Low-Level C Programming

Memory map
Pointers
Arrays
Structures
Memory Map

- Binaries load at 0x20000 by default
- Stack start set by binary when started
- Stack grows downwards
- You will need one stack for each task
**Binaries**

- **Code**
  - All functions of your program
- **Static data**
  - Global variables
  - Static variables
  - Initialized
  - Uninitialized/zeroed
- **All loaded into memory at startup**
  - Code
    - 0x0000_0000
  - Stack
    - Z
    - I
    - Code
      - 0x0002_0000
    - 0x7FFF_FFFF
**Variables**

- All variables have an address
  - **Stack**
    - "auto" variables
    - allocated when function called
    - varies with each function call
  - **Global variables**
    - z/i areas
    - fixed for entire execution of program

```
0x0002_0000
0x0000_0000
```
Functions

- All functions have an address
  - code area
  - fixed for entire execution of program
# Heap

- For dynamic memory allocation
  - Allocated at need
  - Explicit functions
    - “malloc”
- Not mandatory in your project!
Pointers

Basic pointer
**Pointer**

- Variable that contains address
  - Of another variable
    - Global
    - Stack
  - Of dynamically allocated memory
- Must be given something to point at
  - Declaring a pointer does not do this
  - Always initialize a pointer!
Basic pointer

- int a   // variable
- int b
- int *p   // pointer to int

- a = 7   // a=7
- p = &a  // p points to a
- b = *p  // b=a=7
- *p = 5  // a=5, b=7
Pointer to pointer

- int a   // variable
- int b
- int *p  // pointer to int
- int **q // ptr to ptr to int

- a = 7   // a=7
- p = &a  // p points to a
- q = &p  // q points to p
- b = *p  // a=7, b=7
- *p = 5  // a=5, b=7
- **q = 3 // a=3, b=7
### Pointer to pointer

- `int a` // variable
- `int b`
- `int *p` // pointer to int
- `int **q` // ptr to ptr to int

- `p = &a` // p points to a
- `q = &p` // q points to p
- `*q = &b` // repoint p to b
- `*p = 1` // b=1
Pointers & types

- Pointer points to a particular type
- Can only legally address objects of that type
  - Assign "char*" to "int*" is an error
- Each type has a size
  - char = 1 byte
  - short int = 2 bytes
  - int = 4 bytes (on MIPS32)
  - etc.
Pointers & types

- Special pointer type: **void**
  - Compatible with all types
  - Use with care!
  - Cannot be dereferenced!
    - What is the type of the object? None!
Arrays

Identical things in a row
Basic arrays

- `int a[6] // array`
- `int b`

- `a[0] = 1 // assign element`
- `b = a[2] // read element`
- `a[b] = a[5] // variable index`
- `a[17] = 2 // danger!`

Notes
- zero-based indexing
- no bounds checking
- higher index=higher address
Array initializers

- Initialize & size array

- `int a[]={1,1,2,3,5,8,13}
  - 7 elements
  - equivalent size: a[7]`
Arrays = pointers

- In C, array == pointer
  - Array variable = address of first element of array

- int a[6] // array
- int b
- int *p

- p = a
- p = &a[0] // equivalent
  // p = &b would point to b
Arrays = pointers

- Index from pointer

- `int a[6] // array`
- `int b`
- `int *p`

- `p = a`
- `b = (*p)[5] // ( ) needed!`
- `b = *p     // b=a[0]`
Arrays = pointers

- Indexing = arithmetic
  - $a[x] == *(a+x)$

- int a[6] // array
- int b
- int *p, *q
- q = p+3 // q=&a[3]
- q++ // step to next
- b = *q // b=a[4]
- b = *(p+1) // b=a[1]
- b = q-p // pointer difference
Pointer Arithmetic

- Unit of operation: size of type
  - int *p
  - char *q
  - p++
  - p++
  - q++
  - q++
sizeof

- sizeof operator defined in C
- Gives the size in #of bytes of a type
- Used to step pointers
Arrays of pointers

- `int *r[6] // array of ptr`
- `int *p`
- `int b`
- `int a`
- `p = &a`
- `r[3] = &b`
- `*r[3] = 7 // b=7`
- `r[4] = p`
Pointer to array

- int (*p)[6] // ptr to array
- int a[6]
- p = &a  // correct
Strings

- String = array of char
  - Special initializer syntax
  - No other support in C
  - Null-terminated

- char s[] = “hello”
- char *p
- p = s
- *p = ’w’  // s[0]=w
- *p = s[4]  // s[0]=o
Array of Strings

- Array of char*
  - Each string separate
  - Stored in constant memory
  - Not contiguous

- char *s[]={“hello”,“there”,”a”}
Arrays as parameters

- Passed as pointer, not value
- No check for size of array

```c
void foo(char a[]) {
    int i;
    for(i=0;i<MAX_SIZE;i++)
        {...a[i]...}
}
```
Arrays as parameters

- Pointer is alternative, equivalent

```c
void foo(char *a, int b)
{
    int i;
    for(i=0;i<b;i++)
    {
        ...a[i]...
        ...*a++...
    }
}
```
Iterating over Strings

- Compute the length of a string:
  ✷ Watch for the zero termination!

```c
int length (const char * s)
{
    char *p=s;
    while( *p != '\0' )
        p++;
    return p-s;
}
```
Local Arrays

