C Bounds Non-Checking

```c
int main (void) {
    int x = 9;
    char s[4];
    gets(s);
    printf("s is: %s\n", s);
    printf("x is: %d\n", x);
}
```

User input: `abcdefghijkl`
```
s is: abcdefghijkl
x is: 9
```

User input: `abcdefghijklm`
```
s is: abcdefghijklmn
x is: 1828716553 = 0x6d000009
```

User input: `abcdefghijkln`
```
s is: abcdefghijkln
x is: 1845493769 = 0x6e000009
```

User input: `aaa...` (a few thousand characters)
crashes shell
```
Note: your results may vary (depending on machine, compiler, what else is running, time of day, etc.). This is what makes C fun!
```

What does this kind of mistake look like in a popular server?

Reasons Not to Use C

- No bounds checking
  - Programs are vulnerable to buffer overflow attacks
- No automatic memory management
  - Lots of extra work to manage memory manually
  - Mistakes lead to hard to find and fix bugs
- No support for data abstraction, objects, exceptions

So, why would anyone use C today?

Good Reasons to Use C

- Legacy Code: Linux, apache, etc.
- Simple, small
  - Embedded systems, often only have a C compiler
- Low-level abstractions
  - Performance: typically 20-30x faster than interpreted Python
  - Sometimes we need to manipulate machine state directly: device drivers
- Lots of experience
  - We know pitfalls for C programming
  - Tools available that catch them
What are those arrows really?

s = “hello”

Pointers

• In Python, an object reference is really just an address in memory

Pointers in C

• Addresses in memory
• Programs can manipulate addresses directly

&expr Evaluates to the address of the location expr evaluates to

*expr Evaluates to the value stored in the address expr evaluates to

Rvalues and Lvalues

What does = really mean?

```c
int f (void) {
    int s = 1;
    int t = 1;
    t = s;
    t = 2;
}
```

Parameter Passing in C

• Actual parameters are rvalues

```c
void swap (int a, int b) {
    int tmp = b; b = a; a = tmp;
}
```
Parameter Passing in C

```c
void swap (int *a, int *b) {
    int tmp = *b; *b = *a; *a = tmp;
}

int main (void) {
    int i = 3;
    int j = 4;
    swap (&i, &j);
    ...  // The value of &i is passed, which is the address of i
}
```

Is it possible to define swap in Python?

Beware!

```c
int *value (void) {
    int i = 3;
    return &i;
}

void callme (void) {
    int *ip;
    ip = value ();
    printf (*ip == %d
    callme ();
    printf (*ip == %d
}
```

But it could really be anything!

Manipulating Addresses

```c
char s[6];
s[0] = 'h';
s[1] = 'e';
s[2] = 'l';
s[3] = 'l';
s[4] = 'o';
s[5] = '\0';
printf ("s: %s\n", s);
```

s: hello

Obfuscating C

```c
char s[6];
*s = 'h';
*(s + 1) = 'e';
2[s] = 'l';
3[s] = 'l';
5[s] = '\0';
printf ("s: %s\n", s);
```

s: hello

Fun with Pointer Arithmetic

```c
int match (char *s, char *t) {
    int count = 0;
    while (*s == *t) { count++; s++; t++; }
    return count;
}

int main (void) {
    char s1[6] = "hello";
    char s2[6] = "hohoh";
    printf ("match: %d\n", match (s1, s2));
    printf ("match: %d\n", match (s1, s2 + 2));
    printf ("match: %d\n", match (s2[1], &s2[3]));
}
```

match: 1
match: 3
match: 2

Condensing match

```c
int match (char *s, char *t) {
    int count = 0;
    while (*s == *t) { count++; s++; t++; }
    return count;
}
```

```c
s++ evaluates to s_pre, but changes the value of s
Hence, C++ has the same value as C, but has unpleasant side effects.
```
Quiz
• What does s = s++; do?

It is undefined!

If your C programming contains it, a correct interpretation of your program could make s = $s_{\text{pre}} + 1$, s = 37, or blow up the computer.

