UESTC1005 - Introductory Programming

Pointers

Lecture 11

Dr Ahmed Zoha

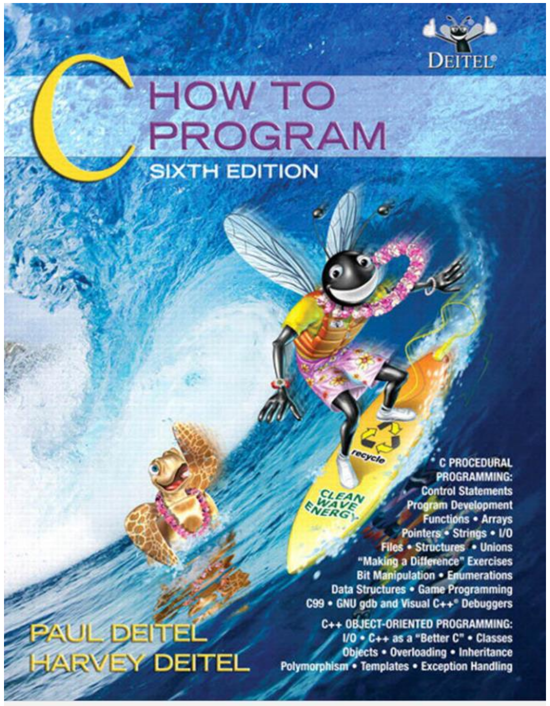
Lecturer in Communication Systems

School of Engineering

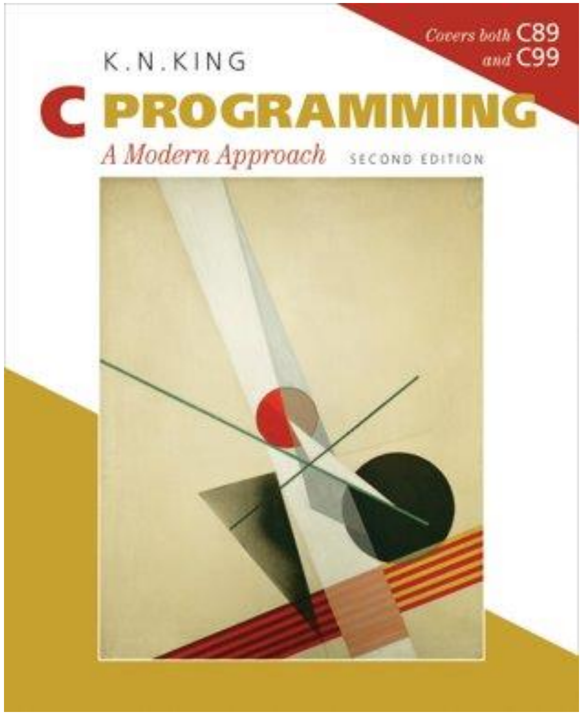
University of Glasgow

Email: ahmed.zoha@Glasgow.ac.uk

Recommended Reading Week 14/15



C How to Program (DETEL)
Chapter 7
Chapter 10, 12
and
Do Exercise



KING C Programming
Chapter 11
Chapter 12, 16, 17

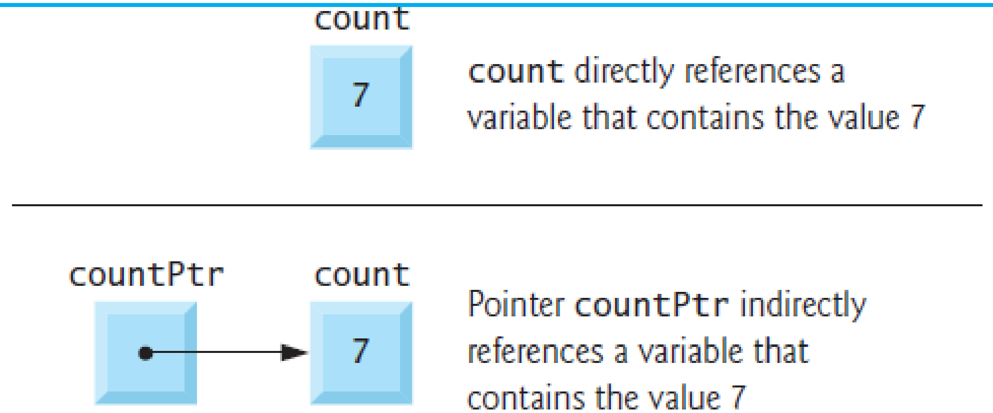
Introduction (RECAP)

- Pointers the most powerful features of C programming language
- Pointers are among C’s most difficult capabilities to master
- Concept of Pointer is very similar to the concept of **indirection**
  - **Example: Purchasing a PC at University of Glasgow (Your Request > IT Department> Vendor) This is an example of indirection**

Pointer Variable Definitions

- In programming languages indirection is the ability to reference something using a name, reference or container instead of a value.
  - Example: Variable names directs to a variable value
  - Pointer provides an indirect means of accessing a value of a particular item.
- Pointers are variables whose values are memory addresses. The main difference between pointers and variables are
  - **A variable directly contains a specific value**
  - **A pointer contains an address of a variable**

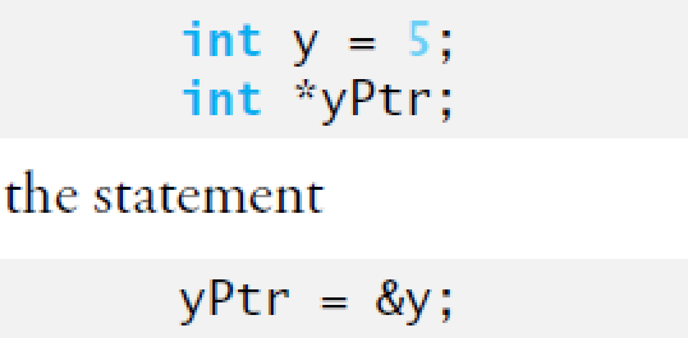
Indirection through Pointers


count
7
count directly references a variable that contains the value 7
countPtr
count
7
Pointer countPtr indirectly references a variable that contains the value 7

- A variable name directly references a value, and a pointer indirectly references a value.
- Referencing a value through a pointer is called indirection.
- The definition: int *countPtr;
- Specifies that variable countPtr is of type int * (i.e., a pointer to an integer)
- **It is read: "countPtr is a pointer to int" or "countPtr points to an object of type int."**

Pointer Operators

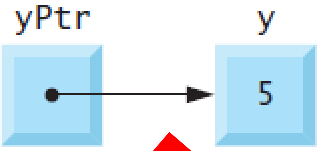
- The &, or address operator, is a unary operator that returns the address of its operand. E.g.,



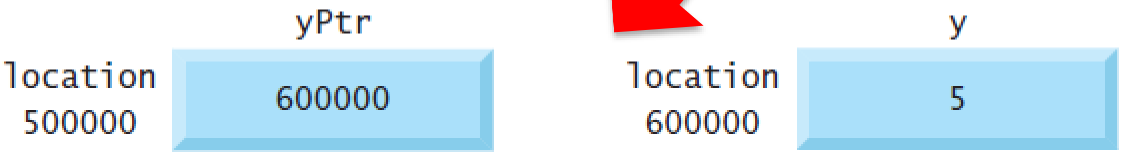
```
int y = 5;
int *yPtr;
```

the statement

```
yPtr = &y;
```



- assigns the address of the variable y to pointer variable yPtr.



