Destructor

- The destructor ensures proper cleanup of objects
- In particular, many classes use some memory from the free store that is allocated by a constructor and de-allocate by a destructor
- 사용자가 정의 하지 않으면 컴파일러가 자동으로 생성함
- 오직 하나만 존재하며 파라미터는 없음

Destructor

```
# include <iostream>
using namespace std ;
class Point {
  int x, y ;
public:
  Point(int x=0, int y=0) {
    this->x = x ; this->y = y ;
    cout << "₩tPoint " ; print() ; cout << " constructed." << endl ;
  }
  ~Point() { cout << "₩tPoint " ; print() ; cout << " destructed." << endl ; }
  void print() const { cout << "(" << x << ", " << y << ")" ; }
} ;
```

Destructor

```
Point gP(100, 100) ;
int main() {
   cout << "P1₩n" ;
   Point p1 ;

   Point* pP2 ;
   {
     cout << "₩nP2₩n" ;
     Point p3(3, 3) ;
     cout << "₩nP3₩n" ;
     pP2 = new Point(2, 2) ;
   }
   delete pP2 ;
}
```

```
Point (100, 100) constructed.
P1
Point (0, 0) constructed.

P2
Point (3, 3) constructed.

P3
Point (2, 2) constructed.
Point (3, 3) destructed.
Point (2, 2) destructed.
Point (0, 0) destructed.
Point (100, 100) destructed.
```

Destructor of Member Objects

```
class Point {
   int x, y ;
public:
   Point(int x=0, int y=0) {
      this->x = x ; this->y = y ;
      cout << "\tPoint " ; print() ; cout << " constructed." << endl ;
   }
   ~Point() { cout << "\tPoint " ; print() ; cout << " destructed." << endl ; }
   void print() const { cout << "(" << x << ", " << y << ")" ; }
} ;
class Rectangle {
   Point rightBottom, leftTop  ;
public:
   Rectangle(const Point& p1, const Point& p2=Point(0,0))
      : leftTop(p1), rightBottom(p2) {
         cout << "Rectangle: " ; print() ; cout << " constructed." << endl ;
   }
   Rectangle(int x1, int y1, int x2=0, int y2=0)
      : leftTop(x1, y1), rightBottom(x2, y2) {
      cout << "Rectangle: " ; print() ; cout << " constructed." << endl ;
   }
   Rectangle() { cout << "Rectangle: " ; print() ; cout << " constructed." << endl ; }
   ~Rectangle() { cout << "Rectangle: " ; print() ; cout << " destructed." << endl ; }
   void print() const { leftTop.print() ; rightBottom.print() ; }
} ;
```

Destructor of Member Objects

```
int main() {
    Point p(2, 2) ;
    Rectangle r2(p) ;
}
```

```
        Point (2, 2) constructed.
        Point (0, 0) constructed.
Rectangle: (2, 2)(0, 0) constructed.
        Point (0, 0) destructed.
Rectangle: (2, 2)(0, 0) destructed.
        Point (2, 2) destructed.
        Point (0, 0) destructed.
        Point (2, 2) destructed.
```

생성자/소멸자 호출 순서 요약

```
class Rectangle {
  //멤버 변수 선언 순서대로 초기화 및 소멸됨
  Point rightBottom ;
  Point leftTop  ;
public:
  Rectangle(int x1, int y1, int x2=0, int y2=0)
    : leftTop(x1, y1), rightBottom(x2, y2)
    { /*  */ }
  ~Rectangle() { /*  */ }
} ;
```

