C++ 언어

# C++이란?

C++은 기존의 C언어에 객체 지향 프로그래밍의 클래스 개념만을 추가하여 만든 언어

C++은 C언어에서 절차 지향적 언어의 특징을 가져왔습니다.

또한, Simula에서 클래스를 사용하는 객체 지향적 언어의 특징을 가져왔습니다.

그와 동시에 템플릿으로 대변되는 일반화 프로그래밍 방식의 언어이기도 합니다.

C++은 C언어를 기초로 삼아 만든 언어이므로, 기존의 C 표준 라이브러리를 그대로 사용할 수 있습니다.

# C++의 역사

C++은 C언어가 개발된 벨 연구소의 비야네 스트롭스트룹(Bjarne Stroustrup)에 의해 개발됩니다.

# C++의 특징

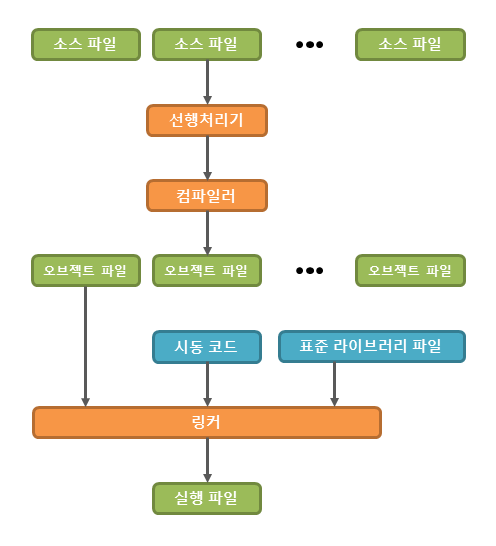
1. C++은 절차 지향적이며 구조적 프로그래밍 언어입니다.

2. C++은 객체 지향 프로그래밍 언어입니다.

3. C++은 일반화 프로그래밍 언어입니다.

# 프로그래밍

C++에서 소스 파일에서 실행 파일을 생성하는 순서



C++을 통해 작성된 소스 파일의 확장자는 대부분 .cpp 가 된다.

\* 선행처리

선행처리(preprocess)란 소스 파일 중에서도 선행처리 문자(#)로 시작하는 선행처리 지시문의 처리 작업을 의미합니다.

이러한 선행처리 작업은 선행처리기(preprocessor)에 의해 처리됩니다.

선행처리기는 코드를 생성하는 것이 아닌, 컴파일하기 전 컴파일러가 작업하기 좋도록 소스를 재구성해주는 역할만을 합니다.

\* 컴파일

컴퓨터는 0과 1로 이루어진 이진수로 작성된 기계어만을 이해할 수 있습니다.

소스 파일은 개발자에 의해 C++ 언어로 작성되기 때문에 컴퓨터가 그것을 바로 이해할 수는 없습니다. 따라서 소스 파일을 컴퓨터가 알아볼 수 있는 기계어로 변환시켜야 하는데, 그 작업을 컴파일(compile)이라고 합니다. 파일은 C/C++컴파일러에 의해 수행되며, 컴파일이 끝나 기계어로 변환된 파일을 오브젝트 파일(object file)이라고 합니다.

이러한 오브젝트 파일의 확장자는 .o 나 .obj 가 됩니다.

\* 링크

컴파일러에 의해 생성된 오브젝트 파일은 운영체제와의 인터페이스를 담당하는 시동 코드(start-up code)를 가지고 있지 않습니다.

또한, 대부분의 C++ 프로그램에서 사용하는 표준 라이브러리 파일도 가지고 있지 않습니다.

하나 이상의 오브젝트 파일과 라이브러리 파일, 시동 코드 등을 합쳐 하나의 파일로 만드는 작업을 링크(link)라고 합니다.

링크는 링커(linker)에 의해 수행되며, 링크가 끝나면 하나의 새로운 실행 파일이나 라이브러리 파일이 생성됩니다.

이처럼 여러 개의 소스 파일을 작성하여 최종적으로 링크를 통해 하나의 실행 파일로 만드는 것을 분할 컴파일이라고 합니다.

\* executable file

소스 파일은 선행처리기, 컴파일러 그리고 링커에 의해 위와 같은 과정을 거쳐 실행 파일로 변환됩니다.

최근 사용되는 개발 툴은 대부분 위에서 소개한 선행처리기, 컴파일러, 링커를 모두 내장하고 있으므로 소스 파일에서 한 번에 실행 파일을 생성할 수 있습니다.

이렇게 생성된 실행 파일의 확장자는 .exe 가 됩니다.

# C++ 프로그램

\* 구조

#include 문

#define 문

int main()

{

명령문;

return 문;

}

ex>

#include <iostream>

#define TEXT "Welcome to C++ Programming!!"

int main()

{

std::cout << TEXT;

return 0;

}

\* main() 함수

C++ 프로그램은 가장 먼저 main() 함수를 찾고, 그곳에서부터 실행을 시작합니다.

따라서 모든 C++ 프로그램은 반드시 하나의 main() 함수를 가지고 있어야 합니다.

만약 main() 함수를 발견하지 못하면 C++ 컴파일러는 오류를 발생시킬 것입니다.

\* 명령문

C++ 프로그램의 동작을 명시하고, 이러한 동작을 컴퓨터에 알려주는 데 사용되는 문장을 명령문(statement)이라고 합니다.

이러한 C++의 모든 명령문은 반드시 세미콜론(;)으로 끝나야 합니다.

실행 결과>

std::cout << "C++ 프로그래밍"; // 정상적으로 출력됨.

std::cout << "C++ 프로그래밍" // 오류가 발생함.

\* 반환문 (return)

반환문은 함수의 종료를 의미하며, 함수를 호출한 곳으로 결괏값을 반환하는 역할을 합니다.

특히 main() 함수가 반환되면, 프로그램 전체가 종료됩니다.

\* 선행처리문

#include 문과 #define 문은 모두 선행처리기에 의해 처리되는 선행처리문입니다.

#include 문은 외부에 선언된 함수나 상수 등을 사용하기 위해서 헤더 파일의 내용을 현재 파일에 포함할 때 사용합니다. C언어에서는 헤더 파일에 .h 확장자를 사용했지만, C++에서는 헤더 파일의 확장자를 사용하지 않기로 합니다. 따라서 기존 C언어 헤더 파일들의 이름 앞에 c를 추가하여 C++ 스타일의 헤더 파일로 변환하기도 합니다.

예제

#include <math.h> // C언어에서는 이 스타일만 허용됨.

#include <cmath> // C++에서는 이 스타일뿐만 아니라 위의 스타일도 사용할 수 있음.

물론 원칙은 위와 같이 바뀌었지만 C++ 프로그램에서는 기존의 C언어 스타일로도 헤더 파일을 사용할 수 있습니다. #define 문은 함수나 상수를 단순화해주는 매크로를 정의할 때 사용합니다.

\* 네임스페이스

네임스페이스란 이름이 기억되는 영역을 뜻하며, 이름이 소속된 공간을 의미합니다. 네임스페이스는 C++ 프로그램을 작성할 때 발생하는 이름에 대한 충돌을 방지해 주는 방법을 제공합니다. 이러한 네임스페이스는 C언어에는 없는 C++ 만의 새로운 기능입니다. C++ 프로그램의 표준 구성 요소인 클래스, 함수, 변수 등은 std라는 이름 공간에 저장되어 있습니다. 따라서 C++ 프로그램에서 표준 헤더 파일인 iostream 내의 정의를 사용하려면 다음과 같이 사용해야 합니다.

ex>

#include <iostream>

#define TEXT "Welcome to C++ Programming!!"

int main()

{

std::cout << TEXT;

return 0;

}

위의 예제처럼 std라는 네임스페이스에 있는 정의를 사용하려면, std:: 접두어를 붙여 해당 정의가 std라는 네임스페이스에 있다는 것을 컴파일러에 알려줘야 합니다.

