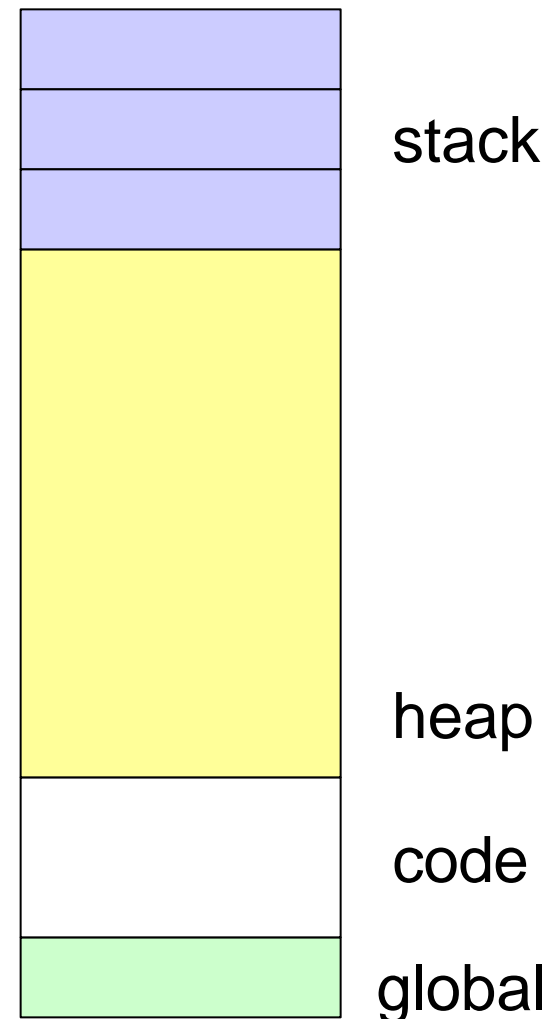


Overview

- 4 major memory segments
 - Global: variables outside stack, heap
 - Code (a.k.a. text): the compiled program
 - Heap: dynamically allocated variables
 - Stack: parameters, automatic and temporary variables
- Key differences from Java
 - Destructors of automatic variables called when stack frame where declared pops
 - No garbage collection: program must explicitly free dynamic memory
- Heap and stack use varies dynamically
- Code and global use is fixed
- Code segment is “read-only”



Illustrating C++ Memory

```
int g_default_value = 1;

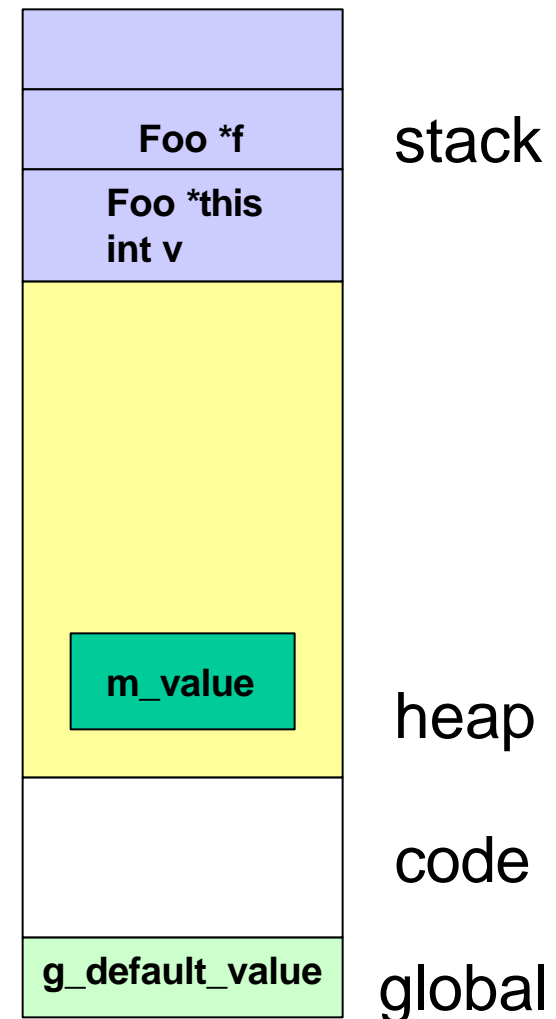
int main (int argc, char **argv) {
    Foo *f = new Foo;

    f->setValue(g_default_value);

    delete f;

    return 0;
}

void Foo::setValue(int v) {
    m_value = v;
}
```



