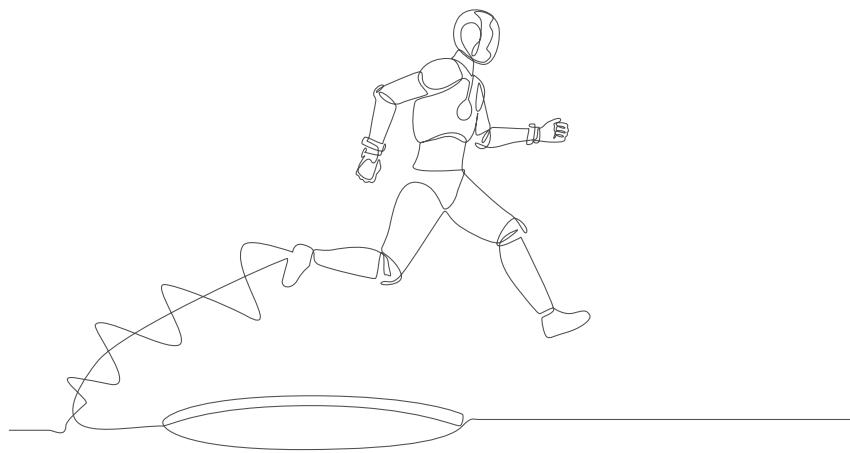


Version Control Systems & Git



Version Control Systems

SUB CHAPTER:

What is Version Control Systems



Version Control Systems (VCS)

- 버전 관리 시스템
 - 파일의 변경 이력을 체계적으로 기록하고 관리하는 시스템이다.
 - 단순히 파일의 백업 기능을 넘어, 누가 언제 어떤 내용을 수정했는지 추적하고, 특정 시점의 상태로 되돌리거나 여러 사람이 동시에 작업하는 환경에서 변경 사항을 병합하고 충돌을 해결하는 데 필수적인 도구이다.
- 버전 관리 시스템의 이점
 - 변경 이력 추적: 파일의 모든 변경 사항을 시간 순서대로 기록하여 필요할 때 특정 시점의 내용을 되돌릴 수 있음
 - 협업 용이성: 여러 명의 개발자가 동시에 하나의 프로젝트에서 작업할 때, 각자의 변경 사항을 효과적으로 병합하고 충돌을 방지하여 효율적인 협업을 지원
 - 안전한 백업: 모든 변경 이력이 저장되므로, 시스템 오류나 예기치 않은 문제 발생 시에도 이전 상태로 복구할 수 있어 데이터의 안전성을 확보
 - 브랜칭 및 병합: 프로젝트의 여러 기능을 동시에 개발하거나 실험적인 변경을 수행할 때, 메인 개발 라인에서 분리된 브랜치를 생성하여 작업하고 완료 후 메인 라인과 병합할 수 있다.
 - 감사 추적: 누가 어떤 변경을 했는지 정확하게 기록되므로, 문제 발생 시 원인을 추적하고 책임 소재를 명확히 할 수 있음



Main Version Control Systems

- 다양한 버전 관리 시스템이 존재하지만, 가장 널리 사용되는 시스템은 Git과 SVN(Subversion)이다.
- Git
 - **분산형 버전 관리 시스템 (Distributed Version Control Systems, DVCS)**: 각 개발자의 로컬 저장소에 전체 프로젝트의 이력이 복제된다. 이는 중상 서버에 장애가 발생하더라도 로컬에서 작업을 계속하고 복구가 가능하다는 장점이 있음
 - **강력한 브랜칭 및 병합 기능**: 유연하고 효율적인 브랜칭 및 병합 기능을 제공하여 복잡한 협업 워크플로우를 효과적으로 관리
 - **빠른 속도**: 로컬 저장소에서 대부분의 작업을 수행하므로 네트워크 연결 없이도 빠르게 작업을 진행할 수 있음
 - **널리 사용되는 플랫폼**: GitHub, GitLab, Bitbucket 등 다양한 플랫폼에서 Git 기반의 저장소를 제공하여 협업 및 프로젝트 관리를 용이하게 함
 - **복잡한 개념**: 처음 사용자는 Git의 다양한 개념(스테이징, 커밋, 브랜치, 머지 등)을 이해하는데 어려움을 느낄 수 있음



Main Version Control Systems

- 다양한 버전 관리 시스템이 존재하지만, 가장 널리 사용되는 시스템은 Git과 SVN(Subversion)이다.
- SVN (Subversion)
 - **중앙 집중형 버전 관리 시스템 (Centralized Version Control System)**: 모든 변경 이력이 중앙 서버에 저장. 개발자는 작업 시 중앙 서버와 통신 해야 함
 - **상대적으로 쉬운 개념**: Git에 비해 개념이 단순하여 초보자가 비교적 쉽게 익힐 수 있음
 - **파일 기반 추적**: 파일 단위로 변경 사항을 추적
 - **네트워크 의존성**: 중앙 서버에 연결되어 있어야 작업을 수행할 수 있음. 서버 장애 시 작업이 중단될 수 있음
 - **브랜칭 및 병합의 어려움**: Git에 비해 브랜칭 및 병합 기능이 상대적으로 복잡하고 충돌 발생 가능성이 높음



Usage of Version Control Systems (Based on Git)

- Git 사용의 기본적인 흐름

1. 저장소 생성 (Repository Creation) :

- 로컬에서 새로운 Git 저장소를 초기화 (`git init`)
- 원격 저장소(GitHub, GitLab 등)를 복제 (`git clone <원격 저장소 URL>`).

2. 파일 변경 (File Modification): 프로젝트 파일을 수정 (실제 코딩을 수행함을 의미)

3. 스테이징 (Staging): 변경된 파일을 커밋할 준비를 함 (`git add <파일 이름>` 또는 `git add .` => 모든 수정된 파일을 스테이징 하라는 의미)

4. 커밋(Commit) : 스테이징된 변경 사항에 대한 설명을 기록하고 로컬 저장소에 저장 (`git commit -m "커밋 메시지"`)

5. 원격 저장소 동기화 (Remote Repository Synchronization):

- 로컬 commit을 원격 저장소에 업로드 (`git push origin <브랜치 이름>`).
- 원격 저장소의 변경 사항을 로컬 저장소로 가져옴 (`git pull origin <브랜치 이름>`).



Usage of Version Control Systems (Based on Git)

- Git 사용의 기본적인 흐름

6. 브랜치 관리 (Branch Management) :

- 새로운 브랜치를 생성 (`git branch <브랜치 이름>`).
- 브랜치를 전환 (`git checkout <브랜치 이름>`).
- 브랜치를 병합 (`git merge <병합할 브랜치 이름>`).



Conclusion

- 버전 관리 시스템은 소프트웨어 개발 뿐만 아니라 텍스트 파일, 이미지 등 다양한 종류의 파일 변경 이력을 관리하는 데 필수적인 도구임
- Git은 현재 가장 널리 사용되는 분산형 버전 관리시스템으로, 효율적인 협업과 안정적인 프로젝트 관리 지원
- Git은 가장 널리 사용되는 분산형 버전 관리 시스템으로, 효율적인 협업과 안정적인 프로젝트 관리를 지원
- 프로젝트의 규모와 협업 환경에 따라 적절한 버전 관리 시스템을 선택하고 사용하는 것은 개발 생산성을 향상시키는 중요한 요소

SUB CHAPTER:

A Deep Dive into Git Staging



What is the Staging Step

- **스테이징(Staging)**은 커밋(Commit)을 만들기 전에 변경 사항을 준비하는 단계를 의미
 - 사진 촬영 전에 원하는 사람과 배경을 골라 프레임에 넣는 것과 비슷
 - 세부 설명
 - **작업 디렉토리 (Working Directory)**: 실제로 파일을 편집하고 작업하는 공간
 - **스테이징 영역 (Staging Area 또는 Index)**: 커밋할 파일과 변경 사항을 임시로 보관하는 곳으로, 다음에 commit에 포함할 사항들을 선택적으로 담을 수 있도록 해 줌
 - **로컬 저장소 (Local Repository)**: 실제로 commit들이 영구적으로 기록되는 곳



What is the Staging Step

- 스테이징(staging)의 중요한 역할:
 - **선별적인 commit**: 작업 디렉토리에서 여러 파일을 수정했더라도, 스테이징 영역을 통해 특정 파일 또는 특정 변경 사항만 골라서 커밋할 수 있도록 지원
 - 예를 들어, 버그 수정과 새로운 기능 개발을 동시에 작업했을 때, 버그 수정 관련 파일만 먼저 커밋하고 나중에 새로운 기능 관련 파일을 커밋할 수 있음
 - **의도 명확화**: 스테이징 영역에 추가된 파일들은 “다음 commit에 포함될 변경 사항”이라는 명확한 의도를 나타냄. 이를 통해 commit 메시지와 실제 변경 사항을 더 일관성 있게 유지할 수 있음
 - **변경 사항 미리보기**: 스테이징된 내용은 git status 명령어나 Git GUI 도구를 통해 쉽게 확인할 수 있음. 이를 통해 실수로 원치 않는 변경 사항이 commit되는 것을 방지할 수 있음
 - **커밋 메시지 작성 용이성**: 스테이징된 변경 사항들을 보면서 해당 commit에 대한 명확하고 의미 있는 [commit](#) 메시지를 작성하는데 도움을 받을 수 있음



Key Git commands related to staging

- `git add <파일 이름>` : 특정 파일을 스테이징 영역에 추가
- `git add .` 또는 `git add *` : 현재 디렉토리와 그 하위 디렉토리의 모든 변경된 파일을 스테이징 영역에 추가
- `git rm -cached <파일 이름>` : 스테이징 영역에서 특정 파일을 제거 (작업 디렉토리의 파일은 유지).
- `git reset HEAD <파일 이름>` : 스테이징 된 특정 파일의 변경 사항을 스테이징 영역에서 제거하고 unstaged 상태로 되돌림
- `git status` : 작업 디렉토리와 스테이징 영역의 상태를 보여줌. 어떤 파일이 변경되었고, 스테이징 되었는지 등을 확인할 수 있음
- `git diff -staged` : 스테이징 영역에 있는 변경 사항들을 보여줌 (마지막 커밋과 비교)
- 정리
 - Git의 스테이징은 **작업한 내용을 커밋이라는 “기록”으로 남기기 전에, 어떤 변경 사항을 포함할지 미리 선택하고 준비하는 중간 단계**에 해당한다. 이 단계를 통해 보다 체계적이고 의미 있는 commit을 만들 수 있으며, 협업 환경에서도 변경 사항을 명확하게 공유할 수 있음

