포인터

- 이미 존재하는 다른 변수를 가리키는(point)는 역할을 하는 변수
- 다른 변수에 대한 주소를 값으로서 가짐

항목	예	설명
포인터 변수의 정의	int* pIntVal ;	int 타입 변수를 가리킬 수 있는 포인터 변수 pIntVal의 정의
포인터 변수의 값 대입	int intVal = 100 ; pIntVal = &intVal ;	pIntVal 변수는 intVal 변수를 가리킴. 즉 intVal 변수의 주소를 저장함
원 변수 값의 조회	cout << *pIntVal ;	*pIntVal은 저장된 주소가 가리키는 공간에서 int 값을 구함. 즉 *pIntVal은 intVal과 동일한 값 100이 된다.

포인터 변수의 정의

- T 타입에 대한 포인터: T*

```
int* pIntVal ;                  // int 타입 변수에 대한 포인터
float* pFloatVal ;              // float 타입 변수에 대한 포인터
```

- 포인터 변수 값의 대입

```
int intVal ;
int* pIntVal1 = &intVal ;           // 다른 변수의 주소 조회 후 대입
int* pIntVal2 = pIntVal1 ;          // 다른 포인터 변수의 값 대입
int* pIntArray = new int[100] ;     // new로 할당된 메모리 주소 대입
```

포인터 변수의 값 대입

- 포인터 변수는 주소를 저장함

```
int intVal = 10 ;
int* pIntVal = &intVal ;    // intVal 변수의 주소가 pIntVal에 저장됨
```

	주소	값
	...	...
	...	...
intVal	**0x1000**	10
	...	...
	...	...
pIntVal	...	**0x1000**

원 변수 값의 조회 및 변경

- ***포인터변수: 원 변수 값의 조회 및 변경

```
int intVal = 10 ;
int* pIntVal = &intVal ;   // pIntVal 은 intVal을 가리킴
cout << *pIntVal ;          // 10
intVal ++ ;
cout << *pIntVal ;         // 11
(*pIntVal) ++ ;              // pIntVal이 가리키는 즉 intVal의 값 1증가
cout << intVal ;           // 12
```

포인터 변수의 정의와 사용 예

```
#include <iostream>
using namespace std ;
int main() {
  int intVal = 100 ;
  int* pIntVal = &intVal ;   // 포인터 변수 pIntVal의 정의와 intVal 주소 대입
  cout << intVal << ' ' << *pIntVal << endl ; // 100 100
  intVal += 100 ;
  cout << intVal << ' ' << *pIntVal << endl ;  // 200 200
  *pIntVal = *pIntVal + 10 ;  // *pIntVal을 변경하므로 intVal도 변경됨
  cout << intVal << ' ' << *pIntVal << endl ; // 210 210
}
```

동적 할당: C 언어

```
#include <stdio.h>
#include <stdlib.h>
int main () {
   int length ;
   char * name ;
   printf ("How long do you want the string? ");
   scanf ("%d", &length);
   name = (char*) malloc (length+1);     // malloc() 함수를 이용한 메모리 할당
   scanf( "%s", name) ;
   printf("%s₩n", name) ;
   free (name);                          // free() 함수를 이용한 메모리 해제
}
```

C++ 동적 할당: new와 delete

```
#include <iostream>
using namespace std ;
int main() {
   cout << "Enter the length of a name" << endl ;
   int length ;
   cin >> length ;
   char* name = new char[length+1] ;    // new를 이용한 메모리 할당
   cin >> name ;
   cout << name << endl ;
   delete [] name ;                     // delete를 이용한 메모리 해제
}
```

메모리 할당: new

- T* = new T ;
- T* = new T[size] ;

예	설명
char* name = **new char[10]** ;	char 타입 10개 변수의 메모리 할당
int size ; cin >> size ; string* names = **new string[size]** ;	개수를 실행시에 결정할 수도 있음; 즉 size는 프로그램이 실행될 때 사용자가 입력한 값에 의해서 결정됨
char* theChar = **new char** ; string* theName = **new string** ;	한 개의 변수도 할당할 수가 있음

메모리 해제: delete

- delete p ;
- delete [] p ;

