EECS 280 – Lecture 2

The Call Stack, Procedural Abstraction, and Testing

1/10/2022

Local Variables/Objects

- At runtime, an **object is created** for each **local variable** when its **definition is run**.

```cpp
int add(int x, int y) {
  int sum = x + y;
  return sum;
}
```

- Variables in **different scopes** are different even if they have the same name.

```cpp
int main() {
  int sum = add(3, 4);
  cout << sum << endl;
}
```

- When a **local variable goes out of scope**, the corresponding **object's lifetime ends**.

Exercise: Object Lifetimes

```cpp
int plus_one(int x) {
  return (x + 1);
}

int plus_two(int y) {
  return (1 + plus_one(y));
}

int main() {
  int result = 0;
  result = plus_one(0);
  result = plus_two(result);
  cout << result; // prints 3
}
```

Question

Which variable in this program needs an object with the longest lifetime?

A) x
B) y
C) result
D) plus_two

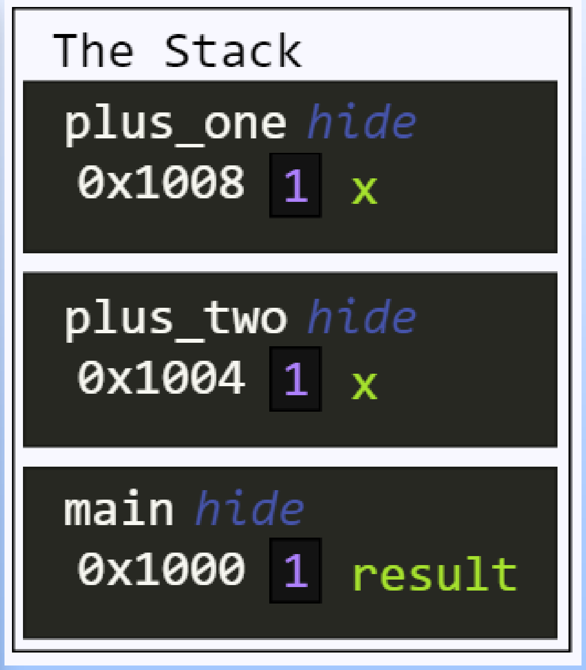
Demo: The Call Stack

- When a function is called, an activation record is created for it and added to the top of the stack.
  - Activation records are often called **stack frames**.

```
int plus_one(int x) {
  return (x + 1);
}

int plus_two(int x) {
  return (1 + plus_one(x));
}

int main() {
  int result = 0;
  result = plus_one(0);
  result = plus_two(result);
  cout << result; // prints 3
}
```



Recap: Call Stacks

- When a function is called, data for the execution of that function is stored as an **activation record.**
  - e.g. local variables, parameters, return address, etc.

- Activation records are typically stored in a **stack**.
- A stack is a container with the Last-In-First-Out (**LIFO**) property.
  - Add/remove things from the "top"[1] of the stack.
  - Can't access the bottom or from the middle.
  - Naturally leads to LIFO.

1 Note: stacks are sometimes drawn growing downward, so "top" is a relative term.

Reference: Function Calls

- To call a function, the computer must…
  1. Evaluate the actual **arguments** to the function
  2. Make a *new* and *unique* invocation of the called function
     - Push a **stack frame** (space for parameters and locals)
     - Pass **parameters**
  3. Pause the **original** function

Active flow

  4. Run the **called** function
  5. Return some value (optional)

Passive flow

  6. Start the **original** function where it left off
  7. Destroy the stack frame. (In simple cases, do nothing.)

Lobster exercise: The Call Stack

- Trace this code or step through the simulation in Lobster.
  - Which function has the largest stack frame?
  - What is the max amount of memory used by the program at any one time (assume `int`s are 4 bytes).
  - How many different stack frames are created for the `min()` function throughout the program's execution?

```
int min(int x, int y) {
  if (x < y) { return x; }
  else { return y; }
}

int minOf3(int x, int y, int z) {
  int a = min(x, y);
  int b = min(y, z);
  return min(a, b);
}
```

```
int main() {
  int a = 3;
  int b = 4;
  int c = 5;

  // prints 3
  cout << minOf3(a, b, c);
}
```

Exercise: The Call Stack

```
int min(int x, int y) {
  if (x < y) { return x; }
  else { return y; }
}

int minOf3(int x, int y, int z) {
  int a = min(x, y);
  int b = min(y, z);
  return min(a, b);
}

int main() {
  int a = 3;
  int b = 4;
  int c = 5;

  // prints 3
  cout << minOf3(a, b, c);
}
```

