http://web.stanford.edu/class/cs106l/

Special Topics

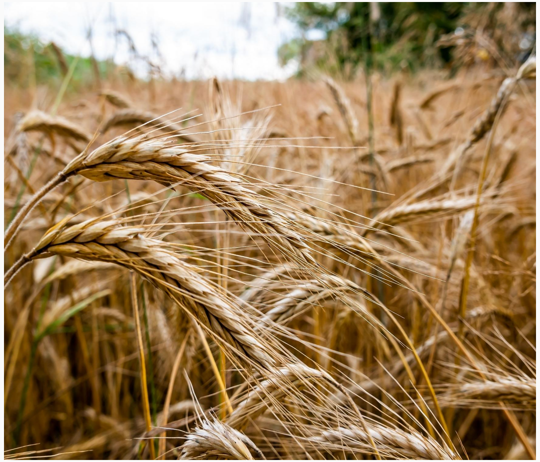
RAII, smart pointers, building projects, and more!

智能指针

CS106L - Fall 22

Attendance!

http://bit.ly/3VgQr06



Announcements!

- **This is our last real class!** Thursday’s class will be an overview of what we’ve covered as well as extra office hours!
- Late days for assignments **are automatic** – no need to let us know if you’re using them!
- For assignments, the general guideline for if it counts as completed is **if it runs**. Build errors result in no completion.

Special Topics

http://web.stanford.edu/class/cs106l/

CONTENTS

Special Topics

http://web.stanford.edu/class/cs106l/

CONTENTS

Identifying code paths

How many code paths exist in this function?

```
string get_name_and_print_sweet_tooth(Person p) {
  if (p.favorite_food() == "chocolate" ||
       p.favorite_drink() == "milkshake") {
     cout << p.first() << " "
          << p.last() << " has a sweet tooth!" << endl;
  }
  return p.first() + " " + p.last();
}
```

Special Topics

http://web.stanford.edu/class/cs106l/

Identifying code paths

How many code paths exist in this function?

```
string get_name_and_print_sweet_tooth(Person p) {
  if (p.favorite_food() == "chocolate" ||
      p.favorite_drink() == "milkshake") {
     cout << p.first() << " "
          << p.last() << " has a sweet tooth!" << endl;
  }
  return p.first() + " " + p.last();
}
```

Code path: A single run-through of the code that the computer would see

Special Topics

http://web.stanford.edu/class/cs106l/

Let's consider each possibility!

Case 1: p doesn't like chocolate or milkshakes

```
string get_name_and_print_sweet_tooth(Person p) {
  if (p.favorite_food() == "chocolate" ||
        p.favorite_drink() == "milkshake") {
      cout << p.first() << " "
           << p.last() << " has a sweet tooth!" << endl;
  }
  return p.first() + " " + p.last();
}
```

Special Topics

http://web.stanford.edu/class/cs106l/

Let's consider each possibility!

Case 1: p doesn't like chocolate or milkshakes

```
string get_name_and_print_sweet_tooth(Person p) {
  if (p.favorite_food() == "chocolate" ||
      p.favorite_drink() == "milkshake") {
    cout << p.first() << " "
         << p.last() << " has a sweet tooth!" << endl;
  }
  return p.first() + " " + p.last();
}
```

Let's consider each possibility!

Case 1: p doesn't like chocolate or milkshakes

```
string get_name_and_print_sweet_tooth(Person p) {
  if (p.favorite_food() == "chocolate" ||
      p.favorite_drink() == "milkshake") {
    cout << p.first() << " "
         << p.last() << " has a sweet tooth!" << endl;
  }
  return p.first() + " " + p.last();
}
```

Special Topics

http://web.stanford.edu/class/cs106l/

Let's consider each possibility!

Case 1: p doesn't like chocolate or milkshakes

```
string get_name_and_print_sweet_tooth(Person p) {
  if (p.favorite_food() == "chocolate" ||
        p.favorite_drink() == "milkshake") {
      cout << p.first() << " "
              << p.last() << " has a sweet tooth!" << endl;
  }
  return p.first() + " " + p.last();
}
```

Let’s consider each possibility!

