# 최소 C 언어 명세 (Min-C)

## Part I: 언어 축소의 기초 원리

이 보고서는 표준 C 언어의 모든 표현력을 유지하면서 불필요한 기능을 제거하여 최소한의 기능 집합만을 갖는 프로그래밍 언어, 'Min-C'의 명세를 정의하는 것을 목표로 한다. 이 과정은 단순히 언어의 구문 요소를 줄이는 것을 넘어, 프로그래밍 언어 설계의 근본적인 원칙과 계산 이론에 기반한 정밀한 축소 작업을 의미한다. 본 파트에서는 Min-C를 설계하는 데 사용된 이론적 프레임워크와 핵심 원칙을 제시한다.

### 제1장: 표현적 동등성과 튜링 완전성

Min-C 설계의 최우선 목표는 표준 C 언어와의 계산적 동등성(computationally equivalent)을 보장하는 것이다. 이는 표준 C로 구현할 수 있는 모든 알고리즘은 Min-C로도 동일하게 구현할 수 있어야 함을 의미한다. 이러한 계산 능력의 기준선은 튜링 완전성(Turing completeness) 개념으로 정의된다. 어떤 프로그래밍 언어가 튜링 기계가 수행할 수 있는 모든 계산을 시뮬레이션할 수 있다면, 그 언어는 튜링 완전하다고 말한다. 표준 C는 튜링 완전하며, 따라서 Min-C 역시 튜링 완전성을 유지해야 한다. 핵심적인 전제는, 언어의 특정 기능(feature)을 줄이는 것이 반드시 계산 능력의 감소로 이어지지는 않는다는 점이다. 필수적인 핵심 원시 기능(primitive) 집합이 유지된다면, 언어는 최소한의 구문으로도 완전한 표현력을 가질 수 있다.

이러한 제어 흐름의 단순화 가능성을 이론적으로 뒷받침하는 것이 바로 봄-야코피니 정리(Böhm-Jacopini theorem)이다.1 이 정리는 어떠한 복잡한 제어 흐름을 가진 프로그램이라도 단 세 가지 기본 제어 구조, 즉 순차(sequence), 선택(selection), 반복(iteration)만을 사용하여 재작성할 수 있음을 증명한다.

* **순차(Sequence):** 문장들이 순서대로 하나씩 실행되는 구조.
* **선택(Selection):** 조건에 따라 두 개 이상의 경로 중 하나를 선택하여 실행하는 구조 (예: if-else).
* **반복(Iteration):** 특정 조건이 만족되는 동안 코드 블록을 반복적으로 실행하는 구조 (예: while 루프).

이 정리는 for 루프, do-while 루프, switch 문과 같은 다양한 제어 구조들이 사실상 if와 while이라는 더 근본적인 구조로 대체될 수 있음을 시사한다. 이는 언어 명세에서 불필요한 제어 구조를 제거하고 핵심적인 요소만 남길 수 있다는 강력한 이론적 근거를 제공한다.

그러나 봄-야코피니 정리가 if와 while의 충분성을 보장하지만, 더 근본적이고 강력한 원시 기능의 조합이 존재한다. 모든 구조화된 루프는 컴파일 과정에서 궁극적으로 기계어 수준의 조건부 점프(conditional jump)와 무조건부 점프(unconditional jump) 명령어로 변환된다.3 C 언어에서 이러한 저수준 점프 기능을 직접적으로 반영하는 고수준 구문이 바로

goto 문이다. while 루프는 if와 goto의 조합으로 완벽하게 구현될 수 있다. 예를 들어, while (condition) 구문은 "만약 condition이 거짓이면 루프를 탈출하고, 참이면 루프 본문을 실행한 뒤 다시 condition 검사 지점으로 돌아가라"는 논리로 구성되며, 이는 if와 goto를 통해 직접적으로 표현할 수 있다.

따라서 진정한 최소주의적 언어 명세를 추구한다면, if와 while 대신 if(조건부 점프)와 goto(무조건부 점프)를 핵심 제어 흐름 원시 기능으로 채택하는 것이 더 우월한 선택이다. 이 조합은 봄-야코피니 정리를 만족시키는 while 루프를 구성할 수 있을 뿐만 아니라, 다중 루프 탈출(multi-level loop break)과 같이 순수 구조적 프로그래밍만으로는 어색하게 구현되는 복잡한 제어 흐름을 훨씬 더 직접적이고 효율적으로 구현할 수 있게 해준다.4 결론적으로, Min-C는 제어 흐름의 최소 원시 기능으로

if와 goto를 채택하여, 최소한의 구문으로 최대의 표현력을 확보한다.

### 제2장: 문법적 설탕 제거를 위한 프레임워크

문법적 설탕(syntactic sugar)은 프로그래밍 언어에서 이미 존재하는 기능들을 더 간결하거나 읽기 쉽게 표현할 수 있도록 제공되는 구문이다. 이러한 구문은 언어의 근본적인 표현력을 확장하지는 않지만, 프로그래머의 편의성을 높이는 역할을 한다. Min-C의 설계 원칙은 이러한 문법적 설탕을 체계적으로 제거하여 언어의 핵심만을 남기는 것이다.

제거의 기준은 명확하다. 만약 언어의 특정 기능 F가 다른 더 근본적인 기능들의 조합인 {f1, f2,...}으로 체계적이고 알고리즘적으로 완벽하게 대체될 수 있다면, F는 문법적 설탕으로 간주되어 제거 대상이 된다. 이 원칙에 따라 제거될 대표적인 C 언어의 문법적 설탕은 다음과 같다.

* **복합 할당 연산자(Compound Assignment Operators):** +=, -=, \*= 등과 같은 연산자들은 문법적 설탕의 전형적인 예이다. 이들은 이항 연산과 할당 연산을 결합한 것으로, x += y는 x = x + y와 완벽하게 동일한 의미를 갖는다.7 따라서 Min-C에서는 기본 할당 연산자  
  =와 기본 이항 연산자들만 유지하고, 모든 복합 할당 연산자는 제거한다.
* **삼항 조건 연산자(Ternary Conditional Operator):** condition? value1 : value2 형태의 삼항 연산자는 값을 반환하는 if-else 문의 간결한 표현이다.11 이 연산자는 항상 표준적인  
  if-else 블록으로 재작성될 수 있으므로, Min-C에서는 if-else 구문만을 남기고 삼항 연산자는 제거한다.
* **콤마 연산자(Comma Operator):** 콤마 연산자는 하나의 표현식 내에서 여러 연산을 순차적으로 실행하는 데 사용되지만, 대부분의 경우 세미콜론으로 문장을 분리하는 것으로 대체할 수 있다.14  
  for 루프의 초기화나 증감식에서 관용적으로 사용되지만, for 루프 자체가 제거되므로 그 필요성은 더욱 감소한다.

