

#include <iostream>

1

- #include <iostream>
 - ▣ 전처리기(C++ Preprocessor)에게 내리는 지시
 - <iostream> 헤더 파일을 컴파일 전에 소스에 확장하도록 지시
- <iostream> 헤더 파일
 - ▣ 표준 입출력을 위한 클래스와 객체, 변수 등이 선언됨
 - ios, istream, ostream, iostream 클래스 선언
 - cout, cin, <<, >> 등 연산자 선언

```
#include <iostream>
```

```
....
```

```
std::cout << "HelloWn";
```

```
std::cout << "첫 번째 맛보기입니다.";
```

화면 출력(<<)

2

□ cout과 << 연산자 이용

```
std::cout << "Hello\n"; // 화면에 Hello를 출력하고 다음 줄로 넘어감  
std::cout << "첫 번째 맛보기입니다.";
```

□ cout 객체

- 스크린 출력 장치에 연결된 **표준** C++ 출력 스트림 객체
- <iostream> 헤더 파일에 선언
- std 이름 공간에 선언: **std::cout**으로 사용

□ << 연산자

- 스트림 연산자(stream operator)
 - C++ 기본 산술 시프트 연산자(<<)$C++$가 스트림 연산자로 재정의됨
 - ostream 클래스에 구현됨
 - 오른쪽 피연산자를 왼쪽 스트림 객체에 삽입
 - cout 객체에 연결된 화면에 출력
- 여러 개의 << 연산자로 여러 값 출력

```
std::cout << "Hello\n" << "첫 번째 맛보기입니다.";
```

namespace 개념

3

- 이름(identifier) 충돌이 발생하는 경우
 - 여러 명이 서로 나누어 프로젝트를 개발하는 경우
 - 오픈 소스 혹은 다른 사람이 작성한 소스나 목적 파일을 가져와서 컴파일하거나 링크하는 경우
 - 해결하는데 많은 시간과 노력이 필요
- namespace 키워드
 - 이름 충돌 해결
 - 2003년 새로운 C++ 표준에서 도입
 - 개발자가 자신만의 이름 공간을 생성할 수 있도록 함
 - 이름 공간 안에 선언된 이름은 다른 이름공간과 별도 구분
- 이름 공간 생성 및 사용(자세한 것은 부록 B 참고)

```
namespace kitae { // kitae 라는 이름 공간 생성
..... // 이 곳에 선언된 모든 이름은 kitae 이름 공간에 생성된 이름
}
```

- 이름 공간 사용
 - 이름 공간 :: 이름

std:: 란?

4

□ std

- C++ 표준에서 정의한 **이름 공간(namespace)** 중 하나
 - <iostream> 헤더 파일에 선언된 모든 이름: std 이름 공간 안에 있음
 - cout, cin, endl 등
- std 이름 공간에 선언된 이름을 접근하기 위해 std:: 접두어 사용
 - std::cout, std::cin, std::endl

□ std:: 생략

- using 지시어 사용

```
using std::cout; // cout에 대해서만 std:: 생략
```

std:: 생략

```
.....  
cout << "Hello" << std::endl; // std::cout에서 std:: 생략
```

```
using namespace std; // std 이름 공간에 선언된 모든 이름에 std:: 생략
```

```
.....  
cout << "Hello" << endl; // std:: 생략
```

std:: 생략

std:: 생략

#include <iostream>과 std

5

- <iostream>이 통째로 std 이름 공간 내에 선언
 - ▣ <iostream> 헤더 파일을 사용하려면 다음 코드 필요

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

cin과 >> 연산자를 이용한 키 입력

6

□ cin

- ▣ 표준 입력 장치인 키보드를 연결하는 C++ 입력 스트림 객체

□ >> 연산자

- ▣ 스트림 연산자(stream operator)

- C++ 산술 시프트 연산자(>>)가 <iostream> 헤더 파일에 스트림 연산자로 재정의됨
- 입력 스트림에서 값을 읽어 변수에 저장

- ▣ 연속된 >> 연산자를 사용하여 여러 값 입력 가능

```
cout << "너비와 높이를 입력하세요>>";  
cin >> width >> height;  
cout << width << 'Wn' << height << 'Wn';
```

