Q: are the addresses of 2D consecutive?

```cpp
int g[2][2];
cout << &g[0][0] << ' ' << &g[0][1] << '
' << &g[1][0] << ' ' << &g[1][1] << '
';
```

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>g[0][0]</td>
<td>g[0][1]</td>
</tr>
<tr>
<td>g[1][0]</td>
<td>g[1][1]</td>
</tr>
</tbody>
</table>
Outline

**Visual Studio**: start project, debug program

**Program Structure**: iostream, std, main, sequential, return, comment

**Integer Variable**: declare, assign, initialize, arithmetic, input/output, round, unsigned, constant.

**Array Variable**: declaration, index, initialization, multi-dimensional array, sparse matrix

**Instruction**: arithmetic/increment, relational/logical, bitwise, if-else, switch, while-loop, for-loop, break

**Other Variables**: float, char, string, enum, vector, structure

**Pointer Variable**: idea, declare, extract address, access content, allocation, relation to array

**Function**: definition, declaration, assignment passing, local/global variable, overload

**Class**: definition, public/private member, constructor/destructor, declaration, memory
Function Overload

We can define multiple functions of the same name but different input lists.

```cpp
int Sum(int a, int b) {
    return a + b;
}

int Sum(int a, int b, int c) {
    return a + b + c;
}

int main() {
    cout << Sum(3, 4);
    cout << Sum(3, 4, 6);
    return 0;
}
```

1st is a Sum function with two integer inputs.

2nd is a Sum function with three integer inputs.

It is legal for us to define two functions in this way.
We can define multiple functions of the same name but different input lists.

1st is a Sum function with two integer inputs.

2nd is a Sum function with three integer inputs.

It is legal for us to define two functions in this way.
Function Overload

When function name is called, the compiler will decide which function to run based on input.

```cpp
int Sum(int a, int b) {
    return a + b;
}

int Sum(int a, int b, int c) {
    return a + b + c;
}

int main()
{
    cout << Sum(3, 4);  // Call the 1st function as there are two inputs.
    cout << Sum(3, 4, 6);  // Call the 2nd function as there are three inputs.
    return 0;
}
```

1st is a Sum function with two integer inputs.
2nd is a Sum function with three integer inputs.
Why function overload?

Function overload is useful to compactly call similar functions.

```cpp
int Sum(int a, int b) {
    return a + b;
}

int Sum(int a, int b, int c) {
    return a + b + c;
}

int main()
{
    cout << Sum(3, 4);
    cout << Sum(3, 4, 6);
    return 0;
}
```

Both aim to add numbers, just of different sizes.
Another Example: Functions with different types of input.

```cpp
void Print(int x) {
    cout << "Integer is: " << x << endl;
}

void Print(string x) {
    cout << "String is: " << x << endl;
}

int main()
{
    Print(10);  // Input is integer. Call the 1st function.
    Print("Hello World!");  // Input is string. Call the 2nd function.
    return 0;
}
```

1st function takes an integer as input.
2nd function takes a string as input.
Outline

Visual Studio: start project, debug program

Program Structure: iostream, std, main, sequential, return, comment

Integer Variable: declare, assign, initialize, arithmetic, input/output, round, unsigned, constant.

Array Variable: declaration, index, initialization, multi-dimensional array, sparse matrix

Instruction: arithmetic/increment, relational/logical, bitwise, if-else, switch, while-loop, for-loop, break

Other Variables: float, char, string, enum, vector, structure

Pointer Variable: idea, declare, extract address, access content, allocation, relation to array

Function: definition, declaration, assignment passing, local/global variable, overload

Class: definition, public/private member, constructor/destructor, declaration, memory
Class is a feature of C++.

Used to describe property and behavior of similar objects, e.g.,
- a course class describes courses
- a student class describes students
- a university class describes universities

Class has member variables and member functions, e.g.,
- a variable of course to store its capacity
- a variable of student to store his/her major
- a function of university to update its enrollment
Define a class and its member variables.

Define a class called Course.

```cpp
class Course {
    public:
        string title;
    private:
        int capacity;
};
```

syntax: class name

Descriptions of the class in a pair of braces.
Define a class and its member variables.

Inside the class definition, we can declare member variables to store properties of the class.

```cpp
class Course {
    public:
        string title;
    private:
        int capacity;
};
```

This member is called “title”. We will use it to hold title of a course.

This member is called “capacity”. We will use it to hold capacity of a course.
Define a class and its member variables.

Among the member variables, we can declare some as public and the others as private.

