Lecture 3 Types and Streams

#CS106L

- **Input/Output Streams**
- **Modern C++ Types**
- **Implementing Simpio**
- **File Streams**

NOTICE: `cin` and `cout` graphic explanation in slide for corresponding chapters

Q1:why `>>` with `cin` is a nightmare?

1. `cin` reads the entire line into the buffer but extracts whitespace-separated tokens.

2. Trash in the buffer will make cin not prompt the user for input at the right time.

3. When `cin` fails, all future `cin` operations fail too.

Q2:how to use throw in c++?

In C++, the `throw` keyword is used to throw an exception, which is a special event that indicates that an error or exceptional condition has occurred in a program. The `throw` statement is typically used in conjunction with a `try-catch` block to handle the exception.

Here is an example of how to use the `throw` keyword to throw an exception:

```
int divide(int a, int b) {

    if (b == 0)  throw runtime_error("Division by zero");

    return a / b;

}
```

In this example, the function `divide` throws a `runtime_error` exception with the message "Division by zero" when the divisor is zero.

To catch the exception, you can use a `try-catch` block, like this:

```
int main() {

    int a = 10, b = 0;

    try {

        int result = divide(a, b);

        cout << result << endl;

    } catch (const runtime_error& e) {

        cerr << e.what() << endl;

    }

    return 0;

}
```

In this example, the code in the `try` block calls the `divide` function. If the function throws an exception, the control is transferred to the `catch` block, where the exception is caught and handled. The `what()` method of the `runtime_error` class is used to get the error message.

It's important to note that when an exception is thrown, the program's flow of control is interrupted and transferred to the nearest matching catch block, if there isn't any matching catch block, the program will terminate.

Also, It's possible to throw different types of exceptions, such as built-in types (e.g. `int`, `double`, `string`, etc), standard library types (e.g. `runtime_error`, `logic_error`, `bad_alloc`, etc) and user-defined types (e.g. custom exception classes).

Q3:when to use auto in c++?

In C++, the `auto` keyword is used to automatically deduce the type of a variable from its initializer. The `auto` keyword can be used in several contexts, such as:

1. Variable Declarations: You can use `auto` to declare a variable and let the compiler deduce the type based on the initializer. For example:

```
        auto x = 5; // x is an int

        auto y = 3.14; // y is a double

        auto str = "Hello"; // str is a const char*
```

2. Iterating through containers: You can use `auto` to iterate through the elements of a container without having to explicitly specify the type of the element. For example:

```
        vector<int> v = {1, 2, 3};

        for (auto x : v) cout << x << endl;
```

3. Lambda functions: You can use `auto` to deduce the type of the return value of a lambda function. For example:

```
        auto func = [](){return 5;}; // func is a function that returns int
```

4. template functions: You can use `auto` to deduce the type of the return value of a template function. For example:

```
        template<typename T>

        auto square(T x) { return x * x; }
```

It's worth noting that the `auto` keyword can make the code more concise and readable, but it can also make it harder to understand the type of a variable, especially when the type is complex or deeply nested. It's important to use it carefully, especially when the type is not obvious.

In general, it's a good practice to use `auto` when the type of a variable is not important or when it's too complex to express explicitly. But also, it's important to be aware of the type of a variable when it's important for the code to work correctly, such as when using specific functions or operators that have different behavior for different types.

It's also worth noting that C++14 and later versions introduce `decltype(auto)` which allows the return type of a function to be deduced, it's useful when the return type depends on the input types.

There are other things to consider when using `auto` keyword, such as the difference between `auto` and `decltype`, when to use `const auto` and `auto&` and so on.

Q4:When does getline fail in c++?

he `getline()` function can fail in several cases when reading input in C++:

1. End of file (EOF) condition: When the `getline()` function reaches the end of a file that it is reading from, it will return a failure condition.
2. Error while reading the stream: If there is an error while reading the input stream, such as a read error or disk error, the `getline()` function will return a failure condition.
3. Invalid arguments: If the `getline()` function is passed invalid arguments, such as a null pointer for the input stream or a null pointer for the output buffer, it will return a failure condition.
4. Insufficient buffer space: If the output buffer passed to the `getline()` function is not large enough to hold the entire line of input, the `getline()` function will return a failure condition, and the input buffer would be truncated.

