**Bitwise operators**

in C++ 6types of bitwise operators

텍스트, 폰트, 번호, 스크린샷이(가) 표시된 사진

자동 생성된 설명

1. **Bitwise left shift(<<)**

3 = 0011

3 << 1 = 0110 = 6

3 << 2 = 1100 = 12

3 << 3 = 1000 = 8

이진수의 끝에서 벗어난 비트는 손실된다. 여기서는 4bit로 가정하고 있으므로, 8bit일 경우

3 << 3은 11000이 되어 24가 된다. 이 경우 4번째 비트는 5번째 비트로 이동되므로 손실되지 않는다.

1. **Bitwise right shift(>>)**

12 = 1100

12 >> 1 = 0110 = 6

12 >> 2 = 0011 = 3

12 >> 3 = 0001 = 1

이진수의 끝에서 벗어난 비트는 손실된다.

**\*변수도 시프트 가능**

unsigned int x =4;

x = x << 1;// x=8

1. **Bitwise NOT(~)**

각 비트에서 0과 1를 바꾼다. 자료형의 크기에 따라 결과값이 다르다.

**4bits**

4 = 0100

~4 = 1011 = 11

**8bits**

4= 0000 0100

~4 = 1111 1011 = 251

1. **Bitwise AND(&)**

두 비트 모두 1인 경우에만 1로 평가된다.

5 = 0101

6 = 0110

5&6 = 0100 = 4

1. **Bitwise OR(|)**

두 비트 중 하나 이상이 1이면 1로 평가된다.

5 = 0101

6 = 0110

5|6 = 0111 = 7

1. **Bitwise XOR(^)**

두 비트 중 하나만 1인 경우에만 1로 평가된다.

5 = 0101

6 = 0110

5^6 = 0011 = 3

1. **Bitwise assignment operators**

**텍스트, 폰트, 번호, 라인이(가) 표시된 사진

자동 생성된 설명**

ex) x= x<< 1 🡪 x <<= 1

**String 라이브러리의 주요기능 정리**

1. size() : 해당 문자열의 사이즈 반환

\*length()함수와 동일한 기능

2. capacity() : 문자열 객체에 할당된 메모리 크기 반환

3. resize() : 문자열 객체를 n만큼의 크기로 만듦. 원래 크기가 n보다 크면 남은 부분은 버림. n보다 작으면 남은 공간을 빈공간으로 채움.

4. clear() : string 객체에 있는 문자열을 지우는 함수

\*size, length는 모두 0이 되고 capacity는 그대로 유지됨.

5. empty() : 문자열 객체가 비어있는지 확인하는 함수, bool타입으로 반환

6. at() : 해당 인덱스 문자 반환

7. front(), back() : 문자열의 가장 앞과 뒤를 반환

8. substr(index, len) : 문자열의 index부터 len만큼 잘라서 반환

9. replace(s, len, str) : 문자열(s)의 인덱스 위치에서 len까지 범위를 str로 대체

10. compare() : 매개변수로 들어온 문자열과 비교해서 같으면 0반환, 다르면 -1반환

11. copy(char\*arr, len, index=0) : 문자열을 arr에 index부터 len만큼 복사

12. find(string str, index=0) : 매개변수로 들어온 문자열과 비교하려는 문자열 중에 일치하는게 있는지 확인, 일치하는게 있다면 첫번째 발견 인덱스 반환, index값부터 있는지, 확인, 없다면 쓰레기 값 반환

13. begin() : 문자열의 첫번째 문자를 가리키는 반복자(iterator) 반환

14. end() : 문자의 마지막의 바로 문자를 가리키는 반복자(iterator) 반환

15. swap() : s1문자열과 s2문자열을 바꾸는 것

16. append() : 문자열 끝에 더함

17. assign() : 문자열을 할당하는 함수

18. insert(index, string s) : 문자열을 지정한 위치에 삽입하는 함수

19. erase(index, num) : 문자열을 지우는 함수(인덱스부터 num개의 문자를 지움)

20. to\_string(num) : 숫자형을 string타입으로 변환(char형 숫자이면 아스키 코드로 변환)

21. stoi(str) : 문자열을 숫자로 반환

22. int to char : 숫자를 아스키 코드를 통해 문자로 반환

23. toupper(char), tolower(char) : 해당 문자를 대문자, 소문자로 변환

**STL(표준 템플릿 라이브러리) 사용법**

: STL(Standard Template Library), 표준 템플릿 라이브러리. 벡터, 리스트, 스택, 큐 등의 컨테이너와 이들을 처리하기 위해 사용할 수 있는 여러가지 연산을 포함함.