이러한 네임스페이스에 속한 정의를 간단하게 사용하려면 다음과 같은 명령문을 추가하면 됩니다.

using namespace std; // std라는 네임스페이스에 속한 정의들은 네임스페이스 이름을 붙이지 않아도 사용할 수 있음.

앞서 살펴본 예제와 정확히 같은 동작을 수행합니다.

#include <iostream>

#define TEXT "Welcome to C++ Programming!!"

using namespace std;

int main()

{

cout << TEXT;

return 0;

}

# 주석

// C++ 한 줄 주석

/\* 여러 줄

// 두 줄 주석 안에 다른 한 줄 주석 삽입 가능 (단, 여러 줄 안에 여러 줄 주석 삽입 불가)

\*/

# iostream

\* 표준 입출력 클래스

C++에서는 cout 객체로 출력 작업을, cin 객체로 입력 작업을 수행하고 있습니다. 또한, C++에선 기존의 C언어 스타일처럼 printf() 나 scanf() 사용 가능!

\* cout 객체

cout 객체는 다양한 데이터를 출력하는 데 사용되는 C++에서 미리 정의된 출력 스트림을 나타내는 객체입니다.

문법 : std::cout << 출력데이터;

<<(삽입 연산자)

\* 소수점 설정

// 이 문장이 없으면 정수부 소수부 합친 길이 = 소수자리수

Cout << fixed; (소수점 아래 숫자의 출력범위만 설정)

Cout.precision (소수자리수); ->

\* cin 객체

cin 객체는 다양한 데이터를 입력받는 데 사용되는 C++에서 미리 정의된 입력 스트림을 나타내는 객체입니다.

문법 -> std::cin >> 저장할변수;

추출연산자(>>)를 통해 사용자가 입력한 데이터를 입력 스트림에서 추출하여, 오른쪽에 위치한 변수에 저장. 이때 cin 객체는 자동으로 사용자가 입력한 데이터를 오른쪽에 위치한 변수의 타입과 동일하게 변환시킨다.

cin >> a >> b >> c; 처럼 a b c 를 여러 값을 입력할 수 있다.

Ex>

#include <iostream>

using namespace std;

int main()

{

int age;

cout << "여러분의 나이를 입력해 주세요 : ";

cin >> age; // age에 string type 입력 시 0이 저장됨.

cout << "여러분의 나이는 " << age << "살 입니다." << endl;

return 0;

}

\* endl : 줄바꿈 문자

# 변수

변수란 데이터를 저장하기 위해 프로그램에 의해 이름을 할당 받은 메모리 공간

숫자표현 관련 변수 - 정수형, 실수형

정수형 : char형, int형, long형 long long형 변수

실수형 : float형, double형

\* bit & byte

비트(bit) : 컴퓨터가 데이터를 처리하기 위해 사용하는 데이터의 최소 단위

바이트(byte) : 위와 같은 비트가 8개 모여서 구성되며, 한 문자를 표현할 수 있는 최소 단위

1byte = 8bit

\* 변수와 메모리 주소

변수는 기본적으로 메모리 주소를 기억하는 역

메모리 주소 : 물리적인 메모리 공간을 서로 구분하기 위해 사용되는 일종의 식별자

변수를 참조할 때는 메모리의 주소를 참조하는 것이 아닌, 해당 주소에 저장된 데이터를 참조하게 된다. 따라서 변수는 데이터가 저장된 메모리의 주소 + 저장된 데이터의 길이와 형태에 관한 정보도 같이 기억해야 한다.

\* 변수 선언 방법

변수 선언만 or 변수 선언후 초기화

Int num; / int num = 20;

반드시 선언한 변수의 타입과 일치하는 타입의 데이터만 할당해야 함.

Int num1, num2; / double num3 = 3.14, num4 = 4.56;

# 상수 (constant)

상수 : 변수와 다르게 값을 변경할 수 없는 데이터를 저장하는 메모리 공간

\* literal constant (리터럴 상수)

데이터가 저장된 메모리 공간을 가리키는 이름을 가지고 있지 않다.

적절한 메모리 공간을 할당받기 위해 변수든 상수든 타입을 가지고 있다.

1. **정수형 리터럴 상수**

아라비아 숫자와 부호로 직접 표현 (8,10,16진수로도 표현 가능

Cout객체는 dec(10진수), hex, oct 조정자를 제공함.

이 세 가지 조정자를 cout 객체에 전달하면 변경 전까지 진법의 형태를 유지 가능

int a = 10;

cout << "숫자 10을 10진수로 표현하면 " << a << "이며, " << endl;

cout << oct;

cout << "숫자 10을 8진수로 표현하면 " << a;

\* 정수형 리터럴 상수는 두 가지 경우를 제외하고 모두 int형으로 저장됨.

1. 데이터 값이 너무 커서 int로 저장할 수 없는 경우

2. 정수형 상수에 접미사를 사용하여, 해당 타입을 직접 명시하는 경우

두 가지 경우를 제외하고 모두 int형으로 저장됨.

-> 정수형 상수

|  |  |
| --- | --- |
| 기본 설정 | (signed) int형 |
| u 또는 U | unsigned int형 |
| l 또는 L | (signed) long형 |
| ul 또는 uL 또는 Ul 또는 UL | unsigned long형 |
| long long 또는 u11 또는 U11 또는 uLL 또는 ULL | unsigned long long형 (C++11부터 제공) |

2. **실수형 리터럴 상수**

실수형 리터럴 상수는 모두 부동 소수점 방식으로 저장됨.

|  |  |
| --- | --- |
| **접미사** | **타입** |
| f 또는 F | float형 |
| 기본 설정 | double형 |
| l 또는 L | long double형 |

3. **포인터 리터럴 상수**

널 포인터 : 아무것도 가리키고 있지 않은 포인터

nullptr 키우드를 사용한 리터럴 상수의 타입 = 포인터 타입 ( 정수형 변환 x)

nullptr == 0 (true) // 0을 사용해 명시 가능

4. **이진 리터럴 상수**

Ex> auto a = 0B010111;

5. **심볼릭 상수**

선언과 동시에 반드시 초기화 해줘야함!

매크로 이용(C언어의 문법) or const 키워드 사용!  
Const int ages = 30;

상수의 타입 명시적 지정 가능 / 구조체와 같은 복잡한 사용자 정의 타입에도 사용 가능

심볼릭 상수를 특정 함수나 파일에서만 사용할 수 있도 제한 가능

# 기본 타입

- 정수형 / 실수형 / 문자형 / bool형

1. 정수형 : 부호를 가지고 있으며, 소수 부분이 없는 수

Unsigned 키워드 추가 시 부호를 나타내는 최상위 비트 MSB까지도 크기를 나타내는 데 사용 가능

Unsigned 정수로는 음의 정수 표현 x , but 0을 포함한 양의 정수는 두 배 더 많이 표현 가능

데이터 타입 : Short (2 byte) / int (4 byte) / long (4 byte) / long long (8 byte)

\* 보편적으로 int 자주 사용

Ex> unsigned int (0~2^32 –1) ) vs (signed) int (-2^31 ~ 2^31 –1 )

정수형 데이터 타입 사용 시 데이터의 최대 크기 고려!

