CME 211: Lecture 25

Topic: C++ memory management

Python memory management

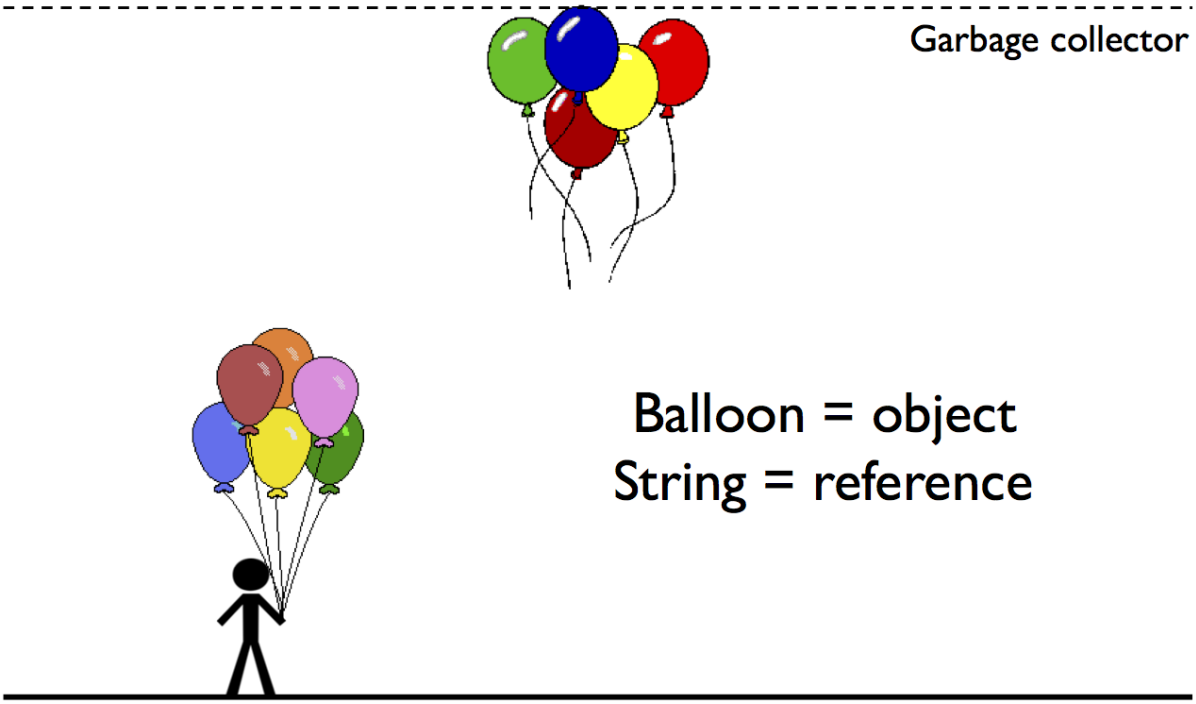

Figure 1: fig

C/C++ memory model

- All data in your application is stored in the same physical memory
- The memory used by each application is logically divided into the *stack* and the *heap*

Stack

- Fixed memory allocation provided to your application
- It is the operating system that specifies the size of the stack
- Stack memory is automatically managed for you by the compiler / processor
- Limited to local variables of fixed size

```
int main()
{
   int a = 2;
   int b = 4;
   return 0;
}
```