문법적 설탕을 제거하는 것은 단순히 키워드의 수를 줄이는 것 이상의 깊은 의미를 지닌다. 이는 특정 연산을 표현하는 유일하고 표준적인(canonical) 방법을 강제하는 설계 철학을 반영한다. 표준 C에서는 동일한 논리를 여러 방식으로 표현할 수 있다 (예: i++, i = i + 1, i += 1). 컴파일러는 이러한 다양한 표현들이 의미적으로 동일하다는 것을 인식해야 하므로, 파싱 및 의미 분석 단계에서 복잡성이 증가한다.

Min-C에서는 각 연산에 대해 단 하나의 표현 방식만을 허용함으로써 (예: 오직 x = x + 1만 유효) 언어의 문법(grammar) 자체가 단순해진다. 이는 Min-C를 위한 컴파일러나 정적 분석 도구를 작성하는 작업을 훨씬 용이하게 만든다. 또한, 프로그래머에게 더 명시적인 코드를 작성하도록 강제함으로써, 콤마 연산자의 우선순위와 관련된 혼동과 같은 특정 유형의 버그를 줄일 수 있다.14 결국 문법적 설탕의 제거는 프로그래머의 편의성이나 코드의 간결함보다 언어의 단순성, 형식적 분석 가능성, 그리고 명시성을 우선시하는 설계적 선택이다. 이 원칙은 Min-C 명세 전반에 걸쳐 일관되게 적용된다.

## Part II: Min-C 언어 명세

본 파트는 Min-C 언어의 형식적인 정의를 담고 있다. 어휘 요소, 데이터 타입, 연산자, 제어 흐름, 함수, 집합 데이터 타입 및 메모리 모델을 포함하여 언어의 모든 측면을 상세히 기술한다.

### 제3장: 어휘 요소와 기본 데이터 타입

Min-C의 가장 기본적인 구성 요소인 어휘적 구조와 데이터 타입을 정의한다. 이는 컴파일러가 소스 코드를 해석하는 첫 단계에 해당한다.

#### 어휘 요소

* **문자 집합(Character Set):** Min-C 소스 코드는 표준 ASCII 문자 집합을 사용하여 작성된다.
* **키워드(Keywords):** Min-C 언어에서 특별한 의미를 갖는 예약어는 다음과 같이 최소한의 집합으로 제한된다.
  + if, else, goto, return
  + struct
  + sizeof
  + char, short, int, long, unsigned
  + void
* **제외된 키워드:** 표준 C의 나머지 키워드들(for, while, do, switch, case, default, union, enum, const, typedef, static, extern, volatile 등)은 Min-C 언어의 일부가 아니다.
* **식별자(Identifiers):** 변수, 함수, 사용자 정의 타입의 이름을 지정하는 식별자는 영문자(a-z, A-Z) 또는 밑줄(\_)로 시작해야 하며, 이후에는 영문자, 숫자(0-9), 밑줄을 포함할 수 있다. 키워드는 식별자로 사용할 수 없다.
* **리터럴(Literals):** 소스 코드에 직접 표현되는 고정된 값이다.
  + **정수 리터럴:** 10진수, 8진수(접두사 0), 16진수(접두사 0x 또는 0X)로 표현된다. 접미사 u 또는 U는 unsigned 타입을, l 또는 L은 long 타입을, ll 또는 LL은 long long 타입을 명시하는 데 사용된다.
  + **문자 리터럴:** 작은따옴표(' ')로 묶인 단일 문자(예: 'a')로 표현된다.
  + **문자열 리터럴:** 큰따옴표(" ")로 묶인 문자들의 연속(예: "hello")으로 표현되며, 메모리 상에서는 널 종료 문자(\0)가 끝에 추가된 char 배열로 저장된다.

#### 기본 데이터 타입

Min-C는 사용자의 요구사항에 따라 모든 표준 정수 타입을 포함하며, C의 기본적인 메모리 모델을 지원하기 위한 필수 타입들을 정의한다.

* **정수 타입(Integer Types):**
  + char: 최소 8비트 크기의 정수 타입.
  + short int (또는 short): 최소 16비트 크기의 정수 타입.
  + int: 최소 16비트 크기의 정수 타입으로, 대상 아키텍처에서 가장 자연스러운 크기를 갖는다.
  + long int (또는 long): 최소 32비트 크기의 정수 타입.
  + long long int (또는 long long): 최소 64비트 크기의 정수 타입.
  + 위의 모든 정수 타입은 signed(기본값) 또는 unsigned 한정자와 함께 사용될 수 있다.
* **부동소수점 타입(Floating-Point Types):** 표준 C와의 완전한 표현력 동등성을 위해 부동소수점 연산을 지원한다.
  + float
  + double
  + long double
* **void 타입:** 타입이 없음을 명시적으로 나타내는 특수한 타입이다. 주로 값을 반환하지 않는 함수의 반환 타입이나, 일반화된 포인터(void\*)를 선언하는 데 사용된다.
* **포인터 타입(Pointer Types):** 모든 데이터 타입 T에 대해, T 타입의 객체를 가리키는 포인터 타입 T\*가 존재한다.

#### const 한정자의 배제

Min-C에서는 const 키워드를 의도적으로 배제한다. C 언어에서 const는 변수가 저장된 메모리를 물리적으로 쓰기 불가능하게 만드는 것을 보장하지 않는다. 대신, 컴파일러가 해당 변수에 대한 쓰기 시도를 컴파일 시점에 오류로 처리하도록 하는 약속에 가깝다.18 포인터를 통해

const 변수의 값을 변경하는 것이 기술적으로 가능하다.

이러한 const의 역할은 프로그래밍 규율을 통해 달성할 수 있으며, 명명된 상수가 필요한 경우, 언어 자체의 기능이 아닌 전처리기(preprocessor)의 #define 지시어를 사용하는 것이 더 원시적이고 미니멀리즘 철학에 부합한다.20 이는 언어의 핵심 명세를 단순하게 유지하면서 필요한 기능은 전처리 단계에서 해결하는 실용적인 타협안이다.

