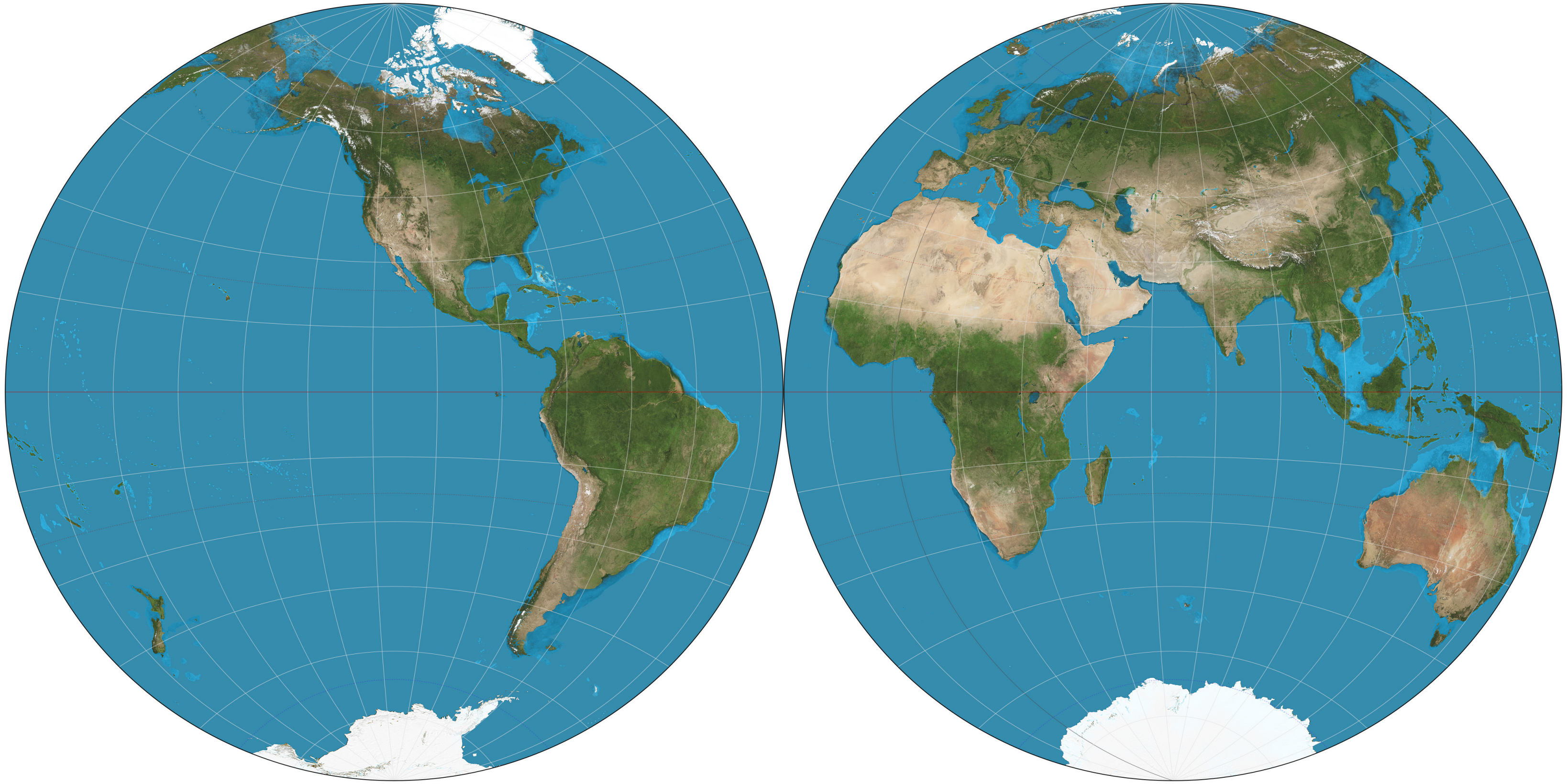


CSE 13S
Spring
2023

Computer Systems and C Programming

Nicolosi Map Projection

$$b = \frac{\pi}{2(\lambda - \lambda_0)} - \frac{2(\lambda - \lambda_0)}{\pi}$$

$$c = \frac{2\varphi}{\pi}$$

$$d = \frac{1 - c^2}{\sin\varphi - c}$$

$$M = \frac{\frac{b\sin\varphi}{d} - \frac{b}{2}}{1 + \frac{b^2}{d^2}}$$

$$N = \frac{\frac{d^2\sin\varphi}{b^2} + \frac{d}{2}}{1 + \frac{d^2}{b^2}}$$

$$x = \frac{\pi}{2}R\left(M \pm \sqrt{M^2 + \frac{\cos^2\varphi}{1 + \frac{b^2}{d^2}}}\right)$$

$$y = \frac{\pi}{2}R\left(N \pm \sqrt{N^2 - \frac{\frac{d^2}{b^2}\sin^2\varphi + d\sin\varphi - 1}{1 + \frac{d^2}{b^2}}}\right)$$

https://en.wikipedia.org/wiki/Nicolosi_globular_projection

Classroom information

Class time and location

M/W/F from 9:20 am – 10:25 am
Performing Arts M110 (Media Theater)

