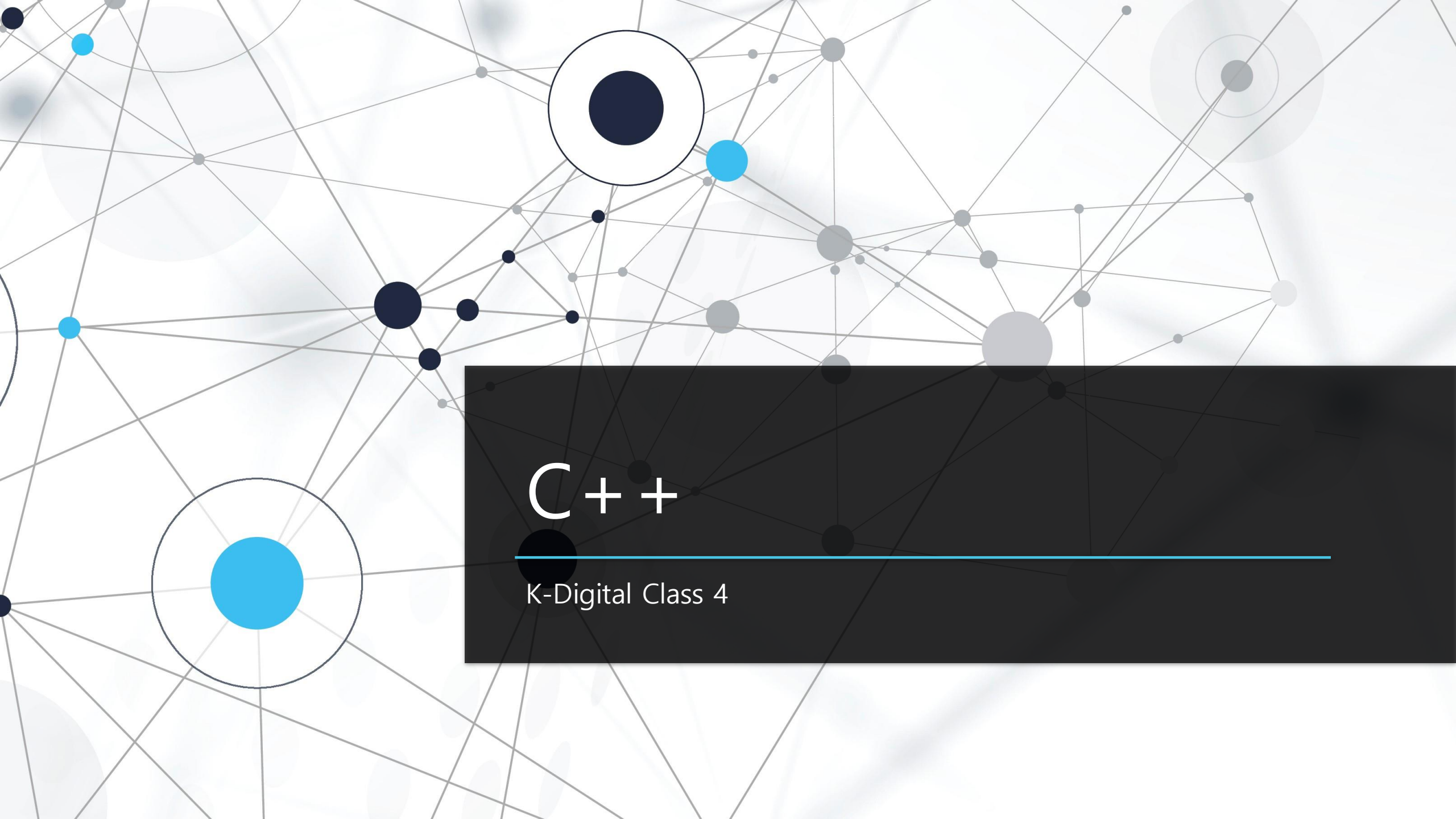


C++

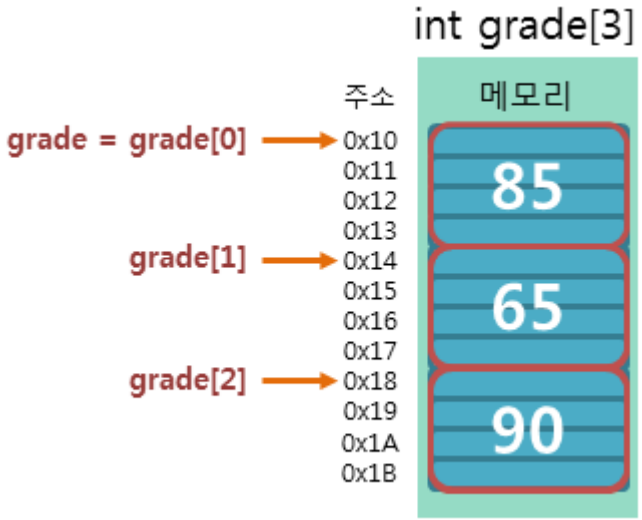
K-Digital Class 4

C++ ARRAYS AND POINTER

C++ Arrays

- 배열은 각 값에 대해 별도의 변수를 선언하는 대신 단일 변수에 여러 값을 저장하는 데 사용된다.
- 배열을 선언하려면 변수 유형을 정의하고 배열 이름과 대괄호를 지정하고 저장해야 하는 요소 수를 지정한다.
  - type arrayName[length];

```
string cars[4];
```



C++ ARRAYS AND POINTER

C++ Omit Arrays

- 배열의 크기를 지정 하지 않고도 선언할 수 있다.
- 그러나 삽입된 요소만큼만 크기가 커진다.

```
string cars[] = {"Volvo", "BMW", "Ford"}; // size of array is always 3
```

C++ ARRAYS AND POINTER

C++ References

- Reference variable(참조 변수)는 기존 변수에 대한 “참조”이며 & 연산자로 생성된다.

```
string food = "Pizza";  // food variable
string &meal = food;    // reference to food
```

C++ ARRAYS AND POINTER

C++ Memory Address

- 이전 페이지의 예에서는 & 연산자를 사용하여 참조 변수를 생성했다.
- 그러나 변수의 메모리 주소를 얻는 데에도 사용할 수 있다. 메모리에 변수가 저장되는 위치이다.
- C++에서 변수를 생성하면 변수에 메모리 주소가 할당된다. 그리고 변수에 값을 할당하면 이 메모리 주소에 저장된다.
