Pointers in C++

Introduction

Pointers are extremely powerful because they allow accessing the addresses and manipulating their data. If we utilizing pointers in correct manner then they could greatly improves the efficiency and performance otherwise they could lead to many problems. The use of pointers makes the program more efficient and compact. Some of the uses of pointers are:

- Improving efficiency
- Accessing dynamically allocated memory
- Returning more than one value from a function
- Implementing data structures

Before going into pointers, it is important to understand how the memory is organized. In computer the memory is made up of bytes (8 bits) arranged in a sequential manner, each byte is identified by an index number called the address of that byte. The address of the bytes starts from zero and the address of the last byte is one less than the size of memory. For example 1KB memory has 1024 bytes and each byte given an address 0 to 1023. Already we have studied the concept of variables in the previous modules a variable is a name for a piece of memory which holds a value. When we declare a variable in the program it automatically access the free memory and assigned to the variable in random manner and any value assigned to that variable is stored in this memory address location.

Address Operator (&)

The address operator allows us to see what memory address is assigned to a variable. This operator can be read as "address of" variable.

Example:

```
int x;
Address of x = '&x'
```

Dereference operator (*)

The dereference operator allows us to get the value at a particular address. It is also called indirection operator.

Finally after the introduction of memory, address operator and dereference operator it is time to talk about pointers.

Pointers

A pointer is a variable that holds a memory address as its value. We perform some operations with pointers very frequently.

- Define pointer variables.
- Assign the address of a variable to a pointer using address operator '&'.
- Access the value at the address of variable using dereference operator '*'.

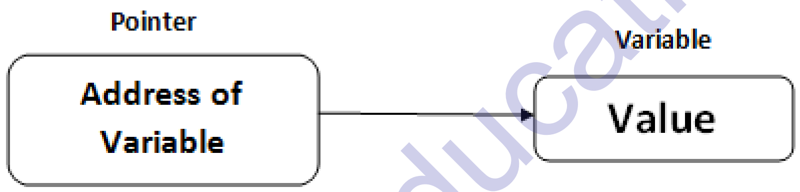


Figure1: Pointer variable

Declaring a pointer

Pointer variables are declared like normal variables, with an asterisk in front of variable name.

Syntax:

data_type *pname;

pname is the name of pointer variable.

data_type is known as base type of pointer.

Examples:

int *iptr; → iptr is a pointer to an integer value

char *cptr; → cptr is a pointer to a character type

float *fptr; → fptr is a pointer to a float value

int *p; or int* p; are same

Initializing pointers

Pointers are variables so the compiler must reserve memory for pointers and they will have some address. When declare a pointer variable, but not initialized then it contains the random address of other memory locations, which is one of the most common errors encountered when using pointers and it harder to debug the code. This situation is very dangerous which may crash the program, so we must need to initialize a pointer variable whenever they declared. Generally the local and global static pointers are automatically initialized to NULL but when we declare an automatic pointer variable it contains the garbage value that is it may points to anywhere in the memory locations. The use of an unassigned pointer may give unpredictable results and it may crash the operating system also.

Example:

```
int a=7;
int *ptr = &a;
```

→ A pointer variable named ‘ptr’ that points to an integer variable named ‘a’.

int ‘*ptr’ declares the pointer to an integer value, which is initialized to address of ‘a’. Once a pointer is declared and initialized then we can deference it using dereference operator ‘*’ which gives the actual value stored at that address.

