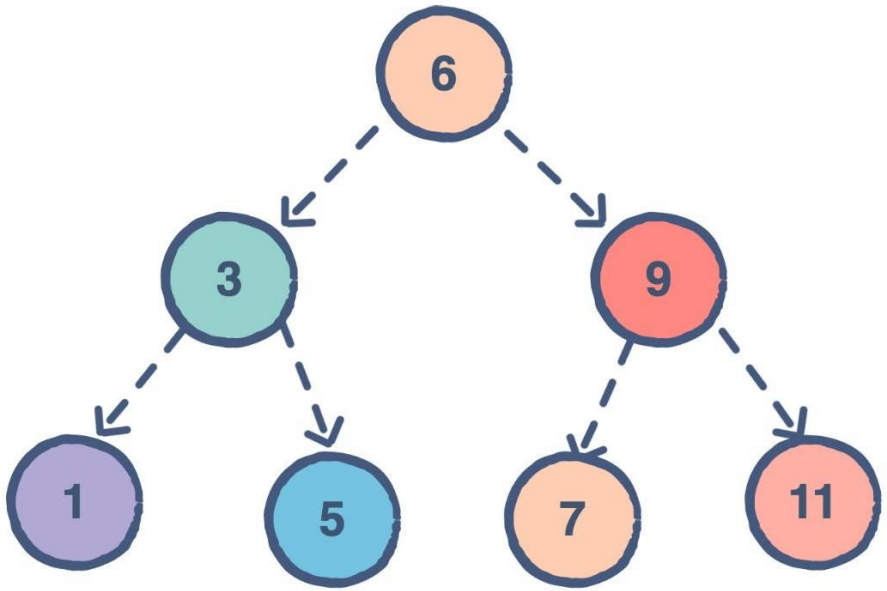
# Huffman coding (HW1)

MediaLab.
Dowan Kwon

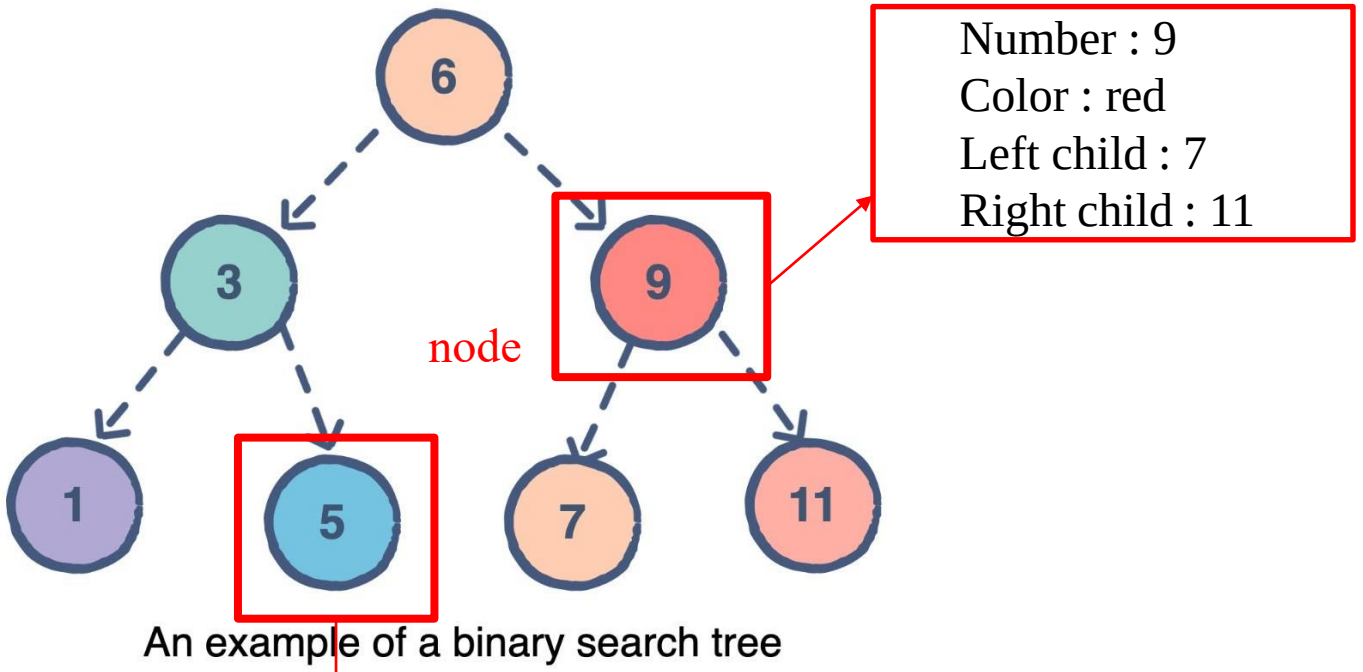
# Contents of Table

## ❖ Binary Search Tree




An example of a binary search tree

## ❖ Binary Search Tree

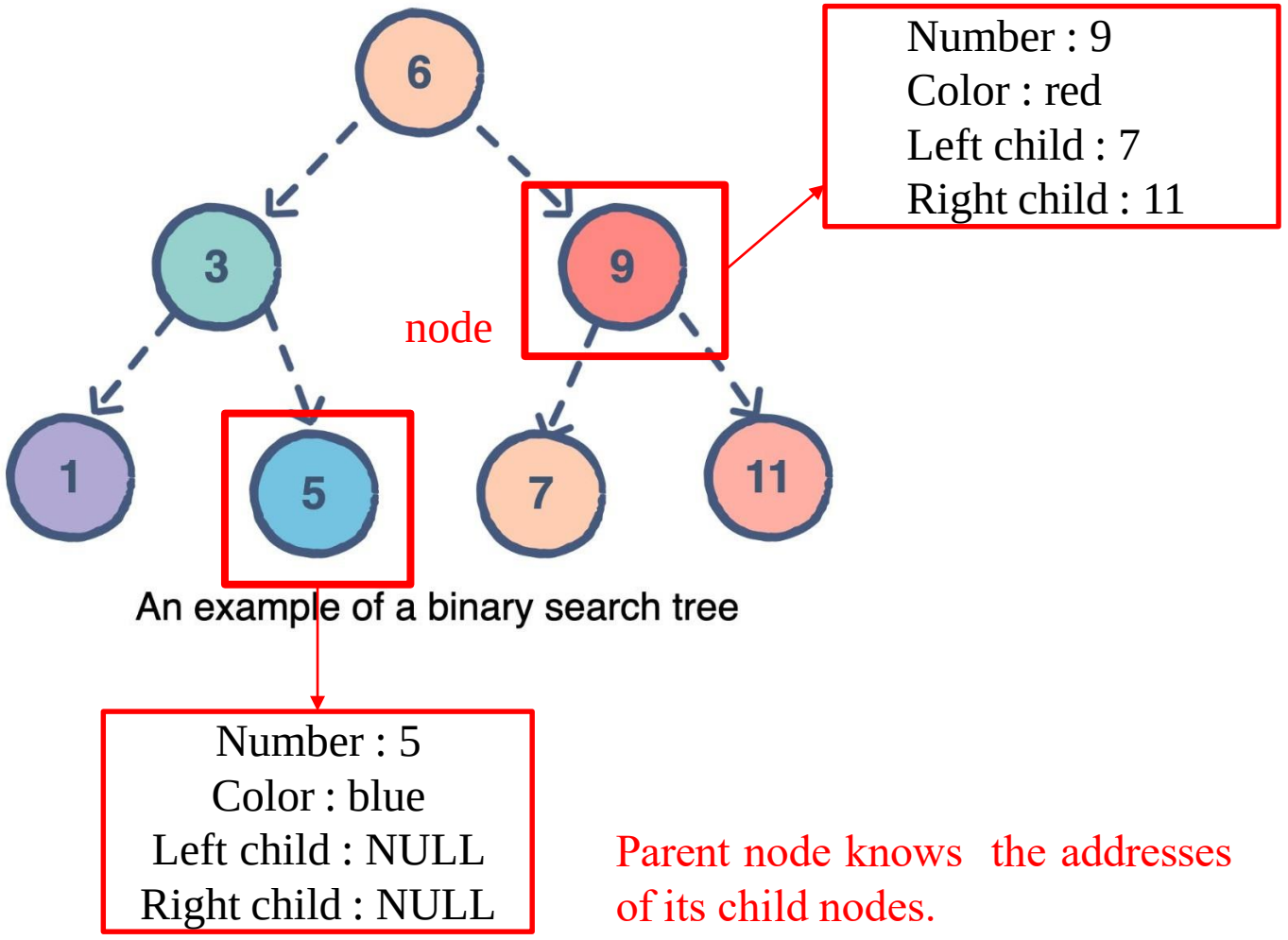



Number : 9
Color : red
Left child : 7
Right child : 11

Number : 5
Color : blue
Left child : NULL
Right child : NULL

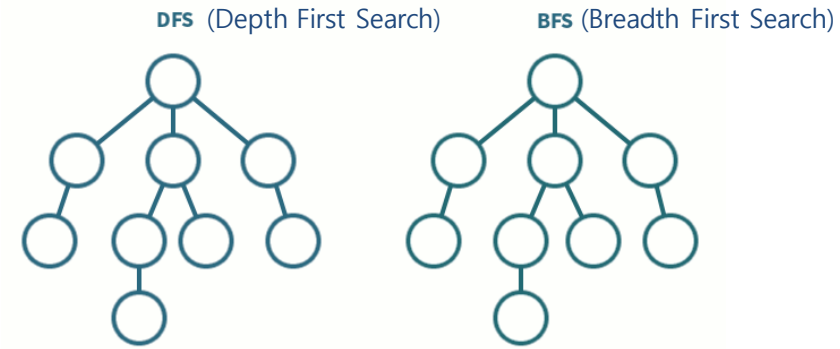
Parent node knows the addresses of its child nodes.

## ❖ Binary Search Tree




Parent node knows the addresses of its child nodes.

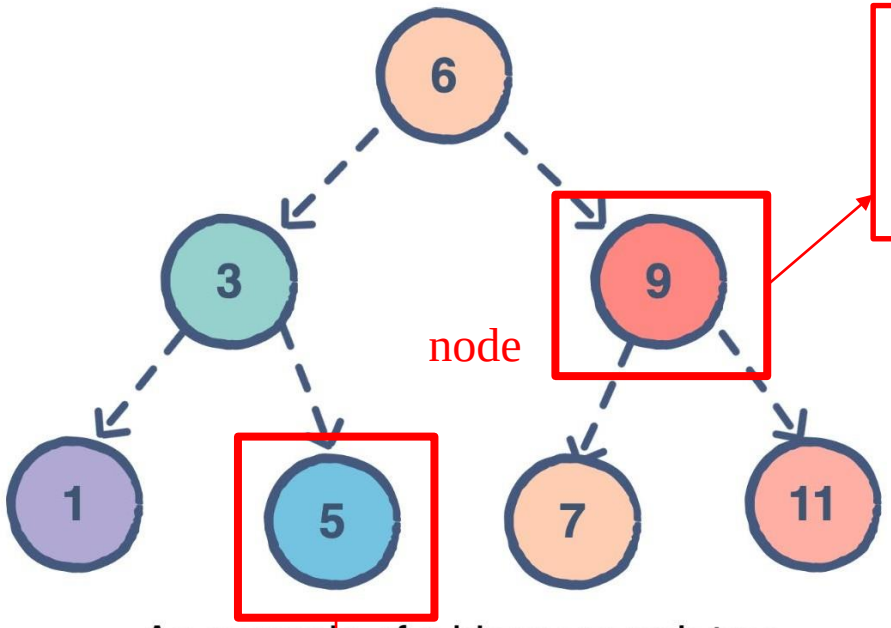
### Search Methods




Question (DFS)
- Is there a 5 inside the tree?
- If it is there, what color is it?

1. Start from root node (6)
2. 6 is not 5
3. Go to left child (3)
4. 3 is not 5
5. Go to left child (1)
6. 1 is not 5
7. Go to left child (NULL)
8. Go back to parent (3)
9. Go to right child (5)
10. 5 is inside the tree, and it is blue

## ❖ Binary Search Tree




Number : 9
Color : red
Left child : 7
Right child :11

Parent node knows the addresses of its child nodes

An example of a binary search tree

Number : 5
Color : blue
Left child : NULL
Right child : NULL

### Main.cpp

```cpp
#include "huffman.h"

int main()
{
    getHuffmanCode();

    return 0;
}
```

### 노드를 나타내는 구조체

```cpp
struct node
{
    string characters;
    unsigned int frequency;
    string code;
    node * leftChild;
    node * rightChild;
};
```

```cpp
vector<node> nodeArray;
```

| A | B | C | … |
|---|---|---|---|
| 7 | 2 | 3 | |

### huffman.h

0

```cpp
void getHuffmanCode()
{
    int size;
    unsigned int tempInt;
    char alphabet;
```

1

```cpp
    //데이터에 사용되는 문자의 종류 개수 입력
    cout << endl;
    cout << "₩t" << "Huffman Tree : ";
    cin >> size;

    cout << endl << endl;
    cout << "₩t" << "문자 빈도" << endl;
    cout << "₩t" << "----------------------" << endl;
```

2

```cpp
    //각 문자별 빈도 수 노드 생성
    for (int i = 0; i < size; i++)
    {
        cout << "₩t";
        node tempNode;
        cin >> alphabet;
        cin >> tempInt;

        tempNode.characters = alphabet;
        tempNode.frequency = tempInt;
        tempNode.leftChild = NULL;
        tempNode.rightChild = NULL;
        nodeArray.push_back(tempNode);
    }
```

3

```cpp
    //Huffman Tree 생성
    node root = getHuffmanTree();

    cout << endl << endl;
    cout << "₩t" << "문자" << "₩t" << "Code" << endl;
    cout << "₩t" << "----------------------" << endl;
```

4

```cpp
    //Huffman Coding Table 생성
    depthFirstSearch(&root, "");
}
```

❖ 다음 데이터 Huffman coding을 통해 압축해보자.

**AAAAAAABBCCCDEEEEFFFFFFG**

① 데이터에서 사용되는 각 문자에 대한 출현 빈도수를 구한다.

| 문자 | A | B | C | D | E | F | G |
|---|---|---|---|---|---|---|---|
| 출현빈도 | 7 | 2 | 3 | 1 | 4 | 6 | 1 |

노드를 나타내는 구조체

```
struct node
{
    string characters;
    unsigned int frequency;
    string code;
    node * leftChild;
    node * rightChild;
};
```

Characters
Frequency
Code


```
int size;
unsigned int tempInt;
char alphabet;

//데이터에 사용되는 문자의 종류 갯수 입력
cout << endl;
cout << "₩t" << "Huffman Tree : ";
cin >> size;
```

①

```
//각 문자별 빈도 수 노드 생성
for (int i = 0; i < size; i++)
{
    cout << "₩t";
    node tempNode;
    cin >> alphabet;
    cin >> tempInt;

    tempNode.characters = alphabet;
    tempNode.frequency = tempInt;
    tempNode.leftChild = NULL;
    tempNode.rightChild = NULL;
    nodeArray.push_back(tempNode);
}
```

②

```
vector<node> nodeArray;
```

| A | B | C | D | E | F | G |
|---|---|---|---|---|---|---|
| 7 | 2 | 3 | 1 | 4 | 6 | 1 |

❖ 다음 데이터 Huffman coding을 통해 압축해보자.

**AAAAAAABBCCCDEEEEFFFFFFG**

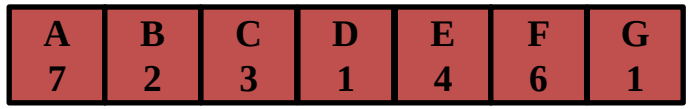
① 데이터에서 사용되는 각 문자에 대한 출현 빈도수를 구한다.

| 문자 | A | B | C | D | E | F | G |
|---|---|---|---|---|---|---|---|
| 출현빈도 | 7 | 2 | 3 | 1 | 4 | 6 | 1 |

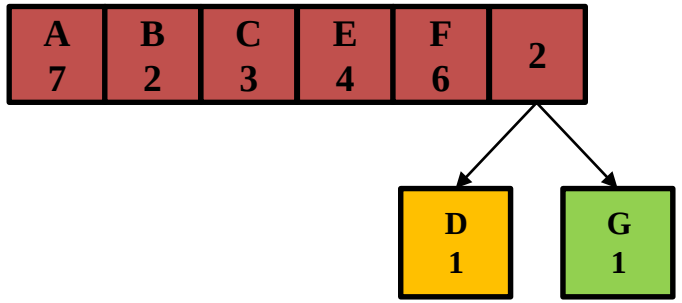
② 빈도가 가장 적은 2개의 문자의 합을 Root에 적는다.

③ 새로운 빈도수의 합을 기준으로 다시 ②과정 **반복**

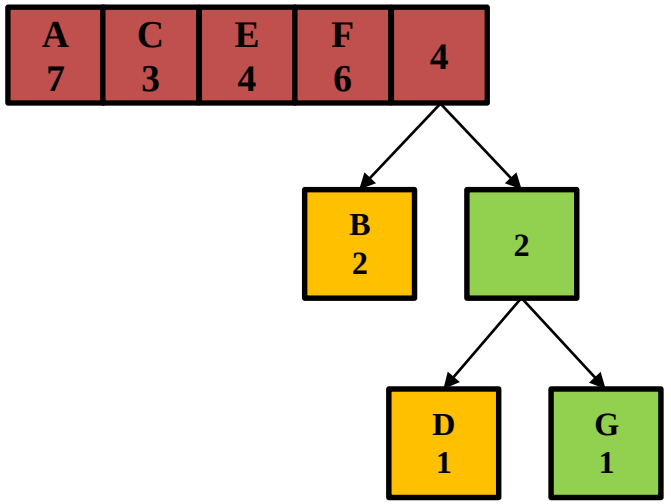
**Step 1**



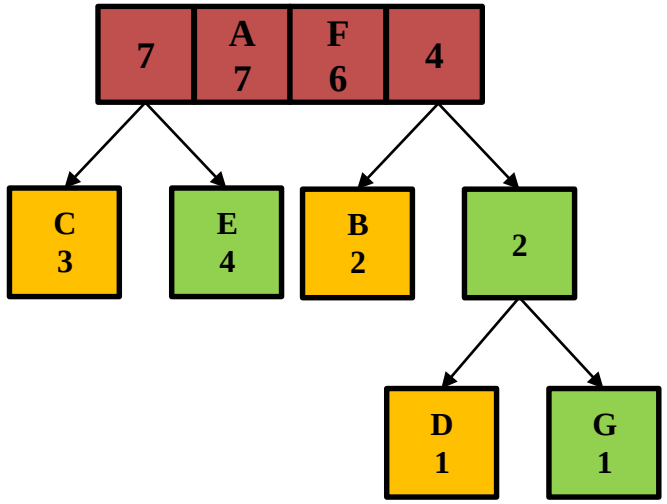

**Step 2**




**Step 3**




**Step 4**



❖ 다음 데이터 Huffman coding을 통해 압축해보자.

**AAAAAAABBCCCDEEEEFFFFFFG**

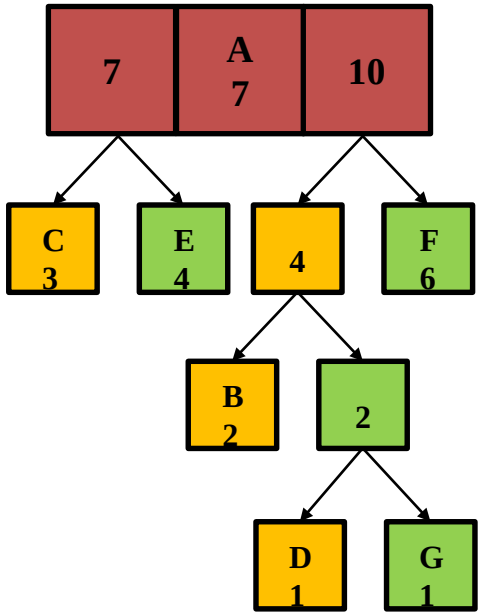
① 데이터에서 사용되는 각 문자에 대한 출현 빈도수를 구한다.

| 문자 | A | B | C | D | E | F | G |
|---|---|---|---|---|---|---|---|
| 출현빈도 | 7 | 2 | 3 | 1 | 4 | 6 | 1 |

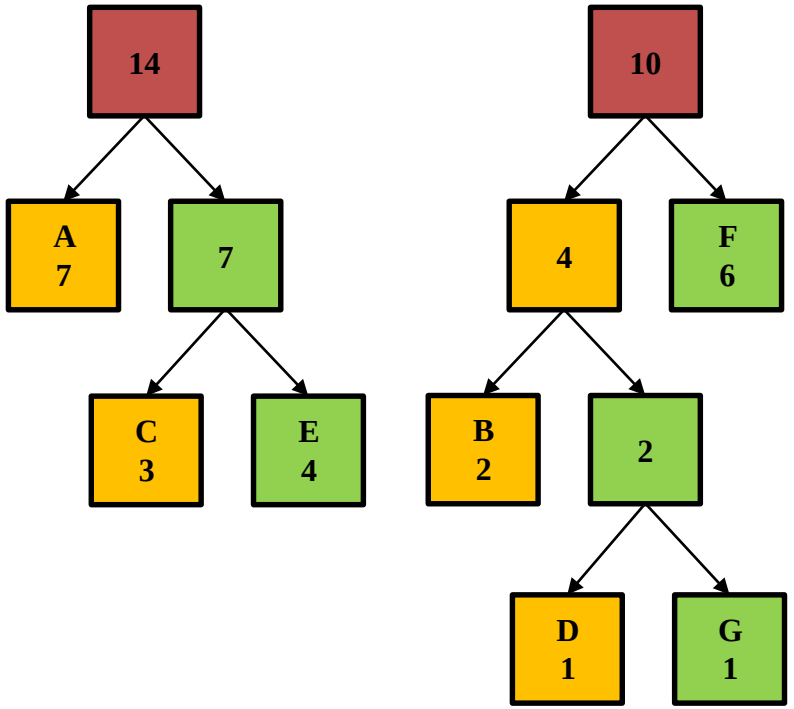
② 빈도가 가장 적은 2개의 문자의 합을 Root에 적는다.

