**Chapter 1. C++에 왔으면 C++의 법을 따릅시다.**

**항목 2. #define을 쓰려거든 const, enum, inline을 먼저 떠올리자.**

\* 단순한 상수를 쓸 때는, #define보다 const 객체 혹은 enum을 우선 생각합시다

\* 함수처럼 쓰이는 매크로를 만들려면, #define 매크로 보다 인라인 함수를 우선 생각해 봅시다.

**항목 3. 낌새만 보이면 const를 들이대 보자**

\* const를 붙여 선언하면 컴파일러가 사용상의 에러를 잡아내는데 도움을 준다. const는 어떤 유효 범위에 있는 객체에도 붙을 수 있으며, 함수 매개변수 및 반환 타입에도 붙을 수 있으며, 멤버 함수에도 붙을 수 있습니다.

\* 컴파일러 쪽에서 보면 비트 수준 상수 성을 지켜야 하지만, 여러분은 개념적인(논리적인) 상수 성을 사용해서 프로그래밍 해야 합니다.

\* 상수 멤버 및 비상수 멤버 함수가 기능적으로 서로 똑같게 구현되어 있을 경우에는 코드 중복을 피하는 것이 좋은데, 이 때 비 상수 버전을 호출하도록 만든다.

**항목 4. 객체를 사용하기 전에 반드시 그 객체를 초기화 하자**

\* 멤버가 아닌 기본 제공 타입 객체는 여러분 손으로 직접 초기화한다. 경우에 따라 저절로 되고도 하고 안되기도 하기 때문

\* 생성자 에서는 데이터 멤버에 대한 대입문을 생성자 본문 내부에 넣는 방법으로 멤버를 초기화 하지 말고 멤버 초기화 리스트를 즐겨 사용하자. 그리고 초기화 리스트에 데이터 멤버를 나열할 때는 클래스에 각 데이터 멤버가 선언된 순서와 똑같이 나열한다.

\* 여러 번역 단위에 있는 비 지역 정적 객체들의 초기화 순서 문제는 피해서 설계해야 한다. 비 지역 정적 객체를 지역 정적 객체로 바꾸면 된다.

**Chapter 2. 생성자, 소멸자 및 대입 연산자**

**항목 5. C++가 은근 슬쩍 만들어 호출해 버리는 함수들에 촉각을 세우자.**

\* 컴파일러는 경우에 따라 클래스에 대해 기본 생성자, 복사 생성자, 복사 대입 연산자, 소멸자를 암시적으로 만들어 놓을 수 있다.

**항목 6. 컴파일러가 만들어낸 함수가 필요 없으면 확실히 이들의 사용을 금해버리자**

\* 컴파일러에서 자동으로 제공하는 기능을 허용치 않으려면, 대응되는 멤버 함수를 private로 선언한 후에 구현은 하지 않은 채로 두십시오. Uncopyable과 비슷한 기본 클래스를 쓰는 것도 한 방법입니다.

**항목 7. 다형성을 가진 기본 클래스에서는 소멸자를 반드시 가상 소멸자로 선언하자.**

\* 다형성을 가진 기본 클래스에는 반드시 가상 소멸자를 선언해야 합니다. 즉, 어떤 클래스가 가상 함수를 하나라도 갖고 있으면, 이 클래스의 소멸자도 가상 소멸자이어야 합니다.

\* 기본 클래스로 설계되지 않았거나 다형성을 갖도록 설계되지 않은 클래스에는 가상 소멸자를 선언하지 말아야 합니다.

**항목 8. 예외가 소멸자를 떠나지 못하도록 붙들어 놓자**

\* 소멸자에서 예외가 빠져나가면 안됩니다. 만약 소멸자 안에서 호출된 함수가 예외를 던질 가능성이 있다면, 어떤 예외이든지 소멸자에서 모두 받아낸 후에 삼켜 버리든지 프로그램을 끝내던지 해야 합니다.

\* 어떤 클래스의 연산이 진행되다가 던진 예외에 대해 사용자가 반응해야 할 필요가 있다면, 해당 연산을 제공하는 함수는 반드시 보통의 함수(소멸자가 아닌 함수)이어야 합니다.