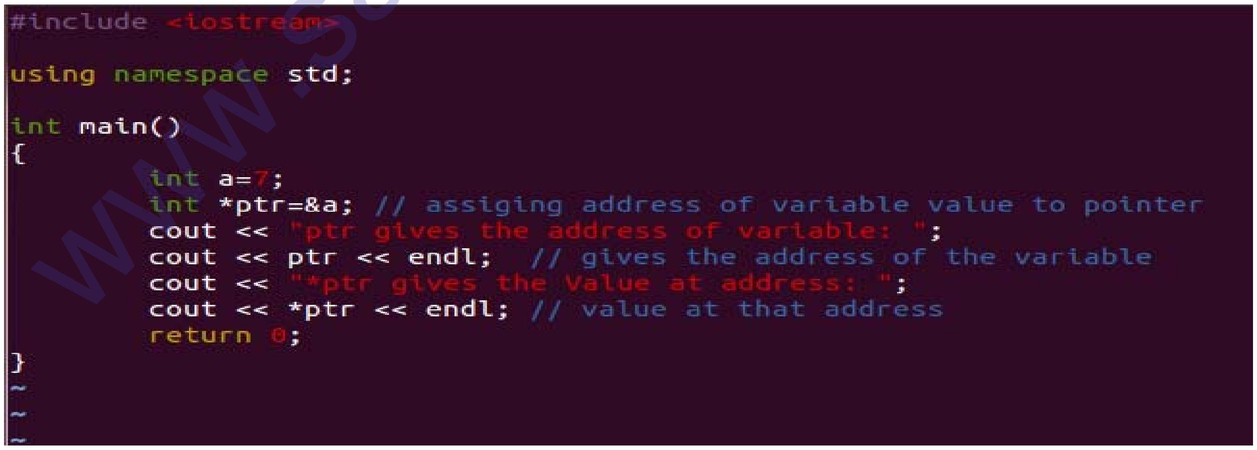
Cout << *ptr; → displays the value pointed by ‘ptr’.

int a = 7;

int *p = &a;

cout << p << endl; → 0003 → displays the address of variable

cout << *p << endl; → 7 → displays the value of variable

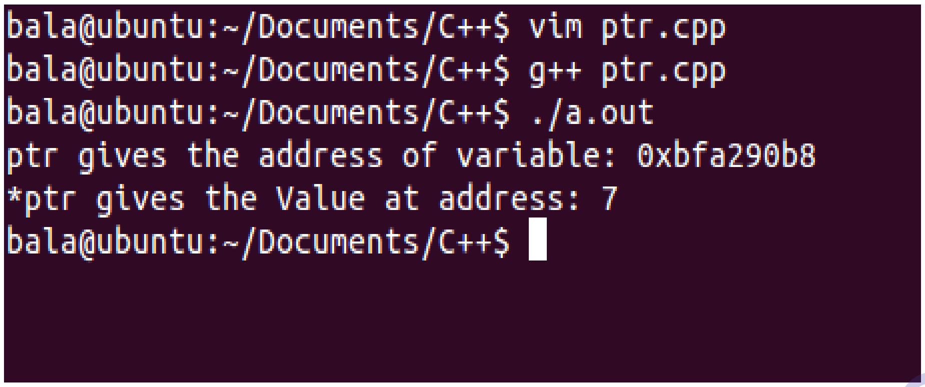


```
#include <iostream>

using namespace std;

int main()
{
        int a=7;
        int *ptr=&a; // assiging address of variable value to pointer
        cout << "ptr gives the address of variable: ";
        cout << ptr << endl;  // gives the address of the variable
        cout << "*ptr gives the Value at address: ";
        cout << *ptr << endl; // value at that address
        return 0;
}
~
~
~
```

