항목 4-21: new를 직접 사용하는 것보다 std::make\_unique와 std::make\_shared를 선호하라  
  
std::make\_unique와 std::make\_shared는   
1.임의의 개수와 형식의 인수들을 받아서 그것들을 생성자로 완벽 전달해서 객체를 동적으로 생성  
2. 그 객체를 가리키는 스마트포인터들을 돌려주는 make함수임  
  
std::allocate\_shared도 std::make\_shared처럼 작동하지만, 첫 인수가 동적 메모리 할당에 쓰일 할당자 객체라는 점이 다름  
  
auto upw1(std::make\_unique<Widget>()); // make 함수 사용  
std::unique\_ptr<Widget> upw2(new Widget); //사용하지 않음  
  
auto spw1(std::make\_shared<Widget>()); // make 함수 사용  
std::shared\_ptr<Widget> spw2(new Widget); // 사용하지 않음  
  
장점1. 단순히 사용법만 보더라도 make함수들이 더 사용하기 편함  
중복된 객체 형식을 1개로 줄일 수 있고, 컴파일 시간 감소, 목적 코드의 크기 감소, 일관성 있는 코드 기반이라는 장점이 있음  
  
장점2. 예외 안정성 측면에서 더 안정적임  
  
void processWidget(std::shared\_ptr<Widget> spw, int priority);  
int computePriority();  
processWidget(std::shared\_ptr<Widget>(new Widget), computePriority());  
// 자원누수 위험이 있음  
  
make함수를 사용하지 않는 위 구문의 경우 new로 생성한 Widget에 대한 누수가 발생할 수 있음.  
  
컴파일러가 목적코드를 만들어내면서 아래와 같은 순서로 실행되도록 할 수 있음  
1. “new Widget” 실행  
2. computePriority 실행 -> 여기서 예외가 발생한다면?!  
3. std::shared\_ptr 생성자 실행  
  
위와 같은 경우 new로 생성한 Widget 객체가 샘.  
여기서 make함수로 대체하게되면 문제가 해결됨  
  
processWidget(std::make\_shared<Widget>(), computePriority());  
실행시점에 두 인자 중 어떤 것이 먼저 오더라도 OK  
  
  
<std::make\_shared를 사용할 때, 특별히 더 좋은 이유>  
또한 std::make\_shared를 사용하면 기존(new사용)보다 더 향상된 효율성을 얻음  
std::shared\_ptr<Widget> spw(new Widget);  
여기서 메모리 할당이 두번 일어나게 됨 (객체, 제어블록)  
->   
auto spw = std::make\_shared<Widget>();  
한번의 할당(객체와 제어블록이 같은 메모리 블록에 한번에 할당됨)  
  
결론적으로 실행 코드의 정적인 크기가 줄어들고, 속도가 빨라짐, 제어블록에 일정 정도의 내부 관리용 정보를 포함할 필요가 없어서 프로그램의 메모리 사용량이 줄어들 수 있음  
(std::make\_shared의 효율성 분석은 std::allocate\_shared도 동일하게 적용가능)  
  
  
하지만! make함수를 선호하라는 것이지 항상 쓰라는 것이 아님!  
  
1. make함수는 커스텀 삭제자를 지정할 수 없음  
  
auto widgetDeleter = [] (Widget\* pw) {…};

std::unique\_ptr<Widget, decltype(widgetDeleter)> upw(new Widget, widgetDeleter);  
std::shared\_ptr<Widget> spw(new Widget, widgetDeleter);  
  
2. std::initializer\_list를 받는 생성자인 경우, 중괄호 초기치로 생성하려면 반드시 new를 직접 사용하거나 ,“항목30”의 설명처럼 우회책을 사용해야함  
  
make함수들은 내부적으로 매개변수들을 완벽전달할 때 중괄호가 아니라 괄호를 사용함  
  
결국 아래와 같은 우회책 사용해야함  
auto initList = {10,20};

auto spv = std::make\_shared<std::vector<int>>(initList);  
  
이 위 2가지는 std::make\_unique, std::make\_shared 모두 문제가 됨  
이 아래 2가지는 std::make\_shared의 문제임  
  
3. 특정 operator new, operator delete가 구성되어있는 형식의 객체를 make함수로 생성하는 것은 대체로 바람직하지 않음  
어떤 형식에, 위 두 함수가 존재한다는 것은, 전역 메모리 할당/해제 루틴이 그 형식의 객체에 적합하지 않는다는 것.  
결국 커스텀 할당, 커스텀 해제가 필요해짐  
  
4. 객체 형식이 상당히 크고 마지막 std::shared\_ptr의 파괴와 std::weak\_ptr의 파괴 사이의 시간이 꽤 길다면, 객체가 파괴된 시점과 객체가 점유하던 메모리가 해제되는 시점에 시간지연이 생김

|  |  |
| --- | --- |
| std::make\_shared사용 | new사용 |
| class ReallyBigType{…};  auto pBigObj =  std::make\_shared<ReallyBigType>();  … // 큰객체를 가리키는 std::shared\_ptr들과 std::weak\_ptr들을 생성해서 사용  … // 여기서 객체를 가리키는 마지막 std::shared\_ptr이 파괴, 아직 std::weak\_ptr는 남아있음   … // 여기서 큰 객체가 차지하는 메모리는 여전히 할당된 상태   … // 여기서 객체를 가리키는 마지막 std::weak\_ptr이 파괴가 되면서 제어블록/객체가 차지하던 메모리가 해제 | class ReallyBigType{…};  std::shared\_ptr<ReallyBigType> pBigObj(new ReallyBigType);  … // 큰객체를 가리키는 std::shared\_ptr들과 std::weak\_ptr들을 생성해서 사용  … // 여기서 객체를 가리키는 마지막 std::shared\_ptr이 파괴, std::weak\_ptr은 남아있음, 하지만 객체의 메모리는 해제됨  … // std::weak\_ptr이 붙잡고 있는 제어블록을 위한 메모리만 할당된 상태  … // 마지막 std::weak\_ptr이 파괴되면서 제어블록 메모리 해제 |

