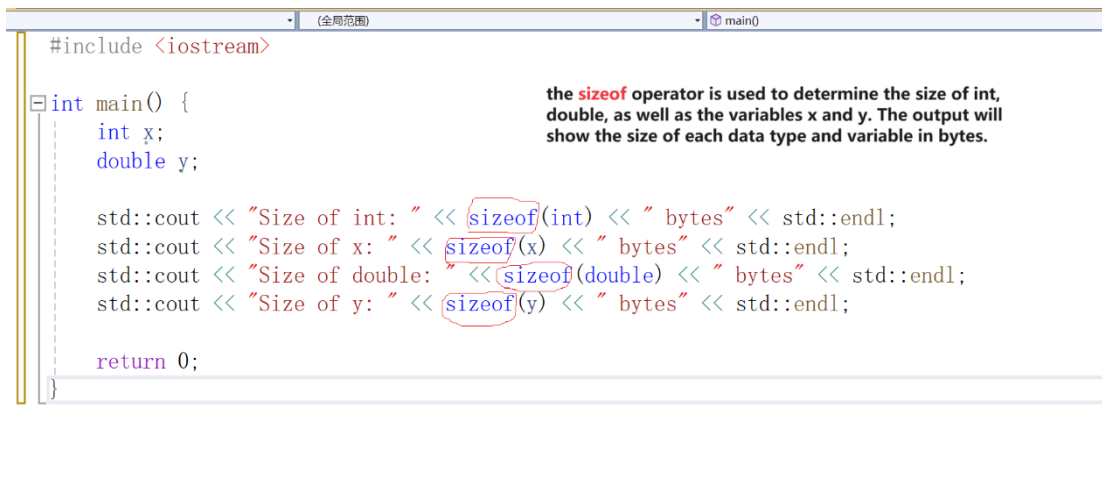


Q1:

The **sizeof** operator is particularly important in C++ because C++ is a statically typed language, meaning that the size of data types is known at compile time. This allows C++ to manage memory efficiently and perform operations on data types based on their size.

The **sizeof** keyword is used to determine the size in bytes of a data type or a variable. It is a compile-time operator that returns the size of a variable or a data type. The **sizeof** operator is very useful in situations where you need to know the size of a data type or a variable in memory.



```
#include <iostream>

int main() {
    int x;
    double y;

    std::cout << "Size of int: " << sizeof(int) << " bytes" << std::endl;
    std::cout << "Size of x: " << sizeof(x) << " bytes" << std::endl;
    std::cout << "Size of double: " << sizeof(double) << " bytes" << std::endl;
    std::cout << "Size of y: " << sizeof(y) << " bytes" << std::endl;

    return 0;
}
```

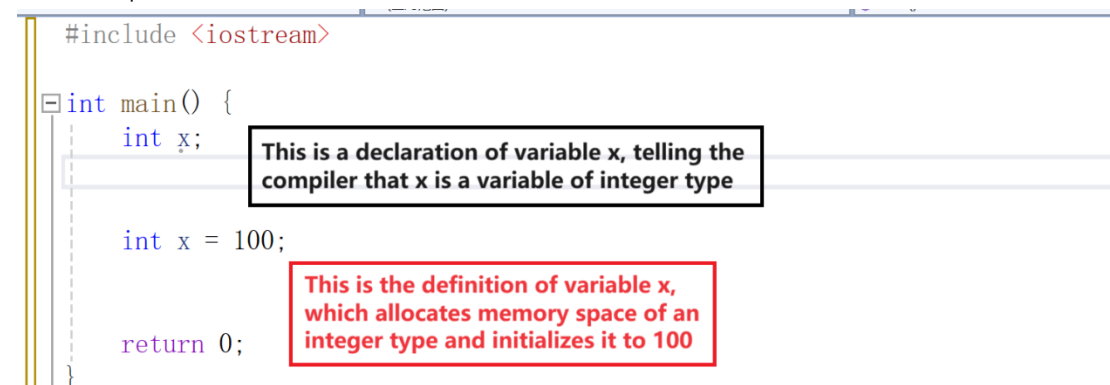
the **sizeof** operator is used to determine the size of int, double, as well as the variables x and y. The output will show the size of each data type and variable in bytes.

