C++ 프로그래밍

학습목표

- C++ 언어로 프로그램을 작성할 수 있다.

레퍼런스

레퍼런스(Reference)는 포인터 타입처럼 간접 참조를 위한 타입으로 포인터의 2가지 불편함을 해결하기 위해 등장했다. (1) 포인터는 널(Null) 값이 포함될 수 있다. (2) 간접 참조로 데이터를 다루기 위해선 역참조 연산자를 계속 사용해야 한다.

```
void Swap(int* lhs, int* rhs)
{
    // 포인터를 안전하게 사용하려면 항상 널 체크를 해줘야 한다.
    if (lhs == NULL || rhs == NULL)
    {
        return;
    }

    // 간접 참조를 하려면 * 연산자를 사용해야 한다.
    int temp = *lhs;
    *lhs = *rhs;
    *rhs = temp;
}

void Swap(int& lhs, int& rhs)
{
    // 레퍼런스는 널 값이 들어올 수 없기 때문에 널 체크를 하지 않아도 된다.

    // 더 이상 * 연산자를 사용하지 않아도 된다.
    int temp = lhs;
    lhs = rhs;
    rhs = temp;
}

int a, b;
```

```
Swap(&a, &b); // 포인터를 사용한 버전
Swap(a, b); // 레퍼런스를 사용할 때는 & 연산자를 굳이 적지 않아도 된다.
```

당연하게도 레퍼런스에 주소 연산은 불가능하다. 그리고 레퍼런스는 꼭 초기화를 해줘야 한다.

```
int& ref; // 컴파일 오류; 초기화를 꼭 해줘야 함.
```

객체지향 프로그래밍

C++에서는 객체지향 프로그래밍을 위한 여러가지 기능을 제공한다. 우선 객체의 청사진이 되는 클래스(Class)부터 알아보자.

클래스

클래스는 사용자 정의 타입으로 아래와 같은 멤버(Member)를 가질 수 있다. 멤버는 클래스의 구성 요소를 의미한다.

- 데이터 멤버
    - 데이터를 저장한다. 다시 말해 객체다.
    - 필드(Field)라고도 한다.
- 멤버 함수
    - 타입의 기능 부분이다.
    - 메소드(Method)라고도 한다.
- 내부 타입(Nested Type)
    - 클래스 안에 다른 클래스를 만들거나, 열거형, 혹은 타입 별칭을 지정할 수 있다.
- 멤버 템플릿
    - 템플릿을 작성할 수도 있다. 템플릿은 후술한다.

클래스 타입은 아래와 같이 정의한다.

```
// class 자리에는 struct, class, union 키워드를 작성할 수 있다.
class A
{
    // 클래스 내부에서 선언된 식별자는 클래스 범위(Class Scope)를 가진다.
    int  _data; // 데이터 멤버

    void foo(); // 멤버 함수

    // 내부 클래스(Nested Class)
    class B
    {
      int _data;
    };

    // 열거형
    enum { TEMP1, TEMP2, TEMP3 };

    // 타입 재정의
    typedef B MyNestedClass;
    using MyNestedClass = B; // 바로 위 구문과 동일하다.
};

// 클래스 타입의 변수는 기본 타입과 똑같이 정의할 수 있다.
// a를 A에 대한 인스턴스(Instance)라고 한다.
A a;
```

각 멤버는 **접근 한정자**(Access Specifier)를 통해 클래스 외부에 공개할 것과 공개하지 않을 것을 구분지어 사용자로 하여금 설계한 의도대로만 사용할 수 있게 강제할 수 있다. 접근 한정자에는 public, protected, private이 있다. protected는 후술할 상속 파트에서 살펴보도록 한다.

```
// class의 기본 접근 지정자는 private이다.
class A
{
    // 이 아래는 private 영역이다.
    // private은 클래스 외부에서 접근할 수 없다.
    int _privateData;

    // 이 아래는 public 영역이다.
    // public은 클래스 외부에서 접근할 수 있다.
public:
```

```
    int PublicData;
};

A a;

a._privateData; // 컴파일 오류; 접근 불가능
a.PublicData; // 접근 가능
```

class 키워드 대신 struct를 사용하면 기본 접근 지정자가 달라진다.

```
// struct의 기본 접근 지정자는 public이다.
struct B
{
  int PublicData;
};
```

- **인스턴스 멤버**

클래스가 가질 수 있는 멤버는 크게 **비정적 멤버**(Non-static Member)와 **정적 멤버**(Static Member)로 나눌 수 있다. 비정적 멤버는 인스턴스가 사용하는 멤버*이며, 정적 멤버는 인스턴스와 관계없이 사용하는 멤버다.

