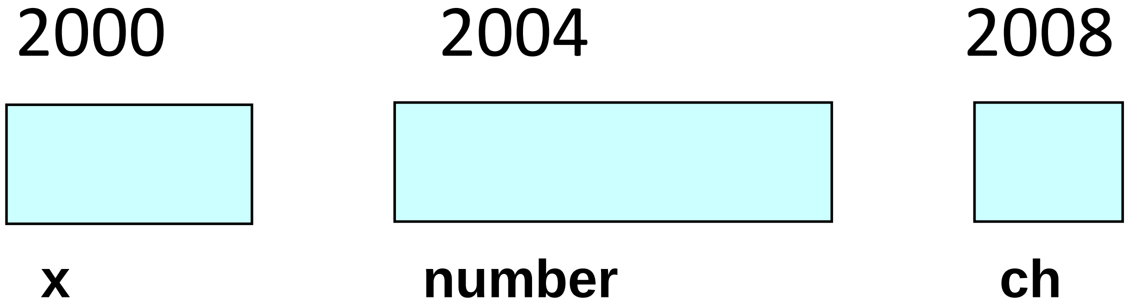
Pointers

Addresses in Memory

- When a variable is declared, enough memory to hold a value of that type is allocated for it at an unused memory location. This is the address of the variable

```
int     x;
float   number;
char    ch;
```



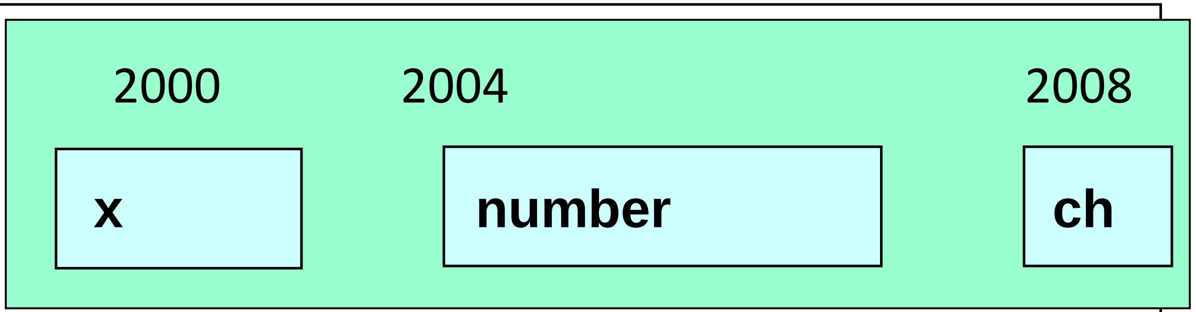
2000 2004 2008

x number ch

Obtaining Memory Addresses

- The address of a *non-array variable* can be obtained by using the address-of operator **&**

```
int     x;
float   number;
char    ch;

cout << “Address of x is “ << &x << endl;

cout << “Address of number is “ << &number << endl;

cout << “Address of ch is “ << &ch << endl;
```



What is a pointer variable?

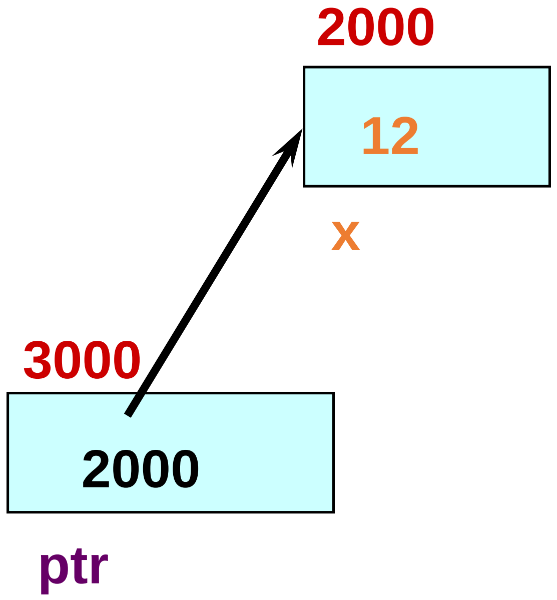
- A pointer variable is a ***variable whose value is the address of a location in memory.***

- To declare a pointer variable, you must specify the type of value that the pointer will point to, for example,

```
int  *ptr; // ptr will hold the address of an int
char  *q;   // q will hold the address of a char
```

Using a Pointer Variable

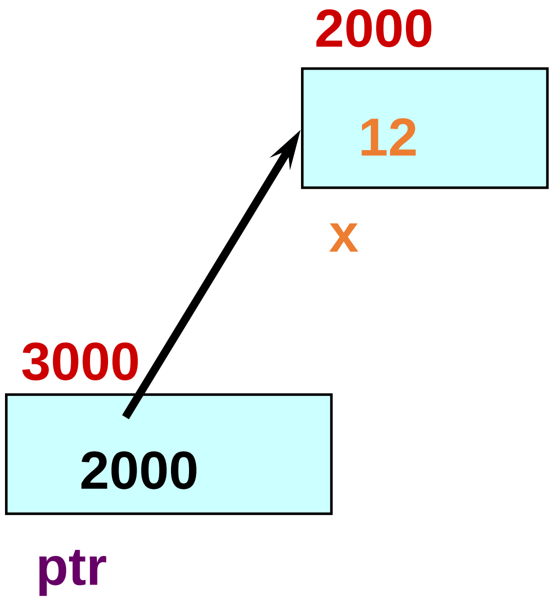
```
int  x;
 x = 12;

 int *ptr;
 ptr = &x;
```



NOTE: Because ptr holds the address of x, we say that ptr “points to” x

* is the dereference operator

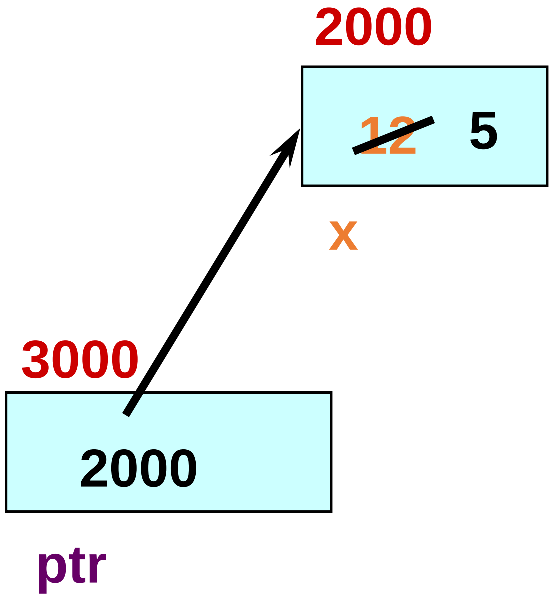
```
int  x;
x = 12;

int *ptr;
ptr = &x;

cout  <<  *ptr;
```



NOTE: The value pointed to by ptr is denoted by *ptr

Using the Dereference Operator

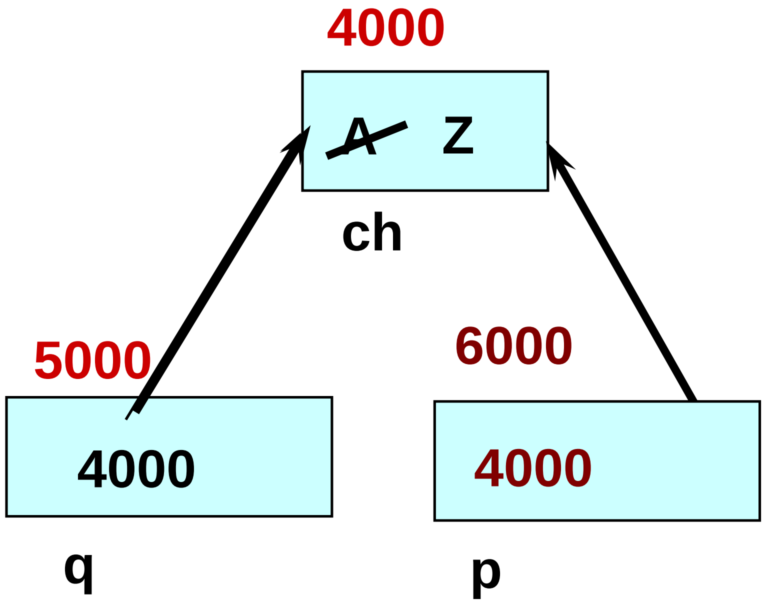
```
int  x;
x = 12;



int *ptr;
ptr = &x;

*ptr = 5;
```



// changes the value at the address ptr points to 5

Self –Test on Pointers

```
char  ch;
ch =  ‘A’;

char *q;
q  = &ch;

*q = ‘Z’;
char *p;
p = q;
```



```
// the rhs has value 4000

// now p and q both point to ch
```

Pointers and arrays

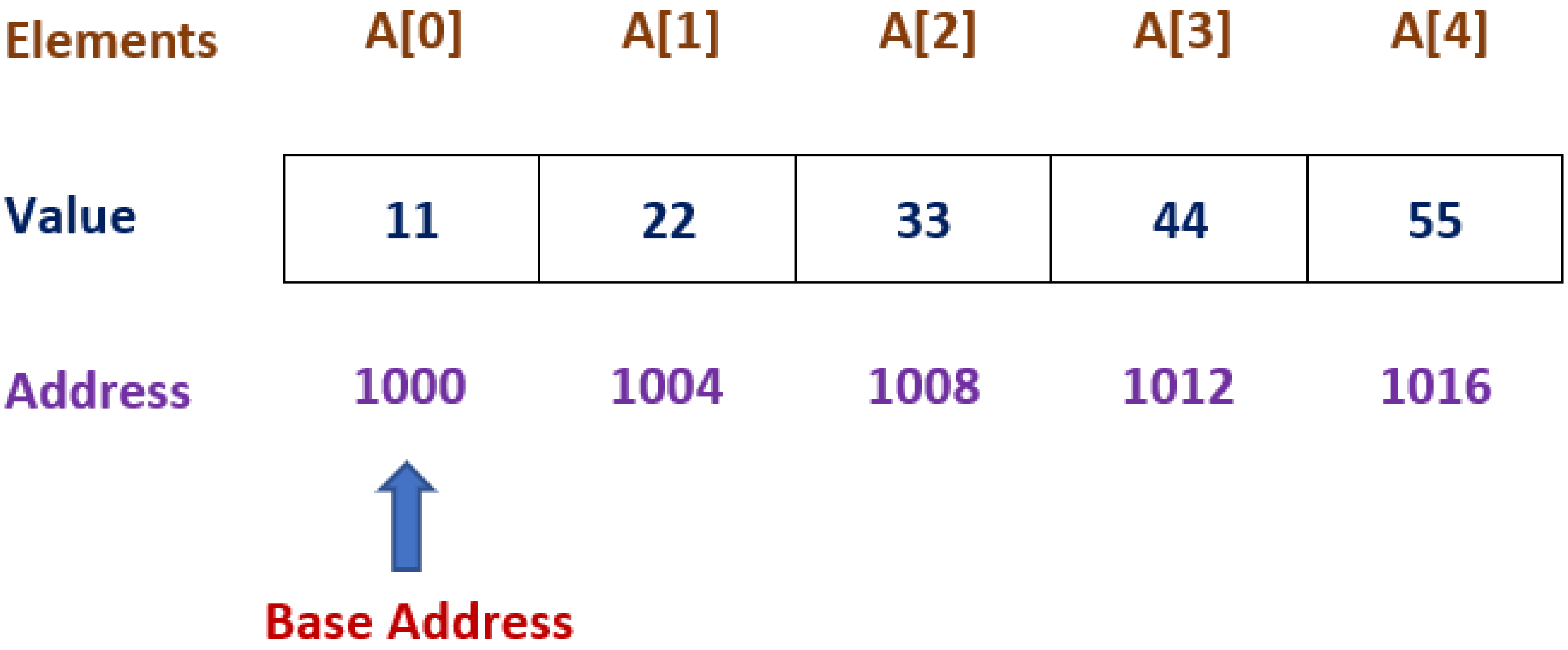
- When an array is declared, the compiler allocates a **base address** and sufficient amount of storage to contain all the elements of the array in contiguous memory locations
- The base address is the location of the first element (index 0) of the array.
- The compiler also defines the **array name as a constant pointer to the first element**.

compiler converts: **x[i] = *(x + i)**

Suppose we declare an array A as follows:

int A[5] ={ 11,22,33,44,55 };

Suppose the base address of A is 1000, and assuming that each integer requires 4 bytes, the five elements will be stored as follows.