SUB CHAPTER:

Practice



Create A Project

- 간단한 스택을 구현하는 프로그램을 만든 후 Git으로 관리하는 실습을 해보자.
- 먼저 프로젝트 “Stack”을 생성한 후 다음과 같이 두 가지 파일을 만든다.
 - Stack.hpp: 스택을 정의한 헤더 파일
 - main.cpp: 스택을 사용하는 파일

Implementation: Stack.hpp (1/2)

```
1 #ifndef _STACK_
2 #define _STACK_
3
4 #include <iostream>
5 #include <exception>
6
7 using namespace std;
8
9 template <typename T>
10 class Stack
11 {
12 private:
13     T* data;
14     int size;
15     int tp;
16 public:
17     Stack(int = 10);
18     ~Stack();
19
20     T top() const;
21     void push(T);
22     void pop();
23
24
25
        bool empty() const;
        bool full() const;
        void increase();
    };
    template <typename T>
    Stack<T>::Stack(int size)
        :data(new T[size]), size(size), tp(-1)
    {}
    template <typename T>
    Stack<T>::~Stack()
    {
        delete[] data;
    }
    template <typename T>
    T Stack<T>::top() const
    {
        return data[tp];
    }
```

Implementation: Stack.hpp (2/2)

```
1 template <typename T>
2 void Stack<T>::push(T v)
3 {
4     if (full())
5         increase();
6     data[++tp] = v;
7 }
8 template <typename T>
9 void Stack<T>::pop()
10 {
11     if (empty())
12         throw std::exception("Stack is empty.");
13     --tp;
14 }
15 template <typename T>
16 bool Stack<T>::empty() const
17 {
18     return tp == -1;
19 }
20
21
22
23
24
25
```

```
template <typename T>
bool Stack<T>::full() const
{
    return tp == size;
}

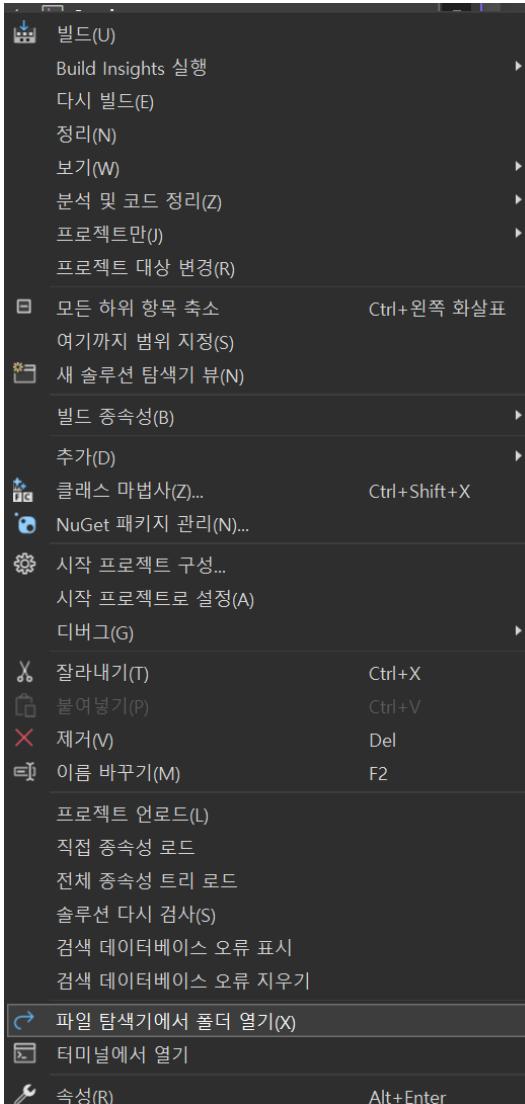
template <typename T>
void Stack<T>::increase()
{
    T* tmp = new T[size * 2];
    memmove(data, tmp, size * sizeof(T));
    size *= 2;
    delete[] data;
    data = tmp;
}
#endif
```

Implementation: main.cpp

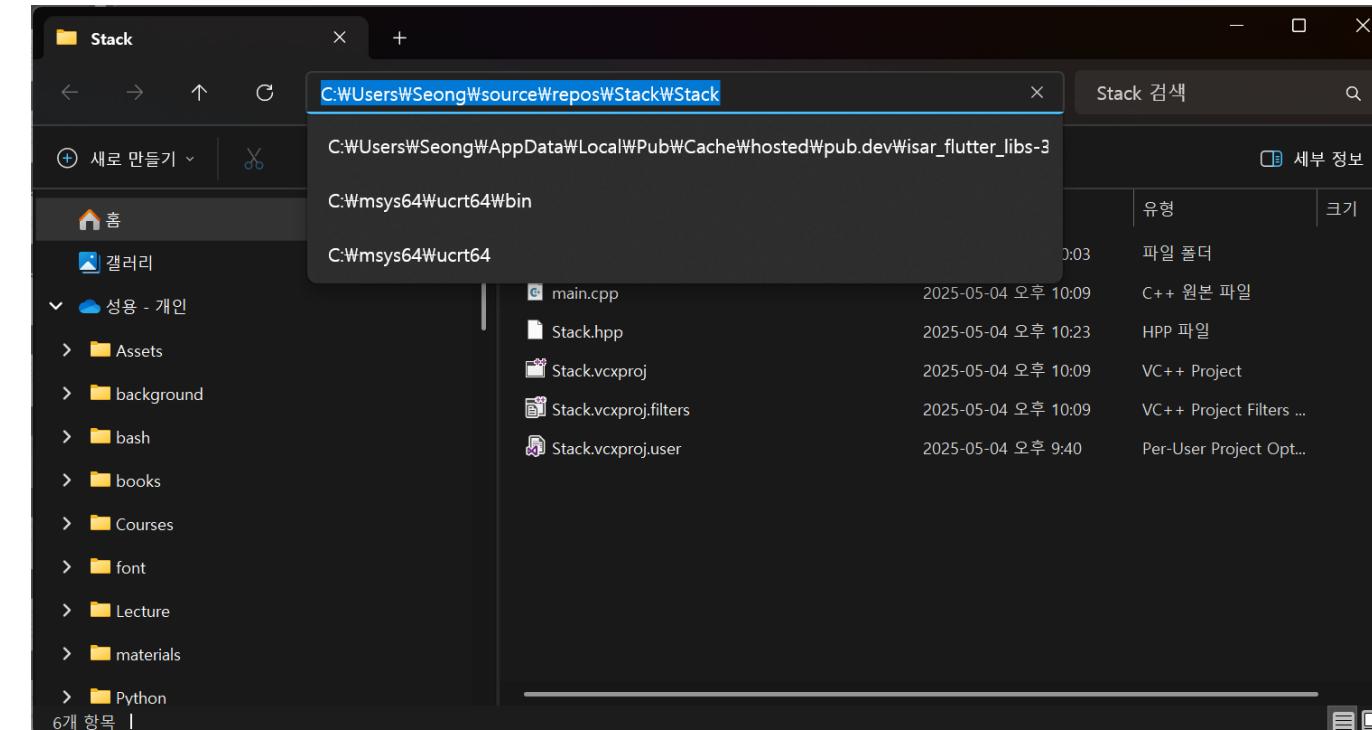
```
1 #include "Stack.hpp"
2 #include <iostream>
3
4 using namespace std;
5
6 int main()
7 {
8     Stack<int> stack;
9
10    stack.push(1);
11    stack.push(2);
12    stack.push(3);
13
14    while (!stack.empty())
15    {
16        cout << stack.top() << endl;
17        stack.pop();
18    }
19
20    return 0;
21 }
22
23
24
25
```



Initialize Git (1/3)



- 프로젝트 이름을 선택하고, 마우스 오른쪽 버튼을 누른 후, 팝업 창에서 “파일 탐색기에 서 폴더 열기”를 선택
- 작업 디렉토리 위치를 복사한다.
 - 예) C:\Users\Seong\source\repos\Stack\Stack





Initialize Git (2/3)

- 복사해둔 디렉토리로 이동한다.
 - 예) cd C:\Users\Seong\source\repos\Stack\Stack

A screenshot of a Windows Command Prompt window. The title bar says "명령 프롬프트". The window shows the command prompt line: "C:\Users\Seong\source\repos\Stack\Stack>".



Initialize Git (3/3)

- 디렉터리로 이동했다면, 다음 명령을 입력해서 git을 초기화한다.

- 예) > git init
- 위 명령어를 입력하면, git을 사용할 수 있도록 초기화 된다.
- 제대로 설치되었는지 확인하기 위해서 다음 명령을 입력해보자.
 - > dir /a
- 그러면, .git이라는 숨김 디렉토리가 만들어졌는지 확인해보자.

```
C:\Users\Seong\source/repos\Stack\Stack>git init
Initialized empty Git repository in C:/Users/Seong/source/repos/Stack/Stack/.git/
C:\Users\Seong\source/repos\Stack\Stack>dir /a
C 드라이브의 볼륨에는 이름이 없습니다.
볼륨 일련 번호 : CEED-3F65

C:\Users\Seong\source\repos\Stack\Stack 디렉터리

2025-05-04 오후 11:03 <DIR> .
2025-05-04 오후 10:21 <DIR> ..
2025-05-04 오후 11:03 <DIR> .git
2025-05-04 오후 10:09 255 main.cpp
2025-05-04 오후 10:23 1,153 Stack.hpp
2025-05-04 오후 10:09 6,688 Stack.vcxproj
2025-05-04 오후 10:09 1,106 Stack.vcxproj.filters
2025-05-04 오후 09:40 168 Stack.vcxproj.user
```



Check .git directory

- 다음 명령어를 입력해서, .git 디렉터리를 확인해보자.

```
> cd .git
```

```
> dir
```

```
명령 프롬프트
C:\Users\Seong\source\repos\Stack\Stack\.git>dir
C 드라이브의 볼륨에는 이름이 없습니다.
볼륨 일련 번호 : CEED-3F65

C:\Users\Seong\source\repos\Stack\Stack\.git 디렉터리

2025-05-04 오후 11:03 <DIR> ..
2025-05-04 오후 11:03 130 config
2025-05-04 오후 11:03 73 description
2025-05-04 오후 11:03 23 HEAD
2025-05-04 오후 11:03 <DIR> hooks
2025-05-04 오후 11:03 <DIR> info
2025-05-04 오후 11:03 <DIR> objects
2025-05-04 오후 11:03 <DIR> refs
            3개 파일           226 바이트
            5개 디렉터리   202,061,955,072 바이트 남음

C:\Users\Seong\source\repos\Stack\Stack\.git>
```