Public members can be directly accessed outside the class.

But private members can not.

* We will see examples shortly.

```cpp
class Course {

public:
    string title;

private:
    int capacity;

};
```

All following “public:” are public members.

All following “private:” are private members.

“title” is public, “capacity” is private.
Declare an object of class.

After defining a class, we need to declare an object of it and play with that object.

a class ~= a variable type.
an object ~= a variable.

```c
int main()
{
    Course x;  // Declare an object of the Course class.
    x.title = "Data Structure";
    x.capacity = 100;
    return 0;
}
```

This object is called “x”.

```java
Course course = new Course();
course.title = "Data Structure";
course.capacity = 100;
```
Access object members using dot.

Syntactically, we access members of an object by “." syntax: object.member.

Object x
public:
    string title;
private:
    int capacity;

* We can play with member variables as if they are regular variables.
Illegal to directly access private member!

```cpp
int main()
{
    Course x;
    x.title = "Data Structure";
    return 0;
}
```

Object x
public:
    string title;
private:
    int capacity;

This line is correct!!
```cpp
x.title = "Data Structure";
```

This line is wrong!!
```cpp
x.capacity = 100;  // Incorrect
```

However, we cannot directly access “capacity” as it is a private member.
Member functions of a class.

Inside a class, we can declare member functions.

Functions are often public, because we want to call them from outside the class.

These functions are often used as interface to access private member variables.

```cpp
class Course {
public:
    string title;
    void setCap(int n);
private:
    int capacity;
};
```

Declare a function `setCap`, which will be used to assign data to private “capacity”.

This is just a declaration, and we still need definition.
Define member function outside the class.

We can certainly define the function inside a class.

More often we define member function outside the class.

```cpp
class Course {
public:
    string title;
    void setCap(int n);
private:
    int capacity;
};
```

When defining member function outside the class, we just need to additionally clarify which class does this function belong to.

Add prefix “class_name::” before the function name.

```cpp
void Course::setCap(int n) {
    capacity = n;
}
```

setCap belongs to class Course.
Member function can directly access private member variables!

```cpp
class Course {
    public:
        string title;
        void setCap(int n);
    private:
        int capacity;
};

void Course::setCap(int n) {
    capacity = n;
}
```

**Important**
A member function can directly access private member variables, or any other member variables.

Directly assign “capacity”.

No need of declaration or any additional statements.
Use member function to assign private member variable in main.

```cpp
class Course {
public:
    string title;
    void setCap(int n);
private:
    int capacity;
};

void Course::setCap(int n) {
    capacity = n;
}

int main()
{
    Course x;
    x.setCap(100);
    return 0;
}
```

We also access a member function by “dot”.

Call public function “setCap”.

Assign 100 to capacity through setCap(), but not directly!
Define another member function for printing capacity.

```cpp
class Course {
    public:
        string title;
        void setCap(int n);
        void printCap();
    private:
        int capacity;
};

void Course::printCap()
{
    cout << capacity;
}
```
Define another member function for printing capacity.

```cpp
int main()
{
    Course x;
    x.setCap(100);
    x.printCap();
    return 0;
}
```

Set capacity to 100.

Print capacity.

Print 100 on screen.
A constructor is a special function which is auto-called when an object is declared.

It is used to initialize member variables of the object.

Syntax: `class_name();`

1. Same name as class.
2. No function type.
3. Can have function input.
Example definition of a constructor.

```cpp
class Course {
public:
    string title;
    Course();
    void setCap(int n);
    void printCap();
private:
    int capacity;
};

Course::Course() {
    capacity = 80;
    title = "Data Structure";
}
```
Auto-calling of constructor when an object is declared in main.

```java
int main()
{
    Course x;
    x.printCap();
    x.setCap(100);
    x.printCap();
    return 0;
}
```

When `x` is declared, constructor is auto-called and member variables are initialized as follows.

- `title = "Data Structure"
- `capacity = 80`

Print 80 on screen.
Reset capacity to 100.
Print 100 on screen.
We can also define a constructor with inputs and use them to initialize member variables.

```cpp
class Course {
public:
    string title;

    Course(int a, string b); // Constructor with inputs

    void setCap(int n);
    void printCap();

private:
    int capacity;
};

Course::Course(int a, string b) {
    capacity = a;
    title = b;
}
```

Constructor with inputs.
Constructor with inputs in main.

