```
#include<iostream>
#include<vector>
// FinalExam.cpp : This file contains the 'main' function. Program execution begins and ends
there.
//
void Q1()
{
    //std::cout << The saint said, "God is one.";
    std::cout << "The saint said, \"God is one\".";
    //std::cout << \"The saint said, "God is one".\";
    //std::cout << "The saint said, \\"God is one\\".";

}
void Q2()
{
    int a = 1, b = 2, c = 3;
    std::cout << (++a) - (b--) + (c++);
}
void Q3()
{
    int a = 5, b = 4, c;
    a = (a > 3) + (b <= 5);
    b = (a == 3) + ((b - 2) >= 3);
    c = (b != 1);
    std::cout << a << ' ' << b << ' ' << c;

}
void Q4()
{
    std::cout << ~(~0 << 5);
}
void Q5()
{
    unsigned int a = 0xb9;
    a = a ^ 0x1 << 4;
    std::cout << std::hex << a;
}
void Q6()
{
    union m { char a; int b; int c; };
    std::cout << sizeof(m);

}
```

```
void Q7()
{
    typedef struct { int a; int b; } M;
    std::cout << sizeof(M);
}
void Q8()
{
    typedef struct { int a; } S;
    S z, * pz{ &z };
    z.a = 10;
    pz->a = z.a * 2;
    std::cout << z.a << ' ' << pz->a;

}
void Q9()
{
    typedef struct { int a; long sq; } SQ;
    int i, j;
    SQ squares[4], * sp;
    for (i = 10, j = 0; i <= 40; i += 10, j++) {
        squares[j].a = i; squares[j].sq = i * i;
    }
    for (j = 0, sp = squares; j < 4; ++j, sp++) {
        std::cout << sp->a << " squared is " << sp->sq << std::endl;
    }

}
#pragma region Q10_13


int* GetArrayOfNumbers(int count) {

    int* p = new int[count];

    memset(p, 0, count * sizeof(int));

    return p;
}
void DisplayNumbers(const int* p, unsigned int count)
{
    while (count-- > 0) std::cout << *p++ << ' ';
}
void Q10_13() {
```

```
    const int count{ 40 };
    int* nums = GetArrayOfNumbers(count);

    for (int i{ 0 }; i < count; i++)
        nums[i] = rand() % count;

    DisplayNumbers(nums, count);
    delete[] nums;

    // potentially more code here
}
#pragma endregion

#pragma region Q14_19
// define Square object structure
struct Square {
    float side;
    float perimeter;
    float area;
};

Square** CreateSquares(const float* side, const int& count)
{
    // Pointer to array of pointers to Square objects.
    Square** pSquarePtrs = new Square*[count];
    Square* pSquare{ nullptr }; //pointer to individual Square object

    // create Square object for each of the side
    for (int row = 0; row < count; row++) {
        // allocate memory on heap to create a Square object
        pSquare = new Square;
        // fill with data
        pSquare->side = side[row];
        pSquare->area = pSquare->side * pSquare->side;
        pSquare->perimeter = 4 * pSquare->side;
        // store the Square object pointer in pSquaresPtrs
        pSquarePtrs[row] = pSquare;
    }

    return pSquarePtrs;
}
void SortSquaresBySide(Square** pSquarePtrs, const int& count)
{
```

```
    for (int i = 0; i < count; i++) {
        for (int j = i + 1; j < count; j++) {
            if (pSquarePtrs[i]->side > pSquarePtrs[j]->side ) {
                //swap
                Square* temp = pSquarePtrs[i];
                pSquarePtrs[i] = pSquarePtrs[j];
                pSquarePtrs[j] = temp;
            }
        }
    }
}
void DisplaySquaresData(Square** pSquarePtrs, const int& count)
{
    Square* pSquare = NULL;
    for (int row = 0; row < count; row++) {
        // let pSquare point to individual Square object present
        pSquare = pSquarePtrs[row];
        printf("Side=%.1f, Perimeter=%.2f, Area=%.2f\n",
            pSquare->side, pSquare->perimeter, pSquare->area);
    }
}
void UseSquares()
{
    float side[] = { 4.0,5.7,2.4,3.5 };
    Square** pSquarePtrs = CreateSquares(side, 4);
    SortSquaresBySide(pSquarePtrs, 4);
    DisplaySquaresData(pSquarePtrs, 4);

    // free all memory allocated on heap
    for (int i{ 0 }; i < 4; i++)
    {
        delete pSquarePtrs[i];
    }
    delete[] pSquarePtrs;
}
#pragma endregion

#pragma region Q20_Q28
template<typename T>
struct Stack
{
    Stack(size_t expectedSize)
    {
```

```
        tos = -1;
        stack = new T[expectedSize];
    }
    void Push(T ele)
    {
        stack[++tos] = ele;
    }
    T Pop()
    {
        return stack[tos--];
    }

private:
    T* stack;
    int tos;
};
struct Player
{
    int gameId;
    int age;
    int pay;
};
void UseStack()
{
    // create stack of integers
    Stack<int> stackInts(6);
    stackInts.Push(5);
    stackInts.Push(3);
    std::cout << stackInts.Pop() << std::endl;

    // create stack of Player objects
    Stack<Player> stackPlayer(11);
