

Floating Point Numbers

String Operations

Command Line Arguments

CS143: lecture 6

Floating Point Numbers

Floating Point Numbers

Floating Point Numbers

- Relying on bits means all information is **discrete**

Floating Point Numbers

- Relying on bits means all information is **discrete**
- Integers can be enumerated → signed integer $[-2^{31}, 2^{31}-1]$

Floating Point Numbers

- Relying on bits means all information is **discrete**
- Integers can be enumerated $\rightarrow$ signed integer $[-2^{31}, 2^{31}-1]$
- Real numbers are dense $\rightarrow$ between any two real numbers there exists another. Innumerable!

Fixed Point Numbers

First idea!

Fixed Point Numbers

First idea!

- Move the decimal point a set number of places

Fixed Point Numbers

First idea!

- Move the decimal point a set number of places
- Divide all numbers by a power of 2, for example 2^{16}

Fixed Point Numbers

First idea!

- Move the decimal point a set number of places
- Divide all numbers by a power of 2, for example 2^{16}
- Range is now $[-2^{15}, 2^{15} - 2^{-16}] = [-32768, 32767.9999847]$

Fixed Point Numbers

First idea!

- Move the decimal point a set number of places
- Divide all numbers by a power of 2, for example 2^{16}
- Range is now $[-2^{15}, 2^{15} - 2^{-16}] = [-32768, 32767.9999847]$
- Fixed precision

Fixed Point Numbers

First idea!

- Move the decimal point a set number of places
- Divide all numbers by a power of 2, for example 2^{16}
- Range is now $[-2^{15}, 2^{15} - 2^{-16}] = [-32768, 32767.9999847]$
- Fixed precision
- Not a very big range despite using 32 bits

Floating Point Numbers

A better idea!

Floating Point Numbers

A better idea!

- Use scientific notation!

Floating Point Numbers

A better idea!

- Use scientific notation!
- $0.0034179688 = +3.4179688\text{E-}5 = + 3.4179688 \times 10^{-3}$

Floating Point Numbers

A better idea!

- Use scientific notation!

- $0.0034179688 = +3.4179688\text{E-}5 = \boxed{+} 3.4179688 \times 10^{-3}$
Sign

Floating Point Numbers

A better idea!

- Use scientific notation!

- $0.0034179688 = +3.4179688\text{E}-5 = \underbrace{+}_{\text{Sign}} \underbrace{3.4179688}_{\text{Coefficient}} \times 10^{-3}$

Floating Point Numbers

A better idea!

- Use scientific notation!

- $0.0034179688 = +3.4179688\text{E}-5 =$

+

3.4179688

x 10

-3

Sign

Coefficient

Exponent

Floating Point Numbers

A better idea!

- Use scientific notation!

• $0.0034179688 = +3.4179688\text{E}-5 = \underbrace{+}_{\text{Sign}} \underbrace{3.4179688}_{\text{Coefficient}} \times 10^{\underbrace{-5}_{\text{Exponent}}}$

- **Sign** tells us whether it is positive or negative

Floating Point Numbers

A better idea!

- Use scientific notation!
- $0.0034179688 = +3.4179688\text{E}-5 = \underbrace{+}_{\text{Sign}} \underbrace{3.4179688}_{\text{Coefficient}} \times 10^{\underbrace{-5}_{\text{Exponent}}}$
- **Sign** tells us whether it is positive or negative
- **Coefficient** starts with an integer 1-9, followed by **significant digits**

Floating Point Numbers

A better idea!

- Use scientific notation!

$$0.0034179688 = +3.4179688\text{E}-5 = \underbrace{+}_{\text{Sign}} \underbrace{3.4179688}_{\text{Coefficient}} \times 10^{\underbrace{-5}_{\text{Exponent}}}$$

- **Sign** tells us whether it is positive or negative
- **Coefficient** starts with an integer 1-9, followed by **significant digits**
- **Exponent** tells us how many digits we need to move the decimal point

Floating Point Numbers

Scientific notation in binary

Floating Point Numbers

Scientific notation in binary

- We need to pack three things in one integer → IEEE-754

Scientific notation in binary

- [illegible]

