



C++ 프로그래밍

김 형 기

hk.kim@jbnu.ac.kr

함수

Today

- 정의
- 프로토타입
- 매개변수(parameter)와 pass-by-value ☆
- return문
- 기본 인수(default argument)
- 오버로딩 ☆
- 함수 호출의 동작 방식 ☆
- 포인터와 참조자 ☆
- inline 함수
- 재귀 함수(recursive)

Functions

- 정의

- 프로토타입
- 매개변수(parameter)와 pass-by-value☆
- return문
- 기본 인수(default argument)
- 오버로딩 ☆
- 함수 호출의 동작 방식 ☆
- 포인터와 참조자☆
- inline 함수
- 재귀 함수(recursive)

Functions

- C++ 프로그램의 함수

- C++ 표준 라이브러리(함수와 클래스)
- 써드 파티 라이브러리(함수와 클래스)
- 직접 구현한 함수와 클래스

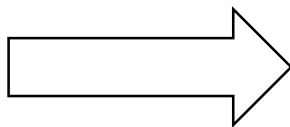
- 함수 → 모듈화 → 재사용성!

- 코드를 독립적인 연산으로 분할
- 연산들을 재사용

Functions

- 가독성 비교

```
1 int main()
2 {
3     //read input
4     statement1;
5     statement2;
6
7     //process input
8     statement3;
9     statement4;
10    statement5;
11
12    //write output
13    statement6;
14    statement7;
15
16    return 0;
17 }
```



```
1 int main()
2 {
3     readInput();
4
5     processInput();
6
7     writeOutput();
8
9     return 0;
10 }
```

모듈화

Functions

● 함수를 사용할 때 알아야 하는 것

- 함수의 기능을 알아야 함
- 함수에서 필요로 하는 정보를 알아야 함
- 함수가 리턴하는 것을 알아야 함
- 어떤 오류가 발생하는지 알아야 함
- 성능상의 제약에 대해 이해해야 함




● 함수를 사용할 때 몰라도 되는 것

- 함수가 내부적으로 어떻게 동작하는지

● 자동차 운전 예시

- 액셀을 밟으면 가속도가 붙고, 브레이크를 밟으면 감속된다. 핸들을 돌리면 바퀴가 돌아간다.
- 액셀을 밟았을 때 연료가 어떻게 분사되고, 구동장치가 어떻게 돌아가는지? 몰라도 운전 가능

Functions

		Name	Description
f_v 	float	<u>ActorGetDistanceToCollision</u> (const FVector& Point, ECollisionChannel TraceChannel, FVector& ClosestPointOnCollision, UPrimitiveComponent** OutP...)	Returns Distance to closest Body Instance surface.
f_b 	bool	<u>ActorHasTag</u> (FName Tag)	See if this actor's Tags array contains the supplied name tag
f_b 	bool	<u>ActorLineTraceSingle</u> (FHitResult& OutHit, const FVector& Start, const FVector& End, ECollisionChannel TraceChannel, const FCollisionQueryParams& Param...)	Trace a ray against the Components of the Actor and return the first blocking hit

언리얼 엔진의 문서 예시

함수의 리턴값, 이름과 매개변수, 기능 설명만 적혀있음

Functions

- <cmath> 예시
 - 수학 연산 함수들 제공

```
1 cout << sqrt(400.0) << endl; // 20;  
2 double result;  
3 result = pow(2.0, 3.0); // 2^3;
```

- 함수가 내부적으로 어떻게 동작하는지는 몰라도 사용할 수 있음

Functions

● C++ 표준 라이브러리

cppreference.com [Create account](#)

Page Discussion View Edit History

C++ Standard Library header files

C++ Standard Library header files

The interface of C++ standard library is defined by the following collection of header files

Concepts library

`<concepts>` (since C++20) Fundamental library concepts

Utilities library

<code><cstdlib></code>	General purpose utilities: program control, dynamic memory allocation, random numbers, sort and search
<code><csignal></code>	Functions and macro constants for signal management
<code><csetjmp></code>	Macro (and function) that saves (and jumps) to an execution context
<code><cstdint></code>	Handling of variable length argument lists
<code><typeinfo></code>	Runtime type information utilities
<code><typeindex></code> (since C++11)	<code>std::type_index</code>
<code><type_traits></code> (since C++11)	Compile-time type information
<code><bitset></code>	<code>std::bitset</code> class template
<code><functional></code>	Function objects, Function invocations, Bind operations and Reference wrappers
<code><utility></code>	Various utility components
<code><ctime></code>	C-style time/date utilities
<code><chrono></code> (since C++11)	C++ time utilities
<code><cstdint></code>	standard macros and typedefs
<code><initializer_list></code> (since C++11)	<code>std::initializer_list</code> class template
<code><tuple></code> (since C++11)	<code>std::tuple</code> class template
<code><any></code> (since C++17)	<code>std::any</code> class
<code><optional></code> (since C++17)	<code>std::optional</code> class template
<code><variant></code> (since C++17)	<code>std::variant</code> class template

<code>log</code>	computes natural (base e) logarithm ($\ln(x)$) (function)
<code>log10</code>	computes common (base 10) logarithm ($\log_{10}(x)$) (function)
<code>log2</code> (C++11)	base 2 logarithm of the given number ($\log_2(x)$) (function)
<code>log1p</code> (C++11)	natural logarithm (to base e) of 1 plus the given number ($\ln(1+x)$) (function)

Power functions

<code>pow</code>	raises a number to the given power (x^y) (function)
<code>sqrt</code>	computes square root (\sqrt{x}) (function)
<code>cbrt</code> (C++11)	computes cubic root ($\sqrt[3]{x}$) (function)
<code>hypot</code> (C++11)	computes square root of the sum of the squares of two given numbers ($\sqrt{x^2+y^2}$) (function)

Trigonometric functions

<code>sin</code>	computes sine ($\sin(x)$) (function)
<code>cos</code>	computes cosine ($\cos(x)$) (function)
<code>tan</code>	computes tangent ($\tan(x)$) (function)

Function Definition

- 함수의 정의에 필요한 요소
 1. 이름
 2. 매개변수 리스트
 3. 리턴 타입
 4. 본문(body)

Function Definition

- 함수의 정의에 필요한 요소
 1. 이름
 - 함수의 이름
 - 변수의 명명 규칙과 동일
 - 의미가 있는 이름이어야 함!
 2. 매개변수 리스트
 3. 리턴 타입
 4. 본문(body)

Function Definition

- 함수의 정의

1. 이름

2. 매개변수 리스트

- 함수에 전달되는 값(인자)들

- 타입이 명시되어야 함

3. 리턴 타입

- 연산 결과의 반환 타입

4. 본문(body)

- 함수가 호출되었을 때 실행되는 명령문

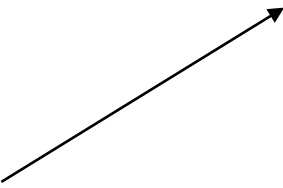
- 대괄호("{ }") 내부

Function Definition

- 함수의 정의

```
1 returnType FunctionName(parameters)
2 {
3     statements;
4
5     return 0;
6 }
```

반환값이 없는 경우 void로 명시



```
1 int FunctionName(int a)
2 {
3     statements;
4
5     return 0;
6 }
7
8 void FunctionName(double d)
9 {
10    statements;
11
12    return; //or 생략
13 }
```

Calling a Function

- 함수의 호출

```
1 void PrintHello()
2 {
3     cout << "Hello" << endl;
4 }
5
6 int main()
7 {
8     for(int i=0; i<10; i++)
9     {
10         PrintHello();
11     }
12     return 0;
13 }
```

