Pointers, Arrays and Structures

Many more applications…

Pointers & Arrays

Pointers and Arrays

When an array is declared:

- **The compiler allocates a *base address* and sufficient amount of storage to contain all the elements of the array in contiguous memory locations.**
- **The *base address* is the location of the first element (*index 0*) of the array.**
- **The compiler also defines the array name as a *constant pointer* to the first element.**

Example

Consider the declaration:

```
int x[5] = {1, 2, 3, 4, 5};
```

- Suppose that the base address of x is 2500, and each integer requires 4 bytes.

Element	Value	Address
x[0]	1	2500
x[1]	2	2504
x[2]	3	2508
x[3]	4	2512
x[4]	5	2516

Example (contd)

Both x and &x[0] have the value 2500.

p = x; and p = &x[0]; are equivalent

- We can access successive values of x by using p++ or p-- to move from one element to another.

Relationship between p and x:

p = &x[0] = 2500
p+1 = &x[1] = 2504
p+2 = &x[2] = 2508
p+3 = &x[3] = 2512
p+4 = &x[4] = 2516

*(p+i) gives the value of x[i]

Arrays and pointers

- An array name is an address, or a pointer value.
- Pointers as well as arrays can be subscripted.
- A pointer variable can take different addresses as values.
- An array name is an address, or pointer, that is fixed.
  - It is a CONSTANT pointer to the first element.

Arrays

Consequences:

- `ar` is a pointer
- `ar[0]` is the same as `*ar`
- `ar[2]` is the same as `*(ar+2)`
- We can use pointer arithmetic to access arrays more conveniently.

Declared arrays are only allocated while the scope is valid

```
char *foo( ) {
  char string[32];
  return string;
}
```

This is incorrect

```
char *foo( ) {
  char *string;
  string = malloc(32);  // Dynamic memory allocation
  return string;
}
```

This is okay

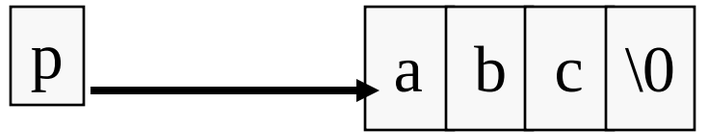
Strings

- 1-d arrays of type char
- By convention, a string in C is terminated by the end-of-string sentinel '\0' (null character)
- char s[21] - can have variable length string delimited with \0
  - Max length of the string that can be stored is 20 as the size must include storage needed for the '\0'
- String constants : "hello", "abc"
- "abc" is a character array of size 4

String Constant

- A string constant is treated as a pointer
- Its value is the base address of the string

```
char *p = “abc”;
```




```
printf (“%s %s\n”,p,p+1); /* abc bc is printed */
```

Arrays In Functions

An array parameter can be declared as an array or a pointer; an array argument can be passed as a pointer

```
int strlen(char s[])
{


}
```

```
int strlen(char *s)
{


}
```

Arrays and pointers

int a[20], i, *p;

The expression a[i] is equivalent to *(a+i)

p[i] is equivalent to *(p+i)

When an array is declared the compiler allocates a sufficient amount of contiguous space in memory. The base address of the array is the address of a[0].

Suppose the system assigns 300 as the base address of a. a[0], a[1], ...,a[19] are allocated 300, 304, ..., 376.

Arrays and pointers

```
#define N 20
int a[2N], i, *p, sum;
```

p = a; is equivalent to p = *a[0];

p is assigned 300.

Pointer arithmetic provides an alternative to array indexing.

p=a+1; is equivalent to p=&a[1]; (p is assigned 304)

```
for (p=a; p<&a[N]; ++p)
        sum += *p ;
```

```
for (i=0; i<N; ++i)
        sum += *(a+i) ;
```

```
p=a;
for (i=0; i<N; ++i)
        sum += p[i] ;
```

Arrays and pointers

int a[N];

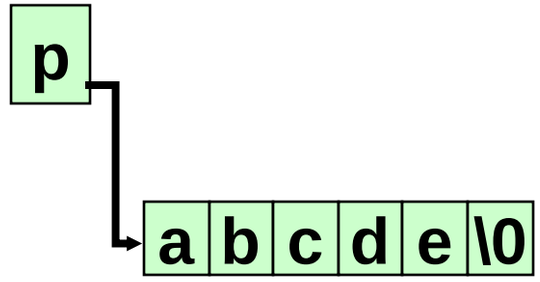
a is a constant pointer.