Scientific notation in binary

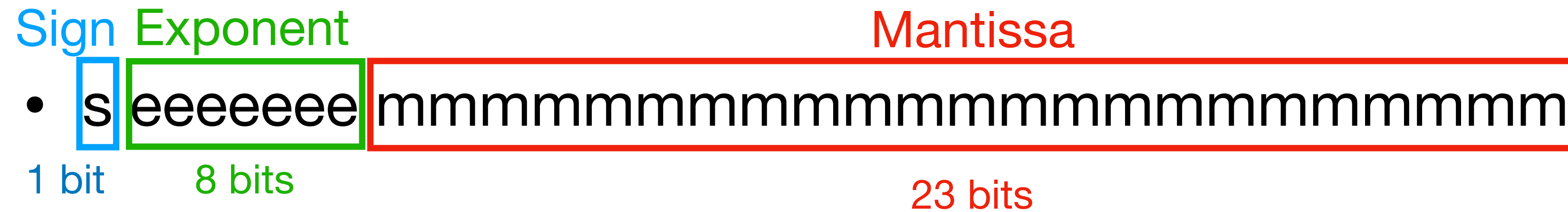
- # Sign Exponent

- 

Floating Point Numbers

Scientific notation in binary

- We need to pack three things in one integer → IEEE-754



Scientific notation in binary

- The diagram illustrates the IEEE 754 single-precision floating-point format, which consists of 32 bits divided into three main sections:

 - Sign (S):** A single bit (1 bit) indicating whether the number is positive or negative.
 - Exponent (E):** An 8-bit field used to represent the power of two by which the mantissa is multiplied.
 - Mantissa (M):** A 23-bit field representing the fractional part of the number.

The total size of the floating-point number is 32 bits (1 + 8 + 23).

Below the diagram, it states: "Sign tells us whether it is positive or negative $S = (-1)^s$ ".

Scientific notation in binary

- We need to pack three things in one integer → IEEE-754

Sign Exponent

Mantissa

- s eeeeeee mmmmmmmmmmmmmmmmmmmmmmmmmmmmm

1 bit

8 bits

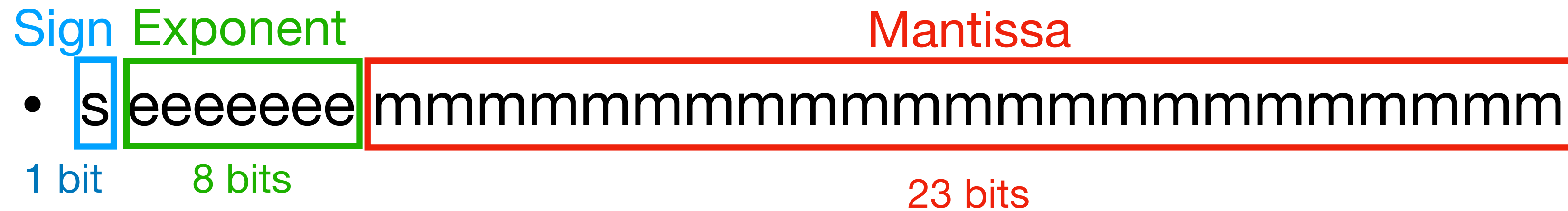
23 bits

- **Sign** tells us whether it is positive or negative $S = (-1)^s$
- **Mantissa** starts with an integer 1, followed by 23 bits of negative powers of two

$$\mathbf{M} = \sum_{i=0}^{22} m_i 2^{(i-23)}$$

Scientific notation in binary

- We need to pack three things in one integer → IEEE-754



- **Sign** tells us whether it is positive or negative $S = (-1)^s$
- **Mantissa** starts with an integer 1, followed by 23 bits of negative powers of two