이렇게 어쩔 수 없이 new를 사용하게 되는 상황에서는 예외 안전성에 신경을 써야함  
new의 결과를 “다른 일은 전혀 하지 않는 문장에서” 스마트 포인터의 생성자에 즉시 넘겨주는 것  
  
std::shared\_ptr<Widget> spw(new Widget, cusDel); //커스텀 삭제자 지정  
processWidget(spw, computePriority);  
위와 같이 스마트 포인터 생성과 이를 사용하는 구문을 분리하여 예외발생의 여지를 줄임  
  
또한 여기서 spw는 processWidget에 값 전달로 넘어가고, 왼값이므로 복사 생성이 일어나게 됨. 그리고 std::shared\_ptr의 복사생성은 참조 횟수를 원자적으로 증가해야 하지만, 이동생성은 참조 횟수를 조작할 필요가 없기 때문에, std::move를 적용해서 spw를 오른값으로 변환해야함  
  
processWidget(std::move(spw), computePriority());  
  
기억해 둘 사항들  
- new의 직접사용에 비해, make함수를 사용하면 소스코드 중복의 여지가 없어지고, 예외 안전성이 향상되고, std::make\_shared와 std::allocate\_shared의 경우 더 작고 빠른 코드가 산출된다.  
- make 함수의 사용이 불가능 또한 부적합한 경우로는 커스텀 삭제자를 지정해야 하는 경우와 중괄호 초기치를 전달해야 하는 경우가 있다.  
- std::shared\_ptr에 대해서는 make함수가 부적합한 경우가 더 있는데, (1)커스텀 메모리 관리 기능을 가진 클래스를 다루는 경우와 (2) 메모리가 넉넉하지 않은 시스템에서 큰 객체를 자주 다루어야 하고 std::weak\_ptr들이 해당 std::shared\_ptr들보다 더 오래 살아남는 경우이다.

항목 4-22: Pimpl 관용구를 사용할 때에는 특수 멤버 함수들을 구현파일에서 정의하라  
  
Pimpl관용구는 빌드시간을 줄이기 위해서 많이 사용됨  
다만 여기에 생 포인터보다 스마트포인터를 이용하여 구성할 때에 특수 멤버 함수들을 구현파일(CPP)에서 정의할 필요가 있음  
  
Widget클래스의 자료 멤버들이 std::string, std::vector, Gadget 형식으로 구성되어 있다면 이들의 형식들이 #include를 이용하여 헤더파일들이 포함되어야 사용할 수 있음  
  
만약 한 헤더의 내용이 변하면 Widget을 사용하는 클라이언트도 헤더의 내용에 의존해야하기 때문에 다시 빌드해야 함  
  
그래서 Pimpl관용구를 이용하여 선언부와 구현부를 분리하여 구현함  
  
기존의 C++98생 포인터 방식 Pimpl  
-widget.h 헤더파일 내부  
class Widget{

public:  
 Widget();  
 ~Widget() //소멸자가 필요함 – 본문

…

private:  
 struct Impl; //구현용 구조체  
 Impl \* pImpl; //구조체를 가리키는 포인터를 선언  
};  
  
-widget.cpp 구현부  
#include “widget.h”  
#include “gadget.h”

#include<string>  
#include<vector>

struct Widget::Impl { //전에 Widget에 있던 자료 멤버들

std::string name;

std::vector<double> data;

Gadget g1,g2,g3;

};

Widget::Widget() //이 Widget 객체를 위한 자료멤버들을 할당

: pImpl(new Impl)

{}

Widget::~Widget() //자료멤버들 해제

{ delete pImpl;}  
  
  
Impl 객체를 동적으로 해제하고 할당하는 코드가 있어서, Widget이 파괴될 때 이 객체도 해제되어야 하기 때문에, 반드시 소멸자가 필요  
  
이를 스마트 포인터를 사용하여 바꾸려고 했을 때, Widget 생성자/소멸자에서 Widget::Impl객체를 할당/해제한다면 (Widget만이 Impl객체를 소유함), std::unique\_ptr을 통해서 구현하는 것이 좋음  
  
-std::unique\_ptr을 사용한 “widget.h”내부  
class Widget {

public:

Widget();

…

private:

struct Impl;

std::unique\_ptr<Impl> pImpl; // 생 포인터 대신 스마트포인터 사용  
};

-std::make\_unique를 사용한 “widget.cpp”내부  
#include “widget.h”  
#include “gadget.h”

#include<string>  
#include<vector>

struct Widget::Impl {

std::string name;

std::vector<double> data;

Gadget g1,g2,g3;

};

Widget::Widget()

:pImpl(std::make\_unique<Impl>()) // make함수를 사용!

{}

이 코드 자체는 잘 컴파일되지만, 아래와 같이 클라이언트가 사용하려하면 컴파일되지 않음  
  
#include “widget.h”

Widget w;