Figure2: Pointer example program

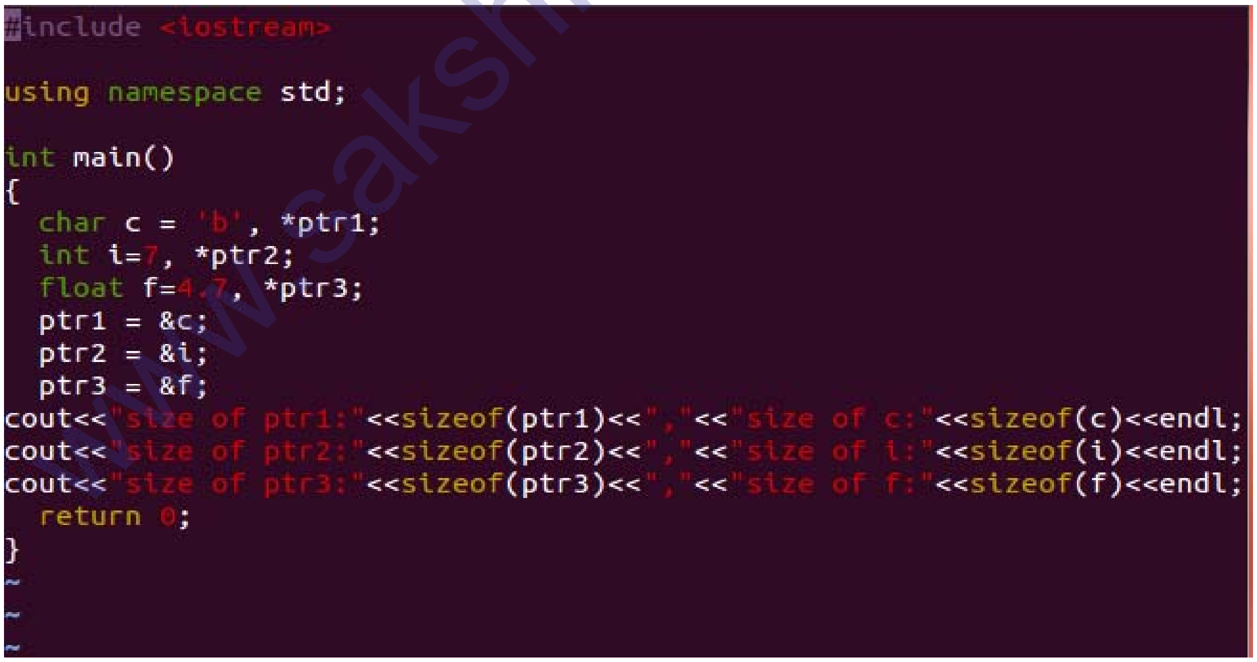


```
bala@ubuntu:~/Documents/C++$ vim ptr.cpp
bala@ubuntu:~/Documents/C++$ g++ ptr.cpp
bala@ubuntu:~/Documents/C++$ ./a.out
ptr gives the address of variable: 0xbfa290b8
*ptr gives the Value at address: 7
bala@ubuntu:~/Documents/C++$
```

Figure3: Output of the program

Size of pointer

The size of pointer is depends upon the architecture. For 32-bit architectures uses 32-bits i.e. 4 bytes and 64-bit architecture uses 64 bits i.e. 8 bytes. Pointers reserve fixed size of memory irrespective of their base type, still we are mentioning data type because the value of pointer only tells the address of the starting byte but it does not know how many bytes of data will be retrieved, so by using data type it knows that how many number of bytes should be retrieved from memory. The following example program shows the size of pointer variables and size of values dereference by them.

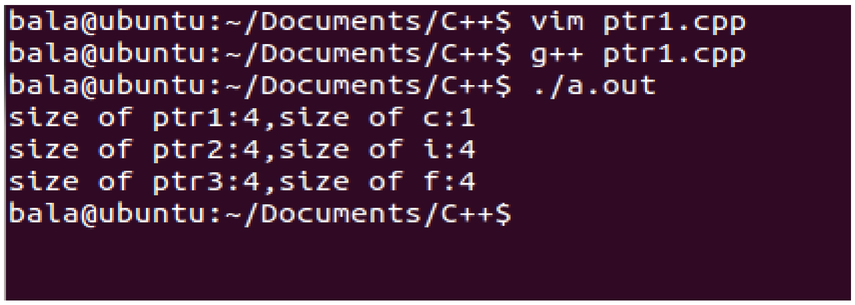


```
#include <iostream>

using namespace std;

int main()
{
  char c = 'b', *ptr1;
  int i=7, *ptr2;
  float f=4.7, *ptr3;
  ptr1 = &c;
  ptr2 = &i;
  ptr3 = &f;
cout<<"size of ptr1:"<<sizeof(ptr1)<<","<<"size of c:"<<sizeof(c)<<endl;
cout<<"size of ptr2:"<<sizeof(ptr2)<<","<<"size of i:"<<sizeof(i)<<endl;
cout<<"size of ptr3:"<<sizeof(ptr3)<<","<<"size of f:"<<sizeof(f)<<endl;
  return 0;
}
~
~
~
```

