Q1:

1.Pass by Value: In this method, a copy of the actual parameter value is passed to the function. Changes made to the parameter within the function do not affect the original value. This method is simple and safe but can be less efficient for large data structures.

2.Pass by Reference: Instead of passing a copy of the value, the memory address (reference) of the actual parameter is passed to the function. Any changes made to the parameter within the function will affect the original value. This method is efficient for large data structures but can lead to unintended side effects if not used carefully.

3.Pass by Pointer: Similar to pass by reference, but a pointer to the actual parameter is passed. This method allows for more explicit manipulation of memory addresses but requires careful handling to avoid issues like null pointer dereferencing.

4.Pass by Constant Reference: This method is similar to pass by reference but ensures that the parameter cannot be modified within the function. It provides efficiency and safety when passing large data structures.

5.Pass by Name (Lazy Evaluation): In this method, the actual parameter expression is passed and evaluated only when it is used in the function. This allows for deferred evaluation of parameters and can be useful in certain scenarios.

When selecting a parameter passing method, consider the following factors:

Efficiency: Pass by value is generally less efficient for large data structures compared to pass by reference or pass by pointer. Choose the method that minimizes unnecessary copying of data.

Safety: Pass by value provides safety by ensuring that the original data is not modified, while pass by reference or pass by pointer allows for direct manipulation of the original data. Choose the method that best suits the requirements of your program.

Mutability: If you want the function to modify the original data, pass by reference or pass by pointer is suitable. If the data should remain unchanged, pass by value or pass by constant reference can be used.

Memory Management: Pass by pointer requires careful memory management to avoid issues like memory leaks or dangling pointers. Ensure proper initialization and deletion of pointers when using this method.

Function Interface: Consider the function's interface and the intended behavior when choosing a parameter passing method. Select the method that best aligns with the function's purpose and expected interactions with the parameters.

Q2:

Parameter Passing:

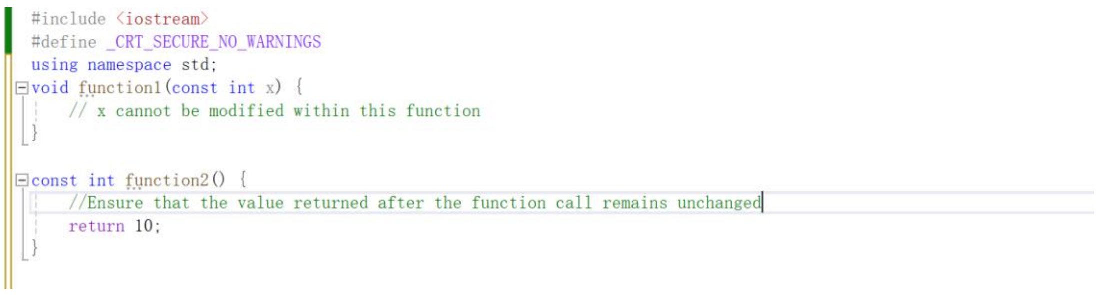
When a parameter is declared as **const** in a function, it means that the function promises not to modify the value of that parameter. This is particularly useful when you want to ensure that a function does not accidentally modify the input parameter.

It also allows you to pass constant values, such as literals or variables declared as **const**, to the function without the risk of unintended changes.

Function Return Value:

When a function returns a const value, it means that the returned value cannot be modified by the caller. This is useful when you want to ensure that the returned value remains constant after the function call.

