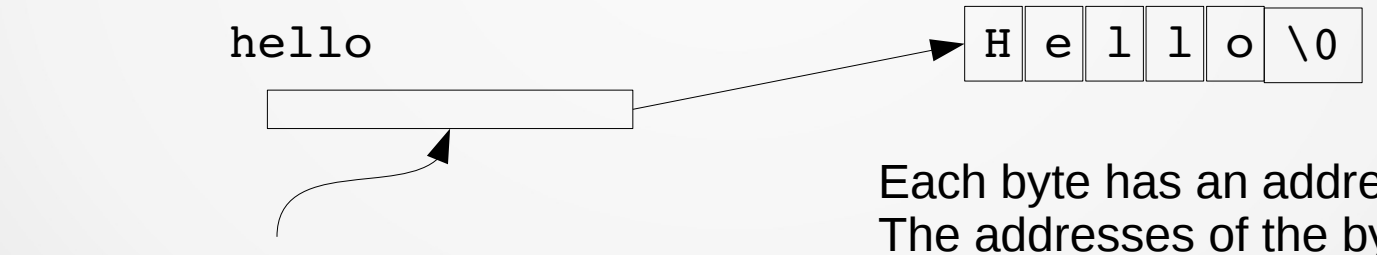
...

# Low level strings

- A C-style string is an array of bytes
- The last byte has the binary value '\0'

```
char* hello = "Hello";
```

This is not allocated in the stack.
This string uses 6 bytes.




This is allocated in the stack frame. An address is 8 bytes (64 bits)

Each byte has an address.
The addresses of the bytes are consecutive:
If the address of 'H' is addr, then the address of 'e' is addr + 1.

# Pointers to functions

- In C one can use pointers to functions
- This is done implicitly by the compiler when functions are passed as parameters
- The syntax for function pointer types is obscure
  - Ritchie, the author of C, considered it an experiment that failed
- This is a low level construct that should be avoided
  - Functors and lambda expressions are more versatile, are object-oriented, and are much safer

# Why and how to use pointers

- Many software systems and libraries use pointers
- So you will need to know about pointers
- The best way to deal with pointers is to wrap them
  - In a user-defined class for this purpose
  - In a smart pointer
- Many STL classes wrap pointers
  - Containers
  - Iterators
  - Smart pointers

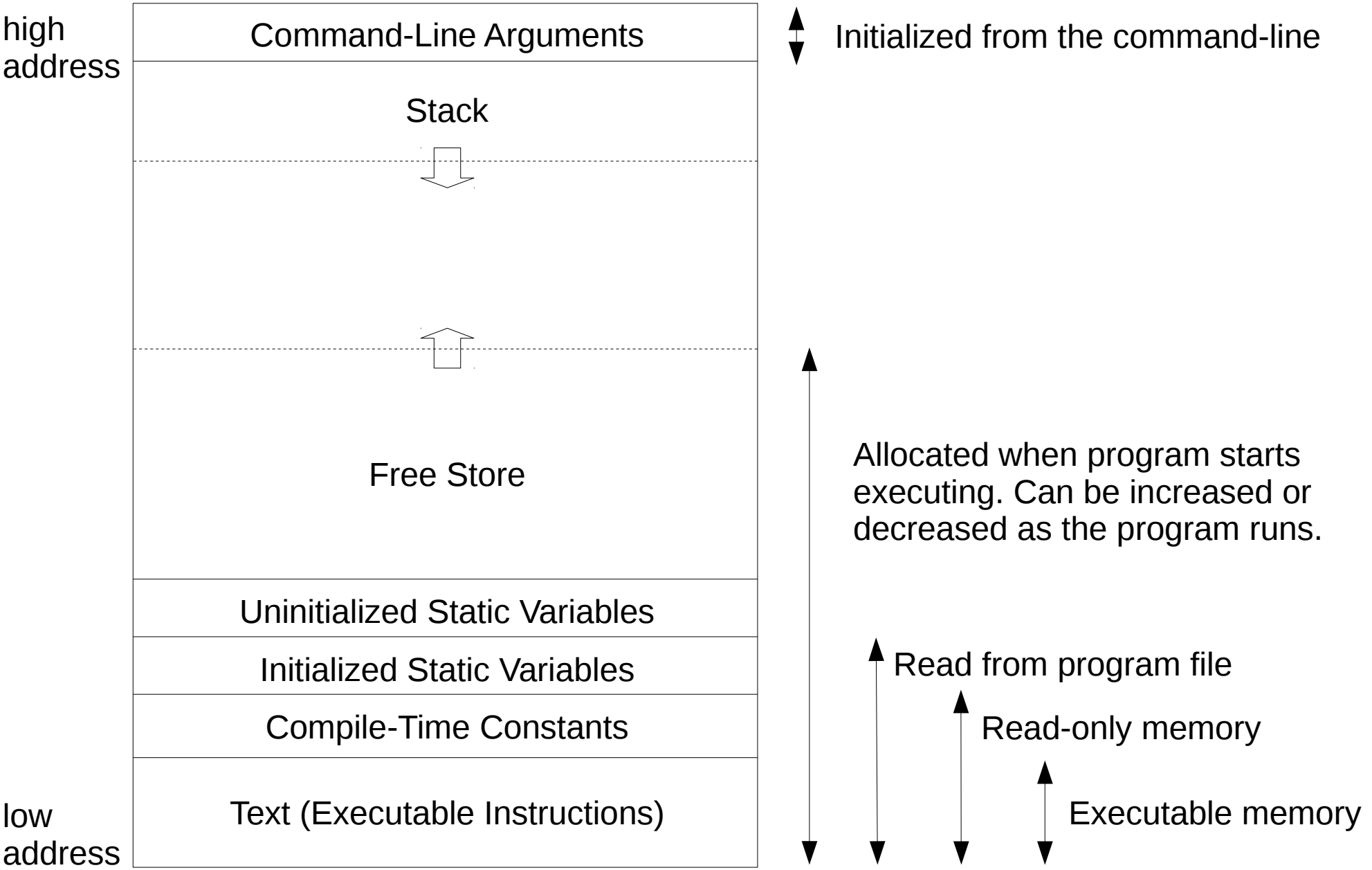
# The SDL2 Graphics Library

- Simple DirectMedia Layer
- Cross-platform development library
  - Access to graphics hardware
  - Low level access to keyboard and mouse
  - Low level access to audio and joystick
- Uses pointers to access its structures
- Uses global functions for all behavior
- Typical example of C-based software libraries

# Memory Organization

# Kinds of memory

- Automatic: objects on the stack
- Static: allocated at compile-time
- Dynamic: objects in the free store
- Text: executable program code
- Command-Line
  - strings on the command-line
  - environment strings



