Lecture notes on Sep 16, 2013

William W. Symes
Scribe: Linlin Zhang

September 16, 2013

There are two ways to define constant, the first way is to declare a variable with keyword const, which specify the value of the variable will not be changed. The second way is using #define, which defines a symbolic constant to be a particular string of characters. src/ex16.c show an example with three ways to define an array with constant array size. You can try to remove const in declaration of arr_size, and rerun the code, it will return the same result. But if you remove const, and change arr_size=4, then it may generate errors (dangerous programming). In the codes of ex16.c, when assigning the values to a1, cast is used, which converts type of i from integer to double.

```
#include <stdio.h>

/* Author: WWS
   Purpose: illustrate array declaration, looping through
   array elements with for, and a simple linear algebra op
   (vector addition)
*/

#define ARR_SIZE 3

int main() {

  const int arr_size = 3;

  double a1[ARR_SIZE]; /*ARR_SIZE is a variable declared with the keyword const*/
  double a2[arr_size]; /*arr_size is a symbolic constant*/
  double a3[3]; /*3 is a literal constant*/

  int i;   /* loop counter */

  for (i=0;i<arr_size;i++) {
    a1[i]=(double)i;
```

```
    a2[i]=2.0 * i;
    printf("a1[%d]=%12.4e a2[%d]=%12.4e\n",i,a1[i],i,a2[i]);
  }

  for (i=0;i<ARR_SIZE;i++) {
    a3[i]=a1[i]+a2[i];
    printf("(a1+a2)[%d]=%12.4e\n",i,a3[i]);
  }

  return 0;
}
```

How to pass the values of a variable to a function? Each variable inside the function scope is a copy (local) of the external variable. The external variable will not be changed no matter what you do to the copy inside the function scope. The two variables have different memory! By running scr/ex25.c, we find that the output of x "before" and "after" are the same, both return the value of 1.

Pointers are variables storing addresses for various types. Declaration of pointers: int * a; float * a; which define a as pointer type of variable. There are two important operations. The operator * is the dereferencing operator, when applied to a pointer, it accesses the object the pointer points to. In the declaration of pointer: int * a; *a as an int (=value stored at address) dereferencing a pointer. The operation & gives the address of an object, the statement x=&a assigns the address of a to the variable x. In ex25.c, b is a copy in the address, pass the address to change the values of variable in function. By runing ex25.c, the value of y is 1 "before" and 2 "after". The function void is no type, and has only one value, and do not return a value on this function. In function argument list, array=address of 1st member.

```
/*
 * Author: WWS
 *
 * Purpose: illustrate pass-by-value and C's fake pass-by-reference
 * using pointers.
 */

#include <stdio.h>

/* void is the "no-type", used when no return value is required
   (amongst other things)
*/
void fun(int a, int * b) {
  a++;
```

```
  (*b)++;
}

int main() {

  int x=1;
  int y=1;

  printf("before: x=%d y=%d\n",x,y);
  fun(x,&y);
  printf("after: x=%d y=%d\n",x,y);

}
```