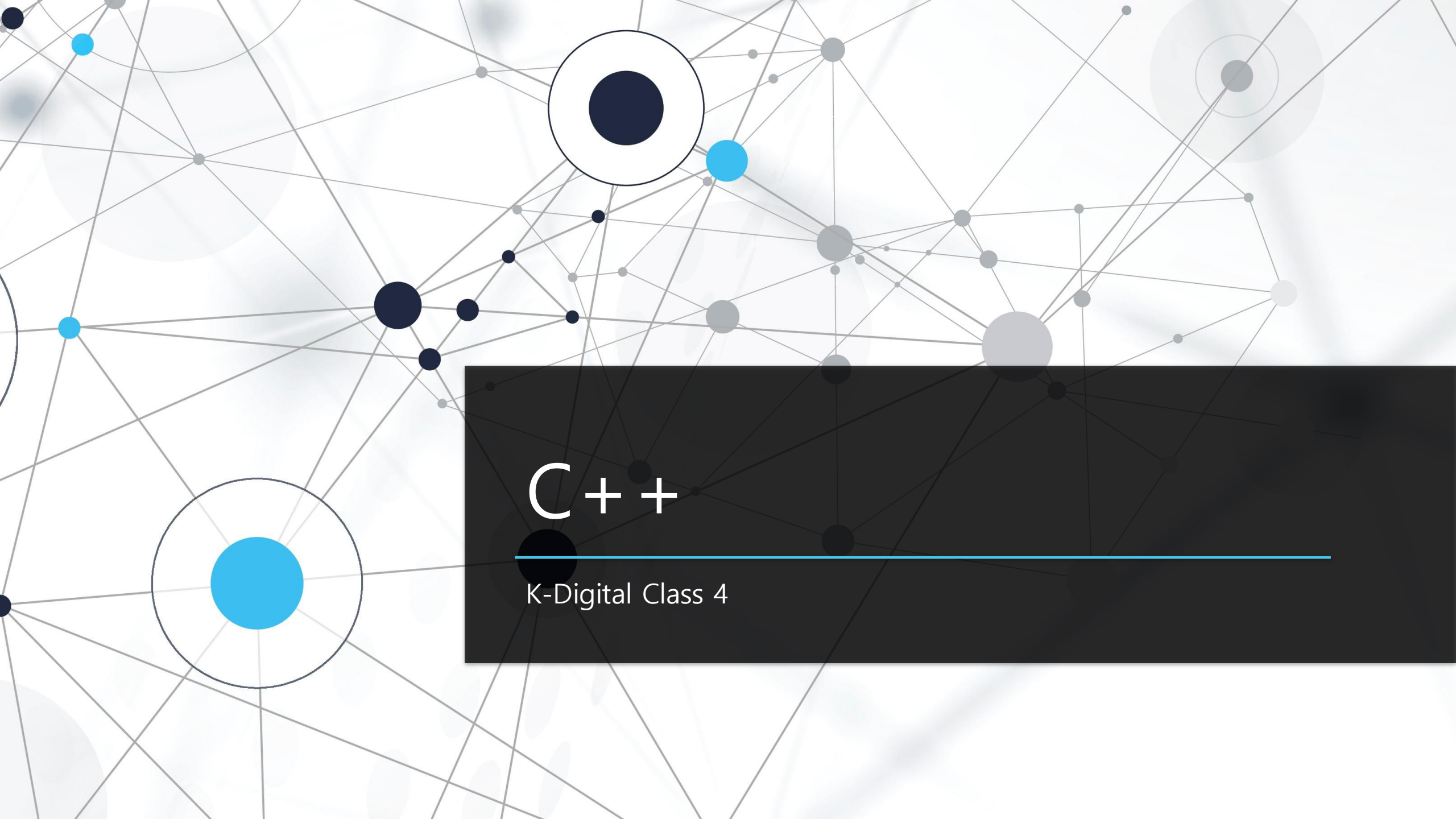


C++

K-Digital Class 4

C++ FILES AND STREAMS

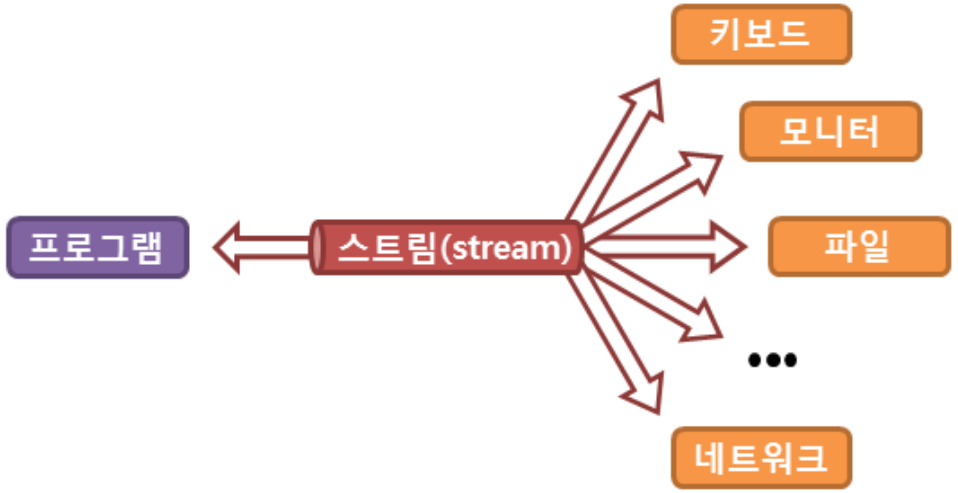
Stream and Buffer

- C++은 C언어와 마찬가지로 입출력에 관한 기능을 언어에서 기본적으로 제공하지 않는다.
- 그 이유는 컴파일러를 만들 때 입출력 기능을 해당 하드웨어에 가장 적합한 형태로 만들 수 있도록 컴파일러가 개발자에게 권한을 주기 위해서다.
- 하지만 대부분의 C++ 컴파일러는 iostream과 fstream 헤더 파일에 정의되어 있는 클래스 라이브러리를 제공한다.