} *PrintHello() 함수 정의*

} *PrintHello() 함수 호출(사용)*

Calling a Function

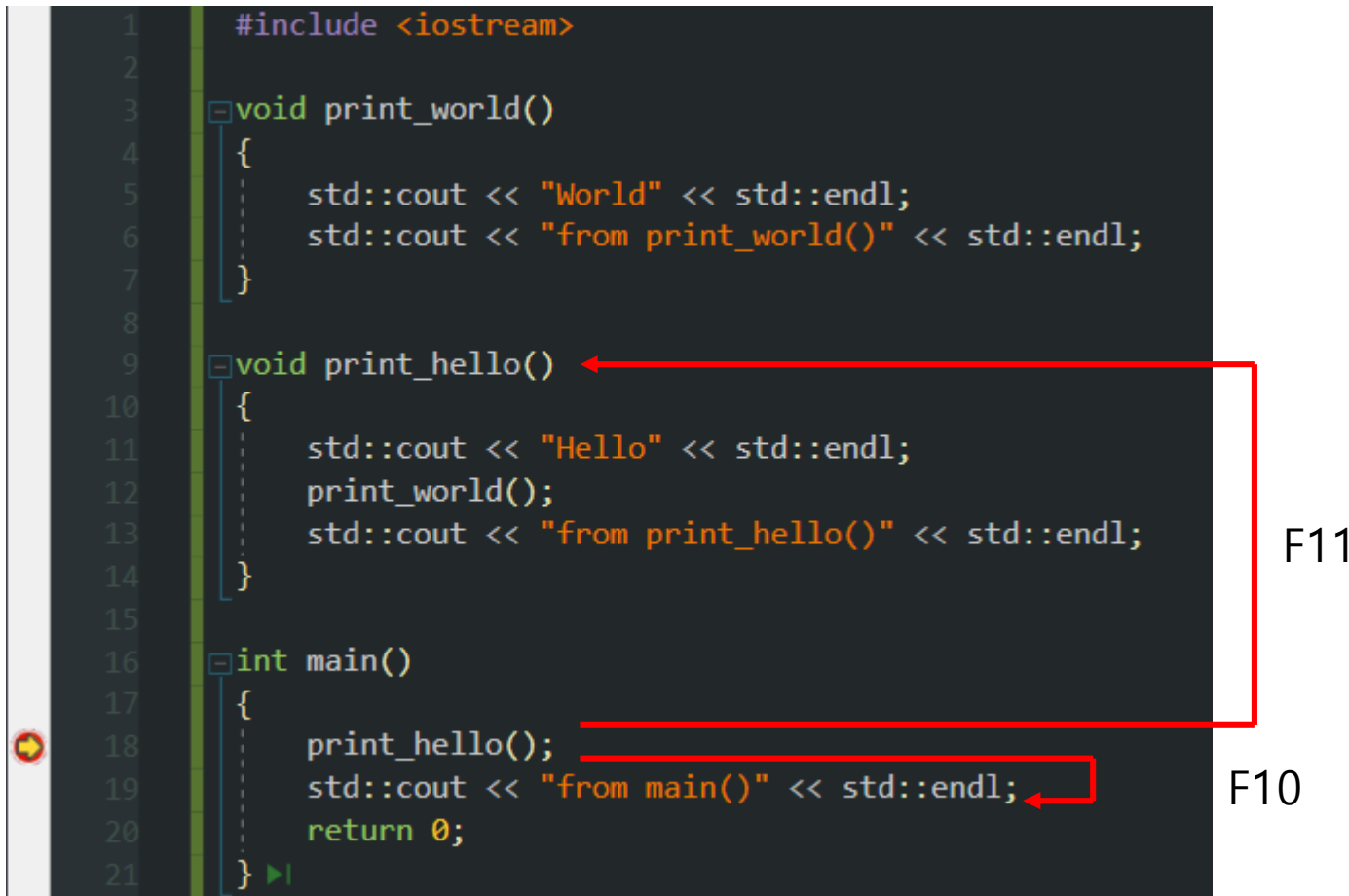
- 함수의 호출

```
1 void PrintWorld()
2 {
3     cout << "World" << endl;
4     cout << "from PrintWorld()" << endl;
5 }
6
7 void PrintHello()
8 {
9     cout << "Hello" << endl;
10    printWorld();
11    cout << "from PrintHello()" << endl;
12 }
13
14 int main()
15 {
16     PrintHello();
17     cout << "from main()" << endl;
18     return 0;
19 }
```

콘솔창에 어떤 순서로 출력될까?

Debugging Function Calls in VS 2017

- 함수의 호출



함수를 실행할 시점에, F11을 누르면 함수 안으로 진입, F10을 누르면 함수 다음줄로 진행

- 컴파일러는 함수의 호출(사용) 이전에 함수의 정의를 알아야 함!

```
1 int main()
2 {
3     SayHello();
4     return 0;
5 }
6
7 void SayHello()
8 {
9     cout << "Hello" << endl;
10 }
```

error C3861: 'say_hello': 식별자를 찾을 수 없습니다.

Functions

- 정의
- **프로토타입**
- 매개변수(parameter)와 pass-by-value ☆
- return문
- 기본 인수(default argument)
- 오버로딩 ☆
- 함수 호출의 동작 방식 ☆
- 배열의 전달과 pass-by-reference ☆
- inline 함수
- 재귀 함수(recursive)

Function Prototypes

- 함수의 **호출 이전**에 함수의 정의를 알 수 있어야 함!
 - 매개변수가 몇 개고, 어떤 타입의 데이터를 리턴하는지를 알려주어야 함
 - 해결방법 1: 항상 함수의 호출보다 위쪽 라인에 함수를 정의
 - 작은 프로그램에서는 OK
 - 일반적으로는 효율적인 방법이 아님
 - 해결방법 2: 함수 프로토타입의 사용
 - 함수의 전체 정의가 아닌 컴파일러가 알아야 할 부분만을 미리 알려주는 개념
 - 전방 선언(forward declaration)이라고도 명칭
 - 프로그램의 초기에 위치
 - 헤더 파일(.h)의 활용

Function Prototypes

- 함수 프로토타입

- 함수를 사용하기 이전에 입력 매개변수와 반환형을 미리 알려주는 명령문
- 함수를 사용할 시점에, 프로토타입과 사용 형식이 맞지 않는다면 오류

```
1 int FunctionName(int, std::string); // Prototype
2
3 int main(){
4     FunctionName(1, "KHK"); // Call(Use)
5 }
6
7 int FunctionName(int a, std::string b) // Definition
8 {
9     statements;
10    return 0;
11 }
```

```
1 void SayHello(); // Prototype
2
3 int main()
4 {
5     SayHello(); //OK
6     SayHello(100); //ERROR
7     cout << SayHello(); //ERROR
8
9     return 0;
10 }
```

- Compile과 Linking, 프로토타입에 대한 깊은 이해
 1. 두 개의 cpp파일에 모두 다 iostream이 필요한가?
 2. 각 파일이 compile이 정상적으로 이루어지는 이유는?
 3. 만일 Log.cpp에 Log 함수가 없다면 어떤 오류가 발생할까?

