**C/C++ 언어특성**

**절차지향** = C

* C : a….b…c… 순서대로 실행
* 소규모 프로젝트는 문제없지만 대규모 프로젝트에서 사용하기 힘듦
* 객체(요소) 분리에 불리 함수 상호 작용

**객체지향 Object Oriented Language** = C++ / java / js …

* Class (C언어에서 Function + struct를 합친구조)
* 컴포넌트 분리 / 덩어리 형태로 작업 / 관계성 유연
* 설계와 구현을 분리

Ex) 함수의 선언과 정의를 따로

* 구조가 간결 해진다 구현이 필요한 부분은 나중에 라도 추가 수정 가능

소감

C 언어는 사용자가 잘 알아야 사용가능 하다 함수가 어떻게 왜 작동되는지 개발자가 잘 알아야 사용가능

C++ 객체지향에선 객체의 작동 원리를 알고 있으면 객체를 쉽게 가져와서 사용한다 라는 느낌??

C: 옷을 빨려면 내가 세탁(함수)을 잘해야 한다.

C++ : 옷을 세탁 할려면 어느 빨래방(객체)이 잘하는지를 알아야한다.

비용(대가) 효율(유지보수관리) 성능

C < C++ C < C++ C > C++

CHATgpt -

OOP(개체 지향 프로그래밍)는 소프트웨어 시스템을 설계하고 구현하기 위해 개체와 개체의 상호 작용을 사용하는 것을 강조하는 프로그래밍 패러다임입니다. OOP는 추상화, 캡슐화, 상속 및 다형성의 원칙을 기반으로 합니다.

추상화는 개체의 필수 기능과 동작을 식별하고 해당 기능과 관련되지 않은 세부 정보를 무시하는 프로세스를 나타냅니다. OOP에서 추상화는 데이터와 동작을 캡슐화하는 클래스와 객체의 생성을 통해 달성됩니다.

캡슐화는 객체의 내부 세부 정보를 외부 세계로부터 숨기고 객체와 상호 작용하기 위한 잘 정의된 인터페이스를 제공하는 방법을 말합니다. 캡슐화는 더 나은 데이터 보안, 유지 관리 및 유연성을 허용합니다.

상속을 통해 한 클래스가 다른 클래스의 속성과 동작을 상속할 수 있습니다. 파생 클래스는 기본 클래스의 속성과 메서드를 상속하며 고유한 기능을 추가할 수 있습니다. 상속은 코드 중복을 줄이고 코드 재사용을 촉진하는 데 도움이 됩니다.

다형성(Polymorphism)은 객체가 다양한 형태를 취할 수 있는 능력을 의미합니다. OOP에서 다형성은 메서드 재정의 및 메서드 오버로딩을 통해 달성됩니다. 메서드 재정의를 통해 파생 클래스는 기본 클래스에 이미 정의된 메서드의 자체 구현을 제공할 수 있습니다. 메서드 오버로드를 사용하면 이름은 같지만 매개 변수가 다른 여러 메서드가 클래스에 공존할 수 있습니다.

객체 지향 언어에서 기본 구성 요소는 클래스, 객체, 메서드 및 속성입니다. 클래스는 개체의 특성과 동작을 정의하는 청사진입니다. 개체는 클래스에 의해 정의된 특성에 대한 고유한 값 집합이 있는 클래스의 인스턴스입니다. 메서드는 개체와 연결되어 개체의 동작을 수정하는 데 사용할 수 있는 함수입니다. 속성은 개체와 연결되고 데이터를 저장하고 검색하는 데 사용할 수 있는 변수입니다.

널리 사용되는 객체 지향 언어의 예로는 C++, Java, Python, Ruby 및 C#이 있습니다. 객체 지향 프로그래밍은 최신 소프트웨어 개발에 널리 사용되며 특히 개발에 유용합니다

자료형

일정길이의 메모리에 저장된 정보를 해석하는 방법

***1. 기본 자료형***

 자료형, 데이터 타입이라고 내게 더 익숙하지만, MS Docs 기준의 명칭으로는 Bulit-in type으로, 기본 자료형이라고 명시되어있다. 이는 C++ 언어의 표준으로 지정되어 헤더파일에 저장되어 있는 게 아닌 컴파일러가 사용하는 이들에게 제공되는 것이다. 기본 자료형은 세 가지 범주로 나타낼 수 있으며, 이는 정수형, 부동 소수점형, void형이 있다. 각각의 자료형은 컴파일러에 의해서 그 자료형에 맞는 고유한 방식으로 처리하게 된다. 다음은 각각의 자료형과 그 자료형이 지니고 있는 값의 범위와 바이트의 크기, 조사하면서 알게 된 특징에 대해 적으려고 한다.

