C++ 프로그래밍 정리

2-1) compile helloWorld

atom이라는 text editor에서 source file을 작성하고, source file의 경우는 text file이기 때문에 실행파일이 아님. -> 실행파일로 만들어주어야 한다 : 해당과정을 컴파일이라고 함

> g++ -o hello hello.cpp

➔ g++는 컴파일러

➔ -o의 경우 컴파일러 수행 후 실행파일이름이 무엇인지 알려주는 역할

➔ hello의 경우 컴파일 결과로 만들어지는 실행파일

➔ hello.cpp의 경우 무엇을 컴파일할 것인지 source file의 이름을 의미

➔ >: prompt를 의미하며, operating system이 현재 user input을 받을 준비가 되었다고 알려주는 문자열

터미널에서 prompt : 유저이름@우분투 설치 시 기계이름

~: 현재 어느 디렉토리에 있는지 보여주는 기호

terminal 내 ls 명령어 : 현재 current directory에 있는 파일들을 보여줌

- 파란색의 경우: directory, 하얀색: regular file, 초록색: executable file (실행파일)

pwd : present working directory, 현재의 디렉토리 위치를 알려줌

cd: change directory 디렉토리의 위치를 바꾸는 역할

2-2) helloWorld - Lab01

/* */ : 주석 comment를 의미, 컴파일러는 해석하지 않고, 프로그래머가 보기 위해 써놓음

/* */: C style comment, // : C++ style comment

cout << "Hello World!₩n"; //write

cout : standard output stream -> 터미널에 출력되는 화면

<< : output operator

"Hello World!₩n" : string(문자열)

; : 문장의 끝을 나타내주는 기호

// : 주석 comment

➔ 쌍따옴표 내에 있는 문자열을 cout에 출력해달라는 의미

2-4) thanks solution – Lab02

~: user의 home directory를 뜻함 / cd만 사용할 경우 cd ~와 동일한 의미

.의 경우는 current directory와 같으므로, cd .의 경우는 present working directory가 변하지 않음

cd .. 은 바로 전 directory로 이동

which 명령어는 executable file의 위치를 찾아서 보여주는 역할

실행파일 실행 시, ./를 붙여야 하는 이유: 현재 디렉토리에 있는 실행파일을 실행하기 위해서

➔ 설정해주지 않으면, 다른 9개의 디렉토리에서 찾아서 실행하려고 시도하기 때문

3-1) Managing Directory

.. : parent directory -> cd .. : parent directory로 이동

mkdir: directory를 하나 생성하는 명령어 -> mkdir C2020의 경우 C2020이름의 디렉토리 생성

cp: copy 복사하는 명령어 -> cp Desktop/* C2020 : Desktop 아래의 모든 파일을 C2020으로 복사

ls Desktop/ : Desktop 디렉토리 아래에 있는 모든 파일을 보여줌

rm: remove -> rm Deskop/* : Desktop 아래의 모든 파일을 삭제

cp hello.cpp a.cpp : hello.cpp를 복사해서 a.cpp 파일을 새로 생성

3-2) Char Variable

endl: new Line Character 출력

홀따옴표는 문자를 표시, 글자하나 -> char / 'apple'은 Compile Error

쌍따옴표는 문자열을 표시 (문자열이란 여러 개 문자들의 sequence를 의미), 순서가 중요

➔ String 타입은 존재하지 않음, char 타입을 여러 개 모아놓은 것

➔ 0개도 여러 개 이기 때문에 null string의 경우 ""로 표현

3-3 Assignment – Lab04

a라는 문자 상수를 출력 -> 주머니에 있는 문자 a를 출력 (var이라는 변수를 사용)

변수는 컴퓨터의 메인 메모리에 저장

= sign의 경우 오른쪽의 값을 왼쪽의 변수에 대입하는 의미, (== 같으면 true, 아니면 false를 출력)

치환문의 경우 RHS의 수식을 evaluation을 한 뒤에 해당 결과값을 LHS의 변수가 가리키고 있는 메모리에 저장하는 의미 / LHS에는 변수의 이름만 사용가능, 메모리의 주소를 가져야 하므로

cat명령: textfile의 내용을 그대로 화면에 보여주는 명령어, 짧은 text파일의 내용을 간단히 읽을때

g++ -g -o a a.cpp

-g: 디버깅을 할 때 필요한 정보를 executable 파일에 같이 저장하라는 의미

-> gdb를 사용할 때, -g option을 같이 사용해야함

3-4) gdb

gdb의 목적어로는 executable 파일의 이름

(gdb) 부터 화면에 타이핑하는 명령어는 바로 operating 명령어로 해석되는 것이 아닌 gdb가 받아서 해석 (수행)

