Pass by value in invoking functions

R. Inkulu
http://www.iitg.ac.in/rinkulu/

Passing arguments to functions (review)

semantically incorrect code

```
void func(int y) {
    //y has 10
    y = 15;
    //y has 15
    return;
}

int main(void) {
    int x=10;
    func(x);
    printf("%d \n", x);  //prints 10 !
    return 0;
}
```

- function parameters are passed *by value*

Simulating pass by reference with pass by value

```
void func(int *y) {
    //now the value of y is the address of x
    //(x is a variable defined in main)
    *y = 15;
}

int main(void) {
    int x=10;
    func(&x);          //address of x is passed to func
    printf("%d \n", x);  //prints 15
}
```

- the adv of of avoiding passing objects to functions by value includes redundant objects being in memory: leading to the reduction in space usage

Passing arguments to functions: swap func

semantically incorrect code

```
void swap(int v, int w) {
    int tmp = v;
    v = w;
    w = tmp;
}

int main(void) {
    int x=10, y=15;
    swap(x, y);
    printf("%d, %d \n", x, y);    //prints 10, 15
}
```

Simulating pass by reference with pass by value: swap func

```
void swap(int *v, int *w) {
    int tmp = *v;
    *v = *w;
    *w = tmp;
}

int main(void) {
    int x=10, y=15;
    swap(&x, &y);
    printf("%d, %d \n", x, y);    //prints 15, 10
}
```

Passing one-dimensional arrays to functions

```
int countNonZeros(int *v, int len) {
   //v has &x[0] and len has 3
   int count = 0, i;
   if (len > 0) v[len-1] = 35;
   for (i=0; i<len; i++)
       if (v[i] != 0) ++count;
    return count;
}
int main(void) {
    int x[3] = {20, 30, 0};
    int k = countNonZeros(x, 3);  //countNonZeros(&x[0], 3)
    printf("%d, %d \n", k, x[2]);    //prints 3, 35
}
```

- change of values of any element of v in *countNonZeros* gets reflects in array x of *main*

Passing one-dimensional arrays to functions (another signature)

```
int countNonZeros(int v[], int len) {
   //'int v[]' gets translated to 'int *v'
   //hence, pointer arithmatic is allowed over 'v'
   int count = 0, i; v++; --v;
   if (len > 0) v[len-1] = 35;
   for (i=0; i<len; i++)
       if (v[i] != 0) ++count;
   return count;
}
int main(void) {
    int x[3] = {20, 30, 0};
    int k = countNonZeros(x, 3);  //countNonZeros(&x[0], 3)
    printf("%d, %d \n", k, x[2]);    //prints 3, 35
}
```

- change of values of any element of v in *countNonZeros* gets reflects in array x of *main*

Passing partial arrays to functions

```
int countNonZeros(int *v, int len) {
   //v has &x[1] and len has 2
   int count = 0, i;
   if (len > 0) v[len-1] = 35;
   for (i=0; i<len; i++)
       if (v[i] != 0)    //equivalently,  if (*(v+i) != 0)
          ++count;
    return count;
}

int main(void) {
    int x[3] = {20, 30, 0};
    int k = countNonZeros(&x[1], 2);
    printf("%d %d \n", k, x[2]);    //prints 1
}
```

Passing partial arrays to functions (another signature)

```
int countNonZeros(int v[], int len) {
   //'int v[]' gets translated to 'int *v'
   int count = 0, i;
   if (len > 0) v[len-1] = 35;
   for (i=0; i<len; i++)
       if (v[i] != 0)    //equivalently,  if (*(v+i) != 0)
          ++count;
    return count;
}

int main(void) {
    int x[3] = {20, 30, 0};
    int k = countNonZeros(&x[1], 2);
    printf("%d %d \n", k, x[2]);    //prints 1
}
```

Passing two-dimensional arrays to functions

```
void func(double (*b)[4], int rownum)
     //b points to a contiguous sequence of double [4]s
{ ...
  (*(b+rownum))[1] = 78;
     //could be written as b[rownum][1] = 78
     //lvalue is (*(b + rownum*sizeof(double [4])))[1]
     //means b[rownum][1]
  ...  } }
int main(void) {
   double a[3][4]; //a is [3] double[4]s
   func(a, 2); }
```

- formal parameter need to include the number of columns; the number of rows are irrelevant: to access a particular element, compiler need to be able to calculate the offset (which could involve number of columns) from the beginning of array b

Passing two-dimensional arrays to functions (another signature)

```
void func(double b[][4], int rownum)
    //equivalently, formal parameter can be b[3][4]
    //double b[][4] is translated to double (*b)[4]
{ ...
  b[rownum][3] = 78;
     //lvalue is *(*(b+rownum)+3), which is
     // *(*(&b[0] + (rownum*sizeof(double [4]))) +3)
     // *(&b[rownum][0] + 3*sizeof(double))
  ...  }
int main(void) {
   double a[3][4];
   func(a, 2); }
```

- as mention in the previous slide, formal parameter need to include the number of columns; the number of rows are irrelevant: to access a particular element, compiler need to be able to calculate the offset (which could involve number of columns) from the beginning of array b

Passing multi-dimensional arrays to functions

Homework!