너비와 높이를 입력하세요>>23 36

23

36

width에
입력

height에
입력

<Enter> 키를 칠 때 변수에 값 전달

7

- cin의 특징
 - ▣ 입력 버퍼를 내장하고 있음
 - ▣ <Enter>키가 입력될 때까지 입력된 키를 입력 버퍼에 저장
 - 도중에 <Backspace> 키를 입력하면 입력된 키 삭제
- >> 연산자
 - ▣ <Enter>키가 입력되면 바로소 cin의 입력 버퍼에서 키 값을 읽어 변수에 전달

C++ 문자열

8

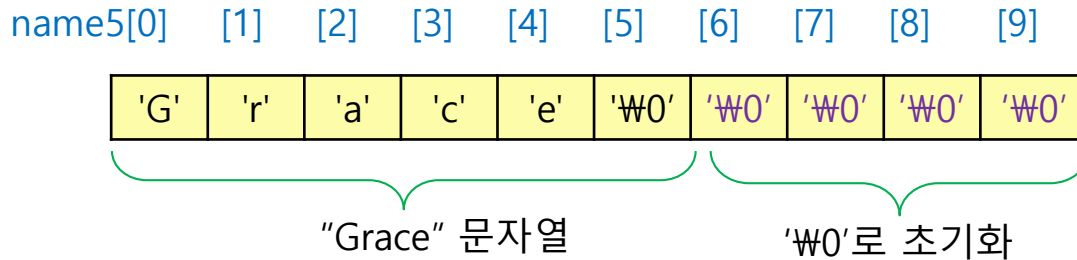
- C++의 문자열 표현 방식 : 2가지
 - ▣ C-스트링 방식 - '\0'로 끝나는 문자 배열

C-스트링
문자열

단순 문자
배열

```
char name1[6] = {'G', 'r', 'a', 'c', 'e', '\0'}; // name1은 문자열 "Grace"  
char name2[5] = {'G', 'r', 'a', 'c', 'e'}; // name2는 문자열이 아니고 단순 문자 배열
```

```
char name5[10] = "Grace";
```



- ▣ string 클래스 이용
 - <string> 헤더 파일에 선언됨
 - 다양한 멤버 함수 제공, 문자열 비교, 복사, 수정 등

C-스트링 방식으로 문자열 다루기

9

- C-스트링으로 문자열 다루기
 - ▣ C 언어에서 사용한 함수 사용 가능
 - strcmp(), strlen(), strcpy() 등
 - ▣ <cstring>이나 <string.h> 헤더 파일 include

```
#include <cstring> 또는  
#include <string.h>  
...  
int n = strlen("hello");
```

- ▣ <cstring> 헤더 파일을 사용하는 것이 바람직함
 - <cstring>이 C++ 표준 방식

cin을 이용한 문자열 입력

10

□ 문자열 입력

```
char name[6]; // 5 개의 문자를 저장할 수 있는 char 배열  
cin >> name; // 키보드로부터 문자열을 읽어 name 배열에 저장한다.
```

Grace

키 입력

name [0] [1] [2] [3] [4] [5]

'G'	'r'	'a'	'c'	'e'	'\0'
-----	-----	-----	-----	-----	------

"Grace" 문자열

cin.getline()으로 공백이 낀 문자열 입력

11

- 공백이 낀 문자열을 입력 받는 방법
- cin.getline(char buf[], int size, char delimiterChar)
 - ▣ buf에 최대 size-1개의 문자 입력. 끝에 '\0' 붙임
 - ▣ delimiterChar를 만나면 입력 중단. 끝에 '\0' 붙임
 - delimiterChar의 디폴트 값은 '\n'(<Enter>키)

```
char address[100];  
cin.getline(address, 100, '\n');
```

