함수(C++)

```
/*
함수는 일부 작업을 수행하는 코드 블록입니다. 함수는 호출자가 함수에 인수를
입력 매개 변수를 필요에 따라 정의할 수 있습니다. 함수는 필요에 따라 출력으로
함수는 이상적으로 함수의 기능을 명확하게 설명하는 이름을 사용하여 재사용 가능
캡슐화하는 데 유용합니다. 다음 함수는 호출자에서 두 개의 정수를 허용하고 해
반환합니다. a 및 b는 형식 int의 매개 변수입니다.
*/

int Sum(int a, int b)
{
    return a + b;
}

/*
함수는 프로그램의 여러 위치에서 호출하거나 호출할 수 있습니다.
함수에 전달되는 값은 함수 정의의 매개 변수 형식과 호환되어야 하는 인수입니다
*/
#include <iostream>
int main()
{
    int j = Sum(10, 32);
    int i = Sum(j, 66);
    std::cout << "The value of j is" << i << std::endl;
}

/*
함수 길이에 대한 실질적인 제한은 없지만 잘 정의된 단일 작업을 수행하는 함수
복잡한 알고리즘은 가능하면 이해하기 쉬운 더 간단한 함수로 세분화해야 합니다.

클래스 범위에서 정의되는 함수는 맴버 함수라고 합니다. 다른 언어와 달리 C++
네임스페이스 범위에서 함수를 정의할 수도 있습니다. 이러한 함수는 자유 함수
라고 하며 표준 라이브러리에서 광범위하게 사용됩니다.
*/
```

함수 선언의 요소

- 최소 함수 선언은 컴파일러에 더 많은 지침을 제공하는 선택적 키워드(keyword)와 함께 반환 형식, 함수 이름 및 매개 변수 목록으로 구성됩니다. 다음 예제는 함수 선언입니다.

```
int Sum(int a, int b);
```

- 함수 정의는 선언과 중괄호 사이의 모든 코드인 본문으로 구성됩니다.

```
int Sum(int a, int b)
{
    return a + b;
}
```

- 함수 선언과 뒤에 오는 세미콜론은 프로그램의 여러 위치에 나타날 수 있으며, 각 변환 단위에서 해당 함수를 호출하기 전에 나타나야 합니다. 함수 정의는 ODR(단일 정의 규칙)에 따라 프로그램에 한 번만 나타나야 합니다.
- 함수 선언의 필수 요소는 다음과 같습니다.
    - 함수가 반환하는 값의 형식을 지정하거나 void 값이 반환되지 않는 경우를 지정하는 반환 형식입니다. C++11 auto에서는 컴파일러가 반환 문에서 형식을 유추하도록 지시하는 유효한 반환 형식입니다. C++14에서도 decltype(auto) 허용됩니다.
    - 문자 또는 밑줄로 시작해야 하며 공백을 포함할 수 없습니다. 일반적으로 표준 라이브러리 함수 이름의 선행 밑줄은 프라이빗 멤버 함수 또는 코드에서 사용할 수 없는 비 멤버 함수를 나타냅니다.
    - 매개 변수 목록 - 중괄호로 구분되거나 쉼표로 구분된 0개 이상의 매개 변수 집합으로, 함수 본문 내에서 값에 엑세스 할 수 있는 형식 및 로컬 이름을 지정합니다.
- 함수 선언의 선택적 요소는 다음과 같습니다.
    - constexpr - 함수의 반환값이 컴파일 시간에 계산할 수 있는 상수 값임을 나타냅니다.

```
constexpr float Exp(float x, int n)
{
    return n == 0 ? 1 :
```

```
            n & 2 == 0 ? Exp(x * x, n / 2) : Exp(x * x, (
}
```

- 링크 사양 extern 또는 static.

```
// Declare printf with C linkage
extern "C" int Printf(const char *fmt, ...);
```

- inline - 함수에 대한 모든 호출을 함수 코드 자체로 바꾸도록 컴파일러에 지시합니다. 인라인 처리는 성능이 중요한 코드 섹션에서 함수가 신속하게 실행되고 반복적으로 호출되는 시나리오의 성능 향상에 도움이 됩니다.

```
inline double Account::GetBalance()
{
    return balance;
}
```

- noexcept 함수가 예외를 throw 할 수 있는지 여부를 지정하는 식입니다. 다음 예제에서는 식이 계산되는 경우 함수에서 예외를 is_pod true throw 하지 않습니다.

```
#include <type_traits>
template <typename T>
T copy_object(T& obj) noexcept(std::is_pod<T>) {...}
```