Main.cpp

```
1 #include <iostream>
2
3 void Log(const char* message);
4
5 int main()
6 {
7     Log("Hello World");
8 }
```

Log.cpp

```
1 #include <iostream>
2
3 void Log(const char* messgae)
4 {
5     std::cout << message << std::endl;
6 }
```

Functions

- 정의
- 프로토타입
- 매개변수(parameter)와 pass-by-value☆
- return문
- 기본 인수(default argument)
- 오버로딩 ☆
- 함수 호출의 동작 방식 ☆
- 배열의 전달과 pass-by-reference ☆
- inline 함수
- 재귀 함수(recursive)

Function Parameters

- 함수 매개변수

- 함수를 호출할 때, 데이터를 전달할 수 있음
 - 함수의 호출에 있어서 전달하는 값은 인수(argument)라 함
 - 함수의 정의에 있어서 전달하는 값은 인자 또는 매개변수(parameter)라 함
- 인수와 매개변수는 개수, 순서와 타입이 일치해야 함

Function Parameters

- 함수 매개변수

```
1 int AddNumbers(int,int); // Prototype
2
3 int main()
4 {
5     int result = 0;
6     result = AddNumbers(100,200); // Call(use)
7     return 0;
8 }
9
10 int AddNumbers(int first, int second) // Definition
11 {
12     return first+second;
13 }
```


- 함수에 데이터를 전달할 때는 값으로 전달(pass-by-value)됨
 - 데이터의 값이 복사되어 전달
 - 함수 내에서는 원본에서 복사해서 만들어진 사본이 사용됨
 - 전달된 인수는 함수를 통해 변화되지 않음
 - 사본을 바꾼다고 원본이 바뀌지 않음
 - 실수로 값을 변화하는 것을 방지
 - 원본을 변화시키는 것이 필요하거나, 복사 비용이 높을 때를 위한 방법 존재 (포인터/참조자)

- 원본과 사본

```
1 void ParamChange(int formal) //formal은 actual의 사본!  
2 {  
3     cout << formal << endl; //50  
4     formal = 100;  
5     cout << formal << endl; //100 ❌ formal은 100으로 변화  
6 }  
7  
8 int main()  
9 {  
10     int actual=50; //원본. Main()함수 안의 actual이라는 변수  
11     cout << actual << endl; //원본의 값은 50  
12     ParamChange(actual); //actual을 함수에 전달  
13     cout << actual << endl; //50 ← 원본 값을 변하지 않음  
14     return 0;  
15 }
```

Functions

- 정의
- 프로토타입
- 매개변수(parameter)와 pass-by-value ☆
- **return문**
- **기본 인수(default argument)**
- 오버로딩 ☆
- 함수 호출의 동작 방식 ☆
- 배열의 전달과 pass-by-reference ☆
- inline 함수
- 재귀 함수(recursive)

Function Return

- 반환(return)

- return 문을 통해서 함수의 결과값을 전달
 - Void형 반환인 경우 return문 생략 가능
- return문은 함수 내 어느 곳에서나 정의 가능
- return문을 통해 함수는 **즉각적으로 종료**

Default Argument Values

● 기본 인수

- (Remind) 함수의 선언에서 정의한 모든 매개변수가 전달되어야 함
- 기본 인수를 사용하면 인수가 주어지지 않을 시 기본값을 사용하도록 정의 가능
 - 동일한 값을 자주 사용할 경우
- 기본값은 함수 프로토타입 **또는** 정의부에 선언
 - 프로토타입에 선언하는 것이 기본
 - 둘 다 선언해서는 안됨!
- 여러 개의 기본값을 사용할 경우 오른쪽부터 선언해야 함

Default Argument Values

- 기본 인수

```
1 double CalcCost(double baseCost, double taxRate = 0.06, double shipping = 3.5);
2
3 double CalcCost(double baseCost, double taxRate, double shipping)
4 {
5     return baseCost += (baseCost * taxRate) + shipping;
6 }
7
8 int main()
9 {
10     double cost=0;
11     cost = CalcCost(100.0, 0.08, 4.5);
12     cost = CalcCost(100.0, 0.08);
13     cost = CalcCost(200.0);
14     return 0;
15 }
```

Functions

- 정의
- 프로토타입
- 매개변수(parameter)와 pass-by-value ☆
- return 문
- 기본 인수(default argument)
- **오버로딩 ☆**
- 함수 호출의 동작 방식 ☆
- 배열의 전달과 pass-by-reference ☆
- inline 함수
- 재귀 함수(recursive)

Function Overloading

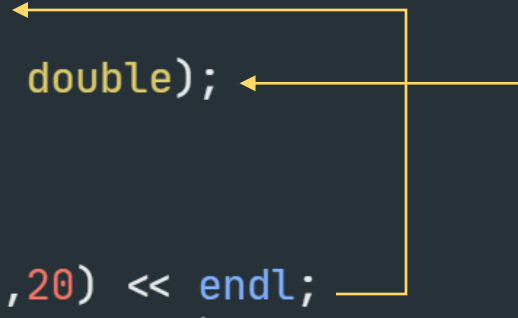
- 함수 오버로딩

- 서로 다른 매개변수 리스트를 갖는 **동일한 이름**의 함수를 정의하는 것
- 추상화의 한 예
- 다형성의 한 예
 - 유사한 개념의 함수를 다른 타입에 대해 정의
- 객체지향 프로그램 구현을 위한 중요한 기법 중 하나
- 컴파일러는 주어진 인수와 함수들의 파라미터 정의를 기반으로 개별적인 함수를 구분할 수 있어야 함

Function Overloading

- 함수 오버로딩

```
1 int AddNumber(int, int);  
2 double AddNumber(double, double);  
3  
4 int main()  
5 {  
6     cout << AddNumber(10,20) << endl;  
7     cout << AddNumber(10.0, 20.0) << endl;  
8     return 0;  
9 }  
10  
11 int AddNumber(int first, int second)  
12 {  
13     return first+second;  
14 }  
15 double AddNumber(double first, double second)  
16 {  
17     return first+second;  
18 }
```



Function Overloading

- 함수 오버로딩
 - 반환 타입만 다른 오버로딩은 불가능하다는 것에 주의

```
1 int GetValue();  
2 double GetValue();  
3  
4 ...  
5 cout << GetValue() << endl; // ?? Int가 반환될지 double이 반환될지 알수없음
```

(Summary) Functions

- 정의
- 프로토타입
- 매개변수(parameter)와 pass-by-value ☆
- return 문
- 기본 인수(default argument)
- 오버로딩 ☆
- **함수 호출의 동작 방식 ☆**
- 배열의 전달과 pass-by-reference ☆
- inline 함수
- 재귀 함수(recursive)

- 지역 범위

- 블록 { } 내의 범위
- **함수의 매개변수까지 함수 범위 내의 지역 변수로 생각해야 함**
 - For문에서 `int i=0`이 블록 내 범위인 것과 마찬가지로
- 따라서 함수의 (복사된) 인자 및 지역 변수들은 함수의 실행 중에만 존재함

- static 지역 변수

- static 한정어를 사용해 예외 변수 지정 가능함
- 초기화가 필요

- 전역 범위

- 함수 밖에 정의된 변수는 어디서나 접근 가능
- 전역 변수는 사용하지 않는 것이 좋음
 - 전역 상수는 OK

Local/Global Scope

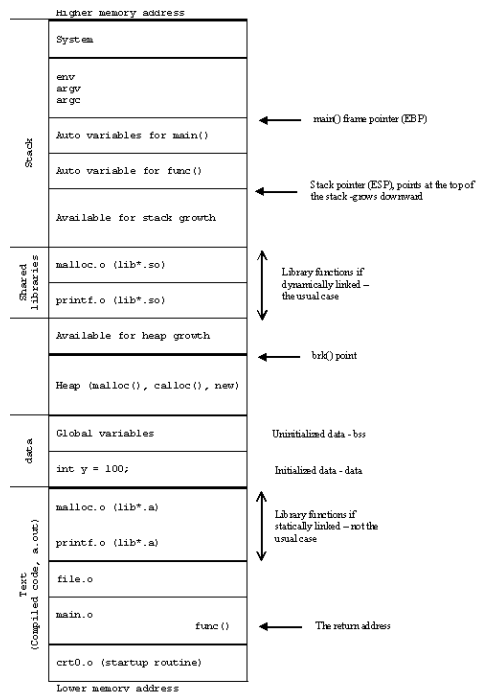
- static 지역 변수

- static 한정어를 사용해 지역 내에 정의된 변수를 지역 밖에 정의한 것처럼 활용 가능
- 단, scope 밖에서 접근할 수는 없음
- 초기화가 필요

```
1 void StaticLocalIncrement()
2 {
3     static int num = 1;
4     cout << "num : " << num << endl;
5     num++;
6     cout << "num : " << num << endl;
7 }
8
9 int main()
10 {
11     StaticLocalIncrement(); // 1 2
12     StaticLocalIncrement(); // 2 3
13     StaticLocalIncrement(); // 3 4
14 }
```