데이터가 타입의 크기를 벗어남 -> overflow (MSB를 벗어난 데이터가 인접 비트를 덮어씀- > 다른결과)

2. 실수형

float (4 byte) : (3.4 x 10^-38) ~ (3.4 x 10^38)

지수 8bit 가수 23bit : [소수점 6자리까지만 정확]

double (8 byte) : (1,7 x 10&-308) ~ (1,7 x 10^308) -> 가장 많이 사용

지수 11bit 가수 52bit : [소수점 15자리까지만 정확]

long double (8 byte) : double과 동일

3. 문자형 (아스키 코드)

아스키코드 : 영문대소문자를 사용하는 7bit 문자 인코딩 방식

Char (1 byte) : 2^-7 ~ 2^7-1

4. bool 형

0 : false 0이 아닌 값 : true [C와 동일]

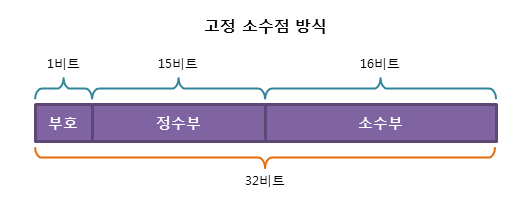
\* auto 키워드 선언

JS의 var과 유사 - 컴파일러가 알아서 데이터를 보고 변수의 타입을 지정해줌.

# 부동 소수점 수

\* 실수의 표현 방식

1. 고정 소수점 방식



정수부 소수부의 자릿수 ↓ -> 표현 범위 매우 작음

2. 부동 소수점 방식

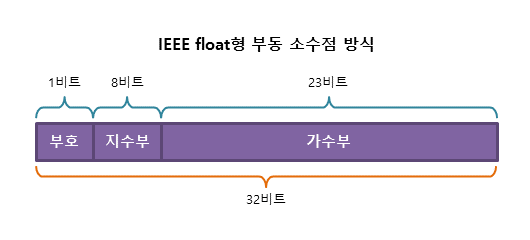
실수를 가수부, 지수부로 나누는 방식

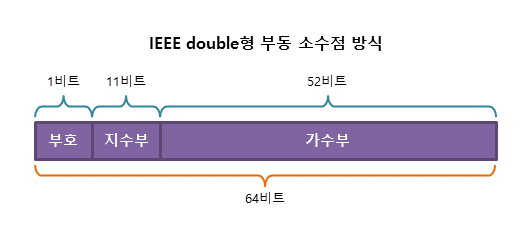
구조> ±(1.가수부) x 2^(지수부-127)

표현 방식> 소수E+16

Ex> -3.14E+16

\* IEEE 부동 소수점 방식





\* 부동 소수점 방식의 오차

부동 소수점 공식에 의해 항상 오차 발생, 실수 표현은 언제나 근사치

# 타입 변환

다른 타입끼리의 연산은 피연산자들을 같은 타입으로 만든 후 수행

\* 자동 타입 변환

대입 연산이나 산술 연산에서 컴파일러가 자동으로 수행해주는 타입 변환

- 대입 연산 : 할당된 데이터가 변수 앞에 선언해준 데이터 타입으로 자동 변환됨.

Int num = 8.3E16; -> int 범위 내의 수가 아니므로 이상한 값 출력

Int num = 3.14 -> num = 3;

- 산술 연산 : 데이터 손실이 최소화되는 방향으로 자동 변환 (범위가 큰 타입으로 변환)

double rst = 5 + 3.14; -> 5가 5.0으로 자동 변환되어 계산 => 8.14

다음과 같은 방향으로 자동타입변환이 이루어짐.

char형 → short형 → int형 → long형 → float형 → double형 → long double형

산술 연산 시 bool형 데이터인 true는 1로, false는 0으로 자동 타입 변환됩니다.

\* 강제 타입 변환

타입 캐스트(cast) 연산자를 사용하여 강제적으로 변환

문법>

(변환할 타입) 변환할 데이터 // 변환할 타입 (변환할 데이터) [ C++에서만 사용 가능한 기능]

-> 이 괄호를 타입캐스트연산자라고 지칭함!

int num1 = 1;

int num2 = 4;

double result1 = num1 / num2; // 0 출력 (int / int이므로)

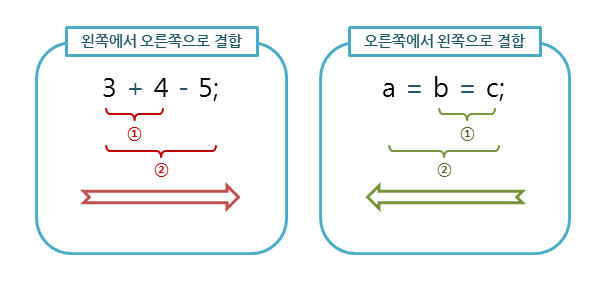
double result2 = (double) num1 / num2; // 0.25 출력 (1.0 / 4 = 0.25 [자동타입변환])

double result3 = double (num1) / num2; // 위와 마찬가지

# 산술 연산자

+ - \* / % (나머지)

\* 우선순위와 결합 방향



# 증감 연산자

전위 증가 연산자 : ++x (x를 먼저 1 증가시킨 후 연산 진행)

후위 증가 연산자 : x++ (연산 수행 후 x를 1증가)

# 논리 연산자

참 거짓 판별

AND ( && ) / OR ( || ) / NOT ( ! )

# 비트 논리 연산자

비트 단위 논리 연산 / 왼or오른 쪽으로 전체 비트 이동 / 1의 보수 만들기

& (AND) / | (OR) / ^ (XOR : exclusiveOR ) : 둘이 달라야 1 반환 / ~ (NOT)

<< n (비트 왼쪽 이동 = x 2^n) / >> (부호 유지 and 오른쪽 이동)

# 삼항 연산자

조건식 ? 반환값(참) : 반환값(거짓)

# sizeof 연산자

Sizeof (피연산자) : 피연산자의 크기를 바이트 단위로 반환

피연산자에 변수나 상수가 전달될 때 해당하는 타입의 크기 반환

Ex> sizeof(short) => 2 / sizeof(long) => 4

# C++ 만의 연산자

1. 범위 지정 연산자 ( :: )

여러 범위에서 사용된 식별자를 식별하고 구분하는데 사용

범위 지정 연산자를 변수의 이름 앞에 붙이면 해당 변수는 전역변수로 명시하는 것.

클래스에 이 연산자를 사용하면 네임스페이스 멤버를 식별하거나, 클래스의 정적 멤버를 호출할 수 있습니다.

Ex> std::cout << “hello”;

- ::식별자

- 클래스이름::식별자

- 네임스페이스::식별자

- 열거체::식별자

2. 멤버 포인터 연산자

클래스의 멤버를 가리키는 포인터

클래스타입의객체 : 객체.\*멤버이름

클래스타입객체의포인터 : 포인터->\*멤버이름

3. typeid(표현식)

객체 타입의 관한 정보 확인 가능

# 조건문

1. If / else if / else 문

\* indent : 쓰기를 통해 코드의 가독성을 높이는 것

2. switch 문

if / else 문보다 가독성이 더 좋으며, 컴파일러가 최적화를 쉽게 할 수 있어 속도가 빠른 편.

switch 문의 조건 값으로는 int형으로 승격할 수 있는(integer promotion) 값만이 사용될 수 있다.

- char형, short형, int형 변수 or 리터럴, 열거

구조>

switch (조건 값)

{

case 값1:

조건 값이 값1일 때 실행하고자 하는 명령문;

break;

case 값1:

조건 값이 값2일 때 실행하고자 하는 명령문;

break;

...

default:

조건 값이 어떠한 case 절에도 해당하지 않을 때 실행하고자 하는 명령문;

있어도 되고 없어도 됨

break;

}

\* 반드시 break 키워드를 포함하고 있어야 함.

\* break가 없다면 다음 케이스의 경우까지 포함하여 연산함.

Ex>

case 'a':

case 'A':

cout << "이 학생의 학점은 A입니다." << endl;

break;

위 경우 a 또는 A인 경우에 대해 문자열이 출력됨.