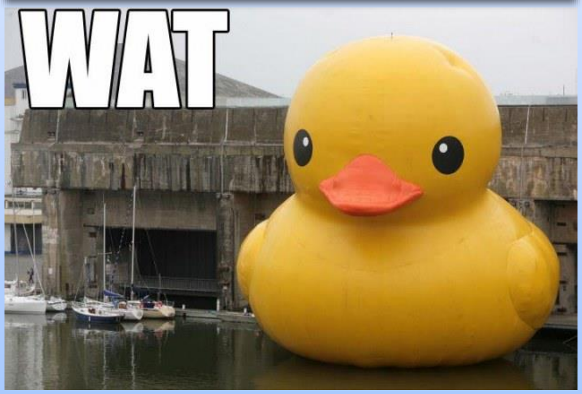
Pass By Value

- Regular parameter passing is done **by value**.
  - "By value" basically means **"make a copy"**.
  - We don't pass objects, just their values!

```
void swap(int x, int y) {
  int temp = x;
  x = y;
  y = temp;
}

int main() {
  int a = 3;
  int b = 7;
  cout << a << ", " << b;
  swap(a, b);
  cout << a << ", " << b;
}
```

It doesn't do anything.



Pass By Reference

- You can also ask for pass **by reference**.
  - "By reference" basically means **"don't make a copy"**.
  - The parameter is an **alias** for the **original** object.

```
void swap(int &x, int &y) {
  int temp = x;
  x = y;
  y = temp;
}

int main() {
  int a = 3;
  int b = 7;
  cout << a << ", " << b;
  swap(a, b);
  cout << a << ", " << b;
}
```

You can also return by reference, instead of by value. We'll come back to that later in the class.

Return By Reference

- You can also return **by reference**.
  - i.e. Return an object rather than its value.

```
int & selectLargest(int &x, int &y) {
  if (x > y) {
    return x;
  }
  else {
    return y;
  }
}

int main() {
  int a = 3; int b = 7;

  // set largest of a or b to 10
  selectLargest(a, b) = 10;
}
```

Procedural Abstraction

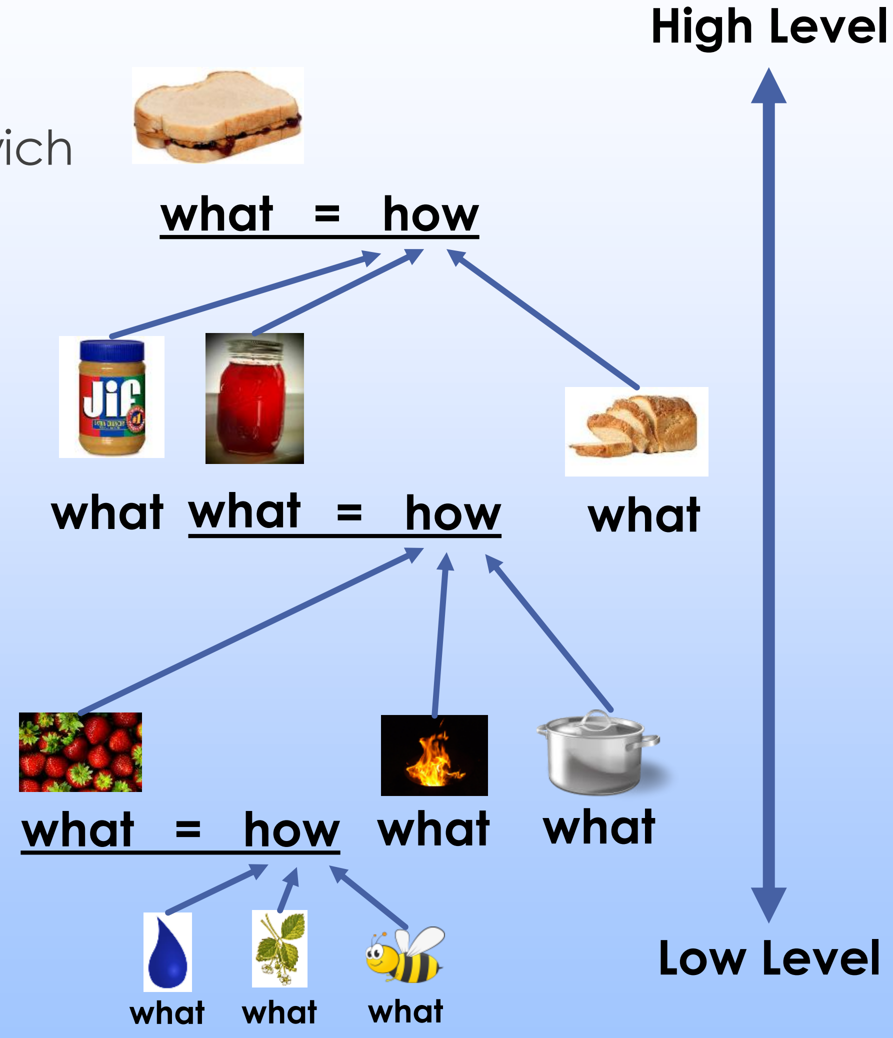
- In general…
  - …helps manage complexity.
  - …hides details.

