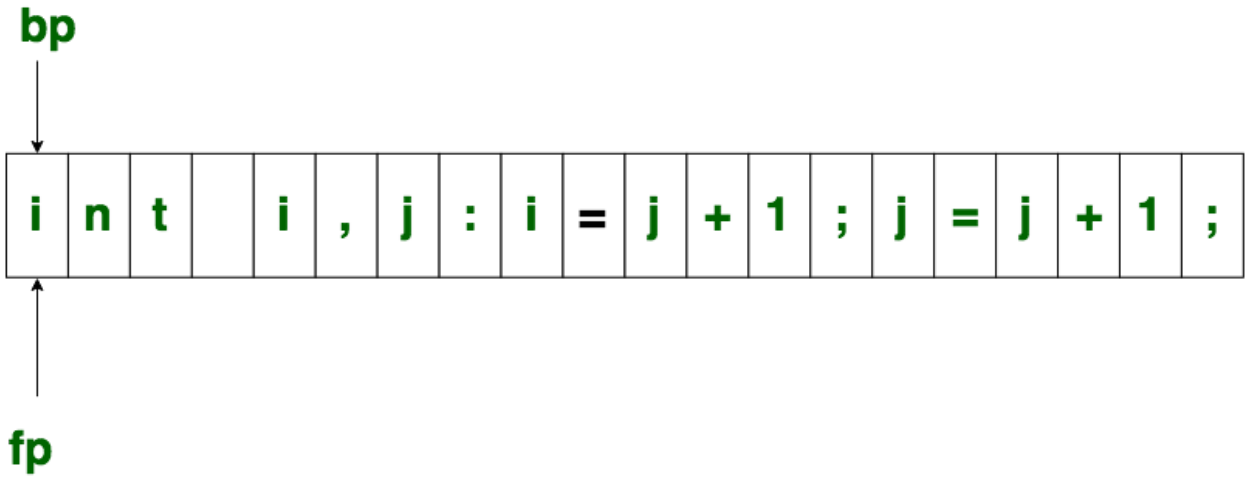
# Input Buffering in Compiler Design

Last Updated : 01 Sep, 2025

In compiler design, the lexical analyzer (or scanner) reads the source program from left to right, one character at a time, to recognize tokens. To achieve this, it uses two pointers:

- **Begin pointer (bp):** Marks the beginning of the current lexeme.
- **Forward pointer (fp):** Moves ahead to detect the end of the lexeme.




**Initial Configuration**

For example, when scanning the input int a;, both bp and fp initially point to i. The fp advances until it encounters a whitespace, indicating the end of the lexeme "int". Then, both pointers move forward to the start of the next token.

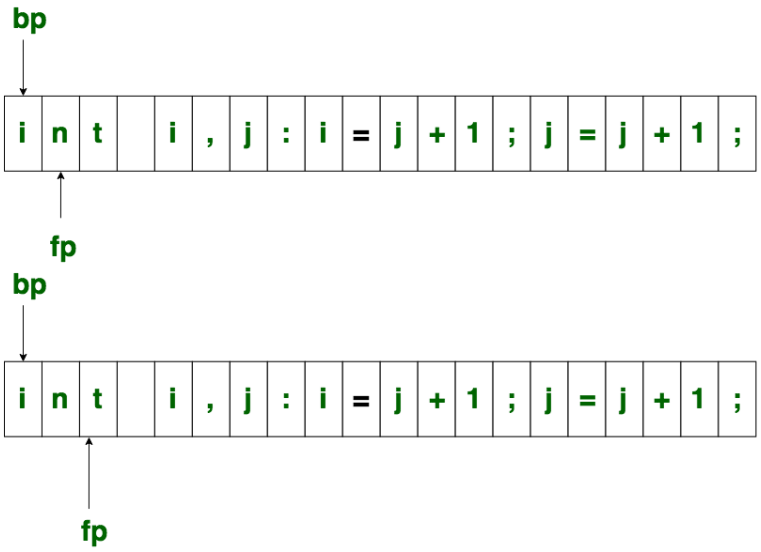


*character at a time is highly inefficient due to costly system calls. To address this, compilers use input buffering.*

## What is Input Buffering?

Input buffering is a technique where the compiler reads input in blocks (chunks) into a buffer instead of character by character from secondary storage. The lexical analyzer then processes characters from this buffer, which significantly reduces the number of system calls and improves performance.

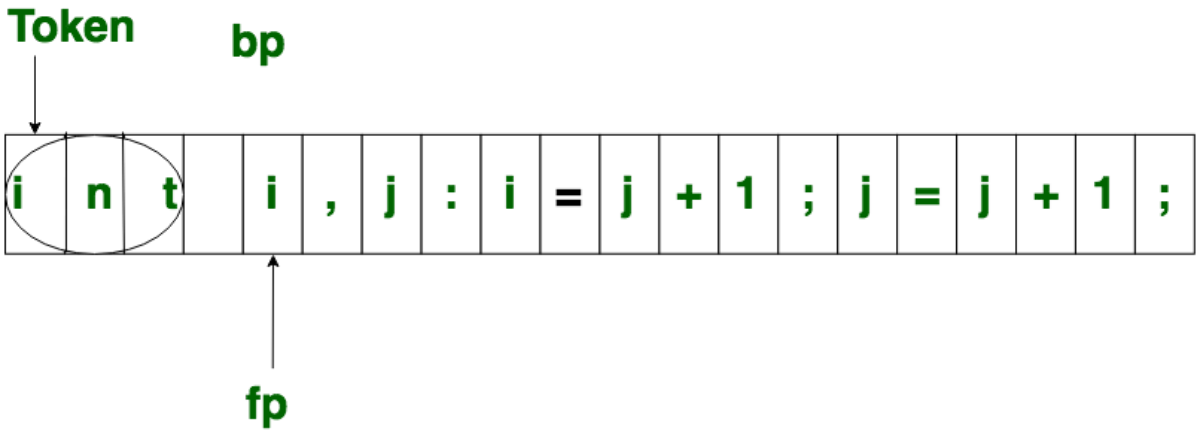
- The basic idea behind input buffering is to read a block of input from the source code into a buffer, and then process that buffer before reading the next block.
- The size of the buffer can vary depending on the specific needs of the compiler and the characteristics of the source code being compiled.
- **Example:** a compiler for a high-level programming language may use a larger buffer than a compiler for a low-level language, since high-level languages tend to have longer lines of code.