stack1.cpp

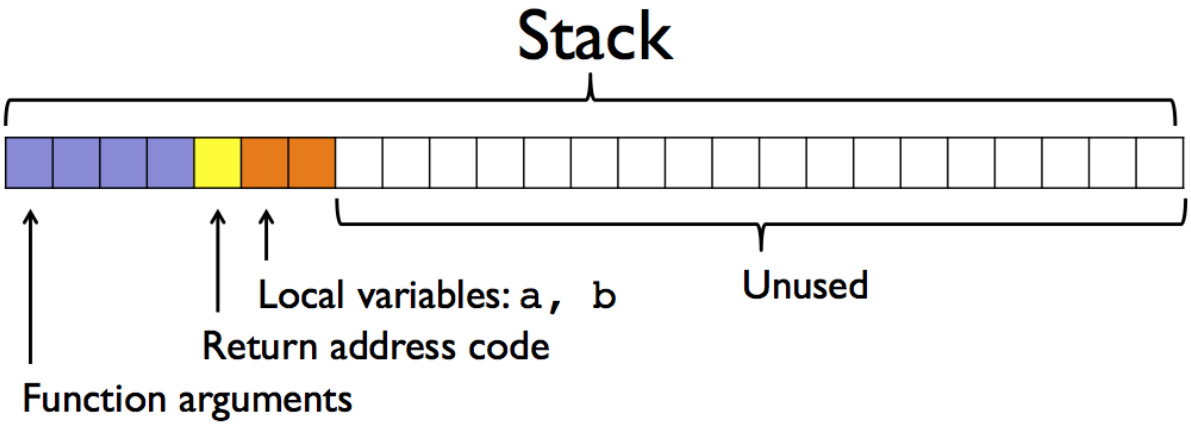

Figure 2: fig

Stack example

Function call

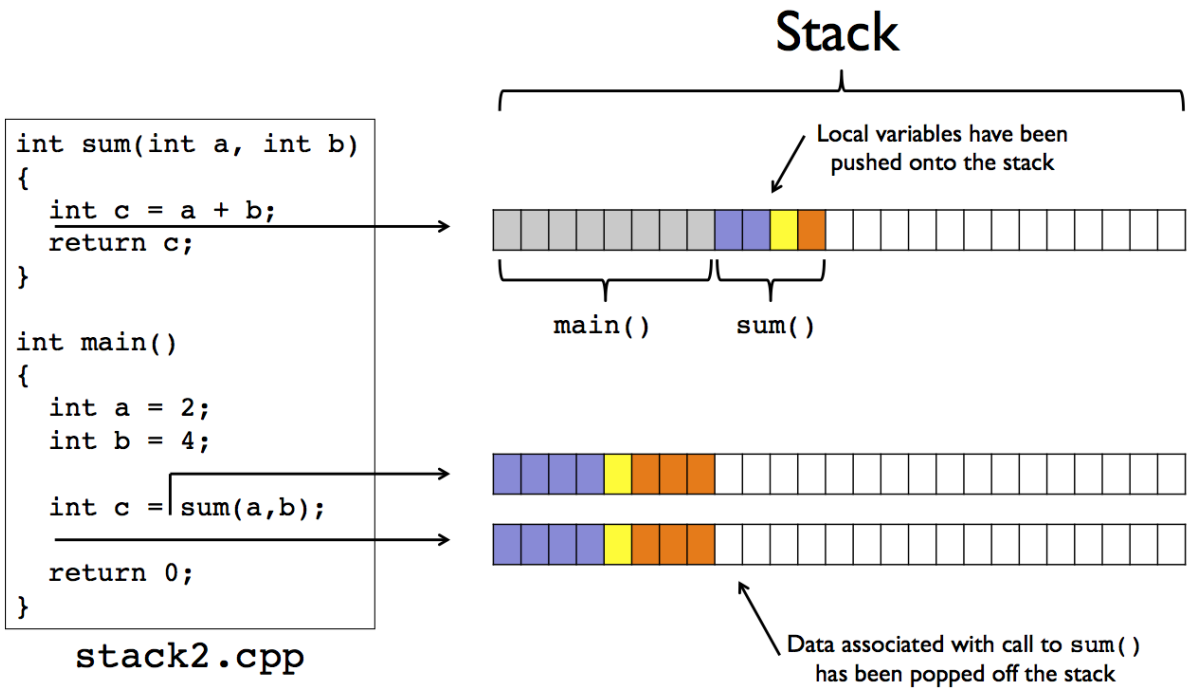

Figure 3: fig

Static arrays

Static array example

src/stack4.cpp:

```
#include <iostream>

int main() {
  int a[2048][2048];
  a[0][0] = 42;
  std::cout << "a[0][0] = " << a[0][0] << std::endl;
  return 0;
}
```

Output:

```
$ g++ -Wall -Wextra -Wconversion src/stack4.cpp -o src/stack4
$ ./src/stack4
Segmentation fault (core dumped)
```

```
int main()
{
  int a = 2;
  int b = 4;
  int c[4];
  int d = 7;

  return 0;
}
```

stack3.cpp

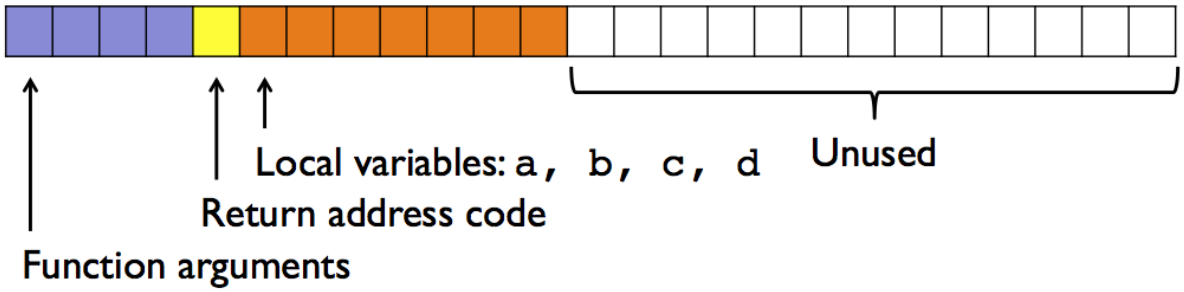

Figure 4: fig

Static size limit

Output:

```
$ ulimit -a
core file size          (blocks, -c) 0
data seg size           (kbytes, -d) unlimited
file size               (blocks, -f) unlimited
max locked memory       (kbytes, -l) unlimited
max memory size         (kbytes, -m) unlimited
open files                      (-n) 256
pipe size            (512 bytes, -p) 1
stack size              (kbytes, -s) 8515
cpu time               (seconds, -t) unlimited
max user processes              (-u) 709
virtual memory          (kbytes, -v) unlimited
```

Modifying the stack size limit

```
$ ulimit -s unlimited
-bash: ulimit: stack size: cannot modify limit: Operation not permitted
$ ulimit -s 16384
-bash: ulimit: stack size: cannot modify limit: Operation not permitted
$ ulimit -s 4096
$
```

- On `corn` we cannot make the stack size larger, but we can make it smaller!

Stack size

src/stack5.cpp:

```
#include <vector>

#include "boost/multi_array.hpp"

int main() {
  std::vector<unsigned int> a;
  for(unsigned int i = 0; i < 8192*8192; i++)
    a.push_back(i);

  boost::multi_array<unsigned int, 2> b(boost::extents[8192][8192]);

  return 0;
}
```

Stack size

Output:

```
$ ulimit -a
core file size          (blocks, -c) 0
data seg size           (kbytes, -d) unlimited
file size               (blocks, -f) unlimited
```

```
max locked memory       (kbytes, -l) unlimited
max memory size         (kbytes, -m) unlimited
open files                      (-n) 256
pipe size            (512 bytes, -p) 1
stack size              (kbytes, -s) 8515
cpu time               (seconds, -t) unlimited
max user processes              (-u) 709
virtual memory          (kbytes, -v) unlimited
$ g++ -std=c++11 -Wall -Wextra -Wconversion src/stack5.cpp -o src/stack5
$ ./src/stack5
```

Heap

- Can contain data of arbitrary size (subject to available computer resources like total memory)
- Accessible by any function (global scope)
- Has the life of the program
- *Managed by programmer*

Using heap memory

- You need to allocate heap memory
- The location of the allocated memory is stored in a pointer, a special variable which stores a memory address
- When you are done using the memory you need to free the memory

Pointers

Declaration of a pointer is denoted by a `*` in front of the variable name (after the type)

- `int a;`: variable `a` will contain an integer
- `int *b;`: variable `b` will contain a memory address where an integer is stored
- `int* b;`: equivalent to `int *b;`. This is my prefered style. I would read it as: "`b` is a variable containing a pointer to an int". Hint: read C and C++ type declarations backwards.

