**Pointers--**reference something using a name

**Indirection**

In programming languages, indirection is the ability to reference something using a name, reference, or container, instead of the value itself.

The most common form of indirection is the act of manipulating a value through its memory address

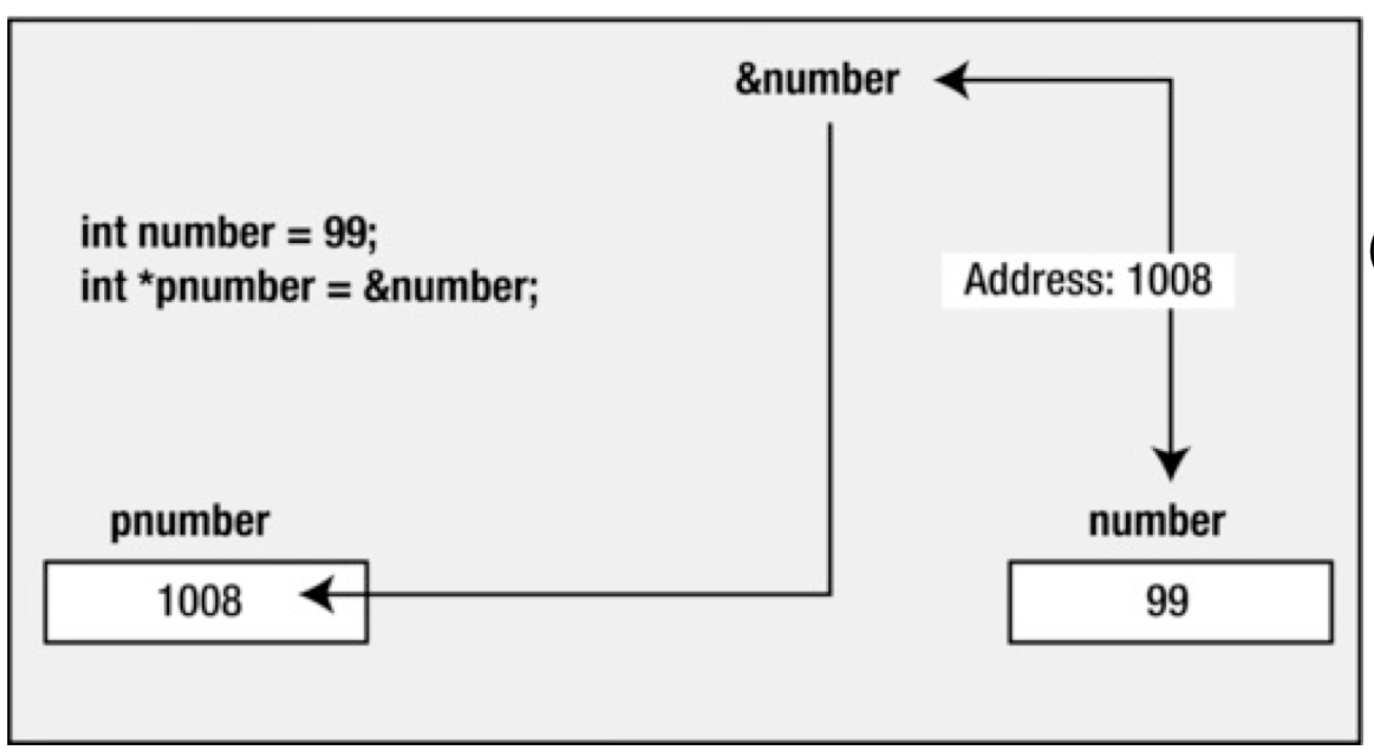
A pointer provides an indirect means of accessing the value of a particular data item

A variable whose value is a memory address

Its value is the address of another location in memory that can contain a value.

