Phys4051: C Lectures

Pointers and Variables: Definition

Variable:
- A variable refers to a memory location that contains a numerical value.

Pointer:
- A pointer refers to a memory location that contains an address.

Pointers: Operators (1)

Address Operator: &
- Note: it looks identical to the bitwise AND operator but it is used in a completely different way!
- Returns the address of a variable
- Example: \texttt{prt\_v = \& x};

Pointers: Operators (2)

Indirection Operator: *
- Note: it looks identical to the multiplication operator but it is used in a completely different way!
- Retrieves a value from the memory location the pointer points to.
- Example: \texttt{*ptr\_v = 77};

Pointer Declaration

A pointer must be declared and the variable type it points to must be specified:

\begin{verbatim}
short *aptr; //pointer declaration
double *bptr;
float* fptr; //same as float *fptr
\end{verbatim}

Assigning an Address to a Pointer (1)

An address is assigned to a pointer using the address operator: &

Example: \texttt{prt\_v = \& x};
Assigning an Address to a Pointer (2)

Example:

```c
short x = 33;
short *aptr; //pointer declaration
aptr = & x;
```

<table>
<thead>
<tr>
<th>x</th>
<th>aptr</th>
</tr>
</thead>
<tbody>
<tr>
<td>33</td>
<td>5012</td>
</tr>
</tbody>
</table>

(The memory addresses are arbitrarily chosen)


Example:

```c
short x = 33;
short *aptr; //declare the ptr
aptr = & x; //ptr points to x
*aptr = -123; //assign a value to “x”
```

<table>
<thead>
<tr>
<th>x</th>
<th>aptr</th>
</tr>
</thead>
<tbody>
<tr>
<td>-123</td>
<td>5012</td>
</tr>
</tbody>
</table>

(The memory addresses are arbitrarily chosen)

Pointer Usage with a Variable:

The following two segments are equivalent in respect what they do to variable “x”:

```c
short x = 33;
short *aptr;
aptr = & x;
*aptr = -123;
short x = 33;
x = -123;
```

Pointers: Additional Comment

Pointers refer to an address which is almost always the address of another variable.

An (arbitrary) address can be directly assigned to a pointer. Doing so makes the program less portable and can be very dangerous!

Example:

```c
short *aptr = 0x300;
*aptr = 0xff; //0x… means HEX
```

Arrays and Pointers

Pointers are most often used in function calls and with arrays.

Because pointers are so often used with arrays, a special pointer has been designated in C to point to the “zeroth” element in an array: the array name itself!

Arrays & Pointers: Pointer to the “Zeroth” Array Element

Example 2a:

```c
float w[128];
float *w_ptr;
w_ptr = & w[0];
```

Example 2b:

```c
float w[128];
float *w_ptr;
w_ptr = w;
```
Each time you declare an array, you also declare implicitly a pointer to the “zeroth” element!

The name of this pointer is the name of the array!

Arrays: Memory Allocation

Arrays: Memory Allocation: Pointer Math (1)

Arrays: Memory Allocation: Pointer Math (2)

Arrays: Memory Allocation: Pointer Math (3)

The following two segments are equivalent statements:

Segment 3a:

```c
short x, y[MAX], val;
y[x] = val;
*(y + x) = val;
```

Segment 3b:

```c
short x, y[MAX], val;
y[x] = val;
```

Functions, Arrays and Pointers

When an array is passed to a function it is passed by reference, i.e., a pointer to the array is passed!

Ex:

```c
short sAr[MAX]; //declare array
SortAr(sAr, MAX); //call function
```
Arrays and Function Calls: Example: Initialize an Array

Write a function that will initialize an entire array, i.e., set each element to 0.

```c
void Init( double* ar, int n )
{
    int i;
    for( i = 0; i < n; i++)
        ar[i] = 0;  //pntr. or arr?
}
```

Functions, Arrays and Pointers: Example 3

A) Write a function that returns the average of an array of type double.
B) Write a program that uses above function.
C) Change the program above to account for the fact that we want to ignore the first 5 data points in the array when calculating the average.

```
double Ave( double* dar, int n )
{
    int i;
    double dtot = 0;
    for( i = 0; i < n; i++)
        dtot += dar[i];  //pntr. or arr?
    return( dtot/n);
}
```

Functions, Arrays and Pointers: Example 3 B

```c
main()
{
    double dAve, dTemp[1000];
    //lines of code to fill array dTemp are omitted here...
    dAve = Ave( dTemp, 1000);
    printf("%d", dAve);
}
```

Functions, Arrays and Pointers: Example 3 C

```c
main()
{
    double dAve, dTemp[1000];
    //lines of code to fill array dTemp are omitted here...
    dAve = Ave( dTemp + 5, 995);
    printf("%d", dAve);
}
```

Example 4: Function to Sort an Array (1): Problem

Assignment:

Write a function that sorts the values contained in an array.
Example 4: Function to Sort an Array (2): Solution

*Pass arrays whenever possible by reference! (Also, no need for global arrays!)

*Passing an array by value takes a long time (and lots of space) because the computer has to make a copy of the array to pass it to the function.

