0. C++ 시작

객체지향과 절차지향

- 절차지향(Procedural) 프로그래밍
  - 절차지향 언어는 문제를 여러 개의 작은 함수(function)로 나누어 그 문제를 해결
  - 절차지향은 동사 중심의 프로그래밍 방식 : **"어떤 처리 함수를 수행하는가"**

- 객체지향(Object oriented) 프로그래밍
  - 관련된 변수(속성)와 함수(기능)를 묶어서 '객체(object)'를 만들고, 객체 단위로 프로그래밍
    - C에서의 구조체? 변수들만 묶여 있음.
    - 세상 모든 것이 객체이다.

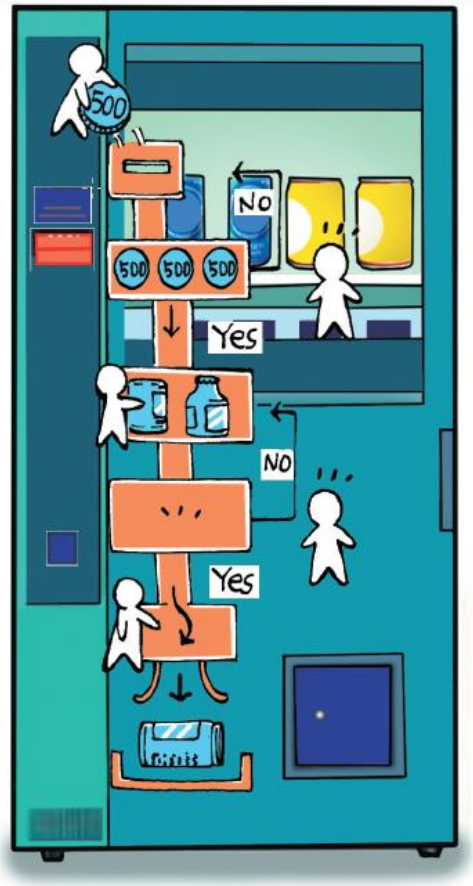
TV A

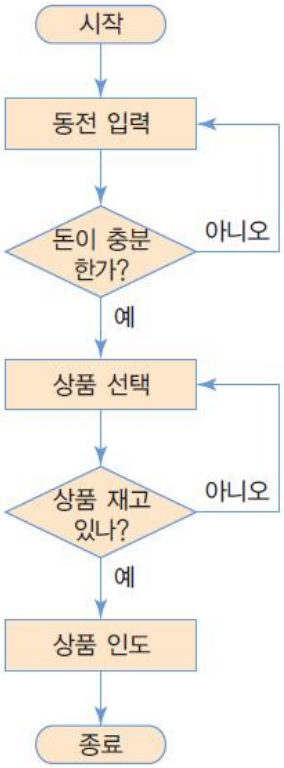

TV B

	TV A	TV B
속성	색깔 : 검은색	색깔 : 흰색+원목색
	크기 : 55인치	크기 : 42 인치
	제조사 : Samsung	제조사 : LG
기능	인터넷 연결 가능	인터넷 연결 안됨
	볼륨 조절 기능	볼륨 조절 기능
	채널 돌림 기능	채널 돌림 기능

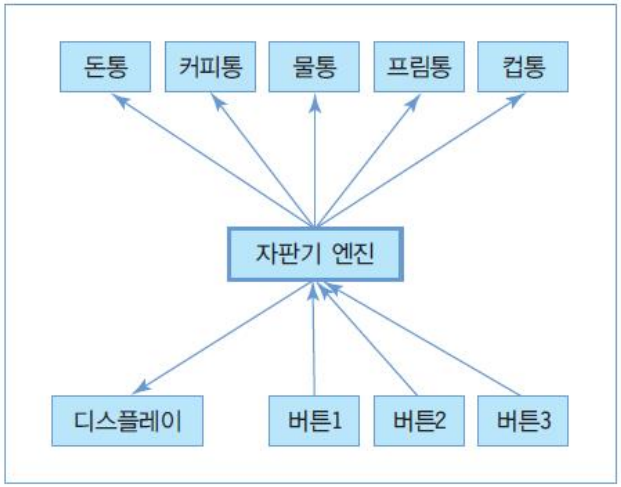
  - 객체지향은 명사 중심의 프로그래밍 방식 : **"어떤 객체가 동작하는가"**
    - 예) TV A의 볼륨을 올려라. TV B의 채널을 돌려라.