**항목 9. 객체 생성 및 소멸 과정 중에는 절대로 가상 함수를 호출하지 말자**

**\*** 생성자 혹은 소멸자 안에서 가상 함수를 호출하지 마세요. 가상 함수라고 해도, 지금 실행중인 생성자나 소멸자에 해당되는 클래스의 파생 클래스 쪽으로는 내려가지 않습니다.

**항목 10. 대입 연산자는 \*this의 참조자를 반환하게 하자.**

**\*** 대입 연산자는 \*this의 참조자를 반환하도록 만드세요

**항목 11. operator=에서는 자기대입에 대한 처리가 빠지지 않도록 하자**

\* operator=을 구현할 때, 어떤 객체가 그 자신에 대입되는 경우를 제대로 처리하도록 만듭시다.

원본 객체와 복사 대상 객체의 주소를 비교 해도 되고, 문자의 순서를 적절히 조정할 수도 있으며, 복사 후 맞바꾸기 기법을 써도 된다.

\* 두 개 이상의 객체에 대해 동작하는 함수가 있다면, 이 함수에 넘겨지는 객체들이 사실 같은 객체인 경우에 정확하게 동작하는지 확인해 보세요

**항목 12. 객체의 모든 부분을 빠짐없이 복사하자.**

**\*** 객체 복사 함수는 주어진 객체의 모든 데이터 멤버 및 모든 기본 클래스 부분을 빠뜨리지 말고 복사하자.

\* 클래스의 복사 함수 두 개를 구현할 때, 한쪽을 이용해서 다른 쪽을 구현하려는 시도는 절대로 하지 마세요. 그대신, 공통된 동작을 제 3의 함수에다 분리해 놓고 양쪽에서 이것을 호출하게 만들어서 해결합시다.

**Chapter 3. 자원 관리**

**항목 13. 자원 관리에는 객체가 그만!**

\* 자원 누출을 막기 위해, 생성자 안에서 자원을 획득하고 소멸자에서 그것을 해제하는 RAII객체를 사용합니다.

\* 일반적으로 널리 쓰이는 RAII클래스는 tr1::shared\_ptr 그리고 auto\_ptr이다. 이둘 가운데 tr1::shared\_ptr이 복사 시의 동작이 직관적이기 때문에 대개 더 좋습니다. 반면, auto\_ptr은 복사되는 객체를 null로 만들어 버립니다.

**항목 14. 자원 관리 클래스의 복사 동작에 대해 진지하게 고찰하자.**

\* RAII 객체의 복사는 그 객체가 관리하는 자원의 복사 문제를 안고 가기 때문에, 그 자원을 어떻게 복사하느냐에 따라 RAII객체의 복사 동작이 결정된다.

\* RAII 클래스에 구현하는 일반적인 복사 동작은 복사를 금지하거나 참조 카운팅을 해주는 선으로 마무리하는 것입니다. 하지만 이 외의 방법들도 가능하니 참고해 둡시다.

**항목 15. 자원 관리 클래스에서 관리되는 자원은 외부에서 접근할 수 있도록 하자.**

**\*** 실제 자원을 직접 접근해야 하는 기존 API들도 많기 때문에, RAII 클래스를 만들 때는, 그 클래스가 관리하는 자원을 얻을 수 있는 방법을 열어 주어야 합니다.

**\***  자원 접근은 명시적 변환 혹은 암시적 변환을 통해 가능합니다. 안전성만 따지면 명시적 변환이 대체적으로 더 낫지만, 고객 편의성을 놓고 보면 암시적 변환이 괜찮습니다.

**항목 16. new 및 delete를 사용할 때는 형태를 반드시 맞추자.**

\*new 표현식에 []를 썻으면, 대응되는 delete 표현식에도 []를 써야합니다. 마찬가지로 new 표현식에 []를 안썻으면, 대응되는 delete 표현식에도 []를 쓰지 말아야 합니다.

**항목 17. new로 생성한 객체를 스마트 포인터에 저장하는 코드는 별도의 한 문장으로 만들자.**

\* new로 생성한 객체를 스마트 포인터로 넣는 코드는 별도의 한 문장으로 만듭시다. 이것이 안되어 있으면, 예외가 발생될 때 디버깅하기 힘든 자원 누출이 초래될 수 있습니다.

**Chapter 4. 설계 및 선언**

**항목 18. 인터페이스 설계는 제대로 쓰기엔 쉽게, 엉터리로 쓰기엔 어렵게 하자.**

\* 좋은 인터페이스는 제대로 쓰기에 쉬우며 엉터리로 쓰기에 어렵습니다. 인터페이스를 만만들 때 이 특성을 지닐 수 있도록 고민하고 또 고민합시다.