Ex. 4: Function to Sort an Array (3): main

```c
#define MAX 10
void SortAr( short *volt, short n );
main(){
    short i, sAr[ MAX ];
    for( i = 0; i < MAX; i++){
        sAr[i] = rand();
        printf("%d %d\n", i, sAr[i] );
    }
    SortAr( sAr, MAX ); // pass by ref
    for( i = 0; i < MAX; i++)
        printf("%d %d\n", i, sAr[i] );
}
```

Ex. 4: Function to Sort an Array (4): Sort Function V1

```c
void SortAr( short *volt, short n ){
    short x, y, stemp;
    for( y = 0; y < n - 1; y++){  
        for( x = 0; x < n - 1 - y; x++ ){  
            if( volt[ x ] > volt[ x + 1 ] ){  
                stemp = volt[x];
                volt[x] = volt[x+1];
                volt[x+1] = stemp;
            }
        }
    }
}
```

Ex. 4: Function to Sort an Array (5): Sort Function V2

```c
void SortAr( short *volt, short n ){
    short x, y, stemp;
    for( y = 0; y < n - 1; y++){  
        for( x = 0; x < n - 1 - y; x++ ){  
            if( *(volt+x) > *(volt+x+1) ){  
                stemp = *(volt+x);
                *(volt+x) = *(volt+x+1);
                *(volatile+x+1) = stemp;
            }
        }
    }
}
```

Example: Uniqueness (1)

*You are given an (integer) array with N elements. Count how many of the values in this array are unique and copy the unique values from this first array into a second array.

*For example, if the first array contained: 8, 7, 7, 3, 8, 5, 3 then it has 4 unique entries and the second array should contain the following values: 8, 7, 3, 5.

Example: Uniqueness Hints:

To check for uniqueness, you must compare each element of the first array with every element of the second array. Specifically:

* If the second array already contains an entry identical to the one you are checking (in the first array) the then go on and check the next element in the first array.

* If the second array does NOT contain an entry identical to the one you are checking (in the first array) then:
  - Copy the value from the first array to the end of the second array.
  - Increment the number of elements that the second array contains.
**Pointers: Function Calls and Function Arguments**

Variables can be passed to a function (as function arguments) either:

a) **by value**
   - (as a copy of a local variable)

b) **by reference**
   - (by a pointer)

**Function Arguments: Passing by Value**

- This is the method you have used so far in these examples.
- A (local) copy of the variable is passed to function.
- Changing the (passed) variable within the calling function has no effect on the (original) variable that was passed.

---

**Ex. 1: Exchange Two Variables: By Value**

Problem:

a) Write a function “Xchange” that will exchange two variables if the “first” variable is greater than the “second” one.

b) You are not allowed to use GLOBAL variables!

---

**Ex. 1a: Exchange Two Variables (2): main**

```c
void Xchange_ByVal(short u, short v);
main()
{
    short x = 10, y = 2;
    Xchange_ByVal(x, y);
    printf("x: %d  ", x);
    printf("y: %d
", y);
}
```

---

**Ex. 1a: Exchange Two Variables (3): Function**

```c
void Xchange_ByVal(short u, short v)
{
    short stemp;
    if( u > v){
        stemp = v; v = u; u = stemp;
    }
    printf("u: %d  ", u);
    printf("v: %d\n", v);
}
```

---

**Ex. 1a: Exchange Two Variables (4): Output**

Output:

```
u:  
v:  
```

Conclusion: Works? (y/n)

Why (not)?
Function Arguments: Passing by Reference

- Allows you to change the value of a variable which is not local to the function without having to make it global.
- Pass a reference (a pointer) to the function which tells the function where “to find” that variable.
- Note: (usually) you don’t change the reference, you change only what the reference points to!

Ex. 1b: Exchange Two Variables (5): By Reference

- **Solution:**
  Pass the function arguments by reference!

Ex. 1b: Exchange Two Variables (6): main

```c
void Xchange_ByRef(short *u, short *v);
main()
{
    short x = 10, y = 2;
    Xchange_ByRef(&x, &y);
    printf("x: %d  ", x);
    printf("y: %d\n", y);
}
```

Ex. 1b: Exchange Two Variables (7): Function

```c
void Xchange_ByRef(short *u, short *v)
{
    short stemp;
    if( *u > *v)
    {
        stemp = *v; *v = *u;
        *u = stemp;
    }
    printf("u: %d  ", *u);
    printf("v: %d\n", *v);
}
```

Ex. 1b: Exchange Two Variables (8): Output

- **Output:**
  
  u:   v: 
  x:   y: 

- **Conclusion:** Works? (y/n)

Function Arguments and Pointers: Summary

- **Passing (a Variable) by Value:**
  - Variable is local to function and, therefore, can not alter the original value.

- **Passing (a Variable) by Reference**
  - Since a reference to the variable is passed, the original value can be accessed and altered.
Summary: Function Calls

a) by Value

Passing a Variable by Value:
void FbyVal( int );
main()
{
    int y = 3;
    FbyVal(y);
}

Passing a Pointer by Value:
void FbyVal( int );
main()
{
    int x = 3;
    int* y = &x;
    FbyVal(y);
}

---

Summary: Function Calls

b) by Reference

Passing a Variable by Reference:
void FbyRef( int* );
main()
{
    int y = 3;
    FbyRef(y);
}

Passing a Pointer by Reference:
void FbyRef( int* );
main()
{
    int x = 3;
    int* y = &x;
    FbyRef(y);
}