절차 지향 프로그래밍과 객체 지향 프로그래밍





(a) 절차 지향적 프로그래밍으로 구현할 때의 흐름도



(b) 객체 지향적 프로그래밍으로 구현할 때의 객체 관계도

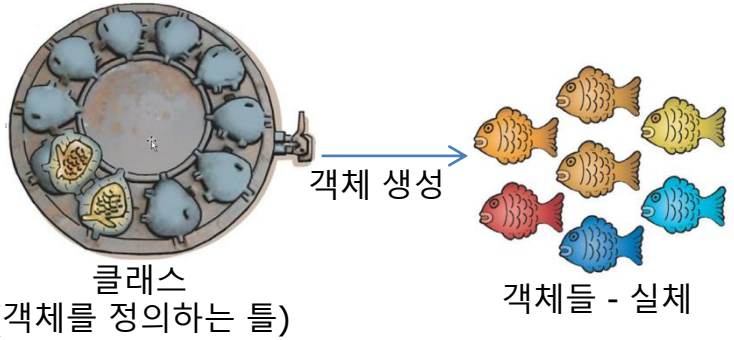
- 실행하고자 하는 절차대로 일련의 명령어 나열.
- 흐름도를 설계하고 흐름도에 따라 프로그램 작성

- 객체들을 정의하고, 객체들의 상호 관계, 상호 작용으로 구현

클래스와 객체

클래스(Class)

- 사용자정의 자료형이라 생각하면 편함(일종의 데이터 형 역할)
- 클래스에는 **변수**라는 저장공간 외에 기능이라는 **프로시저**(함수, 기능)가 포함되어 있음.
- 객체가 가지는 공통된 특성을 기술하는 것
- 클래스에 비유될 수 있는 것 : 붕어빵 틀



객체(Object)

- 클래스 형에 의해 만들어진 실제 사용되는 변수
- 구체적인 값을 갖는 실체(instance)
- 객체에 비유 될 수 있는 것 : 붕어빵 틀에 의해 만들어진 실제 붕어빵
- 객체는 자료와 일련의 처리 명령을 하나로 묶어 놓은 메소드(함수, 프로시저)로 구성되는 프로그램 단위로, 함수보다 높은 수준의 모듈화 방법이라 할 수 있음.

클래스의 정의

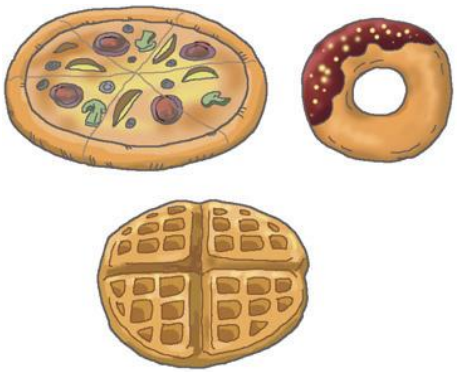
- 원을 추상화 한 Circle 클래스

멤버들

```
class Circle {
public:
    int radius; // 반지름 값
public:
    Circle(int r) { radius = r; }
    double getArea() { return 3.14*radius*radius; }
};
```

- 실제로 만들어진 객체

```
Circle pizza;
Circle donut;
Circle waffle;
```



원 객체들(실체)

객체의 구성요소

- **객체의 선언** `Circle pizza, donut, waffle;`

- **멤버변수** : 객체 내부의 변수
  - 객체 내부의 정보 저장

```
pizza.radius = 15;
```

- **멤버함수**: 객체 내부의 프로시저
  - 객체가 수행할 수 있는 행위를 기술

```
pizza.getArea();
```

생성자

- **생성자(constructor)**
  - 객체가 처음 생성될 때 객체의 초기화에 사용되는 특별한 멤버함수

```
class Circle {
   public:
      int radius; // 반지름 값
   public:
      Circle(int r) {
          radius = r;
      }
       double getArea() {
            return 3.14*radius*radius;
       }
};
```