Case 1: p likes milkshakes

```
string get_name_and_print_sweet_tooth(Person p) {
  if (p.favorite_food() == "chocolate" ||
       p.favorite_drink() == "milkshake") {
     cout << p.first() << " "
          << p.last() << " has a sweet tooth!" << endl;
  }
  return p.first() + " " + p.last();
}
```

Let’s consider each possibility!

Case 1: p likes milkshakes

```
string get_name_and_print_sweet_tooth(Person p) {
  if (p.favorite_food() == "chocolate" ||
        p.favorite_drink() == "milkshake") {
      cout << p.first() << " "
           << p.last() << " has a sweet tooth!" << endl;
  }
  return p.first() + " " + p.last();
}
```

Let’s consider each possibility!

Case 1: p likes milkshakes

```
string get_name_and_print_sweet_tooth(Person p) {
  if (p.favorite_food() == "chocolate" ||
      p.favorite_drink() == "milkshake") {
    cout << p.first() << " "
         << p.last() << " has a sweet tooth!" << endl;
  }
  return p.first() + " " + p.last();
}
```

Let's consider each possibility!

Case 1: p likes milkshakes

```
string get_name_and_print_sweet_tooth(Person p) {
  if (p.favorite_food() == "chocolate" ||
        p.favorite_drink() == "milkshake") {
      cout << p.first() << " "
           << p.last() << " has a sweet tooth!" << endl;
  }
  return p.first() + " " + p.last();
}
```

Special Topics

http://web.stanford.edu/class/cs106l/

Let's consider each possibility!

Case 1: p likes milkshakes

```
string get_name_and_print_sweet_tooth(Person p) {
  if (p.favorite_food() == "chocolate" ||
      p.favorite_drink() == "milkshake") {
    cout << p.first() << " "
         << p.last() << " has a sweet tooth!" << endl;
  }
  return p.first() + " " + p.last();
}
```

Special Topics

http://web.stanford.edu/class/cs106l/

Let's consider each possibility!

Case 1: p likes chocolate (and maybe milkshakes)

```
string get_name_and_print_sweet_tooth(Person p) {
  if (p.favorite_food() == "chocolate" ||
        p.favorite_drink() == "milkshake") {
      cout << p.first() << " "
           << p.last() << " has a sweet tooth!" << endl;
  }
  return p.first() + " " + p.last();
}
```

Let’s consider each possibility!

Case 1: p likes chocolate (and maybe milkshakes)

```
string get_name_and_print_sweet_tooth(Person p) {
  if (p.favorite_food() == "chocolate" ||
        p.favorite_drink() == "milkshake") {
      cout << p.first() << " "
           << p.last() << " has a sweet tooth!" << endl;
  }
  return p.first() + " " + p.last();
}
```

Let’s consider each possibility!

Case 1: p likes chocolate (and maybe milkshakes)

```
string get_name_and_print_sweet_tooth(Person p) {
  if (p.favorite_food() == "chocolate" ||
        p.favorite_drink() == "milkshake") {
      cout << p.first() << " "
           << p.last() << " has a sweet tooth!" << endl;
  }
  return p.first() + " " + p.last();
}
```

Special Topics

http://web.stanford.edu/class/cs106l/

Let's consider each possibility!

Case 1: p likes chocolate (and maybe milkshakes)

```
string get_name_and_print_sweet_tooth(Person p) {
  if (p.favorite_food() == "chocolate" ||
      p.favorite_drink() == "milkshake") {
    cout << p.first() << " "
         << p.last() << " has a sweet tooth!" << endl;
  }
  return p.first() + " " + p.last();
}
```

And now we're done!

TOTAL: 3

```
string get_name_and_print_sweet_tooth(Person p) {
  if (p.favorite_food() == "chocolate" ||
        p.favorite_drink() == "milkshake") {
      cout << p.first() << " "
           << p.last() << " has a sweet tooth!" << endl;
  }
  return p.first() + " " + p.last();
}
```

00
10
01

And now we're done!

...are we?