### 제4장: 연산자와 표현식

Min-C는 계산과 논리 연산을 수행하기 위한 최소한의 연산자 집합을 정의한다. 문법적 설탕에 해당하는 연산자들은 배제되고, 가장 근본적인 연산자들만이 명세에 포함된다.

#### 포함된 연산자

* **산술 연산자(Arithmetic Operators):** + (덧셈), - (뺄셈), \* (곱셈), / (나눗셈), % (나머지)
* **비트 연산자(Bitwise Operators):** & (비트 AND), | (비트 OR), ^ (비트 XOR), ~ (비트 NOT), << (왼쪽 시프트), >> (오른쪽 시프트)
* **논리 연산자(Logical Operators):** && (논리 AND), || (논리 OR), ! (논리 NOT). 이 연산자들은 if와 goto를 이용해 시뮬레이션할 수 있지만, 조건 평가의 핵심적인 역할을 하므로 명료성과 실용성을 위해 유지한다.
* **관계 연산자(Relational Operators):** == (같음), != (같지 않음), < (보다 작음), > (보다 큼), <= (작거나 같음), >= (크거나 같음)
* **할당 연산자(Assignment Operator):** = (단순 할당)
* **포인터 및 메모리 관련 연산자:** & (주소 연산자), \* (역참조 연산자), sizeof (크기 연산자)
* **멤버 접근 연산자(Member Access Operators):** . (직접 멤버 접근), `` (배열 첨자)

#### 배제된 연산자와 그 근거

* **구조체 포인터 멤버 접근 연산자 (->):**
  + **근거:** 사용자의 명시적인 요구사항에 따라 배제된다. 이 연산자는 (\*pointer).member 구문의 문법적 설탕이다.
  + **대체:** Min-C에서는 포인터를 통해 구조체 멤버에 접근할 때 반드시 (\*pointer).member 구문을 사용해야 한다. 이는 포인터를 먼저 역참조(\*)한 후, 그 결과 객체의 멤버에 접근(. )하는 과정을 코드에 명시적으로 드러내어, 연산의 의미를 더 명확하게 한다.
* **증감 연산자 (++, --):**
  + **근거:** 전위 및 후위 증감 연산자는 var = var + 1 또는 var = var - 1 연산의 축약형이다.
  + **대체:** 변수의 값을 1 증가시키거나 감소시키려면 var = var + 1 또는 var = var - 1 형태의 완전한 할당문을 사용해야 한다.
* **삼항 조건 연산자 (?:):**
  + **근거:** if-else 문의 간결한 표현으로, 문법적 설탕에 해당한다.11
  + **대체:** 조건에 따라 다른 값을 할당해야 할 경우, 명시적인 if-else 블록을 사용해야 한다.
* **콤마 연산자 (,):**
  + **근거:** 여러 표현식을 순차적으로 평가하는 기능은 대부분 세미콜론으로 문장을 분리하여 대체할 수 있다.14
  + **대체:** 순차적인 연산이 필요할 경우, 각 연산을 별도의 문장으로 작성한다.
* **복합 할당 연산자 (+=, \*=, 등):**
  + **근거:** var = var op expr 형태의 표현식을 축약한 문법적 설탕이다.7
  + **대체:** 모든 연산은 이항 연산자와 단순 할당 연산자 =를 조합하여 명시적으로 표현해야 한다. 예를 들어, x += 5는 x = x + 5로 작성한다.

### 제5장: 핵심 제어 흐름

Min-C는 제어 흐름을 위해 if와 goto라는 두 가지 원시적인 구문만을 제공한다. 이 최소한의 조합으로 표준 C의 모든 제어 구조(루프, 다중 분기 등)를 완벽하게 합성할 수 있음을 보인다.

#### 선택(Selection)

* **if-else 구문:** Min-C의 유일한 조건부 실행 메커니즘이다.
  + if (expression) statement
  + if (expression) statement1 else statement2

여기서 expression은 0이 아닌 값일 경우 참(true)으로, 0일 경우 거짓(false)으로 평가된다. statement는 단일 문장이거나 중괄호 {}로 묶인 복합 문장(블록)일 수 있다.

#### 분기(Branching)

* **goto 및 레이블(Label):** 프로그램 내의 지정된 위치로 제어를 무조건적으로 이전하는 유일한 메커니즘이다.
  + goto label;
  + label:

레이블은 함수 내에서 유일한 식별자이며, goto는 동일한 함수 내에 있는 레이블로만 점프할 수 있다.

#### 다른 제어 구조의 배제 및 합성

* **switch 문의 배제:**
  + **근거:** switch 문은 특정 변수의 값을 여러 상수와 비교하여 분기하는 구문으로, 이는 if-else if-else 연쇄 구조로 완벽하게 대체될 수 있다.23  
    switch는 가독성과 특정 상황에서의 컴파일러 최적화(점프 테이블 생성 등)를 위해 제공되는 편의 기능이지만, 언어의 최소성을 위해 배제된다.
  + **합성:** switch (var)는 if (var == const1) {... } else if (var == const2) {... }... else {... } 형태로 변환된다.
* **반복문 (for, while, do-while)의 배제:**
  + **근거:** 모든 반복 구조는 조건부 분기(if)와 무조건부 분기(goto)의 조합으로 합성될 수 있다. 이는 제1장에서 논의된 이론적 원칙의 직접적인 적용이다.
  + **합성 과정:**
    1. **for 루프:** for 루프는 while 루프의 문법적 설탕으로 간주될 수 있다.3  
       for (init; cond; inc) { body; }는 다음과 같이 while 루프로 변환된다.  
       C  
       init;  
       while (cond) {  
        body;  
        inc;  
       }
    2. **do-while 루프:** do-while 루프는 최소 한 번의 실행을 보장하는 반복문으로, 이 또한 while 루프와 유사한 구조로 변환 가능하다.34  
       do { body; } while (cond);는 다음과 같이 변환될 수 있다.  
       C  
       body;  
       while (cond) {  
        body;  
       }  
         
       또는 if/goto를 사용하면 더 직접적으로 표현된다.
    3. **while 루프의 if/goto 합성:** 모든 루프의 근간이 되는 while 루프는 if와 goto를 사용하여 다음과 같이 형식적으로 정의된다. while (cond) { body; }는 Min-C에서 다음과 같이 구현된다.  
       C  
       loop\_start:  
        if (!(cond)) goto loop\_end;  
        // body  
        goto loop\_start;  
       loop\_end:  
        // 루프 이후 코드 계속 실행

이러한 변환 규칙을 통해 Min-C는 if와 goto만으로 모든 종류의 반복 로직을 표현할 수 있음을 보장한다.