**Overview:**

* The compiler must know the type of data stored in the variable to which it points.
* Needs to know how much memory is occupied or how to handle the contents of the memory to which it points.
* Every pointer will be associated with a specific variable type
* It can be used only to point to variables of that type
* Pointers of type “pointer to int” can point only to variables of type int
* Pointers of type “pointer to float” can point only to variables of type float



* The value of &number is the address where number is located
* This value is used to initialize pnumber in the second statement

**Why use pointers?**

* Accessing data by means of only variables is very limiting
* With pointers, you can access any location( you can treat any position of memory as a variable for example) and perform arithmetic with pointers.
* Pointers in C make it easier to use arrays and strings
* Pointers allow you to refer to the same space in memory from multiple locations
* Means that you can update memory in one location and the change can be seen from another location in your program
* Can also save space by being able to share components in your data structures
* Pointers allow functions to modify data passed to them as variables
* Pass by reference - passing arguments to function in a way they can be changed by function
* Can also be used to optimise a program to run faster or useless memory than it would otherwise
* Pointers allow us to get multiple values from the function
* A function can return only one value but by passing arguments as pointers we can get more than one values from the pointer
* With pointers dynamic memory can be created according to the program use
* We can save memory from static (compile time) declarations.
* Pointers allow us to design and develop complex data structures like a stack, queue , or linked list.
* Pointers provide direct memory access.

**Defining pointers**

**Declaring pointers**

* Pointers are not declared like normal variables
* You also have to specify the kind of variable to which the pointer points
* Different variable types take up different amounts of storage
* Some pointer operations require knowledge of that storage size.
* You declare a pointer to a variable of type int with:

int \*pnumber;

* The type of the variable with the name pnumber is int\* can store the address of any variable of type int

int \*pi; // pi is a pointer to an integer variable

char \*pc; //pc is a pointer to a character variable

Float \*pf, \*pg; //pf, pg are pointers to float variables

* The space between the \*and the pointer name is optional
* Programmers use the space in a declaration and omit it when dereferencing a variable.
* The value of a pointer is an address, and it is represented internally as an unsigned integer on most systems
* However, you shouldn’t think of a pointer as an integer type.
* Things you can do with integers that you can not do with pointers , and vice versa
* You can multiply one integer by another , but you can not multiply one pointer by another.
* A pointer really is a new type , not an integer type
* %p represents the format specifier for pointers
* The previous declarations creates the variable but does not initialize it;
* Dangerous when not initialized
* You should always initialize a pointer when you declare it.

**NULL POINTERS**

* You can intialize a pointer so that it does not point to anything:

Int \*pnumber = NULL ;

* NULL is a constant that is defined in the standard library
* Is the equivalent of zero for a pointer
* NULL is a value that is guaranteed not to point to any location in memory
* Means that it implicitly prevents the accidental overwriting of memory by using a pointer that does not point to anything specific.
* Add an #include directive for stddef.h to your source file

**Address of operator**

* If you want to initialize your variable with the address of a variable you have already declared
* Use the address of operator, &

int number = 99;

int \*pnumber = &number;

* The initial value of pnumber is the address of the variable number.
* The declaration of number must precede the declaration of the pointer that stores its address
* Compiler must have already allocated space and thus an address for number to use it to initialize pnumber

**Be careful**

* There is nothing special about the declaration of a pointer
* Can declare regular variables and pointers in the same statement.

double value, \*pVal, fnum;

* Only the second variable, pVal, is a pointer

int \*p, q;

* The above declares a pointer, p of type int\*, and a variable, q, that is of type int a common mistake to think that both p and q are pointers
* Also, it is a good idea to use names beginning with p as pointer names.