* 컨테이너(Container) : 데이터 저장
* 반복자(iterartor) : 포인터의 역할
* 알고리즘(algorithm) : 데이터 처리 기능

**1. 컨테이너(Container)**

: 같은 타입의 객체를 저장하는 일종의 집합, 클래스 템플릿으로 컨테이너 변수를 선언할 때 컨테이너에 포함할 요소의 타입을 명시할 수 있음.

* int나 float와 같은 기본자료형 데이터나 사용자 정의 클래스의 객체 등을 저장함.
* 다양한 연산이 제공되어 편리하게 데이터를 활용 가능함.
* 순차 컨테이너 : 동일한 자료형의 객체들을 선형적인 구조로 저장(vetor, list, deque)
* 연상 컨테이너 : 탐색 트리와 같은 인덱스 구조를 이용하는 컨테이너로 키(key)를 이용한 검색 기능을 제공(set, multiset, map, multimap)
* 무순서 연상 컨테이너 : 연상 컨테이너와 같으나 해시함수를 이용하여 검색(unordered-set, unordered-multiset, unordered-map, unordered—multimap)
* 컨테이너 어댑터 : 기본 컨테이너를 기반으로 특정 용도에 맞게 유도된 컨테이너(queue, priority-queue, stack)

텍스트, 스크린샷, 폰트, 번호이(가) 표시된 사진

자동 생성된 설명

**2. 반복자(iterator)**

: 저장된 요소를 반복적으로 순회하여 각각의 요소에 대한 접근을 제공하는 객체, 컨테이너의 구조나 요소의 타입과는 상관없이 컨테이너에 저장된 데이터를 순회하는 과정을 일반화한 표현

* 컨테이너의 유형에 따라 서로 다른 형태의 반복자가 사용됨.
* 가리키는 요소의 값에 접근할 수 있어야함. 따라서, 참조 연산자(\*)가 정의되어야 함.
* 반복자 사이의 대입연산, 비교 연산이 가능해야함. 따라서 대입, 관계 연산자가 정의되어야 함.
* 가리키는 요소의 주변 요소로 이동할 수 있어야 함. 따라서, 증가 연산자(++)가 정의되어야 함.

텍스트, 스크린샷, 폰트, 번호이(가) 표시된 사진

자동 생성된 설명

**3. 알고리즘(Algorithm)**

: 컨테이너의 원소에 대해 적용할 수 있는 여러가지 연산

**텍스트, 스크린샷, 폰트, 번호이(가) 표시된 사진

자동 생성된 설명**

**Vector**

: 1차원 배열의 개념을 구현한 순차 컨테이너 유형의 클래스 템플릿, 배열의 일반적인 기능을 포함하면서 여러가지 유용한 멤버함수 및 관리기능이 도입되어 있음. 배열처럼 크기가 고정되어 있지 않고, 필요에 따라 저장 공간을 확장할 수 있다는 특징이 있음.

* 벡터 객체는 요소가 추가되거나 삭제될 때마다 자동으로 메모리를 재할당하여 크기를 동적으로 변경함.
* 해당 컨테이너는 vector 헤더 파일에 저장되어 있으며, 임의 접근을 제공하는 가장 간단한 시퀀스 컨테이너임.

# Vector 객체 선언 구문

· 템플릿 인수로는 벡터 컨테이너에 저장될 요소의 타입을 전달합니다.

· 생성자 인수로는 벡터 컨테이너의 초기 크기를 전달하며, 생략하면 요소를 가지지 않는 빈 벡터를 생성합니다.

vector<템플릿인수> 객체이름(생성자인수);

# Vector 객체 선언 및 초기화

vector<int> vInt = {1, 2, 3};