# 반복문

1. While

구조>

While (조건식) {

조건식의 결과가 참인 동안 반복적으로 실행하고자 하는 명령문;

}

1. Do / while

do {

조건식의 결과가 참인 동안 반복적으로 실행하고자 하는 명령문;

} while (조건식);

먼저 루프를 한 번 실행한 후에 조건식을 검사한다.

1. For 문

for (초기식; 조건식; 증감식)

{

조건식의 결과가 참인 동안 반복적으로 실행하고자 하는 명령문;

}

for (int i=0; i < num; i++) {

반복문;

}

1. 범위 기반의 for 문

C++11부터는 범위 기반의 for 문이라는 새로운 형태의 반복문이 추가됨.

범위 기반의 for 문은 표현식 안에 포함되어 있는 모든 값에 대해 한 번씩 루프를 실행해 줍니다.

이러한 범위 기반의 for 문은 배열을 자동으로 인식하며, 컨테이너 클래스에서 많이 사용됩니다.

Ex>

int arr[5] = {1, 3, 5, 7, 9};

for (int element : arr)

{

    cout << element << " ";

}

# continue, break, goto

Continue -> 해당 루프의 나머지 부분을 건너 뛰고 바로 다음 조건식의 판단으로 넘어감

Ex>

int except\_num = 2

for (int i = 0; i <= 100; i++)

{

    if (i % except\_num == 0)

    {

        continue;

    }

    cout << i << " ";

}

Break -> 해당 반복문을 완전히 종료시킨 뒤, 반복문 바로 다음에 위치한 명령 실행.

nt num = 1, sum = 0;  
int end\_num = 10;

while (1) // 무한 루프

{

    sum += num;

    if (num == end\_num)

    {

        break;

    }

    num++;

}

cout << "1부터 " << end\_num << "까지 더한 값은 " << sum << "입니다.";

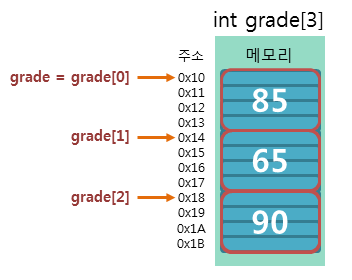
Goto -> 프로그램의 흐름을 지정된 레이블로 무조건 변경시키는 명령문 ( 단점이 많아 디버깅 용으로만 사용 )

# 배열

같은 타입의 변수들로 이루어진 유한 집합

\* 1차원 배열

배열 선언> 타입 배열이름[배열길이];

선언만 하고 초기화하지 않으면 모든 배열 요소가 trash element로 채워짐.

Ex>

int sum = 0;

int grade[3]; // 길이가 3인 int형 배열 선언

// 1. 인덱스를 이용한 배열의 초기화

grade[0] = 85; // 국어 점수

grade[1] = 65; // 영어 점수

grade[2] = 90; // 수학 점수

\* 배열의 선언과 동시에 초기화 하는 법

구조> 타입 배열이름[배열길이] = {배열요소1, 2, …};

(단, 초기화 리스트의 타입과 배열의 타입은 반드시 일치해야 함)

초기화 리스트의 개수가 배열의 총 길이보다 적으면, 배열의 앞에서부터 차례대로 초기화하여 초기화되지 못한 나머지 배열 요소는 모두 0으로 초기화.

배열이 먼저 선언된 후에는 이 방식으로 배열의 요소를 초기화할 수 없다.

Int age[3];

Age[3] = {1,2,3} // 불가능!!

\* 배열의 길이 자동 설정

타입 배열이름[] = {배열요소1, 배열요소2, ...};

\* 배열이 차지하는 메모리의 크기

배열을 복사하거나 배열 요소에 특정 작업을 하고 싶을 때는 해당 배열이 차지하는 메모리의 크기를 정확히 알고 있는 것이 좋다.

배열이 차지하는 메모리의 크기 = 배열의 길이 \* sizeof(타입)

배열의 길이 : 배열의 길이 = sizeof(배열 이름) / sizeof(배열 이름[0])

int len = sizeof(arr) / sizeof(arr[0])

길이 = 메모리 byte / 타입 byte

\* 배열 초기화

1. 배열을 초기화할 때에 대입 연산자(=)를 사용하지 않아도 됩니다.

2. 값을 명시하지 않고 괄호({})만을 사용하여 초기화하면, 모든 배열 요소를 0으로 초기화할 수 있습니다.

3. 초기화 리스트를 사용하여 배열을 초기화할 경우에는 narrowing cast를 할 수 없습니다.

4. array 템플릿 클래스가 추가되었습니다.

\* narrowing cast

초기화를 통해 발생하는 암시적인 데이터 손실 ex> int var = 3.14; // narrowing cast

C++에서는 초기화 리스트를 사용하여 변수나 배열을 초기화할 경우에 narrowing cast를 허용하지 않는다 초기화 리스트를 이용한 초기화에서 narrowing cast가 발생하면, 경고(warning)를 발생시킴.

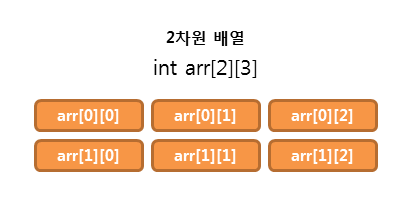
Ex> int var = {3.14}; // 경고 발생!

# 다차원 배열

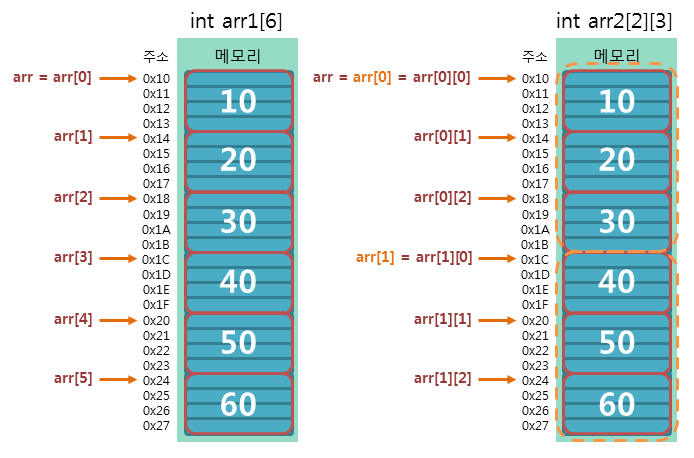
배열 요소로 또 다른 배열을 가지는 배열.

구조> 타입 배열이름[행의길이][열의길이];

ex>



실제로는 다음과 같이 저장됨.



\* 2차원 배열의 선언과 동시 초기화

1. **1차원 배열의 초기화 형태를 따르는 방식**

구조>

타입 배열이름[행의길이][열의길이] = {배열요소[0][0], 배열요소[0][1], ..., 배열요소[1][0], 배열요소[1][1], ..., 배열요소[2][0], 배열요소[2][1], ...};

이 방식도 마찬가지로 초기화하려는 배열 요소가 배열의 길이보다 적으면, 나머지는 모두 0으로 초기화

2. **배열의 모든 요소 초기화**

타입 배열이름[행의길이][열의길이] =

{

    {배열요소[0][0], 배열요소[0][1], ...},

    {배열요소[1][0], 배열요소[1][1], ...},

    {배열요소[2][0], 배열요소[2][1], ...},

    ...

};

보통 이 방식을 많이 사용함!

3. 배열의 일부 요소만을 초기화하는 방식

int arr[3][4] = {

    {10, 20},

    {30, 40, 50, 60},

    {0, 0, 70, 80}

};

\* 2차원 배열의 길이

(열) arr\_column\_len = sizeof(arr[0]) / sizeof(arr[0][0]);

(행) arr\_row\_len = (sizeof(arr) / arr\_column\_len) / sizeof(arr[0][0]);

\* 배열의 길이 자동 설정

배열의 길이를 명시하지 않고 자동으로 설정 가능

단, 행의 길이는 생략할 수 있지만, 열의 길이는 반드시 명시해야 한다.