메모리 할당	메모리 해제
char* name = new char[10] ;	delete [] name ;
int size ; cin >> size ; string* names = new string[size] ;	delete [] names ;
char* theChar = new char ; string* theName = new string ;	delete theChar ; delete theName ;

C 언어와 C++ 언어의 비교

	C 언어	C++ 언어
할당	int* pIntArray = (int*) malloc (size);	int* pIntArray = new int[size] ; int* pInt = new int ;
해제	free(pIntArray) ;	delete [] pIntArray ; delete pInt ;

Good Design: 포인터 관련 오류 최소화[1]

- 표준 라이브러리(std::vector, std::string 등)나 검증된 라이브러리를 사용
- 클래스를 정의하고 동적 메모리 할당 및 해제를 관리함(RAII 원칙)
- 스마트 포인터를 사용
  - (일반) 포인터는 주로 두 가지 용도로 사용함
    - 개체를 참조함 (문제: 해제된 메모리 참조, double free)
    - 동적 메모리를 관리함 (문제: 메모리 누수 발생)

  → **(일반)포인터는 누군가가 개체를 참조하고 있는지, 더 이상 필요하지 않아 메모리를 해제해야 하는지 알 수 없음**

[1] Peter Gottschling, Discovering Modern C++, Addison-Wesley Professional, 2015

Smart Pointer

- Smart pointers are used to make sure that an object is deleted if it is no longer used (referenced)

- 스마트 포인터의 유형
  - unique_ptr: 참조한 데이터의 고유 소유권(unique ownership)을 나타내며 포인터 사용이 만료되면 메모리가 자동으로 해제됨
  - shared_ptr: 여러 곳에서 공통으로 사용하는 메모리를 관리(reference count 등)하며, 더 이상 데이터를 참조하지 않는 즉시 메모리를 해제

std::unique_ptr<>

```
#include <memory>
int main() {
    std::unique_ptr<int> pui {new int(3)};
    auto pui2 = std::make_unique<int>(3);
    int* pint = pui2.get();                                  //get the stored pointer
    auto pustr = std::make_unique<std::string>("good bye");
    pustr.reset(new std::string());
    std::cout << *pustr << std::endl;
   std::unique_ptr<int[]> puarr {new int[3]};
    for (int i=0; i < 3; ++i)
        puarr[i] = i + 2;
    int b;
    std::unique_ptr<int> pub{&b};          //동적으로 할당하지 않음!
    //std::unique_ptr<int> pui2{pui};      //컴파일 에러 (복사 생성자가 삭제되어 있음)
    //pui2 = pui;                          //컴파일 에러 (할당금지)
}
```

std::unique_ptr<> (move semantics)

```
#include <memory>
std::unique_ptr<int> generate_pu_int(int arg) {
   return std::unique_ptr<int> {new int(arg)};
}
int main(){
   std::unique_ptr<int>pui {new int(3)};
   std::unique_ptr<int> pui2{std::move(pui)};  //pui is dangling pointer
   auto pui3 = std::move(pui2);                //pui2 is dangling pointer

   std::unique_ptr<int> pui4;
   pui4 = generate_pu_int(3);                  //move semantics
}
```

std::shared_ptr<>

```
#include <iostream>
#include <memory>
int main() {
    std::shared_ptr<int> psi {new int(3)};        //내부적으로 관리 데이터(ref. count)를 생성
    std::shared_ptr<int> psi2{psi};  //가능함 (관리 데이터는 공유되고 있음)
    std::cout << psi.use_count() << " " << psi.use_count() <<std::endl;  // 2 2

    auto psi3 = std::make_shared<int>(4);//효율적으로  shared_ptr 생성함 (권장)
    std::shared_ptr<int> psi4{psi3};
    std::cout << psi4.use_count() << " " << psi4.use_count() << std::endl;  // 2 2

    //int* pint = new int(4);
    //std::shared_ptr<int> psi5(pint)            // please use std::make_shared or new int{4}
    //std::shared_ptr<int> psi6(pint);
    //std::cout << psi5.use_count() << " " << psi6.use_count() <<std::endl;  // 1 1
}
```