***2. 정수형***

 모든 자료형의 범위는 해당 자료형의 바이트 크기의 영향을 받는다. 예를 들어서 int형 자료형은 바이트 크기가 4바이트인데, 이를 컴퓨터가 표현할 수 있는 최소 데이터 단위인 비트로 바꾸면 32비트라는 값을 얻을 수 있다. 데이터 범위는 2의 (비트)제곱만큼의 수를 가진다. 예를 들어, unsigned char는 바이트 크기가 1바이트인데 이는 8비트이므로 데이터 범위는 2의 8제곱을 계산하여 256이라는 것을 알 수 있다. 하지만 아래에 기재된 표에 의하면 unsigned char의 데이터 범위는 0 ~ 255이다. 이는 8비트가 표현할 수 있는 최대의 수가 255이기 때문이다. 각 비트가 표현할 수 있는 최대의 수는 2의 (해당 비트)제곱 - 1이다. 그러므로 8비트의 최대가 표현할 수 있는 최대 범위는 255가 된 것이다.

 양수는 모든 정수형 자료형에서 표현할 수 있다. 하지만 음수와 같이 부호를 지니고 있는 정수의 표현은 어느 자료형을 사용하느냐에 따라서 사용이 제한될 수 있다. 이에 대한 구분은 signed와 unsigned 키워드가 있다. 사용하고자 하는 자료형의 앞에 붙여서 사용하고자 하는 의미를 살리는데, signed는 양수, 음수 값을 데이터 범위로 모두 사용하는 키워드이고, unsigned는 음수를 제외한, 0을 포함한 양수 값을 데이터 범위로 사용하는 키워드이다. 예를 들어, unsigned char 자료형은 0 이상의 양수 범위를 사용하므로, 바이트 크기 1바이트에 해당되는 값인 256을 양수에 맞춰서 표현하는 것으로 범위는 0 ~ 255가 된다. 하지만 char 자료형은 signed 키워드를 묵시적으로 앞부분에 붙인 것을 생략한 모습이므로, 음수와 양수를 모두 표현할 수 있는데 바이트 크기 1바이트에 해당되는 값 256만큼 음수와 양수를 표현해야하기에 약 절반을 나눠 음수와 양수를 표현한다. 그렇기에 char형 데이터 범위는 -128 ~ 127를 가진다.

 정수형은 기본적인 형식으로 int 자료형을 사용한다. 하지만 int 자료형의 데이터 범위로 표현하고자 하는 수가 없는 경우나 int 자료형만큼의 바이트 크기를 사용하지 않아도 되는 경우 등 다른 정수 자료형을 사용할 때도 있다. 이를 위해 short, long, long long 자료형이 있다. 각 자료형의 크기 관계는 다음과 같다.

1 == sizeof(char) <= sizeof(short) <= sizeof(int) <= sizeof(long) <= sizeof(long long)

 short 자료형은 16비트 이상, long 자료형은 32비트 이상, long long 자료형은 64비트 이상이어야 한다. 그리고 사용하고 있는 short, long, long long, unsigned, signed은 여러가지 동의어를 가지고 있다. 아래는 동의어끼리 모아서 한 줄로 정리해놓은 것이다.

**○ short, short int, signed short, signed short int**

**○ unsigned short, unsigned short int**

**○ int, signed, signed int**

**○ unsigned, unsigned int**

**○ long, long int, signed long, signed long int**

**○ unsigned long, unsigned long int**

**○ long long, long long int, signed long long, signed long long int**

**○ unsigned long long, unsigned long long int**

***2. 부동 소수점형***

 부동 소수점형, 다른 말로 실수형이라고 하는 이것은 정수형에 소수점을 포함하여 더욱 세밀하고 광범위한 값을 표현하기 위해 제공되는 자료형이다. 부동 소수점형의 종류로 float, double, long double이 있다. 이 세 자료형에 대한 크기 관계는 다음과 같다.

sizeof(float) <= sizeof(double), sizeof(double) <= sizeof(long double)

 한 가지 눈에 들어오는 것은 float 자료형과 double 자료형이 같을 수 있다고해서 float 자료형이 long double 자료형과 같을 수 있는 것은 아니다. MS Docs에서는 long double과 double을 컴파일러에서 동일하게 처리한다고 한다.