# 포인터의 개념

\* 주소값의 이해

해당 데이터가 저장된 메모리의 시작 주소(address)

C++에서는 이러한 주소값을 1바이트 크기의 메모리 공간으로 나누어 이해할 수 있습니다.

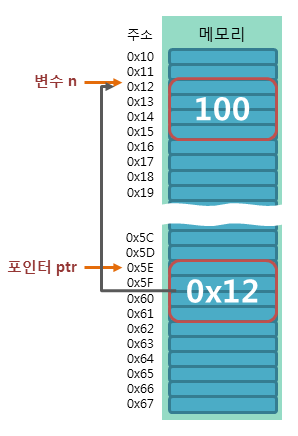
예를 들어, int형 데이터는 4바이트의 크기를 가지지만, int형 데이터의 주소값은 시작 주소 1바이트만을 가리키게 됩니다.

\* 포인터

메모리의 주소값을 저장하는 변수, 포인터 변수라고도 부름.

Ex>

Int n = 100; // 변수 선언

Int \*ptr = &n; // 포인터의 선언

\* 포인터 연산자

1. **주소 연산자 (&) (address operator)**

변수의 이름 앞에 사용, 해당 변수의 주소값 반환

& (ampersand - 앰퍼샌드)라고 읽으며, 번지 연산자 라고도 불림.

2. **참조 연산자( \* ) (reference operator)**

포인터의 이름이나 주소 앞에 사용하여, 포인터에 저장된 주소에 저장돼있는 값 반환.

\*기호 – 역참조연산자 (에스크리터 asterisk operator)

\* 포인터의 선언

문법> 타입\*포인터이름;

\* 포인터의 동시 선언

int \*ptr1, \*ptr2;

\* 포인터의 선언과 초기화

초기화하지 않은 채로 참조 연산자 사용 시, 어딘지 알 수 없는 메모리 장소에 값을 저장하는 것이 됨. 따라서 선언과 동시에 초기화 함께하는게 좋음.

문법>

타입\* 포인터이름 = &변수이름;

또는

타입\* 포인터이름 = &주소값;

\* 포인터의 참조

Ex> 포인터의 주소값과 함께 포인터가 가리키고 있는 주소값의 데이터 참조

int x = 7;            // 변수의 선언

int \*ptr = &x;      // 포인터의 선언

int \*\*pptr = &ptr; // 포인터의 참조

ex>

int num1 = 1234;

double num2 = 3.14;

int\* ptr\_num1 = &num1;

double\* ptr\_num2 = &num2;

cout >> ptr\_num1 ->  0x7fff789fab54

cout >> \*ptr\_num1 -> 1234

# 포인터 연산

값을 더하거나 빼는 등의 제한된 연산만을 할 수 있다.

규칙

1. 포인터끼리의 덧셈 곱셈 나눗셈은 아무런 의미가 없다.
2. 포인터끼리의 뺄셈 : 두 포인터 사이의 상대적 거리
3. 포인터에 정수를 더하거나 빼기 가능 but, 실수와의 연산 x
4. 포인터끼리 대입하거나 비교 가능

\* 타입별 포인터 연산

포인터 연산 후 각각의 포인터가 가리키는 주소는 포인터의 타입에 따라 다르다

Ptr = 0x00 이라고 할 때, int \*ptr이면

Ptr+1 = 0x04, ptr + 2 = 0x08 … (4byte만큼 증가)

\* 포인터와 배열의 관계

배열의 이름은 그 값을 변경할 수 없는 상수라는 점을 제외하곤 포인터와 같다.

배열의 이름을 포인터처럼 사용할 수 있고, 포인터를 배열의 이름처럼 사용할 수도 있다.

-> 배열의 이름이 주소로 해석됨, 해당 배열의 첫 번째 요소의 주소와 같다.

Ex>

Int arr[3] = {10, 20, 30};

Int\* ptr\_arr = arr; // 포인터에 배열의 이름 대입

Arr[0]~arr[2] = 10, 20, 30

Ptr\_arr[0] ~ ptr\_arr[2] = 10, 20, 30

Sizeof(arr) = 12 // 배열의 크기

Sizeof(ptr\_arr) = 8 // 포인터 변수 자체의 크기

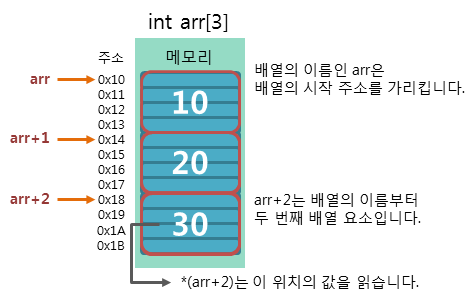
\* 배열의 포인터 연산

배열의 이름을 포인터처럼 사용하는 경우

Int arr[3] = {10, 20, 30};

\*(arr+0) = 10

\*(arr+1) = 20

\*(arr+2) = 30

arr이 배열의 이름이거나 포인터이고 n이 정수일 때,

arr[n] == \*(arr + n) 이라는 공식이 성립함! (다차원 배열에서도 언제나 성립)

# 메모리의 동적 할당

데이터 영역과 스택 영역에 할당되는 메모리 크기 -> 컴파일 타임에 미리 결정됨.

힙 영역의 크기 -> 런 타임(프로그램이 실행되는 도중 시간)에 사용자가 직접 결정

메모리의 동적 할당 : 런타임에 메모리를 할당 받는 것

포인터 = 런 타임에 이름 없는 메모리를 할당받아 포인터에 할당, 할당받은 메모리에 접근

C에선 malloc() 함수 등의 라이브러리로 가능함.

하지만 C++에선 메모리의 동적 할당 및 해제를 위한 효과적인 방법이 있다

1. new 연산자

문법>

타입\* 포인터이름 = new 타입;

첫 번째 타입 : 데이터에 맞는 포인터 선언

두 번째 타입 : 메모리의 종류 지정

사용할 수 있는 메모리가 부족하여 새 메모리를 만들지 못헀다면, new 연산자는 null 포인터 반환

New 연산자 -> 자유 기억 공간(free store)이라고 불리는 메모리 공간(pool)에 객체를 위한 메모리를 할당받는다.

new연산자를 통해 할당받은 메모리는 따로 이름이 없으므로 해당 포인터로만 접근 가능

2. delete 연산자

C에서는 free() 함수를 이용하여 동적으로 할당받은 메모리를 다시 운영체제로 반환

마찬가지로 C++에서는 delete 연산자 이용하여 더 이상 사용하지 않는 메모리를 다시 메모리 공간에 돌려줄 수 있다.

delete 포인터이름;

ex>

int\* ptr\_int = new int;

\*ptr\_int = 100;

double\* ptr\_double = new double;

\*ptr\_double = 100.123;

Delete ptr\_int

Delete ptr\_double;

\*ptr\_int = 100

Ptr\_int = 0x1d~~

\*ptr\_double = 100.123

Ptr\_double = 0x1d~~

이때 new 연산자를 통해 생성한 메모리가 아닌 변수를 선언하여 생성한 메모리는 delete연산자로 해제할 수 없다. New 연산자를 통해 할당한 메모리를 해제할 때만 delete 사용해야 함!

# 문자열(string)

메모리에 저장된 일련의 연속된 문자(character)들의 집합

1. C언어 스타일의 문자열

큰따옴표(“”)를 사용해 표현되는 문자열 – 문자열 상수(string constant)

-> 문자형 배열 선언 시 이 배열이 문자열 변수가 됨.