**ACCESSING POINTER VALUES**

* You use the indirection operator, \*, to access the variable pointer to by a pointe also referred to as the dereference operator because you use it to ‘dereference’ a pointer

int number = 15;

int \*pointer = &number;

int result = 0;

* The pointer variable contains the address of the variable number you can use this in an expression to calculate a new value for result

result = \*pointer + 5;

* The expression \*pointer will evaluate to the value stored at the address contained in the pointer the value stored in number, 15, so result will be set to 15 + 5, which is 20
* The expression \*pointer will evaluate to the value stored at the address contained in the pointer the value stored in the number, 15, so result will be set to 15 + 5, which is 20.
* The indirection operator, \*, is also the symbol for multiplication, and it is used to specify pointer types depending in where the asterisk appears, the compiler will understand whether it should interpret it as an indirection operator, as a multiplication sign, or as part of a type specification context determines what it means in any instance.

**Displaying a pointer value**

* To output the address of a variable, you use the output format specifier %p outputs a pointer a pointer value as a memory address in hexadecimal form

int number = 0; // a variable of type int initialized to 0

int \*pnumber = NULL; // a pointer that can point to type int

number = 10;

pnumber = &number;

printf(“pnumber’s value: %p\n, pnumber); //output the value (an address)

* Pointers occupy 8 bytes and the address have 16 hexidecimal digits if a machine has 64-bit operating system and my compiler supports 64-bit addresses some compilers only support 32-bit addressing, in which case addresses will be 32-bit addresses

printf(“number’s address: %p\n”, &number); //Output the address

printf(“pnumber’s address: %p\n”, (void\*) &pnumber); //Output the address

* Remember, a pointer itself has an address, just like any other variable you use %p as the conversion specifier to display an address.
* You use the &(address of) operator to reference the address that the pnumber variable occupies
* The cast to void\* is to prevent a possible warning from the compiler. The %p specification expects the value to be some kind of pointer type, but the type of &pnumber is “pointer to pointer to int”

**Displaying the number of bytes a pointer is using**

* You use the sizeof operator to obtain the number of bytes a pointer occupies.
* On my machine this shows that a pointer occupies 8bytes.
* A memory address in my machine is 64bits
* You may get a compiler warning when using sizeof this way
* Size\_t is an implementation-defined integer type
* To prevent the warning , you could cast the argument to type int like this:

printf(“pnumber’s size: %d bytes\n”,(int)sizeof(pnumber)); //output the size

**Example:**

int main(void)

{

int number = 0; //a variable of type int initialized to 0

int \*pnumber = NULL; //a pointer that can point to type int

number = 10;

printf(“number’s address:%p/n”, &number); //output the address

printf(“number’s value:%d/n/n”, number); //output the value;

pnumber = &number; //store the address of number in pnumber

printf(“pnumber’s address: %p/n”, (void\*)(pnumber)); //output the address

printf(“pnumber’s size: %zd bytes/n”, sizeof(pnumber); //output the size

printf(“pnumber’s value: %p/n”, pnumber); //output the value(an address)

printf(“value pointer to: %d/n”, \*pnumber); //value at the address

return (0);

}

**USING POINTERS**:

* You can assign an address to a pointer
* Assigned value can be an array name, a variable preceded by address operator (&), or another second pointer.
* You can dereference a pointer
* The \* operator gives the value stored in the pointed-to-location
* You can take a pointer address
* The & operator tells you where the pointer itself is stored.
* You can perform pointer arithmetic
* Use the +operator to add an integer to a pointer to an integer (integer is multiplied by the number of bytes in the pointed-to type and added to the original address)
* Increment a pointer by one (useful in arrays when moving to the next element)
* Use the - operator to subtract an integer from a pointer (integer is multiplied by the number of bytes in the pointed-to type and subtracted from the original address.
* Decrementing a pointer by one(useful in arrays when going back to the previous element).
* You can find the difference between two pointers
* You do this for two pointers to elements that are in the same arrays to find out how far apart the elements are.
* You can use the relational operators to compare the values of two pointers
* Pointers must be the same type
* Remember, there are two forms of subtraction
* You can subtract one pointer from another to get an integer
* You can subtract an integer from a pointer and get a pointer
* Be careful when incrementing or decrementing pointers and causing an array “out of bounds” error
* Computer does not keep track of whether a pointer still points to an array element

