Function Pointers, Vulnerabilities

CS 2130: Computer Systems and Organization 1

Xinyao Yi Ph.D.
Assistant Professor

UNIVERSITY of VIRGINIA | ENGINEERING

Announcements

- Homework 10 due Monday
- Quiz 10 open today, due Sunday on Gradescope
- Final exam: 7pm Dec 12, Physics 338 (different room!)
  - Cumulative, see practice tests
  - Exam conflict form in email
- Remember to fill out course evaluations – 5 pts extra credit on final exam if completed by **Wednesday, Dec 10 at 5pm**!

Using write
pig latin example continued

1. Pig Latin Rules

1. **If a word starts with a consonant**, move the consonant (or consonant cluster) to the end and add **"ay"**
    - pig → **igpay**
    - smile → **ilesmay**
2. **If a word starts with a vowel**, add **"yay"** (or a similar syllable) to the end
3. **Non-letters** (punctuation, digits, symbols) remain unchanged

2. Design Strategy

The program is built using a **bottom-up** approach:

- First identify the smallest useful pieces
- Then combine them to form the full solution

Steps needed:

1. Read input text
2. Detect words
3. Convert each word to Pig Latin
4. Output converted text while preserving punctuation

3. Finding the First Vowel

Use `strpbrk()` to locate the first vowel in a word:

```
const char *findVowel(const char *word) {
   return strpbrk(word, "aeiouAEIOU");
}
```

If no vowel is found, the function should return the original word:

```
const char *findVowel(const char *word) {
   const char *ans = strpbrk(word, "aeiouAEIOU");
   if (ans == NULL)
      return word;
   return ans;
}
```

4. Converting a Word to Pig Latin

Steps:

1. Find the first vowel
2. Print the part from the vowel to the end
3. Print the initial consonants (the part before the vowel)
4. Add `"ay"`

Example implementation:

```
void showPig(const char *word) {
   const char *vowel = findVowel(word);
   printf("%s", vowel);                              // vowel → end
   fwrite(word, sizeof(char), vowel - word, stdout);   // consonant
part
   printf("ay");                                 // add suffix
}
```

5. Reading Input One Character at a Time

To correctly separate words and punctuation, input is read **one byte at a time** using:

```
read(0, buffer + index, 1);
```

- If the character is a **letter**, append it to the word buffer
- If it is **not a letter**, the current word is complete
- Convert the word (if any) and print the punctuation as-is

Use `isalpha()` from `<ctype.h>` to identify letters.

6. Detecting Word Boundaries

The program builds words based on sequences of alphabetic characters.

Logic:

1. If the current character is alphabetic:
   - append it, continue building the word
2. If it's not alphabetic:
   - finish the word
   - convert it
   - output the non-letter character
   - reset the buffer

7. Full Program Structure

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

    char buffer[1500];      // word buffer
    int index = 0;

    while (read(0, buffer + index, 1) == 1) {

        if (isalpha(buffer[index])) {
            index++;
        } else {
            char keepme = buffer[index];

            if (index > 0) {
                buffer[index] = '\0';
                showPig(buffer);
            }