\* NULL 문자

문자형 배열로 선언된 문자열 변수는 문자열의 끝을 프로그램에 따로 알려주어야 함.

문자열에 속한 값과 그 외의 쓰레깃값을 구분할 수 있다.

문자열 속 데이터가 끝나면 ‘\0’으로 표시(NULL 문자) – 아스키코드값은 0이다.

-> 문자열을 저장하기 위한 문자형 배열을 선언할 때에는 반드시 널 문자까지 포함해서 생각해야 한다.

\* 문자열 입력

문자열이 저장될 문자형 배열을 미리 생성해 놔야 한다. 따라서 입력할 문자열의 길이를 미리 설정해줘야한다.

C++에서 cin 객체는 띄어쓰기를 포함한 탭 문자, 캐리지 리턴 문자 등을

모두 문자열의 끝으로 인식한다.

띄어쓰기를 포함한 문자열을 전부 입력받으려면 cin 객체의 get() 메소드를 사용

20바이트 이상의 이름을 입력 시 cin.ignore() 메소드 사용

const int SIZE = 20;

char address[SIZE];

char name[SIZE];

cout << "자신의 이름을 적어주세요 : ";

cin.get(name, SIZE).ignore(SIZE, ‘\n’);

cout << "자신이 살고 있는 도시를 적어주세요 : ";

cin.get(address, SIZE). ignore(SIZE, ‘\n’);

cout << address << "에 살고 있는 " << name << "님~ 감사합니다!";

1. String 클래스를 이용한 문자열

string 클래스는 문자형 배열보다 사용하기 편리하며, 문자열을 하나의 타입처럼 표현할 수 있게 해준다. string 클래스를 사용하기 위해서는 우선 string 헤더 파일을 포함해야 한다.

Ex>

#include <iostream>

#include <string>

using namespace std;

int main(void)

{

    string address, name;

    cout << "자신의 이름을 적어주세요 : ";

    getline(cin, name); // 한 줄 전체 입력값으로 받아오는 메소드

    cout << "자신이 살고 있는 도시를 적어주세요 : ";

    getline(cin, address);

    cout << address << "에 살고 있는 " << name << "님~ 감사합니다!";

    return 0;

}

\* 문자열 선언 및 초기화

string str1;              // 문자열의 선언 (길이가 0인 string 객체)

str1 = "C++ Programming"; // 문자열의 초기화

string str2 = "C++";      // 문자열의 선언과 동시에 초기화

string 객체는 위와 같이 문자열을 배열 형태가 아닌 단순한 변수로 선언한다 그 후 문자열을 초기화하면 C++ 프로그램이 문자열 str1의 길이를 자동으로 조절한다.

또한, string 객체로 만든 문자열도 배열처럼 인덱스를 통해 다룰 수 있다

Ex>

string dog;

cout << "현재 dog 변수의 길이는 " << dog.length() << "입니다." << endl;

dog = "Navi";

cout << dog << "! 정말 이쁜 이름이네요!" << endl;

cout << "현재 dog 변수의 길이는 " << dog.length() << "입니다." << endl;

cout << "강아지 이름의 첫 글자는 바로 " << dog[0] << "입니다.";

-> 컴퓨터에서 한글은 2바이트로 취급된다. 따라서 위의 예제에서 한글을 입력하면 이름의 첫 글자로 이상한 문자가 출력될 것입니다.

-> length() , size() 메소드 사용 시 string 객체의 현재 길이를 알 수 있다.

\* 문자열 처리

C언어에서 문자형 배열 사이의 대입 작업은 strcpy() 함수나 strncpy() 함수를 사용했다.

하지만 string 객체를 이용한 문자열에서는 대입 연산자 만으로도 대입 작업 가능

string str1 = "C++ is Cool!";

string str2;

str2 = str1; // 문자열 대입 연산

cout << str2;

또, 문자형 배열 사이의 추가나 결합 작업은 strcat()이나 strncat() 함수를 사용했다면, string 객체를 이용하면 복합 대입 연산자(+=)나 append() 메소드를 이용하여 손쉽게 추가나 결합을 할 수 있습니다.

Ex>

string str1 = "C++ is ";

string str2 = "Cool! and funny!";

string str3;

str3 = str1 + str2; // 문자열 결합 연산

cout << str3 << endl;

str1 += str2;       // 문자열 추가 연산

cout << str1;

\* 문자열 입출력

cin 객체를 이용한 문자열의 입력은 한 단어 단위로 수행되는데 문자열을 한 행(line)씩 읽고 싶을 때는 getline() 함수를 사용해야 한다.

Ex>

string name, subject;

cout << "자신의 이름을 적어주세요 : ";

getline(cin, name);

cout << "가장 자신있는 과목을 적어주세요 : ";

getline(cin, subject);

# string 메소드

1. Length() & size() 메소드

Length() 메소드 : 문자열의 길이, size() 메소드 : string 객체가 메모리에서 실제 사용하는 크기

1. Append() 메소드

문자열.append(추가할문자열, [시작위치], [개수]); [ ]는 생략 가능

문자열.append(개수, 추가할 문자);

1. Find() 메소드

문자열.find(찾을 문자열 or 찾을 문자, [시작위치] ); // [시작위치]부터 찾을 문자열을 찾아 문자열의 인덱스 반환

\* string::npos => index = -1 을 의미함

1. Compare() 메소드

문자열 간의 내용 비교 -> true / false 반환

Str1.compare(str2) == 0 // 둘이 같은 경우

Str1.compare(str2) < 0 // str1 이 사전 순으로 앞일 경우

Str1.compare(str2) > 0 // str2가 사전 순으로 앞일 경우

1. Replace() 메소드

문자열.replace(대체할문자열의시작위치, 대체할문자열의길이, 새로운문자열);

Ex>

string str1 = "C++ is very nice!";

string str2 = "nice";

string str3 = "awesome";

string::size\_type index = str1.find(str2);

if (index != string::npos) // string::npos 는 인덱스가 -1일때의 경우로 -1을 리턴함.

{

    str1.replace(index, str2.length(), str3);

}

cout << str1;

1. Capacity() 메소드와 max\_size() 메소드

Str.Capacity() : 해당 문자열이 재할당받지 않고 저장할 수 있는 최대 문자열의 길이

Str.Max\_size() : 해당 문자열이 최대한 대입받으면 가질 수 있는 최대 문자열의 길이

ex>

string str = "C++ Programming";

cout << "문자열 str의 length는 " << str.length() << "입니다." << endl;

cout << "문자열 str의 capacity는 " << str.capacity() << "입니다." << endl;

cout << "문자열 str의 max\_size는 " << str.max\_size() << "입니다.";

\* 문자열 공백 분리

C++ 에서는 STL에서 파이썬의 split 같은 함수를 지원하지 않기에, sstream 이용!  
ex> 공백 분리 후 변수에 할당

#include <iostream>

#include <string>

#include <sstream>

#include <vector>

using namespace std;

int main(){

// 공백 분리할 문자열 선언

string input = "abc def ghi";

// 문자열을 스트림화

stringstream ss(input);

// 스트림을 통해, 문자열을 공백 분리해 변수에 할당

string first, second, third;

ss >> first >> second >> third;

}

Ex> 공백 분리 후 배열에 저장

#include <iostream>

#include <string>

#include <sstream>

#include <vector>

using namespace std;

int main(){

// 공백 분리할 문자열 선언

string input = "abc def ghi";

// 문자열을 스트림화

stringstream ss(input);

// 공백 분리 결과를 저장할 배열

vector<string> words;

string word;

// 스트림을 한 줄씩 읽어, 공백 단위로 분리한 뒤, 결과 배열에 저장

while (getline(ss, word, ' ')){

words.push\_back(word);

}

}

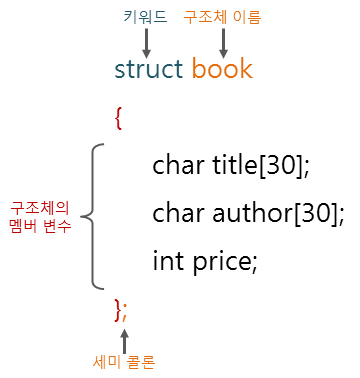
# 구조체 (structure type) -> 객체지향프로그래밍의 클래스의 기초

C++의 기본 타입을 가지고 새롭게 정의할 수 있는 사용자 정의 타입

배열이 같은 타입의 변수 집합이라면 구조체는 다양한 타입의 변수 집합을 하나의 타입으로 나타낸 것

이때 구조체를 구성하는 변수 = member variable // 함수도 멤버 변수로 가질 수 있다.

