C 언어의 복잡한 규약들

서울대학교 ▌허충길*

1. C 언어 규약들의 필요성

많은 C 프로그래머들이 컴파일러 최적화와 관련된 복잡한 C 언어의 규약들을 잘 모르고 있다. 이러한 규약을 어기고 프로그램을 작성하면 컴파일러가 아무런 경고 없이 프로그램을 잘 못 컴파일 할 수 있다. 이럴 경우 프로그래머도 모르는 오류가 조용히 삽입되어 보안 문제 등이 생길 수도 있으며, 오류를 발견했다 하더라도 그 원인을 찾는데 큰 어려움을 겪을 수 있다.

Java와 같은 고수준(High-level) 언어와는 달리 C 언어가 컴파일러 최적화 관련 규약을 필요로 하는 이유는 성능 최적화를 최우선 목표로 설계되었기 때문이다. 고수준 언어에서 프로그래머는 프로그램의 행동을 고수준에서 기술하도록 되어 있고 컴파일러는 저수준(Low-level) 상세사항을 채우도록 되어 있어 둘 사이에 큰 충돌이 발생하지 않는다. 반면 C 언어에서는 프로그래머가 자신이 원하는 성능 최적화를 위해 프로그램의 행동을 저수준에서 직접 기술할 수 있으며, 컴파일러는 또한 이렇게 기술된 저수준 행동을 더욱 최적화 한다. 이와 같이 프로그래머와 컴파일러 모두 저수준에서 작업하므로 조심하지 않으면, 즉 복잡한 규약을 지키지 않으면, 의도치 않게 잘못된 결과가 초래될 수 있다.

2. C 언어 규약들 소개

C 언어의 복잡한 규약들은 ISO C Standard [1]에 정의되어 있다. 이 중 몇 가지 대표적인 규약들을 예를 들어 소개하겠다. 예제 프로그램들 컴파일에는 Apple, Google등에서 많이 사용하는 clang version 3.8.1을 사용했다.

2.1 부호 있는 정수 넘침(Signed Integer Overflow)에 관한 규약

부호 있는 정수의 계산이 넘쳤을 때(Signed Integer Overflow 발생시) 적용되는 규약을 소개한다. 아래의 예제 프로그램 ex1.c를 보자.

[ex1.c]

```
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>
int main(int argc, char** argv){
  int n = atoi(argv[1]);
  int len = atoi(argv[2]);
  if (len >= 0 && n <= n + len)
    printf("OK: %d <= %d \n", n, n + len);
}
```

이 프로그램은 정수 n과 길이 len을 사용자로부터 입력받아 len이 0이상이고 n+len에서 넘침(Overflow)이 발생하지 않는 경우 그 값을 출력한다. 여기서 넘침이 발생하는지 확인하는 간단한 방법으로 n <= n+len을 계산해 볼 수 있다. 왜냐하면 n+len에서 넘침이 발생했다면 그 값은 반드시 n보다 작기 때문이다.

하지만 이 프로그램을 clang으로 컴파일해서 수행해보면 다음과 같다.

```
$ clang -O2 ex1.c -o ex1
$ ex1 100 2147483640
OK: 100 <= -2147483556
```

입력으로 주어진 len이 너무 커 넘침이 발생했음에도 잘못된 결과를 화면에 출력한다. 그 이유는 컴파일러 입장에서 len이 0 이상일 경우 n <= n+len은 수학적으로 항상 성립하는 식이라 생각하고 n <= n+len을 지우는 최적화를 수행했기 때문이다.

---

* 종신회원

† 이 글에서는 사람들이 잘 모르고 있는 C 언어의 여러 복잡한 규약들에 대해 소개한다. 또한 이러한 규약들에 관한 연구들도 간략히 소개한다.

이러한 최적화를 허용하기 위해 ISO C Standard는 부호 있는 정수 넘침을 발생시키는 모든 프로그램을 잘못된 프로그램으로 규정하고 있다. 위 프로그램은 넘침이 발생하는지 확인하는 과정(n <= n+len)에서 넘침을 발생시키므로 잘못된 프로그램이다. 따라서 위 프로그램에서 if문 조건을 아래와 같이 수정하면 규약을 지키게 되고 컴파일러 최적화와 충돌을 피할 수 있다.

```
if (len >= 0 && n <= INT_MAX - len)
```

2.2 초기화 되지 않은 변수(Uninitialized Variable)에 관한 규약

초기화 되지 않은 변수를 사용했을 때 적용되는 규약을 소개한다. 아래의 예제 프로그램 ex2.c를 보자.