- Iostream과 fstream 클래스 라이브러리의 중요 개념 중 하나가 바로 스트림(stream)이다.

C++ FILES AND STREAMS

Stream

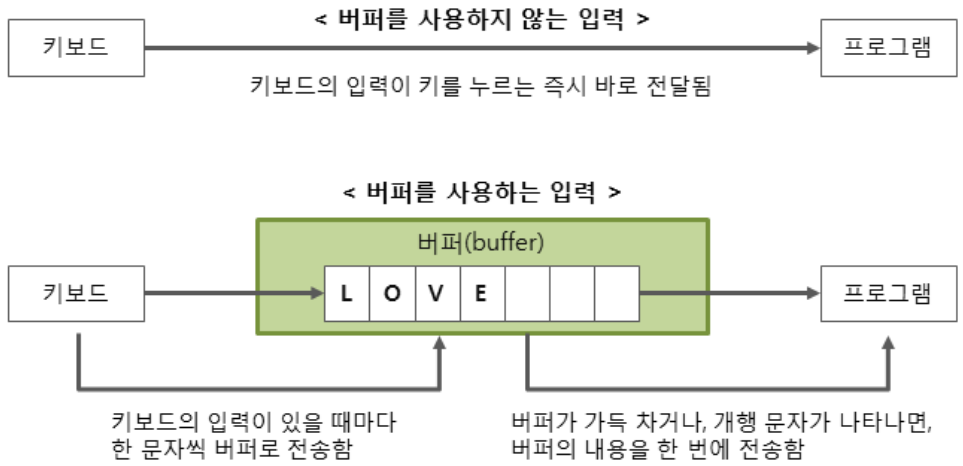
- C++ 프로그램은 파일이나 콘솔의 입출력을 직접 다루지 않고, 스트림(stream)이라는 흐름을 통해 다룬다.
- 스트림(stream)이란 실제의 입력이나 출력이 표현된 데이터의 이상 화된 흐름을 의미한다.
- 즉, 스트림은 운영체제에 의해 생성되는 가상의 연결 고리를 의미하며, 중간 매개자 역할을 한다.