Create .gitignore

- VS Community에서 작업하다보면, 컴파일 시 많은 파일들이 만들어지며, 특히 실행 파일 등을 서버에 업로드 하다 보면 불필요한 공간을 많이 차지하게 된다. 이 경우 불필요한 디렉토리들은 서버에 저장할 필요가 없다.
- 이를 위해서 매번 특정 파일을 지정하면 번거롭기 때문에 git에서 관리하지 않을 파일들을 .gitignore 파일에 작성해두면 .git은 이 파일들을 제외하고 관리하게 된다.
- 먼저 다음 명령어를 입력하여 .gitignore 파일을 만든다.

```
> echo. > .gitignore
```

- 파일이 만들어졌는지 확인해보자. 루트 디렉토리에서 다음과 같이 입력해본다.

```
> dir /a
```



Create .gitignore

- ‘.’로 시작하는 파일이나 디렉토리들은 숨김 파일이나 디렉토리이기 때문에, 그냥은 보이지 않는다.
- 숨김 파일을 보기 위해서는 `dir /a`와 같이 숨김 파일을 볼 수 있는 옵션과 함께 사용된다.

명령 프롬프트

볼륨 일련 번호 : CEED-3F65

C:\Users\Seong\source\repos\Stack\Stack 디렉터리

날짜	시간	타입	파일/디렉터리	크기
2025-05-04	오후 11:32	<DIR>	.	
2025-05-04	오후 10:21	<DIR>	..	
2025-05-04	오후 11:03	<DIR>	.git	
2025-05-04	오후 11:32		27 .gitignore	
2025-05-04	오후 10:09		255 main.cpp	
2025-05-04	오후 10:23		1,153 Stack.hpp	
2025-05-04	오후 10:09		6,688 Stack.vcxproj	
2025-05-04	오후 10:09		1,106 Stack.vcxproj.filters	
2025-05-04	오후 09:40		168 Stack.vcxproj.user	
2025-05-04	오후 10:03	<DIR>	x64	
		6개 파일	9,397 바이트	
		4개 디렉터리	202,060,144,640 바이트 남음	

C:\Users\Seong\source\repos\Stack\Stack>



Create .gitignore

- .gitignore 파일은 다음과 같은 패턴으로 작성한다.

```
# 특정 파일 무시  
*.log  
*.tmp  
  
# 특정 폴더 무시  
/node_modules/  
/build/  
  
# 특정 파일 제외 (추적 유지)  
!important.txt
```

- 특정 파일 제외는 제외 폴더에 있는 파일 중 git에 포함할 파일이 있다면, !important.txt와 같이 작성하면, 이 파일은 git에 의해서 관리된다.
- 여기서는 “/x64” 디렉토리를 git에서 관리하지 않도록 설정해보자.



Create .gitignore

- .gitignore 파일을 연 다음, 다음과 같이 입력하자.

```
# 무시할 폴더  
/x64/
```

- 규칙적용

- 파일을 작성했으면, .gitignore 파일을 Git에 적용하기 위해서 다음 명령을 실행한다.

```
> git add .gitignore
```

```
> git commit -m "Add .gitignore file"
```

The screenshot shows a terminal window with the following command history:

```
명령 프롬프트          설정  
2025-05-04 오후 10:23      1,153 Stack.hpp  
2025-05-04 오후 10:09      6,688 Stack.vcxproj  
2025-05-04 오후 10:09      1,106 Stack.vcxproj.filters  
2025-05-04 오후 09:40      168 Stack.vcxproj.user  
2025-05-04 오후 10:03      <DIR> x64  
                           6개 파일      9,397 바이트  
                           4개 디렉터리  202,060,144,640 바이트 남음  
C:\Users\Seong\source\repos\Stack\Stack>code .ignore  
C:\Users\Seong\source\repos\Stack\Stack>git add .gitignore  
C:\Users\Seong\source\repos\Stack\Stack>git commit -m "Add .gitignore file"  
[master (root-commit) e36acff] Add .gitignore file  
 1 file changed, 1 insertion(+)  
  create mode 100644 .gitignore  
C:\Users\Seong\source\repos\Stack\Stack>
```



Check current status

- 현재까지의 git 상태를 확인하기 위해서 다음 명령을 실행해보자.

> git status

```
명령 프롬프트
설정

1 file changed, 1 insertion(+)
create mode 100644 .gitignore

C:\Users\Seong\source\repos\Stack\Stack>git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)
    .ignore
    Stack.hpp
    Stack.vcxproj
    Stack.vcxproj.filters
    Stack.vcxproj.user
    main.cpp
    x64/

nothing added to commit but untracked files present (use "git add" to track)

C:\Users\Seong\source\repos\Stack\Stack>
```



Check current status

- 확인 가능한 정보
 - 추적되지 않은 파일 (Untracked files)
 - 스테이징된 파일 (Changes to be committed)
 - 수정되었지만 스테이징되지 않은 파일 (Changes not staged for commit)
 - 현재 브랜치 정보(On branch <브랜치명>)
 - 리모트 저장소와 동기화 상태 (예: Your branch is up to date with ‘original/main’.)
- 현재는 어떤 파일도 추가하지 않은 상태이므로, Untracked files만 보인다.



Add all files

- 이제 작성한 파일들을 모두 Git에 추가해보자. 다음 명령을 실행해보자.

```
> git add .
```

- ‘.’은 모든 파일들을 스테이징(staging) 상태로 변경하라는 의미이다.
- 그리고, 다시 상태를 확인해보자.

```
명령 프롬프트
main.cpp
nothing added to commit but untracked files present (use "git add" to track)
C:\Users\Seong\source\repos\Stack\Stack>git add .
C:\Users\Seong\source\repos\Stack\Stack>git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    new file:   Stack.hpp
    new file:   Stack.vcxproj
    new file:   Stack.vcxproj.filters
    new file:   Stack.vcxproj.user
    new file:   main.cpp

C:\Users\Seong\source\repos\Stack\Stack>
```



- 스테이징(staging) 파일을 commit 하기 위해서, 다음 명령을 실행해보자.

```
> git commit -m "First commit all files."
```

The screenshot shows a Windows terminal window with a dark theme. The title bar says "명령 프롬프트". The command "git commit -m \"First commit all files.\" " was run, resulting in the following output:

```
(use "git restore --staged <file>..." to unstage)
new file: Stack.hpp
new file: Stack.vcxproj
new file: Stack.vcxproj.filters
new file: Stack.vcxproj.user
new file: main.cpp

C:\Users\Seong\source\repos\Stack\Stack>git commit -m "First commit all files."
[master f223b44] First commit all files.
 5 files changed, 276 insertions(+)
 create mode 100644 Stack.hpp
 create mode 100644 Stack.vcxproj
 create mode 100644 Stack.vcxproj.filters
 create mode 100644 Stack.vcxproj.user
 create mode 100644 main.cpp

C:\Users\Seong\source\repos\Stack\Stack>
```



- 현재 상태를 확인해보자.

> git status

```
명령 프롬프트  설정
new file: Stack.vcxproj.user
new file: main.cpp

C:\Users\Seong\source\repos\Stack\Stack>git commit -m "First commit all files."
[master f223b44] First commit all files.
 5 files changed, 276 insertions(+)
 create mode 100644 Stack.hpp
 create mode 100644 Stack.vcxproj
 create mode 100644 Stack.vcxproj.filters
 create mode 100644 Stack.vcxproj.user
 create mode 100644 main.cpp

C:\Users\Seong\source\repos\Stack\Stack>git status
On branch master
nothing to commit, working tree clean

C:\Users\Seong\source\repos\Stack\Stack>
```

- 상태를 보면 스테이징(Staging 상태)에 있던 모든 파일이 commit이 되어서, 작업 중이던 트리가 정리되었음을 확인할 수 있다.



Modifying file

- 프로그램을 수정한 후 다시 git의 상태를 확인해보자.
 - main.cpp을 다음 페이지와 같이 수정해보자.

Modifying main.cpp

```
1 int main()
2 {
3     Stack<int> stack;
4
5     stack.push(1);
6     stack.push(2);
7     stack.push(3);
8
9     while (!stack.empty())
10    {
11         cout << stack.top() << endl;
12         stack.pop();
13     }
14
15
16
17
18
19
20
21
22
23
24
25
```

```
stack.push(rand() % 100);

while (!stack.empty())
{
    cout << stack.top() << endl;
    stack.pop();
}

return 0;
```



Check current status

- 상태를 확인해보면, commit할 stage에 변경된 사항이 없음을 보여줌
 - 그렇지만, `main.cpp` 파일이 수정되었음을 보여준다.

```
명령 프롬프트
설정
x + v - □ ×

create mode 100644 Stack.vcxproj.filters
create mode 100644 Stack.vcxproj.user
create mode 100644 main.cpp

C:\Users\Seong\source\repos\Stack\Stack>git status
On branch master
nothing to commit, working tree clean

C:\Users\Seong\source\repos\Stack\Stack>git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
    modified:   main.cpp

no changes added to commit (use "git add" and/or "git commit -a")

C:\Users\Seong\source\repos\Stack\Stack>
```



Commit the modified file

- 수정한 파일을 스테이지에 추가한 후, 커밋을 수행해보자.

```
> git add main.cpp
```

```
> git commit -m "Add functions to main.cpp"
```

또는

```
> git commit -a -m "Add functions to main.cpp."
```

또는

```
> git commit -am "Add functions to main.cpp."
```

```
C:\Users\Seong\source\repos\Stack\Stack>git reset --hard HEAD^
More?
More? No
fatal: ambiguous argument 'HEAD'
No': unknown revision or path not in the working tree.
Use '--' to separate paths from revisions, like this:
'git <command> [<revision>...] -- [<file>...]'


C:\Users\Seong\source\repos\Stack\Stack>git reset --hard HEAD~1
HEAD is now at f223b44 First commit all files.


C:\Users\Seong\source\repos\Stack\Stack>git commit -a -m "Add functions to main.cpp"
.
On branch master
nothing to commit, working tree clean

C:\Users\Seong\source\repos\Stack\Stack>
```



Check commit history

- git의 commit 이력을 확인하기 위해서 다음 명령을 실행해보자.

