C Intro (Part II)

Agenda

- Assert and Assignment 1
- Pointers
- Memory Model for C programs
- Header Files
- C preprocessor

Assert and Assignment 1

- Use `assert.h` library and `assert` for tests

Assert and Assignment 1

fact.c

```
#include <stdio.h>
#include <assert.h>

int fact(int n) {
  if (n == 1 || n == 0) {
    return 1;
  } else {
    return n * fact(n - 1);
  }
}

int main(void) {
  // fact tests
  assert(fact(0) == 1);
  assert(fact(1) == 1);
  assert(fact(5) == 120);

  return 0;
}
```

Assert and Assignment 1

fact.c

```
#include <stdio.h>
#include <assert.h> (1)

int fact(int n) {
  if (n == 1 || n == 0) {
    return 1;
  } else {
    return n * fact(n - 1);
  }
}

int main(void) {
  // fact tests
  assert(fact(0) == 1);
  assert(fact(1) == 1);
  assert(fact(5) == 120);

  return 0;
}
```

1. include `assert.h`

Assert and Assignment 1

fact.c

```
#include <stdio.h>
#include <assert.h> (1)

int fact(int n) {           (2)
  if (n == 1 || n == 0) {
    return 1;
  } else {
    return n * fact(n - 1);
  }
}

int main(void) {
  // fact tests
  assert(fact(0) == 1);
  assert(fact(1) == 1);
  assert(fact(5) == 120);

  return 0;
}
```

1. include `assert.h`
2. define all your functions before `main`

Assert and Assignment 1

fact.c

```
#include <stdio.h>
#include <assert.h> (1)

int fact(int n) {          (2)
  if (n == 1 || n == 0) {
    return 1;
  } else {
    return n * fact(n - 1);
  }
}

int main(void) {
  // fact tests          (3)
  assert(fact(0) == 1);
  assert(fact(1) == 1);
  assert(fact(5) == 120);

  return 0;
}
```

1. include `assert.h`
2. define all your functions before `main`
3. inside main for each functionm write tests using `assert`

Pointers: Declaration and Initialization

```
int *p
```

- `p` is a pointer to an `int`
  - think of it as: `p` is going to point to an integer value
- `p` is declared but not initialized!

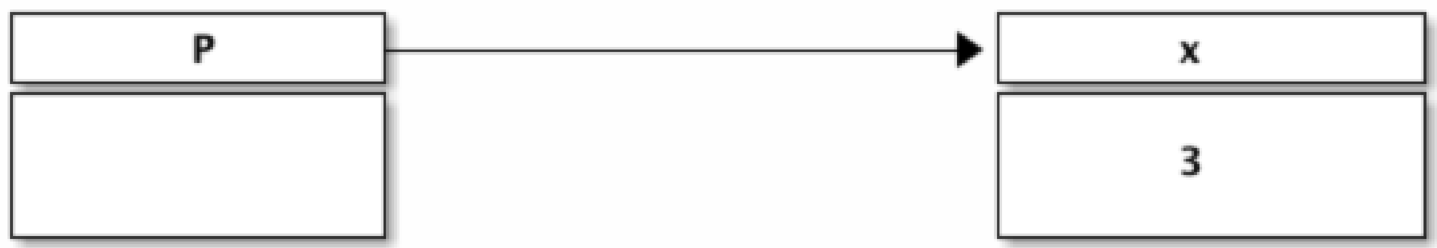
Pointers: Declaration and Initialization

```
int x = 3;
int *p = &x;
```

- We declare and initialize `x` to hold the value `3`
- We declare and initialize `p` to point to `x`

Pointers: Declaration and Initialization

```
int x = 3;
int *p = &x;
```



Pointers: Declaration and Initialization

What if I do not have a value to point to right now?

```
int *p = NULL;
```

- `NULL` is special!

Pointers: Declaration and Initialization

```
int *p = NULL;
```

Pointers: Dereference

```
int x = 3;
int *p = &x;

printf("The variable x is %d\n", x);
printf("The pointer p points to %d\n", *p);
printf("The pointer p is %p\n", p);
printf("The address of x is %p\n", &x);
printf("The address of p is %p\n", &p);
```

Pointers: Dereference

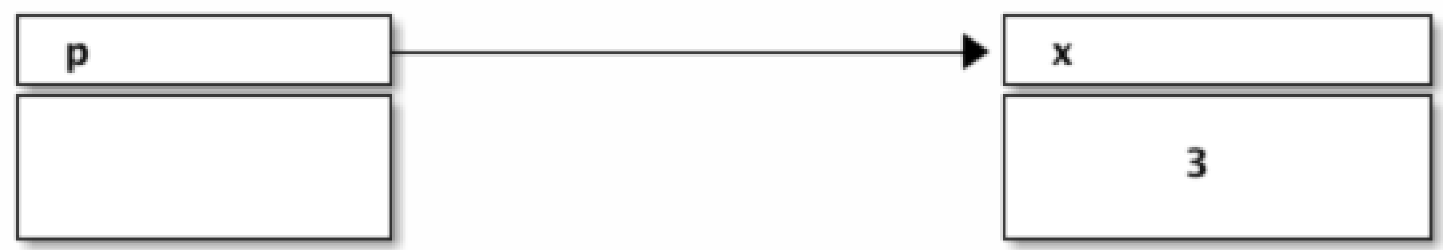
```
int x = 3;
int *p = &x;

printf("The variable x is %d\n", x);
printf("The pointer p points to %d\n", *p);
printf("The pointer p is %p\n", p);
printf("The address of x is %p\n", &x);
printf("The address of p is %p\n", &p);
```

Outputs:

```
The variable x is 3
The pointer p points to 3
The pointer p is 0xbfa01958
The address of x is 0xbfa01958
The address of p is 0xbf961ba8
```

Pointers: Dereference

Our original diagram



Pointers: Dereference

`p` holds the address of `x`, i.e., `&x`. That is what the arrow represented.