C++ FILES AND STREAMS

Buffer

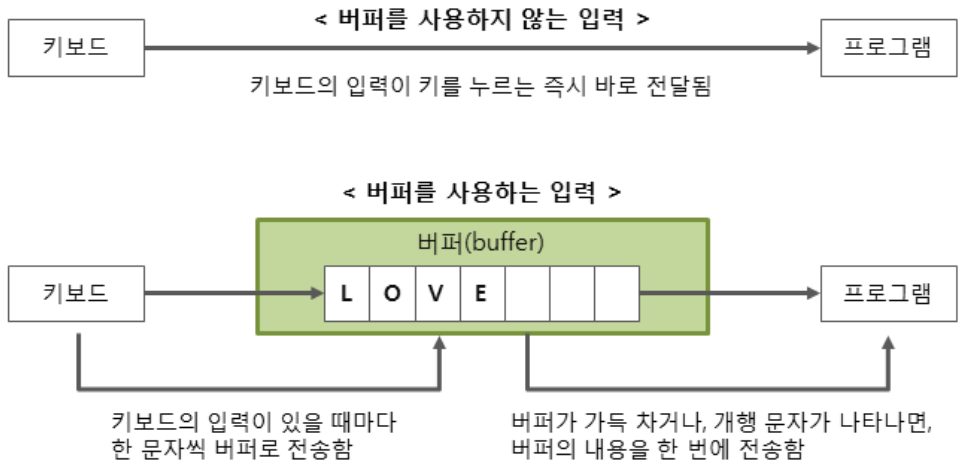
- Stream은 내부에 버퍼(buffer)라는 임시 메모리 공간을 가지고 있다.
- 이러한 버퍼를 이용하면 입력과 출력을 좀 더 효율적으로 처리할 수 있게 된다.



C++ FILES AND STREAMS

Buffer

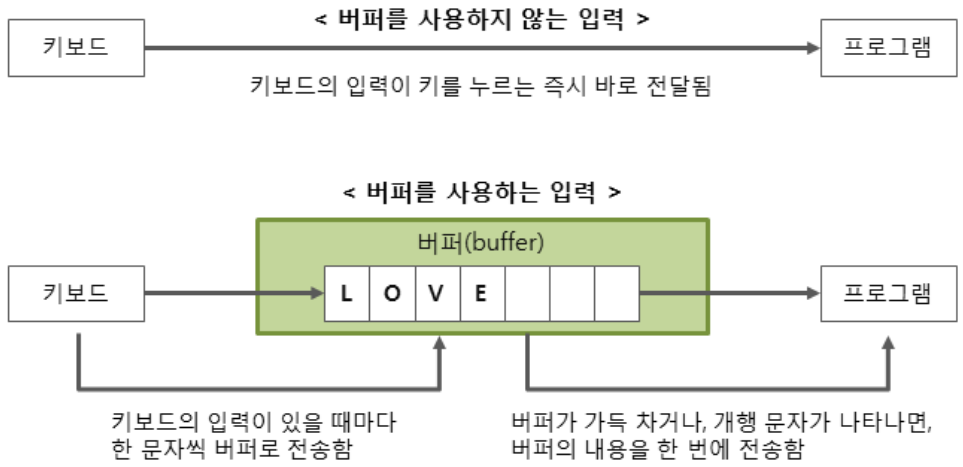
- 버퍼를 사용하면서 얻을 수 있는 장점은 다음과 같다.
  - 문자를 하나씩 전달하는 것이 아닌 묶어서 한 번에 전달하므로, 전송 시간이 적게 걸려 성능이 향상된다.
  - 사용자가 문자를 잘못 입력했을 경우 수정을 할 수가 있다.
- 하지만 입력 작업에 버퍼를 사용하는 것이 반드시 좋은 것만은 아니다.
- 빠른 반응이 요구되는 게임과 같은 프로그램에서는 키를 누르는 즉시 바로 전달 되어야 한다.
- 이렇게 버퍼를 사용하는 입력과 버퍼를 사용하지 않는 입력은 서로 다른 용도로 사용된다.



