BIL 104E Introduction to Scientific and Engineering Computing

Lecture 8

Address (Left Value) vs. Content (Right Value)

- A memory location inside the computer may contain an **instruction** (an action to be taken by the computer) or it may contain **data** (an item that is acted upon by the action of the instruction).

- Each memory location has a unique address so that the computer can read from or write to the memory location without any confusion.

- When a variable is **declared**, a piece of unused memory will be reserved for the variable, and the unique address to the memory will be associated with the name of the variable. The address associated with the variable name is usually called the **left value** of the variable.

- Then, when the variable is **assigned** a value, the value is stored into the reserved memory location as the content. The content is also called the **right value** of the variable.

Address (Left Value) vs. Content (Right Value)

- In C, all variables have four important attributes:
  1. name
  2. type (which determines their size)
  3. value (direct reference)
  4. storage location (or address)

- **Example**:

```
int x;
x = 7;
/*Left value: 1000, Right value: 7    */
```

The left value, 1000, is the address of the memory location reserved for x. Note that depending on computers and operating systems, the left value of x can be different from one machine to another.

The right value, 7, is the content stored in the memory location.

The Address of Operator (&)

- Address-of operator (**&**) returns the address (i.e. left value) of a variable.

```
include <stdio.h>
main(){
        char c;int x;float y;
        printf("c: address=0x%p, content=%c\n", &c, c);
        printf("x: address=0x%p, content=%d\n", &x, x);
        printf("y: address=0x%p, content=%5.2f\n", &y, y);
        c = 'A'; x = 7;y = 123.45;
        printf("c: address=0x%p, content=%c\n", &c, c);
        printf("x: address=0x%p, content=%d\n", &x, x);
        printf("y: address=0x%p, content=%5.2f\n", &y, y);
        return 0;
}
```

```
c: address=0x1AF4, content=@
x: address=0x1AF2, content=-32557
y: address=0x1AF6, content=0.00
c: address=0x1AF4, content=A
x: address=0x1AF2, content=7
y: address=0x1AF6, content=123.45
```

Pointers

- Using pointers is simply another method of getting data in and out of computer's memory.

- But unlike other methods this is a very powerful one that provides great programming flexibility.

- A **pointer** is a type of variable that can hold the address of another variable in memory. In addition to the four attributes common to all variables, pointers have a fifth attribute called an **indirect value**. So pointer variables have:
    1. name
    2. type (which determines their size)
    3. value (direct reference)
    4. storage location (or address)
    5. indirect value

Pointer Declaration

- Pointer declarations are composed of three parts – the data type, an asterisk * (or star) character, and the pointer name itself.
  - **Example**:

```
int *countPtr;
```

- Multiple pointers require using a ***** before each variable declaration.
  - **Example**:

```
int *countPtr1, *countPtr2;
```

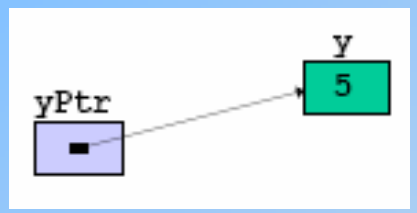
- Pointers to any data type can be declared.
  - **Example**:

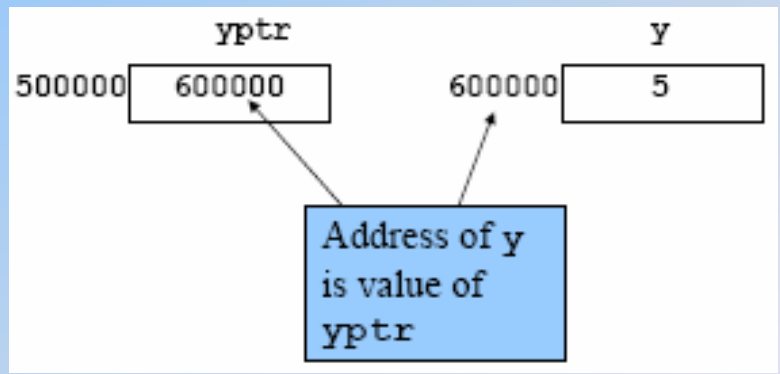
```
char *ptr_c;        /* declare a pointer to a character      */
int   *ptr_int;     /* declare a pointer to an integer       */
float *ptr_flt;     /* declare a pointer to a float          */
```