- access하려면 & 연산자를 사용하 변수가 저장된 위치를 나타낸다.

```
string food = "Pizza";

cout << &food; // Outputs 0x6dfed4
```

메모리 주소를 아는 것이 왜 유용한가?
참조와 포인터는 C++에서 중요하다.
왜냐하면 컴퓨터 메모리의 데이터를 조작할 수 있는 능력을 제공하기 때문이다.
이러한 것은 code를 줄이고 성능을 향상시킬 수 있다.
이 두 가지 기능은 C++를 Python 및 Java와 같은 다른 프로그래밍 언어와 차별화하는 요소 중 하나이다.

C++ ARRAYS AND POINTER

C++ Pointer

- 변수가 선언되면 그 값을 저장하는 데 필요한 메모리의 특정 위치(메모리 주소)가 할당된다.
- 일반적으로 C++ 프로그램은 변수가 저장되는 정확한 메모리 주소를 능동적으로 결정하지 않는다.
- 다행히도 그 작업은 프로그램이 실행되는 환경에 맡겨집니다. 일반적으로 런타임 시 특정 메모리 위치를 결정하는 것은 운영 체제이다.
- 그러나 프로그램이 런타임 중에 변수에 대한 특정 위치에 있는 데이터 셀에 access(접근)하기 위해 변수의 주소를 얻어오는 것이 유용할 수 있다.
- C++에서 포인터(pointer)란 메모리의 주소 값을 저장하는 변수이며, 포인터 변수라고도 부른다.
- Char형 변수가 문자를 저장하고, int형 변수가 정수를 저장하는 것처럼 포인터는 주소 값을 저장하는 데 사용된다.

C++ ARRAYS AND POINTER

Address-of (주소) operator - &

- 변수의 주소는 변수 이름 앞에 주소 연산자라고 하는 ampersand 기호 (&)를 붙여서 얻을 수 있다.

