RAII
Game Plan

- RAII
- Examples
- Smart Pointers
Without any extra information, how many potential code paths are in this function?

```
string EvaluateSalaryAndReturnName( Employee e ) {
    if ( e.Title() == "CEO" || e.Salary() > 100000 ) {
        cout << e.First() << " " << e.Last() << " is overpaid" << endl;
    }
    return e.First() + " " + e.Last();
}
```
**Code path 1**

(both conditionals false)

```cpp
string EvaluateSalaryAndReturnName( Employee e ) {

    if ( e.Title() == "CEO" || e.Salary() > 100000 ) {
        cout << e.First() << " " << e.Last() << " is overpaid" << endl;
    }
    return e.First() + " " + e.Last();

}
```
Code path 1
(both conditionals false)

string EvaluateSalaryAndReturnName( Employee e ) {

    if ( e.Title() == "CEO" || e.Salary() > 100000 ) {
        cout << e.First() << " " << e.Last() << " is overpaid" << endl;
    }
    return e.First() + " " + e.Last();

}
Code path 1
(both conditionals false)

```cpp
string EvaluateSalaryAndReturnName( Employee e ) {

    if ( e.Title() == "CEO" || e.Salary() > 100000 ) {
        cout << e.First() << " " << e.Last() << " is overpaid" << endl;
    }

    return e.First() + " " + e.Last();
}
```
string EvaluateSalaryAndReturnName( Employee e ) {
    if ( e.Title() == "CEO" || e.Salary() > 100000 ) {
        cout << e.First() << " " << e.Last() << " is overpaid" << endl;
    }
    return e.First() + " " + e.Last();
}
string EvaluateSalaryAndReturnName(Employee e) {
    if ( e.Title() == "CEO" || e.Salary() > 100000 ) {
        cout << e.First() << " " << e.Last() << " is overpaid" << endl;
    }
    return e.First() + " " + e.Last();
}
Code path 2
(first conditional false, second true)

string EvaluateSalaryAndReturnName( Employee e ) {

    if ( e.Title() == "CEO" || e.Salary() > 100000 ) {
        cout << e.First() << " " << e.Last() << " is overpaid" << endl;
    }
    return e.First() + " " + e.Last();
}