### 제6장: 함수와 프로그램 구조

함수는 코드의 재사용과 모듈화를 위한 가장 기본적인 추상화 메커니즘이다. 이는 문법적 설탕이 아닌 언어의 핵심 기능이므로, Min-C는 표준 C의 함수 관련 기능을 대부분 그대로 유지한다.

#### 함수 정의 및 선언

* **함수 정의(Function Definition):** 함수의 실제 코드를 포함하며, 반환 타입, 함수 이름, 매개변수 목록, 그리고 함수 본체(중괄호로 묶인 문장들의 집합)로 구성된다.  
  C  
  return\_type function\_name(parameter\_list) {  
   // 함수 본체  
  }
* **함수 선언(Function Declaration) / 프로토타입(Prototype):** 함수를 호출하기 전에 컴파일러에게 함수의 시그니처(반환 타입, 이름, 매개변수 타입)를 알려주는 역할을 한다. 이는 타입 검사를 가능하게 하고, 함수가 정의되기 전에 호출될 수 있도록 한다.  
  C  
  return\_type function\_name(parameter\_list);

#### 함수 호출 및 매개변수 전달

* **함수 호출(Function Call):** function\_name(argument\_list) 구문을 사용하여 함수를 실행한다.
* **매개변수 전달(Parameter Passing):** Min-C는 표준 C와 동일하게 값에 의한 전달(pass-by-value) 방식을 사용한다. 함수에 인자를 전달할 때, 인자의 값이 복사되어 함수의 매개변수에 할당된다. 따라서 함수 내에서 매개변수의 값을 변경해도 호출자(caller)의 원본 변수에는 영향을 미치지 않는다. 포인터를 전달함으로써 참조에 의한 전달(pass-by-reference)을 시뮬레이션할 수 있다.

#### return 문

return 문은 함수의 실행을 종료하고 제어를 호출자에게 반환한다. void가 아닌 함수는 반드시 return 문을 통해 값을 반환해야 하며, 반환되는 값의 타입은 함수 선언에 명시된 반환 타입과 호환되어야 한다. void 함수에서는 값을 반환하지 않는 return;을 사용하거나, 함수 본체의 끝에 도달하면 암묵적으로 반환된다.

#### 프로그램 진입점 (main 함수)

모든 Min-C 프로그램은 main이라는 이름의 함수에서 실행을 시작해야 한다. main 함수는 프로그램의 필수적인 진입점(entry point)이다. main 함수의 표준적인 형태는 다음과 같다.

C

int main(void) {  
 // 프로그램 코드  
 return 0;  
}  
  
int main(int argc, char\* argv) {  
 // 프로그램 코드  
 return 0;  
}

argc는 명령행 인자의 개수를, argv는 인자 문자열들의 배열을 가리키는 포인터이다.

#### 재귀(Recursion)

Min-C는 함수 호출 메커니즘을 통해 재귀를 자연스럽게 지원한다. 함수는 자기 자신을 직접 또는 간접적으로 호출할 수 있으며, 이는 재귀적인 알고리즘을 구현하는 데 필수적이다.

#### 함수 포인터 (Function Pointers)

함수 포인터는 C 언어의 핵심 기능 중 하나로, 문법적 설탕이 아닌 필수적인 요소이다. 함수 포인터는 함수의 메모리 주소를 변수에 저장하여, 런타임에 어떤 함수를 호출할지 동적으로 결정할 수 있게 해준다. 이는 콜백(callback) 메커니즘, 고차 함수(higher-order functions) 시뮬레이션, 상태 기계(state machine) 구현 등 고급 프로그래밍 기법에 필수적이므로 Min-C에 포함된다.

* **선언:** 함수 포인터는 반환 타입과 매개변수 타입을 명시하여 선언한다.  
  C  
  return\_type (\*pointer\_name)(parameter\_type\_list);  
    
  예를 들어, int를 반환하고 두 개의 int를 인자로 받는 함수를 가리키는 포인터 func\_ptr은 다음과 같이 선언한다.  
  C  
  int (\*func\_ptr)(int, int);
* **할당:** 함수의 이름(주소)을 함수 포인터 변수에 할당한다.  
  C  
  func\_ptr = &add; // & 연산자는 선택 사항  
  func\_ptr = add; // 이 또한 유효함
* **호출:** 함수 포인터를 통해 함수를 호출할 때는 포인터를 역참조하여 호출한다.  
  C  
  int result = (\*func\_ptr)(10, 20);  
    
  Min-C의 명시성 원칙에 따라, 포인터를 역참조하는 (\*func\_ptr) 형태의 호출을 표준으로 간주한다.

### 제7장: 집합 데이터 타입과 메모리 모델

Min-C는 데이터를 구조화하고 그룹화하기 위한 핵심적인 메커니즘으로 struct와 배열을 제공한다. union, 비트 필드, enum, typedef와 같은 편의 기능들은 배제되며, 이들의 기능은 더 근본적인 구조를 통해 시뮬레이션하는 방법을 제시한다.

#### struct (구조체)

* **정의:** struct는 서로 다른 타입의 변수들을 하나의 논리적 단위로 묶는 사용자 정의 복합 데이터 타입이다. 이는 Min-C에서 새로운 데이터 타입을 생성하는 주된 방법이다.  
  C  
  struct tag\_name {  
   type1 member1;  
   type2 member2;  
   //...  
  };
* **사용:** 구조체 변수는 struct tag\_name var;와 같이 선언하며, 멤버에는 . 연산자를 사용하여 접근한다 (var.member1).

#### 배열(Arrays)

* **정의:** 배열은 동일한 타입의 요소들이 메모리에 연속적으로 저장된 집합이다.
* **선언:** type array\_name[size]; 형태로 선언한다.
* **사용:** 배열 요소에는 첨자 연산자 ``를 사용하여 접근한다 (array\_name[index]).

