濟大論文集(自然), 37, 247-256, 1993
Cheju Univ. Jour. (Natural Sci.),
37, 247-256, 1993

C++ 프리프로세서의 설계와 구현

곽 호 영*

A Design and Implementation of C++ Preprocessor

*Ho-Young Kwak**

Summary

The current programming style gaurantees concept of abstraction and effects of information hiding. The concentration on the object-oriented programming which supports the concept expressing situation of realworld is increasing. This study has implemented C++ preprocessor which translate C++ language with object-oriented programming concept to C language. The translated C program can be executed regardless of a type of C compiler and a run-time library in a computer system.

서 론

현재의 프로그래밍 언어의 형식은 추상화 개념과 정보은폐가 보장되고 현실 세계의 상황들을 가장 적절히 표현할 수 있는 개념을 지원하고 있으며, 이들을 도구로 하는 객체지향 프로그래밍의 관심이 증가하는 가운데 Smalltalk, Ada, Actor, C++ 등의 많은 객체지향 개념을 포함한 언어들이 발표되면서 그 사용이 점차 증대되고 있다[Goldburg and Robinson, 1983; Lai, 1989; Pressman, 1987; Stroustrup, 1991]. 이 중에서도 가장 널리 사용되는 시스템 프로그래밍 언어인 C 언어가 객체지향 개념을 포함한 슈퍼셋(superset)인 C++ 언어로 확장되면서 C++ 언어는 사용자들에게 가장 친숙한 객체지향 언어로 그 사용이 증가되는 추세에 있다[Booch 1986, 1991, Ege 1989, Wiener and Pinson, 1988].

이러한 추세에 따라 기존에 C 언어에 친숙한 사용자들이 C++ 언어와 C 언어의 차이를 비교 검토할 수 있고 객체지향 개념을 쉽게 이해할 수 있도록 C++ 프리프로세서를 설계, 구현하였다. 이때 기존의 C컴파일러만 구비하고 있으면 구비된 C 컴파일러의 헤더화일과 라이브러리를 그대로 이용할 수 있는 범용 C++ 프리프로세서로 구현하여 시스템의 종류나 컴파일러의 종류에 관계없이 C++ 언어의 사용이 가능하도록 하였다[Aho *et al.* 1986; AT&T 1989].

본 연구에서 C++ 원시 프로그램의 번역 방법을 프리프로세서 방식으로 채택한 이유는 첫째, 컴퓨터 하드웨어에 의존하지 않는 소프트웨어로 개발하려는 것이며, 둘째로는 생성되는 목적 언어가 기존의 C 컴파일러만 구비되어 있으면 컴퓨터 시스템의 기종과 컴파일러의 종류에 관계없이 사용가능하도록 하

* 공과대학 정보공학과(Dept. of Information Engineering, Cheju Univ., Cheju-do, 690-756, Korea)

기 위한 것이다[원, 1991]. C++ 언어를 프리프로세서로 번역한 결과는 C 프로그램으로 생성되도록 하여 C++ 언어를 학습하는데 유용한 자료로 이용되도록 사용자가 요구할때 즉시 생성할 수 있게 처리하였다. 그리고, 기존의 C 컴파일러와 같이 실행시간 라이브러리(run-time library)에 독립적인 C++ 번역기를 구현함으로써 호환성(portability)을 높이도록 설계하였다. 따라서, 본 연구에서는 실행시간 라이브러리에 의존하지 않고 완전한 C 프로그램 형태로 목적 프로그램이 생성되도록 하였으며, RISC 기종인 MIPS R-2000에서 구현하였다.

방 법

1. 프리프로세서의 설계

C++ 프리프로세서는 Fig. 1과 같은 흐름에 의해서 목적 프로그램인 C 프로그램을 생성하며, 생성된 C 프로그램은 기존의 C 컴파일러에서 번역이 가능하도록 변환되고, 그 실행결과는 C++ 컴파일러에서 실행한 결과와 일치한다. C++ 언어의 문법은 ISO '93 Standardization Grammar와 호환되도록 하였다 [ISO, 1993].