string EvaluateSalaryAndReturnName( Employee e ) {
    if ( e.Title() == "CEO" || e.Salary() > 100000 ) {
        cout << e.First() << " " << e.Last() << " is overpaid" << endl;
    }
    return e.First() + " " + e.Last();
}
Code path 2
(first conditional false, second true)

```cpp
string EvaluateSalaryAndReturnName( Employee e ) {
    if ( e.Title() == "CEO" || e.Salary() > 100000 ) {
        cout << e.First() << " "
             << e.Last() << " is overpaid" << endl;
    }
    return e.First() + " " + e.Last();
}
```
Code path 2
(first conditional false, second true)

```cpp
string EvaluateSalaryAndReturnName( Employee e ) {

    if ( e.Title() == "CEO" || e.Salary() > 100000 ) {
        cout << e.First() << " "
            << e.Last() << " is overpaid" << endl;
    }

    return e.First() + " " + e.Last();
}
```
Code path 3
(first conditional true, short circuit)

```cpp
string EvaluateSalaryAndReturnName( Employee e ) {
    if ( e.Title() == "CEO" || e.Salary() > 100000 ) {
        cout << e.First() << " "
            << e.Last() << " is overpaid" << endl;
    }
    return e.First() + " " + e.Last();
}
```
Code path 3
(first conditional true, short circuit)

```cpp
string EvaluateSalaryAndReturnName( Employee e ) {
    if ( e.Title() == "CEO" || e.Salary() > 100000 ) {
        cout << e.First() << " "
             << e.Last() << " is overpaid" << endl;
    }
    return e.First() + " " + e.Last();
}
```
Code path 3
(first conditional true, short circuit)

```cpp
string EvaluateSalaryAndReturnName( Employee e ) {

    if ( e.Title() == "CEO" || e.Salary() > 100000 ) {
        cout << e.First() << " "
             << e.Last() << " is overpaid" << endl;
    }

    return e.First() + " " + e.Last();
}
```
Code path 3
(first conditional true, short circuit)

```cpp
string EvaluateSalaryAndReturnName( Employee e ) {
    if ( e.Title() == "CEO" || e.Salary() > 100000 ) {
        std::cout << e.First() << " " << e.Last() << " is overpaid" << std::endl;
    }
    return e.First() + " " + e.Last();
}
```
Are there any more code paths?

```cpp
string EvaluateSalaryAndReturnName( Employee e ) {
    if ( e.Title() == "CEO" || e.Salary() > 100000 ) {
        cout << e.First() << " " << e.Last() << " is overpaid" << endl;
    }
    return e.First() + " " + e.Last();
}
```
Hidden function calls can all throw exceptions

```cpp
string EvaluateSalaryAndReturnName( Employee e ) {
    if ( e.Title() == "CEO" || e.Salary() > 100000 ) {
        cout << e.First() << " "
             << e.Last() << " is overpaid" << endl;
    }
    return e.First() + " " + e.Last();
}
```
User defined function calls can throw exceptions.

```cpp
string EvaluateSalaryAndReturnName(Employee e) {

    if ( e.Title() == "CEO" || e.Salary() > 100000 ) {
        cout << e.First() << " "
             << e.Last() << " is overpaid" << endl;
    }

    return e.First() + " " + e.Last();
}
```
Each of these operators could potentially be overloaded, could throw exceptions.

```cpp
string EvaluateSalaryAndReturnName( Employee e ) {
    if ( e.Title() == "CEO" || e.Salary() > 100000 ) {
        cout << e.First() << " " << e.Last() << " is overpaid" << endl;
    }
    return e.First() + " " + e.Last();
}
```

It’s possible `e.Title()` returns an object, so we might instead overload operator===(Title, string).
Initialization for string and int can throw exceptions.

```cpp
string EvaluateSalaryAndReturnName( Employee e ) {
    if ( e.Title() == "CEO" || e.Salary() > 100000 ) {
        cout << e.First() << " " << e.Last() << " is overpaid" << endl;
    }
    return e.First() + " " + e.Last();
}
```

The string constructor may throw exceptions (in C++17 it is declared noexcept).
Don’t forget the copy constructors can throw!

```cpp
string EvaluateSalaryAndReturnName( Employee e ) {

    if ( e.Title() == "CEO" || e.Salary() > 100000 ) {
        cout << e.First() << " " << e.Last() << " is overpaid" << endl;
    }
    return e.First() + " " + e.Last();
}
```
Summary

With exceptions, there are many, many code paths even in 5 lines of code.

How many code paths would a large project have?

Why does this matter?
Review: Memory Leak

A memory leak occurs when heap-allocated memory is not released after its use.

```cpp
{
    Employee* ptr = new Employee(idNumber);
}
// memory not accessible ptr is gone.
// but memory has not been released.
```
Followup example: what might go wrong?

```cpp
string EvaluateSalaryAndReturnName(int idNumber) {
    Employee* e = new Employee(idNumber);

    if ( e->Title() == "CEO" || e->Salary() > 100000 ) {
        cout << e->First() << " " << e->Last() << " is overpaid" << endl;
    }
    auto result = e->First() + " " + e->Last();

    delete e; // free memory of Employee
    return result;
}
```
Can you guarantee this function will not have a memory leak?

```cpp
string EvaluateSalaryAndReturnName(int idNumber) {
    Employee* e = new Employee(idNumber);

    if ( e->Title() == "CEO" || e->Salary() > 100000 ) {
        cout << e->First() << " " << e->Last() << " is overpaid" << endl;
    }
    auto result = e->First() + " " + e->Last();

    delete e; // free memory of Employee
    return result;
}
```
Can you guarantee this function will not have a memory leak?

```cpp
string EvaluateSalaryAndReturnName(int idNumber) {
    Employee* e = new Employee(idNumber);

    // throws an exception, control flow jumps to // a catch statement, skipping the delete call.

    delete e; // free memory of Employee
    return result;
}
```
More general concern: resources that need to be released.