- (맴버 함수만 해당) 함수인지 여부를 지정하는 cv 한정자입니다.(const volatile)
- (맴버 함수만 해당) virtual 또는 override final, virtual은 함수를 파생 클래스에서 재정의 할 수 있도록 지정합니다. override는 파생 클래스의 함수가 가상 함수를 재정의하고 있음을 의미합니다. final은 함수를 더이상 파생된 클래스에서 재정의 할 수 없음을 의미합니다.
- (맴버 함수만 해당) static 맴버 함수에 적용된다는 것은 함수가 클래스의 개체 인스턴스와 연결되지 않음을 의미합니다.
- (비정적 맴버 함수만 해당)암시적 개체 매개 변수(*this)가 rvalue 참조와 lvalue 참조일 때 선택할 함수의 오버로드를 컴파일러에 지정하는 ref-qualifier입니다.

함수 정의

- 함수 정의는 변수 선언, 문 및 식을 포함하는 중괄호로 묶인 선언과 함수 본문으로 구성됩니다.

```
int Foo(int i, std::string s)
{
    int value {i};
    MyClass mc;
    if (strcmp(s, "default") != 0)
    {
        value = mc.do_something(i);
    }
    return value;
}
```

- 본문 내에 선언된 변수는 지역 변수 또는 지역이라고 합니다. 이러한 변수는 함수가 종료될 때 범위를 벗어나므로 함수는 지역에 대한 참조를 반환할 수 없습니다.

```
MyClass& Boom(int i, std::string s)
{
    int value {i};
    MyClass mc;
    mc.Initialize(i, s);
    return mc;
}
```

const 및 constexpr 함수

- 맴버 함수 const를 선언하여 함수가 클래스의 데이터 맴버 값을 변경할 수 없도록 지정할 수 있습니다. 맴버 함수를 const 선언하면 컴파일러가 const-corrctness를 적용할 수 있습니다. 누군가가 실수로 선언된 const 함수를 사용하여 개체를 수정ㄹ하려고 하면 컴파일러 오류가 발생합니다.
- 함수가 생성하는 값이 컴파일 시간에 결정될 수 있는 경우 constexpr 함수를 선언한다. constexpr 함수는 일반 함수보다 빠르게 실행된다.

함수 템플릿

- 함수 템플릿은 클래스 템플릿과 유사하며 템플릿 인수에 따라 구체적인 함수를 생성합니다. 대부분의 경우 템플릿은 형식 인수를 유추할 수 있으므로 명시적으로 지정할 필요

가 없습니다.

```
template<typename Lhs, typename Rhs>
auto Add2(const Lhs& lhs, const Rhs& rhs)
{
    return lhs + rhs;
}

auto a = Add2(3.13, 2.895);
auto b = Add2(string {"Hello"}, string{" World"});
```

함수 매개변수 및 인수

- 함수에는 0개 이상 형식의 쉼표로 구분된 매개 변수 목록이 있으며, 각각에는 함수 본문 내에서 엑세스 할 수 있는 이름이 있습니다. 함수 템플릿은 더 많은 형식 또는 값 매개 변수를 지정할 수 있습니다. 호출자는 형식이 매개 변수 목록과 호환되는 구체적인 값인 인수를 전달합니다.
- 기본적으로 인수는 값으로 함수에 전달됩니다. 즉, 함수는 전달할 개체의 복사본을 받는다는 의미입니다. 큰 개체의 경우 복사본을 만드는 데 비용이 많이 들 수 있으며 항상 필요한 것은 아닙니다. 인수가 참조(특히 lvalue 참조)로 전달되도록 하려면 매개 변수에 참조 수량자를 추가합니다.

```
void DoSomething(std::string& input) {...}
```

- 함수에서 참조로 전달된 인수를 수정하면 로컬 복사본이 아니라 원래 개체가 수정됩니다. 함수가 이러한 인수를 수정하지 못하도록 하려면 매개 변수를 const로 한정합니다.

```
void DoSomething(const std::string& input) {...}
```

- C++11: rvalue-reference 또는 lvalue-reference로 전달되는 인수를 명시적으로 처리하려면 매개 변수에 이중 엠퍼샌드를 사용하여 범용 참조를 나타냅니다.

```
void DoSomething(const std::string&& input) {...}
```

- 키워드 인수 선언 목록의 첫 번째이자 유일한 맴버인 경우 매개 변수 선언 목록에서 단일 void 키워드 void 선언된 함수는 인수를 사용하지 않습니다. 목록의 다른 위치에 있는 형식 void의 인수는 오류를 생성합니다.

```
long GetTickCount( void );
```

- 여기에 설명된 경우를 제외하고 인수를 지정 void 하는 것은 불법이지만 형식에서 void 파생된 형식은 인수 선언 목록의 void 아무 곳에서나 나타날 수 있습니다.

