Overview

• Abstract data types
• Introduction to C++
  • Minor C differences
• Object-oriented programming
• Objects
Abstract Data Types
High-Level Example

• You have chosen to drive from Santa Barbara to Los Angeles

• You have a driver’s license
Questions

• Are these first-priority concerns?
  • The number of cylinders in the vehicle?
  • The gas mileage?
  • Manual or automatic transmission?
Discussion

• Previous questions probably not first concerns

• If it has a steering wheel, brakes, and a gas pedal, it probably is just fine

• The implementation, that is, how the engine works, is **abstract**
Another Example

• We have a program that performs arithmetic on some numbers (+, −, *, /)

• For basic correctness, how important is the mechanism used to represent integers?
Discussion

- Again, representation could vary
  - One’s complement
  - Two’s complement
  - Binary-coded decimal
  - Inductive definition

- Which representation chosen is not absolutely critical, due to abstraction
Abstract Data Types

- A way to abstract over data and the operations on said data
- Intentionally hides detail away
Question

• Why is hiding detail good? (Two big answers)
Answers

• Less information to keep track of

• Implementations can vary independently of how they are used

• E.g., with a license, you can drive a wide variety of vehicles
Three Levels of ADTs

- There are three levels to an abstract data type:
  - Application/user level
  - Logical/abstract level
  - Implementation level
Application/User Level

- Defines the problem domain
- What we need to do with it
  - Examples?
Application/User Level

• Defines the problem domain
• What we need to do with it
• Examples?
  • Where we are driving to
• The arithmetic we need to perform
Three Levels of ADTs

• There are three levels to an abstract data type:
  • Application/user level
  • Logical/abstract level
  • Implementation level
Logical/Abstract Level

• An *abstracted* view of the domain and the operations on the domain
• An interface for using the ADT
• Examples?
Logical/Abstract Level

• An abstracted view of the domain and the operations on the domain
• An interface for using the ADT
• Examples?
  • The steering wheel, brake and gas pedals
  • The +, −, *, and / operations, along with int
Three Levels of ADTs

- There are three levels to an abstract data type:
  - Application/user level
  - Logical/abstract level
  - Implementation level
Implementation Level

• How the ADT is implemented “under the hood”
• The code behind the interface
• Examples?
Implementation Level

- How the ADT is implemented “under the hood”
- The code behind the interface
- Examples?
  - The actual engine for the vehicle
  - The integer representation chosen, along with the algorithms for performing the operations
ADT Example in C
Motivation

- A programmer wants to write a 2D platforming game
- The visuals boil down to a grid of rectangles
Important Features

- Players, enemies, platforms, and walls are all rectangles
- In order for the game mechanics to work as expected, we need to be able to
  - Determine and modify the width and height of a rectangle
  - Determine the perimeter of a rectangle
  - Determine the area of a rectangle
Rectangle ADT

• What is the application level?
Rectangle ADT

- What is the application level?
- The visuals and mechanics of a 2D platforming game
Rectangle ADT

- What about the logical level from a high-level?
- Recall: width, height, perimeter, area
Rectangle ADT

• What about the logical level as a C interface? (width, height, perimeter, area)
• Data representation?
• Representation of operations?
• Try it yourself!
Example in Code
Rectangle ADT

- The implementation level is pretty simple in this case
- Area = width * height
- Perimeter = 2 * (width + height)
Problems

• With respect to how we defined ADTs, the C implementation has some issues

• What are these (two major problems)?
Problems

• With respect to how we defined ADTs, the C implementation has some issues

• What are these?

  • A rectangle is a struct, and we can always see its internal details

  • The interface is tied to this implementation
Seeing Internal Details

• We often say is this a *leaky* abstraction - it does not abstract over everything it should

• How do we hide function implementation?

• How do we hide struct implementation?
Seeing Internal Details

• We often say is this a leaky abstraction - it does not abstract over everything it should

• How do we hide function implementation?
  • Header files for interfaces, C files for implementation

• How do we hide struct implementation?
  • Just plain hard in C; no “accepted” way, and it’s an uphill battle
Interface Tied to Implementation

• The interface-defined `getArea` has only one implementation

• It is not possible to have two functions named `getArea` in C

• Necessary for a drop-in replacement
Why this Matters - Example

- The game developer notices that the game spends 50% of its time calculating area and perimeter
- The rectangles rarely change their width and height
- How might we make things faster?
Why this Matters - Example

- The game developer notices that the game spends 50% of its time calculating area and perimeter
- The rectangles rarely change their width and height
- How might we make things faster?
  - Precompute area and perimeter, and store them in the rectangle itself
Interface Tied to Implementation

- Precomputing is great for this example
- What if we only need width and height, and we want to minimize the amount of memory used?
Interface Tied to Implementation

• Precomputing is great for this example

• What if we only need width and height, and we want to minimize the amount of memory used?

• Our original implementation was the best!

• There is rarely a single perfect implementation
Interface Tied to Implementation

- This can be addressed in C, but it gets very messy
- Doing it properly requires features we won’t discuss
- Very error-prone, and leads to bulky code
- Code basically must determine which implementation is used and respond accordingly
So what if C is bad for this?