Q2:

A variable **declaration** is the process of telling the compiler in code that a variable will be used, but not allocating memory or initializing the variable. Declaring a variable simply tells the compiler the type and name of the variable so that it can be used later in the code. When declaring a variable, the compiler is aware of its existence, but does not allocate memory space for it.

A variable **definition** is the process of allocating memory space to a variable and possibly initializing it. Defining a variable means allocating storage space in memory for the variable so that it can be used in the program. When a variable is defined, the compiler allocates memory for that variable.

For example:



```
#include <iostream>

int main() {
    int x;

    int x = 100;

    return 0;
}
```

This is a declaration of variable x, telling the compiler that x is a variable of integer type

This is the definition of variable x, which allocates memory space of an integer type and initializes it to 100

Q3:

In C++, the **auto** specifier is a keyword that allows the compiler to automatically deduce the data type of a variable based on its initializer. This feature is known as "type inference" and was introduced in C++11 to simplify code and reduce redundancy when declaring variables.

When you use **auto** to declare a variable, the compiler determines the type of that variable by looking at the expression used to initialize it. This can be particularly useful when dealing with complex data types, such as iterators or lambda functions, where explicitly specifying the type can be cumbersome or error-prone.

Variables must be initialized when using **auto** to define them, and the compiler needs to derive the actual type of **auto** based on the initialization expression during the compilation phase. Therefore, **auto** is not a "type" declaration, but a "placeholder" when a type declaration, and the compiler replaces **auto** with the actual type of the variable during compilation.

```
#include<iostream>
using namespace std;
int TestAuto()
{
    return 10;
}
int main()
{
    int a = 10;
    auto b = a; //variable
    auto c = 'a'; //character
    auto d = TestAuto(); //The return of a function
    cout << typeid(b).name() << endl;
    cout << typeid(c).name() << endl;
    cout << typeid(d).name() << endl;

    //auto e;
    return 0;
}
```

It cannot be compiled, and variables must be initialized when defining them with **auto**

Q4:

(The example given is in the picture)

1. Differences in Bottom logic:

A pointer is essentially a variable that stores the address of a variable, which is logically independent, while a reference is an alias that is not logically independent, and its existence is dependent. Pointer is an entity, while reference is only an alias.

Example:

```
1  #include <iostream>
2  using namespace std;
3  int main() {
4      int num = 10;
5      int* ptr = &num;
6      int& ref = num;
7      cout << "Original value of num: " << num << endl;
8      *ptr = 20;
9      cout << "Value of num after modifying through pointer: " << num << endl;
10
11     ref = 30;
12     cout << "Value of num after modifying through reference: " << num << endl;
13
14     cout << "Address of num: " << &num << endl;
15     cout << "Address stored in ptr: " << ptr << endl;
16
17     // int *ptrToRef = &ref;
18     // Attempting to create a pointer to a reference will result in a compilation error
19
20     return 0;
}
```

Ptr is a pointer to a num variable, which stores the address of the num variable and can be modified by indirectly referencing * ptr to modify the value of num.

And ref is a reference to num, it actually does not have its own storage spaceso modifying the value of num through ref is actually directly modifying num itself.

2. Initialization necessary and memory allocation:

The reference must be initialized, but storage space is not allocated. Initialize when pointer is not declared, and allocate storage space during initialization.

Example:

```
#include <iostream>
using namespace std;
int main() {
    int num = 10;
    int* ptr;
    int& ref = num;

    ptr = &num;

    cout << "Value of num: " << num << endl;
    cout << "Value of num through pointer: " << *ptr << endl;
    cout << "Value of num through reference: " << ref << endl;

    return 0;
}
```

Ptr is a pointer that is declared but not initialized, and then in subsequent code, `ptr = &num` Initialized and allocated storage space. And ref is a reference that is initialized at the same time as declaration, and it does not require additional storage space.

3. Modifiability

Pointers can be changed, including changes to the address they point to and changes to the data stored in the address they point to. But the referenced object cannot be changed throughout its entire lifecycle, and can only be attached to the same variable from beginning to end.

Example:

```
#include <iostream>
using namespace std;
int main() {
    int num1 = 10;
    int num2 = 20;

    int* ptr = &num1;
    int& ref = num1;

    ptr = &num2; // we can modify pointer to point to num2
    // ref = num2;
    // This line of code will cause a compilation error as the reference cannot change the bound object

    return 0;
}
```

4 Can or can not be empty:

References cannot be empty, pointers can be empty.

