Interlude

- I'll skim through style notes.
  - But please do read them carefully yourselves
- Superquiz 2 now live.
- Test is Monday 10 September - first Monday of term 3.
  - Announcements on test Friday next week
- Programming Contests, anyone?
- How not to answer Q13 & Q14, Lab 4
- My answers to Superquiz 1

Lab 4, Q13

```
int isInData(char* p)
{
    int ans = 0;
    for (int i = 0; i < 100; i++) {
        if (p == &data[i]) {
            ans = 1;
        }
    }
    return ans;
}
```

No! ✗

```
int isInData(char* p)
{
    return p >= data && p <= &data[99];
}
```

Yes!

✔

8. Style

- My ENCE260 “Official” Style guideline
- Some wisdom from “The Elements of Programming Style”, Kernighan and Plauger, McGraw Hill, 1974.
  - K & P (not to be confused with K & R)

My "official" ENCE260 style guide

https://learn.canterbury.ac.nz/mod/resource/view.php?id=722806

Please read it!

Demo

K & P Rules (an edited subset)

- Write clearly - don't be too clever.
- Don't sacrifice clarity for small gains in "efficiency".
- Parenthesize to avoid ambiguity.
- Use library functions.
- Replace repetitive expressions by calls to a common function.
- Let your compiler do the simple optimizations.
- Choose variable names that mean something and won't be confused.
- Avoid multiple exits from loops.
- Make sure special cases are truly special.
- Don't use conditional branches as a substitute for a logical expression.

K & P Rules (cont'd)

- Choose a data representation that makes the program simple.
- Use data arrays to avoid repetitive control sequences.
- Use recursive procedures for recursively-defined data structures.
- Write and test a big program in small pieces.
- Write first in an easy-to-understand pseudo-language; then translate into whatever language you have to use.
- Don't stop with your first draft.
- Modularize. Use subroutines.
- Don't patch bad code – rewrite it.
- Don't comment bad code – rewrite it.

K & P Rules (cont'd)

- Make sure input cannot violate the limits of the program.
- Identify bad input; recover if possible.
- Localize input and output in subroutines.
- Watch out for off-by-one errors.
- Make sure all variables are initialized before use.
- Make sure your code "does nothing" gracefully.
- Test programs at their boundary values.
- Program defensively.
- 10.0 times 0.1 is hardly ever 1.0.
- Don't compare floating point numbers just for equality.

The Rules (cont'd)

- Make it right before you make it faster.
- Keep it right when you make it faster.
- Make it clear before you make it faster.
- Keep it simple to make it faster.
- Don't diddle code to make it faster – find a better algorithm.
- Instrument your programs. Measure before making "efficiency" changes.
- Make sure comments and code agree.
- Don't just echo the code with comments – make every comment count.
- Document your data layouts.
- Don't over-comment.

Example 1:

"Write clearly, don't be too clever."

```
// What does this program do?!

double M[SIZE][SIZE];

...

for (i = 0; i < SIZE; i++) {
    for (j = 0; j < SIZE; j++) {
        M[i][j] = (i/j)*(j/i);
    }
}
```

Example 1 (cont'd):

```
// Define an identity matrix, take 2.


for (i = 0; i < SIZE; i++) {
    for (j = 0; j < SIZE; j++) {
        M[i][j] = i == j ? 1 : 0;
    }
}
```

Example 1 (cont'd):

```
// Define an identity matrix. Take 3.

for (i = 0; i < SIZE; i++)
    for (j = 0; j < SIZE; j++) {
        if (i == j)  {          // Diagonal?
            M[i][j] = 1;        // yes
            } else {
            M[i][j] = 0;        // no
         }
    }
}
```

Which of the three versions do you prefer?

Example 2:

"Use data arrays to avoid repetitive control sequences."

```
void printCoinage(int numCents)
{
    int num2dollars = numCents / 200;
    if (num2dollars > 0) {
        printf("%d 2 dollar coins\n", num2dollars);
        numCents = numCents - num2dollars * 200;
    }
    if (numCents >= 100) {
        printf("%1 1 dollar coin\n");
        numCents -= 100;
    }
    if (numCents >= 50) {
        printf("1 50c coin\n");
        numCents -= 50;
    }
```

etc for 20c, 10c and 5c coins.

Example 2 (cont'd)

```
void printCoinage(int numCents)
{
     int coins[ ] = {200, 100, 50, 20, 10, 5};
     int i = 0;
     for (i = 0; i < 6; i++) {
        int numCoins = numCents / coins[i];
        if (numCoins > 0) {
            printf("%d %dc coins\n", numCoins, coins[i]);
            numCents -= numCoins * coins[i];
        }
     }
}
```

What's wrong with that?

Example 2 (cont'd)

```
void printCoinage(int numCents)
{
    int coins[ ] = {200, 100, 50, 20, 10, 5};
    char *names[ ] = {"2 dollar", "1 dollar", "50c",
       "20c","10c","5c"};
    int i = 0;
    for (i = 0; i < 6; i++) {
       int numCoins = numCents / coins[i];
       if (numCoins > 0) {
           printf("%d %s coins\n", numCoins, names[i]);
           numCents -= numCoins * coins[i];
       }
    }
}
```

Still imperfect. Why? Can you fix it?

Example 3

"Make sure special cases are truly special"

```
// Insert newElem into na-element array a in position pos
void insertElementIntoArray(int a[], int na, int newElem, int pos)
{
    assert(pos >= 0 && pos <= na);
    if (pos == na) {           // Is it going at the end?
        a[pos] = newElem;      // Yes. Put it there.
    } else {                   // No. Have to make room for it first
        for (int i = na; i > pos; i--) {
            a[i] = a[i - 1]; // Shift elements right to make room
        }
        a[pos] = newElem;
    }
}
```

Can it be improved?

Example 3 (cont'd)

```
// Insert newElem into na-element array a in position pos. Take 2.
void insertElementIntoArray(int a[ ], int na, int newElem, int pos)
{
    assert(pos >= 0 && pos <= na);
    if (pos != na) {  // Do we have to make room for it?
        for (int i = na; i > pos; i--)  {
            a[i] = a[i - 1];   // Shift elements right.
        }
    }
    a[pos] = newElem;
}
```

Can it be improved?