* 그래서 비정적 멤버를 인스턴스 멤버라고도 한다.

```
class A
{
  int _data;
  double  _data2;
};

// A 타입의 인스턴스 a는 메모리에 다음과 같이 표현된다.
// [ _data(4) ][ padding(4) ][ _data2(8) ]
A a;
```

비정적 멤버 함수에서는 인스턴스의 주소를 담고 있는 this 포인터라는 것으로 인스턴스의 데이터를 접근할 수 있다.

```
class A
```

```
{
public:
  void Add(int value, double value2)
  {
    // this 포인터로 인스턴스 데이터를 접근할 수 있다.
    this->_data += value;

    // this 포인터를 생략할 수도 있다.
    _data2 += value2;
  }

private:
  int _data;
  double  _data2;
};

A a;

// 인스턴스 메소드를 호출할 때 암시적으로
// 인스턴스의 주소값이 전달되어 this 포인터의 초기값으로 사용된다.
// 즉, 인스턴스 메소드의 첫 번째 매개변수는
// 클래스 타입의 포인터라고도 할 수 있다.
// Add(int value, double value2)는
// Add(A* const this, int value, double value2)인 것이다.
a.Add(1, 1.2); // 이 예시에서는 A::Add(&a, 1, 1.2)라고도 볼 수 있다.
```

비정적 멤버 함수에는 const 수식이 가능하며, const로 수식된 비정적 멤버 함수에서는 데이터 수정이 불가능하다.

```
class A
{
public:
   // const로 수식된 인스턴스 메소드
  void Foo() const
  {
    _data += 10; // 컴파일 오류! 데이터 수정 불가능
  }

  // 오버로딩이 가능하다.
  void Foo() { _data += 10; }
```

```
  void Boo() { _data += 20; }

private:
  int _data;
};

// const로 수식 되었다면 const 메소드만 호출 가능하게 된다.
const A a;
a.Foo(); // const로 수식된 Foo()만 호출 가능
a.Boo(); // 컴파일 오류!
```

- **특수한 멤버 함수**

비정적 멤버 함수 중 컴파일러가 자동으로 만들어주는 메소드가 있다. 이를 **기본 메소드**(Default Method)라고 한다. 기본 메소드에는 **기본 생성자**(Default Constructor), **복사 생성자**(Copy Constructor), **이동 생성자**(Move Constructor), **소멸자**(Destructor), **복사 할당 연산자**(Copy Assignment Operator), **이동 할당 연산자**(Move Assignment Operator)가 있다. 차례대로 살펴보자.

- **기본 생성자**

생성자는 인스턴스의 초기화를 담당한다.

```
class A
{
public:
    // 생성자의 문법은 기본적으로 반환타입을 갖지 않고,
    // 타입의 이름과 동일하게 식별자를 사용한다.
    // 매개변수가 없는 생성자를 기본 생성자라고 한다.
    A()
     : _data(0) // 이 부분을 멤버 초기화 목록(Member Initializer
List)이라고 한다.
    {
      // 초기화 외에 인스턴스를 만들면서 해야할 일들을 적는다.
    }

    // 생성자는 필요한만큼 오버로딩이 가능하다.
    A(int data) : _data(data) { }
```

```
private:
    int _data;
};

A a; // 기본 생성자가 호출된다. a._data는 0이다.
A b(10); // A(int data) 생성자가 호출된다. b._data는 10이다.
```

기본 생성자는 클래스 안에 아무런 생성자도 적어주지 않을 시 자동으로 합성된다.

```
class A
{
    // 생성자를 정의해주지 않았지만 자동으로 컴파일러에 의해 만들어진다.
    // 이 때 자동으로 만들어지는 생성자는 아래와 같다.
    // A() { }
    int _data;
};

// 멤버 중에 클래스 타입이 있는 경우 기본 생성자를 호출하게 된다.
A a;

class B
{
public:
    // 생성자를 하나 만들어주었으므로
    // 기본 생성자는 더 이상 만들어지지 않는다.
    B(int data)
     : _data(data)
    { }

private:
    int _data;
};

B b; // 컴파일 오류; 기본 생성자가 없으므로 만들 수 없음.
```

모든 생성자는 기본적으로 **변환 생성자**(Converting Constructor)라고 한다. 다시 말해 생성자에 사용한 매개변수 목록에서 클래스 타입으로 암시적 변환이 일어난다.

```
class A
```

```
{
public:
    A(int data)
     : _data(data) { }

private:
    int _data;
};

void Foo(A a);

// int 타입에서 A 타입으로의 암시적 변환이 일어남.
// void Foo(int); 함수가 있다면 의도하지 않은 결과가 일어날 수 있음.
Foo(10);
```