\* 인터페이스의 올바른 사용을 이끄는 방법으로는 인터페이스 사이의 일관성 잡아주기, 그리고 기본제공 타입과의 동작 호환성 유지하기가 있습니다.

\* 사용자의 실수를 방지하는 방법으로는 새로운 타입 만들기, 타입에 대한 연산을 제한하기, 객체의 값에 대해 제약 걸기, 자원 관리 작업을 사용자 책임으로 놓지 않기가 있습니다.

\* tr1::shared\_ptr은 사용자 정의 삭제자를 지원합니다. 이 특징 때문에 tr1::shared\_ptr은 교차 DLL 문제를 막아주며, 뮤텍스 등을 자동으로 잠금 해제하는 데 쓸 수 있습니다.

**항목 19. 클래스 설계는 타입 설계와 똑같이 취급하자.**

\* 클래스 설계는 타입 설계 입니다. 새로운 타입을 정의하기 전에, 이번 항목에 나온 모든 고려사항을 빠짐없이 점검해 보십시오.

**항목 20. ‘값에 의한 전달’ 보다는 ‘상수 객체 참조자에 의한 전달 방식’을 택하는 편이 대게 낫다.**

\* ‘값에 의한 전달’보다는 ‘상수 객체 잠조자에 의한 전달’을 선호합시다. 대체적으로 효율적일 뿐만 아니라 복사손실 문제까지 막아줍니다.

\* 이번 항목에서 다룬 법칙은 기본 제공 타입 및 STL 반복자, 그리고 함수 객체 타입에는 맞지 않습니다. 이들에 대해서는 ‘값에 의한 전달’이 더 적절합니다.

**항목 21. 함수에서 객체를 반환해야 할 경우에 참조자를 반환하려고 들지 말자.**

\* 지역 스택 객체에 대한 포인터나 참조자를 반환하는 일, 혹은 힙에 할당된 객체에 대한 참조자를 반환하는 일, 또는 지역 정적 객체에 대한 포인터나 참조자를 반환하는 일은 그런 객체가 두 개 이상 필요해질 가능성이 있다면 절대로 하지마세요 (항목 4를 보시면 지역 정적 객체에 대한 참조자를 반환하도록 설계된 올바른 코드 예제를 찾을 수 있습니다. 최소한 단일 스레드 환경에서는 통합니다.)

**항목 22. 데이터 멤버가 선언될 곳은 private 영역임을 명심하자.**

\*데이터 멤버는 private 멤버로 선언 합시다. 이를 통해 클래스 제작자는 문법적으로 일관성 있는 데이터 접근 통로를 제공할 수 있고, 필요에 따라서는 세밀한 접근 제어도 가능하며, 클래스의 불변속성을 강화할 수 있을 뿐 아니라, 내부 구현의 융통성도 발휘할 수 있습니다.

\* protected는 public보다 더 많이 ‘보호’받고 있는 것이 절대로 아닙니다.

**항목 23. 멤버 함수보다는 비멤버 프렌드 함수와 더 가까워지자.**

\* 멤버 함수보다는 비멤버 프렌드 함수를 자주 쓰도록 합시다. 캡슐화 정도가 높아지고, 패키징 유연성도 커지며, 기능적인 확장성도 늘어납니다.

**항목 24. 타입 변환이 모든 매개변수에 대해 적용되어야 한다면 비 멤버 함수를 선언하자.**

\* 어떤 함수에 들어가는 모든 매개변수(this 포인터가 가리키는 객체도 포함해서)에 대해 타입 변환을 해 줄 필요가 있다면, 그 함수는 비 멤버이어야 합니다.

**항목 25. 예외를 던지지 않는 swap에 대한 지원도 생각해 보자.**

\* std::swap이 여러분의 타입에 대해 느리게 동작할 여지가 있다면 swap 멤버 함수를 제공합시다. 이 멤버 swap은 예외를 던지지 않도록 만듭시다.

\* 멤버 swap을 제공했으면, 이 멤버를 호출하는 비멤버 swap도 제공합니다. 클래스(템플릿이 아닌)에 대해서는, std::swap도 특수화해 둡시다.