Resources that need to be returned.

- Heap memory

<table>
<thead>
<tr>
<th>Acquire</th>
<th>Release</th>
</tr>
</thead>
<tbody>
<tr>
<td>new</td>
<td>delete</td>
</tr>
</tbody>
</table>
More general concern: resources that need to be released.

Resources that need to be returned.

- Heap memory
- Files
- Locks
- Sockets

<table>
<thead>
<tr>
<th>Acquire</th>
<th>Release</th>
</tr>
</thead>
<tbody>
<tr>
<td>new</td>
<td>delete</td>
</tr>
<tr>
<td>open</td>
<td>close</td>
</tr>
<tr>
<td>try_lock</td>
<td>unlock</td>
</tr>
<tr>
<td>socket</td>
<td>close</td>
</tr>
</tbody>
</table>
Aside: Exceptions

Exceptions are a way to transfer control and information to a (potential) exception handler.

```java
try {
    // code associated with exception handler
    throw [exception object];
} catch ( [exception type] e ) {
    // exception handler
} catch ( [exception type] e ) {
    // exception handler
} // etc.
```
Aside: Exceptions

Exceptions are a way to transfer control and information to a (potential) exception handler.

We aren’t cover in depth how to use exceptions (you’ve used them, kinda, in 106B, and they’re easy to figure out).

However, the idea of exception safety is extremely important in object-oriented programming.
We can’t guarantee the ‘delete’ call is called if an exception is thrown.

```cpp
string EvaluateSalaryAndReturnName(int idNumber) {
    Employee* e = new Employee(idNumber);

    if ( e->Title() == "CEO" || e->Salary() > 100000 ) {
        cout << e->First() << " "
             << e->Last() << " is overpaid" << endl;
    }
    auto result = e->First() + " " + e->Last();

    delete e; // free memory of Employee
    return result;
}
```
How do we guarantee classes release their resources?

Even if exceptions are used throughout the code?
One option: enforcing exception safety

Functions can have four levels of exception safety:

- **Nothrow exception guarantee**
  - absolutely does not throw exceptions: destructors, swaps, move constructors, etc.
- **Strong exception guarantee**
  - rolled back to the state before function call
- **Basic exception guarantee**
  - program is in valid state after exception
- **No exception guarantee**
  - resource leaks, memory corruption, bad...
Another option: avoiding exceptions entirely

Exceptions

We do not use C++ exceptions.

Source: https://google.github.io/styleguide/cppguide.html#Exceptions
Another option: avoiding exceptions entirely

Decision:
On their face, the benefits of using exceptions outweigh the costs, especially in new projects. However, for existing code, the introduction of exceptions has implications on all dependent code. If exceptions can be propagated beyond a new project, it also becomes problematic to integrate the new project into existing exception-free code. Because most existing C++ code at Google is not prepared to deal with exceptions, it is comparatively difficult to adopt new code that generates exceptions.

Given that Google's existing code is not exception-tolerant, the costs of using exceptions are somewhat greater than the costs in a new project. The conversion process would be slow and error-prone. We don't believe that the available alternatives to exceptions, such as error codes and assertions, introduce a significant burden.

Our advice against using exceptions is not predicated on philosophical or moral grounds, but practical ones. Because we'd like to use our open-source projects at Google and it's difficult to do so if those projects use exceptions, we need to advise against exceptions in Google open-source projects as well. Things would probably be different if we had to do it all over again from scratch.