#### 배제된 집합 데이터 타입 및 시뮬레이션

* **union의 배제:**
  + **근거:** union은 여러 멤버가 동일한 메모리 공간을 공유하도록 하는 기능이다. 이는 타입 시스템을 우회하여 메모리를 다른 방식으로 해석하는, 즉 타입 퍼닝(type-punning)에 주로 사용된다. 이 기능은 더 명시적인 방법으로 시뮬레이션 가능하다.
  + **시뮬레이션:** union의 기능은 충분한 크기의 char 배열을 할당하고, 해당 배열을 가리키는 포인터를 원하는 타입의 포인터로 명시적으로 형변환(casting)하여 구현할 수 있다.40 이 방식은 메모리 재해석이 일어나는 지점을 코드 상에 명확히 드러내어, 저수준 메모리 조작의 의도를 분명히 한다.  
    C  
    // union { int i; float f; } u; 를 시뮬레이션  
    char buffer[43]; // sizeof(int)와 sizeof(float) 중 큰 값 이상  
    int\* i\_ptr = (int\*)buffer;  
    float\* f\_ptr = (float\*)buffer;  
    \*i\_ptr = 10; // 정수로 사용  
    // 이제 \*f\_ptr을 통해 해당 메모리를 부동소수점으로 접근
* **비트 필드(Bit-fields)의 배제:**
  + **근거:** 비트 필드는 구조체 내에서 멤버 변수가 차지할 비트 수를 직접 지정하여 메모리를 절약하는 구문이다.44 이 기능은 표준 정수 타입과 비트 연산자(마스크, 시프트)를 사용하여 완벽하게 복제할 수 있다.
  + **시뮬레이션:** 비트 단위의 데이터 패킹은 unsigned int와 같은 변수를 선언하고, 비트 마스크(&)와 시프트 연산(<<, >>)을 통해 특정 비트들을 읽고 쓰는 방식으로 구현한다. 이는 하드웨어를 직접 제어하는 임베디드 프로그래밍에서 흔히 사용되는 기법이며, 언어에 내장된 특수 기능 대신 기본적인 연산자를 사용한다는 Min-C의 철학과 일치한다.  
    C  
    // struct { unsigned int flag1:1; unsigned int flag2:1; } s; 를 시뮬레이션  
    unsigned int flags;  
    // flag1 설정:  
    flags = flags | (1 << 0);  
    // flag2 값 확인:  
    unsigned int is\_flag2\_set = (flags >> 1) & 1;
* **enum (열거형)의 배제:**
  + **근거:** enum은 연관된 정수 상수들에 이름을 부여하는 문법적 설탕이다.45
  + **시뮬레이션:** 이 기능은 전처리기 지시어인 #define을 사용하여 간단히 대체할 수 있다. #define은 더 원시적인 메커니즘이며, 언어 명세 자체를 간결하게 유지한다.  
    C  
    // enum { RED, GREEN, BLUE } color; 를 시뮬레이션  
    #define RED 0  
    #define GREEN 1  
    #define BLUE 2  
    int color;
* **typedef의 배제:**
  + **근거:** typedef는 기존 타입에 새로운 별칭(alias)을 부여하는 순수한 편의 기능이다.49 이는 새로운 기능을 추가하지 않으며, 언어의 최소성을 위해 제거된다.
  + **대체:** 프로그래머는 typedef으로 축약하는 대신 항상 완전한 기본 타입 이름을 사용해야 한다. 예를 들어, typedef struct MyStruct\_s MyStruct; 대신, 변수를 선언할 때마다 struct MyStruct\_s my\_var;와 같이 전체 타입을 명시해야 한다.

### 제8장: 포인터, 주소, 그리고 역참조

포인터는 C 언어의 정체성이자 강력함의 원천이다. 메모리 주소를 직접 다루고, 동적 메모리 할당을 가능하게 하며, 효율적인 데이터 구조를 구현하는 데 필수적이다. 따라서 포인터와 관련된 핵심 기능들은 Min-C의 필수 불가결한 부분으로, 어떠한 축소도 없이 온전히 보존된다.

#### 포인터의 기본 개념

* **포인터 변수:** 다른 변수의 메모리 주소를 값으로 저장하는 변수이다.
* **선언:** type \*pointer\_name; 형태로 선언한다. 이는 pointer\_name이 type 타입의 변수를 가리키는 포인터임을 의미한다.

#### 핵심 포인터 연산자

* **주소 연산자 (&):** 변수 앞에 사용하여 해당 변수의 메모리 주소를 반환한다.  
  C  
  int var = 10;  
  int \*ptr = &var; // ptr은 var의 주소를 저장한다.
* **역참조 연산자 (\*):** 포인터 변수 앞에 사용하여 해당 포인터가 가리키는 메모리 위치에 저장된 값을 읽거나 쓴다. 이를 역참조(dereferencing)라고 한다.  
  C  
  int value = \*ptr; // ptr이 가리키는 곳의 값(10)을 value에 할당한다.  
  \*ptr = 20; // ptr이 가리키는 곳(var)의 값을 20으로 변경한다.

#### 포인터 연산(Pointer Arithmetic)

Min-C는 포인터에 정수를 더하거나 빼는 포인터 연산을 지원한다. 포인터에 정수 n을 더하면, 포인터는 n \* sizeof(\*pointer) 바이트만큼 메모리 주소상에서 앞으로 이동한다. 이는 배열의 요소들을 순회하는 데 매우 유용하다.

#### 배열과 포인터의 관계

C 언어의 핵심 개념 중 하나인 배열과 포인터의 긴밀한 관계는 Min-C에서도 동일하게 유지된다. 대부분의 표현식에서 배열의 이름은 배열의 첫 번째 요소를 가리키는 포인터로 암묵적으로 변환된다. 따라서 다음 두 표현은 동일한 의미를 갖는다.

* array[i]
* \*(array + i)

이러한 특성 덕분에 배열을 함수에 전달할 때 효율적인 포인터 전달이 이루어진다.

#### void\* (일반 포인터)

void\*는 특정 타입을 가리키지 않는 일반적인(generic) 포인터 타입이다. 어떠한 타입의 포인터든 void\*로 변환될 수 있으며, 그 반대도 가능하다(명시적 형변환 필요). 이는 malloc과 같은 메모리 할당 함수나, 다양한 데이터 타입을 처리해야 하는 일반화된 함수를 작성하는 데 필수적이다. void\* 포인터는 직접 역참조할 수 없으며, 반드시 구체적인 타입의 포인터로 형변환한 후에 사용해야 한다.

