C Retrospective

CS 211

Getting the Code

There's no C code for this lecture, but there is some Python code, which you can access in your Linux shell account:

```
% cd cs211
```

Getting the Code

There's no C code for this lecture, but there is some Python code, which you can access in your Linux shell account:

```
% cd cs211
% cp ~cs211/lec/13/tr_bench.py .     # get a copy
```

Getting the Code

There's no C code for this lecture, but there is some Python code, which you can access in your Linux shell account:

```
% cd cs211
% cp ~cs211/lec/13/tr_bench.py .     # get a copy
% ./tr_bench.py                      # run it
```

Getting the Code

There's no C code for this lecture, but there is some Python code, which you can access in your Linux shell account:

```
% cd cs211
% cp ~cs211/lec/13/tr_bench.py .     # get a copy
% ./tr_bench.py                      # run it
Length = 1024:
  tr1   0.019s
  tr2   0.027s
⋮                                     # watch it go
```

Getting the Code

There's no C code for this lecture, but there is some Python code, which you can access in your Linux shell account:

```
% cd cs211
% cp ~cs211/lec/13/tr_bench.py .     # get a copy
% ./tr_bench.py                      # run it
Length = 1024:
  tr1   0.019s
  tr2   0.027s
⋮                                    # watch it go
^C                                   # kill it
```

Getting the Code

There's no C code for this lecture, but there is some Python code, which you can access in your Linux shell account:

```
% cd cs211
% cp ~cs211/lec/13/tr_bench.py .     # get a copy
% ./tr_bench.py                      # run it
Length = 1024:
  tr1   0.019s
  tr2   0.027s
⋮                                     # watch it go
^C                                    # kill it
% emacs tr_bench.py                  # edit it
```

Road map

for

C Retrospective

Road map

for

translate()

C Retrospective

Road map

for

translate()

charseq_length()

Up next

for

translate()

charseq_length()

C Retrospective

What's C for?

What's C for?

Very particular things.

What's C for?

Very particular things.

Systems programming: providing efficient services for other programs

What's C for?

Very particular things.

Systems programming: providing efficient services for other programs

What's C for?

Very particular things.

Systems programming: providing efficient services for other programs

- When you need to control every detail of:
  - data layout
  - memory allocation
  - other low-level hardware stuff

What's C for?

Very particular things.

Systems programming: providing efficient services for other programs

- When you need to control every detail of:
  - data layout
  - memory allocation
  - other low-level hardware stuff
- When you can't afford (or get along with) a garbage collector

What's C for?

Very particular things.

Systems programming: providing efficient services for other programs

- When you need to control every detail of:
  - data layout
  - memory allocation
  - other low-level hardware stuff
- When you can't afford (or get along with) a garbage collector
- When you can't afford heap allocation! (embedded systems)

When should I use C?

You probably shouldn't.

When should I use C?

You probably shouldn't.

Stronger:

Don't use C.

When should I use C?

You probably shouldn't.

Stronger:

Don't use C.

Stronger still (& what I actually believe):

Using C when you could use a safer language* is engineering malpractice.

* nearly any other modern programming language

Why?

How a double-free bug in WhatsApp turns to RCE

14 minute read

In this blog post, I'm going to share about a double-free vulnerability that I discovered in WhatsApp for Android, and how I turned it into an RCE. I informed this to Facebook. Facebook acknowledged and patched it officially in WhatsApp version 2.19.244. Facebook reserved CVE-2019-11932 for this issue.

WhatsApp users, please do update to latest WhatsApp version (2.19.244 or above) to stay safe from this bug.

https://awakened1712.github.io/hacking/hacking-whatsapp-gif-rce/

Why?

GOT ROOT? —

Serious flaw that lurked in sudo for 9 years hands over root privileges

Flaw affecting selected sudo versions is easy for unprivileged users to exploit.

DAN GOODIN - 2/4/2020, 3:07 PM

https://arstechnica.com/information-technology/2020/02/serious-flaw-that-lurked-in-sudo-for-9-years-finally-gets-a-patch/

Why?

https://mobile.twitter.com/capitalist_void/status/1229900018821292032

Why did we spend five weeks learning it?

Why did we spend five weeks learning it?