생성자는 객체가 생성될 때 radius에 값을 배정한다.

```
Circle pizza(15);
```

클래스 특성 – 캡슐화

- **캡슐화(encapsulation) :** 접근지정자를 통한 캡슐화 구현
  - 객체의 내부 구성요소에 대한 접근을 제한하기 위한 수단
    - 객체를 사용하기 위해서 필요한 부분을 제외한 나머지 부분을 캡슐로 감싸서 숨기는 것
  - 전용(private) 속성과 공용(public) 속성

```
class Circle
{
private:
    int radius; // 반지름 값

public:
    Circle(int r) {
        radius = r;
    }
    double getArea() {
        return 3.14*radius*radius;
    }
};
```

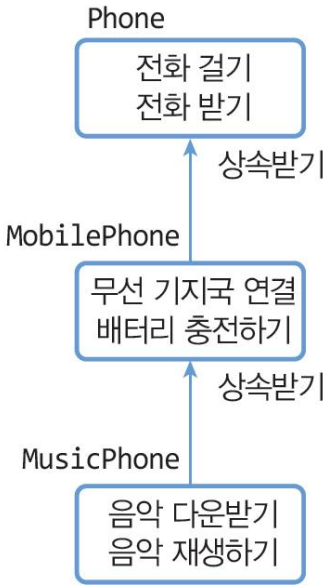
클래스 안의 구성요소들은 다른 프로그램 단위에서 접근할 수 있는지 여부에 따라 공용이나 전용으로 표시됨.

```
pizza.radius = 15; (오류)
```

클래스 특성 – 상속

- **객체 지향 상속(Inheritance)**
  - 자식이 부모의 유전자를 물려 받는 것과 유사

- **C++ 상속**
  - 객체가 자식 클래스의 멤버와 부모 클래스에 선언된 모양 그대로 멤버들을 가지고 탄생



상속 관계 표현

```
class Phone {
   void call();
   void receive();
};

class MobilePhone : public Phone {
   void connectWireless();
   void recharge();
};

class MusicPhone : public MobilePhone {
   void downloadMusic();
   void play();
};
```

Phone을 상속받는다.

MobilePhone을 상속받는다.

C++로 상속 선언

전화기

휴대 전화기

음악 기능 전화기

클래스 특성 – 다형성

- **다형성(polymorphism)**
  - 하나의 기능이 경우에 따라 다르게 보이거나 다르게 작동하는 현상
  - 연산자 중복, 함수 중복, 함수 재정의(overriding)

+ 연산자 중복

```
2 + 3                          → 5
"남자" + "여자"                 → "남자여자"
redColor 객체 + blueColor 객체  → purpleColor 객체
```