Rectangle 생성 시

rightBottom
leftTop
Rectangle() 본문

Rectangle 소멸 시

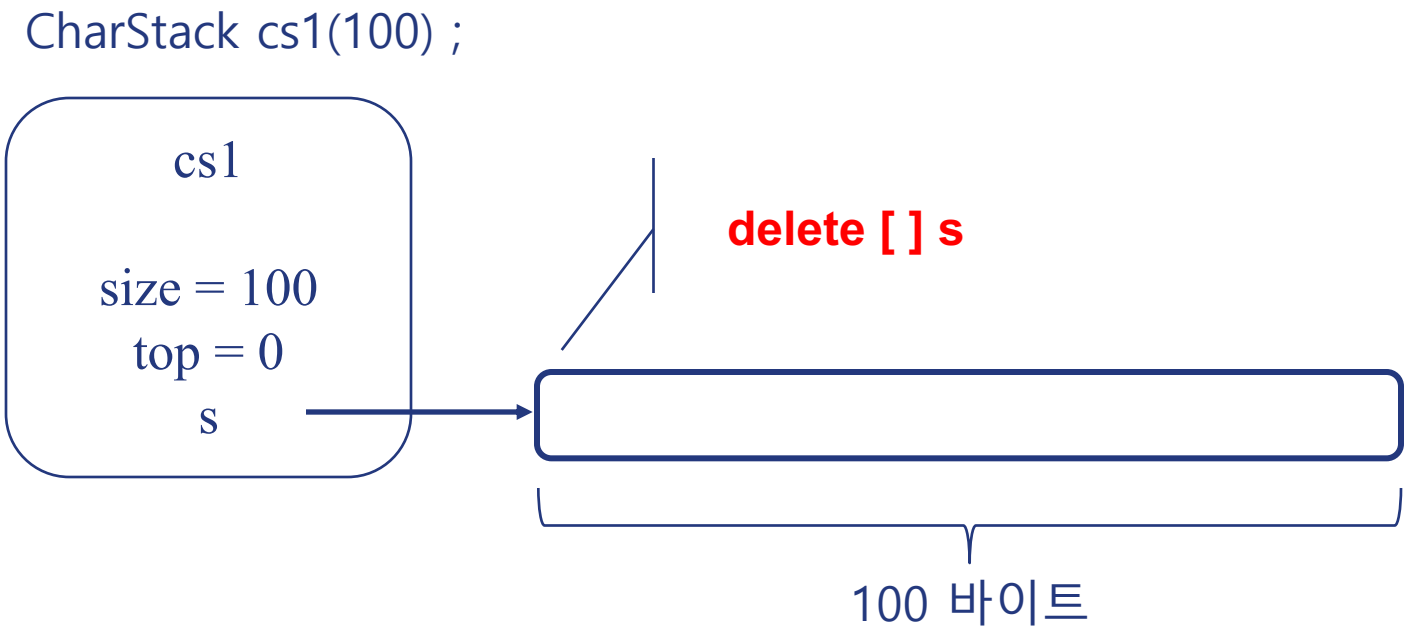
~Rectangle()
leftTop
rightBottom

소멸자의 역할

```
#include <iostream>
using namespace std ;
class CharStack {
    int size ;
    int top ;
    char* const s ;
public:
   CharStack(int sz) : size(sz), s(new char[sz]) {
      top = 0 ;
      cout << "constructor invoked for Stack[" << size << "]" << endl ;
    }
    CharStack(const CharStack& another)
      : size(another.size), top(another.top), s(new char[another.size]) {
      for ( int i = 0 ; i <= top ; i ++ ) s[i] = another.s[i] ;
    }
    ~CharStack() {
      delete [] s ;
      cout << "destructor invoked for Stack[" << size << "]" << endl ;
    }
    …
} ;
```

```
int main() {
    CharStack cs1(100) ;
    for ( int i=0; i < 10; i++ ) {
        CharStack cs2(200) ;
        …
    }
}
```

소멸자의 역할



소멸자의 역할

- What if no destructor for CharStack?

```
#include <iostream>
using namespace std ;
class CharStack {
    int size ;
    int top ;
    char* const s ;
public:
   CharStack(int sz) : size(sz), s(new char[sz]) {
      top = 0 ;
      cout << "constructor invoked for Stack[" << size << "]" << endl ;
    }
    CharStack(const CharStack& another)
      : size(another.size), top(another.top), s(new char[another.size]) {
      for ( int i = 0 ; i <= top ; i ++ ) s[i] = another.s[i] ;
    }
    ~CharStack() {
      delete [] s ;
      cout << "destructor invoked for Stack[" << size << "]" << endl ;
    }
    …
} ;
```

```
int main() {
    for ( int i=0; i < 1000; i++) {
        CharStack cs(1000) ;
        …
        // 1K Leaked
    }  // 1K * 1K Leaked
}
```

Good Design :
Does this class need a destructor?

