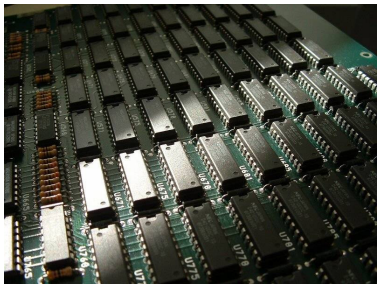
CS216: Program and Data Representation
University of Virginia Computer Science
Spring 2006 David Evans

Lecture 11: Managing Memory



http://www.informatics.bangor.ac.uk/~emlynl/photos/lab2/chips1.jpg

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

Levels of Abstraction

Python: a → "hello"

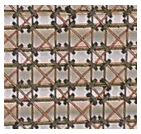
C: a: 0x80496f8

0x80496f8	h
0x80496f9	e
0x80496fa	l
0x80496fb	l
0x80496fc	o
0x80496fd	\0

Digital Abstraction:
01101000011001010110110001101100011011000110111100000000

Physical Memory

Vacuum Tube



Magnetic Core

Capacitor (store charge) DRAM

Memory in C

Stack
- Managed by compiler automatically
- Lifetime is determined by program scope
  - Cannot outlive procedure return

Heap
- Managed by programmer explicitly
- Lifetime is controlled by programmer
  - Lives until freed by program

Static/Dynamic Allocation

- Static: space required is known before program starts (at "compile time")
- Dynamic: space required is not known before the program starts
  - Can be placed on either the stack or the heap

malloc

```
void *malloc (size_t size)
```

Returns an untyped pointer (can point to anything)

Size in bytes

Malloc returns an address that is the location of at least size bytes of previously unused memory, and reserves that space.
Or, returns **null** is there isn't enough space.

Memory Allocators

		Lifetime	
		Scoped	Unlimited
Size	Known	local variable declarations	global, static variable declarations
	Unknown	alloca	malloc

alloca – like malloc, but on the stack, not heap (rarely used)

Malloc Example

char *s = (char *) malloc (sizeof(*s) * n)

type cast
malloc returns void *, cast tells compiler we are using it as a char *

sizeof operator
Takes a type or expression, evaluates to the number of bytes to store it

String Concatenation

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>

int main (int argc, char **argv)
{
  int i = 0;
  char *result = (char *) malloc (sizeof (*result));
  *result = '\0';

  while (i < argc)
    {
      char *s = (char *) malloc (sizeof (*s)
                                 * (strlen (result) + strlen (argv[i]) + 1));
      strcpy (s, result);
      strcat (s, argv[i]);
      result = s;
      i++;
    }
  printf ("Concatenation: %s\n", result);
  return 0;
}
```

Some serious problems with this, we'll discuss soon…

Concating Strings

```
char *s = (char *)
  malloc (sizeof (*s)
          * (strlen (result)
             + strlen (argv[i]) + 1));
strcpy (s, result);
strcat (s, argv[i]);
```

Why is the parameter to malloc what it is?

string functions

int strlen(char *s) – returns number of chars in *s* (not counting null terminator)

char *strcpy(char *s1, const char *s2) – Copies the string pointed to by *s2* (including the terminating null byte) into the space pointed to by *s1*. If copying takes place between objects that overlap, the behavior is undefined.

char *strcat(char *s1, const char * s2) - appends a copy of the string pointed to by *s2* (including the terminating null byte) to the end of the string pointed to by *s1*. The initial byte of *s2* overwrites the null byte at the end of *s1*. If copying takes place between objects that overlap, the behavior is undefined.

Concating Strings

```
char *s = (char *)
  malloc (sizeof (*s)
          * (strlen (result)
             + strlen (argv[i]) + 1));
strcpy (s, result);
strcat (s, argv[i]);
```

We need enough space for s to hold all the chars in result, all the chars in argv[i], and the null terminator.

Memory Lifetimes

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>

int main (int argc, char **argv)
{
  int i = 0;
  char *result = (char *) malloc (sizeof (*result));
  *result = '\0';

  while (i < argc)
    {
      char *s = (char *) malloc (sizeof (*s)
                                 * (strlen (result) + strlen (argv[i]) + 1));
      strcpy (s, result);
      strcat (s, argv[i]);
      result = s;
      i++;
    }
  printf ("Concatenation: %s\n", result);
  return 0;
}
```

When is space allocated by malloc reclaimed?