> git log

```
명령 프롬프트
commit f223b4472f193434215c146d9ac374a827f3128f (HEAD -> master)
Author: 성용 주 <SeongYongJ@gmail.com>
Date: Mon May 5 00:32:15 2025 +0900

    First commit all files.

commit 1f1643597420339eb3d7cb3e4d98e6bede3653ba
Author: 성용 주 <SeongYongJ@gmail.com>
Date: Mon May 5 00:25:40 2025 +0900

    Add .gitignore file

commit e36acffbd86a4840c35445ea1e1f9f5333d02777
Author: 성용 주 <SeongYongJ@gmail.com>
Date: Mon May 5 00:03:19 2025 +0900

    Add .gitignore file

C:\Users\Seong\source\repos\Stack\Stack>
```

SUB CHAPTER:

Restore



git restore

- 작업 디렉토리에서 변경 사항을 되돌리는 명령어
- 파일을 삭제하는 것이 아니라, 이전 커밋 상태로 복구
- 스테이징된 변경 사항을 취소하거나, 특정 파일을 원래 상태로 되돌릴 수 있음
- 사용 예제:

```
git restore <파일명> # 작업 디렉토리에서 변경 사항 취소
```

```
git restore --staged <파일명> # 스테이징 영역에서 변경 취소
```



Cancelling changes in the working directory and in the staging area

- 작업 디렉토리(Working directory)와 스테이징 영역(Staging area) 내 변경 사항을 취소하는 것의 차이점
 - 작업 디렉토리에서 변경 사항 취소 (`git restore <파일명>`)
 - 로컬에서 수정한 파일을 원래 상태로 되돌림
 - 이전 커밋 상태로 복구하지만, 스테이징 영역에는 영향을 주지 않음
 - 예) `git restore index.html` # `index.html` 파일을 마지막 `commit` 상태로 복구
 - 즉, 이전 `commit` 상태로 되돌리고 싶을 때 사용
 - 스테이징 영역에서 변경 사항 취소 (`git restore --staged <파일명>`)
 - 스테이징된 파일을 취소하고 작업 디렉토리 상태로 되돌림
 - 즉, `git add`로 스테이징 상태로 등록한 파일을 다시 추적되지 않은 상태(untracked files)로 되돌림
 - 예) `git restore --staged index.html` # `index.html`을 스테이징 제거
 - 파일 자체는 그대로 유지됨. 즉, Git에서 추적하지 않을 뿐 파일의 내용은 그대로 유지됨



Cancelling changes in the working directory and in the staging area

- 차이점 정리

명령어	역할	파일 변경 여부	Git 축적 여부
git restore <파일명>	작업 디렉토리 변경 취소	변경됨	추적 유지
git restore --staged <파일명>	스테이징 영역에서 취소	변경 없음	추적 안함

- 즉, 작업 디렉토리에서 취소하면 파일이 원래 상태로 돌아가고, 스테이징 영역에서 취소하면 Git에서 추적하지 않도록 되돌리는 차이가 있음



git rm

- Git에서 파일을 추적하지 않도록 제거하는 명령어
- git commit을 실행하면 해당 파일은 저장소에서 완전히 삭제됨
- 작업 디렉토리에서 해당 파일도 제거됨
- 사용 예제:

```
git rm <파일명> # 파일 삭제 후 Git에서 추적 제거
```

```
git rm -r <폴더명> # 폴더 전체 삭제
```

```
git rm -cached <파일명> # 파일을 Git에서 제거하지만 로컬에 유지
```



git rm

- 차이점 정리

명령어	역할	파일 삭제 여부	Git 추적 상태
git restore	변경 사항 복구	아니요	유지
git rm	파일 제거	삭제됨	추적 안함

- 즉, `git restore`은 파일을 원래 상태로 되돌리는 역할을 하고, `git rm`은 Git에서 완전히 제거하는 역할을 함

SUB CHAPTER:

Revert to a previous version



Revert to a previous version

- `git reset`과 `git revert`는 Git에서 변경 사항을 되돌리는 데 사용되는 명령어이지만, 작동 방식과 목적이 다름
- `git reset`
 - `git reset`은 commit 기록 자체를 변경하는 명령어
 - 특정 commit 이전의 상태로 되돌리고, 필요에 따라 작업 디렉토리와 스테이징 영역까지 해당 상태로 되돌릴 수 있음
 - 주의해서 사용해야 하며, 특히 이미 원격 저장소에 푸시(push)한 commit에 대해서는 사용을 자양해야 함
 - 협업하는 팀원들의 로컬 저장소와 기록이 불일치하게 되어 혼란을 줄 수 있음
 - `git reset` 명령어는 세 가지 주요 옵션
 - `--soft` : (기본 옵션, 생략 가능)
 - HEAD 포인터를 지정된 commit으로 이동
 - 스테이징 영역과 작업 디렉토리는 변경하지 않고 그대로 유지
 - 이전 commit 이후의 변경 사항은 “Staged changes” 상태로 남아있어 다시 commit할 수 있음
 - 용도: 마지막 commit 메시지를 수정하거나, 여러 개의 commit을 하나의 commit으로 합치고 싶을 때 유용



Revert to a previous version

- `git reset` 명령어는 세 가지 주요 옵션
 - `--soft` : (기본 옵션, 생략 가능)
> `git reset --soft <commit 해시 또는 참조>`
 - `--mixed` : (명시적으로 사용)
 - HEAD 포인터를 지정된 commit으로 이동
 - 스테이징 영역의 변경 사항을 제거. 즉, 이전 commit 이후의 변경 사항은 “Changes not staged for commit” 상태가 됨
 - 작업 디렉토리의 파일은 그대로 유지
 - 용도: 마지막 commit의 내용을 되돌리고 싶지만, 변경 사항은 유지하여 다시 선택적으로 스테이징하고 commit하고 싶을 때 유용
> `git reset --mixed <커밋 해시 또는 참조>`



Revert to a previous version

- `git reset` 명령어는 세 가지 주요 옵션
 - `--hard` : (매우 중의해서 사용해야 함)
 - HEAD 포인터를 지정된 커밋으로 이동
 - 스테이징 영역과 작업 디렉토리의 모든 변경 사항을 **완전히 삭제**. 이전 commit 이후의 모든 작업 내용이 사라지므로 데이터 손실의 위험이 큼.
 - **용도**: 로컬 저장소를 특정 commit 시점으로 완전히 되돌리고 싶을 때 사용하지만, **작업 내용을 잃을 수 있으므로 신중하게 결정해야 함**

```
> git reset --hard <commit 해시 또는 참조>
```



Revert to a previous version

- `git revert`:
 - 새로운 commit을 생성하여 특정 commit의 변경 사항을 “취소”하는 명령어
 - 이전 commit의 내용을 되돌리는 새로운 commit이 만들어지므로, **기존 commit 기록은 그대로 유지됨**
 - 따라서, 이미 원격 저장소에 푸시(push)한 채 commit에 대해서도 안전하게 사용할 수 있으며, 협업 환경에서 변경 사항을 되돌리는 일반적인 방법
 - `git revert <commit 해시 또는 참조>` 명령어를 실행하면 해당 commit의 변경 사항을 취소하는 새로운 commit을 생성하고 commit 메시지 편집기가 열림. 기본적으로 revert된 commit에 대한 정보가 포함된 메시지가 생성되며, 필요에 따라 수정하고 저장하면 revert commit이 생성
 - `git revert --no-commit <commit 해시 또는 참조>` 옵션을 사용하면 revert에 필요한 변경 사항만 스테이징 영역에 추가하고 commit은 생성하지 않음. 이를 통해 revert된 변경 사항을 추가적으로 수정하거나 다른 변경 사항과 함께 하나의 commit으로 만들 수 있음



Revert to a previous version

- `git revert`
 - `git revert`의 장점:
 - **기존 커밋 기록 유지:** commit 히스토리를 변경하지 않아 협업하는 팀원들에게 혼란을 주지 않음
 - **안전한 되돌리기:** 이미 푸시된 commit에 대해서도 안전하게 변경 사항을 되돌릴 수 있음
 - **명확한 의도:** “이 commit의 변경 사항을 되돌린다”는 명확한 의도를 새로운 commit으로 기록
 - `git revert`의 단점:
 - **commit 히스토리 증가:** 변경 사항을 되돌릴 때마다 새로운 commit이 생성되어 히스토리가 길어질 수 있음



Revert to a previous version

- 요약 및 사용 시점

기능	git reset	git revert
커밋 기록 변경	변경함	변경하지 않고 새로운 commit 생성
안정성	주의 필요 (특히 푸시된 commit에 대해)	비교적 안전 (푸시된 commit에도 사용 가능)
작업 방식	특정 시점으로 되돌림	특정 commit의 변경 사항을 취소하는 새 commit 생성
주요 용도	로컬에서 최근 commit 수정/병합, 되돌리기	원격 저장소에 푸시된 commit의 변경 사항 되돌리기



When to use

- `git reset`:
 - **로컬에서만 작업한 commit**을 되돌리거나 수정하고 싶을 때 (아직 푸시하지 않은 경우)
 - 최근의 잘못된 커밋을 완전히 없애고 싶을 때 (--hard 사용 시 주의!).
 - 스테이징된 내용을 되돌리고 싶을 때 (--mixed).
 - 마지막 커밋 메시지를 수정하고 싶을 때 (--soft).
- `git revert`:
 - **이미 원격 저장소에 푸시한 commit**의 변경 사항을 되돌리고 싶을 때
 - commit 히스토리를 유지하면서 특정 변경 사항을 취소하고 싶을 때
 - 협업하는 팀원들과 변경 사항 되돌리기를 명확하게 공유하고 싶을 때
- 결론적으로, 협업 환경에서는 `git revert`를 사용하여 변경 사항을 되돌리는 것이 일반적이고 안전한 방법이다.
`git reset`은 주로 로컬에서 개인적으로 작업하는 동안 실수를 수정하거나 히스토리를 정리하는 용도로 신중하게 사용해야 함

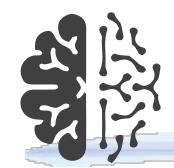
SUB CHAPTER:

Branch



What is branch?