TOTAL: 3?

```
string get_name_and_print_sweet_tooth(Person p) {
  if (p.favorite_food() == "chocolate" ||
        p.favorite_drink() == "milkshake") {
      cout << p.first() << " "
           << p.last() << " has a sweet tooth!" << endl;
  }
  return p.first() + " " + p.last();
}
```

Special Topics

http://web.stanford.edu/class/cs106l/

Now entering: exceptions!

When a function has an error, it can crash the program.

Now entering: exceptions!

When a function has an error, it can crash the program.

- This is known as “throwing” an exception.

Now entering: exceptions!

When a function has an error, it can crash the program.

- This is known as “throwing” an exception.

However, we can write code to handle these to let us continue!

Now entering: exceptions!

When a function has an error, it can crash the program.

- This is known as “throwing” an exception.

However, we can write code to handle these to let us continue!

- This is “catching” the exception!

Now entering: exceptions!

抛出异常

When a function has an error, it can crash the program.

- This is known as “throwing” an exception.

However, we can write code to handle these to let us continue!

- This is “catching” the exception!

```
try {
      // code that we check for exceptions
}
catch([exception type] e1) { // "if"
      // behavior when we encounter an error
}
catch([other exception type] e2) { // "else if"
      // ...
}
catch { // the "else" statement
      // catch-all (haha)
}
```

Special Topics

http://web.stanford.edu/class/cs106l/

Now, how many code paths do we see?

TOTAL: 3

What happens when a function throws an exception?

```
string get_name_and_print_sweet_tooth(Person p) {
  if (p.favorite_food() == "chocolate" ||
      p.favorite_drink() == "milkshake") {
    cout << p.first() << " "
         << p.last() << " has a sweet tooth!" << endl;
  }
  return p.first() + " " + p.last();
}
```

Special Topics

http://web.stanford.edu/class/cs106l/

Now, how many code paths do we see?

TOTAL: ~~3~~ 23!

What happens when a function throws an exception?

```
string get_name_and_print_sweet_tooth(Person p) {
  if (p.favorite_food() == "chocolate" ||
        p.favorite_drink() == "milkshake") {
      cout << p.first() << " "
           << p.last() << " has a sweet tooth!" << endl;
  }
  return p.first() + " " + p.last();
}
```

Hidden Code Paths

There are (at least) 23 code paths in the code before!

Hidden Code Paths

There are (at least) 23 code paths in the code before!

- (1) copy constructor of Person parameter may throw
- (5) constructor of temp string may throw

Hidden Code Paths

There are (at least) 23 code paths in the code before!

- (1) copy constructor of Person parameter may throw
- (5) constructor of temp string may throw
- (6) call to favorite_food, favorite_drink, first (2), last (2), may throw

Hidden Code Paths

There are (at least) 23 code paths in the code before!

- (1) copy constructor of Person parameter may throw
- (5) constructor of temp string may throw
- (6) call to favorite_food, favorite_drink, first (2), last (2), may throw
- (10) operators may be user-overloaded, thus may throw

Hidden Code Paths

There are (at least) 23 code paths in the code before!

- (1) copy constructor of Person parameter may throw
- (5) constructor of temp string may throw
- (6) call to favorite_food, favorite_drink, first (2), last (2), may throw
- (10) operators may be user-overloaded, thus may throw
- (1) copy constructor of string for return value may throw

Takeaway

There are often more code paths than meet the eye!

Takeaway

There are often more code paths than meet the eye!

- Make sure to cover all possible paths in test cases for production code.

Special Topics

http://web.stanford.edu/class/cs106l/

Takeaway

There are often more code paths than meet the eye!

- Make sure to cover all possible paths in test cases for production code.
- Or, catch any errors that could create other potential paths!

What else could go wrong?

Beyond exceptions, keep an eye out for anything else that could potentially go awry.

Do you see anything suspicious about this code?

```
string get_name_and_print_sweet_tooth(int id_number) {
  Person* p = new Person(id_number); // assume the constructor fills in variables
  if (p->favorite_food() == "chocolate" ||
      p->favorite_drink() == "milkshake") {
      cout << p->first() << " "
           << p->last() << " has a sweet tooth!" << endl;
  }

  auto result = p->first() + " " + p->last();
  delete p;

  return result;
}
```

What else could go wrong?

except exceptions

Beyond exceptions, keep an eye out for anything else that could potentially go awry.