- You need to understand it if you want to learn:
  - systems, generally (CS 213)
  - operating systems (CS 343)
  - C++ (many places and things) or Rust (growing quickly)

Why did we spend five weeks learning it?

- You need to understand it if you want to learn:
  - systems, generally (CS 213)
  - operating systems (CS 343)
  - C++ (many places and things) or Rust (growing quickly)
- It's an on-ramp to understanding lower-level machine stuff, which will help you:
  - understand and write more efficient code in nearly every language
  - understand compilers, machine architecture, data structures,...

Why did we spend five weeks learning it?

- You need to understand it if you want to learn:
  - systems, generally (CS 213)
  - operating systems (CS 343)
  - C++ (many places and things) or Rust (growing quickly)
- It's an on-ramp to understanding lower-level machine stuff, which will help you:
  - understand and write more efficient code in nearly every language
  - understand compilers, machine architecture, data structures,...
- Without some experience, it can be hard to understand why it's dangerous

Up next

for

translate()

charseq_length()

C Retrospective

Why is C “fast”?

Have you heard someone say
that a particular language

(*e.g.,* C, C++, Java, Python, JavaScript)

is fast or s l o w?

Some things that might affect performance

- The choice of algorithm
- How much work the basic operations of the language actually require
- How much the compiler knows about the meaning of the program (vs. how flexible it is)
- How well the programmer can understand and control the performance implications of what they write

Choice of algorithm

```
void tr0(char* s, const char* fr, const char* to)
{
    for (size_t i = 0; s[i]; ++i)
        s[i] = tr_char(s[i], fr, to);
}

void tr1(char* s, const char* fr, const char* to)
{
    for ( ; strlen(s) > 0; ++s)
        *s = tr_char(*s, fr, to);
}
```

Choice of algorithm

```
void tr0(char* s, const char* fr, const char* to)
{
    for (size_t i = 0; s[i]; ++i)
        s[i] = tr_char(s[i], fr, to);
}

void tr1(char* s, const char* fr, const char* to)
{
    for ( ; strlen(s) > 0; ++s)
        *s = tr_char(*s, fr, to);
}

void tr2(char* s, const char* fr, const char* to)
{
    for (size_t n = strlen(s); n > 0; --n, ++s)
        *s = tr_char(*s, fr, to);
}
```

Choose an algorithm

```
void tr3(char* s, const char* fr, const char* to)
{
    for (size_t i = 0; i < strlen(s); ++i)
        s[i] = tr_char(s[i], fr, to);
}

void tr4(char* s, const char* fr, const char* to)
{
    while ( (*s = tr_char(*s, fr, to)) )
        ++s;
}
```

Comparison to Java

```
void tr(char* s, const char* fr, const char* to)
{
    for (size_t i = 0; s[i]; ++i)
        s[i] = tr_char(s[i], fr, to);
}
```

```
static void tr(char[] s, char[] fr, char[] to) {
    for (int i = 0; i < s.length; ++i)
        s[i] = trChar(s[i], fr, to);
}
```

Comparison to Java

```
void tr(char* s, const char* fr, const char* to)
{
    for (size_t i = 0; s[i]; ++i)
        s[i] = tr_char(s[i], fr, to);
}


static String tr(String s, char[] fr, char[] to) {
    char[] buf = new char[s.length()];

    for (int i = 0; i < s.length(); ++i)
        buf[i] = trChar(s.charAt(i), fr, to);

    return new String(buf);
}
```

Comparison to Java

```
void tr(char* s, const char* fr, const char* to)
{
    for (size_t i = 0; s[i]; ++i)
        s[i] = tr_char(s[i], fr, to);
}


static String tr(CharSequence s,
                 char[] fr,
                 char[] to)
{
    char[] buf = new char[s.length()];
    for (int i = 0; i < buf.length; ++i)
        buf[i] = trChar(s.charAt(i), fr, to);
    return new String(buf);
}
```

Java teleology

```
public static class Tr {
    ...
    public String apply(CharSequence s) { ... }

    public String apply(String s) {
        char[] buf = s.toCharArray();
        for (int i = 0; i < buf.length; ++i)
            buf[i] = trChar(buf[i]);
        return new String(buf);
    }

    private char trChar(char c) { ... }
    private CharSet fr;
    private CharSet to;
}
```