• The bulk of this class discusses different kinds of ADTs
• C is really not the language for implementing these properly
• We need a better language for this
C++
Motivation for C++

• C++ has additional features that makes implementing ADTs much cleaner
• Can hide implementation details much, much better
• Can vary implementation used relatively easily
• Can tightly couple data representation with the operations on said data
Design Goals

• Be as close to C as possible
  • Nearly backwards compatible - a superset of C

• Incorporate better support for handling ADTs, and especially \textit{object-oriented programming}
For Now

• Will talk about fundamental differences of C++ next lecture

• For now, I will be covering minor differences

• You may have to learn these on your own

• Fundamental differences need a whole lecture
Minor C++ Differences
Memory Allocation
new instead of malloc - non-arrays

// in C
int *x1 = malloc(sizeof(int));

// in C++
int *x2 = new int;
new instead of malloc - arrays

// in C
int *x1 = malloc(sizeof(int) * 5);

// in C++
int *x2 = new int[5];
delete instead of free - non-arrays

// in C
int *x1 = malloc(sizeof(int));
free(x1);

// in C++
int *x2 = new int;
delete x2;
delete[] instead of free - arrays

// in C
int *x1 = malloc(sizeof(int) * 5);
free(x1);

// in C++
int *x2 = new int[5];
delete[] x2;
delete vs. delete[

• Intuitively:
  
  • delete just frees the area
  
  • delete[] frees the area, and calls object destructors if it is an array of objects (more on those later)

• Undefined what happens if you delete (as opposed to delete[]) an array
Intermixing Old and New

- Anything allocated with `malloc` should be deallocated with `free`
- Anything allocated with `new` should be deallocated with `delete`
- Intermixing is undefined (`new/free` and `malloc/delete`)
- Unless you are interoperating with C, use `new and delete` exclusively
Overloading
Motivation

• Sometimes, a single operation makes sense in multiple different contexts
  • The `+` operator for `int` and `double`
  • `getArea` for rectangles, squares, and circles

• C limits us here. How?
Motivation

• Sometimes, a single operation makes sense in multiple different contexts

• The + operator for int and double

• getArea for triangles, rectangles, and circles

• C limits us here. How?

• + is built-in and works this way, but we cannot define anything like this
Solution

• We want to *overload* the definition of getArea

• Overloading based on the *signature* of the function
  • Name of the function
  • Number of arguments
  • Types of arguments
  • **Not** the return type (in C++)
Example

double getArea(triangle* t);
double getArea(square* s);
double getArea(circle* c);
const
Motivation

• A lot of bugs are rooted in unexpected state changes

• Something unexpectedly changes a variable’s value

• A “read-only” operation wasn’t read-only

• We would like a way to guarantee that state cannot change
Example

<table>
<thead>
<tr>
<th>What is pointed to is constant</th>
<th>The pointer itself is constant</th>
</tr>
</thead>
<tbody>
<tr>
<td>void foo(const char* const s) {</td>
<td></td>
</tr>
<tr>
<td>s[0] = 'a'; // disallowed</td>
<td></td>
</tr>
<tr>
<td>s = NULL; // disallowed</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
</tbody>
</table>
References
Motivation

- Pointers allow us to indirectly refer to data, which is very powerful
- ...but it’s also very error-prone
- We want something in between
References

• These “reference” some other data directly

• References are indirect, but they behave as if they were direct

• Unlike pointers, references are not a distinct kind of data that lives in memory (more restricted)

• Trying to get the address of a reference gets the address of what it references
References Example 1

```c
void swapPointers(int* x, int* y) {
    int temp = *x;
    *x = *y;
    *y = temp;
}

void swapRef(int& x, int& y) {
    int temp = x;
    x = y;
    y = temp;
}
```
struct point {
    int x;
    int y;
};

void swap(struct point& p) {
    int temp = p.x;
    p.x = p.y;
    p.y = temp;
}

int addedPoint(const struct point& p) {
    return p.x + p.y;
}
#include
#include

- No longer correct to put `.h` after the filename for `system-provided` files
- Still expected for your own files

```cpp
// provided by system:
#include <iostream>

// provided by you:
#include "myfile.h"
```
Namespaces
Motivation

• Every name (variable, function, struct) in C lives in the some distinct namespace

• Means we cannot define two variables with the same name at the same scope

• Global variable pain
Namespaces

• A way for the programmer to define custom namespaces

• In this class, you won’t be defining your own, but you will be using existing ones

• Most notable: \texttt{std} for the standard library
Namespaces

• Need to fully specify the name of something

• For example, `endl` is defined in namespace `std`, so to use it we must say:

  • `std::endl`
Namespaces

• Repeatedly typing out the namespace can be annoying, so we can instead say:

  • using std::endl;

• ...and then later simply say endl everywhere we would have said std::endl
Namespaces

• Sometimes we want everything from a namespace. For that, we can say:
  • using namespace std;
  • ...to put everything in the std namespace in scope (no more need to prepend std:: to everything)
Terminal I/O
Terminal I/O

• Terminal input and output are modeled as *streams* that can be read from and written to

  • `cin`: input stream
  • `cout`: output stream
  • `cerr`: error stream (often synonymous with the output stream)
Reading and Writing

- Can be done using `>>` and `<<`, respectively

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

int main() {
    int x;
    cin >> x;
    cout << "Saw: " << x << endl;
    return 0;
}
```