Procedural Abstraction

- Example:
  Making A PBJ sandwich

- **what**
  The abstract idea,
  a holistic view.

- **How**
  All the details.
  Don't necessarily
  need to know this.



Procedural Abstraction

- In general…
  - …helps manage complexity.
  - …hides details.
- In computer programs…
  - …makes programs easier to maintain and modify.
  - …separates **what** code does from **how** it works. **Functions** are the main tool for this in C++.

Interfaces vs. Implementations

- In CS terminology…
  - an **interface** specifies **what** something does.
  - an **implementation** specifies **how** it works.

- For example, consider an ATM machine.
  - Most ATMs have very similar **interfaces**:
    - Card reader
    - Screen for display
    - Buttons for withdraw/deposit
  - But internally may use different **implementations:**
    - Might be a bunch of machinery?
    - Might just be a person hiding inside?

Procedural Abstraction in Code

- We can use a function as long as we know **what** it does, without having to know **how** it works.

```
#include <iostream>
#include <vector>
using namespace std;

double mean(vector<double> v) { // <--- interface
  // implementation not shown. a bunch of fancy math
}

int main() {
  vector<double> v;
  // put data in v

  double m = mean(v); // only care about the interface here
  cout << m;
}
```

Project 1 File Structure

Let's look at the project 1 structure as an example of how programs are split into **modules** based on **abstractions**…

We'll also see how **interfaces** are generally separated from their **implementations**.

Project 1 File Structure

- What's wrong here?

main.cpp

```
#include <iostream>
#include <vector>
using namespace std;

int main() {
  vector<double> v;
  // put data in v

  double m = mean(v);
  cout << m;
}

double mean(vector<double> v) {
  // implementation not shown
}
```

Compile error here: mean has not yet been declared.

Project 1 File Structure

- Potential fix #1: Place the definition of `mean` first.

`main.cpp`

```
#include <iostream>
#include <vector>
using namespace std;

double mean(vector<double> v) {
  // implementation not shown
}

int main() {
  vector<double> v;
  // put data in v

  double m = mean(v);
  cout << m;
}
```

Ok. Compiler knows what `mean` is now.

But there's still a design problem… everything is in the same file!

Project 1 File Structure

- Potential fix #2: Use a function prototype.

main.cpp

```
#include <iostream>
#include <vector>
using namespace std;

double mean(vector<double> v);

int main() {
  vector<double> v;
  // put data in v

  double m = mean(v);
  cout << m;
}

double mean(vector<double> v) {
  // implementation not shown
}
```

The function prototype acts as a declaration. It specifies the interface of mean, but doesn't fully define it.

A declaration is sufficient at this point.

The definition of the function comes later and gives the implementation of the function.

Project 1 File Structure

- Modularity goes hand in hand with abstraction.
- Let's split our P1 code into several modules:

`p1_library.cpp` `stats.cpp` `main.cpp`

main.cpp

```
#include <iostream>
#include <vector>
using namespace std;

int main() {
  vector<double> v;
  // put data in v

  double m = mean(v);
  cout << m;
}
```

stats.cpp

```
#include <vector>
using namespace std;

double mean(vector<double> v) {
  // implementation not shown
}
```

A similar problem: `main.cpp` and `stats.cpp` are compiled individually, and mean isn't declared in `main.cpp`.

Give both files to g++.

```
g++ main.cpp stats.cpp -o main.exe
```

Project 1 File Structure

- Solution: Add a header (`.h`) file for each module (except the driver).

stats.h

```
#include <vector>

double mean(std::vector<double> v);
```

The header contains all relevant <u>declarations</u>.

main.cpp

```
#include <iostream>
#include <vector>
#include "stats.h"
using namespace std;

int main() {
  vector<double> v;
  // put data in v

  double m = mean(v);
  cout << m;
}
```

stats.cpp

```
#include <vector>
#include "stats.h"
using namespace std;

double mean(vector<double> v) {
  // implementation not shown
}
```

