Extra C Material


Two dimensional array vs pointer to array

```c
char *students[] = {"Fi", "April", "Raghd", "Jack", "Bobby"};
```

```plaintext
FI
April
Raghd
Jack
Bobby
```

```c
char stu_arr[][10] = {"Fi", "April", "Raghd", "Jack", "Bobby"};
```

```plaintext
stu_arr:
<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>FI</td>
<td>April</td>
<td>Raghd</td>
<td>Jack</td>
<td>Bobby</td>
</tr>
</tbody>
</table>
```

students[2][3]; and stu_arr[2][3]; are both valid

Access two dimensional array with one index

- Two dimensional arrays are stored in memory as a large one dimensional array
- Sometimes it is convenient to index into this array with a single index
  - First dimension is rows
  - Second dimension is columns

```c
int a[3][4];
tt x;
x = a[3][3]; // this is equivalent to
x = a[7];  // this
```

- Formula for conversion
  - Index = row index * number of columns + column index

1-2

1-3
char *argv[
• argv: Argument Vector
• Pointer to array of character strings
• argv[0]: name of current program
  • Number of args at least 1
• Args[1- n]: command line arguments
• Terminated by a null pointer

Function Pointers
Functions are not variables in C
Pointers to functions are Things you can do with a function pointer
• Assignment
• Store an array of function pointers
• Use as a function argument
• Return function pointer from a function
Ex. int (*foo)(char *, char *);  
• foo is a pointer to a function that takes two char * arguments and returns an int;
• Parenthesis needed  
  ➢ int * foo(char *, char *);
  ➢ foo is a function that takes two char * arguments and returns an int *.

Function Pointers
Why parenthesis around function pointer name:
• precedence of * vs ()
Typed function pointer
• typedef int (*foo)(int, int);
• foo now has the type pointer to function that takes two int arguments
Store a function in a function pointer
int bar(int a, int b){
  return a + b;
}
int (*foo)(int, int);
foo = &bar;

typedef int (*foo)(int, int);
foo f = &bar;
Unions

Unions are variables that use the same memory area to hold objects of different types and possibly sizes
- Only one object can be stored at any one time
- Bits in memory do not change only how they are interpreted
- Programmers job to keep track of what type is currently being stored in the union

Same operations as a struct:
- . for union variable member
- -> for union pointer member
- Assign to
- Copy
- Take address of

Ex.
```c
int main() {
    union tag {
        float f;
        int x;
        char *s;
    } t; // all members reference the same memory/data/bits

    t.f = 999999;
    printf("%.0f\n", t.f); // print value of float member as a float
    printf("%d\n", t.x); // print value of int member as an int
}
```

- Can be members of structs or have structs as members
- Can only be initialized with a value of the same type as the first member
  - in this case float

Bit Fields

Bit Fields are a way to directly access bits
- Save space
- Change individual bit values without masks
  - x = x | xfff;
- Implementation dependent
- Not very portable
- Fields declared as int
- Specify unsigned or signed for better portability
- Fields behave like small integers

Ex.
```c
struct car{
    unsigned int ignition_on: 1; // unnamed fields used as padding
    unsigned int engine_status: 3; // fields can have different width
} p_car;

p_car.ignition_on = 1; // easy to change a bits value
```