● 메모리 레이아웃

- 지금은 스택 영역만 생각



스택 영역은 빠르지만 작음
메모리가 자동으로 정리(해제)됨

힙 영역은 크지만 느림
메모리를 직접 해제해주어야 함
(다음 강의)

- 함수 호출의 동작방식
 - Function call stack
 - LIFO(Last in first out)
 - Stack Frame (Activation Record)
 - 함수의 **호출이 발생할 때마다 일종의 구분선**이 정의됨
 - 함수의 **지역 변수와 매개변수는 그 구분선 영역 내에** 생성됨
 - 함수의 호출이 끝나면 **구분선 내의 메모리는 자동으로 해제**됨
 - 스택은 유한하고 작아서, stack overflow 발생할 수 있음

```
1 int Func2(int x, int y, int z)
2 {
3     x += y + z;
4     return x;
5 }
6 int Func1(int a, int b)
7 {
8     int result;
9     result = a + b;
10    result = Func2(result, a, b);
11    return result;
12 }
13 int main()
14 {
15     int x = 10;
16     int y = 20;
17     int z;
18     z = Func1(x, y);
19     cout << z << endl;
20     return 0;
21 }
```



```
1 int Func2(int x, int y, int z)
2 {
3     x += y + z;
4     return x;
5 }
6 int Func1(int a, int b)
7 {
8     int result;
9     result = a + b;
10    result = Func2(result, a, b);
11    return result;
12 }
13 int main()
14 {
15     int x = 10;
16     int y = 20;
17     int z;
18     z = Func1(x, y);
19     cout << z << endl;
20     return 0;
21 }
```

Main함수
지역변수들

스택 메모리

CC는 초기화되지
않은 쓰레기값을
표시하기 위해 사용

Int기 때문에
실제로는
4칸씩(4바이트씩)을
차지하므로 주소가
4씩 감소함!

Int z

CC

0x1000 - 8

Int y

CC

0x1000 - 4

Int x

CC

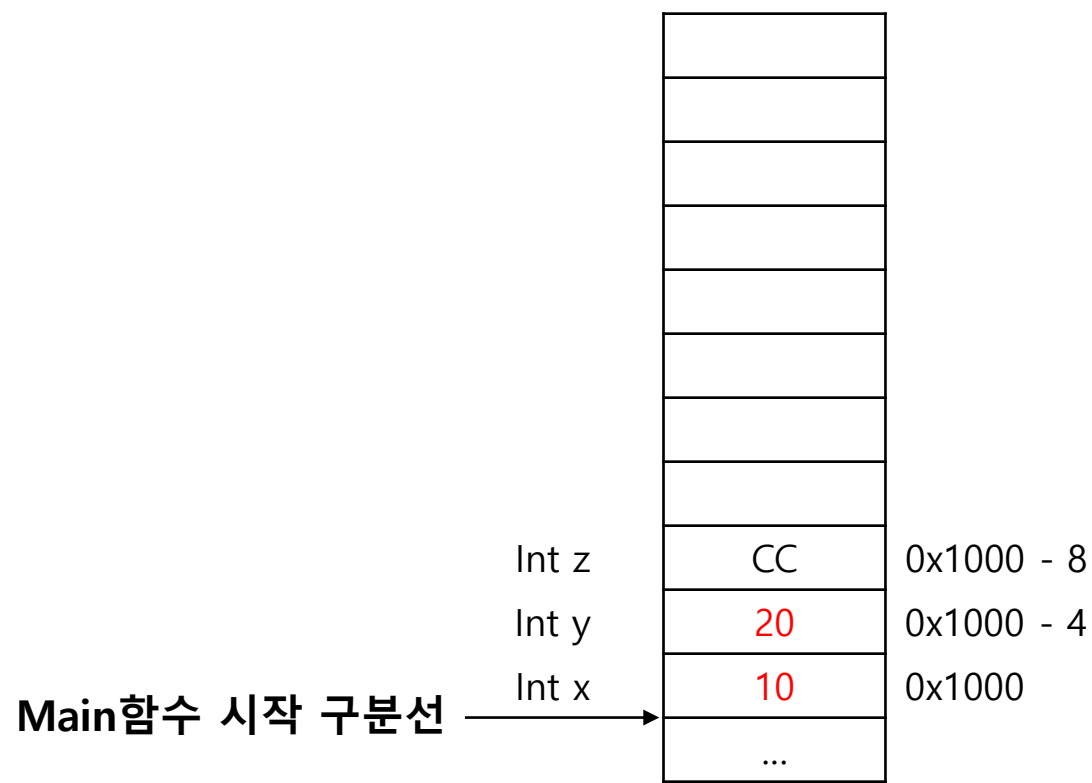
0x1000

...

Main함수 시작 구분선

```
1 int Func2(int x, int y, int z)
2 {
3     x += y + z;
4     return x;
5 }
6 int Func1(int a, int b)
7 {
8     int result;
9     result = a + b;
10    result = Func2(result, a, b);
11    return result;
12 }
13 int main()
14 {
15     int x = 10;
16     int y = 20;
17     int z;
18     z = Func1(x, y);
19     cout << z << endl;
20     return 0;
21 }
```

스택 메모리

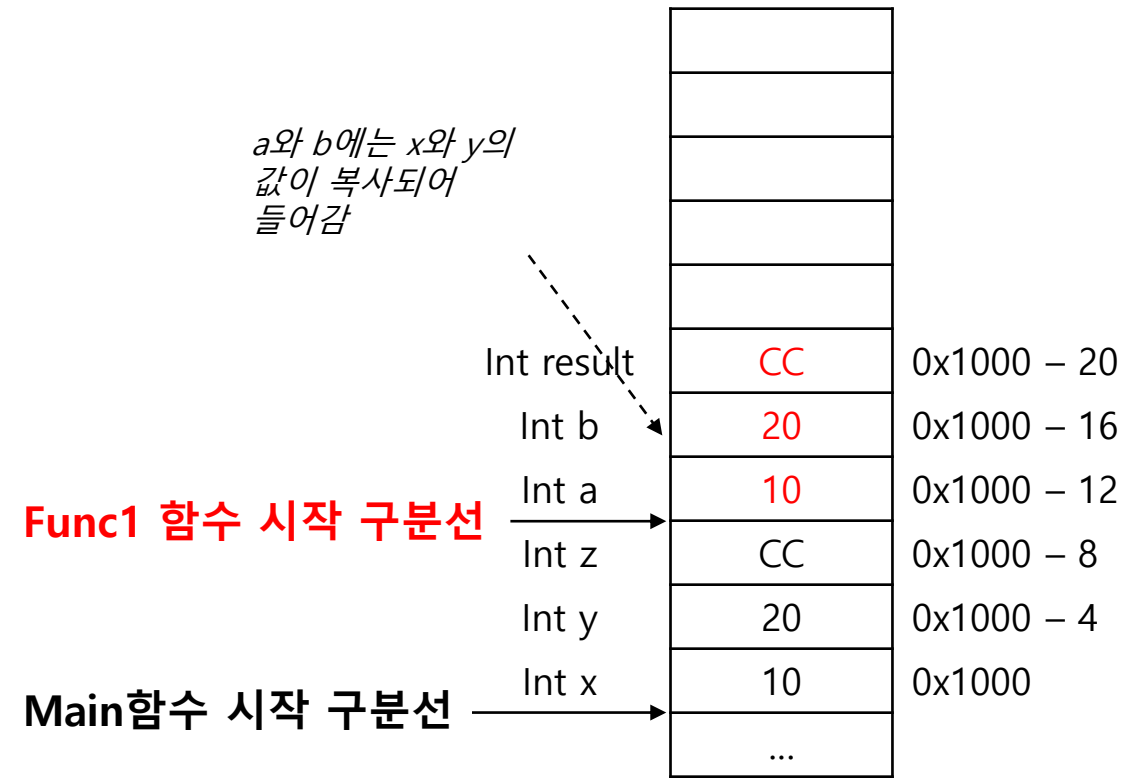


```
1 int Func2(int x, int y, int z)
2 {
3     x += y + z;
4     return x;
5 }
6 int Func1(int a, int b)
7 {
8     int result;
9     result = a + b;
10    result = Func2(result, a, b);
11    return result;
12 }
13 int main()
14 {
15     int x = 10;
16     int y = 20;
17     int z;
18     z = Func1(x, y);
19     cout << z << endl;
20     return 0;
21 }
```