add 함수 중복

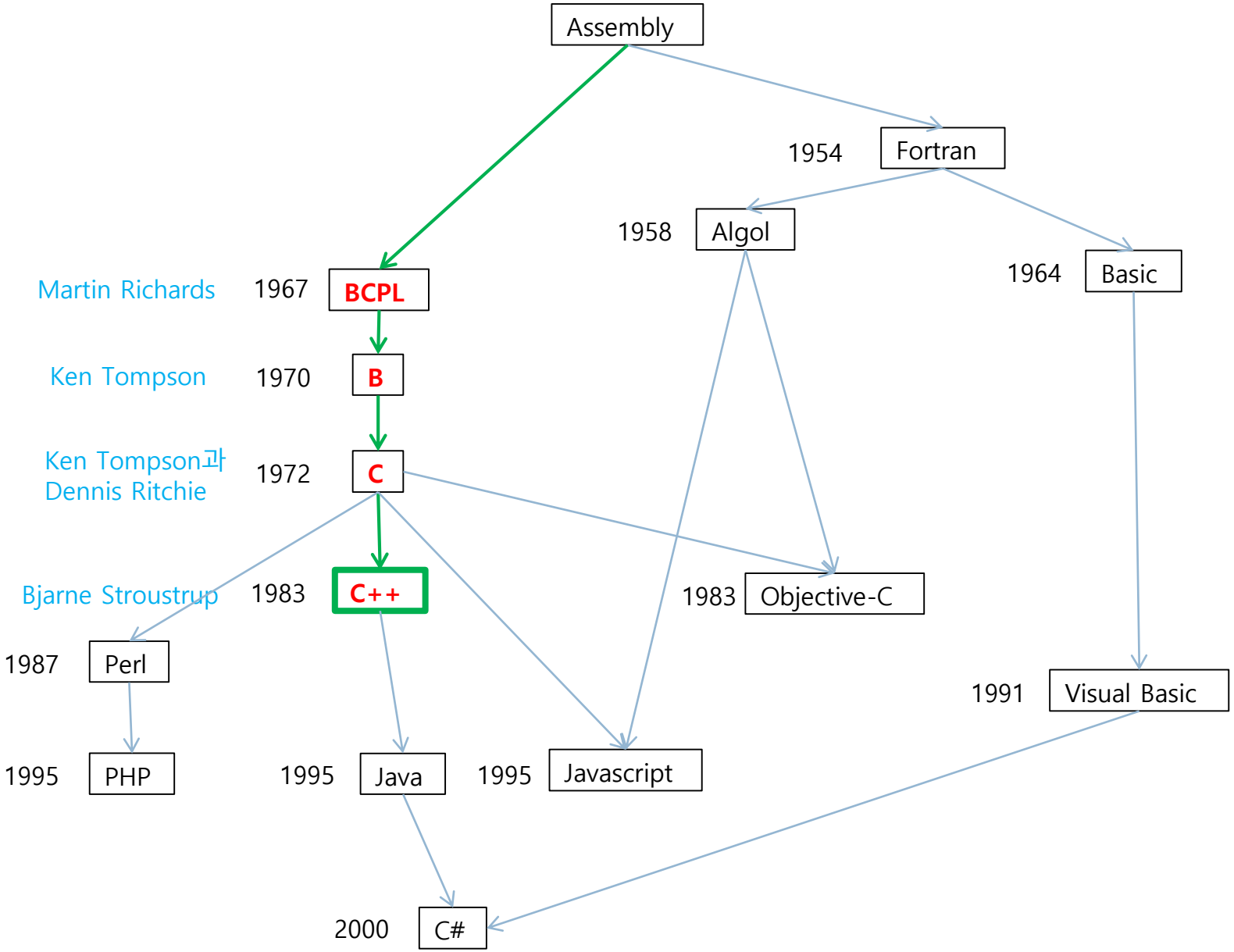
```
void add(int a, int b) { ... }
void add(int a, int b, int c) { ... }
void add(int a, double d) { ... }
```

함수 재정의(오버라이딩)

객체지향프로그래밍언어 C++

우리는 C를 배웠으므로 C++을 통해 객체지향프로그래밍을 해본다!

프로그래밍 언어의 진화와 C++의 기원



C ++ 언어에서 객체 지향을 도입한 목적

- 소프트웨어 생산성 향상
  - 소프트웨어의 생명 주기 단축 문제 해결 필요
  - 기 작성된 코드의 재사용 필요
  - C++ 클래스 상속 및 객체 재사용으로 해결

- 실세계에 대한 쉬운 모델링
  - 과거의 소프트웨어
    - 수학 계산이나 통계 처리에 편리한 절차 지향 언어가 적합
  - 현대의 소프트웨어
    - 물체 혹은 객체의 상호 작용에 대한 묘사가 필요
    - 실세계는 객체로 구성된 세계
    - 객체를 중심으로 하는 객체 지향 언어 적합