Pointers contain addresses

Many roles of the `*`

- We've already seen that the asterisk is used to denote the declaration of a pointer
- The asterisk is also used to access the data at the memory address stored in a pointer
- This operation is typically call *dereferencing*

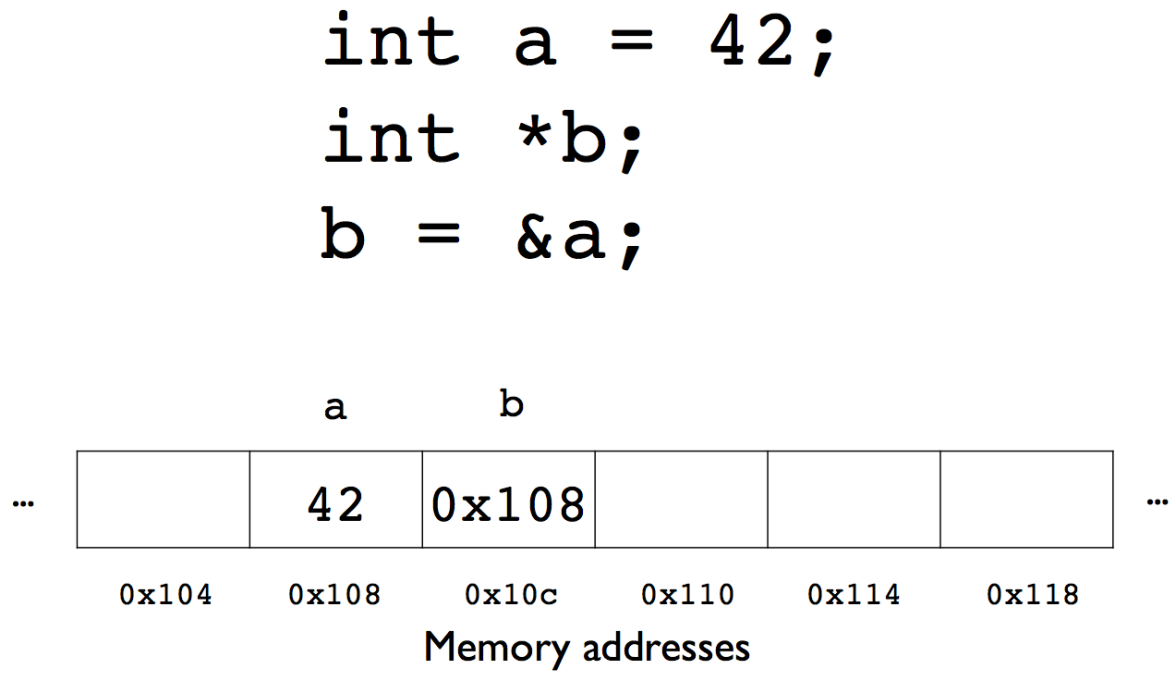

Figure 5: fig

Dereferencing a pointer

src/pointer1.cpp:

```
#include <iostream>

int main() {
  int a = 42;
  int* b; // b is a pointer to an int

  std::cout << " a = " << a << std::endl;
  std::cout << "&a = " << &a << std::endl;

  b = &a; // here & is the "address of" operator

  // show the value of the pointer
  std::cout << " b = " << b << std::endl;

  // dereference the pointer
  std::cout << "*b = " << *b << std::endl;

  return 0;
}
```

Output:

```
$ g++ -std=c++11 -Wall -Wextra -Wconversion src/pointer1.cpp -o src/pointer1
$ ./src/pointer1
 a = 42
```

```
&a = 0x7fff5a43fad8
 b = 0x7fff5a43fad8
*b = 42
```

Store a value at a memory address

The asterisk in front of a pointer has a different meaning when it appears on the left of the assignment operator (=)

```
int a = 42;
int *b;
b = &a;
// store the value 7 at the memory address in b
*b = 7;
```

Storing a value

src/pointer2.cpp:

```
#include <iostream>

int main() {
  int a = 42;
  int *b;
  b = &a;

  std::cout << " a = " << a << std::endl;
  std::cout << "&a = " << &a << std::endl;
  std::cout << " b = " << b << std::endl;
  std::cout << "*b = " << *b << std::endl;

  // Store the value 7 at the
  // memory address stored in b
  *b = 7;

  std::cout << " a = " << a << std::endl;
  std::cout << "&a = " << &a << std::endl;
  std::cout << " b = " << b << std::endl;
  std::cout << "*b = " << *b << std::endl;

  return 0;
}
```

Output:

```
$ g++ -std=c++11 -Wall -Wextra -Wconversion src/pointer2.cpp -o src/pointer2
$ ./src/pointer2
 a = 42
&a = 0x7fff5ebc9a98
 b = 0x7fff5ebc9a98
*b = 42
 a = 7
&a = 0x7fff5ebc9a98
 b = 0x7fff5ebc9a98
*b = 7
```

Increment

src/increment.cpp:

```cpp
#include <iostream>

void increment(int *a) {
  // Value at the memory
  // address is incremented
  (*a)++;
}

int main() {
  int a = 2;
  std::cout << "a = " << a << std::endl;

  // increment() receives copy of memory address for a
  increment(&a);
  std::cout << "a = " << a << std::endl;

  return 0;
}
```

Output:

```
$ g++ -std=c++11 -Wall -Wextra -Wconversion src/increment.cpp -o src/increment
$ ./src/increment
a = 2
a = 3
```

