Casting pointers

Casting pointers

- Java is considered to have **static** and **strong** typing.
- C is considered to have **static** and **weak** typing.
- What this means can be exemplified by the ability to cast pointers to pointers of one type to pointers of other, arbitrary types.
  - See pointers/casting/weak_typing.c
- The results are surprising at first, but as we begin to understand bit-level representations of numbers, we'll see why the program behaves the way it does.

Casting pointers to arrays

- Pointer (as other types) can be cast to other types.
- What does this program do?
  - pointers/casting/casting.c
- The explanation of what it prints is something you should make sure you understand.
- Figure it out, talk about it on Piazza if you are not sure.

Strings

Strings are arrays

- C has very little native support for strings.
- A string in C is (literally) an array of chars terminated by a null character ('\0'). Sometimes called the 'zero code'.
- Most functions that perform operations on strings uses that null character in some form of fashion, its presumed to always be there when handling a string.
- A missing null character in a string is historically a common source of errors related to strings in C.

Strings are arrays *cont*.

- Whenever we have a double quoted string in a program it is stored as a string *literal* terminated by a *null character*.

```
printf("Hello World");
```

- The following creates and array of **6** characters (5 letters + the null character).

```
char h[] = "hello";
```

- It is equivalent to writing…

```
char h[6]= "hello";
```

- Note that the following lines will compile as well and sometimes even run without obvious problems, at least initially.

```
char h[5] = "hello";
char h[2] = "hello";
```

String length

- Since strings are simply arrays of characters, we have no size information accompanying the string (as we do in Java)
- The assumption is that there is null character that terminates the string as this is the only way of knowing where the string ends.
- The term *length of a string* is used to describe the number of bytes *preceding* the null character.
- See strings/strings.c

String copy, concat, append, etc…

- You can write all these operations yourselves now that you know what you know about the null character.

- However, many of these things have been implemented for us in **<string.h>**

- The string.h header file contains declarations of many useful functions for working with null terminated string.

- You can learn about whats available by using the man pages or reading through the Wikipedia page on the subject

  - https://en.wikipedia.org/wiki/C_string_handling

String misconceptions

- A common misconception is that all char arrays are strings, because *string literals* are converted to arrays during compilation.
- It is important to remember that a string ends at the first zero code. Therefore..
  - A char array that contains a null character before the last byte contains a string, or possibly several strings, but is not itself a string.
  - Conversely, it is possible to create a char array that is not null-terminated and is thus not a string.

C Structs

First, a quick word on types

- We've seen a few types at this point: **char**, **int**, **float**, **char***
- Types are important because:
  - They allow your program to impose logical structure on memory
  - They help the compiler tell when you're making a mistake
- Next we will discuss:
  - How to create logical layouts of different types (structs)
  - How to create new types using typedef

What is a struct?

- Java and C++ have classes. C has structs.
- Structs are something like classes *without* functions.
- Sometimes they are called a record or “compound data
- A struct is a type which is a collection of variables, possibly of different types, grouped together under a name.

```
struct <struct-typename>{
    <type> <identifier_list>;
    <type> <identifier_list>;
    ...
};
```

Each identifier defines a member of the structure.

A simple struct

- Here is a definition for a type called **point** with two members x and y.

```
struct point {
   float x;
   float y;
};
```

- In order to declare a variable of type point…

```
struct point p;
```

- The keyword *struct* is needed in the declaration of the variable.