# Command-Line Arguments

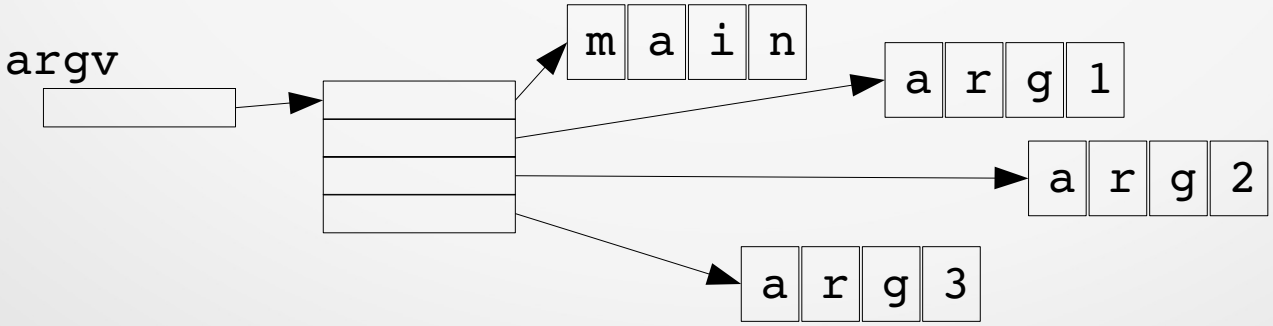
- One can specify strings on the command-line

```
./main arg1 arg2 arg3
```

- The full syntax for the main function declartion is this:

```
int main(int argc, char** argv)
```

- The number of command-line arguments is `argc`
- The second parameter is a pointer to a pointer (array)



# Free Store

- Objects that are allocated at run-time are either on the stack or in the free store.
- All local variables in a function are allocated on the stack
- What is in the free store?
  - Containers use the free store for their elements
  - Strings use the free store for the array of characters
  - Objects that have been allocated with `new`
- Objects in the free store are deallocated with `delete`
  - The deallocated space is reused

# Stack vs Free Store

- Allocation and deallocation of stack space is more efficient
  - Change the stack pointer
- Allocation in the free store requires finding unused space
  - It must be large enough
- Stack objects are less persistent
  - Only exist while a method is executing
- Free store objects are more persistent
  - Continue to exist after method returns

# Stack vs Free Store

- Stack allocation and deallocation is automatic
  - One name for stack objects is “automatic”
- Free store objects must be explicitly created and destroyed
  - The free store is also called the “heap”
- The free store is not as safe as the stack
  - Dangling pointers
  - Null pointers
  - Memory leaks
- However, the free store is essential for dynamic storage

# Memory Allocation and Deallocation

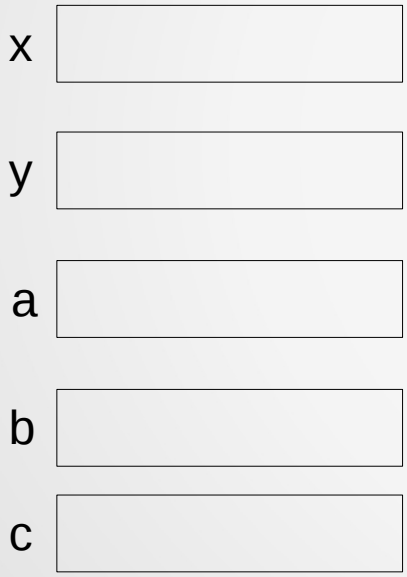
# Creation and Destruction

- Create objects in the free store with `new`
  - Memory is allocated
  - Constructor is called
- Destroy objects in the free store with `delete`
  - Destructor is called
  - Memory is deallocated