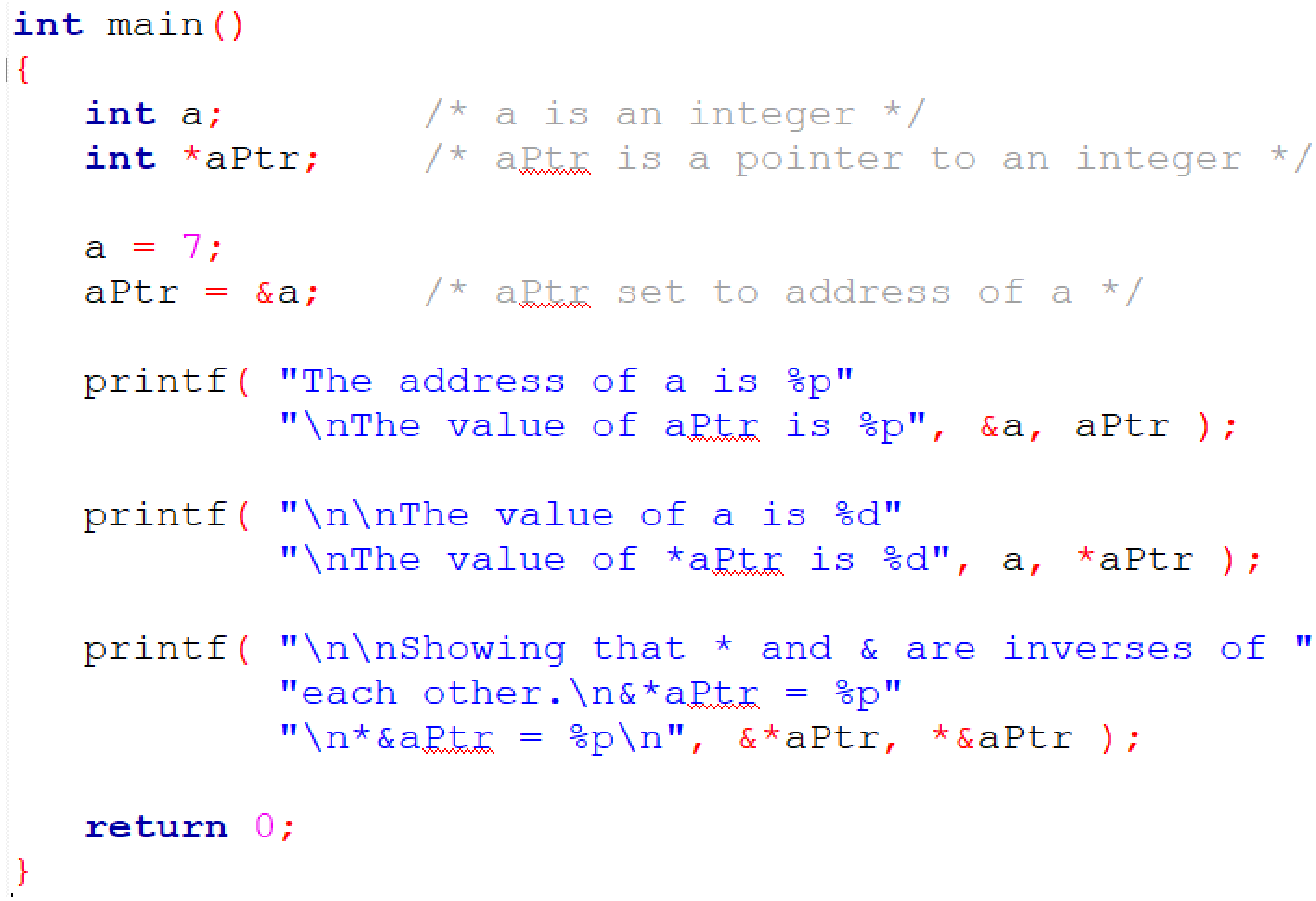
Pointer Operators and Declarations

- The unary * operator referred to as the indirection operator or dereferencing operator, returns the value of the object to which its operand (i.e., a pointer) points. E.g.,

  printf(**"%d"**, *yPtr); prints the value of variable y, namely 5.