③ 새로운 빈도수의 합을 기준으로 다시 ②과정 **반복**

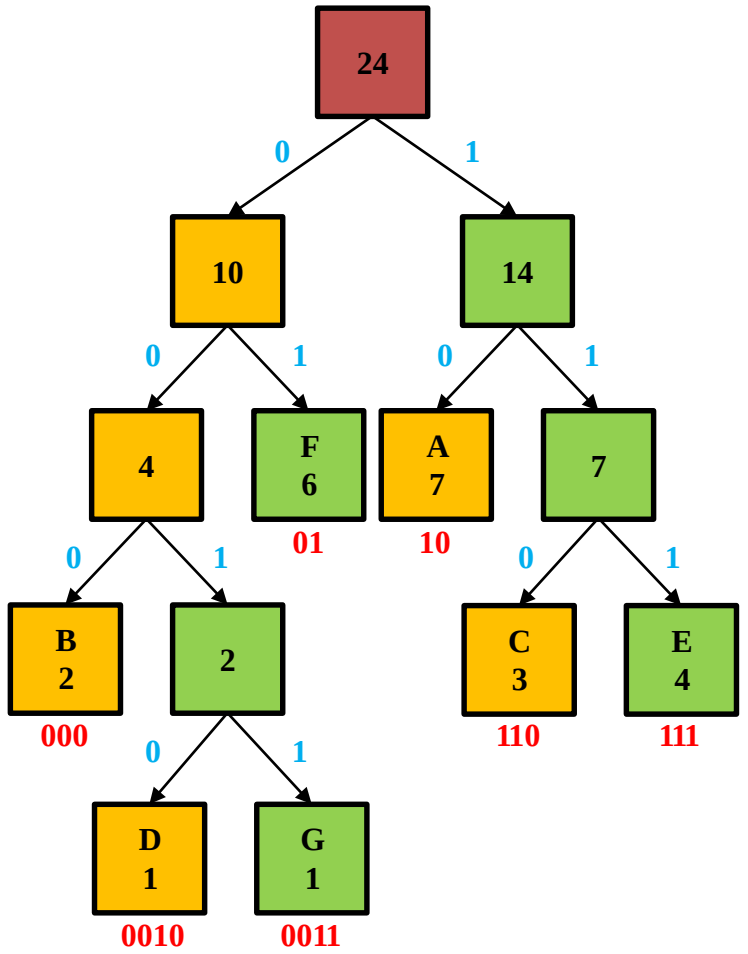
**Step 5**




**Step 6**




**Final Tree**



❖ 다음 데이터 Huffman coding을 통해 압축해보자.

**AAAAAAABBCCCDEEEEFFFFFFG**

① 데이터에서 사용되는 각 문자에 대한 출현 빈도수를 구한다.

| 문자 | A | B | C | D | E | F | G |
|---|---|---|---|---|---|---|---|
| 출현빈도 | 7 | 2 | 3 | 1 | 4 | 6 | 1 |

② 빈도가 가장 적은 2개의 문자의 합을 Root에 적는다.

③ 새로운 빈도수의 합을 기준으로 다시 ②과정 **반복**

```
struct node
{
    string characters;
    unsigned int frequency;
    string code;
    node * leftChild;
    node * rightChild;
};
```

| A | B | C | D | E | F | G |
|---|---|---|---|---|---|---|
| 7 | 2 | 3 | 1 | 4 | 6 | 1 |

Characters
Frequency
Code

**Root** **L** **R**

```
node getHuffmanTree()
{
    while (!nodeArray.empty())
    {
        node * tempNode = new node;   - Parent
        node * tempNode1 = new node;  - Children
        node * tempNode2 = new node;
        *tempNode1 = extractMin();
        *tempNode2 = extractMin();

        tempNode->leftChild = tempNode1;
        tempNode->rightChild = tempNode2;
        tempNode->frequency = tempNode1->frequency + tempNode2->frequency;
        nodeArray.push_back(*tempNode);

        //Root Node만 남았으므로 Huffman Tree 완성
        if (nodeArray.size() == 1)  break;
    }
    return nodeArray[0];
}
```

**Step 1**

| A | B | C | D | E | F | G |
|---|---|---|---|---|---|---|
| 7 | 2 | 3 | 1 | 4 | 6 | 1 |

**Step 2**

| A | B | C | E | F | 2 |
|---|---|---|---|---|---|
| 7 | 2 | 3 | 4 | 6 | |

| D | G |
|---|---|
| 1 | 1 |

**Step 3**

| A | C | E | F | 4 |
|---|---|---|---|---|
| 7 | 3 | 4 | 6 | |

| B | 2 |
|---|---|
| 2 | |

| D | G |
|---|---|
| 1 | 1 |

❖ 다음 데이터 Huffman coding을 통해 압축해보자.

**AAAAAAABBCCCDEEEEFFFFFFG**

① 데이터에서 사용되는 각 문자에 대한 출현 빈도수를 구한다.

| 문자 | A | B | C | D | E | F | G |
|---|---|---|---|---|---|---|---|
| 출현빈도 | 7 | 2 | 3 | 1 | 4 | 6 | 1 |

② 빈도가 가장 적은 2개의 문자의 합을 Root에 적는다.

③ 새로운 빈도수의 합을 기준으로 다시 ②과정 **반복**

**Step 1**

| A | B | C | D | E | F | G |
|---|---|---|---|---|---|---|
| 7 | 2 | 3 | 1 | 4 | 6 | 1 |

**Step 2**

| A | B | C | E | F | 2 |
|---|---|---|---|---|---|
| 7 | 2 | 3 | 4 | 6 | |

| D | G |
|---|---|
| 1 | 1 |

```
struct node
{
    string characters;
    unsigned int frequency;
    string code;
    node * leftChild;
    node * rightChild;
};
```

Characters
Frequency
Code

**Root** **L** **R**

| A | B | C | D | E | F | G |
|---|---|---|---|---|---|---|
| 7 | 2 | 3 | 1 | 4 | 6 | 1 |

```
node getHuffmanTree()
{
    while (!nodeArray.empty())
    {
        node * tempNode = new node;
        node * tempNode1 = new node;
        node * tempNode2 = new node;
        *tempNode1 = extractMin();
        *tempNode2 = extractMin();

        tempNode->leftChild = tempNode1;
        tempNode->rightChild = tempNode2;
        tempNode->frequency = tempNode1->frequency + tempNode2->frequency;
        nodeArray.push_back(*tempNode);

        //Root Node만 남았으므로 Huffman Tree 완성
        if (nodeArray.size() == 1)  break;
    }
    return nodeArray[0];
}
```

-nodeArray에서 빈도수가 가장 적은 노드를 반환함.
- nodeArray에서 해당 노드 제거.

```
node extractMin()
{
    unsigned int min = UINT_MAX;
    vector<node>::iterator iter, position;
    for (iter = nodeArray.begin(); iter != nodeArray.end(); iter++)
    {
        if (min > (*iter).frequency)
        {
            position = iter;
            min = (*iter).frequency;
        }
    }

    node tempNode = (*position);
    nodeArray.erase(position);

    return tempNode;
}
```

- nodeArray 탐색하여 빈도수가 최소인 노드 골라냄.

❖ 다음 데이터 Huffman coding을 통해 압축해보자.

**AAAAAAABBCCCDEEEEFFFFFFG**

① 데이터에서 사용되는 각 문자에 대한 출현 빈도수를 구한다.

| 문자 | A | B | C | D | E | F | G |
|---|---|---|---|---|---|---|---|
| 출현빈도 | 7 | 2 | 3 | 1 | 4 | 6 | 1 |

② 빈도가 가장 적은 2개의 문자의 합을 Root에 적는다.

③ 새로운 빈도수의 합을 기준으로 다시 ②과정 **반복**

**Step 1**

| A | B | C | D | E | F | G |
|---|---|---|---|---|---|---|
| 7 | 2 | 3 | 1 | 4 | 6 | 1 |

**Step 2**

| A | B | C | E | F | 2 |
|---|---|---|---|---|---|
| 7 | 2 | 3 | 4 | 6 | |

| D | G |
|---|---|
| 1 | 1 |

```
struct node
{
    string characters;
    unsigned int frequency;
    string code;
    node * leftChild;
    node * rightChild;
};
```

Characters
Frequency
Code

**Root** **L** **R**

| A | B | C | D | E | F | G |
|---|---|---|---|---|---|---|
| 7 | 2 | 3 | 1 | 4 | 6 | 1 |

```
node getHuffmanTree()
{
    while (!nodeArray.empty())
    {
        node * tempNode = new node;
        node * tempNode1 = new node;
        node * tempNode2 = new node;
        *tempNode1 = extractMin();
        *tempNode2 = extractMin();

        tempNode->leftChild = tempNode1;
        tempNode->rightChild = tempNode2;
        tempNode->frequency = tempNode1->frequency + tempNode2->frequency;
        nodeArray.push_back(*tempNode);

        //Root Node만 남았으므로 Huffman Tree 완성
        if (nodeArray.size() == 1)  break;
    }
    return nodeArray[0];
}
```

-nodeArray에서 빈도수가 가장 적은 노드를 반환함.
- nodeArray에서 해당 노드 제거.

```
node extractMin()
{
    unsigned int min = UINT_MAX;
    vector<node>::iterator iter, position;
    for (iter = nodeArray.begin(); iter != nodeArray.end(); iter++)
    {
        if (min > (*iter).frequency)
        {
            position = iter;
            min = (*iter).frequency;
        }
    }

    node tempNode = (*position);
    nodeArray.erase(position);

    return tempNode;
}
```

- 해당 노드 nodeArray 에서 제거

# Huffman Coding

❖ 다음 데이터 Huffman coding을 통해 압축해보자.

**AAAAAAABBCCCDEEEEFFFFFFG**

① 데이터에서 사용되는 각 문자에 대한 출현 빈도수를 구한다.

| 문자 | A | B | C | D | E | F | G |
|---|---|---|---|---|---|---|---|
| 출현빈도 | 7 | 2 | 3 | 1 | 4 | 6 | 1 |

② 빈도가 가장 적은 2개의 문자의 합을 Root에 적는다.

③ 새로운 빈도수의 합을 기준으로 다시 ②과정 **반복**

**Step 1**

| A | B | C | D | E | F | G |
|---|---|---|---|---|---|---|
| 7 | 2 | 3 | 1 | 4 | 6 | 1 |

**Step 2**

| A | B | C | E | F | 2 |
|---|---|---|---|---|---|
| 7 | 2 | 3 | 4 | 6 | |

| D | G |
|---|---|
| 1 | 1 |

```
struct node
{
    string characters;
    unsigned int frequency;
    string code;
    node * leftChild;
    node * rightChild;
};
```

Characters
Frequency
Code

**Root** **L** **R**

| A | B | C | D | E | F | G |
|---|---|---|---|---|---|---|
| 7 | 2 | 3 | 1 | 4 | 6 | 1 |

```
node getHuffmanTree()
{
    while (!nodeArray.empty())
    {
        node * tempNode = new node;
        node * tempNode1 = new node;
        node * tempNode2 = new node;
        *tempNode1 = extractMin();
        *tempNode2 = extractMin();

        tempNode->leftChild = tempNode1;
        tempNode->rightChild = tempNode2;
        tempNode->frequency = tempNode1->frequency + tempNode2->frequency;
        nodeArray.push_back(*tempNode);

        //Root Node만 남았으므로 Huffman Tree 완성
        if (nodeArray.size() == 1)  break;
    }
    return nodeArray[0];
}
```

-nodeArray에서 빈도수가 가장 적은 노드를 반환함.
- nodeArray에서 해당 노드 제거.

```
node extractMin()
{
    unsigned int min = UINT_MAX;
    vector<node>::iterator iter, position;
    for (iter = nodeArray.begin(); iter != nodeArray.end(); iter++)
    {
        if (min > (*iter).frequency)
        {
            position = iter;
            min = (*iter).frequency;
        }
    }

    node tempNode = (*position);
    nodeArray.erase(position);

    return tempNode;
}
```

- 빈도수 가장 적은 노드 반환

❖ 다음 데이터 Huffman coding을 통해 압축해보자.

**AAAAAAABBCCCDEEEEFFFFFFG**

① 데이터에서 사용되는 각 문자에 대한 출현 빈도수를 구한다.

| 문자 | A | B | C | D | E | F | G |
|---|---|---|---|---|---|---|---|
| 출현빈도 | 7 | 2 | 3 | 1 | 4 | 6 | 1 |

② 빈도가 가장 적은 2개의 문자의 합을 Root에 적는다.

③ 새로운 빈도수의 합을 기준으로 다시 ②과정 **반복**

```
struct node
{
    string characters;
    unsigned int frequency;
    string code;
    node * leftChild;
    node * rightChild;
};
```

| A | B | C | D | E | F | G |
|---|---|---|---|---|---|---|
| 7 | 2 | 3 | 1 | 4 | 6 | 1 |

Characters
Frequency
Code

**Root** **L** **R**

```
node getHuffmanTree()
{
    while (!nodeArray.empty())
    {
        node * tempNode = new node;
        node * tempNode1 = new node;
        node * tempNode2 = new node;
        *tempNode1 = extractMin();
        *tempNode2 = extractMin();

        tempNode->leftChild = tempNode1;
        tempNode->rightChild = tempNode2;
        tempNode->frequency = tempNode1->frequency + tempNode2->frequency;
        nodeArray.push_back(*tempNode);

        //Root Node만 남았으므로 Huffman Tree 완성
        if (nodeArray.size() == 1)  break;
    }
    return nodeArray[0];
}
```

- 부모 노드에 자식 노드의 주소값 배정

**Step 1**

| A | B | C | D | E | F | G |
|---|---|---|---|---|---|---|
| 7 | 2 | 3 | 1 | 4 | 6 | 1 |

**Step 2**

| A | B | C | E | F | 2 |
|---|---|---|---|---|---|
| 7 | 2 | 3 | 4 | 6 | |

| D | G |
|---|---|
| 1 | 1 |

**Step 3**

| A | C | E | F | 4 |
|---|---|---|---|---|
| 7 | 3 | 4 | 6 | |

| B | 2 |
|---|---|
| 2 | |

| D | G |
|---|---|
| 1 | 1 |

❖ 다음 데이터 Huffman coding을 통해 압축해보자.

**AAAAAAABBCCCDEEEEFFFFFFG**

① 데이터에서 사용되는 각 문자에 대한 출현 빈도수를 구한다.

| 문자 | A | B | C | D | E | F | G |
|---|---|---|---|---|---|---|---|
| 출현빈도 | 7 | 2 | 3 | 1 | 4 | 6 | 1 |

② 빈도가 가장 적은 2개의 문자의 합을 Root에 적는다.

③ 새로운 빈도수의 합을 기준으로 다시 ②과정 **반복**

```
struct node
{
    string characters;
    unsigned int frequency;
    string code;
    node * leftChild;
    node * rightChild;
};
```

Characters
Frequency
Code

**Root** **L** **R**

| A | B | C | D | E | F | G |
|---|---|---|---|---|---|---|
| 7 | 2 | 3 | 1 | 4 | 6 | 1 |

```
node getHuffmanTree()
{
    while (!nodeArray.empty())
    {
        node * tempNode = new node;
        node * tempNode1 = new node;
        node * tempNode2 = new node;
        *tempNode1 = extractMin();
        *tempNode2 = extractMin();

        tempNode->leftChild = tempNode1;
        tempNode->rightChild = tempNode2;
        tempNode->frequency = tempNode1->frequency + tempNode2->frequency;
        nodeArray.push_back(*tempNode);

        //Root Node만 남았으므로 Huffman Tree 완성
        if (nodeArray.size() == 1)  break;
    }
    return nodeArray[0];
}
```

- 빈도수 합 설정 후 nodeArray에 삽입

**Step 1**

| A | B | C | D | E | F | G |
|---|---|---|---|---|---|---|
| 7 | 2 | 3 | 1 | 4 | 6 | 1 |

**Step 2**

| A | B | C | E | F | 2 |
|---|---|---|---|---|---|
| 7 | 2 | 3 | 4 | 6 | |

| D | G |
|---|---|
| 1 | 1 |

**Step 3**

| A | C | E | F | 4 |
|---|---|---|---|---|
| 7 | 3 | 4 | 6 | |

| B | 2 |
|---|---|
| 2 | |

| D | G |
|---|---|
| 1 | 1 |

❖ 다음 데이터 Huffman coding을 통해 압축해보자.

**AAAAAAABBCCCDEEEEFFFFFFG**

① 데이터에서 사용되는 각 문자에 대한 출현 빈도수를 구한다.

| 문자 | A | B | C | D | E | F | G |
|---|---|---|---|---|---|---|---|
| 출현빈도 | 7 | 2 | 3 | 1 | 4 | 6 | 1 |

② 빈도가 가장 적은 2개의 문자의 합을 Root에 적는다.

③ 새로운 빈도수의 합을 기준으로 다시 ②과정 **반복**

```
struct node
{
    string characters;
    unsigned int frequency;
    string code;
    node * leftChild;
    node * rightChild;
};
```

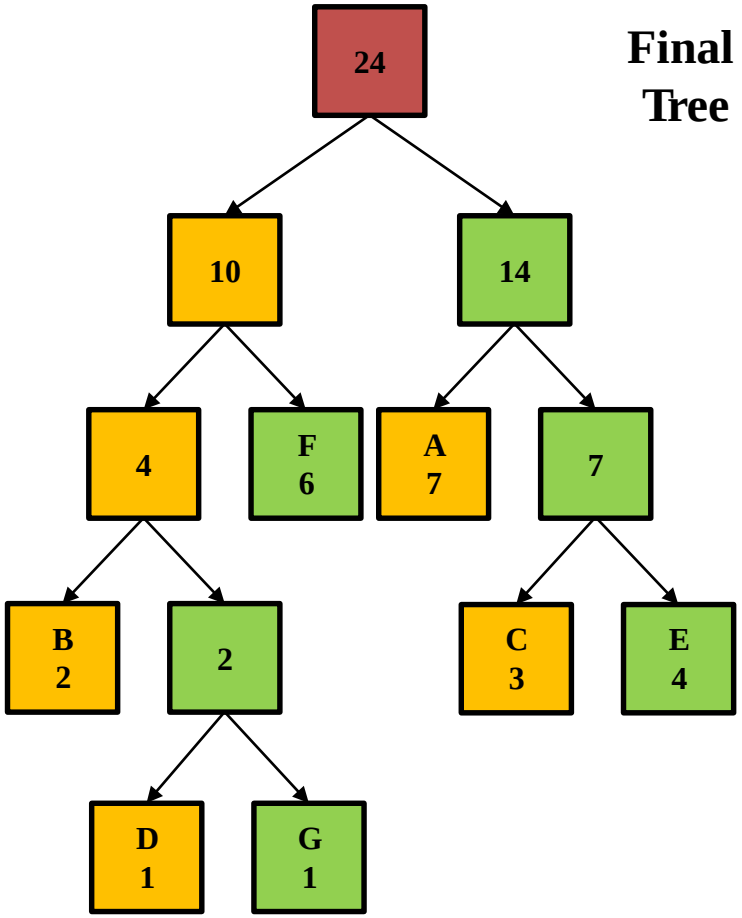
Characters
Frequency
Code

**Root** **L** **R**

| A<br>7 | B<br>2 | C<br>3 | D<br>1 | E<br>4 | F<br>6 | G<br>1 |
|---|---|---|---|---|---|---|

```
node getHuffmanTree()
{
    while (!nodeArray.empty())
    {
        node * tempNode = new node;
        node * tempNode1 = new node;
        node * tempNode2 = new node;
        *tempNode1 = extractMin();
        *tempNode2 = extractMin();

        tempNode->leftChild = tempNode1;
        tempNode->rightChild = tempNode2;
        tempNode->frequency = tempNode1->frequency + tempNode2->frequency;
        nodeArray.push_back(*tempNode);

        //Root Node만 남았으므로 Huffman Tree 완성
        if (nodeArray.size() == 1)  break;
    }
    return nodeArray[0];
}
```

- Root Node 반환



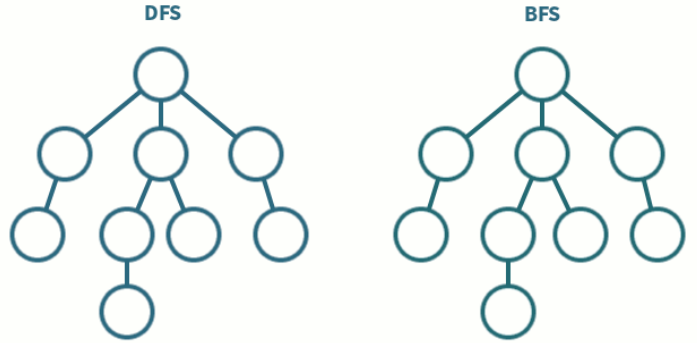

**Final Tree**

# Huffman Coding

❖ 다음 데이터 Huffman coding을 통해 압축해보자.