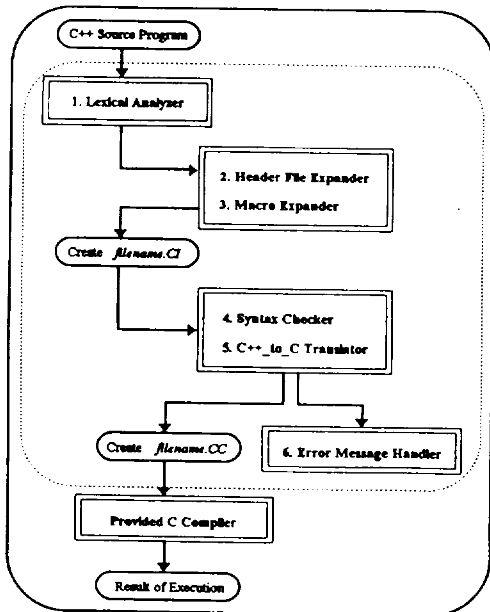

Fig. 1. Structure of C++ Preprocessor.

Fig. 1의 각 처리기들은 다음과 같은 기능을 갖는다.

1. 1. 어휘분석기(Lexical Analyzer)

이 부분은 C++ 원시 프로그램의 토큰(token) 인식을 위하여 오토마타를 이용한 어휘분석기이다. 이 어휘분석기는 UNIX 시스템에서 제공되는 어휘분석기 생성기인 Lex를 이용하여 처리했으며, Lex 입력에 필요한 C 프로그램 일부와 ISO Standard C++ 언어에서 정의된 예약어(reserved word)를 정의하였다. 이 어휘분석기의 출력은 다음 단계인 헤더화일 확장기와 매크로 확장기의 입력으로 사용되며, 주요 함수로는 yylex( )가 있다.

1. 2. 헤더화일 확장기(Header file Expander)

C++ 원시 프로그램에 선언된 헤더화일을 찾아 원시 프로그램에 확장시키고, 이 결과를 "*filename. CI*"에 출력한다. 이때 확장될 헤더화일은 크게 두 가지 형태를 가지며, 그 처리는 다음과 같다.

 # include "file1. h"

현재의 디렉토리(directory)에서 정의된 화일명 "file1. h"라는 화일을 찾아 출력화일인 *filename. CI*에 포함시킨다.

 # include <*file2. h*>

시스템의 시스템 포함 화일(*system include file*)이 있는 디렉토리에서 정의된 화일명 "*file2. h*"라는 화일을 찾아 출력인 *filename. CI*에 포함시킨다. (구현 기종인 MIPS R-2000에서 시스템 포함 화일이 있는 디렉토리는 "*/usr/c++/include*"이다.) 위의 두 경우 모두 내포(nested)되어 정의될 수 있다.

1. 3. 매크로 확장기(Macro Expander)

원시 프로그램에 매크로 정의 부분을 찾아 매크로 정의표(macro define table)에 그 해당 식별자(identifier)와 그 식별자의 행위를 저장한다. 저장된 식별자가 프로그램 문장에서 발견되면 매크로 정의표에 저장된 행위로 대치한다. 식별자의 행위에 또 다른 매크로 확장의 사용도 가능하며, 이 때 대치될 수 있는 형태는 상수, 변수 또는 함수의 형식을 갖는다.

1. 4. 문법 검사기(Syntax Checker)

이 문법 검사기는 헤더화일 확장기와 매크로 확장기에 의해 확장된 원시 프로그램이 C++ 언어에서 정

의된 프로그램 문법에 타당한지의 여부를 검사하는 부분으로서 모든 C'' 언어로 작성된 원시 프로그램은 이 검사기를 통과하여 오류가 없는 프로그램으로 판정되어야만이 다음 과정의 처리를 진행할 수 있게 된다. 이때 발생되는 문법적 오류들은 각각의 오류 종류에 따라 고유의 오류 번호를 가지며, 이 오류 번호를 오류 메시지 처리기에 전달함으로써 해당되는 오류 정보를 출력하도록 하였다.