Final-exam day/time

Monday, June 12, 8:00 am – 11:00 am

Instructor

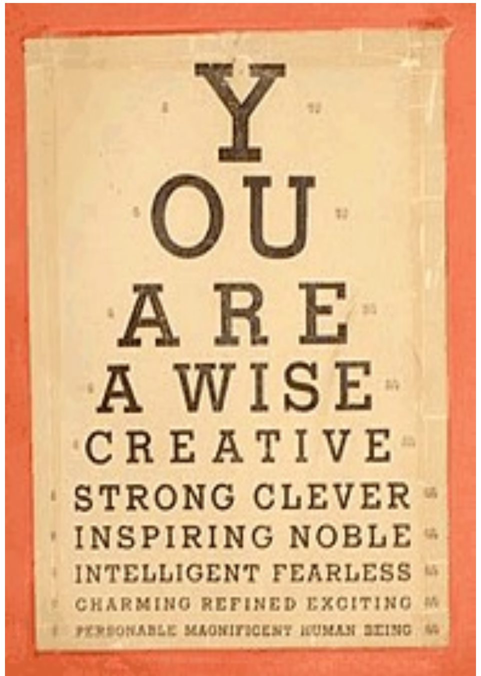
Dr. Kerry Veenstra
veenstra@ucsc.edu

Engineering 2 Building, Room 247A
(this is a shared office)

Office hours:
Tuesday 10:30 am – 12:30 pm
Thursday 2:00 pm – 4:00 pm

I'm totally supportive of DRC accommodations

- Bring me or email me your form ASAP
- Some folks need accommodations for the final only, some may need something for the quizzes: if so, we need to talk SOON!



So where does your grade come from?

- 20% Quizzes (top *n*–1 scores)
  - In class every Friday
  - I drop your lowest quiz score
- 50% Programming Assignments
- 30% Final Exam

I record the classes and post slides. **You** choose if you come to lecture—except for the quizzes.

NOTE: Assigned seats for the final exam

Canvas Web Site

- https://canvas.ucsc.edu/courses/62884

- Staff & Schedules (*still* under construction)
  - Office Hours
  - Discussion Section Times
  - Tutors & Times

Painless Way to Learn a Programming Language

Write a series of tiny programs to verify your understanding of what you read.

Assignment 1

- Will be posted soon
  - I'm still working on part of it
  - I've already written my version of the C program!
- Simulate a simplified version of the dice game "Pass the Pigs"

Defining Constant Values: `#define`

- Treated by the compiler like global text substitution

```
#define MAX_PLAYERS 10
#define DEFAULT_SEED 2023
```

- Be sure to use parentheses if you use an expression
  - Wrong

```
#define LAST_PLAYER MAX_PLAYERS - 1
```

  - Right

```
#define LAST_PLAYER (MAX_PLAYERS – 1)
```

Defining Constant Values: **#define**

- What happens if you don't use parentheses?

```
    #define A 10
    #define B A – 1
    printf("%d\n", B * B);
```

- Expands like this:

```
    printf("%d\n", A – 1 * A – 1);
    printf("%d\n", 10 – 1 * 10 – 1);
    printf("%d\n", 10 – 10 – 1);
    printf("%d\n", –1);
```

Defining Constant Values: #define

- What happens if you **do** use parentheses?

```
#define A 10
#define B (A – 1)
printf("%d\n", B * B);
```

- Expands like this:

```
printf("%d\n", (A – 1) * (A – 1));
printf("%d\n", (10 – 1) * (10 – 1));
printf("%d\n", (9) * (9));
printf("%d\n", 81);
```

Defining Constant Values: `#define`

- Best to use parentheses around all macros in a numeric expression

```
#define A 5 + 5 // oops! they forgot parens!
#define B ((A) – 1)
printf("%d\n", B * B);
```

- Expands like this:

```
printf("%d\n", ((A) – 1) * ((A) – 1));
printf("%d\n", ((5 + 5) – 1) * ((5 + 5) – 1));
printf("%d\n", (9) * (9));
printf("%d\n", 81);
```

Defining Constant Values: **`const`**

- Declare a variable but make it constant.

```
    const int BIRD = 0;
    const int CAT = 1;
    const int DOG = 2;
```

- **`const`** is okay with any type (float, double, etc.)

Defining Constant Values: **`enum`**

- Creates a set of constants

```
enum {BIRD, CAT, DOG};
```

- As if you had done this

```
const int BIRD = 0;
const int CAT = 1;
const int DOG = 2;
```

Defining Constant Values: **`enum`**

- First value defaults to 0, but you can specify another starting value.

```
enum {BIRD = 10, CAT, DOG};
```

- As if you had done this

```
const int BIRD = 10;
const int CAT = 11;
const int DOG = 12;
```

Defining a new type: **typedef**

- Similar to declaring a variable
  - Declare a variable **a**:

```
int a;
```

  - Prefix with **typedef** to declare a new type called **a**:

```
typedef int a;
```

- Now **a** is a type

```
int myint1;   // myint1 is an int
a myint2;     // myint2 also is an int
```

- Why? For more complex types, such as structures.

`enum` and **`typedef`**

- Declare an enumerated type and define its values:

```
typedef enum {CALICO, TABBY} cats;
typedef enum {BULLDOG, TERRIER} dogs;
```

- Then can declare a variable like this:

```
cats c = TABBY;
dogs d = TERRIER;
```

- Unfortunately, all enums are the same.

```
c = BULLDOG;     // the compiler allows this
```

Modulus (also known as remainder)

- You know multiplication and division

```
int a = b * c;
int fraction = n / d;     // rounds down
```

- Integer division has a remainder operator: modulus (`%`)

```
int remainder = n % d;
```

- Great for mapping a large range of values into `0 .. N - 1`

```
int i = some_number % N;  // then 0 <= i < N
```

Modulus restrictions

- Assume **`n`** and **`d`** are nonnegative

```
    int remainder = n % d;
```

- C defines what happens when n or d is negative
  - But it's not obvious
  - So I usually don't use `%` with negative numbers.