EE 441 – CH2

ARGUMENT PASSING, RECURSIVE FUNCTIONS

Instructor: UĞUR HALICI

ARGUMENT PASSING

Suppose that we desire to swap the contents of two integer variables. Consider the following function:

```
void swap(int v1, int v2)
{ int tmp=v2;
   v2=v1;
   v1=tmp;
}

main()
{ int i=10;
   int j=20;
   cout << "Before swap():\ti:"<<i<<"\tj:"<<j<<endl;
   swap(i,j);
   cout << "After swap():\ti:"<<i<<"\tj:"<<j<<endl;
}
```

when executed the result is not as we desired, but it is

```
Before swap(): i:10 j:20
After swap():    i:10  j:20
```

Two alternatives to solve the problem:

The first alternative is to use pointers as parameters:

```
void  pswap(int *v1, int *v2) // parameters are pointers
{  int temp=*v2;
    *v2=*v1;
    *v1=temp;
}

main ()
{.....
pswap(&i, &j); // send address of i and j as parameter
...
}
```

When executed, we will have:

Before swap(): i:10 j:20
After swap(): i:20 j:10

Second alternative is to use reference:

```
void rswap(int &v1, int &v2)
{ int temp=v2;
        v2=v1;
        v1=temp;
}


main ()
{.....
rswap(i, j);
...
}
```

When compiled and executed, we will have

```
Before swap(): i:10 j:20
After swap():    i:20  j:10
```

However if it is declared as

```
void crswap(const int &v1, const int &v2)
{ int temp=v2;
        v2=v1;
        v1=temp;
}
main ()
{.....
crswap(i, j);
...
}
```

When executed, we will have

Before swap(): i:10 j:20

After swap():   i:10 j:20

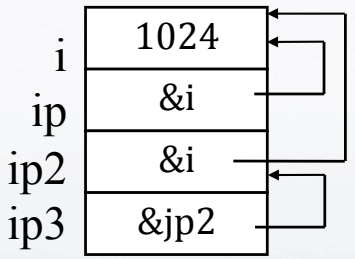
Pointer Types:

A pointer variable holds values that are the addresses of objects in memory. Typical uses of pointers are the creation of linked lists and the management of objects allocated during execution.

```
int* ip // pointer decleration
int *ip // the same as above
int *ip1, *ip2 // two pointers
int* ip1, ip2 // a pointer, an integer
```

```
int ip, *ip2 // an integer and an integer pointer
long *lp, lp2 // a long integer pointer and a long integer
float fp, *fp2 // a floating point and a floating point pointer
```

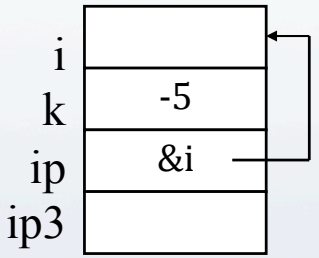
```
int i=1024
//create an  pointer that points to i
int *ip=i // error, type mismatch
int *ip=&i // ok. the operator &
            //is referred as address-of operator
// create another pointer that also points to i
int *ip2=ip // ok. now ip2 also addresses i
// to create a pointer that points  ip2
int *ip3=&ip // error, type mismatch
int **ip3=&ip2 // ok , it is a pointer to a pointer
```




```
int i=1024
int *ip=&i; //ip points to i
int k=ip; //error
int k=*ip; // k now contains 1024
```

In general:

```
int *ip=… //  type decleration for the pointer and initialization
*ip= … // the item pointed by ip
…=*ip // the item pointed by ip
 … =ip // the pointer itself
ip=... // the pointer itself
```

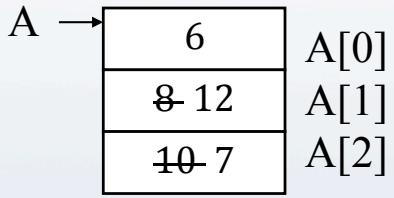



```
int i;
int k=-5;
int *ip=&i;
*ip=k; // i=k
*ip=abs( *ip) // i=abs(i);
// int: 4 bytes, char: 1 byte, double: 8 bytes
// these sizes depend on the system for which compiler is written
// ip points to an integer that is 4 bytes

ip=ip+1;
// The value of the address it contains is increased by the size of
// the object.,i.e. it is incremented by sizeof(int)=4
// However, some compilers needs it is to be written explicitly as  ip
    +1*sizeof(int)
```

```
int i,j,k;
int *ip=&i;
*ip= *ip+2 // add two to i, that is  i=i+2
ip=ip+2; // now it  contains ip+2* sizeof(int)
```

```
int A[3]={6,8,10};
*(A+2)=7;
*(A+1)=A[2]+5
```