Example:

```
#include <iostream>
using namespace std;
int main() {
    int num = 10;
    int* ptr = nullptr;
    int& ref = num;

    return 0;
}
```

5. Method of Parameter Passing:

Pointer passing parameters is essentially a way of value passing, passing an address value. During the value transfer process, the formal parameters of the called function are treated as local variables of the called function, which opens up memory space on the stack to store the values of the arguments put in by the main calling function, thus becoming a copy of the arguments. The characteristic of value transfer is that any operation of the called function on the formal parameter is performed as a local variable and does not affect the value of the argument variable of the main calling function.

In the process of reference passing, although the formal parameters of the called function also open up memory space in the stack as local variables, the address of the actual parameter variable put in by the calling function is stored at this time. Any operation on formal parameters by the called function is treated as indirect addressing, which accesses the actual parameter variables in the calling function through the address stored on the stack. Because of this, any operation on formal parameters by the called function affects the actual parameter variables in the main calling function.

Example:

```
1 #include <iostream>
2 using namespace std;
3 void ByPointer(int* numPtr) {
4     (*numPtr)++;
5     numPtr = nullptr;
6 }
7
8 void ByReference(int& numRef) {
9     numRef++;
10    numRef = 0;
11 }
12
13 int main() {
14     int num = 5;
15     int* ptr = &num;
16     ByPointer(ptr);
17     cout << "Value of num after incrementing by pointer: " << num << endl;
18
19     int& ref = num;
20     ByReference(ref);
21     cout << "Value of num after incrementing by reference: " << num << endl;
22     return 0;
23 }
```

In the ByPointer function, although we change the pointer's pointing inside the function, it does not affect the value of the argument variable in the calling function.

In the ByReference function, since the reference itself is an alias of a variable, the operation on the reference directly affects the value of the argument variable in the calling function

Value of num after incrementing by pointer: 6
Value of num after incrementing by reference: 0

E:\Project12\x64\Debug\Project12.exe (进程 19420)已退出, 代码为 0。
要在测试停止时自动关闭控制台, 请启用“工具”->“选项”->“测试”->“测试停止时自动关闭控制台”

6. Security:

References are type safe because type checks are performed during compilation; Pointers may not be type safe, as they can point to any type of data, which may result in type mismatch or illegal memory access.

Q5:

New and malloc:

1.New is an operator in C++, while malloc is a library function in C language. New calls the constructor to initialize the allocated memory, while malloc only allocates a block of memory space and does not call the constructor.

2.New returns a pointer to an object, while malloc returns a void * pointer that requires type conversion.

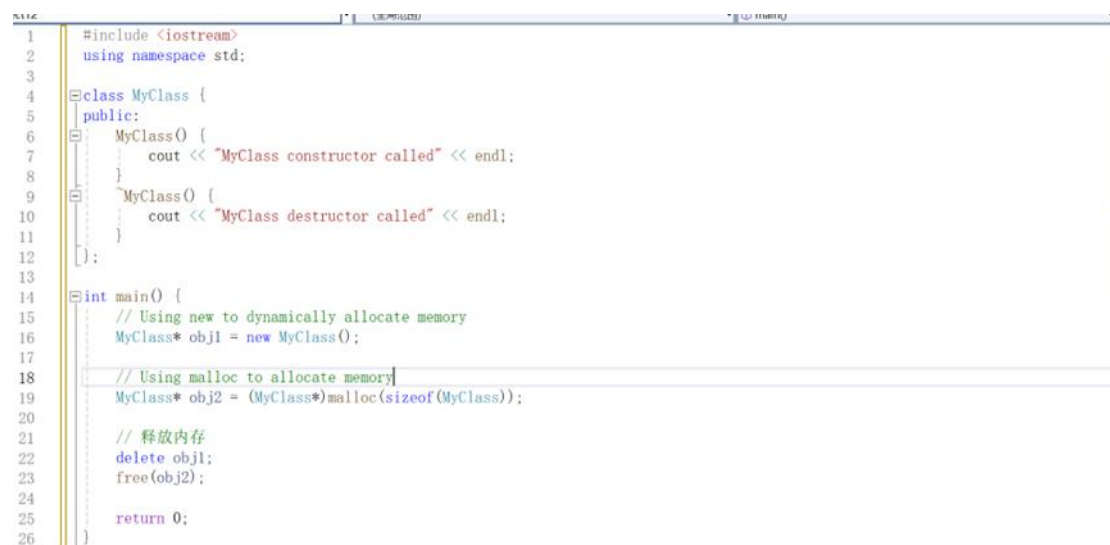
3.New can be overloaded as needed, while malloc is a fixed function that cannot be overloaded.