    stackPlayer.Push(Player{ 10,26,60000 });
    stackPlayer.Push(Player{ 10,27,50000 });
    stackPlayer.Push(Player{ 10,24,20000 });
    auto p = stackPlayer.Pop();
    std::cout << p.gameId << ' ' << p.age << ' ' << p.pay << std::endl;
}
template<typename T>
void Sort(T* items, size_t count, bool (*comparer)(T left, T right))
{
    for (int i{ 0 }; i < count; i++)
    {
```

```
        for (int j{ i }; j < count; j++)
        {
            if (comparer(items[i], items[j]) > 0)
            {
                // swap
                auto temp = items[i];
                items[i] = items[j];
                items[j] = temp;
            }
        }
    }
}
bool ComparePlayerByPay(Player p1, Player p2)
{
    return p1.pay > p2.pay ? true : false;
}
void UseSort()
{
    const int count{ 5 };
    Player players[count]{
        {10, 24, 20000},
        {10, 25, 60000},
        {10, 22, 30000},
        {10, 24, 50000},
        {10, 27, 80000},
    };
    Sort(players, count, ComparePlayerByPay);
    for (int i{ 0 }; i < count; i++)
    {
        std::cout << players[i].pay << ' ' << players[i].age << std::endl;
    }
}
#pragma endregion

#pragma region Q29_Q40

struct Student{
    Student(int a, int i) { age = a; id = i; }
    int age;
    int id;
};

static bool operator== (Student& s1, Student& s2)
```

```
{
   // check equality of two student objects by Id
   return s1.id == s2.id ? true : false;
}

template<typename T>
struct LLNode    //Node of Doubly Linked List
{
   std::unique_ptr<T> data; // each node owns its data object provided by the user of DLL
   std::shared_ptr<LLNode<T>> next; // pointer to the next node
   std::weak_ptr<LLNode<T>> prev; // pointer to the previous node, avoiding cyclic reference
};

template<typename T>
class DLL // doubly linked list
{
public:
   DLL(bool (*callback) (T&, T&))
   {
      // need a callback to ask user of DLL to check for equality
      comparer = callback;
   }

   void Add(std::unique_ptr<T> data) // add a node to linked list with data
   {
      std::shared_ptr<LLNode<T>> newNode = CreateNode(std::move(data));

      if (startLL == nullptr)
      {
         startLL = newNode;
      }
      else
      {
         newNode->next = startLL;
         startLL->prev = newNode;
         startLL = newNode;
      }
   }

   std::unique_ptr<T> Extract(T id) // search for a node, remove from list and return it
   {
      auto p{ startLL };
      std::unique_ptr<T> extract;
```

```
        while (p != nullptr && p->next != nullptr)
        {
            T* obj1 = p->data.get();

            if (*obj1 == id) // use comparer callback
            {
                // got a match, let's remove it from list and return it
                extract = std::move(p->data); // hold a pointer to data of the LL node
                if (p == startLL)
                {
                    // there was only one node the list
                    startLL = nullptr;
                }
                else
                {
                    // remove the node and let the prev & next nodes be linked
                    auto prevNode = p->prev.lock();
                    prevNode->next = p->next;
                    p->next->prev = prevNode;
                }
                break; // we found the node, no need to iterate further
            }
            p = p->next; // go to next node
        }

        return extract;
    }

private:
    std::shared_ptr<LLNode<T>> CreateNode(std::unique_ptr<T> data)
    {
        // create a node
        std::shared_ptr<LLNode<T>> pNode = std::make_shared<LLNode<T>>();
        pNode->data = std::move(data); // get exclusive ownership of the data
        return pNode;
    }

private:
    std::shared_ptr<LLNode<T>> startLL; // pointer to start of the node
    bool (*comparer)(T&, T&);
};

void UseDLL()
```

```
{
    DLL<Student> doubleLL(operator==);

    // add few nodes
    auto s = std::make_unique<Student>(17, 20001020);
    doubleLL.Add(std::move(s));

    s = std::make_unique<Student>(19, 20001022);
    doubleLL.Add(std::move(s));

    s = std::make_unique<Student>(18, 20001023);
    doubleLL.Add(std::move(s));

    s = std::make_unique<Student>(19, 20001024);
    doubleLL.Add(std::move(s));

    Student search(0, 20001023); // set a search by Id
    // see if we get the data back from DLL
    s = doubleLL.Extract(search);

    std::cout << "Got back from doubly linked list: \n" << "Age: " << s->age << " Id: " << s->id <<
std::endl;
}
#pragma endregion

void main()
{
    UseDLL();
}
```