***3. void형***

 void형은 값이 없는 집합을 이야기한다. void형은 값을 아무것도 반환하지 않는 함수를 선언하거나 임의로 지정하여 어느 형식으로 변환 혹은 캐스팅 가능한 generic 포인터로 주로 사용된다. 하지만 다음 3가지 상황에서 사용이 제한된다.

 - 표현 구문, 콤마 연산자 의 왼쪽 피연산자, 삼항 연산자의 2번째나 3번째 피연산자

***4. std::nullptr\_t***

 다음은 MS Docs에서 주된 세 가지라고 표현한 자료형 이외의 설명을 기재해놓았다. C++11에 추가된 nullptr은 포인터에 0이나 NULL 대신 어떤 값도 가리키지 않는 포인터로 명시하기 위해 존재하는데 이것의 자료형이 std::nullptr\_t이다.

이것 역시 기본 자료형으로 취급된다.

***5. bool형***

 bool형은 참과 거짓을 구분하는 true와 false만을 나타내기 위한 자료형이다. 2가지 수를 위해 1바이트라는 크기는 낭비일 수 있으나 최소 1바이트의 의미를 갖는 8비트가 되어야 메모리에 대한 액세스 등 여러가지 필요한 이유에 부합되기에1바이트를 할당하는 것 같다.

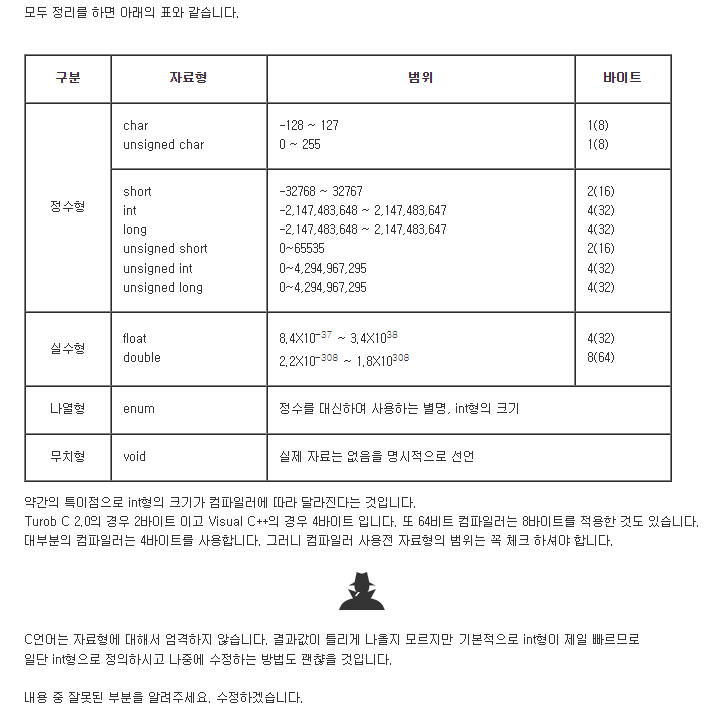
***6. 문자형***

문자형인 char 자료형은 문자들을 효율적으로 인코딩하기 위한 문자 표시 형식이다. C++에서는 char를 signed char, unsigned char로 서로 다른 형식으로 처리한다. 이 둘은 비트단위 작업의 경우 표현되는 범위의 차이에 따라 형식을 나눈 것으로 보인다.

 wchar\_t는 확장 문자 또는 멀티바이트 문자 형식이며 일반적인 1바이트 문자보다 큰 유니코드 문자를 표현하기 위해 사용된다. 문자열 앞에 L 접두사를 붙여 wchar\_t임을 명시하는 방법이 있다. char\*\_t은 utf-\* 문자 표현에 사용된다. 여기서 \*는 8, 16, 32의 숫자를 의미한다.