최대 99개의 문자를 읽어 address 배열
에 저장. 도중에 <Enter> 키를 만나면
입력 중단

사용자가 'Seoul Korea<Enter>'를 입력할 때,

address[0] [1] [2] [3] [4] [5] [6] [7] [8] [9] [10] [11] [99]

'S'	'e'	'o'	'u'	'l'	' '	'K'	'o'	'r'	'e'	'a'	'\0'
-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	------	-----	-----

"Seoul Korea" 문자열

C++에서 문자열을 다루는 string 클래스

12

□ string 클래스

- ▣ C++에서 강력 추천
- ▣ C++ 표준 클래스
- ▣ 문자열의 크기에 따른 제약 없음
 - string 클래스가 스스로 문자열 크기게 맞게 내부 버퍼 조절
- ▣ 문자열 복사, 비교, 수정 등을 위한 다양한 함수와 연산자 제공
- ▣ 객체 지향적
- ▣ <string> 헤더 파일에 선언
 - #include <string> 필요
- ▣ C-스트링보다 다루기 쉬움

표준 C++ 헤더 파일은 확장자가 없다

13

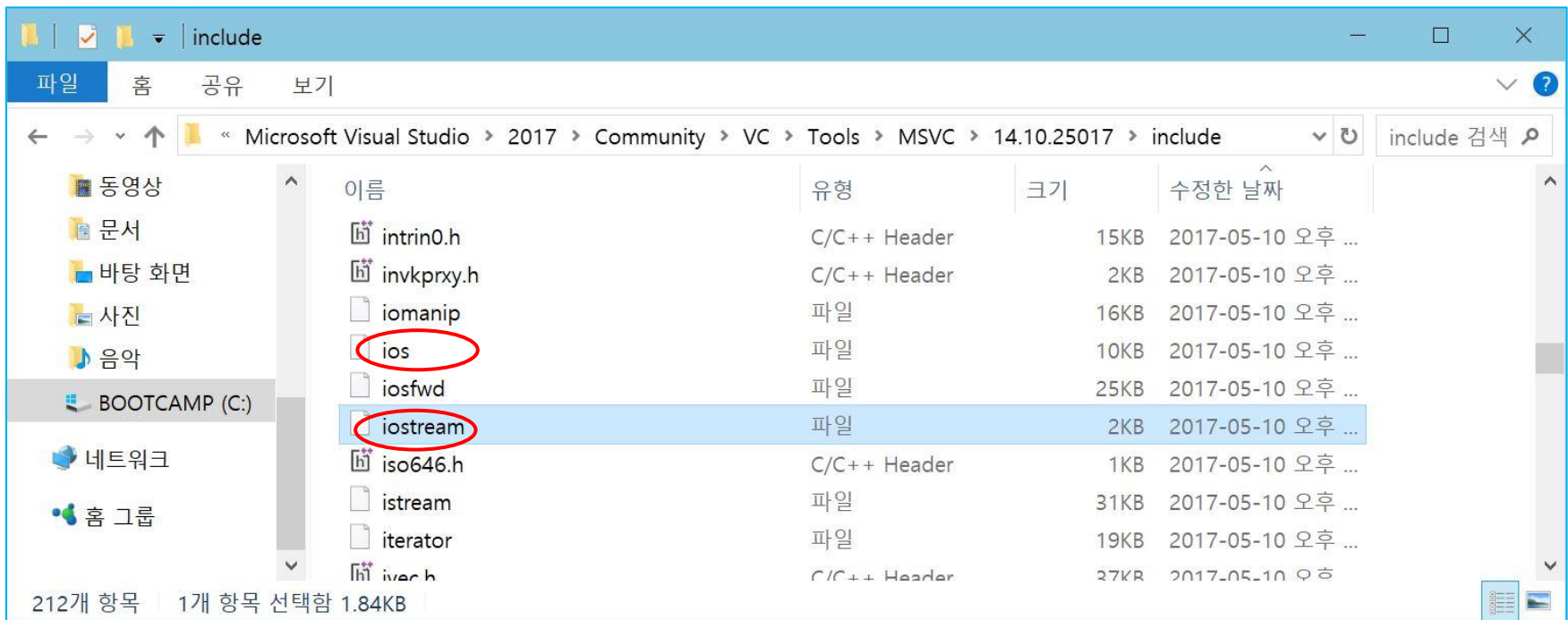
- 표준 C++에서 헤더 파일 확장자 없고, std 이름 공간 적시
 - ▣ `#include <iostream>`
 - ▣ `using namespace std;`
- 헤더 파일의 확장자 비교

