DECLARAING AND INITIALIZING POINTERS

Passing arguments – Call by Address

Introduction to Pointers

- Within the computer's memory, every stored data item occupies one or more contiguous memory cells (i.e., adjacent bytes).
- The number of cells required to store a data item *depends on the type of data item*. For example, a single character will typically be stored in one byte (8 bits) of memory; an integer usually requires two contiguous bytes; a floating-point number may require four contiguous bytes.

Introduction to Pointers

- Suppose v is a variable that represents some particular data item. The compiler will automatically assign memory cells for this data item.
- The data item can then be accessed if we know the location (i.e., the address) of the first memory cell.
- The address of v's memory location can be determined by the expression &v, where & is a unary operator, called the ***address operator***, that evaluates the address of its operand.

Introduction to Pointers

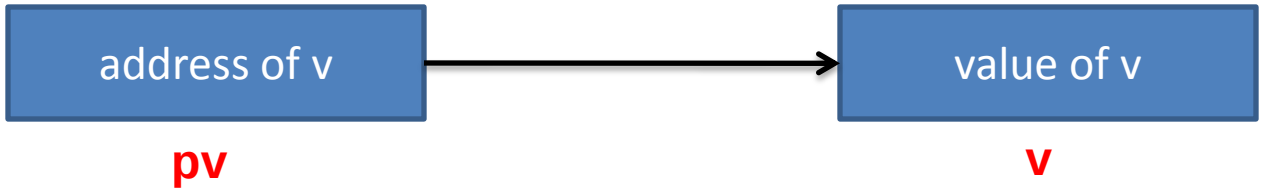
- Now let us assign the address of v to another variable, **pv**. Thus,

```
pv = &v;
```

- This *new variable is called a pointer to v*, since it "points" to the location where v is stored in memory.
- Remember, however, that pv represents v's address, not its value. Thus pv is referred to as a pointer variable.

Introduction to Pointers

- The relationship between pv and v is illustrated in Fig.



- Relationship between pv and v (where pv = &v and v = *pv)

Introduction to Pointers

- *So a pointer is a variable that represents the location (rather than the value) of a data item, such as a variable or an array element.*

Pointer Declarations

- Pointer variables, like all other variables, must be declared before they may be used in a C program.
- The interpretation of a pointer declarations differs, however, from the interpretation of other variable declarations.
- **When a pointer variable is declared, the variable name must be preceded by an asterisk (*). This identifies the fact that variable is a pointer.**

Pointer Declarations

- The data type that appears in the declaration refers to the object of the pointer, i.e., *the data item that is stored in the address represented by the pointer*, rather than the pointer itself.
- **Thus, a pointer declaration may be written in general terms as**

```
data-type   *ptvar;
```

Where ptvar is the name of the pointer variable, and data – type refers to the data type of the pointer's object.

Passing Arguments (Pointers) to a Function – Call By Address

- Pointers are often passed to a function as arguments. This allows data items within the calling portion of the program to be **accessed by the function**, **altered within the function**, and then returned to the calling portion of the program in altered form. We refer this use of pointers as passing arguments by reference (or by address or by location), in contrast to passing arguments by value as discussed earlier.

Passing Arguments (Pointers) to a Function – Call By Address

- When an argument is passed by value, the data item is copied to the function. *Thus, any alteration made to the data item within the function is not carried over into the calling routine*.
- When an argument is passed by reference, however, the address of data item is passed to the function. The *contents of that address can be accessed freely*, either within the function or within the calling routine.

Passing Arguments (Pointers) to a Function – Call By Address

- Moreover, any change that is made to the data item (i.e., to the contents of the address) will be recognized in both the function and the calling routine.
- Thus, the use of a pointer as a function argument permits the corresponding data item *to be altered globally from within the function*.