Elements	A[0]	A[1]	A[2]	A[3]	A[4]
Value	11	22	33	44	55
Address	1000	1004	1008	1012	1016

Base Address

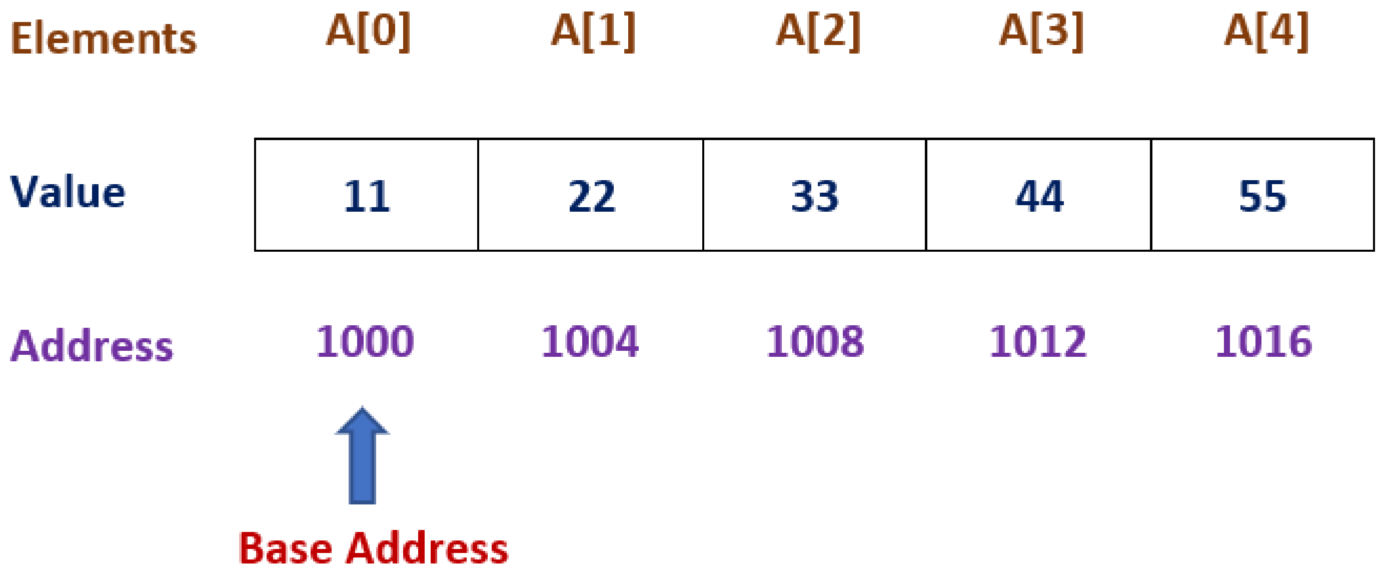
- The name A is defined as a constant pointer pointing to the first element, A[0] and therefore the value of A is 1000, the location where A[0] is stored.

  That is **A =&A[0] =1000**

- If we declare p as an integer pointer, then we can make the pointer p to the array A by the following statement.

  p=A; This is equivalent to **p=&A[0]**;

Now we can access every value of A using p++ to move from one element to another.



Elements	A[0]	A[1]	A[2]	A[3]	A[4]
Value	11	22	33	44	55
Address	1000	1004	1008	1012	1016

p= &A[0]   (=1000)

p+1=&A[1] (=1004)

p+2=&A[2] (=1008)

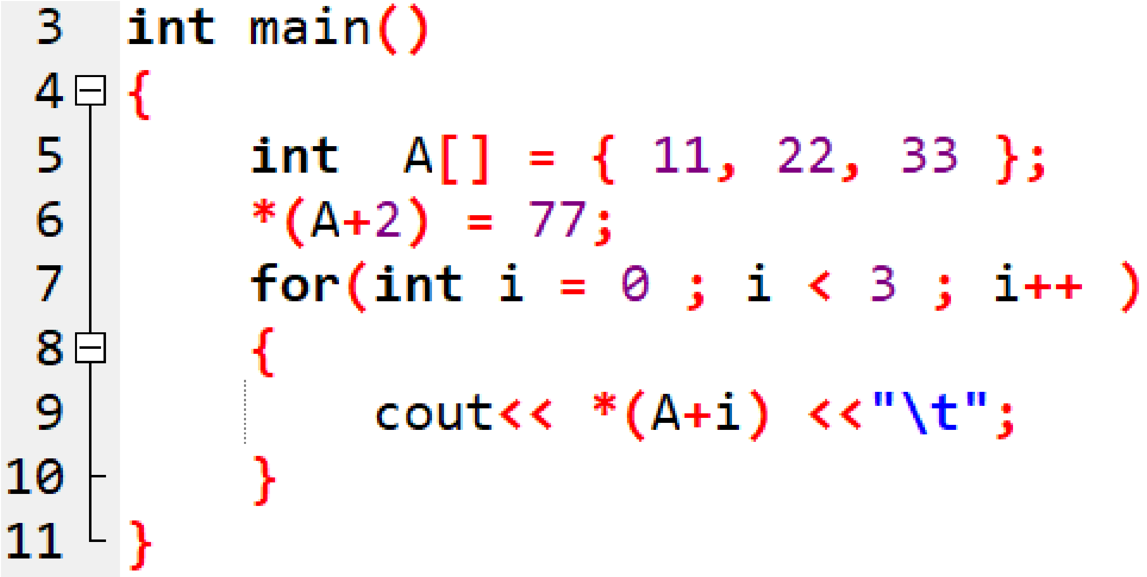
p+3=&A[3] (=1012)

p+4=&A[4] (=1016)

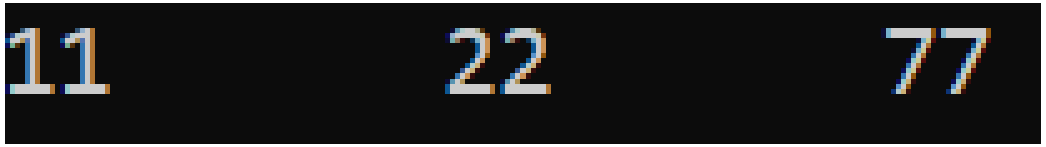
Address of A[3] = base address + (3 x scale factor of integer)

=1000 +(3*4) =1012

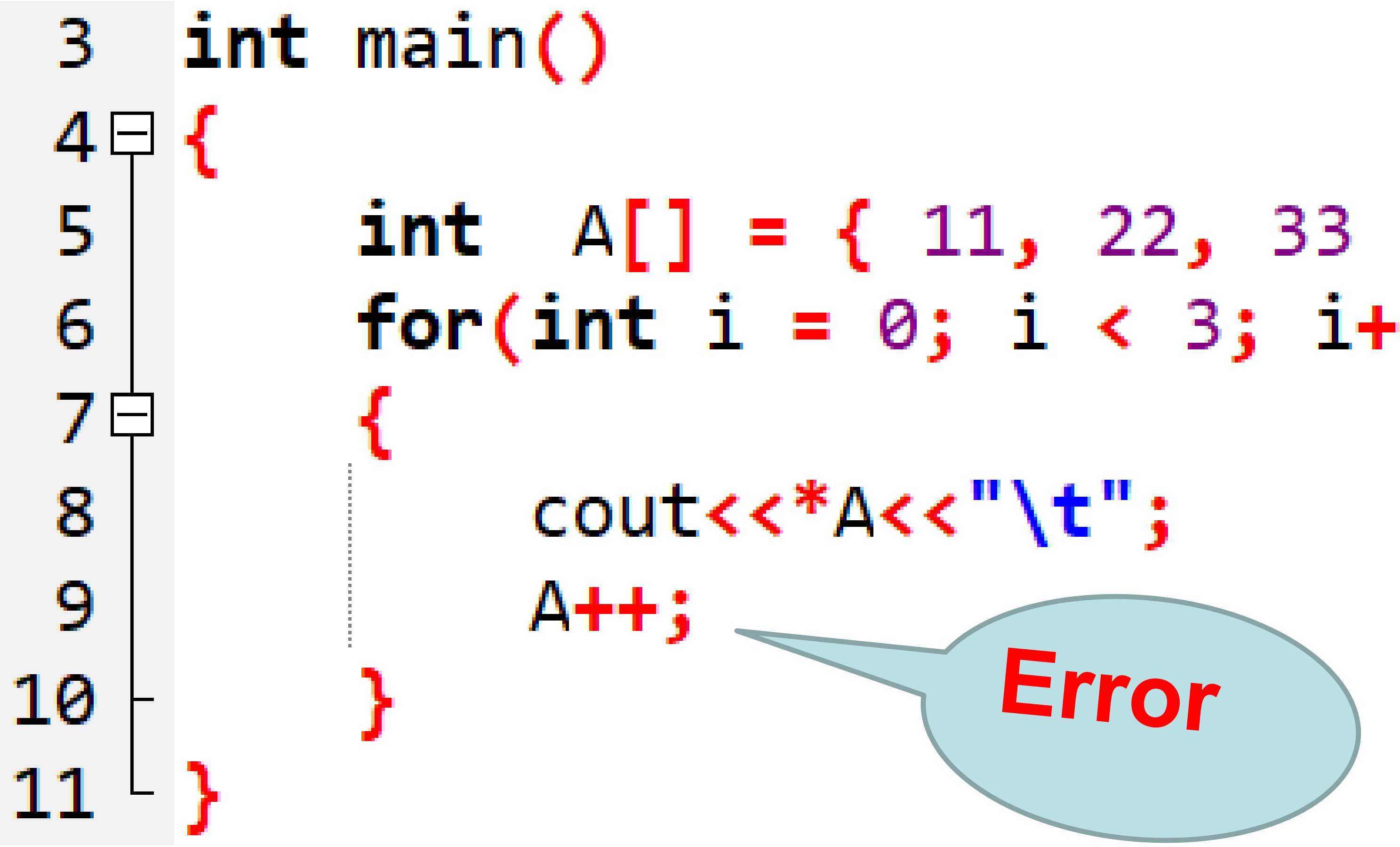
Accessing array elements using constant pointer



```
int main()
{
	int  A[] = { 11, 22, 33 };
	*(A+2) = 77;
	for(int i = 0 ; i < 3 ; i++ )
	{
		cout<< *(A+i) <<"\t";
	}
}
```



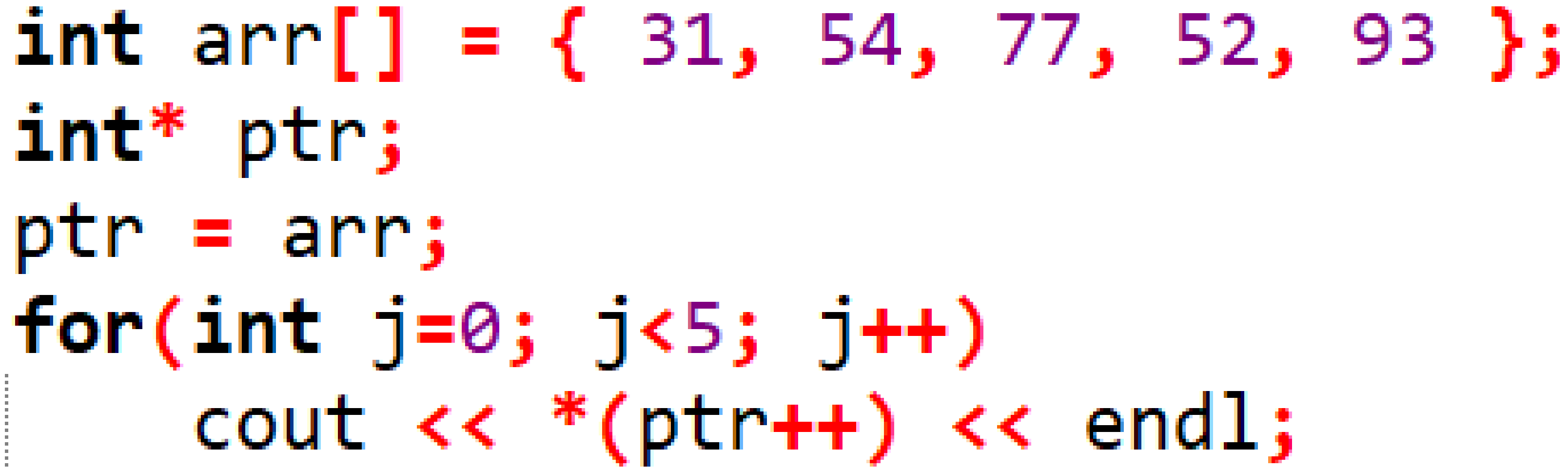
```
11      22      77
```



```
int main()
{
    int  A[] = { 11, 22, 33 };
    for(int i = 0; i < 3; i++)
    {
        cout<<*A<<"\t";
        A++;
    }
}
```

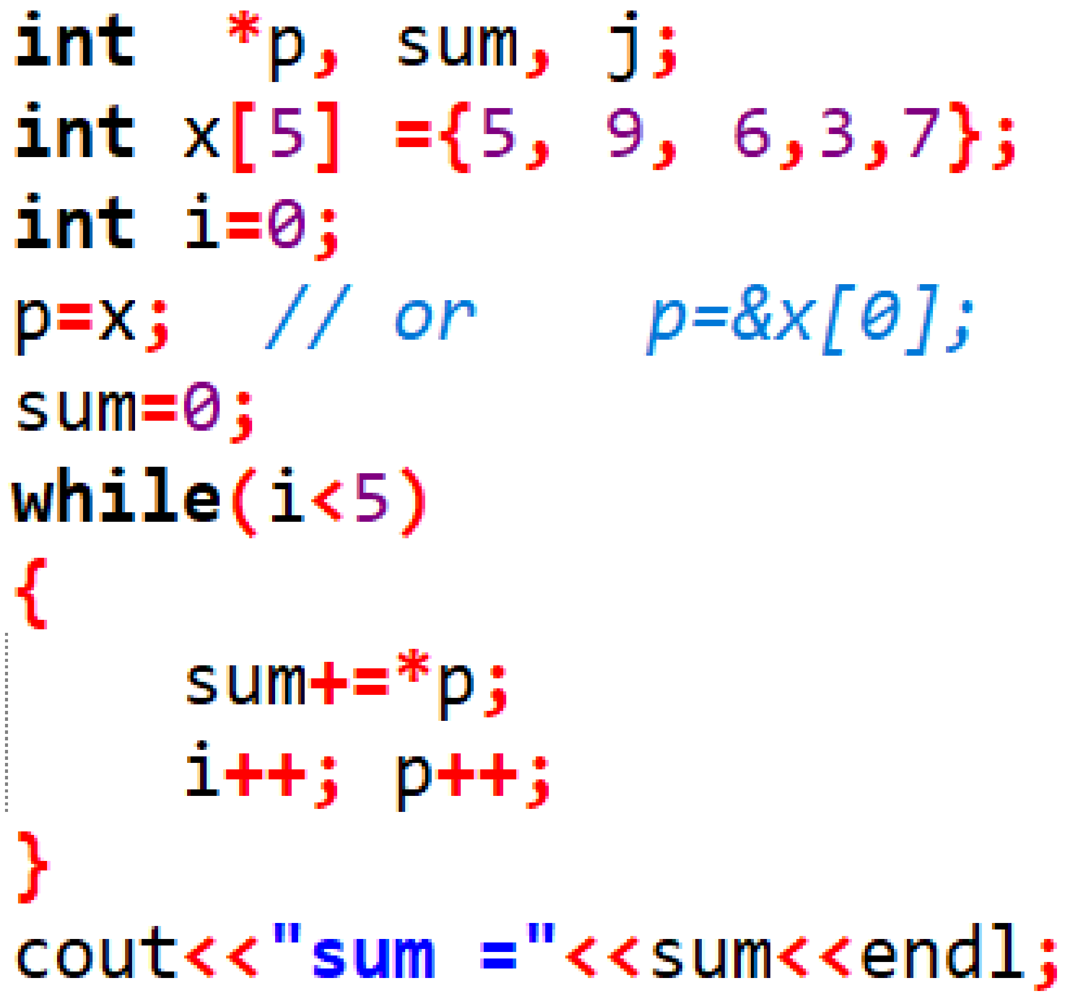
Error

Array accessed with pointer



```
int arr[] = { 31, 54, 77, 52, 93 };
int* ptr;
ptr = arr;
for(int j=0; j<5; j++)
    cout << *(ptr++) << endl;
```