언어	헤더 파일 확장자	사례	설명
C	.h	<code><string.h></code>	C/C++ 프로그램에서 사용 가능
C++	확장자 없음	<code><cstring></code>	<code>using namespace std;</code> 와 함께 사용해야 함

<iostream> 헤더 파일은 어디에?

14

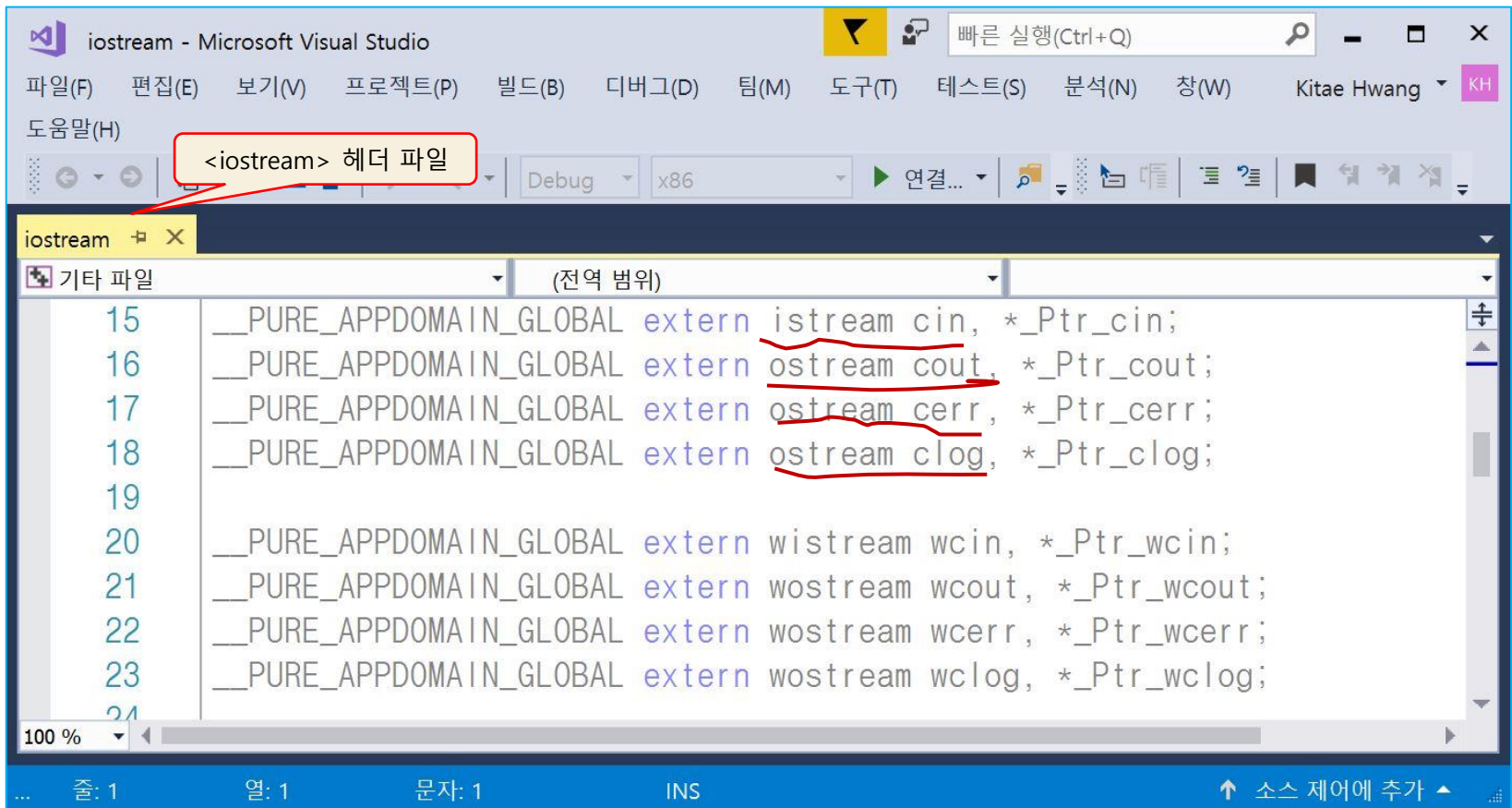
- iostream 파일은 확장자 없는 텍스트 파일
- 컴파일러가 설치된 폴더 아래 include 폴더에 존재
 - C:\Program Files(x86)\Microsoft Visual Studio\2017\Community\VC\Tools\MSVC\14.10.25017\include