$$\mathbf{M} = \sum_{i=0}^{22} m_i 2^{(i-23)}$$

- **Exponent** tells us what power of 2 we are multiplying the number

Floating Point Numbers

Scientific notation in binary

Scientific notation in binary

- s eeeeeeee mmmmmmmmmmmmmmmmmmmmmmmmmmmmm

Scientific notation in binary

- s eeeeeeee mmmmmmmmmmmmmmmmmmmmmmm

1 bit

Floating Point Numbers

Scientific notation in binary

Sign Exponent

- 
1 bit 8 bits

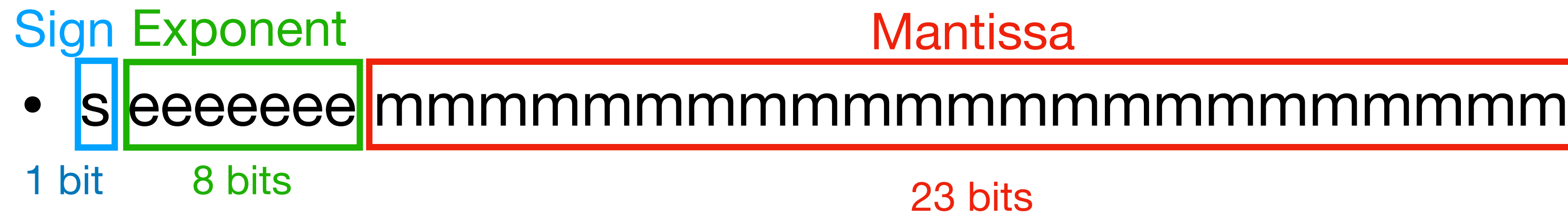
Scientific notation in binary

- s eeeeeee mmmmmmmmmmmmmmmmmmmmmmmmmmmmm

23 bits

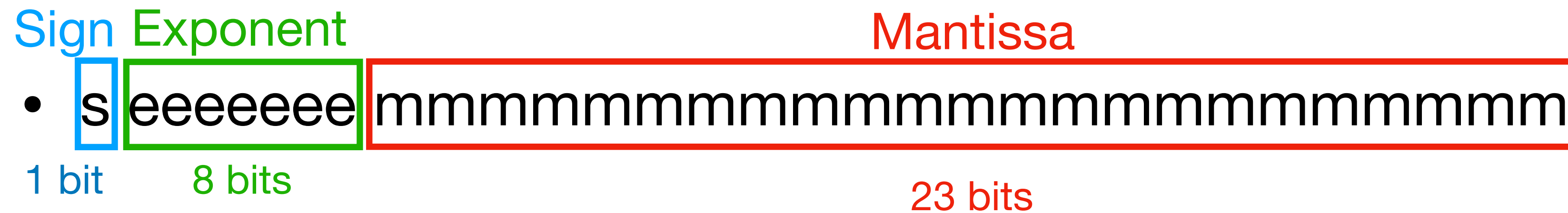
Floating Point Numbers

Scientific notation in binary



- $F = S \times M \times E$

Scientific notation in binary



- $F = S \times M \times E$

- 0.0034179688
 - $= (-1)^0 \times 1.1100\dots0 \times 2^{(118-127)}$
 - $= 1 \times 1.75 \times 2^{(-9)}$
 - $= 0\ 01110110\ 110000000000000000000000$

Floating Point Numbers

Subnormal numbers

Floating Point Numbers

Subnormal numbers

- When **exponent** is 00...00 or 11...11 then the number is called **subnormal**

Floating Point Numbers

Subnormal numbers

- When **exponent** is 00...00 or 11...11 then the number is called **subnormal**
- If **exponent** is all 0 then $F = (-1)^s \times 0.\text{mm}...\text{m} \times 2^{(-126)}$ the leading bit is 0

Floating Point Numbers

Subnormal numbers

- When **exponent** is 00...00 or 11...11 then the number is called **subnormal**
- If **exponent** is all 0 then $F = (-1)^s \times 0.\text{mm}...\text{m} \times 2^{(-126)}$ the leading bit is 0
 - Graceful degrading precision down to 1 bit for the smallest non-zero value

Floating Point Numbers

Subnormal numbers

- When **exponent** is 00...00 or 11...11 then the number is called **subnormal**
- If **exponent** is all 0 then $F = (-1)^s \times 0.\text{mm}...\text{m} \times 2^{(-126)}$ the leading bit is 0
 - Graceful degrading precision down to 1 bit for the smallest non-zero value
- **Exponent** all 1s represents infinity ($M = 0$) or NaN ($M \neq 0$)

Floating Point Numbers

Subnormal numbers

- When **exponent** is 00...00 or 11...11 then the number is called **subnormal**
- If **exponent** is all 0 then $F = (-1)^s \times 0.\text{mm}...\text{m} \times 2^{(-126)}$ the leading bit is 0
 - Graceful degrading precision down to 1 bit for the smallest non-zero value
- **Exponent** all 1s represents infinity (**M** = 0) or NaN (**M** ≠ 0)
- **Floats** or **single precision** can hold about $\log_{10}(2^{24}) \approx 7.225$ significant digits