Source: https://google.github.io/styleguide/cppguide.html#Exceptions

tl;dr We forgot to write RAII code initially, so it’s too hard to get started now.
RAII

Resource Acquisition Is Initialization
RAII

"The best example of why I shouldn't be in marketing"
"I didn't have a good day when I named that”
-- Bjarne Stroustrup
SBRM

Scope Based Resource Management
(better name for RAII)
CADRE

Constructor Acquires, Destructor Releases
(better name for RAII)
PIMPL

Pointer to Implementation.

(this is not another name for RAII. just wanted to bring it up since we are talking about bad C++ acronyms).
What is RAII?

All resources should be **acquired** in the constructor.

All resources should be **released** in the destructor.
What is the rationale?

There should **never** be a “half-valid” state of the object. Object **immediately usable** after its creation.

The destructor is **always called** (even with exceptions), so the resource is always freed.
You learned this in CS 106B. Is it RAII Complaint?

```cpp
void printFile () {
    ifstream input();
    input.open("hamlet.txt");

    string line;
    while (getline(input, line)) {
        cout << line << endl;
    }

    input.close();
}
```
Resource not acquired in the constructor or released in the destructor.

```cpp
void printFile () {
    ifstream input();
    input.open("hamlet.txt");

    string line;
    while (getline(input, line)) {
        cout << line << endl;
    }

    input.close();
}
```
You don’t need the close call. The ifstream library is RAII-compliant.

```cpp
void printFile () {
    ifstream input("hamlet.txt");

    string line;
    while (getline(input, line)) { // might throw exception
        cout << line << endl;
    }

    // no close call needed!
} // stream destructor, releases access to file
```
Resource not acquired in the constructor or released in the destructor.

```cpp
template
void cleanDatabase (mutex& databaseLock, map<int, int>& database) {

databaseLock.lock();

// other threads will not modify database
// modify the database
// if exception thrown, mutex never unlocked!

databaseLock.unlock();
}
```
The fix: an object whose sole job is to release the resource when it goes out of scope.

```cpp
void cleanDatabase (mutex& databaseLock, 
                   map<int, int>& database) {
    lock_guard<mutex>(databaseLock);
    // other threads will not modify database
    // modify the database
    // if exception thrown, that’s fine!
    // no release call needed
} // lock always unlocked when function exits.
```
How do you think lock_guard is implemented?
Here’s a non-template version.

class lock_guard {
    public:
    lock_guard(mutex& lock) : acquired_lock(lock) {
        acquired_lock.lock();
    }

    ~lock_guard() {
        acquired_lock.unlock();
    }

    private:
    mutex& acquired_lock;
}
Sidenote: mutexes cannot be copied, and cannot be moved!

```cpp
class mutex {
public:
    mutex(const mutex& other) = delete;
    mutex& operator=(const mutex& rhs) = delete;
    mutex(mutex&& other) = delete;
    mutex& operator=(mutex&& rhs) = delete;
};
```
That’s why we initialize lock_guard using the initializer_list.

class lock_guard {
public:
    lock_guard(mutex& lock) : acquired_lock(lock) {
        acquired_lock.lock();
    }
    ~lock_guard() {
        acquired_lock.unlock();
    }
private:
    mutex& acquired_lock;
}
Summary for RAII

Acquire resources in the constructor, release in the destructor.

Clients of your class won’t have to worry about acquiring/freeing resources.
RAII for Memory!
This is what we are approaching (kinda)!

C++ Will No Longer Have Pointers

(note: this is an April Fools joke by a C++ blog, even if it has some truth, please do not take this as a truth!). Do not blindly believe this.
More accurately...

R.11: Avoid calling **new** and **delete** explicitly

**Reason**

The pointer returned by **new** should belong to a resource handle (that can call **delete**). If the pointer returned by **new** is assigned to a plain/naked pointer, the object can be leaked.