C++ FILES AND STREAMS

Buffer Synchronization(버퍼 동기화)

- 버퍼가 꽉 찼을 때, 버퍼에 있는 모든 데이터는 물리적인 매체에(만약 출력 스트림이라면) 기록되어지거나 (만약 입력 스트림이면)단순히 지워집니다. 이러한 처리를 동기화(synchronization)라고 부르고, 다음의 상황에서 일어나게 된다.
  - 파일이 닫힐 때 : 모든 버퍼는 읽기 또는 쓰기가 완전히 끝나지 않은 채 닫혀지려고 하면 그 이전에 동기화가 이루어진다.
  - 버퍼가 꽉 찼을 때 : Buffers 의 크기가 정해져 있고, buffer가 꽉 찼을 때 자동적으로 동기화가 이루어진다.
  - 조작자(manipulators)를 명시됐을 때 : 스트림을 동조 시키기 위한 명확한 조작자를 사용했을 때 이며, 이들 조작자는 flush 와 endl 이다.
  - sync() 함수를 명시했을 때 : 인수없이 sync() 멤버함수를 호출하여 즉시 동기화가 일어나게 할 수 있다. 이 함수는 스트림과 연결된 버퍼가 없거나 오류가 있는 경우에는 -1과 같은 int값을 돌려주게 된다.



C++ FILES AND STREAMS

- 지금까지 standard input(표준 입력)에서 읽고 standard output(표준 출력)으로 읽고 쓰기 위한 cin 및 cout 메서드를 각각 제공하는 iostream 표준 라이브러리(Standard Library)를 사용했다.
- 이번에는 file을 읽고(read) 쓰는(write)하는 방법을 배워보자.

Sr.No	Data Type & Description
1	**ofstream** This data type represents the output file stream and is used to create files and to write information to files.
2	**ifstream** This data type represents the input file stream and is used to read information from files.
3	**fstream** This data type represents the file stream generally, and has the capabilities of both ofstream and ifstream which means it can create files, write information to files, and read information from files.

C++ FILES AND STREAMS

- fstream 라이브러리를 사용하려면 표준 <iostream>과 <fstream> 헤더 파일을 모두 포함한다.
- fstream 라이브러리에는 파일을 생성, 쓰기 또는 읽는 데 사용되는 세 가지 클래스가 포함되어 있다.

Example

```
#include <iostream>
#include <fstream>
```

C++ FILES AND STREAMS

Opening a File

- 파일을 읽거나 쓰려면 먼저 파일을 열어(Open) 야 한다. Ofstream 또는 fstream 객체는 쓰기를 위해 파일을 여는 데 사용할 수 있다. 그리고 ifstream 객체는 읽기 전용으로 파일을 여(Open) 는 데 사용된다.
- 다음은 fstream, ifstream 및 ofstream 객체의 멤버인 open() 함수의 표준 구문이다.
- 여기서 첫 번째 매개변수는 열려는 파일의 이름과 위치(file path)를 지정하고 open() 멤버 함수의 두 번째 매개변수는 파일을 열어야 하는 모드(open option)를 정의한다.

```
void open(const char *filename, ios::openmode mode);
```

C++ FILES AND STREAMS

Opening a File

Sr.No	Mode Flag & Description
1	**ios::app** Append mode. All output to that file to be appended to the end.
2	**ios::ate** Open a file for output and move the read/write control to the end of the file.
3	**ios::in** Open a file for reading.
4	**ios::out** Open a file for writing.
5	**ios::trunc** If the file already exists, its contents will be truncated before opening the file.