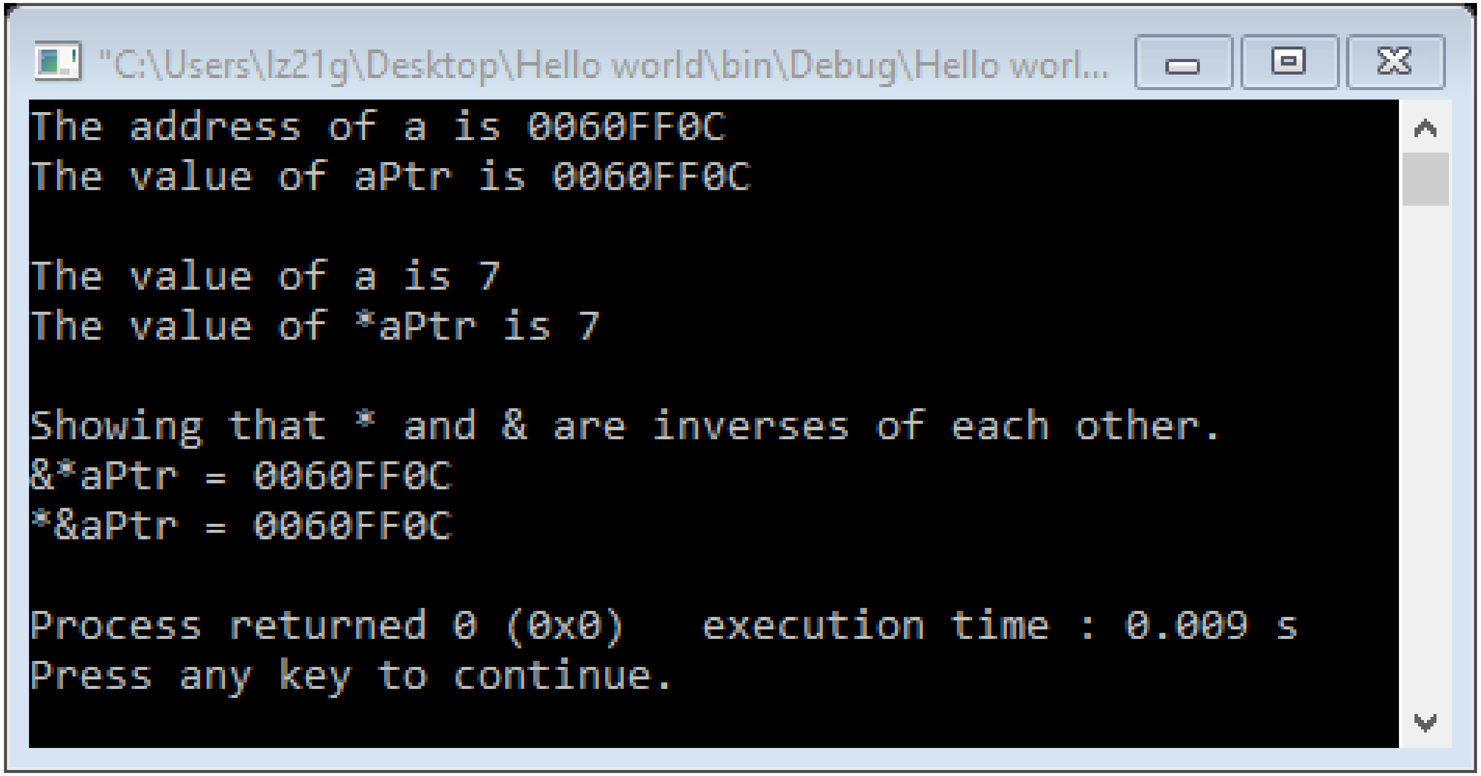
  printf(**"%p"**, yPtr); prints the value of pointer yPtr
- Declaring Pointers: the space between * and the pointer is optional
  - Convention: Programmers use * while declaring and omit the space while dereferencing the pointer
- Value of a pointer: unsigned integer
  - However you shouldn’t think of pointers as integers, Why?

Example



```
int main()
{
    int a;          /* a is an integer */
    int *aPtr;      /* aPtr is a pointer to an integer */

    a = 7;
    aPtr = &a;      /* aPtr set to address of a */

    printf( "The address of a is %p"
            "\nThe value of aPtr is %p", &a, aPtr );

    printf( "\n\nThe value of a is %d"
            "\nThe value of *aPtr is %d", a, *aPtr );

    printf( "\n\nShowing that * and & are inverses of "
            "each other.\n&*aPtr = %p"
            "\n*&aPtr = %p\n", &*aPtr, *&aPtr );

    return 0;
}
```

Example



```
The address of a is 0060FF0C
The value of aPtr is 0060FF0C

The value of a is 7
The value of *aPtr is 7

Showing that * and & are inverses of each other.
&*aPtr = 0060FF0C
*&aPtr = 0060FF0C

Process returned 0 (0x0)   execution time : 0.009 s
Press any key to continue.
```

- Address of a and the value of aPtr are identical.
- The & and * operators are complements of one another, when they are applied consecutively in either order, same result printed.

NULL Pointer

- Remember to declare a pointer always (refer to previous example)
- What if you have no address to initialize a pointer??
  - int * pnumber = **NULL;**
- **NULL** is the constant that is defined by the standard library
  - **It is equivalent to zero for a pointer**
  - **It guarantees that the pointer does not point to any location in the memory and prevents accidental overwriting of a memory**
  - **You need to add #include directive for stddef.h to your source file**

Be cautious

- You can declare regular variables and pointers in the same line
  - double value, *pVal, fnum;
  - Only the second variable pVal is a pointer of type double
- int *p, q;
  - A common mistake to think is that both p and q are pointers
- **Also it is a good idea to start pointer names beginning with p**

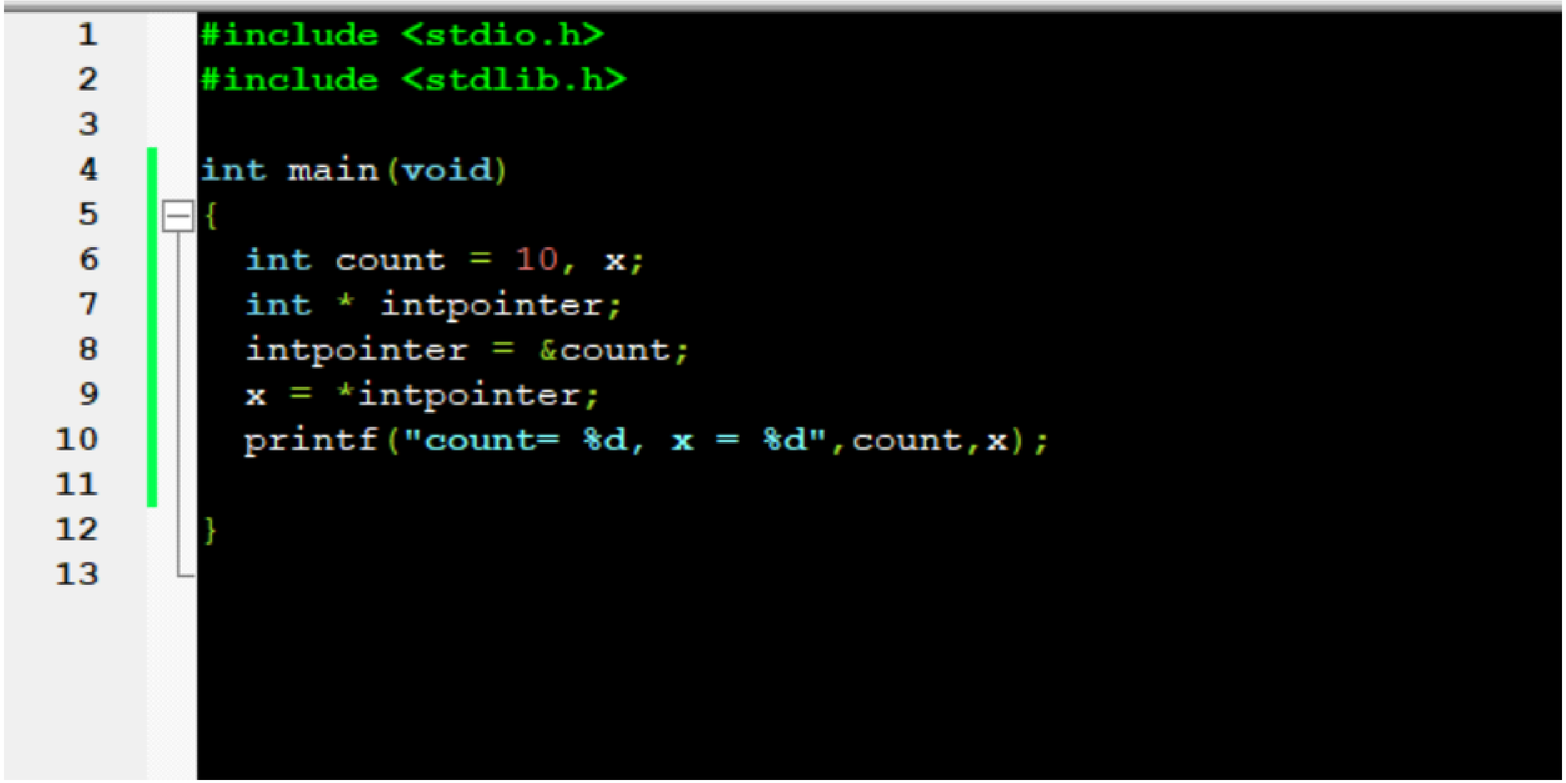
Why use Pointers (1/2)