a=p; ++a; a+=2; illegal

Differences : Array & Pointers

char *p = “abcde”;

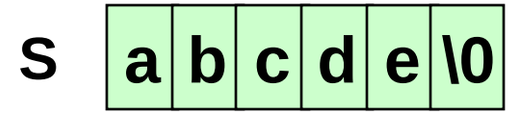
The compiler allocates space for p, puts the string constant “abcde” in memory somewhere else, initializes p with the base address of the string constant



char s[ ] = “abcde”;

≡ char s[ ] = {‘a’,’b’,’c’,’d’,’e’.’\0’};

The compiler allocates 6 bytes of memory for the array s which are initialized with the 6 characters



Pointer arithmetic and element size

```
double * p, *q ;
```

The expression `p+1` yields the correct machine address for the next variable of that type.

Other valid pointer expressions:

- `p+i`
- `++p`
- `p+=i`
- `p-q` `/* No of array elements between p and q */`

Pointer Arithmetic

Since a pointer is just a mem address, we can add to it to traverse an array.

`p+1` returns a ptr to the next array element.

`(*p)+1` vs `*p++` vs `*(p+1)` vs `*(p)++`?

- `x = *p++` ⇒ `x = *p ; p = p + 1;`
- `x = (*p)++` ⇒ `x = *p ; *p = *p + 1;`

What if we have an array of large structs (objects)?

- C takes care of it: In reality, `p+1` doesn't add `1` to the memory address, it adds the size of the array element.

Pointer Arithmetic

We can use pointer arithmetic to "walk" through memory:

```
void copy(int *from, int *to, int n) {
    int i;
    for (i=0; i<n; i++)  *to++ = *from++;
}
```

- C automatically adjusts the pointer by the right amount each time (i.e., 1 byte for a char, 4 bytes for an int, etc.)

Arrays of Structures

We can define an array of structure records as

```
struct stud class[100];
```

The structure elements of the individual records can be accessed as:

```
class[i].roll
class[20].dept_code
class[k++].cgpa
```

Pointers and Structures

Once `ptr` points to a structure variable, the members can be accessed as:

```
ptr -> roll;
ptr -> dept_code;
ptr -> cgpa;
```

- The symbol "–>" is called the *arrow* operator.

A Warning

When using structure pointers, we should take care of operator precedence.

- **Member operator "." has higher precedence than "*".**

  ptr –> roll and (*ptr).roll mean the same thing.

  ***ptr.roll will lead to error.**

- **The operator "–>" enjoys the highest priority among operators.**

  ++ptr –> roll will increment roll, not ptr.

  (++ptr) –> roll will do the intended thing.

Use of pointers to structures

```
#include <stdio.h>
struct complex  {
                float  real;
                float  imag;
};

main( )
{
   struct  complex  a, b, c;
   scanf ( "%f %f", &a.real, &a.imag );
   scanf ( "%f %f", &b.real, &b.imag );
   add( &a, &b, &c ) ;
   printf ( "\n %f %f", c,real, c.imag );
}
```

```
void add (x, y, t)
struct complex  *x, *y, *t;
{
   t->re = x->real + y->real;
   t->im = x->imag + y->imag;
}
```

Practice Problems

1. Write a function to search for an element in an array of integers that returns 1 if the element is found, 0 otherwise. If found, it also returns the index in the array where found
2. Write a function that returns the number of lowercase letters, uppercase letters, and digit characters in a string
3. Define a structure POINT to store the coordinates (integer) of a point in 2-d plane. Write a function that returns the two farthest (largest distance) points in an array of POINT structures
4. Write a function that takes two arrays of integers A and B and returns the size of the union set and the size of the intersection set of A and B
5. Write a function that returns the lengths of the largest palindromes formed by any substring (sequence of consecutive characters) of the string. It should also return the index in the string from which the palindrome starts.

For all of the above, add suitable main() functions to call the functions. Also, decide on what parameters you will need; for better practice, for all problems other than problems 1, assume that the return type of the function is void.