Reclaiming Storage

- Storage allocated by malloc is reserved forever
- Give it back by passing it to free

void free(void *ptr);

The *free*() function shall cause the space pointed to by *ptr* to be deallocated; that is, made available for further allocation. If *ptr* is a null pointer, no action shall occur. Otherwise, if the argument does not match a pointer earlier returned by *malloc*(), … (or a few other allocators) function, or if the space has been deallocated by a call to *free*(), the behavior is undefined.

http://www.opengroup.org/onlinepubs/009695399/functions/free.html

Plugging Memory Leaks

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>

int main (int argc, char **argv)
{
  int i = 0;
  char *result = (char *) malloc (sizeof (*result));
  *result = '\0';

  while (i < argc)
    {
      char *s = (char *) malloc (sizeof (*s)
                                 * (strlen (result) + strlen (argv[i]) + 1));
      strcpy (s, result);
      strcat (s, argv[i]);
      result = s;
      i++;
    }
  printf ("Concatenation: %s\n", result);
  return 0;
}
```

free (result);

Memory Leak

- There is no reference to allocated storage
  - It can never be reached
  - It can never be reclaimed
- Losing references
  - Variable goes out of scope
  - Variable reassigned

References

```
char *result = (char *) malloc (sizeof (*result));
…
while (i < argc)
  {
    char *s = (char *) malloc (…);
    strcpy (s, result);
    strcat (s, argv[i]);
    result = s;
    i++;
  }
```

} scope of s is closed – should we free(s) first?

No! result now references same storage

References

```
char *result = (char *) malloc (sizeof (*result));
…
while (i < argc)
  {
    char *s = (char *) malloc (…);
    strcpy (s, result);
    strcat (s, argv[i]);
    result = s;
    i++;
  }
```

after this assignment, no way to reach storage result previously pointed to

C vs. Python

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>

int main (int argc, char **argv)
{
  int i = 0;
  char *result = (char *) malloc (sizeof (*result));
  *result = '\0';

  while (i < argc)
    {
      char *s = (char *) malloc (sizeof (*s)
                                 * (strlen (result) + strlen (argv[i]) + 1));
      strcpy (s, result);
      free (result);
      strcat (s, argv[i]);
      result = s;
      i++;
    }
  printf ("Concatenation: %s\n", result);
  return 0;
}
```

```
import sys

res = ""
for arg in sys.argv:
    res += arg
print res
```


C vs. Python

```
int main (int argc, char **argv)
{
  char result[6] = "hello";

  while (strlen(result) < 1000000000:)
    {
      char *s = (char *) malloc (sizeof (*s) * (2 * strlen (result) + 1));
      strcpy (s, result);
      strcat (s, result);
      free (result);
      result = s;
    }
  return 0;
}
```

```
res = "hello"
while len(res) < 1000000000:
    res += res
```

```
> time python concat.py
4.46u 11.70s 9:40.04
```


Python's List Implementation

http://svn.python.org/view/python/trunk/Objects/listobject.c

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;
}
```


list_resize

```
static int
list_resize(PyListObject *self, Py_ssize_t newsize)
{
   PyObject **items;
   size_t new_allocated;
   Py_ssize_t allocated = self->allocated;

   /* Bypass realloc() when a previous overallocation is large enough … */
   if (allocated >= newsize && newsize >= (allocated >> 1)) {
      assert(self->ob_item != NULL || newsize == 0);
      self->ob_size = newsize;
      return 0;
   }

   …
```

expr1 >> *expr2*
Value is value of *expr1* with bits moved right value of *expr2* times

010000 => 001000

x >> 1 is similar to (x / 2)


```
…
/* 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() in the presence of a
 * poorly-performing system realloc().
 * The growth pattern is:  0, 4, 8, 16, 25, 35, 46, 58, 72, 88, ...
 */
new_allocated = (newsize >> 3) + (newsize < 9 ? 3 : 6) + newsize;
if (newsize == 0)
   new_allocated = 0;
items = self->ob_item;
if (new_allocated <= ((~(size_t)0) / sizeof(PyObject *)))
   PyMem_RESIZE(items, PyObject *, new_allocated);
else
   items = NULL;
if (items == NULL) {
   PyErr_NoMemory();
   return -1;
}
self->ob_item = items;
self->ob_size = newsize;
self->allocated = new_allocated;
return 0;
```

PyMem_Resize