Pointer Initialization

- A pointer can be initialized to the address of a valid object of the specified type.

```
int y = 5;
int *yPtr;
yPtr = &y;
/* yPtr gets the address of y */
```





NULL Pointer

- A NULL pointer setting indicates that a pointer does not point to an object.
- Although a NULL pointer does not point to any object, an uninitialized pointer might point to anywhere.
- The NULL macro is defined in stdio.h.

```
#include <stdio.h>
int main()
{
      int *ipPtrToInt1, *ipPtrToInt2 ;
      ipPtrToInt1 = NULL;
      ipPtrToInt2 = 0;
}
```

Void Pointer

- Unlike regular pointers a ***void pointer*** can point to any type in C. A void pointer can hold any type of pointer, and any pointer can be assigned to a void * without a cast

```
int iVal1;
char cIn;
void *vpHold;
char *cpChar = &cIn;
int *ipValue = &iVal1;
int *ipExtent;
vpHold = ipValue;
ipValue = (int *) vpHold;
ipExtent = (int *) vpHold;
```

In the final two statements a cast is required when a void * is assigned to any other type of pointer. This makes sense since the void * caries no type information with it.

Indirection/Dereferencing Operator

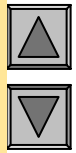
- The indirect value of a pointer is obtained by preceding a pointers name with the ***** operator. The result is that the actual value of the object referenced by the pointer. * operator can also be used for assignment.
- **Example:**

```
int x, y = 5;
int *yPtr;
yPtr = &y;          /* yPtr gets the address of y */

x = *yptr;          /* *yptr returns y (since yptr points to y) */
/* OR  */
x = y;              /* equivalent to the above line */

*yPtr = 7;         /* changes y to 7   */
/* OR  */
y = 7;              /* equivalent to the above line */
```

- ***** and **&** are inverses, they cancel each other out.

```c
#include <stdio.h>
main() {
    char c, *ptr_c;
    int x, *ptr_x;
    float y, *ptr_y;
    c = 'A'; x = 7; y = 123.45;
    ptr_c = &c; ptr_x = &x; ptr_y = &y;

    printf("c: address=0x%p, content=%c\n", &c, c);
    printf("x: address=0x%p, content=%d\n", &x, x);
    printf("y: address=0x%p, content=%5.2f\n", &y, y);
    printf("ptr_c: address=0x%p, content=0x%p\n", &ptr_c, ptr_c);
    printf("*ptr_c => %c\n", *ptr_c);
    printf("ptr_x: address=0x%p, content=0x%p\n", &ptr_x, ptr_x);
    printf("*ptr_x => %d\n", *ptr_x);
    printf("ptr_y: address=0x%p, content=0x%p\n", &ptr_y, ptr_y);
    printf("*ptr_y => %5.2f\n", *ptr_y);
    return 0;
}
```

```
c: address=0x1B38, content=A
x: address=0x1B36, content=7
y: address=0x1B32, content=123.45
ptr_c: address=0x1B30, content=0x1B38 *ptr_c => A
ptr_x: address=0x1B2E, content=0x1B36 *ptr_x => 7
ptr_y: address=0x1B2C, content=0x1B32 *ptr_y => 123.45
```