l: source file을 list해주는 명령어, compile 했던 파일의 첫번째 10줄을 화면에 보여줌

r: compile은 이미 되어있기 때문에 run 명령어 수행 시, program start

debugging은 프로그램을 중간에 멈추고, 메모리 내용이 뭔 지, 에러가 왜 발생했는지 찾아야함

b: 프로그램을 중간에 멈추는 명령어 breakpoint / b 7: 7번 line에 breakpoint 설정

➔ Breakpoint 설정 후에 r을 수행하면, breakpoint에서 멈추고 멈춘 라인을 보여줌

➔ 해당 경우에는 멈춘 라인 전까지 수행

p: print a value of expression, 화면에 출력하는 명령어

n: next, 프로그램의 한 라인만 수행하고 멈추는 명령어

변수이름 앞에 &를 붙이면 : 변수의 주소 값을 의미 -> 주소 값은 16진수로 보는 것이 편함

c: continue running, r의 경우는 처음부터 프로그램을 실행, c의 경우 멈춘 부분부터 실행

➔ Breakpoint가 없다면 프로그램이 끝날 때까지 실행

q: quit, gdb 수행을 끝내고, operating prompt로 넘어감

3-5) Declaring Variables

변수란 프로그램에서 데이터를 저장하는 공간 -> 값을 나중에 사용하기 위해서 저장

변수 선언시, [자료형 변수이름;] 순으로 선언 -> 같은 타입이라면 한번에 여러 개 선언 가능

변수이름은 식별자 규칙을 따름

➔ 알파벳 문자, 숫자, 밑줄 문자로 구성

➔ 첫번째 문자는 반드시 알파벳 or 밑줄문자

➔ 대소문자를 구별

➔ C언어의 키워드와 같은 이름은 허용하지 않음

키워드란 C++ 언어에서 고유한 의미를 가지고 있는 특별한 단어, 예약어라고도 함

4-1) ArrayIndex – Lab05

배열이란 동일한 타입의 데이터가 여러 개 저장되어 있는 데이터 저장 장소를 의미

자료형 배열이름[배열크기]; 방식으로 선언

선언문 내의 []은 배열의 크기, 실행문 내의 []은 인덱스를 의미

배열의 원소는 치환문에서 LHS, RHS 모두 가능

4-3) RefOp gdb2

p arr : 배열을 프린트하는 경우에 배열의 원소들을 모두 보여줌 / 배열의 크기만큼의 string

인덱스 범위를 넘어가는 부분에 access를 할 경우 syntax error가 발생하지 않으므로 주의해야함

-1, -2 등 음수의 범위 인덱스도 주소를 찾을 수는 있지만, 값을 이용해서 쓰면 안됨

l: 현재 수행해야하는 라인을 가운데로 두고, 앞뒤로 총 10줄을 보여주는 명령어

bt: backtrace, 현재 내가 몇번째 명령어에서 멈춰있는지 확인하는 명령어 (수행 안한 상태)

del break: 모든 breakpoint를 다 지우는 명령어, del b의 경우 b가 bookmark, breakpoint인지 혼란

4-4) Array Initialization

배열의 크기보다 작은 개수만큼 초기화 한 경우 나머지 값은 0으로 채워짐

➔ 초기값이 일부일 경우 나머지 원소는 0으로 초기화

배열의 크기가 주어지지 않은 경우에는 자동적으로 초기값 개수만큼 배열의 크기로 설정

다차원 배열에서는 첫번째 차원만 생략가능, 나머지 차원은 생략 불가

➔ 리스트의 총 개수로 유추하는 것이 가능하기 때문

➔ 2차원 배열에서 두번째 인덱스의 크기보다 작은 값이 초기화될 경우 나머지는 0으로 채움

4-5) Multi Dim Array

주소가 1씩 증가하는 것이 아닌 4씩 증가하는 이유

➔ arr 내의 각각의 변수가 integer (4bytes)로 선언되었기 때문

4-6) Row Major Storage

주소는 1차원으로 연속적으로 증가

2차원 배열에서 원소를 메모리에 저장할 때, 2차원으로 저장할 수 없고, 1차원으로 만들어야 함

다차원 배열을 메모리에 저장하는 순서는 마지막 index가 먼저 변하는 순서로 저장

➔ 행 우선 저장 (row 저장 후 다음 row 저장하는 순서로) ; C언어에서 사용

열 우선 저장의 경우 첫번째 index가 먼저 변하는 순서로 저장 ; fortran에서 사용

5-1) Control Structure

block: 여러 개의 문장을 {}로 묶은 것을 의미하여 compound statements

statement가 한 개인 경우에는 {}로 묶지 않아도 됨

if문이 여러 개 있는데 else가 하나만 있다면, 해당 else는 가장 가까운 if와 연결

if 문장 안에 있는 문장이 break, continue일 경우, if 바깥의 문장들이 else로 들어가도 동일

switch문에서 break를 의도적으로 생략한 경우에는 즉, break문을 만나지 않으면 순서대로 실행

➔ 여러 개의 경우를 하나로 묶어서 처리하기 위한 기법

➔ switch문은 값이 같은 경우만 비교하므로, 범위를 나타낼 때는 if-else문을 사용

➔ switch 문에서 마지막 break는 생략해도 상관 없다.