# vector 멤버 함수

at(): 직접접근을 위한 멤버함수, 지정된 첨자가 범위를 벗어날 경우 예외 발생

size(): 벡터의 논리적인 사이즈 반환

capacity(): 벡터의 물리적인 사이즈 반환, 실제 할당된 메모리의 크기

push\_back(): 벡터의 끝에 데이터를 삽입

pop\_back(): 벡터의 끝에 데이터를 꺼냄

insert(it, value): 지정된 위치에 데이터를 삽입 (반복자 it가 가리키는 위치에 value 삽입)

erase(it): 지정된 위치에 데이터를 삭제 (반복자 it가 가리키는 위치 삭제)

erase(it1, it2): [it1, it2] 범위의 데이터 삭제

resize(n): 논리적 크기 변경

reserve(n): capacity()가 최소한 n을 반환하도록 확장

empty(): 비어 있는 벡터의 경우 true

sort(first, last): 랜덤 액세스 반복자에 의해 지정된 범위의 값들을 정렬, first ~ last 범위의 값들을 정렬

sort(first, last, comp): comp는 정렬 순서를 정하는 함수(callback), 정렬 순서에 대한 기준을 판단할 수 있는 함수

merge(first1, last1, first2, last2, dest)

merge(first1, last1, first2, last2, dest, comp)

- 동일한 기준으로 정렬된 두 개의 데이터 집합을 동일한 기준으로 정렬된 하나의 데이터 집합으로 결합

- first1, last1: 첫 번째 정렬된 데이터 범위

- first2, last2: 두 번째 정렬된 데이터의 범위

- dest: 합병 결과가 저장될 시작 위치

- comp: 합병 순서를 정하는 함수

# 반복자의 값을 구하는 vector의 멤버함수

begin(): 첫 번째 원소를 가리키는 랜덤 액세스 반복자를 반환

end(): 마지막 원소의 다음 위치를 가리키는 랜덤 액세스 반복자를 반환

# 반복자 선언

vector<템플릿인수>::iterator it;

// ex) vector<int>::iterator it = intVec.begin();

# 주의 사항: 첨자의 범위를 벗어날 경우 예외 발생

vector<float fVector(10);

fVector[1] = 1.0f; // 정상

fVector[12] = 12.0f; // 에러 발생

**Vector 사용 예시**

#include <iostream>

#include <iterator>

#include <vector>

#include  <cstdlib>

#include  <algorithm>

using namespace std;

// 내림차순 정렬하기 위한 콜백함수 (gt<int>로 사용)

// sort를 하기위한 순서를 결정할 때 매번 호출되기 때문에, 함수 객체 방식을 이용하는 것이 좋음

template<typename T> bool get(const T& a, const T& b) {

    return a > b;

}

// 함수객체: 함수처럼 사용될 수 있는 객체 -> () 연산자를 다중 정의

// 내림차순 정렬하기 위한 콜백함수 (함수객체) (GREATER<int>()로 사용)

// 오름차순일 경우: a가 b보다 작을 때 참을 반환 (앞에가 작은 값, 뒤에가 큰 값)

// 내림차순일 경우: a가 b보다 클 때 참을 반환 (앞에가 큰 값, 뒤에가 작은 값)

template<typename T> struct GREATER {

    bool operator()(const T& a, const T& b) const {

        return a > b;

    }

};

int main() {

    vector<int> vInt = {1, 2, 3}; // vector 객체의 선언 및 초기화

    cout << "벡터의 크기: " << vInt.size() << endl; // 벡터의 크기: 3

    vInt.push\_back(4); // 벡터의 끝에 데이터 삽입, 논리적 크기:4, 물리적 크기:4

    vInt.pop\_back();   // 벡터의 끝에 데이터를 꺼냄, 논리적 크기:3, 물리적 크기:4

    cout << "벡터의 논리적 크기 : " << vInt.size() << endl;     // 논리적 크기: 3

    cout << "벡터의 물리적 크기 : " << vInt.capacity() << endl; // 물리적 크기: 4

    // Vector의 값 출력

    cout << "저장된 데이터 : ";

    for (int i = 0; i < vInt.size(); i++)

        cout << vInt[i] << " ";

    // Vector의 값 출력(반복자 이용)

    // vector<int>::iterator it = vInt.begin();

    auto it = vInt.begin(); // 자료형 추론을 이용한 반복자 선언

    cout << endl << "저장된 데이터 : ";

    for (; it < vInt.end(); it++)

        cout << \*it << " ";

    cout << endl;

    // Vector의 3번째 데이터 출력

    it = vInt.begin(); // it 반복자가 vector의 앞부분을 가리킴

    cout << "3번째 데이터 : ";

    cout << \*(it + 2) << endl; // 포인터로 it + 2 번째(3) 값을 출력

    // 난수를 이용하여 벡터를 생성하고, 정렬하기

    // 벡터1

    srand((unsigned)time(NULL));   // 난수 발생기 초기화

    vector<int> iv1(5);

    cout << "벡터1 : ";

    for (auto& i : iv1) {

        i = rand() % 100;         // 0~99의 난수 발생

        cout << i << " ";

