Crash Course in C

Dynamic Allocation

ingolia@mit.edu

Dynamic Allocation of Memory

Dynamic allocation

- Requests memory from the system
- Size can be calculated at run–time
- Persists beyond a single function activation

Dynamic Allocation of Memory

`malloc`

The malloc function is used to allocate memory

```
void *malloc(size_t sz);
```

`sz` This argument specifies the size, in bytes, of the memory to be allocated.

The function returns an untyped pointer to the allocated memory, or NULL if the memory cannot be allocated.

Dynamic Allocation of Memory

sizeof

Always use sizeof to determine the size of data.

At compile time, a sizeof expression is expanded into an integer.

```
int main(void)
{
  printf("|double| = %d\n",
         sizeof(double));
  return 0;
}
```

```
  printf("|double| = %d\n", 8);
```

```
|double| = 8
```

Dynamic Allocation of Memory

Size specified at run–time

```
void f(char *str)
{
  char buf[strlen(str) + 1];

  strcpy(buf, str);
```

Unfortunately, we cannot do this.

The compiler must know how big the buffer is.

We could make a "very large" buffer.

However, this wastes space, and someone will eventually give us a string that is too large.

Dynamic Allocation of Memory

Size specified at run–time

```
void f(char *str)
{
   int   buflen;
   char *buf;

   buflen = strlen(str) + 1;
   buf = malloc(sizeof(char)*buflen);

   strcpy(buf, str);
```

The size of the buffer is specified at run–time, so any string can be accomodated.

Note the use of sizeof for the size of a character.

Dynamic Allocation of Memory

Persistence of Allocated Memory

```
char *get_name(void)
{
  char name[64];
  strcpy(name, "Nicholas Ingolia");
  return name;
}
```

The storage associated with name disappears when the function returns.

Thus, the returned pointer is useless and dangerous.

Dynamic Allocation of Memory

Persistence of Allocated Memory

```
char *lookup(char *key)
{
  static char val[256];
  int         idx = keyidx(key);

  strcpy(val, keyvals[idx].val);
  return val;
}
```

This is an acceptable solution...

Except, the next call to lookup overwrites the old val, which is a problem.

Dynamic Allocation of Memory

Persistence of Allocated Memory

```
char *lookup(char *key)
{
  char *val;
  int   idx = keyidx(key);

  val = malloc(sizeof(char)
          * (strlen(vals[idx]) + 1);
  strcpy(val, vals[idx]);
  return val;
}
```

Val will persist until explicitly deallocated.

A new val will be created on every call to lookup.

Dynamic Allocation of Memory

Allocating Strings

```
char *strdup(const char *str)
{
  int   len = strlen(str);
  char *newstr;

  newstr = malloc(sizeof(char)
                  * (len + 1));
  strcpy(newstr, str);
  return newstr;
}
```

Strings need an extra char to hold the 0 terminator.

Dynamic Allocation of Memory

Allocating Strings

The strdup function copies strings with allocation.

```
char *strdup(const char *str);
```

`str` is the string to be copied.

The function returns a freshly–allocated copy of the string, or NULL if it cannot allocate memory.

We did not address the error case in our function.

Dynamic Allocation of Memory

Allocating Arrays

```
double *normalize(double *d, int nd)
{
  double dnorm[nd];
  int    i;

  for (i = 0; i < nd; i++) {
    ...

double *normalize(double *d, int nd)
{
  double *dnorm;

  dnorm = malloc(sizeof(double)*nd);
  ...
```