암시적 변환이 있어 사용이 편하기도 하지만, 때에 따라 실수로 이어질 수도 있다. 만일 이런 암시적 변환을 원치 않는다면 explicit 한정자를 사용할 수 있다.

```
class A
{
public:
    // 더 이상 암시적 변환이 불가능하며 모든 코드에서 명시적으로
    // 생성자를 호출해야 한다.
    explicit A(int data)
        : _data(data) { }

private:
    int _data;
};

void Foo(A a);

Foo(10); // 컴파일 오류! 암시적 변환 불가능.
Foo(A(10)); // 사용 가능. 명시적으로 표기해줌.
```

explicit 한정자를 사용하면 암시적 변환으로 인해 발생할 수 있는 오류를 줄일 수 있다. 즉, 좀 더 안전한 코드를 작성할 수 있다.

- **복사 생성자**

또한 생성자 중 동일 타입의 레퍼런스를 인자로 받는 생성자를 **복사 생성자**(Copy Constructor)라고 한다.

```
class A
{
public:
    A(int data) : _data(data) { }

    // 복사 생성자의 용도는 말 그대로 다른 인스턴스의
    // 데이터를 완전히 복사하는 것이다.
    A(const A& other) : _data(other._data) { }

private:
    int _data;
};

A a(10);
A a2(a); // 복사 생성자 호출. a2._data는 a._data와 동일하다.
A a3 = a; // 이 경우에도 복사 생성자가 호출된다.
```

복사 생성자도 기본 생성자와 마찬가지로 아무런 복사 생성자가 정의되어 있지 않을 시 자동으로 합성된다.

```
class A
{
public:
    A(int data) : _data(data) { }

    // 복사 생성자를 만들지 않았으므로
    // 컴파일 시 기본 복사 생성자가 만들어진다.
    // A(const A& other) { memcpy(this, &other, sizeof(A)); }

private:
    int _data;
};

A a = 10; // 암시적 변환.
A b = a; // 기본 복사 생성자가 호출된다.
```

특히 멤버에 포인터가 있는 경우 복사 생성자를 만들 때 주의해야 한다. 아래 예시를 보자.

```
class A
{
public:
    A() : _p(malloc(sizeof(int) * 3)) { }

    void Foo(int index, int value)
    {
      _p[index] = value;
    }
private:
    int* _p;
};

A a;
A a2 = a; // 주의! a._p와 a2._p가 같은 메모리를 가리키고 있음.

a2.Foo(2, 10); // 주의! a도 영향을 받는다.
```

a와 a2는 서로 다른 인스턴스임에도 불구하고 각자의 _p가 같은 메모리 영역을 가리키고 있어 서로 영향을 주고 있다는 것을 알 수 있다. 이를 **얕은 복사**(Shallow Copy)라고 한다. 따라서 얕은 복사를 의도한 게 아니라면 아래와 같이 바꿔줘야 한다.

```
class A
{
public:
    A() : _p(malloc(sizeof(int) * 3)) { }

    A(const A& other)
      : _p(malloc(sizeof(int) * 3))
    {
      for (int i = 0; i < 3; ++i)
      {
        _p[i] = other._p[i];
      }
    }

    void Foo(int index, int value)
```

```
    {
        _p[index] = value;
    }
private:
    int* _p;
};

A a;
A a2 = a; // 이제는 서로 다른 영역을 가리킨다.
a2.Foo(2, 10); // 이제는 a에게 영향을 주지 않는다.
```

복사 생성자에서 서로 다른 메모리 영역을 가리키되 데이터는 똑같게 만들어줬다. 이를 **깊은 복사**(Deep Copy)라고 한다.

- **소멸자**

소멸자는 객체의 수명이 다할 때 자동으로 호출되는 특수한 메소드로 자원을 정리하기 위해 사용한다.

```
class A
{
public:
    A() : _p(malloc(sizeof(int) * 3)) { }

    // 소멸자도 생성자와 마찬가지로 반환 타입이 없으며
    // 식별자는 타입 이름 앞에 ~를 붙이고
    // 아무런 매개변수도 적지 않는다.
    // 보통 이와 같이 자원을 정리할 때 사용한다.
    ~A() { free(_p); _p = nullptr; }
private:
    int*    _p;
};

{
    A a;
} // 이 부분에서 a.~A(); 가 호출된다.
```