    }

    sort(iv1.begin(), iv1.end());   // 정렬 알고리즘

    cout << endl << "정렬된 벡터1 : ";

    for (auto i : iv1)

        cout << i << " ";

    cout << endl << endl;

    // 벡터2

    vector<int> iv2(5);

    cout << "벡터2 : ";

    for (auto &i : iv2) {

        i = rand() % 100;         // 0~99의 난수 발생

        cout << i << " ";

    }

    sort(iv2.begin(), iv2.end());   // 정렬 알고리즘

    cout << endl << "정렬된 벡터2 : ";

    for (auto i : iv2)

        cout << i << " ";

    cout << endl << endl;

    // 벡터3: 벡터1과 벡터2 합병

    // 합병 결과를 저장할 벡터

    vector<int> iv3(iv1.size() + iv2.size());

    // iv1과 iv2를 합병한 결과를 iv3에 저장

    merge(iv1.begin(), iv1.end(),

        iv2.begin(), iv2.end(), iv3.begin());

    cout << "벡터1과 벡터2를 합병한 결과 : ";

    for (auto i : iv3)

        cout << i << " ";

    cout << endl << endl;

    // 내림차순 정렬하기

    sort(iv3.begin(), iv3.end(), GREATER<int>()); // 정렬 알고리즘

    cout << endl << "내림차순 정렬된 벡터3 : ";

    for (auto i : iv3)

        cout << i << " ";

}

**MAP**

: 키와 값의 쌍으로 데이터를 관리하는 연관 컨테이너

해당 컨테이너는 집합 컨테이너와 마찬가지로 정렬된 위치에 요소를 삽입하므로 검색 속도가 매우 빠름. 키는 유일해야 하므로, 키의 중복을 허용하지 않음. 따라서, 하나의 키에 하나의 값만이 연결 될 수 있음. 하지만, 멀티맵(multimap)은 값의 중복을 허용하므로, 하나의 키가 여러 개의 값과 연결될 수 있음.

* 저장하는 데이터의 형태 : 키와 값의 쌍
* 해당 컨테이너는 map 헤더 파일에 정의되어 있으며, 키를 이용하여 데이터에 직접 접근할 수 있는 연상 컨테이너임.

# Vector 객체 선언 구문

· 템플릿 인수로는 맵 컨테이너에 저장될 키의 타입과 값의 타입을 전달합니다.

· keyType: 키의 자료형

· valueType: 값의 자료형

· Traits: map 내에서 상대적 순서를 결정하는 함수 객체의 클래스(기본값 less<KeyType>)

map<템플릿인수> 객체이름;

map<keyType, valueType, Traits> objName;

// ex) map<string, int> mp;

# 데이터 저장: insert() 함수

map<string, string> addrMap;

addrMap.insert(make\_pair("김철수", "서울시 종로구"));

addrMap.insert({"홍길동", "서울시 중구"});

# 데이터 접근: []연산자, find() 함수

cout << addrMap["홍길동"];

auto it = addrMap.find("김철수"); // auto = map<string, string>::iterator

addrMap["김영희"] = "대전시 동구"; // 키가 존재하지 않는 경우 삽입

addrMap["김철수"] = "서울시 성동구"; // 키가 존재하는 경우 수정

# 데이터 삭제: erase(it1), erase(it1, it2), erase(keyValue)

addrMap.erase(it1); // 삭제할 데이터를 가리키는 반복자를 지정하는 방법

addrMap.erase(it1, it2); // 삭제할 데이터를 가리키는 반복자의 범위를 지정하는 방법

addrMap.erase("김철수"); // 키를 지정하여 데이터를 삭제하는 방법

**Map 사용 예시**

#include    <iostream>

#include    <string>

#include    <map>

using namespace std;

template<typename T>

class LESS\_T {

public:

    bool operator()(const T& a, const T& b) const {

        return a < b;

    }

};

int main()