C++ FILES AND STREAMS

Opening a File – C++ use often combination

파일 모드 상수	C언어 파일 모드	설명
ios_base::out | ios_base::trunc	"w"	파일을 쓰는 것만이 가능한 모드로 개방함. 파일이 없으면 새 파일을 만들고, 파일이 있으면 해당 파일의 모든 데이터를 지우고 개방함.
ios_base::out | ios_base::app	"a"	파일을 쓰는 것만이 가능한 모드로 개방함. 파일이 없으면 새 파일을 만들고, 파일이 있으면 해당 파일의 맨 끝에서부터 데이터를 추가함.
ios_base::in | ios_base::out	"r+"	파일을 읽고 쓰는 것이 둘 다 가능한 모드로 개방함.
ios_base::in | ios_base::out | ios_base::trunc	"w+"	파일을 읽고 쓰는 것이 둘 다 가능한 모드로 개방함. 파일이 없으면 새 파일을 만들고, 파일이 있으면 해당 파일의 모든 데이터를 지우고 개방함.

C++ FILES AND STREAMS

Opening a File

- 이들 값을 OR로 결합하여 둘 이상의 값을 결합할 수 있다. 예를 들어 쓰기 모드에서 파일을 열고 이미 존재하는 파일을 자르고 싶다면 구문은 다음과 같다.
- 비슷한 방법으로 다음과 같이 읽기 및 쓰기 목적으로 파일을 열 수 있다.

```
ofstream outfile;
outfile.open("file.dat", ios::out | ios::trunc );
```

```
fstream  afile;
afile.open("file.dat", ios::out | ios::in );
```

C++ FILES AND STREAMS

Closing a File

- C++ 프로그램이 종료되면 모든 스트림을 자동으로 플러시하고 할당된 모든 메모리를 해제하고 열려 있는 모든 파일을 닫는다. 그러나 프로그래머는 프로그램을 종료하기 전에 열려 있는 모든 파일을 닫아야 한다.
- 이는 아주 좋은 습관이며 불필요한 메모리 공간을 정리할 수 있다.
- 다음은 fstream, ifstream 및 ofstream 객체의 멤버인 close() 함수의 표준 구문입니다.

```
void close();
```

C++ FILES AND STREAMS

Create and Write To a File

- C++ 프로그래밍을 수행하는 동안 화면에 정보를 출력하기 위해 해당 연산자를 사용하는 것처럼 스트림 삽입 연산자(<<)를 사용하여 프로그램에서 파일에 정보를 쓴다.
- 유일한 차이점은 cout 객체 대신 ofstream 또는 fstream 객체를 사용한다는 것이다.

```
#include <iostream>
#include <fstream>
using namespace std;

int main() {
  // Create and open a text file
  ofstream MyFile("filename.txt");

  // Write to the file
  MyFile << "Files can be tricky, but it is fun enough!";

  // Close the file
  MyFile.close();
}
```

C++ FILES AND STREAMS

Read a File

- 파일에서 읽으려면 ifstream 또는 fstream 클래스와 파일 이름을 사용한다.
- 파일을 한 줄씩 읽고 파일 내용을 인쇄하기 위해 (ifstream 클래스에 속하는) getline() 함수와 함께 while 루프도 사용한다.

```
// Create a text string, which is used to output the text file
string myText;

// Read from the text file
ifstream MyReadFile("filename.txt");

// Use a while loop together with the getline() function to read the file line by line
while (getline (MyReadFile, myText)) {
  // Output the text from the file
  cout << myText;
}

// Close the file
MyReadFile.close();
```

C++ FILES AND STREAMS

Read and Write Example

- 다음은 읽기 및 쓰기 모드에서 파일을 여는 C++ 프로그램이다.
- 사용자가 입력한 정보를 afile.dat라는 파일에 쓴 후 프로그램은 파일에서 정보를 읽어 화면에 출력한다.

```
#include <fstream>
#include <iostream>
using namespace std;

int main () {
   char data[100];

   // open a file in write mode.
   ofstream outfile;
   outfile.open("afile.dat");

   cout << "Writing to the file" << endl;
   cout << "Enter your name: ";
   cin.getline(data, 100);

   // write inputted data into the file.
   outfile << data << endl;

   cout << "Enter your age: ";
   cin >> data;
   cin.ignore();

   // again write inputted data into the file.
   outfile << data << endl;

   // close the opened file.
   outfile.close();

   // open a file in read mode.
   ifstream infile;
   infile.open("afile.dat");

   cout << "Reading from the file" << endl;
   infile >> data;

   // write the data at the screen.
   cout << data << endl;

   // again read the data from the file and display it.
   infile >> data;
   cout << data << endl;

   // close the opened file.
   infile.close();

   return 0;
}
```