 다음은 기본 자료형의 데이터 범위와 바이트 크기를 정리해 놓은 것이다.

|  |  |  |
| --- | --- | --- |
| 이름 | 범위 | 바이트 크기 |
| int | –2,147,483,648 ~ 2,147,483,647 | 4 |
| unsigned int | 0 ~ 4,294,967,295 | 4 |
| bool | false 또는 true | 1 |
| char | -128 ~ 127 | 1 |
| signed char | -128 ~ 127 | 1 |
| unsigned char | 0 ~ 255 | 1 |
| short | –32,768 ~ 32,767 | 2 |
| unsigned short | 0 ~ 65,535 | 2 |
| long | –2,147,483,648 ~ 2,147,483,647 | 4 |
| unsigned long | 0 ~ 4,294,967,295 | 4 |
| long long | –9,223,372,036,854,775,808 ~ 9,223,372,036,854,775,807 | 8 |
| unsigned long long | 0 ~ 18,446,744,073,709,551,615 | 8 |
| enum |  | 상황마다 다름 |
| float | 3.4E+/-38(7개의 자릿수) | 4 |
| double | 1.7E+/-308(15개의 자릿수) | 8 |
| long double | double과 동일 | double과 동일 |
| wchar\_t | 0 ~ 65,535 | 2 |



* void 는 없다라는 걸 형식적으로 알려준다

function() 에서 매개변수로 받는 인자가 생략 된건지 받지 않는지를 식별가능하게

function(void) 매개변수 없음!

* 유도형 : \*포인터 / [ ] 배열 / 구조체 / 공용체 (동일메모리에 다양한 해석방법부여)

# https://gerrk.tistory.com/3?category=901417

**부동 소수점**

실수 : 오차를 가지고 있음

오차의 이유? 근사값 처리 3.4 정수처리 하면 3

float 소수점 이하 6자리 / float 에서 정수 사이 1 ~ 2 사이는 128만큼의 단위로 수로 나눔

double 소수점 이하 15자리 / float 보단 double 사용 why? 오차범위 인식크기 차이

long double double 이상

문자열 = 상수화된 문자 ‘배열’

int main(void)

{

    printf("%f\n", 120.45F); // float 소숫점 6번짜 자리까지만 출력 이하는 유효값 X

    printf("%f\n", 120.45);  // double

    // 120.449997

    // 120.450000

    return 0;

}

**형 변환 (Type promotion)**

서로 다른 타입의 변수를 연산할 때 작은 타입의 변수가 피연산자의 타입으로 형 변환되어 연산 된다.

a(char type) + 24(int) = a(int type으로 변환(아스키코드)) a= **97?** 그정도 + 24 가되어 121이됨

**형식 문자 (Format Character)**

데이터를 입/출력 할 때, 컴퓨터가 그 Type을 인식할 수 있도록, 데이터의 해석을 지시해주는 문자이다. 우리가 흔히 사용하는 scanf와 printf에서 주로 사용되며, %c, %d, %f 등과 같은 문자를 의미한다. 형식 문자가 사용되는 이유는, 같은 데이터라도 해석하는 방향에 따라 다른 문자가 될 수 있기 때문이다. 예로, 'a'라는 알파벳 소문자 데이터는 단일 character로 인식하는 경우(%c) a를 출력하지만, 정수로 인식하는 경우(%d) 97을 출력할 수 있다. 따라서 C 언어에서는 다양한 형식 문자를 지원한다.

- %c : char, short, int, 단일 바이트 문자 입/출력에 사용.

- %d, %i : char, short, int, 부호 있는 10진수 정수.

- %lld : long long, 부호 있는 10진수 정수.

- %u : unsinged int, 부호 없는 10진수 정수.

- %o : unsinged int, 부호 없는 8진수 정수.

- %x(%X) : unsinged int, 부호 없는 16진수 정수.(x는 a,b,c,d,e,f, X는 A,B,C,D,E,F를 사용)

- %f : float, double, 부호 있는 10진수 값을 표현하며, 부동소수점 형식으로 사용. 기본적으로 소수점 6자리까지 표현.

- %lf : long double, 부호 있는 10진수 값을 표현하며, 부동소수점 형식으로 사용. 기본적으로 소수점 6자리까지 표현.

- %e(%E) : float, double, 부호 있는 10진수 값을 표현하며, e, E를 이용한 지수형으로 표현.

- %g(%G) : float, double, 부호 있는 10진수 값을 표현하며, 기본적으로 %f와 동일하나, 필요시 %e와 동일.

- %s : 문자열, 단일 바이트 문자의 연속(array of character).

- %p : 포인터 유형, 인수를 16진수로 된 주소로 표시.