cin과 cout은 어디에 선언되어 있는가?

15

- cout이나 cin은 모두 <iostream>에 선언된 객체



```
iostream - Microsoft Visual Studio
빠른 실행(Ctrl+Q)
파일(F) 편집(E) 보기(V) 프로젝트(P) 빌드(B) 디버그(D) 팀(M) 도구(T) 테스트(S) 분석(N) 창(W) Kitae Hwang KH
도움말(H)
<iostream> 헤더 파일
Debug x86 연결...
iostream
기타 파일 (전역 범위)
15 __PURE_APPDOMAIN_GLOBAL extern istream cin, *_Ptr_cin;
16 __PURE_APPDOMAIN_GLOBAL extern ostream cout, *_Ptr_cout;
17 __PURE_APPDOMAIN_GLOBAL extern ostream cerr, *_Ptr_cerr;
18 __PURE_APPDOMAIN_GLOBAL extern ostream clog, *_Ptr_clog;
19
20 __PURE_APPDOMAIN_GLOBAL extern wistream wcin, *_Ptr_wcin;
21 __PURE_APPDOMAIN_GLOBAL extern wostream wcout, *_Ptr_wcout;
22 __PURE_APPDOMAIN_GLOBAL extern wostream wcerr, *_Ptr_wcerr;
23 __PURE_APPDOMAIN_GLOBAL extern wostream wclog, *_Ptr_wclog;
24
100 %
... 줄: 1 열: 1 문자: 1 INS ↑ 소스 제어에 추가
```

헤더 파일에는 무엇이 들어 있는가?

16

- 질문1) <cstring>파일에 strcpy() 함수의 코드가 들어 있을까?
 - (1) strcpy() 함수의 코드가 들어 있다(O, X).
 - 답은 X
 - (2) strcpy() 함수의 원형이 선언되어 있다(O, X).
 - 답은 O

- 질문 2) 그러면 strcpy() 함수의 코드는 어디에 있는가?
 - 답) strcpy() 함수의 코드는 컴파일된 바이너리 코드로, 비주얼 스튜디오가 설치된 lib 폴더에 libcmt.lib 파일에 들어 있고
 - 링크 시에 strcpy() 함수의 코드가 exe에 들어간다.

- 질문 2) 그러면 헤더 파일은 왜 사용되는가?
 - 답) 사용자 프로그램에서 strcpy() 함수를 호출하는 구문이 정확한지 확인하기 위해 컴파일러에 의해 필요

참조자?

17

- 레퍼런스(Reference)의 이해 : 개체에게 지어주는 별명이다.
- 레퍼런스 선언이란 이름이 존재하는 메모리 공간에 하나의 이름을 더 부여하는 행위이다.

```
int a=100;
```

```
int &r = a;    // 레퍼런스 선언을 위한 연산자
```

```
int *p = &a;   // 주소 값을 얻기 위한 주소 연산자
```

- 레퍼런스의 제약

```
int & ref1;      //오류, 선언과 동시에 초기화되어야 한다
```

```
int & ref2  = 10; //오류, 이름이 부여된 메모리 공간,  
                즉 변수이어야 한다.
```