- 클래스가 멤버 소멸자의 일부가 아닌 코드를 실행해야 하는 경우에만 기본이 아닌 소멸자를 정의하십시오.
- 사용자 정의 소멸자가 필요한 클래스에는 두 가지 일반 범주가 있습니다.
  - 리소스가 있는 클래스로 아직 소멸자가 표현되지 않은 클래스 (예 : 벡터 또는 트랜잭션 클래스).
  - 주로 소멸시 어떤 액션을 실행하는 것이 존재하는 클래스 (트레이싱 등)
- 기본 소멸자가 더 잘 동작하고, 더 효과적이며, 틀리지 않는다.

```
class Foo {   // bad; use the default destructor
    public:
        // ...
        ~Foo() { s = ""; i = 0; vi.clear(); }  // clean up
    private:
        string s;
        int i;
        vector<int> vi;
    };
```

Good Design :
destructor must not fail

- 소멸자가 실패할 때 에러 없는(error-free) 코드를 작성하는 방법을 우리는 모름
- 표준 라이브러리는 모든 클래스에 예외(except)로 종료되지 않는 소멸자가 있어야 한다고 요구함
- 소멸자를 noexcept로 선언하라
  - 프로그램이 정상적으로 완료되거나 프로그램이 종료되는 것을 보장함

```
class X {
public:
    ~X() noexcept;
    // ...
};
```

```
X::~X() noexcept {
    // ...
    if (cannot_release_a_resource)  terminate();
}
```

Good Design (RAII)

- Resource Acquisition Is Initialization(RAII) binds the life cycle of a **resource that must be acquired before use** (allocated **heap memory**, thread of execution, open socket, **open file**, locked mutex, disk space, database connection, and et. al..)
- It also guarantees that **all resources are released** when the lifetime of their controlling object ends, in reverse order of acquisition
- RAII encapsulates each resource into a class, where
  - **the constructor** acquires the resource and establishes all class invariants or throws an exception if that cannot be done,
  - **the destructor** releases the resource and never throws exceptions;
- RAII always uses the resource via an instance of a RAII-class that either
  - has automatic storage duration or temporary lifetime itself, or
  - has lifetime that is bounded by the lifetime of an automatic or temporary object

기본 연산(default operators) 생성 정리

- default, delete로 선언된 기본 연산이 있으면 다른 연산을 생성하지 않을 수 있으므로, 6개를 모두 명시적으로 선언하거나 아무 연산도 선언하지 않는다.
- 생성규칙
  - 클래스(class)의 멤버 변수 중 하나라도 생성 가능한 기본 연산이 존재하지 않으면, 해당 클래스(the class)의 기본 연산도 생성되지 않음 (클래스와 멤버 변수들의 기본연산에서 교집합만 생성됨)
  - 레퍼런스 타입 (기본 생성자로 생성 안됨), unique_ptr (복사 연산이 없음)
  - 소멸자를 생성하지 않으면 무조건 컴파일러가 생성한다.
  - 기본 생성자는 다른 생성자가 존재하면 생성되지 않는다. 따라서 기본 생성자를 정의하거나 명시적으로 default 선언해야 한다.

기본 연산(default operators) 생성 정리

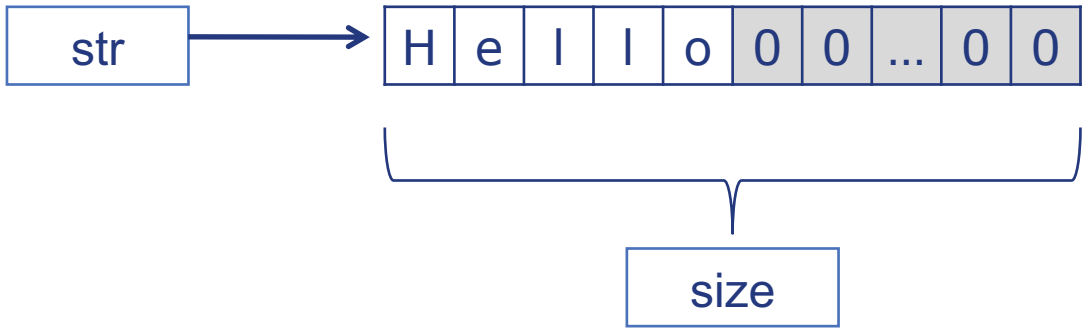
- 복사 연산 (C++03, 하위 호환성)
  - 사용자 정의 이동 연산이 있을 때 묵시적으로 생성되지 않음
  - 사용자 정의 복사 할당 연산자가 있을 때, 복사 생성자도 생성
  - 사용자 정의 복사 생성자가 있을 때, 복사 할당 연산자도 생성
  - 사용자 정의 소멸자가 있을 때, 묵시적으로 생성됨
- 이동 연산 (C++11)
  - 묵시적(자동)으로 생성되지 않을 때
    - 사용자 정의 복사 연산이 있을 때
    - 다른 사용자 정의 이동 연산이 있을 때
    - 사용자 정의 소멸자가 있을 때
    - 비정적 멤버가 참조(reference) 나 const일 경우
- 올바르게 동작하려면 기본 연산을 하나도 구현하지 않거나(0의 규칙) 모두 구현(6의 규칙)해야 한다!
- RAII를 준수하는 멤버들(파일 스트림, 스마트 포인트 등)만 사용하면 0의 규칙을 따를 수 있음