```
#define PyMem_RESIZE(p, type, n) \
      ( (p) = (type *) PyMem_REALLOC((p), (n) * sizeof(type)) )
```

/* PyMem_MALLOC(0) means malloc(1). Some systems would return NULL for malloc(0), which would be treated as an error. Some platforms would return a pointer with no memory behind it, which would break pymalloc. To solve these problems, allocate an extra byte. */

```
#define PyMem_MALLOC(n)         malloc((n) ? (n) : 1)
#define PyMem_REALLOC(p, n)     realloc((p), (n) ? (n) : 1)
```

PyMem_RESIZE(items, PyObject *, new_allocated)
=>
items = (PyObject *)
realloc (items, new_allocated * sizeof(PyObject *))

realloc

void *realloc(void *ptr, size_t size);
The *realloc()* function changes the size of the memory object pointed to by *ptr* to the size specified by *size*. The contents of the object will remain unchanged up to the lesser of the new and old sizes. If the new size of the memory object would require movement of the object, the space for the previous instantiation of the object is freed. If the new size is larger, the contents of the newly allocated portion of the object are unspecified. If *size* is 0 and *ptr* is not a null pointer, the object pointed to is freed. If the space cannot be allocated, the object remains unchanged.

http://www.opengroup.org/onlinepubs/007908799/xsh/realloc.html

realloc is risky

```
int main (int argc, char **argv)
{
  char *s = (char *) malloc (sizeof(*s) * 6);
  char *t;

  t = s;
  strcpy (s, "hello");
  printf ("s = %s [%p] [%p]\n" , s, s, t);
  s = (char *) realloc (s, sizeof(*s) * 7);
  printf ("s = %s [%p] [%p]\n" , s, s, t);
  strcpy (s, "cheers");
  printf ("s = %s, t = %s\n", s, t);
}
```

20

s = hello [20a80] [20a80]
s = hello [20e88] [20a80]
s = cheers, t = hello

s = hello [20a80] [20a80]
s = hello [20a80] [20a80]
s = cheers, t = cheers

realloc

void *realloc(void *ptr, size_t size);
The *realloc()* function changes the size of the memory object pointed to by *ptr* to the size specified by *size*. The contents of the object will remain unchanged up to the lesser of the new and old sizes. **If the new size of the memory object would require movement of the object, the space for the previous instantiation of the object is freed.** If the new size is larger, …

After a realloc, any use of the original location of ptr has undefined behavior!

Running time of realloc

- Time to reserve new space: $O(1)$
- Time to copy old data into new space:
  - $O(n)$ where n is the size of the old data

So how can Python's list append be $O(1)$?


```
static int
list_resize(PyListObject *self, Py_ssize_t newsize)
{
   PyObject **items;
   size_t new_allocated;
   Py_ssize_t allocated = self->allocated;
   …
/* 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() in the presence of a
 * poorly-performing system realloc(). The growth pattern is:  0, 4, 8, 16, 25, …*/
new_allocated = (newsize >> 3) + (newsize < 9 ? 3 : 6) + newsize;
if (newsize == 0) new_allocated = 0;
items = self->ob_item;
if (new_allocated <= ((~(size_t)0) / sizeof(PyObject *)))
   PyMem_RESIZE(items, PyObject *, new_allocated);
else
   items = NULL;
if (items == NULL) { PyErr_NoMemory(); return -1; }
self->ob_item = items;
self->ob_size = newsize;
self->allocated = new_allocated;
return 0;
```

When to grow

```
new_allocated = (newsize >> 3)
    + (newsize < 9 ? 3 : 6) + newsize;
```

- Approximately: newsize / 8 + newsize
- So, we have to do $O(n)$ work every approximately every n/8 calls to append
- Amortized: $O(n / (n / 8)) = O(1)$
- But…some calls will be more expensive than others

Charge

- Section: practice with pointers
- Reading: Chapter 10
- Wednesday:
  - Reference Counting
  - Garbage Collection
  - PS4 out