레퍼런스를 이용한 성능의 향상

18

- `void ShowData(Person p) // 구조체가 메모리에 할당된다`
`{ ... }`
- `void ShowData(Person &p) // 같은 메모리 참조하여 메모리는 절약되나`
`{ ... }` 데이터를 변경하는 우를 범할 수도 있다
- `void ShowData(const Person &p) // 같은 메모리 참조하여 메모리`
`{ ... }` 도 절약하고 데이터를 변경하는 우를 범하지 않는다.
- 레퍼런스 선언앞에 `const`를 주면 `p`를 통한 데이터 조작은 허용하지 않겠다
- 지역변수는 레퍼런스로 리턴되어서는 안된다.
(그 함수에서 소멸하기 때문)

메서드 오버로딩

- 함수 오버로딩이란? (함수 중복정의)
 - 동일한 이름의 함수를 중복해서 정의하는 것!
- 함수 오버로딩의 조건
 - 매개 변수의 개수 혹은 타입이 일치하지 않는다.
- 함수 오버로딩이 가능한 이유
 - 호출할 함수를 매개 변수의 정보까지 참조해서 호출
 - 함수의 이름 + 매개 변수의 정보
 - 함수의 반환형은 오버로딩 조건에 포함되지 않는다.

메서드 오버로딩

```
int function1(int n) {...}
```

```
int function1(char c) {...}
```

```
int function2(int v) {...}
```

```
int function2(int v1, int v2) {...}
```

디폴트 매개 변수

- 디폴트 매개 변수란?
 - 전달되지 않은 인자를 대신하기 위한 기본 값이 설정되어 있는 변수

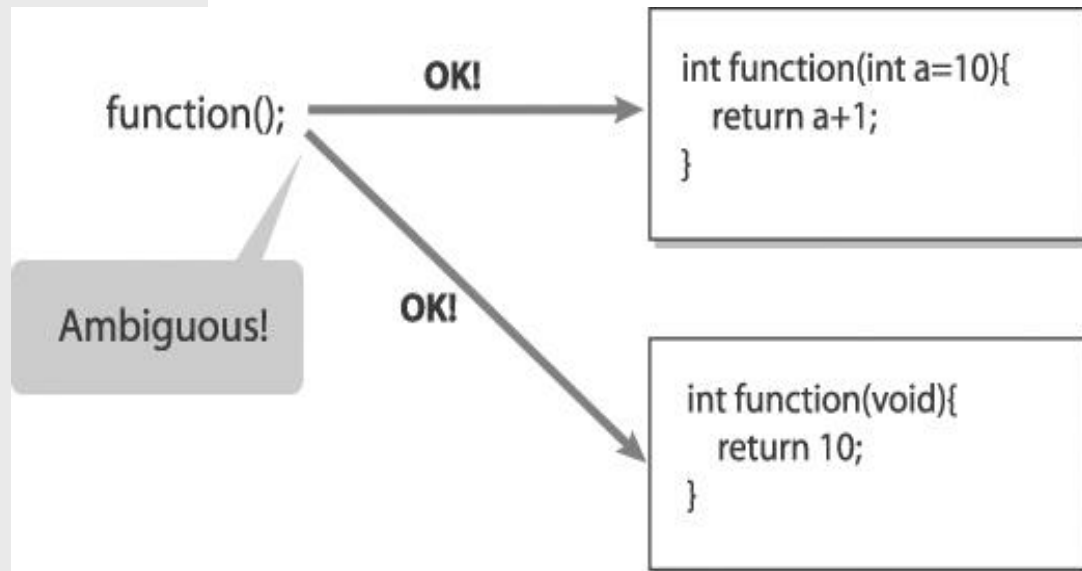
```
int function( int a = 0 )  
{  
    return a+1;  
}
```

디폴트
매개변수

디폴트 매개 변수

■ 디폴트 매개변수 vs. 함수 오버로딩

```
int function(int a=10){  
    return a+1;  
}  
// 모호한 호출  
int function(){  
    return 10;  
}  
int main(void)  
{  
    std::cout<<function()<<std::endl;  
    std::cout<<function(10)<<std::endl;  
    return 0;  
}
```