Java teleology

```
public static class Tr {
    ...

    public Stream<Char> apply(Stream<Char> s) {
        return s.map(c -> trChar(c));
    }

    ...
}
```

Comparison to Python

```
void tr(char* s, const char* fr, const char* to)
{
    for (size_t i = 0; s[i]; ++i)
        s[i] = tr_char(s[i], fr, to);
}
```

```
def tr1(s: str, fr: str, to: str) -> str:
    result = ''
    for c in s:
        result += tr_char(c, fr, to)
    return result
```

Comparison to Python

```
void tr(char* s, const char* fr, const char* to)
{
    for (size_t i = 0; s[i]; ++i)
        s[i] = tr_char(s[i], fr, to);
}
```

```
def tr2(s: str, fr: str, to: str) -> str:
    result = ''
    for c in s:
        dummy = result  # forces next line to copy
        result += tr_char(c, fr, to)
    return result
```

Comparison to Python

```
void tr(char* s, const char* fr, const char* to)
{
    for (size_t i = 0; s[i]; ++i)
        s[i] = tr_char(s[i], fr, to);
}
```

```
def tr3(s: str, fr: str, to: str) -> str:
    buf = []
    for c in s:
        buf.append(tr_char(c, fr, to))
    return ''.join(buf)
```

Comparison to Python

```
void tr(char* s, const char* fr, const char* to)
{
    for (size_t i = 0; s[i]; ++i)
        s[i] = tr_char(s[i], fr, to);
}
```

```
def tr4(s: str, fr: str, to: str) -> str:
    return ''.join(tr_char(c, fr, to) for c in s)
```

Up next

for

translate()

charseq_length()

C Retrospective

Plus in Python (1/5)

```
typedef struct {
    size_t ref_count;
    PyType* ob_type;
} PyObject;
```

Plus in Python (1/5)

```
typedef struct {
    size_t ref_count;
    PyType* ob_type;
} PyObject;
```

```
typedef struct {
    size_t ref_count;
    PyType* ob_type;
    double value;
} PyFltObject;
```

Plus in Python (1/5)

```c
typedef struct {
    size_t ref_count;
    PyType* ob_type;
} PyObject;
```

```c
typedef struct {
    size_t ref_count;
    PyType* ob_type;
    size_t len;
    char data[0];
} PyStrObject;
```

Plus in Python (1/5)

```
typedef struct {
    size_t ref_count;
    PyType* ob_type;
} PyObject;
```

```
typedef struct {
    size_t ref_count;
    PyType* ob_type;
    ssize_t len;
    uint32_t digits[1];
} PyIntObject;
```

Plus in Python (2/5)

```
PyObject* op_plus(PyObject* a, PyObject* b)
{
    if (a->ob_type == &INT_TYPE &&
            b->ob_type == &INT_TYPE)
        return op_plus_int((PyIntObject*) a,
                           (PyIntObject*) b);









}
```

Plus in Python (2/5)

```
PyObject* op_plus(PyObject* a, PyObject* b)
{
    if (a->ob_type == &INT_TYPE &&
            b->ob_type == &INT_TYPE)
        return op_plus_int((PyIntObject*) a,
                           (PyIntObject*) b);

    if (a->ob_type == &STR_TYPE &&
            b->ob_type == &STR_TYPE)
        return op_plus_str((PyStrObject*) a,
                           (PyStrObject*) b);




}
```

Plus in Python (2/5)

```
PyObject* op_plus(PyObject* a, PyObject* b)
{
    if (a->ob_type == &INT_TYPE &&
            b->ob_type == &INT_TYPE)
        return op_plus_int((PyIntObject*) a,
                           (PyIntObject*) b);

    if (a->ob_type == &STR_TYPE &&
            b->ob_type == &STR_TYPE)
        return op_plus_str((PyStrObject*) a,
                           (PyStrObject*) b);

    // mixed floats and ints?

    ...
}
```

Plus in Python (3/5)

```
PyObject* op_plus_float(PyFltObject* a, PyFltObject* b)
{
    PyStrObject* result =
        py_malloc(sizeof(struct PyFltObject));

    result->ref_count = 1;
    result->ob_type = &FLOAT_TYPE;
    result->value = a->value + b->value;

    return (PyObject*) result;
}
```