기본 인수

- 함수 서명에서 마지막 매개 변수에 기본 인수를 할당할 수 있습니다. 즉, 일부 다른 값을 지정하려는 경우가 아니면 함수를 호출할 때 호출자가 인수를 제외할 수 있습니다.

```
int DoSomething(int num, string str, Allocator& alloc = defau
{...}

int DoSomethingEls(int num, string str = string {"Working"}, 
{...}

int DoMore(int num = 5, string str, Allocator& = defaultAlloc
```

함수 반환 형식

- 함수는 다른 함수 또는 기본 제공 배열을 반환할 수 없습니다. 그러나 이러한 형식 또는 함수 개체를 생성하는 람다에 대한 포인터를 반환할 수 있습니다. 이러한 경우를 제외하고 함수는 범위에 있는 모든 형식의 값을 반환하거나 값을 반환하지 않을 수 있습니다. 이 경우 void 반환 형식이 됩니다.

후행 반환 형식

- 일반 반환 형식은 함수 서명의 왼쪽에 있습니다. 후행 반환 형식은 서명의 오른쪽에 있으며 연산자가 앞에 옵니다. → 후행 반환 형식은 반환 값의 형식이 템플릿 매개 변수에 따라 달라지는 경우 함수 템플릿에서 특히 유용합니다.

```
template<typename Lhs, typename Rhs>
auto Add(const Lhs& lhs, const Rhs& rhs) -> decltype(lhs + rh
{
    return lhs + rhs;
}
```

- 후행 반환 형식과 함께 사용되는 경우 auto decltype 식이 생성하는 모든 항목에 대한 자리 표시자로만 사용되며 그 자체가 형식 추론을 수행하지 않습니다.

함수 지역 변수

- 함수 본문 내에서 선언된 변수를 지역 변수 또는 단순히 로컬 변수라고 부릅니다. 비정적 로컬은 함수 본문 내부에만 표시되며, 스택에 선언된 경우 함수가 종료될 때 범위를 벗어집니다. 지역 변수를 생성하고 값으로 반환하는 경우 컴파일러는 일반적으로 명명된 반환 값 최적화를 수행하여 불필요한 복사 작업을 방지할 수 있습니다. 지역 변수를 참조로 반환하는 경우 해당 참조를 사용하려는 호출자의 모든 시도가 지역이 제거된 후 수행되므로 컴파일러에서 경고가 발생합니다.
- C++에서는 지역 변수를 정적으로 선언할 수 있습니다. 이 변수는 함수 본문 내에만 표시되지만 함수의 모든 인스턴스에 대해 변수의 단일 복사본이 존재합니다. 로컬 정적 개체는 atexit로 지정된 종료 중에 소멸됩니다. 프로그램의 제어 흐름이 해당 선언을 무시했기 때문에 정적 개체가 생성되지 않은 경우 해당 개체를 삭제하려고 시도하지 않습니다.

반환 형식의 형식 추론 (C++ 14)

- C++114에서는 후행 반환 형식을 제공하지 않고도 함수 본문에서 반환 형식을 유초하도록 컴파일레 지시하느 데 auto 사용할 수 있습니다. auto 항상 값별 반환을 추론합니다. auto&&를 사용하여 참조를 추론하도록 컴파일러에 지시합니다.
- 이 예제 autoi에서는 lhs 및 rhs 합계의 비 -const 값 복사본으로 추론됩니다.

```
template<typename Lhs, typename Rhs>
auto Add2(const Lhs& lhs, const Rhs& rhs)
{
    return lhs + rhs;
}
```

