```
#include<iostream>
#include<stdlib.h>
#include<omp.h>
using namespace std;


void mergesort(int a[],int i,int j);
void merge(int a[],int i1,int j1,int i2,int j2);

void mergesort(int a[],int i,int j)
{
        int mid;
        if(i<j)
        {
        mid=(i+j)/2;

        #pragma omp parallel sections
        {

        #pragma omp section
        {
                mergesort(a,i,mid);
        }

        #pragma omp section
        {
                mergesort(a,mid+1,j);
        }
        }

        merge(a,i,mid,mid+1,j);
        }

}

void merge(int a[],int i1,int j1,int i2,int j2)
{
        int temp[1000];
        int i,j,k;
        i=i1;
        j=i2;
        k=0;

        while(i<=j1 && j<=j2)
        {
        if(a[i]<a[j])
        {
        temp[k++]=a[i++];
        }
        else
        {
        temp[k++]=a[j++];
    }
        }

        while(i<=j1)
        {
        temp[k++]=a[i++];
        }

        while(j<=j2)
        {
        temp[k++]=a[j++];
        }

        for(i=i1,j=0;i<=j2;i++,j++)
        {
        a[i]=temp[j];
        }
}


int main()
{
        int *a,n,i;
        cout<<"\n enter total no of elements=>";
        cin>>n;
        a= new int[n];
```

```
	cout<<"\n enter elements=>";
	for(i=0;i<n;i++)
	{
	cin>>a[i];
	}
   //	 start=.......
//#pragma omp…..
	mergesort(a, 0, n-1);
//	  stop…….
	cout<<"\n sorted array is=>";
	for(i=0;i<n;i++)
	{
	cout<<"\n"<<a[i];
	}
	// Cout<<Stop-Start
	return 0;
}
```

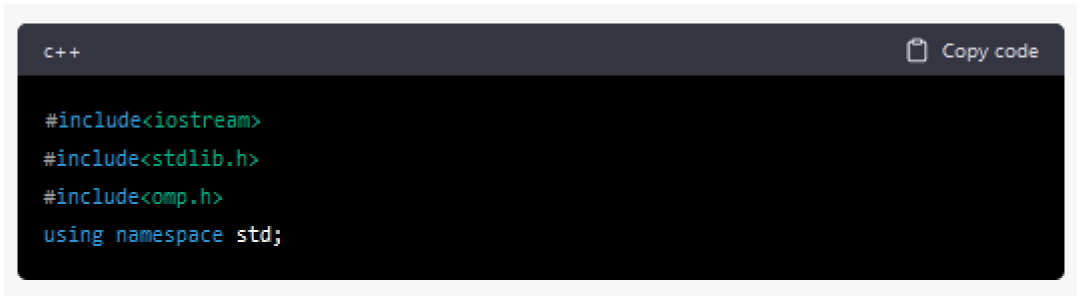
Second Code:

```
#include <iostream>
#include <omp.h>

void merge(int* arr, int l, int m, int r) {
	int i, j, k;
	int n1 = m - l + 1;
	int n2 = r - m;

	int L[n1], R[n2];

	for (i = 0; i < n1; i++)
	L[i] = arr[l + i];
	for (j = 0; j < n2; j++)
	R[j] = arr[m + 1 + j];

	i = 0;
	j = 0;
	k = l;
	while (i < n1 && j < n2) {
	if (L[i] <= R[j]) {
	arr[k] = L[i];
	i++;
	}
```

```
		else {
		arr[k] = R[j];
		j++;
		}
		k++;
		}

		while (i < n1) {
		arr[k] = L[i];
		i++;
		k++;
		}

		while (j < n2) {
		arr[k] = R[j];
		j++;
		k++;
		}
}

void mergeSort(int* arr, int l, int r) {
		if (l < r) {
		int m = l + (r - l) / 2;
		#pragma omp parallel sections
		{
		#pragma omp section
		{
			mergeSort(arr, l, m);
		}
		#pragma omp section
		{
			mergeSort(arr, m + 1, r);
		}
		}

		merge(arr, l, m, r);
		}
}

int main() {
		int arr[] = { 12, 11, 13, 5, 6, 7 };
		int n = sizeof(arr) / sizeof(arr[0]);
  double start, stop;
		std::cout << "Given array is: ";
		for (int i = 0; i < n; i++)
		std::cout << arr[i] << " ";
		std::cout << std::endl;
```

```
start = omp_get_wtime();
  #pragma omp parallel
  {
          mergeSort(arr, 0, n - 1);
}
  stop = omp_get_wtime();
          std::cout << "Sorted array is: ";
          for (int i = 0; i < n; i++)
          std::cout << arr[i] << " ";
          std::cout << std::endl;

std::cout<<stop-start;
          return 0;
}
```