동적 메모리 할당 및 반환

23

- 정적 할당
 - ▣ 변수 선언을 통해 필요한 메모리 할당
 - 많은 양의 메모리는 배열 선언을 통해 할당
- 동적 할당
 - ▣ 필요한 양이 예측되지 않는 경우. 프로그램 작성시 할당 받을 수 없음
 - ▣ 실행 중에 힙 메모리에서 할당
 - 힙(heap)으로부터 할당
 - 힙은 운영체제가 프로세스(프로그램)의 실행을 시작 시킬 때 동적 할당 공간으로 준 메모리 공간
- C 언어의 동적 메모리 할당 : malloc()/free() 라이브러리 함수 사용
- C++의 동적 메모리 할당/반환
 - ▣ new 연산자
 - 기본 타입 메모리 할당, 배열 할당, 객체 할당, 객체 배열 할당
 - 객체의 동적 생성 - 힙 메모리로부터 객체를 위한 메모리 할당 요청
 - 객체 할당 시 생성자 호출
 - ▣ delete 연산자
 - new로 할당 받은 메모리 반환
 - 객체의 동적 소멸 - 소멸자 호출 뒤 객체를 힙에 반환

new와 delete 연산자

24

- C++의 기본 연산자
- new/delete 연산자의 사용 형식

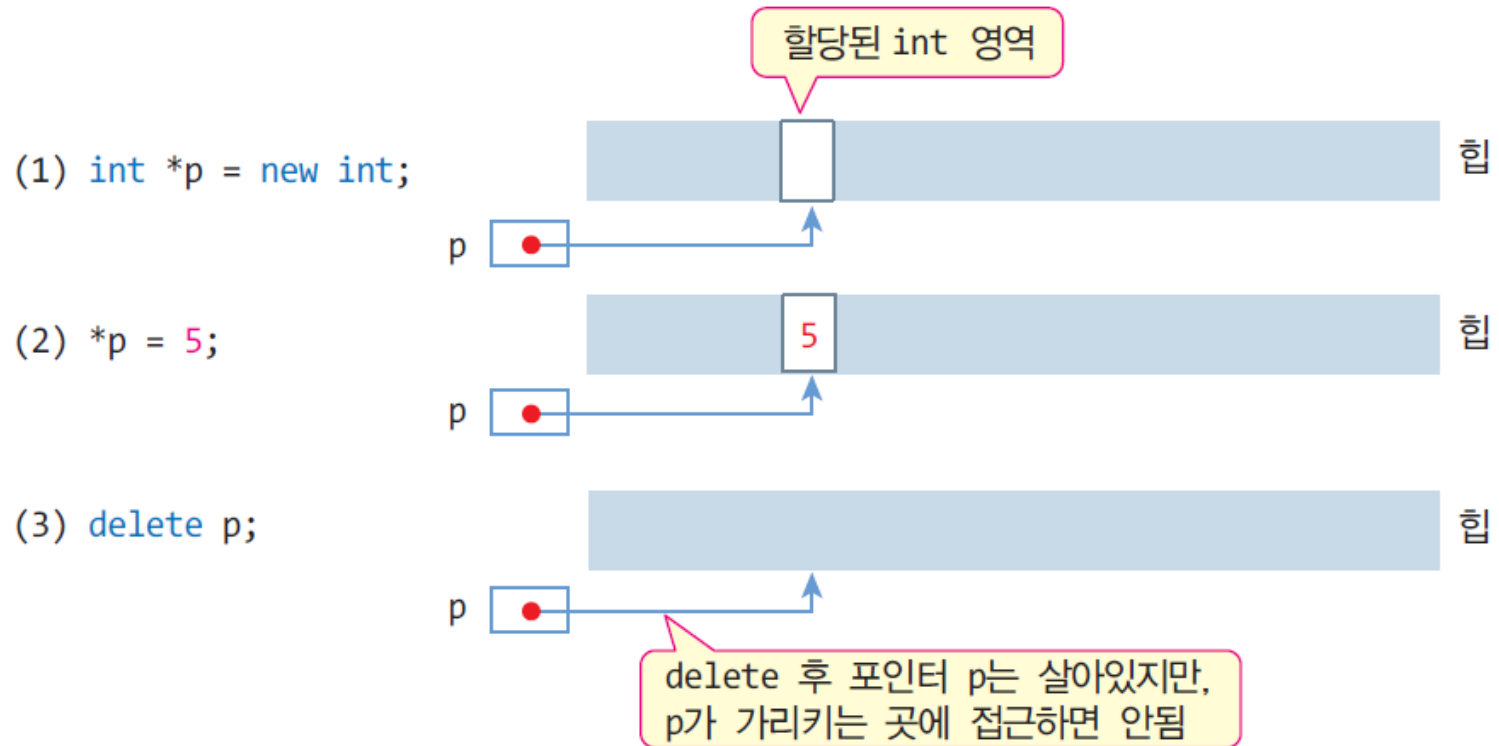
```
데이터타입 *포인터변수 = new 데이터타입 ;  
delete 포인터변수;
```