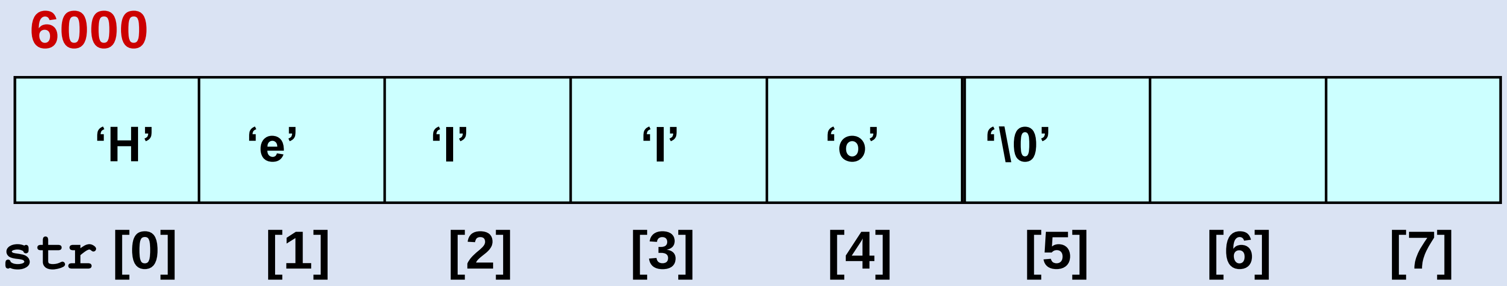
- Write a C++ program to compute the sum of all elements stored in an array using pointers.



```
int  *p, sum, j;
int x[5] ={5, 9, 6,3,7};
int i=0;
p=x;  // or    p=&x[0];
sum=0;
while(i<5)
{
    sum+=*p;
    i++; p++;
}
cout<<"sum ="<<sum<<endl;
```

If you remember

- char str [8];
- **str** is the ***base address*** of the array.
- We say ***str is a pointer*** because its value is an address.
- It is a pointer constant because the value of str itself cannot be changed by assignment. It “points” to the memory location of a char.

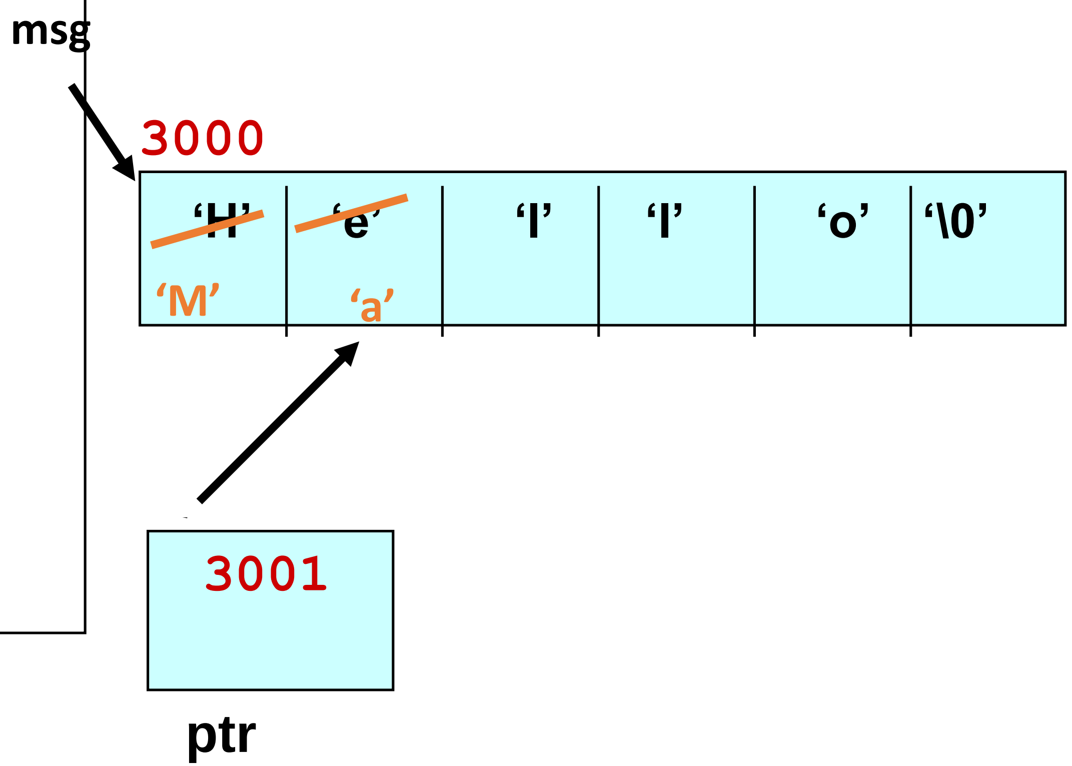


6000

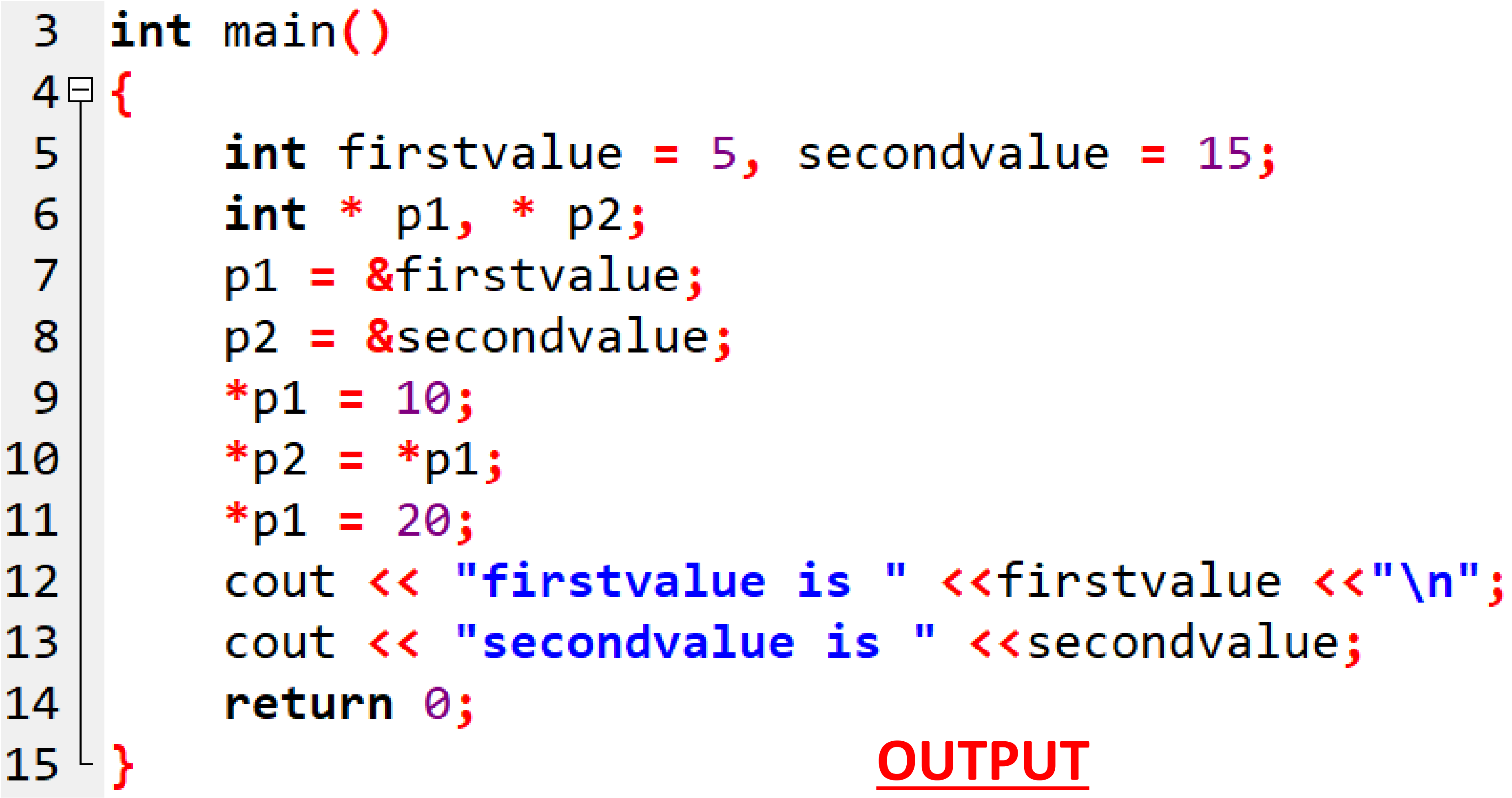
‘H’	‘e’	‘l’	‘l’	‘o’	‘\0’		
str [0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]

Using a Pointer to Access the Elements of a String

```
char   msg[ ] =“Hello”;
char *ptr;
ptr  =  msg;
*ptr  = ‘M’ ;
ptr++;

*ptr = ‘a’;
```

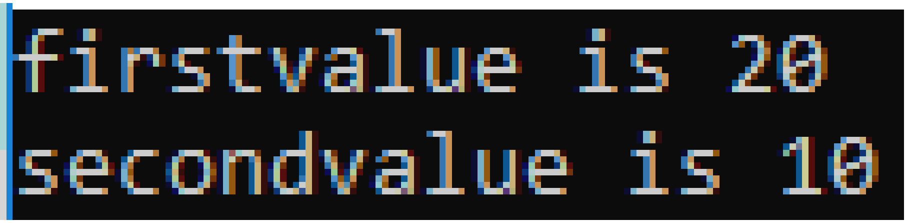


Question-1: What is the output ?



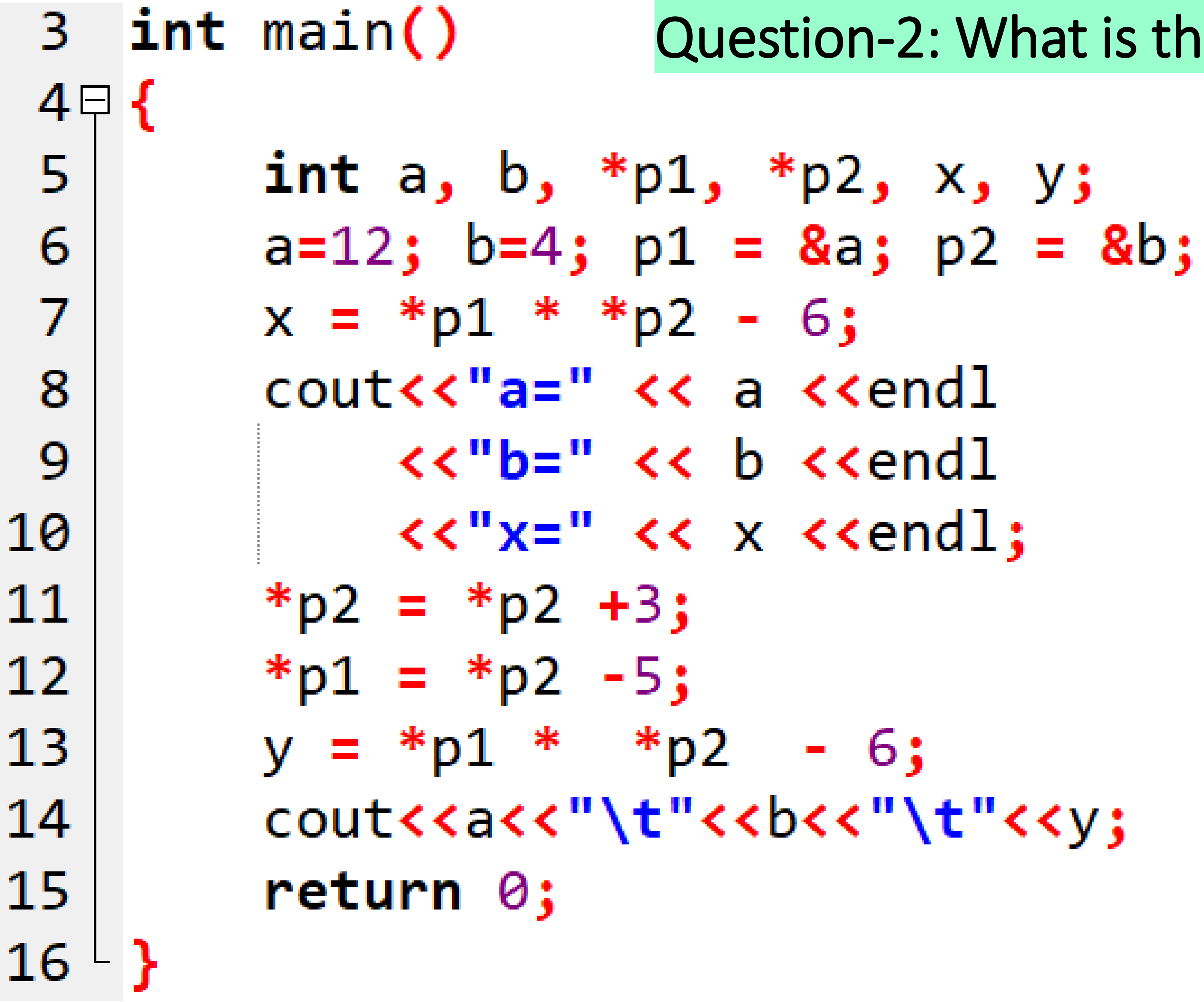
```
int main()
{
    int firstvalue = 5, secondvalue = 15;
    int * p1, * p2;
    p1 = &firstvalue;
    p2 = &secondvalue;
    *p1 = 10;
    *p2 = *p1;
    *p1 = 20;
    cout << "firstvalue is " <<firstvalue <<"\n";
    cout << "secondvalue is " <<secondvalue;
    return 0;
}
```