Increment

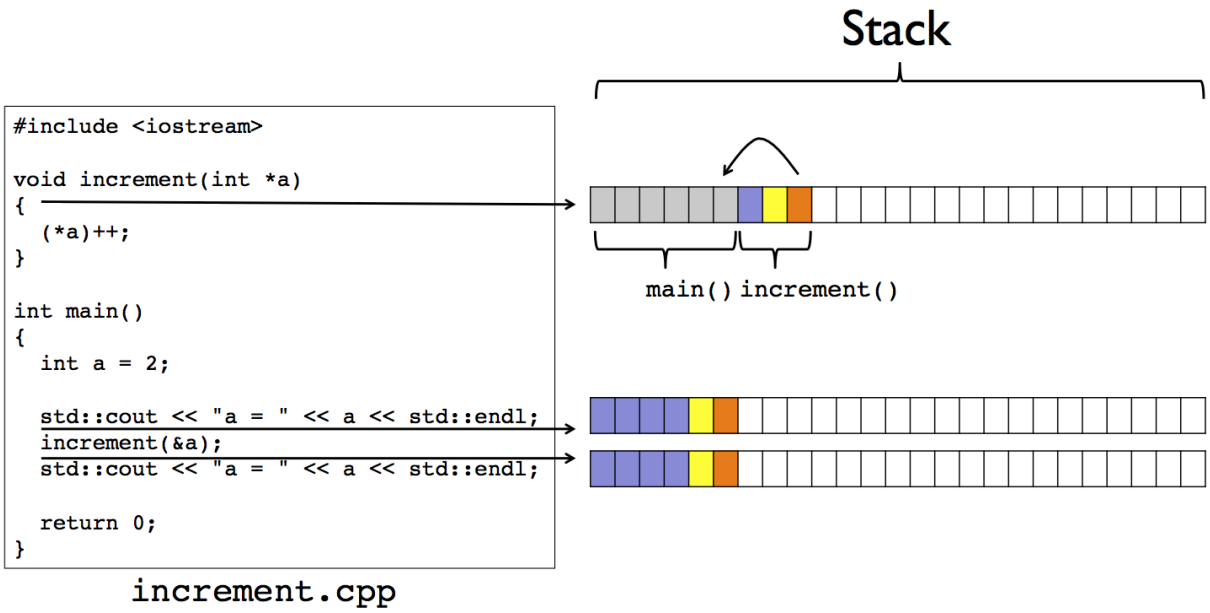

Figure 6: fig

Returning pointers

src/func.cpp:

```
#include <iostream>

int* func(void) {
  int b = 2;
  return &b;
}

int main() {
  int *a = func();

  std::cout << " a = " << a << std::endl;
  std::cout << "*a = " << *a << std::endl;

  return 0;
}
```

Output:

```
$ g++ -std=c++11 -Wall -Wextra -Wconversion src/func.cpp -o src/func
src/func.cpp:5:11: warning: address of stack memory associated with local variable 'b' returned [-Wretu
  return &b;
         ^
1 warning generated.
$ ./src/func
 a = 0x7fff5bcf4acc
*a = 32767
```

Returning pointers

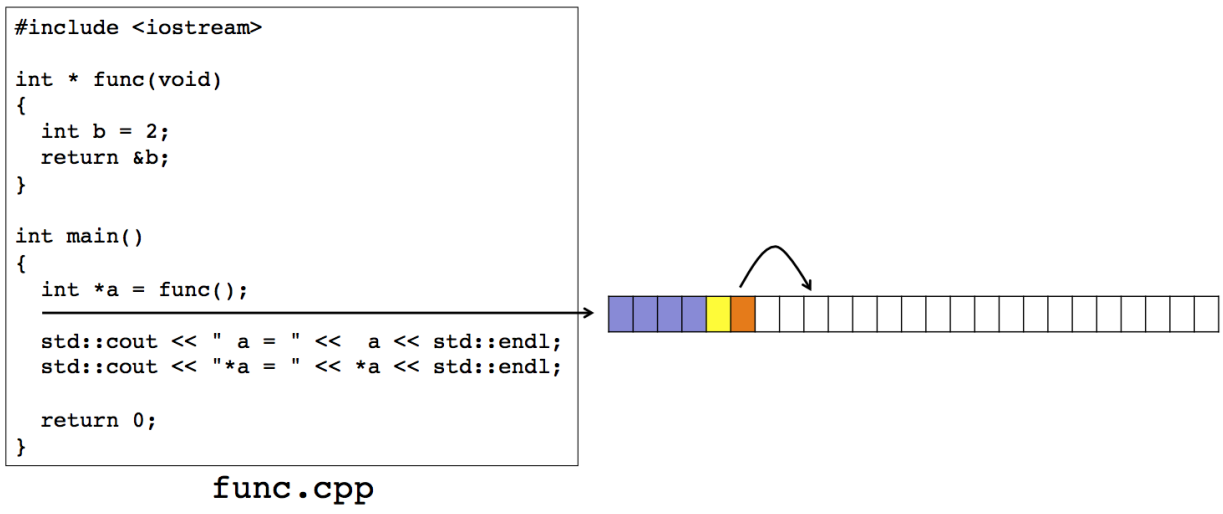

Figure 7: fig

Common mistake: pointer declaration

(There are many!)

`double *a, b;`

- `a` is a pointer to a double
- `b` is a double

`double *a, *b;`

- `a` is a pointer to a double
- `b` is a pointer to a double

`double* a, b;`

- `a` is a pointer to a double
- `b` is a **double**

Many uses of *

src/pointer3.cpp:

```
#include <iostream>

int main() {
  int a = 4;
  int *b = &a;

  // * used for dereferencing, multiplication, and storage
  *b = *b**b;

  std::cout << "a = " << a << std::endl;

  return 0;
}
```

Output:

```
$ g++ -std=c++11 -Wall -Wextra -Wconversion src/pointer3.cpp -o src/pointer3
$ ./src/pointer3
a = 16
```

Common mistake: uninitialized pointer

src/pointer4.cpp:

```
#include <iostream>

int main() {
  int *a;
  std::cout << "*a = " << *a << std::endl;
  return 0;
}
```

Output:

```
$ g++ -std=c++11 -Wall -Wextra -Wconversion src/pointer4.cpp -o src/pointer4
src/pointer4.cpp:5:28: warning: variable 'a' is uninitialized when used here [-Wuninitialized]
  std::cout << "*a = " << *a << std::endl;
                           ^
src/pointer4.cpp:4:9: note: initialize the variable 'a' to silence this warning
  int *a;
       ^
        = nullptr
1 warning generated.
$ ./src/pointer4
/bin/sh: line 1: 61024 Segmentation fault: 11  ./src/pointer4
```

Suggestion

src/pointer5.cpp:

```
#include <iostream>

int main() {
  int *a = nullptr;
  std::cout << "*a = " << *a << std::endl;
  return 0;
}
```

Output:

```
$ g++ -std=c++11 -Wall -Wextra -Wconversion src/pointer5.cpp -o src/pointer5
$ ./src/pointer5
/bin/sh: line 1: 61031 Segmentation fault: 11  ./src/pointer5
```

new

- the `new` keyword *allocates* dynamic memory in the *heap*
- Works by setting aside a specified amount of *contiguous memory* and returning the *starting address*
- No guarantees about the state of initialization (i.e. the memory will have "random" data in it)

Memory allocation

src/new1.cpp:

```
#include <iostream>
#include <string>

int main(int argc, char *argv[]) {
  if (argc < 2) return 1;
  unsigned int n = std::stoi(argv[1]);

  // Allocate storage for n double values and
  // store the starting address in a
  double *a = new double[n];
  std::cout << "a = " << a << std::endl;

  for (unsigned int i = 0; i < n; i++)
```

```
    a[i] = i+3;

  for (unsigned int i = 0; i < n; i++)
    std::cout << "a[" << i << "] = " << a[i] << std::endl;

  // Free the memory
  delete[] a;
  std::cout << "a = " << a << std::endl;

  return 0;
}
```

Output:

```
$ g++ -std=c++11 -Wall -Wextra -Wconversion src/new1.cpp -o src/new1
src/new1.cpp:6:20: warning: implicit conversion changes signedness: 'int' to 'unsigned int' [-Wsign-conv
  unsigned int n = std::stoi(argv[1]);
               ~   ^~~~~~~~~~~~~~~~~~
1 warning generated.
$ ./src/new1 2
a = 0x7fb562e00000
a[0] = 3
a[1] = 4
a = 0x7fb562e00000
$ ./src/new1 4
a = 0x7fc033c031a0
a[0] = 3
a[1] = 4
a[2] = 5
a[3] = 6
a = 0x7fc033c031a0
```

Memory allocation sequence

Step 1:

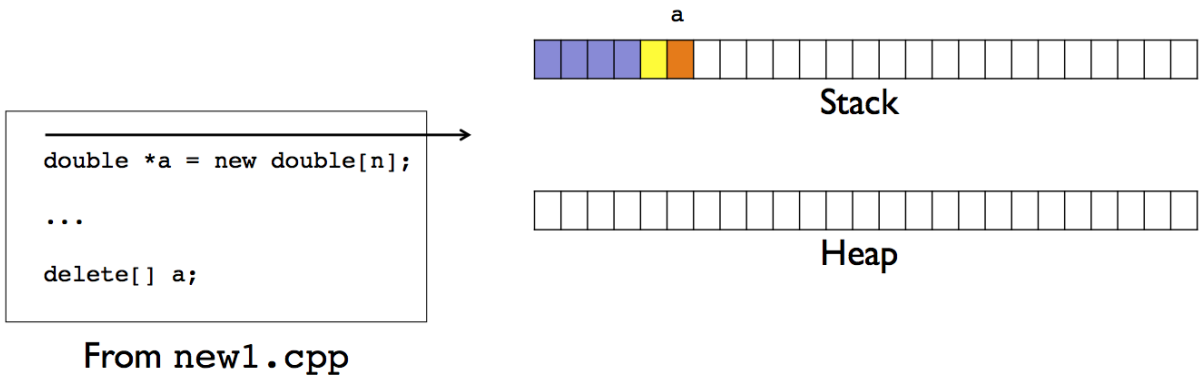

Figure 8: fig

Step 2:

Step 3:

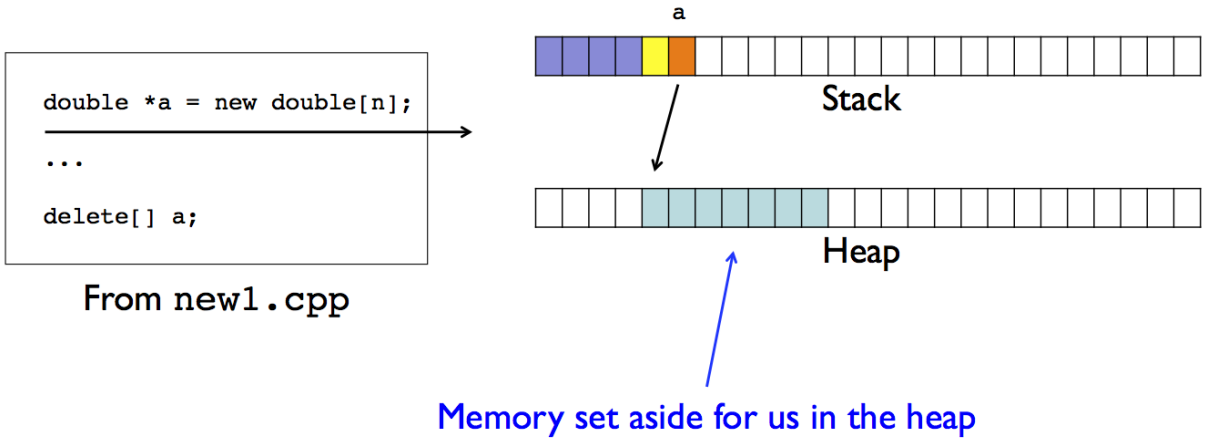

Figure 9: fig

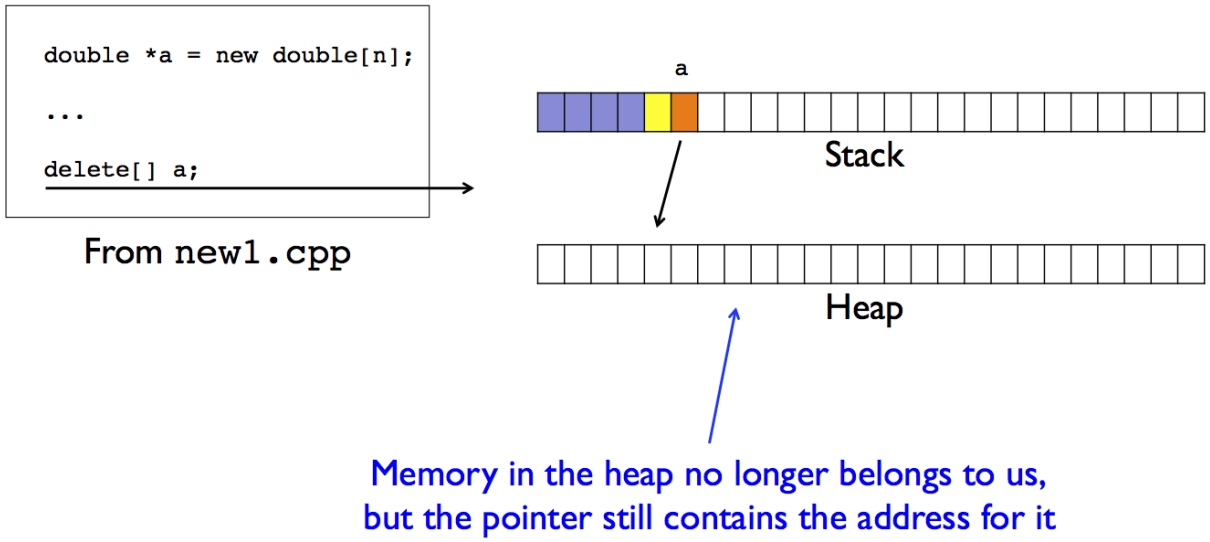

Figure 10: fig

Out of bounds access

src/new2.cpp:

```
#include <iostream>
#include <string>

int main(int argc, char *argv[]) {
  if (argc < 2) return 1;
  unsigned int n = std::stoi(argv[1]);

  double *a = new double[n];
  std::cout << "a = " << a << std::endl;

  delete[] a;
  std::cout << "a = " << a << std::endl;

  for (unsigned int i = 0; i < n; i++)
    a[i] = i+3;

  for (unsigned int i = 0; i < n; i++)
    std::cout << "a[" << i << "] = " << a[i] << std::endl;

  return 0;
}
```

Output:

```
$ g++ -std=c++11 -Wall -Wextra -Wconversion src/new2.cpp -o src/new2
$ ./src/new2 2
a = 0xe98040
a = 0xe98040
a[0] = 3
a[1] = 4
$ ./src/new2 1048576
a = 0x7f8bf1c0b010
a = 0x7f8bf1c0b010
Segmentation fault (core dumped)
```

Use valgrind

- compile with `-g` flag
- run with `valgrind`

Output:

```
$ g++ -g -std=c++11 -Wall -Wextra -Wconversion src/new2.cpp -o src/new2
src/new2.cpp:6:20: warning: implicit conversion changes signedness: 'int' to 'unsigned int' [-Wsign-conv
  unsigned int n = std::stoi(argv[1]);
               ~   ^~~~~~~~~~~~~~~~~
1 warning generated.
$ valgrind ./src/new2 4
==61046== Memcheck, a memory error detector
==61046== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
==61046== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
```

```
==61046== Command: ./src/new2 4
==61046==
a = 0x100a72ea0
a = 0x100a72ea0
==61046== Invalid write of size 8
==61046==    at 0x100001099: main (new2.cpp:15)
==61046==  Address 0x100a72ea0 is 0 bytes inside a block of size 32 free'd
==61046==    at 0x10000B2F7: free (in /usr/local/Cellar/valgrind/3.11.0/lib/valgrind/vgpreload_memcheck
==61046==    by 0x10000101E: main (new2.cpp:11)
==61046==  Block was alloc'd at
==61046==    at 0x10000AEBB: malloc (in /usr/local/Cellar/valgrind/3.11.0/lib/valgrind/vgpreload_memche
==61046==    by 0x10004E7DD: operator new(unsigned long) (in /usr/lib/libc++.1.dylib)
==61046==    by 0x100000FAB: main (new2.cpp:8)
==61046==
==61046== Invalid read of size 8
==61046==    at 0x10000112F: main (new2.cpp:18)
==61046==  Address 0x100a72ea0 is 0 bytes inside a block of size 32 free'd
==61046==    at 0x10000B2F7: free (in /usr/local/Cellar/valgrind/3.11.0/lib/valgrind/vgpreload_memcheck
==61046==    by 0x10000101E: main (new2.cpp:11)
==61046==  Block was alloc'd at
==61046==    at 0x10000AEBB: malloc (in /usr/local/Cellar/valgrind/3.11.0/lib/valgrind/vgpreload_memche
==61046==    by 0x10004E7DD: operator new(unsigned long) (in /usr/lib/libc++.1.dylib)
==61046==    by 0x100000FAB: main (new2.cpp:8)
==61046==
a[0] = 3
a[1] = 4
a[2] = 5
a[3] = 6
==61046==
==61046== HEAP SUMMARY:
==61046==     in use at exit: 38,600 bytes in 193 blocks
==61046==   total heap usage: 258 allocs, 65 frees, 44,344 bytes allocated
==61046==
==61046== LEAK SUMMARY:
==61046==    definitely lost: 80 bytes in 1 blocks
==61046==    indirectly lost: 68 bytes in 2 blocks
==61046==      possibly lost: 0 bytes in 0 blocks
==61046==    still reachable: 16,384 bytes in 1 blocks
==61046==         suppressed: 22,068 bytes in 189 blocks
==61046== Rerun with --leak-check=full to see details of leaked memory
==61046==
==61046== For counts of detected and suppressed errors, rerun with: -v
==61046== ERROR SUMMARY: 8 errors from 2 contexts (suppressed: 0 from 0)
```

Suggestion

src/new3.cpp:

```cpp
#include <iostream>
#include <string>

int main(int argc, char *argv[]) {
  if (argc < 2) return 1;
  unsigned int n = std::stoi(argv[1]);
```

```
  double *a = new double[n];

  delete[] a;
  a = nullptr;

  for (unsigned int i = 0; i < n; i++)
    a[i] = i+3;

  for (unsigned int i = 0; i < n; i++)
    std::cout << "a[" << i << "] = " << a[i] << std::endl;

  return 0;
}
```

```
$ g++ -std=c++11 -Wall -Wextra -Wconversion src/new3.cpp -o src/new3
$ ./src/new3 2
Segmentation fault (core dumped)
```

Memory allocation in a function

src/new4.cpp:

```
#include <iostream>
#include <string>

double * AllocateArray(unsigned int n) {
  //Memory allocated, accessed, and pointer to it returned
  double *a = new double[n];
  for (unsigned int i = 0; i < n; i++) a[i] = 0.;
  return a;
}

int main(int argc, char *argv[]) {
  if (argc < 2) return 1;
  unsigned int n = std::stoi(argv[1]);

  // Returned memory address stored in stack variable
  double *a = AllocateArray(n);

  // Memory is now used by main()
  for (unsigned int i = 0; i < n; i++)
    a[i] = i+3;
  for (unsigned int i = 0; i < n; i++)
    std::cout << "a[" << i << "] = " << a[i] << std::endl;

  delete[] a; // Memory is freed
  a = NULL;

  return 0;
}
```

Output:

```
$ g++ -std=c++11 -g -Wall -Wextra -Wconversion src/new4.cpp -o src/new4
```

```
src/new4.cpp:13:20: warning: implicit conversion changes signedness: 'int' to 'unsigned int' [-Wsign-co
  unsigned int n = std::stoi(argv[1]);
               ~   ^~~~~~~~~~~~~~~~~
1 warning generated.
$ valgrind ./src/new4 4
==61053== Memcheck, a memory error detector
==61053== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
==61053== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==61053== Command: ./src/new4 4
==61053==
a[0] = 3
a[1] = 4
a[2] = 5
a[3] = 6
==61053==
==61053== HEAP SUMMARY:
==61053==     in use at exit: 38,600 bytes in 193 blocks
==61053==   total heap usage: 258 allocs, 65 frees, 44,344 bytes allocated
==61053==
==61053== LEAK SUMMARY:
==61053==    definitely lost: 80 bytes in 1 blocks
==61053==    indirectly lost: 68 bytes in 2 blocks
==61053==      possibly lost: 0 bytes in 0 blocks
==61053==    still reachable: 16,384 bytes in 1 blocks
==61053==         suppressed: 22,068 bytes in 189 blocks
==61053== Rerun with --leak-check=full to see details of leaked memory
==61053==
==61053== For counts of detected and suppressed errors, rerun with: -v
==61053== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Memory leaks

src/new5.cpp:

```cpp
#include <iostream>
#include <string>

void ProcessData(double *a, unsigned int n) {
  // temporary allocation for processing a
  // Memory is allocated but never freed
  double *tmp = new double[n];
  for (unsigned int i = 0; i < n; i++) tmp[i] = 0.;

  // Process a
  a[0] = tmp[0];

  return;
}

int main(int argc, char *argv[]) {
  if (argc < 2) return 1;
  unsigned int n = std::stoi(argv[1]);

  double *a = new double[n];
```

```
  // Process a
  ProcessData(a, n);

  delete[] a;
  a = nullptr;

  return 0;
}
```

Output:

```
$ g++ -std=c++11 -g -Wall -Wextra -Wconversion src/new5.cpp -o src/new5
src/new5.cpp:18:20: warning: implicit conversion changes signedness: 'int' to 'unsigned int' [-Wsign-co
  unsigned int n = std::stoi(argv[1]);
               ~   ^~~~~~~~~~~~~~~~~
1 warning generated.
$ valgrind ./src/new5 4
==61060== Memcheck, a memory error detector
==61060== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
==61060== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==61060== Command: ./src/new5 4
==61060==
==61060==
==61060== HEAP SUMMARY:
==61060==     in use at exit: 22,100 bytes in 190 blocks
==61060==   total heap usage: 255 allocs, 65 frees, 27,844 bytes allocated
==61060==
==61060== LEAK SUMMARY:
==61060==    definitely lost: 32 bytes in 1 blocks
==61060==    indirectly lost: 0 bytes in 0 blocks
==61060==      possibly lost: 0 bytes in 0 blocks
==61060==    still reachable: 0 bytes in 0 blocks
==61060==         suppressed: 22,068 bytes in 189 blocks
==61060== Rerun with --leak-check=full to see details of leaked memory
==61060==
==61060== For counts of detected and suppressed errors, rerun with: -v
==61060== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

C++ memory management

Containers

- Object is a stack variable
- One (or more) data attributes point to heap memory

Vector implementation

src/MyVector1.hpp:

```
#pragma once

class MyVector
```

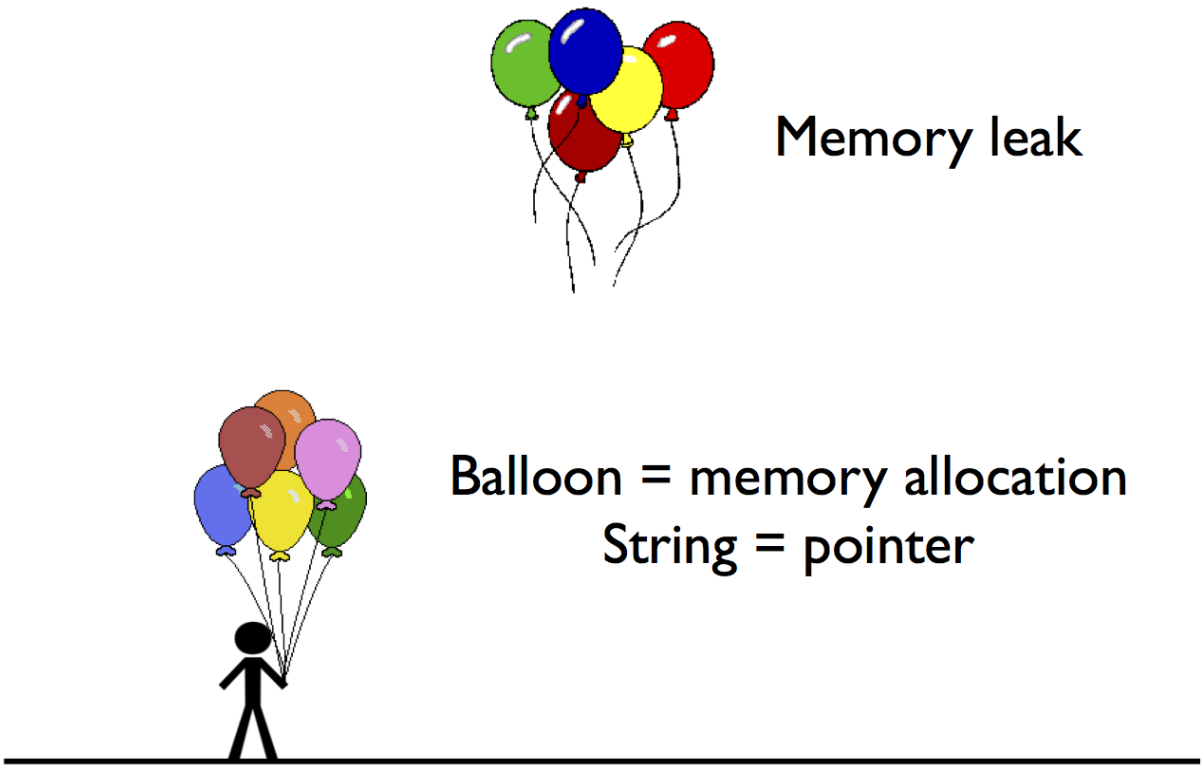

Figure 11: fig

```
{
 private:
  int *data;
  unsigned int size;
  unsigned int capacity;
 public:
  MyVector();
  void push_back(int val);
  void print(void);
};
```

src/MyVector1.cpp:

```
#include <iostream>

#include "MyVector1.hpp"

MyVector::MyVector() {
  size = 0;
  capacity = 10;
  data = new int[capacity];
}

void MyVector::push_back(int val) {
  if (size < capacity) {
    data[size] = val;
    size++;
  }
  else {
    // A real implementation would resize the capacity
    std::cerr << "Vector is full" << std::endl;
    exit(1);
  }
}

void MyVector::print() {
  using std::cout;
  using std::endl;
  cout << "[";
  bool comma = false;
  for (unsigned int i = 0; i < size; ++i) {
    if (comma) {
      cout << ", ";
    }
    else {
      comma = true;
    }
    cout << data[i];
  }
  cout << ']';
}
```

src/main1.cpp:

```
#include <iostream>
#include "MyVector1.hpp"
```

```
void func(void) {
  // Create an instance of the MyVector class
  MyVector v;
  v.push_back(7);
  v.push_back(42);
  v.print();
  std::cout << std::endl;
}

int main() {
  func();
  return 0;
}
```

