Slides by Andrew DeOrio

EECS 280

Programming and Introductory Data Structures

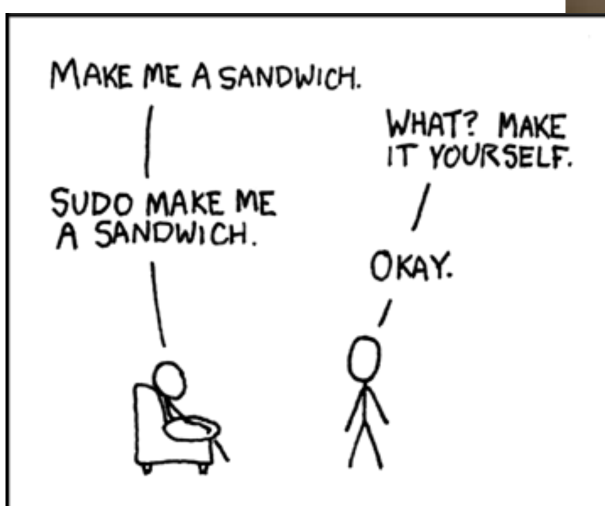
Procedural Abstraction and Recursion

Abstraction

- Abstraction helps manage complexity
- Abstraction hides details

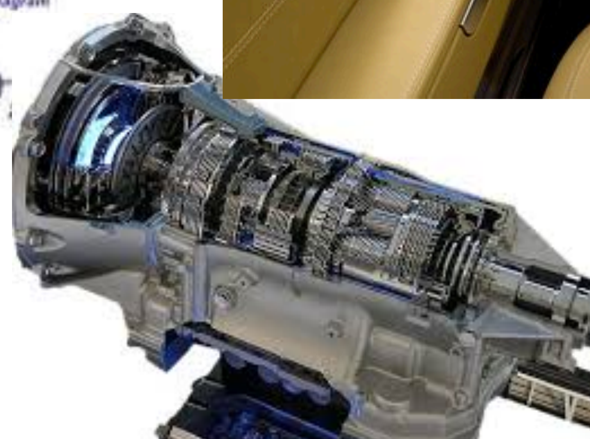
Abstraction example

- **How** make a PBJ sandwich?
- You just need to know **what** the ingredients are combine them
- You don't need to know **how** to make the ingredients
- The processes for making the ingredients have been abstracted away



Abstraction example

- Steering wheel, gear shift and pedals are an abstraction
- Hides the details of rack-and-pinion steering, transmissions, and internal combustion engine

