A wild pointer is a pointer in C++ that is uninitialized or has been deleted. This means that the pointer does not point to a valid memory location, and accessing or dereferencing a wild pointer can result in undefined behavior.

Here are some common scenarios where wild pointers can occur:

1. Uninitialized pointers If you declare a pointer variable without initializing it, the pointer will contain a random value that points to some memory location in the computer's memory. If you try to access or dereference this pointer, you may access memory that you should not access, which can cause your program to crash or behave unpredictably.

For example, consider the following code:

```
int *ptr;

cout << *ptr << endl;
```

In this example, `ptr` is declared but not initialized. When we try to dereference it using `*ptr`, we will get undefined behavior.

2. Deleting pointers If you delete a pointer and then try to access or dereference it, you will be accessing memory that has already been deallocated. This can cause your program to crash or behave unpredictably.

For example, consider the following code:

```
int *ptr = new int;

delete ptr;

cout << *ptr << endl;
```

In this example, we allocate memory for an integer using the `new` operator and assign the address of the allocated memory to `ptr`. We then delete the memory pointed to by `ptr` using the `delete` operator. When we try to access or dereference `ptr` using `*ptr`, we will get undefined behavior.

3. Pointers to non-existent variables If you create a pointer that points to a non-existent variable or object, you will be accessing memory that does not contain a valid object. This can cause your program to crash or behave unpredictably.

For example, consider the following code:

```
int *ptr = &x;

cout << *ptr << endl;
```

In this example, we create a pointer `ptr` that points to the address of an integer variable `x`. However, `x` has not been declared or initialized, so the pointer `ptr` points to a non-existent variable. When we try to access or dereference `ptr` using `*ptr`, we will get undefined behavior.

To avoid wild pointers in C++, you should always initialize your pointers to a valid memory location or to `nullptr` if they are not pointing to anything. You should also avoid deleting pointers that have not been allocated using the `new` operator. If you do delete a pointer, make sure to set it to `nullptr` to prevent it from becoming a wild pointer. Finally, you should always check if a pointer is pointing to a valid memory location before dereferencing it to avoid accessing a wild pointer.