참조 : <https://msdn.microsoft.com/ko-kr/library/hf4y5e3w.aspx>

**이스케이프 시퀀스 (Escape sequence)**

프로그래밍 언어 특성상 표현할 수 없는 기능, 문자를 표현해준다.  
컴퓨터를 제어하는 목적으로 사용되는 특수한 문자이다.  
제어 시퀀스(control sequence), 이스케이프 문자, 확장 비트열 이라고도 한다.  
\와 ₩는 같은 의미이다.

| 이스케이프 시퀀스 | 의미 |
| --- | --- |
| \a | 경고(alert) |
| \n | 줄바꿈(new line) |
| \t | 수평 탭(horizontal tab) |
| \v | 수직 탭(vertical tab) |
| \b | 백스페이스(backspace) |
| \f | 폼 피드(form feed) |
| \r | 캐리지 리턴(carriage return) |
| \\ | 백슬래시( \, backslash) |
| \’ | 작음따옴표( ‘, single quotes) |
| \” | 큰따옴표( “, double quotes) |
| \ooo | 8진수 숫자를 사용하여 ASCII 코드의 문자 표현 |
| \xhh | 16진수 숫자를 사용하여 ASCII 코드의 문자 표현 |

위의 형식문자들을 이용해 출력 형식에 옵션을 줄 수도 있다. %10s와 같이 숫자를 적는 경우, 출력에 있어서 숫자로 기입한 만큼의 공간을 할당해 준다. 정수와 실수의 출력에 있어 %05d와 같은 형식은 출력될 공간을 할당해주고, 남은 부분을 0으로 채워준다.  또한 할당받은 공간에서 기본적으로 오른쪽 정렬이나, -옵션을 통해서 왼쪽정렬로 변경할 수 있다. 예로 %-5d와 같은 경우는, 5칸의 공간에 정수를 왼쪽 정렬해 출력하겠다는 의미이다. 실수의 경우에는 소숫점 아래 몇자리 까지 표현할 것인지 %.2f등과 같이 표현할 수 있다.

#include<stdio.h>

int main(void){

    int integer=12;

    long long long\_int=-150;

    float fraction=0.0000000000392;

    char character='a';

    char \*str="HELLO WORLD!";

    printf("\

           Int : %d\n\

           long long : %lld\n\

           OCT : %o\n\

           HEX : %X\n\

           float : %f,%e,%G\n\

           char : %c\n\

           str : %s\n\

           ",integer,long\_int,integer,integer,fraction,fraction,fraction,character,str);

    return 0;

}

변수

값이 정해지지 않은 수 (컴퓨터 입장에서)

자료형은 (int ,char 등) 변수에 선언된 메모리값을 해석하는 방법 00FF420A 등을 어떻게 해석? int로? char로? double 로?

extern

    extern int tData;

    // ~~~.c에서 이미 선언 되어있는 변수 tData를 전역변수로 선언해서 접근할수있다

상수

값이 확정된 수 (컴퓨터 입장에서)

# (컴파일)전 처리기 Free processor

#include, #define …

헤더의 선언(~~.h 파일은 #pragma once 선언 꼭 필요)

컴파일러가 소스코드를 컴파일하기에 앞서 선행 처리하는 구문

* 코드변경
* 컴파일 타임에만 일어남 cpu연산과 상관없다

(꺽쇠)브라켓 과 < > 따옴표는 다르다 “ “

<stdio.h>

* 검색 -> 컴파일러 설정

“~~~.h“

* 현재폴더에서 검색

#if, #ifdef #else #endif

전처리기 지시어

-value는 메모리 위치를 나타내는 식이며 할당 연산자의 왼쪽에 나타날 수 있습니다. l-value는 참조하는 개체의 값을 변경하는 데 사용할 수 있습니다. l-value의 예로는 변수, 참조 및 배열이 있습니다.

대로 r-값은 메모리 위치가 아닌 값을 참조하는 표현식입니다. r-값은 할당 연산자의 오른쪽에 나타날 수 있지만 참조하는 개체의 값을 변경하는 데 사용할 수는 없습니다. r-값의 예로는 리터럴, 상수 및 값을 계산하는 표현식이 있습니다.