**AAAAAAABBCCCDEEEEFFFFFFG**

④ 각 가지의 왼쪽에는 0, 오른쪽에는 1을 쓴다. (**Depth First Search**를 이용하였다.)



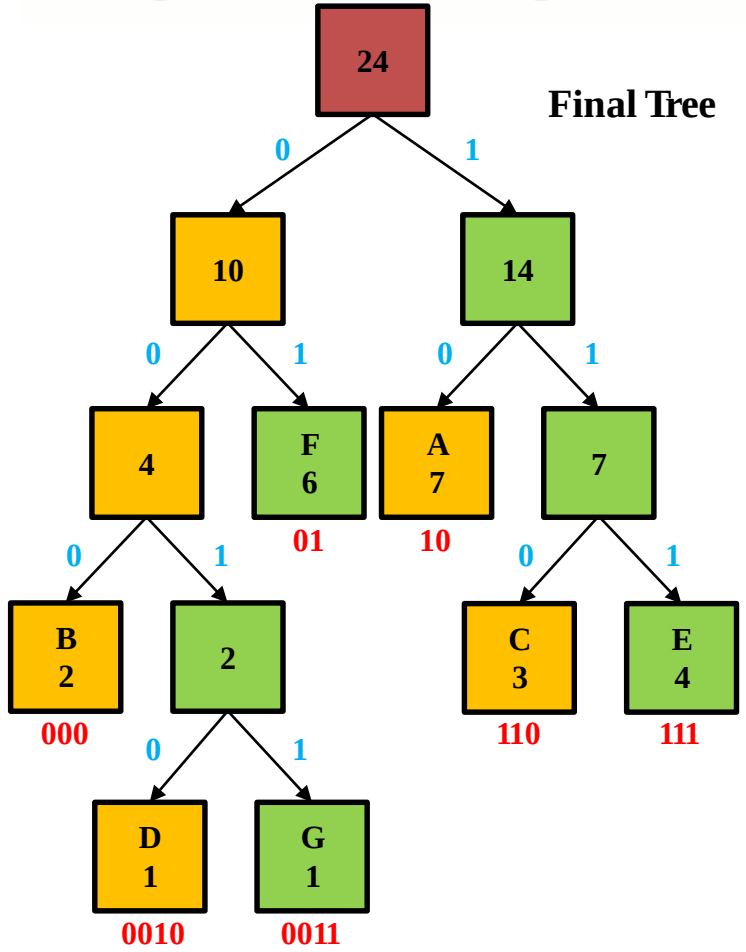

```
//Huffman Coding Table 생성
depthFirstSearch(&root, "");
```

```
void depthFirstSearch(node* tempRoot, string s)
{
    node* root1 = tempRoot;

    root1->code = s;
    if (root1 == NULL)
    {

    }
    else if (root1->leftChild == NULL && root1->rightChild == NULL)
    {
        cout << "₩t" << root1->characters << "₩t" << root1->code << endl;
    }
    else
    {
        root1->leftChild->code = s.append("0");
        s.erase(s.end() - 1);
        root1->rightChild->code = s.append("1");
        s.erase(s.end() - 1);

        depthFirstSearch(root1->leftChild, s.append("0"));
        s.erase(s.end() - 1);
        depthFirstSearch(root1->rightChild, s.append("1"));
        s.erase(s.end() - 1);
    }
}
```

- 각 노드별 코드 부여하는 재귀함수

Microsoft Visual Studio 디버그 콘솔

```
Huffman Tree : 7

문자 빈도
--------------------
A 7
B 2
C 3
D 1
E 4
F 6
G 1


문자    Code
--------------------
B       000
D       0010
G       0011
F       01
A       10
C       110
E       111
```



# Huffman Coding

❖ 다음 데이터 Huffman coding을 통해 압축해보자.

**AAAAAAABBCCCDEEEEFFFFFFG**

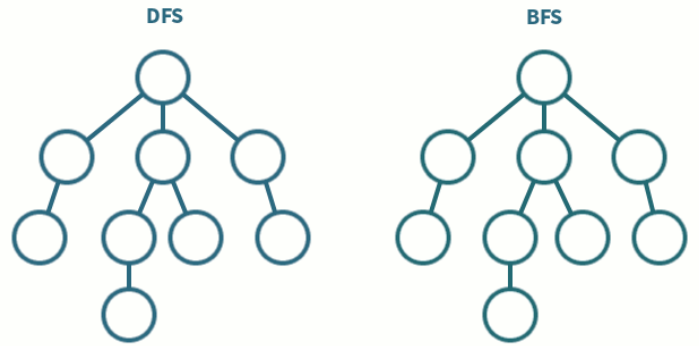
④ 각 가지의 왼쪽에는 0, 오른쪽에는 1을 쓴다. (**Depth First Search**를 이용하였다.)

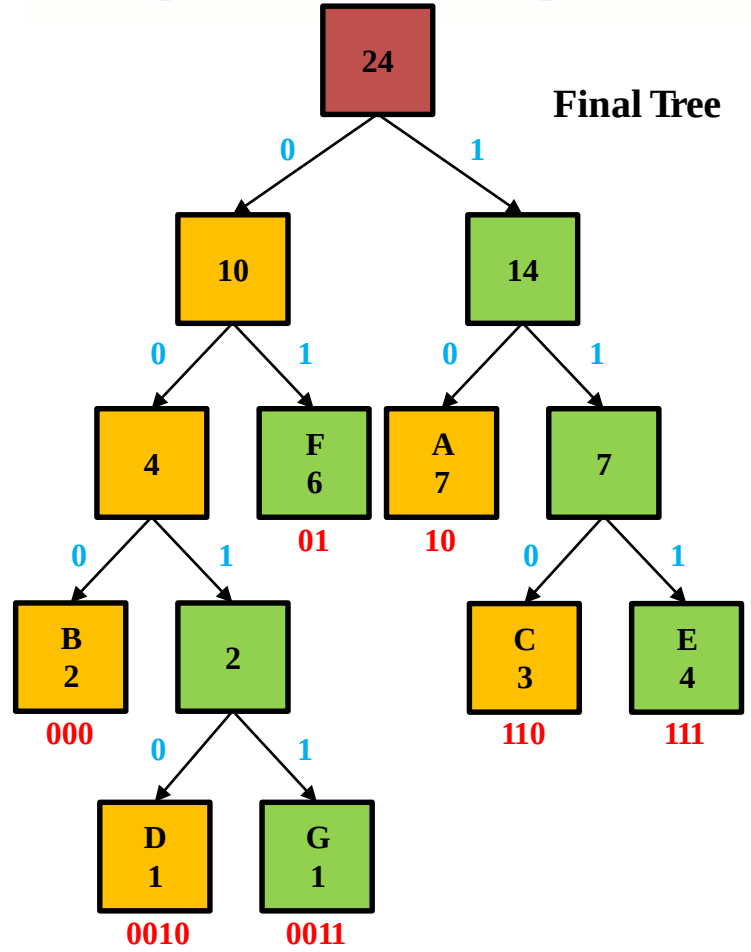
```
//Huffman Coding Table 생성
depthFirstSearch(&root, "");
}
```

```
void depthFirstSearch(node* tempRoot, string s)
{
    node* root1 = tempRoot;

    root1->code = s;
    if (root1 == NULL)
    {

    }
    else if (root1->leftChild == NULL && root1->rightChild == NULL)
    {
        cout << "₩t" << root1->characters << "₩t" << root1->code << endl;
    }
    else
    {
        root1->leftChild->code = s.append("0");
        s.erase(s.end() - 1);
        root1->rightChild->code = s.append("1");
        s.erase(s.end() - 1);

        depthFirstSearch(root1->leftChild, s.append("0"));
        s.erase(s.end() - 1);
        depthFirstSearch(root1->rightChild, s.append("1"));
        s.erase(s.end() - 1);
    }
}
```

- 허프만 코드 (시작할 때는 비어있음)

```
Microsoft Visual Studio 디버그 콘솔

Huffman Tree : 7

문자 빈도
--------------------
A 7
B 2
C 3
D 1
E 4
F 6
G 1


문자    Code
--------------------
B       000
D       0010
G       0011
F       01
A       10
C       110
E       111
```

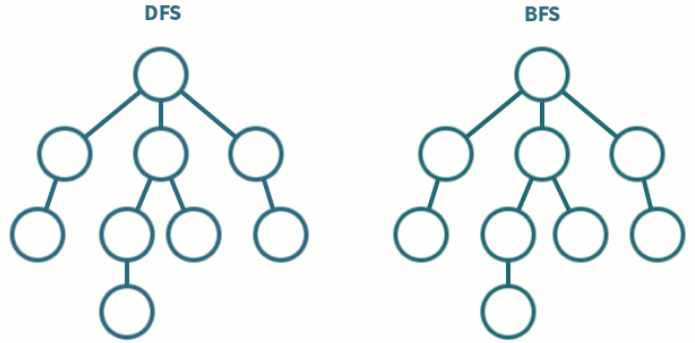


# Huffman Coding

❖ 다음 데이터 Huffman coding을 통해 압축해보자.

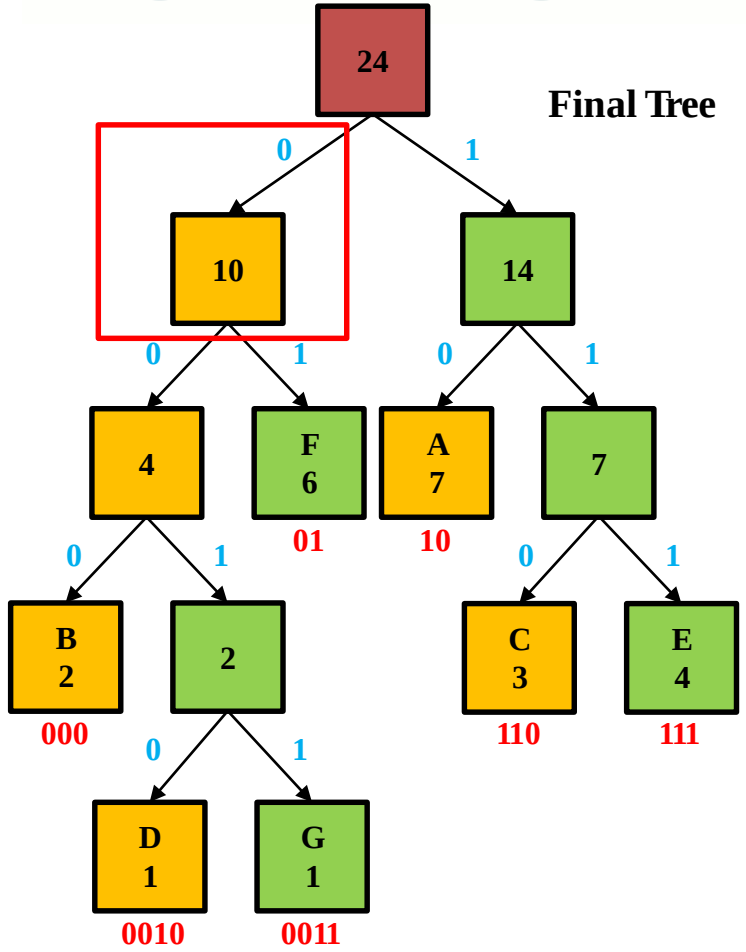
**AAAAAAABBCCCDEEEEFFFFFFG**




④ 각 가지의 왼쪽에는 0, 오른쪽에는 1을 쓴다. (**Depth First Search**를 이용하였다.)

```
void depthFirstSearch(node* tempRoot, string s)
{
    node* root1 = tempRoot;

    root1->code = s;
    if (root1 == NULL)
    {

    }
    else if (root1->leftChild == NULL && root1->rightChild == NULL)
    {
        cout << "₩t" << root1->characters << "₩t" << root1->code << endl;
    }
    else
    {
        root1->leftChild->code = s.append("0");
        s.erase(s.end() - 1);
        root1->rightChild->code = s.append("1");
        s.erase(s.end() - 1);

        depthFirstSearch(root1->leftChild, s.append("0"));
        s.erase(s.end() - 1);
        depthFirstSearch(root1->rightChild, s.append("1"));
        s.erase(s.end() - 1);
    }
}
```

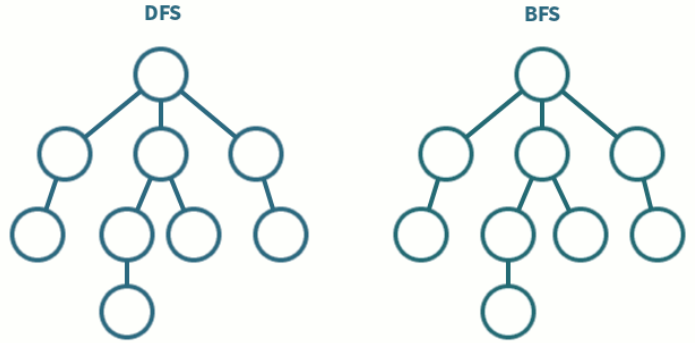
-s("")에 0 붙인 후 왼쪽 자식에게 코드 부여.
-s("0")의 마지막 인덱스 제거.

# Huffman Coding

❖ 다음 데이터 Huffman coding을 통해 압축해보자.

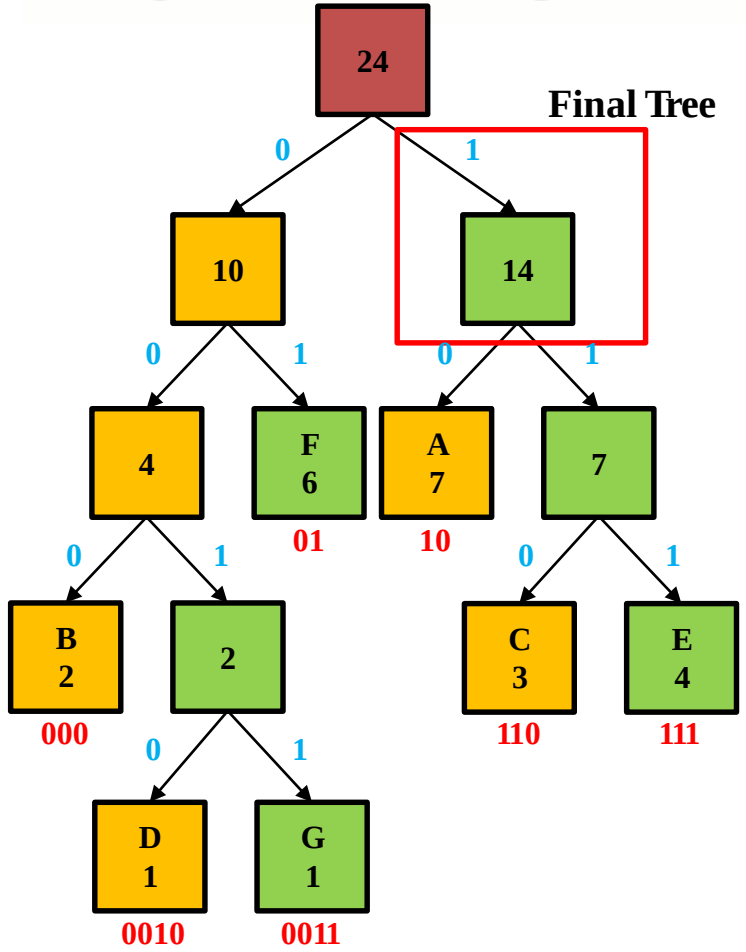
**AAAAAAABBCCCDEEEEFFFFFFG**




④ 각 가지의 왼쪽에는 0, 오른쪽에는 1을 쓴다. (**Depth First Search**를 이용하였다.)

```
void depthFirstSearch(node* tempRoot, string s)
{
    node* root1 = tempRoot;

    root1->code = s;
    if (root1 == NULL)
    {

    }
    else if (root1->leftChild == NULL && root1->rightChild == NULL)
    {
        cout << "₩t" << root1->characters << "₩t" << root1->code << endl;
    }
    else
    {
        root1->leftChild->code = s.append("0");
        s.erase(s.end() - 1);
        root1->rightChild->code = s.append("1");
        s.erase(s.end() - 1);

        depthFirstSearch(root1->leftChild, s.append("0"));
        s.erase(s.end() - 1);
        depthFirstSearch(root1->rightChild, s.append("1"));
        s.erase(s.end() - 1);
    }
}
```

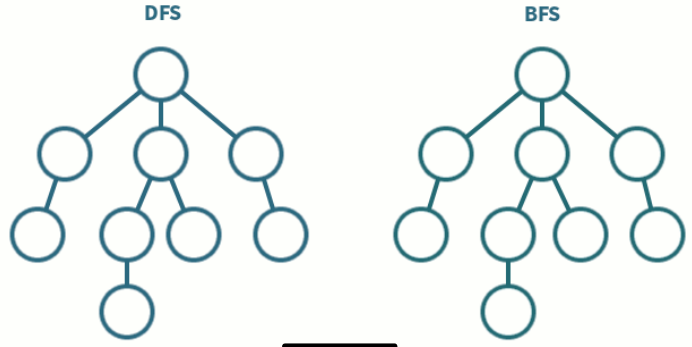
- s("")에 1 붙인 후 오른쪽 자식에게 코드 부여.
- s("1")의 마지막 인덱스 제거.

# Huffman Coding

❖ 다음 데이터 Huffman coding을 통해 압축해보자.

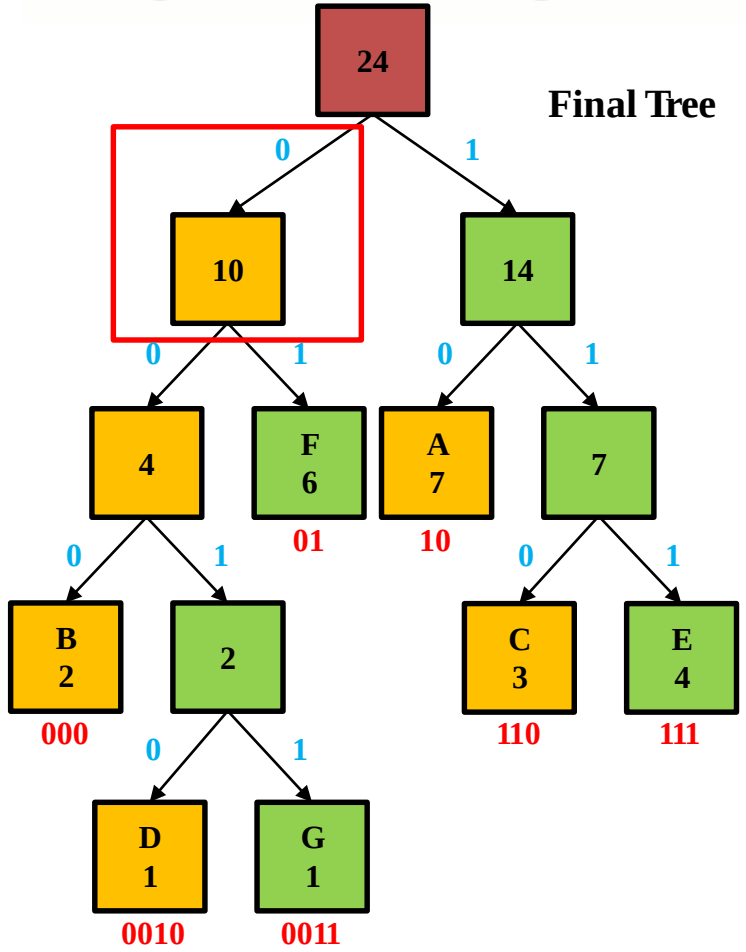
**AAAAAAABBCCCDEEEEFFFFFFG**




④ 각 가지의 왼쪽에는 0, 오른쪽에는 1을 쓴다. (**Depth First Search**를 이용하였다.)

```
void depthFirstSearch(node* tempRoot, string s)
{
    node* root1 = tempRoot;

    root1->code = s;
    if (root1 == NULL)
    {

    }
    else if (root1->leftChild == NULL && root1->rightChild == NULL)
    {
        cout << "₩t" << root1->characters << "₩t" << root1->code << endl;
    }
    else
    {
        root1->leftChild->code = s.append("0");
        s.erase(s.end() - 1);
        root1->rightChild->code = s.append("1");
        s.erase(s.end() - 1);

        depthFirstSearch(root1->leftChild, s.append("0"));
        s.erase(s.end() - 1);
        depthFirstSearch(root1->rightChild, s.append("1"));
        s.erase(s.end() - 1);
    }
}
```

- 왼쪽 자식으로 depthFirstSearch 실행.
- 왼쪽 자식의 현재 코드는 "0"

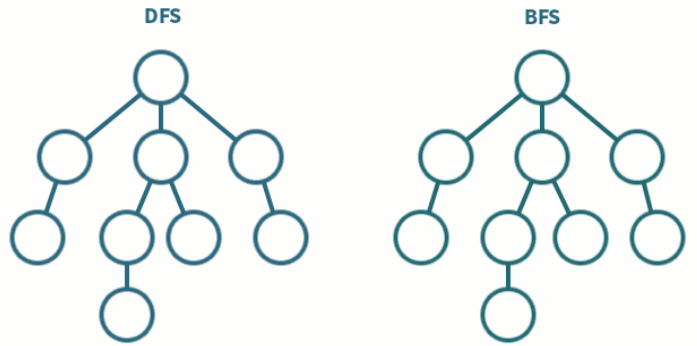



# Huffman Coding

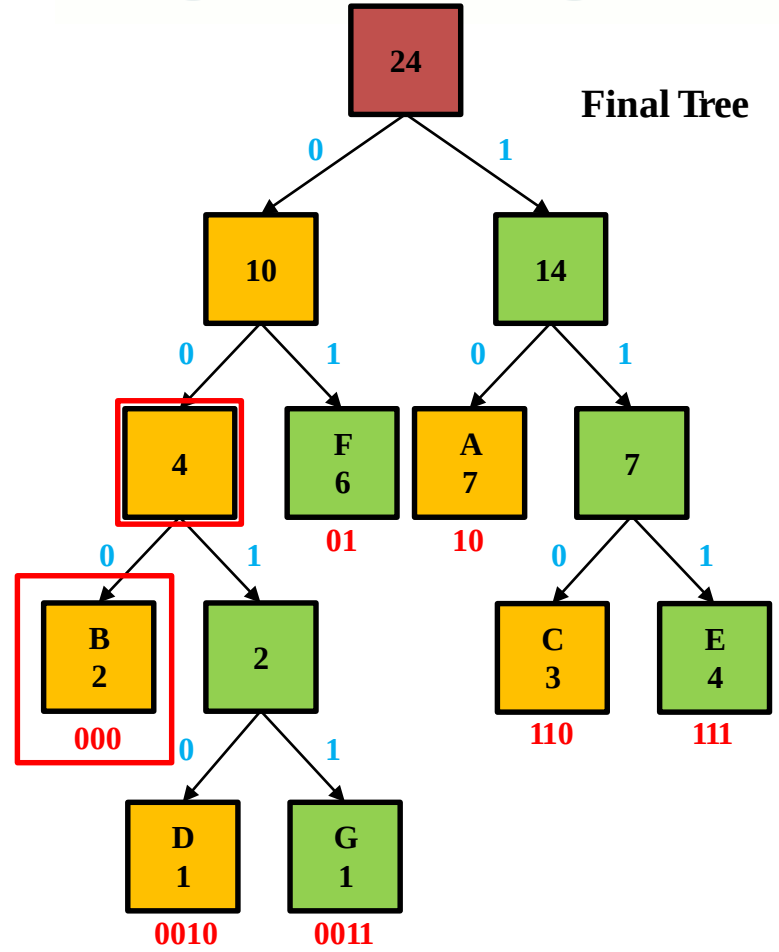
❖ 다음 데이터 Huffman coding을 통해 압축해보자.