- Accessing data by only means of variables is very limiting
  - With pointers, you can access any location in the memory as a variable (for example) and perform arithmetic operations
- Pointers make it easy to use Strings and Arrays
- Pointers allow functions to modify the data passed to them as variables
  - Pass by reference: passing arguments to a function in a way they can be changed by function
- Also be used to optimize the program to run faster and use less memory

Why use Pointers (2/2)

- With pointers dynamic memory can be created according to the program use
  - We can save memory from static (compile time) declarations , for example int, float or defining the size of an array
- Pointers allow us to design and develop complex data structures like stack, queue, or linked lists.

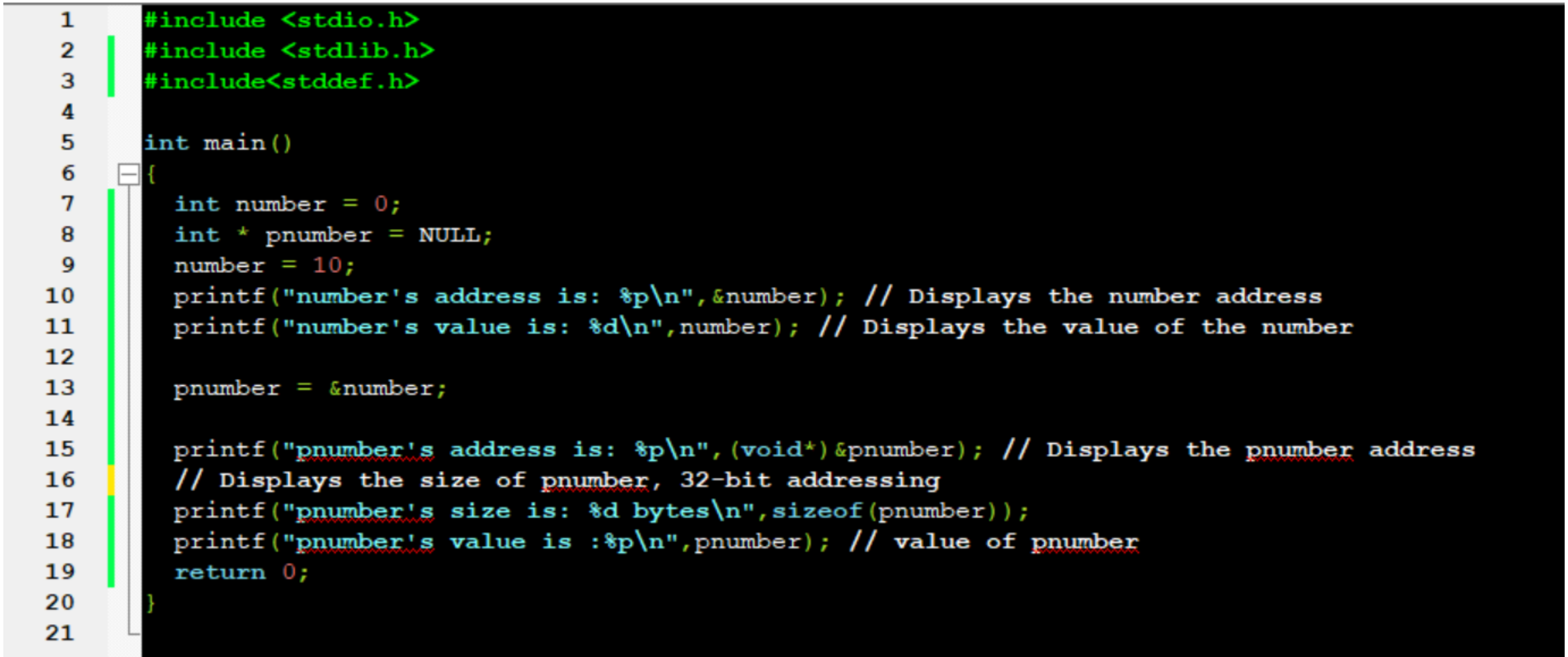
What is the value of x??



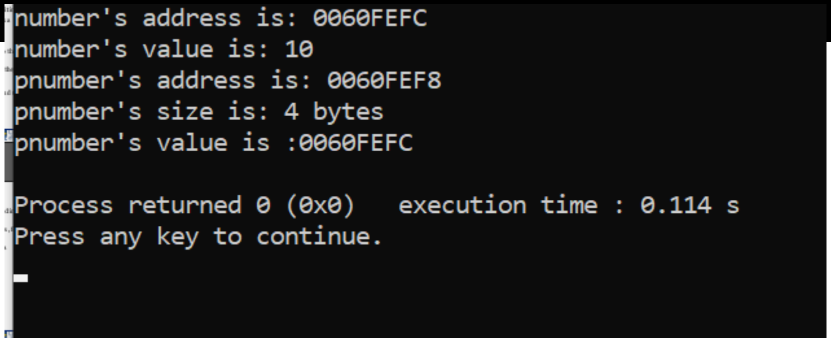
```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
   int count = 10, x;
   int * intpointer;
   intpointer = &count;
   x = *intpointer;
   printf("count= %d, x = %d",count,x);

}
```