# The `new` operator

Creates a single object or
an array of objects

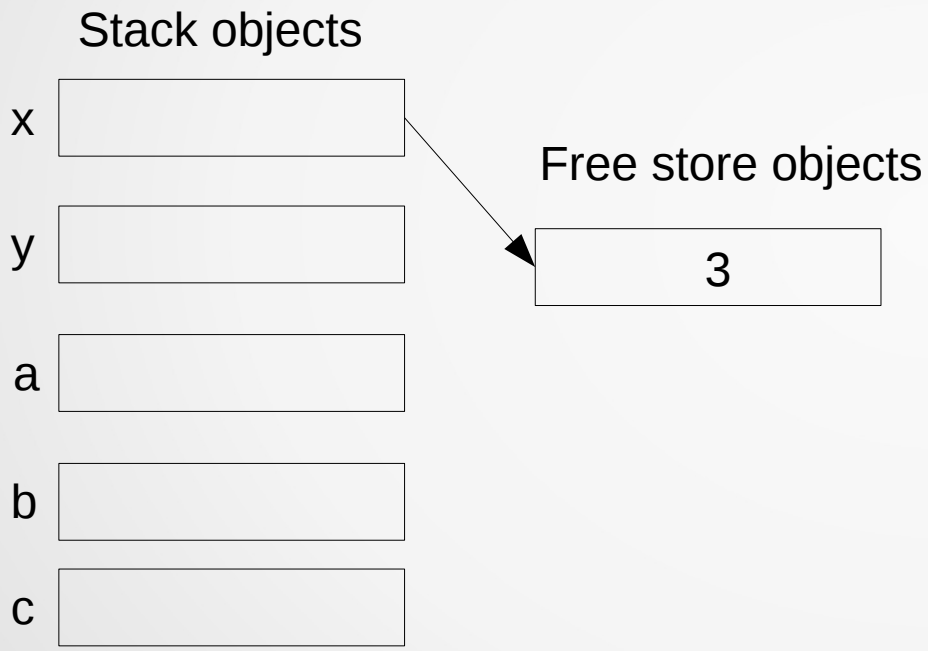
Stack objects



# The `new` operator

Creates a single object or an array of objects

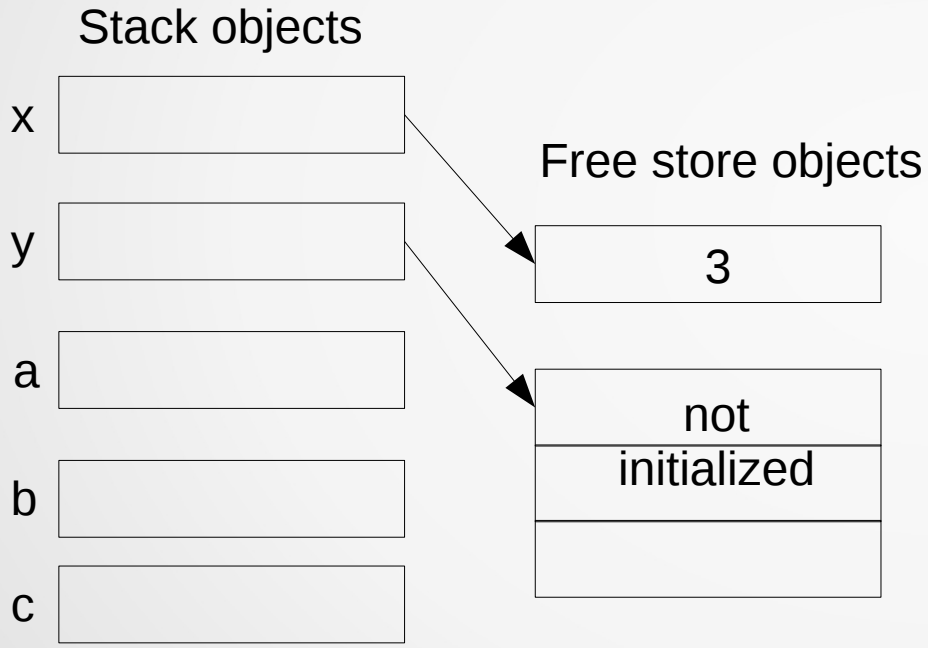
```
int* x = new int(3);
```



# The `new` operator

Creates a single object or an array of objects

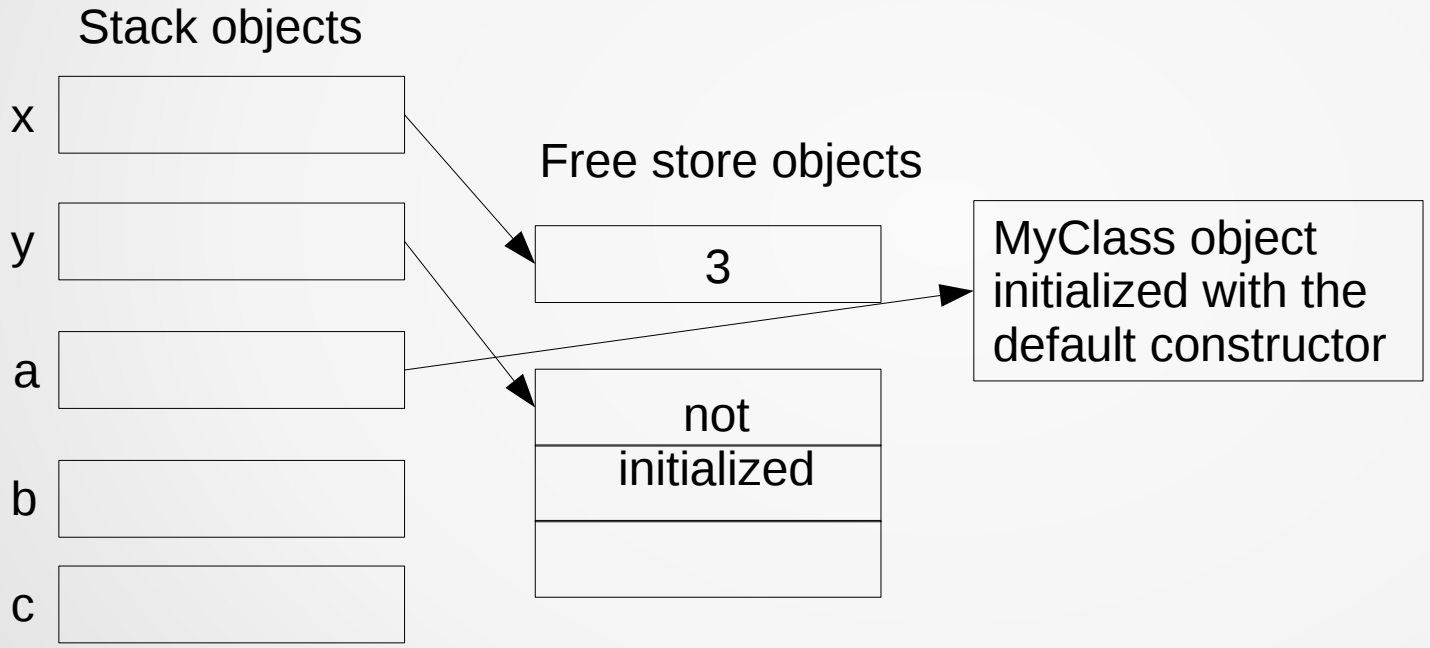
```
int* x = new int(3);
int* y = new int[3];
```



# The `new` operator

Creates a single object or an array of objects

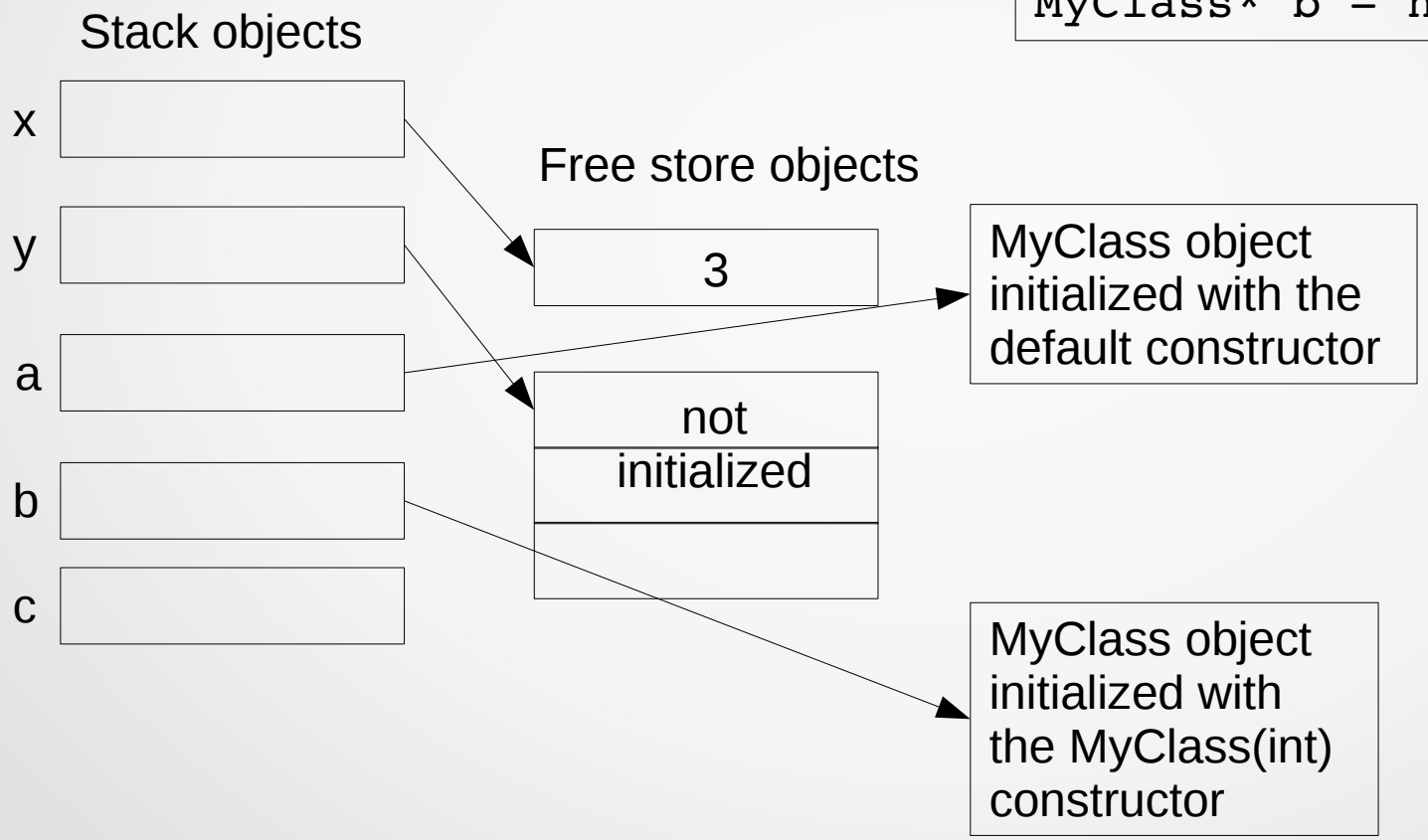
```
int* x = new int(3);
int* y = new int[3];
MyClass* a = new MyClass;
```