**AAAAAAABBCCCDEEEEFFFFFFG**

④ 각 가지의 왼쪽에는 0, 오른쪽에는 1을 쓴다. (**Depth First Search**를 이용하였다.)



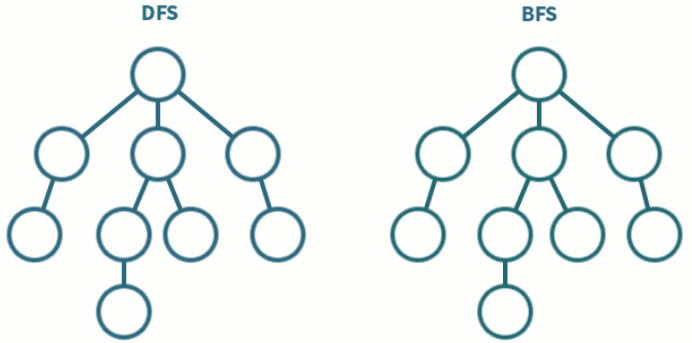

```
void depthFirstSearch(node* tempRoot, string s)
{
    node* root1 = tempRoot;

    root1->code = s;
    if (root1 == NULL)
    {

    }
    else if (root1->leftChild == NULL && root1->rightChild == NULL)
    {
        cout << "₩t" << root1->characters << "₩t" << root1->code << endl;
    }
    else
    {
        root1->leftChild->code = s.append("0");
        s.erase(s.end() - 1);
        root1->rightChild->code = s.append("1");
        s.erase(s.end() - 1);

        depthFirstSearch(root1->leftChild, s.append("0"));
        s.erase(s.end() - 1);
        depthFirstSearch(root1->rightChild, s.append("1"));
        s.erase(s.end() - 1);
    }
}
```

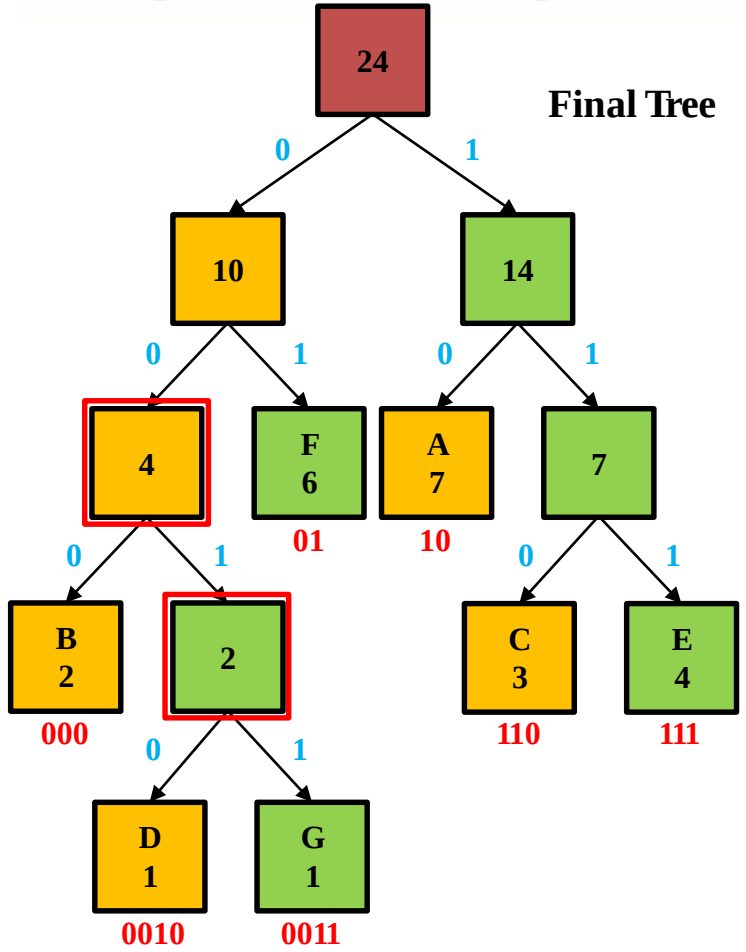
❖ 다음 데이터 Huffman coding을 통해 압축해보자.

**AAAAAAABBCCCDEEEEFFFFFFG**

④ 각 가지의 왼쪽에는 0, 오른쪽에는 1을 쓴다. (**Depth First Search**를 이용하였다.)




```
void depthFirstSearch(node* tempRoot, string s)
{
    node* root1 = tempRoot;

    root1->code = s;
    if (root1 == NULL)
    {

    }
    else if (root1->leftChild == NULL && root1->rightChild == NULL)
    {
        cout << "₩t" << root1->characters << "₩t" << root1->code << endl;
    }
    else
    {
        root1->leftChild->code = s.append("0");
        s.erase(s.end() - 1);
        root1->rightChild->code = s.append("1");
        s.erase(s.end() - 1);

        depthFirstSearch(root1->leftChild, s.append("0"));
        s.erase(s.end() - 1);
        depthFirstSearch(root1->rightChild, s.append("1"));
        s.erase(s.end() - 1);
    }
}
```

# Huffman Coding

❖ 다음 데이터 Huffman coding을 통해 압축해보자.

**AAAAAAABBCCCDEEEEEFFFFFFG**

**요약.**

① 캐릭터 종류는 몇가지인가?
② 각 캐릭터별 빈도수 입력 받고 노드를 생성한다.
③ 빈도수에 따른 hierarchy로 binary search tree를 생성한다.
④ Root부터 시작하여 각 노드별 huffman code를 부여한다.

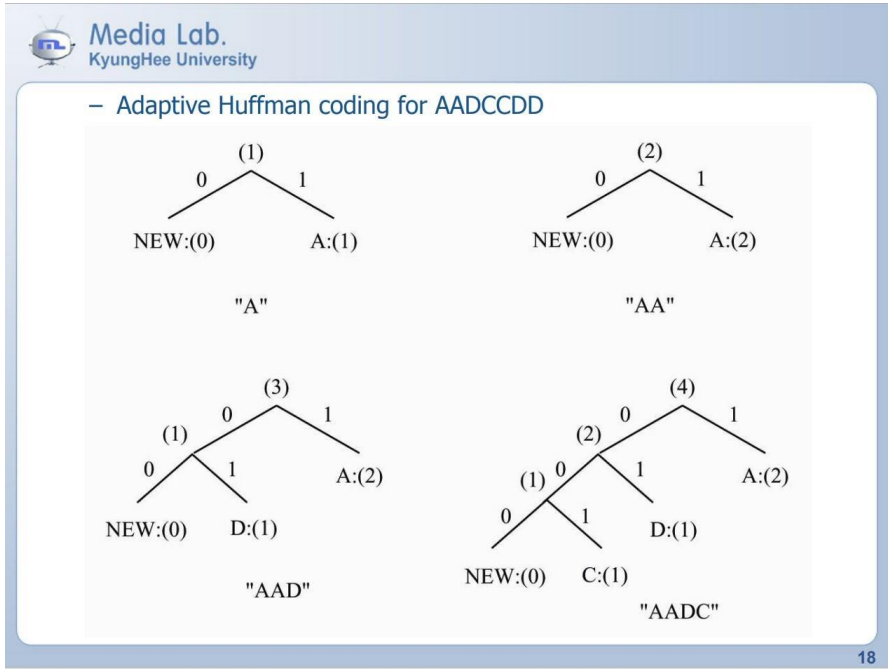
```
void getHuffmanCode()
{
    int size;
    unsigned int tempInt;
    char alphabet;

    //데이터에 사용되는 문자의 종류 개수 입력
    cout << endl;
    cout << "₩t" << "Huffman Tree : ";
    cin >> size;

    cout << endl << endl;
    cout << "₩t" << "문자 빈도" << endl;
    cout << "₩t" << "----------------------" << endl;

    //각 문자별 빈도 수 노드 생성
    for (int i = 0; i < size; i++)
    {
        cout << "₩t";
        node tempNode;
        cin >> alphabet;
        cin >> tempInt;

        tempNode.characters = alphabet;
        tempNode.frequency = tempInt;
        tempNode.leftChild = NULL;
        tempNode.rightChild = NULL;
        nodeArray.push_back(tempNode);
    }

    //Huffman Tree 생성
    node root = getHuffmanTree();

    cout << endl << endl;
    cout << "₩t" << "문자" << "₩t" << "Code" << endl;
    cout << "₩t" << "----------------------" << endl;

    //Huffman Coding Table 생성
    depthFirstSearch(&root, "");
}
```

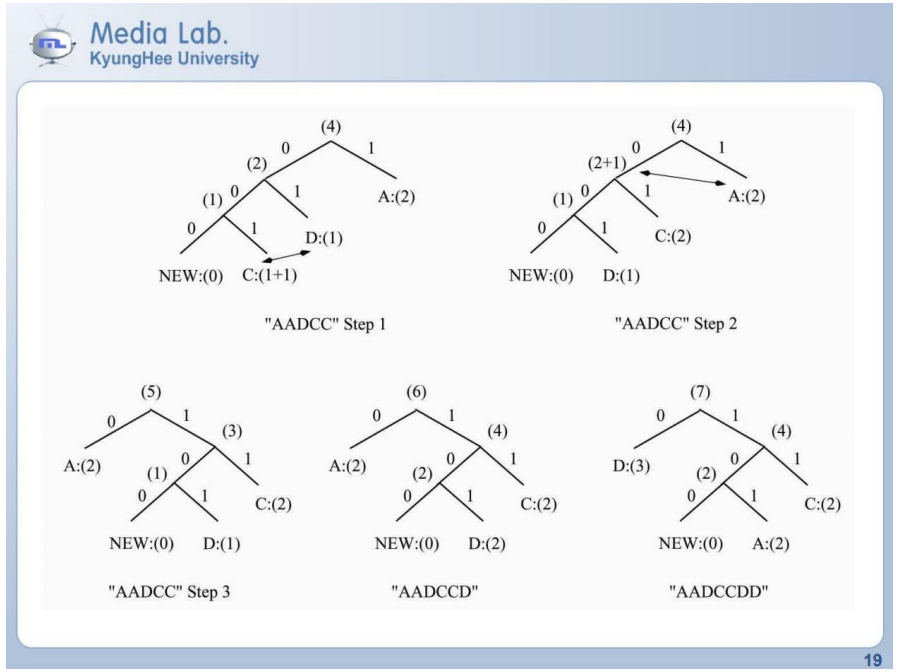
① ② ③ ④

# Coding Homework

## ❖ Assignment

- Realization of **Adaptive Huffman Coding** in the lecture note




```
#include "adpative_huffman.h"

int main()
{
    get_adaptive_huffman_code();

    return 0;
}
```

- 인터페이스는 우측 그림과 유사하게 만들어주세요

```
input : A

        data      count
        ------------
        A         1

        data      code
        ------------
        NEW       0
        A         1
```

```
input : B

        data      count
        ------------
        A         3
        B         1

        data      code
        ------------
        NEW       00
        B         01
        A         1
```

# Coding Homework

Lab 1.

| 제출일 | | 학번 | |
|---|---|---|---|
| 전공 | | 이름 | |

| | |
|---|---|
| ① 문제 | |
| ② 주요 변수 | |
| ③ 알고리즘 | |
| ④ 결과 | |

1. 파일명 : "실감미디어_1st_학번_이름.zip"
   - EX) 실감미디어_1st_2023000100_홍길동.zip
2. 압축 파일 구성
   - 소스 파일, .hwp(.doc) 2개 파일 (Python 의 경우)
   - 소스 파일, .hwp(.doc), .exe 3개 파일 (C/C++ 의 경우)
3. 파일 내용
   - 보고서 : 이름, 학번, 문제, 주요 변수, 알고리즘, 결과
   - (.py, .ipynb)파이썬 소스코드 or (.cpp, .c) C/C++ 소스 코드
   - .exe : 실행파일 (C/C++ 의 경우만 해당)
4. 주의 사항
   - 제출이 늦을 경우 감점은 있지만, 학기 종료까지 제출하시면 점수가 있습니다.

조　교 : 권도완
이메일 : kdwys97@khu.ac.kr
연구실 : Media Lab.3 (전자정보대학 567호)

# Appendix

# Visual Studio Community Install (Window)

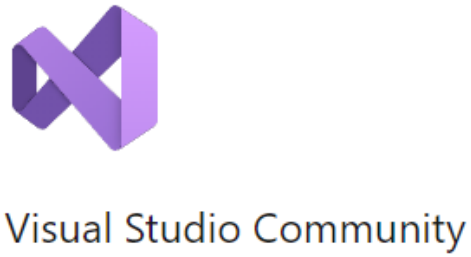
# Visual Studio 설치 (Window)

## ❖Visual Studio

VS 마이크로 소프트에서 개발한 통합 개발 환경(IDE)

윈도우, 리눅스, macOS에서 작동하며, 다양한 언어(C, C#, C++, Python, node.js 등)로 프로그래밍이 가능합니다.

다운로드 페이지 : https://visualstudio.microsoft.com/ko/free-developer-offers/



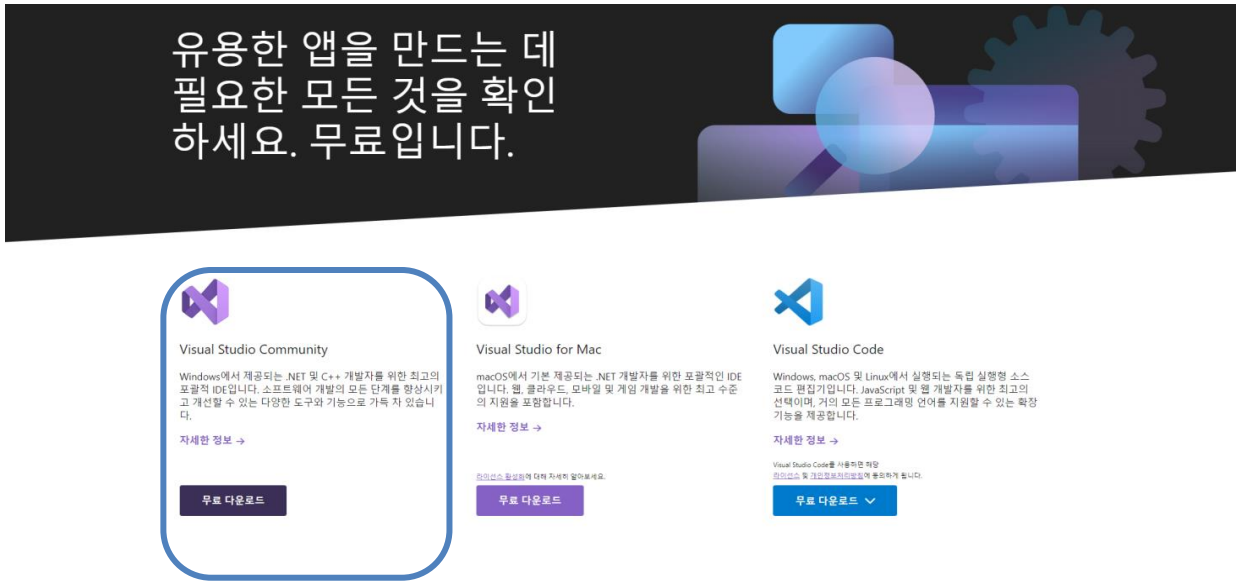
# Visual Studio 설치 (Window)