Dereference(역 참조) operator-*

- 다른 변수의 주소를 저장하는 변수를 포인터라고 한다.
- 포인터는 주소가 저장되어 있는 변수를 "가리키는" 것이다.

```
string food = "Pizza";  // A string variable
string* ptr = &food;  // A pointer variable that stores the address of food
```

C++ ARRAYS AND POINTER

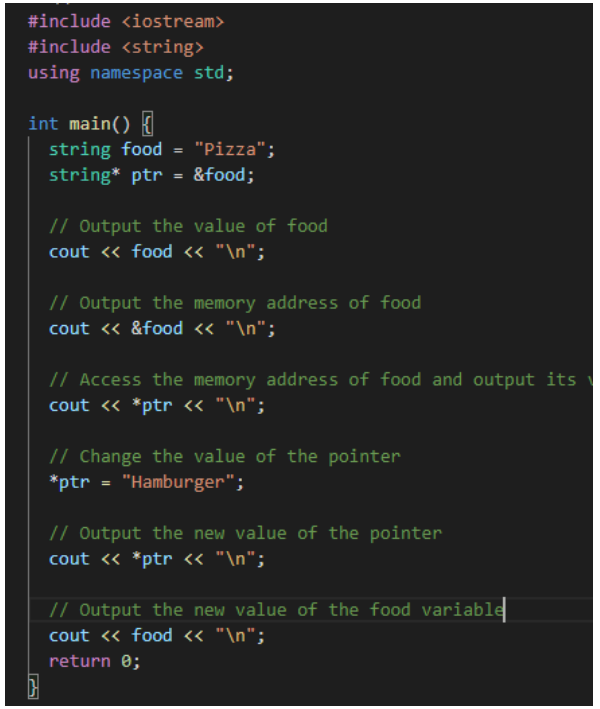
Declaring pointers

- 포인터 변수 선언의 일반적인 형식은 다음과 같다.
- type * name;
  - int * number;
  - char * character;
  - double * decimals;

C++ ARRAYS AND POINTER

C++ Modify Pointers

- 포인터의 값을 변경할 수도 있다.
- 그러나 이렇게 하면 원래 변수의 값도 변경된다.



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

int main() {
  string food = "Pizza";
  string* ptr = &food;

  // Output the value of food
  cout << food << "\n";

  // Output the memory address of food
  cout << &food << "\n";

  // Access the memory address of food and output its
  cout << *ptr << "\n";

  // Change the value of the pointer
  *ptr = "Hamburger";

  // Output the new value of the pointer
  cout << *ptr << "\n";

  // Output the new value of the food variable
  cout << food << "\n";
  return 0;
}
```

C++ ARRAYS AND POINTER

C++ Pointers and arrays

- 배열의 개념은 포인터의 개념과 관련이 있다. 실제로 배열의 첫 번째 요소는 포인터와 매우 유사하게 작동하며 실제로 배열은 항상 적절한 유형의 포인터로 묵시적으로 변환될 수 있다.
- 포인터와 배열은 동일한 작업 집합을 지원하며 둘 다 동일한 의미를 갖는다. 주요 차이점은 포인터에는 새 주소를 할당할 수 있지만 배열에는 할당할 수 없다는 것이다.

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

int main ()
{
  int numbers[5];
  int * p;
  p = numbers;  *p = 10;
  p++;  *p = 20;
  p = &numbers[2];  *p = 30;
  p = numbers + 3;  *p = 40;
  p = numbers;  *(p+4) = 50;
  for (int n=0; n<5; n++)
    cout << numbers[n] << ", ";
  return 0;
}
```

C++ ARRAYS AND POINTER

C++ Pointers and arrays

- 배열에 대한 장에서 대괄호([])는 배열 요소의 인덱스를 지정하는 것이다. 사실 이 괄호는 오프셋 연산자로 알려진 역참조 연산자이다. 그들은 뒤따르는 변수를 *처럼 역참조하지만 역참조되는 주소에 대괄호 사이의 숫자도 추가한다.
- 이 두 표현식은 포인터인 경우 뿐 아니라 가 배열인 경우에도 동일하고 유효하다. 배열인 경우 해당 이름을 첫 번째 요소에 대하여 포인터처럼 사용할 수 있음을 기억해야 한다.

```
a[5] = 0;       // a [offset of 5] = 0
*(a+5) = 0;     // pointed to by (a+5) = 0
```