When declaring an object, we need to include input.
syntax: class object(input)

```java
int main()
{
    Course x(100, "Data Structure");
    Course y(40, "Program Structure");
    x.printCap();                      -> Print 100.
    y.printCap();                      -> Print 40.
    y.setCap(60);                      -> y.capacity = 60.
    return 0;
}
```

Object x with capacity 100, title "Data Structure"
Object y with capacity 40, title "Prog Structure"
Destructor of a class.

```cpp
class Course {
public:
    Course();
    ~Course();

private:
    int *p;
    int capacity;
};

Course::~Course() {
    delete[] p;
}
```

A destructor is used to delete the object when it is out of scope.

There is a default destructor, but we should manually delete some members like pointer.
Constructor and destructor.

```cpp
class Course {

public:

    Course();  // Neither constructor or destructor are syntactically necessary when defining a class.
    ~Course();

private:

    int *p;
    int capacity;

};
```
Define class *before* declaring object in main.

Define class `Course` before main. 

```
class Course {...};
int main()
{
    Course x(100, "Data Structure");
    Course y(40, "Program Structure");
    x.printCap();
    y.setCap(60);
    return 0;
}
```

Main function

Member functions *can* be defined after main, since we know their class-ship.
Declare an array of objects.

An array of Course objects.

First object $x[0]$ has capacity = 10

Title = “data structure”

Second object $x[1]$ has capacity = 20

Title = “program structure”

Third object $x[2]$ has capacity = 30

Title = “Java”

```java
int main()
{
    Course x[3] = {
        {10, "Data Structure"},
        {20, "Program Structure"},
        {30, "Java"}
    };

    x[0].printCap();
    x[1].printTitle();
    return 0;
}
```
Access elements of an object array.

```java
int main()
{
    Course x[3] = {
        {10, "Data Structure"},
        {30, "Program Structure"},
        {30, "Java"}
    };

    x[0].printCap(); // Print 10.
    x[1].printTitle(); // Print "Program Structure".
    return 0;
}
```
Declare a pointer of class, and access members using “→”.

```cpp
class Course {
public:
    int getCap();
    void setCap(int n);
private:
    int capacity;
};
```
We are done with the concept of class!

Next let’s briefly introduce dynamic memory allocation.
Allocation of memory for a declared variable `x`.

```cpp
int main()
{
    for (int i = 0; i < 5; i++) {
        int x;
        x = i;
        cout << x << ' ' << &x << '
';
    }
    return 0;
}
```

We have a for loop.

In each iteration, we aim to declare an integer variable `x` and assign a value to it.

Output `x` and its address.
Allocation of memory for a declared variable x.

Before running the program, the compiler will allocate a fixed chunk of memory for a variable called x.

This allocation is fixed, and does not depend on input.

```c
int main()
{
    for (int i = 0; i < 5; i++) {
        int x;
        x = i;
        cout << x << ' ' << &x << ' \n';
    }
    return 0;
}
```

All declared x's are allocated with the same chunk of memory.

Static memory allocation.
Allocation of memory for a declared variable x.

If we want to allocate a new chunk of memory to every newly declared variable, we can apply the dynamic memory allocation method.

We will access the variable using pointer.

```cpp
int main()
{
    for (int i = 0; i < 5; i++) {
        int *x = new int;
        *x = i;
        cout << *x << ' ' << x << '\n';
    }
    return 0;
}
```

allocate a new chunk for an integer variable.

Address of this chunk is assigned to a pointer x.
Allocation of memory for a declared variable x.

If we want to allocate a new chunk of memory to every newly declared variable, we can apply the dynamic memory allocation method.

We will access the variable using pointer.

```c++
int main()
{
    for (int i = 0; i < 5; i++) {
        int *x = new int;
        *x = i;
        cout << *x << ' ' << x << '
';
    }

    return 0;
}
```

Each newly declared chunk has an address.

Dynamic memory allocation.
Allocation of memory for a declared variable x.

```cpp
int main()
{
    for (int i = 0; i < 5; i++) {
        int *x = new int;
        *x = i;
        cout << *x << ' ' << x << '
';
    }
    return 0;
}
```

We can replace “int” with
- float
- char
- a class
Dynamic memory allocation is particularly powerful when used in combination with a loop of input.
Take input of arbitrary size using “while + cin”.

```cpp
int main()
{
    int temp;
    while(cin >> temp) {
        int x;
        x = temp;
        cout << x << ' ' << &x << '
';
    }
    return 0;
}
```

One way to take an arbitrary size of input is using “while” + “cin”.

As long as the input is an integer, condition (cin >> temp) is true and we get to continue taking the next input!