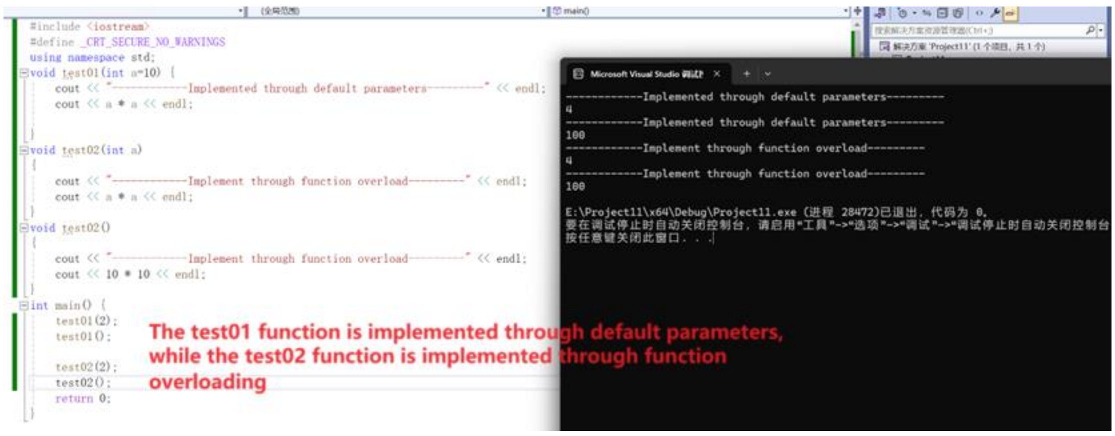
It helps prevent accidental modification of the returned value and ensures that the caller treats the returned value as read-only.



```
#include <iostream>
#define _CRT_SECURE_NO_WARNINGS
using namespace std;
void function1(const int x) {
    // x cannot be modified within this function
}

const int function2() {
    //Ensure that the value returned after the function call remains unchanged
    return 10;
}
```

Q3:

Default arguments in functions allow you to provide a default value for a parameter, which can be omitted when calling the function. This can also be achieved by overloading the function with different sets of parameters.

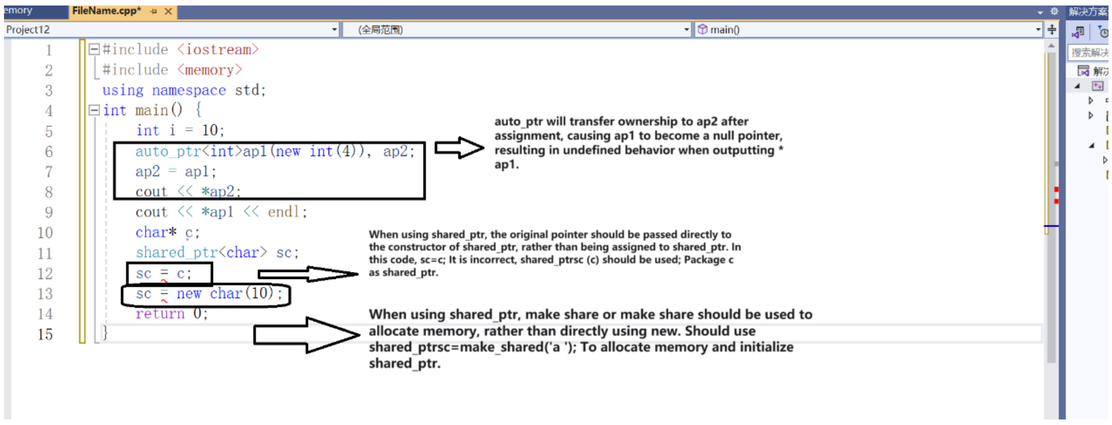


```
#include <iostream>
#define _CRT_SECURE_NO_WARNINGS
using namespace std;
void test01(int a=10) {
    cout << "------------Implemented through default parameters---------" << endl;
    cout << a * a << endl;

}
void test02(int a)
{
    cout << "------------Implement through function overload---------" << endl;
    cout << a * a << endl;
}
void test02()
{
    cout << "------------Implement through function overload---------" << endl;
    cout << 10 * 10 << endl;
}
int main() {
    test01(2);
    test01();

    test02(2);
    test02();
    return 0;
}
```

```
------------Implemented through default parameters---------
4
------------Implemented through default parameters---------
100
------------Implement through function overload---------
4
------------Implement through function overload---------
100

E:\Project11\x64\Debug\Project11.exe (进程 28472)已退出，代码为 0。
要在调试停止时自动关闭控制台，请启用"工具"->"选项"->"调试"->"调试停止时自动关闭控制台
按任意键关闭此窗口. . .
```

The test01 function is implemented through default parameters, while the test02 function is implemented through function overloading

Q4:

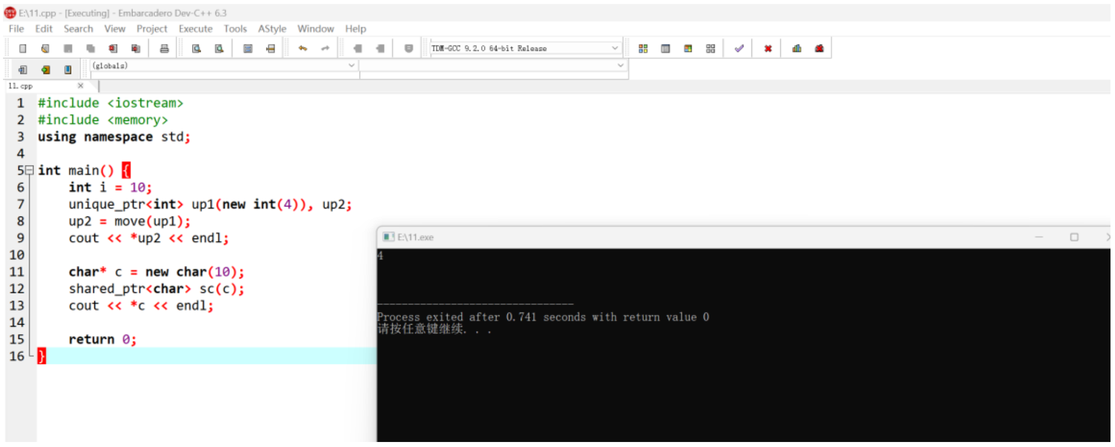


```cpp
#include <iostream>
#include <memory>
using namespace std;
int main() {
    int i = 10;
    auto_ptr<int>ap1(new int(4)), ap2;
    ap2 = ap1;
    cout << *ap2;
    cout << *ap1 << endl;
    char* c;
    shared_ptr<char> sc;
    sc = c;
    sc = new char(10);
    return 0;
}
```

auto_ptr will transfer ownership to ap2 after assignment, causing ap1 to become a null pointer, resulting in undefined behavior when outputting * ap1.

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.