Input Buffering

- One of the main advantages of input buffering is that it can reduce the number of system calls required to read input from the source code. Since each system call carries some overhead, reducing the number of calls can improve performance. Additionally, input buffering can simplify the design of the compiler by reducing the amount of code required to manage input.
- The forward ptr moves ahead to search for end of lexeme. As soon as the blank space is encountered, it indicates end of lexeme. In above

example as soon as ptr (fp) encounters a blank space the lexeme "int" is identified.



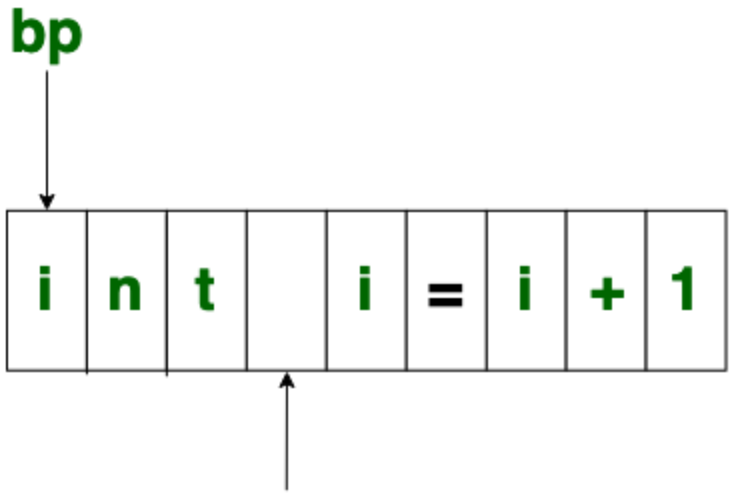

Input buffering

- The fp will be moved ahead at white space, when fp encounters white space, it ignore and moves ahead. then both the begin ptr(bp) and forward ptr(fp) are set at next token.
- The input character is thus read from secondary storage, but reading in this way from secondary storage is costly.
- hence buffering technique is used.A block of data is first read into a buffer, and then second by lexical analyzer.

## Methods of Input Buffering

There are two methods used in this context: One Buffer Scheme, and Two Buffer Scheme:

### 1. One-Buffer Scheme

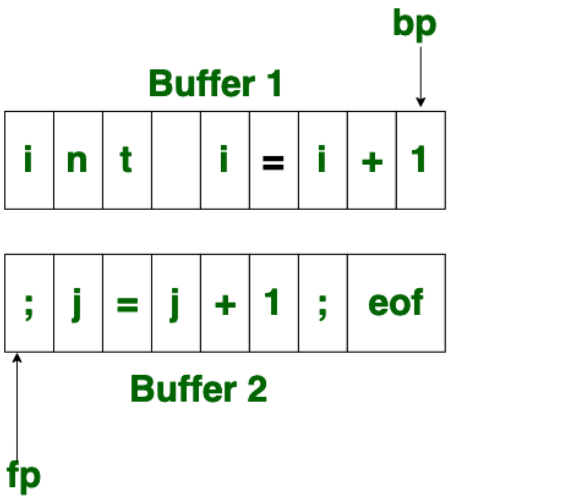



One buffer scheme storing input string

- In this scheme, a single buffer is used to store a block of input.
- The fp scans characters within this buffer until the end of the buffer is reached.
- **Problem:** If a lexeme is longer than the buffer size, part of the lexeme may cross the buffer boundary. Refilling the buffer overwrites the beginning of the lexeme, causing errors.

### 2. Two-Buffer Scheme

To overcome the limitations of the one-buffer scheme, the two-buffer scheme is used.




**Two buffer scheme storing input string**

- Two buffers of equal size are maintained.
- They are filled alternately: when the fp reaches the end of one buffer, the other buffer is refilled with the next block of input.
- At the end of each buffer, a special sentinel character (EOF marker) is placed to indicate buffer boundaries.
- Initially, both bp and fp point to the beginning of the first buffer. The fp moves ahead until a whitespace (end of lexeme) or sentinel is encountered.
- When the sentinel is reached, the analyzer switches to the other buffer. This process continues until the source program is completely scanned.
- Even in this scheme, if a lexeme is longer than the buffer size, it still cannot be scanned completely.

***Sentinel Usage:*** *The sentinel character (eof) at the end of each buffer helps the lexical analyzer detect when to switch buffers without performing repeated boundary checks. This makes scanning more efficient.*

### Advantages:

- **Reduced system calls:** Reading in large blocks lowers I/O overhead.