**POINTERS USED IN EXPRESSIONS**

* The value reference by a pointer can be used in arithmetic expressions\
* If a variable is defined to be of type “pointer to integer” then it is evaluated using the rules of integer arithmetic.

int number = 0; // a variable of type int initialized to 0

int \*pnumber = NULL; // a pointer that can point to type int

number = 10;

pnumber = &number; // store the address of pnumber

\*pnumber += 25;

* Increments the value of the number variable by 25
* \*indicates you are accessing the contents to which the variable called pnumber is pointing to
* If a pointer points to a variable x
* That pointer has been defined to be a pointer to the same data type as is x
* Use of \*pointer in an expression is identical to the use of x in the same expression.
* A variable defined as a “pointer to int” can store the address of any variable of any variable of type int

int value =999;

pnumber =&value;

\*pnumber +=25;

* The statement will operate with the new variable, value
* The new contents of value will be 1024
* A pointer can contain the address of any variable of the appropriate type
* You can use one pointer variable to change the values of many different variables as long as they are of a type compatible with the pointer type.

int main(void)

{

long num1 = 0L;

long num2 = 0L;

long \*pnum = NULL;

pnum = &num1;

\*pnum = 2L;

++num2;

num2 += \*pnum;

pnum = &num2;

++\*pnum;

printf(“num1 = %ld num2 = %ld \*pnum = %ld \*pnum + num2 = %ld\n”, num1, num2, \*pnum, \*pnum + num2);

return (0);

}

Output : num1 = 2 num2 = 4 \*pnum = 4 \*pnum +num2 = 8

**When receiving input:**

* When we have used scanf() to input values, we have used the & operator to obtain the address of a variable
* On the variable that is to store the input (second argument)
* 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; //sets pointer to refer to value

printf(“input an integer:”);

scanf(“%d”, pvalue); // read into value via the pointer

printf(“You entered %d.\n”, value); // Output the value entered

**Testing for NULL**:

There is one rule you should burn into your memory - DO NOT DEREFERENCE AN UNINITIALIZED POINTER.

int \*pt; //an uninitialized pointer

\*pt =5; //a terrible error

* The second line means store the value 5 in the location to which pt points
* pt has a random value, there is no knowing where the 5 will be placed.
* It might go somewhere harmless, it might overwrite data or code, or it might cause the program to crash.
* Creating a pointer only allocates memory to store the pointer itself
* It does not allocate memory to store data
* Before you use a pointer, it should be assigned a memory location that has already been allocated
* Assign the address of an existing variable to the pointer
* Or you can use the malloc() function to allocate memory first.
* We already know that when declaring a pointer that does not point to anything, we should initialize it to NULL

int \*pvalue = NULL;

* NULL is a special symbol in C that represents the pointer equivalent to 0 with ordinary numbers
* The below also sets a pointer to NULL using 0

int \*pvalue = 0;

* Because NULL is the equivalent of zero, if you want to test whether pvalue is NULL, you can do this:

if(!pvalue)...

* You will want to check for NULL before you dereference a pointer
* Often when pointers are passed to functions.

**POINTERS AND CONST**

* When we use the const modifier on a variable or an array it tells the compiler that the contents of the variable/array **will not be changed by the program.**
* With pointers, we have to consider two things when using the const modifier.
* Whether the pointer will be changed
* Whether the value that 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; // defines a pointer to a const

* You have declared the value pointed to by pvalue to be const
* The compiler will check for any statements that attempted to modify the value pointed to by pvalue and flag such statements as an error
* The following statement will now result in an error message from the compiler

\*pvalue = 8888L; //Error -attempt to change const location

* You can still modify value(you have only applied const to the pointer)

value =7777L;

* The value pointed to has changed, but you did not use the pointer to make the change.
* The pointer itself is not constant, so you can still change what it points to:

Long number = 8888L;

pvalue = &number; // ok - change the address in pvalue

* Will change the address stored in pvalue to point to number
* Still cannot use the pointer to change the value that is stored
* You can change the address stored in the pointer as much as you like
* Using the pointer to change the value pointed to is not allowed, even after you have changed the address stored in the pointer.
* You might also want to ensure that the address stored in a pointer cannot be changed
* You can do this by using the const keyword in the declaration of the pointer

int count = 43;

int \*const pcount = &count; // Defines a constant pointer

* The above ensures that a pointer always points to the same thing
* Indicates that the address stored must not be changed
* Compiler will check that you do not inadvertently attempt to change what the pointer points to elsewhere in your code.

int item =34;

pcount =&item; //error- attempt to change a constant pointer

* It is all about where you place the const keyword, either before the type or after the type/

const int\* … // value can not be changed

int \*const … // pointer address cannot change

* You can still change the value that pcount points to using pcount

\*pcount = 345; //OK - changes the value of count

* Reference the value stored in count through the pointer and changes its value to 345.
* You can create a constant pointer that points to a value that is also constant:

int item = 25;

const int \*const pitem = &item;

* The pitem is a constant pointer to a constant so everything is fixed
* Cannot change the address stored in pitem
* Cannot use pitem to modify what it points to
* You can still change the value of item directly
* If you wanted to make everything not change , you could specify item as const as well.

**VOID POINTER**

* The type name **void** means absence of any type
* A pointer of type void\* can contain the address of a data item of any type
* void\* is often used as a parameter type or return value type with functions that deal with data in a type-independent way.
* Any kind of pointer does not know what type of object it is pointing to, so, it cannot be dereferenced directly.
* The void pointer does not know what type of object it is pointing to, so, it cannot be dereferenced directly.
* The void pointer must first be explicitly cast to another pointer type before it is dereferenced.
* The address of a variable of type int can be stored in pointer variable of type void\*
* When you want to access the integer value at the address stored in the void\* pointer, you must first cast the pointer to type int\*.

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 = %.2f/n”, \*(float\*)vptr);

vptr =&ch;

printf(“Value of ch = %c/n”, \*(char\*)vptr);

**POINTERS AND ARRAYS**

* An array is a collection of objects of the same type that you can refer to using a single name.
* A pointer is a variable that has as its value a memory address that can reference another variable or constant of a given type.
* You can use a pointer to hold the address of different variables at different times(must be same type).
* Arrays and pointers seem quite different, but they are very closely related and can sometimes be used interchangeably
* One of the most common uses of pointers in C is as pointers to arrays
* The main reasons for using pointers to arrays are ones of notational convenience and of program efficiency
* Pointers to arrays generally result in code that uses less memory and executes faster
* If you have an array of 100 integers

int values[100];

* You can define a pointer called valuesPtr, which can be used to access the integers contained in this array.

int \*valuesPtr;

* When you define a pointer that is used to point to the elements of an array, you do not designate the pointer as type “pointer to array”
* You designate the pointer is pointing to the type of element that is contained in the array.
* To set valuesPtr to point to the first element in the values array, you write

valuePtr = values;

* The address operator is not used
* The C compiler treats the appearance of an array name without a subscript as a pointer to the array.
* Specifying values without a subscript has the effect of producing a pointer to the first element of values.
* An equivalent way of producing a pointer to the start of values is to apply the address operator to the first element of the array

valuePtr = &values[10];

* So , you can use the above example or the one on the previous slide

valuesPtr = values;

* either one is fine and a matter of programmer perference

**POINTER ARITHMETIC**

* The real power of using pointers to arrays 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 array, that is, values[0]

* To reference values[3] through the valuesPtr variable, you can add 3 to valuesPtr and then apply the indirection operator.