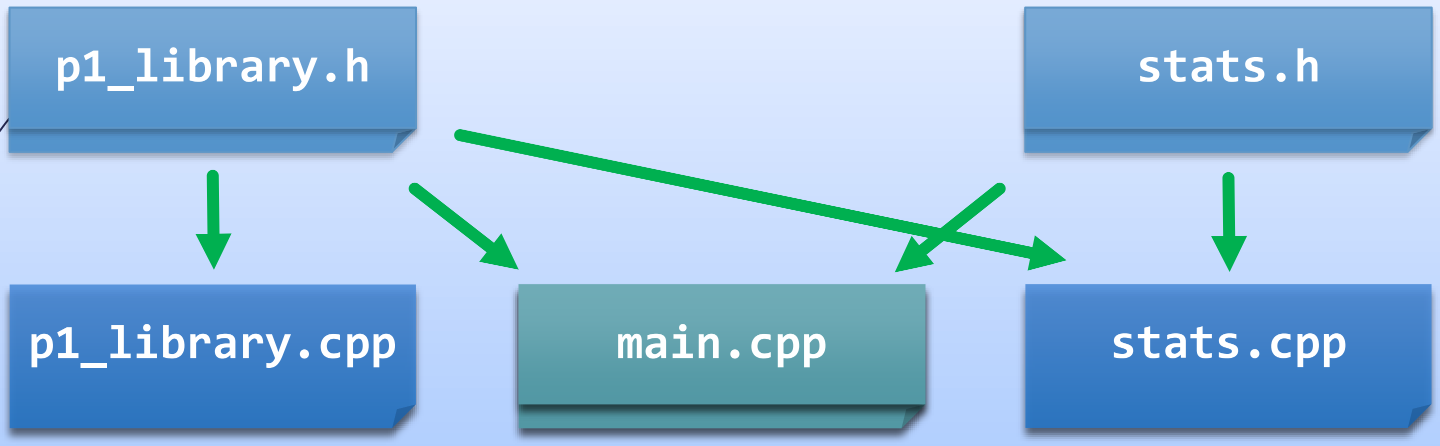
<u>Definitions</u> still go in the .cpp file.

Give both files to g++.

```
g++ main.cpp stats.cpp -o main.exe
```

Project 1 File Structure

- Here's the overall structure.
  - Caution! Never `#include` a `.cpp` file.
  - Caution! Never put a `.h` file in the `g++` command.




```
g++ p1_library.cpp stats.cpp main.cpp -o main.exe
```

P1 Abstraction

p1_library.h

```
// EFFECTS: extracts one column of data from a tab
//   separated values file (.tsv)
//   Prints errors to stdout and exits with non-zero
//   status on errors
std::vector<double> extract_column(
  std::string filename, std::string column_name);
```

You can understand **what** the function does by reading `p1_library.h`.

main.cpp

```
#include "p1_library.h"
int main() {
  // ...
  std::vector<double> v = extract_column(
    filename, column_name);
  // do something with v
}
```

You can use the `extract_column` function in your `main.cpp` without ever knowing **how** it works!

P1 Abstraction

main.cpp

```
#include "p1_library.h"
int main() {
  // ...
  std::vector<double> v = extract_column(
    filename, column_name);
  // do something with v
}
```

You can use the `extract_column` function in your `main.cpp` without ever knowing **how** it works!

Break Time!

We’ll start again in five minutes.

Specification Comments (RMEs)

- A comment that specifies the **interface** for a function.
- Requires
- Modifies
- Effects

EFFECTS and MODIFIES

- EFFECTS specifies what the function actually does.
  - What is the meaning of the **return value**?
  - Are there any **side effects**? (e.g modifying a data structure)
- MODIFIES indicates which things are potentially changed as a result of side effects.
  - e.g. A reference parameter, `cout`, global variables, etc.

```
// MODIFIES: v
// EFFECTS:  sorts v in ascending order
void sort(std::vector<double> &v);
```

Example: `sort` has the effect of sorting the input vector `v`. Of course, this may modify `v`.

REQUIRES (i.e. “ASSUMES”)

- Prerequisites for the function to make sense.
- Behavior in cases that break the `REQUIRES` clause is **undefined** by the **interface**.
- The function **implementation** only needs to cover cases allowed by the **interface’s** `REQUIRES` clause.

```
// REQUIRES: v is not empty
// EFFECTS: returns median of the numbers in v
double median(std::vector<double> v);
```

Example: `median` doesn’t make any sense for empty `v`.

- A function with no requirements is called **complete**.
- A function with requirements is called **partial**.