But if input is not integer, condition is false and we stop taking inputs.
Example Process

```cpp
int main()
{
    int temp;

    while(cin >> temp) {
        int x;
        x = temp;
        cout << x << ' ' << &x << '
';
    }

    return 0;
}
```

Input a sequence of numbers ended with a character ‘s’.

Press ‘enter’.
Example Process

```cpp
int main()
{
    int temp;
    while(cin >> temp) {
        int x;
        x = temp;
        cout << x << ' ' << &x << '
';
    }
    return 0;
}
```

**Round 1**

<table>
<thead>
<tr>
<th>temp.val</th>
<th>x.address</th>
<th>x.value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Input an integer 1 to temp, condition (cin>>temp) is true.

So run one iteration of loop.
Example Process

Round 1

<table>
<thead>
<tr>
<th>temp.val</th>
<th>x.address</th>
<th>x.value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>005</td>
<td></td>
</tr>
</tbody>
</table>

```c
int main()
{
    int temp;
    while(cin >> temp) {
        int x;
        x = temp;
        cout << x << ' ' << &x << '\n';
    }
    return 0;
}
```

Declare an int variable x.
Example Process

```c++
int main()
{
    int temp;
    while(cin >> temp) {
        int x;
        x = temp;
        cout << x << ' ' << &x << '
';
    }
    return 0;
}
```

Round 1

<table>
<thead>
<tr>
<th>temp.val</th>
<th>x.address</th>
<th>x.value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>005</td>
<td>1</td>
</tr>
</tbody>
</table>

Assign temp to x.
Example Process

```cpp
int main()
{
    int temp;

    while(cin >> temp) {
        int x;
        x = temp;

        cout << x << ' ' << &x << '
';
    }

    return 0;
}
```

This iteration is done!

<table>
<thead>
<tr>
<th>temp.val</th>
<th>x.address</th>
<th>x.value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>005</td>
<td>1</td>
</tr>
</tbody>
</table>
Example Process

```c
int main()
{
    int temp;
    while(cin >> temp) {
        int x;
        x = temp;
        cout << x << ' ' << &x << '
';
    }
    return 0;
}
```

**Round 2**

<table>
<thead>
<tr>
<th>temp.val</th>
<th>x.address</th>
<th>x.value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>005</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Input an integer 5 to temp, condition `(cin>>temp)` is true

So run one iteration of loop.
Example Process

Round 2

```
int main()
{
    int temp;

    while(cin >> temp) {
        int x;
        x = temp;
        cout << x << ' ' << &x << '
';
    }

    return 0;
}
```

<table>
<thead>
<tr>
<th>temp.val</th>
<th>x.address</th>
<th>x.value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>005</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>005</td>
<td></td>
</tr>
</tbody>
</table>

Declare an int variable x.
Example Process

```cpp
int main()
{
    int temp;
    while(cin >> temp) {
        int x;
        x = temp;
        cout << x << ' ' << &x << '
';
    }
    return 0;
}
```

<table>
<thead>
<tr>
<th>Round 2</th>
<th>temp.val</th>
<th>x.address</th>
<th>x.value</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>005</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>005</td>
<td>5</td>
</tr>
</tbody>
</table>

Assign temp to x.
Example Process

```cpp
int main()
{
    int temp;
    while(cin >> temp) {
        int x;
        x = temp;
        cout << x << ' ' << &x << '\n';
    }
    return 0;
}
```

Round 2

<table>
<thead>
<tr>
<th>temp.val</th>
<th>x.address</th>
<th>x.value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>005</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>005</td>
<td>5</td>
</tr>
</tbody>
</table>

Output x value and address.

This iteration is done!
### Example Process

```c
int main()
{
    int temp;

    while(cin >> temp) {
        int x;
        x = temp;
        cout << x << ' ' << &x << '
';
    }

    return 0;
}
```

#### Round 3

<table>
<thead>
<tr>
<th>temp.val</th>
<th>x.address</th>
<th>x.value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>005</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>005</td>
<td>5</td>
</tr>
<tr>
<td>32</td>
<td>005</td>
<td>32</td>
</tr>
</tbody>
</table>

Output x value and address.

This iteration is done!
Example Process

```c
int main()
{
    int temp;

    while(cin >> temp) {
        int x;
        x = temp;
        cout << x << ' ' << &x << '\n';
    }

    return 0;
}
```

Output x value and address.