C++ FILES AND STREAMS

File Position Pointers

- ifstream과 ofstream은 모두 파일 위치 포인터를 재배치하기 위한 멤버 함수를 제공한다.
- 이러한 멤버 함수는 ifstream의 경우 seekg(" seek get ")이고 ofstream의 경우 seekp(" seek put ")이다.
- Seekg 및 seekp에 대한 인수는 일반적으로 long integer입니다. 탐색 방향을 나타내기 위해 두 번째 인수를 지정할 수 있다.
- 탐색 방향은 스트림의 시작 부분을 기준으로 위치 지정을 위한 ios::beg(default)일 수 있다.
- 스트림의 현재 위치를 기준으로 위치를 지정하려면 ios::cur를, 스트림 끝을 기준으로 위치를 지정하려면 ios::end를 사용한다.

```
// position to the nth byte of fileObject (assumes ios::beg)
fileObject.seekg( n );

// position n bytes forward in fileObject
fileObject.seekg( n, ios::cur );

// position n bytes back from end of fileObject
fileObject.seekg( n, ios::end );

// position at end of fileObject
fileObject.seekg( 0, ios::end );
```

C++ EXCEPTION HANDLING

- Exception(예외)는 Handler라는 특수 기능에 제어를 전달하여 프로그램의 예외적인 상황(예: 런타임 오류)에 대응하는 방법을 제공한다.
- 예외(Exception)를 잡기(catch) 위해 코드의 일부가 예외 검사를 받는다. 이는 코드의 해당 부분을 try 블록으로 묶음으로써 수행된다. 해당 블록 내에서 예외적인 상황이 발생하면 예외 처리기로 제어를 전송하는 예외가 발생한다. 예외가 발생하지 않으면 코드가 정상적으로 계속되고 모든 예외 처리기가 무시된다.
- try 블록 내부에서 throw 키워드를 사용하면 예외가 발생한다. Exception Handler는 catch 키워드로 선언되며 try 블록 바로 뒤에 위치해야 한다.

C++ EXCEPTION HANDLING

C++ try and catch

- try 문을 사용하면 실행되는 동안 오류를 테스트할 코드 블록을 정의할 수 있다.
- throw 키워드는 문제를 감지하면 예외를 throw하므로 사용자 지정 오류를 생성할 수 있다.
- catch 문을 사용하면 try 블록에서 오류가 발생할 경우 실행할 코드 블록을 정의할 수 있다.
- try 및 catch 키워드는 쌍으로 제공된다.

```
try {
  // Block of code to try
  throw exception; // Throw an exception when a problem arise
}
catch () {
  // Block of code to handle errors
}
```

C++ EXCEPTION HANDLING

Throwing Exceptions

- 예외는 throw 문을 사용하여 코드 블록 내 어디에서나 throw될 수 있다. throw 문의 피연산자는 예외의 유형을 결정하고 모든 표현식이 될 수 있으며 표현식의 결과 유형은 throw된 예외의 유형을 결정한다.
- 다음은 0으로 나누는 조건이 발생한 경우 예외를 throw하는 예이다.

```
double division(int a, int b) {
   if( b == 0 ) {
      throw "Division by zero condition!";
   }
   return (a/b);
}
```

C++ EXCEPTION HANDLING

Catching Exceptions

- try 블록 다음에 오는 catch 블록은 모든 예외를 catch한다. catch할 예외 유형을 지정할 수 있으며 이는 catch 키워드 다음에 괄호 안에 나타나는 예외 선언에 의해 결정된다.

```
try {
   // protected code
} catch( ExceptionName e ) {
  // code to handle ExceptionName exception
}
```

C++ EXCEPTION HANDLING

Handle Any Type of Exceptions (...)