5-2) Iteration

while문의 경우는 조건이 만족할 때까지 반복하는 구조, for문은 반복횟수를 지정하는 경우

5-3) break

while 조건 중에서 조건식이 0이 아니면 모두 참으로 간주하여 무한루프 -> break // 0=False

nested loop에서 outer loop를 종료하기 위해서는 bool 변수의 flag를 사용

➔ inner loop를 break하기 전에 flag를 변경하고, outer loop 조건에 따라 break

break의 경우는 전체의 loop를 끝내지만, continue는 현재의 iteration을 중단하고 다음 반복

➔ continue의 경우는 continue 아래의 문장은 실행하지 않고 다음 반복을 진행

6-1) Types – gdb3

unsigned의 경우에는 양수만 표현하기 때문에 더 큰 범위로 표현 가능

char 배열의 경우는 문자의 sequence = 문자열 -> "문자들을 순서대로 써줌"

sizeof() 연산자를 통하여 해당 변수나, 타입의 크기를 알 수 있음

배열의 이름을 sizeof() 연산자에 넣을 경우 배열의 사이즈를 알 수 있음

➔ 타입의 크기 X 배열의 크기

gdb에서 아무 명령어를 입력하지 않고 Enter를 입력하면, 이전 명령어를 반복해서 사용

p c를 수행

➔ c라는 값이 character이기 때문에 1byte만 읽어서 값을 2가지로 보여줌 숫자(십진수), 문자

p &c

➔ c라는 변수가 저장된 메모리의 주소값을 출력 / 주소, 주소가 가리키고 있는 곳의 문자열

print 명령을 수행하면 gdb는 print한 변수의 값을 자체 생성한 gdb 내부 변수에 치환

➔ 방금 print 했던 값을 나중에 쉽게 지칭할 수 있도록 하게 하기 위해 편리하도록

gdb에서 x라는 명령어는 argument로 주소값을 받음

➔ 해당 숫자에 해당하는 메모리 주소와, 해당 주소에 저장된 값을 출력

6-2) gdb – x

x/FMT ADDRESS -> default : x/1xw // 16진수 8자리 -> 4byte // 2진수는 8자리가 16진수 2자리

➔ FMT은 생략가능, repeat count + format letter (hex 16진수의 경우 x, 2진수 t) + size letter(b, w)