\*(valuesPtr+3)

* The expression, \*(valuePtr + i) can be used to access the value contained in value[i]
* To set values[10] to 27, you could do the following

value[10] = 27;

* Or , using valuesPtr, you could

\*(valuePtr + 10) = 27;

* To set valuesPtr to point to the second element of the values array, you can apply the address operator to values[1] and assign the result to valuesPtr

valuePtr = &values[1];

* If valuesPtr points to values[0], you can set it to point to values[1] by simply adding 1 to the value of valuesPtr

valuesPtr += 1;

* This is a perfectly valid expression in C and can be used for pointers to any data type.
* The increment and decrement operators ++ and -- are particularly useful when dealing with pointers.
* Using the increment operator on a pointer has the same effect as adding one to the pointer .
* Using the decrement operator has the same effect as subtracting one from the pointer.

++valuesPtr;

* Sets valuesPtr pointing to the next integer in the values array (values[1])

--textPtr;

* Sets valuesPtr pointing to the previous integer in the values array, assuming that valuesPtr was not pointing to the beginning of the values array.

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 value[10]= {3,7,-9,3,6,-1,7,9,1,-5};

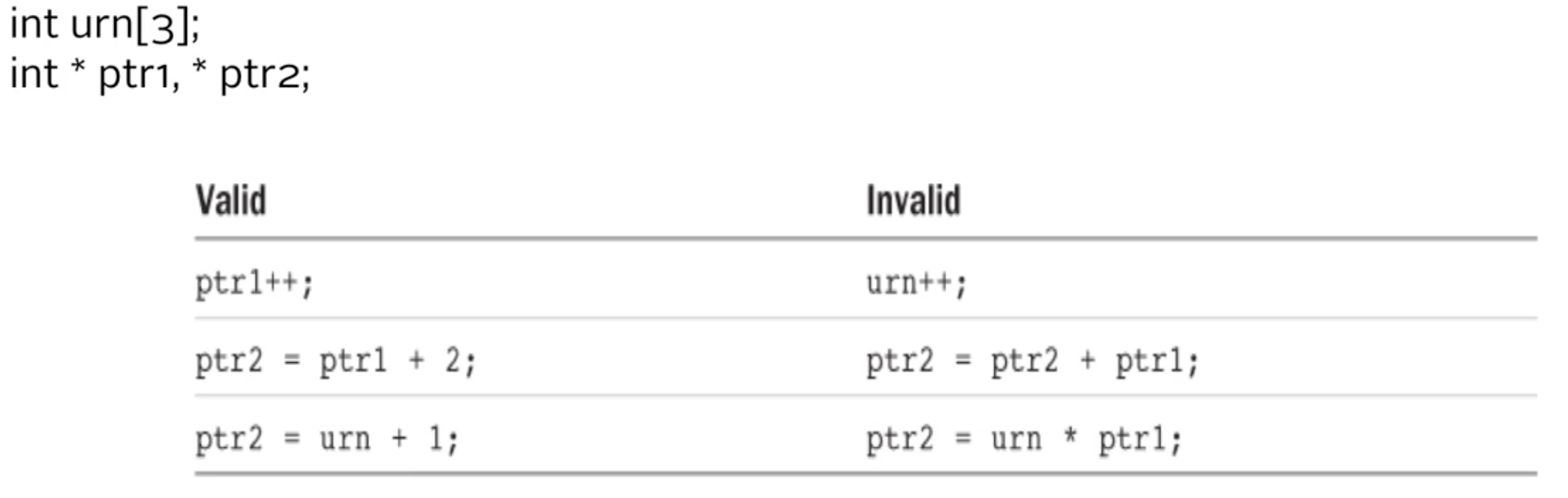
printf(“The sum is %i\n”, arraySum(values, 10));

}

* To pass an array to a function, you simply specify the name of the array.
* To produce a produce a pointer to an array, you need only specify the name of the array.
* This implies that in the call to the arraySum() function, what was passed to the function was actually a pointer to the array values.
* Explains why you are able to change the elements of an array from within a function.
* So you might wonder why the formal parameter inside the function is not declared to be a pointer

int arraySum(int \*array, const int n)

* The above is perfectly valid
* Pointers and arrays are intimately related in C
* This is why you can declare array to be type “array of ints” inside the arraySum function or to be type “pointer to int”.
* If you are going to be using index numbers to reference the elements of an array that is passed to a function, declare the corresponding formal parameter to be an array.
* More correctly reflects the use of the array by the function.