1.5. C''-to-C 번역기 (Translator)

이 번역기는 헤더화일 확장기와 매크로 확장기를 거쳐 생성된 화일인 *"filename. CI"* 를 입력으로 하여 C'' 언어의 구문 구조를 C 언어의 구문 구조로 변환하는 일을 수행한다. 이때 C''-to-C 번역기는 객체(object)의 개념과 접근 제어(access control) 등 객체지향 개념이 그대로 유지될 수 있도록 변환하였다.

1.6. 오류 정보 처리기 (Error Messages Handler)

문법 검사기에서 오류가 발생되면 해당 오류에 대한 오류번호가 오류 정보 처리기에 보내지고, 오류 정보 처리기는 오류 번호에 해당하는 오류 메시지와 오류가 발생한 원시 프로그램의 줄번호(line number)를 출력한다. 오류가 복구되면 다음 줄부터는 처음의 어휘분석 단계부터 다시 반복된다.

다음 2절의 프리프로세서의 구현에서는 위의 6가지 처리기중 가장 중점적으로 설계, 구현된 매크로 확장기와 C''-to-C 번역기의 구현시 고려된 세부 사항에 대해 논한다.

2. 프리프로세서의 구현

2.1. 매크로 확장기의 구현

C'' 프리프로세서의 구현의 가장 중요한 요소중의 하나인 매크로 확장기의 구현과 관련된 전처리 변환 규칙은 다음과 같다.

2.1.1 전처리 변환 규칙

① *# ifdel* 〈*id*〉

만일 〈id〉가 원시 프로그램에 정의되어 있으면 #elif나 #else, 혹은 #endif문전까지의 문장을 수행하도록 하고, 정의되어 있지 않으면 다음의 전처리 과정을 수행한다.

```
#ifdef DEBUG
printf("Debug ok!!! /n");
#endif
```

DEBUG가 이미 정의되어 있으면 print 문장을 수행한다.

② *# ifndef* 〈*id*〉

만일 〈*id*〉가 원시 프로그램에 정의되어 있지 않으면 *#elif*나 *#else*, 혹은 *#endif*문전까지의 문장을 수행하도록 하고 이후의 처리는 *#ifdef*과 같다.

#if 〈*constant-expression*〉

#else 또는 *#elif* 〈*constant-expression*〉

#endif

③ 〈*constant-expression*〉을 평가한 결과가 참(nonzero)일 때, 그 이하의 문장을 해당 범위(*#else*, *#elif*, 혹은 *#endif*)까지 수행한다. 이 정의는 일반적인 if 조건문의 수행과 동일하다. 또한 *#if*나 *#elif*의 정의에서는 *defined* [() 〈*id*〉 ()] 함수를 사용할 수 있으며, 이 함수는 〈*id*〉가 정의되었는지를 검사하는데 사용되고 *#ifdef* 혹은 *#ifndef*과 같은 효과를 갖는다.

```
#if !defined(NULL)
    #define NULL 0
#else
     printf("NULL is defined/n");
#endif
```

만약 NULL이 정의되어 있지 않으면 NULL을 0으로 정의한다.

④ *#undef* 〈*id*〉

이미 *#define*으로 정의된 〈*id*〉를 매크로 정의표에서 제거한다. 이로 인해 해당 〈*id*〉는 더 이상 확장되지 않는다. 매크로 정의표에 없는 경우에 무시한다.

```
#undef DEBUG
```

DEBUG가 이미 정의되어 있으면 매크로 정의 표에서 DEBUG를 제거한다.

⑤ *#error ⟨message⟩*
*⟨message⟩*의 내용을 오류메시지로 내보낸다.

```
#if  !defined (MODEL)
#error Building model not define
#endif
```

MODEL이 정의되어 있지 않으면 오류메시지를 출력한다.

⑥ *#line ⟨constant⟩ [⟨id⟩]*
다음 원시 문장(source line)의 문번호가 *⟨constant⟩*이고, 현재 입력 화일은 *⟨id⟩*이다. 만약 *⟨id⟩*가 정의되지 않았으면 입력화일은 현재의 화일이다.

```
#line 55 main. c
```

입력화일의 이름은 "main. c"이고 문장 번호는 55 line이다.