## Part III: 표현력 분석 및 실제 적용

이 파트에서는 Min-C가 이론적으로 완전할 뿐만 아니라, 실제 복잡한 프로그래밍 작업에도 적용 가능하다는 것을 증명한다. 표준 C의 모든 기능을 Min-C로 변환하는 포괄적인 가이드를 제공하고, 구체적인 사례 연구를 통해 Min-C의 실용성을 입증한다.

### 제9장: 표준 C에서 Min-C로: 변환 가이드

Min-C가 표준 C와 표현적으로 동등하다는 주장을 구체적으로 입증하기 위해, 표준 C의 배제된 기능들을 Min-C의 핵심 기능들로 변환하는 체계적인 방법을 제시한다. 아래 표는 각 표준 C 구문을 그에 상응하는 Min-C 패턴으로 매핑하는 "번역 사전" 역할을 한다. 이 변환 가이드는 프로그래머에게 실질적인 지침을 제공하며, Min-C 설계 과정에서 어떠한 표현력도 손실되지 않았음을 명확히 보여준다.

이 표는 Min-C가 단순한 이론적 실험이 아니라, 실제 C 코드를 변환하고 이해하는 데 사용할 수 있는 실용적인 명세임을 증명하는 핵심적인 증거이다. 각 변환 패턴은 Min-C가 어떻게 더 적은 구문 요소로 동일한 논리적 작업을 수행하는지를 명확하게 보여준다.

| 표준 C 구문 | Min-C 등가 패턴 |
| --- | --- |
| for (init; cond; inc) { body; } | init; loop\_start: if (!(cond)) goto loop\_end; body; inc; goto loop\_start; loop\_end: |
| while (cond) { body; } | loop\_start: if (!(cond)) goto loop\_end; body; goto loop\_start; loop\_end: |
| do { body; } while (cond); | loop\_start: body; if (cond) goto loop\_start; |
| switch (expr) { case C1: S1; break;... } | if (expr == C1) { S1; } else if... |
| var->member | (\*var).member |
| var++ / ++var | var = var + 1 |
| var-- / --var | var = var - 1 |
| var += expr | var = var + expr |
| result = cond? val1 : val2; | if (cond) { result = val1; } else { result = val2; } |
| union { T1 m1; T2 m2; } | char buffer; T1\* p1 = (T1\*)&buffer; T2\* p2 = (T2\*)&buffer; |
| struct { int x:1; int y:2; } | unsigned int flags; /\* x 조작 \*/ flags = (flags & ~1) | (val << 0); /\* y 조작 \*/ flags = (flags & ~6) | (val << 1); |
| enum { A, B, C }; | #define A 0 #define B 1 #define C 2 |
| typedef T new\_name; | *등가 패턴 없음. 항상 전체 타입 T를 사용해야 함.* |

### 제10장: 사례 연구 - 커널 스타일 연결 리스트 구현

Min-C가 이론적 완전성을 넘어 복잡한 시스템 프로그래밍 작업에 실질적으로 사용될 수 있음을 보이기 위해, 운영체제 커널에서 흔히 볼 수 있는 원형 이중 연결 리스트(circular, doubly-linked list)를 Min-C로 구현하는 사례를 제시한다. 이 예제는 포인터 조작, 구조체 사용, 그리고 if/goto를 이용한 제어 흐름 구성이 어떻게 결합되어 실용적인 자료구조를 만들어내는지를 보여준다.

이 사례 연구의 핵심은 Min-C의 제약 조건, 특히 -> 연산자의 부재가 코드에 어떻게 반영되는지를 명확히 보여주는 데 있다. 모든 포인터를 통한 멤버 접근은 (\*pointer).member 구문을 사용해야 하므로, 포인터 역참조 과정이 코드에 명시적으로 드러난다. 이는 Min-C가 추구하는 명시성의 원칙이 실제 코드에서 어떻게 작동하는지를 보여주는 강력한 예시이다.

(참고: 동적 메모리 할당을 위해 표준 라이브러리 함수 malloc과 free가 제공된다고 가정한다.)

#### 연결 리스트 노드 구조체 정의

C

/\* list\_head 구조체 정의 \*/  
struct list\_head {  
 struct list\_head \*next;  
 struct list\_head \*prev;  
};

#### 리스트 초기화 함수

C

/\* 리스트 헤드를 초기화하는 함수. 헤드가 자기 자신을 가리키도록 함. \*/  
void list\_init(struct list\_head \*head) {  
 (\*head).next = head;  
 (\*head).prev = head;  
}

#### 리스트에 노드 추가 함수 (list\_add)

C

/\* 새로운 노드(new\_node)를 헤드(head) 바로 뒤에 추가 \*/  
void list\_add(struct list\_head \*new\_node, struct list\_head \*head) {  
 struct list\_head \*next\_node;  
 next\_node = (\*head).next;  
  
 (\*next\_node).prev = new\_node;  
 (\*new\_node).next = next\_node;  
 (\*new\_node).prev = head;  
 (\*head).next = new\_node;  
}

#### 리스트 순회 및 출력 예제 (for 루프 대신 if/goto 사용)

C

/\* 데이터 저장을 위한 구조체 \*/  
struct my\_data {  
 int value;  
 struct list\_head list; /\* 리스트 헤드를 멤버로 포함 \*/  
};  
  
/\* 리스트를 순회하며 각 노드의 값을 출력하는 함수 \*/  
void print\_list(struct list\_head \*head) {  
 struct list\_head \*current;  
 struct my\_data \*data\_entry;  
  
 current = (\*head).next;  
  
loop\_start:  
 /\* current가 head와 같으면 루프 종료 (리스트의 끝) \*/  
 if (current == head) goto loop\_end;  
  
 /\*  
 \* list\_head 포인터로부터 my\_data 구조체의 시작 주소를 계산.  
 \* 이는 C 커널에서 container\_of 매크로가 하는 역할과 유사하며,  
 \* Min-C의 기본 연산만으로 구현 가능.  
 \* (단순화를 위해 여기서는 직접적인 주소 계산 대신 개념만 설명)  
 \* 실제 구현에서는 (char\*)current - offsetof(struct my\_data, list) 와 같은  
 \* 포인터 연산이 필요하지만, offsetof 또한 매크로이므로  
 \* Min-C에서는 주소 계산을 수동으로 해야 함.  
 \* 여기서는 개념적 설명을 위해 캐스팅을 사용.  
 \*/  
 data\_entry = (struct my\_data \*)current;  
   
 /\* 값 출력 - printf가 제공된다고 가정 \*/  
 /\* printf("Value: %d\n", (\*data\_entry).value); \*/  
  
 /\* 다음 노드로 이동 \*/  
 current = (\*current).next;  
 goto loop\_start;  
  
loop\_end:  
 return;  
}

이 사례 연구는 for 루프, -> 연산자, typedef 와 같은 편의 기능 없이도, Min-C의 핵심 구성 요소만으로 복잡하고 실용적인 코드를 작성할 수 있음을 명백히 보여준다. (\*ptr).member와 같은 명시적인 구문과 if/goto를 통한 제어 흐름 구성은 코드를 더 장황하게 만들 수 있지만, 동시에 모든 연산과 제어의 흐름을 가감 없이 드러내는 효과를 가져온다. 이는 저수준 시스템을 분석하고 디버깅하는 데 있어 오히려 장점이 될 수 있다.