상수에 대한 포인터

- 포인터가 가리키는 값을 변경하지 못하는 포인터

	일반적인 포인터	상수에 대한 포인터
정의	int* pIntVal = &intVal ;	**const** int* pcIntVal = &intVal ;
원 변수 값 조회	cout << *pIntVal ;	cout << *pcIntVal ;
원 변수 값 변경	**(*pIntVal) ++ ; // 허용**	**(*pcIntVal) ++ ; // 불허**

- 상수에 대한 포인터의 활용 예

```
size_t strlen ( const char * str );
char* strcat ( char* destination, const char* source) ;
const char * strstr ( const char * str1, const char * str2 );
```

Good Design: 상수에 대한 포인터의 활용

- 포인터가 가리키는 변수의 값이 변경되지 않아야 한다면 상수에 대한 포인터를 항상 사용하는 것이 권장

```
int countUppercase(const char* str, const int length) {
   int count = 0 ;
   for ( int i = 0 ; i < length ; i ++ )
      if ( str[i] >= 'A' && str[i] <= 'Z' ) count ++ ;
   return count ;
}
int main() {
   char* msg = "Hello World" ;
   cout << countUppercase(msg, strlen(msg)) ; // 2
}
```

상수 포인터

- 일반 포인터는 다른 주소를 대입함으로써 새로운 변수를 가리키는 것이 가능함

```
int intVal1 = 100, intVal2 = 200 ;
int* pIntVal = &intVal1 ;
pIntVal = &intVal2 ; // pIntVal이 다른 변수 intVal2를 가리킨다.
```

- **상수 포인터**는 초기화된 후에 다른 주소를 대입하는 것이 불가능함

```
int intVal1 = 100, intVal2 = 200 ;
int* const pIntVal = &intVal1 ;
pIntVal = &intVal2 ; // ERROR
```

상수 포인터

- 일반 포인터: 다른 주소를 대입하여 다른 변수를 가리키는 것이 가능
- 상수 포인터: 초기화 한 후에 다른 주소를 대입시키는 것을 불가능

	일반 포인터	상수 포인터
정의	int* pIntVal = &intVal1 ;	int* **const** cpIntVal = &intVal1 ;
원 변수 값 조회	cout << *pIntVal ;	cout << *cpIntVal ;
원 변수 값 변경	(*pIntVal) ++ ;	(*cpIntVal)++ ;
초기화 필수 여부	// 초기화 필수 아님 int* pIntVal1 ; // 허용	**// 초기화 필수** int* const cpIntVal1 ; // 불허
다른 주소의 대입	pIntVal = &intVal2 ; // 허용	cpIntVal = &intVal2 ; **// 불허**

포인터와 상수 요약

	일반 포인터	상수에 대한 포인터	상수 포인터	상수에 대한 상수 포인터
정의 방법	int* pIntVal	**const** int* pIntVal	int* **const** pIntVal=&intVal1	**const** int* **const** pIntVal=&intVal1
초기화 필수 여부	필수 아님	필수 아님	필수	필수
원 변수 값 조회 cout << *pIntVal ;	OK	OK	OK	OK
원 변수 값 변경 *pIntVal ++ ;	OK	ERROR	OK	ERROR
다른 주소의 대입 pIntVal = &intVal2	OK	OK	ERROR	ERROR

```
int main() {
  int intVal1 = 10, intVal2 = 20 ;

  // 일반 포인터
  int* pIntVal ;
  pIntVal = &intVal1 ;                  // ok
  *pIntVal += 10 ;                      // ok

  // 상수에 대한 포인터
  const int* pcIntVal = &intVal1 ;
  *pcIntVal += 10 ;                     // error; pcIntVal은 상수에 대한 포인터이므로
  pcIntVal = &intVal2 ;                 // ok

  // 상수 포인터
  int *const cpIntVal = &intVal1 ;
  *cpIntVal += 10 ;                     // ok
  cpIntVal = &intVal2 ;                 // error; cpIntVal은 상수 포인터이므로

  // 상수에 대한 상수 포인터
  const int *const ccIntVal = &intVal1 ;
  *ccIntVal += 10 ;                     // error; ccIntVal은 상수에 대한 포인터이므로
  ccIntVal = &intVal2 ;                 // error; ccIntVal은 상수 포인터이므로
}
```