\* 사용자 입장에서 swap을 호출할때는, std::swap에 대한 using 선언을 넣어 준 후에 네임스페이스 한정 없이 swap을 호출합시다.

\* 사용자 정의 타입에 대한 std 템플릿을 완전 특수화하는 것은 가능합니다. 그러나 std에 어떤 것이라도 새로 ‘추가’하려고 들지는 마십시오.

**Chapter 5. 구현**

**항목 26. 변수 정의는 늦출 수 있는 데까지 늦추는 근성을 발휘하자.**

\* 변수 정의는 늦출 수 있을 때까지 늦춥시다. 프로그램이 더 깔끔해지며 효율도 좋아집니다.

**항목 27. 캐스팅은 절약, 또 절약! 잊지 말자.**

\* 다른 방법이 가능하다면 캐스팅은 피하십시오. 특히 수행 성능에 민감한 코드에서 dynamic\_cast는 몇 번이고 다시 생각하십시오. 설계 중에 캐스팅이 필요해졌다면, 캐스팅을 쓰지 않는 다른 방법을 시도해 보십시오.

\* 캐스팅이 어쩔 수 없이 필요하다면, 함수 안에 숨길 수 있도록 해 보십시오. 이렇게 하면 최소한 사용자는 자신의 코드에 캐스팅을 넣지 않고 이 함수를 호출할 수 있게 됩니다.

\* 구형 스타일의 캐스트를 쓰려거든 C++ 스타일의 캐스트를 선호하십시오. 발견하기도 쉽고, 설계자가 어떤 역할을 의도 했는지가 더 자세하게 드러납니다.

**항목 28. 내부에서 사용하는 객체에 대한 ‘핸들’을 반환하는 코드는 되도록 피하자**

\* 어떤 객체의 내부 요소에 대한 핸들(참조자, 포인터, 반복자)을 반환하는 것은 되도록 피하세요. 캡슐화 정도를 높이고, 상수 멤버함수가 객체의 상수성을 유지한 채로 동작할 수 있도록 하며, 무효참조 핸들이 생기는 경우를 최소화할 수 있습니다.

**항목 29. 예외 안정성이 확보되는 그날 위해 싸우고 또 싸우자.**

\* 예외 안정성을 갖춘 함수는 실행 중 예외가 발생하더라도 자원을 누출시키지 않으며 자료구조를 더럽힌채 내버려두지 않는다. 이런 함수들이 제공할 수 있는 예외 안정성 보장은 기본적 / 강력한 / 예외 금지 보장이 있다.

\* 강력한 예외 안정성 보장은 복사 후 맞바꾸기 방법을 써서 구현 할 수 있지만, 반드시 실용적인 것은 아니다.

\* 어떤 함수가 제공하는 예외 안정성 보장의 강도는 그 함수가 내부적으로 호출 하는 함수들이 제공하는 가장 약한 보장을 넘을 수 없다.

**항목 30. 인라인 함수는 미주알고주알 따져서 이해해 두자.**

\* 함수 인라인은 작고, 자주 호출되는 함수에 대해서만 하는 것으로 묶어 둡시다. 이렇게 하면 디버깅 및 라이브러리의 바이너리 업그레이드가 용이해지고, 자칫 생길 수 있는 코드 부풀림 현상이 최소화되며, 프로그램의 속력이 더 빨라질 수 있는 여지가 최고로 많아집니다.

\* 함수 템플릿이 대개 헤더 파일에 들어간다는 일반적인 부분만 생각해서 이들을 inline으로 선언하면 안됩니다

**항목 31. 파일 사이의 컴파일 의존성을 최대로 줄이자.**

\* 컴파일 의존성을 최소화 하는 작업의 배경이 되는 가장 기본적인 아이디어는 ‘정의’대신에 ‘선언’에 의존하게 만들자는 것입니다. 이 아이디어에 기반한 두 가지 접근 방법은 핸들 클래스와 인터페이스 클래스입니다.

\* 라이브러리 헤더는 그 자체로 모든 것을 갖추어야 하며 선언부만 갖고 있는 형태여야 합니다. 이 규칙은 템플릿이 쓰이거나 쓰이지 않거나 동일하게 적용합시다.