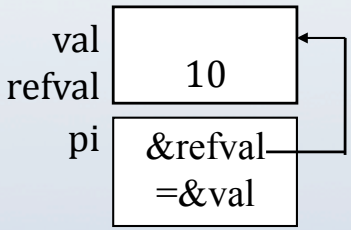
Reference Types:

All the operations applied to the reference act on the object to which it refers

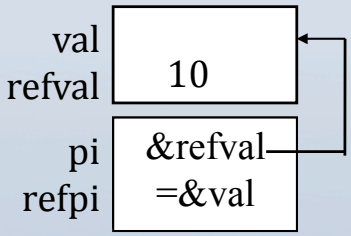
```
int val=l0
int  &refval=val
```

demonstrate reference as


```
int *pi=&refval   //initializes pi with the address of  val
```




```
int  *&refpi=pi   // refpi is a reference  to a pointer
```



```
int &*ptr=&refval          //USELESS
// ptr is a pointer  to  a reference,
//however  the address of a reference is same as the referenced
//variable, therefore instead  &*ptr simply use *ptr as below
int *ptr=&refval
// or
int *ptr=&val
```

A pointer argument can also be declared as a reference when the programmer wish to modify the pointer itself rather than the object addressed by the pointer

```
void prswap(int *&v1, int *&v2)
//NOTE:  *&v1 should be read from right to left  v1 is a reference to a pointer to
// an  object of type int
{ int temp=v2;
        v2=v1;
        v1=temp;
}
main()
{ int i=10;
   int j=20;
   int *pi=&i;
   int *pj=&j;

   cout << "Before swap(): i:"<<*pi<<"j:"<<*pj<<endl;
   prswap(pi,pj);
   cout << "After swap():  i:"<<*pi<<"j:"<<*pj<<endl;
}
```

When compiled and executed, we will have

```
Before swap(): *pi:10 *pj:20
After swap():    *pi:20 *pj:10
Return from prswap destroys v1, v2, temp
```

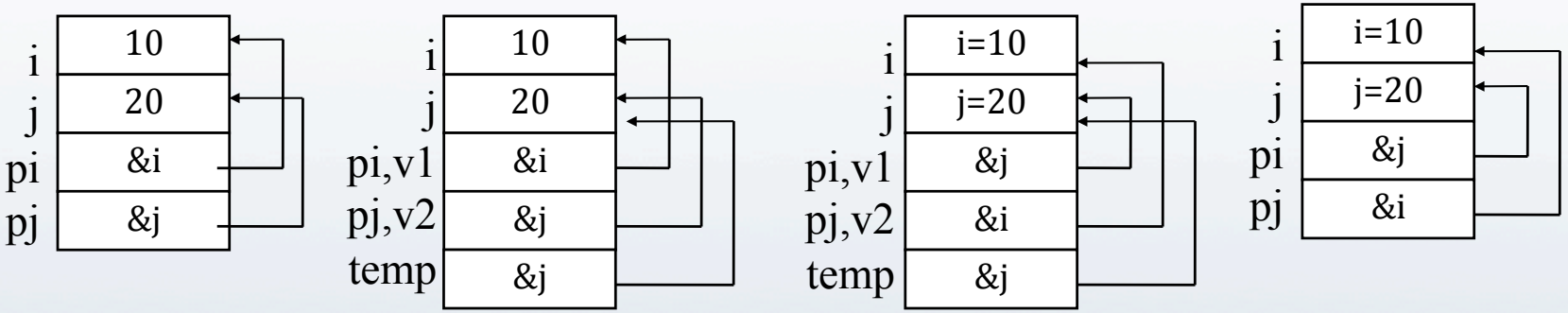
If traced we will have

int i=10
int j=20
int *pi=&i;
int *pj=&j

prswap (int *&v1,*&v2)
int *temp=v2;

v2=v1;
v1=temp;

Return from prswap destroys v1, v2, temp



before swap *pi: 10 *pj: 20
after swap *pi: 20 *pj:10

Notes:

```
const int* p; // pointer to a constant
  p=…  OK
*p=      not OK

int *  const  p; // a constant pointer
  p=…  not OK
*p=…   OK

const int* const p; // a constant pointer to a constant
  p=…   not OK
*p=…    not OK

const int &p; // reference to a constant
 p=…              not OK

int & const p; // constant reference
// useless, a reference is already a constant, you can not change shortcut
```

RECURSIVE FUNCTIONS.