C++에서는 할당과 같은 특정 작업의 왼쪽에 l-value가 필요하기 때문에 l-value와 r-value를 구별하는 것이 중요합니다. 또한 일부 함수 및 연산자는 r-값만 인수로 사용할 수 있습니다. 예를 들어 인수를 값으로 받는 함수는 r-값만 받을 수 있는 반면, 참조로 인수를 받는 함수는 l-value나 r-value를 받을 수 있습니다.

**표준 입출력**

입출력(I/O) 주체는 kernel

시스템 프로그래밍 socket/ File I/O / TCP protocol 공부할것

buffer 의개념

getchar()

gets() 보안오류 overflow / get\_s() 사용

scanf()

fflush() 버퍼를 비우는 함수

ASLR(address Space Layout Randomization)

* **스택, 힙, 라이브러리 등의 주소를 랜덤한 영역에 배치하여, 공격에 필요한 target address를 예측하기 어렵게 만든다**.

**함수의 정의**

호출자함수(callen)가 다른 피호출자함수(callee)를 불러오려면 피호출자 함수의 **매개변수(인수 또는 인자라고도 사용)**에 초기값을 기술해야 한다. (실인수)

int(반환형) main(함수이름) (매개변수)

항 > 구문 > 함수 > 프로그램

{ { } Block scope (변위)

string 문자(배)열

Auto 형식의 부모의 형식을 따라간다 (c++)

ex) int a = 10;

auto b = a; (b의 자료형식은 int)

선언 및 정의

int a = 10; ( C 형식 )

int a(10); = ( C++형식 ) 두개의 차이점은 없다.

Namespace (소속)

메모리 동적할당

C : malloc(size\_t); 할당 인수 자료형 size\_t(양의 정수형태의 크기)

* 주소만 반환 용도상관안함

free(); 해제

**연산자**

결합성 : 연산자 우선 순위가 같을 시 왼쪽에서 오른쪽으로 연산

산술 연산자 : + - \* / **(나눗셈은 주의해야 할것들이 많음 형변환/무한루프)**

컴퓨터에서 나눗셈의 방식 7 / 2 의 예시 (int 형변환으로 나머지 이하는 절사 3이됨)

7 – 2 = 5 (몫1)

5 – 2 = 3 (몫2)

3 – 2 = 1(몫3)

1 (나머지 값)

7 / 0 는 루한루프가 됨

7 / 2 의 계산을 할 때 의 대상이 값이 크면

나머지 연산자 : % ( A % B 의 몫을 구하고 나서 나머지값 C를 구한다.)

단항 증감 연산사

++ / --

비트 연산자

마스크 연산 \*비트 연산자가 활용되는 대표적인 예

shist << , >>

0x34 << 3 : 오른쪽에 000을 채움 (왼쪽으로 3칸이동)

0x47 >> 5 왼쪽끝에 있는 수에 맞춰 왼쪽에 채움 오른쪽 5는 절사

typeof() 연산 (자료형 구분 ) – compiler type

인자를 자료형으로 구분하고 자료형의 값을 정수로 반환

sizeof(1) = 4;

상등(==)/부등(>,<)연산에서 실수는 사용하지 않는다 (오차값이 생기므로)

논리 연산자

다항 연산자 = &&(and) ||(or)

단한 연산자 = ! (true)

삼항 연산자 ndata = (조건) ? A : B

조건이 참이면 ndata = A / 거짓이면 ndata = B

**제어문**

조건문

if / if else / switch

반복분

for / while / do while

리팩토링? s/w기능 / 개발자가 코드를 최적화하여 성능을 높이는 것 (결과의 변경없이 코드 가독성/유지보수등을 구조를 재조정)

Lookup-table : 성능향상을 위한 배열

goto문

continue / break;

가급적 continue 사용 자제 / 소스코드의 복잡성 증가 knot

C++ /

범위지정 for

for (auto : n(변수), 배열)

pirnf(“%d:, n) (읽기만가능 / 전체요소 접근)

배열의 형식에 맞게 자동으로 반복한다

참조자 사용하면 쓰기가능

auto : &n

n= 0

**배열**

배열은 상대적 주소

문자(배)열 + 가변길이

배열 = 고정길이

다차원 배열 (배열의 배열)

char sting[행][열]

char sting[3][4]