→

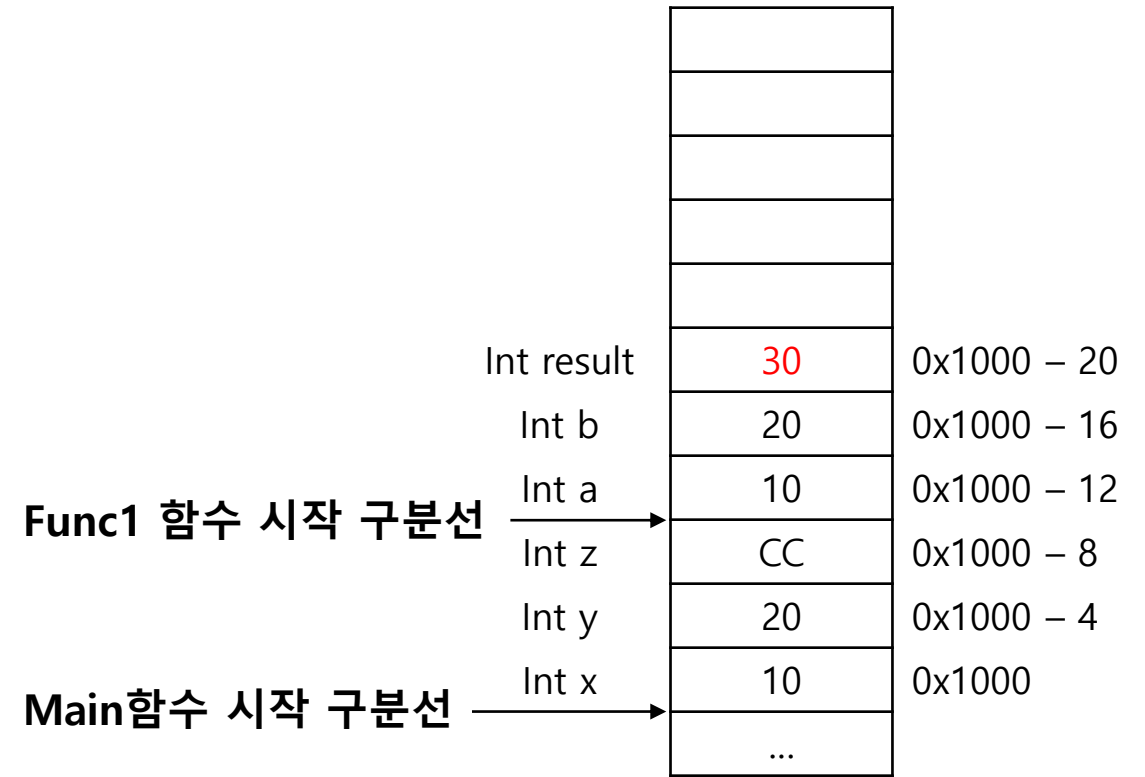
Func1함수의
지역변수들

스택 메모리



```
1 int Func2(int x, int y, int z)
2 {
3     x += y + z;
4     return x;
5 }
6 int Func1(int a, int b)
7 {
8     int result;
9     result = a + b;
10    result = Func2(result, a, b);
11    return result;
12 }
13 int main()
14 {
15     int x = 10;
16     int y = 20;
17     int z;
18     z = Func1(x, y);
19     cout << z << endl;
20     return 0;
21 }
```

스택 메모리



```
1 int Func2(int x, int y, int z)
2 {
3     x += y + z;
4     return x;
5 }
6 int Func1(int a, int b)
7 {
8     int result;
9     result = a + b;
10    result = Func2(result, a, b);
11    return result;
12 }
13 int main()
14 {
15     int x = 10;
16     int y = 20;
17     int z;
18     z = Func1(x, y);
19     cout << z << endl;
20     return 0;
21 }
```

Func2함수의 지역변수들

스택 메모리

Func2 함수 시작 구분선

Func1 함수 시작 구분선

Main함수 시작 구분선

Int z	20	
Int y	10	
Int x	30	...
Int result	30	0x1000 - 20
Int b	20	0x1000 - 16
Int a	10	0x1000 - 12
Int z	CC	0x1000 - 8
Int y	20	0x1000 - 4
Int x	10	0x1000
	...	

```
1 int Func2(int x, int y, int z)
2 {
3     x += y + z;
4     return x;
5 }
6 int Func1(int a, int b)
7 {
8     int result;
9     result = a + b;
10    result = Func2(result, a, b);
11    return result;
12 }
13 int main()
14 {
15     int x = 10;
16     int y = 20;
17     int z;
18     z = Func1(x, y);
19     cout << z << endl;
20     return 0;
21 }
```