# The `new` operator

Creates a single object or an array of objects

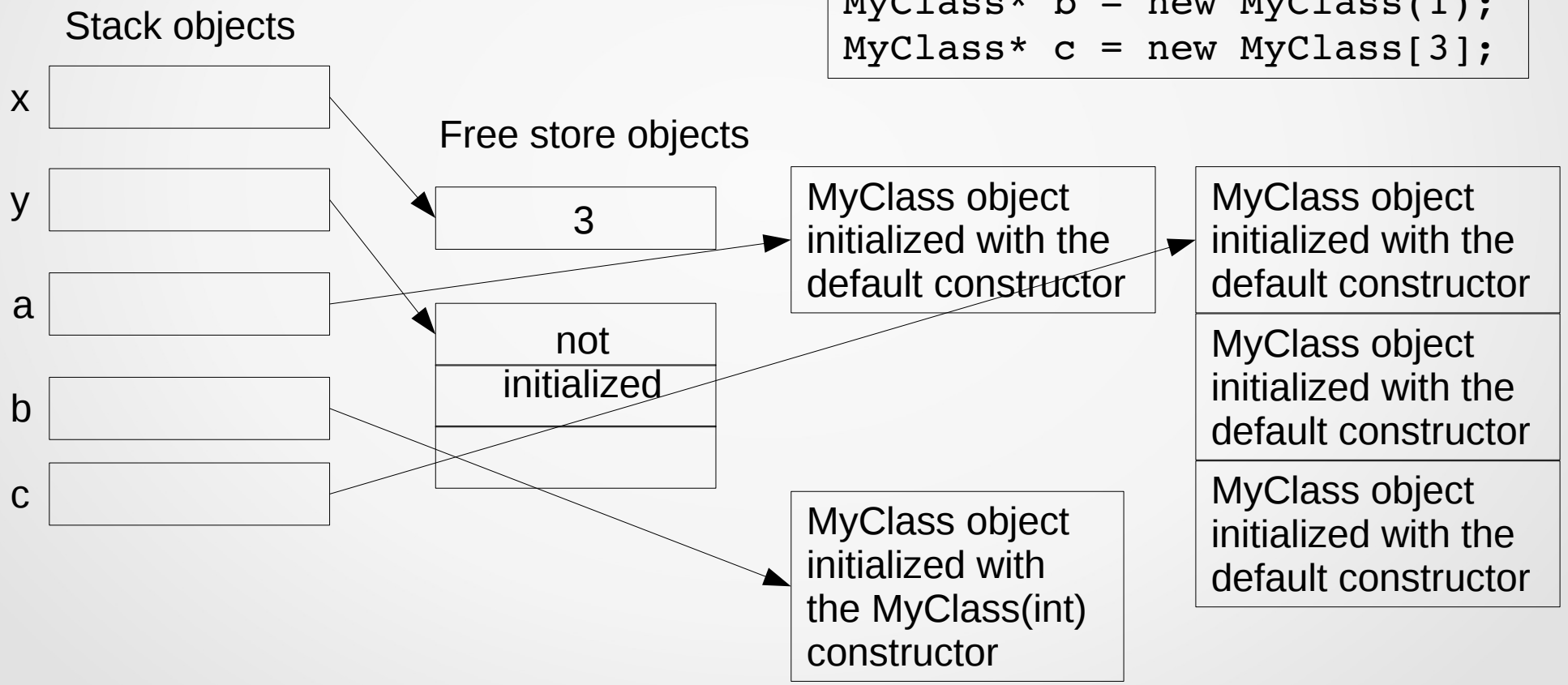
```
int* x = new int(3);
int* y = new int[3];
MyClass* a = new MyClass;
MyClass* b = new MyClass(1);
```



# The `new` operator

Creates a single object or an array of objects

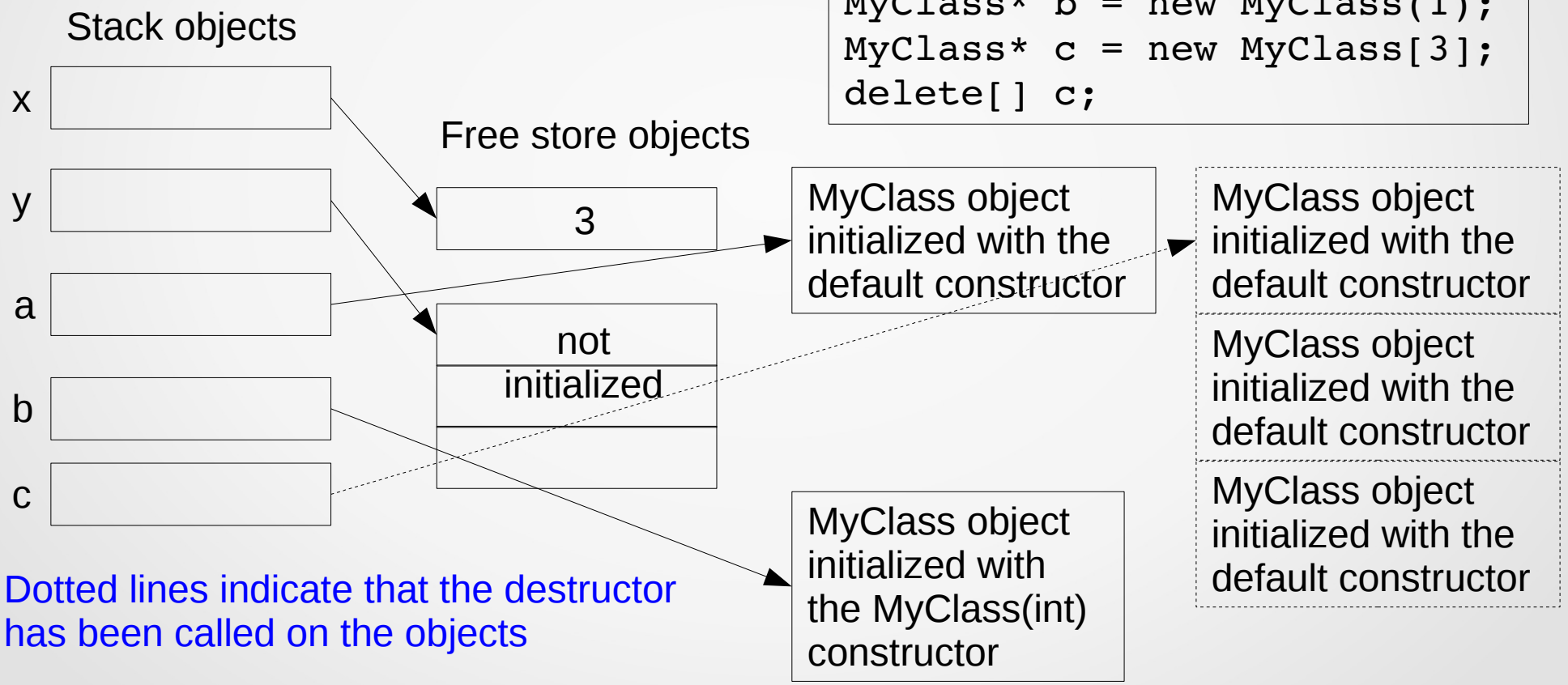
```
int* x = new int(3);
int* y = new int[3];
MyClass* a = new MyClass;
MyClass* b = new MyClass(1);
MyClass* c = new MyClass[3];
```