{

    map<string, string, LESS\_T<string>> pBook{

        { "김철수", "010-1111-1111" },

        { "김영희", "010-2222-2222" }

    };

    pBook["김철수"] = "010-3333-3333";

    pBook.insert(make\_pair("홍길동", "010-1234-5678"));

    pBook.insert({ "이순신", "010-0000-0000" });

    for (auto pb = pBook.begin(); pb != pBook.end(); ++pb)

        cout << pb->first << "  " << pb->second << endl;

    cout << pBook.size() << "명이 등록되어 있습니다." << endl;

    string  str;

    cout << endl << "찾을 이름 : ";

    cin >> str;

    auto result = pBook.find(str);

    if (result != pBook.end()) {

        cout << result->first << "님의 전화번호는 ";

        cout << result->second << " 입니다." << endl;

    }

    else

        cout << str << "님을 찾을 수 없습니다." << endl;

    return 0;

}

**Try-catch 예시**

try

{

if (inAccount != AC.getAccount() || inPass != AC.getPass())

{

throw 1;//int throw

}

else

{

cout << "출금액 입력: "; cin >> amount;

if (amount > AC.getBalance())

{

cout << "현재 잔액: " << AC.getBalance() << endl;

/\*

throw 'e';

//char throw

\*/

throw string("Underflow!");

return 0;

}

else

{

lbalance = AC.getBalance() - amount;

cout << "현재 잔액: " << lbalance << endl;

cout << AC.getName() << "님의 현재 잔액은: " << lbalance << "입니다." << endl;

}

return 0;

}

}

catch (int exception)

{

cout << "\n다음 입력을 다시 한 번 확인하세요!!!" << endl;

Account AC("", inAccount, inPass, 0);

//입력받은 계좌번호와 비밀번호로 객체 생성

AC.showAccount();

return 0;

}

//계좌번호와 비밀번호가 다를 경우의 예외처리

/\*catch (char exception)

{

cout << "희망 출금액: (" << amount << ")이 너무 많네요." << endl;

return 0;

}\*/

//희망출금액이 잔액보다 적을 경우의 예외처리

catch (string error)

{

cout << "희망 출금액: (" << amount << ")이 너무 많네요." << endl;

return 0;

}

**C++ 복습**

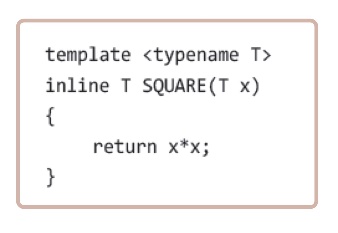
**Ch01**

함수 오버로딩(fuction overloading) : 매개변수의 선언이 다르면, 동일함 이름의 함수도 정의 가능

but, 반환형의 차이는 함수 오버로딩의 조건을 만족시키지 않는다.

매개변수의 디폴트 값 : 함수의 선언을 별도로 둘 때에는 디폴트 값의 선언을 함수의 선언부에 위치시켜야 함.

인라인 함수(inlint function) : 매크로 함수와 달리 자료형에 독립적이지 못하지만, 템플릿으로 극복가능



이름공간(namespace) : 존재하는 이름공간이 다르면 동일한 이름의 함수 및 변수를 선언하는 것이 가능함.

+함수의 선언과 정의의 분리시, 선언은 반드시 이름공간 안에서 해야함.

+선언된 이름공간의 이름이 동일할 경우, 동일한 이름공간으로 간주된다.

+이름공간을 명시하지 않고 함수를 호출하면, 함수의 호출문이 존재하는 함수와 동일한 이름공간안에서 호출할 함수를 찾음.

+이름공간은 중첩이 가능하여, 계층적 구조를 갖게끔 이름공간을 구성할 수 있다.

+이름공간의 별칭 지정 ex) namespace ABC = AAA::BBB::CCC;

+범위 지정 연산자를 이용한 전역변수의 접근 ex) ::val->전역변수는 이름공간이 없는 것이 이름

**Ch02**

**키워드 const의 의미 : 변수의 상수화, 포인터의 상수화**

Const int num=10;

* 변수 num을 상수화

Const int \* ptr1 = &val1;

* 포인터 ptr1을 이용해서 val1의 값을 변경할 수 없음.

Int \* const ptr2 = &val2;

* 포인터 ptr2가 상수화 됨

Const int\* const ptr3 = &val3;

* 포인터 ptr3가 상수화 되었으며, ptr3을 이용해서 val3의 값을 변경할 수 없음.