- new/delete의 사용

```
int *pInt = new int; // int 타입의 메모리 동적 할당  
char *pChar = new char; // char 타입의 메모리 동적 할당  
Circle *pCircle = new Circle(); // Circle 클래스 타입의 메모리 동적 할당  
  
delete pInt; // 할당 받은 정수 공간 반환  
delete pChar; // 할당 받은 문자 공간 반환  
delete pCircle; // 할당 받은 객체 공간 반환
```


기본 타입의 메모리 동적 할당 및 반환

25



delete 사용 시 주의 사항

26

- 적절치 못한 포인터로 delete하면 실행 시간 오류 발생
 - ▣ 동적으로 할당 받지 않는 메모리 반환 - 오류

```
int n;  
int *p = &n;  
delete p; // 실행 시간 오류  
// 포인터 p가 가리키는 메모리는 동적으로 할당 받은 것이 아님
```

- ▣ 동일한 메모리 두 번 반환 - 오류

```
int *p = new int;  
delete p; // 정상적인 메모리 반환  
delete p; // 실행 시간 오류. 이미 반환한 메모리를 중복 반환할 수 없음
```

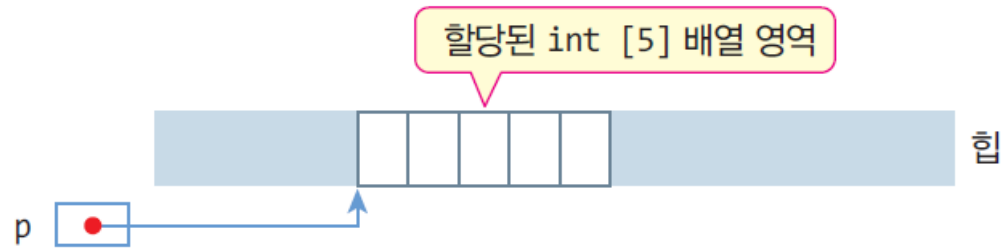
배열의 동적 할당 및 반환

27

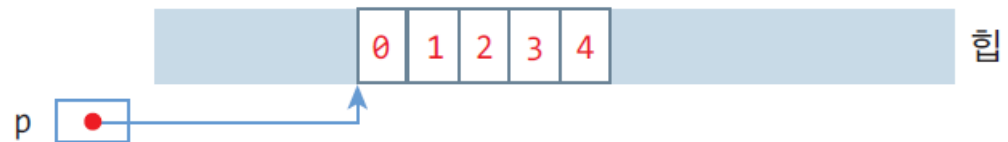
□ new/delete 연산자의 사용 형식

데이터타입 *포인터변수 = **new** 데이터타입 [배열의 크기]; // 동적 배열 할당
delete [] 포인터변수; // 배열 반환

(1) `int *p = new int [5];`



(2) `for(int i=0; i<5; i++)`
 `p[i] = i;`



(3) `delete [] p;`