# The `delete` operator

Destroys a single object or an array of objects

```
int* x = new int(3);
int* y = new int[3];
MyClass* a = new MyClass;
MyClass* b = new MyClass(1);
MyClass* c = new MyClass[3];
delete[] c;
```

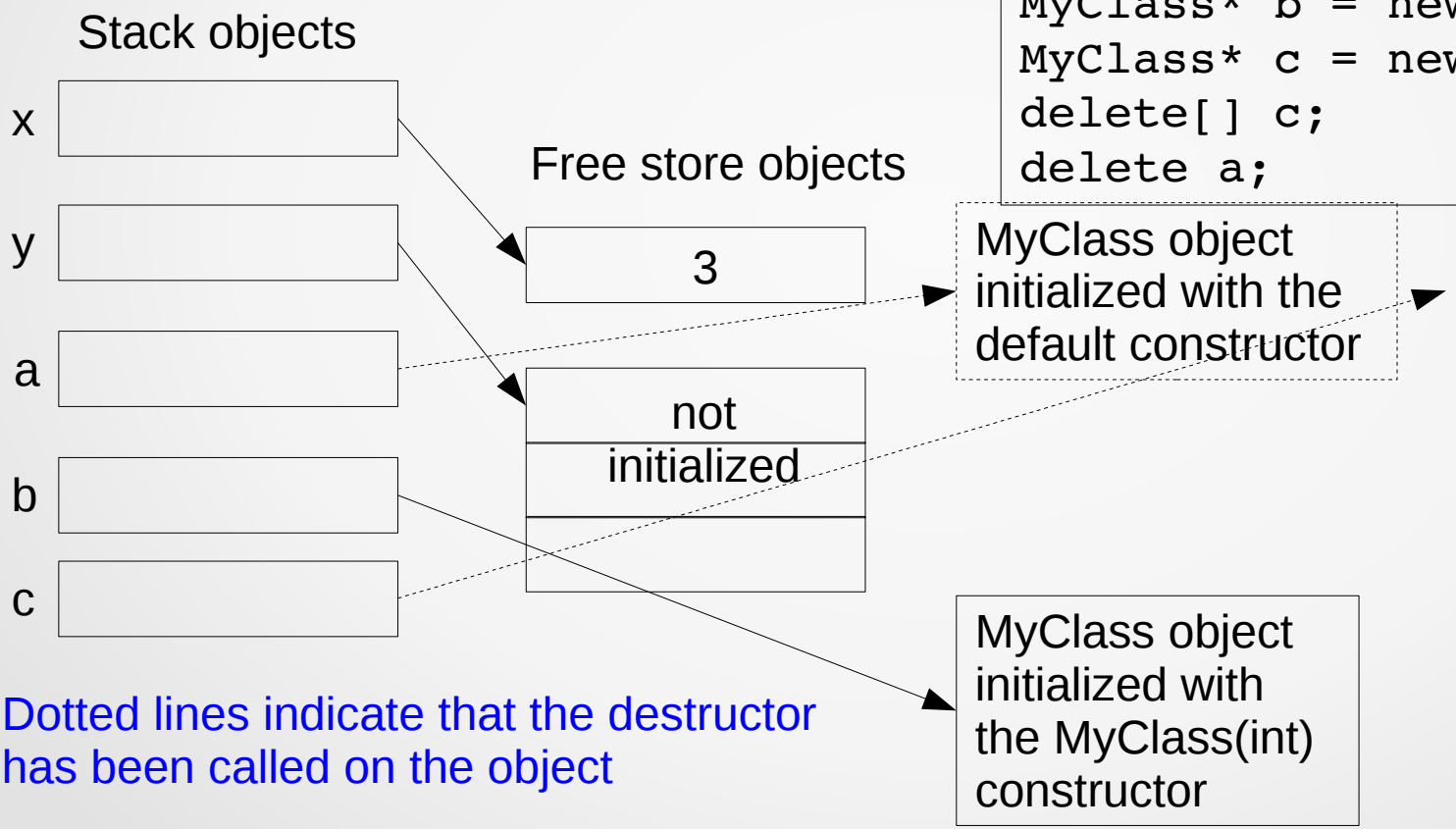



Dotted lines indicate that the destructor has been called on the objects

# The `delete` operator

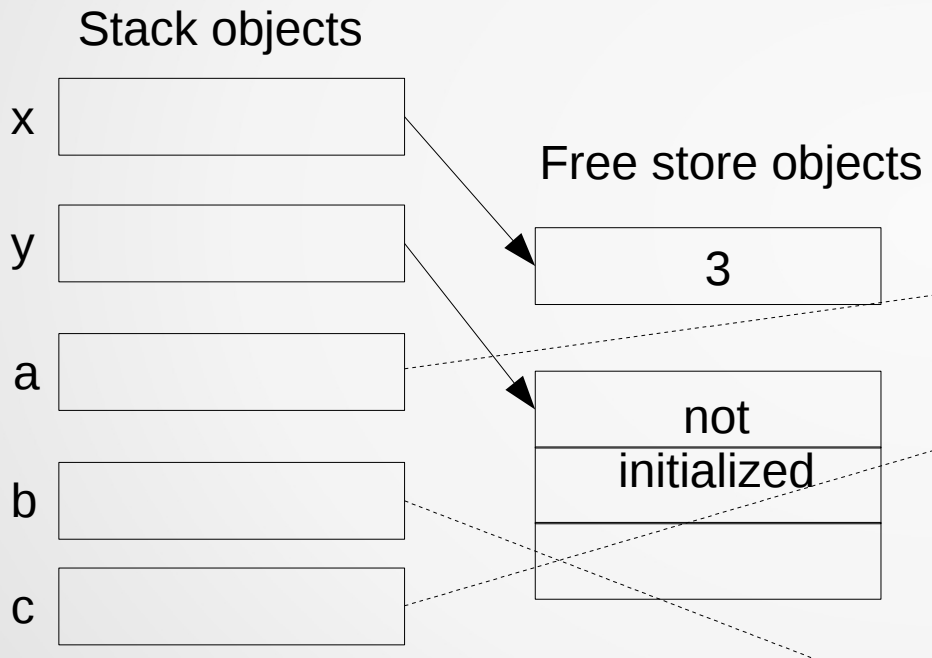
Destroys a single object or an array of objects

```
int* x = new int(3);
int* y = new int[3];
MyClass* a = new MyClass;
MyClass* b = new MyClass(1);
MyClass* c = new MyClass[3];
delete[] c;
delete a;
```