OUTPUT



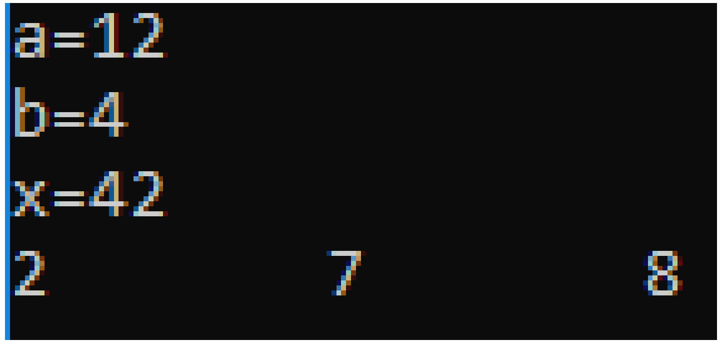
```
firstvalue is 20
secondvalue is 10
```

Question-2: What is the output ?



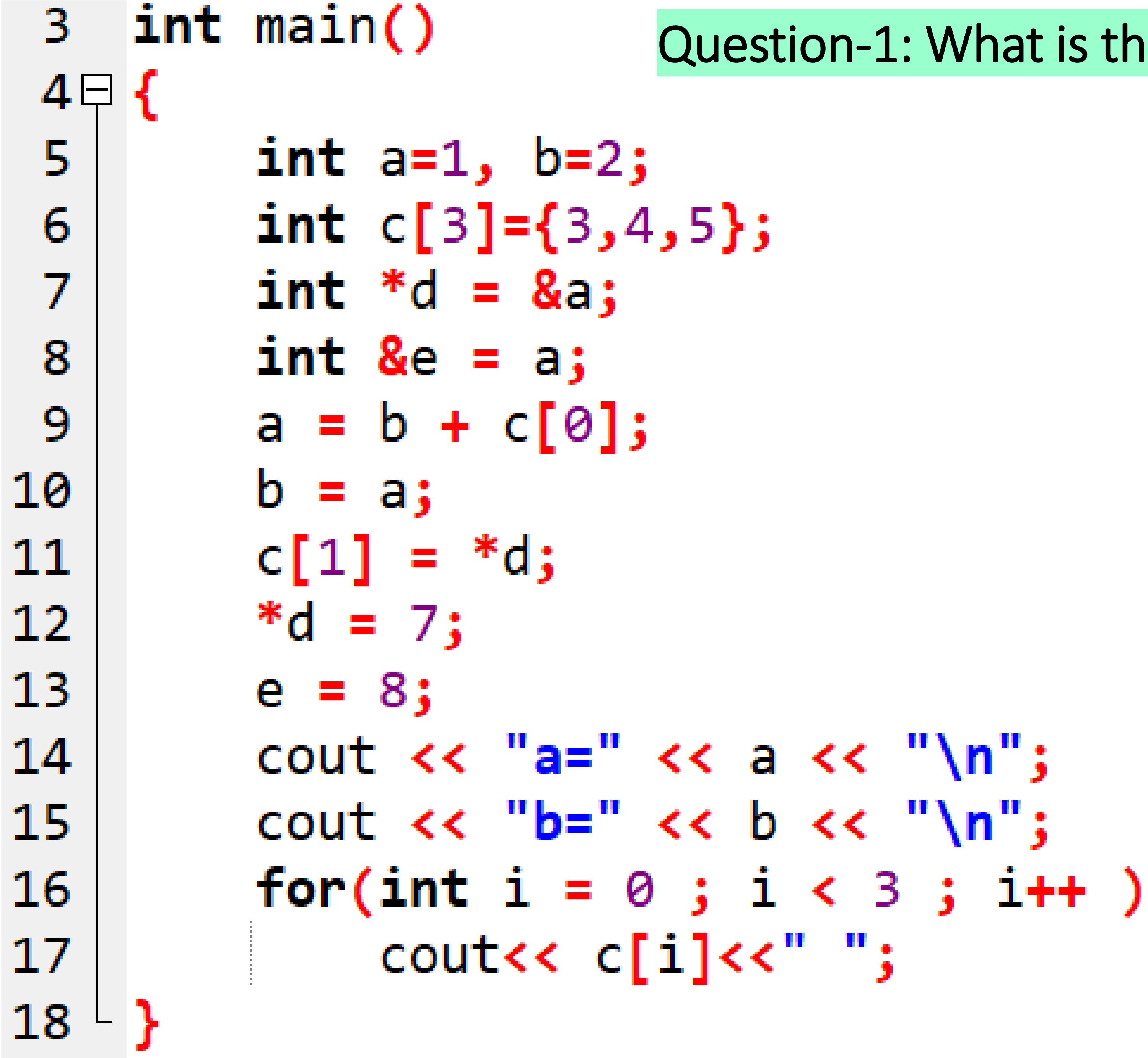
```cpp
int main()
{
	int a, b, *p1, *p2, x, y;
	a=12; b=4; p1 = &a; p2 = &b;
	x = *p1 * *p2 - 6;
	cout<<"a=" << a <<endl
	    <<"b=" << b <<endl
	    <<"x=" << x <<endl;
	*p2 = *p2 +3;
	*p1 = *p2 -5;
	y = *p1 *  *p2  - 6;
	cout<<a<<"\t"<<b<<"\t"<<y;
	return 0;
}
```

OUTPUT



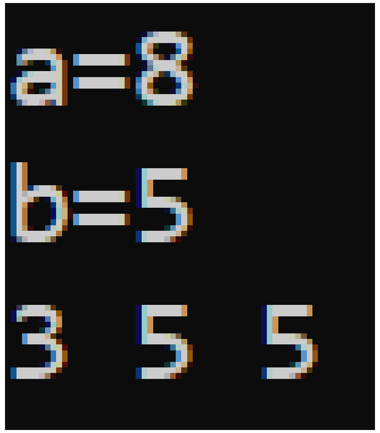
```
a=12
b=4
x=42
2       7       8
```

Question-1: What is the output ?



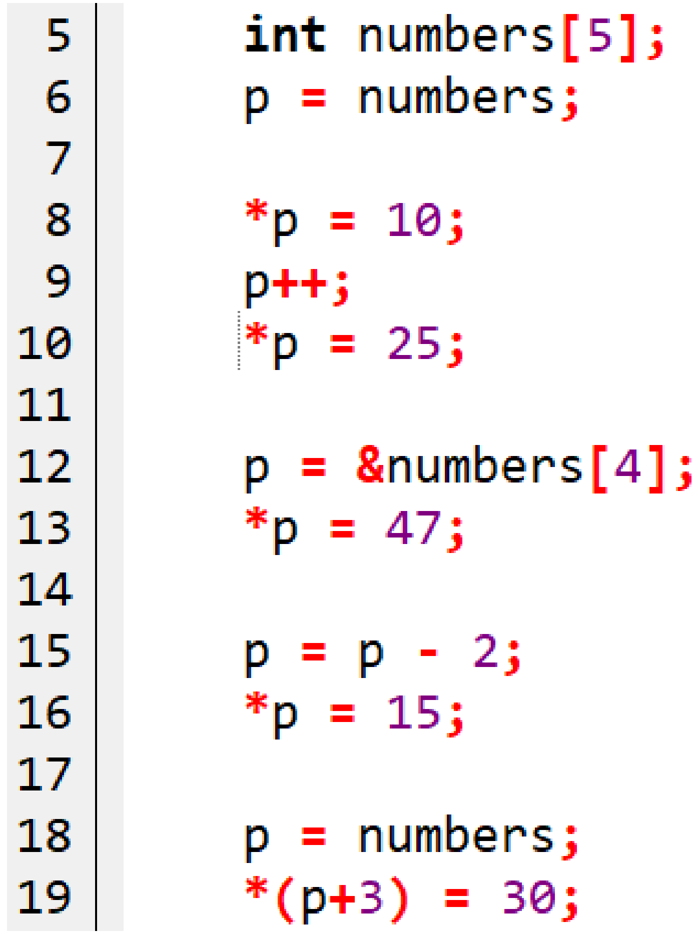
```cpp
int main()
{
    int a=1, b=2;
    int c[3]={3,4,5};
    int *d = &a;
    int &e = a;
    a = b + c[0];
    b = a;
    c[1] = *d;
    *d = 7;
    e = 8;
    cout << "a=" << a << "\n";
    cout << "b=" << b << "\n";
    for(int i = 0 ; i < 3 ; i++ )
        cout<< c[i]<<" ";
}
```

OUTPUT:

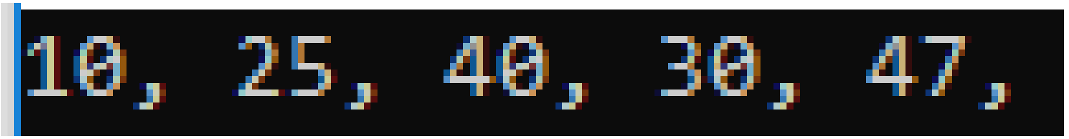


```
a=8
b=5
3 5 5
```

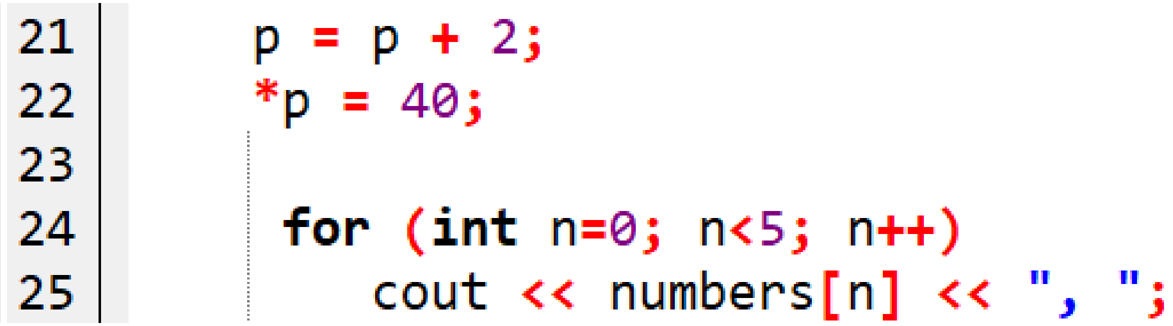
Question-2: What is the content of the array?



```
int numbers[5]; int * p;
p = numbers;

*p = 10;
p++;
*p = 25;

p = &numbers[4];
*p = 47;

p = p - 2;
*p = 15;

p = numbers;
*(p+3) = 30;
```