## ❖Visual Studio

Visual Studio Community : 개인 사용자용 무료 version

Visual Studio Professional : 유료 version

Visual Studio Enterprise : Professional로는 대규모 개발이 충분하지 않은 사용자를 위한 버전



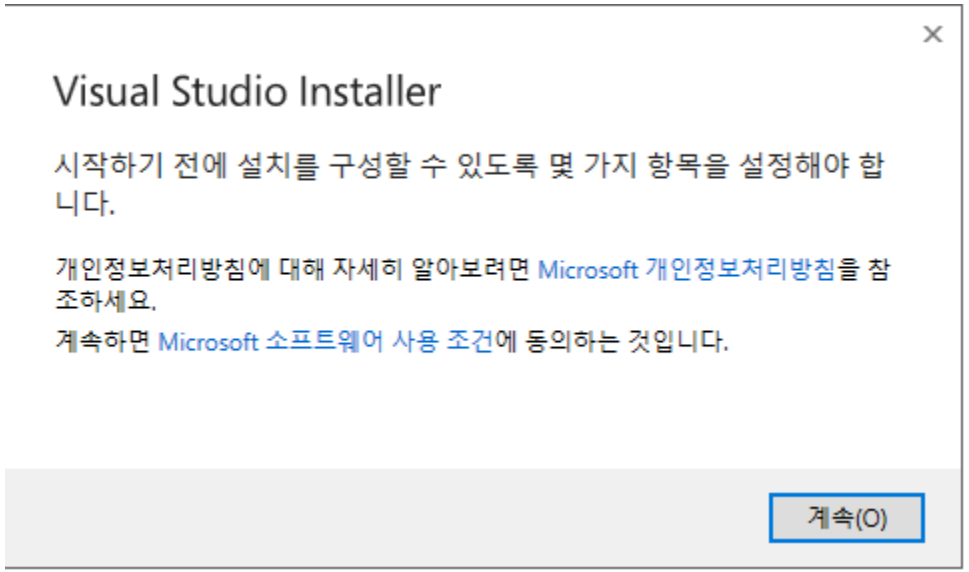

Microsoft 개발자 클라우드의 일부

# Visual Studio 설치 (Window)

## ❖Visual Studio(Window)

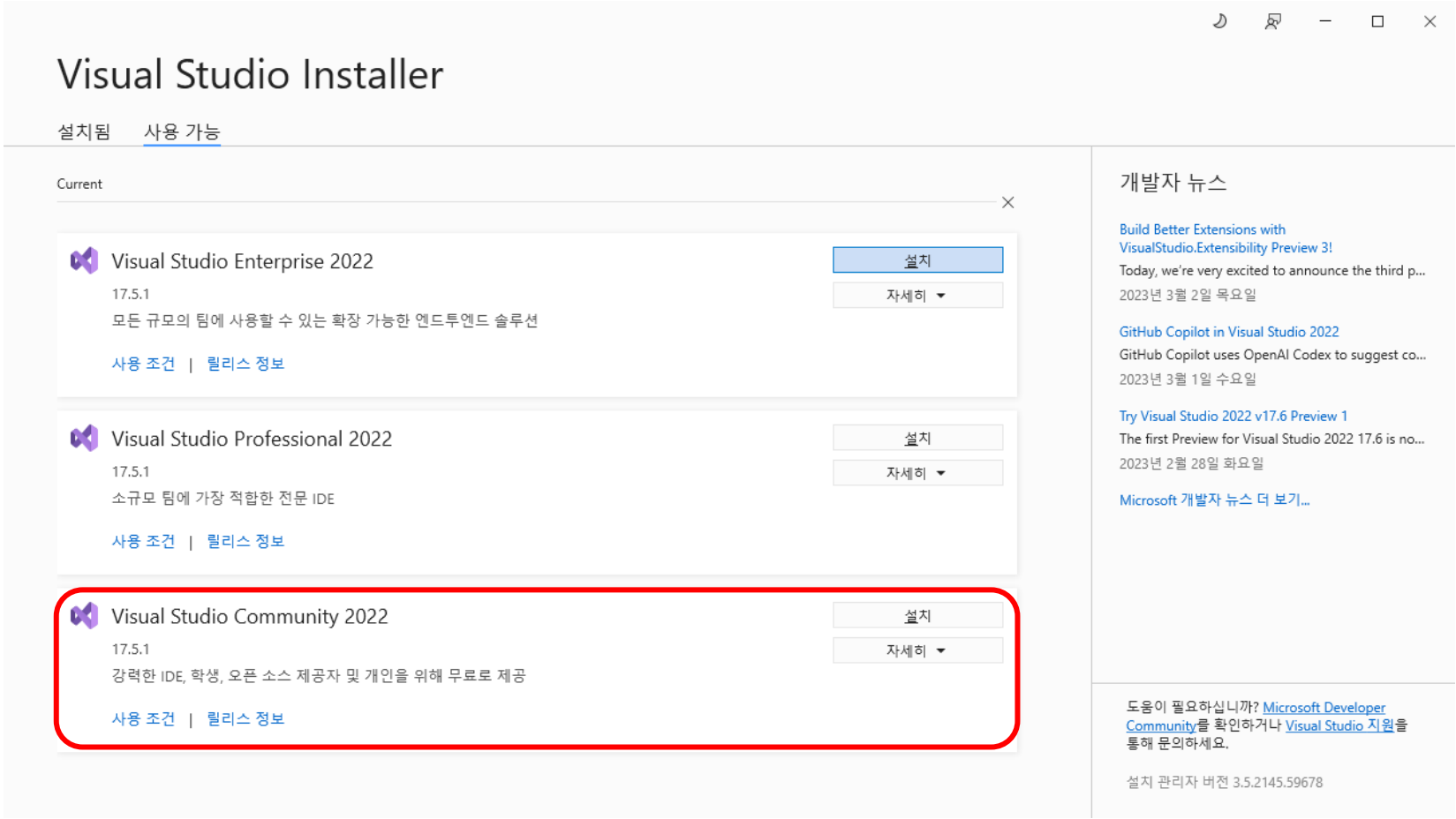
VisualStudioSetup.exe 파일 실행

해당 파일을 설치를 시작하면 아래와 같은 창 발생 : "계속" 클릭



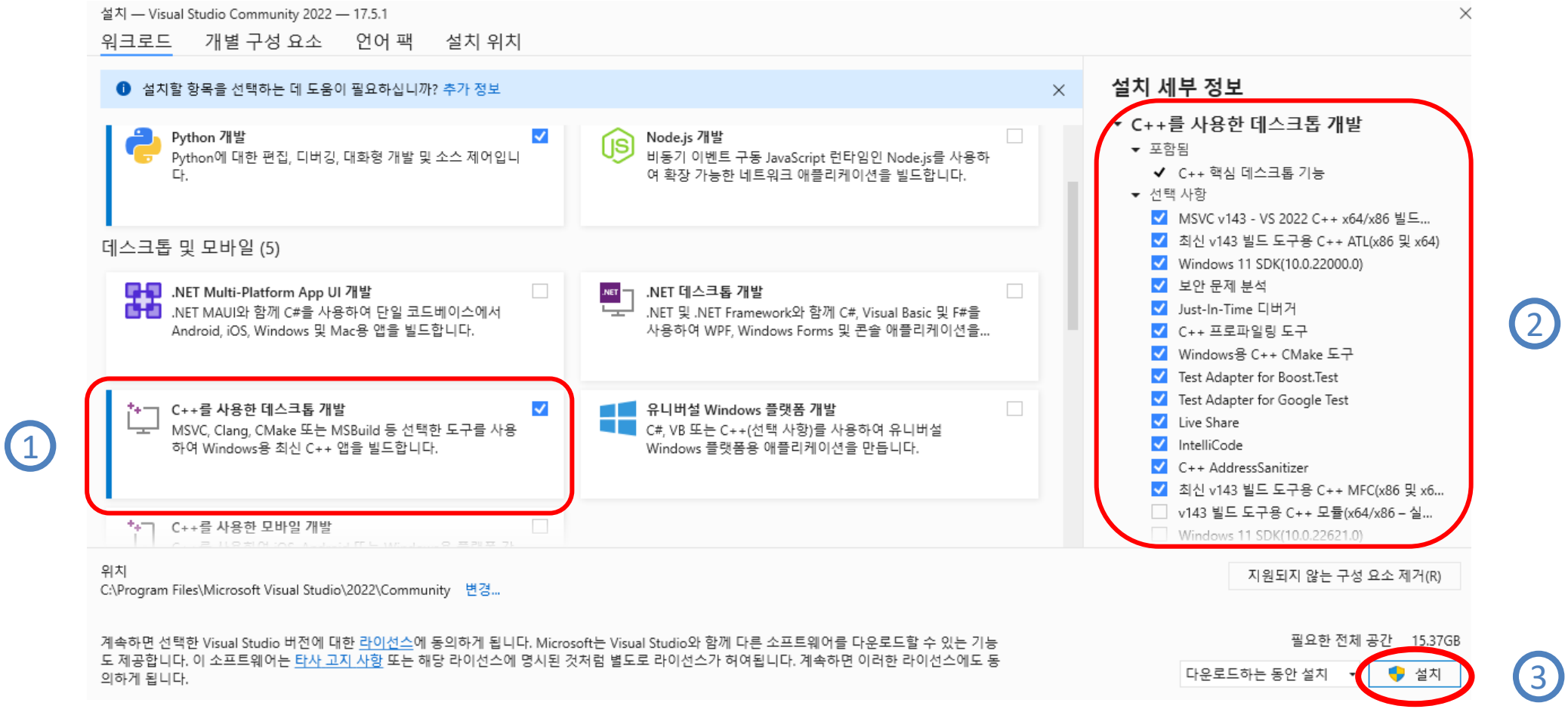
# Visual Studio 설치 (Window)

## ❖Visual Studio(Window)



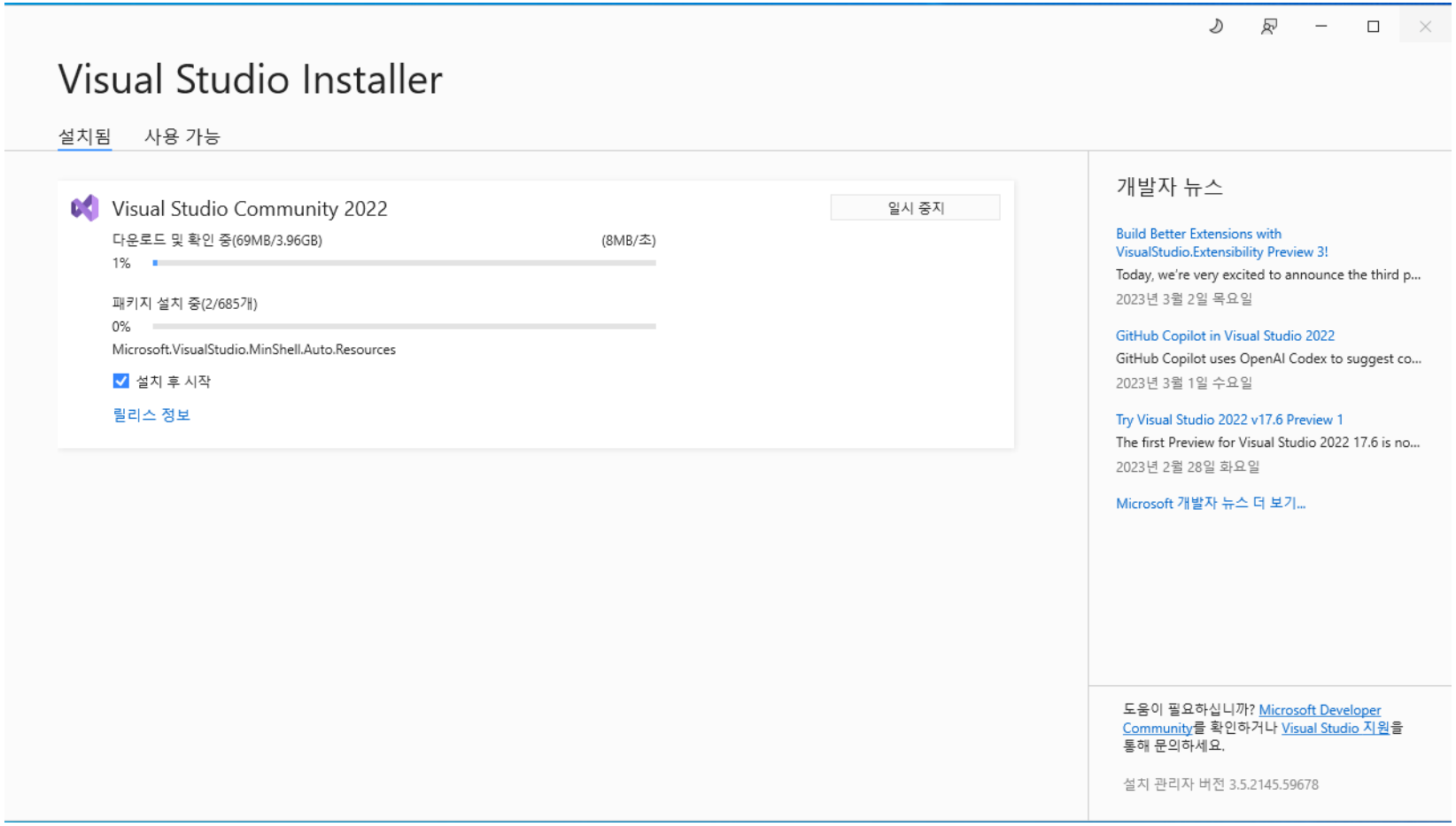
# Visual Studio 설치 (Window)

## ❖Visual Studio(Window)



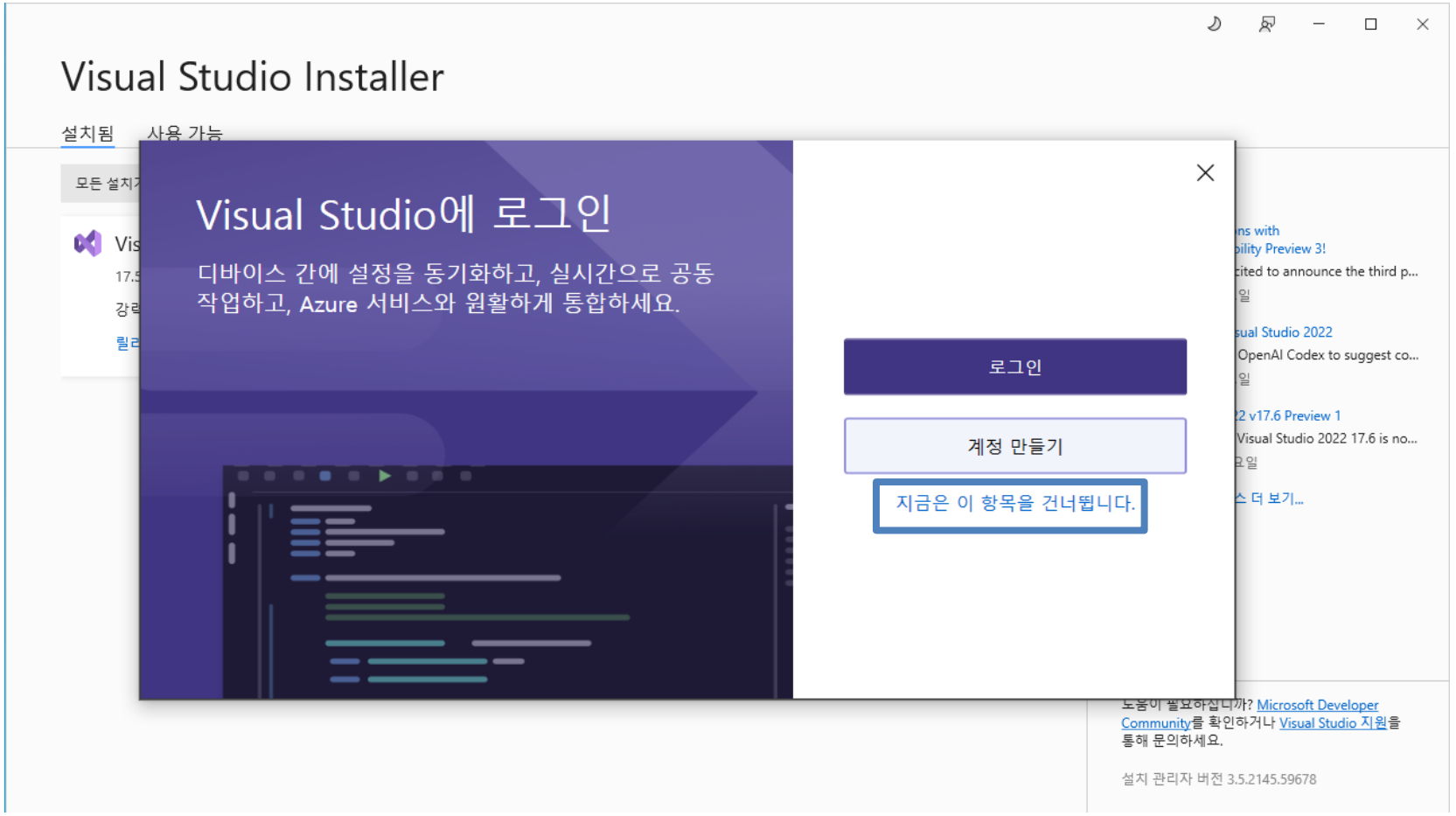
# Visual Studio 설치 (Window)

## ❖Visual Studio(Window)



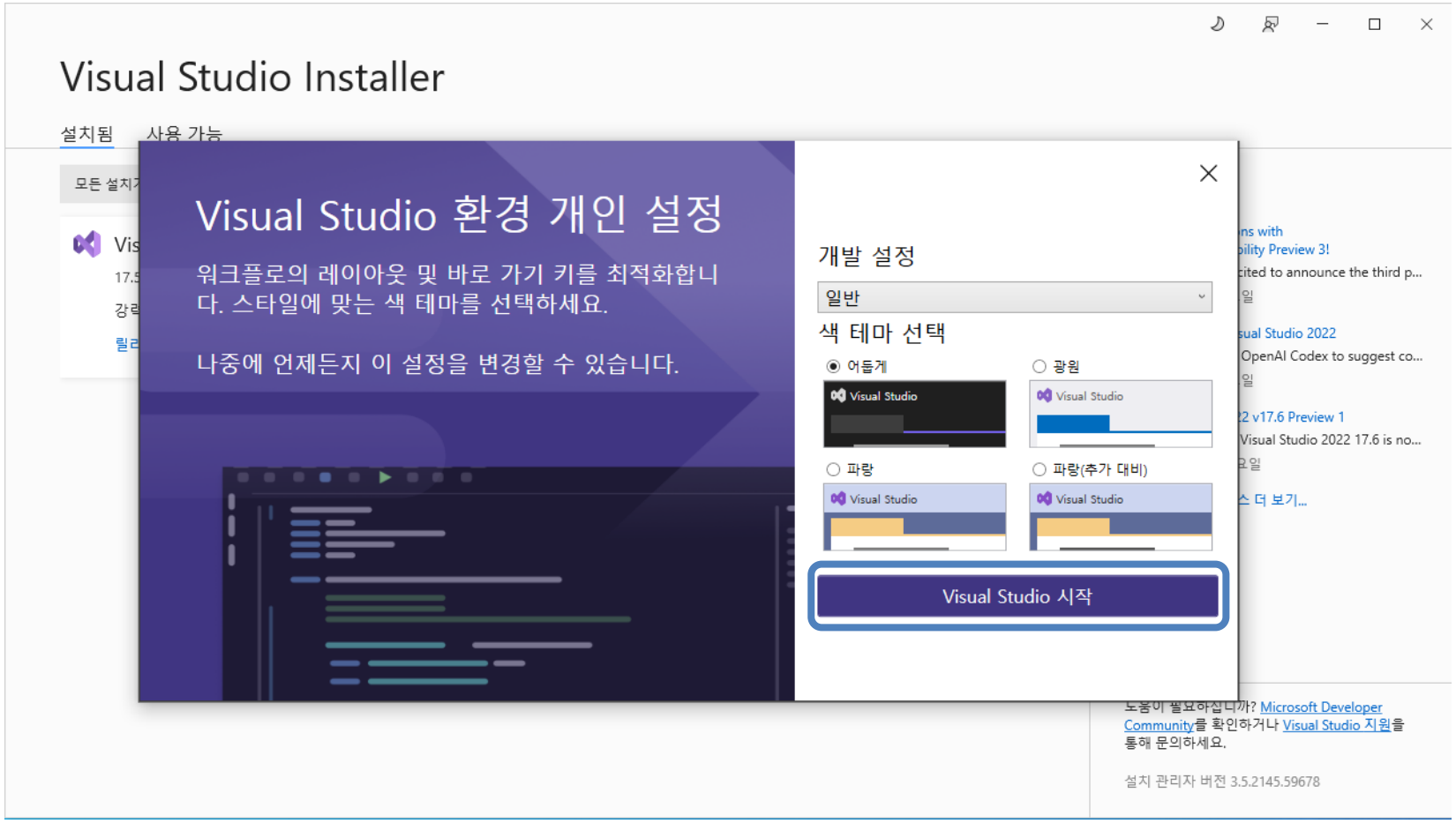
# Visual Studio 설치 (Window)

## ❖Visual Studio(Window)



# Visual Studio 설치 (Window)

## ❖Visual Studio(Window)



# 프로젝트 생성

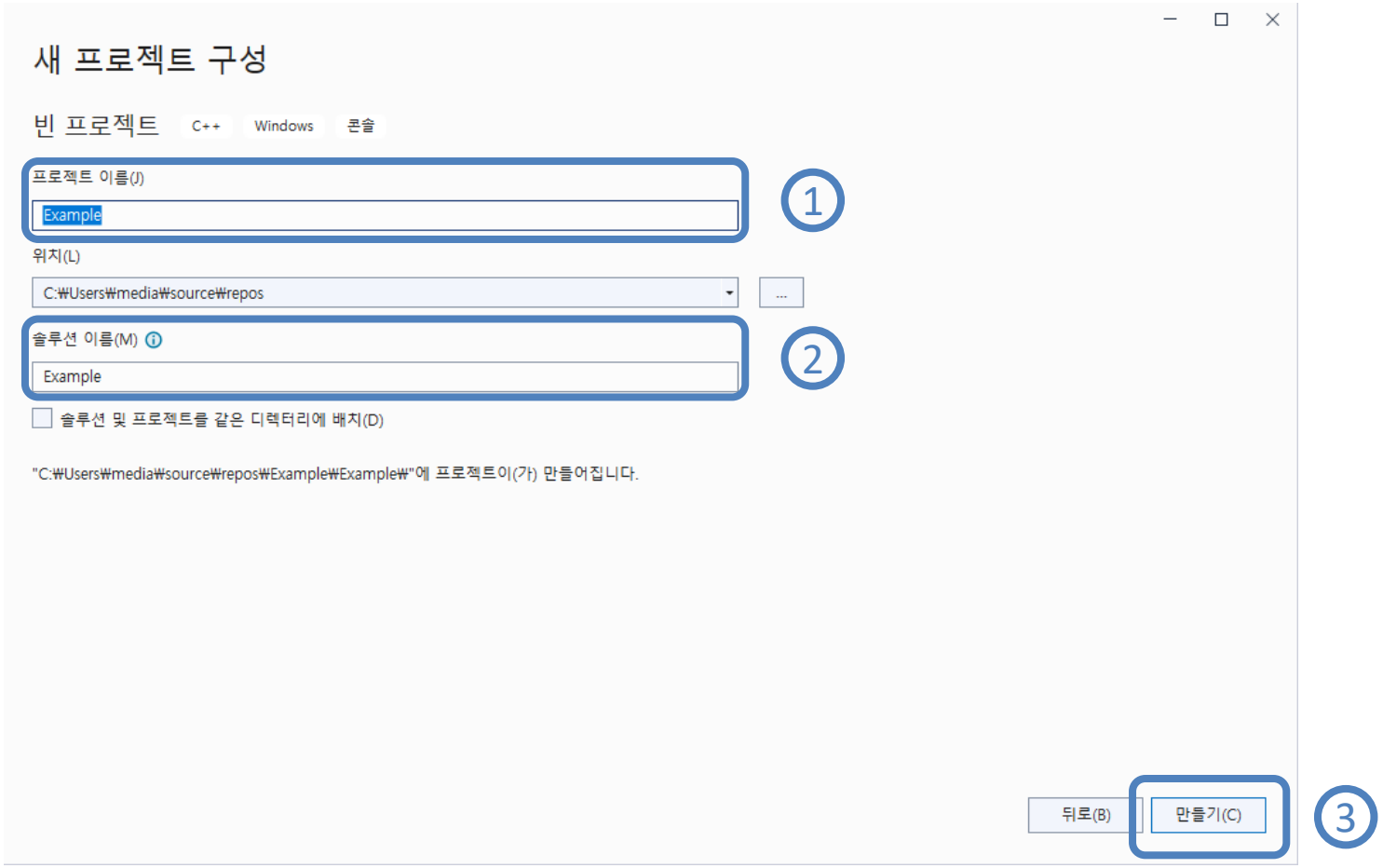
## ❖Visual Studio(Window)