## 결론

본 보고서는 표준 C 언어의 모든 표현력을 보존하면서, 불필요한 중복과 문법적 설탕을 제거하여 최소한의 기능 집합만을 갖는 'Min-C' 언어 명세를 제시했다. Min-C의 설계는 다음과 같은 핵심 원칙에 기반한다.

1. **표현적 동등성 유지:** Min-C는 튜링 완전하며, 표준 C로 작성 가능한 모든 알고리즘을 구현할 수 있다. 이는 if와 goto를 핵심 제어 흐름 원시 기능으로 채택하고, 다른 모든 제어 구조를 이들의 조합으로 합성함으로써 보장된다.
2. **문법적 설탕의 체계적 제거:** 프로그래머의 편의를 위해 추가된 구문들(->, ++, ?:, += 등)을 제거하고, 모든 연산이 가장 근본적인 형태로 표현되도록 강제한다. 이는 언어의 문법을 단순화하고, 코드의 의미를 명확하게 하며, 컴파일러 및 정적 분석 도구의 구현을 용이하게 한다.
3. **핵심 원시 기능 중심:** 함수, 구조체, 배열, 포인터와 같이 C 언어의 힘과 유연성의 근간이 되는 핵심 기능들은 온전히 보존한다. 반면, union, 비트 필드, enum과 같이 그 기능이 핵심 원시 기능의 조합으로 시뮬레이션될 수 있는 부가 기능들은 배제한다.

결론적으로, Min-C는 표준 C의 복잡성을 제거한 순수한 형태의 C 언어이다. 비록 간결함이나 일부 편의성은 희생되었지만, 그 대가로 언어의 명료성, 예측 가능성, 그리고 형식적 분석의 용이성을 얻었다. 이러한 특성은 Min-C를 컴퓨터 과학 교육, 형식 검증, 고도로 특화된 컴파일러 개발, 그리고 프로그래밍 언어의 근본 원리를 탐구하고자 하는 연구자들에게 매우 강력하고 가치 있는 도구로 만들어준다. Min-C는 C 언어의 본질이 무엇인지에 대한 깊은 통찰을 제공하며, 최소한의 요소만으로도 완전한 계산 능력을 갖출 수 있음을 증명하는 구체적인 실체이다.