Figure4: Pointer example with size of operator



```
bala@ubuntu:~/Documents/C++$ vim ptr1.cpp
bala@ubuntu:~/Documents/C++$ g++ ptr1.cpp
bala@ubuntu:~/Documents/C++$ ./a.out
size of ptr1:4,size of c:1
size of ptr2:4,size of i:4
size of ptr3:4,size of f:4
bala@ubuntu:~/Documents/C++$
```

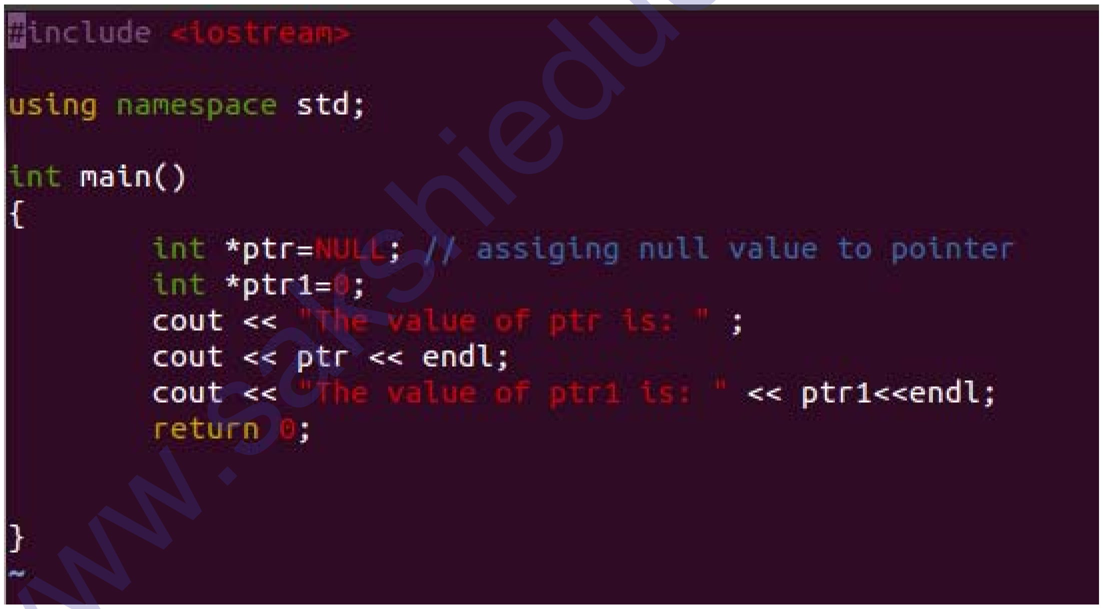
Figure5: Output of the program

Null pointers

A pointer holding a null value is called a null pointer i.e. initializes a pointer to '0' or NULL means points to nothing. While declaring pointers if unable to initialize with exact value then initialize as NULL pointers. NULL pointers are safer than the uninitialized pointers because the computer stops the program immediately.

Example:

int *ptr = 0; or int *ptr=NULL;



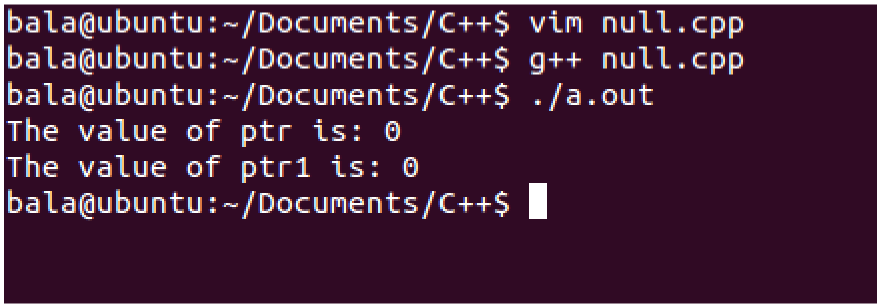
```
#include <iostream>

using namespace std;

int main()
{
        int *ptr=NULL; // assiging null value to pointer
        int *ptr1=0;
        cout << "The value of ptr is: " ;
        cout << ptr << endl;
        cout << "The value of ptr1 is: " << ptr1<<endl;
        return 0;


}
~
```