스택 메모리

Func2 함수 시작 구분선

Func1 함수 시작 구분선

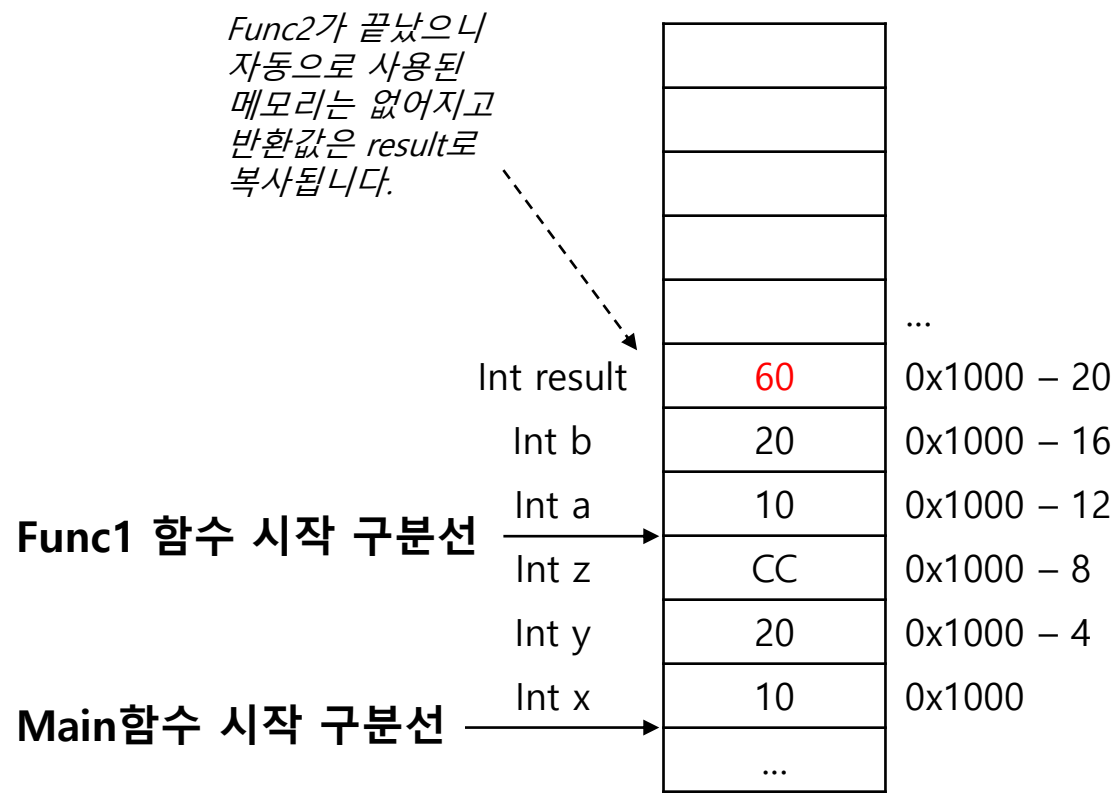
Main 함수 시작 구분선

Int z	20	
Int y	10	
Int x	60	...
Int result	30	0x1000 - 20
Int b	20	0x1000 - 16
Int a	10	0x1000 - 12
Int z	CC	0x1000 - 8
Int y	20	0x1000 - 4
Int x	10	0x1000
	...	

```
1 int Func2(int x, int y, int z)
2 {
3     x += y + z;
4     return x;
5 }
6 int Func1(int a, int b)
7 {
8     int result;
9     result = a + b;
10    result = Func2(result, a, b);
11    return result;
12 }
13 int main()
14 {
15     int x = 10;
16     int y = 20;
17     int z;
18     z = Func1(x, y);
19     cout << z << endl;
20     return 0;
21 }
```

Func2를 호출한
지점으로
돌아옵니다.

스택 메모리



```
1 int Func2(int x, int y, int z)
2 {
3     x += y + z;
4     return x;
5 }
6 int Func1(int a, int b)
7 {
8     int result;
9     result = a + b;
10    result = Func2(result, a, b);
11    return result;
12 }
13 int main()
14 {
15     int x = 10;
16     int y = 20;
17     int z;
18     z = Func1(x, y);
19     cout << z << endl;
20     return 0;
21 }
```

스택 메모리

Func1 함수 시작 구분선

Main함수 시작 구분선

		...
Int result	60	0x1000 - 20
Int b	20	0x1000 - 16
Int a	10	0x1000 - 12
Int z	CC	0x1000 - 8
Int y	20	0x1000 - 4
Int x	10	0x1000
	...	

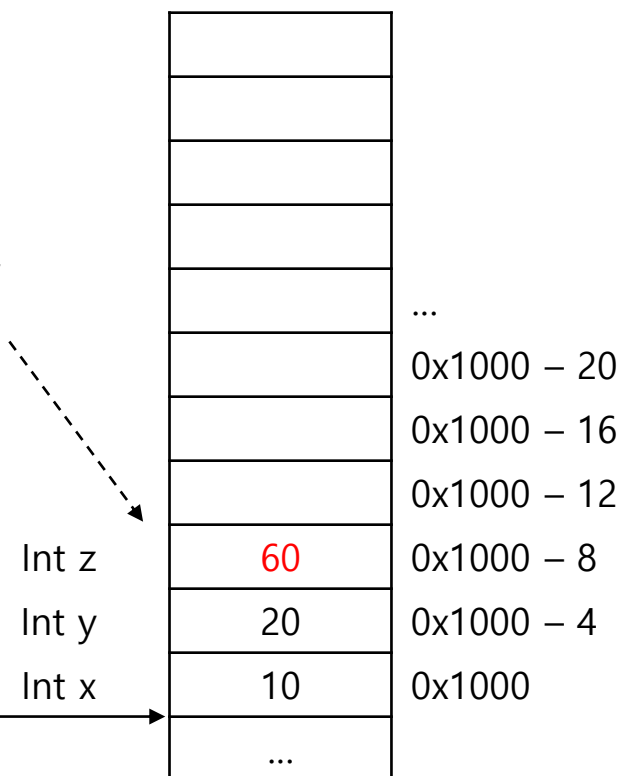

```
1 int Func2(int x, int y, int z)
2 {
3     x += y + z;
4     return x;
5 }
6 int Func1(int a, int b)
7 {
8     int result;
9     result = a + b;
10    result = Func2(result, a, b);
11    return result;
12 }
13 int main()
14 {
15     int x = 10;
16     int y = 20;
17     int z;
18     z = Func1(x, y);
19     cout << z << endl;
20     return 0;
21 }
```

Func1를 호출한
지점으로
돌아옵니다.

스택 메모리

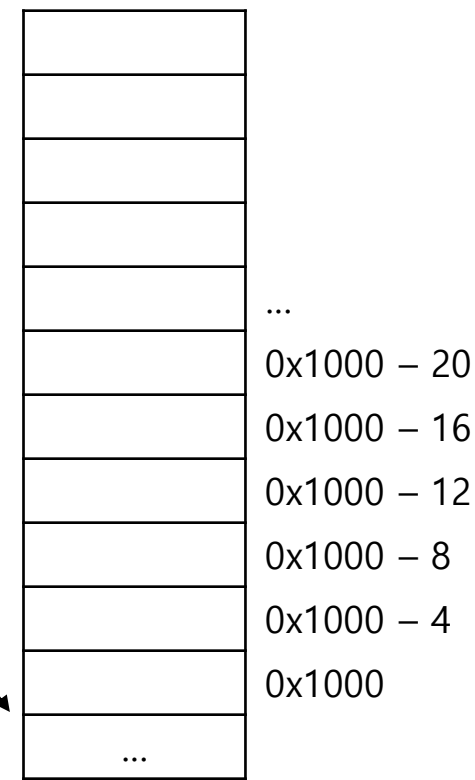
Func1이 끝났으니
자동으로 사용된
메모리는 없어지고
반환값은 z로
복사됩니다.

Main함수 시작 구분선



```
1 int Func2(int x, int y, int z)
2 {
3     x += y + z;
4     return x;
5 }
6 int Func1(int a, int b)
7 {
8     int result;
9     result = a + b;
10    result = Func2(result, a, b);
11    return result;
12 }
13 int main()
14 {
15     int x = 10;
16     int y = 20;
17     int z;
18     z = Func1(x, y);
19     cout << z << endl;
20     return 0;
21 }
```