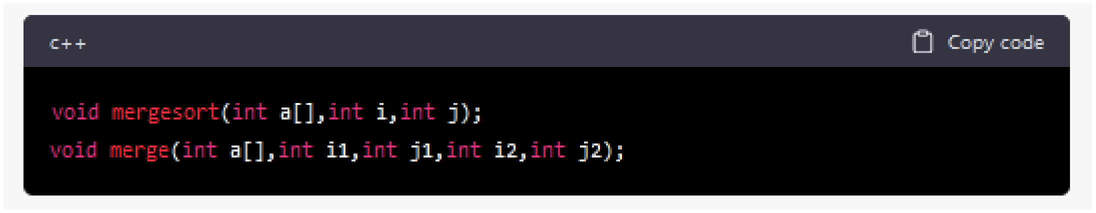
Explanation



c++ Copy code

```cpp
#include<iostream>
#include<stdlib.h>
#include<omp.h>
using namespace std;
```

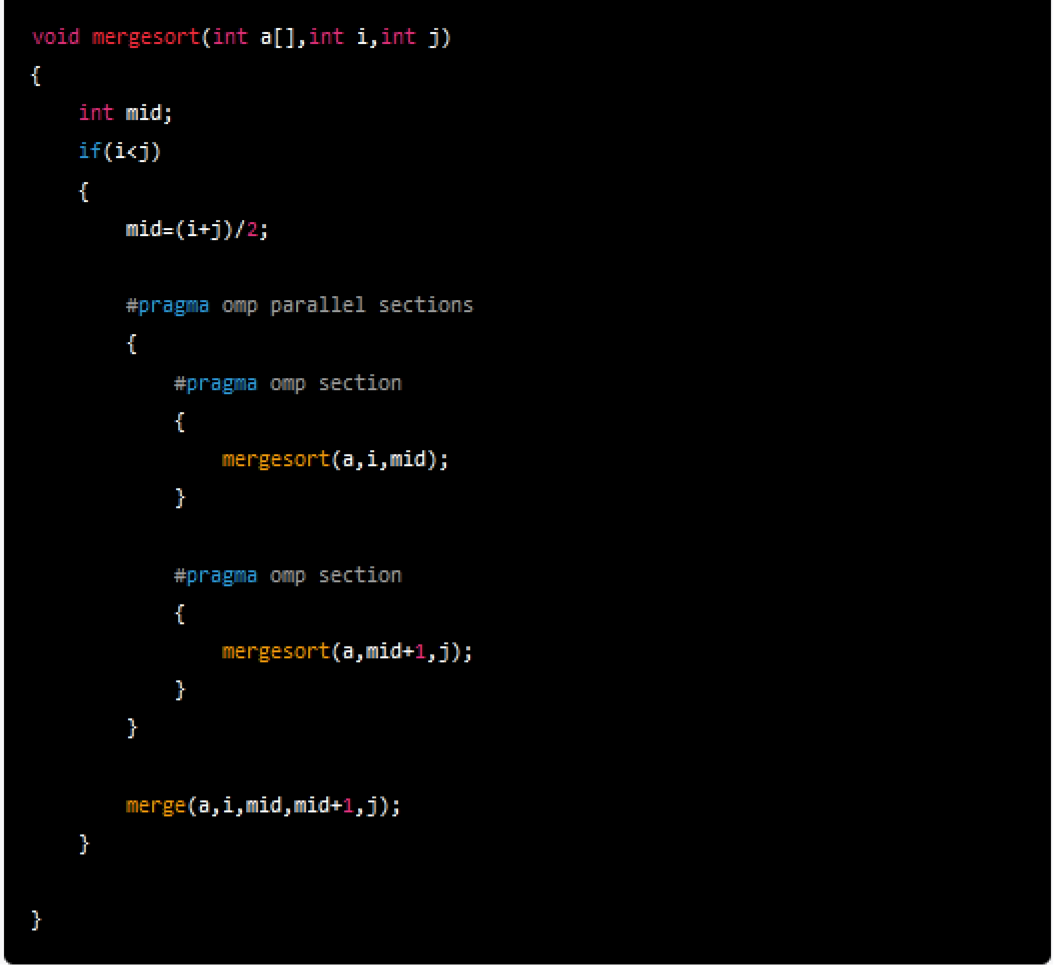
This block of code is a set of header files and a namespace declaration. iostream and stdlib.h are header files, and omp.h is the header file for OpenMP (Open Multi-Processing) library that is used for parallel programming.



c++ Copy code

```cpp
void mergesort(int a[],int i,int j);
void merge(int a[],int i1,int j1,int i2,int j2);
```

These are function declarations for mergesort and merge functions.