Dynamic Allocation of Memory

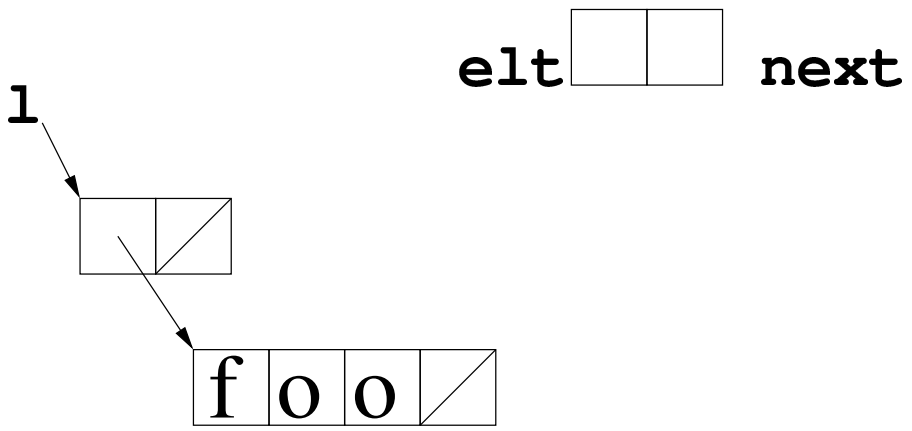
Allocation of Data Structures

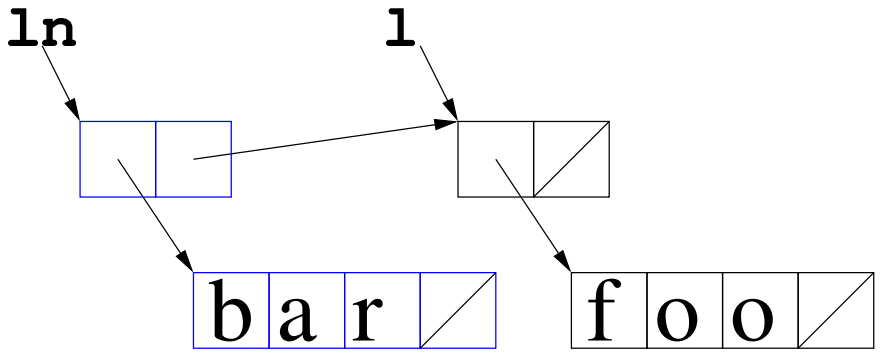
```
typedef foo_struct {
   char *a;
   char *b;
   int   c;
} foo;

foo *foo_new(const char *a,
                const char *b, int c)
{
   foo *f;
   f = malloc(sizeof(foo));
   f->a = strdup(a);
   f->b = strdup(b);
   f->c = c;
   return f;
}
```

Dynamic Allocation of Memory

Allocation of Data Structures

```
llist *llist_add(llist *l,
                 const char *elt)
{
  llist *ln = malloc(sizeof(llist));

  ln->elt = strdup(elt);
  ln->next = l;
  return ln;
}
```





Dynamic Allocation of Memory

Persistence of Allocated Memory

Allocated memory persists

Thus, it is necessary to release unneeded memory

Failure to release unallocated memory causes the program to consume system resources.

Free all "temporary" allocated memory; ensure that memory is freed before all possible returns.

Note which functions return pointers to new memory

Free allocated memory "inside" data structures

Dynamic Allocation of Memory

Releasing Allocated Memory

The free function releases allocated memory

`void free(void *p)`

`p` A pointer to allocated memory

It is very important that free only be called on a pointer returned by malloc or strdup.

Free may be called only once on such a pointer.

Neither the pointer nor any other derived from it may be used after it is freed.

Dynamic Allocation of Memory

Releasing Allocated Memory

```
void foo(const char *str)
{
  char *buf;

  buf = strdup(str);
  munge(buf);
  display(buf);
  free(buf);
}
```

Dynamic Allocation of Memory

Releasing Data Structures

```
void llist_free(llist *l)
{
   llist *next;

   next = l->next;
   free(l->elt);
   free(l);
   if (next != NULL) {
      llist_free(next);
   }
}
```

We use next because we cannot use l after freeing it.

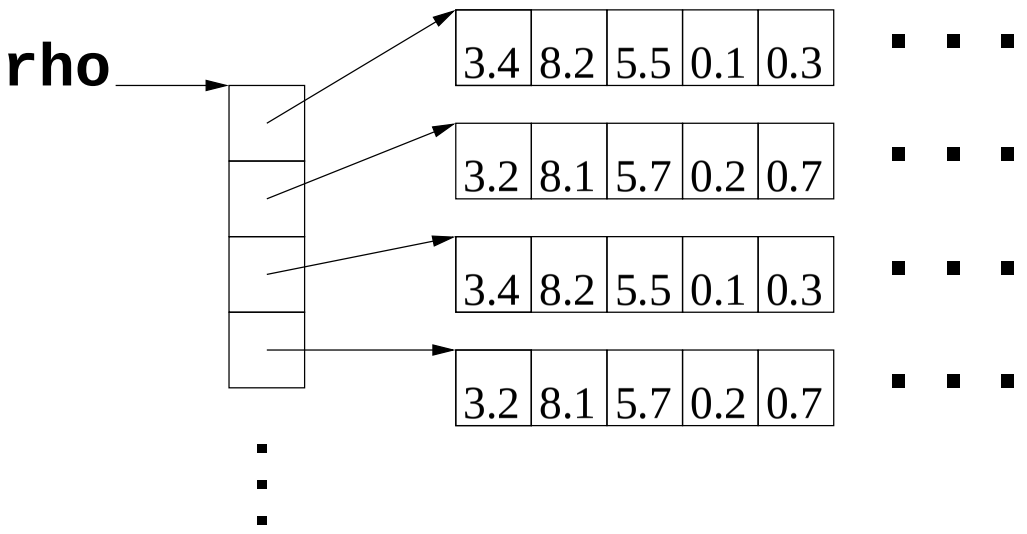
We must release the elt field, because we allocated it.

Dynamic Allocation of Memory

Multidimensional Arrays

A multidimensional array in C is an array of arrays.

```
double rho[NROWS][NCOLS];
rho[17][37] = 0.666;
```



Dynamical Allocation of Memory

Multidimensional Arrays

```
double *new_2d(int lx, int ly)
{
  double **dx;
  int      x;

  dx = malloc(sizeof(double *)*lx);
  for (x = 0; x < lx; x++) {
    dx[x] = malloc(sizeof(double)
                   * ly);
  }

  return dx;
}
```

Dynamic Allocation of Memory

Releasing Multidimensional Arrays

```
void free_2d(double **d,
             int nx, int ny)
{
  int x;

  for (x = 0; x < nx; x++) {
    free(d[x]);
  }

  free(d);
}
```

All allocated memory must be freed.