- Allocated on the stack
- Not cleared before use
  - Always initialize
  - Contains garbage data when created
- Deallocated when function returns
  - Do not return a local pointer!
Structures

Organizing data into records
Structures in C

- Collection of related data
  - Use typedef for clarity

- Syntax:
  ```c
  typedef struct TAG {     // TAG is optional but good form
    TYPE element;
    TYPE2 element2;
    ...
  } TAG_T;

  TAG_T x; // declare variable
  ```
In Memory

- Elements stored in order
- Potentially padded
  * To align evenly in memory
- Towards higher addresses

```c
struct bar {
    char   a;   // 1 byte
    // invisible 3-byte padding
    int    b;   // 4 bytes
    int   *c;   // 4 bytes
} x;
```
Structs

- typedef struct {
  int i;
  char c;
} S_t;

- S_t s
- int a = 112
- char b = ‘y’

- s.i = a
- s.c = ‘x’
- b = s.c
Struct Pointers

- `S_t` s
- `S_t` t
- `S_t` *p

- `p = &s` // p points at s
- `(*p).i = t.i` // assign element
- `p->i = t.i` // special syntax!
- `t = *p` // t=s
```c
typedef struct T {
    int i;
    struct T *p;  // ptr to own type
} T_t

T_t s
T_t t
T_t *q     // “struct T* q”

q = &s
q->p = &t  // changing s.p
q = q->p;  // t points to itself
```
Structure Parameters

- Call-by-value
  - Gets copied to the stack of the callee

```c
void foo(struct bar x, int b)
{
  ...x.a...
}
```
Structure Initialization

- Initialize all elements of a struct

```c
struct bar z = {'a', 45, &b};
```

- Combine with array initializers

```c
struct bar z[3] = { {'a', 45, &b},
                    {'b', 47, &c},
                    {'d', 10, &d} }
```
Linked Lists

Structs

Pointers
Linked list

- **Head pointer**
  - Points at first element in list
  - Outside the list per se

- **Next pointer**
  - Points to the next element
  - Stored inside the list

- **End marked by NULL**

- **Elements are structs in C**
Defining a list

- **Element type:**
  ```c
  typedef struct T {
    int value;  // short: v
    struct T *next;  // short: n
  } T_t
  ```

- **Head pointer:**
  ```c
  T_t *head
  head = NULL
  ```
First element

- Create a new element
  - Declare as variable
  - Allocate dynamically
  - We get a pointer to it

- \( T_t \ e \)
- \( T_t \ *q \)
- \( e.n = \text{NULL} // \\text{initialize!} \)
- \( q = \&e // \text{ptr to newcomer} \)
Insert into list

- Update head
  - head = q  //
  - q = &e  // ptr to newcomer

- First element
  - Special case
One more

- Link in at start of list
  - Update head
    - But first new element points at the old head of the list!

- T_t f // new element
- f.n = NULL // initialize next
- q = &f
- q->n = h
- h = q
Traverse a list

- Visit all elements
  - Start at head
  - Use a “walker”

- \( T_t *w \) // walker
- \( w = h \) // start
- \( \text{while}(w!=\text{NULL}) \)
  
  \[
  \begin{align*}
  &\quad \text{visit}(w) \\
  &\quad w = w->\text{next}
  \end{align*}
  \]
Remove element

- Link “over” the element
  - Change the predecessor
  - Special case for head!
- Unlink it from the list
  - Make removed element have NULL next pointer
Remove element

Example function

```c
void remove( T_t *e )
{
    if (h==e)
        h = h->next;
    else {
        T_t *w = h;
        while (w!=NULL) {
            if (w->next == e)
                w->next = w->next->next;
            w = w->next;
        }
    }
    e->next = NULL;
}
```
Arrays of Structures
Array of Structures

- $T_t \ a[4]$
- $T_t \ t = \{10, \text{NULL}\}$
- $T_t \ *p$

- $a[0] = t$
- $p = a+2 \quad // \quad p = &a[2]$
- $*p = t$
Static Memory

- Global array of struct:
  - Statically allocated
  - Gives you a memory area to work on
  - Good alternative to malloc() in OS
Exercises

To work on to familiarize yourself with pointers
Exercises

- Work on Sun/Linux/Windows
  - Get it right on host first!
  - Use malloc() to create elements
    - T_t *p = malloc( sizeof(T_t) )
  - Use printf() to see contents of lists
Sorted linked list

- Create a linked list that is always sorted, with the following functions:
  - `newelement(value)`: creates a new element using `malloc()`
  - `insert(list,newelement)`: inserts the new element in the right place in the list
  - `delete(list,element)`: removes the element from the list
  - `print(list)`: print the list
Double-linked lists

- Each element has two pointers:
  - next and previous
- Basic functions to create:
  - No requirements for sorting here!
  - newElement(value)
  - insert(list, newElement)
  - insert_after(list, element, newElement)
  - delete(list, element)
  - print(list, direction)
Linked list in array

- Declare a large array
  - \( T_t \ a[1000] \)
- Instead of malloc(), allocate new elements from unused array items
- "Thread" the linked list through the array
Linked list in array

- Two lists:
  - Used elements
  - Unused/free elements

- Allocate = move from free to used
  - Find a free item
  - Relink to the other list