Ex>

struct 구조체이름

{

    멤버변수1의타입 멤버변수1의이름;

    멤버변수2의타입 멤버변수2의이름;

    ...

};

선언된 구조체 타입을 가지고 변수 선언

구조> Struct 구조체이름 구조체변수이름;

Ex> (struct)[생략가능] Book web\_book;

\* 구조체 멤버로의 접근 방법

배열에서는 인덱스 이용 -> 구조체는 멤버 참조 연산자 (.) 사용

문법> 구조체변수이름.멤버변수이름;

Ex> web\_book.author

\* 구조체 변수의 초기화

구조체변수이름 = {멤버변수1의초깃값, 멤버변수2의초깃값, ...};

Ex> web\_book = {"HTML과 CSS", "홍길동", 28000};

멤버 변수가 정의된 순서에 따라 차례대로 초깃값이 설정되며, 나머지 멤버 변수는 0으로 초기화

# 함수와 구조체

함수를 호출할 때 전달되는 인수나, 함수가 종료될 때 반환되는 반환값으로도 구조체를 사용할 수 있다.

Ex> 구조체의 멤버 변수를 함수의 인자로 전달하는 예제

struct Prop

{

    int savings;

    int loan;

};

int CalcProperty(int, int);

int main(void)

{

    int hong\_prop;

    Prop hong = {10000000, 4000000};

    hong\_prop = CalcProperty(hong.savings, hong.loan); // 구조체의 멤버 변수를 함수의 인수로 전달함

    cout << "홍길동의 재산은 적금 " << hong.savings << "원에 대출 " << hong.loan

        << "원을 제외한 총 " << hong\_prop << "원입니다.";

    return 0;

}

int CalcProperty(int s, int l)

{

    return (s - l);

}

구조체를 인수로 전달하는 방식은 함수가 원본 구조체의 복사본을 가지고 작업하므로 안전하다.

Ex2 > 함수의 인수로 구조체의 주소를 직접 전달

int CalcProperty(Prop\*);

int main(void)

{

    ...

    hong\_prop = CalcProperty(&hong); // 구조체의 주소를 함수의 인수로 전달함.

    ...

}

int CalcProperty(Prop\* money)

{

    money->savings = 100; // 호출된 함수에서 원본 구조체의 데이터를 변경

    return (money->savings - money->loan);

}

구조체를 가리키는 포인터를 인수로 전달하는 방식 -> 구조체의 복사본이 아닌 주소 하나만을 전달 -> 빠름

But, 호출된 함수에서 원본 구조체에 직접 접근, 원본 데이터의 보호 측면에선 위험함

따라서 예제의 CalcProperty() 함수처럼 const 키워드를 사용하여 함수에 전달된 인수를 함수 내에선 직접 수정할 수 없도록 하는게 좋음.

Prop InitProperty(void);

int CalcProperty(const Prop\*);

int main(void)

{

    ...

    hong = InitProperty();

    hong\_prop = CalcProperty(&hong); // 구조체의 멤버 변수를 함수의 인수로 전달함

    ...

}

Prop InitProperty(void)

{

    Prop hong\_prop = {10000000, 4000000};

    return hong\_prop; // 구조체를 함수의 반환값으로 반환함.

}

int CalcProperty(const Prop\* money) // const 키워드를 사용하여 구조체의 데이터를 직접 수정하는 것을 방지함.

{

    //money->savings = 100; // 호출된 함수에서 원본 구조체의 데이터를 변경

    return (money->savings - money->loan);

}

\* 중첩된 구조체

구조체 정의 시 멤버 변수로 또 다른 구조체 포함시킬 수 있음.

struct Name

{

    string first;

    string last;

};

struct Friends

{

    Name first\_name;

    string address;

    string job;

};

\* 구조체의 크기

-> 멤버 변수들의 크기에 따라 결정

-> 구조체의 크기 != 멤버 변수들의 크기 총합

struct TypeSize

{

    char a;

    int b;

    double c;

};

int main(void)

{

    cout << "구조체 TypeSize의 각 멤버의 크기는 다음과 같습니다." << endl;

    cout << sizeof(char) << ", " << sizeof(int) << ", " << sizeof(double) << endl; // 1 4 8

    cout << "구조체 TypeSize의 크기는 다음과 같습니다." << endl;

    cout << sizeof(TypeSize); // 16

    return 0;

}

멤버 변수 크기 합 : 13

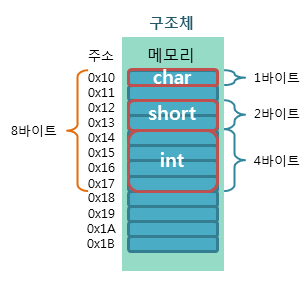
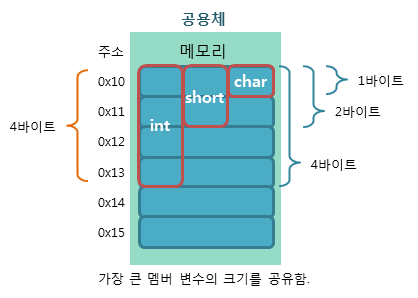
구조체 크기 : 16

-> 구조체를 메모리에 할당할 때 컴파일러는 프로그램의 속도 향상을 위해 바이트 패딩(byte padding) 이라는 규칙 이용

구조체는 다양한 크기의 타입을 멤버 변수로 가질 수 있는 타입이지만 컴파일러는 메모리의 접근을 쉽게 하려고 크기가 가장 큰 멤버 변수를 기준으로 모든 멤버 변수의 메모리 크기를 맞추게 된다. 이것을 바이트 패딩이라고 하며, 이때 추가되는 바이트를 패딩 바이트(padding byte)라고 합니다.

# 공용체

Union 키워드로 선언, 모든 면에서 구조체와 같지만 모든 멤버 변수가 하나의 메모리 공간을 공유한다는 점이 차이점이다.

그래서 한 번에 하나의 멤버 변수밖에 사용할 수 없다.

공용체는 순서가 규칙적이지 않고, 미리 알 수 없는 다양한 타입의 데이터를 저장할 수 있도록 설계된 타입이다. 또한 크기가 가장 큰 멤버 변수의 크기로 메모리를 할당 받는다.

따라서 공용체 배열을 사용하면, 같은 크기로 구성된 배열 요소에 다양한 크기의 데이터를 저장 가능

union ShareData

{

    unsigned char a;

    unsigned short b;

    unsigned int c;

};

int main(void)

{

    ShareData var;

   var.c = 0x12345678;

    cout << hex;

    cout << var.a << endl;

    cout << var.b << endl;

    cout << var.c;

    return 0;

}

실행 결과

X

5678

12345678

공용체에 저장된 값의 의미는 값을 저장할 때 공용체의 어떤 멤버 변수를 사용했는지에 따라 달리 해석된다. 따라서 공용체가 어떤 멤버 변수를 사용하여 저장했는지를 별도의 변수로 저장하든가 하여, 사용할 때에도 같은 멤버 변수를 사용해야 합니다.