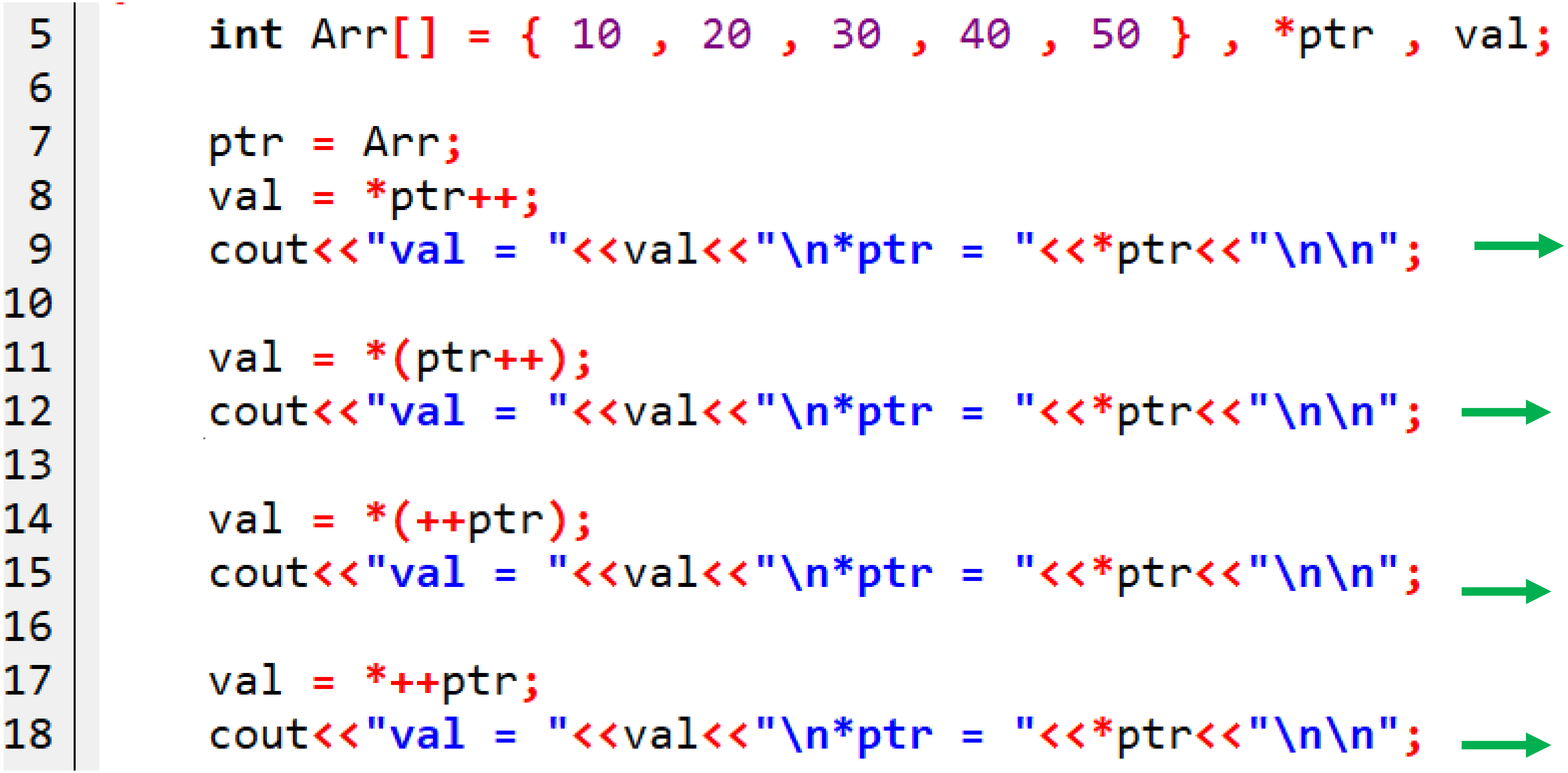


```
10, 25, 40, 30, 47,
```



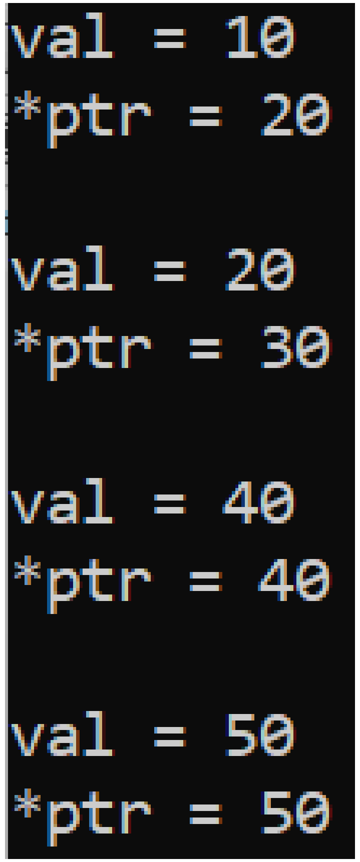
```
p = p + 2;
*p = 40;

 for (int n=0; n<5; n++)
     cout << numbers[n] << ", ";
```

Question-3:



```
int Arr[] = { 10 , 20 , 30 , 40 , 50 } , *ptr , val;

ptr = Arr;
val = *ptr++;
cout<<"val = "<<val<<"\n*ptr = "<<*ptr<<"\n\n";

val = *(ptr++);
cout<<"val = "<<val<<"\n*ptr = "<<*ptr<<"\n\n";

val = *(++ptr);
cout<<"val = "<<val<<"\n*ptr = "<<*ptr<<"\n\n";

val = *++ptr;
cout<<"val = "<<val<<"\n*ptr = "<<*ptr<<"\n\n";
```

OUTPUT:



```
val = 10
*ptr = 20

val = 20
*ptr = 30

val = 40
*ptr = 40

val = 50
*ptr = 50
```

Pointers and character strings

- The statement **char *cp =name**;

  declares **cp** as a pointer to a character and assigns address of the first character of name as the initial value.

- The statement **while(*cp!=‘\0’)** is true until the end of the string is reached.

- When the while loop is terminated, the pointer **cp** holds the address of the null character.

- The statement **length = cp – name**; gives the length of the string name.

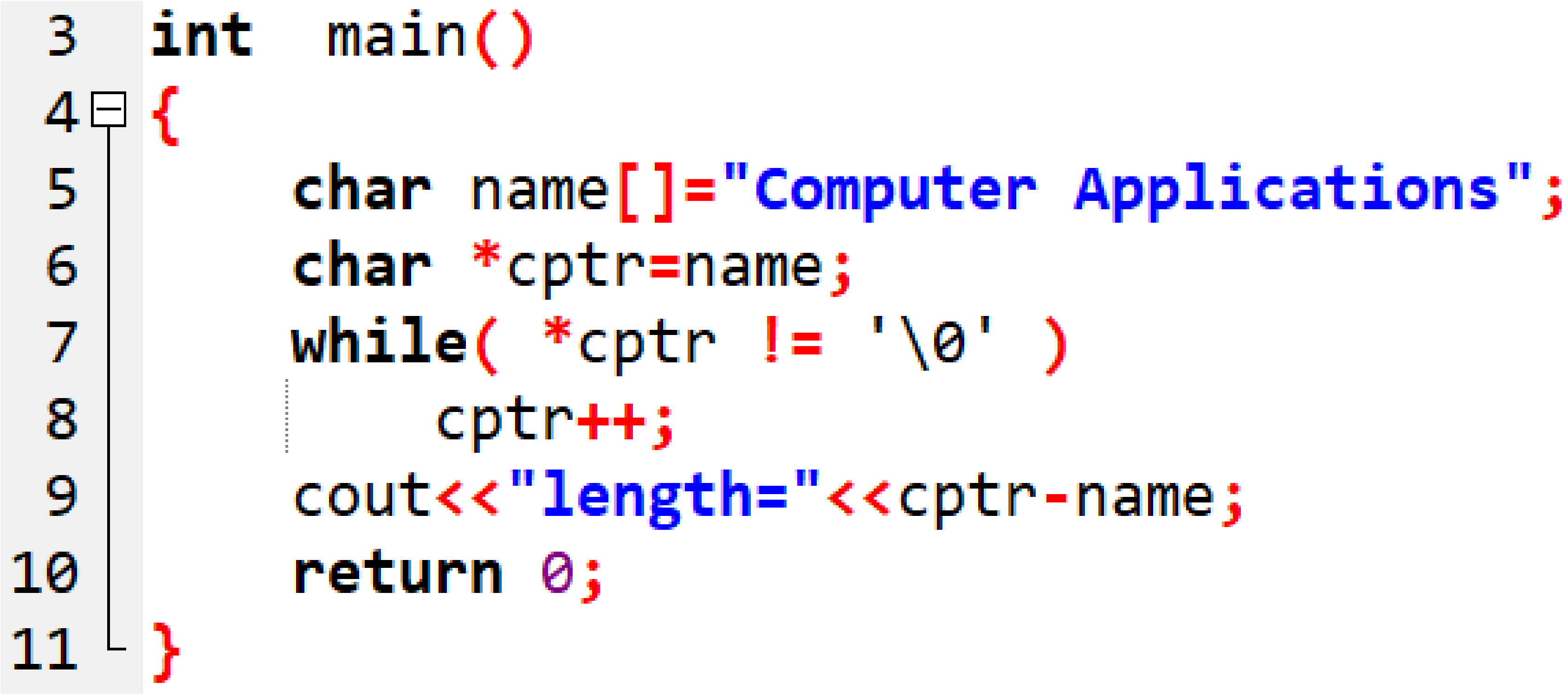
The following statements are valid.

char *name;

name =“Delhi”;

These statements will declare name as a pointer to character and assign to name the constant character string “Delhi”

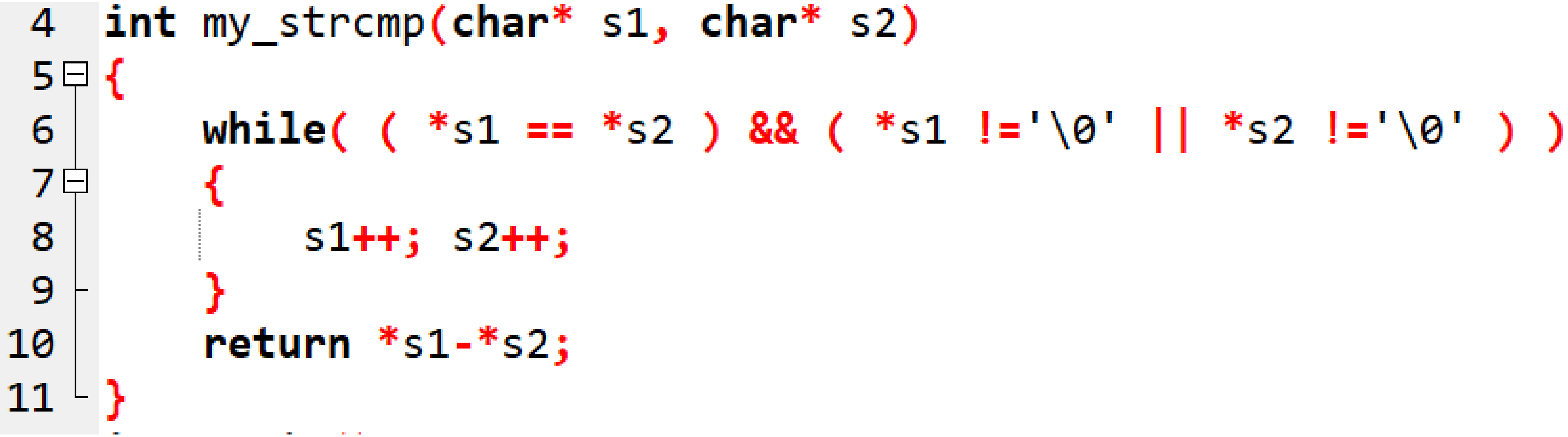
//Program to find the length of the string



```cpp
int  main()
{
    char name[]="Computer Applications";
    char *cptr=name;
    while( *cptr != '\0' )
        cptr++;
    cout<<"length="<<cptr-name;
    return 0;
}
```

Strcmp() function:

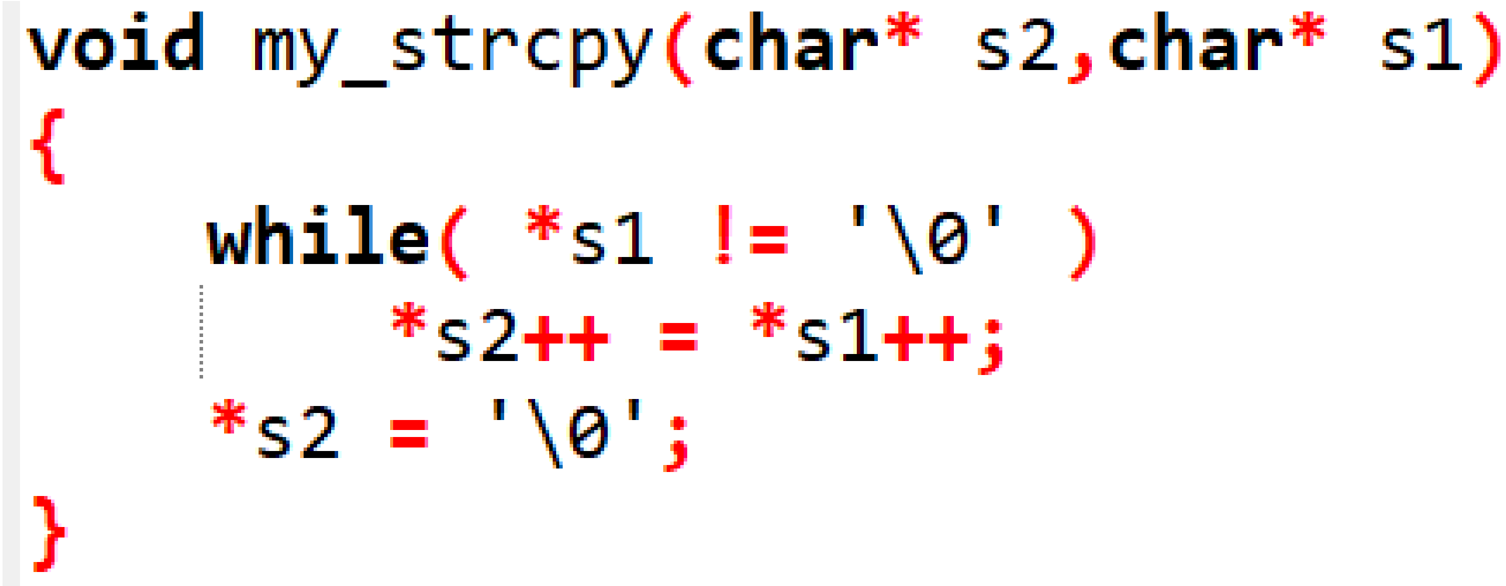
int my_strcmp (char *s1 , char *s2)



```
int my_strcmp(char* s1, char* s2)
{
    while( ( *s1 == *s2 ) && ( *s1 !='\0' || *s2 !='\0' ) )
    {
        s1++; s2++;
    }
    return *s1-*s2;
}
```

Strcpy() function:

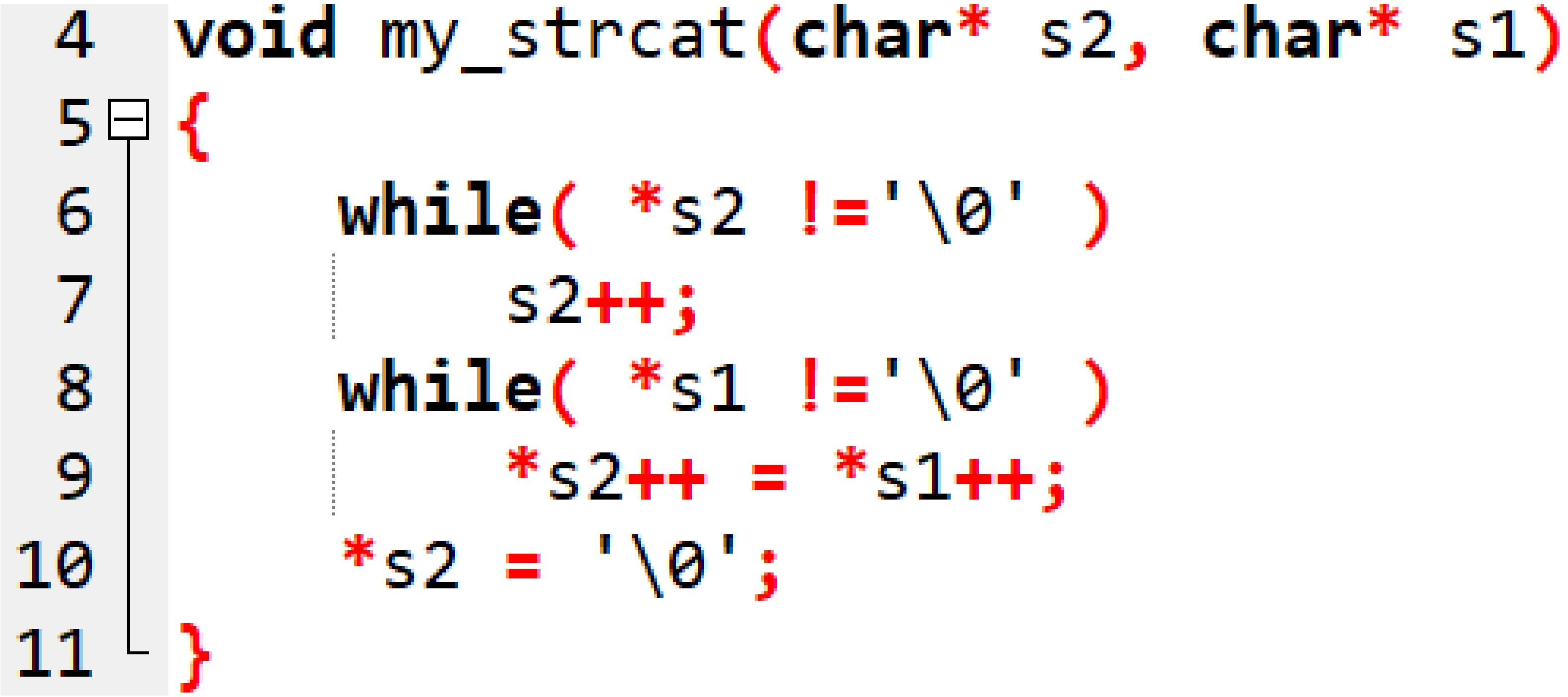
void **my_strcpy** (char *dest, char *src)



```
void my_strcpy(char* s2,char* s1)
{
    while( *s1 != '\0' )
        *s2++ = *s1++;
    *s2 = '\0';
}
```

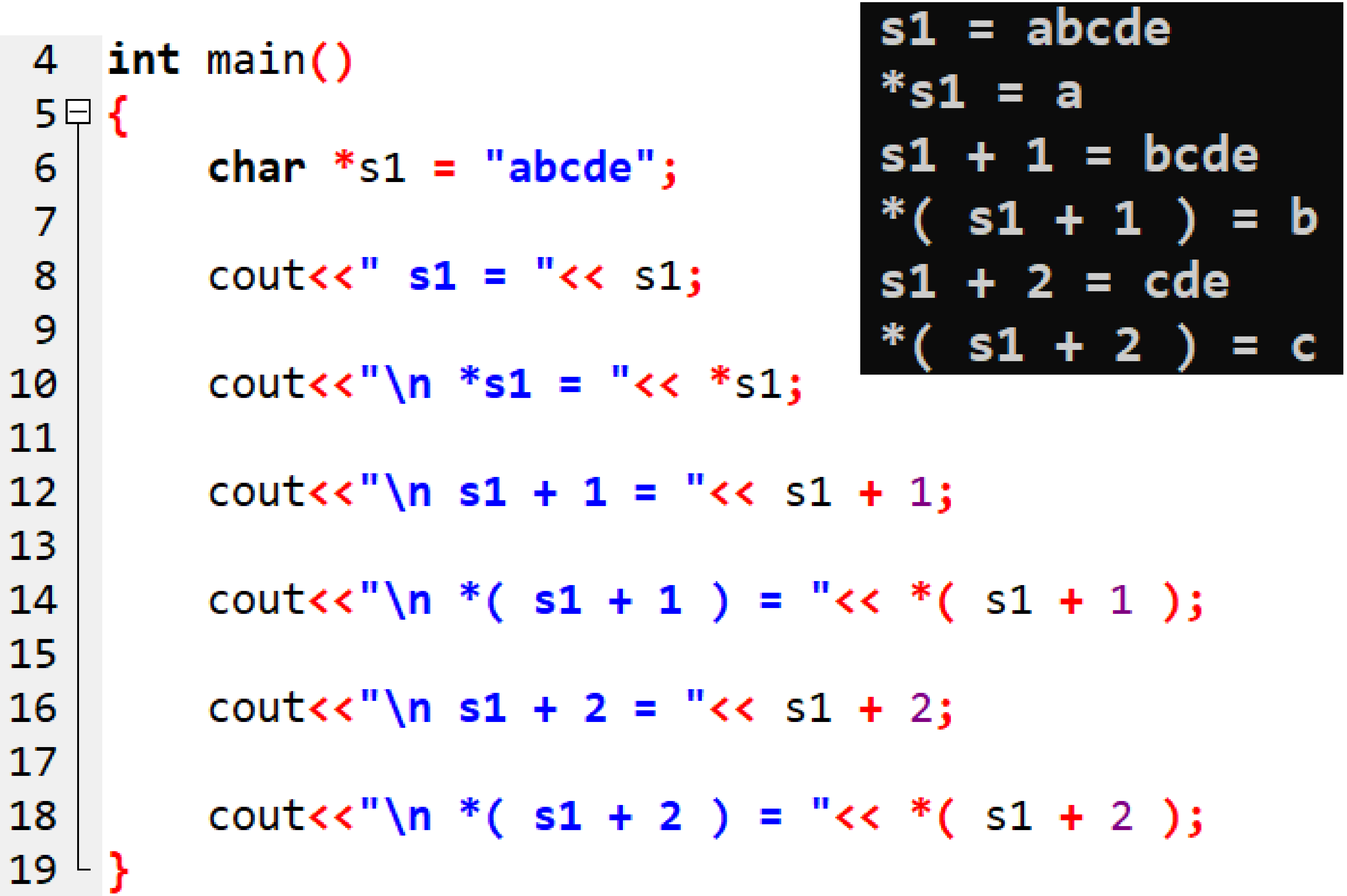
Strcat() function:

```
void  my_strcat ( char   *dest,   char *src )
```



```
void my_strcat(char* s2, char* s1)
{
    while( *s2 !='\0' )
        s2++;
    while( *s1 !='\0' )
        *s2++ = *s1++;
    *s2 = '\0';
}
```

Question



```
int main()
{
	char *s1 = "abcde";

	cout<<" s1 = "<< s1;

	cout<<"\n *s1 = "<< *s1;

	cout<<"\n s1 + 1 = "<< s1 + 1;

	cout<<"\n *( s1 + 1 ) = "<< *( s1 + 1 );

	cout<<"\n s1 + 2 = "<< s1 + 2;

	cout<<"\n *( s1 + 2 ) = "<< *( s1 + 2 );
}
```

OUTPUT:

```
s1 = abcde
*s1 = a
s1 + 1 = bcde
*( s1 + 1 ) = b
s1 + 2 = cde
*( s1 + 2 ) = c
```

Handling Table of Strings

char name[3][25];

This says that the name is a table containing 3 names, each with a maximum length of 25 characters (including null character)

- The **total storage** requirements for the name table are **75 bytes**
- We know that rarely the individual strings will be of equal lengths.
- Therefore, instead of making each row a fixed number of characters, we can make it a pointer to a string of varying length.

Array of strings

```
char *name[3] = { “New Zealand”, “Australia”, India”};
```

Declares name to be an array of 3 pointers to characters, each pointer pointing to a particular name as shown below.

name[0] → New Zealand

name[1]→ Australia

name[2]→ India

The following statement would print out all the 3 names.

```
for( i = 0 ; i < 3 ; i++ )
    cout << name[ i ];
```

Dynamic memory allocation

- **new** and **delete**
  - operators used to allocate and free memory at run time
- The **new** operator allocates memory and returns a pointer to the start of it
- **delete** operator frees memory previously allocated using **new**

Dynamic Memory Management : new and delete:

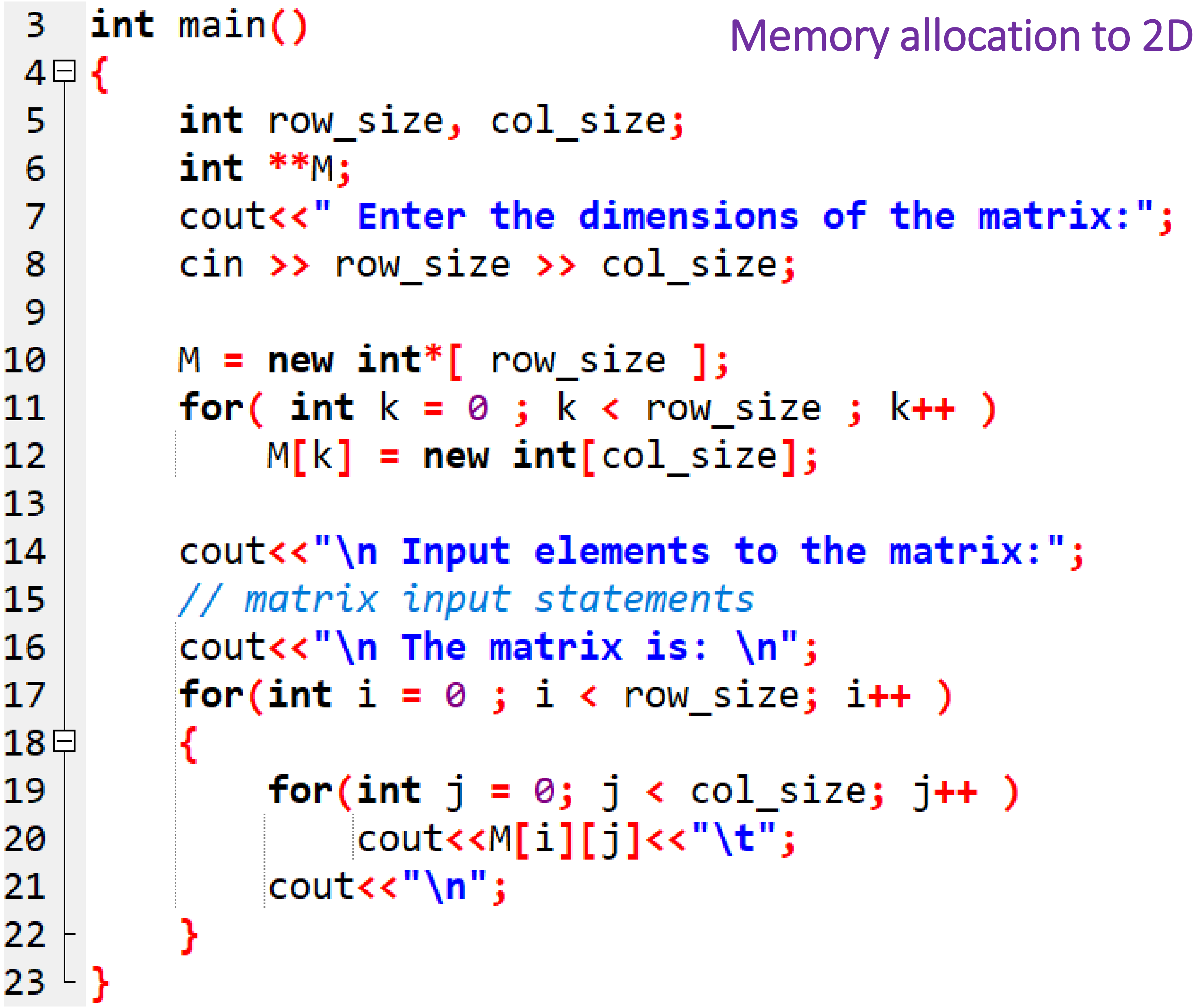
Syntax:

```
Pointer_variable =  new  data_type;
Pointer_variable =  new  data_type(value);
Pointer_variable =  new  data_type[size];
```

Eg:

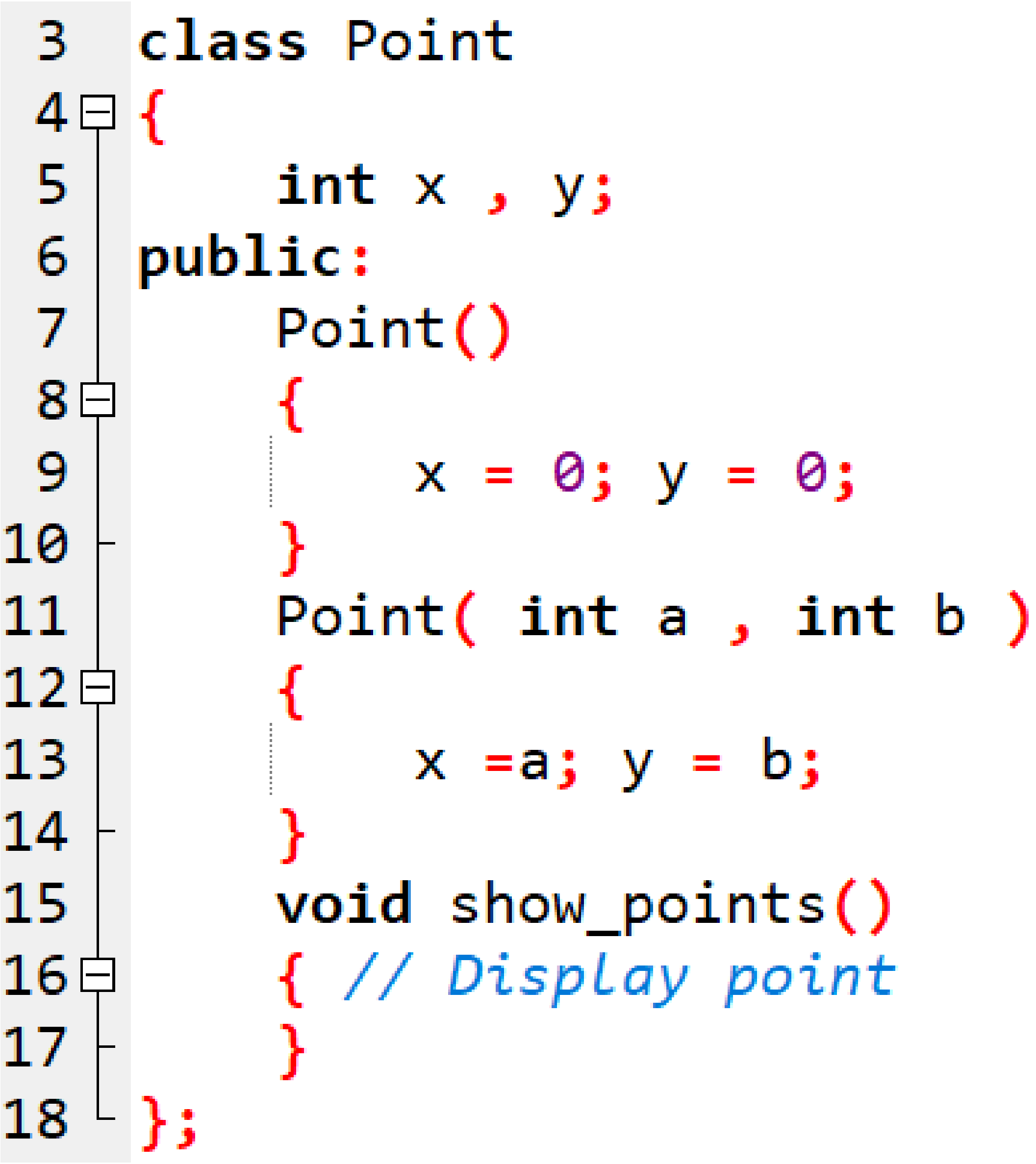
```
int   *p1   =  new  int;
int   *p2   =  new  int(5);
int   *p_arr   =  new  int[10];
```

Memory allocation to 2D array

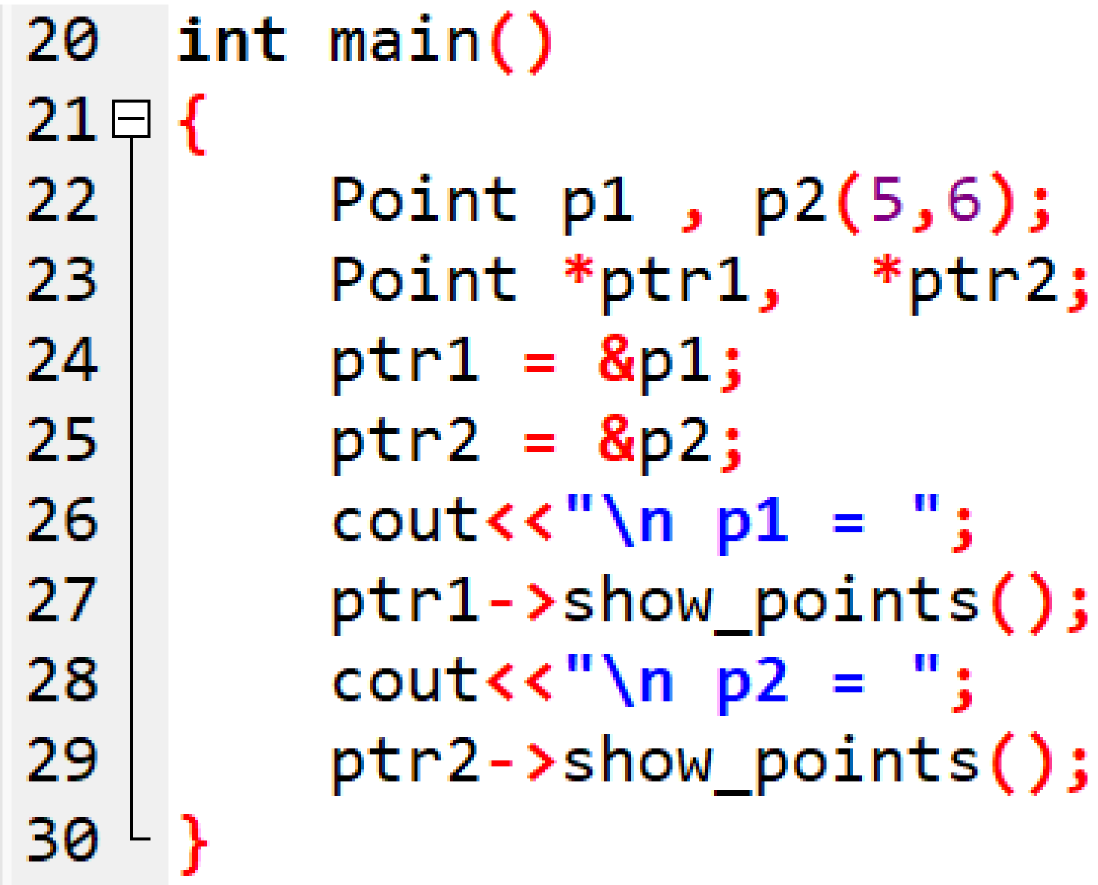


```cpp
int main()
{
    int row_size, col_size;
    int **M;
    cout<<" Enter the dimensions of the matrix:";
    cin >> row_size >> col_size;

    M = new int*[ row_size ];
    for( int k = 0 ; k < row_size ; k++ )
        M[k] = new int[col_size];

    cout<<"\n Input elements to the matrix:";
    // matrix input statements
    cout<<"\n The matrix is: \n";
    for(int i = 0 ; i < row_size; i++ )
    {
        for(int j = 0; j < col_size; j++ )
            cout<<M[i][j]<<"\t";
        cout<<"\n";
    }
}
```

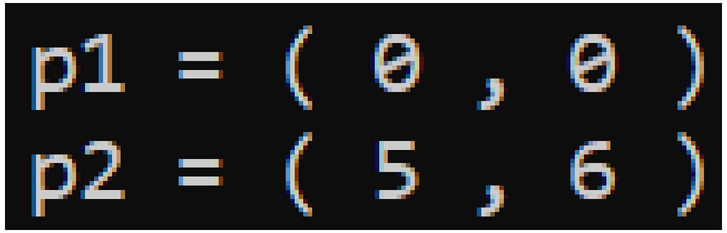
Pointer to object: Example-1



```cpp
class Point
{
    int x , y;
public:
    Point()
    {
        x = 0; y = 0;
    }
    Point( int a , int b )
    {
        x =a; y = b;
    }
    void show_points()
    { // Display point
    }
};
```

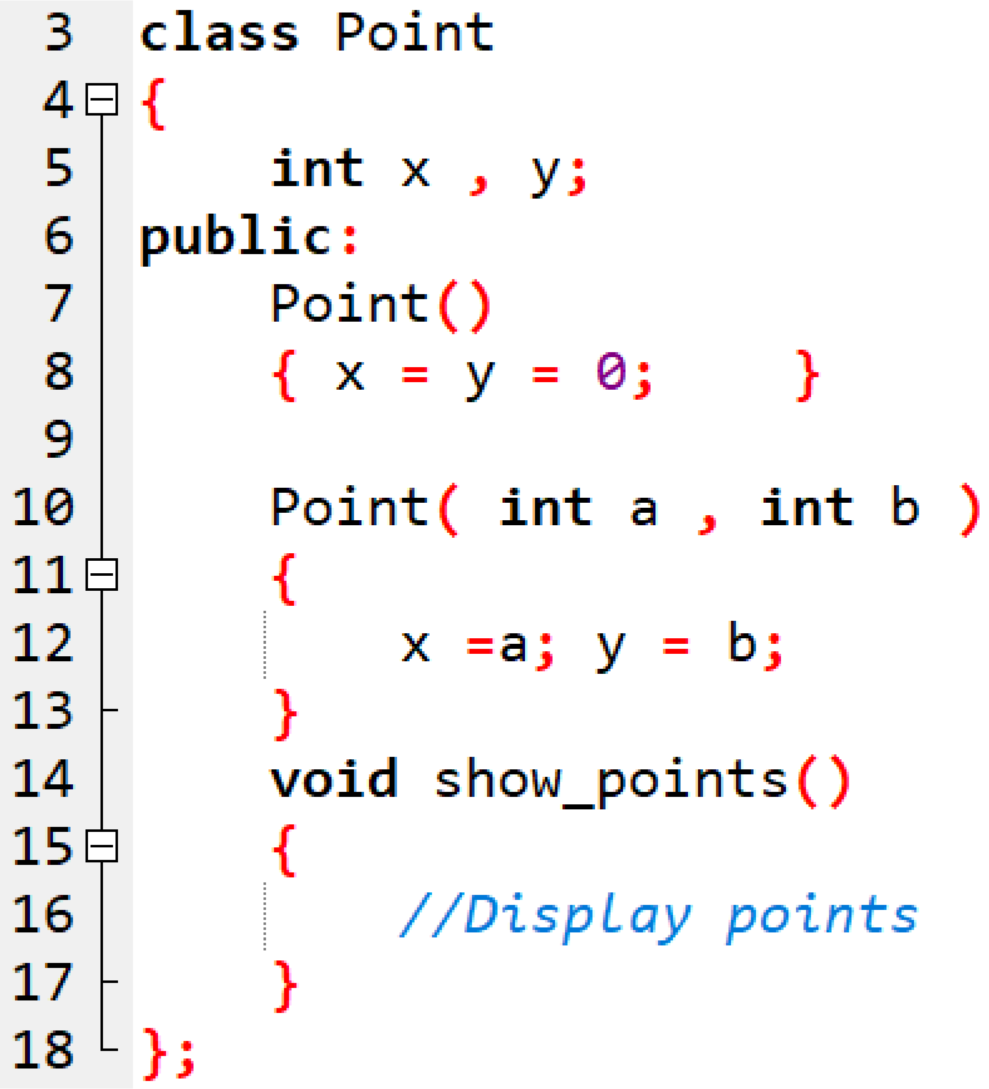


```cpp
int main()
{
    Point p1 , p2(5,6);
    Point *ptr1,  *ptr2;
    ptr1 = &p1;
    ptr2 = &p2;
    cout<<"\n p1 = ";
    ptr1->show_points();
    cout<<"\n p2 = ";
    ptr2->show_points();
}
```

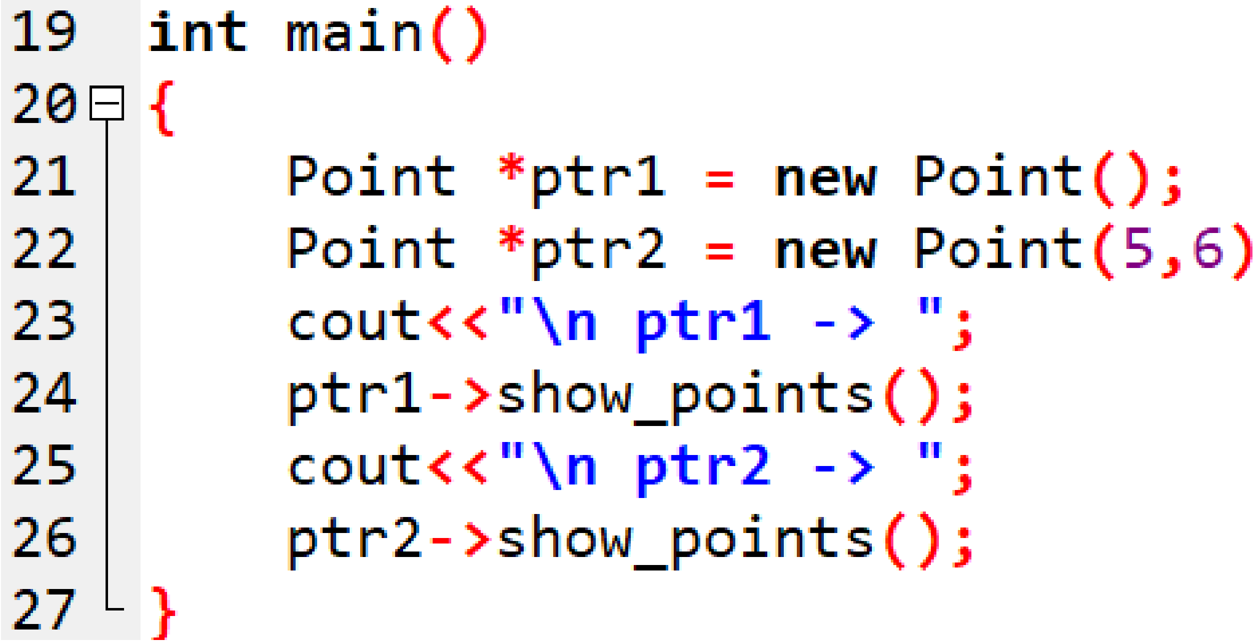


```
p1 = ( 0 , 0 )
p2 = ( 5 , 6 )
```

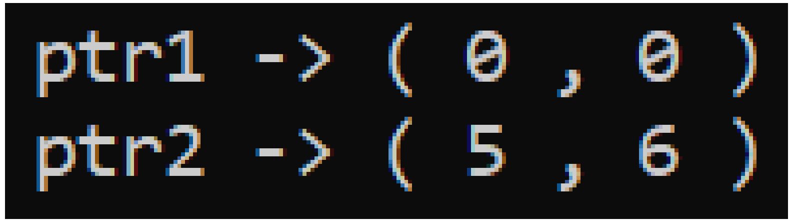
Pointer to object: Example-2



```cpp
class Point
{
    int x , y;
public:
    Point()
    { x = y = 0;     }

    Point( int a , int b )
    {
        x =a; y = b;
    }
    void show_points()
    {
        //Display points
    }
};
```