            fwrite(&keepme, sizeof(char), 1, stdout);
            index = 0;
        }
    }
}
```

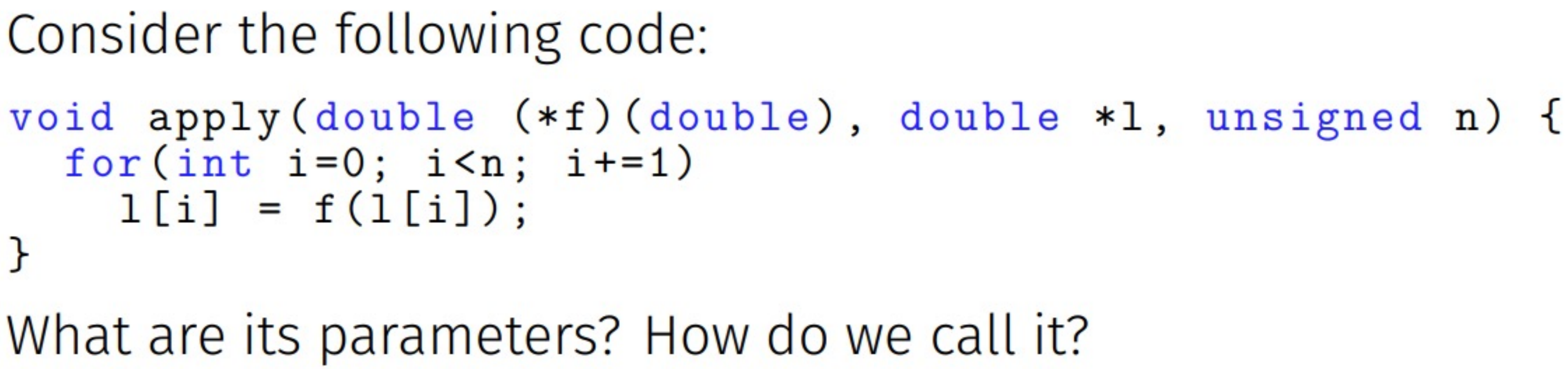
```c
#include <string.h>
#include <stdio.h>
#include <ctype.h>
#include <unistd.h>

const char *findVowel(const char *word) {
    const char *ans = strpbrk(word, "aeiouAEIOU");
    if (ans != NULL) return ans;
    return word;
}

void showPig(const char *word) {
    const char *vowel = findVowel(word);
    printf("%s", vowel); // ig
    fwrite(word, sizeof(char), vowel - word, stdout);// p
    printf("%s", "ay"); // ay
}

int main(int argc, const char *argv[]) {
    char buffer[1500]; // watch buffer overflow attack!
    int index = 0;
    while (read(0, buffer+index, 1) == 1) {
        if(isalpha(buffer[index])) {
            index += 1;
        } else {
            char keepme = buffer[index];
            if (index > 0) {
                buffer[index] = '\0';
                showPig(buffer);
            }
            fwrite(&keepme, sizeof(char), 1, stdout);
            index = 0;
        }
    }
}
```

Example Code



Consider the following code:

```
void apply(double (*f)(double), double *l, unsigned n) {
  for(int i=0; i<n; i+=1)
    l[i] = f(l[i]);
}
```

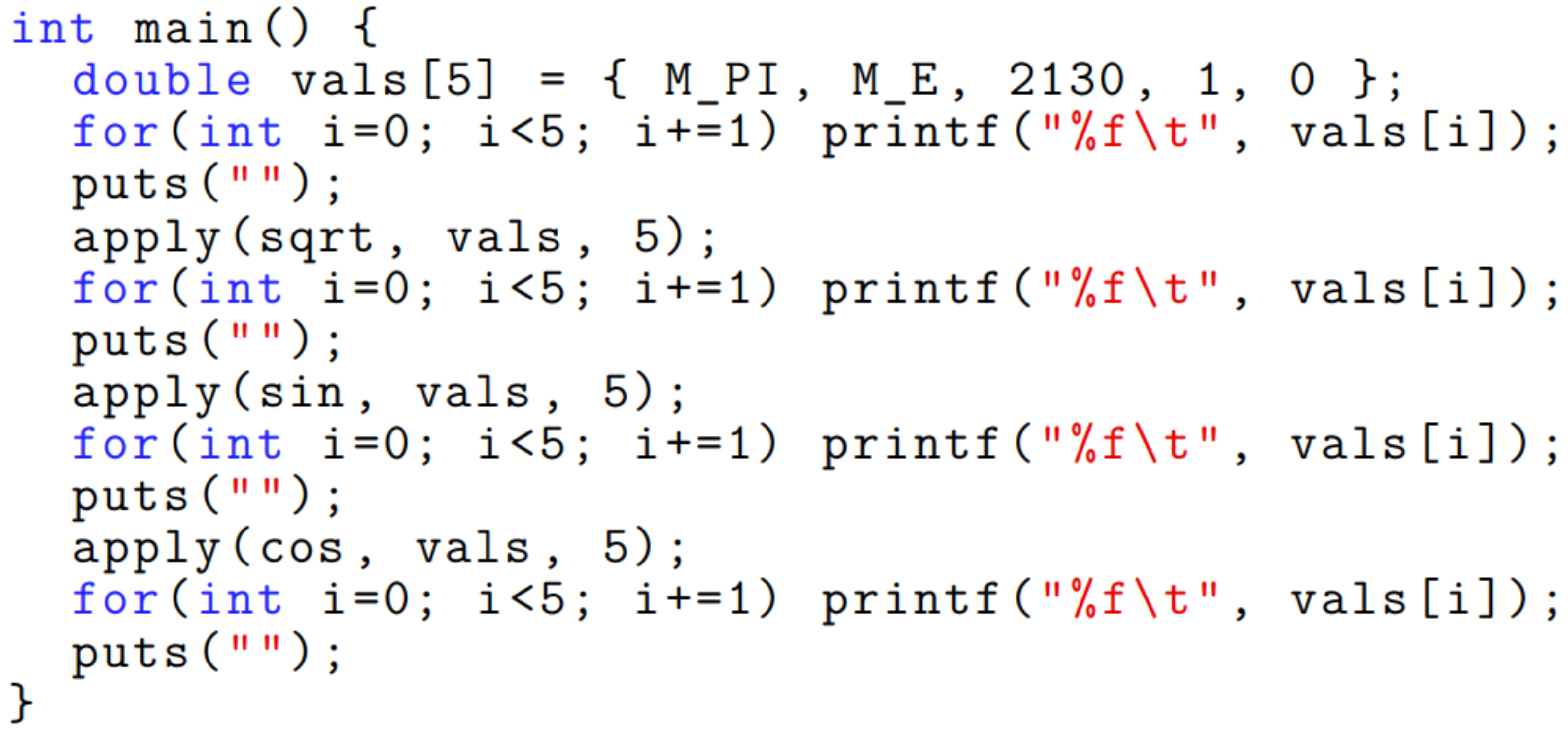
What are its parameters? How do we call it?

It takes three parameters:

1. A function pointer: `double (*f)(double)`
   This means *f is a pointer to a function that takes a double and returns a double*.
2. A pointer to a list of doubles: `double *l`.
3. The length of the list: `unsigned n`.

Inside the function, we simply loop from `0` to `n - 1`, and we replace each element `l[i]` with `f(l[i])`.
So this function applies **whatever function we pass in** to every element of the array.

Example Code



```
int main() {
  double vals[5] = { M_PI, M_E, 2130, 1, 0 };
  for(int i=0; i<5; i+=1) printf("%f\t", vals[i]);
  puts("");
  apply(sqrt, vals, 5);
  for(int i=0; i<5; i+=1) printf("%f\t", vals[i]);
  puts("");
  apply(sin, vals, 5);
  for(int i=0; i<5; i+=1) printf("%f\t", vals[i]);
  puts("");
  apply(cos, vals, 5);
  for(int i=0; i<5; i+=1) printf("%f\t", vals[i]);
  puts("");
}
```

We start with an array of five doubles, including `M_PI` and `M_E` from the math library.

We print the array so we can see the initial values.

Then we call:

```
apply(sqrt, vals, 5);
```

Notice the important detail:
We pass **`sqrt` without parentheses**.
That means we are *not calling* `sqrt` here. We are passing **the address of the function**.

`apply` will call it later for each element in the array.

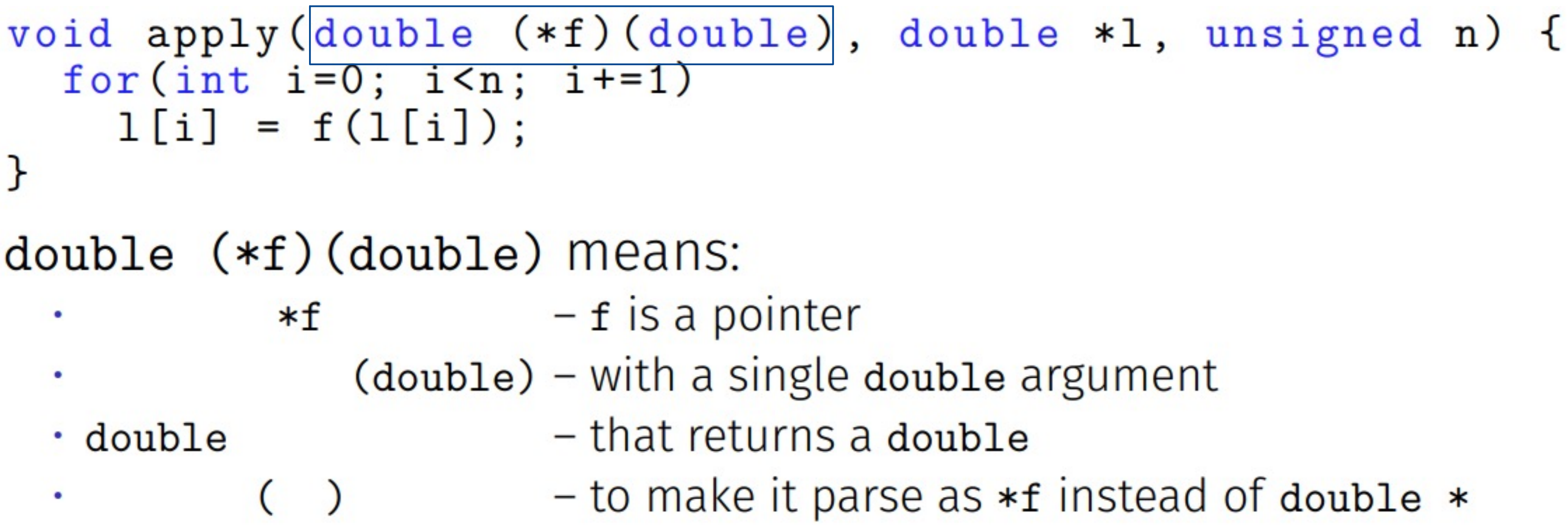
We repeat the same process with:

```
apply(sin, vals, 5);
apply(cos, vals, 5);
```

Each time, the array gets transformed in place using the function we passed in.

So the point of this example is:
we can write a generic function that applies any math function to an array, as long as it takes and returns a double.

Function Pointers