[ex2.c]

```
#include <stdio.h>
int foo();
int main() {
  int x = foo();
  printf(x <  0 ? "negative \n" : "");
  printf(x >= 0 ? "non-negative \n" : "");
}
int bar(int i) { return i; }
int foo() { int i; return bar(i); }
```

이 프로그램은 foo()함수 호출 후 결과 값(x)이 음수면 “negative”를 출력하고, 음수가 아니면 “non-negative”를 출력한다.

하지만 이 프로그램을 clang으로 컴파일해서 수행해보면 다음과 같다.

```
$ clang -O2 ex2.c -o ex2
$ ex2
```

실행 후 “negative”나 “non-negative” 어떤 것도 출력되지 않는다. 그 이유는 foo() 함수가 반환한 값이 초기화 되지 않은 변수 i에서 읽은 값이기 때문이다. 컴파일러 입장에서는 이러한 값을 정의되지 않은 값(undefined value)으로 보고 x < 0 과 x >= 0 둘 다 0으로 최적화 해버린다.

이러한 최적화를 허용하기 위해 ISO C Standard는 초기화 되지 않은 변수에서 읽은 값을 사용하는 모든 프로그램을 잘못된 프로그램으로 규정하고 있다. 따라서 위 프로그램의 foo() 함수에서 아래와 같이 변수 i를 초기화하면 제대로 “non-negative”가 출력된다.

```
int foo() { int i = 0; return bar(i); }
```

2.3 타입이 다른 포인터 값(Pointer Value)들에 관한 규약

타입이 다른 포인터 값들을 참조할 때의 규약을 소개한다. 아래의 예제 프로그램 ex3.c를 보자.

[ex3.c]

```
#include <stdio.h>
#include <string.h>
int foo(int* iptr, short* sptr);
int main() {
  int x;
  printf("%X \n", foo(&x, (short*) &x));
}
int foo(int* iptr, short* sptr) {
  *iptr = -1;
  *sptr = 0;
  return *iptr;
}
```

이 프로그램은 변수 x 의 주소에 int 타입으로 -1 (즉, 0xFFFFFFFF)을 저장한 후 다시 short 타입으로 0(즉, 0x0000)을 덮어쓴다. 그 후 변수 x 에 저장된 정수 값을 읽어 출력한다. 결과는 FFFF0000로 clang -O0로 최적화 없이 컴파일 해 확인할 수 있다.

```
$ clang -O0 ex3.c -o ex3
$ ex3
FFFF0000
```

하지만 clang -O2로 최적화 기능을 켜면 FFFFFFFF가 출력된다.

```
$ clang -O2 ex3.c -o ex3
$ ex3
FFFFFFFF
```

그 이유는 컴파일러가 foo함수를 최적화 할 때 iptr과 sptr은 타입이 다르므로 서로 다른 메모리 영역을 가리킨다고 가정하기 때문이다. 이에 따라 컴파일러는 return *iptr 을 return -1 로 최적화한다.

이러한 최적화를 허용하기 위해 ISO C Standard는 서로 다른 타입으로 같은 메모리 영역을 참조하지 못하도록 규정하고 있다 (단, char*는 제외). 따라서 ex3.c는 C standard에 따르면 잘못된 프로그램이다.

이를 수정하기 위해서는 다음과 같이 memcpy 함수

를 사용할 수 있다.

```
int foo(int* iptr, short* sptr) {
  *iptr = -1;
  // *sptr = 0;
  short s = 0; memcpy(sptr, &s, sizeof(s));
  return *iptr;
}
```

이 프로그램은 C standard의 규약을 지키므로 clang -O2로 컴파일해도 FFFF0000이 제대로 출력된다. 뿐만 아니라 C 컴파일러들은 memcpy 함수를 특별하게 처리해서 memcpy를 사용하더라도 성능을 잃지 않도록 효율적으로 컴파일 한다. 예를 들어 memcpy를 사용한 위의 foo 함수는 clang -O2에 의해 아래와 같은 어셈블리 코드로 컴파일 된다.

```
_foo:
     movl    8(%esp), %eax
     movl    4(%esp), %ecx
     movl    $-1, (%ecx)
     movw    $0, (%eax)
     movl    (%ecx), %eax
     retl
```

여기서 보듯이 memcpy 함수는 없어지고 short s = 0; memcpy(sptr, &s, sizeof(s)) 코드가 *sptr = 0 만큼 효율적으로 컴파일 됐다.