Plus in Python (4/5)

```
PyObject* op_plus_str(PyStrObject* a, PyStrObject* b)
{
    size_t len = a->len + b->len;
    PyStrObject* result =
        py_malloc(sizeof(struct PyStrObject) + len);

    result->ref_count = 1;
    result->ob_type = &STR_TYPE;
    result->len = len;
    memcpy(result->data, a->data, a->len);
    memcpy(result->data + a->len, b->data, b->len);

    return (PyObject*) result;
}
```

Plus in Python (5/5)

```
PyObject* op_plus_int(PyIntObject* a, PyIntObject* b)
{
    if (a->len == 1 && b->len == 1 &&
            a->digits[0] <= PY_DIGIT_MAX - b->digits[0])
    {
        uint32_t sum = a->digits[0] + b->digits[0];
        if (sum < 256) return INTERNED_INT_TABLE[sum]

        PyIntObject* result =
            py_malloc(sizeof(struct PyIntObject));
        result->ref_count = 1;
        result->ob_type = &INT_TYPE;
        result->digits[0] = sum;

        return (PyObject*) result;
    } else
        return bignum_plus(a, b);
}
```

People call Python “dynamically typed”

What does this mean?

People call Python “dynamically typed”

What does this mean?

It means that the class of a variable can’t (always) be determined from the program source:

```
if random.randint(0, 2) == 0:
    x = 'hello'
else:
    x = 6
```

People call Python “dynamically typed”

What does this mean?

It means that the class of a variable can’t (always) be determined from the program source:

```
if random.randint(0, 2) == 0:
    x = 'hello'
else:
    x = 6
```

So is C dynamically typed?

What are dynamic types?

How I like to think of it:

- Variables (and expressions more generally) have static types—types known at compile time
- Objects have dynamic types—possibly not known until run time

What are dynamic types?

How I like to think of it:

- Variables (and expressions more generally) have static types—types known at compile time
- Objects have dynamic types—possibly not known until run time
- Type soundness: The static type is correct with respect to the dynamic type

What are dynamic types?

How I like to think of it:

- Variables (and expressions more generally) have static types—types known at compile time
- Objects have dynamic types—possibly not known until run time
- Type soundness: The static type is correct with respect to the dynamic type

In this view, Python has one static type `TPT` (The Python Type), and every Python class is a dynamic type.

Example of dynamic types in C

```
double sum(double* p, size_t len) { ... }

void g()
{
    double a1[] = {2, 3}, a2[] = {2, 3, 4, 5};
    sum(a1, sizeof a1 / sizeof *a1);
    sum(a2, sizeof a2 / sizeof *a2);
}
```

Example of dynamic types in C

```
double sum(double* p, size_t len) { ... }

void g()
{
    double a1[] = {2, 3}, a2[] = {2, 3, 4, 5};
    sum(a1, sizeof a1 / sizeof *a1);
    sum(a2, sizeof a2 / sizeof *a2);
}
```

- The static type of `p` is `double*`.

Example of dynamic types in C

```
double sum(double* p, size_t len) { ... }

void g()
{
    double a1[] = {2, 3}, a2[] = {2, 3, 4, 5};
    sum(a1, sizeof a1 / sizeof *a1);
    sum(a2, sizeof a2 / sizeof *a2);
}
```

- The static type of `p` is `double*`.
- The static and dynamic type of `a1` is `double[2]`
- The static and dynamic type of `a2` is `double[4]`

Example of dynamic types in C

```
double sum(double* p, size_t len) { ... }

void g()
{
    double a1[] = {2, 3}, a2[] = {2, 3, 4, 5};
    sum(a1, sizeof a1 / sizeof *a1);
    sum(a2, sizeof a2 / sizeof *a2);
}
```

- The static type of `p` is `double*`.
- The static and dynamic type of `a1` is `double[2]`
- The static and dynamic type of `a2` is `double[4]`
- When `sum(a1)` is active, the dynamic type of `p` is `double(*)[2]`
- When `sum(a2)` is active, the dynamic type of `p` is `double(*)[4]`

– The End –