Exercise

- Implement class MyString equivalent to STL string.

```
class MyString {
    char* const str ;
    int size ;
public:
    ...
} ;
```

MyString str("Hello");



Exercise

- Implement class MyString equivalent to STL string.

```
class MyString {
    char* const str ;
    int size ;
public:
    ...
} ;
```

```
int main() {
    // constructor test
    MyString str1("ABC") ;
    str1.print() ;

    // copy constructor test
    {
        MyString str2(str1) ;
        // set test
        str2.set(0, 'D') ;
        str2.print() ;
    }
    str1.print() ;

    // length, at test
    for ( int i = 0 ;i < str1.length() ; i ++ )
        cout << str1.at(i) ;
}
```

```
ABC
DBC
ABC
ABC
```

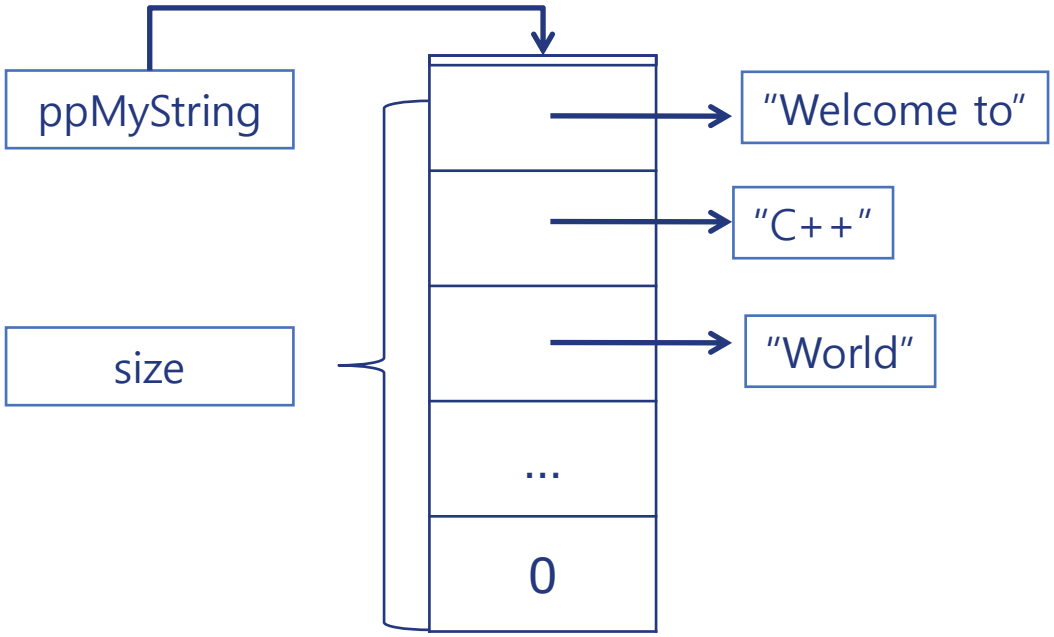
MyStringList

```
class MyStringList {
private:
   MyString** const ppMyString ;
    int size ;
public:
    …
} ;
```

```
int main() {
    MyStringList list1 ;

    list1.set(0, "Welcome to") ;
    list1.set(2, "World") ;
    list1.print() ;

    MyStringList list2(list1) ;
    list1.clear() ;
    list2.set(1, "C++") ;
    list2.print() ;
}
```



Q&A