**실행중인 프로그램의 메모리 공간**

데이터 : 전역변수가 저장되는 영역

스택 : 지역변수 및 매개변수가 저장되는 영역(함수호출을 위한 메모리 공간)

힙 : malloc 함수호출에 의해 프로그램이 실행되는 과정에서 동적으로 할당이 이뤄지는영역

Malloc&free : malloc 함수호출에 의해 할당된 메모리 공간은 free 함수호출을 통해서 소멸하지 않으면 해제되지 않는다.

**Bool 자료형**

True와 false는 bool형 데이터이다.

True와 false 정보를 저장할 수 있는 변수는 bool형 변수이다.

True : “참”을 의미하는 1바이트 데이터

False : “거짓”을 의미하는 1바이트 데이터

* 정수가 와야할 위치에 오게 되면, 각각 1과 0으로 반환이 된다.

**참조자의 이해**

참조자 : 기존에 선언된 변수에 붙이는 ‘별칭’

1. 참조자의 수에는 제한이 없으며, 참조자를 대상으로 참조자를 선언하는 것도 가능함.
2. 배열의 요소 역시 변수의 성향을 지니기 때문에, 참조자의 선언이 가능하다.
3. Const 참조자

* 참조자를 const로 선언하면, 함수의 원형 선언만 봐도 값의 변경이 일어나지 않음을 판단할 수 있고, 실수로 인한 값의 변경이 일어나지 않는다.

**\* const참조자의 상수 참조**

참조자는 상수를 참조할 수 없지만, const 참조자는 상수를 참조할 수 있다. 상수를 메모리 공간에 임시적으로 저장하기 때문에,

**New&delete(malloc&free를 대신하는)**

Type형 변수의 할당 : type \* ptr1 = new type;

길이가 n인 type형 배열의 할당 : type \* arr1 = new type[3];

Type형 변수의 소멸 : delete ptr1;

Type형 배열의 소멸 : delete []arr1;

**C++에서 C언어의 표준 함수 호출 방법**

C를 더하고 .h를 빼라

#include <stdio.h> -> #include <cstido>

#include < string.h) -> #incldue <cstring>

**C복습**

**Part02**

**11-1. 배열의 이해와 배열의 선언 및 초기화 방법**

1차원 배열의 선언에 필요한 것 세 가지 : 배열이름, 자료형, 길이정보

배열의 위치 정보를 명시하는 인덱스 값은 1이 아닌 0에서부터 시작

배열의 모든 요소는 반복문을 이용해서 순차적으로 접근하는 것이 가능

**11-2. 배열을 이용한 문자열 변수의 표현**

Char형 배열

문자열의 끝에 자동으로 삽입되는 문자 : ‘\0’, ‘널(null)’문자

널(‘\0’) 문자의 아스키 코드 값 : 0 (정수형 출력시 0이 나옴)

문자의 형태로 출력할 경우, 아무런 출력이 발생하지 않음.

공백(‘ ‘) 문자의 아스키 코드 값 : 32

* 널 문자와 공백 문자 혼동 x

scanf함수를 이용해 문자열 입력 받을시, 서식문자 %s를 사용하고, &를 붙이지 않는다.

* scanf(“%s”, str);

C언어에서 표현하는 모든 문자열의 끝에는 널 문자가 자동으로 삽입된다. -> 널 문자가 존재하면 문자열이고, 널 문자가 존재하지 않으면 문자열이 아니다.

* C언어의 관점에서는 어떻게 선언이 되든 널 문자가 마지막에 존재하는 문자열이 된다.

문자열의 시작 : str[0]/문자열의 끝 : 널 문자->printf함수도 이 기준으로 문자열을 구분함.

\*scanf함수가 데이터를 구분짓는 기준 : 공백

**12-1. 포인터란 무엇인가?**

변수의 선언과 메모리 할당 -> 1바이트의 메모리 공간을 단위로 하나의 주소 값이 할당되며, 주소 값도 1씩 증가함.

포인터 변수 : 메모리의 주소 값은 저장하기 위한 변수

type \* : type형 포인터

type \* ptr; : type형 변수의 주소 값을 저장하는 포인터 변수 ptr의 선언

**12-2. 포인터와 관련 있는 연산자 : &연산자와 \*연산자**

&연산자 : 피연산자의 주소 값을 반환하는 연산자

\*연산자 : 포인터가 가리키는 메모리 공간에 접근할 때 사용하는 연산자

포인터의 형이 존재하는 이유는 포인터 기반의 메모리 접근기준을 마련하기 위함.