⑦ *#define ⟨id₁⟩[(⟨id₂⟩, …)]⟨token-string⟩*
프로그램 명령문에 나타나는 *⟨id₁⟩*에 대해 정의된 *⟨token-string⟩*으로 대치한다. 여기에는 다음과 같은 두 가지 형태가 있다.
ⓐ #define MAX 100
ⓑ #define Fun(x, y) x+y

```
#define MAX 100
#define funl (A, B)  A+B+MAX
#define fun2 (a, b, c)  a+funl (b, c)
main( )
{
int a, p1, p2, p3;
a=fun2(p1, p2, p3) ;
}
```

```
main( )
{
int a, p1, p2, p3;
a=p1+p2+p3+100;
}
```

2. 1. 2 저장과 탐색

이 처리기에서의 매크로 정의표 탐색 과정은 *define* 내의 앞부분의 이름이 동일하고 인수(argument)의 수가 동일할 때 탐색이 성공된 것으로 하였고, 형식 매개변수의 인수들을 관리하여 탐색된 정의의 행위 또는 해당 값으로 대치한다. 탐색이 성공한 정의의 행위가 또다른 정의를 포함할 수 있으므로 새로운 정의가 없을 때까지 계속 탐색을 수행한다. 이의 구현을 위한 주요 함수와 매크로 정의표를 위한 자료 구조는 다음과 같다.

* 주요 함수

defsym. h
어휘 분석기를 위한 토큰의 종류와 번호를 정의하였다.

del. h
프로그램에 사용되는 모든 함수와 기타 전역 함수 등을 정의하였다.

lex. c
어휘 분석기로 yylex( )함수를 호출하면 현재 정의된 토큰을 분리하고 해당 번호를 반환한다.

fun. c
매크로 확장을 위한 보조 함수들을 모은 화일로 탐색과 변환, 그리고 출력 화일에 변환된 내용을 쓰는 등의 실질적인 번역을 수행한다.

cpptoci. c
main 함수로서 토큰의 종류를 보고 해당 함수를 호출하는 역할을 수행한다.

* 주요 저장 장소의 구조(struct type)

```
struct DefTab{          struct ArgList{
  int ArgNum;             char ArgName;
```

```
  char *front;              struct ArgList *next;
  char *end;              }
  struct ArgList *argptr;
  struct DefTab *next;
}
```

여기서 헤더화일 확장기와 매크로 확장기, 또 매크로 확장기내의 ①에서 ⑦번까지 전처리 규칙들은 순차적으로 수행되는 것이 아니라 상호 협력하에 원시 프로그램을 해석하고 변경시키며, 이렇게 확장된 내용을 갖는 화일을 확장자가 *"CI"*가 되는 임시 화일(tempory file)을 생성하여 C 프로그램으로 번역하는 C''-to-C 번역기의 입력 화일로 사용하였다.

2.2 C''-to-C 번역기의 구현

C'' 프리프로세서의 구현의 가장 중요한 요소중의 하나로 본 연구의 주요 부분인 C''-to-C 번역기의 구현과 이에 관련된 변환 규칙은 다음과 같다.

2.2.1. class 선언문의 변환

객체지향 언어인 C'' 언어는 클래스(class) 구조를 채택하고 있다. 이러한 클래스 구조를 표준 C 프로그램으로 변환하기 위해서는 클래스 개념이 구조적 자료형(structual data type)의 개념과 동일하므로 C 언어 구문의 *"struct"* 선언문을 사용하여 변환하였다. 이 변환 방법에서 반적인 변수에 대한 자료형 변환과 클래스 구조내에 선언되어 있는 멤버 함수(member function)의 사용에 대한 허용 여부를 관리하기 위해 클래스 이름, 클래스 변수, 멤버 함수를 유지하는 테이블을 생성한다. 클래스 변환에 대한 세부적인 사항은 다음과 같다.