C++ ARRAYS AND POINTER

C++ Pointer initialization

- 포인터를 선언한 후 참조 연산자(*)를 사용하기 전에 포인터는 반드시 초기화되어야 한다.
- 초기화하지 않은 채로 참조 연산자를 사용하게 되면, 어딘지 알 수 없는 메모리 장소에 값을 저장하는 것이 된다.

```cpp
#include <iostream>

using namespace std;

int main () {
   int  var = 20;   // actual variable declaration.
   int  *ip;        // pointer variable

   ip = &var;       // store address of var in pointer variable

   cout << "Value of var variable: ";
   cout << var << endl;

   // print the address stored in ip pointer variable
   cout << "Address stored in ip variable: ";
   cout << ip << endl;

   // access the value at the address available in pointer
   cout << "Value of *ip variable: ";
   cout << *ip << endl;

   return 0;
}
```

C++ ARRAYS AND POINTER

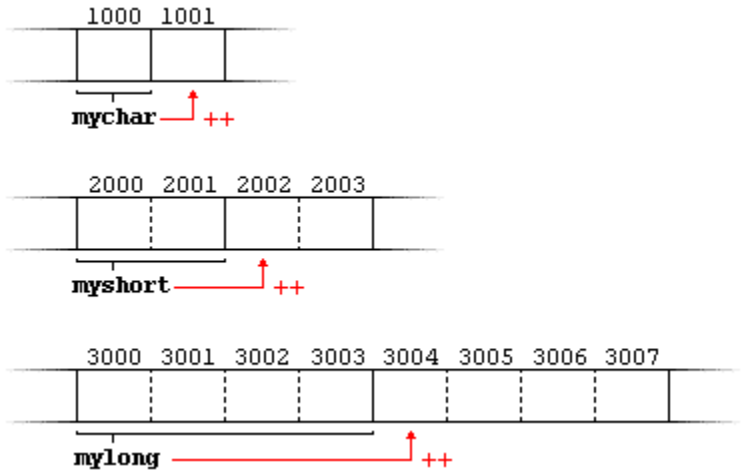
C++ Pointer arithmetics(pointer 연산)

- 포인터에서 산술 연산을 수행하는 것은 일반 정수 유형에서 수행하는 것과 약간 다르다.
- 우선 덧셈과 뺄셈 연산만 허용되며 나머지 연산은 포인터의 세계에서 의미가 없다.
- 그러나 덧셈과 뺄셈은 포인터가 가리키는 데이터 유형의 크기에 따라 포인터에 대해 다른 동작을 한다.
- 기본 데이터 유형은 크기가 서로 다른 것을 배웠다. 예를 들어: char의 크기는 항상 1바이트이고 short는 일반적으로 그보다 크며 int와 long은 훨씬 더 큰다. 이들의 정확한 크기는 시스템에 따라 다르다. 예를 들어, 사용하는 시스템에서 char은 1바이트, short는 2바이트, long은 4를 사용한다고 가정해 보겠다.

C++ ARRAYS AND POINTER

C++ Pointer arithmetics(pointer 연산)

- char *mychar;
- short *myshort;
- long *mylong;

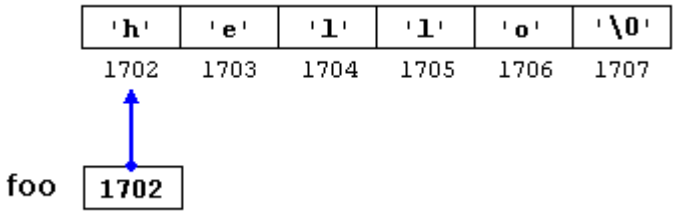


- ++mychar;
- ++myshort;
- ++mylong;

C++ ARRAYS AND POINTER

C++ Pointers and string literals

- 문자열 literal은 null로 끝나는 문자 sequence를 포함하는 배열이다.
- 문자열 literal은 cout에 직접 사용되어 문자열을 초기화하고 문자 배열을 초기화하는 데 사용되었다.
  - const char * foo = "hello";
- "hello"에 대한 literal 표현으로 배열을 선언하고 첫 번째 요소에 대한 포인터가 foo에 할당된다. " hello " 가 주소 1702에서 시작하는 메모리 위치에 저장되어 있다고 상상하면 이전 선언을 다음과 같이 나타낼 수 있다.