#### 참고 자료

1. Böhm-Jacopini theorem - Stack Overflow, 9월 27, 2025에 액세스, <https://stackoverflow.com/questions/40333950/b%C3%B6hm-jacopini-theorem>
2. APP Bohm -Jacopini Structured Programming Theorem A foundation for Structured Programming - YouTube, 9월 27, 2025에 액세스, <https://www.youtube.com/watch?v=Oyl-6xmWScA>
3. What the difference between a while and for loop? : r/C\_Programming - Reddit, 9월 27, 2025에 액세스, <https://www.reddit.com/r/C_Programming/comments/1npr9pt/what_the_difference_between_a_while_and_for_loop/>
4. goto Statement in C - GeeksforGeeks, 9월 27, 2025에 액세스, <https://www.geeksforgeeks.org/c/goto-statement-in-c/>
5. goto Statement in C Programming (With Examples) - WsCube Tech, 9월 27, 2025에 액세스, <https://www.wscubetech.com/resources/c-programming/goto>
6. Valid use of goto for error management in C? - Stack Overflow, 9월 27, 2025에 액세스, <https://stackoverflow.com/questions/788903/valid-use-of-goto-for-error-management-in-c>
7. Augmented assignment - Wikipedia, 9월 27, 2025에 액세스, <https://en.wikipedia.org/wiki/Augmented_assignment>
8. Assignment Operators In C Explained With Proper Code Examples // Unstop, 9월 27, 2025에 액세스, <https://unstop.com/blog/assignment-operators-in-c>
9. Assignment operators - IBM, 9월 27, 2025에 액세스, <https://www.ibm.com/docs/en/xl-c-aix/13.1.0?topic=expressions-assignment-operators>
10. Assignment Operators in C - GeeksforGeeks, 9월 27, 2025에 액세스, <https://www.geeksforgeeks.org/c/assignment-operators-in-c-c/>
11. Ternary conditional operator - Wikipedia, 9월 27, 2025에 액세스, <https://en.wikipedia.org/wiki/Ternary_conditional_operator>
12. Conditional or Ternary Operator (?:) in C - GeeksforGeeks, 9월 27, 2025에 액세스, <https://www.geeksforgeeks.org/c/conditional-or-ternary-operator-in-c/>
13. Should I utilise the ternary operator over a simple if statement? - Quora, 9월 27, 2025에 액세스, <https://www.quora.com/Should-I-utilise-the-ternary-operator-over-a-simple-if-statement>
14. Comma in C - GeeksforGeeks, 9월 27, 2025에 액세스, <https://www.geeksforgeeks.org/c/comma-in-c/>
15. Comma operator - Wikipedia, 9월 27, 2025에 액세스, <https://en.wikipedia.org/wiki/Comma_operator>
16. Dark Corners of C - The Comma Operator - Stephen Friederichs - EmbeddedRelated.com, 9월 27, 2025에 액세스, <https://www.embeddedrelated.com/showarticle/804.php>
17. Why does C provide both the comma operator and the semicolon to separate statements?, 9월 27, 2025에 액세스, <https://softwareengineering.stackexchange.com/questions/368618/why-does-c-provide-both-the-comma-operator-and-the-semicolon-to-separate-stateme>
18. const (computer programming) - Wikipedia, 9월 27, 2025에 액세스, <https://en.wikipedia.org/wiki/Const_(computer_programming)>
19. Const Qualifier in C - GeeksforGeeks, 9월 27, 2025에 액세스, <https://www.geeksforgeeks.org/c/const-qualifier-in-c/>
20. Constant In C | Types, Syntax, Uses (Explained With Examples) // Unstop, 9월 27, 2025에 액세스, <https://unstop.com/blog/constant-in-c>
21. Constants in C - GeeksforGeeks, 9월 27, 2025에 액세스, <https://www.geeksforgeeks.org/c/constants-in-c/>
22. How do I best use the const keyword in C? - Stack Overflow, 9월 27, 2025에 액세스, <https://stackoverflow.com/questions/14401856/how-do-i-best-use-the-const-keyword-in-c>
23. Difference Between if-else and switch in C - GeeksforGeeks, 9월 27, 2025에 액세스, <https://www.geeksforgeeks.org/cpp/difference-between-if-else-and-switch-in-c/>
24. Switch statement - Wikipedia, 9월 27, 2025에 액세스, <https://en.wikipedia.org/wiki/Switch_statement>
25. switch Statement (C) - Microsoft Learn, 9월 27, 2025에 액세스, <https://learn.microsoft.com/en-us/cpp/c-language/switch-statement-c?view=msvc-170>
26. When to use If/Else, and when to use Switch? : r/C\_Programming - Reddit, 9월 27, 2025에 액세스, <https://www.reddit.com/r/C_Programming/comments/1f9oc8b/when_to_use_ifelse_and_when_to_use_switch/>
27. Switch statement vs if else - C++ Forum, 9월 27, 2025에 액세스, <https://cplusplus.com/forum/beginner/139288/>
28. When to use If-else if-else over switch statements and vice versa [duplicate] - Stack Overflow, 9월 27, 2025에 액세스, <https://stackoverflow.com/questions/427760/when-to-use-if-else-if-else-over-switch-statements-and-vice-versa>
29. loops - For vs. while in C programming? - Stack Overflow, 9월 27, 2025에 액세스, <https://stackoverflow.com/questions/2950931/for-vs-while-in-c-programming>
30. Convert A For Loop To A While Loop | C Programming Tutorial - YouTube, 9월 27, 2025에 액세스, <https://www.youtube.com/watch?v=q7vdQJ3Y0CE>
31. Difference between For, While and Do-While Loop in Programming - GeeksforGeeks, 9월 27, 2025에 액세스, <https://www.geeksforgeeks.org/dsa/difference-between-for-while-and-do-while-loop-in-programming/>
32. How can I convert for loop into while loop? - Quora, 9월 27, 2025에 액세스, <https://www.quora.com/How-can-I-convert-for-loop-into-while-loop>
33. Loops in C - GeeksforGeeks, 9월 27, 2025에 액세스, <https://www.geeksforgeeks.org/c/c-loops/>
34. Difference between while and do-while loop in C, C++, Java - GeeksforGeeks, 9월 27, 2025에 액세스, <https://www.geeksforgeeks.org/java/difference-between-while-and-do-while-loop-in-c-c-java/>
35. 'do...while' vs. 'while' - Stack Overflow, 9월 27, 2025에 액세스, <https://stackoverflow.com/questions/3347001/do-while-vs-while>
36. Do/While vs. While loops? : r/csharp - Reddit, 9월 27, 2025에 액세스, <https://www.reddit.com/r/csharp/comments/tb19nj/dowhile_vs_while_loops/>
37. Do while loop - Wikipedia, 9월 27, 2025에 액세스, <https://en.wikipedia.org/wiki/Do_while_loop>
38. I really don't get while, do-while, and for loops in C. : r/learnprogramming - Reddit, 9월 27, 2025에 액세스, <https://www.reddit.com/r/learnprogramming/comments/1855jn7/i_really_dont_get_while_dowhile_and_for_loops_in_c/>
39. Comparing while vs do-while loops - Stack Overflow, 9월 27, 2025에 액세스, <https://stackoverflow.com/questions/65865052/comparing-while-vs-do-while-loops>
40. Character Array and Character Pointer in C - C Programming Tutorial - OverIQ.com, 9월 27, 2025에 액세스, <https://overiq.com/c-programming-101/character-array-and-character-pointer-in-c/index.html>
41. Type-casting with Structure Pointer, 9월 27, 2025에 액세스, <http://c-pointers.com/typecasting_n_ptr/sp_struct_ptr.html>
42. Using char array inside union - c++ - Stack Overflow, 9월 27, 2025에 액세스, <https://stackoverflow.com/questions/15952204/using-char-array-inside-union>
43. Bit Fields in C - GeeksforGeeks, 9월 27, 2025에 액세스, <https://www.geeksforgeeks.org/c/bit-fields-c/>
44. When to Use Enum Instead of Define in C? - GeeksforGeeks, 9월 27, 2025에 액세스, <https://www.geeksforgeeks.org/c/when-to-use-enum-instead-of-define-in-c/>
45. Tips and Tricks – Using enum over #define | Beningo Embedded Group, 9월 27, 2025에 액세스, <https://www.beningo.com/tips-and-tricks-using-enum-over-define/>
46. learn.microsoft.com, 9월 27, 2025에 액세스, <https://learn.microsoft.com/en-us/cpp/c-language/c-enumeration-declarations?view=msvc-170#:~:text=Enumerations%20provide%20an%20alternative%20to,constant%20always%20have%20int%20type.>
47. typedef enum vs enum : r/cprogramming - Reddit, 9월 27, 2025에 액세스, <https://www.reddit.com/r/cprogramming/comments/oqjfkj/typedef_enum_vs_enum/>
48. What are some alternatives to using typedef in C++? - Quora, 9월 27, 2025에 액세스, <https://www.quora.com/What-are-some-alternatives-to-using-typedef-in-C>
49. What is the difference between 'typedef' and 'using'? - Codemia.io, 9월 27, 2025에 액세스, <https://codemia.io/knowledge-hub/path/what_is_the_difference_between_typedef_and_using>
50. typedefs best alternative - Amr Shams's blog, 9월 27, 2025에 액세스, <https://nightbird007.hashnode.dev/typedefs-best-alternative>
51. What is the difference between 'typedef' and 'using'? - Stack Overflow, 9월 27, 2025에 액세스, <https://stackoverflow.com/questions/10747810/what-is-the-difference-between-typedef-and-using>