This iteration is done!
Example Process

```c
int main()
{
    int temp;
    while(cin >> temp) {
        int x;
        x = temp;
        cout << x << ' ' << &x << '\n';
    }
    return 0;
}
```

Round 5

<table>
<thead>
<tr>
<th>temp.val</th>
<th>x.address</th>
<th>x.value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>005</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>005</td>
<td>5</td>
</tr>
<tr>
<td>32</td>
<td>005</td>
<td>32</td>
</tr>
<tr>
<td>17</td>
<td>005</td>
<td>17</td>
</tr>
</tbody>
</table>

Input ‘s’ to temp, condition (cin >> temp) is false!

Terminate the loop.
Example Process: Observation

```cpp
int main()
{
    int temp;
    while(cin >> temp) {
        int x;
        x = temp;
        cout << x << ' ' << &x << '\n';
    }

    return 0;
}
```

<table>
<thead>
<tr>
<th>temp.val</th>
<th>x.address</th>
<th>x.value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>005</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>005</td>
<td>5</td>
</tr>
<tr>
<td>32</td>
<td>005</td>
<td>32</td>
</tr>
<tr>
<td>17</td>
<td>005</td>
<td>17</td>
</tr>
</tbody>
</table>

All values are stored in the same chunk of memory, overwriting their ancestors.

We can’t retrieve the previous inputs.
We see a limitation of static memory allocation, now let’s try dynamic memory allocation instead.
Dynamic Memory Allocation: Example Process

```cpp
int main()
{
    int temp;

    while(cin >> temp) {
        int *x = new int;
        *x = temp;
        cout << *x << ' ' << x << '\n';
    }

    return 0;
}
```
Dynamic Memory Allocation: Example Process

```cpp
int main()
{
    int temp;
    while(cin >> temp) {
        int *x = new int;
        *x = temp;
        cout << *x << ' ' << x << '
';
    }
    return 0;
}
```

**Round 1**

Input an integer 1 to `temp`, condition `(cin>>temp)` is true

<table>
<thead>
<tr>
<th>temp.val</th>
<th>x.address</th>
<th>x.value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Dynamic Memory Allocation: Example Process

```
int main()
{
    int temp;

    while(cin >> temp) {
        int *x = new int;
        *x = temp;
        cout << *x << ' ' << x << '
';
    }

    return 0;
}
```

Round 1

<table>
<thead>
<tr>
<th>temp.val</th>
<th>x.address</th>
<th>x.value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>005</td>
<td></td>
</tr>
</tbody>
</table>

Allocate a new chunk and assign its address to x.
Dynamic Memory Allocation: Example Process

```c
int main()
{
    int temp;

    while(cin >> temp) {
        int *x = new int;
        *x = temp;
        cout << *x << ' ' << x << '
';
    }
    return 0;
}
```

<table>
<thead>
<tr>
<th>temp.val</th>
<th>x.address</th>
<th>x.value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>005</td>
<td>1</td>
</tr>
</tbody>
</table>

Round 1

Place temp in that chunk.
Dynamic Memory Allocation: Example Process

```
int main()
{
    int temp;
    while(cin >> temp) {
        int *x = new int;
        *x = temp;
        cout << *x << ' ' << x << '\n';
    }
    return 0;
}
```

Input an integer 5 to temp, condition (cin>>temp) is true
Dynamic Memory Allocation: Example Process

```c
int main()
{
    int temp;

    while(cin >> temp) {
        int *x = new int;
        *x = temp;
        cout << *x << ' ' << x << '\n';
    }

    return 0;
}
```

Allocate a new chunk and assign its address to `x`.

This address is different from the previous address for 1.
Dynamic Memory Allocation: Example Process

```cpp
int main()
{
    int temp;

    while(cin >> temp) {
        int *x = new int;
        *x = temp;
        cout << *x << ' ' << x << '\n';
    }

    return 0;
}
```

**Round 2**

<table>
<thead>
<tr>
<th>temp.val</th>
<th>x.address</th>
<th>x.value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>005</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>007</td>
<td>5</td>
</tr>
</tbody>
</table>

Place temp in that chunk.
int main()
{
    int temp;
    while(cin >> temp) {
        int *x = new int;
        *x = temp;
        cout << *x << ' ' << x << '\n';
    }
    return 0;
}
Dynamic Memory Allocation: Example Process

```cpp
int main()
{
    int temp;

    while(cin >> temp) {
        int *x = new int;
        *x = temp;
        cout << *x << ' ' << x << '
';
    }

    return 0;
}
```