소멸자도 정의되지 않으면 자동으로 컴파일러에 의해 합성된다.

```
class A
{
private:
    int _data;
};

{
    A a; // a의 기본 생성자가 호출된다.
} // a의 기본 소멸자가 호출된다.
```

- **복사 할당 연산자**

C++에서는 **연산자 오버로딩**(Operator Overloading)이 가능하다. 그 중 하나의 매개변수만 가지며 그 매개변수의 타입이 클래스 타입과 동일한 할당 연산자를 복사 할당 연산자라고 한다.

```
// 복사 할당 연산자를 구현하는 방법에는 여러가지가 있지만
// 그 중 하나인 복사 후 교환 관용구(Copy-and-Swap Idiom)에 대해서 살펴본다.
class A
{
    int _data;
public:
    // 복사 후 교환 관용구를 사용하려면 매개변수의 타입은 레퍼런스가 아니어야 한다.
    A& operator=(A a)
    {
        // std::swap()을 이용해 멤버 별로 교체해준다.
        std::swap(_data, a._data);

        return *this;
    }
};
```

이런 기본 메소드에 대해서는 default와 delete 키워드를 사용해 컴파일러가 제공하는 기본 함수를 사용하거나 삭제할 수 있다.

```
class A
{
public:
    A() = default; // 컴파일러가 제공하는 기본 생성자 사용
```

```
    ~A() = default; // 컴파일러가 제공하는 기본 소멸자 사용
    A(const A&) = delete; // 복사 생성자 삭제. A 타입의 객체는 복사 불가능.
    A& operator=(const A&) = delete; // 복사 할당 연산자 삭제. A 타입의
객체는 복사 불가능.
private:
    int _data = 10; // C++11부터 기본 멤버 초기자(Default Member
Initializer)가 제공됨
};

A a;
A b(a); // 컴파일 오류! 복사 생성자 삭제됨.
A c;
c = a; // 컴파일 오류! 복사 할당 연산자 삭제됨.
```

- **정적 멤버**

멤버를 사용할 때 static 키워드를 붙이면 정적 멤버가 된다. 정적 멤버는 프로세스 주소 공간 중 스택이나 힙이 아닌 데이터 영역을 사용하며, 타입 별로 단일의 멤버로 존재하게 된다.

```
class A
{
  static int s_data; // 정적 데이터 멤버. 이 부분은 선언이다.
  static const int CONST = 10; // 상수는 정의가 가능하다.

  int _data;
public:
  // 정적 멤버 함수
  // 인스턴스 메소드가 아니기에 const 수식이 불가능하다.
  static void Foo()
  {
    _data = 10; // 컴파일 오류! 정적 메소드기 때문에 this 포인터가 존재하지
않는다.
    Boo(); // 컴파일 오류! 마찬가지로 this 포인터가 존재하지 않는다.

    ++s_data;
  }

  void Boo()
  {
    ++_data;
```

```
    Foo(); // 인스턴스 메소드에서 정적 메소드는 호출 가능하다.
  }
};

int A::s_data = 0; // 정적 데이터 멤버 정의

A a, a2;
a.Foo(); // A::s_data == 1
a2.Foo(); // A::s_data == 2

A::Foo(); // 단독으로도 호출 가능하다.
```

상속

C++에서 상속을 하고 싶다면 클래스를 정의할 때, 뒤에 콜론을 붙이고 상속할 클래스를 적어주면 된다. 상속할 클래스를 적을 때 접근 지정자를 함께 적어줄 수 있는데, 여기서는 public에 대해서만 다룬다. 그 외의 것은 범위를 벗어나므로 여기서 다루지 않는다.*

*하지만 찾아보고 싶은 사람은 여기를 참고하라.

```
class Base
{
    // 생략
};

// 상속할 클래스를 : 뒤에 적어준다.
class Derived : public Base
{
    // 생략
};
```

상속을 받을 때는 객체의 생성과 소멸 순서가 어떻게 되는지 명확히 알고 있는 것이 중요하다.

```
class Base
{
public:
    Base() { std::cout << "Base Constructor\n"; }
    ~Base() { std::cout << "Base Destructor\n"; }
};
```

```
// 상속할 클래스를 : 뒤에 적어준다.
class Derived : public Base
{
public:
    Derived() { std::cout << "Derived Constructor\n"; }
    ~Derived() { std::cout << "Derived Destructor\n"; }
};

{
    // 생성은 부모 클래스부터 자식 클래스 순으로 하게 된다.
    Derived d;
} // 소멸은 반대로 자식 클래스에서 부모 클래스 순으로 하게 된다.
```

- **protected 접근 지정자**

protected 접근 지정자는 클래스 내부 및 자식 클래스에게 접근 권한을 부여한다.

```
class Base
{
protected:
    int _baseData;

private:
    int _privateData;
};

// 상속할 클래스를 : 뒤에 적어준다.
class Derived : public Base
{
public:
    void Foo()
    {
        _baseData = 10; // 접근 가능
        _privateData = 20; // 컴파일 오류! private은 자식 클래스에서도 접근 불가능
    }
};
```

- **public 상속의 의미**

public 상속은 부모 클래스의 모든 public 멤버를 자식 클래스의 public 멤버로 만들고, 부모 클래스의 모든 protected 멤버를 자식 클래스의 protected 멤버로 만든다.