- 브랜치(Branch)는 **독립된 작업 공간**임
 - 프로젝트를 수행할 때 여러 목적으로 개발 라인을 분리해서 진행할 수 있음
 - 특정 목적에 따라 테스트 코드 등을 삽입할 수 있는데 메인 브랜치에서 작업하면 협업자에게 영향을 줄 수 있으며, 코드 개발이나 수정에 실패할 경우 원래대로 돌리지 못할 우려가 있음
 - 메인 개발 라인(보통 main 또는 master 브랜치)에서 분리되어 특정 기능 개발, 버그 수정, 실험적인 시도 등을 다른 사람의 작업에 영향을 주지 않고 진행할 수 있도록 해줌
 - 각 브랜치는 커밋의 역사를 복사하여 만들어지며, 해당 브랜치에서의 변경 사항은 다른 브랜치에 격리됨. 따라서 여러 작업을 동시에 진행하고, 완료된 작업만 메인 라인에 합칠 수 있음
 - **브랜치의 핵심적인 특징:**
 - **독립적인 개발:** 각 브랜치에서 코드를 수정하고 커밋하는 것은 다른 브랜チ에는 영향을 미치지 않음
 - **병렬 작업:** 여러 명의 개발자가 동시에 다른 기능 개발을 위해 각각의 브랜치에서 작업할 수 있음
 - **안전한 실험:** 새로운 아이디어나 위험한 변경을 시도할 때, 메인 브랜치에 영향을 주지 않고 별도의 브랜치에서 실험할 수 있음



What is branch?

- **브랜치의 핵심적인 특징:**
 - **체계적인 기능 관리:** 각 기능 개발 단위를 브랜치로 관리하여 작업 단위를 명확하게 구분하고 추적할 수 있음
 - **코드 리뷰 용이성:** 개발이 완료된 기능 브랜치를 메인 브랜치에 합치기 전에 코드 리뷰를 통해 코드 품질을 향상시킬 수 있음



Why are branches necessary?

- Git 브랜치는 소프트웨어 개발 워크플로우를 효율적이고 안전하게 관리하는 데 필수적인 기능임. 브랜치가 없다면 다음과 같은 문제가 발생할 수 있음
 - **동시 작업의 어려움**: 여러 개발자가 동시에 하나의 코드베이스를 수정하면 충돌이 빈번하게 발생하고, 코드의 안전성을 유지하기 어려워짐
 - **실험적인 시도의 위험성**: 검증되지 않은 코드를 바로 메인 라인에 적용하면 오류 발생 가능성이 높아지고, 안정적인 버전 관리가 어려워짐
 - **기능별 관리의 어려움**: 여러 기능 개발을 동시에 진행할 때, 각 기능의 변경 사항을 추적하고 관리하기가 매우 복잡해짐
 - **코드 리뷰의 어려움**: 변경 사항을 논리적인 단위로 분리하기 어려워 코드 리뷰 과정이 비효율적이고 오류를 놓칠 가능성이 커짐
 - **버전 관리의 혼란**: 특정 시점의 안정적인 버전을 유지하고 관리하기 어려워 룰백이나 문제 해결이 복잡해짐



Why are branches necessary?

- Git 브랜치 사용의 주요 이점:
 - **협업 효율성의 증대**: 여러 개발자가 독립적인 공간에서 동시에 작업하고, 완료된 작업만 안전하게 통합할 수 있어 협업 효율성을 크게 향상
 - **코드 안전성 확보**: 실험적인 기능 개발이나 위험한 변경을 메인 브랜치와 분리하여 진행함으로써 코드의 안정성을 유지할 수 있음
 - **체계적인 기능 개발**: 각 기능별로 브랜치를 생성하여 개발 단위를 명확하게 관리하고 추적할 수 있음
 - **효율적인 코드 리뷰**: 기능별 브랜치를 기반으로 코드 변경 사항을 논리적으로 검토하고 피드백을 주고 받아 코드 품질을 향상시킬 수 있음
 - **유연한 버전 관리**: 특정 시점의 코드를 브랜치로 분리하여 유지하고 관리함으로써 필요에 따라 이전 버전으로 쉽게 롤백하거나 특정 기능만 배포할 수 있는 유연성을 제공



Usage of Git Branch

- 브랜치 확인
 - 현재 로컬 저장소에 있는 브랜치 목록을 확인하고, 현재 활성화된 브랜치를 알 수 있음

```
usage > git branch
```
 - 활성화된 브랜치 이름 앞에는 * 표시가 붙음
- 브랜치 생성
 - 새로운 작업을 시작하기 전에 메인 브랜치(일반적으로 `main` 또는 `master`)에서 새로운 브랜치를 생성

```
usage > git branch <새로운 브랜치 이름>
```
 - 예를 들어, “`feature-login`”이라는 새로운 기능 개발을 위한 브랜치를 생성하려면 다음과 같이 입력

```
usage > git branch feature-login
```



Usage of Git Branch

- 브랜치 만들고 이동하기
 - “show-type” 브랜치를 만들기 위해서 다음과 같이 수행

```
> git branch show-type
```

- show-type 브랜치로 이동하기

```
> git checkout show-type
```

```
명령 프롬프트
* e36acff Add .gitignore file
C:\Users\Seong\source\repos\Stack\Stack>git branch
* master

C:\Users\Seong\source\repos\Stack\Stack>git branch show-type

C:\Users\Seong\source\repos\Stack\Stack>git branch
* master
  show-type

C:\Users\Seong\source\repos\Stack\Stack>git checkout show-type
Switched to branch 'show-type'

C:\Users\Seong\source\repos\Stack\Stack>git branch
  master
* show-type

C:\Users\Seong\source\repos\Stack\Stack>
```



Usage of Git Brach

- 브랜치 생성과 이동 동시 수행

```
usage > git checkout -b show-type
```

- “show-type” 브랜치로 이동했다면, 코드를 수정하고, 다시 git에 add하고 commit 해보자.
- 코드 수정

- Stack.hpp의 Stack 클래스에 다음과 같은 함수를 선언하고, 함수를 구현해보자.

- 함수 선언

```
string type() const;
```

- 함수 구현

```
template <typename T>
```

```
string Stack<T>::type() const
```

```
{
```

```
    return string( typeid(*this).name() );
```

```
}
```



Usage of Git Branch

- 코드 수정 후 add와 commit 동시 수행

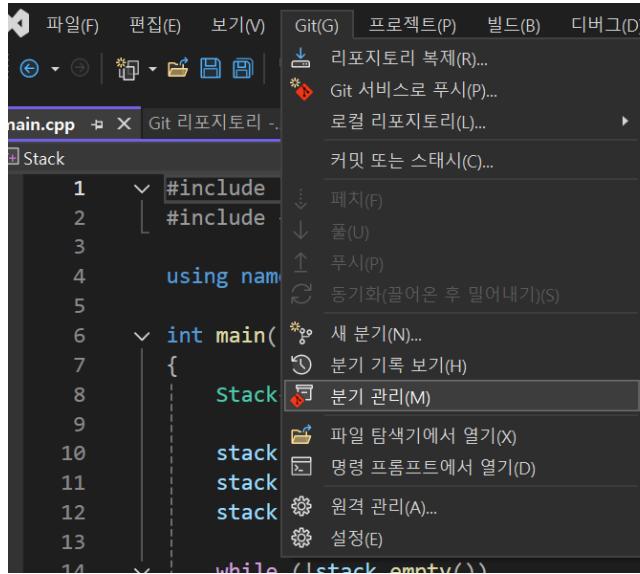
```
> git commit -am "Add type()"
```

- 위 명령은 수정된 모든 파일을 스테이징에 추가한 후, 곧 바로 commit 한다.
- 현재 “show-type” 브랜치에서 코드를 수정했기 때문에, “master” 브랜치에서 파일을 열어 보면 코드가 수정되지 않았음을 알 수 있다.
- VS Community를 실행후 “master” 브랜치에서 파일을 확인해보자.



Usage of Git Branch

- VS Community에서 git을 실행해보자.



- “git > 분기 관리”를 선택한다.
- 만일 이 메뉴가 보이지 않는다면, 먼저 “로컬 저장소”를 열어야 한다.



Usage of Git Brach

- 다음 창의 왼쪽 리스트에서 “master”를 두 번 클릭해보자.
- 이 동작은 “master” 브랜치로 이동한다.

The screenshot shows the Microsoft Visual Studio Code interface with the Git History panel open. The title bar indicates the project is 'Stack' and the file 'main.cpp' is open. The Git History panel shows a commit history for the 'master' branch. The commits are:

ID	날짜	만든 사람	내용
b04fa6b7	2025-05-05 오전 3:23:19	성용 주	Add pushes to code.
f223b447	2025-05-05 오전 12:32:15	성용 주	First commit all files.
1f164359	2025-05-05 오전 12:25:40	성용 주	Add .gitignore file
e36acffb	2025-05-05 오전 12:03:19	성용 주	Add .gitignore file

- 파일이 열려 있다면, 모두 닫은 후 다시 열어보면, 수정한 내용이 없다는 것을 알 수 있다.