C++ ARRAYS AND POINTER

C++ Pointers and string literals

- 여기에서 foo는 포인터이고 주소 값 1702를 포함하고 ' h ' 나 " hello " 가 아니라 실제로 1702가 이 두 가지의 주소이다.
- 포인터 foo는 일련의 문자를 가리킨다. 그리고 포인터와 배열은 표현식에서 본질적으로 같은 방식으로 동작하기 때문에 foo는 null로 끝나는 문자 sequence의 배열과 같은 방식으로 문자에 액세스하는 데 사용할 수 있다.

- *(foo+4)
- foo[4]
- 두 표현식 모두 'o'(배열의 다섯 번째 요소) 값을 갖는다.

C++ ARRAYS AND POINTER

C++ Pointers to pointers(이중 포인터)

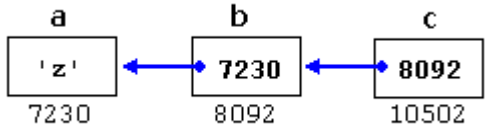
- C++에서는 포인터를 가리키는 포인터를 사용할 수 있다.
- 이 포인터는 차례로 데이터(또는 다른 포인터도 가리킴)를 가리킨다.
- 구문에는 포인터 선언의 각 간접 참조 수준에 대해 별표(*)가 필요하다.

- char a;
- char * b;
- char ** c;
- a = 'z';
- b = &a;
- c = &b;

C++ ARRAYS AND POINTER

C++ Pointers to pointers(이중 포인터)

- 선택된 메모리 위치를 7230, 8092 및 10502의 각 변수에 대해 임의로 가정하면 다음과 같이 나타낼 수 있다.
- 이 예에서 새로운 점은 포인터에 대한 포인터인 변수 c이며 세 가지의 다른 간접 참조 수준에서 사용할 수 있으며 각각은 다른 주소 값이다.



C++ ARRAYS AND POINTER

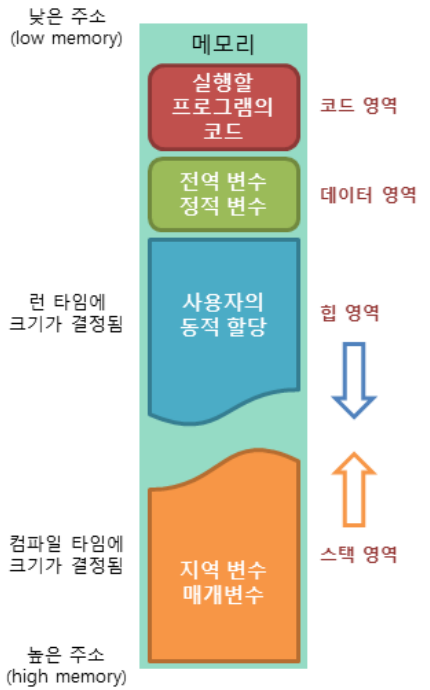
C++ void pointer

- void 유형의 포인터는 특별한 유형의 포인터입니다. C++에서 void는 type(유형)이 없음을 나타낸다. 따라서 void 포인터는 유형이 없는 값을 가리키는 포인터이다.
- 이것은 정수 값이나 부동 소수점에서 문자열에 이르기까지 모든 데이터 유형을 가리킬 수 있으므로 void 포인터에 큰 유연성을 제공한다.
- 그 대가로 그들은 큰 한계를 가지고 있다.
  - 그들이 가리키는 데이터는 직접 역 참조될 수 없으며(역 참조할 유형이 없기 때문에 논리적임), 이러한 이유로 void 포인터의 모든 주소는 구체적인 포인터 유형을 가리키는 다른 포인터 유형으로 변환되어야 한다.

C++ ARRAYS AND POINTER

Struct of Memory(메모리의 구조)