ㅁㅁㅁㅁ / ㅁㅁㅁㅁ / ㅁㅁㅁㅁ

char [4] 배열 이 3개있음

최소값 최대값 / 정렬 / 교차 / 알고리즘 공부

선택 정렬 선택한 기준점을 정하고 비교대상과 위치를 교환 – 자리교환을 최소화한다

버블 정렬 – 자리교환을 매번한다

quick sort 알고리즘이 빠르다

**포인터**

int \* pnData

\*pnData (간접지정) pnData가 변수이기때문에

배열 연산

    // 배열

    int aList[5] = {40, 50, 30, 20, 70};

    // \*paList 주소에 aList 주소를 입력

    // 메모리 주소를 저장한다 0x0023FFA0 등등

    int \*paList = aList; // 배열의 이름은 0번 요소에 부여한 식별자

                         //\*paList = &aList[0] 같은의미

    // paList 에 들어있는 메모리 주소의 자료형에 따라 1칸만큼 옆에있는 메모리주소에 5를 대입

    //  aList[1] = 5; 같은의미

    \*(paList + 1) = 5;

    for (int i = 0; i < 5; i++)

    {

        // 출력하면 40, 5, 30, 20, 70이 나온다

        printf("%d\t", &aList[i]);

    }

    printf("\n%d\t %d", &aList[2], paList);

    // 포인터는 메모리 주소를 변수화해서 저장

    // \*paList = aList    ==    aList 배열의 메모리 주소를 포인터변수 \*Palist에 저장

    // 변수 앞에 &val == 변수의 메모리 주소를 가리킴

    // 변수 앞에 \*val (선언이 아닐 떄) 간접지정자 포인터 변수에 들어있는 값 접근

    //  피연산자는 포인터 형식이어야 합니다. 연산 결과는 피연산자가 가리키는 주소에 있는 값입니다

    // 12바이트 (인트형 \* 3) 메모리를 동적할당하고 그 메모리 시작주소를 nData에 저장

    int \*nData = (int \*)malloc(sizeof(int) \* 3);

    \*nData = 0;

    printf("\nnData : %d" , \*nData);

    // 동적할당 받은 메모리를 반환(해제)한다.

    free(nData);

    return 0;

// 포인터는 메모리 주소를 변수화해서 저장 선언할 때 변수 앞에 \*(자료형 \*변수;

// \*paList = aList    ==    aList 배열의 메모리 주소를 포인터변수 \*Palist에 저장

// 변수 앞에 &val == 변수의 메모리 주소를 가리킴

// 변수 앞에 \*val (선언이 아닐 떄) 간접지정자 안의 값을 가리킴

// 변수가 포인터 형식인데 변수처럼 쓰일 경우 자료형의 크기만큼 인덱스 처리

// paList + 2 == paList 배열의 메모리주소의 인덱스를 +2만큼 이동한 주소를가리킴

//  피연산자는 포인터 형식이어야 합니다. 연산 결과는 피연산자가 가리키는 주소에 있는 값입니다 \*paList[0] == 배열 첫번째의 값에 접근

단항 간접 참조 연산자(\*)는 포인터를 통해 값에 간접적으로 액세스합니다.

피연산자는 포인터 형식이어야 합니다. 연산 결과는 피연산자가 가리키는 주소에 있는 값입니다.

결과의 형식은 피연산자가 표시하는 형식과 동일합니다.

    //nData2의 메모리주소가 0x0018FF28일때 밑에 두개는 같은 의미이다

    int nData2 = 10;

    \*((int\*)0x0018FF28) = 0;

**C++ /**

참조자 (일종의 별명)

참조자를 사용하는 이유??

포인터의 문제점부터 파악 :

**메모리 종류**

Stack – 1mb **(정적 할당)**

* scope 영역

Heap – 크기제한x 자유영역 **(동적 할당)**

* scope를 뺀 나머지 다

// 별개의 구문

PE – text 기계영역

image – data 영역?

* 읽기쓰기o 전역변수(정적) data
* 읽기만o 문자열 상수 data

**메모리 동적 할당**

**<stdlib.h>**

대량의 메모리 사용을 하기위해 OS한테 공간(쓸수있는 공간이 있을 시)을 허가 받는다. ( 동적 할당 시 해제를 꼭 해줘야 한다.)

malloc(); 동적할당 함수

* 주소 void \* 형식으로 넘어온다

free(); 반환 함수

* 반환을 안하면? 메모리 유실 / 누수등 오류발생

**C++ /**

new 변수이름 자료형; - 메모리 할당 / 크기는 자료형에 맞게

delete 변수이름;

int aNewData = new int[5];

delete [] aNewData; // 형식에 맞게 메모리 해제

    // 12바이트 (인트형 \* 3) 메모리를 동적할당하고 그 메모리 시작주소를 nData에 저장

    int \*nData = (int \*)malloc(sizeof(int) \* 3);