Usage of Git Branch

- 브랜치 병합 (Merge):
 - 개발이 완료된 브랜치의 변경 사항을 메인 브랜치 또는 다른 브랜치로 합치는 과정
 - 이 작업은 하위 브랜치에서 master 브랜치의 내용으로 수정할 때도 사용
 - 과정
 - 병합을 브랜치로 이동: 예를 들면, master 또는 show-type

```
> git checkout main
```

- 병합 실행: merge 명령어를 사용하여 특정 브랜치의 변경 사항을 현재 브랜치로 병합

```
usage > git merge <병합할 브랜치 이름>
```

- 예 > git merge show-type
 - 이 명령은 main 브랜치에서 show-type 브랜치와 합침
 - 두 브랜치를 합칠 때는 show-type 브랜치에 새로 추가된 내용만 합침



Usage of Git Branch

- 브랜치 병합 (Merge):

- 병합 충돌 해결: 만일 동일한 파일을 master 브랜치와 show-type 브랜치 둘 다 수정했다면 충돌이 발생. 이 경우 Git은 충돌이 발생한 부분을 표시해주며, 개발자는 직접 파일을 수정하여 충돌을 해결한 후 다시 스테이징하고 커밋해야 함
- 충돌을 발생시키기 위해서 “master” 브랜치의 main.cpp 파일의 main 함수에 아래와 같이 코드를 추가해보자.
 - 추가할 코드: `cout << "Solving Merge Conflict.\n";`
- 이제 “master” 브랜치에서 main.cpp를 add하고, commit 해보자.

```
C:\Users\Seong\source\repos\Stack\Stack>git checkout master
M      main.cpp
Already on 'master'

C:\Users\Seong\source\repos\Stack\Stack>git add main.cpp

C:\Users\Seong\source\repos\Stack\Stack>git commit -m "commit main.cpp to conflic w
ith show-type branch."
[master c73009e] commit main.cpp to conflic with show-type branch.
 1 file changed, 2 insertions(+)

C:\Users\Seong\source\repos\Stack\Stack>git merge show-type
Auto-merging main.cpp
CONFLICT (content): Merge conflict in main.cpp
Automatic merge failed; fix conflicts and then commit the result.
```



Usage of Git Branch

- 브랜치 병합 (Merge):

- 앞의 그림은 충돌이 발생했음을 보여준다.
- 충돌 내용을 살펴보기 위해서 다음과 같이 명령어를 실행한다.

```
> git diff main.cpp
```

```
C:\Users\Seong\source\repos\Stack\Stack>git diff main.cpp
diff --cc main.cpp
index e9dc5eb,c8e321e..0000000
--- a/main.cpp
+++ b/main.cpp
@@@ -33,7 -33,7 +33,11 @@ int main(
                      stack.pop();
}

++<<<<< HEAD
+     cout << "Solving Merge Confliction.\n";
+=====
+     cout << stack.type() << endl;
+>>>>> show-type

                     return 0;
}
```



Usage of Git Branch

- 브랜치 병합 (Merge):
 - VS Community에서 main.cpp를 열어보면, 다음과 같이 충돌이 표시되어 있다.

```
36 | <<<<< HEAD
37 |     cout << "Solving Merge Conflict.\n";
38 | =====
39 |     cout << stack.type() << endl;
40 | >>>>> show-type
41 |
```

- 위 코드를 수정하고 다시 add와 commit을 실행하고, 정상적으로 commit이 수행되면 다시 merge를 해보자.

```
명령 프롬프트
+
+     cout << "Solving Merge Conflict.\n";
+=====
+     cout << stack.type() << endl;
++>>>>> show-type

        return 0;
    }

C:\Users\Seong\source\repos\Stack\Stack>git commit -am "Solving conflict"
[master 00e7422] Solving conflict

C:\Users\Seong\source\repos\Stack\Stack>git merge show-type
Already up to date.

C:\Users\Seong\source\repos\Stack\Stack>git branch
* master
  show-type

C:\Users\Seong\source\repos\Stack\Stack>S|
```



Usage of Git Branch

- 브랜치 삭제

- 더 이상 필요 없는 브랜치는 삭제하여 저장소를 깔끔하게 유지할 수 있음

- 로컬 브랜치 삭제: 병합이 완료된 로컬 브랜치를 삭제

- 방법

```
> git branch -d <삭제할 브랜치 이름>
```

- 병합되지 않은 브랜치를 강제로 삭제하려면 **-D** 옵션을 사용 (주의해서 사용).

- 방법

```
> git branch -D <삭제할 브랜치 이름>
```

SUB CHAPTER:

Remote Repository



Remote Repository

- 원격 저장소(**remote repository**)는 인터넷이나 네트워크 상에 위치한 Git 저장소를 의미
- 로컬 저장소(사용중인 컴퓨터에 있는 Git 저장소)와는 달리, 원격 저장소는 주로 다음과 같은 목적으로 사용
 - 코드 공유 및 협업: 여러 개발자가 동일한 프로젝트에서 작업할 때, 변경 사항을 공유하고 통합하는 중앙 집중적인 역할을 함. 각 개발자는 로컬에서 작업한 내용을 원격 저장소에 푸시(push)하고, 다른 개발자의 변경 사항을 풀(pull)하여 동기화
 - 코드 백업 및 버전 관리: 로컬 저장소의 모든 변경 이력을 원격 저장소에 백업하여 데이터 손실 위험을 줄이고, 프로젝트의 모든 버전을 안전하게 관리
 - 배포 및 CI(지속적 통합, Continuous Integration)/CD(지속적 배포, Continuous Deployment) 파이프라인: 원격 저장소는 빌드, 테스트, 배포 등의 자동화된 프로세스(CI/CD)의 트리거 역할을 하기도 함
- 주요 특징:
 - 접근 가능성: 네트워크 연결을 통해 여러 사람이 접근할 수 있음
 - 중앙 허브: 협업의 중심 역할을 하며, 변경 사항을 주고받는 기준점이 됨
 - 다양한 형태: GitHub, GitLab, Bitbucket과 같은 클라우드 기반 서비스나, 자체적으로 구축한 Git 서버 형태로 존재할 수 있음



- 원격 저장소 관련 주요 Git 명령어:
 - `git remote`: 등록된 원격 저장소 목록 확인. `-v` 옵션을 사용하면 URL도 함께 표시
 - `git remote add <이름> <url>`: 새로운 원격 저장소를 등록. 관례적으로 원격 저장소 이름을 `origin`으로 등록
 - `git remote remove <이름>`: 등록된 원격 저장소를 제거
 - `git remote rename <기존 이름> <새로운 이름>`: 등록된 원격 저장소의 이름을 변경
 - `git fetch <이름>`: 원격 저장소의 변경 사항을 로컬 저장소로 가져오지만, 현재 작업중인 브랜치에 자동으로 병합하지는 않음. 가져온 변경 사항은 `origin/브랜치` 이름 형태로 확인할 수 있음
 - `git pull <이름> <브랜치 이름>`: 원격 저장소의 특정 브랜치 변경 사항을 로컬 저장소로 가져와 현재 브랜치에 병합. 이 명령은 `git fetch`와 `git merge`를 합친 명령어이다.
 - `git push <이름> <브랜치 이름>`: 로컬 저장소의 특정 브랜치 커밋을 원격 저장소의 해당 브랜치로 업로드. `-u` 옵션을 처음 사용할 때 지정하면 로컬 브랜치와 원격 브랜치를 연결하여 이후에는 브랜치 이름을 생략하고 `git push`만으로 푸시할 수 있음



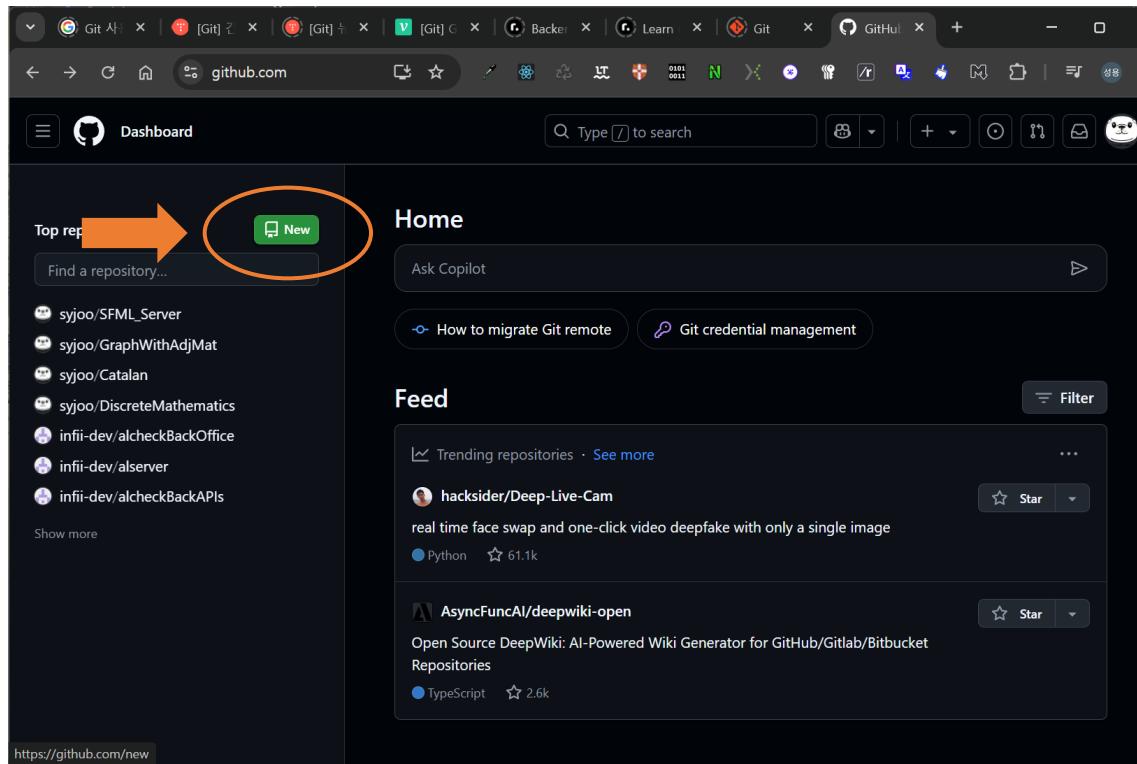
Remote Repository

- 원격 저장소의 필요성:
 - 협업: 여러 개발자가 효율적으로 코드를 공유하고 함께 작업할 수 있도록 지원
 - 백업: 로컬 환경에서 발생할 수 있는 데이터 손실 위험으로부터 프로젝트 보호
 - 버전 관리의 중앙 집중화: 모든 변경 이력을 한 곳에서 관리하여 프로젝트의 안전성과 추적성을 높임