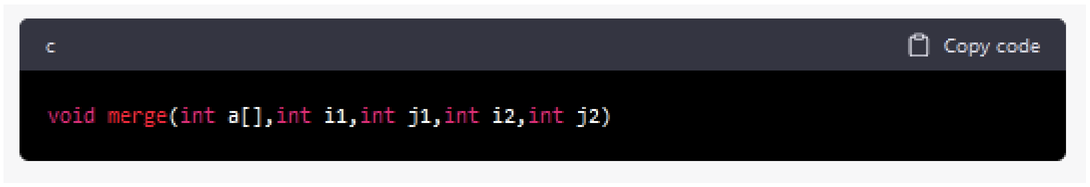
```c
void mergesort(int a[],int i,int j)
{
    int mid;
    if(i<j)
    {
        mid=(i+j)/2;

        #pragma omp parallel sections
        {
            #pragma omp section
            {
                mergesort(a,i,mid);
            }

            #pragma omp section
            {
                mergesort(a,mid+1,j);
            }
        }

        merge(a,i,mid,mid+1,j);
    }

}
```

This is the mergesort function, which implements the merge sort algorithm recursively. a[] is the input array that is to be sorted, i and j are the starting and ending indices of the subarray being sorted.

First, it checks whether i is less than j because if i is greater than or equal to j, it means that there is only one element in the subarray, which is already sorted. Then it calculates the midpoint of the subarray using (i+j)/2.

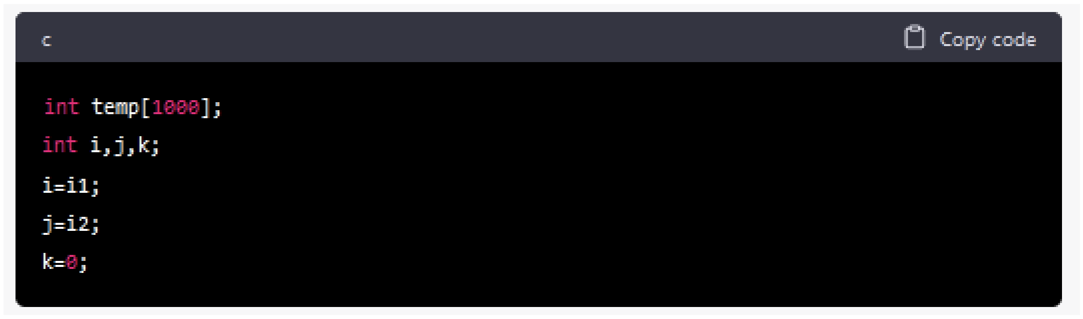
The #pragma omp parallel sections directive creates a parallel region that specifies two parallel sections that can be executed concurrently. The mergesort function is then recursively called on the left and right halves of the subarray in two separate parallel sections.

Finally, the merge function is called to merge the sorted halves of the subarray.



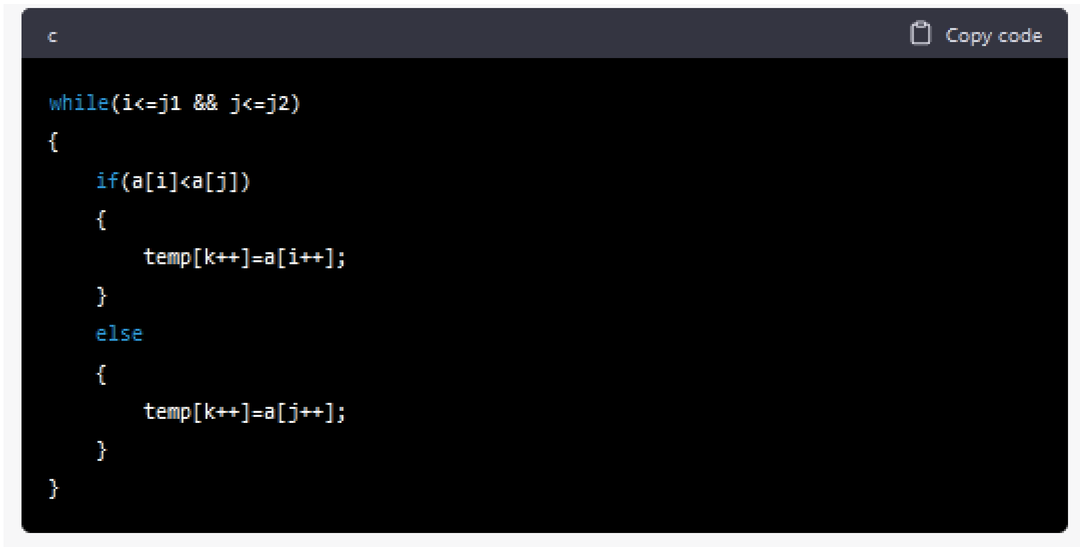
```c
void merge(int a[],int i1,int j1,int i2,int j2)
```