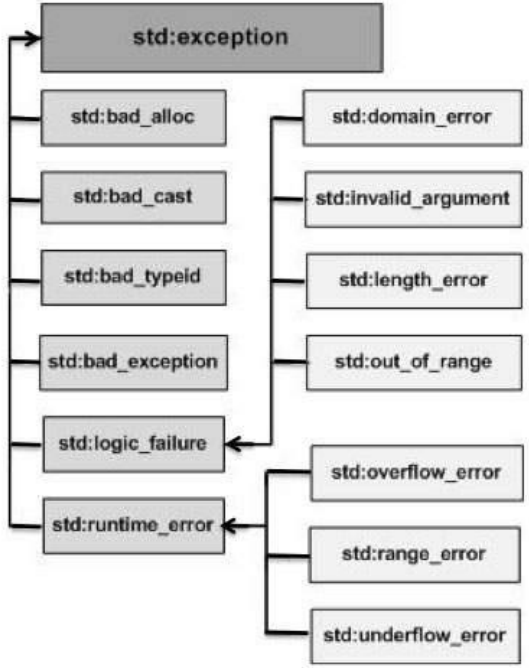
- 앞의 코드는 ExceptionName 유형의 예외를 catch합니다. catch 블록이 try 블록에서 throw된 모든 유형의 예외를 처리하도록 지정하려면 다음과 같이 예외 선언을 묶는 괄호 사이에 줄임표 ...를 넣어야 합니다.

```
try {
   // protected code
} catch(...) {
  // code to handle any exception
}
```

C++ EXCEPTION HANDLING

C++ Standard Exceptions

- C++는 우리 프로그램에서 사용할 수 있는 <exception>에 정의된 standard exception list를 제공한다. 이들은 그림에 표시된 부모-자식 클래스 계층 구조로 정렬된다.



C++ EXCEPTION HANDLING

C++ Standard Exceptions

Sr.No	Exception & Description
1	std::exception 모든 표준 C++ 예외의 예외 및 상위 클래스이다.
2	std::bad_alloc 이것은 new 에 의해 던질 수 있다
3	std::bad_cast 이것은 dynamic_cast 에 의해 던질 수 있다
4	std::bad_exception 이것은 C++ 프로그램에서 예기치 않은 예외를 처리하는 데 유용한 장치이다.
5	std::bad_typeid 이것은 typeid 에 의해 던질 수 있다.
6	std::logic_error 이론적으로 코드를 읽어 감지할 수 있는 예외이다.
7	std::domain_error 이것은 수학적으로 유효하지 않은 도메인이 사용될 때 발생하는 예외이다.

Sr.No	Exception & Description
8	std::invalid_argument 잘못된 인수로 인해 발생한다.
9	std::length_error 이것은 너무 큰 std::string이 생성될 때 발생한다.
10	std::out_of_range 이것은 'at' 메서드에 의해 발생될 수 있다(예: std::vector 및 std::bitset<>::operator[]()).
11	std::runtime_error 이론적으로 코드를 읽어서 감지할 수 없는 예외이다.
12	std::overflow_error 수학적 오버플로가 발생하면 throw된다.
13	std::range_error 범위를 벗어난 값을 저장하려고 할 때 발생한다.
14	std::underflow_error 수학적 언더플로가 발생하면 throw된다.

C++ EXCEPTION HANDLING

Define New Exceptions

- exception class 기능을 inheriting 및 overriding하여 고유한 예외를 정의할 수 있다.
- 다음은 std::exception class를 사용하여 표준 방식으로 자신의 예외를 구현하는 예제이다.

```
#include <iostream>
#include <exception>
using namespace std;

struct MyException : public exception {
    const char * what () const throw () {
        return "C++ Exception";
    }
};

int main() {
    try {
        throw MyException();
    } catch(MyException& e) {
        std::cout << "MyException caught" << std::endl;
        std::cout << e.what() << std::endl;
    } catch(std::exception& e) {
        //Other errors
    }
}
```