The process of using a remote repository

- 선행 조건
 - GitHub이나 GitLab 등 원격 저장소에 가입 후 새 프로젝트를 만듦

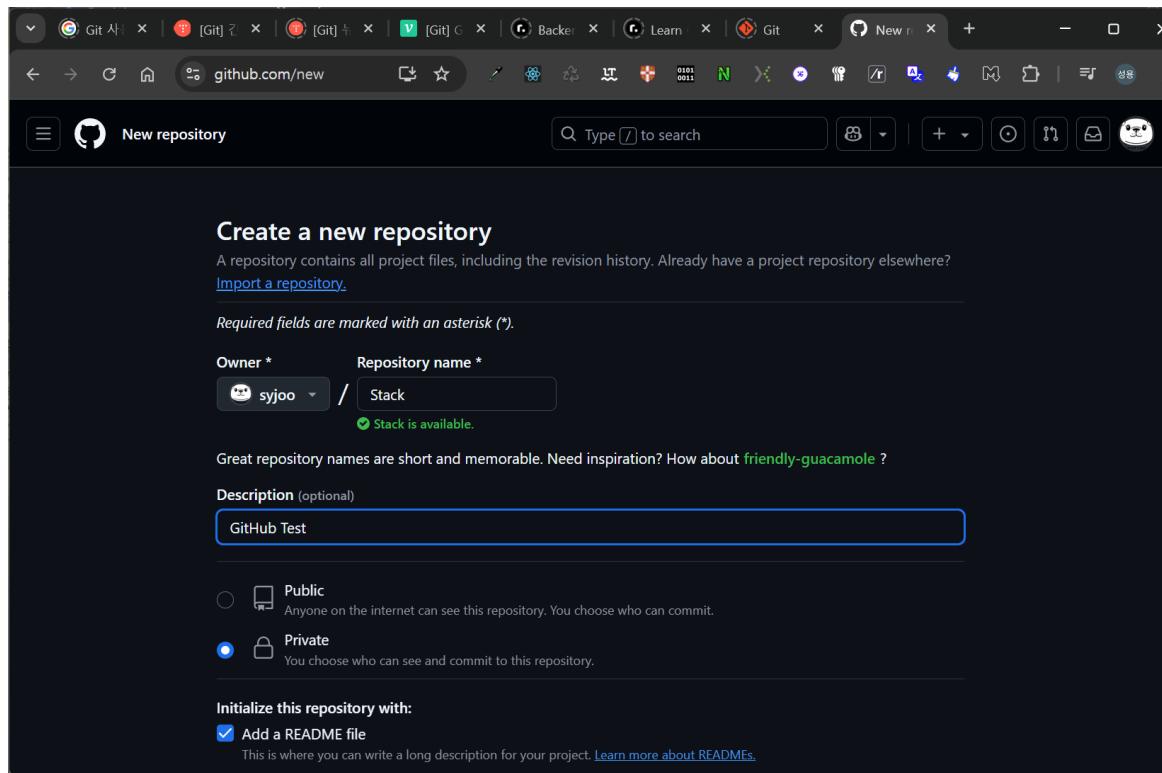


- 왼쪽의 New 버튼을 눌러서 새로운 프로젝트를 만든다.



The process of using a remote repository

- 프로젝트 생성을 위한 항목을 채우고 하단의 “Create repository” 버튼을 누름



- Repository 생성을 위해서 채워야 할 항목
 - 1) Repository name: Stack
 - 2) Description: 필요할 경우 작성. 다른 저장소와 구분하기 위해서 작성하는 것이 좋음.
예) GitHub Test
 - 3) 공개 유무:
 - Public: 누구나 보거나 사용할 수 있는 저장소
 - Private: 팀원들만 사용할 수 있는 저장소
 - 4) Add a README file: README 파일을 자동 생성할 것인지를 확인
 - MD(Markdown format으로 작성하는 것이 일반적)



The process of using a remote repository

5) Add .gitignore

공개 저장소에 .gitignore를 처음부터 만들기를 원하면 이 버튼에서 필요한 항목을 선택

6) Choose a license

필요한 라이센스를 선택한다.

주로 상업적으로 활용하거나 프로젝트를 공개할 때, 소스 코드의 활용할 수 있는 범위를 지정

- 예)

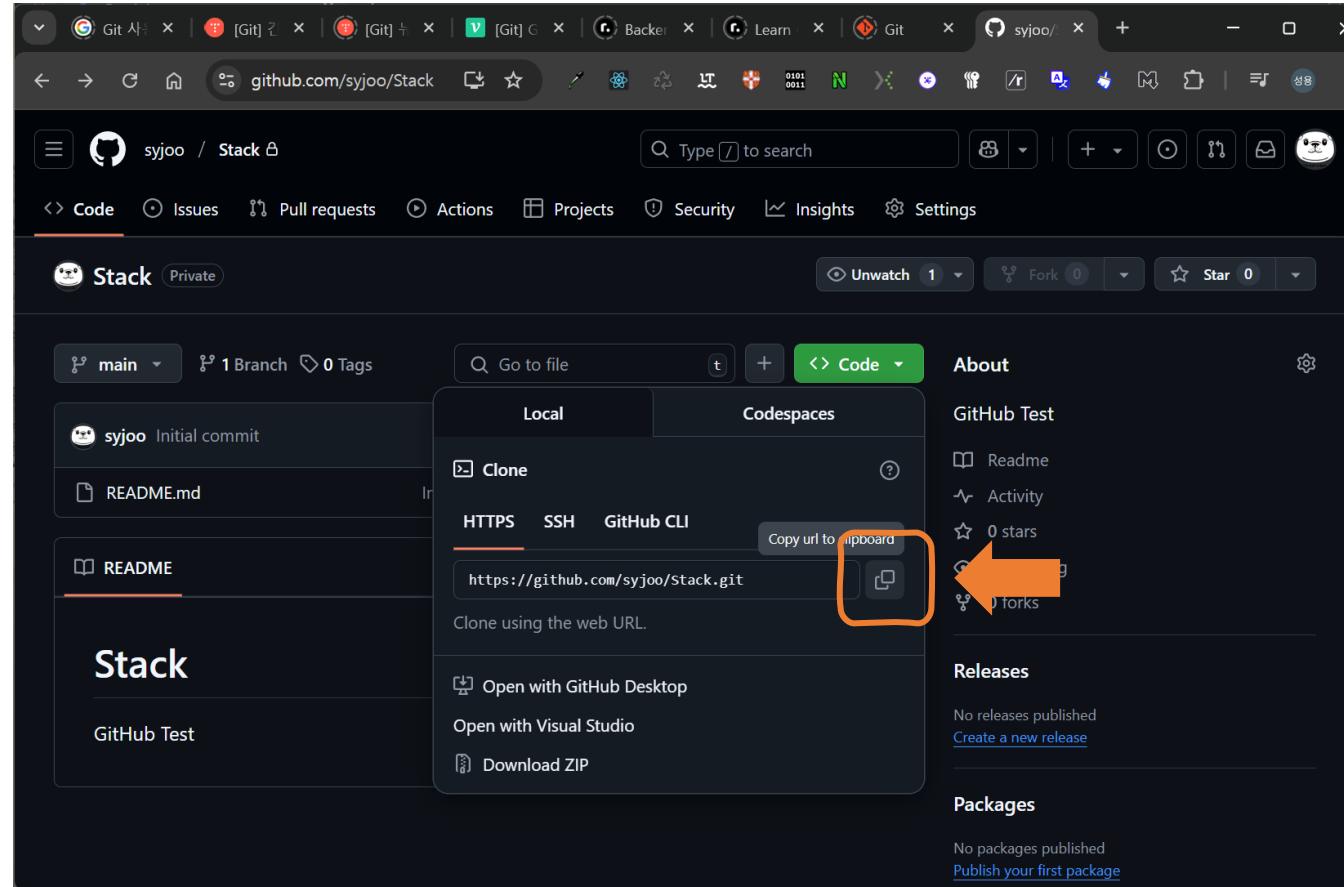
- Repository Name: owner / stack
- Description: GitHub Test
- Private 선택
- Add a README file: unchecked
- Add .gitignore: unchecked
- Choose a license: NONE



The process of using a remote repository

7) 원격 저장소 주소 확인

원격 저장소의 주소를 확인하기 위해서 상단의 “Code” 버튼을 누른다.



- “Copy url to clipboard” 버튼을 눌러 주소를 복사한다.



The process of using a remote repository

8) 원격 저장소와 연결

- GitHub에 원격 저장소를 만들면 기본 브랜치 이름이 `main`이다.
- 반면 로컬 브랜치 이름은 `master`가 기본인 경우가 많은데, 이를 일치시키기 위해서 로컬 브랜치의 이름을 다음과 같이 변경할 수 있다.

```
> git branch -m master main
```

- 원격 저장소와 로컬 저장소를 연결하기 위해서, 아래 명령어를 실행한다. (일반적으로 원격 저장소의 이름은 `origin`으로 설정)

```
> git remote add origin https://github.com/syjoo/Stack.git
```

- `git remote`를 입력하여 원격 저장소를 확인한다.

```
> git remote
```

```
명령 프롬프트
C:\Users\Seong\source/repos\Stack>git remote origin
C:\Users\Seong\source/repos\Stack>git remote -v
origin  https://github.com/syjoo/Stack.git (fetch)
origin  https://github.com/syjoo/Stack.git (push)
C:\Users\Seong\source/repos\Stack>
```



The details of working with remote repository

1. 원격 저장소 등록 (Register Remote Repository) :

- 로컬 저장소를 처음으로 원격 저장소와 연결하기 위해서 `git remote add` 명령어를 사용. 일반적으로 원격 저장소의 이름은 `origin`으로 사용

```
> git remote add origin <원격 저장소 url>
```

- <원격 저장소 URL>은 GitHub 등의 플랫폼에서 해당 저장소의 “Code” 버튼을 클릭해서 확인할 수 있는 HTTPS 또는 SSH 주소

2. 등록된 원격 저장소 확인:

- 등록된 원격 저장소 목록과 URL을 확인

```
> git remote -v
```

- 결과에 `origin`과 해당 URL이 표시



The details of working with remote repository

3. 원격 저장소로 푸시 (Push):

- 로컬 저장소의 Commit된 변경 사항을 원격 저장소로 업로드. 처음 푸시할 때 -u 옵션을 사용하여 로컬 브랜치와 원격 브랜치를 연결(tracking)하는 것이 좋음

```
> git push -u origin <원격 브랜치 이름>
```

- 브랜치를 만들지 않은 경우 <원격 브랜치 이름>은 main 또는 master이다. 원격 브랜치가 main인지 master인지 확인 후, 해당 브랜치로 푸시한다. 예를 들어 main 브랜치로 푸시하면 다음과 같다.

```
> git push -u origin master
```

- 이후 단순히 git push 명령어만으로 연결된 원격 브랜치로 푸시할 수 있음



The details of working with remote repository

3. 원격 저장소로 푸시 (Push):

```
명령 프롬프트
show-type

C:\Users\Seong\source\repos\Stack\Stack>git push -u origin master
Enumerating objects: 26, done.
Counting objects: 100% (26/26), done.
Delta compression using up to 28 threads
Compressing objects: 100% (22/22), done.
Writing objects: 100% (26/26), 4.33 KiB | 2.17 MiB/s, done.
Total 26 (delta 10), reused 0 (delta 0), pack-reused 0 (from 0)
remote: Resolving deltas: 100% (10/10), done.
remote:
remote: Create a pull request for 'master' on GitHub by visiting:
remote:   https://github.com/syjoo/Stack/pull/new/master
remote:
To https://github.com/syjoo/Stack.git
 * [new branch]      master -> master
branch 'master' set up to track 'origin/master'.

C:\Users\Seong\source\repos\Stack\Stack>
```

- 위 명령어는 원격 저장소에 존재하지 않는 브랜치를 등록하는 것이기 때문에, 새로운 브랜치를 만들어 push를 하게 된다.
- 이 경우 원격 저장소에서 merge를 수행해야 하기 때문에 불편할 수 있다.