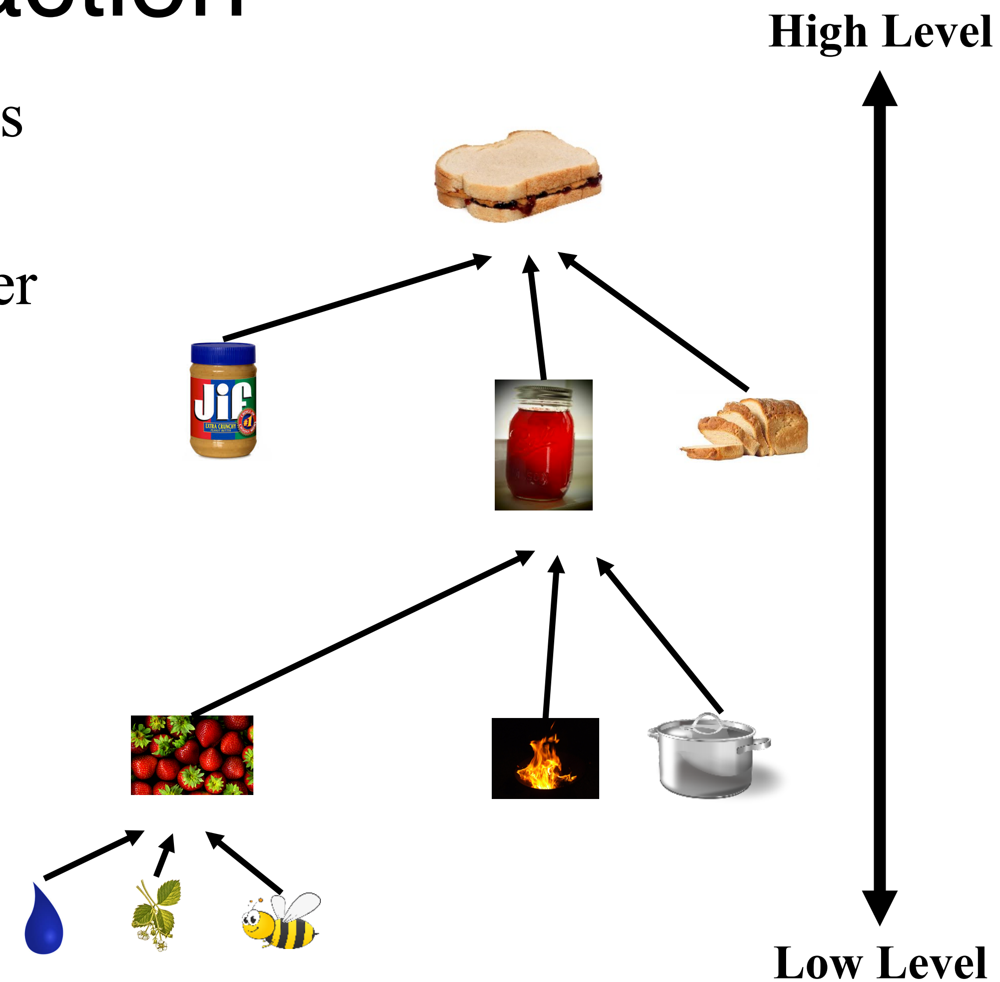


Abstraction exercise

- Try to tell your neighbor how to do something **without** using abstraction
- Examples:
  - How to ride a bike
  - How to bake a cake
  - How to submit to the autograder

Layers of Abstraction

- Abstractions of abstractions
- You don't need to worry about the "**how**" from lower layers



Abstraction in computer programs

- Abstraction lets us separate *what* code does from *how* it works
- Abstraction helps us model complex phenomena
  - Like statistical methods in project 1
- Abstraction makes programs easier to maintain and modify
  - You can change the implementation and no users of the code can tell

- We'll cover two kinds of abstraction in this class
  - Procedural abstraction
  - Data abstraction

Abstraction in computer programs

- **Procedural abstraction** lets us separate *what* a procedure does from *how* it is implemented

- In C++, we use functions to implement procedural abstraction

Example

- Here's an example from project 1
- *What* the functions do, but not *how*

`p1_library.h`

- *How* the functions work

`p1_library.cpp`

- Finally, we use our functions

`main.cpp`

Example

- Works well when you have multiple programmers

- Prof. DeOrio and student Alice agree on an abstraction

`p1_library.h`

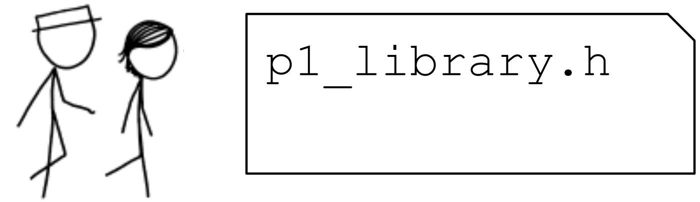
- Prof. DeOrio codes `p1_library.cpp`
  - Implements procedural abstraction

`p1_library.cpp`

- Alice codes `main.cpp`
  - Uses procedural abstraction

`main.cpp`

Cartoon credit: xkcd.com



Example

```
//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`

Cartoon credit: xkcd.com

main.cpp

Example

```
#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!

Cartoon credit: xkcd.com

Building an abstraction

p1_library.h

- Describe abstraction using function inputs, outputs, and "what it's supposed to do"
- Function signature include inputs and outputs

```
std::vector<double> extract_column(
  std::string filename, std::string column_name);
```

- Need a comment to describe "what it's supposed to do"

```
//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
```

p1_library.h

Building an abstraction

- Anatomy of a *specification comment*
- Need to answer three questions:
  - What pre-conditions must hold to use the function?
  - Does the function change any inputs (even implicit ones)? If so, how?
  - What does the procedure actually do?

`p1_library.h`

Building an abstraction

- Anatomy of a *specification comment*
- Need to answer three questions:
  - What pre-conditions must hold to use the function?
    `REQUIRES`: the pre-conditions that must hold, if any
  - Does the function change any inputs (even implicit ones)? If so, how?
    `MODIFIES`: how inputs are modified, if any
  - What does the procedure actually do?
    `EFFECTS`: what the procedure computes given legal inputs.

- You can omit `REQUIRES` or `MODIFIES` if the function doesn't require or modify anything

MODIFIES example

- Here's another example from `p1_library.h`

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

- `v` is passed by reference
- `sort` will modify `v`
- Modifying global state would also go here
- We say a function that modifies things has *side effects*

REQUIRES example

- Here's an example from project 1's `stats.h`

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

- This function REQUIRES that the input is not empty
- `sum` doesn't make any sense if there aren't any numbers!
- The function can safely assume that `v` will have size >= 1
- Functions with REQUIRES clauses are called *partial*
- Functions without REQUIRES clauses are called *complete*

Checking the requires clause

- A function implementation is free to assume that another programmer hasn't violated the REQUIRES clause
- It's a good habit to check yourself any ways

assert()

`assert( /*EXPRESSION*/ );`

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

```
#include <cassert>
int main () {
  assert(true);  // does nothing
  assert(false); // crash with debug message
}
```

```
Assertion failed: (false), function main, file test.cpp, line 5.
```

Checking the requires clause

- We can use `assert()` to check a requires clause
- GOOD HABIT!!!

```
double sum(std::vector<double> v) {
  assert(!v.empty());
  // ...
}
```

another way to do it

```
double sum(std::vector<double> v) {
  assert(v.size() > 0);
  // ...
}
```

Properties of procedural abstraction

- Local
  The implementation of an abstraction can be understood without examining any other abstraction implementation

- Substitutable
  You can replace one (correct) implementation of an abstraction with another (correct) one, and no callers of that abstraction will need to be modified

Example: substitutable

- Here's the current implementation in `p1_library.cpp`

```
void sort(std::vector<double> &v) {
  std::sort(v.begin(), v.end());
}
```

- And let's say your `mode()` function in `stats.cpp` uses `sort()`:

```
double mode(std::vector<double> v) {
  assert(!v.empty());
  sort(v);
  //...
}
```

- If the staff changes the implementation of `sort()`, do you need to change your mode function?

Example: substitutable

- If the staff changes the implementation of `sort()`, do you need to change your mode function?
- No! As long as no one changes the *abstraction* in `p1_library.h`, your code in `stats.cpp` still works!
- This is a big benefit of abstraction

The stack

- Some questions we'd like to answer:

- Each function has it's own local variables, how can the program keep track of them all?

- How does the flow of control happen between callers and callees?

The stack

- 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
  - You can add/remove things from the “top”[1] of the stack
  - You can’t take them off the bottom or out of the middle
  - Naturally leads to LIFO

[1] Note: stacks are sometimes drawn growing downward, so “top” is a relative term

The 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; //3
  return 0;
}
```

The stack

Since environments are lexically scoped, `plus_one` cannot see `plus_two`'s `x`. Instead, a copy of `plus_two`'s x is passed to `plus_one`, and stored in `plus_one`'s x

```
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; //3
  return 0;
}
```

Function Calls

1. Make a new stack frame
2. Pause the original function
3. Run the called function
4. Restart the original function where it left off
5. Destroy the stack frame

Stack example

```
main
result: ??
paused: (running)
return: ??
```

Stack

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

int main() {
1.  int result = 0;
2.  result = plus_one(0);
3.  result = plus_two(result);
4.  cout << result; //3
5.  return 0;
}
```

Stack example

main
result: ??
paused: line 2
return: ??

plus_one
x: 0
paused: (running)
return: ??

Stack

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

int main() {
1.   int result = 0;
2.   result = plus_one(0);
3.   result = plus_two(result);
4.   cout << result; //3
5.   return 0;
}
```

Stack example

```
main
result: ??
paused: line 2
return: ??
```

```
plus_one
x: 0
paused: (running)
return: 0 + 1
```

Stack

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

int main() {
1.   int result = 0;
2.   result = plus_one(0);
3.   result = plus_two(result);
4.   cout << result; //3
5.   return 0;
}
```

Stack example

<u>main</u>
result: 1
paused: (resumed)
return: ??

Stack

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

int main() {
1.   int result = 0;
2.   result = plus_one(0);
3.   result = plus_two(result);
4.   cout << result; //3
5.   return 0;
}
```

Stack example

```
main
result: 1
paused: line 3
return: ??
```

```
plus_two
x: 1
paused: (running)
return: ??
```

Stack

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

int main() {
1.   int result = 0;
2.   result = plus_one(0);
3.   result = plus_two(result);
4.   cout << result; //3
5.   return 0;
}
```

Stack example

```
main
result: 1
paused: line 3
return: ??
```

```
plus_two
x: 1
paused: line 1
return: ??
```

```
plus_one
x: 1
paused: (running)
return: ??
```

Stack

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

int main() {
1.   int result = 0;
2.   result = plus_one(0);
3.   result = plus_two(result);
4.   cout << result; //3
5.   return 0;
}
```

Stack example

<u>main</u>
result: 1
paused: line 3
return: ??

<u>plus_two</u>
x: 1
paused: line 1
return: ??

<u>plus_one</u>
x: 1
paused: (running)
return: 1 + 1

Stack

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

int main() {
1.   int result = 0;
2.   result = plus_one(0);
3.   result = plus_two(result);
4.   cout << result; //3
5.   return 0;
}
```

Stack example

<u>main</u>
result: 1
paused: line 3
return: ??