2.4 끝나지 않는 프로그램에 관한 규약

아무 일도 하지 않으면서 끝나지 않는 프로그램에 관한 규약을 소개한다. 아래의 예제 프로그램 ex4.c를 보자.

[ex4.c]

```
#include <stdio.h>
#include <unistd.h>
void foo(int loop) {
  while (loop) {};
}
int main() {
  foo(1);
  printf("boom!\n");
}
```

이 프로그램은 foo(1) 을 호출한 후 무한 루프에 빠져 끝나지 않는다.

하지만 이 프로그램을 clang -O2로 컴파일해서 수행해보면 다음과 같다.

```
$ clang -O2 ex4.c -o ex4
$ ex4
boom!
```

즉, 실행하면 “boom!” 을 출력하면서 끝난다. 그 이유는 컴파일러가 foo 함수를 read-only 함수로 분석한 후, 반환값이 사용되지 않은 read-only 함수 호출(즉, foo(1))을 삭제하는 최적화를 수행했기 때문이다.

이러한 최적화를 허용하기 위해 ISO C Standard는 관찰 가능한 행동 없이 (즉, 아무 일도 하지 않으면서) 무한히 수행되는 것을 허용하지 않는다. 즉, ex4.c는 C standard를 따르지 않은 잘못된 프로그램이다.

이를 수정하기 위해서는 아래와 같이 pause() 시스템 콜을 사용할 수 있다.

```
void foo(int loop) {
  // while (loop) {};
  if (loop) pause();
}
```

이 프로그램은 pause()라는 시스템 콜(즉, 아무 일도 하지 말고 기다리라는 시스템 콜)을 명시적으로 호출하고, 이는 C standard에서 허용된다. 따라서 clang -O2로 컴파일해도 제대로 수행 된다(즉, 끝나지 않는다).

2.5 포인터 값(Pointer Value) 비교에 관한 규약

포인터 값을 비교할 때의 규약을 소개한다. 아래의 예제 프로그램 ex5.c를 보자.

[ex5.c]

```
#include <stdio.h>
int foo(int* arr, int len) {
  int sum = 0;
  for (int* end = arr+len; arr < end; arr += 2)
    sum += *arr;
  return sum;
}
int main() {
  int a[9];
  for (int i=0; i < 9; i++)
    a[i] = i;
  printf("%d\n", foo(a, 9));
}
```

우선 foo(arr, len) 함수는 배열 arr과 그 길이 len을 받아 짝수 번째 항의 값을 모두 더해 반환한다. 따라서 이 프로그램은 0,...,8로 구성된 배열 a의 짝수 번째 항의 합(즉, 0+2+4+6+8=20)을 출력한다.

하지만 이 프로그램은 ISO C Standard에 따르면 잘못된 프로그램이다. 그 이유는 foo 함수에서 for문이 끝날 때 마지막으로 변수 arr에 a+10 이 저장되기 때문이다. 구체적으로 말하면 길이가 len인 배열 a에 대해서는 오직 a+0부터 a+len까지만 포인터 값으로 사용되는 것이 허용된다. 위 프로그램에서 사용된 주소 a+10은 이 범위(a+0 ~ a+9)를 벗어나므로 규약을 어겼다. 이 프로그램이 비록 현재는 clang에 의해 제대로 컴파일 되지만, 어쨌든 규약을 어겼으므로 앞으로 새로 추가되는 최적화에 의해 잘못 컴파일 될 여지가 있다.

이를 해결하기 위해 위 foo함수에서 for문을 아래와 같이 수정하면 C standard를 지킬 수 있다.

```
for (int i = 0; i < len; i += 2)
  sum += arr[i];
```

2.6 C 언어 규약들에 관한 더 많은 자료들

지금까지 대표적인 규약들 몇 가지를 살펴보았다. C 언어 규약들에 관한 더 자세한 내용은 아래 웹사이트에 잘 설명되어 있다.

https://www.securecoding.cert.org/confluence/display/c

이 외에도 다음과 같은 재미있는 글들이 있다.