Outline

```
1  /* Fig. 7.4: fig07_04.c
2     Using the & and * operators */
3  #include <stdio.h>
4
5  int main()
6  {
7     int a;        /* a is an integer */
8     int *aPtr;    /* aPtr is a pointer to an integer */
9
10    a = 7;
11    aPtr = &a;    /* aPtr set to address of a */
12
13    printf( "The address of a is %p"
14            "\nThe value of aPtr is %p", &a, aPtr );
15
16    printf( "\n\nThe value of a is %d"
17            "\nThe value of *aPtr is %d", a, *aPtr );
18
19    printf( "\n\nShowing that * and & are inverses of "
20            "each other.\n&*aPtr = %p"
21            "\n*&aPtr = %p\n", &*aPtr, *&aPtr );
22
23    return 0;
24 }
```

The address of **a** is the value of **aPtr.**

Declare variables

Initialize variables

The ***** operator returns an alias to what its operand points to. **aPtr** points to **a**, so ***aPtr** returns **a**.

Notice how ***** and **&** are inverses

Program Output

```
The address of a is 0012FF88
The value of aPtr is 0012FF88

The value of a is 7
The value of *aPtr is 7
Showing that * and & are inverses of each other.
&*aPtr = 0012FF88
*&aPtr = 0012FF88
```

```
/* Updating Variables via Pointers */
*/ As long as a variable is linked up to a pointer variable, the value of the variable can be
     obtained by using the pointer variable. In other words, the value can be read by
     pointing to the memory location of the variable and using the dereferencing operator.
     */
#include <stdio.h>
main(){
      char c, *ptr_c;
      c = 'A';
      printf("c: address=0x%p, content=%c\n", &c, c);
      ptr_c = &c;
      printf("ptr_c: address=0x%p, content=0x%p\n", &ptr_c, ptr_c);
      printf("*ptr_c => %c\n", *ptr_c);
      *ptr_c = 'B';
      printf("ptr_c: address=0x%p, content=0x%p\n", &ptr_c, ptr_c);
      printf("*ptr_c => %c\n", *ptr_c);
      printf("c: address=0x%p, content=%c\n", &c, c);
      return 0;
}
```

```
c: address=0x1828, content=A
ptr_c: address=0x1826, content=0x1828
*ptr_c => A
ptr_c: address=0x1826, content=0x1828
*ptr_c => B
c: address=0x1828, content=B
```

```c
/* Pointing to the Same Thing */
/* So, a memory location can be pointed to by more than one pointer. */

main(){
        int x;
        int *ptr_1, *ptr_2, *ptr_3;
        x = 1234;
        printf("x: address=0x%p, content=%d\n", &x, x);
        ptr_1 = &x;
        printf("ptr_1: address=0x%p, content=0x%p\n", &ptr_1, ptr_1);
        printf("*ptr_1 => %d\n", *ptr_1);
        ptr_2 = &x;
        printf("ptr_2: address=0x%p, content=0x%p\n", &ptr_2, ptr_2);
        printf("*ptr_2 => %d\n", *ptr_2);
        ptr_3 = ptr_1;
        printf("ptr_3: address=0x%p, content=0x%p\n", &ptr_3, ptr_3);
        printf("*ptr_3 => %d\n", *ptr_3);
        return 0;
}
```

```
x: address=0x1838, content=1234
ptr_1: address=0x1834, content=0x1838
*ptr_1 => 1234
ptr_2: address=0x1836, content=0x1838
*ptr_2 => 1234
ptr_3: address=0x1832, content=0x1838
*ptr_3 => 1234
```

```c
/* A pointer can be defined to point to another pointer.*/
/* Demonstrate use of pointer to pointer*/

#include <stdio.h>
int main ( void ) {
        int iValue = 10;
        int *ipPtrToInt = &iValue;
        int **ippPtrToPtrToInt = &ipPtrToInt;
        printf (" Direct Access : iValue = %4d\n", iValue );
        printf (" Via *ipPtrToInt : iValue = %4d\n", *ipPtrToInt );
        printf (" Via **ippPtrToPtrToInt : iValue = %4d\n", **ippPtrToPtrToInt );
}
```

```
Direct Access : iValue =   10
Via *ipPtrToInt : iValue =   10
Via **ippPtrToPtrToInt : iValue =   10
```