C Wrap Up

CS 211

Winter 2020

Getting the Code

There's no C code for this lecture, but there is some Python code, which you can view directly here; or you can download it to your Linux shell and try it there:

```
% wget $URL211/lec/08cwrapup/tr_bench.py
⋮
% chmod +x tr_bench.py   # mark it as executable
% ./tr_bench.py          # run it
⋮
% emacs -nw tr_bench.py  # edit it
```

Road map

- `for`
- `main()`
- `translate()`
- `charset_length()`
- `::` and postfix &

What is C for?

What is C for?

Systems programming: providing efficient services for other programs

What is C for?

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 is C for?

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 is C for?

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)

`main()`

What’s wrong with this code?

```
if (expand_charset(argv[1]) == NULL ||
        expand_charset(argv[2]) == NULL) {
    fprintf(stderr, OOM_MESSAGE, argv[0]);
    return 2;
}

char* from = expand_charset(argv[1]);
char* to   = expand_charset(argv[2]);

if (charset_length(to) != charset_length(from)) {
    fprintf(stderr, LENGTH_MESSAGE, argv[0]);
    return 2;
}
```

What’s wrong with this code?

```
if (expand_charset(argv[1]) == NULL ||
        expand_charset(argv[2]) == NULL) {
    fprintf(stderr, OOM_MESSAGE, argv[0]);
    return 2;
}

char* from = expand_charset(argv[1]);
char* to   = expand_charset(argv[2]);

if (charset_length(to) != charset_length(from)) {
    fprintf(stderr, LENGTH_MESSAGE, argv[0]);
    return 2;
}
```

- First two calls to `expand_charset` leak!

What’s wrong with this code?

```
if (expand_charset(argv[1]) == NULL ||
        expand_charset(argv[2]) == NULL) {
    fprintf(stderr, OOM_MESSAGE, argv[0]);
    return 2;
}

char* from = expand_charset(argv[1]);
char* to   = expand_charset(argv[2]);

if (charset_length(to) != charset_length(from)) {
    fprintf(stderr, LENGTH_MESSAGE, argv[0]);
    return 2;
}
```

- First two calls to `expand_charset` leak!
- But `from` and `to` might be `NULL` anyway

What’s wrong with this code?

```
if (expand_charset(argv[1]) == NULL ||
        expand_charset(argv[2]) == NULL) {
    fprintf(stderr, OOM_MESSAGE, argv[0]);
    return 2;
}

char* from = expand_charset(argv[1]);
char* to   = expand_charset(argv[2]);

if (charset_length(to) != charset_length(from)) {
    fprintf(stderr, LENGTH_MESSAGE, argv[0]);
    return 2;
}
```

- First two calls to `expand_charset` leak!
- But `from` and `to` might be `NULL` anyway
- Applying `charset_length` to a literal charset?

What’s wrong with this code?

```
if (expand_charset(argv[1]) == NULL ||
        expand_charset(argv[2]) == NULL) {
    fprintf(stderr, OOM_MESSAGE, argv[0]);
    return 2;
}

char* from = expand_charset(argv[1]);
char* to   = expand_charset(argv[2]);

if (charset_length(to) != charset_length(from)) {
    fprintf(stderr, LENGTH_MESSAGE, argv[0]);
    return 2;
}
```

- First two calls to `expand_charset` leak!
- But `from` and `to` might be `NULL` anyway
- Applying `charset_length` to a literal charset?
- Leaks `from` and `to` if lengths don’t match

How do we fix this code?

```
char* from = expand_charset(argv[1]);
if (from == NULL) {
    fprintf(stderr, OOM_MESSAGE, argv[0]);
    return 2;
}

char* to = expand_charset(argv[2]);
if (to == NULL) {
    fprintf(stderr, OOM_MESSAGE, argv[0]);
    return 2;
}

if (strlen(to) != strlen(from)) {
    fprintf(stderr, LENGTH_MESSAGE, argv[0]);
    return 2;
}
```