**Chapter 6. 상속, 그리고 객체 지향 설계**

**항목 32. Public 상속 모형은 반드시 “is-a(..는 ..의 일종이다)” 를 따르도록 만들자.**

\* public 상속의 의미는 “is-a(..는 ..의 일종)”입니다. 기본 클래스에 적용되는 모든 것들이 파생 클래스에 그대로 적용되어야 합니다. 왜냐하면 모든 파생 클래스 객체는 기본 클래스 객체의 일종이기 때문입니다.

**항목 33. 상속된 이름을 숨기는 일은 피하자.**

\* 파생 클래스의 이름은 기본 클래스의 이름을 가립니다. Public 상속에서는 이런 이름 가림 현상은 바람직하지 않습니다.

\* 가려진 이름을 다시 볼 수 있게 하는 방법으로, using 선언 혹은 전달 함수를 쓸 수 있습니다.

**항목 34. 인터페이스 상속과 구현 상속의 차이를 제대로 파악하고 구별하자.**

\* 인터페이스 상속은 구현 상속과 다릅니다. Public 상속에서 파생 클래스는 항상 기본 클래스의 인터페이스를 모두 물려 받습니다.

\* 순수 가상 함수는 인터페이스 상속만을 허용합니다.

\* 단순(비순수) 가상 함수는 인터페이스 상속과 더불어 기본 구현의 상속도 가능하도록 지정합니다.

\* 비가상 함수는 인터페이스 상속과 더불어 필수 구현의 상속도 가하도록 기정합니다.

**항목 35. 가상 함수 대신 쓸 것들도 생각해 두는 자세를 시시 때때로 길러 두자.**

\* 가상 함수 대신에 쓸 수 있는 다른 방법으로 NVI 관용구 및 전략 패턴을 들 수 있습니다.

\* 객체에 필요한 기능을 멤버 함수로부터 클래스 외부의 비멤버 함수로 옮기면, 그 비멤버 함수는 그 클래스의 public 멤버가 아닌 것들을 접근할 수 없다는 단점이 생깁니다.

\* tr1::function 객체는 일반화된 함수 포인터처럼 동작합니다. 이 객체는 주어진 목적 시그니처와 호환되는 모든 함수호출성 개체를 지원합니다.

**항목 36. 상속받은 비가상 함수를 파생 클래스에서 재정의 하는 것은 절대 금물!**

\* 상속받은 비가상 함수를 재정의하는 일은 절대로 하지 맙시다.

**항목 37. 어떤 함수에 대해서도 상속받은 기본 매개변수 값은 절대로 재정의하지 말자**

\* 상속받은 기본 매개변수 값은 정대로 재정의해서는 안됩니다. 왜냐하면 기본 매개변수 값은 정적으로 바인딩되는 반면, 가상 함수(여러분이 오버라이드할 수 있는 유일한 함수이죠)는 동적으로 바인딩되기 때문입니다

**항목 38. “has-a(..는..를 가짐)” 혹은 “is-implemented-in-terms-of(..는..를 써서 구현됨)” 를 모형화할 때는 객체 합성을 사용하자.**

\* 객체 합성(composition)의 의미는 public 상속이 가진 의미와 완전히 다릅니다.

\* 응용 영역에서 객체 합성의 의미는 has-a(~는 ~를 가짐)입니다. 구현 영역에서는 is-implemented-in-terms-of(~는 ~를 써서 구현됨)의 의미를 갖습니다.

**항목 39. Private 상속은 심사숙고해서 구사하자**

\* private 상속의 의미는 is-implemented-in-terms-of (~는 ~를 써서 구현됨)이다. 파생 클래스측에서 기본 클래스의 protected 멤버에 접근해야 할 경우 혹은 상속받은 가상함수를 재정의 해야할때 private 상속은 의미를 가질 수 있다.

\* 객체 합성과 달리 private 상속은 공백 기본 클래스 최적화를 활성화 시킬 수 있다.

**항목 40. 다중 상속은 심사숙고해서 사용하자.**

\* 다중 상속은 단일 상속보다 확실히 복잡합니다. 새로운 모호성 문제를 일으킬 뿐만 아니라 가상 상속이 필요해질 수도 있습니다.

\* 가상 상속을 쓰면 크기 비용, 속도 비용이 늘어나며, 초기화 및 대입 연산의 복잡도가 커집니다. 따라서 가상 기본 클래스에는 데이터를 두지 않는 것이 현실적으로 가장 실용적입니다.