Type Checking in C
• Java: only allow programs the compiler can prove are type safe
  Exception: run-time type errors for downcasts and array element stores.
• C: trust the programmer. If she really wants to compare apples and oranges, let her.
• Python: don’t trust the programmer or compiler – check everything at runtime.

Type Checking
int main (void) {
char *s = (char *) 3;
printf ("s: %s", s);
}

Windows XP (SP 2)

Python’s List Implementation
(A Whirlwind Tour)
http://svn.python.org/view/python/trunk/Objects/listobject.c

listobject.c
/* List object implementation */
#include <Python.h>
#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h> /* For size_t */

/* Ensure ob_item has room for at least numnew elements, and set ob_size to numnew. If numnew > ob_size on entry, the content of the new slots at exit is undefined heap trash; it’s the caller’s responsibility to override them with some value. The number of allocated elements may grow, shrink, or stay the same. Failure is impossible if numnew <= self.allocated on entry, although that partly relies on an assumption that the system malloc() never fails when passed a number of bytes <= the number of bytes last allocated (the C standard doesn’t guarantee this, but it’s hard to imagine a realistic implementation where it wouldn’t be true). Note that self->ob_item may change, and even if numnew is less than ob_size on entry. */
/* static */
int list_resize(PyListObject *self, Py_ssize_t numnew) {
...
listobject.h

typedef struct

PyObject_VAR_HEAD
/* Vector of pointers to list elements. list[0] is ob_item[0], etc. */
PyObject **ob_item;
/* ob_item contains space for 'allocated' elements. The number
* currently in use is ob_size.
* Invariants:
* Now we know our answer to PS1 #6 (Python's list
implementation is continuous) is correct!
*/
Py_ssize_t allocated;
} PyListObject;

http://svn.python.org/view/python/trunk/Include/listobject.h

Now we can be (slightly) more confident that our
answer to PS1 #4 (append is O(1)) was correct!

Append

int
PyList_Append(PyObject *op, PyObject *newitem)
{
if (PyList_Check(op) && (newitem != NULL))
return app1((PyListObject *)op, newitem);
PyErr_BadInternalCall();
return -1;
}

app1

static int
app1(PyListObject *self, PyObject *v)
{
Py_ssize_t n = PyList_GET_SIZE(self);
assert (v != NULL);
if (n == INT_MAX) {
PyErr_SetString(PyExc_OverflowError,
"cannot add more objects to list");
return -1;
}
if (list_resize(self, n+1) == -1)
return -1;
Py_INCREF(v);
PyList_SET_ITEM(self, n, v);
return 0;
}

Set Item (in listobject.h)

/* Macro, trading safety for speed */
define PyList_SET_ITEM(op, i, v) \((\{\{PyListObject *\}(op))\rightarrow\text{ob_item}[n] = (v)\))

Macro: text replacement, not procedure calls.
PyList_SET_ITEM(self, n, v);
((PyListObject *)(self))\rightarrow\text{ob_item}[n] = (v)\)

Set Item (in listobject.h)

((PyListObject *)(self))\rightarrow\text{ob_item}[n] = (v)\)

/* Vector of pointers to list elements. list[0] is ob_item[0], etc. */
PyObject **ob_item;
/* ob_item contains space for 'allocated' elements. The number
* currently in use is ob_size.
* Invariants:
*/
Py_ssize_t allocated;
} PyListObject;

Now we can be (slightly) more confident that our
answer to PS1 #4 (append is O(1)) was correct!
list_resize

static int
list_resize(PyListObject *self, Py_ssize_t newsize)
{
    /* This over-allocates proportional to the list size, making room for additional growth. The over-allocation is mild, but is enough to give linear-time amortized behavior over a long sequence of appends()... */

Monday’s class will look at list_resize

Charge

• This is complicated, difficult code
  – We could (but won’t) spend the rest of the semester without understanding it all completely
• Now we trust PS1 #4
  – But...only amortized O(1) – some appends will be worse than average!
  – We shouldn’t trust Python’s developers’ comments
• Exam 1 is out now, due Monday
  – Work alone, read rules on first page carefully