- **다중 상속**

상속할 클래스가 꼭 한 개일 필요는 없다. 여러 개를 상속 받는 것도 가능하다.

```
class Base1 { };
class Base2 { };

// 다중 상속을 할 때는 ,로 적어주면 된다.
class Derived : public Base1, public Base2
{
    // 생략
};
```

다만 다중 상속에는 큰 문제가 있다. 아래를 보자.

```
class Base1
{
public:
    void Foo() { std::cout << "Base1의 Foo()\n"; }
};

class Base2
{
public:
    void Foo() { std::cout << "Base2의 Foo()\n"; }
};

class Derived : public Base1, public Base2
{

};

Derived d;
d.Foo(); // 컴파일 오류! Base1::Foo()와 Base2::Foo() 중 누굴 호출할 것인지 모호함.
```

이런 경우를 [죽음의 다이아몬드](#)(The Deadly Diamond of Death) 현상이라고 한다.

- **가상 함수**

가상 함수(Virtual Function)는 다형성을 지원하기 위한 기능이다. 가상 함수를 작성하려면 메소드 앞에 virtual 한정자를 적는다.

```
class Base
{
public:
    // 가상 함수는 앞에 virtual 키워드를 붙이면 된다.
    // Foo()는 가상 함수다.
    virtual void Foo()
    {
        std::cout << "Base::Foo()\n";
    }
};
```

가상 함수의 내용을 재정의하는 것을 **오버라이딩**(Overriding)이라고 한다.

```
class Base
{
public:
    virtual void Foo()
    {
        std::cout << "Base::Foo()\n";
    }
};

class Derived : public Base
{
public:
    // 가상 함수 재정의는 그냥 정말 재정의를 해주면 된다.
    void Foo()
    {
        std::cout << "Derived::Foo()\n";
    }
};
```

그럼 가상 함수를 작성하기만 하면 다형성을 사용할 수 있는 걸까? 그것은 아니다. 부모 클래스 타입을 가리키는 포인터나 레퍼런스로 **업캐스팅**(upcasting)*하여 다뤄야 다형성이 적용된다.

* 자식 클래스 타입에서 부모 클래스 타입으로 변환하는 것을 말한다. 반대는 **다운캐스팅**(downcasting)이다.

```
class Base
{
public:
    virtual void Foo()
    {
        std::cout << "Base::Foo()\n";
    }
};

class Derived : public Base
{
public:
    // 가상 함수 재정의는 그냥 정말 재정의를 해주면 된다.
    void Foo()
    {
        std::cout << "Derived::Foo()\n";
    }
};

Base b;
Derived d;

Base& b1 = d; // 업캐스팅
b1.Foo(); // "Derived::Foo()"
```

