CS113: Lecture 9

Topics:

- Dynamic Allocation
- Dynamic Data Structures

What's wrong with this?

```
char *big_array( char fill )
{
   char a[1000];
   for( i = 0; i < 1000; i++ )
      a[i] = fill;
   return a;
}

void main()
{
   char *b;
   b = big_array( 'z' );
}
```

A fixed version

```
char *big_array( char fill )
{
   char *a;
   a = (char *)malloc( sizeof( char ) * 1000 );
   if( a == NULL ) return a;
   for( i = 0; i < 1000; i++ )
      a[i] = fill;
   return a;
}

void main()
{
   char *b;
   b = big_array( 'z' );
   /* do something with b */
   free( b );
}
```

Notice:

- `malloc` used instead of static allocation.

  `malloc` expects an `unsigned int` as argument and has return type `void *`

- Result of `malloc` is *casted* into `char *`.

- Notice `free` at end of program. Every pointer obtained through `malloc` should be "freed" using `free` before program termination.

More flexible version

```
char *big_array( char fill, int size )
{
   char *a;
   a = (char *)malloc( sizeof( char ) * size );
   if( a == NULL ) return a;
   for( i = 0; i < size; i++ )
      a[i] = fill;
   return a;
}
```

What good is this malloc thing?

- Suppose you want to write a program which stores names (of people) along with their addresses.

- One way to implement would be to define a struct holding all of this information, and then define an array of structs at the beginning of the program:

```
struct person_struct
{
   char name[30];
   char address[60];
};

struct person_struct  database[6000];
```

- Difficulties:

  Need to know ahead of time the maximum size of the database.

  If the maximum size is 6000 and only 50 people stored, much memory is wasted.

A naive implementation

```
struct person_struct
{
   char name[30];
   char address[60];
};

struct database_struct
{
   struct person_struct people[100];
   int num_people;
};

void add_person( struct database_struct *db,
                 char *pname, char *add )
{
   strcpy( (db->people[db->num_people]).name, pname );
   strcpy( (db->people[db->num_people]).address, add );
   (db->num_people)++;
}

void main()
{
   struct database_struct db;
   db.num_people = 0;

   add_person( &db, "Chen, Hubie",
      "1234 Street Ave., Ithaca, NY 14853" );
}
```

Linked list implementation

```
#include <stdio.h>

#define NAME_SIZE 30
#define ADD_SIZE  60

struct list_item_struct
{
   char name[NAME_SIZE];
   char address[ADD_SIZE];
   struct list_item_struct *next;
};

struct database_struct
{
  struct list_item_struct *first;
};

typedef struct list_item_struct list_item;
typedef struct database_struct database;

int initialize_db( database *db )
{
  db->first = NULL;
}
```

Linked list 2

```
int add_to_db( database *db, char *name, char *address )
{
  list_item *new_item_ptr;
  new_item_ptr = (list_item *)malloc(sizeof( list_item ));

  if( new_item_ptr == NULL ) return -1;
  if( strlen( name ) >= NAME_SIZE ||
      strlen( address ) >= ADD_SIZE ) return -1;

  strcpy( new_item_ptr->name, name );
  strcpy( new_item_ptr->address, address );
  new_item_ptr->next = db->first;

  db->first = new_item_ptr;
  return 0;
}
```

Linked list 3

```
void print_item( list_item l )
{
  printf( "Name: %s\n", l.name );
  printf( "Address: %s\n\n", l.address );
}

void print_db( database db )
{
  list_item *l = db.first;
  while( l != NULL )
    {
      print_item( *l );
      l = l->next;
    }
}


void main()
{
  database db;
  initialize_db( &db );
  add_to_db( &db, "Hubie Chen", "1234 Street Road" );
  add_to_db( &db, "Homer Simpson",
               "742 Evergreen Terrace" );

  print_db( db );
}
```

Linked list 4

```
void remove_from_db( database *db, list_item *item )
{
  list_item *l = db->first;
  if( l == NULL ) return;
  if( l == item )
    {
      db->first = l->next;
      free( l );
      return;
    }
  while( l->next != NULL && l->next != item )
    l = l->next;
  if( l->next == item )
    {
      l->next = item->next;
      free( item );
    }
}
```