Figure 6: NULL Pointer Example



```
bala@ubuntu:~/Documents/C++$ vim null.cpp
bala@ubuntu:~/Documents/C++$ g++ null.cpp
bala@ubuntu:~/Documents/C++$ ./a.out
The value of ptr is: 0
The value of ptr1 is: 0
bala@ubuntu:~/Documents/C++$ 
```

Figure7: Output of the program

Pointer Arithmetic

The arithmetic operations are possible on pointers but not all arithmetic operations. The only valid operator's that can be used on pointers are addition, subtraction, increment and decrement. The pointer arithmetic is somewhat different from ordinary arithmetic operations. Here all arithmetic operations are performed relative to the size of base type of pointer. To better understanding pointer arithmetic let us consider an example integer pointer 'ptr' which points to address 8000 and perform increment operation on 'ptr' is 'ptr++'. Now the 'ptr' will point to the location 8004 instead of 8001. This is because the size of int is four bytes. This operation will move the pointer to next memory location without changing actual value at the memory location. If 'ptr' is a character pointer whose address is 2000 and perform increment operation on this will point to memory location 2001 because the size of char is one byte.

Pointer comparisons

The pointers can be compared with relational operators such as '= =', '! =', '>', '>=', '<', '<='. The operators '= =', '! =' are used to compare two pointers for finding whether they contain same address or not and they are equal if both pointers contain NULL or they points to address of same variable. The operators '>', '>=', '<', '<=' are used for only same type pointers. These operations are making sense only when both pointers point to elements of same array.

Pointer to pointer

A pointer to a pointer is a form of multiple indirection of pointer that is a pointer points to another pointer which points to the variable. As shown in below figure the first pointer contains the address of second pointer which contains the address of variable which contains the value. The pointer to pointer concept can extended to any limit but most of the time pointer to pointer is used. The pointer to pointer variable is declared as an additional asterisk comes in front of pointer variable as shown in below example,

Example:

int **ptr;