How do we fix this code?

```
char* from = expand_charset(argv[1]);
if (from == NULL) {
    fprintf(stderr, OOM_MESSAGE, argv[0]);
    return 2;
}

char* to = expand_charset(argv[2]);
if (to == NULL)
    free(from);
    fprintf(stderr, OOM_MESSAGE, argv[0]);
    return 2;
}

if (strlen(to) != strlen(from)) {
    fprintf(stderr, LENGTH_MESSAGE, argv[0]);
    return 2;
}
```

Now it’s correct…

```c
char* from = expand_charset(argv[1]);
if (from == NULL) {
    fprintf(stderr, OOM_MESSAGE, argv[0]);
    return 2;
}

char* to = expand_charset(argv[2]);
if (to == NULL) {
    free(from);
    fprintf(stderr, OOM_MESSAGE, argv[0]);
    return 2;
}

if (strlen(to) != strlen(from)) {
    free(from);
    free(to);
    fprintf(stderr, LENGTH_MESSAGE, argv[0]);
    return 2;
}
```

Cleaning up (1/2)

```c
char* from = expand_charset(argv[1]);
char* to = expand_charset(argv[2]);

if (from == NULL || to == NULL) {
    free(to);
    free(from);
    fprintf(stderr, OOM_MESSAGE, argv[0]);
    return 2;
}

if (strlen(to) != strlen(from)) {
    free(from);
    free(to);
    fprintf(stderr, LENGTH_MESSAGE, argv[0]);
    return 2;
}
```

Cleaning up (2/2)

```c
char* from = expand_charset(argv[1]);
char* to = expand_charset(argv[2]);

if (from == NULL || to == NULL) {
    fprintf(stderr, OOM_MESSAGE, argv[0]);
    goto cleanup;
}

if (strlen(to) != strlen(from)) {
    fprintf(stderr, LENGTH_MESSAGE, argv[0]);
    goto cleanup;
}

// ...

cleanup:
    free(from);
    free(to);
    return 2;
```

A goto chain

(No space on slide for `fprintf`s.)

```c
char* from = expand_charset(argv[1]);
if (from == NULL) goto from_failed;

char* to = expand_charset(argv[2]);
if (to == NULL) goto to_failed;

if (strlen(to) != strlen(from)) goto len_failed;

// ...

len_failed:
    free(to);
to_failed:
    free(from);
from_failed:
    return 2;
```

C++ foreshadowing: object lifecycles

In C++, each `struct` we define can have a *destructor*: A function we write that runs every time an instance “goes away.”

C++ foreshadowing: object lifecycles

In C++, each `struct` we define can have a *destructor*: A function we write that runs every time an instance “goes away.”

(For now, “goes away” means it goes out of scope.)

C++ foreshadowing: object lifecycles

In C++, each `struct` we define can have a *destructor*: A function we write that runs every time an instance “goes away.”

(For now, “goes away” means it goes out of scope.)

In fact, we will be able to customize:

- construction
- copying (into a new instance)
- assignment (into an existing instance)
- moving! (into a new or existing instance)
- destruction

`translate()`

Why is C “fast”?

Have you heard people say that a particular language
(*e.g.,* C, C++, Java, Python, JavaScript)
is fast or slow?

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

```c
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);
}
```

```python
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

```c
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);
}
```

```python
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

```c
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);
}
```

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

charset_length()

Plus in Python (1/5)

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

Plus in Python (1/5)

```c
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)

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

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:

```python
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

```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]`

Two C++ concepts

- pass-by-reference (postfix &)
- member functions (`::`)

C is completely pass-by-value

```
void f(int x, int* p) { ... }
```

In C, every variable names its own object:

- x stands for 4 bytes, not overlapping with any other variable's object
- p stands for 8 bytes, not overlapping with any other variable's object

C is completely pass-by-value