Passing Arguments (Pointers) to a Function – Call By Address

- When pointers are used as arguments to a function, some care is required with the formal argument declarations within the function.
- **Specifically, formal pointer arguments that must each be preceded by an asterisk.**

Passing Arguments to a Function – Call By Address - Example

- Here is a simple C program containing a function that alters the value of its argument:

```
# include <stdio.h>
void modify ( int *ptr );
void  main (void) {
int a = 2;
printf (“\na=%d ( from main, before calling the function)”,a);
modify(&a);
printf (“\na=%d ( from main, after calling the function)”,a);
}
```

```
void modify ( int *ptr)
{
*ptr += 3;
printf(“\n\na = %d ( from the function, after being modified )”, *ptr);
return;
}
```

Passing Arguments to a Function – Call By Address - Example

- The original value of a (i.e., a = 2) is displayed when main begins execution.
- The address of variable a is then passed to the function modify, where 3 is added 3 to it and the **new value** will be displayed.

Passing Arguments to a Function – Call By Address - Example

- Note that it is the *altered value of the formal argument* that is displayed within the function.
- Finally, the value of a within main (i.e., the actual argument) is again displayed, after control is transferred back to the main from modify.

Passing Arguments to a Function – Call By Address - Example

- When the program is executed, the following output is generated:

a =2 (from main, before calling the function)
a =5 (from the function, after being modified)
a =5 (from main, after calling the function)

- These results show that a is altered within the main, and the corresponding value of a is changed within modify.

Passing Arguments to a Function – Call By Address - Important

- **The following points must be noted about passing arguments using call by address mechanism:**

- The actual arguments can only be variables.
- When the control is transferred from the calling function to the called function, the memory for formal arguments and local variables is **allocated**, address *of the actual arguments are substituted in the corresponding formal arguments*, and the statements in the function body are executed.

Passing Arguments to a Function – Call By Address - Important

- **The following points must be noted about passing arguments using call by value mechanism:**

- As soon as the called function finishes its execution, the **memory allocated for it is de-allocated**, i.e., the values of formal arguments and local variable are destroyed, and finally the control is transferred back to the calling function.
- Any change made to the formal arguments will have immediate effect on actual arguments, since the function will be working on actual arguments through pointers.

Call By Value – Call By Address - DIFFERENCE

Call by Value	Call by Address
The actual arguments can be constants, variables, or expressions.	The actual arguments can only be variables.
The values of actual arguments are substituted in formal arguments which are ordinary variables.	The address of the actual arguments are substituted in formal arguments which are pointer variable.
Any change made to the formal arguments will have no effect on actual arguments, since the function will only be using the local copy of the arguments.	Any change made to the formal arguments will have immediate effect on actual arguments, since the function will be working on actual arguments through pointers.

Practice Questions

- Predict the result of the following code segment:

```
void test(int *address)
{
    printf("%u : %u : %u", address, &address, *address);
}
void main(void)
{
    int value = 50;
    test(&value);
}
```

Practice Questions

- Predict the result of the following code segment:

```
void main(void)
{
    int value = 50;
    int *address = &value;
    printf("%d : %d : %d", address, &address, *address);
}
```

Also state why the output of this code differs from that of previous problem.

Practice Questions

- Fill in the blank with suitable expression that performs the task equivalent to the statement: int *address = &value;

  int *address;

  _____= &value ;

- Fill in the blanks in the context of statement:

  int *address = &value;

  – The expressions _____ and ***address** will return same value.

  – The expressions **&address** will return address of _____.

  – The expression **address** and _____ will return same value/address.

Practice Questions

- What is wrong with the following code segment?

```
void testptrs(int iptr, float *fptr, char *cptr)
{
   printf("The address of i is: %u", iptr);
   printf("The value of f is: %f", fptr);
   printf("The address of c is: %u", cptr);
}
void main()
{
   int i = 100; float f = 35.79, char c = '$';
   testprts(&i, &f, &c);
}
```

Practice Questions

- Write the problem given in LECTURE 13 using pointers rather than returning values from functions.