Free and delete:

1.Free is a library function in C language used to release memory allocated through functions such as malloc, calloc, or realloc; And delete is an operator in C++ used to release memory allocated through the new operator.

2.Free only releases memory and does not call the object's destructor; And delete will first call the object's destructor, and then release memory.

3.When using free to release memory, it is necessary to manually calculate the size of the memory to be released and pass it to the free function; And delete will automatically call the appropriate destructor based on the type and release the corresponding size of memory.



```
1  #include <iostream>
2  using namespace std;
3
4  class MyClass {
5  public:
6      MyClass() {
7          cout << "MyClass constructor called" << endl;
8      }
9      ~MyClass() {
10         cout << "MyClass destructor called" << endl;
11     }
12 };
13
14 int main() {
15     // Using new to dynamically allocate memory
16     MyClass* obj1 = new MyClass();
17
18     // Using malloc to allocate memory
19     MyClass* obj2 = (MyClass*)malloc(sizeof(MyClass));
20
21     // 释放内存
22     delete obj1;
23     free(obj2);
24
25     return 0;
26 }
```

Q6:

The screenshot shows a C++ code editor with the following code and annotations:

```

1 #include <iostream>
2 #include <memory>
3 using namespace std;
4 int main() {
5     int i = 10;
6     auto_ptr<int> ap1(new int(4)), ap2;
7     ap2 = ap1;
8     cout << *ap2;
9     cout << *ap1 << endl;
10    char* c;
11    shared_ptr<char> sc;
12    sc = c;
13    sc = new char(10);
14    return 0;
15 }

```

Annotations:

- Line 6: `auto_ptr<int> ap1(new int(4)), ap2;` and `ap2 = ap1;` are boxed. An arrow points to the text: "auto_ptr will transfer ownership to ap2 after assignment, causing ap1 to become a null pointer, resulting in undefined behavior when outputting *ap1."
- Line 12: `sc = c;` is boxed. An arrow points to the text: "When using shared_ptr, the original pointer should be passed directly to the constructor of shared_ptr, rather than being assigned to shared_ptr. In this code, sc=c; It is incorrect, shared_ptrsc(c) should be used; Package c as shared_ptr."
- Line 13: `sc = new char(10);` is boxed. An arrow points to the text: "When using shared_ptr, make share or make share should be used to allocate memory, rather than directly using new. Should use shared_ptrsc=make_shared('a '); To allocate memory and initialize shared_ptr."

The screenshot shows a C++ code editor with the following code:

```

1 #include <iostream>
2 #include <memory>
3 using namespace std;
4
5 int main() {
6     int i = 10;
7     unique_ptr<int> up1(new int(4)), up2;
8     up2 = move(up1);
9     cout << *up2 << endl;
10
11    char* c = new char(10);
12    shared_ptr<char> sc(c);
13    cout << *c << endl;
14
15    return 0;
16 }

```

The execution output is shown in a terminal window:

```

Process exited after 0.741 seconds with return value 0
请按任意键继续. . .

```

Q7:

(1):

a=12

b=3

c=45

d=634

(2):

Answer as shown in the picture

```
123456789012345678901234567890
14          12          18
24          22          30
          20          18          24
```

(3):

2356

Q8:

```
#include <iostream>
#include <iomanip>
using namespace std;
int main() {
    int height = 5;
    for (int i = 1; i <= height; i++)
    {
        cout << setw(height - i + 1);
        for (int j = 1; j <= 2 * i - 1; j++) {
            cout << "*";
        }
        cout << endl;
    }
    return 0;
```

this code is used to control the printing position of each line, aligning the bottom of the pyramid with the left.

The following is the result of the program execution:

