Get out your computer or other quiz-taking device

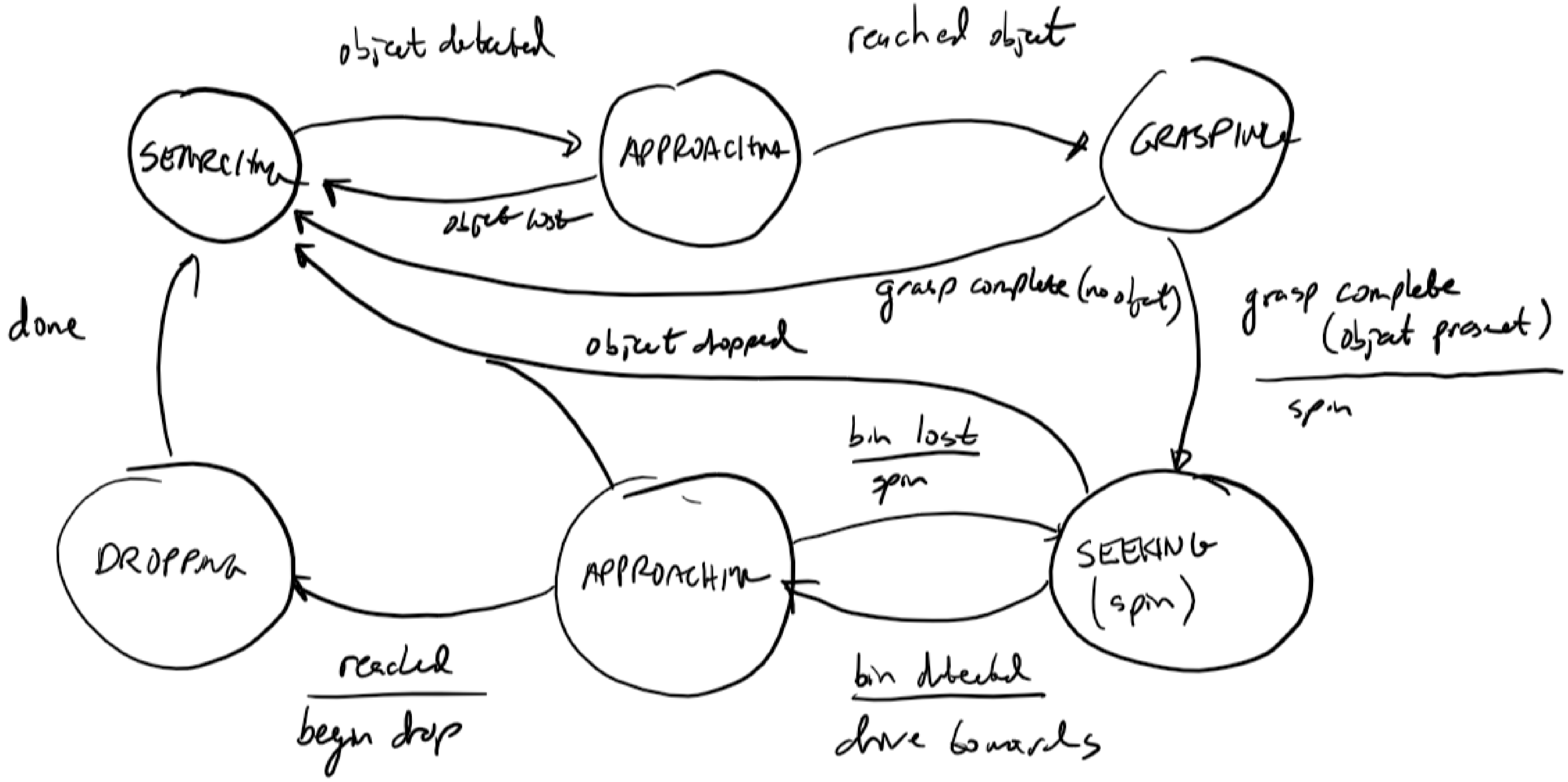
217states

Coding a state machine

We’ve seen event-driven programming…

```
while(1)
{
  if(SomeEvent() == true) HandleThatEvent();
  if(SomeOtherEvent() == true) HandleThatOtherEvent();
  if(YetAnotherEvent()) HandleThatOtherOtherEvent();
}
```

…and state machines



We defined a handler function

```
enum QUIZ_STATE {READY, IN_PROGRESS, GRADING, CLOSED};
QUIZ_STATE quizState = READY;

void HandleOpenQuizButton(void)
{
  if(quizState == READY)
  {
    quiz.Open();
    timer.Start();
    quizState = IN_PROGRESS;
  }
}
```

Pretty soon, this gets messy

```
void HandleObjectDetectButtonRelease(void)
{
 if(state == SEEKING_BIN) {...}
 else if(state == APPROACHING_BIN) {...}
 else if(state == DROPPING) {...}
}
```

The `switch` statement is your friend

```
void HandleObjectDetectButtonRelease(void)
{
 switch(state)
 {
  case SEEKING_BIN:
    ...do stuff...
    break;
  case APPROACHING _BIN:
    ...do stuff...
    break;
  case DROPPING:
    ...do stuff...
    break;
 }
}
```

Anatomy of a `switch` statement

test variable

```
switch(<variable>)
{
        case <A>:
                ...do stuff...
        case <B>:
                ...do stuff...
                break;
        case <C>
                ...do stuff...
                break;
        default:
                ...do stuff
}
```

if(<variable) == (A) { . . .

done!

Cleaning up your mess

```
void HandleObjectDetectButtonRelease(void)
{
 switch(state)
 {
  case SEEKING_BIN:
  case APPROACHING _BIN:
    state = SEEKING_ITEM;
    SeekItem();
    break;
  case DROPPING:
    itemsDeposited++;
    state = SEEKING_ITEM;
    SeekItem();
    break;
 }
}
```

Even neater, if you want (but do be careful!)

```
void HandleObjectDetectButtonRelease(void)
{
 switch(state)
 {
 case DROPPING:
   itemsDeposited++;
 case SEEKING_BIN:
 case APPROACHING _BIN:
   state = SEEKING_ITEM;
   SeekItem();
   break;
 }
}
```

Arrays

- Arrays are useful when you have a lot of similar pieces of data:
  - A number of detected objects: the camera returns an array of objects found
  - Data collected over several iterations: perhaps your BaseBot tracks the amount of time to deliver each pizza
  - Readings from a sensor array: The 2001 BaseBot has an array of eight line sensors
  - Motor positions for individual floors

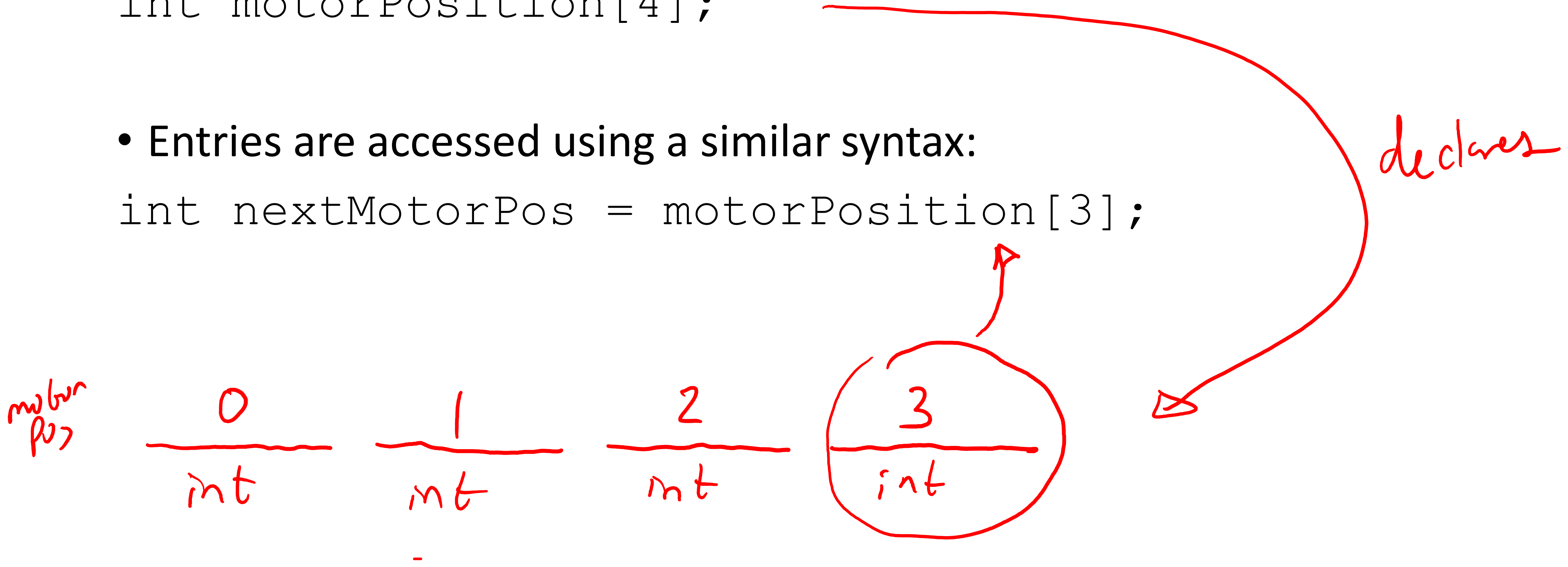
Syntax by example

- Declare an array that holds four motor positions:

```
int motorPosition[4];
```

- Entries are accessed using a similar syntax:

```
int nextMotorPos = motorPosition[3];
```



Syntax by example

- Declare an array that holds eight line sensor (ADC) readings:

```
int lineSensorReadings[8];
```

Formal syntax

```
<variable_type> <name>[<size>];
```

- Variables can be the pre-defined types:
  - int, float, char, bool, etc.

```
float temperatures[10];
```

- Or custom defined classes:

```
class Sensor;
Sensor sensors[4];
```

$\underbrace{\texttt{sensors[1]}}\texttt{.GetValue();}$

Arrays are 0 indexed!

- Declare an array that holds four motor positions:

```
int motorPosition[4];
```

- First entry:

```
int nextMotorPos = motorPosition[0];
```

- Last entry:

```
int nextMotorPos = motorPosition[3];
```

- **Bad**:

```
int nextMotorPos = motorPosition[4];
motorPosition[4] = 73;
```

You might access all the entries in a `for` loop

```
#include <stdio.h>
#include <iostream>
using namespace std;
int main(){
    int values[10];
    for(int i = 0; i < 10; i++){
      values[i] = i*i;
    }
    for(int i = 0; i < 10; i++){
        cout << "i = " << i << ": i^2 = " << values[i] << endl;
    }    return 0;
}
```

Use a *size* parameter

Const int TOTAL_MOTOR_POSITIONS = 8;

- Declare an array that holds four motor positions:

```
int motorPosition[TOTAL_MOTOR_POSITIONS];



for(int i = 0; i < TOTAL_MOTOR_POSITIONS; i++)
{
    motorPosition[i] = ...;
}
```