# 프로젝트 생성

## ❖Visual Studio(Window)

# 프로젝트 생성

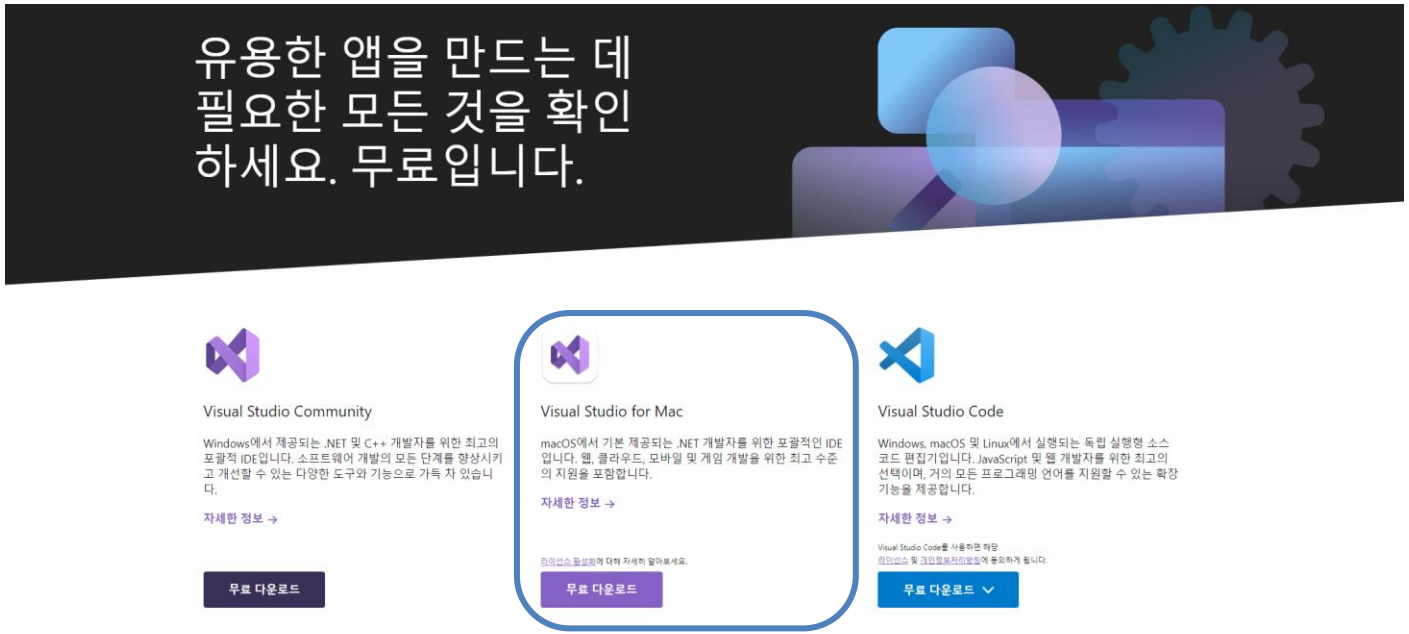
## ❖Visual Studio(Window)

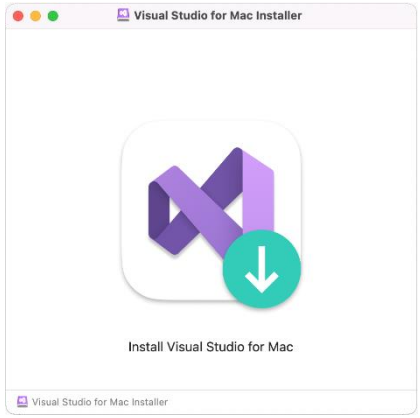


# Visual Studio Community Install (Mac)

# Visual Studio 설치 (Mac)

## ❖Visual Studio(Mac)

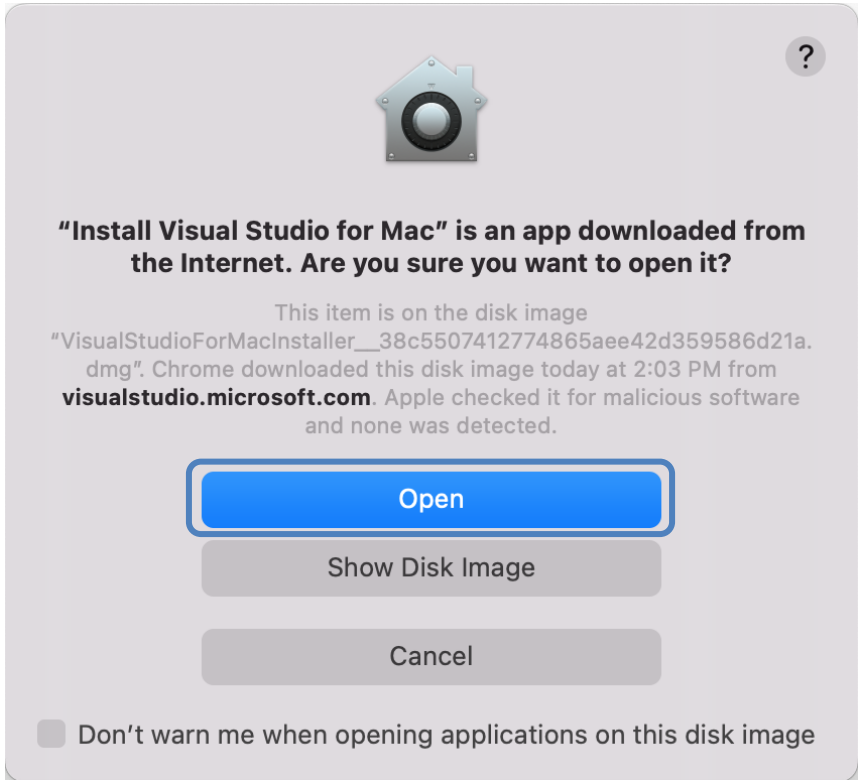



Microsoft 개발자 클라우드의 일부

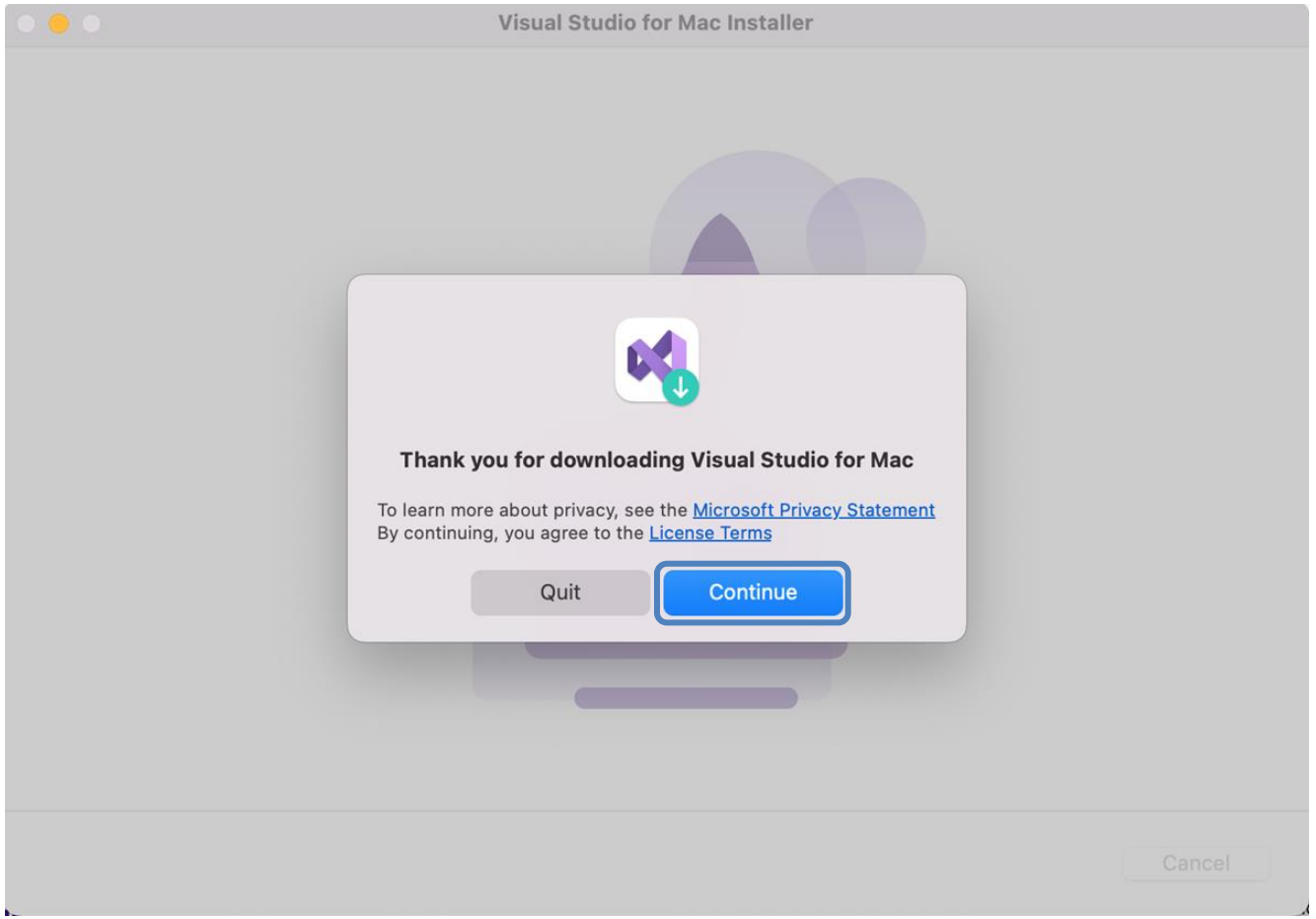
# Visual Studio 설치 (Mac)

## ❖Visual Studio(Mac)



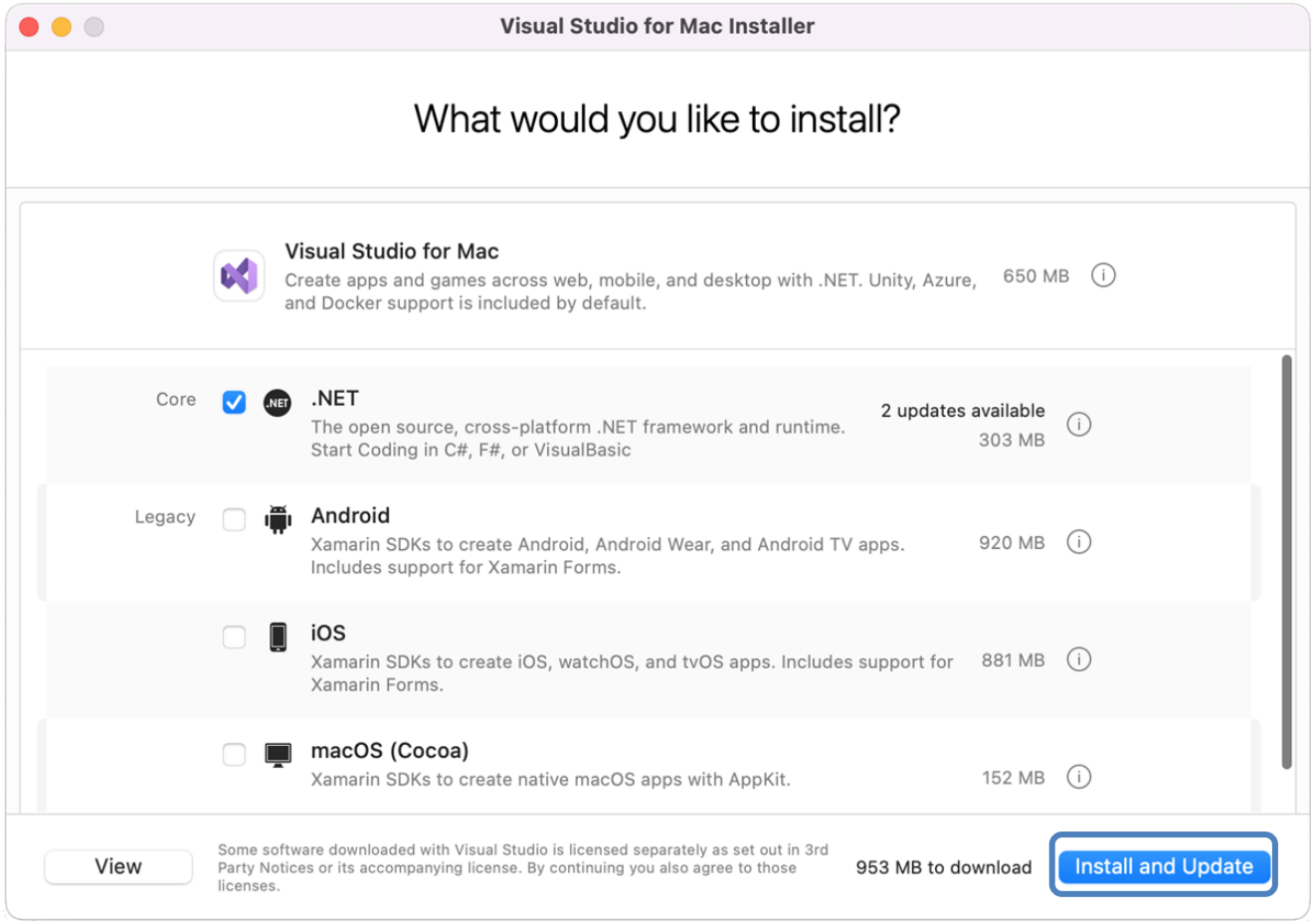
# Visual Studio 설치 (Mac)

## ❖Visual Studio(Mac)



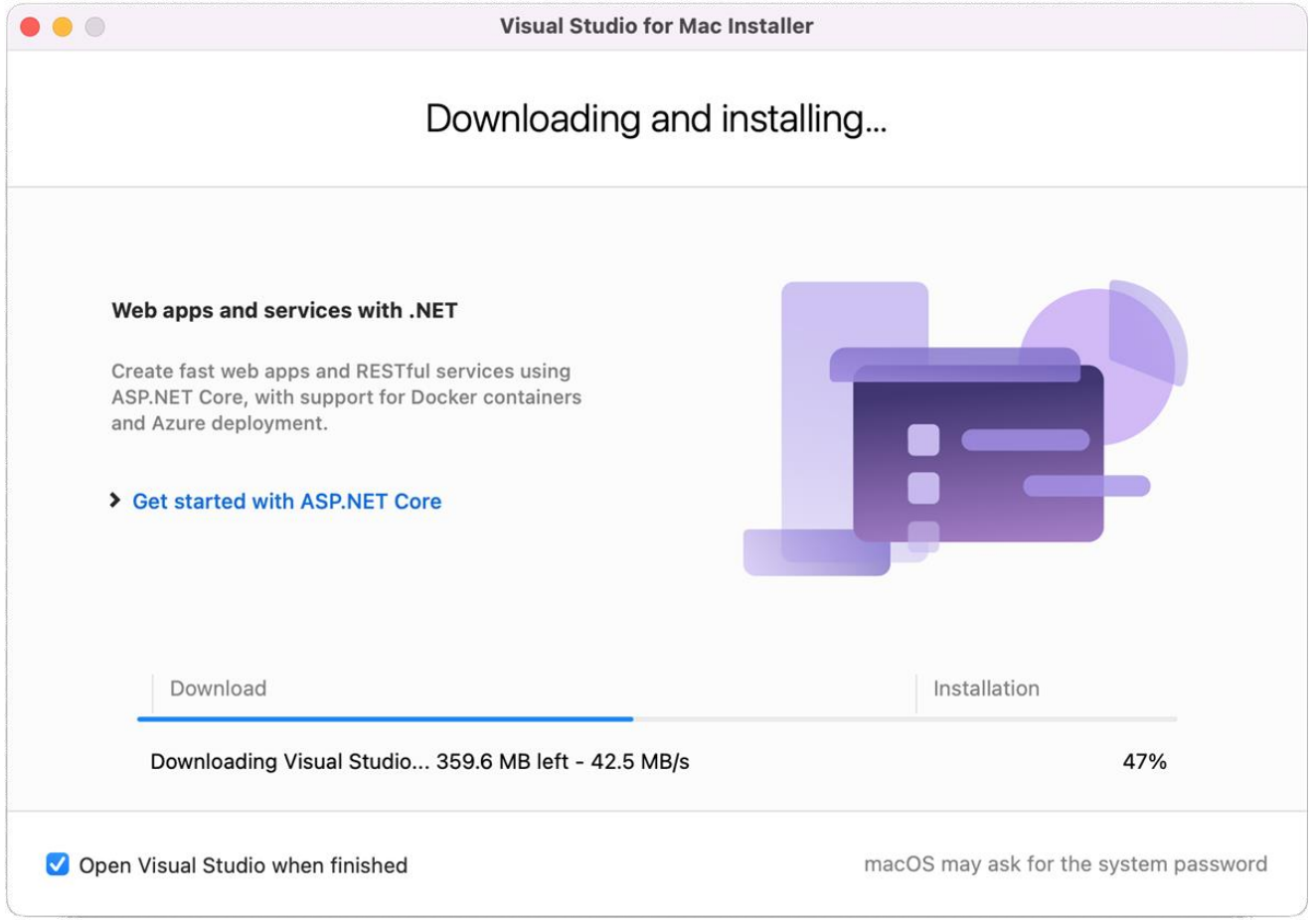
# Visual Studio 설치 (Mac)

## ❖Visual Studio(Mac)



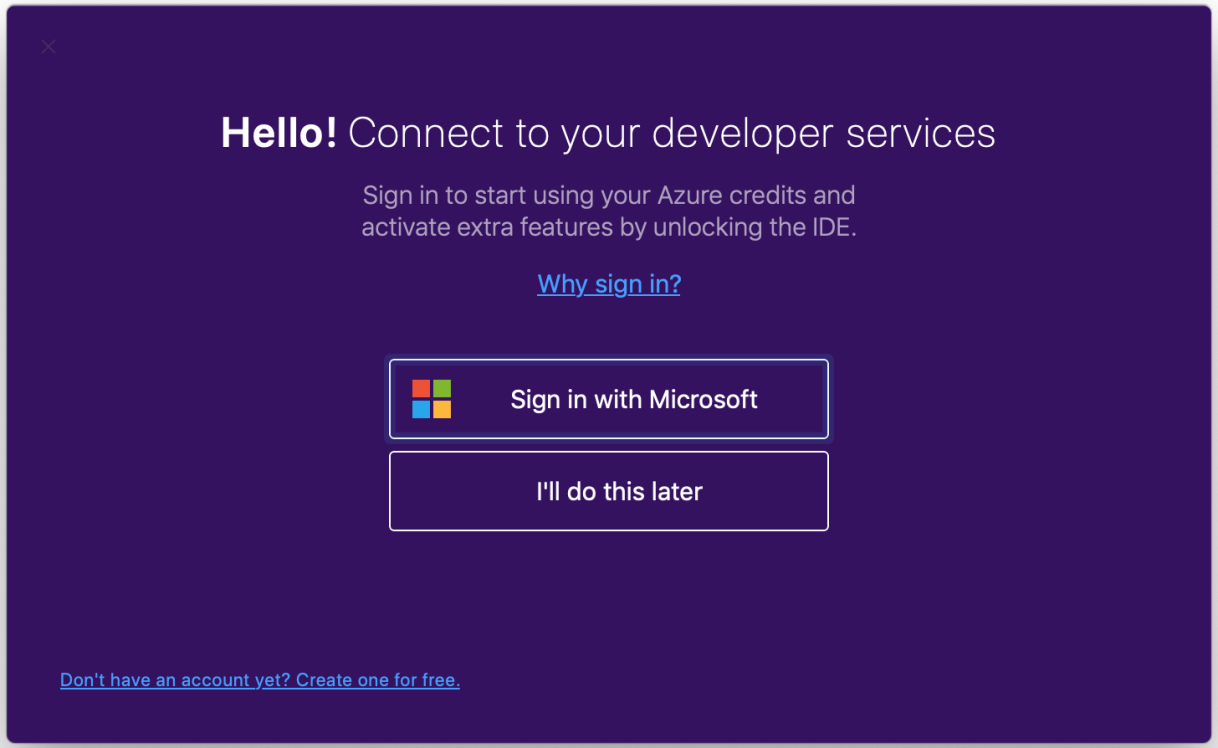
# Visual Studio 설치 (Mac)