# The `delete` operator

Destroys a single object or an array of objects

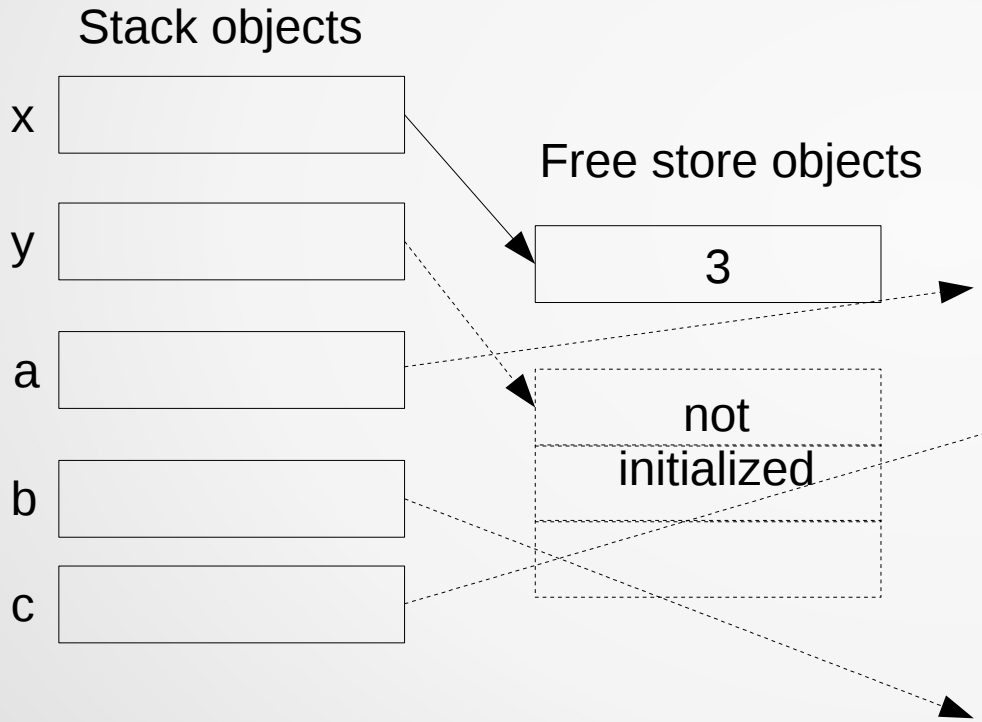



```
int* x = new int(3);
int* y = new int[3];
MyClass* a = new MyClass;
MyClass* b = new MyClass(1);
MyClass* c = new MyClass[3];
delete[] c;
delete a;
delete b;
```

Dotted lines indicate that the destructor has been called on the object

# The `delete` operator

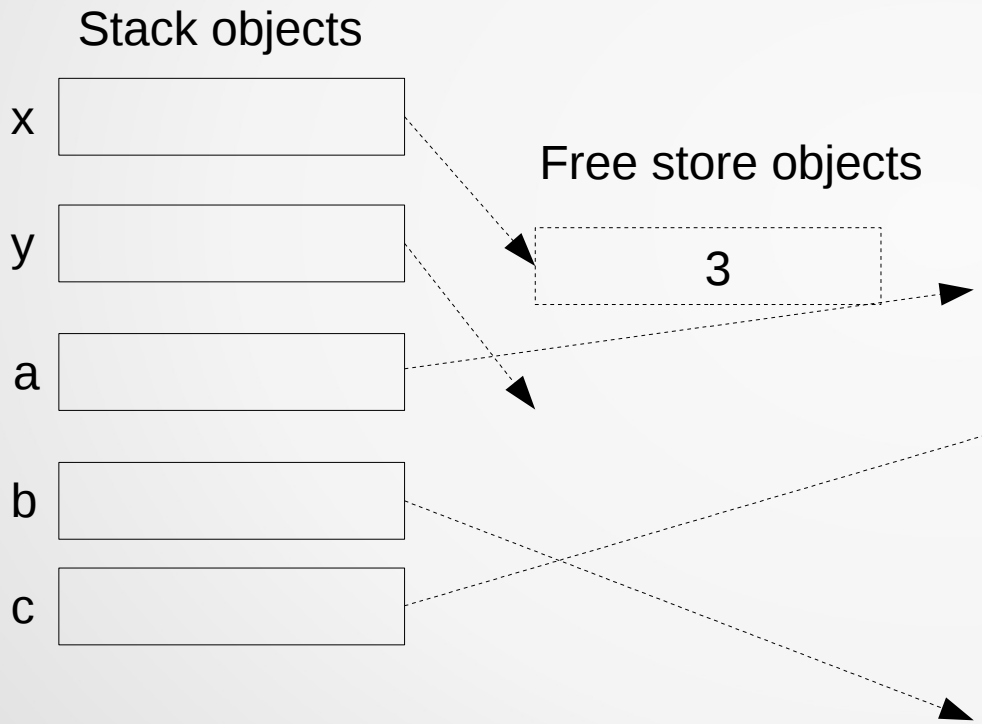
Destroys a single object or an array of objects




```
int* x = new int(3);
int* y = new int[3];
MyClass* a = new MyClass;
MyClass* b = new MyClass(1);
MyClass* c = new MyClass[3];
delete[] c;
delete a;
delete b;
delete[] y;
```

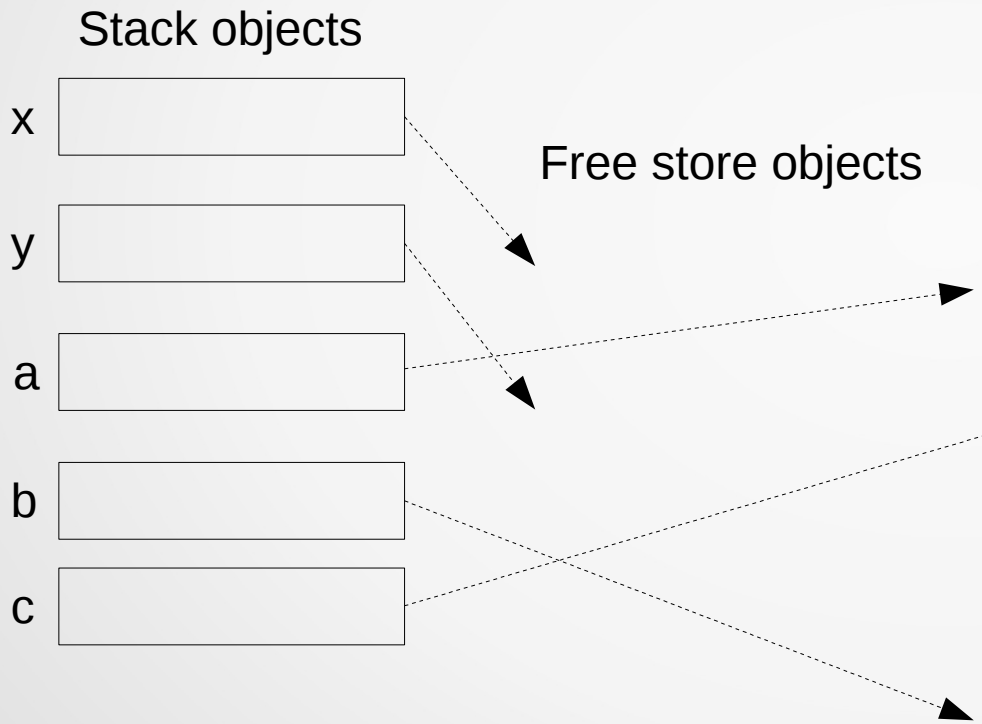
# The `delete` operator

Destroys a single object or an array of objects




```
int* x = new int(3);
int* y = new int[3];
MyClass* a = new MyClass;
MyClass* b = new MyClass(1);
MyClass* c = new MyClass[3];
delete[] c;
delete a;
delete b;
delete[] y;
delete x;
```

# The `delete` operator




```
int* x = new int(3);
int* y = new int[3];
MyClass* a = new MyClass;
MyClass* b = new MyClass(1);
MyClass* c = new MyClass[3];
delete[] c;
delete a;
delete b;
delete[] y;
delete x;
```

The delete operator does not change the pointer. All of these pointers are now dangling.