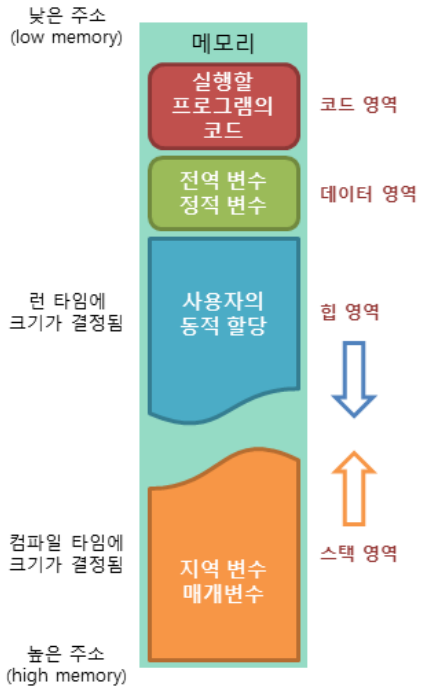
- 프로그램이 실행되기 위해서는 먼저 프로그램이 메모리에 로드(load)되어야 한다.
- 또한, 프로그램에서 사용되는 변수들을 저장할 메모리도 필요하다.
- 따라서 컴퓨터의 운영체제(Operating System)는 프로그램의 실행을 위해 다양한 메모리 공간을 제공하고 있다.
- 프로그램이 운영체제로부터 할당 받는 대표적인 메모리 공간은 다음과 같다.
  - 코드(code) 영역
  - 데이터(data) 영역
  - 스택(stack) 영역
  - 힙(heap) 영역



C++ ARRAYS AND POINTER

Struct of Memory(메모리의 구조)

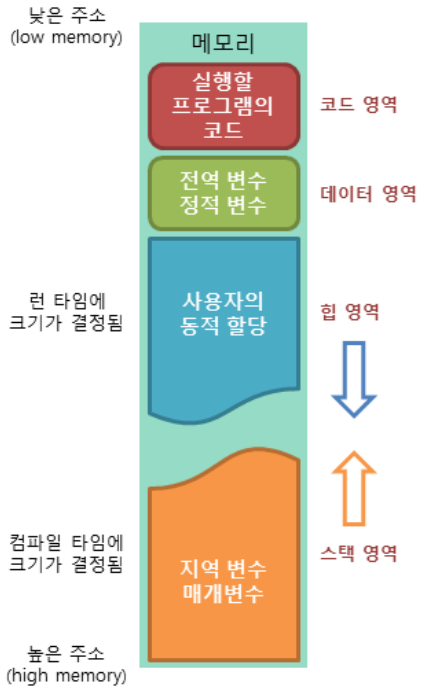
- Code 영역
  - 메모리의 코드(code) 영역은 실행할 프로그램의 코드가 저장되는 영역으로 텍스트(code) 영역이라고도 부른다.
  - CPU는 코드 영역에 저장된 명령어를 하나씩 가져가서 처리하게 된다.
- Data 영역
  - 메모리의 데이터(data) 영역은 프로그램의 전역 변수와 정적(static) 변수가 저장되는 영역이다.
  - 데이터 영역은 프로그램의 시작과 함께 할당되며, 프로그램이 종료되면 소멸한다.



C++ ARRAYS AND POINTER

Struct of Memory(메모리의 구조)

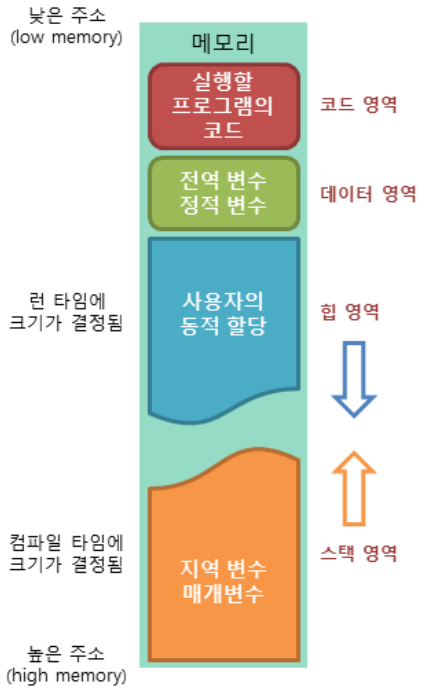
- Stack 영역
  - 메모리의 스택(stack) 영역은 함수의 호출과 관계되는 지역 변수와 매개변수가 저장되는 영역이다.
  - 스택 영역은 함수의 호출과 함께 할당되며, 함수의 호출이 완료되면 소멸한다.
  - 이렇게 스택 영역에 저장되는 함수의 호출 정보를 스택 프레임(stack frame)이라고 한다.
  - 스택 영역은 푸시(push) 동작으로 데이터를 저장하고, 팝(pop) 동작으로 데이터를 인출한다.
  - 이러한 스택은 후입선출(LIFO, Last-In First-Out) 방식에 따라 동작하므로, 가장 늦게 저장된 데이터가 가장 먼저 인출된다.
  - 스택 영역은 메모리의 높은 주소에서 낮은 주소의 방향으로 할당된다.