➔ Size letter의 경우 b(byte), h(halfword), w(word), g(giant),

➔ repeat count default값 1, format letter, size letter의 경우 이전에 사용했던 값

x/1xw $1 -> 1개의 word (4byte)를 보여주는데 hex(16진수)로 표현

x/4xb $1 -> 4개의 byte를 16진수로 표현

➔ 1word와 4byte는 개수는 똑같으나, 띄어쓰기가 다름, 4byte는 byte를 띄어쓰기로 끊어서

8byte -> 16진수로하면 16자리, 2진수로 하면 64자리

6-3) 기본자료형 char

char 1byte를 이용해서 ASCII set에 포함되는 문자들을 사용할 수 있는 타입

문자도 메모리에 표현하기 위해서는 2진수로 표현되어야 함

➔ 각각의 문자를 어떠한 2진수로 표현할 지를 정하는 표준들 중에 가장 많이 사용하는 것 ASCII, unicode

➔ ASCII는 7비트를 사용해서 영어의 알파벳, 특수문자를 표현

ASCII 코드 중 printable character가 아닌 문자들 -> 제어문자

➔ 제어 목적으로 사용, 줄바꿈 문자, 탭 문자, 백스페이스 문자 등

➔ 백슬래시의 경우 글자로 보는 것이 아닌 escape sequence가 시작됨을 알림

논리형 타입의 경우 정수형 변수로 변환이 될 경우 true = 1, false = 0

2진수를 8진수로 바꾸려면 3자리 씩 끊어서 변환, 16진수로 바꾸려면 4자리씩 끊어서 변환

6-4) 기본자료형 4fp

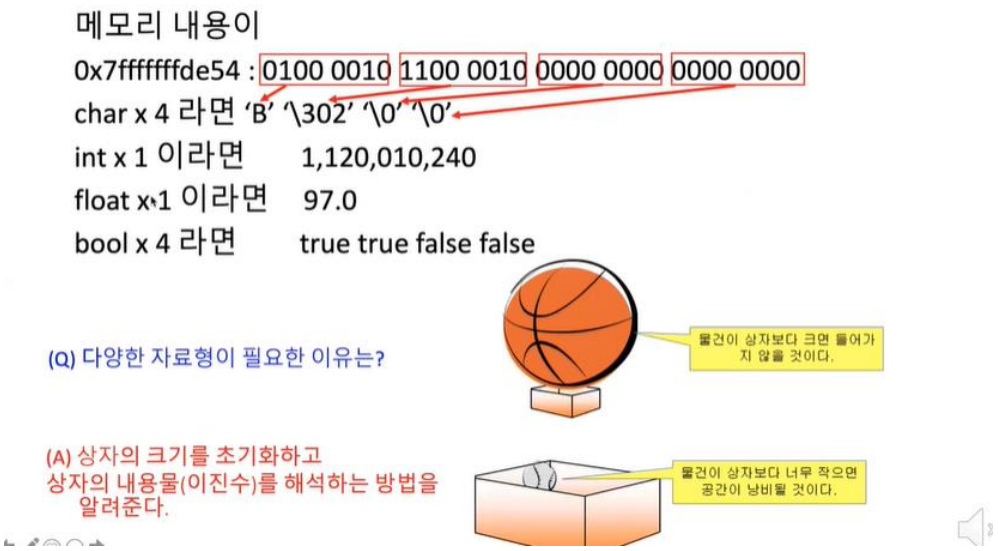
지수표현으로 하지말고 소수점을 찍어서 표현

➔ cout.setf(ios_base::fixed) : 지수로 숫자를 표현하지 말고 소수점을 찍어서 유효숫자를 표현

➔ cout.precision(30) : 소수점 아래 30자리까지

float type은 32bits로 숫자를 표현 -> 약 40억 개의 숫자만 표현 가능, 나머지는 근사치로

double은 64bits로 숫자를 표현 -> float type보다는 정확한 숫자 표현 가능



7-1) Pointer Variables

주소값을 $4 로 저장한 것처럼 gdb 에서만 주소값을 저장하는 것이 아닌 프로그램 내에 저장

➔ 포인터 타입의 변수

➔ 메모리의 주소 값을 가지고 있는 변수

➔ 포인터를 이용하여 메모리 내용에 직접적으로 접근할 수 있음

메모리의 단위는 바이트, 즉 8bit / 변수의 크기에 따라 차지하는 메모리 공간이 달라짐

➔ 포인터 변수는 변수의 주소를 가지고 있는 변수로, 선언문에서 *를 이용

➔ Int *p = &I; 의 경우 integer 변수의 포인터로 선언과 동시에 초기화

➔ **Int *p; p = &I; 와 같은 역할 // *p = &i 는 아님**

포인터 타입의 변수의 크기는 시스템마다 다름, 주소의 크기는 같은 시스템 내에서 같음

➔ 가리키는 데이터의 타입과는 상관없이 포인터 자체는 메모리의 주소 값이므로 동일

➔ 64-bit machine 에서는 8byte 를 차지

➔ 포인터 변수에 저장되는 것은 주소이므로, char, int 에 대한 주소인지 상관없이 같음

7-2) Indirection Operator 간접 참조 연산자 *

C/C++ 프로그램 소스에서 선언문과 실행문을 구분하면 혼동을 줄일 수 있음

➔ 같은 기호가 선언문, 실행문에서 서로 다른 의미를 가지고 있음

➔ 선언문에서의 *는 포인터 변수라는 것을 선언하기 위해 사용된 기호

➔ 실행문 내에서 변수 앞에 *가 있을 경우, 간접 참조 연산자를 의미 : 주소에 해당하는 값

➔ 간접 참조 연산자를 사용하기 위해서 변수는 포인터 타입이어야 함

➔ 포인터 변수에 상수 값을 주는 것은 Compile Error 발생, 주소 값을 할당해야 함

➔ 실행문 내의 *는 간접 참조 연산자를 의미하며, &의 반대 역할

➔ 포인터 변수가 가리키는 변수의 값을 가져오는 연산자

➔ 간접 참조 연산자를 이용하여 값을 치환할 수 있음 -> 실제 값을 치환하는 것과 같음

7-3) Type Casting Operator 형 변환 연산자

명시적 형 변환 (Explicit Type Casting): 사용자가 데이터 타입을 변경하는 것

묵시적 형 변환 (Implicit Type Casting): 형 변환 연산자가 없어도 자동으로 형 변환

➔ 명시적 형 변환을 하지 않더라도, 컴파일러가 알아서 형 변환을 수행

➔ 오른쪽에 있는 자료형을 왼쪽에 있는 자료형에 자동으로 맞춤

Pointer Type Casting

Float Pointer -> Interger Pointer : i_p = (int *)f_p -> 같은 주소에 있는 값을 interger 로 해석