The value of X is 10

Size and Actual Address of Pointer

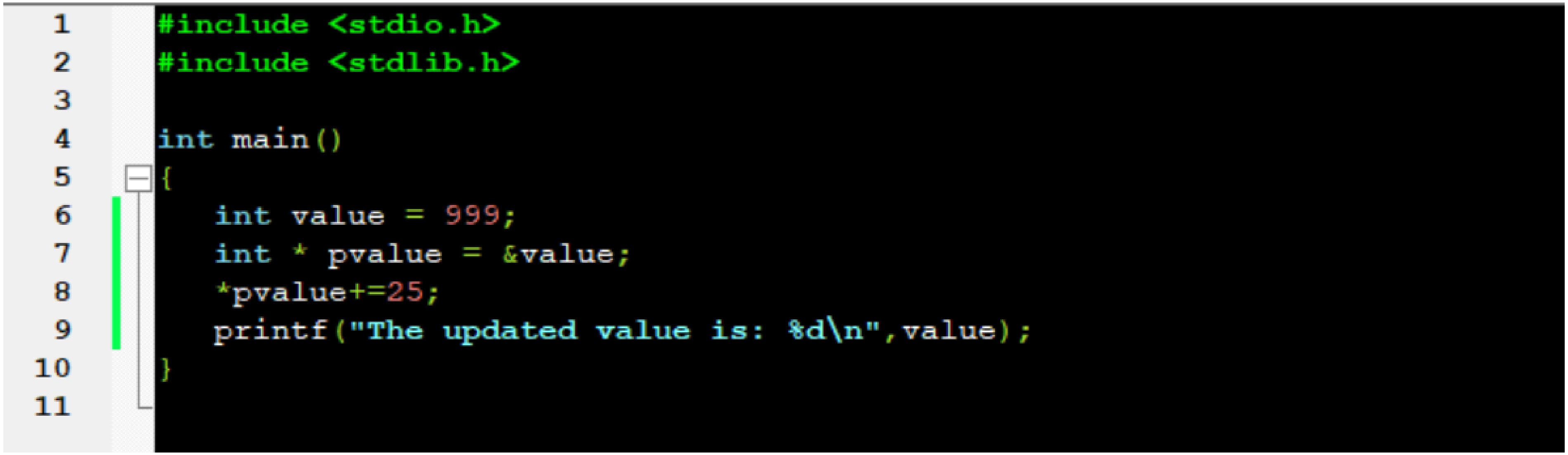


```c
#include <stdio.h>
#include <stdlib.h>
#include<stddef.h>

int main()
{
    int number = 0;
    int * pnumber = NULL;
    number = 10;
    printf("number's address is: %p\n",&number); // Displays the number address
    printf("number's value is: %d\n",number); // Displays the value of the number

    pnumber = &number;

    printf("pnumber's address is: %p\n",(void*)&pnumber); // Displays the pnumber address
    // Displays the size of pnumber, 32-bit addressing
    printf("pnumber's size is: %d bytes\n",sizeof(pnumber));
    printf("pnumber's value is :%p\n",pnumber); // value of pnumber
    return 0;
}
```



```
number's address is: 0060FEFC
number's value is: 10
pnumber's address is: 0060FEF8
pnumber's size is: 4 bytes
pnumber's value is :0060FEFC

Process returned 0 (0x0)   execution time : 0.114 s
Press any key to continue.
```