    // 동적할당 받은 메모리를 반환(해제)한다.

    free(nData);

**메모리 초기화**

**<string.h>**

memset(\*변수이름, 초기화값, 동적할당받은 량)

동적할당한 크기를 알아내는 함수 \_memset(\*변수)

**메모리 복사**

**<string.h>**

메모리 주소 복사 (shallow copy) / 내용을 복사(Deep copy)하는 건 다르다

deep copy = memcpy(타겟변수, 소스변수, 복사할 메모리 량); / 딥카피 하는건 중요

shallow copy = 배열변수 = 배열변수 == ? 주소만 복사하는 것

**메모리 비교**

memcmp(비교 원본 메모리 주소 , 비교 대상 메모리 주소, 비교할 메모리의 바이트 크기)

문자열 비교

strcmp

**메모리 재 할당**

realoc(), sprint()

**다중 포인터**

int naList[3] = {0}; / 해석순서

1. NaList 배열은
2. 인트형의 자료를 가진 배열이고
3. 값은 0으로 초기화한다.

int \*npList = npList; / 해석순서

1. nNumList 포인터이고
2. 가리키는 대상은 int type 이고
3. 대상은 배열이다

int\*\* \*ppList = 0; / 해석순서

1. pplist는 포인터이고
2. 가리키는 대상은 인트형 포인터로 (why? 자료형에 포인터를 쓰면 지정주소를 int로 인스턴스화 한다)

**함수**

재귀 호출 = 반복문 + 스택

인라인함수

\_\_inline

함수형식 (배열처럼 포인터접근가능 / 배열형태의 한 부분)

* 반환형식 호출규칙 이름 ( 매개변수 )
* 컴파일러가 알아서 인라인함수로지정 지정된인라인함수는 Func call 안함 / 안하면 스택상승 x / 성능상승

**함수 포인터 역 호출 (callback)**

qsort

함수 주소를 등록한다 / 등록을 받은쪽에서 call을한다 / 직접 사용x

**구조체 / 공용체**

배열 = 동일 형식의 연속된 ‘집합체’

구조체 = 서로 같거나 다른 형식들의 연속된 ‘집합체’

공용체 = 한 형식(같은 크기)의 여러 해석방법을 부여하는 문법

파일 입/출력

streaming > buffer > queue

**컴파일러 최적화**

문법 코드에서 군살을 제외 (논리적으로 당연한 것은 연산을 하지않는다)

특정 변수(자료)에 대해 의존성이 존재하는 연산들을 구별

변수가 많을수록 / 변수 값의 범위 / 경우의 수가 많아 논리적 구조가 복잡해진다 최적화하기 어려움

컴파일러가 최적화 하기 좋게 문법쓰기

최적화 방해요소

* 변수의 개수
* 포인터 사용 주의

하드코딩 금지

심볼릭 상수 (자주)사용 const / #define / 적재적소에 알아 볼 수 있도록

형한정어

* const : 상수화
* volatile : 변수와 관련된 모든 연산에 대해 컴파일러가 최적화 규칙을 적용하지 않는다 / const와는 정 반대
* typedef : 형식 재선언 ( typedef 원형식 새형식 변수이름

형식 재선언

열거형 상수 (enum)

* 여러 개의 심볼릭 상수를 한 번에 정의하고 구조체처럼 사용 중요

**매크로**

**클래스**

구조체 + 멤버함수 //

접근제어 – public / private

class TEST\_CLASS

{

private:

    /\* data \*/

public:

    TEST\_CLASS(/\* args \*/);

    ~TEST\_CLASS();

};

TEST\_CLASS::TEST\_CLASS(/\* args \*/)

{

}

TEST\_CLASS::~TEST\_CLASS()

{

}

생성자 / 소멸자

* 호출시점 / 자동호출 정해진 / 문법에따라 / 객체가 언제 소멸? 지역변수의 소멸시점 (콜러)의스코프가 닫히는 시점
* 반환형식 X

변환 생성자 중요

this 포인터

**상속**

코드를 재사용하는 방법 / 규모확장 / 관계의 한 유형

**템플릿**