Do you see anything suspicious about this code?

```
string get_name_and_print_sweet_tooth(int id_number) {
  Person* p = new Person(id_number); // assume the constructor fills in variables
  if (p->favorite_food() == "chocolate" ||
      p->favorite_drink() == "milkshake") {
      cout << p->first() << " "
           << p->last() << " has a sweet tooth!" << endl;
  }

  auto result = p->first() + " " + p->last();
  delete p;

  return result;
}
```

Special Topics

http://web.stanford.edu/class/cs106l/

What else could go wrong?

What happens if an exception is thrown?

Can we guarantee that we won't leak memory?

```
string get_name_and_print_sweet_tooth(int id_number) {
  Person* p = new Person(id_number); // assume the constructor fills in variables
  if (p->favorite_food() == "chocolate" ||
     p->favorite_drink() == "milkshake") {
     cout << p->first() << " "
          << p->last() << " has a sweet tooth!" << endl;
  }

  auto result = p->first() + " " + p->last();
  delete p;

  return result;
}
```

Special Topics

http://web.stanford.edu/class/cs106l/

new delete
open close
try_lock unlock
socket close

This problem isn't unique to pointers!

There are many resources that need to be returned after use:

	Acquire	Release
Heap memory	`new`	`delete`
Files	`open`	`close`
Locks	`try_lock`	`unlock`
Sockets	`socket`	`close`

This problem isn't unique to pointers!

There are many resources that need to be returned after use:

	Acquire	Release
Heap memory	`new`	`delete`
Files	`open`	`close`
Locks	`try_lock`	`unlock`
Sockets	`socket`	`close`

How do we guarantee resources are returned even in the event of exceptions?

Special Topics

http://web.stanford.edu/class/cs106l/

RAII: Resource Acquisition is Initialization

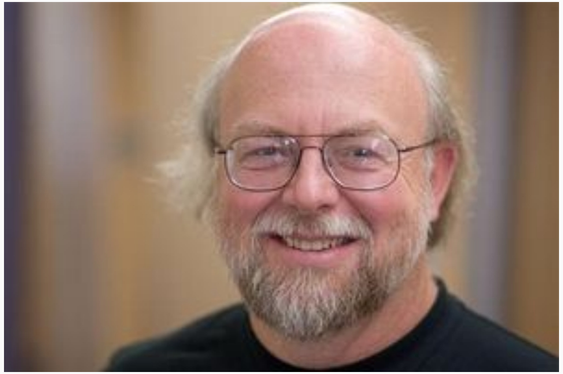
RAII is a concept developed by our good friend Bjarne and a driving philosophy behind C++, Java, and other languages.

Special Topics

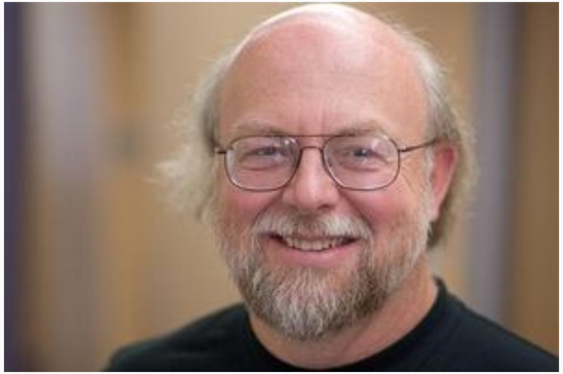
http://web.stanford.edu/class/cs106l/

RAII: Resource Acquisition is Initialization

RAII is a concept developed by our good friend Bjarne and a driving philosophy behind C++, Java, and other languages.



