NWEN 241
Systems Programming

Sue Chard

suechard@ecs.vuw.ac.nz

Content

- Helpdesk **CO246** for assignment

- Recap
- Pointer Arithmetic

Recap Pointers

Pointer Declaration *

```
Data_type * myptr;        //declare a pointer variable
```

Address of operator &

```
myptr = &z;    //assign the memory address of variable z to myptr
```

Indirection /dereferencing operator *

```
*myptr = y;    //dereference myptr and assign the value from y
```

Recap Array Pointers

- The variable that you declare for an array is actually a fixed pointer to the first array element
- Pointers can also be used to access array elements

```
int z[10], *ip;

ip = &z[0];
z[0] = 7;

printf("z= %d \n",z);
printf("z[0] = %d \n",z[0]);
printf("&z[0] = %d \n",&z[0]);
printf("*z = %d \n",*z);
printf("ip = %d \n", ip);
printf("*ip = %d\n",*ip);
```

```
z= 6422264
z[0] = 7
&z[0] = 6422264
*z = 7
ip = 6422264
*ip = 7
```

Pointer Arithmetic

- Addition and subtraction can be performed on pointers , this is useful to iterate through arrays
- **++** Increments the pointer to the next element in the array
- **--** decrements the pointer to the previous element in the array
- **+** adds the specified number of positions in the array to the pointer eg. pointer + 4 moves 4 elements in the array
- **-** subtracts the specified number of positions in the array from the pointer
- You can also add or subtract two pointers of the same type

Iterating through an array incrementing each element

```
int j;
for(j = 0; j < n; j++)
     a[j]++;
```

vs

```
int *pj;
for(pj = a; pj < a + n; pj++)
       (*pj)++;
```

n is the number of elements in the array

pj is a pointer to an **int**

a is a pointer to the beginning of an array of **n** elements;

Start with **pj** pointing at **a**, i.e., **pj** points to **a[0]**

The loop iterates while **pj < a + n**

pj is a pointer, so it is an address

so **a+n** is the size of the array

pj++ increments the pointer to point at the next element in the array

The instruction **(*pj)++** says “take what **pj** points to and increment it”

Subtraction on pointers

```
int a[10] = {…};
int *ip;
for(ip = &a[9]; ip >= a; ip--)
```

Iterate through an array from last element to first element

Pass the address of the 3rd element of an array **`&a[2]`** to a function and use pointer subtraction to get to elements **`a[0]`** and **`a[1]`**.

```
int sub(int *ip)
{
    int temp;
    temp = *ip + *(ip – 1) + *(ip – 2);
    return temp;
}
```

To call the function

```
int a[3] = {5,7,9};
printf("%d",sub(&a[2]));
```

Example ways of coding a string copy function

```
void strcpy (char *s, char *t)
{
    int i = 0;
    while((s[i] = t[i]) != '\0')
        i++;
}
```

strcpy using indexes

```
void strcpy (char *s, char *t)
{
    while((*s = *t) != '\0')
    {
        s++;   t++;
    }
}
```

strcpy using pointers

```
void strcpy (char *s, char *t)
{
    while((*s++ = *t++) != '\0');
}
```

The conciseness of this last strcpy makes it hard to understand

Example ways of coding a string compare function

```
int strcmp (char *s, char *t)
 {
     int i;
     for(i=0;s[i] = = t[i];i++)
          if(s[i] = = '\0')
               return 0;
     return s[i] – t[i];
 }
```

Strcmp using indexes

```
int strcmp (char *s, char *t)
 {
     for ( ; *s = = *t; s++, t++)
          if (*s = = '\0') return 0;
     return *s - *t;
 }
```

strcmp using pointers

What is the return value of these functions?

If the strings are the same, return value is 0

If the strings are not the same, return value is *s - *t

Return value indicates lexicographical order of s and t: Positive if s appears after t, negative otherwise