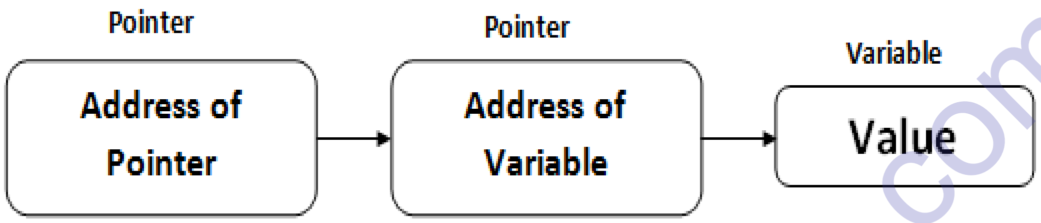
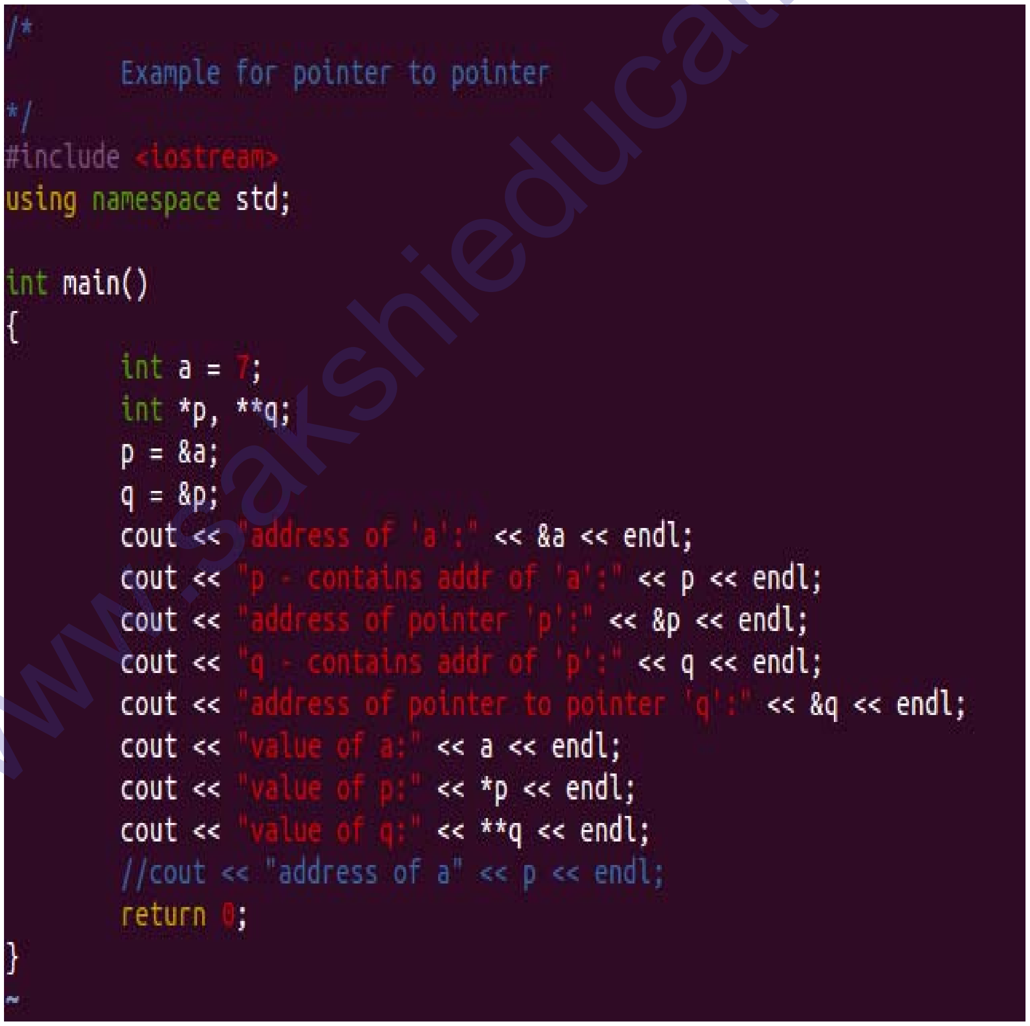


Figure8: Pointer to Pointer



```
/*
        Example for pointer to pointer
*/
#include <iostream>
using namespace std;

int main()
{
        int a = 7;
        int *p, **q;
        p = &a;
        q = &p;
        cout << "address of 'a':" << &a << endl;
        cout << "p - contains addr of 'a':" << p << endl;
        cout << "address of pointer 'p':" << &p << endl;
        cout << "q - contains addr of 'p':" << q << endl;
        cout << "address of pointer to pointer 'q':" << &q << endl;
        cout << "value of a:" << a << endl;
        cout << "value of p:" << *p << endl;
        cout << "value of q:" << **q << endl;
        //cout << "address of a" << p << endl;
        return 0;
}
~
```