참조(reference) 변수

- 다른 변수를 가리키는 역할
- T&: T 타입 변수에 대한 참조

```
#include <iostream>
using namespace std ;
int main() {
   // T& x: 변수 x는 타입 T 변수에 대한 참조이다.
   int intVal = 10 ;
   int& rIntVal = intVal ;
   cout << intVal << '₩t' << rIntVal << endl ; // 10 10
   // 원래 변수의 값이 변경되면 참조 변수의 값도 변경됨
   intVal = 20 ;
   cout << intVal << '₩t' << rIntVal << endl ; // 20 20
   // 참조 변수의 값이 변경되면 원래 변수의 값도 변경됨
   rIntVal = 30 ;
   cout << intVal << '₩t' << rIntVal << endl ; // 30 30
}
```

참조 변수의 초기화

- 참조 변수는 기존의 변수를 가리키도록 초기화되며 이후에는 다른 변수를 가리킬 수가 없다

코드	설명
float floatVal = 10.1F ; float& rFloatVal1 = floatVal ;	Ok: 참조 변수가 정의와 동시에 일반 변수로 초기화 함
float floatVal1 = 10.1F ; float floatVal2 = 20.2F ; float& rFloatVal2 = floatVal1 ; rFloatVal2 = floatVal2 ;	OK: 참조 변수가 정의와 동시에 일반 변수로 초기화 함 Error: 초기화 후에 다른 변수를 가리킬 수 없음
float& rFloatVal3 = 10.0F ;	Error: 참조 변수는 리터럴 값으로 초기화 될 수 없음.
const float constVal = 20.0 ; float& rFloatVal4 = constVal	Error: 참조 변수는 상수 변수를 가리킬 수가 없음

참조 변수와 포인터 변수

	참조 변수	포인터 변수
정의 방법	int intVal=10, otherVal=20 ; int& rIntVal = intVal ;	int intVal=10, otherVal=20 ; int* pIntVal = &intVal ;
초기화 필수 여부	필수 int& rIntVal ; 가 불허됨	필수 아님 int* pIntVal ; 가 허용됨
다른 변수의 지칭	불가능	가능 pIntVal = &otherVal ; 가 허용됨
연산의 적용 대상	참조 변수가 가리키는 원 변수 rIntInt ++ ; intVal ++ 와 동일	pIntVal ++ ; 다음 주소를 가리킴 원래 변수 값을 접근하려면 (*pIntVal) ++ ; 로 함

Good Design: 포인터 대신 참조 변수가 권장

- 포인터는 +, ++, -- 등의 연산자를 이용해서 임의의 메모리를 가리킬 경우 문제를 유발
- 참조 변수는 기존 변수로 일단 초기화된 후에는 다른 변수를 임의로 가리키도록 변경될 수가 없음

```
#include <iostream>
using namespace std ;
int main() {
    int intVal = 10 ;
    int* pIntVal = &intVal ;
    cout << *pIntVal << endl ;
    pIntVal ++ ;
    cout << *pIntVal << endl ;
}
```

```
// 10
// intVal 다음의 메모리를 가리킴
// 확인되지 않은 값이 출력됨
```

Good Design: 참조 변수의 사용

- 상수 포인터 대신에 참조를 사용하는 것이 권장

```
int intVal = 100 ;
int * const cpIntVal = &intVal ;      // C/C++ 언어에서의 상수 포인터
int& rIntVAl = intVal;                // C++ 언어에서의 참조 변수
```

nullptr (Since C++11)

- nullptr denotes NULL Pointer. Use nullptr instead of 0

```
#include <iostream>
using namespace std ;
int main() {
   cout << "Enter the length of a name" << endl ;
   int length ;
   cin >> length ;
   char* name = new char[length+1] ;
   if ( name != nullptr ) {
      cin >> name ;
      cout << name << endl ;
      delete [] name ;
   }
}
```

auto (Since C++11)