REQUIRES

- Program:

```cpp
// REQUIRES: v is not empty
double median(std::vector<double> v);
```

- Me:

```cpp
std::vector<double> clearlyEmptyVec;
cout << median(clearlyEmptyVec) << endl;
```

- Program: *crashes*
- Me:

Exercise: RME


```
// REQUIRES: ???
//
// MODIFIES: ???
//
// EFFECTS:  ???
//
int mystery(vector<int> &v) {
   for (int i = 0; i < v.size(); i++) {
      v[i] = v[i] / v.size();
   }
}
```

Question

Which RME clauses can be omitted for this function?

A) none
B) REQUIRES
C) MODIFIES
D) REQUIRES and MODIFIES
E) REQUIRES, MODIFIES, EFFECTS

Checking the REQUIRES Clause?

```
// REQUIRES: v is not empty
// EFFECTS: returns median of the numbers in v
double median(std::vector<double> v) {
  if (v.empty()) {
    // try to salvage the situation
  }
}
```

Don't do this.

```
// REQUIRES: v is not empty
// EFFECTS: returns median of the numbers in v
double median(std::vector<double> v) {
  assert(!v.empty()); // sound the alarms!
}
```

Do this.

assert([EXPRESSION])

- `assert()` is a programmer's friend for debugging
- Does nothing if `EXPRESSION` is true
- Exits and prints an error message if `EXPRESSION` is false

```
#include <cassert>
int main() {
  int x = 3;
  int y = 4;
  assert(x < y); // ok, does nothing
  assert(x > y); // crash with debug message
}
```

```
$ ./test
Assertion failed: (false), function main, file
test.cpp, line 6.
```

Motivation for Testing

- Good testing yields correctly working software
  - It's easier to catch bugs than to never make mistakes!
- A thorough set of tests makes development easier
  - Re-run tests after all changes and make sure you didn't break anything!
- Testing is not the same as debugging
  - **Testing**: Discovering that something is broken.
  - **Debugging**: Fixing something once you know it's broken.

Types of Testing

- **Unit testing**
  - One piece at a time (e.g., a function)
  - Find and fix bugs early! This saves you time!
    - Test smaller, less complex, easier to understand units.
    - You just wrote the code – so it's easier to debug.
- **System testing**
  - Entire project (code base)
  - Do this *after* unit testing
- **Regression testing**
  - Automatically run all unit and system tests after a code change

Kinds of Test Cases

Consider test cases for the `mode` function from project 1…

```
// REQUIRES: v is not empty
// EFFECTS:  Returns the mode of the numbers in v.
//           http://en.Wikipedia.org/wiki/Mode_(statistics)
double mode(std::vector<double> v);
```

Don't write these.

Generally not used in 280.

Type Prohibited	`assert(mode("cat") == "sleeping");`
REQUIRES Prohibited	`vector<double> empty;` `assert(mode(empty) == 0);`
Simple	`assert(mode({1, 2, 3, 2}) == 2);`

(Edge) Special	`assert(mode({3}) == 3); // single element`
	`assert(mode({1, 2, 1, 2}) == 1); // a tie`

Stress	Take the mode of a really, really big vector. Used for performance-critical applications like web servers to test how well they handle an intense load.

Example – Unit Tests

- Let's take a look at some unit tests for project 1.

```
void test_mean_basic() {
  std::vector<double> data = {1, 2, 3};
  double expected = 2;
  double actual = mean(data);
  assert(actual == expected);
  ...
}

int main() {
  test_mean_basic();
  test_mean_edge();
  test_median_basic();
 ...
}
```

If this fails, we need to <u>debug</u> the implementation of `mean`.

Debugging `mean`

- The essential nature of debugging is to figure out **precisely** where your program goes wrong.
- We can narrow down where the problem is by observing the state of the program at key points.

```
double mean(vector<double> v) {
  double s = sum(v);
  cout << "sum: " << s << endl;

  double c = count(v);
  cout << "count: " << c << endl;

  return s / c;
}
```

Think of debugging as <u>hypothesis testing</u>. For example, this line tests the hypothesis "something is wrong with the sum function".

- Using print statements can be kind of clunky. You'll see how to do this with a **debugger** in lab.

Exercise: Testing

- Let's say you're testing the **`mode()`** function…
- …and you're stranded on a desert island and can only bring 4 tests with you…

- Which 4 would you bring?

```
assert(mode({1, 2, 2, 2}) == 2);
assert(mode({1, 2, 1, 2}) == 1);
assert(mode({1, 42, 42, 42}) == 42);
assert(mode({1, 2, 2, 2, 2, 2, 2, 2, 2, 2}) == 2);
assert(mode({5, 5, 5, 3, 3}) == 5);
assert(mode({5, 3, 5, 3, 5}) == 5);
assert(mode({3}) == 3);
assert(mode({}) == 0);
```