<u>plus_two</u>
x: 1
paused: (resumed)
return: 1 + 2

Stack

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

int main() {
1.   int result = 0;
2.   result = plus_one(0);
3.   result = plus_two(result);
4.   cout << result; //3
5.   return 0;
}
```

Stack example

<u>main</u>
result: 1
paused: line 3
return: ??

<u>plus_two</u>
x: 1
paused: (resumed)
return: 3

Stack

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

int main() {
1.  int result = 0;
2.  result = plus_one(0);
3.  result = plus_two(result);
4.  cout << result; //3
5.  return 0;
}
```

Stack example

main
result: 3
paused: (resumed)
return: ??

Stack

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

int main() {
1.   int result = 0;
2.   result = plus_one(0);
3.   result = plus_two(result);
4.   cout << result; //3
5.   return 0;
}
```

Stack example

```
main
result: 3
paused: line 4
return: ??
```

```
cout
...
```

Stack

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

int main() {
1.   int result = 0;
2.   result = plus_one(0);
3.   result = plus_two(result);
4.   cout << result; //3
5.   return 0;
}
```

Stack example

```
main
result: 3
paused: (resumed)
return: ??
```

Stack

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

int main() {
1.  int result = 0;
2.  result = plus_one(0);
3.  result = plus_two(result);
4.  cout << result; //3
5.  return 0;
}
```

Stack example

```
main
result: 3
paused: (resumed)
return: 0
```

Stack

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

int main() {
1.  int result = 0;
2.  result = plus_one(0);
3.  result = plus_two(result);
4.  cout << result; //3
5.  return 0;
}
```

Stack example

Stack

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

int main() {
1.   int result = 0;
2.   result = plus_one(0);
3.   result = plus_two(result);
4.   cout << result; //3
5.   return 0;
}
```

Recursion

- Functions can call other functions
- Can a function call itself?
- Let's try it

```
void countToInfinity(int x) {
  cout << start << endl;
  countToInfinity(x + 1);
}

int main() {
  countToInfinity(0);
}
```


Slide Credit: James Juett

Solving problems with recursion

- Two features of problems make recursion an attractive solution:
- Subproblems that are:
  - Similar
  - “Smaller” (closer to a base case)
- A Base Case
  - Can be solved without recursion
- The next few lectures, and most of project 2, is all about solving problems using recursion

Slide Credit: James Juett

Base case

- You have to stop somewhere

```
void countToTen(int x) {
  cout << start << endl;
  if (x == 10) return;
  countToTen(x + 1);
}

int main() {
  countToTen(0);
}
```

Recursive step

- Solve a “smaller” problem
- One that’s closer to the base case

```
void countToTen(int x) {
  cout << start << endl;
  if (x == 10) return;
  countToTen(x + 1);
}

int main() {
  countToTen(0);
}
```

Exercise: factorial

- Recall the factorial function from math class:
  - 0! = 1
  - n! = n * (n-1)!
- Try to write `factorial` using recursion

```
// REQUIRES: n >= 0
// EFFECTS: computes and returns n!
int factorial(int n){




}
```

Writing recursive functions

- Don’t try to do it all in your head
- Instead, treat it like an inductive proof

- Identify the “trivial” base case and write it explicitly
- For all other cases
  - Assume there is a function that can solve smaller versions of the same problem
  - Figure out how to get from the smaller solution to the bigger one

Solution: factorial

- Recall the factorial function from math class:
  - 0! = 1
  - n! = n * (n-1)!
- Try to write an implementation for factorial using recursion

```
// REQUIRES: n >= 0
// EFFECTS: computes and returns n!
int factorial(int n){
  if (n == 0) { // BASE CASE
    return 1;
  }
  else{
    return n * factorial(n-1); // RECURSIVE CASE
  }
}
```

Solution: factorial

- Another correct solution

```
// REQUIRES: n >= 0
// EFFECTS: computes and returns n!
int factorial(int n){
  if (n == 0) return 1;
  return n * factorial(n-1);
}
```

Exercise: draw the call stack

```
int main() {
  int x;
  x = factorial(3);
  return 0;
}

int factorial (int n) {
  if (n == 0) return 1;
  return n * factorial(n-1);
}
```