pointer 변수에 상수 값을 주는 것은 compile error -> (float or int *)로 type casting 필요

변수의 자료형은 변수의 크기와 변수의 내용물을 해석하는 방법을 알려줌

주의점

1. 포인터 타입과 변수의 타입은 일치해야 함
2. 초기화가 되지 않은 포인터를 사용하면 안됨

- 초기화하지 않으면 의미가 없는 garbage value 를 가리킴 -> run-time error

3. 포인터가 아무 것도 가리키고 있지 않은 경우에는 NULL 로 초기화

- NULL 포인터를 가지고 간접 참조 시 하드웨어로 감지할 수 있음
- 포인터의 유효성 여부 판단이 쉬움

배열의 이름만 사용하면 배열의 시작 주소와 같은 의미 int a[10]; // a = &a[0]

포인터 변수에 절대값 상수를 취하면 안됨

8-1) Literal Constant 리터럴 상수

치환문에서 LHS 에는 1 차적으로 변수의 이름이 올 수 있고, *(포인터변수) 형태도 가능

치환문 자체도 수식이므로 RHS 부분에 다른 치환문을 쓰는 것도 가능

literal constant: 이름이 없는 상수

symbolic constant: 이름이 있는 상수

➔ literal constant 를 표현하기 위해서는 변수와 마찬가지로 메모리에 상자를 마련하여 저장

➔ 변수와 다르게 상자에는 이름이 없음 -> literal constant 의 값을 바꿀 방법이 없음

➔ 하지만 Symbolic Constant 는 이름이 있는 상수

➔ 만드는 방법은 2 가지, **const** int CONST2 = 12; #define CONST1 12

➔ 프로그램 내부에서 상수 CONST2 의 값을 변경하면 안됨, 상수를 치환 시, Compile Error

➔ 치환문의 LHS 에는 변수의 이름만 가능하므로, 상수의 이름은 불가

➔ 변수와 상수를 구분하기 위해서 상수의 이름을 대문자로 설정하는 것이 Convention

➔ Convention 을 어긴다고 하더라도 Compile Error 는 발생하지 않지만 이해하기 쉽도록

➔ 변수를 선언할 때는 초기화를 하지 않더라도 괜찮지만, 상수는 초기화 필수

➔ 왜냐하면 상수는 값을 초기화할 때만 값을 선언할 수 있기 때문

문자열은 항상 끝에 문자열이 끝났다는 것을 알리기 위해 끝에 0 의 값을 가지는 Null 문자가 있음

➔ sizeof("abcde") = 5*1 + 1(null) = 6byte

실수 리터럴 상수는 기본적으로 double type, 정수 리터럴 상수는 기본적으로 signed int

8-2) Type Casting 상수의 형 변환

정수 덧셈 연산자와, 실수 덧셈 연산자는 눈에는 같아 보이지만, 기계어 코드가 다름

복잡한 타입으로의 변환: upcasting // 단순한 타입으로의 변환: downcasting

1 바이트 = 16 진수에서 2 자리

8-3) Symbolic Constant 기호상수

기호 상수: 기호를 이용하여 상수를 표현한 것

➔ 가독성이 높아지고, 값을 쉽게 변경할 수 있다는 장점

➔ C style 은 #define 이용, C++ style 은 const 키워드 사용

➔ Symbolic Constant 의 값을 실행문에서 바꿔줄 경우 Compile Error 발생

➔ unsigned keyword 는 음수가 아닌 값만을 나타내고, default 의 경우 signed

➔ enumeration type define -> enum ANIMALS {DOG, CAT, BIRD}; ANIMALS a;