- http://blog.llvm.org/2011/05/what-every-c-programmer-should-know.html
- http://blog.regehr.org/archives/213
- http://blog.regehr.org/archives/1180

3. C 언어 규약 관련 연구 소개

ISO C Standard는 700페이지에 달할 정도로 복잡하고 또한 종종 정의가 모호하다. 이러한 문제를 해결하기 위해 크게 두 가지 방향의 연구가 진행 중이다. 간략하게 이들을 소개하는 것으로 이글을 마친다.

3.1 규약을 어긴 부분을 찾아주는 분석도구

주어진 C 프로그램이 여러 가지 규약을 잘 지켰는지 확인하고 어긴 부분을 찾아주는 분석 도구들을 개발하는 연구들이 진행되고 있다. 최근 연구결과로 다음과 같은 논문들이 있다.

[2] Towards optimization-safe systems: analyzing the impact of undefined behavior. In *SOSP*, 2013.

[3] Defining the undefinedness of C. In *PLDI*, 2015.

[4] A differential approach to undefined behavior detection. *CACM*, 2016.

3.2 C 언어의 규약을 개선하는 연구

복잡하고 모호한 C 언어의 규약을 간단하고 엄밀하게 만드는 연구도 진행 중이다. 이 연구는 우리 연구실이 세계적으로 주도하고 있다. 우리 연구실의 관련 연구 결과는 다음과 같다.

[5] A formal C memory model supporting integer-pointer casts. In *PLDI*, 2015.

[6] Taming undefined behavior in LLVM. In *PLDI*, 2017.

[7] A promising semantics for relaxed-memory concurrency. In *POPL*, 2017.

[8] Repairing sequential consistency in C/C++11. In *PLDI*, 2017.

관련 연구로 Robbert Krebbers는 현재의 ISO C Standard를 최대한 엄밀하게 정의하는 연구를 해왔고 그의 박사 논문에 잘 정리되어 있다.

[9] The C standard formalized in Coq. PhD thesis, 2015.

참고문헌

[1] ISO. ISO/IEC 9899:2011 Information technology – Programming languages – C. 2011.

[2] Xi Wang, Nickolai Zeldovich, M. Frans Kaashoek, Armando Solar-Lezama. Towards optimization-safe systems: analyzing the impact of undefined behavior. In *SOSP*, 2013.

[3] Chris Hathhorn, Chucky Ellison, Grigore Roşu. Defining the undefinedness of C. In *PLDI*, 2015.

[4] Xi Wang, Nickolai Zeldovich, M. Frans Kaashoek, Armando Solar-Lezama. A differential approach to undefined behavior detection. *CACM*, 2016.

[5] Jeehoon Kang, Chung-Kil Hur, William Mansky, Dmitri Garbuzov, Steve Zdancewic, Viktor Vafeiadis. A formal C memory model supporting integer-pointer casts. In *PLDI*, 2015.

[6] Juneyoung Lee, Yoonseung Kim, Youngju Song, Chung-Kil Hur, Sanjoy Das, David Majnemer, John Regehr, Nuno P. Lopes. Taming undefined behavior in LLVM. In *PLDI*, 2017.

[7] Jeehoon Kang, Chung-Kil Hur, Ori Lahav, Viktor Vafeiadis, Derek Dreyer. A promising semantics for relaxed-memory concurrency. In *POPL*, 2017.

[8] Ori Lahav, Viktor Vafeiadis, Jeehoon Kang, Chung-Kil Hur, Derek Dreyer. Repairing sequential consistency in C/C++11. In *PLDI*, 2017.

[9] Robbert Krebbers. *The C standard formalized in Coq.* PhD thesis, Radboud University Nijmegen, 2015.

약 력

허충길

2000 KAIST 전산학과 및 수학과 졸업 (학사)
2010 영국 University of Cambridge 졸업 (박사)
2009-2010 프랑스 Laboratoire PPS 박사후 연구원
2010-2012 독일 Max Planck Institute for Software Systems 박사후 연구원
2012-2013 영국 Microsoft Research Cambridge 박사후 연구원
2013-현재 한국 서울대학교 컴퓨터 공학부 조교수
관심분야: 소프트웨어 검증, 프로그래밍 언어, 컴파일러
Email: gil.hur@sf.snu.ac.kr