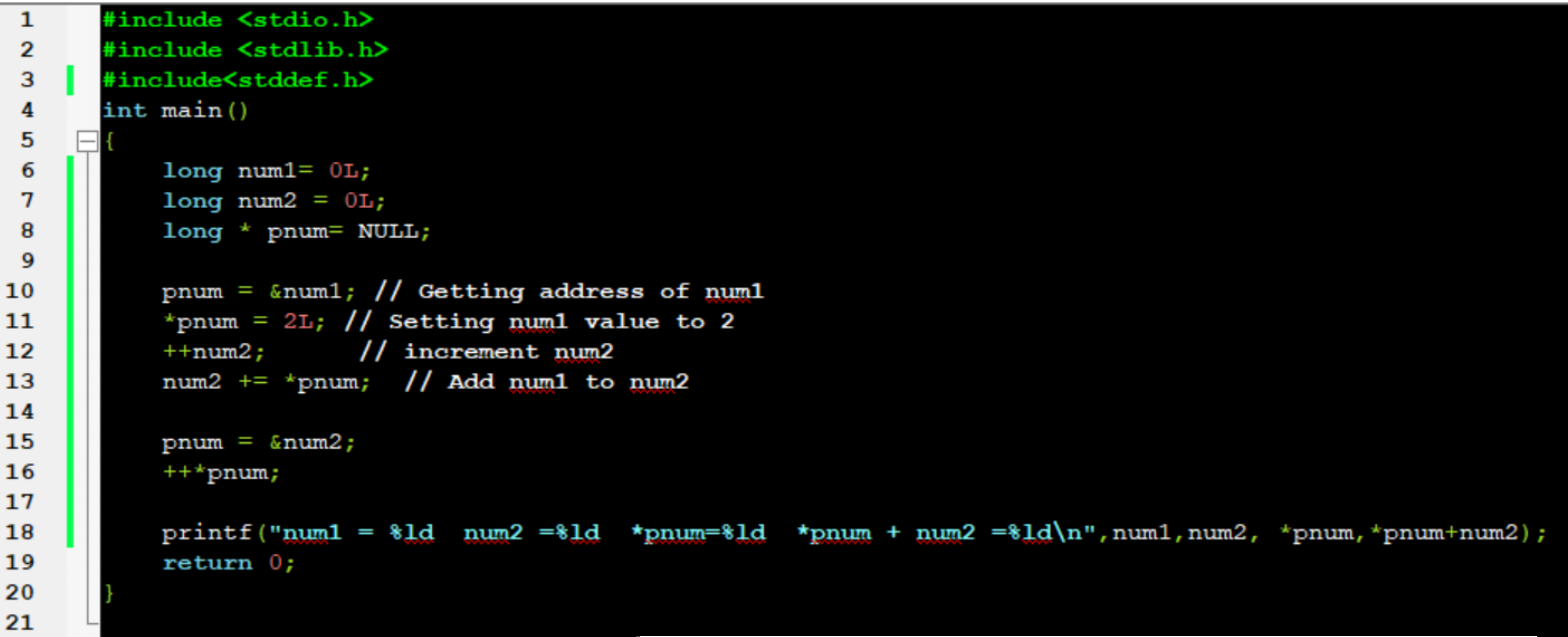
Pointers used in Expression



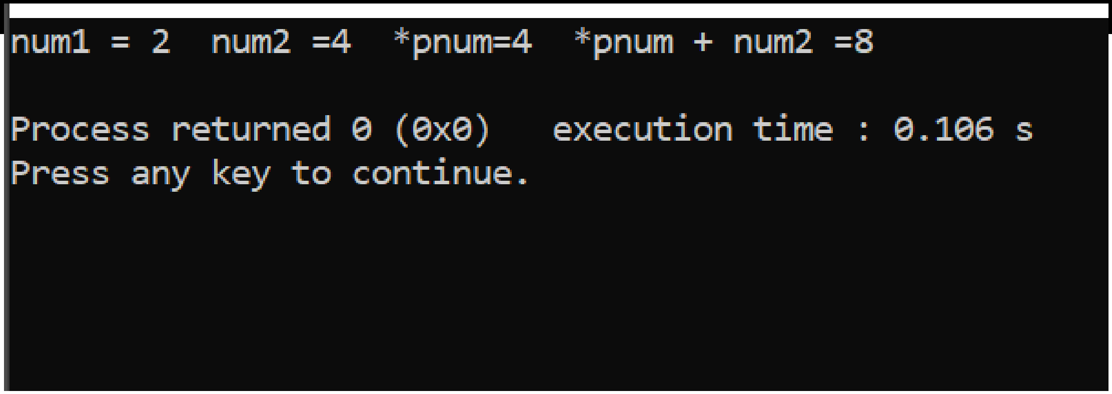
```
#include <stdio.h>
#include <stdlib.h>

int main()
{
    int value = 999;
    int * pvalue = &value;
    *pvalue+=25;
    printf("The updated value is: %d\n",value);
}

```

The updated value is 1024

Pointers and arithmetic operations



```c
#include <stdio.h>
#include <stdlib.h>
#include<stddef.h>
int main()
{
    long num1= 0L;
    long num2 = 0L;
    long * pnum= NULL;

    pnum = &num1; // Getting address of num1
    *pnum = 2L; // Setting num1 value to 2
    ++num2;       // increment num2
    num2 += *pnum;   // Add num1 to num2

    pnum = &num2;
    ++*pnum;

    printf("num1 = %ld  num2 =%ld  *pnum=%ld  *pnum + num2 =%ld\n",num1,num2, *pnum,*pnum+num2);
    return 0;
}
```



```
num1 = 2  num2 =4  *pnum=4  *pnum + num2 =8

Process returned 0 (0x0)   execution time : 0.106 s
Press any key to continue.
```

scanf() and pointers

- Remember we use & operator in the scanf() when we need to store an input (except character types)
- When you have a pointer that already contains an address, you can use the pointer name as an argument for scanf()

```
int value = 0;

int *pvalue = &value;

printf(“input an integer:”);

scanf(“%d”. pvalue);  // Read into value via the pointer
```

Importance of NULL

- **Rule to remember**
  - **Never dereference an uninitialized pointer**

```
int * pt;  // uninitialized pointer

*pt = 5; // store the value 5 to a location where pt points
```

Problem : pt has a random value and there is no knowing where pt will be placed

1. **It might go somewhere harmless, it might overwrite data or code, or might cause the program to crash**

2. **Creating a pointer only allocated memory to store the pointer itself and it does not allocate memory to store the data itself.**

Using Const with Pointers

- Const modifier on array or variable tells the compiler that the contents of the variables/array cannot be changed.
- With pointers we have to take into consideration two things
  - Whether we like the pointer to be changed
  - Or the value of the pointer points to will be changed
- You can use the const keyword when you declare a pointer to indicate that the value pointed to must not be changed

```
long value = 9999L;

const long *pvalue = &value;
```

We have declared the value pointed to by the pvalue to be const

The compiler will throw error if there is an attempt to modify the value

Using Const with Pointers

Now, if you try to reassign a value to pvalue

*pvalue = 8888L;

The compiler will throw an error which states that you are attempting to change the const location

- But you can still change the value because it is not const
  - Value = 7777L;
- Pointer itself is not constant, so you can still change what it points to

long number = 8888L;

pvalue = &number; // OK, changing address in pvalue

Fixing the Pointer Address

- You might want to ensure that address stored in pointer do not change

```
int count = 43;
int *const pcount = &count; // Define a const pointer
```

- The above insures that the pointer always point to the same thing
- Note the change in the position of const keyword.

When everything is fixed (the address and the value)

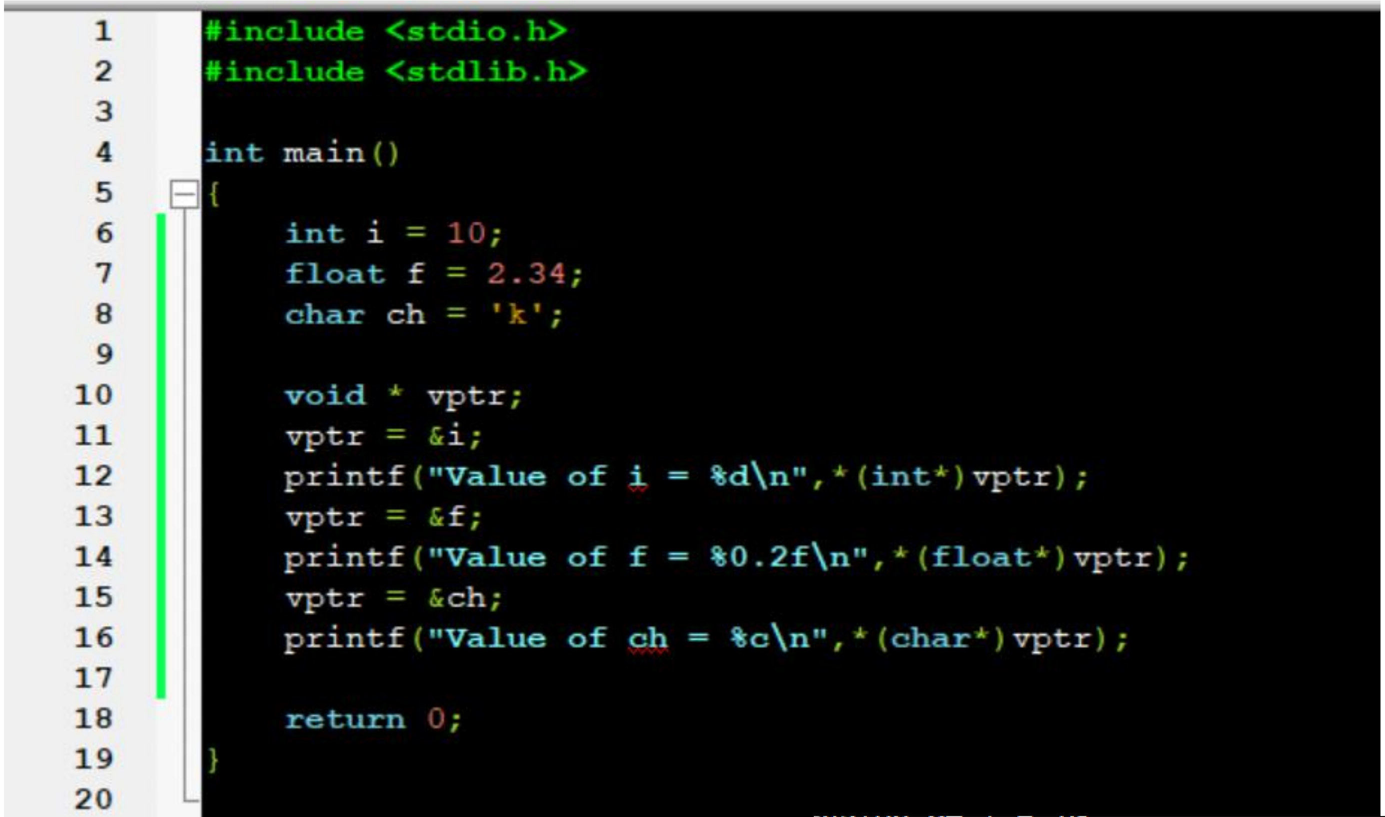
```
int item = 25;
const int * const pitem = &item;
```