➔ DOG, CAT, BIRD 를 순서대로 0, 1, 2, 3 이라고 지정하는 것과 같음

➔ 증가하는 정수값들을 가지도록 선언하는 것을 한번에 symbolic constant 선언

➔ 실수의 경우에는 정수와 달라서 양수로 표현가능 한 것이 모두 음수로 표현 가능

9-1) function 함수

C++에서는 모듈러 프로그래밍을 위해서 함수라는 개념 사용

➔ 각 모듈들은 독자적으로 개발 가능, 다른 모듈과 독립적으로 변경 가능

➔ 유지보수가 쉬워지고, 모듈의 재사용이 가능

gdb 의 commands 에서 n(next)와 s(step)의 차이

➔ n 의 경우 함수의 호출을 한 후에 함수를 끝까지 수행 후 그 다음 문장에서 멈춤

➔ s 의 경우 함수를 호출하고, 함수 호출의 첫번째 문장에서 멈춤

9-2) Parameters 인자와 매개변수

Main 함수에서 호출하는 함수에 전달하는 값을 argument (인자)라고 하며

호출되는 함수에서 전달되는 인자를 받는 변수를 parameter (매개변수)라고 한다

인수는 여러개가 가능하지만, 반환값은 하나만 가능 반환하기 위해 return 사용

함수 원형 (function prototype) : 컴파일러에게 함수에 대해 미리 알리는 것

Main 함수에서 함수를 호출할 때, 함수의 원형이 main 함수 앞에 선언 X -> Compile Error

➔ 반환형 함수이름 (매개변수 1, 매개변수 2); 매개변수 리스트에 매개변수 이름을 안써도 됨

디폴트 매개변수 : 함수 호출 시, 인자를 전달하지 않아도 디폴트 값을 대신 넣어주는 기능

➔ 함수 원형 정의할 때, 미리 매개변수에 치환문을 통해 선언 필요

➔ 여러 개의 매개변수가 있다면 뒤에서부터 앞쪽으로만 정의할 수 있음

중복 함수 (overloading functions)

같은 이름을 가지는 함수를 여러 개 정의하는 것 -> 함수의 매개변수가 다르면, 서로 다른 함수

같은 일을 하는 함수를 같은 이름으로 함수를 사용할 수 있다는 장점 -> 함수이름 재사용 가능

function 의 signature 가 함수의 이름만으로 구분되는 것이 아닌, 파라미터 갯수, 순서, 타입 포함

➔ 함수의 반환 값은 function signature 가 아님, 따라서 반환형이 다른 경우는 중복 불가

9-3) Library Functions 인라인 함수

컴파일러가 실행 파일을 만들 때, 함수 호출로 번역하지 않고, 코드를 복사해서 실행파일 생성

➔ 인라인 사용시, 함수 호출을 하는 것이 아닌, 코드 자체가 바뀌어서 컴파일 진행

➔ 인라인으로 선언되지 않은 함수도 컴파일러가 최적화 정도에 따라 inline 으로 번역할 수 있음

➔ 실제로 인라인 함수로 만드는 것이 수행시간을 더 빠르게 할 수 있음

라이브러리 함수 : 컴파일러에서 제공하는 함수

표준 입출력, 수학 연산, 문자열 처리, 시간처리, 오류처리, 데이터 검색과 정렬 등

9-4) rand function

standard library 에 구현되어 있는 rand 함수와 srand 함수

rand 함수를 다시 실행할 경우 처음과 같은 랜덤 값이 출력

random number generator 를 seed 를 이용하여 initialize 하는 것 -> srand()

➔ seed 값이 같으면 같은 random sequence 가 반복됨

➔ 프로그램 수행 시 마다 seed 값을 다르게 하기 위해서는 상수로 주면 안됨

➔ 수행할 때마다 달라지는 값을 주어야 함

➔ time 함수의 인자를 null 로 줄 경우, 현재의 시간 return

➔ cin.ignore(); 입력을 받지만, 입력받은 값을 아무 곳에도 치환하지 않고 다음줄로 넘어감

➔ 입력할 때까지 프로그램이 다음 줄을 수행하지 않고 기다림 -> 시간 간격을 두기 위해

9-5) Command Line Argument // 포인터 타입의 데이터를 가지는 Array 사용

Character pointer 의 경우에는 특별하게 취급되어서 pointer value 를 16 진수로 보여주는 대신에 해당 포인터가 가리키고 있는 장소에 가서, 저장되어 있는 문자열을 보여줌

주소 값이 무엇인지 16 진수로 보고 싶을 때는, character pointer 가 아닌 다른 pointer 로 변환

string 을 integer 로 바꿔주는 함수 = atoi() -> cstdlib 에 prototype 이 정의됨

➔ character pointer 를 인자로 주면, atoi 함수는 숫자로 바꿔서 integer type 으로 return

10-1) Local Variable 변수의 범위

범위: 변수가 접근되고 사용되는 범위 -> 변수가 선언되는 위치에 따라서 결정

전역변수: 함수의 외부에서 선언 (초기값 지정하지 않아도 자동 0, 프로그램 시작 ~ 종료)

지역변수: 블록 안에서 선언 (선언된 블록이 지역변수의 scope)

변수의 생존기간: 변수가 생겨서 없어질 때까지의 기간 (선언시점 ~ 블록 끝날 때까지)

➔ lifetime 중간에도 변수의 사용이 불가능할 수도 있음 -> lifetime 은 scope 와 같지 않음

➔ lifetime 은 변수가 access 가 가능하지 않더라도 메모리에 존재하는 기간

함수의 매개변수도 지역변수로 볼 수 있음 -> 함수의 function body 가 scope

10-2) Scope GDB – 변수의 범위

전역변수와 지역변수의 이름이 같다면, 지역변수는 전역변수를 가린다.