Floating Point Numbers

Subnormal numbers

- When **exponent** is 00...00 or 11...11 then the number is called **subnormal**
- If **exponent** is all 0 then $F = (-1)^s \times 0.\text{mm}...\text{m} \times 2^{(-126)}$ the leading bit is 0
 - Graceful degrading precision down to 1 bit for the smallest non-zero value
- **Exponent** all 1s represents infinity (**M** = 0) or NaN (**M** ≠ 0)
- **Floats** or **single precision** can hold about $\log_{10}(2^{24}) \approx 7.225$ significant digits
 - Mantissa has 23 bits and the "implied" 24th bit that is always 1

Floating Point Numbers

Double precision

Floating Point Numbers

Double precision

- $S \rightarrow 1$ bit

Floating Point Numbers

Double precision

- $S \rightarrow 1$ bit
- $E \rightarrow 11$ bits

Floating Point Numbers

Double precision

- $S \rightarrow 1$ bit
- $E \rightarrow 11$ bits
- $M \rightarrow 52$ bits

Floating Point Numbers

Double precision

- **S** → 1 bit
- **E** → 11 bits
- **M** → 52 bits
- **Double** or **double precision** can hold about $\log_{10}(2^{53}) \approx 15.955$ significant digits

Floating Point Numbers

Double precision

- **S** → 1 bit
- **E** → 11 bits
- **M** → 52 bits
- **Double** or **double precision** can hold about $\log_{10}(2^{53}) \approx 15.955$ significant digits
- Goes from very small ($2^{-1074} \approx 4.9406564584124654 \times 10^{-324}$) to very big ($2^{1023} \times (2 - 2^{-52}) \approx 1.7976931348623157 \times 10^{308}$)

String Operations

Strings

Strings are `char` arrays

```
char str[20] = "Hello, World!";
```

Strings end with the special character `'\0'`

Strings

Useful operations

Strings

Useful operations

- Length of the “useful” part of the string
`size_t strlen(const char str[]);`

Strings

Useful operations

- Length of the “useful” part of the string
`size_t strlen(const char str[]);`
- Compare `str1` and `str2` alphabetically, return 0 if equal, <0 if `str1` comes before and >0 if `str1` comes after `str2`
`int strcmp(const char str1[], const char str2[]);`

Strings

Useful operations

- Length of the “useful” part of the string
`size_t strlen(const char str[]);`
- Compare `str1` and `str2` alphabetically, return 0 if equal, <0 if `str1` comes before and >0 if `str1` comes after `str2`
`int strcmp(const char str1[], const char str2[]);`
- Copy the source to destination, return the number of characters copied
`size_t strcpy(char destination[], const char source[]);`

Strings

Useful operations

- Length of the “useful” part of the string
`size_t strlen(const char str[]);`
- Compare `str1` and `str2` alphabetically, return 0 if equal, <0 if `str1` comes before and >0 if `str1` comes after `str2`
`int strcmp(const char str1[], const char str2[]);`
- Copy the source to destination, return the number of characters copied
`size_t strcpy(char destination[], const char source[]);`
- Concatenate source to destination, return the number of characters copied
`size_t strcat(char destination[], const char source[]);`

Strings

Useful operations

Strings

Useful operations

- Compare **up to num characters** between `str1` and `str2` alphabetically, return 0 if equal, <0 if `str1` comes before and >0 if `str1` comes after `str2`
`int strncmp(const char str1[], const char str2[], size_t num);`

Strings

Useful operations

- Compare **up to num characters** between `str1` and `str2` alphabetically, return 0 if equal, <0 if `str1` comes before and >0 if `str1` comes after `str2`
`int strncmp(const char str1[], const char str2[], size_t num);`
- Copy **up to num characters** from the `source` to `destination`, return the number of characters copied
`size_t strncpy(char destination[], const char source[], size_t num);`

Strings

Useful operations

- Compare **up to num characters** between `str1` and `str2` alphabetically, return 0 if equal, <0 if `str1` comes before and >0 if `str1` comes after `str2`
`int strncmp(const char str1[], const char str2[], size_t num);`
- Copy **up to num characters** from the `source` to `destination`, return the number of characters copied
`size_t strncpy(char destination[], const char source[], size_t num);`
- Concatenate **up to num characters** from `source` to `destination`, return the number of characters copied
`size_t strncat(char destination[], const char source[], size_t num);`

strlen

```
size_t strlen(const char str[]) {  
    unsigned long len = 0;  
    for (len = 0; str[len] != '\0'; len++);  
    return len;  
}
```

Use a for loop!