- You cannot change the address stored in pitem
- Cannot use pitem to modify what it points to

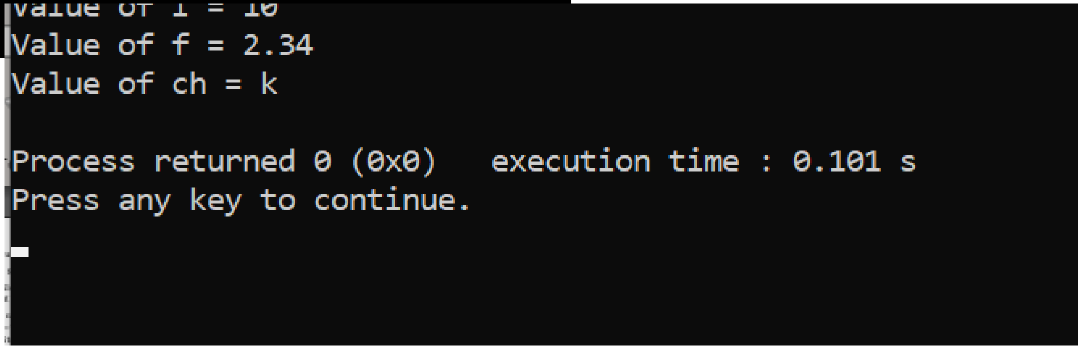
Void Pointers

- The term void means here absence of any type
- A pointer of type void* can contain the address of data item of any type
- void* is often used as a parameter type or return value type with functions that deal with data in type-independent way
- Void pointer does not know the type of object it is pointing to, so it cannot be dereferenced directly
- Void pointer must be first explicitly cast to another pointer type before it can be dereferenced.

Example: Void Pointer



```c
#include <stdio.h>
#include <stdlib.h>

int main()
{
    int i = 10;
    float f = 2.34;
    char ch = 'k';

    void * vptr;
    vptr = &i;
    printf("Value of i = %d\n",*(int*)vptr);
    vptr = &f;
    printf("Value of f = %0.2f\n",*(float*)vptr);
    vptr = &ch;
    printf("Value of ch = %c\n",*(char*)vptr);

    return 0;
}
```



```
Value of i = 10
Value of f = 2.34
Value of ch = k

Process returned 0 (0x0)   execution time : 0.101 s
Press any key to continue.
```

Pointers and Arrays (1/2)

- Recall: Array is the collection of items of same type referred by a single name
- Pointers hold memory address and it can hold the memory address of different variables at different times (must be of same type)
- One of the most common uses of pointers is : Pointers to arrays
- Pointers to arrays generally uses less memory and executes faster.

Pointers and Arrays (2/2)

- Let’s take an example of array values with 100 integers
  - int values[100];
  - You can define a pointer called **valuesPtr**, which can be used to access the elements of the array **values**
  - **int * valuesPtr;**
- ***Remember:*** *array names are themselves pointers and therefore you do not use* **&** *operator when pointing towards an arrayname*
  - **valuesPtr = values;**
- When specifying the array name without a subscript, this has an effect of producing a pointer to the first element of the **values or**
  - **valuesPtr = &values[0];**

Pointer, Arrays and Arithmetic's (1/2)

- Real power of using pointers for array comes into play when you want to sequence through the elements of an array
- * valuesPtr can be used to access the first integer of the values array (i.e., values[0])
- To reference values[3] via pointer, you can simply add 3 to valuesPtr such as *(valuesPtr +3)
- If you want to assign a specific value (say 27) at a specific location (say 10) in the values array via pointer, you could do the following

```
*(valuesPtr + 10) = 27;
```

Pointers, Arrays and Arithmetic’s (2/2)

- To set the valuesPtr to point the second element of the array, you do the following