➔ 전역변수는 lifetime 이지만, 접근이 불가능, 전역변수 사용 불가

프로그램이 시작되기 전에도 전역변수는 값을 볼 수 있음, 지역변수는 볼 수 없음

gdb 에서 info locals 는 local 변수들을 모두 보여줌

전역변수와는 다르게 지역변수는 값이 0 으로 초기화되지 않음, 랜덤하게

bt: backtrace 라는 명령어 -> 현재 수행하고 있는 함수의 인자값, 함수를 호출한 것은 누구인지

disp a: gdb 에서 명령어를 하나씩 실행할 때마다 a 라는 변수를 스토킹 하는 것 -> 계속 보여줌

10-3) Static External – 변수의 범위

정적으로 할당되는 변수의 생존기간 = 프로그램 실행 시간 동안 계속 유지 (전역변수)

자동으로 할당되는 변수의 생존기간 = 블록 생성 ~ 블록 나올 때 (지역변수)

저장 유형 지정자 static (for local variables)

➔ static keyword 의 역할: local 변수를 정적변수로 만들어줌 -> 프로그램 시작 ~ 끝 유지

➔ 초기화는 한 번만 일어나고, 더 이상 일어나지 않음, 블록이 끝나서도 메모리에 유지

저장 유형 지정자 extern

컴파일러에게 변수가 다른 곳에서 선언 되었음을 알림, 새로 선언되는 것이 아님

extern 을 이용하여 다중 소스 파일에서 변수들을 연결할 수 있음

static 을 전역변수나 전역함수에 쓸 경우, 지역변수 static 선언의 의미와 다름

➔ 선언된 file 외부에서 접근 불가

function prototype 을 하나하나 만드는 것보다는 헤더파일로 만드는 것이 좋음

➔ 헤더파일의 경우 컴파일을 따로 하는 것이 아닌, #include “scope2-1.h”와 같이 사용

11) Make Utility

source file 이 여러 개일 때, 컴파일 시 컴파일 명령이 길어짐

➔ 이러한 입력을 매번 단순화하기 위해서 Make Utility 사용

make 를 사용하기 위해서는 소스 파일들끼리의 의존성과 각 파일에 필요한 명령을 정의함으로써 프로그램을 컴파일 할 수 있으며, 최종 프로그램을 만들 수 있는 과정을 서술할 수 있는 표준적인 문법을 가지고 있음

➔ 개발 명령을 자동화하여 개발자의 수고를 덜고, 시간을 절약할 수 있음

정해진 시스템파일이 저장된 위치에서 헤더파일을 찾을 수 있으면 <>

현재 컴파일하고 있는 디렉토리에서 헤더파일을 찾을 수 있다면 ""

#include 를 해주면 file 들에 있는 내용들이 copy & paste 로 소스파일 내로 들어감

-c 옵션은 link 는 하지 않고, object file 까지만 생성하는 옵션

dependency list 파일의 수정시간이 target file 보다 최근이거나, target file 이 없으면 명령어 수행

명령어 앞에는 반드시 tab 문자가 있어야 함

Make {-f filename} {target}.

➔ default input file : Makefile // default target : make file 중 첫번째 target

Makefile 에서 비어있는 행은 무시하고, 주석처리는 #으로, 라인이 길어질 경우 ₩를 이용

;로 명령 행의 구분, dependency list 에 아무 파일도 없을 경우 -> 항상 명령문 수행

Macro 사용 -> $(variable) : 괄호 내에 매크로 변수를 치환

➔ 일괄적으로 다른 컴파일러를 사용하고 싶을 때, 용이 ex) CC = g++ $(CC)

➔ CPPFLAGS = -W -Wall 의 경우 warning error 를 다 보여달라는 option

➔ target, object list 또한 매크로로 지정 가능

➔ &^: dependency list 에 있는 모든 파일 이름을 써주는 것과 같음

➔ &@: target 에 있는 파일로 치환, &<: dependency list 중 첫번째만

ls -l f1* : f1 으로 시작되는 파일들의 수정 시간을 확인

12-1) Expression 수식

+, - 부호 연산자의 경우에는 단항 연산자 (피연산자의 수가 1 개이므로)