- 추론하는 `auto` 형식의 const-ness는 유지되지 않습니다. 반환 값이 인수의 const-ness 또는 ref-ness를 유지해야 하는 전달 함수의 경우 형식 유추 규칙을 사용하고 `decltype` 모든 형식 정보를 유지하는 키워드(keyword) 사용할 `decltype(auto)` 수 있습니다. `decltype(auto)` 는 왼쪽의 일반 반환 값 또는 후행 반환 값으로 사용될 수 있습니다.
- 다음 예제(N3493의 코드 기반)는 템플릿이 인스턴스화될 때까지 알려지지 않은 반환 형식에서 함수 인수를 완벽하게 전달하도록 설정하는 데 사용되는 방법을 보여 `decltype(auto)` 줍니다.

```
template<typename F, typename Tuple = tuple<T...>, int... I>
decltype(auto) apply_(F&& f, Tuple&& args, index_sequence<I..
{
    return std::forward<F>(f)(std::get<I>(std::forward<Tuple>
}

template<typename F, typename Tuple = tuple<T...>,
            typename Indices = make_index-sequence<tuple_size
            decltype(auto) apply(F&& f, Tuple&& args)
{
    return apply_(std::forward<F>(f), std::forward<Tuple>(arg
}
```

함수에서 여러 값 반환

- 함수에서 두 개 이상의 값을 반환하는 다양한 방법이 있습니다.
    - 명명된 클래스 또는 구조체 개체의 값을 캡슐화 합니다. 호출자에게 클래스 또는 구조체 정의를 표시해야 합니다.

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

struct S
{
        string name;
        int num;
};

S g()
{
        string t{"hello"};
        int u{42};
        return {t,u};
}

int main()
```

```
{
        S s = g();
        cout << s.name << " " << s.num << endl;
        return 0;
}
```

- std::tuple 또는 std:;pair 개체를 반환합니다.

```
#include <tuple>
#include <string>
#include <iostream>

using namespace std;

tuple<int, string, double> f()
{
        int i{108};
        string s{"Some text"};
        double d {.01};
        return {i, s, d};
}

int main()
{
        auto t = f();
        cout << get<0>(t) << " " << get<1>(t) << " " << ge

        int myVal;
        string myname;
        double mydecimal;
        tie(myVal, myname, mydecimal) = f();
        cout << myVal << " " << myname << " " << mydecimal
        return 0;
}
```

- **Visual Studio 2017 버전 15.3 이상** (모드 이상에서 `/std:c++17` 사용 가능): 구조적 바인딩을 사용합니다. 구조화된 바인딩의 장점은 반환 값을 저장하는 변수가 선언되는 동시에 초기화되므로 경우에 따라 훨씬 더 효율적일 수 있습니다.

문 `auto[x, y, z] = f();` 에서 대괄호는 전체 함수 블록의 범위에 있는 이름을 도입하고 초기화합니다.

```
#include <tuple>
#include <string>
#include <iostream>

using namespace std;

tuple<int, string, double> f()
{
    int i{ 108 };
    string s{ "Some text" };
    double d{ .01 };
    return { i,s,d };
}
struct S
{
    string name;
    int num;
};

S g()
{
    string t{ "hello" };
    int u{ 42 };
    return { t, u };
}

int main()
{
    auto[x, y, z] = f(); // init from tuple
    cout << x << " " << y << " " << z << endl;

    auto[a, b] = g(); // init from POD struct
    cout << a << " " << b << endl;
```

```
    return 0;
}
```

- 반환 값 자체를 사용하는 것 외에도 함수가 호출자가 제공하는 개체의 값을 수정하거나 초기화할 수 있도록 참조를 통해 사용할 매개 변수 수를 정의하여 값을 "반환"할 수 있습니다.

함수 포인터

- C++은 C 언어와 동일한 방식으로 함수 포인터를 지원합니다. 그러나 일반적으로 함수 개체를 사용하면 형식이 보다 더 안전합니다. 함수 포인터 형식을 반환하는 함수를 선언하는 경우 함수 포인터 형식에 대한 별칭을 선언하는 데 사용하는 `typedef` 것이 좋습니다. 예를 들면 다음과 같습니다.

```
typedef int (*fp)(int);
fp myFunction(char* s); // function returning function pointe
```

- 이 작업이 수행되지 않으면 다음과 같이 식별자(`fp` 위 예제의 경우)를 함수 이름 및 인수 목록으로 바꿔 함수 포인터의 선언자 구문에서 함수 선언에 대한 적절한 구문을 추론할 수 있습니다.

```
int (*myFunction(char* s))(int);
```

- 앞의 선언은 앞의 선언과 `typedef` 동일합니다.