This is the function signature which takes an integer array a and four integer variables i1, j1, i2, and j2. i1 and j1 define the start and end indices of the first sorted subarray, and i2 and j2 define the start and end indices of the second sorted subarray.



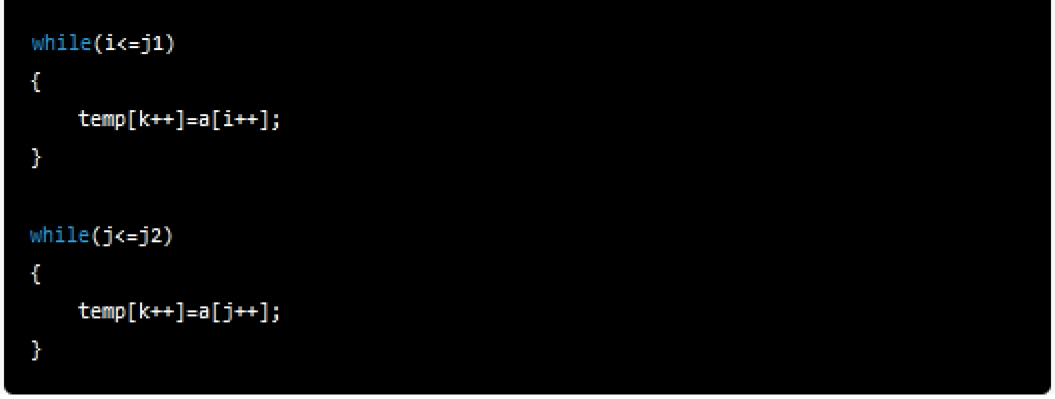
```c
int temp[1000];
int i,j,k;
i=i1;
j=i2;
k=0;
```

Here, a temporary array temp is created with a size of 1000. Three integer variables i, j, and k are initialized. i and j are set to the start indices of the two subarrays, while k is set to 0.



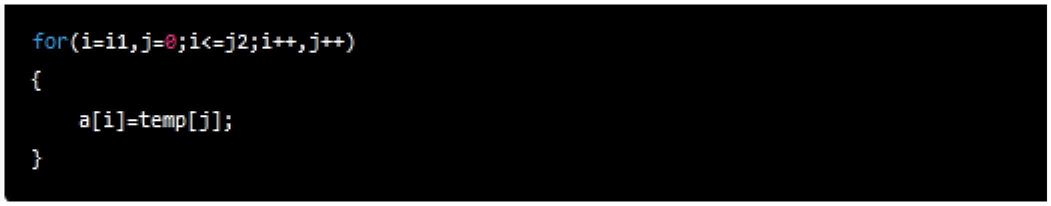
```c
while(i<=j1 && j<=j2)
{
    if(a[i]<a[j])
    {
        temp[k++]=a[i++];
    }
    else
    {
        temp[k++]=a[j++];
    }
}
```

This is a while loop that runs as long as i is less than or equal to j1 and j is less than or equal to j2. Inside the loop, if the element at index i of the first subarray is less than the element at index j of the second subarray, then the element at index i is copied to the temp array at index k, and i and k are incremented. Otherwise, the element at index j is copied to the temp array at index k, and j and k are incremented.



```
while(i<=j1)
{
    temp[k++]=a[i++];
}

while(j<=j2)
{
    temp[k++]=a[j++];
}
```

After the above loop terminates, there may be some elements left in one of the subarrays. These loops copy the remaining elements into the temp array.



```
for(i=i1,j=0;i<=j2;i++,j++)
{
    a[i]=temp[j];
}
```

Finally, the sorted temp array is copied back to the original a array. The loop runs from i1 to j2 and copies the elements of temp array to the corresponding indices in the a array. The loop variable j starts from 0 and increments alongside i.