RAII: Resource Acquisition is Initialization

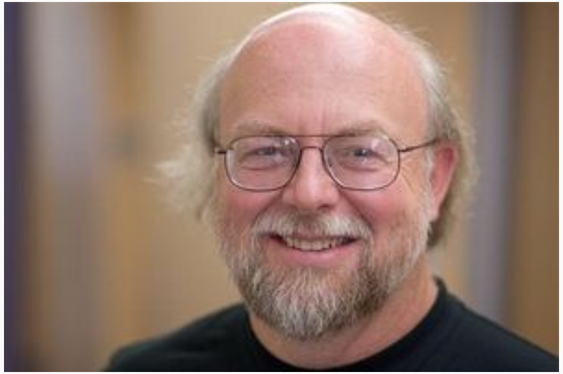


RAII is a concept developed by our good friend Bjarne and a driving philosophy behind C++, Java, and other languages.

In RAII:

RAII: Resource Acquisition is Initialization

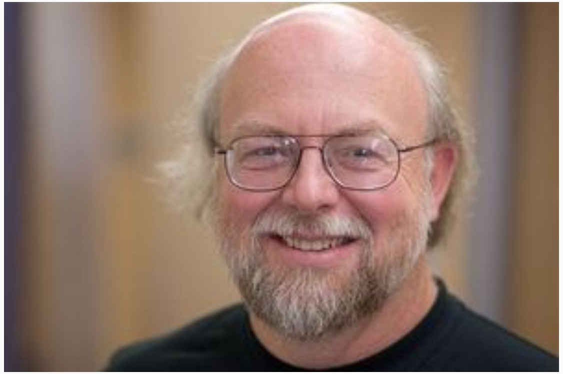
RAII is a concept developed by our good friend Bjarne and a driving philosophy behind C++, Java, and other languages.



In RAII:

- All resources used by a class should be acquired in the constructor

RAII: Resource Acquisition is Initialization



RAII is a concept developed by our good friend Bjarne and a driving philosophy behind C++, Java, and other languages.

In RAII:

- All resources used by a class should be acquired in the constructor 构造函数获取
- All resources used by a class should be released in the destructor 析构函数释放

Why RAII?

Why care about this?

Why RAII?

Why care about this?

- Objects should be usable immediately after creation.

Why RAII?

Why care about this?

- Objects should be usable immediately after creation.
- There should never be a "half-valid" state of an object, where it exists in memory but is not accessible to/used by the program.

Why RAII?

Why care about this?

- Objects should be usable immediately after creation.
- There should never be a "half-valid" state of an object, where it exists in memory but is not accessible to/used by the program.
- The destructor is always called (when the object goes out of scope), so the resource is always freed!

Is this RAII-compliant?

You've seen this in 106B!

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

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

  input.close();
}
```

Special Topics

http://web.stanford.edu/class/cs106l/

No!

The ifstream is not opened and closed in the constructor and destructor.

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

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

  input.close();
}
```

Neither is a naked mutex!

Check out CS111 for more on what this is!

```
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();
}
```

How do we fix it?

Let's implement a class whose entire job is to acquire the lock in the constructor and release it in the destructor.

```
void cleanDatabase (mutex& databaseLock,
                    map<int, int>& database) {
  lock_guard<mutex> lg(databaseLock);

  // other threads will not modify database
  // modify the database
  // if exception thrown, mutex is unlocked!

  // no need to unlock at end, as it's handle by the lock_guard
}
```

Special Topics

http://web.stanford.edu/class/cs106l/

CONTENTS

What about RAII for memory?

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.

Special Topics

http://web.stanford.edu/class/cs106l/

What about RAII for memory?

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.

The same fix works!

We fixed mutexes by creating a new object that acquires the resource in the constructor and releases it in the destructor.

The same fix works!

We fixed mutexes by creating a new object that acquires the resource in the constructor and releases it in the destructor.

We can do the same thing for memory!

The same fix works!

We fixed mutexes by creating a new object that acquires the resource in the constructor and releases it in the destructor.

We can do the same thing for memory!

- These wrapper pointers are called “smart pointers.”

The same fix works!

There are three types of smart (RAII-safe) pointers:

The same fix works!

There are three types of smart (RAII-safe) pointers:

- `std::unique_ptr`
  - Uniquely owns its resource, can't be copied

The same fix works!