The details of working with remote repository

3. 원격 저장소로 푸시 (Push):

- 앞 슬라이드의 불편함을 해소하기 위해서 로컬의 master 브랜치를 main 브랜치로 변경 후 push를 할 수 있다.
- 이 경우, 로컬 저장소 보다 원격 저장소가 늦게 만들어졌다면, git push 시 다음과 같은 오류가 발생할 수 있다.

```
! [rejected] main -> main (non-fast-forward)
```

```
error: failed to push some refs to 'https://github.com/syjoo/Stack.git'
```

- 이 오류는 main의 commit 시점이 더 최신이기 때문이다.
- 위 문제를 해결하기 위해서 다음 명령을 수행한다.

```
> git pull origin main --rebase
```

- 만일 서버와 로컬 모두 파일을 변경했다면, rebase에 실패할 수 있다. 이 경우 오류 메시지가 표시되며, 소스를 열어 직접 해결 해야 한다.
- `git pull --rebase`는 로컬의 변경사항을 일시적으로 저장하고, 원격 브랜치의 Commit을 먼저 적용 후 로컬 변경사항을 다시 적용한다.
- 두 번째 오류가 발생하고 해결한 경우 `git rebase --continue`를 실행한다.



The details of working with remote repository

3. 원격 저장소로 푸시 (Push):

- 모든 문제를 해결했다면, 다음 명령어를 실행해서 서버에 데이터를 전송한다.

```
> git push -u origin main
```

A screenshot of a Windows Command Prompt window titled "명령 프롬프트". The window shows the command `git push -u origin main` being run and its output. The output indicates that 24 objects were enumerated, counted, and compressed. It shows the progress of writing objects at 4.09 KiB/s and 2.04 MiB/s. The total number of objects was 23, with 9 deltas reused and 0 pack-reused. The remote repository is https://github.com/syjoo/Stack.git, and the branch 'main' was set up to track 'origin/main'. The command prompt then returns to the directory C:\Users\Seong\source\repos\Stack\Stack>.

```
C:\Users\Seong\source\repos\Stack\Stack>git push -u origin main
Enumerating objects: 24, done.
Counting objects: 100% (24/24), done.
Delta compression using up to 28 threads
Compressing objects: 100% (21/21), done.
Writing objects: 100% (23/23), 4.09 KiB | 2.04 MiB/s, done.
Total 23 (delta 9), reused 0 (delta 0), pack-reused 0 (from 0)
remote: Resolving deltas: 100% (9/9), done.
To https://github.com/syjoo/Stack.git
 67ab2c5..6731b05  main -> main
branch 'main' set up to track 'origin/main'.

C:\Users\Seong\source\repos\Stack\Stack>
```



The details of working with remote repository

4. 원격 저장소 확인

A screenshot of a web browser displaying a GitHub repository page for a user named 'syjoo'. The repository is named 'Stack' and is private. The 'Code' tab is selected. On the left, there is a list of files and their commit history:

File	Commit Message	Time
.gitignore	Add .gitignore file	25 minutes ago
README.md	Initial commit	40 minutes ago
Stack.hpp	Add type	23 minutes ago
Stack.vcxproj	First commit all files.	25 minutes ago
Stack.vcxproj.filters	First commit all files.	25 minutes ago
Stack.vcxproj.user	First commit all files.	25 minutes ago
main.cpp	Add type	23 minutes ago

At the bottom of the list, there is a link to 'README'.

The right sidebar contains sections for 'About', 'GitHub', 'Releases', and 'Packages'.

- About**: Shows 1 branch, 0 tags, 7 commits, and 1 watching.
- GitHub**: Shows Readme, Activity, 0 stars, 1 watching, and 0 forks.
- Releases**: Shows 'No releases published' and a link to 'Create a new release'.
- Packages**: Shows 'No packages published' and a link to 'Publish your first package'.

- 옆의 그림을 보면, 로컬 저장소에 있던 파일들이 원격 저장소에 업로드 되었음을 볼 수 있다.



The details of working with remote repository

5. 원격 저장소에서 가져오기 (Pull):

- 여러 사람이 협업할 경우, 새로 작업하기 전 최신 버전을 가져온 후 작업을 진행하는 것이 좋다.
- 원격 저장소에서 변경 내용을 가져오기 위해서 다음 명령을 수행한다.

```
> git pull origin <원격 브랜치 이름>
```

- 예를 들어 `origin`의 `main` 브랜치에서 변경사항을 가져 오려면 다음과 같음.

```
> git pull origin main
```

6. 원격 저장소 정보 가져오기 (Fetch):

5. 원격 저장소의 변경 사항을 로컬 저장소로 가져오지만, 현재 작업 중인 브랜치에 자동으로 병합하지 않는다. 변경 사항을 확인하고 수동으로 병합하고 싶을 때 사용

```
> git fetch origin
```

6. 가져온 변경 사항은 `origin/<브랜치 이름>` 형태로 로컬에 저장되며, `git merge origin/<브랜치 이름>` 명령어를 사용하여 현재 브랜치에 병합할 수 있음

SUB CHAPTER:

Other Options



git log

- `git log` 명령어는 Git 저장소의 Commit 이력을 보는데 사용
 - 누가 언제 어떤 변경 사항을 Commit했는지, 그리고 각 Commit에 대한 메시지를 확인할 수 있음
 - 기본 사용법

```
> git log
```

- 기본적으로 `git log`는 다음과 같은 정보를 순서대로 (최신 Commit부터) 보여줌
 - commit 해시: 각 Commit을 고유하게 식별하는 40자 길이의 SHA-1 해시 값
 - Author: Commit을 수행한 사람의 이름과 이메일 주소
 - Date: Commit이 이루어진 날짜와 시간
 - Commit message: Commit 시 작성된 설명
- 자주 사용되는 옵션:
 - `--oneline`: 각 Commit 정보를 한 줄로 간략하게 표시. Commit 해시의 앞부분과 Commit 메시지만 보여줌

```
> git log --oneline
```



git log

- 자주 사용되는 옵션:
 - `--graph` : 브랜치와 병합 이력을 텍스트 기반의 그래프로 시각화하여 보여줌. 다른 옵션과 함께 사용하면 유용.
`> git log --graph --oneline --decorate --all`
 - `--decorate` : 브랜치, 태그 등의 정보를 함께 표시
 - `--all` : 모든 브랜치의 Commit 이력을 보여줌
 - `--reverse` : Commit 이력을 오래된 순으로 보여줌
`> git log --reverse`
 - `-n <숫자>` : 최근 <숫자>개의 Commit만 보여줌
`> git log -n 5`
 - `--author=<이름>` : 특정 작성자가 수행한 Commit만 보여줌
`> git log --author="John Doe"`



git log

- 자주 사용되는 옵션:
 - `--since=<날짜>` 또는 `--until=<날짜>` : 특정 기간 동안의 Commit만 보여줌. 다양한 날짜 형식을 사용할 수 있음
 - 예: “yesterday”, “2 weeks ago”, “2023-10-26”
`> git log --since="yesterday"`
`> git log --until="2023-10-20"`
 - `--grep=<패턴>` : Commit 메시지에 특정 <패턴>이 포함된 Commit만 보여줌
`> git log --grep="fix bug"`
 - <파일 이름> : 특정 파일의 변경 이력과 관련된 Commit만 보여줌
`> git log index.html`



git diff

- `git diff` 명령어는 Git에서 다양한 작업 영역 간의 변경 사항을 비교하는데 사용. 작업 디렉토리, 스테이징 영역, 그리고 Commit 간의 차이를 확인할 수 있음

- **기본 사용법:**

- **작업 디렉토리와 스테이징 영역 비교:** 현재 작업 디렉토리에서 변경되었지만 아직 스테이징되지 않은 내용을 보여줌

```
> git diff
```

- **스테이징 영역과 마지막 커밋 비교:** 스테이징 영역에 있는 내용과 마지막 커밋(HEAD)의 차이를 보여줌. 즉, 다음 Commit에 포함될 변경 사항을 미리 확인할 수 있음.

```
> git diff --staged
```

또는

```
> git diff --cached
```



- **기본 사용법:**
 - **작업 디렉토리와 특정 커밋 비교:** 작업 디렉토리의 내용과 특정 커밋의 내용을 비교
 > `git diff <커밋 해시 또는 참조>`
 - **스테이징 영역과 특정 커밋 비교:** 스테이징 영역의 내용과 특정 커밋의 내용을 비교
 > `git diff <커밋 해시 또는 참조> --staged`
 - **두 커밋 간 비교:** 두 개의 특정 커밋 간의 변경 사항을 보여줌
 > `git diff <커밋 해시1> <커밋 해시2>`
 - **특정 파일의 변경 사항 비교:** 특정 파일에 대한 변경 사항만 보여줌
 > `git diff <커밋 해시1> <커밋 해시2> <파일 이름>`



git diff

- git diff 출력 형식:

- git diff 명령어는 일반적으로 다음과 같은 형식으로 출력을 보여줌
 - --- a/<파일 경로> : 비교 대상 1(일반적으로 이전 상태)
 - +++ b/<파일 경로> : 비교 대상 2(일반적으로 현재 상태)
 - @@ -<시작 라인>, <라인 수> +<시작 라인>, <라인 수> @@ : 변경된 코드 블록의 위치 정보
 - - : <파일 경로> a에만 있는 라인(제거된 라인)
 - + : <파일 경로> b에만 있는 라인(추가된 라인)