You can check the return value of `getline()` to see if it failed or not, it returns an `istream&` object and you can use it in a conditional statement. For example:

```
        string name;

        if(getline(cin, name)) {

    cout << "Your name is: " << name << endl;

        } else {

    cout << "Error reading input" << endl;

 }
```

Also you can use the `eof()` method from the input stream to check if the end of file was reached or the `fail()` method to check if the stream failed.

```
        string name;

        getline(cin, name);

        if (cin.eof()) {

    cout << "Error: End of file reached" << endl;

        } else if (cin.fail()) {

    cout << "Error: Input failed" << endl;

        } else {

    cout << "Your name is: " << name << endl;

 }
```

It's worth noting that in some cases, the `getline()` function may return a failure condition, but the input stream may not be in a failed state, for example when the buffer passed to `getline()` is too small, the getline() will return a failure condition but the stream is not failed.

In general, it's a good practice to check the return value of `getline()` and the state of the input stream, and handle any errors that may occur while reading the input. This can help to prevent any potential issues that may arise from unexpected input or errors while reading the input stream. This can include things like displaying error messages to the user, providing default values, or retrying the input operation.

Additionally, it's important to always keep in mind the format of the input you expect, and the use of appropriate input methods, as well as using appropriate error-checking methods, can help ensure that your program works as expected and can handle any unexpected input or errors that may occur.

Q5:Does the position pointer always read up to a whitespace? If not, come up with a counter example.

The position pointer will read up to a whitespace when using the `>>` operator with the `cin` object to read input from the standard input, but it doesn't always read up to a whitespace.

By default, `cin` reads input until it encounters whitespace, such as a space or a newline character, and stores it in the variable or object specified on the left side of the operator.When using the `>>` operator with the `cin` object to read input from the standard input, the position pointer will skip whitespace before the token, not after the token.

you can use the `ws` manipulator before `>>` to clear leading whitespaces, like this:

```
        string name;

        int age;

        cin >> ws >> name >> age;
```

This will clear any leading whitespaces before reading the input, this way you don't have to worry about leading whitespaces affecting the input.

However, you can use the `getline()` function to read an entire line of input, including whitespaces, and store it in a string or a character array.

For example:

```
        string name;

        getline(cin, name);
```

In this example, `getline()` reads an entire line of input, including any whitespaces, and stores it in the variable `name`.

Another example is the use of formatted input, where the `>>` operator stops reading the input when it encounters a specific delimiter. For instance, you can use the `setw()` and `setfill()` manipulators to read a fixed-width field, like this:

```
        int age;

        char gender;

        cin >> setw(2) >> age >> setw(1) >> gender;
```

In this example, the `cin` reads a 2-character field as the age and then a single character field as the gender, using the delimiter set by `setw()` manipulator.

It's worth noting that you can also use custom delimiters, like this:

```
string name;

cin >> get_delim(name, ':');
```

in this case the `cin` stops reading input when it encounters the ':' character.

In summary, the position pointer does not always read up to a whitespace and it depends on the method you use to read the input and the input format you expect.

Q6:Why does the cout operation not immediately print the output onto the console? When is the output printed?

The `cout` operation does not immediately print the output onto the console because the output is buffered. When you perform an output operation such as `cout << "Hello World"`, the data is first written to a buffer in memory, rather than being sent directly to the console. The buffer is then periodically flushed to the console, typically when it becomes full or when the program terminates.

The reason for buffering the output is to improve performance by reducing the number of writes to the console, which can be a relatively slow operation. By buffering the output, the system can write to the buffer in memory much faster, and then periodically flush the buffer to the console. This can greatly improve the performance of the program, especially when it needs to output a large amount of data.

You can use the `flush` manipulator to force the buffer to be flushed, like this:

```
cout << "Hello World" << flush;
```

You can also use the `endl` manipulator which also flush the buffer, like this:

```
cout << "Hello World" << endl;
```

Additionally, the buffer will be automatically flushed in the following cases:

- When the buffer becomes full
- When the program terminates
- When the buffer is associated with an input stream and the user enters an end-of-file condition

It's worth noting that flushing the buffer will cause a write operation that may decrease performance, but it's necessary to ensure that the data is displayed on the console immediately.