CSE 143

Event-Driven Applications and Frameworks

Traditional Data Processing

- Start at beginning of main and execute statements until finished

```
int main( ) {
  initialize;
  open input files;
  while (!end of file){
     process current input;
     advance to next input;
  }
  produce final summaries;
  return 0;
}
```

- Execution sequence determined by program

Event-Driven Programs

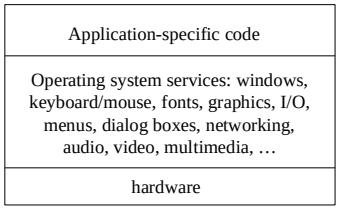
- Program reacts to *events* or *messages*: key press, mouse drag, window open/close/move, network traffic, disk I/O, etc….

```
initialize application;
done = false;
while (!done) {
   get-next-event(e);
   process-event(e);
}
```

- Execution sequence determined by user
- Essentially all desktop applications work this way (Word, PageMaker, Visual C++, …)

Application/System Design

- Contemporary operating systems provide large libraries of standard routines (APIs)
- Applications use these services as needed



Application Event Loop

- Grossly simplified idea behind a Windows app

```
initialize( );
while(GetMessage(msg)) {
   switch(msg->kind) {
      case mouseClick:
         switch(msg->where) {
            case inDialog: …
            case inWindow: …
              …
         }
      case keyPress: …
      case diskIO: …
   }
}
```

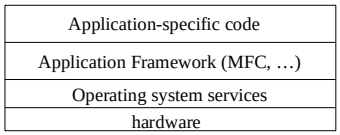
- Execution continues until **`GetMessage`** is false, which is caused by close box or File>Quit event.

Resources

- Applications are usually built with:
  - Code: C++ classes, declarations, statements
  - Resources: Icons, dialog box text, menu entries, pictures, fonts, …
- Resources are not hard-wired into the code
  - Loaded into dynamic storage as needed
  - Can be changed without having to rewrite code - ideal for producing application versions for different natural languages

Application Frameworks

- Most applications need similar code to handle standard user-interface and processing duties
- Application framework: Library (classes) with generic implementations of standard tasks
  - Examples: Microsoft Foundation Classes (MFC), Java Abstract Window Toolkit (AWT)
  - Reusable, don't need to reinvent for each application



Working with a Framework 1

- Rule #1:

Don't Panic!

- You don't need to understand everything
- You hardly need to understand anything
- Applications are never written from scratch
  - Find a chunk of code that is close to what you want and adapt it
  - Use a "wizard" or graphical interface to generate instances of standard components

Working with a Framework 2

- Main job is finding the part of the program that contains code you need to understand/modify
  - With luck, can ignore everything else
- Debugging
  - Can't just single-step starting in **`main`**
  - Set breakpoints at the beginning of your routines and other useful places
  - No cin/cout - need to use debugger
  - Examine variables on routine entry/exit
  - Step through your code, but step over/out of other code