치환문(대입 연산자)에서는 왼쪽에는 항상 변수 또는 alias(포인터의 간접참조)가 와야 함

12-2) Expression 2

&&의 연산자는 ||보다 우선 순위가 높음

Short circuit (단축 계산) : &&의 경우 처음이 거짓이면 거짓, ||의 경우 처음이 참이면 참

결합 규칙: 만약 같은 우선순위를 가지는 연산자가 많을 때, 어떤 것부터 먼저 수행할까

12-3) Expression 3

XOR: 두 값이 같으면 0, 다를 경우 1

포인터끼리의 덧셈을 하는 것은 아무 의미가 없기 때문에 안됨

포인터 연산에서 덧셈연산이 무조건 1 을 증가하는 것이 아닌 포인터 타입에 따라 증가 결정

뺄셈도 포인터 타입의 크기에 따라서 나누어서 결정 ex) int 20 의 경우 20/4 = 5

13-1) Call by Value 값에 의한 호출

포인터 타입을 이용하면, 함수를 호출했을 때, 함수 내에서 caller 의 변수 값을 고칠 수 있음

➔ Call by Reference 같은 효과, 변수의 주소를 인자로 넘겨주었기 때문

13-2 Reference Type

Reference 를 선언하는 방법은 선언문에 &기호를 추가

➔ Reference type 으로 선언된 변수는 새로 메모리를 할당 받지 않음

➔ sum, diff 는 각각 k, l 의 별명이 됨 -> k 의 별명이므로 k 의 값이 바뀌게 됨

➔ call by reference 를 포인터를 사용할 때보다 훨씬 더 깔끔하게 사용할 수 있음

reference 는 선언문에서만 초기화가 가능하므로 반드시 초기화를 해줘야 함

reference 는 변수로만 초기화될 수 있음 **ex) int &r = i; -> r 은 i 의 별명**

실행문에서 &(reference 기호)는 주소생성연산자

선언문에서는 reference 선언을 하는데 사용

➔ 함수 인자도 아닌데 reference 선언을 하는 것은 아무 의미가 없음

➔ 함수 인자로만 사용할 것

변수의 별명을 통해서도 변수의 값을 바꿀 수 있음

Reference 인자 초기화 시, & 붙이면 안됨 -> int &sum = k; 의미이므로

포인터 인자 초기화할 경우에는 -> int *sum = &k; 의미이므로 & 필요

인자의 타입을 reference 로 주는 이유는 call by reference 를 구현하기 위해

➔ 효율성을 위해서도 가능

➔ 객체의 크기가 큰 경우 복사는 오래 걸려서 레퍼런스로 처리하는 것이 유리

const int *p1

➔ p1 은 const int 에 대한 포인터 -> **p1 이 가리키는 내용이 상수**

int * const p2

➔ **정수를 가리키는 p2 가 상수** -> p2 의 내용이 변경될 수 없음

const 를 reference 앞에 붙이면 레퍼런스가 가리키고 있는 변수가 변경 불가

➔ const int &p

➔ p 는 reference 로 전달받은 인자

➔ 단순히 복사하지 않기 위해 전달 받은 경우, 함수 내에서 값이 변경되지 않도록

14-1) Dynamic Allocation 메모리의 동적할당

배열의 이름만 사용했을 경우에는 첫번째 원소의 주소와 같음

배열이름에 1 을 더했을 때, 배열의 두번째 원소의 주소를 의미, *표시 -> 저장된 값

Integer pointer 가 1 증가할 경우, 주소 상으로는 저장할 만큼 뛰어넘은 다음 주소 +4

Character pointer 는 주소값을 출력하는 대신에 포인터가 가리키고 있는 위치부터 문자열 출력

➔ (void *) 이용하여 주소 값 확인 가능

포인터를 사용한 배열에 접근 -> 인덱스 표기법보다 빠르다, 원소의 주소를 계산할 필요 X

➔ sum += *p++

프로그램 시작 전에 미리 정해진 크기의 메모리를 할당 받는 것 -> 정적 메모리 할당

동적 메모리 할당: 프로그램이 실행되는 도중에 동적으로 메모리를 할당 받는 것

➔ 사용이 끝나면 시스템에 메모리 반납, 필요만 만큼만 할당 받으므로 효율적인 메모리 사용

➔ new, delete 사용 (C 언어에서는 malloc(), free() 사용)

14-2) Passing Array

포인터에 대해서도 array 연산자를 쓸 수 있음 -> pointer 변수와 array 이름은 interchangeable

➔ p[0] = *(p+0) 같은 의미 // void f(int *p) == f(int p[])

지역 변수는 함수가 종료되면 소멸되기 때문에 주소를 반환하면 안됨 -> 컴파일은 되고 경고

core file: segmentation fault 가 발생했을 때, 프로그램 수행 시 메모리를 그대로 file 로 dump

➔ core file 의 size limit 을 0 으로 했을 때, core file 이 안 생길 수도 있음

➔ core file 에 대한 system limit 을 확인 : ulimit -c -> ulimit -c unlimited 는 무한대로 늘림

➔ core file 은 gdb 를 이용해서 열 수 있음 : gdb (executable 파일 이름) core

동적으로 할당 받은 주소를 반환하는 것은 가능 -> delete 로 지우기 전까지는 heap 에 존재

14-3) Allocation 1D // 1 차원 배열을 동적으로 할당

14-4) Allocation 2D // 2 차원 배열을 동적으로 할당

14-5) malloc

delete 를 한다고 해서 pointer 의 값이 바뀌지는 않음

➔ delete 후에는 pointer 를 이용해 메모리 액세스 하면 안됨

return 타입이 void pointer 이므로, int pointer 로 변경 필요