Illustrating C++ Memory

```
int g_default_value = 1;

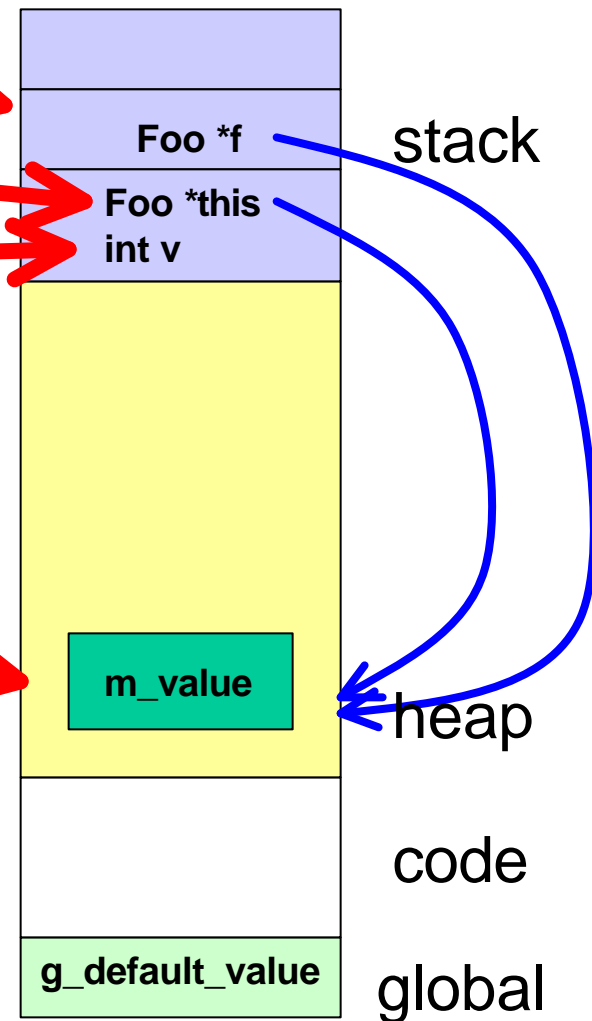
int main (int argc, char **argv) {
    Foo *f = new Foo;

    f->setValue(g_default_value);

    delete f;

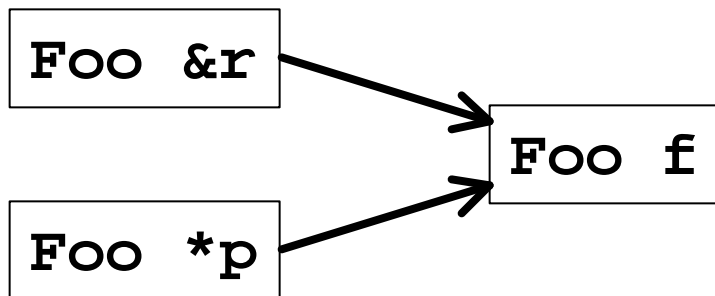
    return 0;
}

void Foo::setValue(int v) {
    m_value = v;
}
```



A Basic Issue: Aliasing

```
int main (int argc, char **argv) {  
    Foo f;  
    Foo *p = &f;  
    Foo &r = f;  
  
    delete p;  
  
    return 0;  
}
```



- Multiple aliases for same object
 - f is a simple alias, the object itself
 - g is a variable holding a pointer
 - h is a variable holding a reference
- What happens when we call delete on p?
 - Destroy a stack variable (may get a bus error there if we're lucky)
 - If not, we may crash in destructor of f at function exit
 - Or worse, a local stack corruption that may lead to problems later
- Problem: object destroyed but another alias to it was then used

Memory Initialization Errors

```
int main(int argc, char **argv) {  
    Foo *f;  
  
    cout << f->value() << endl;  
  
    for (int i; i < argc; ++i)  
        cout << argv [i] << endl;  
  
    return 0;  
}
```

- Heap and stack memory is not initialized for us in C++
 - Get whatever was out in memory
 - Unless we initialize it ourselves
 - So we should always do so
- What happens when we dereference pointer f?
 - If we're lucky, a program crash
 - If not, we get bad output
 - Or worse, a local over-write, leading to problems later
- Is there anything else wrong with this code?