```
void f(int x, int* p) { ... }
```

In C, every variable names its own object:

- x stands for 4 bytes, not overlapping with any other variable's object
- p stands for 8 bytes, not overlapping with any other variable's object

C *simulates* pass-by-reference by letting you pass pointers, but you are still passing a value (a pointer value)

C++ has pass-by-reference as well

```
void f(int x, int* p, int& r) { ... }
```

- x and p are as in C
- r refers to some other, existing int object

C++ has pass-by-reference as well

```
void f(int x, int* p, int& r) { ... }
```

- `x` and `p` are as in C
- `r` refers to some other, existing `int` object

Use `r` like an ordinary `int`—no need to dereference

C++ reference example: inc

```
void inc_p(int* p)
{
    *p += 1;
}

void inc(int& r)
{
    r += 1;
}
```

C++ reference example: `inc`

```
void inc_p(int* p)
{
    *p += 1;
}

void inc(int& r)
{
    r += 1;
}

void h()
{
    int x = 0;
    inc_p(&x);
    inc(x);
}
```

C++ reference example: swap

```
void swap_p(int* p, int* q) { ... }

void swap(int& r, int& s)
{
    int temp = r;
    r = s;
    s = temp;
}
```

C++ reference example: swap

```
void swap_p(int* p, int* q) { ... }

void swap(int& r, int& s)
{
    int temp = r;
    r = s;
    s = temp;
}

void h()
{
    int x = 1, y = 2;
    swap(x, y);
}
```

C++ references *desugar* to pointers

- Replace every variable declaration `T& x` with `T* xp`.

C++ references *desugar* to pointers

- Replace every variable declaration `T& x` with `T* xp`.
- Replace every initialization `T& x = e;` with `T* xp = &e;`.

C++ references *desugar* to pointers

- Replace every variable declaration `T& x` with `T* xp`.
- Replace every initialization `T& x = e;` with `T* xp = &e;`.
- Replace every use of `x` with `*xp`.

C++ references *desugar* to pointers

- Replace every variable declaration `T& x` with `T* xp`.
- Replace every initialization `T& x = e;` with `T* xp = &e;`.
- Replace every use of `x` with `*xp`.

```
void swap(int& r, int& s)
{
    int temp = r;
    r = s;
    s = temp;
}
```

C++ references *desugar* to pointers

- Replace every variable declaration `T& x` with `T* xp`.
- Replace every initialization `T& x = e;` with `T* xp = &e;`.
- Replace every use of `x` with `*xp`.

```
void swap(int& r, int& s)
{
    int temp = r;
    r = s;
    s = temp;
}
// becomes
void swap(int* rp, int* sp)
{
    int temp = *rp;
    *rp = *s;
    *s = temp;
}
```

C++ references *desugar* to pointers

- Replace every variable declaration `T& x` with `T* xp`.
- Replace every initialization `T& x = e;` with `T* xp = &e;`.
- Replace every use of `x` with `*xp`.

```
void swap(int& r, int& s)         swap(x, y);
{
    int temp = r;
    r = s;
    s = temp;
}
// becomes
void swap(int* rp, int* sp)       swap(&x, &y);
{
    int temp = *rp;
    *rp = *s;
    *s = temp;
}
```

C++ member functions

```
struct Posn
{
    double x, y;
    double dist(Posn const&) const;
};
```

C++ member functions

```cpp
struct Posn
{
    double x, y;
    double dist(Posn const&) const;
};

double Posn::dist(Posn const& other) const
{
    double dx = this->x - other.x;
    double dy = this->y - other.y;
    return sqrt(dx * dx + dy * dy);
}
```

C++ member functions

```
struct Posn
{
    double x, y;
    double dist(Posn const&) const;
};

double Posn::dist(Posn const& other) const
{
    double dx = this->x - other.x;
    double dy = this->y - other.y;
    return sqrt(dx * dx + dy * dy);
}

Posn p1{3, 4};
Posn p2{8, -8};
double d = p1.dist(p2);
```

– Next time: the C++ object lifecycle —