스택 메모리

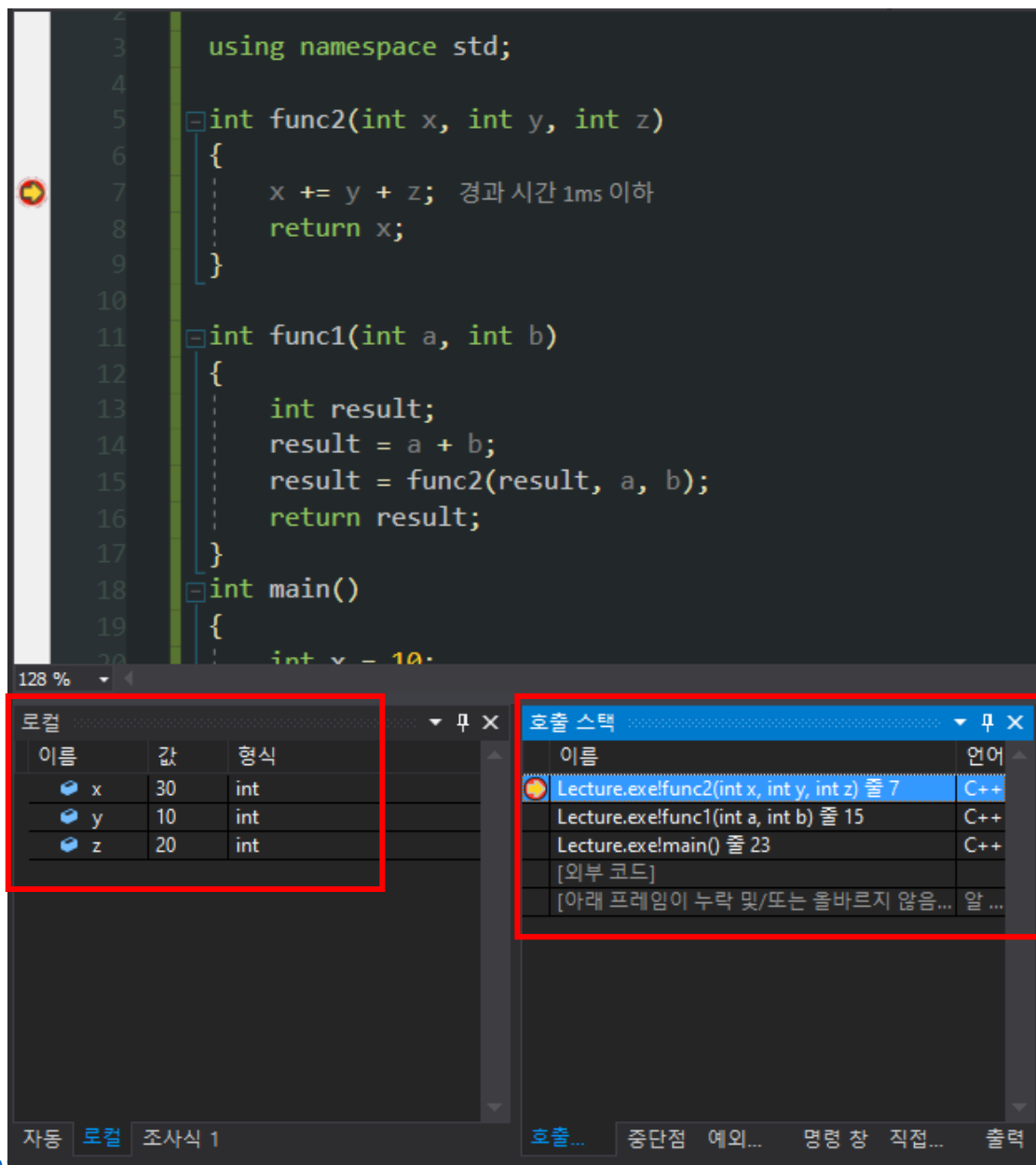


main이 끝났으니
자동으로 사용된
메모리는 없어지고
아무일도 없던것처럼
정리됩니다.



Check Function Calls in VS

중요!



로컬 창은 현재 호출 스택에 포함된 메모리(=지역변수 + 매개변수)의 상태를 보여줍니다.

호출 스택 창은 현재 시점(중단점이 걸린 시점)의 호출 스택(구분선이 생성된 상태)를 보여줍니다.

호출 스택창이 보이지 않으면 디버그→창→호출 스택 클릭 (디버깅(F5) 상태에서)

The screenshot shows the Visual Studio 2017 IDE. The main editor displays a C++ file named 'Lecture.cpp'. The code includes a `using namespace std;` statement and two functions: `func2` and `func1`. `func2` takes three integers and returns their sum. `func1` takes two integers, calculates their sum, and then calls `func2` with the sum and the two integers. The `main` function is partially visible at the bottom. The left margin shows line numbers 1 through 20. A red arrow points to the call stack icon in the left margin. Below the code editor, the 'Call Stack' window is open, showing a list of function calls. The top call is `Lecture.exe!func2(int x, int y, int z) 줄 7`. The second call is `Lecture.exe!func1(int a, int b) 줄 15`, which is highlighted with a red arrow. The third call is `Lecture.exe!main() 줄 23`. The bottom of the window shows the 'Locals' window with variables `a`, `b`, and `result`.

```
1  using namespace std;
2
3
4
5  int func2(int x, int y, int z)
6  {
7      x += y + z; 경과 시간 1ms 이하
8      return x;
9  }
10
11 int func1(int a, int b)
12 {
13     int result;
14     result = a + b;
15     result = func2(result, a, b);
16     return result;
17 }
18 int main()
19 {
20     int x = 10;
```

128 %

이름	값	형식
a	10	int
b	20	int
result	30	int

호출 스택	언어
Lecture.exe!func2(int x, int y, int z) 줄 7	C++
Lecture.exe!func1(int a, int b) 줄 15	C++
Lecture.exe!main() 줄 23	C++
[외부 코드]	
[아래 프레임이 누락 및/또는 올바르게 표시되지 않음...	알 ...

호출... 중단점 예외... 명령 창 직접... 출력

2) 그 호출 스택에 포함된 메모리의 상태를 보여줍니다.

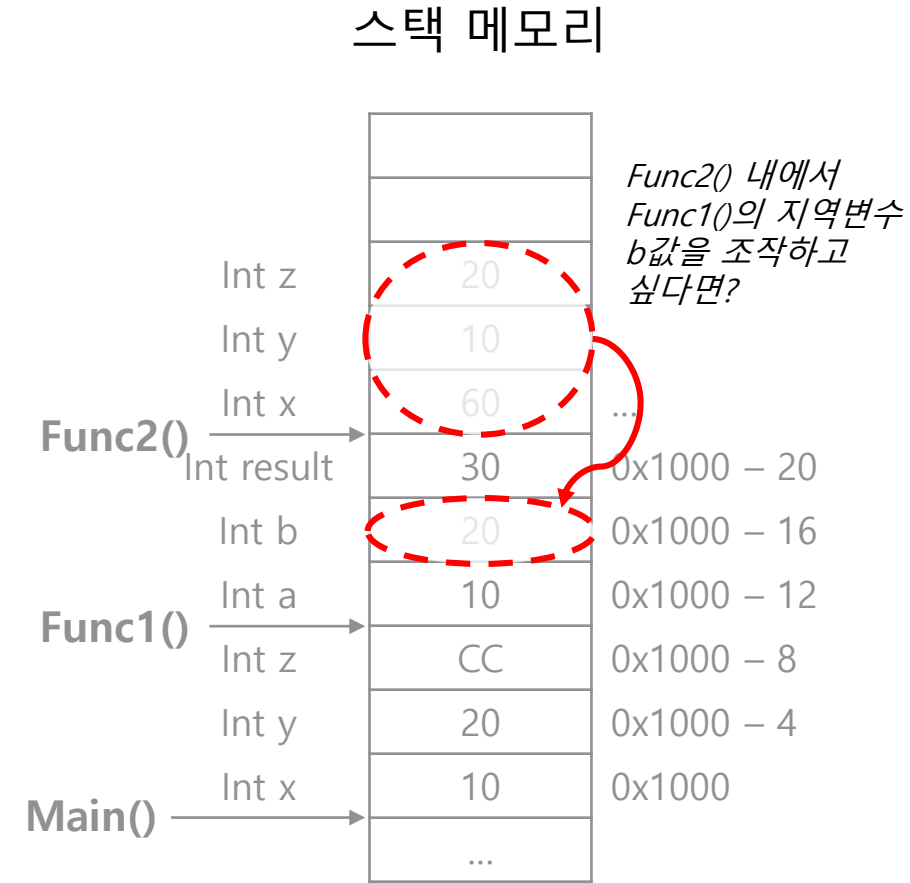
1) 다른 호출 스택을 더블클릭 하면

(Summary) Functions

- 정의
- 프로토타입
- 매개변수(parameter)와 pass-by-value☆
- return문
- 기본 인수(default argument)
- 오버로딩 ☆
- 함수 호출의 동작 방식 ☆
- 포인터와 참조자 ☆
- inline 함수
- 재귀 함수(recursive)