Figure9: Pointer-to-Pointer example program

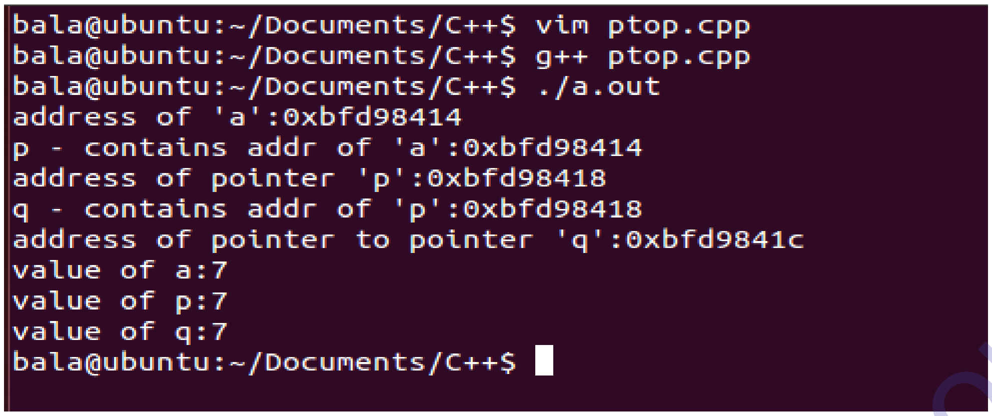

```
bala@ubuntu:~/Documents/C++$ vim ptop.cpp
bala@ubuntu:~/Documents/C++$ g++ ptop.cpp
bala@ubuntu:~/Documents/C++$ ./a.out
address of 'a':0xbfd98414
p - contains addr of 'a':0xbfd98414
address of pointer 'p':0xbfd98418
q - contains addr of 'p':0xbfd98418
address of pointer to pointer 'q':0xbfd9841c
value of a:7
value of p:7
value of q:7
bala@ubuntu:~/Documents/C++$ 
```

Figure10: Output of the program

Void Pointer

The void pointer is also called as generic pointer. A void pointer can hold the address of any data type. Void pointer is a special type of pointer which can be pointed to any data type. The void pointer can be assigned to a pointer of any data type and address of any data type without any explicit casts. The void pointer is declared like a normal pointer using the 'void' keyword as pointers type. We cannot operate on the object pointed by void pointer because the type is unknown. The void pointers can be used compare with another address.

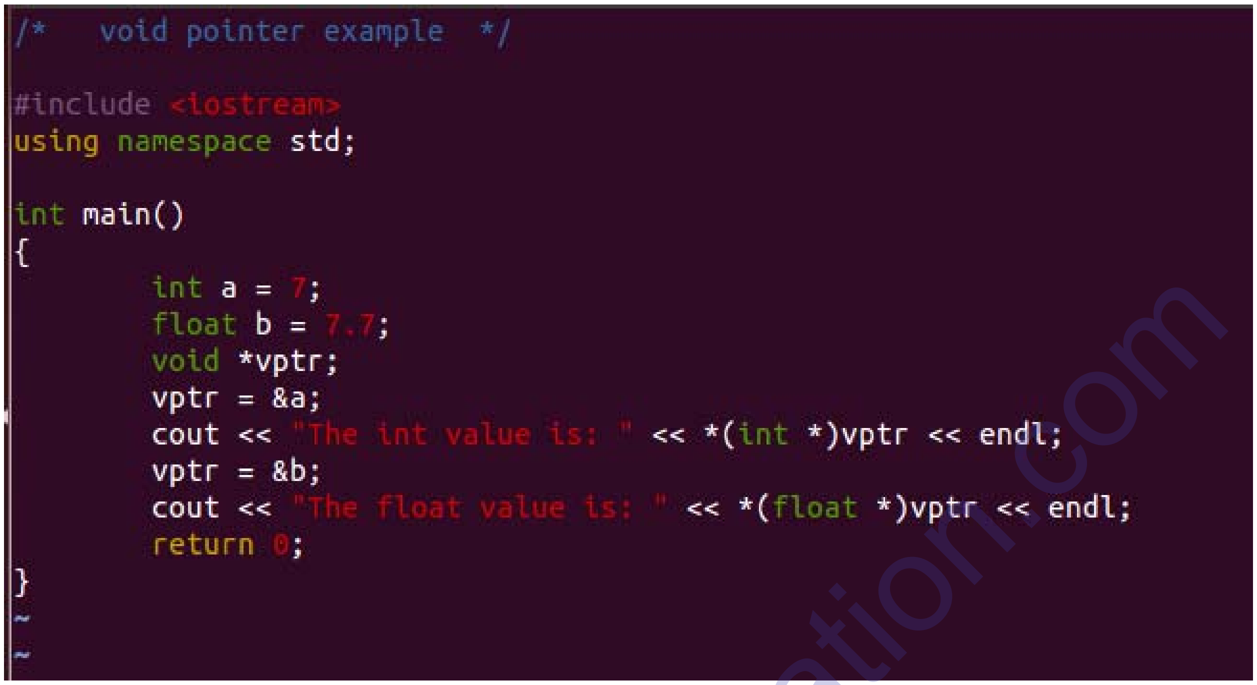
Syntax:

void *ptr; → ptr is a void pointer.