**Note**

In a large program, a naked **delete** (that is a **delete** in application code, rather than part of code devoted to resource management) is a likely bug: if you have N **delete**s, how can you be certain that you don't need N+1 or N-1? The bug may be latent: it may emerge only during maintenance. If you have a naked **new**, you probably need a naked **delete** somewhere, so you probably have a bug.

**Enforcement**

(Simple) Warn on any explicit use of **new** and **delete**. Suggest using **make_unique** instead.
Is automatic memory management a good or bad thing?
Announcements
Announcements

• Survey going out today for final lecture topic(s)! Please fill out by 5 pm tomorrow for your vote to count.

• Avery’s last day! :'(

• Keep in touch with us after 106L - we’ll gladly still answer questions about C++, CS, Stanford, Berkeley, life, etc.
  • Avery: averywang@stanford.edu
  • Anna: aszeng@stanford.edu
Announcements

• Survey going out today for final lecture topic(s)! Please fill out by 5 pm tomorrow for your vote to count.

• Avery’s last day! :'(  

• Keep in touch with us after 106L - we’ll gladly still answer questions about C++, CS, Stanford, Berkeley, life, etc.
  • Avery: averywang@stanford.edu
  • Anna: aszeng@stanford.edu

Just kidding. Avery was super confused when he went through he slides.
Smart Pointers
Up till now we’ve seen how file reading and locks can be non-RAII compliant...

```cpp
void printFile () {
    ifstream input();
    input.open("hamlet.txt");

    string line;
    while (getline(input, line)) {
        cout << line << endl;
    }

    input.close();
}

void cleanDatabase (mutex& databaseLock,
                    map<int, int>& database) {
    databaseLock.lock();
    // other threads will
    // not modify database
    databaseLock.unlock();
}
```
...where the fix was to wrap it in an object to ensure that the resource is released...

```cpp
void printFile () {
    ifstream input();
    input.open("hamlet.txt");

    string line;
    while (getline(input, line)) {
        cout << line << endl;
    }
    // no close call needed!
} // stream destructor
// releases access to file

void cleanDatabase (mutex& databaseLock, map<int, int>& database) {

    lock_guard<mutex> guard(databaseLock);
    // other threads will
    // not modify database

} // destructor of lock_guard
// unlocks mutex
```
So let’s do it again! 
From CS 106B: is it RAII-compliant?

```c
void rawPtrFunc() {
    Node* n = new Node();

    // linked list fun

    delete n;
}
```
Nope! Exceptions will cause memory leaks.

```cpp
void rawPtrFunc() {
    Node* n = new Node();

    // linked list fun
    // if exception thrown, node never freed.

    delete n;
}
```
C++ has built in “smart” (RAII-compliant) pointers.

```cpp
std::unique_ptr;
std::shared_ptr;
std::weak_ptr;
// boost::scoped_ptr;
// boost::intrusive_ptr;

// std::auto_ptr – deprecated in C++11
```
Unique pointer owns its resource and deletes it when the object is destroyed.

```cpp
void rawPtrFunc() {
    Node* n = new Node();
    // linked list fun
    // if exception thrown
    // node never freed.
    delete n;
}
```

```cpp
void rawPtrFunc() {
    std::unique_ptr<Node> n(new Node);
    // linked list fun
    // if exception thrown
    // that’s fine!
}

} // Freed!
```
We can’t copy a unique_ptr! Why?

```cpp
void rawPtrFunc() {
    Node* n = new Node();
    // linked list fun
    // if exception thrown
    // node never freed.
    delete n;
}
```

```cpp
void rawPtrFunc() {
    std::unique_ptr<Node> n(new Node);
    // linked list fun
    // if exception thrown
    // that’s fine!
}
```

```cpp
} // Freed!
```
We can’t copy a unique_ptr! Why?