C++ ARRAYS AND POINTER

Struct of Memory(메모리의 구조)

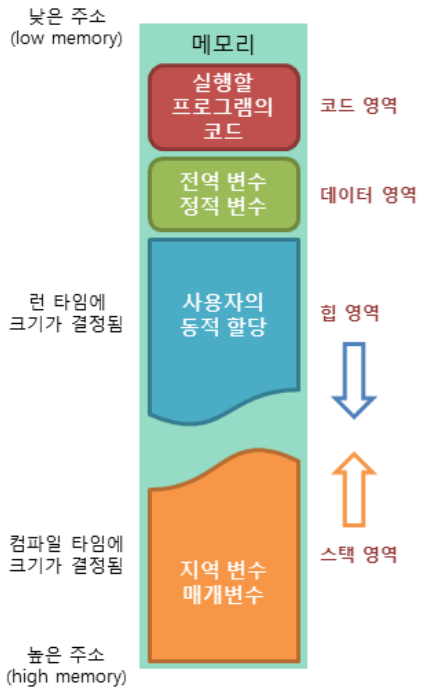
- Stack frame
  - 함수가 호출되면 스택에는 함수의 매개변수, 호출이 끝난 뒤 돌아갈 반환 주소 값, 함수에서 선언된 지역 변수 등이 저장된다.
  - 이렇게 스택 영역에 차례대로 저장되는 함수의 호출 정보를 스택 프레임(stack frame)이라고 한다.
  - 이러한 스택 프레임 덕분에 함수의 호출이 모두 끝난 뒤에, 해당 함수가 호출되기 이전 상태로 되돌아갈 수 있다.



C++ ARRAYS AND POINTER

Struct of Memory(메모리의 구조)

- Heap 영역
  - 메모리의 힙(heap) 영역은 사용자가 직접 관리할 수 있는 '그리고 해야만 하는' 메모리 영역이다.
  - 힙 영역은 사용자에 의해 메모리 공간이 동적으로 할당되고 해제된다.
  - 힙 영역은 메모리의 낮은 주소에서 높은 주소의 방향으로 할당된다.



C++ ARRAYS AND POINTER

C++ Dynamic Allocation(Memory의 동적 할당)

- 데이터 영역과 스택 영역에 할당되는 메모리의 크기는 컴파일 타임(compile time)에 미리 결정된다.
- 하지만 Heap 영역의 크기는 프로그램이 실행되는 도중인 런 타임(run time)에 사용자가 직접 결정하게 된다.
- 이렇게 런 타임에 메모리를 할당 받는 것을 메모리의 동적 할당(dynamic allocation)이라고 한다.
- 포인터의 가장 큰 목적은 런 타임에 이름 없는 메모리를 할당 받아 포인터에 할당하여, 할당 받은 메모리에 접근하는 것이다.
- C언어에서는 malloc() 함수 등의 라이브러리 함수를 제공하여 이러한 작업을 수행할 수 있게 해준다.
- C++에서도 C언어의 라이브러리 함수를 사용하여 메모리의 동적 할당 및 해제를 할 수 있다.
- 하지만 C++에서는 메모리의 동적 할당 및 해제를 위한 더욱 효과적인 방법을 new 연산자와 delete 연산자를 통해 제공한다.

C++ ARRAYS AND POINTER

C++ Dynamic Allocation(Memory의 동적 할당)

- C malloc() 함수
  - Malloc() 함수는 프로그램이 실행 중일 때 사용자가 직접 힙 영역에 메모리를 할당할 수 있게 해줍니다.
  - Malloc() 함수의 원형은 다음과 같습니다.
  - <stdlib.h> 에 포함 된 C standard library이다.
  - malloc() 함수는 인수로 할당 받고자 하는 메모리의 크기를 바이트 단위로 전달받는다.
  - 이 함수는 전달받은 메모리 크기에 맞고, 아직 할당되지 않은 적당한 블록을 찾는다.
  - 이렇게 찾은 블록의 첫 번째 바이트를 가리키는 주소 값을 반환한다.
  - 힙 영역에 할당할 수 있는 적당한 블록이 없을 때에는 널 포인터를 반환한다.
  - 주소 값을 반환 받기 때문에 힙 영역에 할당된 메모리 공간으로 접근하려면 포인터를 사용해야 한다.

```
#include <stdlib.h>
void *malloc(size_t size);
```