포인터 변수를 선언만하고 초기화하지 않으면, 포인터 변수는 쓰레기 값으로 초기화 됨.

Int \* ptr1=0;

Int \* ptr2=NULL;

키워드 NULL -> 값 0 -> 널포인터 -> “아무데도 가리키지 않는다.”

**13-1. 포인터와 배열의 관계**

배열 이름의 포인터형

\* %p는 주소 값의 출력에 사용되는 서식문자

배열의 이름=배열의 시작 주소 값, 값의 저장이 불가능한 상수=상수 형태의 포인터= 포인터 상수

|  |  |  |
| --- | --- | --- |
|  | 포인터 변수 | 배열의 이름 |
| 이름 여부 | 존재 | 존재 |
| 무엇을 | 메모리의 주소값 | 메모리의 주소 값 |
| 변경여부 | 가능 | **불가능** |

* 배열의 이름을 피연산자로 하는 \*연산이 가능함.

Int \* ptr = &arr[0]; // int \* ptr =arr;

**13-2. 포인터 연산**

TYPE형 포인터를 대상으로 n 증가 -> n x sizeof(TYPE)의 크기만큼 증가

결론 : arr[i]=\*(arr+i);

**13-3. 상수 형태의 문자열을 가리키는 포인터**

배열을 기반으로 문자열 선언 : **변수 형태의 문자열**

* Char str1[] = “My string”;

포인터를 기반으로 문자열을 선언 : **상수 형태의 문자열**

* Char \* str2 = “Your string”; : char형 문자 Y의 주소 값이 저장됨. 포인터 변수 str2가 가리키는 문자열은 그 내용의 변경이 불가능하다.

차이점 : 배열이름 str1은 계속해서 문자 M이 저장된 위치를 가리키는 상태이어야 하지만 포인터 변수 str2는 다른 위치를 가리킬 수 있다.

결론 : printf 함수는 문자열을 통째로 전달받는 함수가 아닌, 문자열의 주소값을 전달받는 함수이다. 따라서, 문자열 입력시 &를 쓰지 않음. 문자열은 그 자체로 주소 값을 갖고 있다.

**13-4. 포인터 변수로 이뤄진 배열 : 포인터 배열**

**14-1. 함수의 인자로 배열 전달하기**

함수호출 시 전달되는 인자의 값은 매개변수에 복사가 된다!! ->함수가 호출되고 나면, 전달되는 인자와 매개변수는 별개가 된다.

**배열을 함수의 인자로 전달하는 방식**

배열의 주소 값을 인자로 전달해서 이를 통해서 접근하도록 유도

ex) 배열의 주소 값 : arr, int \* param, int param[ ]

-> 배열의 주소 값을 알고 있으므로, 어디서든 배열에 접근하여 저장된 값을 참조하고 변경할 수 있음

**14-2. Call-by-value vs Call-by-reference**

단순히 값을 전달하는 형태의 함수호출 : Call-by-value

메모리 접근에 사용되는 주소 값을 전달하는 형태의 함수호출 : Call-by-reference

-> 함수의 인자로 전달되는 대상으로 구분

**scanf함수호출 시 &연산자를 붙이는 이유**

scanf함수의 호출도 Call-by-reference 형태의 함수 호출임.

문자열을 입력 받을 때, &연산자를 붙이지 않은 이유 : 문자열은 이름 그 자체로 배열의 주소 값이기 때문

**14-3. 포인터 대상의 const 선언**

**포인터 변수가 참조하는 대상의 변경을 허용하지 않는 const 선언**

const int \* ptr = &num; -> ptr이 가리키는 변수에 저장된 값을 변경하는 것을 허용하지 않겠습니다.

**포인터 변수의 상수화**

int \* const ptr = &num; -> 한번 주소 값이 저장되면 그 값의 변경이 불가능함.

ptr = &num2; //컴파일 에러 -> 가리키는 대상을 바꿀 수는 없음.

\*ptr = 40;; //컴파일 성공 -> ptr이 가리키는 대상에 저장된 값을 변경하는 것은 가능함.

**동시에 const 선언**

const int \* const ptr = &num;

\*ptr = 20; //컴파일 에러 -> 가리키는 대상을 바꿀 수 없음.

ptr = &age; //컴파일 에러 -> ptr이 가리키는 대상에 저장된 값을 변경하는 것도 불가능함.

**const선언이 갖는 의미**

프로그램 코드의 안전성을 높일 수 있다.