● 범위 밖 메모리의 조작

- Func2()를 실행 중일 때, main() 및 Func1()의 지역 변수들에 대한 접근은 불가능한 것일까?
- No. 하지만 이런 접근이 가능하려면 다른 타입의 변수를 사용해야 함
 1. **포인터**: 메모리 주소값을 갖는 변수
 2. **참조자**: 변수에 또다른 이름(별명)을 부여
- 정확한 내용은 다음 강의에서 설명하지만, 우선 소개



- 포인터를 함수로 전달하는 예시
 - 주소값을 명시하는 포인터를 활용하여 범위 밖 메모리에 접근할 수 있음

```
1 void ChangeValue(int* number);
2
3 int main()
4 {
5     int number = 10;
6     cout << number << endl; // 10
7     ChangeValue(&number);
8     cout << number << endl; // 20
9 }
10
11 void ChangeValue(int* number)
12 {
13     *number = 20;
14 }
```

**number는 main()의 지역
변수이지만 ChangeValue()
함수를 통해 값이 바뀌었음*

*즉, ChangeValue 함수는
자신의 범위 밖에 변수에
접근한 것.*

- 참조자로 전달

- 함수 내에서 범위 밖 변수값을 바꾸고 싶은 경우 사용하는 또다른 방법
 - 값의 변환을 위해서는 매개변수의 주소값(포인터)이 필요
- 배열이 아닌 경우에도 C++에서는 참조자(reference)를 통해 가능
 - C 언어를 사용한다면 포인터를 사용할 수밖에 없음
 - C++에서는 포인터 또는 참조자 두 가지 옵션이 존재
- 형식 매개변수를 실제 매개변수의 별명처럼 사용하는 개념

Pass-by-reference

- 참조자

- & 기호 사용

```
1 void ScaleNumber(int &num);
2
3 int main()
4 {
5     int number = 1000;
6     scaleNumber(number);
7     cout << number << endl; // 100
8     return 0;
9 }
10
11 void ScaleNumber(int &num)
12 {
13     if(num > 100)
14     {
15         num = 100;
16     }
17 }
```

**numbers는 main()의 지역
변수이지만 ScaleNumber()
함수를 통해 값이 바뀌었음!*

Pass-by-reference

- 참조자

- Swap 예제

```
1 void Swap(int &a, int &b);
2
3 int main()
4 {
5     int x=10, y=20;
6     cout << x << " " << y << endl; //10 20
7     Swap(x,y);
8     cout << x << " " << y << endl; //20 10
9     return 0;
10 }
11
12 void Swap(int &a, int &b)
13 {
14     int temp = a;
15     a = b;
16     b = temp;
17 }
```

● 참조자

- 참조자를 사용하지 않는 경우에는 print를 위해 메모리 두 배 사용
- 참조자를 사용하되, 값의 변경이 필요 없을 시에는 const로 안정성 확보

꼭 알아두어야 하는 내용이지만,
지금은 참조자에 대해 자세히
설명하지 않았기 때문에 일단
고급으로 분류합니다.

```
1 void print(const std::vector<int> &v);  
2  
3 int main()  
4 {  
5     std::vector<int> data {1,2,3,4,5};  
6     print(data);  
7     return 0;  
8 }  
9  
10 void print(const std::vector<int> &v)  
11 {  
12     for(auto num: v)  
13     {  
14         cout << num << endl;  
15     }  
16 }
```

(Summary) Functions

- 정의
- 프로토타입
- 매개변수(parameter)와 pass-by-value ☆
- return 문
- 기본 인수(default argument)
- 오버로딩 ☆
- 함수 호출의 동작 방식 ☆
- 배열의 전달과 pass-by-reference ☆
- **inline 함수**
- 재귀 함수(recursive)

inline function

- inline 함수

- 함수의 호출에는 어느 정도 오버헤드가 존재
 - Activation stack 생성, 파라미터 처리, pop stack, 리턴값 처리...
- 함수를 inline으로 정의하면 컴파일 단계에서 함수내의 명령문으로 함수 호출이 대체
 - 일반적인 함수 호출보다 빠름
 - 단, 바이너리 파일의 용량이 커질 수 있음
 - 내가 명시하지 않아도 컴파일러에서 최적화를 위해 내부적으로 알아서 처리하기도 함

```
1 inline int AddNumbers(int first,int second){
2     return first+second;
3 }
4 int main(){
5     int result = 0;
6     result = AddNumbers(100,200); // → 컴파일하면 함수 본문인 100 + 200으로 대체됨
7     return 0;
8 }
```

(Summary) Functions

- 정의
- 프로토타입
- 매개변수(parameter)와 pass-by-value ☆
- return 문
- 기본 인수(default argument)
- 오버로딩 ☆
- 함수 호출의 동작 방식 ☆
- 배열의 전달과 pass-by-reference ☆
- inline 함수
- 재귀 함수(recursive)

Recursive Function

- 재귀 함수

- 스스로를 호출하는 함수
- Factorial

➤ 재귀 호출을 끝내는 base case가 반드시 실행되어야 함 (stack overflow 주의!)

```
1 unsigned long long Factorial(unsigned long long n)
2 {
3     if(n==0) // Base case
4     {
5         return 1;
6     }
7     return n*Factorial(n-1);
8 }
9 int main()
10 {
11     cout << Factorial(5) << endl;
12     return 0;
13 }
```

(Summary) Functions

- 정의 (모듈화, 이름+매개변수+본문+리턴)
- 프로토타입 (호출 전에 컴파일러가 함수의 인자와 반환형을 알 수 있도록)
- 매개변수(parameter)와 pass-by-value (인자는 복사하여 전달된다)
- Return문
- 기본 인자(default argument) (기본값, 오른쪽부터 선언해준다)
- 오버로딩 (동일 이름, 유사 동작, 다른 타입)
- 함수 호출 동작방식
 1. 함수를 호출하면? → 구분선이 생김
 2. 구분선 안에는? → 지역변수 공간(매개변수 포함)
 3. 함수가 끝나면? → 자동으로 사라짐
- 포인터와 참조자 (원래는 접근할 수 없는 다른 메모리 영역의 변수를 변환하고 싶을 때 사용)
- inline 함수
- 재귀 함수(recursive)

추가 슬라이드

Multiple Return Values

- 참조자/포인터를 인자로 전달
 - 인자의 전달에 복사 비용이 적어 효율적
- 같은 타입인 경우 배열(std 배열, vector 등)을 활용
- `std::tuple` 활용
 - `Std::make_pair()`, `std::get()`
- struct/class 활용

Memory Layout Additions

- Release 모드에서 컴파일하면 지역변수들이 연속된 메모리 공간에 할당되나, 디버그 모드에서 컴파일하면, 중간중간에 공백이 존재함

[illegible]

- 스택 프레임에는 지역 변수, 인자 이외에 반환 주소값 등이 포함됨
 - http://tcpschool.com/c/c_memory_stackframe
 - <http://www.cs.uwm.edu/classes/cs315/Bacon/Lecture/HTML/ch10s07.html>