그럼 이런 것이 어떻게 동작하는 것일까? 우리가 일반적으로 작성하는 함수는 컴파일 시간에 어떤 함수를 호출할 것인지 결정하는 **정적 바인딩**(Static Binding)인 반면에, 가상 함수는 실행 시간에 어떤 함수를 호출할 것인지 결정하는 **동적 바인딩**(Dynamic Binding)으로 동작한다. 동적 바인딩을 위해 가상 함수의 주소가 저장되어 있는 **가상 함수 테이블**(Virtual Function Table)이 각 타입마다 존재하게 되며, 각 인스턴스마다 이를 자기 타입의 가상 함수 테이블을 가리키는 **가상 함수 포인터**(Virtual Function Pointer)를 갖고 있게 된다.

```
#include <iostream>

using namespace std;

struct Base
{
  // Base 타입의 가상 함수 테이블은 아래와 같이 구성된다.
```

```
  // vftable[0] = Base::Foo
  // vftable[1] = Base::Boo
  virtual void Foo() { cout << "Base Foo\n"; }
  virtual void Boo() { cout << "Base Boo\n"; }

  // Coo는 가상 함수가 아니기 때문에
  // 가상 함수 테이블에 들어가지 않는다.
  void Coo() { cout << "Base Coo\n"; }

// 가상 함수가 존재하기 때문에 아래와 같이
// 모든 인스턴스는 가상 함수 포인터를 갖게 된다.
// private: void** __vfptr = Base::vftable;

// 가상 함수 포인터는 인스턴스가 생성될 때 초기화 되며
// 초기값은 각 타입의 가상 함수 테이블이 된다.
};

struct Derived : Base
{
  // Derived의 가상 함수 테이블은 아래와 같이 구성된다.
  // vftable[0] = Derived::Foo
  // vftable[1] = Base::Boo
  // vftable[2] = Derived::Aoo
  void Foo() { cout << "Derived Foo\n"; }
  virtual void Aoo() { cout << "Derived Aoo\n"; }

  void Coo() { cout << "Derived Coo\n"; }
};

Base b;
Base* p = &b;
p->Foo(); // Base Foo
p->Boo(); // Base Boo
p->Coo(); // Base Coo

Derived d;
p = &d;
p->Foo(); // Derived Foo
p->Boo(); // Base Boo

// 가리키고 있는 타입이 Base이므로 Base::Coo가 호출된다.
p->Coo(); // Base Coo.
```

```
// 다운캐스팅을 하여 Derived::Coo가 호출된다.
((Derived*)p)->Coo(); // Derived Coo
```

- **추상 클래스**

추상 클래스(Abstract Class)는 인스턴스를 만들 수 없는 클래스로 상위 타입을 정의하는 데 사용한다.

```
class AbstractClass
{
  // 아래와 같은 가상 함수를 순수 가상 함수(Pure Virtual Function)라 하며
  // 보통 선언만 한다.
  virtual void Foo() = 0;
}

AbstractClass obj; // 컴파일 오류! 추상 타입은 인스턴스를 생성할 수 없다.
```

순수 가상 함수는 하위 타입으로 하여금 오버라이딩을 강제하는 효과가 있다.

```
// Abstract는 추상 클래스
struct Abstract
{
  virtual void Foo() = 0;
};

struct Base : Abstract
{
  // 순수 가상 함수를 오버라이딩 하지 않으면
  // Abstract::Foo()를 그대로 Base가 갖게 되며,
  // Base 또한 추상 클래스가 된다.
};

// Derived는 Foo()를 오버라이딩 하였으므로
// 추상 클래스가 아닌 구체 클래스(Concrete Class)다.
struct Derived : Base
{
  void Foo() { }
};
```

```
Abstract temp; // 컴파일 오류! Abstract는 추상 타입
Base temp; // 컴파일 오류! Base는 추상 타입
Derived d; // OK. Derived는 구체 타입
```

- **동적 할당**

객체지향 프로그래밍을 지원하면서 동적 할당의 방법도 바뀌게 되었다. 왜냐하면 클래스 타입의 객체를 동적 할당할 때, 생성자와 소멸자를 호출해야 하기 때문이다.

```
struct A
{
  A() { std::cout << "Constructor"; }
  ~A() { std::cout << "Desturctor"; }
};

int main()
{
  int* p = new int; // 할당에는 new 연산자를
  delete p; // 해제에는 delete 연산자를 사용한다.

  p = new int[3]; // 배열의 할당이 필요할 때는 new[] 연산자를 사용하며
  delete[] p; // 해제에는 delete[] 연산자를 사용한다.

  // 가장 크게 달라지는 부분은 클래스 타입을 동적 할당 할 때이다.
  // new / delete 연산자는 클래스 타입일 때
  // 자동으로 생성자와 소멸자를 호출한다.
  A* a = new A();
  delete a;

  return 0;
}
```

new / delete 연산자의 쌍은 꼭 맞춰줘야 하는데, 동작 방식이 서로 다르기 때문이다.

- **가상 소멸자**

동적 할당을 알게 됨으로써 문제점이 있다. 아래를 보자.

```
struct Base
```

```
{
  Base() { std::cout << "Constructor"; }
  ~Base() { std::cout << "Destructor"; }
};

struct Derived : Base
{
  Derived() { std::cout << "Constructor"; }
  ~Derived() { std::cout << "Destructor"; }
};

{
  Derived d;
} // 정상적으로 생성과 소멸이 이뤄짐.
```

위와 같은 경우에는 객체의 생성과 소멸이 잘 이뤄진다. 하지만 동적할당일 땐 어떨까?

```
Base* b = new Derived();
delete b; // ~Base()만 호출됨.
```