표준 C++ 프로그램의 중요성

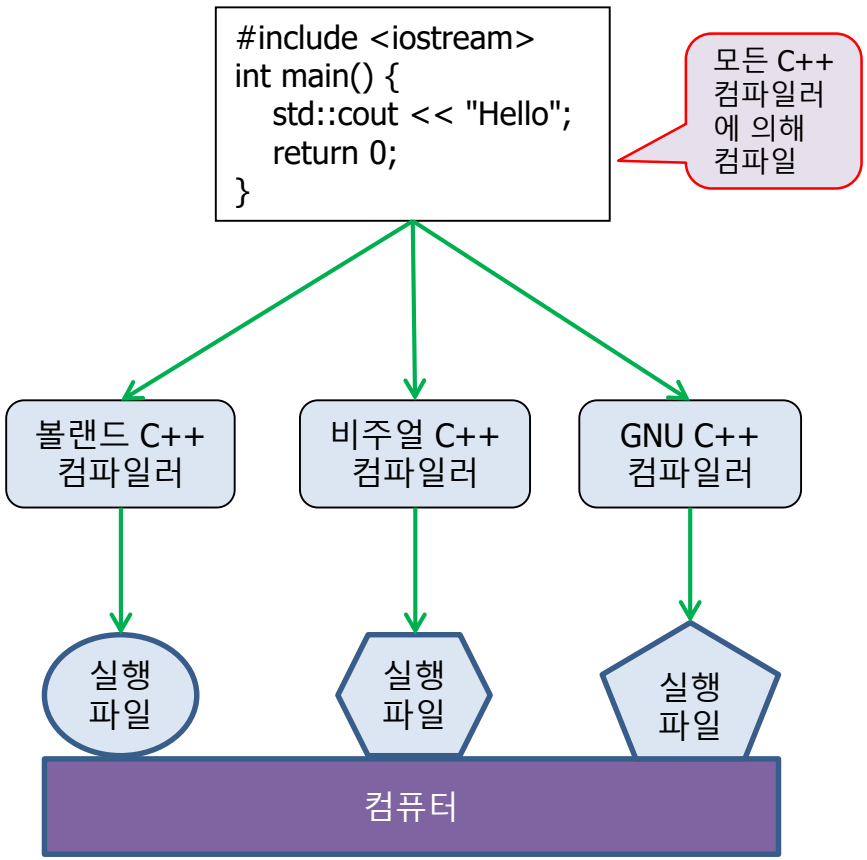
- C++ 언어의 표준
  - 1998년 미국 표준원(ANSI, American National Standards Institute)
    - C++ 언어에 대한 표준 설정
  - ISO/IEC 14882 문서에 작성됨. 유료 문서
  - 표준의 진화
    - 1998년(C++98), 2003년(C++03), 2007년(C++TR1), 2011년(C++11)

- 표준의 중요성
  - 표준에 의해 작성된 C++ 프로그램
    - 모든 플랫폼. 모든 표준 C++ 컴파일러에 의해 컴파일
    - 동일한 실행 결과 보장
    - 운영체제와 컴파일러의 종류에 관계없는 높은 호환성

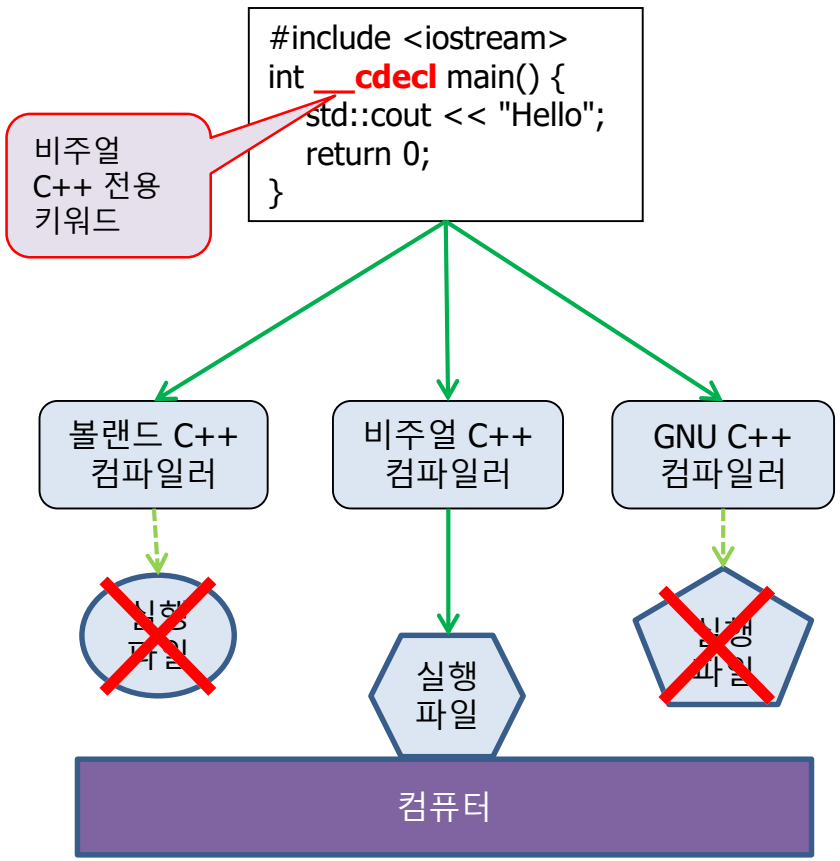
- 비 표준 C++ 프로그램
  - Visual C++, Borland C++ 등 컴파일러 회사 고유의 비 표준 구문
    - 특정 C++ 컴파일러에서만 컴파일
  - 호환성 결여