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.



```cpp
#include <iostream>
#include <memory>
using namespace std;

int main() {
    int i = 10;
    unique_ptr<int> up1(new int(4)), up2;
    up2 = move(up1);
    cout << *up2 << endl;

    char* c = new char(10);
    shared_ptr<char> sc(c);
    cout << *c << endl;

    return 0;
}
```

```
4

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

Q5:

625
18.4
6.6
9409
9
81

Q6:
（1）
Program error

Reason:
Since C/C++passes parameters by value, modifying pointers inside a function does not affect the value of pointers outside the function. Therefore, the str in the function Test is still NULL, and attempting to copy a string onto a null pointer in the strcpy function can result in undefined behavior, which may cause the program to crash.
Therefore, there may be errors during the runtime of this code, and the output result is uncertain. It may cause program crashes or other abnormal situations.

（2）
Program error

Reason:
When the GetMemory function returns after execution, the memory space of the local variable p will be freed, so the returned pointer will point to an invalid memory address. In the Test function, assigning this invalid address to the str pointer and attempting to print its contents will result in undefined behavior.
Due to p being a local array, its lifecycle is limited to within the GetMemory function. When the GetMemory function returns, the memory occupied by p will be released, and the memory address pointed to by the str pointer will become a dangling pointer. Accessing the value of the dangling pointer will result in undefined behavior.

(3)
hello

reason:
The program passes a pointer to a pointer, allowing the allocated memory address to be accessed outside of the function. So "hello" can be successfully printed.

(4)
world

reason:
Str was allocated 100 bytes of space, and then the string "hello" was copied into this memory, which was then released using the free function. Due to the previous release of memory, the memory pointed to by str is no longer valid, but str itself has not been set to NULL. Therefore, the following steps can be carried out normally through the if condition

If the above program is changed from C style to C++style, malloc is replaced with new, and free is replaced with delete, the results of the program may be different.

In C++, using new to allocate memory calls the object's constructor, while malloc simply allocates memory space. Similarly, delete calls the destructor of the object, while free only frees up memory space. Therefore, if there are class constructors and destructors in the program, using new and delete may result in different behaviors.

In addition, in C++, new returns a pointer to an object, while malloc returns a void pointer, so there is no need for type conversion when using new. When using delete, there is no need to specify the size of the memory block to be released. Delete will automatically call the destructor and release the memory.

Programming suggestions about dynamic memory management in C++:

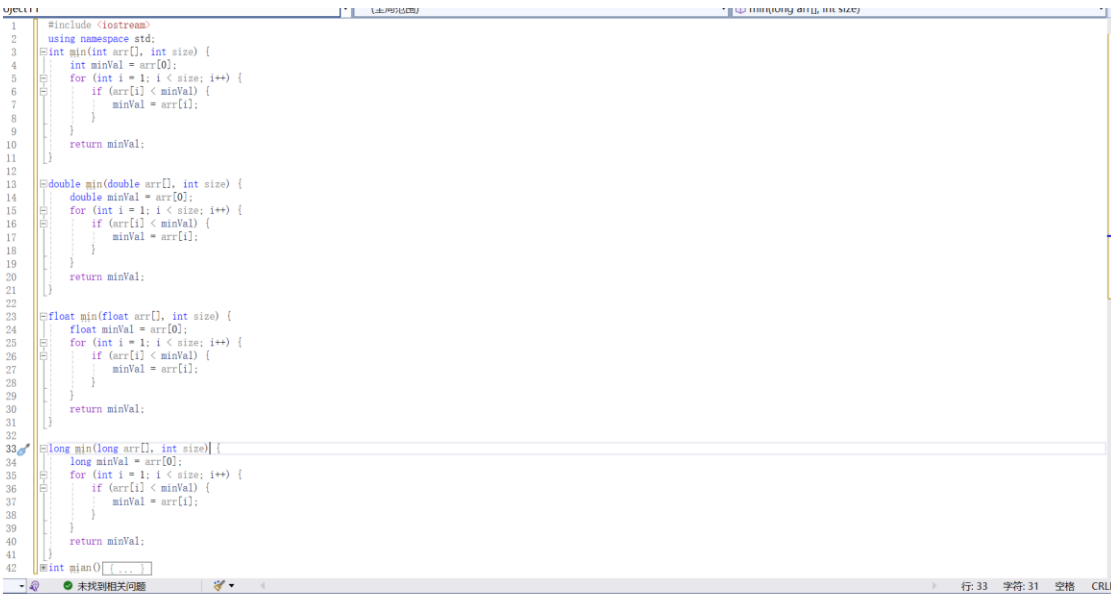
Prefer Smart Pointers: Use smart pointers like std::unique_ptr and std::shared_ptr to manage dynamic memory. They automatically handle memory deallocation when they go out of scope.

Avoid Raw Pointers: Minimize the use of raw pointers and prefer using smart pointers or containers like std::vector whenever possible to reduce the chances of memory leaks and errors.

Always Check for Null Pointers: Check if a pointer is nullptr before dereferencing it to avoid accessing invalid memory.

Avoid Manual Memory Management: Whenever possible, avoid manual memory management using new and delete. Use standard containers and smart pointers to manage memory automatically.

Q7:



```cpp
#include <iostream>
using namespace std;
int min(int arr[], int size) {
    int minVal = arr[0];
    for (int i = 1; i < size; i++) {
        if (arr[i] < minVal) {
            minVal = arr[i];
        }
    }
    return minVal;
}

double min(double arr[], int size) {
    double minVal = arr[0];
    for (int i = 1; i < size; i++) {
        if (arr[i] < minVal) {
            minVal = arr[i];
        }
    }
    return minVal;
}

float min(float arr[], int size) {
    float minVal = arr[0];
    for (int i = 1; i < size; i++) {
        if (arr[i] < minVal) {
            minVal = arr[i];
        }
    }
    return minVal;
}

long min(long arr[], int size) {
    long minVal = arr[0];
    for (int i = 1; i < size; i++) {
        if (arr[i] < minVal) {
            minVal = arr[i];
        }
    }
    return minVal;
}
int mian() { ... }
```