Memory Initialization Errors

```
int main(int argc, char **argv) {  
    Foo *f;  
  
    cout << f->value() << endl;  
  
    for (int i; i < argc; ++i)  
        cout << argv [i] << endl;  
  
    return 0;  
}
```

- Heap and stack memory is not initialized for us in C++
 - Get whatever was out in memory
 - Unless we initialize it ourselves
 - So we should always do so
- What happens when we dereference pointer f?
 - If we're lucky, a program crash
 - If not, we get bad output
 - Or worse, a local over-write, leading to problems later
- Is there anything else wrong with this code?

Memory Lifetime Errors

```
Foo *bad() {  
    Foo f;  
    return &f;  
}
```

```
Foo &alsoBad() {  
    Foo f;  
    return f;  
}
```

```
Foo mediocre() {  
    Foo f;  
    return f;  
}
```

```
Foo *better() {  
    Foo *f = new Foo;  
    return f;  
}
```

- Automatic variables
 - Are destroyed on function return
 - But in bad, we return a pointer to a variable that no longer exists
 - Reference from also_bad similar
 - Like an un-initialized pointer
- What if we returned a copy?
 - Ok, we avoid the bad pointer, and end up with an actual object
 - But we do twice the work (why?)
 - And, it's a temporary variable (more on this next)
- We really want dynamic allocation here

Memory Lifetime Errors

```
int main() {  
    Foo *f = &mediocre();  
    cout << good()->value() <<  
        endl;  
    return 0;  
}
```

```
Foo mediocre() {  
    Foo f;  
    return f;  
}
```

```
Foo *good() {  
    return new Calculator;  
}
```

- Dynamically allocated variables
 - Are not garbage collected
 - But are lost if no one refers to them: called a “memory leak”
- Temporary variables
 - Are destroyed at end of statement
 - Similar to problems w/ automatics
- Can you spot 2 problems?
 - One with a temporary variable
 - One with dynamic allocation
- Important intuition
 - Incorrect *use* is the real problem here
 - Need to understand the details

Memory Lifetime Errors

```
int main() {  
    Foo *f = &mediocre();  
    cout << good()->value() <<  
        endl;  
    return 0;  
}  
  
Foo mediocre() {  
    Foo f;  
    return f;  
}  
  
Foo *good() {  
    return new Foo;  
}
```

- Dynamically allocated variables
 - Are not garbage collected
 - But are lost if no one refers to them: called a “memory leak”
- Temporary variables
 - Are destroyed at end of statement
 - Similar to problems w/ automatics
- Can you spot 2 problems?
 - One with a temporary variable
 - One with dynamic allocation
- Important intuition
 - Incorrect *use* is the real problem here
 - Need to understand the details

A More General View: Scopes

- Temporary variables
 - Are scoped to an expression, e.g., `a = b + 3 * c;`
- Automatic variables
 - Are scoped to the duration of the function in which they are declared
- Dynamically allocated variables
 - Are scoped from explicit creation (`new`) to explicit destruction (`delete`)
- Global variables
 - Are scoped to the entire lifetime of the program
 - Includes static class and namespace members
 - May still have initialization ordering issues
- Member variables
 - Are scoped to the lifetime of the object within which they reside
 - Depend on whether object itself is global, dynamic, automatic, temporary

Scopes of Variables vs. References

- Variables and the references to them may have different scopes
 - May result in obvious kinds of lifetime errors we saw earlier
 - But may also introduce more subtle issues of aliasing
 - Still can risk destroying an object too soon or too late
- Solution in C++ is a set of idioms
 - Constructor allocates, destructor de-allocates
 - Guard: ties dynamic resource scopes to automatic scopes
 - Shallow copy, deep copy semantics
 - Reference counting: ties dynamic lifetime to a group of references
 - Copy-on-write: allows more efficient management of multiple aliasing