There are three types of smart (RAII-safe) pointers:

- `std::unique_ptr`
    - Uniquely owns its resource, can't be copied
- `std::shared_ptr`
    - Can make copies, destructed when underlying memory goes out of scope

Special Topics

http://web.stanford.edu/class/cs106l/

std::unique_ptr
std::shared_ptr
std::weak_ptr

The same fix works!

There are three types of smart (RAII-safe) pointers:

- `std::unique_ptr`
  - Uniquely owns its resource, can't be copied
- `std::shared_ptr`
  - Can make copies, destructed when underlying memory goes out of scope
- `std::weak_ptr`
  - Models temporary ownership: when an object only needs to be accessed if it exists (convert to shared_ptr to access)

Special Topics

http://web.stanford.edu/class/cs106l/

In practice

From this...

```
void rawPtrFn() {
  Node* n = new Node;
  // do things with n
  delete n;
}
```

Special Topics

http://web.stanford.edu/class/cs106l/

In practice

自动释放

From this…

```
void rawPtrFn() {
  Node* n = new Node;
  // do things with n
  delete n;
}
```

…to this!

```
void rawPtrFn() {
  std::unique_ptr<Node> n(new Node);
  // do things with n
  // automatically freed!
}
```

Why can't we copy unique_ptr?

When a unique_ptr goes out of scope, it frees the memory associated with it.

Why can’t we copy unique_ptr?

When a unique_ptr goes out of scope, it frees the memory associated with it.

What if we had a unique_ptr, copied it, then the original destructor was called?

Why can't we copy unique_ptr?

When a unique_ptr goes out of scope, it frees the memory associated with it.

What if we had a unique_ptr, copied it, then the original destructor was called?

The copy would be pointing at deallocated memory!

Why can't we copy unique_ptr?

When a unique_ptr goes out of scope, it frees the memory associated with it.

What if we had a unique_ptr, copied it, then the original destructor was called?

The copy would be pointing at deallocated memory!

shared_ptr gets around this for us by only deallocating memory when all of the shared_ptrs have gone out of scope.

Special Topics

http://web.stanford.edu/class/cs106l/

Creating smart pointers...

```
std::unique_ptr<T> up{new T};

std::shared_ptr<T> sp{new T};

std::weak_ptr<T> wp = sp;
```

Creating smart pointers...

This is still explicitly calling new!

```
std::unique_ptr<T> up{new T};

std::shared_ptr<T> sp{new T};

std::weak_ptr<T> wp = sp;
```

列表初始化

We can fix it!

```
std::unique_ptr<T> up{new T};
std::unique_ptr<T> up = std::make_unique<T>();

std::shared_ptr<T> sp{new T};
std::shared_ptr<T> sp = std::make_shared<T>();

std::weak_ptr<T> wp = sp;
// can only be copy/move constructed (or empty)!
```

Which is better?

Always use std::make_unique<T> and std::make_shared<T>!

std::make_unique<T>

std::make_shared<T>

Which is better?

Always use std::make_unique<T> and std::make_shared<T>!

- If we don't use make_shared, then we're allocating memory twice (once for sp, and once for new T)!

Which is better?

Always use std::make_unique<T> and std::make_shared<T>!

- If we don't use make_shared, then we're allocating memory twice (once for sp, and once for new T)!
- We should be consistent across smart pointers – if we use make_shared, also use make_unique!

Special Topics

http://web.stanford.edu/class/cs106l/

CONTENTS

What do `make` and `Makefiles` do?

- `make` is a "build system"
- Uses g++ as its main engine
- Several stages to the compiler system
- Can be utilized through a `Makefile`!
- Let's take a look at a simple makefile to get some practice!

Special Topics

http://web.stanford.edu/class/cs106l/

CS111 Example

```
TARGET = sh111

CXXBASE = g++
CXX = $(CXXBASE) -std=c++17
CXXFLAGS = -ggdb -O -Wall -Werror

CPPFLAGS =
LIBS =

OBJS = sh111.o
HEADERS =

all: $(TARGET)

$(OBJS): $(HEADERS)

$(TARGET): $(OBJS)
  $(CXX) -o $@ $(OBJS) $(LIBS)


clean:
  rm -f $(TARGET) $(LIB) $(OBJS) $(LIBOBJS) *~ .*~ _test_data*

.PHONY: all clean starter
```

So then what is `cmake`?

If we have `Makefiles` already, why use `cmake`?

So then what is `cmake`?

If we have `Makefiles` already, why use `cmake`?

- `cmake` is a cross-platform `make`!

Special Topics

http://web.stanford.edu/class/cs106l/

So then what is `cmake`?

If we have `Makefiles` already, why use `cmake`?

- `cmake` is a cross-platform `make`! 跨平台
- `make` is a build system, and `cmake` creates entire build systems!
  - Another level of abstraction that takes in an even higher-level config file, ties in external libraries, and outputs a `Makefile`, which is then run.

Example cmake file

```
cmake_minimum_required(VERSION 3.0)
project(wikiracer)

set(CMAKE_CXX_STANDARD 17)
set(CMAKE_CXX_STANDARD_REQUIRED True)

find_package(cpr CONFIG REQUIRED)

# adding all files
add_executable(main main.cpp wikiscraper.cpp.o error.cpp)

target_link_libraries(main PRIVATE cpr)
```

Special Topics

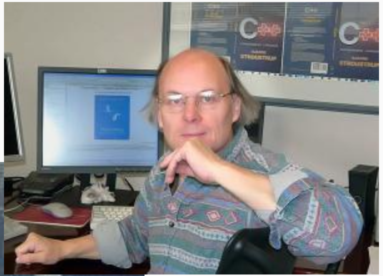
http://web.stanford.edu/class/cs106l/

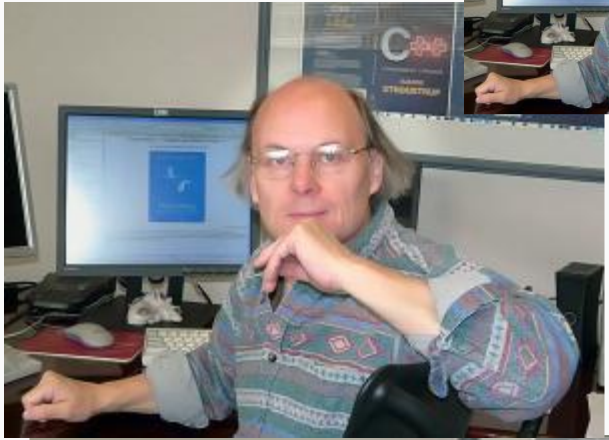
Example cmake file (ours!)

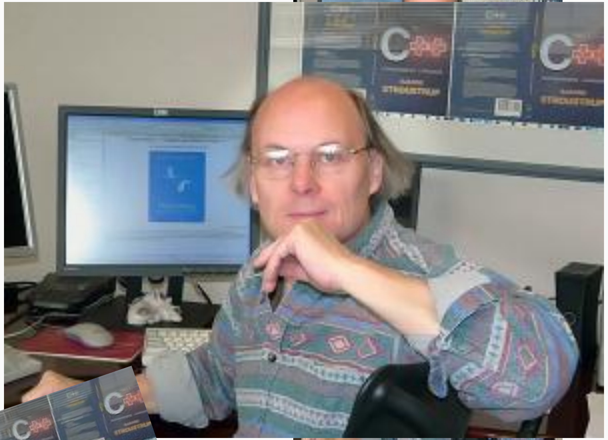
```
cmake_minimum_required(VERSION 3.0)
project(wikiracer)

set(CMAKE_CXX_STANDARD 17)
set(CMAKE_CXX_STANDARD_REQUIRED True)

find_package(cpr CONFIG REQUIRED)

# adding all files
add_executable(main main.cpp wikiscraper.cpp.o error.cpp)

target_link_libraries(main PRIVATE cpr)
```

Looks closer to a coding language as we know it!

http://web.stanford.edu/class/cs106l/







Thanks!

Next up: final class review! :(