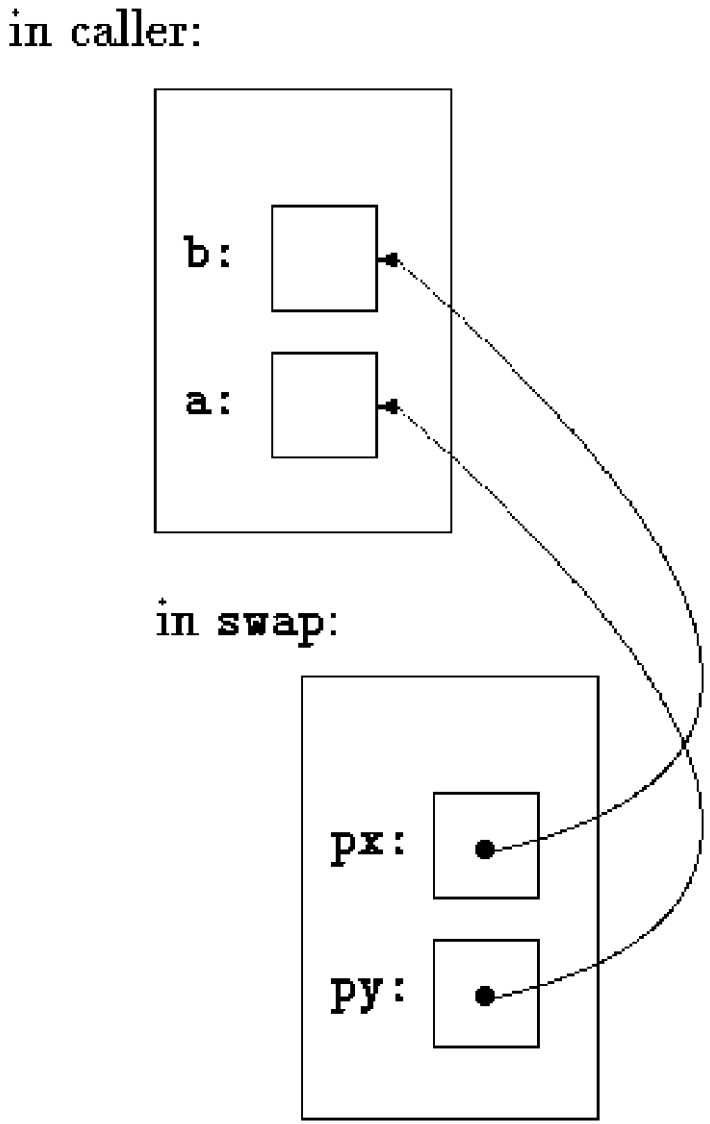
Reference / variable parameters

To make changes to a variable that persist after a function ends, pass the address of (a pointer to) the variable to the function (a reference parameter)

Then use the indirection operator inside the function to change the value the parameter points to:

```
In caller:
int a, b;
…
swap (&a, &b);

void swap(int *px, int *py)
{ int temp;
  temp = *px;
  *px = *py;        // values stored at
  *py = temp;       // addresses of 'a'
}
// and 'b' are swapped
```



Reference / variable parameters

A function can also return a pointer value

```
float *findMax(float A[], int N) {
  int I;
  float *theMax = &(A[0]);
  for (I = 1; I < N; I++)
    if (A[I] > *theMax) theMax = &(A[I]);
  return theMax;
}

void main() {
  float A[5] = {0.0, 3.0, 1.5, 2.0, 4.1};
  float *maxA;
  maxA = findMax(A,5);
  *maxA = *maxA + 1.0;
  printf("maxA %.1f A[4]%.1f\n",*maxA,A[4]);
}
```

The output is:

maxA 5.1 A[4] 5.1

Returning pointers from functions

Caution!!!

- In functions, do not do **`return p`**, where **`p`** is a pointer to a **local** variable

Recap:

- local variables are deallocated when the function ends
  - so whatever **`p`** is pointing to will no longer be available
  - but if you return the pointer, then you still are pointing at that memory location even though you no longer know what is there

Why can the local variable declared in the previous example as **`float *theMax = &(A[0]);`** be returned ?

Pointer to pointers

A pointer can also be made to point to a pointer variable (but the pointer must be of a type that allows it to point to a pointer)

Example:

```
int V = 101;
int *P = &V;                          /* P points to int V */
int **Q = &P;                         /* Q points to int pointer P */

printf("%d %d %d\n", V, *P, **Q); /* prints 101 3 times */
```

Pointer Types and Casting

Pointers are generally of the same size (enough bytes to represent all possible memory addresses), but it is generally inappropriate to assign an address of one type of variable to a pointer to a different type

Doing the following will generally result in a warning, not an error, because C will allow you to do this and it is appropriate in some situations

Example:

```
int V = 101;
float *P = &V; /* Generally results in a Warning not an error*/
```

When assigning a memory address of a variable of one type to a pointer that points to another type, it is best to use the cast operator to indicate the cast is intentional (this will remove the warning), but it is still unsafe to do this !!!

Example:

```
int V = 101;
float *P = (float *) &V;   /* Casts int address to float * */
```

General (void) pointers

A **`void *`** is considered to be a general pointer

No cast is needed to assign an address to a **`void *`** or from a **`void *`** to another pointer type

Example:

```
int V = 101;
void *G = &V; /* No warning */
float *P = G; /* No warning, still unsafe */
```

Certain library functions return **`void *`** results

A Useful quote to remember

*"Pointers have been lumped with the **goto** statement as a marvelous way to create impossible-to-understand programs. This is certainly true when they are used carelessly, and it is easy to create pointers that point somewhere unexpected. With discipline, however, pointers can also be used to achieve clarity and simplicity."*

– Kernighan and Ritchie, 1988.