unique_ptr<Node> x;

resource

Data (heap)
We can’t copy a unique_ptr! Why?

```
unique_ptr<Node> x;

resource  

unique_ptr<Node> y;

resource  

Data (heap)
```
We can’t copy a `unique_ptr`! Why?

```cpp
unique_ptr<Node> x;

resource

unique_ptr<Node> y;

Freed

Freed
```
We can’t copy a `unique_ptr`! Why?

```cpp
unique_ptr<Node> x;
```

resource

Freed
We can’t copy a unique_ptr! Why?

```
unique_ptr<Node> x;
```

Freed

Double freed!
We can’t copy a unique_ptr! Why?
The fix: delete the copy constructors. Followup: can you still move?

class unique_ptr {
public:
    unique_ptr(const unique_ptr& other) = delete;
    unique_ptr& operator=(const unique_ptr& rhs) = delete;
    unique_ptr(unique_ptr&& other);
    unique_ptr& operator=(unique_ptr&& rhs);
};
But often times you want multiple pointers to the same object!

```cpp
{   Node* ptr1 = new Node(5);
    {   Node* ptr2 = ptr1;
    }
}
```
Cannot copy unique_ptrs!

```cpp
{
    unique_ptr<Node> ptr1 = unique_ptr(new Node(5));

    {
        unique_ptr<Node> ptr2 = ptr1;
        // error – call to deleted copy constructor
    }
}
```
A resource can be pointed to by any number of shared_ptrs

```cpp
{
    shared_ptr<Node> ptr1 = shared_ptr(new Node(5));

    {
        shared_ptr<Node> ptr2 = ptr1;
        // Copy constructor OK!
    }
}
```
Memory is freed when no pointers point to it.

```cpp
{
    shared_ptr<Node> ptr1 = shared_ptr(new Node(5));
    // Use ptr1
    {
        shared_ptr<Node> ptr2 = ptr1;
        // Use ptr1 and ptr2
    }
    // Use ptr1
} // No pointers pointing to Node, Node is freed.
```
Memory is freed when no pointers point to it.

```cpp
{
    shared_ptr<Node> ptr1 = shared_ptr(new Node(5));
    // Use ptr1
    {
        shared_ptr<Node> ptr2 = ptr1;
        // Use ptr1 and ptr2
    }
    // Use ptr1
} // No pointers pointing to Node, Node is freed.
```

Important: this only works if new shared_ptrs created by copying!
Implementing shared_ptr.

- How are these implemented?
  
  **Reference counting!**

- **Idea:** Store an int that keeps track of the number currently referencing that data
  - Gets incremented in copy constructor/copy assignment
  - Gets decremented in destructor or when overwritten with copy assignment
  - Frees the resource when reference count hits 0
Consider our previous example.

```cpp
void rawPtrFunc() {
    Node* n = new Node();
    // linked list fun
    // if exception thrown
    // node never freed.
    delete n;
}
```

```cpp
void rawPtrFunc() {
    std::unique_ptr<Node> n(new Node);
    // linked list fun
    // if exception thrown
    // that’s fine!
}
```

} // Freed!
Consider our previous example.

```cpp
void rawPtrFunc() {
    Node* n = new Node();
    // linked list fun
    // if exception thrown
    // node never freed.
    delete n;
}

void rawPtrFunc() {
    std::unique_ptr<Node> n(new Node);
    // linked list fun
    // if exception thrown
    // that’s fine!
}
```
Consider our previous example.

```cpp
void rawPtrFunc() {
    Node* n = new Node();

    // linked list fun
    // if exception thrown
    // node never freed.

    delete n;
}
```

```cpp
void rawPtrFunc() {
    std::shared_ptr<Node> n(new Node);

    // linked list fun
    // if exception thrown
    // that’s fine!

} // Freed!
```
This still works! Why?

```cpp
void rawPtrFunc() {
    Node* n = new Node();
    // linked list fun
    // if exception thrown
    // node never freed.
    delete n;
}
```

```cpp
void rawPtrFunc() {
    std::shared_ptr<Node> n = new Node();
    // linked list fun
    // if exception thrown
    // that’s fine!
}
} // Freed!
```
Next time

Final lecture