## ❖Visual Studio(Mac)



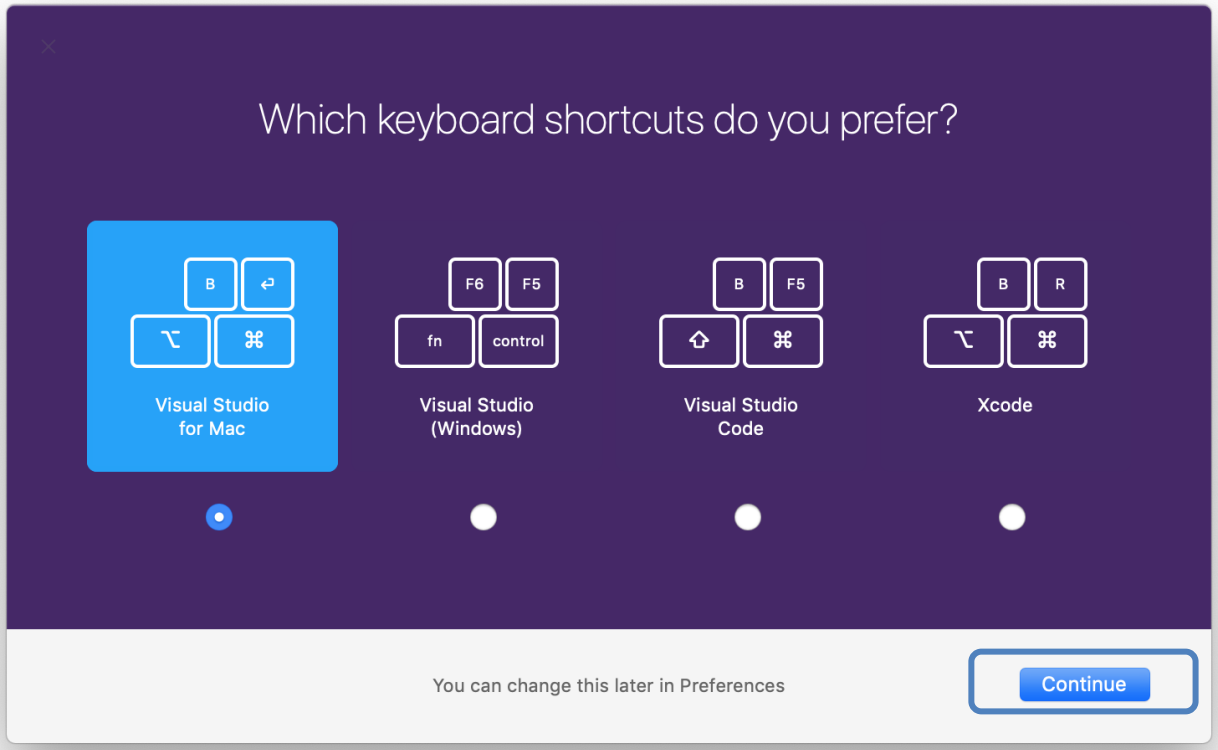
# Visual Studio 설치 (Mac)

## ❖Visual Studio(Mac)



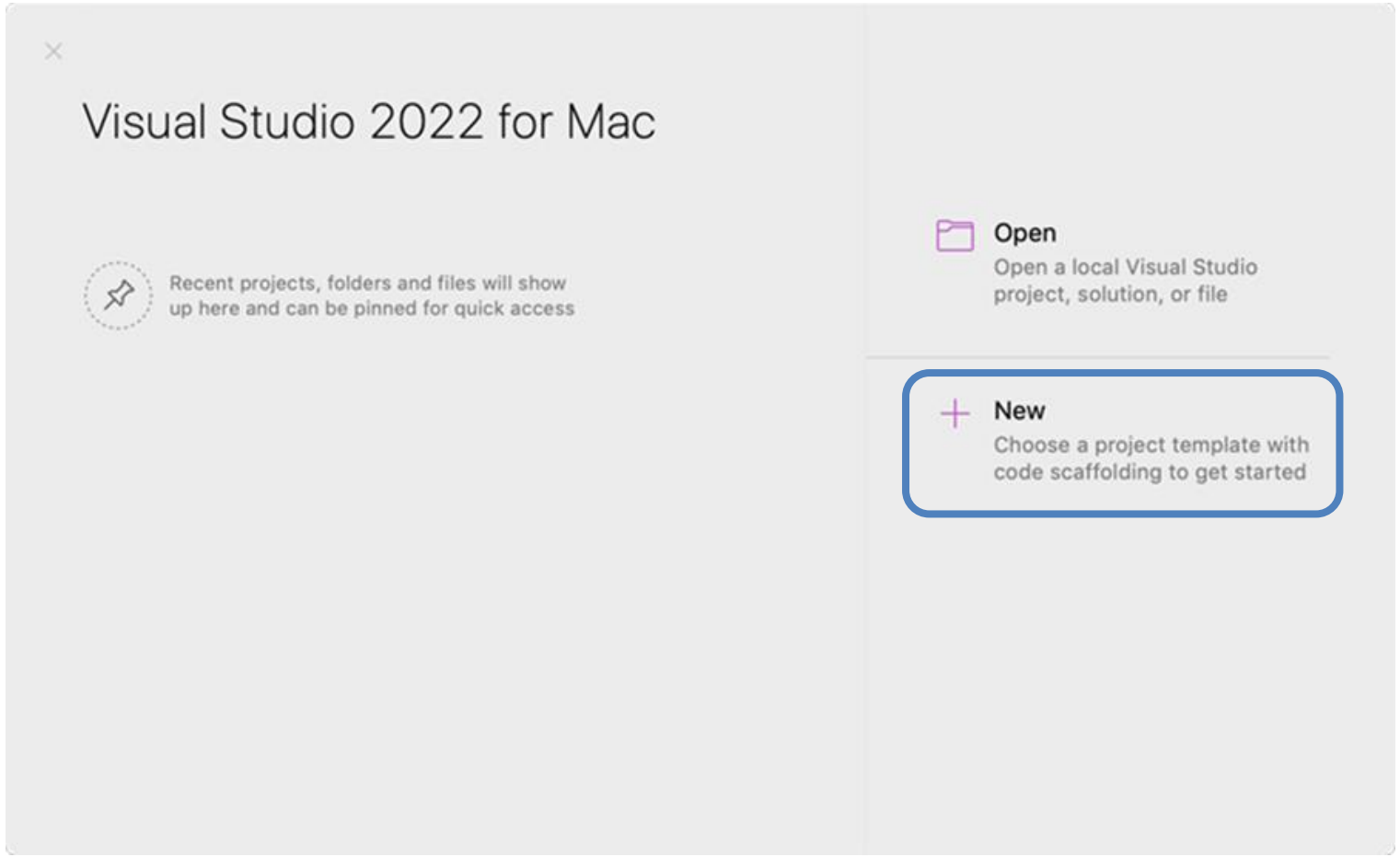
# Visual Studio 설치 (Mac)

## ❖Visual Studio(Mac)



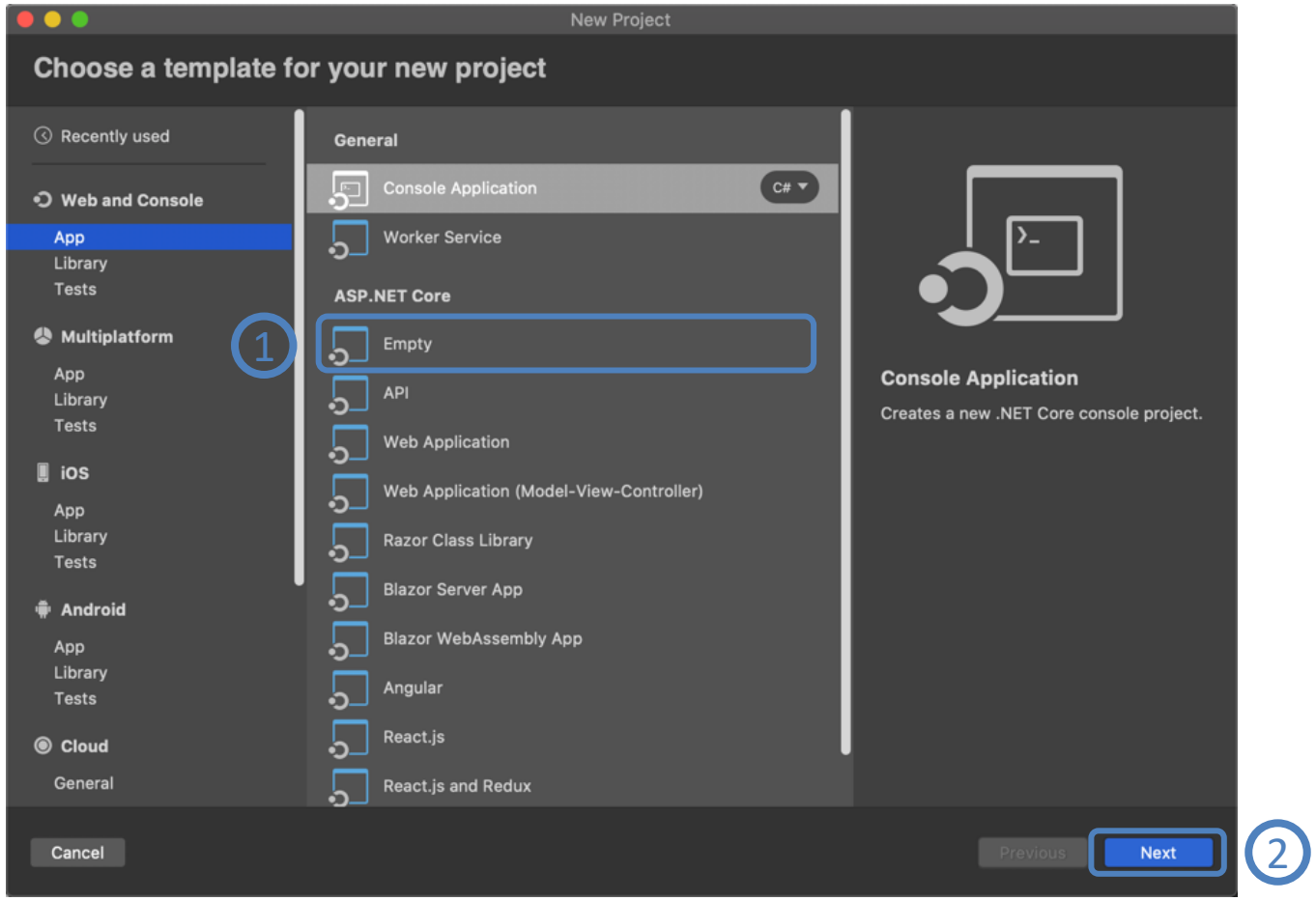
# 프로젝트 생성

## ❖Visual Studio(Mac)



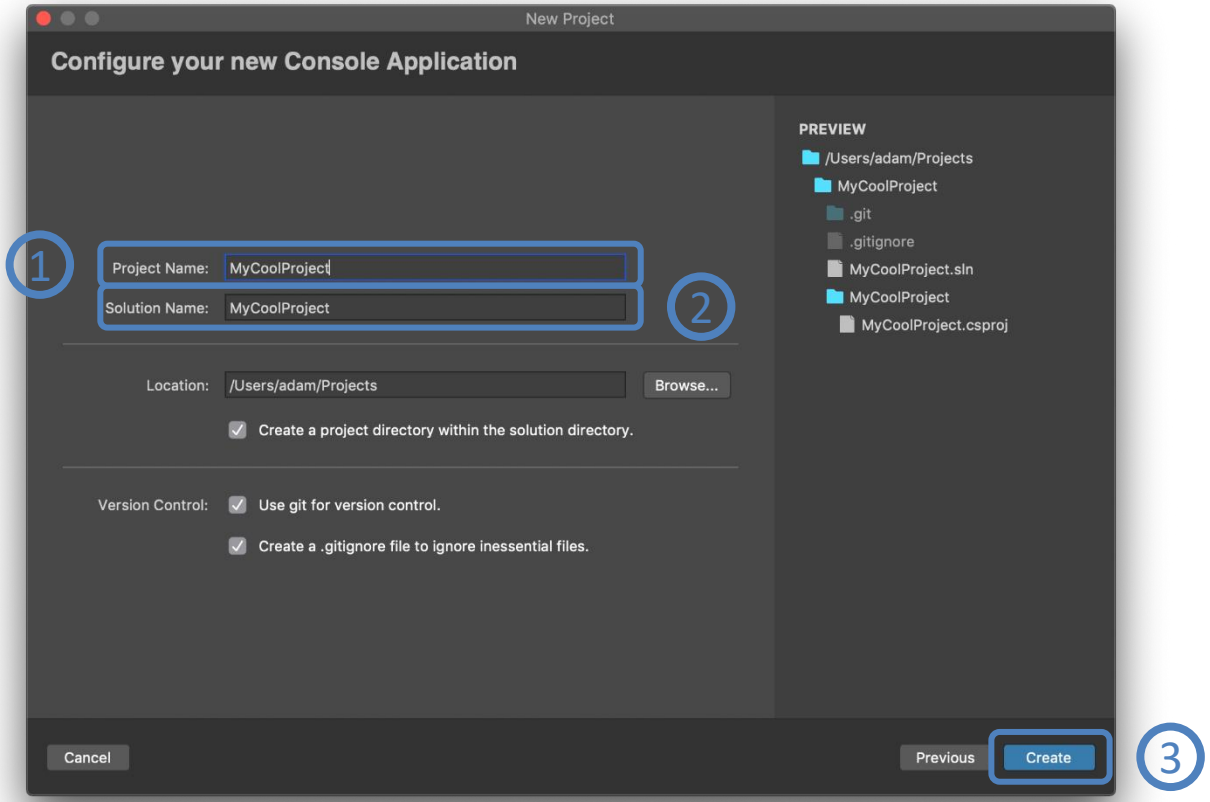
# 프로젝트 생성

## ❖Visual Studio(Mac)



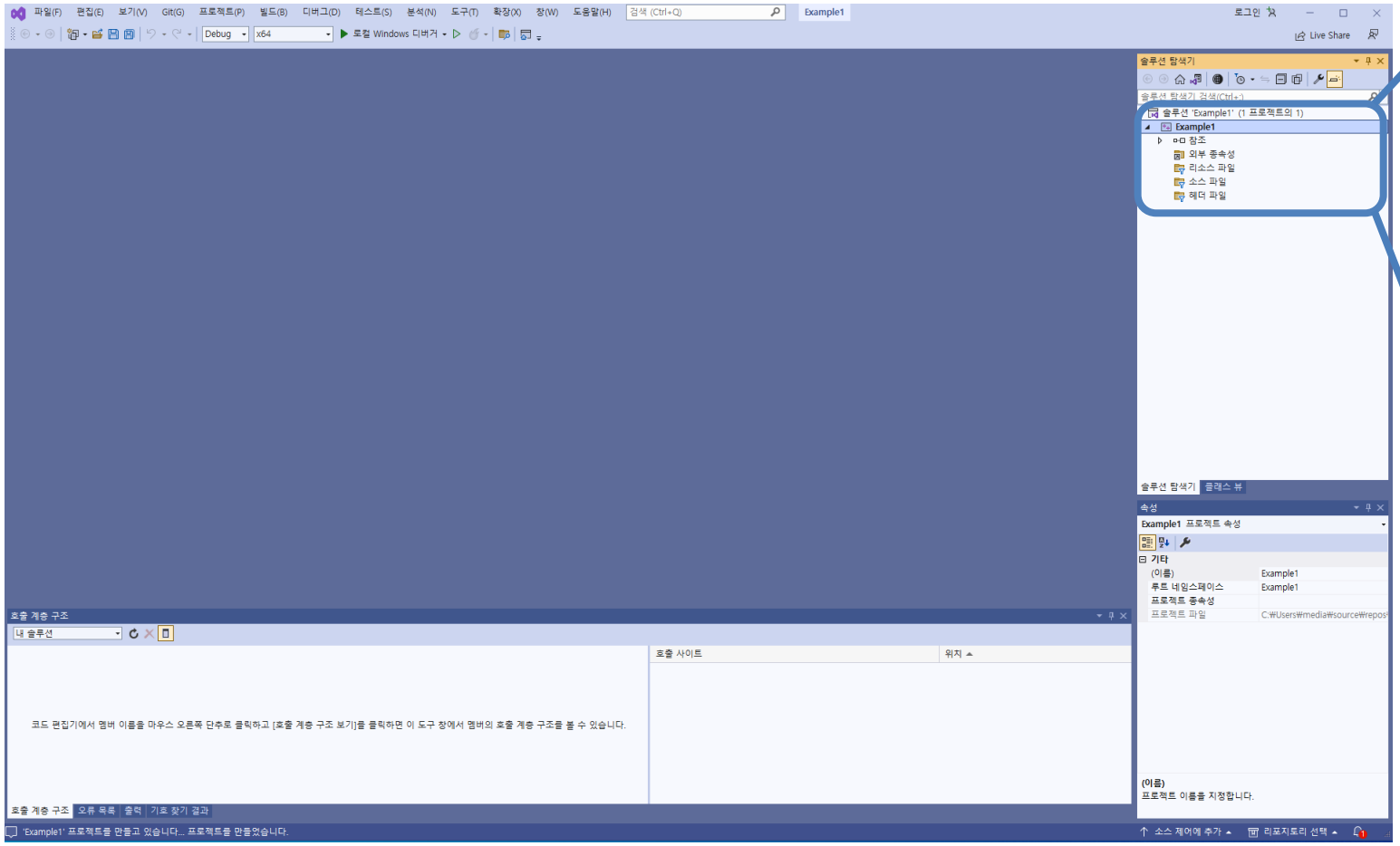
# 프로젝트 생성

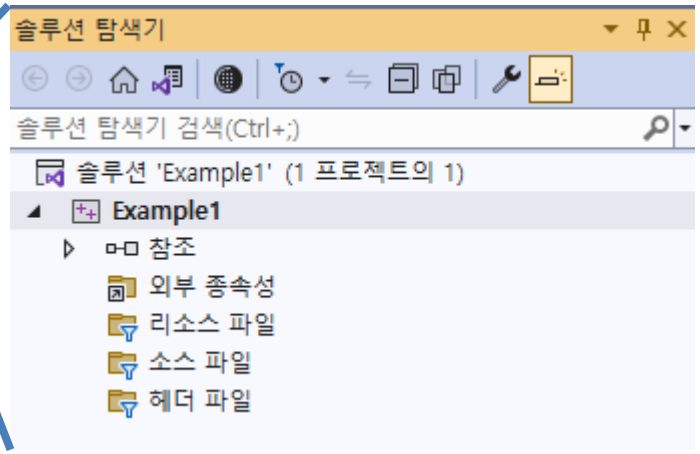
## ❖Visual Studio(Mac)