- For variables, specifies that the type of the variable that is being declared will be automatically deduced from its initializer
- Type is deduced using the rules for template argument deduction(TAD)[1].

```
#include <iostream>
#include <string>
int main() {
    auto x = 4;                 // type of a is int
    auto y = 3.37;
    auto pz = &x;
    std::cout  << typeid(x).name() << std::endl          //i
               << typeid(y).name() << std::endl          //d
               << typeid(pz).name() << std::endl;        //Pi

    std::string str ="hello";
    for(auto it=str.begin(); it!=str.end(); ++it) {
        std::cout << typeid(it).name() << std::endl;
        std::cout << *it <<std::endl;
    }
}
```

[1] https://en.cppreference.com/w/cpp/language/template_argument_deduction

(추가) AAA (Almost Always Auto)

- 타입에 신경쓰기 않으면, auto x = *initializer*, 를 선호하라
- 타입을 명시하고 싶으면, auto x = type { *expression* }

```
int i = 42;
long v = 42;
Customer c{"Jim", 77};
std::vector<int>::const_iterator p = v.begin();
```

```
auto i = 42;
auto v = 42l;
auto c = Customer{"Jim", 77};
auto p = v.cbegin();
```

- 사소한 문제?

```
int i = {42};


std::string x = "42";


std::array<int, 5> r{};

long long ll{getInt()};
```

```
auto i = {42};          //initializer_list<int>
auto i = int {42};//OK!

using namespace std::literals;
auto x = "42"s; //C++14

auto r = std::array(5); //C++17

//auto ll = long long{getInt()}; //ERROR
auto ll = static_cast<long long>(getInt());
```

auto for Return Type(C++11, C++14)

- Function declaration uses the trailing return type syntax, the keyword auto does not perform automatic type detection(C++11)
- No need of trailing return type; return type will be deduced from the operand of its return statement(C++14)

```
#include <iostream>
#include <vector>
//error: 'generate' function uses 'auto' type specifier without trailing return type
//auto generate() { return 10; }                          //C++11 → error
auto generate() -> int {return 10;}                       //trailing return type
auto add(int op1, int op2) { return op1+op2; }            //C++14 → OK
//std::vector::operator[] → Returns a reference to the element at position n in the vector container.
auto get(std::vector<int> vec, int idx) { return vec[idx]; } //return int& type

int main() {
    auto a = generate();                                  // type of a is int
    auto b = add(1, 2);                                   // type of b is int
    std::vector<int> vec = {1, 2, 3};
    auto c = get(vec, 1);                                 // type of c is int or int& ?
}   //If P is a reference type, the type referred to by P is used for deduction from TAD.
```

https://en.cppreference.com/w/cpp/language/auto

decltype (Since C++11)

- Inspects the declared type of an entity or the type and value category of an expression

id-expression 예) 변수명, 클래스 멤버변수, std::endl 등

- If the argument is an unparenthesized id-expression or an unparenthesized class member access expression, then decltype yields the type of the entity named by this expression.

- If the argument is any other expression of type T, and
  - a) if the value category of expression is xvalue, then decltype yields T&&;
  - b) if the value category of expression is lvalue, then decltype yields T&;
  - c) if the value category of expression is prvalue, then decltype yields T.

- Note that if the name of an object is parenthesized, it is treated as an ordinary lvalue expression, thus decltype(x) and decltype((x)) are often different types.

https://en.cppreference.com/w/cpp/language/decltype

decltype (Since C++11)

```
#include <iostram>

decltype(auto) get(std::vector<int> vec, int idx) { return vec[idx]; }

int main() {
    int a = 3;
    const int b = 10;

    auto b1 = b;              //type of b1 is int
    decltype(b) b2 = b;       //type of b2 is const int

    decltype(a) c1 = a;   // type of c1 is int, holding a copy of a
    decltype((a)) c2 = a; // type of c2 is int&, "(a)" is lvalue

    ++c1;
    std::cout << "a=" << a << << ", c1=" << c1 <<std::endl; // a=3, c1=4
    ++c2;
    std::cout << "a=" << a << << ", c2=" << c2 <<std::endl; // a=4, c2=4
}
```

https://en.cppreference.com/w/cpp/language/decltype

Q & A