Pointers: Dereference

Let's take one more step and replace the names p and x with their addresses.

Pointers: Dereference

What happens when we alter the value stored in `x`

```
int x = 3;
int *p = &x;

printf("The variable x is %d\n", x);
printf("The pointer p points to %d\n", *p);
printf("The pointer p is %p\n", p);
printf("The address of x is %p\n", &x);
printf("The address of p is %p\n", &p);

x = 500;

printf("\n\nThe variable x is %d\n", x);
printf("The pointer p points to %d\n", *p);
printf("The pointer p is %p\n", p);
printf("The address of x is %p\n", &x);
printf("The address of p is %p\n", &p);
```

Pointers: Dereference

What happens when we alter the value stored in `x`

Outputs

```
The variable x is 3
The pointer p points to 3
The pointer p is 0xbfa01958
The address of x is 0xbfa01958
The address of p is 0xbf961ba8


The variable x is 500
The pointer p points to 500
The pointer p is 0xbfa01958
The address of x is 0xbfa01958
The address of p is 0xbf961ba8
```

Pointers: Dereference

Let's go back to our images. What happened. We started with

Pointers: Dereference

Then we executed `x=500` and we got

Pointers: Dereference

We **mutated** `x`; we deleted `3` and replaces it with `500`. Any variable that was pointing to the address of `x` **sees** the update.

Dereferencing NULL

What happens when we run this code?

```
int *p1;
int *q = NULL;

printf("What does p1 point to? %d\n", *p1);
printf("What does q point to? %d\n", *q);
```

Dereferencing NULL

What happens when we run this code?

```
int *p1;
int *q = NULL;

printf("What does p1 point to? %d\n", *p1);
printf("What does q point to? %d\n", *q);
```

Outputs

```
What does p1 point to? -1079514593
zsh: segmentation fault  ./a.out
```

Pointers and Arrays

- Arrays are formed by placing the elements contiguously in memory.

```
int array[4];

array[1]; // is of type int
array;    // is a pointer to the first array
element

int *p = (array + 1);  // points to array[1]
int x = array[1];      // the value at index 1
                       // what p points to!

p = p + 1;   //  moves p by one int to point to
array[2]
```

Heap

- Space in memory that allows for dynamic allocation and deallocation.
- Request memory using `void *malloc(size_t size)`
- Release memory using `void free(void *block)`
- Reuse memory using `void *realloc(void *block, size_t size)`

And we need a way to tell how much memory we need for each type!

- `size_t sizeof(type)`, looks like a function it is not!
- `size_t sizeof expression`, it is an expression.

Heap and Stack

```
int a[1000];   // stack allocated

int *b;

b = (int*) malloc (sizeof(int) * 1000);
assert(b != NULL);

a[100] = 7;
b[100] = 7; // we can still use [] to index the
array

free(b);  // give the memory back!
```

Heap and Stack: function calls

Whiteboard!

Singly Linked List of `int`

- Design each node, what do we have to store?
- List needs to dynamically grow and shrink.
- Operations

1. `Node *list_create(int element)`
   - create a new list and add `element`
2. `void list_add(int element, Node *list)`
   - add `element` as the first item to `list`
3. `int list_get_first(Node *list)`
   - return the first item. List is unchanged
4. `Node *list_get_rest(Node *list)`
   - return the list without it's first item

Prototypes

- Functions need to be defined before use.
- A function prototype tells the compiler the **signature**. This is the declaration of a function.
    - `int total_tax(int sum);`

Header Files: Organizing code

- `#include <stdio.h>` - grab `stdio.h` and paste in here.
  - Where is `stdio.h`?
- We can make our own header files and include them using `#include "list.h"`
  - **NOTE** quotes instead of < >. Quotes mean ***relative*** to the source file.

Header Files: Organizing code

- Header files define the interface to our module for clients
  - functions and types
- Clients
  - include our header file
  - prefix prototypes with `extern` (more on `extern` in a minute)
- Implementors
  - include the header file
  - provide the implementation for each function prototype in the header file
- Java coders, header files kinda like Java interfaces.

Scope

- A `.c` file is one compilation unit.
- We have seen local function variables.
- Variables visible to all functions in a `.c` file.
  - define **once** outside any function
  - use `extern` to declare the use of it inside a function

Scope

```
#include <stdio.h>

int max;     //scope is the whole file

int is_max(int val) {
    extern int max;     /* refers to max above */

    if (max > val) {
        return 0;
    } else {
        max = val;
        return 1;
    }
}

int get_max() {
    extern int max; /* refers to max above */
    return max;
}
```

Scope

- There is also `static`
  - can be used for variables and functions
- `static int x`
  - visible to functions in the same file as `x`
  - invisible to function defined outside the file where `x` is defined
- similar use for functions
- think **private** to the compilation unit.

Preprocessor

- Recall `gcc -E`?
- include other files, e.g, `#include <stdio.h>`
- define constants, e.g., `#define SIZE 100`
- `gcc` has the `-I` argument that allows us to add more directories to search for `.h` files.
- we can also
  - free/remove a definition using `#undef`
  - check if it is already define `#ifdef` or not `#ifndef`
  - if-else control flow with `#if`, `#elif` and `#else`
  - and more complex macros `#define INC(x) x++`
- MACROS perform substitution with arguments ***unevaluated***. Be careful!