보통 객체지향에선 업캐스팅하여 객체를 다루게 되는데, 이 경우 객체의 소멸이 제대로 안되는 것을 확인할 수 있다. 왜냐하면 결국 상위 타입으로 다루고 있기 때문이다. 따라서 올바르게 소멸자를 호출하고 싶다면 **가상 소멸자**(Virtual Destructor)를 정의해야 한다.

```
struct Base
{
  Base() { std::cout << "Constructor"; }
  virtual ~Base() { std::cout << "Destructor"; }
};

struct Derived : Base
{
  Derived() { std::cout << "Constructor"; }
  ~Derived() { std::cout << "Destructor"; }
};

Base* b = new Derived();
delete b; // 이제는 올바르게 소멸된다.
```

따라서 상위 타입이 될 여지가 있는 클래스라면 꼭 가상 소멸자를 정의해주도록 하자.

일반화 프로그래밍

일반화 프로그래밍(Generic Programming)은 타입에 관계없이 알고리즘을 기술하는 프로그래밍 패러다임이다. C++에서는 이를 **템플릿**(Template)이라는 기능으로 지원하고 있다. 이 파트에서는 템플릿에 대해 알아보고자 한다.

템플릿 문법

템플릿은 기본적으로 **클래스 템플릿**(Class Template)과 **함수 템플릿**(Function Template)으로 구분된다.*

* C++14부터는 별칭 템플릿(Alias Template), 변수 템플릿(Variable Template), 컨셉(Concept) 등을 만들 수 있다.

```
// 문법은 아래와 같이 구성된다.
// template <parameter-list> declaration

// 클래스 템플릿을 정의했다.
template <typename T>
struct A { T Data; };

// 함수 템플릿을 정의했다.
template <typename T>
void Swap(T& lhs, T& rhs)
{
    T temp = lhs;
    lhs = rhs;
    rhs = lhs;
}

// 템플릿을 사용하려면 매개변수 목록에 인자를 전달하면 된다.
// 이를 특정 타입(Specific Type)으로 대체한다고 하여
// 특수화(Specialization)라고 한다.
A<int> a; // a는 int 타입의 Data를 가진다.
A<double> b; // b는 double 타입의 Data를 가진다.

int a = 10, b = 20;
Swap<int, int>(a, b);
Swap(a, b); // 함수 템플릿은 <>를 생략할 수 있다.
```

템플릿은 하나 이상의 **템플릿 파라미터**(Template Parameter)와 함께 매개변수화 된다. 템플릿 파라미터에는 **비 타입 파라미터**(Non-type Parameter), **타입 파라미터**(Type Parameter), **템플릿 파라미터**(Template Parameter)가 있다. 정리하자면 타입도 마치 인자처럼 전달할 수 있다는 것이다.

```
// 타입 파라미터는 타입을 받을 수 있다.
template <typename T> class A { };
template <class T> class B { }; // typename 대신 class를 사용할 수 있다.
template <typename T = int> class C { }; // 기본 인자 전달도 가능하다.
// C<> c; c는 C<int> 타입이다.

// 비 타입 파라미터는 구체적인 타입을 지정하는 것이다.
// 비 타입 파라미터에 넣을 수 있는 타입으로는
// 정수형, 포인터, 레퍼런스 등이 있다.
// 마찬가지로 기본 인자 전달이 가능하다.
template <size_t N = 10> struct Array { int Container[N]; };

// 템플릿 파라미터는 템플릿을 전달하는 것이다.
template <typename T> class Container { };
template <template<typename> typename Container = Container<int>>
class Temp { };
```

특수화

특수화는 컴파일 타임에 일어나며, 컴파일러가 템플릿을 기반으로 특정 타입을 생성한다.

```
template <typename T, size_t N>
struct IdiotArray { T Container[N]; };

IdiotArray<int, 10> arr;
// 컴파일러는 컴파일 타임에 아래와 같은 코드를 생성한다.
// template <>
// struct IdiotArray { int Container[10]; };
```

따라서 일반적으로 템플릿은 헤더 파일에 모든 내용을 적으며, 정의를 소스 파일에 나눠서 적으면 링크 오류가 발생한다.

```
// Temp.h
```

```cpp
template <typename T>
class A
{
public:
    void SetData(const T& data);
    T GetData() const;
private:
    T _data;
};

// Temp.cpp
// 이 파일 아래에 있는 템플릿은 인스턴스가 만들어지지 않는다.
template <typename T>
void A<T>::SetData(const T& data)
{
    _data = data;
}

template <typename T>
T A<T>::GetData() const
{
    return _data;
}

// Main.cpp
int main()
{
    A<int> a;
    // 아래와 같이 인스턴스가 만들어진다.
    // 정의가 없음을 볼 수 있다.
    // template <>
    // class A
    // {
    // public:
    //     void SetData(const int& data);
    //     int GetData() const;
    // private:
    //     int _data;
    // };

    a.SetData(10); // 링크 오류! 정의가 없음.
}
```