Constructor/Destructor

```
#ifndef IntArray_H
#define IntArray_H

class IntArray {
public:
    IntArray(const int
        size);
    ~IntArray();
private:
    int m_size;
    int *m_values;
};

#endif
```

- Resources allocated by an object need to be freed
- Constructor, destructor offer good start/end points
- Fixed size array
 - Can store integer values
 - Can access elements
 - Size may differ for each object
 - Once set, array size is fixed

Constructor/Destructor

```
#include "IntArray.h"
```

```
IntArray::IntArray(int s)
    :m_size(s), m_values(NULL) {
    if (m_size > 0) {
        m_values = new int[m_size];
        if (m_values == NULL) {
            m_size = 0;
        }
    }
}
```

```
IntArray::~~IntArray() {
    delete [] m_values;
}
```

- **IntArray** initialization
- Sets **m_size**, but **NULLS** out the pointer
- Only allocate if length is greater than zero
- What does **new** do?
 - Allocates memory
 - Calls constructor(s)
- Destructor releases the allocated array

Constructor/Destructor

```
int main(int argc, char**argv) {
    IntArray a(6);
    // When a is created, it
    // allocates memory

    return 0;

    // Now when a is destroyed,
    // so is the memory it
    // allocated
}
```

- So, scopes are tied
 - `IntArray` and its memory
- What does `delete` do?
 - Calls destructor(s)
 - Frees memory
- What are `new []` and `delete []`
 - vs. `new` and `delete`
- What if `m_values` was 0 when `delete` was called?
 - Remember the `NULL`?

Applying Guard: Function Scope

```
Foo *createAndInit() {  
    Foo *f = new Foo;  
    auto_ptr<Foo> p(f);  
    init(f); // may throw exception  
    p.release();  
    return f;  
}
```

```
int run () {  
    try {  
        Foo *d = createAndInit();  
    } catch (...) {  
    }  
}
```

- Guard idiom revisited
- C++ has an `auto_ptr` class template
- `auto_ptr<X>` assumes ownership of a pointer-to-X
- `auto_ptr` destructor calls `delete` on the owned pointer
- Call `release` to break ownership by `auto_ptr` when it's safe to do so

Copy Constructor: Shallow

```
IntArray::IntArray(const IntArray &a)
    :m_size(a.m_size), m_values(a.m_values) {
}
```

- Two ways to “copy”
 - Shallow: aliases existing resources
 - Deep: makes a complete and separate copy
- Version above shows shallow copy
 - Efficient
 - But may be risky. Why?
- What’s the invariant?
 - m_size, m_values

Copy Constructor: Deep

```
IntArray::IntArray(const IntArray &a)
:m_size(a.m_size), m_values(NULL) {

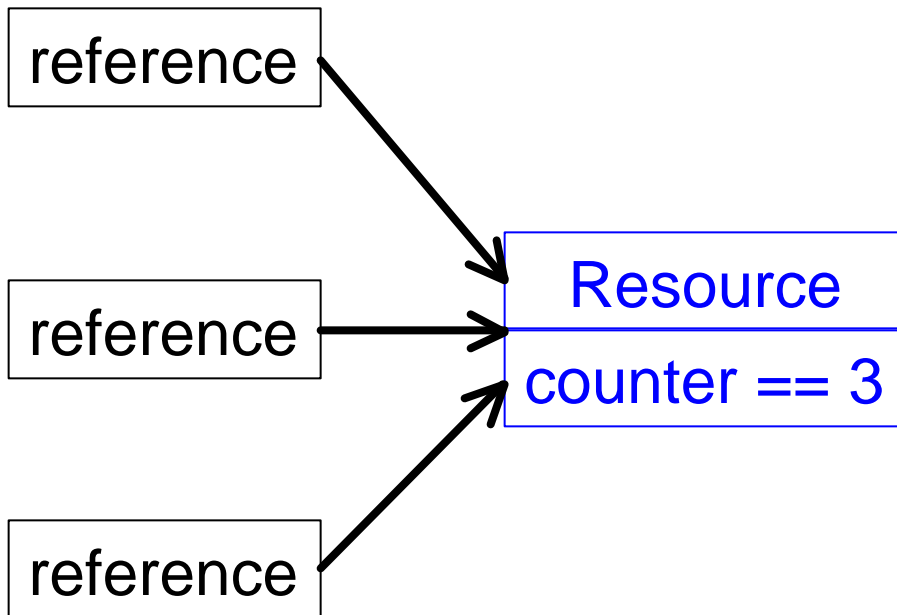
    if (m_size > 0) {
        m_values = int[m_size];

        if (m_values == NULL) {
            m_size = 0;
        }

        for(int i = 0; i < m_size; ++i) {
            m_values[i] = a.m_values[i];
        }
    }
}
```

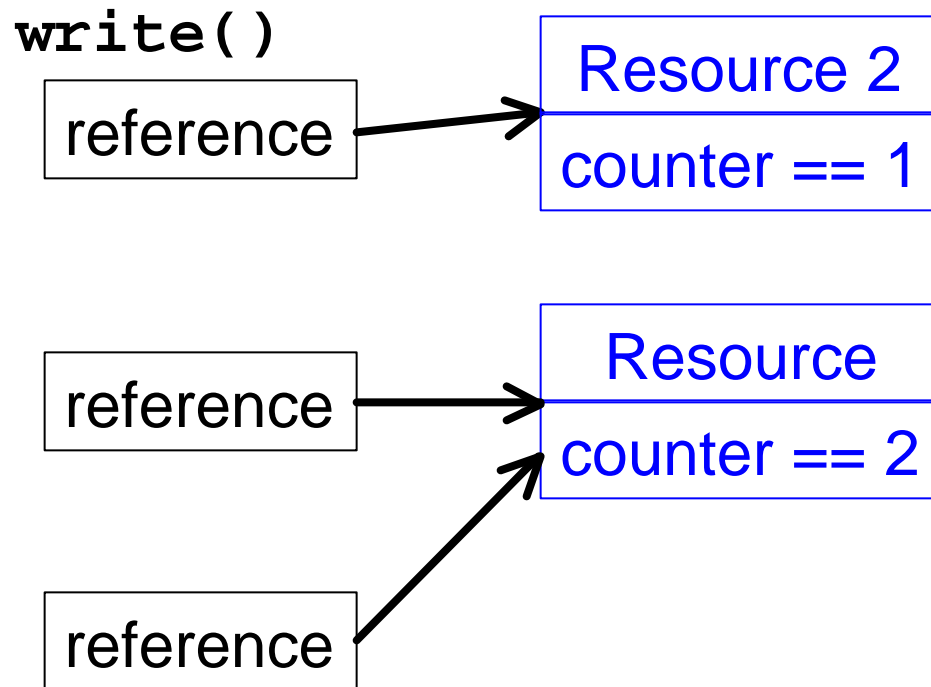
- Version to left shows deep copy
 - Safe: no aliasing
 - But may not be very efficient
- Note trade-offs with arrays
 - Allocate memory once
 - More efficient than multiple calls to new (heap search)
 - Constructor and assignment called on each array element
 - Less efficient than block copy
 - But sometimes more correct

Intro to Reference Counting



- Basic Problem
 - Resource sharing is efficient
 - But hard to tell when done
 - Must avoid early deletion
 - Must avoid leaks
- Solution
 - Share resource and a counter
 - Each new reference increments the counter
 - When a reference is done, it decrements the counter
 - When count drops to zero, delete resource and counter

Intro to Copy on Write



- Basic Problem
 - Reference counting enables safe and efficient sharing
 - But what about modifications?
 - May want logically separate copies of resource
- Solution
 - Start with reference counting
 - Writer checks for count > 1
 - Copies reference & counter
 - Updates both counters
 - Performs the write
 - Readers all share a copy, each writer can get its own

Memory Management Tips

- Know what goes where in memory
- Understand mechanics of stack and dynamic allocation
- Watch for simple lifetime errors
- Consider more subtle scope/lifetime issues
- Pay attention to aliasing (draw a picture)
- Think about shallow versus deep copying trade-offs
- Master useful idioms for C++ memory management
 - Learn how they work
 - Understand when to apply them
 - Look for ways to apply them in the labs and beyond
- Avoid pointers when you can
 - Encapsulate them within classes