\* 가상 상속을 적법하게 쓸 수 있는 경우가 있습니다. 여러 시나리오 중 하나는 인터페이스 클래스로부터 public 상속을 시킴과 동시에 구현을 돕는 클래스로부터 private 상속을 시키는 것입니다.

**Chapter 7. 템플릿과 일반화 프로그래밍**

**항목 41. 템플릿 프로그래밍의 천릿길도 암시적 인터페이스와 컴파일 타임 다형성부터**

\* 클래스 및 템플릿은 모두 인터페이스와 다형성을 지원합니다.

\* 클래스의 경우, 인터페이스는 명시적이며 함수의 시그너처를 중심으로 구성되어 있습니다. 다형성은 프로그램 실행 중에 가상 함수를 통해 나타납니다.

\* 템플릿 매개 변수의 경우, 인터페이스는 암시적이며 유효 표현식에 기반을 두어 구성됩니다. 다형성은 컴파일 중에 템플릿 인스턴스와 함수 오버로딩 모호성 해결을 통해 나타납니다.

**항목 42. Typename의 두 가지 의미를 제대로 파악하자.**

\* 템플릿 매개 변수를 선언할 때, class 및 typename은 서로 바꾸어 써도 무방합니다.

\* 중첩 의존 타입 이름을 식별하는 용도에는 반드시 typename을 사용합니다. 단,중첩 의존 이름이 기본 클래스 리스트에 있거나 멤버 초기화 리스트 내의 기본 클래스 식별자로 있는 경우에는 예외이다.

**항목 43. 템플릿으로 만들어진 기본 클래스 안의 이름에 접근하는 방법을 알아 두자.**

**\*** 파생클래스 템플릿에서 기본 클래스 이름을 참조할 떄는 “this->”를 접투사로 붙이거나 기본 클래스 한정문을 명시적으로 써주는 것으로 해결합시다.

**항목 44. 매개변수에 독립적인 코드는 템플릿으로부터 분리시키자.**

\* 템플릿을 사용하면 비슷비슷한 클래스와 함수가 여러 벌 만들어집니다. 따라서 템플릿 매개변수에 종속되지 않은 템플릿 코드는 비대화의 원인이 됩니다.

\* 비 타입 템플릿 매개변수로 생기는 코드 비대화의 경우, 템플릿 매개변수를 함수 매개변수 혹은 클래스 데이터 멤버로 대체함으로써 비대화를 종종 없앨 수 있습니다.

\* 타입 매개 변수로 생기는 코드 비대화의 경우, 동일한 이진 표현구조를 가지고 인스턴스 화되는 타입들이 한 가지 함수 구현을 공유하게 만듦으로써 비대화를 감소시킬 수 있습니다.

**항목 45. “호환되는 모든 타입”을 받아들이는 데는 멤버 함수 템플릿이 직방!**

\* 호환되는 모든 타입을 받아들이는 멤버 함수를 만들려면 멤버 함수 템플릿을 사용합시다.

\* 일반화된 복사 생성 연산과 일반화된 대입 연산을 위해 멤버 템플릿을 선언했다 하더라도, 보통의 복사 생성자와 복사 대입 연산자는 여전히 직접 선언해야 합니다.

**항목 46. 타입 변환이 바람직할 경우에는 비 멤버 함수를 클래스 템플릿 안에 정의해 두자.**

\* 모든 매개 변수에 대해서 암시적 타입 변환을 지원하는 템플릿과 관계가 있는 함수를 제공하는 클래스 템플릿을 만들려고 한다면, 이런 함수는 클래스 템플릿 안에 프렌드 함수로서 정의합니다.

**항목 47. 타입에 대한 정보가 필요하다면 특성 정보 클래스를 사용하자.**

\* 특성정보 클래스는 컴파일 도중에 사용할 수 있는 타입 관련 정보를 만들어냅니다. 또한 특성정보 클래스는 템플릿 및 템플릿 특수 버전을 사용하여 구현합니다.

\* 함수 오버로딩 기법과 결합하여 특성정보 클래스를 사용하면, 컴파일 타임에 결정되는 타입별 if . . . else 점검문을 구사할 수 있습니다.

**항목 48. 템플릿 프로그래밍, 하지 않겠는가?**

\* 템플릿 메타 프로그래밍은 기존 작업을 런타임에서 컴파일타임으로 전환하는 효과를 가진다. 따라서 TMP를 사용하면 선행 에러 탐지와 높은 런타임 효율을 손에 넣을 수 있다.