사용자가 특정 템플릿 인자에 대해 코드를 커스터마이징 할 수도 있는데, 모든 템플릿 파라미터에 대해 특수화를 하는 걸 **명시적 특수화**(Explicit Specialization), 일부만 하는 걸 **부분 특수화**(Partial Specialization)라고 한다.

```
template <typename T1, typename T2>
void Print(const T1& a, const T2& b)
{
    std::cout << a << '\n';
    std::cout << b << '\n';
}

// 명시적 특수화
template <>
void Print(const int& a, const double& b)
{
    std::cout << "이것은 명시적으로 특수화 되었습니다.\n";
}

// 부분 특수화
template <typename T>
void Print(const T& a, const int& b)
{
    std::cout << "이것은 부분적으로 특수화 되었습니다.\n";
}

double d = 1.0;
Print(d, d); // Print<double, double>; 암시적 특수화 버전 호출

int i = 1;
Print(i, d); // Print<int, double>; 명시적 특수화 버전 호출

Print(d, i); // Print<double, int>; 부분 특수화 버전 호출
```

STL

STL의 사용법에 대해서 알아보자

<vector>

```cpp
#include <iostream>
#include <vector>

using namespace std;

int main()
{
    // 벡터 생성하기
    vector<int> vec;
    cout << boolalpha << vec.empty() << endl;
    cout << vec.size() << endl;
    cout << vec.capacity() << endl;

    // 삽입
    // push_back : 벡터의 맨 끝에 데이터 삽입
    vec.push_back(3);
    vec.push_back(4);
    vec.push_back(2);
    // { 3, 4, 2 }

    // 반복자
    // begin() : 컨테이너의 첫 원소를 가리키는 반복자
    // end() : 컨테이너의 마지막 원소의 다음을 가리키는 반복자
    for (vector<int>::iterator iter = vec.begin(); iter != vec.end();
++iter)
    {
        cout << *iter << ",";
    }
    cout << endl;

    // rbegin() : 컨테이너의 마지막 원소를 가리키는 반복자
    // rend() : 컨테이너의 첫 원소의 이전을 가리키는 반복자
    for (vector<int>::reverse_iterator iter = vec.rbegin(); iter !=
vec.rend(); ++iter)
    {
        cout << *iter << ",";
    }
    cout << endl;

    // insert : pos 이전에 데이터를 삽입
    vector<int>::iterator iter = vec.begin();
    ++iter; // 2번째 원소 : 4
```

```
    iter = vec.insert(iter, 10); // vec : { 3, 10, 4, 2 }, iter :
&vec[1] (10)
    iter = vec.insert(iter, 3, 5); // vec : { 3, 5, 5, 5, 10, 4, 2 },
iter : &vec[1] (5)
    iter = vec.insert(iter, vec.begin() + 3, vec.end() - 1); // vec :
{ 3, 5, 10, 4, 5, 5, 5, 10, 4, 2 }
                                        // iter : &vec[1] (5)
    for (vector<int>::iterator iter = vec.begin(); iter != vec.end();
++iter)
    {
        cout << *iter << ",";
    }
    cout << endl;

    // 삭제
    // pop_back : 맨 끝에 있는 데이터를 삭제
    // vec : { 3, 5, 10, 4, 5, 5, 5, 10, 4, 2 }
    vec.pop_back();
    vec.pop_back();
    vec.pop_back();
    // vec : { 3, 5, 10, 4, 5, 5, 5 }

    // erase : pos에 있는 데이터를 삭제
    vec.erase(vec.begin());
    // vec : { 5, 10, 4, 5, 5, 5 }
    vec.erase(vec.begin() + 1, vec.begin() + 3);
    // vec : { 5, 5, 5, 5 }

    // 읽기
    cout << vec.front() << endl; // 5
    cout << vec.back() << endl; // 5
    cout << vec[2] << endl; // 5

    // 검색 => 메소드가 아니라 알고리즘 라이브러리에 존재

    // 다른 생성
    vector<int> vec2(5); // { 0, 0, 0, 0, 0 }
    vector<int> vec3(5, 10); // { 10, 10, 10, 10, 10 }

    return 0;
}
```

참고자료

- cppreference.com
- 5 일반화 프로그래밍(Generic Programming) - C++ Template Note
- Generic programming - Wikipedia