<table>
<thead>
<tr>
<th>temp.val</th>
<th>x.address</th>
<th>x.value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>005</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>007</td>
<td>5</td>
</tr>
<tr>
<td>32</td>
<td>012</td>
<td>32</td>
</tr>
<tr>
<td>17</td>
<td>020</td>
<td>17</td>
</tr>
</tbody>
</table>
Dynamic Memory Allocation: Example Process

```cpp
int main()
{
    int temp;

    while(cin >> temp) {
        int *x = new int;
        *x = temp;
        cout << *x << ' ' << x << '
';
    }
    return 0;
}
```

Round 5

<table>
<thead>
<tr>
<th>temp.val</th>
<th>x.address</th>
<th>x.value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>005</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>007</td>
<td>5</td>
</tr>
<tr>
<td>32</td>
<td>012</td>
<td>32</td>
</tr>
<tr>
<td>17</td>
<td>020</td>
<td>17</td>
</tr>
</tbody>
</table>

Input ‘s’ to temp, condition (cin >> temp) is false!

Terminate the loop.
Dynamic Memory Allocation: Observation

```c
int main()
{
    int temp;
    while(cin >> temp) {
        int *x = new int;
        *x = temp;
        cout << *x << ' ' << x << '\n';
    }
    return 0;
}
```

<table>
<thead>
<tr>
<th>temp.val</th>
<th>x.address</th>
<th>x.value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>005</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>007</td>
<td>5</td>
</tr>
<tr>
<td>32</td>
<td>012</td>
<td>32</td>
</tr>
<tr>
<td>17</td>
<td>020</td>
<td>17</td>
</tr>
</tbody>
</table>

Different values are stored in different chunks of memory.

We can retrieve them in future (with properly stored addresses).
More about input and output from “while” + “cin”.

```cpp
int main()
{
    int temp;
    while(cin >> temp) {
        int *x = new int;
        *x = temp;
        cout << x << ' ' << &x << '\n';
    }
    return 0;
}
```
Outline

**Visual Studio**: start project, debug program

**Program Structure**: iostream, std, main, sequential, return, comment

**Integer Variable**: declare, assign, initialize, arithmetic, input/output, round, unsigned, constant.

**Array Variable**: declaration, index, initialization, multi-dimensional array, sparse matrix

**Instruction**: arithmetic/increment, relational/logical, bitwise, if-else, switch, while-loop, for-loop, break

**Other Variables**: float, char, string, enum, vector, **structure**

**Pointer Variable**: idea, declare, extract address, access content, allocation, relation to array

**Function**: definition, declaration, assignment passing, local/global variable, overload

**Class**: definition, public/private member, constructor/destructor, declaration, memory
7. Structure Variable

A structure is a user defined aggregation of elements.

These element can be variables of different types.

We may interpret Course as a type of “meta-variable”.

```c
int main()
{
    enum day {Mon, Tue, Wed, Thu, Fri};

    struct Course {
        string title;
        int capacity;
        day time ;
    };

    return 0;
}
```

Define a structure “Course”.

It contains three elements
- a string “title”
- an int capacity
- an enum

We can include any other type of variables as members of this structure.
Declaration and member access.

Now we have a new type of meta-variable called Course.

```
struct Course {
    string title;
    int capacity;
    day time;
};
```

Declare a Course variable x.

```
Course x;
```

Access member variables through indexing.

```
x.title = "Data Structure";
x.capacity = 140;
x.time = Mon;
```

Play with them as if they are regular variables.

- assign title in x.
- assign capacity.
- assign time[0].

Syntax: x.member
Initialization

We can initialize a structure by a set of ordered values.

```cpp
struct Course {
    string name;
    int capacity;
    day time;
};

Course x = { "Data Structure", 140, Mon };

cout << "Title: " << x.name << 'n';
cout << "Capacity: " << x.capacity << 'n';
cout << "Time: " << x.time << 'n';
```

In this set, 1st value goes to 1st member. 2nd value goes to 2nd member. 3rd value goes to 3rd member.

Syntax also applies in structure var assignment.
Initialization

```c++
struct Course {
    string name;
    int capacity;
    day time);

Course x = { "Data Structure", 140, Mon };

cout << "Title: " << x.name << '
';

cout << "Capacity: " << x.capacity << '
';

cout << "Time: " << x.time << '
';
```

0 means Mon.
Copy of a structure variable.

```cpp
struct Course {
    string name;
    int capacity;
    string time;
};

Course x, y;

x = {"Data Structure", 140, Mon};
y = x;
```

Two variables have the same structure. Copy in an element-wise way.