\* TMP는 정책 선택의 조합에 기반하여 사용자 정의 코드를 생성하는데 쓸 수 있으며, 또한 특정 타입에 대해 부적절한 코드가 만들어지는 것을 막는 데도 쓸 수 있다.

**Chapter 8. new와 delete를 내 맘대로**

**항목 49. new 처리자의 동작 원리를 제대로 이해하자.**

\* set\_new\_handler 함수를 사용하면 메모리 할당 요청이 만족되지 못 했을 때 호출하는 함수를 지정할 수 있다.

\* 예외불가(nothrow) new는 영향력이 제한되어있다. 메모리 할당 자체에만 적용되기 때문이다. 이후에 호출되는 생성자에서는 얼마든지 예외를 던질 수 있다.

**항목 50. new 및 delete를 언제 바꿔야 좋은 소리를 들을지를 파악해 두자.**

\* 개발자가 스스로 사용자 정의 new 및 delete를 작성하는 데는 여러 가지 나름대로 타당한 이유가 있습니다. 여기에는 수행 성능을 향상시키려는 목적, 힙 사용 시의 에러를 디버깅하려는 목적, 힘 사용 정보를 수집하려는 목적 등이 포함됩니다.

**항목 51. new 및 delete 를 작성할 때 따라야 할 기존의 관례를 잘 알아 두자.**

\* 관례적으로, operator new 함수는 메모리 할당을 반복해서 시도하는 무한 루프를 가져야 하고, 메모리 할당 요구를 만족시킬 수 없을 때 new-handler를 호출해야 하며, 0바이트에 대한 대책도 있어야 합니다. 클래스 전용 버전은 자신이 할당하기로 예정된 크기보다 더 큰(틀린) 메모리 블록에 대한 요구도 처리해야 합니다.

\* operator delete 함수는 널 포인터가 들어 왔을 때 아무 일도 하지 않아야 합니다. 클래스 전용 버전의 경우에는 예정 크기보다 더 큰 블록을 처리해야 합니다.

**항목 52. 위치지정 new 를 작성한다면 위치지정 delete도 같이 준비하자**

\* operator new 함수의 위치지정 버전을 만들 때는 짝이 맞는 위치지정 delete 함수도 만들어 주어야 한다.

\* new 및 delete의 위치지정 버전을 선언 할 때는 의도한 바가 아닌데 이들의 표준 버전이 가려지지 않도록 주의해야 한다.

**항목 53. 컴파일러의 경고를 지나치지 말자**

\* 컴파일러의 경고를 쉽게 지나치지 말자. 여러분의 컴파일러에서 지원하는 최고 경고 수준에서도 경고 메시지를 내지 않고 컴파일 되는 코드를 만들자

\* 컴파일러 경고에 너무 기대지는 말자. 컴파일러가 경고를 내는 부분은 천 차 만별이기 때문이다. 이 컴파일러에서는 경고였지만 다른 컴파일러에서는 경고를 하지 않는 경우도 있다.

**항목 54. TR1을 포함한 표준 라이브러리 구성요소와 편안한 친구가 되자.**

\* 최초에 상정된 표준 C++ 라이브러리의 주요 구성요소는 STL, iostream, 로케일 등입니다. 여기에는 C89의 표준 라이브러리도 포함되어 있습니다.

\* TR1이 도입되면서 추가된 것은 스마트 포인터(tr1::shared\_ptr), 일반화 함수 포인터(tr1::function), 해시 기반 컨테이너, 정규 표현식 그리고 그 외의 1개 구성요소입니다.

\* TR1 자체는 단순히 명세서일 뿐입니다. TR1의 기능을 사용하기 위해서는 명세를 구현한 코드를 구해야 합니다 TR1 구현을 구할 수 잇는 자료처 중 한 군덱 바로 부스트입니다

**항목55. 부스트를 늘 여러분 가까이에**

\* 부스트는 동료 심사를 거쳐 등록되고 무료로 배포되는 오픈 소스 C++ 라이브러리를 개발하는 모임이자 웹사이트입니다. 또한 C++ 표준화에 있어서 영향력을 있는 역할을 맡고 있습니다.

\* 부스트에서 배포되는 라이브러리들 중엔 TR1 구성요소에 들어간 것도 잇지만, 그 외에 다른 라이브러리들도 아주 많습니다.