C++ ARRAYS AND POINTER

C++ Dynamic Allocation(Memory의 동적 할당)

- C free() 함수
  - free() 함수는 힙 영역에 할당 받은 메모리 공간을 다시 운영 체제로 반환해 주는 함수이다.
  - 데이터 영역이나 스택 영역에 할당되는 메모리의 크기는 컴파일 타임에 결정되어, 프로그램이 실행되는 내내 고정된다.
  - 하지만 메모리의 동적 할당으로 힙 영역에 생성되는 메모리의 크기는 런 타임 내내 변화된다.
  - 따라서 free() 함수를 사용하여 다 사용한 메모리를 해제해 주지 않으면, 메모리가 부족해지는 현상이 발생할 수 있다.
  - 이처럼 사용이 끝난 메모리를 해제하지 않아서 메모리가 부족해지는 현상을 메모리 누수(memory leak)라고 한다.
  - free() 함수의 원형은 다음과 같습니다.
  - free() 함수는 인수로 해제하고자 하는 메모리 공간을 가리키는 포인터를 전달받는다.
  - 인수의 타입이 void형 포인터로 선언되어 있으므로, 어떠한 타입의 포인터라도 인수로 전달될 수 있다.

```
#include <stdlib.h>
void free(void *ptr);
```

C++ ARRAYS AND POINTER

C++ Dynamic Allocation(Memory의 동적 할당)

- calloc() 함수
  - calloc() 함수는 malloc() 함수와 마찬가지로 힙 영역에 메모리를 동적으로 할당해주는 함수이다.
  - 이 함수가 malloc() 함수와 다른 점은 할당하고자 하는 메모리의 크기를 두 개의 인수로 나누어 전달받는 점이다.
  - 또한, calloc() 함수는 메모리를 할당 받은 후에 해당 메모리의 모든 비트 값을 전부 0으로 초기화해 준다.
  - Calloc() 함수도 malloc() 함수와 마찬가지로 free() 함수를 통해 할당 받은 메모리를 해제해 주어야 한다.

```
#include <stdlib.h>
void *calloc(size_t nmemb, size_t size);
```

C++ ARRAYS AND POINTER

C++ Dynamic Allocation(Memory의 동적 할당)

- realloc() 함수
  - realloc() 함수는 이미 할당된 메모리의 크기를 바꾸어 재할당할 때 사용하는 함수이다.
  - realloc() 함수의 첫 번째 인수는 크기를 바꾸고자 하는 메모리 공간을 가리키는 포인터를 전달받는다.
  - 두 번째 인수로는 해당 메모리 공간에 재할당할 크기를 전달한다.
  - 따라서 첫 번째 인수로 NULL이 전달되면, malloc() 함수와 정확히 같은 동작을 하게 된다.

```
#include <stdlib.h>
void *realloc(void *ptr, size_t size);
```

C++ ARRAYS AND POINTER

C++ new and delete operator

- new operator
  - C++ 모든 데이터 유형에 대해 동적으로 메모리를 할당하기 위해 new 연산자를 사용하는 다음 일반 구문이 있다.
  - Type은 데이터에 맞는 포인터를 선언하기 위해, 두 번째 타입은 메모리의 종류를 지정하기 위해 사용된다.
  - 만약 사용할 수 있는 메모리가 부족하여 새로운 메모리를 만들지 못했다면, new 연산자는 널 포인터를 반환한다.
  - 또한, new 연산자를 통해 할당 받은 메모리는 따로 이름이 없으므로 해당 포인터로만 접근할 수 있게 된다.

- type* ptrName = new type;

C++ ARRAYS AND POINTER

C++ new and delete operator

- delete operator
  - C언어에서는 free() 함수를 이용하여 동적으로 할당 받은 메모리를 다시 운영체제로 반환한다.
  - 이와 마찬가지로 C++에서는 delete 연산자를 사용하여, 더는 사용하지 않는 메모리를 다시 메모리 공간에 돌려줄 수 있다.
  - C++에서 delete 연산자는 다음과 같은 문법으로 사용합니다.

- delete ptrName;