① *class*는 *struct*로 1대1 변환한다.

```
class name {              struct name {
              ⇨
  ...                       ...
};                        };
```

② 클래스내에서 선언된 모든 변수들은 변수 혹은 함수명 앞에 접두어(prefix)로 클래스의 이름을 붙인다. 이는 C'' 언어의 특징인 객체의 접근 제어 효과를 갖게 되는데, 클래스에서 선언된 변수는 그 접근이 해당 클래스에만 가능하다. 그러나 *struct*에서 선언된 변수는 해당 영역-선언된 위치에 따라 지역(local) 혹은 전역적(global)으로-에서만 접근이 가능하므로 다른 영역에서의 접근을 막기 위해서는 클래스의 내부에 접근이 가능한 곳(*public*인 경우는 일반 함수에서도 가능하지만 *private*나 *protected*인 경우는 클래스의 멤버 함수에서만 가능)에서만 사용할 수 있도록 클래스의 이름을 모든 변수 혹은 함수 앞에 첨가한다.

```
class name {              struct name {
  char * first;    ⇨        char * _name_first;
  char * last;              char * _name_last;
};                        };
```

③ 상위 클래스(super class)에서 선언된 모든 변수들을 하위 클래스(subclass)에 포함시킨다. 이는 클래스 상속(class inheritance)의 문제를 해결한다. 즉, 상위 클래스에서 선언된 변수들은 모두 하위 클래스에 접근 가능해야 하므로 그 접근이 가능하도록 하기 위해서는 변수의 참조 때마다 상위 클래스를 검사하거나 상위 클래스의 모든 변수를 하위 클래스에 포함하는 두 가지 방법이 있다. 전자는 속도가 상대적으로 늦어지고, 후자는 기억 장소를 많이 사용하는 상반된 장단점을 갖는다. 본 연구에서 전자 보다 후자를 선택한 이유는 현재 구현된 기종이 고용량의 기억 장소를 사용할 수 있고, 속도의 향상에 보다 더 역점을 두었기 때문이다.

```
class super {                    struct super {
  int i;                           int _super_i;
};                               }
                    ⇨
class sub:public super {         struct sub {
public :                           int _super_i;
  int j;                           int _sub_i;
};                               };
```

④ *friend*로 선언된 클래스의 모든 변수들을 *friend*를 선언한 클래스에 포함시킨다. 이는 *friend*를 선언한 클래스에서 *friend*로 선언된 클래스의 모든 변수를 참조 가능하도록 하는데, 상위 클래스를 하위 클래스에

포함시키는 것과 동일한 역할을 한다.

⑤ *public, private, protected* 등의 접근 제어는 ②의 변환으로 처리가 가능하므로 무시한다.

```
class access {                 struct access {
public :                         int _access_j;
  int j;              ⇨        };
};
```

⑥ 내포 클래스(nested class)는 분리된 *struct* 자료형으로 변환한다. 내포 클래스는 실제로 두 개의 분리된 클래스이고, 외부 클래스가 내부의 클래스의 모든 변수를 포함한 형태이므로 두 개의 *struct*로 완전 분리가 가능하고 상속받은 것처럼 변수를 복사한다.

```
class out {                    struct inner {
  int j;                         int _inner_i;
  int j;              ⇨        };
  class inner {                struct out {
    int i;                       int _out_i;
  };                             int _out_j;
                                 int _inner_i;
};                             };
```

⑦ 클래스내에서 선언된 멤버 함수는 클래스 이름을 붙여 *struct* 자료형 밖으로 분리한다. C++ 언어에서는 멤버 함수가 생성된 클래스의 객체 자신을 가리키는 포인터인 *this* 변수를 갖는데, C 언어에서는 객체의 개념이 없으므로 인위적으로 모든 멤버 함수의 매개변수에 *this*라는 매개변수를 첨가시킨다. 멤버 함수를 호출할 때 *this*에 해당하는 객체의 주소를 매개변수에 포함하여 호출한다.

```
class member {                 struct member {
  int i;                         int_member_i;
  fcn();                       };
};                             _member_fcn(this)
member :: fcn()                struct member *this;
  i=100;                       {
}                                this->_member_i=100;
  …                            }
main()                ⇨        …
{                              main()
  member obj;                  {
  …                              struct member obj;
  obj.fnc();                     …
  …                              _member_fcn(&obj);
}                                …
                               }
```