No statements needed!

What happens if string is not properly terminated?

Undefined behavior!

strcmp

```
int strcmp(const char str1[], const char str2[]) {  
    int cmp = 0;  
    for (size_t i = 0; ((cmp = str1[i] - str2[i]) != 0) && (str1[i] != '\0'); i++);  
    return cmp;  
}
```

strcpy

```
size_t strcpy(char destination[], const char source[]) {  
    size_t len;  
    for (len = 0; (destination[len] = source[len]) != '\0'); len++;  
    return len;  
}
```

Putting it all together

```
int main(void) {
    char first[20] = "Hello, world!";
    char second[20];
    printf("This string is %lu characters long\n", strlen(first);

    strncpy(second, first, 5);
    printf("first: %s\n", first);
    printf("second: %s\n", second);

    strcpy(second, first);
    printf("second after copying all characters: %s\n", second);

    return EXIT_SUCCESS;
}
```

Putting it all together

```
int main(void) {
    char first[20] = "Hello, world!";
    char second[20];
    printf("This string is %lu characters long\n", strlen(first);

    strncpy(second, first, 5);
    printf("first: %s\n", first);
    printf("second: %s\n", second);

    strcpy(second, first);
    printf("second after copying all characters: %s\n", second);

    return EXIT_SUCCESS;
}
```

```
./strings
This string is 13 characters long
first: Hello, world!
second: Hello
second after copying all characters: Hello, world!
```

Command Line Arguments

Command Line Arguments

Command Line Arguments

```
wc tests/hello.txt  
1  2 14 tests/hello.txt
```

Command Line Arguments

```
wc tests/hello.txt  
1  2 14 tests/hello.txt
```

- We want to use more than just the stdin

Command Line Arguments

```
wc tests/hello.txt  
1  2 14 tests/hello.txt
```

- We want to use more than just the stdin
- Or maybe we want to provide our encrypt/decrypt program a different key each time

Command Line Arguments

```
wc tests/hello.txt  
1  2 14 tests/hello.txt
```

- We want to use more than just the stdin
- Or maybe we want to provide our encrypt/decrypt program a different key each time

```
int main(int argc, char *argv[])
```

Command Line Arguments

```
int main(int argc, char *argv[])
```

Command Line Arguments

Number of
arguments

```
int main(int argc, char *argv[])
```

Command Line Arguments

Number of
arguments

Argument
vector

```
int main(int argc, char *argv[])
```

Command Line Arguments

Number of
arguments

Argument
vector

```
int main(int argc, char *argv[])
```

- `argc` — argument count

Command Line Arguments

Number of
arguments

Argument
vector

```
int main(int argc, char *argv[])
```

- `argc` — argument count
- `argv` — argument vector

Command Line Arguments

Number of
arguments

Argument
vector

```
int main(int argc, char *argv[])
```

- `argc` — argument count
- `argv` — argument vector
- `argv[i]` is a string

Command Line Arguments

Number of arguments
`int main(int argc,`
Argument vector
`char *argv[])`

- `argc` — argument count
- `argv` — argument vector
- `argv[i]` is a string
- `argv[0]` is the name of the program

Command Line Arguments

Number of arguments
`int main (int argc, Argument vector char *argv[])`

- `argc` — argument count
- `argv` — argument vector
- `argv[i]` is a string
- `argv[0]` is the name of the program
- Any additional arguments passed are saved in `argv[1], ..., argv[argc-1]`

Command Line Arguments

```
int main(int argc, char *argv[]) {  
    for (int i = 0; i < argc; i++) {  
        printf("%s\n", argv[i]);  
    }  
  
    return EXIT_SUCCESS  
}
```

Command Line Arguments

```
int main(int argc, char *argv[]) {  
    for (int i = 0; i < argc; i++) {  
        printf("%s\n", argv[i]);  
    }  
  
    return EXIT_SUCCESS  
}
```

```
./argv hello 1 a CMSC143  
./argv  
hello  
1  
a  
CMSC143
```