# Using new and delete

- An object allocated with new *must* be deleted exactly once
- If allocated with new[] then one *must* delete with delete[]
- Undefined behavior:
  - Deleting an object twice
  - Deleting a null pointer
  - Dereferencing a pointer to a deleted object
  - Dereferencing a null pointer
  - Deleting with delete[] when the object was allocated with new
  - Deleting with delete when the object was allocated with new[]
- Problematic behavior:
  - Not deleting an object which is not pointed to
  - This is called a *memory leak*

# Memory Management Strategies

# Strategies

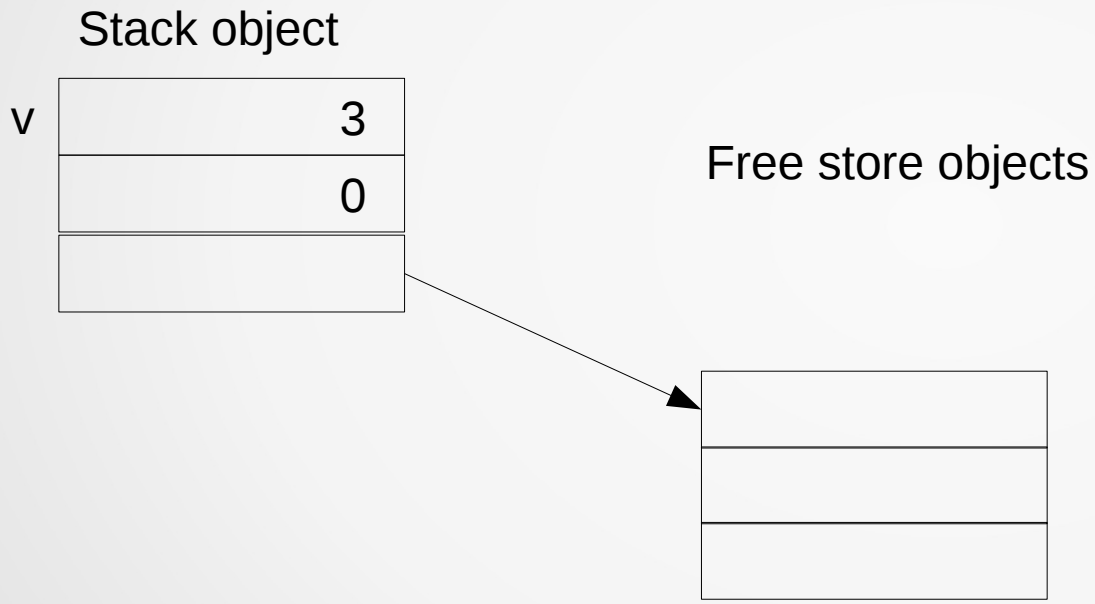
- Managing the free store requires a memory management *strategy*.
- Each class that will be using the free store must have a strategy.
- A group of related classes may have a single strategy to handle all of them.
- Some strategies
  - Container strategy
  - Tree strategy
  - Reference counting
  - Garbage collection

# Container Strategy

- Requirements for this strategy:
  - The container owns its objects
  - The container allocates and deallocates its objects
  - When the container is deleted all of its objects are deleted.
- The vector class strategy
  - An array of unused slots is initially allocated
  - The vector keeps track of how many slots are in use
  - Each call to `push_back` uses one of the allocated slots
  - When all slots are in use, the next `push_back` allocates a new, larger array, copies to the new array and deallocates the old array
  - When a vector is deleted, the array is deallocated

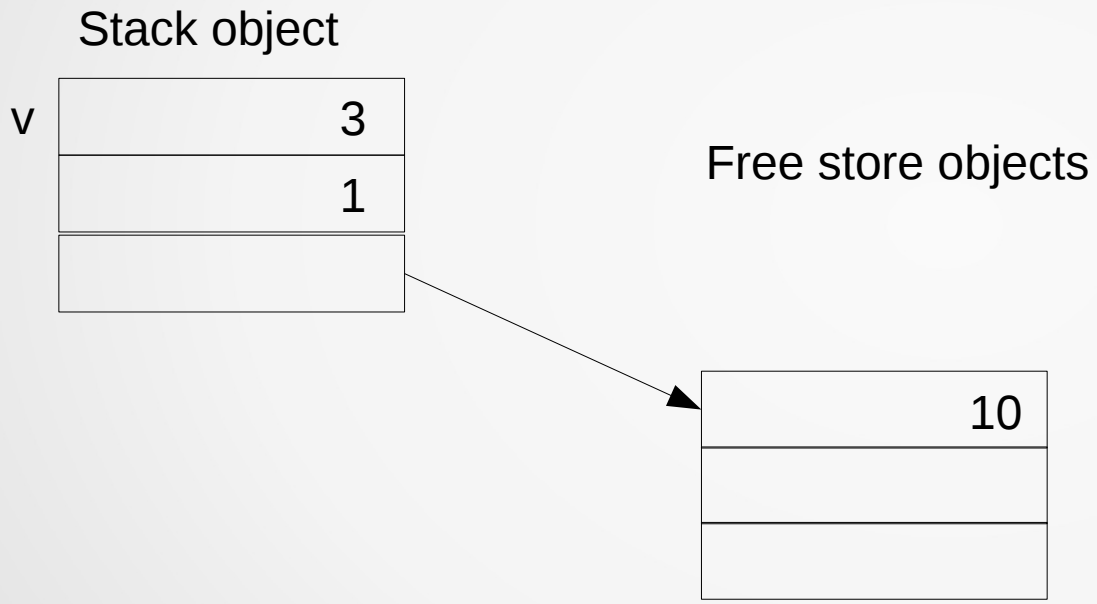
# The vector structure

```
vector<int> v(3);
```



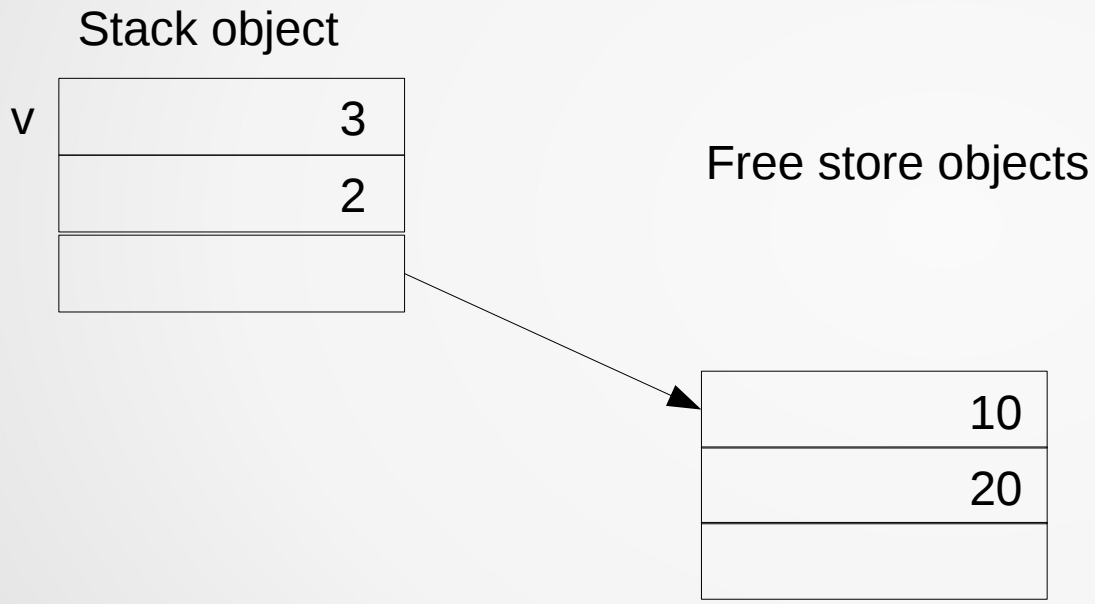
# The vector structure

```
vector<int> v(3);
v.push_back(10);
```