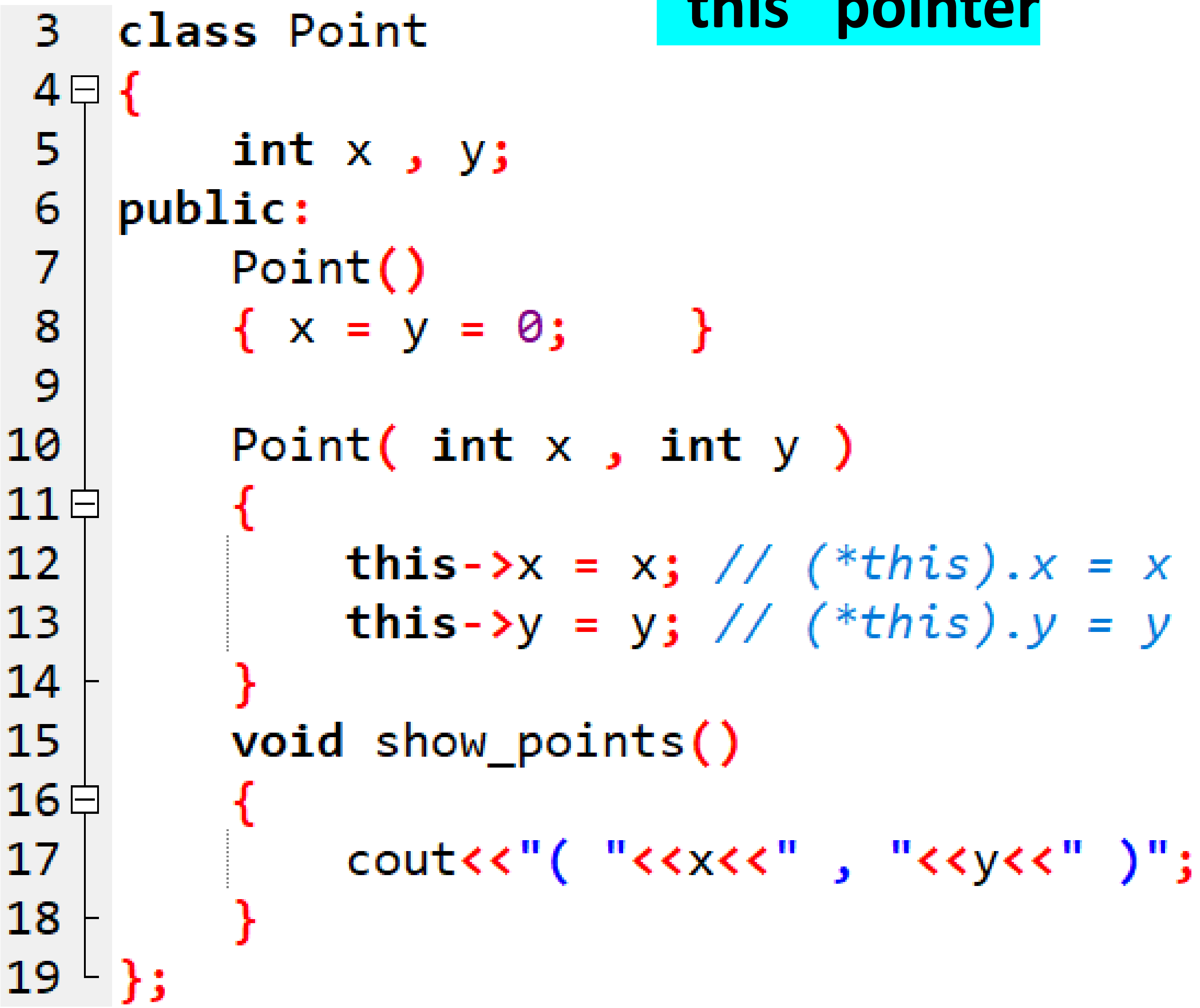


```cpp
int main()
{
    Point *ptr1 = new Point();
    Point *ptr2 = new Point(5,6)
    cout<<"\n ptr1 -> ";
    ptr1->show_points();
    cout<<"\n ptr2 -> ";
    ptr2->show_points();
}
```

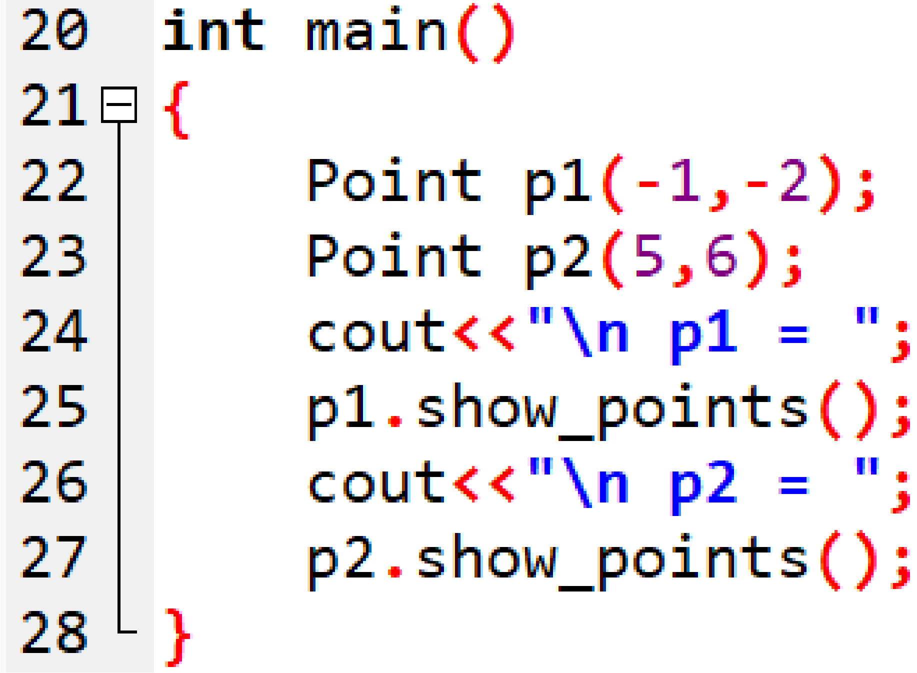


```
ptr1 -> ( 0 , 0 )
ptr2 -> ( 5 , 6 )
```

“this” pointer



```cpp
class Point
{
    int x , y;
public:
    Point()
    { x = y = 0;     }

    Point( int x , int y )
    {
        this->x = x; // (*this).x = x
        this->y = y; // (*this).y = y
    }
    void show_points()
    {
        cout<<"( "<<x<<" , "<<y<<" )";
    }
};
```



```cpp
int main()
{
    Point p1(-1,-2);
    Point p2(5,6);
    cout<<"\n p1 = ";
    p1.show_points();
    cout<<"\n p2 = ";
    p2.show_points();
}
```

Destructors

- **`delete`** calls the object's **destructor**.
- **`delete`** frees space occupied by the object.

- A **destructor** cleans up after the object.
- Releases resources such as memory.

Syntax: delete pointer_variable;

delete []array_pointer;

Malloc	New
Standard C Function	Operator
Used sparingly in C++; used frequently in C	Only in C++
Does not invoke any constructor	Invokes constructor of the class for object initialization.
Returns void* and requires explicit casting	Returns the proper type
Returns NULL when there is not enough memory	Throws an exception when there is not enough memory
Every malloc() should be matched with a free()	Every new/new[] should be matched with a delete/delete[]

Pointers and structures

Consider the following structure

```
struct inventory
{
   char  name[30];
   int  number;
   float  price;
} product[2], *p;
```

p=product; assigns the address of the zeroth element of **product** to **p**

or **p** points to **product[0];**

Pointers and structures (Contd…)

Its members are accessed using the following notation

p->name

p->number

p->price

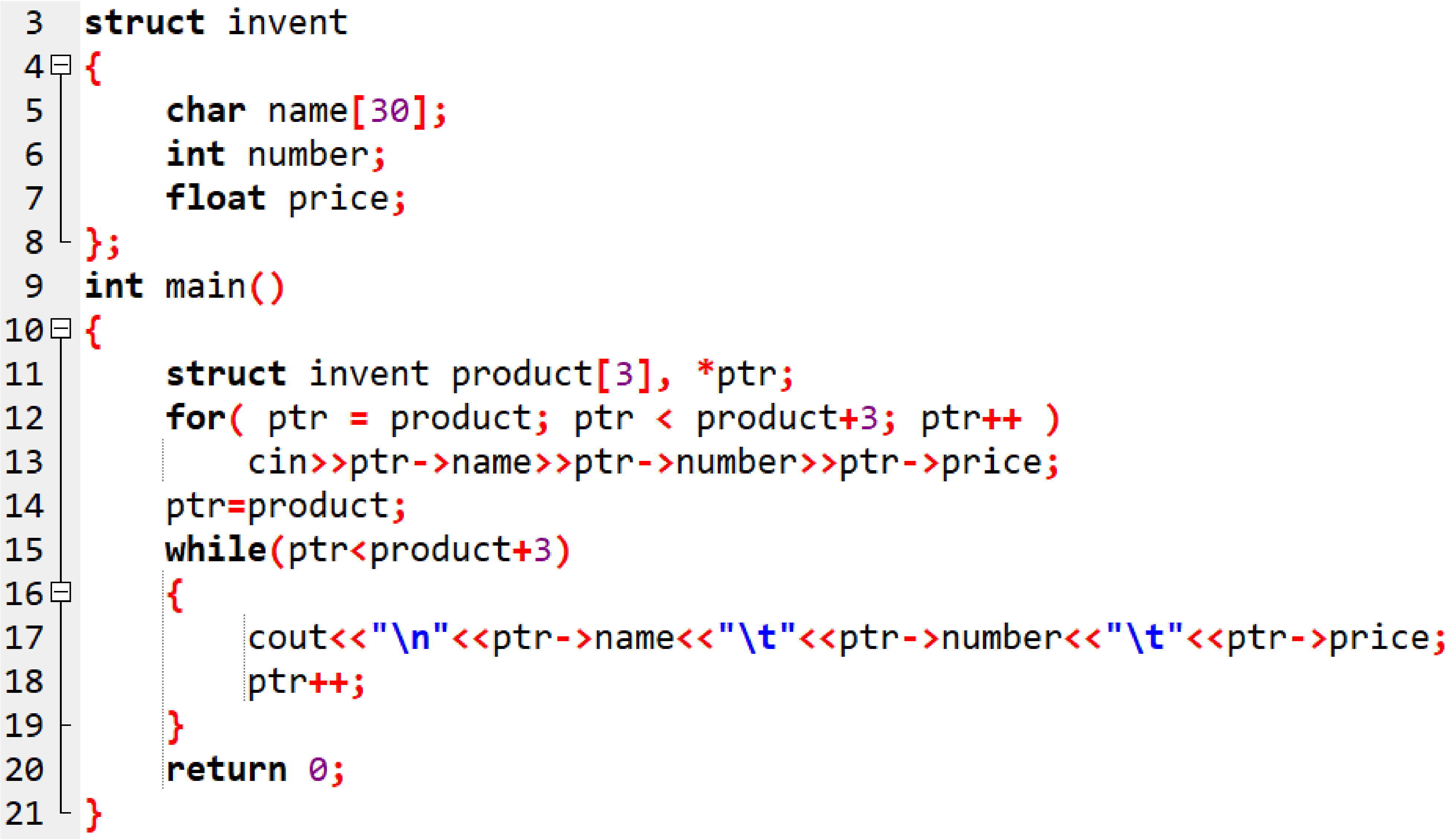
The symbol **->** is called **arrow operator** (also known as **member selection operator**)

The member number can also be accessed using

(*p).number

Parantheses is required because **‘.’** has higher precedence than the operator *

Program to illustrate the use of structure pointers



```cpp
struct invent
{
    char name[30];
    int number;
    float price;
};
int main()
{
    struct invent product[3], *ptr;
    for( ptr = product; ptr < product+3; ptr++ )
        cin>>ptr->name>>ptr->number>>ptr->price;
    ptr=product;
    while(ptr<product+3)
    {
        cout<<"\n"<<ptr->name<<"\t"<<ptr->number<<"\t"<<ptr->price;
        ptr++;
    }
    return 0;
}
```