```
void apply(double (*f)(double), double *l, unsigned n) {
  for(int i=0; i<n; i+=1)
    l[i] = f(l[i]);
}
```

`double (*f)(double)` means:

- `*f` – `f` is a pointer
- `(double)` – with a single `double` argument
- `double` – that returns a `double`
- `(  )` – to make it parse as `*f` instead of `double *`

`double (*f)(double)` means:

- `*f` → f is a pointer
- `(double)` → it takes one double argument
- `double` → it returns a double
- the extra `( )` around `*f` are required because of operator precedence; otherwise the compiler would think we are declaring a function returning a pointer.

▵ Once we have a function pointer, calling it is easy.
We just write: `f(x);`

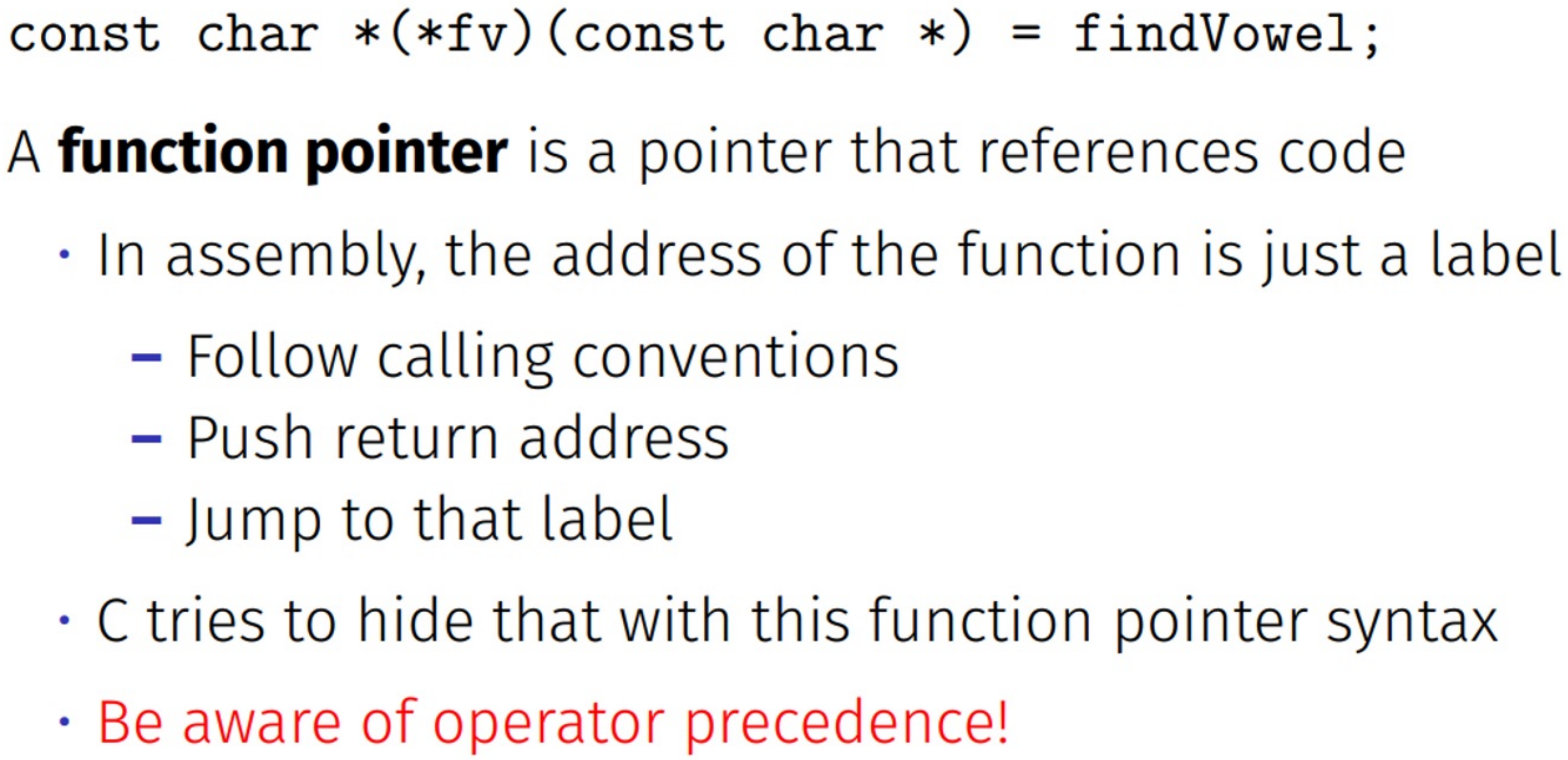
Exactly the same way we call a normal function.

▵ One more note: math functions like `sqrt`, `sin`, and `cos` are in the math library, so we must compile with:

```
clang file.c -lm
```

`-lm` means ‘link with the math library’.
C does not link it automatically.

Function Pointers



```
const char *(*fv)(const char *) = findVowel;
```

A **function pointer** is a pointer that references code

- In assembly, the address of the function is just a label
  - Follow calling conventions
  - Push return address
  - Jump to that label
- C tries to hide that with this function pointer syntax
- Be aware of operator precedence!

Conceptually, a function pointer is very simple: it is a pointer that references code.

At the assembly level, a function is just a label—an address in memory. Calling a function means jumping to that address, following the standard calling conventions, and returning when done.

C hides this behind syntax like `double (*f)(double)`, but the meaning is straightforward:
you store the address of a function in a variable, and later you jump to it when you call `f`.

There is one thing to be careful about: **operator precedence**.
This is why the parentheses around `*f` are required.

Function pointers exist because C does not have first-class functions or closures.
But they still let us pass behavior—like `sin`, `sqrt`, or `cos`—into generic code, which can be a very powerful idea.