# 소스 파일(CPP) 추가

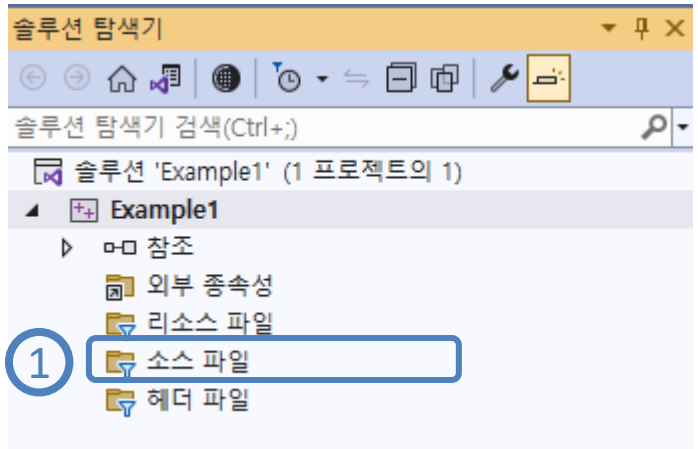
## ❖Visual Studio

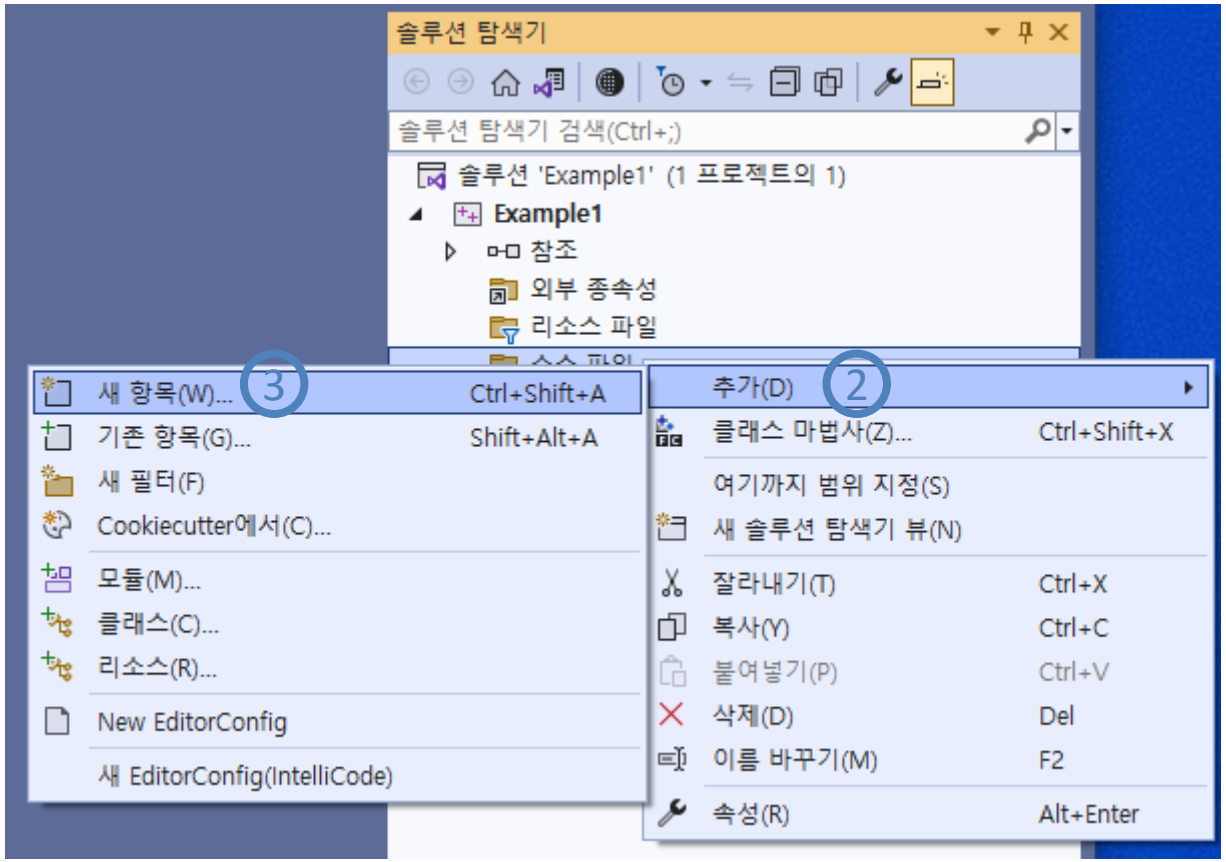




# 소스 파일(CPP) 추가

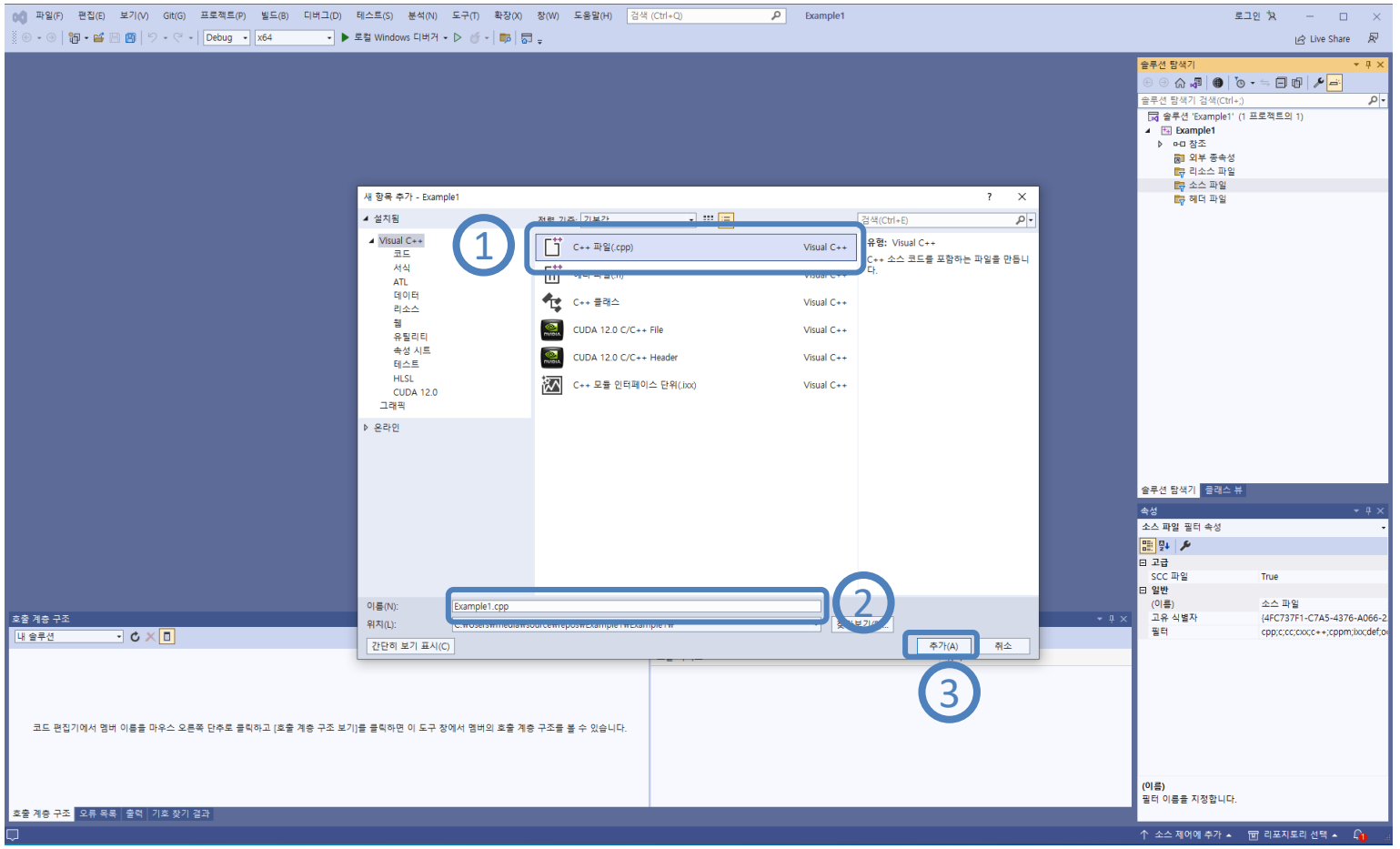
## ❖Visual Studio



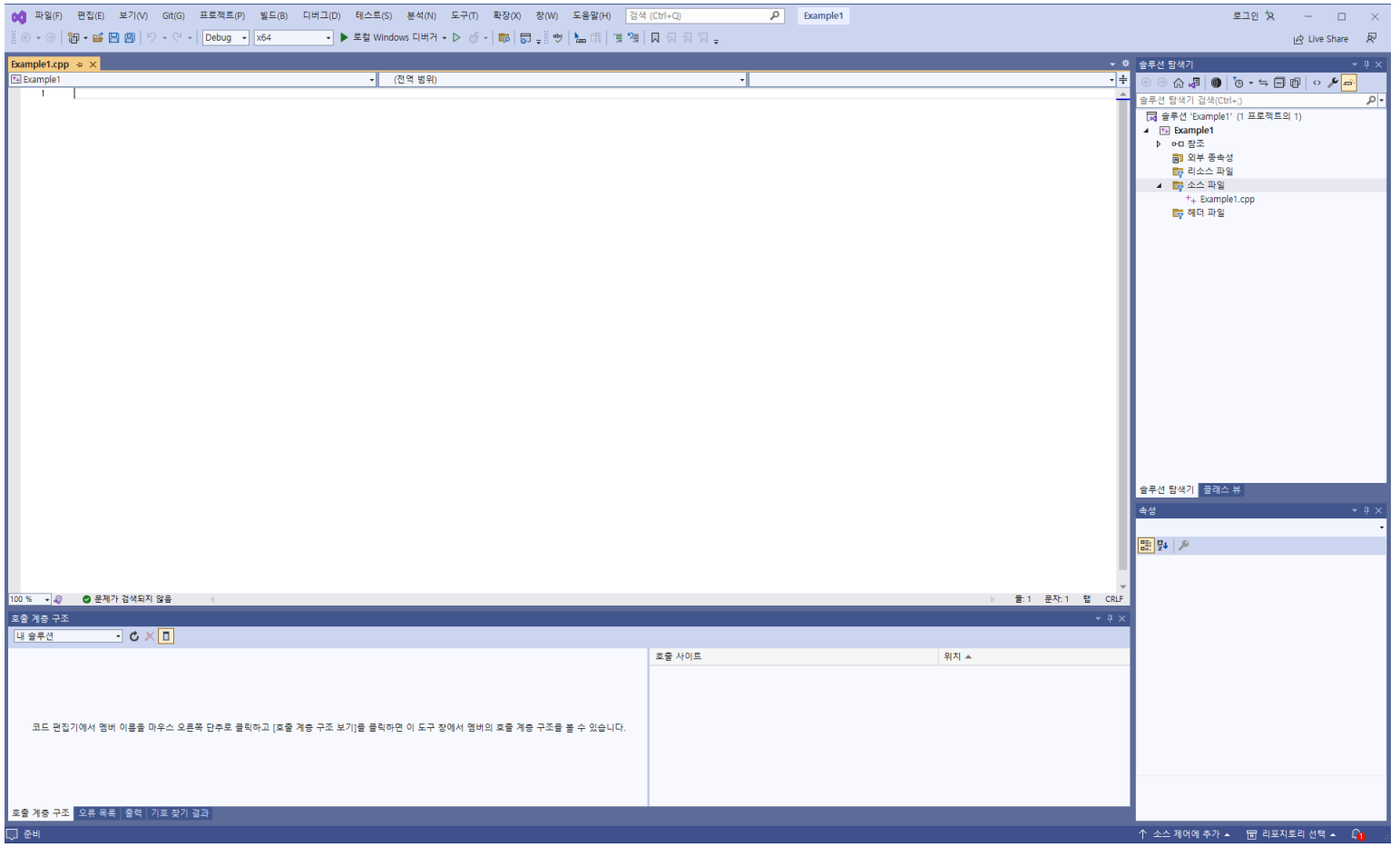
# 소스 파일(CPP) 추가

## ❖Visual Studio



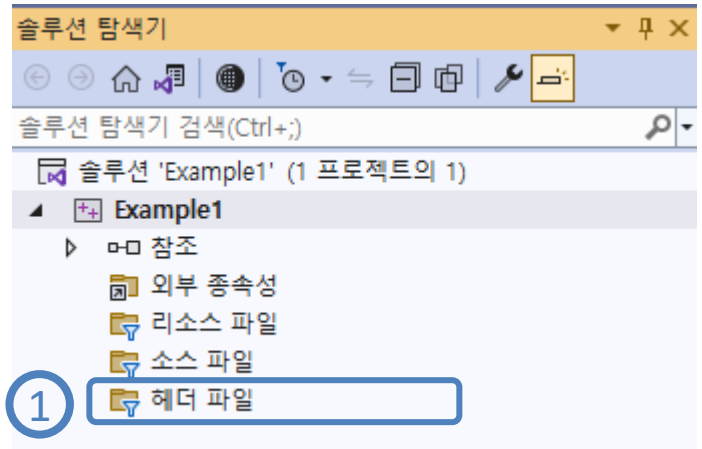
# 소스 파일(CPP) 추가

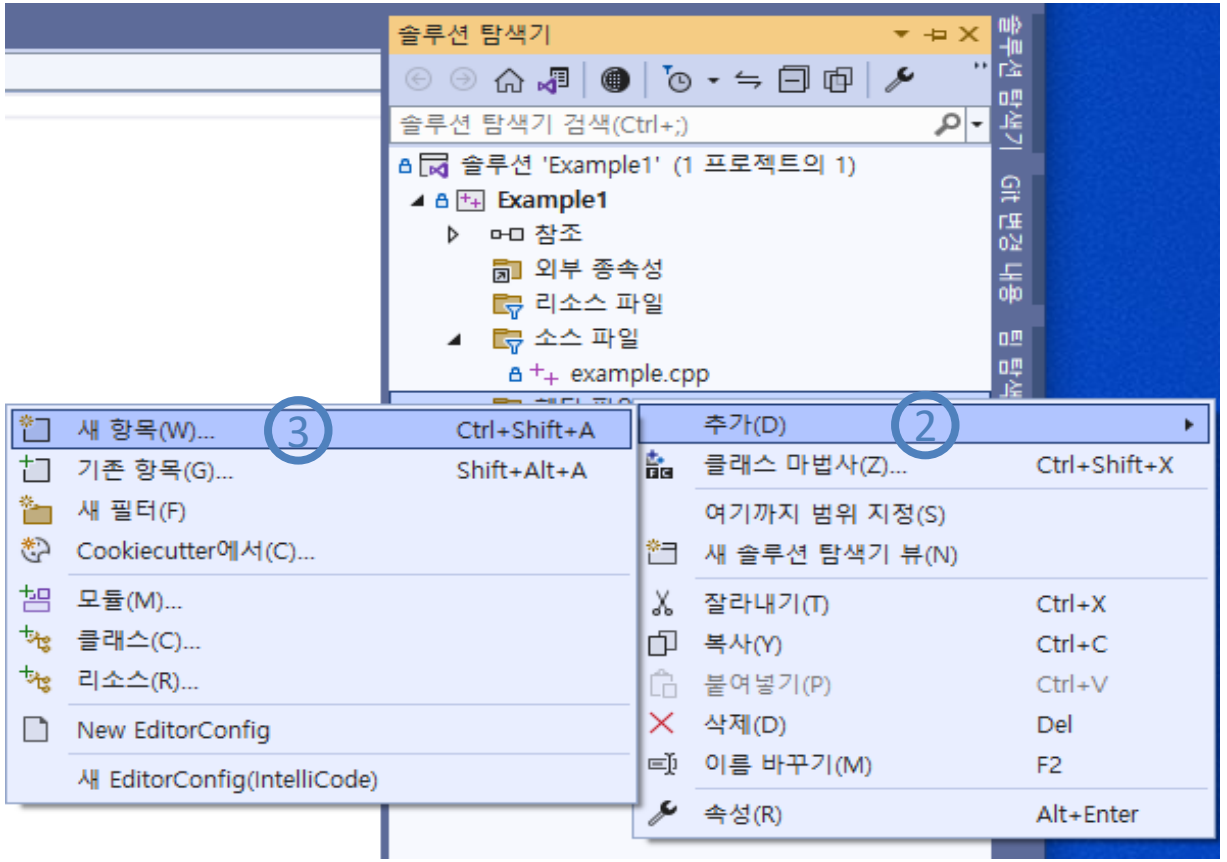
## ❖Visual Studio



# 헤더 파일(CPP) 추가

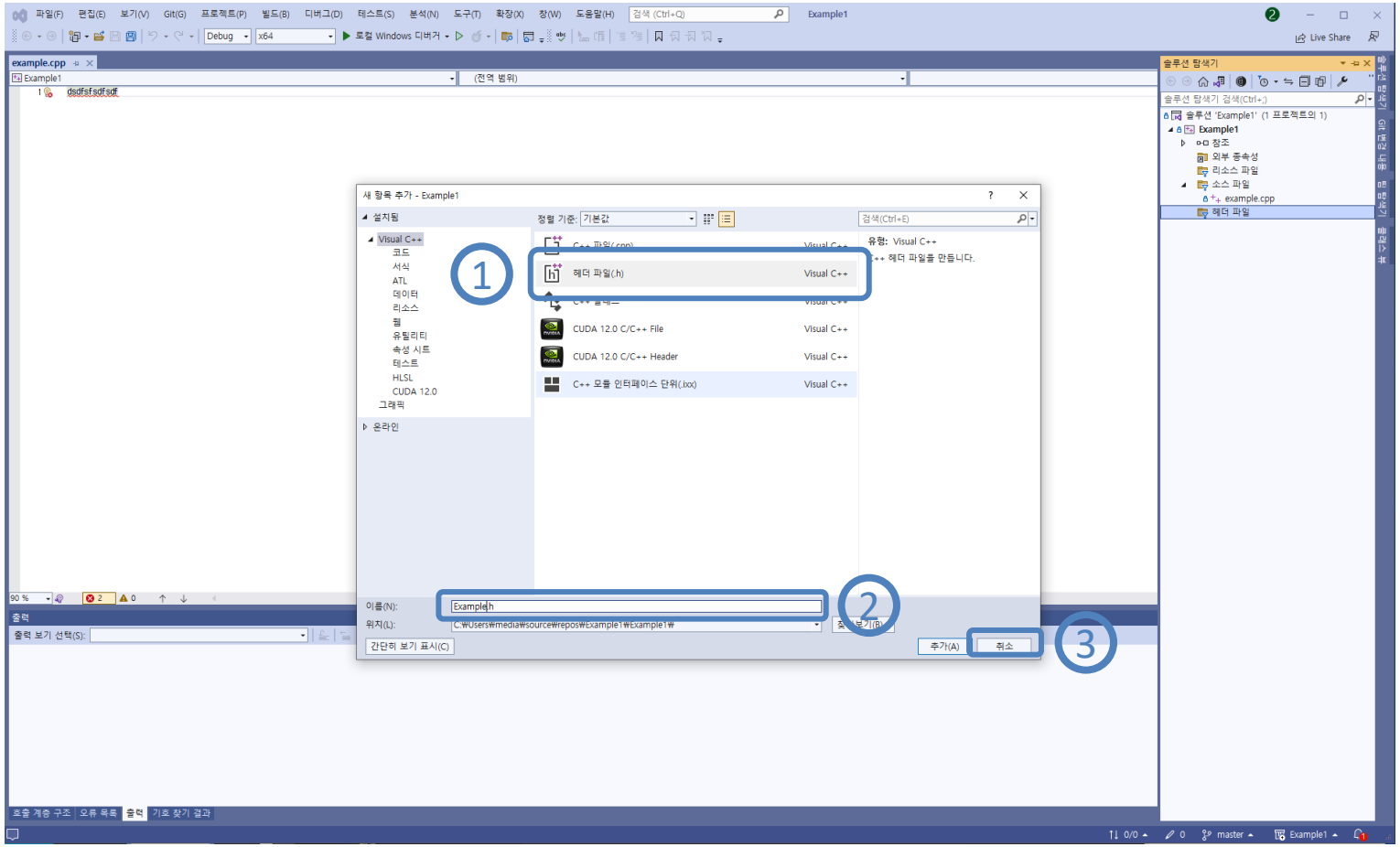
## ❖Visual Studio



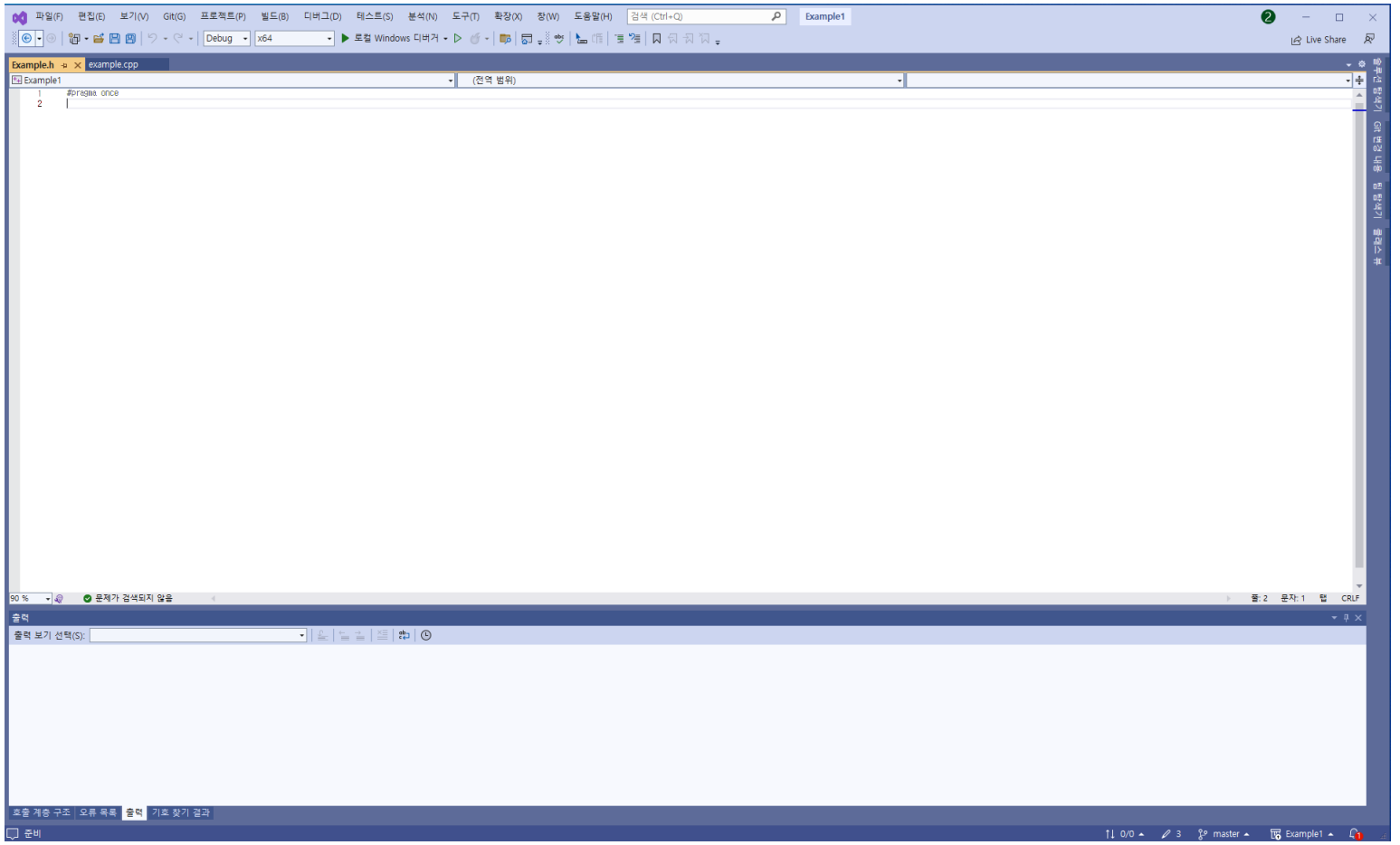
# 헤더 파일(CPP) 추가

## ❖Visual Studio



# 헤더 파일(CPP) 추가

## ❖Visual Studio



# Thank you