Output:

```
$ g++ -g -std=c++11 -Wall -Wextra -Wconversion src/main1.cpp src/MyVector1.cpp -o src/main1
$ ./src/main1
[7, 42]
```

Memory leak

Output:

```
$ valgrind ./src/main1
==61070== Memcheck, a memory error detector
==61070== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
==61070== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==61070== Command: ./src/main1
==61070==
[7, 42]
==61070==
==61070== HEAP SUMMARY:
==61070==     in use at exit: 38,492 bytes in 191 blocks
==61070==   total heap usage: 255 allocs, 64 frees, 44,204 bytes allocated
==61070==
==61070== LEAK SUMMARY:
==61070==    definitely lost: 40 bytes in 1 blocks
==61070==    indirectly lost: 0 bytes in 0 blocks
==61070==      possibly lost: 0 bytes in 0 blocks
==61070==    still reachable: 16,384 bytes in 1 blocks
==61070==         suppressed: 22,068 bytes in 189 blocks
==61070== Rerun with --leak-check=full to see details of leaked memory
==61070==
==61070== For counts of detected and suppressed errors, rerun with: -v
==61070== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Destructor

src/MyVector2.hpp:

```
#pragma once
```

```
class MyVector
{
 private:
  int *data;
  unsigned int size;
  unsigned int capacity;
 public:
  MyVector();
  void push_back(int val);
  void print(void);
  ~MyVector();
};
```

From `src/MyVector2.cpp`:

```
MyVector::~MyVector() {
  delete[] data;
  data = nullptr;
}
```

Output:

```
$ g++ -g -std=c++11 -Wall -Wextra -Wconversion src/main2.cpp src/MyVector2.cpp -o src/main2
$ ./src/main2
[7, 42]
$ valgrind ./src/main2
==61080== Memcheck, a memory error detector
==61080== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
==61080== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==61080== Command: ./src/main2
==61080==
[7, 42]
==61080==
==61080== HEAP SUMMARY:
==61080==     in use at exit: 38,452 bytes in 190 blocks
==61080==   total heap usage: 255 allocs, 65 frees, 44,204 bytes allocated
==61080==
==61080== LEAK SUMMARY:
==61080==    definitely lost: 0 bytes in 0 blocks
==61080==    indirectly lost: 0 bytes in 0 blocks
==61080==      possibly lost: 0 bytes in 0 blocks
==61080==    still reachable: 16,384 bytes in 1 blocks
==61080==         suppressed: 22,068 bytes in 189 blocks
==61080== Rerun with --leak-check=full to see details of leaked memory
==61080==
==61080== For counts of detected and suppressed errors, rerun with: -v
==61080== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

C++ memory management

Reading

C++ Primer, Fifth Edition by Lippman et al:

- Section 2.3.2: Pointers

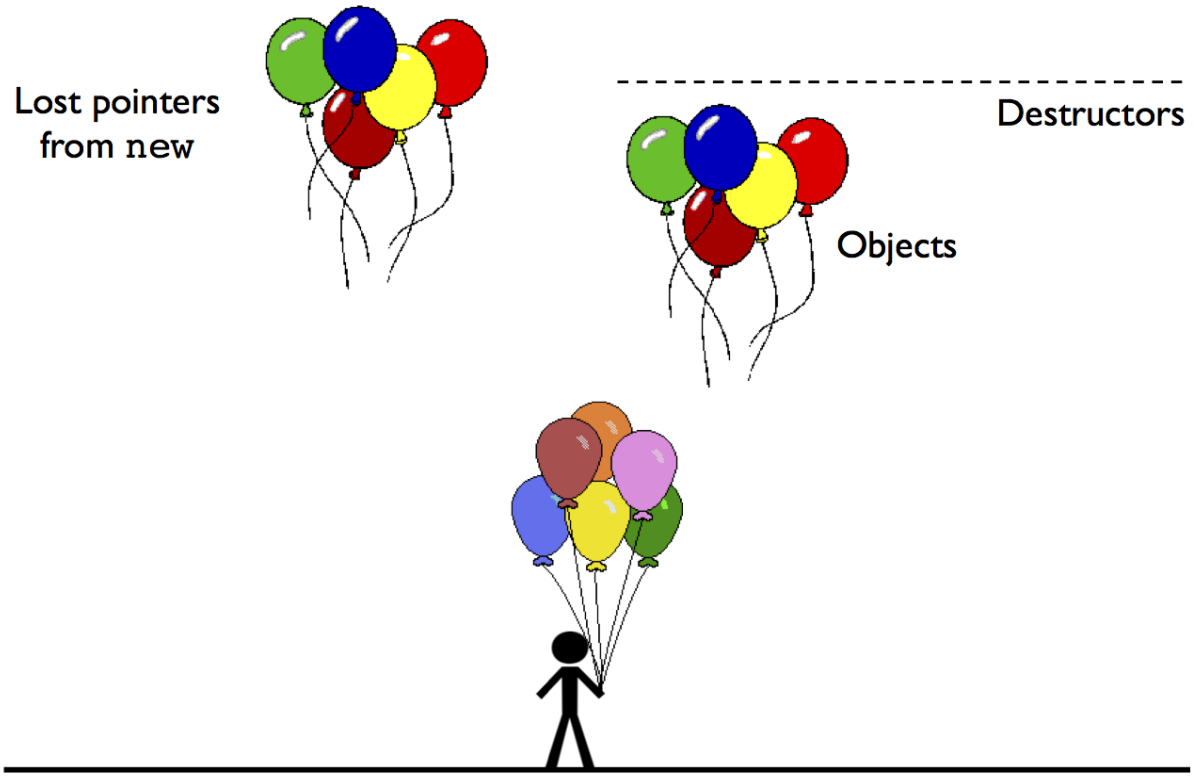

Figure 12: fig

- Section 12.2: Dynamic Arrays
- Section 7.1.5: Destruction