# 열거체

새로운 타입을 선언하면서 동시에 그 타입이 가질 수 있는 정수형 상숫값도 같이 명시합니다.

enum Weather {SUNNY = 0, CLOUD = 10, RAIN = 20, SNOW = 30};

int main(void)

{

    int input;

    Weather wt;

    cout << "오늘의 날씨를 입력해 주세요 : " << endl;

    cout << "(SUNNY=0, CLOUD=10, RAIN=20, SNOW=30)" << endl;

    cin >> input;

    wt = (Weather)input;

    switch (wt)

    {

        case SUNNY:

            cout << "오늘의 날씨는 아주 맑아요!";

            break;

        case CLOUD:

            cout << "오늘의 날씨는 흐리네요!";

            break;

        case RAIN:

            cout << "오늘의 날씨는 비가 주룩주룩 오네요!";

            break;

        case SNOW:

            cout << "오늘의 날씨는 하얀 눈이 내려요!";

            break;

        default:

            cout << "정확한 상숫값을 입력해 주세요!";

            break;

    }

    cout << endl << "열거체 Weather의 각 상숫값은 " << SUNNY << ", " << CLOUD << ", "

        << RAIN << ", " << SNOW << "입니다.";

    return 0;

}

상숫값을 따로 명시하지 않으면 0부터 시작되며, 그 다음은 바로 앞의 상숫값보다 1만큼 증가되며 정의됩니다.

# 함수

특별한 목적의 작업을 수행하기 위해 독립적으로 설계된 코드의 집합

-> 특정 작업 캡슐화하는 데 유용

\* 사용 이유

반복적인 프로그래밍 회피 가능

프로그램을 여러 개의 함수로 나누어 작성 시, 모듈화로 인해 전체적인 코드의 가독성이 좋아짐.

기능 변경 필요 시 손쉽게 유지보수 가능

\* 함수의 선언

int sum (x, y)

{

//함수 내용

}

리턴 값으로 배열을 제외한 모든 타입 사용 가능 (단, 구조체나 객체에 포함된 배열은 반환 가능)

\* 함수의 원형 선언

Main 함수보다 뒤에 함수를 선언하는 경우 컴파일러에 의해 함수가 컴파일 될 때, main함수에 등장하는 함수를 아직 알지 못하여 컴파일 오류 발생

-> 함수 원형 선언

반환타입 함수명(매개변수목록);

Ex>

#include <iostream>

using namespace std;

int SmallNum(int, int);

int main(void)

{

    int result;

    result = SmallNum(4, 6);

    return 0;

}

int SmallNum(int num1, int num2)

{

    …

}

\* 인수 전달 방법

1. 값에 의한 전달 (call by value)

인수로 전달되는 변수가 가지고 있는 값을 함수 내의 매개변수에 복사하는 방식

void Local(int);

int main(void)

{

    int var = 10;

    cout << "변수 var의 초깃값은 " << var << "입니다." << endl;

    Local(var);

    cout << "Local() 함수 호출 후 변수 var의 값은 " << var << "입니다.";

    return 0;

}

void Local(int num)

{

    num += 10;

}

Var = 10

함수 호출 후 Var = 10

-> 함수 내에서 매개변수 num의 값을 아무리 변경하더라도 원래 인수로 전달된 변수 var의 값은 절대 변경되지 않는다.

\* 참조에 의한 전달 (call by reference)

원본 데이터 직접 전달 -> 참조자 사용

void Local(int&);

int main(void)

{

    ...

    Local(var);

    ...

}

void Local(int& num)

{

    num += 10;

}

Var = 10

함수 이후 var = 20

위의 예제에서 Local() 함수의 매개변수 num는 인수로 변수 var의 참조를 전달받는다. 따라서 함수 내에서 참조자 num의 값을 변경하면 원래 인수인 변수 var의 값도 같이 변경된다.

\* main() 함수의 인수 전달

Void(또는 int) main(int argc, char \*argv[]);

main() 함수의 첫 번째 인수인 int형 변수 argc는 인수로 전달되는 문자열의 개수를 명시합니다.

두 번째 인수 char형 포인터의 포인터인 argv는 인수로 전달된 각각의 문자열이 포함되는 배열을 가리킵니다.

# 재귀 호출

int RecursiveSum(int n)

{

    if (n == 1)                   // n이 1이면, 그냥 1을 반환함.

    {

          return 1;

    }

    return n + RecursiveSum(n-1); // n이 1이 아니면, n을 1부터 (n-1)까지의 합과 더한 값을 반환함.

}

\* 함수 포인터

프로그램에서 정의된 함수는 프로그램이 실행될 때 모두 메인 메모리에 올라갑니다. 이때 **함수의 이름**은 메모리에 올라간 함수의 시작 주소를 가리키는 포인터 상수(constant pointer)가 됩니다.

이렇게 함수의 시작 주소를 가리키는 포인터 상수를 함수 포인터(function pointer)라고 부릅니다.

-> 포인터 상수(constant pointer)란 포인터 변수가 가리키고 있는 주소 값을 변경할 수 없는 포인터를 의미하며,

상수 포인터(pointer to constant)란 상수를 가리키는 포인터를 의미합니다.

함수 포인터의 포인터 타입은 함수의 반환값과 매개변수에 의해 결정됩니다.

즉 함수의 원형을 알아야만 해당 함수에 맞는 함수 포인터를 만들 수 있습니다.

void Func(int, int); // 함수 원형

void (\*ptr\_func)(int, int); // 함수 포인터

함수 포인터는 함수를 또 다른 함수의 인수로 전달할 때 유용하게 사용됩니다.

#include <iostream>

using namespace std;

double Add(double, double);

double Sub(double, double);

double Mul(double, double);

double Div(double, double);

double Calculator(double , double, double (\*func)(double, double));

int main(void)

{

    double (\*calc)(double, double) = NULL; // 함수 포인터 선언

    double num1 = 3, num2 = 4, result = 0;

    char oper = '\*';

    switch (oper)

    {ㅍ

        case '+' :

            calc = Add;

            break;

        case '-':

            calc = Sub;

            break;

        case '\*':

            calc = Mul;

            break;

        case '/':

            calc = Div;

            break;

        default:

            cout << "사칙연산(+, -, \*, /)만을 지원합니다."; }

            break;

    }

    result = Calculator(num1, num2, calc);

    cout << "사칙 연산의 결과는 " << result << "입니다.";

    return 0;

}

double Add(double num1, double num2) { return num1 + num2; }

double Sub(double num1, double num2) { return num1 - num2; }

double Mul(double num1, double num2) { return num1 \* num2; }

double Div(double num1, double num2) { return num1 / num2; }

double Calculator(double num1, double num2, double (\*func)(double, double)) { return func(num1, num2); }

앞서 함수 포인터를 표기했을 때

Double (\*calc)(double, double) = NULL; 라는 표기는 너무 복잡하다,

따라서 typedef / auto 키워드를 사용하여 단순하게 나타낼 수 있다.

typedef double (\*CalcFunc)(double, double); // 함수 포인터에 calcFunc이라는 새로운 이름을 붙임.

auto ptr\_func = calc; //

typedef를 사용하면 다음과 같이 나타낼 수 있다.

#include <iostream>

using namespace std;

typedef double (\*Arith)(double, double); // typedef 키워드를 이용한 새로운 이름 선언

...

double Calculator(double , double, Arith);

int main(void)

{

    Arith calc = NULL; // 함수 포인터 선언

    double num1 = 4, num2 = 5, result = 0;

    char oper = '+';

    ...

}

...

double Calculator(double num1, double num2, Arith func) { return func(num1, num2); }