* Functions that process arrays actually use pointers as arguments
* You have a choice between array notation and pointer notation for writing array processing functions.
* Using array notation makes it more obvious that the function is working with arrays.
* Array notation has a more familiar look to programmers vs in FORTRAN, Pascal, Modula-2, or BASIC.
* Other programmers might be more accustomed to working with pointers and might find the pointer notation more natural.
* Closer to machine language and, with some compilers , leads to more efficient code.

**POINTERS AND STRINGS**

* We now know how arrays relate to pointers and the concept of pointer arithmetic.
* These concepts can be very useful when applied to character arrays (strings).
* One of the most common applications of using a pointer to an array is as a pointer to a character string.
* The reasons are one of notational convenience and efficiency.
* Using a variable of type pointer to char to reference a string gives you a lot of flexibility.
* Let's look at an example that uses an array to copy a string

void copyString (char to[], char from[])

{

int i;

for(i = 0; from[i] != ‘\0’, ++i)

to[i] = from[i];

to[i] = ‘\0’;

}

void copyString(char \*to, char \*from)

{

for( ; \*from != ‘\0’; ++from, ++to)

\*to = \*from;

\*to = ‘\0’;

}

* If you have an array of characters called text, you could similiarly define a pointer to be used to point to elements in text

char \*textPtr;

* If textPtr is set pointing to the beginning of an array of chars called text.

++textPtr;

* The above sets textPtr pointing to the next character in text, which is text[1]

--textPtr;

* The above sets textPtr pointing to the previous character in text, assuming that textPtr was not pointing to the beginning of text prior to the execution of this statement

void copyString (char \*to, char \*from)

{

int i;

While (\*from) // the null character is equal to the value 0, so will jump out then

\*to++ = \*from++;

\*to = ‘\0’;

}

int main(void)

{

char string1[]=”A string to be copied.”;

char string2[50];

copyString(string2, string1);

printf(“%s/n”, string2);

}

Task, are performed more easily with pointers, and other tasks, such as dynamic memory allocation, cannot be performed without using pointers. So it becomes necessary to learn pointers.

Every variable is a memory location and every memory location has its address defined which can be accessed using ampersand (&) operator, which denotes an address in memory. Consider the following example :

#include <stdio.h>

int main()

{

Int var1;

char var2[10];

printf(“address of var1 variable: %x\n”, &var1 );

printf(“address of var2 variable: %x\n”, &var2 );

Return (0);

}

When the above code is compiled and executed, it produces the following result:

Address of var1 variable: bff5a400

Address of var2 variable: bff5a3f6

**What are pointers ?**

A pointer is a variable whose value is the address of another variable , ie, direct address of the memory location. Like any variable or constant , you must declare a pointer using it to store any variable address. The general form of a pointer before using it to store any variable address. The general form of a pointer declarations-

Int \*ip; /\* pointer to an integer\*/

double \*dp; /\* pointer to a double\*/

float \*fp; /\* pointer to a float \*/

char \*ch; /\* pointer to a character \*/

The actual data type of the value of all pointers, whether integer, float , character, or otherwise, is the same, a long hexadecimal number that represents a memory address. The only difference between pointers of different data types is the data type of the variable or constant that the pointer points to.

**How to use pointers?**

There are a few important operations, which we do the help of pointers very frequently. (a) We define a pointer variable, (b) assign the address of a variable to a pointer and (c) finally access the value at the address available in the pointer variable. This is done by using unary operator \* that returns the value of the variable located at the address specified by its operand. The following example makes uses of these operations-