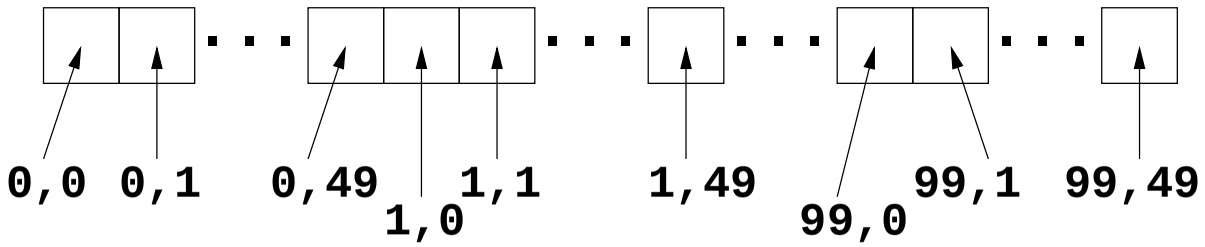
Dynamic Allocation of Memory

Alternate Multidimensional Arrays

Rather than using an array of arrays, it is possible to use a single array of size **(xy)**

```
double rho[NROWS * NCOLS];
rho[(17 * NCOLS) + 37] = 0.666;
```

For 100 rows, 50 colums:



This requires only a single block of memory, but the actual index must be computed.

Modular Programming in C

Advantages of Modular Programming

Programs ought to be divided into smaller modules

Modules provide conceptual boundaries

Individual modules can be tested, debugged, and reused

Modules assist in allocation management

Modular Programming in C

Interfaces and Implementations

A C program can be broken into multiple source files

Using one source file per module helps enforce module boundaries

Header files specify the interface to a module
  Interfaces include function and data type declarations

Source files provide the implementation of a module
  Implementations provide definitions of functions promised by the module in the interface.

Modular Programming in C

Header Files

```
/* llist.h */
#if !defined(_llist_h)
#define _llist_h 1
struct llist_struct;
typedef struct llist_struct llist;

llist *llist_new(void);
void llist_free(llist *);
llist *llist_insert(llist *,
                    const char *);
llist *llist_remove(llist *,
                    const char *);
int llist_contains(const llist *,
                   const char *);

#endif
```

Modular Programming in C

Headers

The C processor can be used to protect against multiple inclusions of a header file

```
#if !defined(_header_h)
#define _header_h 1
  ...
#endif
```

Modular Programming in C

Private Structure Definitions

llist.h

```
struct llist_struct;
typedef struct llist_struct llist;
```

llist.c

```
struct llist_struct {
   char  *elt;
   llist *next;
};
```

Pointers to an llist can be used anywhere.

The fields of an llist can be accessed only in within the llist.c source file implementing an llist.

Modular Programming in C

Immutability

A constant pointer in a function declaration is a promise not to change the argument.

```
char *strdup(const char *str);
```

The calling function knows that the str passed to strdup will not be changed.

C enforces this agreement, though it can be overridden

```
str[0] = 'a';
```

Modular Programming in C

Compilation and Linking

C source must be compiled and linked to produce a program

Compilation translates C into quasi–executable code but does not resolve function names

```
int a, b;

a = b + 2;
foo(a);
```

```
lwz r9, r31 + 12
addi r0, r9, 2
stw r0, r31 + 8
lwz r3, r31 + 8
bl foo
```

Modular Programming in C

Compiling and Linking

A compiled object is a collection of functions awaiting completion by linking

Modular Programming in C

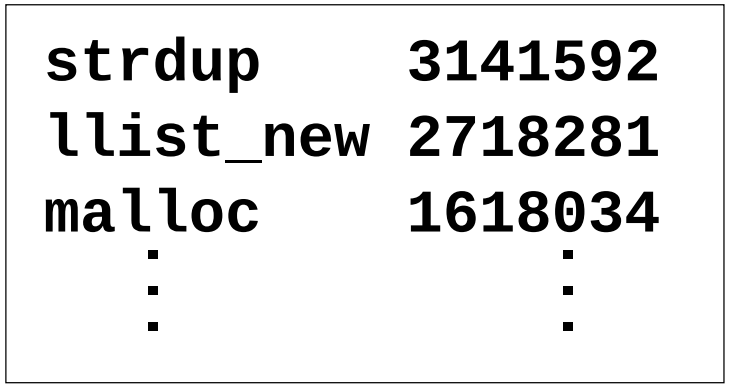
Compilation and Linking

Linking assembles objects into a program

The linker builds a table of functions and global variables provided by various objects

The table is used to "complete" the objects

bl *llist_new* → **bl 2718281**



Modular Programming in C

Compiling and Linking

`gcc -c prog.c` Compile prog.c into prog.o

`gcc -c llist.c` Compile llist.c into llist.o

`gcc prog.o llist.o -o prog`

Link prog.o and llist.o into a program prog.

`gcc prog.c llist.c -o prog`

Compile prog.c and llist.c into temporary objects and link them into prog