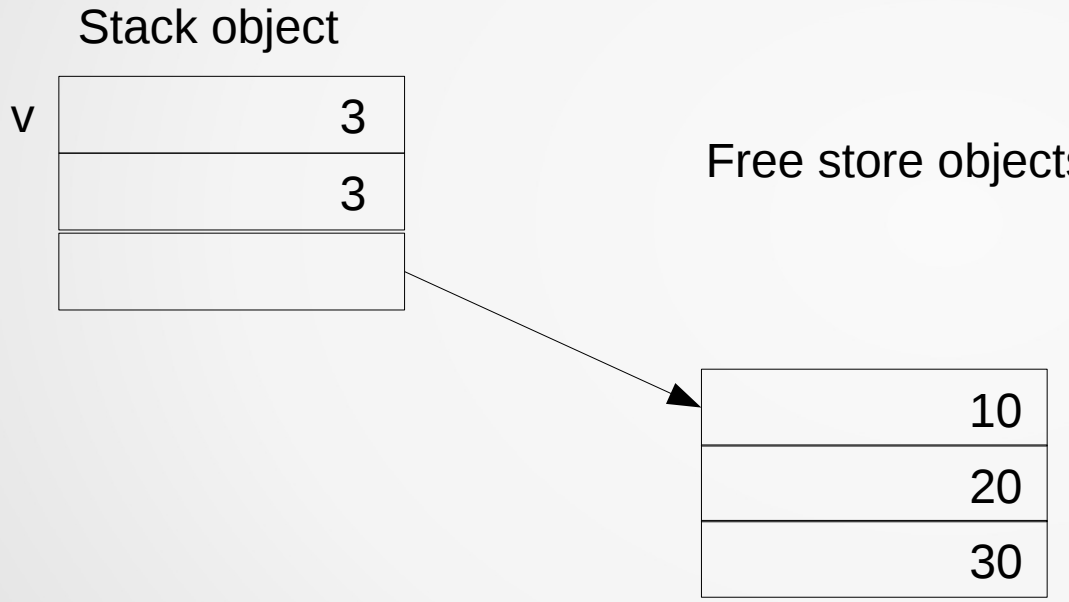
# The vector structure

```
vector<int> v(3);
v.push_back(10);
v.push_back(20);
```



# The vector structure

```
vector<int> v(3);
v.push_back(10);
v.push_back(20);
v.push_back(30);
```



# The vector structure

```
vector<int> v(3);
v.push_back(10);
v.push_back(20);
v.push_back(30);
v.push_back(40);
```

# Tree Strategy

- The pointers form a tree (or forest) structure
- In other words, one has a container of containers of containers ... to any number of levels
- Sufficient strategy as long as each object has a unique owner
- Memory is deallocated in the destructors

# Reference Counting Strategy

- The pointers form a directed acyclic graph (DAG) structure
- Objects may have multiple owners
- An object is deleted when it no longer has any owners
- Track of how many references (owners) an object has
  - This is the *reference count*
  - When the reference count reaches 0, the object is deleted.
- This is a sufficient strategy as long as there are no cycles
- The `shared_ptr` class implements reference counting

# Garbage Collection Strategy

- The pointers form an arbitrary graph structure
- Objects may have multiple owners
- An object is deleted when there is no way to access it using a pointer path starting from one of the root pointers.
  - Usually one defines the root pointers to be the ones that are in the stack somewhere
  - One can also have an explicit function that registers/unregisters a pointer as a root pointer
- The strategy is to follow all pointers recursively and mark the objects that can be accessed. Then delete all the rest.
  - For example, one can use depth first search
  - There are many garbage collection algorithms

# Low level operations

# Low-level memory allocation

- The C memory allocator is `malloc`
- Never mix `new` and `malloc`
  - The malloc function does not invoke the constructor
  - This function is also not type safe
- The C memory copy function is `memcpy`
- Never use `memcpy`
  - The `memcpy` function does not invoke a copy constructor
  - This function is not type safe

# Even lower-level operations

- One can insert an assembly language instruction with the `asm` statement.
  - Violates type safety
  - Not portable
  - Very dangerous
- Never use an `asm` statement unless one is writing highly specialized code for a specific machine.

# Assignment #6

# Requirements

- Develop four classes:
  - `Task` A task to be performed
  - `TaskFactory` A factory for randomly generates tasks
  - `SharedMap` A shared multimap for mapping integer codes to tasks
  - `Action` The action to be performed by a thread
- The main program constructs a collection of threads, each running one of the actions as well as various tasks to be performed by the threads. It then waits for the threads to finish and finally prints the current shared mapping.
- The namespace for this assignment is `asst06`. You will lose points if your program has a different namespace than this.

# Task Methods

- Constructor specifying the code and the quit flag
  - The quit flag is false by default
- Getters for the code and quit flag
- `toString()` method that shows the task properties, for example:
  - If the code is 3 and the quit flag is false show this:

    ```
    Continue(3)
    ```

  - If the code is 2 and the quit flag is true show this:

    ```
    Quit(2)
    ```

# Task Factory Methods

- Constructor specifying the number of codes and the seed for random number generation
- `getNextTask` method with a boolean parameter
  - If the boolean parameter is true then generate a random quit task
  - If the boolean parameter is false then generate a random continue task
  - The default value of the parameter is false
- The codes are integers starting with 0
  - Generate the codes uniformly

# Shared Map Methods Part 1

- Constructor has no parameters
- `map` method has a task parameter
  - The code of the task is mapped to the task by the multimap
- `unmap` method has a code parameter
  - One of the tasks that have the specified code is removed from the multimap and returned
  - If there are no tasks with the specified code, then wait until there is one

# Shared Map Methods Part 2

- `showMap` method returns a string with the current mappings
  - Here is an example:

```
2 -> Continue(2)
4 -> Continue(4)
1 -> Continue(1)
1 -> Continue(1)
1 -> Continue(1)
```

# Action Methods

- This is a functor.
- Constructor specifies the code, shared map and task factory
- The function performs the following in an infinite loop:
  - Get a task from the shared map
  - Perform the task
    - If the task is a quit task then end the loop
  - Add a randomly generated task to the shared map

# Main program

- Read two numbers from the standard input
  - The number of codes
  - The seed for the random number generator
- Construct the shared map and task factory
- Add initial tasks to the shared map, one per code
- Construct threads, one per code
- Add quit tasks to the shared map, one per code
- Wait for the threads to finish
- Print the shared map

# Next Class

- Introduction to graphics with SDL2