표준/비표준 C++ 프로그램의 비교

표준 C++ 규칙에 따라
작성된 C++ 프로그램



표준 C++ 규칙에 따라
작성되지 않는 비주얼 C++ 프로그램



C언어의 개선을 통해 탄생된 C++

- **C 언어의 개선**을 통해 탄생됨
  - 클래스와 여러 기능을 추가하여 C++라는 언어 탄생
  - 다른 언어에 비해 정보를 보다 효과적으로 관리하고 처리

- **C와의 호환성**
  - C 언어의 문법 체계 계승
    - 소스 레벨 호환성 - 기존에 작성된 C 프로그램을 그대로 가져다 사용
    - 링크 레벨 호환성 – C 목적 파일과 라이브러리를 C++ 프로그램에서 링크

- **객체 지향 개념 도입**
  - 캡슐화, 상속, 다형성
  - 소프트웨어의 재사용을 통해 생산성 향상
  - 복잡하고 큰 규모의 소프트웨어의 작성, 관리, 유지보수 용이

- 엄격한 타입 체크. 실행 시간 오류의 가능성을 줄임

- 실행 시간의 효율성 저하 최소화
  - 작은 크기의 멤버 함수 잦은 호출 가능성 → 인라인 함수로 실행 시간 저하 해소

C언어의 개선을 통해 탄생된 C++

- C언어에 추가한 기능들
  - 인라인 함수(inline function)
    - 함수 호출 대신 함수 코드의 확장 삽입
  - 디폴트 인자(default parameter)
    - 매개 변수에 디폴트 값이 전달되도록 함수 선언
  - 함수 중복(function overloading)
    - 매개 변수의 개수나 타입이 다른 동일한 이름의 함수들 선언
  - 참조와 참조 변수(reference)
    - 하나의 변수에 별명을 사용하는 참조 변수 도입
  - 참조에 의한 호출(call-by-reference)
    - 함수 호출 시 참조 전달
  - new/delete 연산자
    - 동적 메모리 할당/해제를 위해 new와 delete 연산자 도입
  - 연산자 재정의
    - 기존 C++ 연산자에 새로운 연산 정의
  - 제네릭 함수와 클래스
    - 데이터 타입에 의존하지 않고 일반화시킨 함수나 클래스 작성 가능

C언어의 개선을 통해 탄생된 C++

```
#include <iostream>

// 원형 선언
void star(int a=5);

// 함수 구현
void star(int a) {
        for(int i=0; i<a; i++) std::cout << '*';
        std::cout << '\n';
}

int main() {
    star();          // star(5);와 동일
    star(10);

    return 0;
}
```

```
#include <iostream>

int GetSum(int x, int y);
int GetSum(const int arr[], int size);

int main() {
    int a, b;
    std::cout << "두 정수를 입력하세요 : ";
    std::cin >> a >> b;
    std::cout << GetSum(a, b) << '\n';

    int arr[] = {10, 20, 30, 40, 50};
    int n = sizeof(arr) / sizeof(arr[0]);
    std::cout  << GetSum(arr, n) << '\n';

    return 0;
}

int GetSum(int x, int y) {
    return x + y;
}

int GetSum(const int arr[], int size) {
    int sum = 0;
    for(int i = 0 ; i < size ; i++)
        sum += arr[i];
    return sum;
}
```