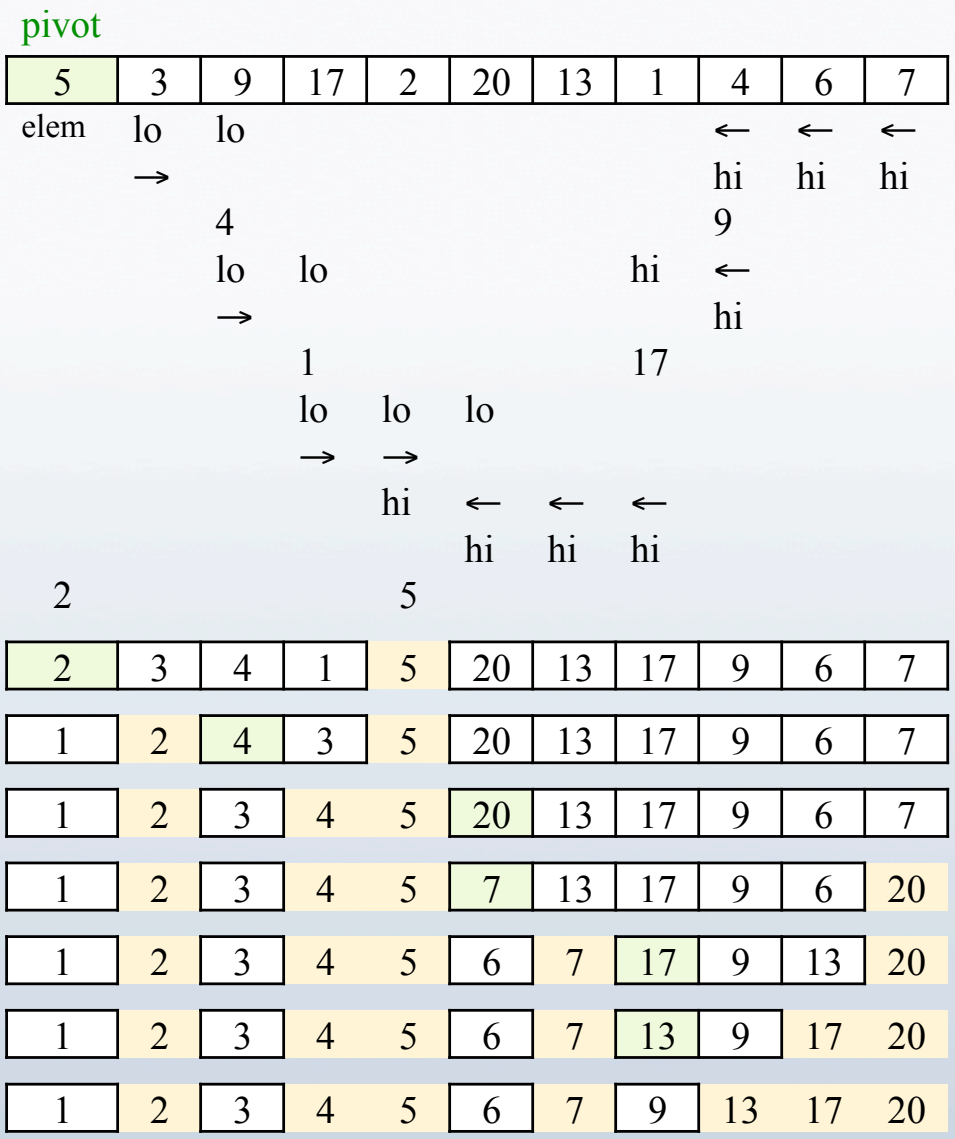
Some calculations have recursive character: for example N!=N*(N-1)!. To be able implement this kind of functions we write functions that call themselves. Such functions are called recursive functions.

Example:

```
int factorial (int n)
{if n>1
     return n*factorial(n-1)
else
    return 1
}
```

Example: Quick Sort is an algorithm for sorting numbers in a recursive way.

```
Template <classType>
void qsort(Type *ia, int low, int high)
{//quick sort
    if (low<high) {
       int lo=low+1;
       int hi=high;
       Type elem=ia[low];
       while (True)
       {
    while (ia[lo] <elem) {lo++};
          while (ia[hi] >elem) {hi--};
    if (lo<hi)
       swap(ia,lo,hi);
    else break;
       }
      swap(ia,low,hi);
      qsort(ia,low,hi-1);
      qsort(ia, hi+1,high);
   }
}
```



Quicksort makes use of a helping function swap (). This, too, needs to be defined as a template function:

```
template <class Type>
void swap (Type ia[], int i, int j)
{// swap two elements of an array
   Type tmp=ia[i];
    ia[i]=ia[j];
      ia[j]=tmp
}
```

or it can be implemented as

```
template <class Type>
void swap (Type *ia, int i, int j)
{// swap two elements of an array
   Type tmp=*(ia+i);
    *(ia+i)=*(ia+j)
      *(ia+j)=tmp
}
```

```
#include "qsort.c"
main(){
int    A[11]={5,3,9,17,18,20,13,2,4,6,7}
double B[5]={12.8,76.0,87.7,98.6, 65.7}
main ()
qsort(A,0,10)
qsort(B,0,4)
}
```