CS216: Program and Data Representation
University of Virginia Computer Science
Spring 2006 David Evans

Lecture 10: *&!%[]++

http://www.cs.virginia.edu/cs216

C Bounds Non-Checking

```
int main (void) {
  int x = 9;
  char s[4];

  gets(s);
  printf ("s is: %s\n", s);
  printf ("x is: %d\n", x);
}
```

Note: your results may vary (depending on machine, compiler, what else is running, time of day, etc.). This is what makes C fun!

```
> gcc -o bounds bounds.c
> bounds
abcdefghijkl
s is: abcdefghijkl
x is: 9
> bounds
abcdefghijklm
s is: abcdefghijklmn
x is: 1828716553
> bounds
abcdefghijkln
s is: abcdefghijkln
x is: 1845493769
> bounds
aaa... [a few thousand characters]
crashes shell
```

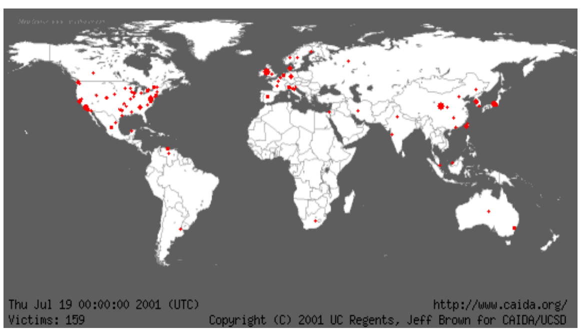
(User input)

= 0x6d000009

= 0x6e000009

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

Code Red



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

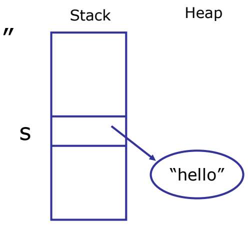
So, why would anyone use C today?

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

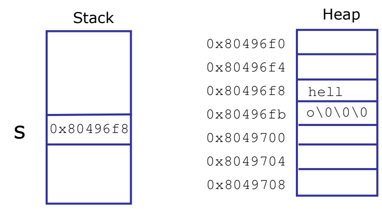
What are those arrows really?

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

&*%&@#*!

```
int f (void) {
  int s = 1;
  int t = 1;
  int *ps = &s;
  int **pps = &ps;
  int *pt = &t;
                  s == 1, t == 1
  **pps = 2;      s == 2, t == 1
  pt = ps;
  *pt = 3;        s == 3, t == 1
  t = s;          s == 3, t == 3
}
```


Rvalues and Lvalues

What does **=** really mean?

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

left side of = is an “lvalue”
it evaluates to a location (address)!
right side of = is an “rvalue”
it evaluates to a value

There is an implicit ***** when a variable is used as an rvalue!

Parameter Passing in C

- Actual parameters are rvalues

```
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 (3) is passed, not its location!
swap does nothing

Parameter Passing in 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!

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

void callme (void)
{
  int x = 35;
}

int main (void) {
  int *ip;
  ip = value ();
  printf ("*ip == %d\n", *ip);
  callme ();
  printf ("*ip == %d\n", *ip);
}
```

*ip == 3

*ip == 35

But it could really be anything!

Manipulating Addresses

```
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);
```

expr1[expr2] in C is just syntactic sugar for *(expr1 + expr2)

s: hello

Obfuscating C

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

s: hello

Fun with Pointer Arithmetic

```
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 (s2, s2 + 2));
  printf ("match: %d\n", match (&s2[1], &s2[3]));
}
```

The **\0** is invisible!

&s2[1]
➔ &(*(s2 + 1))
➔ s2 + 1

match: 1
match: 3
match: 2

Condensing match

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

```
int match (char *s, char *t) {
  char *os = s;
  while (*s++ == *t++);
  return s – os - 1;
}
```

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!

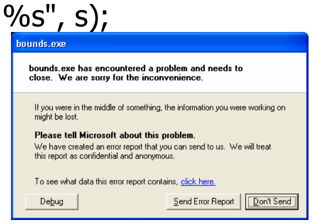
If your C programming contains it, a correct interpretation of your program could make $s = s_{pre} + 1$, $s = 37$, or blow up the computer.

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

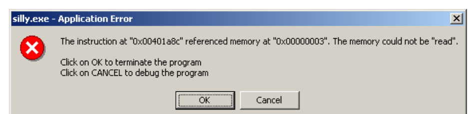
```
int main (void) {
  char *s = (char *) 3;
  printf ("s: %s", s);
}
```



Windows XP (SP 2)

Type Checking

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



Windows 2000

(earlier versions of Windows would just crash the whole machine)

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"

#ifdef STDC_HEADERS
#include <stddef.h>
#else
#include <sys/types.h> /* For size
#endif

/* Ensure ob_item has room for at least newsize elements, and set ob_size to newsize.  If
newsize > ob_size on entry, the content of the new slots at exit is undefined heap trash; it's the
caller's responsiblity to overwrite them with sane values. The number of allocated elements may
grow, shrink, or stay the same. Failure is impossible if newsize <= self.allocated on entry,
although that partly relies on an assumption that the system realloc() 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 realloc implementation where it wouldn't be true).
Note that self->ob_item may change, and even if newsize is less than ob_size on entry.
 */
static int
list_resize(PyListObject *self, Py_ssize_t newsize)
{
   ...
```

We'll get back to this…
but you should be convinced that you are lucky to have been using a language with automatic memory management so far!

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:
    *
    *
    *
    * Items must normally not be NULL, except during construction when
    * the list is not yet visible outside the function that builds it.
    */
   Py_ssize_t allocated;
} PyListObject;
```

Now we know our answer to PS1 #6 (Python's list implementation is continuous) is correct!

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

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;
}
```

Checks there is enough space to add 1 more element (and resizes if necessary)

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;
}
```

Complicated macro for memory management: needs to keep track of how many references there are to object v

Set Item (in listobject.h)

```
/* Macro, trading safety for speed */
#define PyList_SET_ITEM(op, i, v) \
   (((PyListObject *)(op))->ob_item[i] = (v))
```

Macro: text replacement, not procedure calls.

PyList_SET_ITEM(self, n, v);

⇩

(((PyListObject *)(self))->ob_item[n] = (v))

Set Item (in listobject.h)

(((PyListObject *)(self))->ob_item[n] = (v))

```
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: ...   */
   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