객체지향 및 제너릭 프로그래밍이 가능한 C++

제네릭 프로그래밍(generic programming) 기능

- 템플릿을 사용하여 정보의 타입과 관계없이 동일한 방법으로 처리할 수 있는 알고리즘을 구현할 수 있게 함.
  - 제네릭 함수와 제네릭 클래스를 활용하여 프로그램을 작성하는 새로운 프로그래밍 패러다임
  - 점점 중요성이 높아지고 있음

제네릭 함수와 제네릭 클래스

- 제네릭 함수(generic function)
  - 동일한 프로그램 코드에 다양한 데이터 타입을 적용할 수 있게 일반화 시킨 함수
- 제네릭 클래스(generic class)
  - 동일한 프로그램 코드에 다양한 데이터 타입을 적용할 수 있게 일반화시킨 클래스
- **template** 키워드로 선언
  - 템플릿 함수 혹은 템플릿 클래스라고도 부름

객체지향 및 제너릭 프로그래밍이 가능한 C++

```
#include <iostream>

template<class T>
T max(T a, T b) {
        return (a > b ? a : b);
}

int main() {
        int i1 = 5, i2 = 3;
        int i3 = max(i1, i2);    // i3 = 5   max<int>(i1, i2)와 동일
        std::cout << i3 << endl;

        double d1 = 0.9, d2 = 1.0;
        double d3 = max(d1, d2);    // d3 = 1.0 5   max<double>(d1,d2)와 동일
        std::cout << d3 << endl;
}
```

C++ 언어의 아킬레스

- C++ 언어는 C 언어와의 호환성 추구
  - 장점
    - 기존에 개발된 C 프로그램 코드 활용
  - 단점
    - 캡슐화의 원칙이 무너짐
      - C++에서 전역 변수와 전역 함수를 사용할 수 밖에 없음
      - 부작용(side effect) 발생 염려

C++ 프로그램 개발 과정

C++ 소스 프로그램 작성

```
#include <iostream>
int main() {
   std::cout << "Hello";
   return 0;
}
```

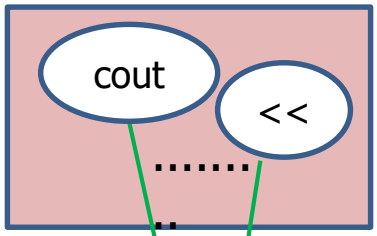
소스 파일
(hello.cpp)

컴파일

```
_main,12#
$<<01010
00000111
_Hello001
```

목적 파일
(hello.obj)

C++ 라이브러리



링킹

```
01010000
01000101
01001111
01011010
10100101
11010101
```

실행 파일
(hello.exe)

실행

오류 발생

디버깅

오류 수정

C++ 표준 라이브러리

- C++ 표준 라이브러리는 3개의 그룹으로 구분
  - **C 라이브러리**
    - 기존 C 표준 라이브러리를 수용, C++에서 사용할 수 있게 한 함수들
    - 이름이 c로 시작하는 헤더 파일에 선언됨
  - **C++ 입출력 라이브러리**
    - 콘솔 및 파일 입출력을 위한 라이브러리
  - **C++ STL 라이브러리**
    - 제네릭 프로그래밍을 지원하기 위해 템플릿 라이브러리

STL 라이브러리

C 라이브러리

C++ 입출력 라이브러리

algorithm	complex	exception	list	stack
bitset	csetjmp	fstream	locale	stdexcept
cassert	csignal	functional	map	strstream
cctype	cstdarg	iomanip	memory	streambuf
cerrno	cstddef	ios	new	string
cfloat	cstdio	iosfwd	numeric	typeinfo
ciso646	cstdlib	iostream	ostream	utility
climits	cstring	istream	queue	valarray
clocale	ctime	iterator	set	vector
cmath	deque	limits	sstream	

다음 수업

- C++ 프로그래밍의 기본
  - 1_ C++ 기본요소와 화면 출력
  - 2_ namespace
  - 3_ 입출력