Examples:

```
int i = 7;
float f = 7.7;
void *ptr;
ptr = &i;
ptr = &f;
```

A void pointer can't be dereferences by using indirection operator, before dereferencing it should be type cast to appropriate pointer data type. The pointer arithmetic operations are not performed on void pointers without typecasting. The void pointers are mostly used to pass pointer to functions which have to perform same operations on different data types.

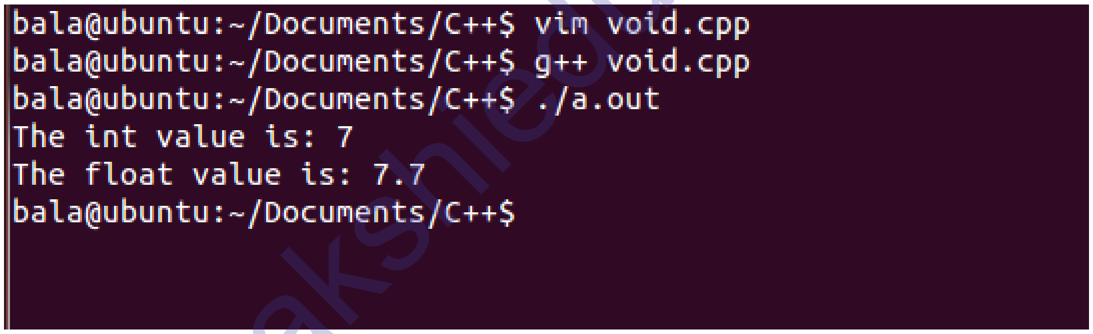


```
/*   void pointer example  */

#include <iostream>
using namespace std;

int main()
{
        int a = 7;
        float b = 7.7;
        void *vptr;
        vptr = &a;
        cout << "The int value is: " << *(int *)vptr << endl;
        vptr = &b;
        cout << "The float value is: " << *(float *)vptr << endl;
        return 0;
}
~
~
```

Figure11: Void pointer example



```
bala@ubuntu:~/Documents/C++$ vim void.cpp
bala@ubuntu:~/Documents/C++$ g++ void.cpp
bala@ubuntu:~/Documents/C++$ ./a.out
The int value is: 7
The float value is: 7.7
bala@ubuntu:~/Documents/C++$
```

Figure12: Output of the program

Pointers and Constants

Non const pointer to const value

A non constant pointer to a const value is a pointer that points to a constant value. To declare a pointer to const value, use const keyword before the data type. The value pointed to cannot be changed but the pointer can be changed to another value.

Example:

```
const int a = 7;
const int *ptr = &a;
int a = 7;
const int *ptr = &a;
```

Const pointers to non const value

A const pointer is a pointer whose value cannot be changed after initialization. The const pointer can be declared using the const keyword between the asterisk and pointer name. The value pointed to can be changed but pointer cannot be changed to point another value.

```
int a = 7;
int *const ptr = &a;
```

Const pointer to a const value

A const pointer to const value is declared by using const keyword both before the type and variable name. This pointer cannot be set to point another address nor can the value it is pointing to be changed through the pointer. The value pointed to cannot be changed and pointer cannot be changed to point to another value.

```
int a = 7;
const int *const ptr = &a;
```

Non const pointer to non const value

The value pointed to can be changed and the pointer can be changed to point to another value.

```
int a = 7;
int *ptr = &a;
```

Const pointers are used in function parameters to ensure that the function does not accidently change the passed in the argument. A non const pointer can be redirected to point to other addresses. A const pointer always points to the same address and this address cannot be changed. A pointer to non const value can change the value it is pointing to.