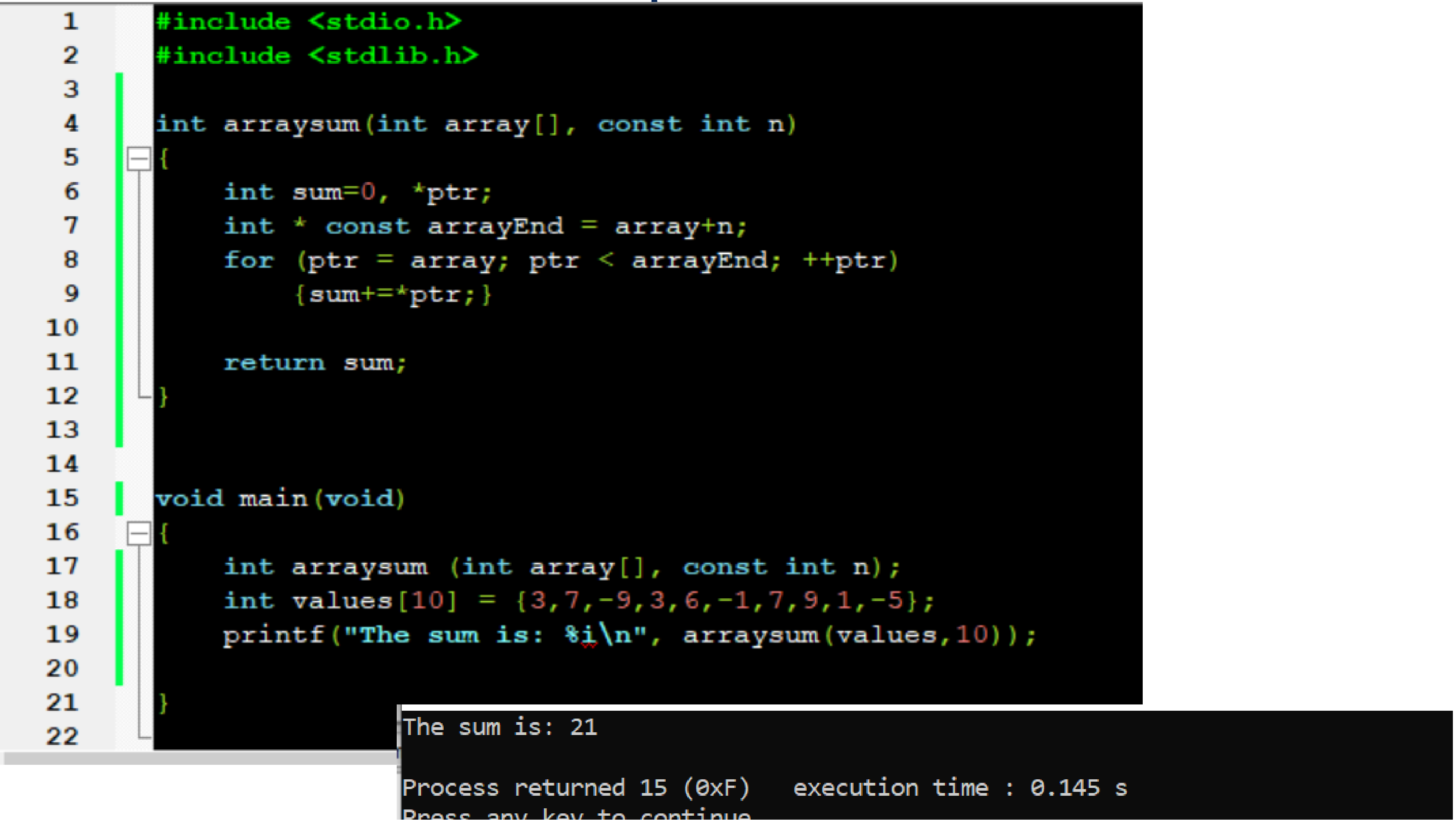
valuesPtr = &values[1];

Or

valuesPtr +=1;

- Increment and Decrement operators have same effect on pointers as adding one and subtracting one to the pointer, respectively.
  - ++valuesPtr Or –valuesPtr (Be careful of out of bound error)

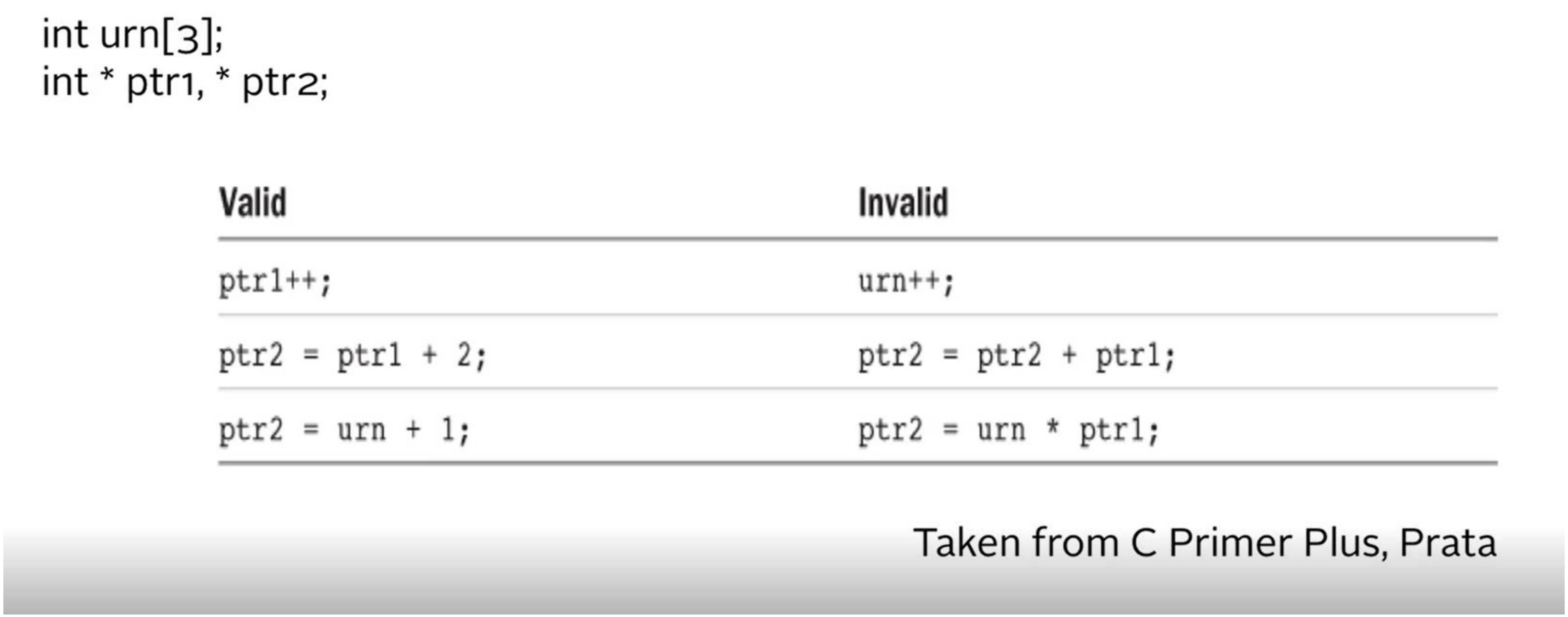
Example : Pointers, Arrays and Arithmetic Operations



```c
#include <stdio.h>
#include <stdlib.h>

int arraysum(int array[], const int n)
{
    int sum=0, *ptr;
    int * const arrayEnd = array+n;
    for (ptr = array; ptr < arrayEnd; ++ptr)
        {sum+=*ptr;}

    return sum;
}


void main(void)
{
    int arraysum (int array[], const int n);
    int values[10] = {3,7,-9,3,6,-1,7,9,1,-5};
    printf("The sum is: %i\n", arraysum(values,10));

}
```

```
The sum is: 21

Process returned 15 (0xF)   execution time : 0.145 s
Press any key to continue
```

Summary



```
int urn[3];
int * ptr1, * ptr2;
```

Valid	Invalid
`ptr1++;`	`urn++;`
`ptr2 = ptr1 + 2;`	`ptr2 = ptr2 + ptr1;`
`ptr2 = urn + 1;`	`ptr2 = urn * ptr1;`

Taken from C Primer Plus, Prata

You cannot add addresses or multiply them. Likewise, ++ operator does not work on array names directly unlike pointers.