- To access individual coordinates of the point we use the dot operator:

```
p.x = 3.5;
p.y = 8.9;
printf(”p = (%f,%f)”, p.x, p.y);
```

More complex structs

- Structure definitions can contain any number of members of any type. This includes pointers. Ex.

```
struct student {
   char* id;
   char* name;
   float gpa;
   int num_of_credits;
};
```

- There can be a structure whose members are other types of structures. For example, a rectangle defined by its two diagonally opposite corners.

```
struct rectangle {
   struct point c1;
   struct point c2;
};
```

Struct layout in memory

- Usually the fields are stored in consecutive positions in memory, in the same order as they are declared in the record type.
- Ex

```
struct example {
   char a;
   char b;
};

struct example ex = {'a', 'b'};
```

Name	Addr	Value
	0	
	1	
	2	
	3	
ex.a	4	‘a’ (97)
ex.b	5	‘b’ (98)
	6	
	7	
	8	
	9	
	10	

Pointers to structs

- We can create a variable of any struct type. Therefore, it stands to reason that we could have pointers to structs.

```
struct point p;
p.x = 1.5;
p.y = 3.1;

struct point* pp;
pp = &p;   // pp is a pointer to a struct
```

- Using the standard pointer dereference operator we can access the values of the members of **p** via **pp**

```
float x = (*pp).x;
float y = (*pp).y;
```

- The parentheses are needed since the . operator has higher precedence than the * operator

+ Pointers to structs *con't*

- Pointers to structures are frequently used, therefore there is a shorthand notation for accessing members of structs via a pointer.

```
float x = pp->x;
float y = pp->y;
```

  - See struct/simple/struct.c

- Once the arrow operator is used, we can then use the dot notation to reference the properties, if the members of the structs *are other structs*.

```
struct rectangle r =  // initialized properly
float x = pr->c1.x;
```

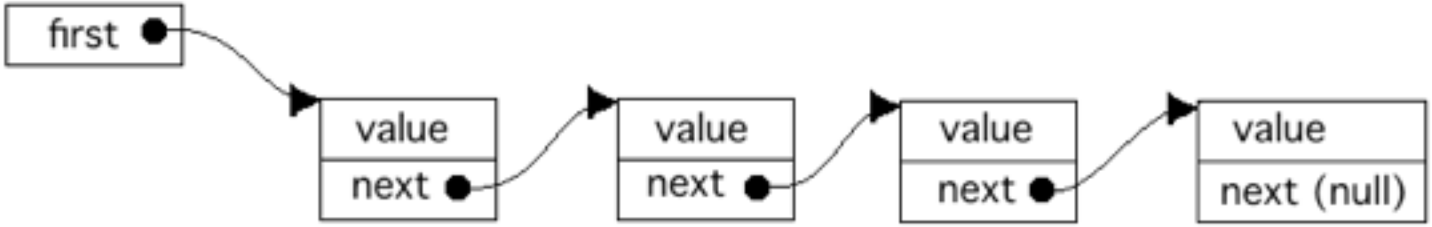
  - See struct/structs_in_structs/struct.c

Structs & functions

- Structures can be passed to functions either by copying all the values contained within or by a pointer (just like any other variable).
- Its generally far more efficient to use pointers to structs to pass them as arguments.
- See structs/functions/struct.c

Structs & data structures

- Structs are key in implementing data structures in C.
- For example, with a linked list, we could have a struct ‘node’ that contains a value and a ‘next’ pointer to a node.



- See structs/data_structures/list.c

How to read C types & expressions

Precedence rules are important

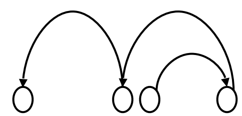
- When you use multiple operators, make sure you are clear on what happens in what order.
- C has its precedence rules and has no trouble parsing expressions like the this `*p->str++;`

Precedence rules are important

- When you use multiple operators, make sure you are clear on what happens in what order.
- C has its precedence rules and has no trouble parsing expressions like the this `*p->str++;`
- What actually happens is `(*(p->str))++`
  - p is a pointer to a structure, so -> accesses its member.
  - str is a member of p and is a pointer itself
  - * dereferences that pointer.
  - Finally, ++ is applied to the value after dereferencing str

How to read types

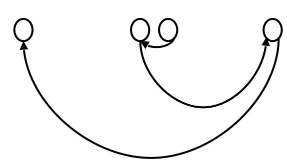
- C type names can be understood by starting at the name and working outwards according to the rules of precedence:



`int *x[10];`

x is an array of pointers to int

`int (*x)[10];`



x is a pointer to an array of int

Operator precedence

<table>
<tr><th>Operator Type</th><th>Operator</th><th>Associativity</th></tr>
<tr><td>Primary Expression Operators</td><td><code>() [] . -> expr++ expr--</code></td><td>left-to-right</td></tr>
<tr><td>Unary Operators</td><td><code>* & + - ! ~ ++expr --expr (typecast) sizeof</code></td><td>right-to-left</td></tr>
<tr><td rowspan="10">Binary Operators</td><td><code>* / %</code></td><td rowspan="10">left-to-right</td></tr>
<tr><td><code>+ -</code></td></tr>
<tr><td><code>>> <<</code></td></tr>
<tr><td><code>< > <= >=</code></td></tr>
<tr><td><code>== !=</code></td></tr>
<tr><td><code>&</code></td></tr>
<tr><td><code>^</code></td></tr>
<tr><td><code>|</code></td></tr>
<tr><td><code>&&</code></td></tr>
<tr><td><code>||</code></td></tr>
<tr><td>Ternary Operator</td><td><code>?:</code></td><td>right-to-left</td></tr>
<tr><td>Assignment Operators</td><td><code>= += -= *= /= %= >>= <<= &= ^= |=</code></td><td>right-to-left</td></tr>
<tr><td>Comma</td><td><code>,</code></td><td>left-to-right</td></tr>
</table>

Typedefs

Using typedef

- At this point we pretty much seen then entire type system of C.
- It turns out we can create our own aliases for types with typedefs.
- Typedefs are a way of creating more convenient or shorter names for existing types.
- For example, the following creates a new type called **int_pointer** that is equivalent to **int***

```
typedef int* int_pointer;
```

- See typedefs/typedef.c

Using typedef *con't*

- Typedefs tend to be often used with the structures.
- We could rewrite the node structure we saw before as follows…

```
typedef struct {
        char * word;
        struct node * next;
} node;
```

- …..which allows us to use *'node'* as the typename, rather than *'struct node'* everywhere.. (saves us some typing).
- Moreover, these two lines become equivalent..

```
struct node n;
node n;
```

Macros

Using Macros

- A macro is a fragment of code which has been given a name. Whenever the name is used, it is replaced by the contents of the macro.
  - This happens during 'preprocessing'
- There are two kinds of macros. They differ mostly in what they look like when they are used.
  - object-like macros resemble variables when used
  - function-like macros resemble function calls.
- For now, we will concern ourselves with only 'object-like' macros.

Object-like macros

- An object-like macro is an identifier which will be replaced by a code fragment.
- They are most commonly used to give symbolic names to numeric constants.
- You create macros with the '#define' directive. Ex.

```
#define ARRAY_SIZE 100
```

- Then later in your code you can use that macro name like so..

```
int[ARRAY_SIZE] my_int_array;
```

- See macros/macros.c

Unix Commands

Unix basic command cheat sheet

`man 'command'`	display a manual page (or simply *help*) for the command (this is the easiest way to learn about options to the commands that you know and about new commands)
`pwd`	print the name of the present working directory
`ls`	list content of the current working directory
`ls dir_name`	list content of the directory named `dir_name`
`cd dir_name`	cd stands for change directory, changes the current working directory to `dir_name`
`cd ..`	move one directory up in the directory tree
`cd`	change the current working directory to your home directory
`cp file1 file2`	copy `file1` into `file2`, where `file1` and `file2` can be either relative or complete path names
`mv file1 file2`	move `file1` into `file2`, where `file1` and `file2` can be either relative or complete path names
`rm file`	remove a file (there is no undoing it, so be very careful!)
`mkdir path`	make a directory at the specified `path`
`rmdir path`	remove the directory specified by the `path` (there is no undoing it, so be very careful!)
file file_name	determine the type of a file
less file_name	view the file in the terminal
more file_name	view the file in the terminal
cat file_name(s)	concatenate files and print them to standard output