⑧ 클래스 생성자(constructor)는 매개변수의 자료형과 반환형(return type) 모두를 함수의 이름에 첨가하여 새로운 함수를 생성하는데, 이는 C++ 언어에서 같은 이름으로 여러가지 행위를 갖는 함수의 정의가 가능하기 때문이다. 따라서, 동일한 명칭으로 사용 가능하므로 그대로 C 언어로 변환하면 중복 정의 오류(redeclaration error)가 발생한다. 그러므로 함수의 이름을 매개변수의 자료형과 반환형으로 구분하여 중복 정의 오류의 발생을 막는다. 생성자에서 정의되지 않은 반환형은 해당 클래스의 객체를 가리키는 포인터형으로 재정의한다. 이는 생성자가 해당 객체를 생성하고 이를 가리키는 this 포인터를 생성하는데, 이 this 포인터를 함수에서 계속 유지해야 하기 때문이다.

```
class member {                 struct member {
  int i;                         int_inner_i;
  member(int a);               };
member :: member(int a)        struct member*
                               _i_member(this, a):
  i+=a;               ⇨        struct member *this;
}                              int a;
                               {
                                 this->_inner_i+=a;
                                 return(this);
                               }
```

⑨ 클래스 소멸자(destructor)는 클래스 이름에 "*dtor*"을 첨가시켜 함수의 이름을 새롭게 정의한다. 이는 C 언어의 함수 이름에 "~"을 사용하지 못하기 때문에 대신 "*dtor*"을 사용한다.

```
class dest {                   struct dest {
  char s;                        char*_dest_s;
```

```
public :                          };
  dest (int sz) {                 _dtor_dest (this)
  ~det () {delete s;}    ⇨        struct dest*this;
                                  {
                                  free (this->_dest_s;
};                                }
```

2.2.2 특정 함수의 변환

① *cout* 함수는 표준 출력함수인 *printf* ( ) 함수로 변환한다. 이때 C++ 언어에서는 출력하고자 하는 변수의 자료형의 선언없이 사용 가능하지만 C 언어의 *printf* ( ) 함수는 출력하고자 하는 변수의 출력 명세(format)가 필요하므로 여기에서는 변수가 선언되면 선언된 모든 변수의 형을 저장하고 변수의 형이 필요할 때 저장된 내용을 참조하도록 하였다. 구현은 하나의 *cout* 함수를 여러 개의 *printf* ( ) 함수로 분리하여 처리하였다.

```
int i;                    int i;
cout<<"int i="<<i;   ⇨    printf ("int i=");
                          printf ("%d", i);
```

② *cin* 함수는 *scanf* ( ) 함수로 변환하며, *cout* 함수와 마찬가지로 값을 저장하려는 변수의 형을 저장하였다가 해당 변수의 입력 명세에 맞게 변환한다. *cout* 함수와 동일하게 여러 개의 분리된 *scanf* ( ) 함수 형태를 갖는다.

```
int i;                    int i;
cout>>i;         ⇨        scanf ("%d", &i);
```

③ *new*는 *calloc* ( ) 함수로 변환된다. 여기서, *malloc* ( ) 함수를 사용하지 않고 *calloc* ( ) 함수를 사용한 이유는 할당하고자 하는 크기가 하나 이상일 경우가 많고, 또 그 크기가 하나일 때도 *calloc* ( ) 함수로 가능하기 때문이다.

```
int *ip=new int[10]; ⇨ int *ip;
                       ip= (int*) calloc (10, sizeof (int));
```

④ *delete*는 *free*( ) 함수로 변환한다. 변환 예는 2.2.1의 소멸자의 변환 예에서 이미 기술하였다.

⑤ *enum* 자료형은 변환 자료형으로 정수형을 취하고, *enum*의 각 변수는 *const* 선언으로 변환하였다. 이는 구현시 enum 자료형 모두를 저장하는 불합리를 제거할 수 있도록 하기 위해 *enum* 내에 선언된 변수를 상수 정의로 대치하고, 변수의 자료형은 int형으로 대치하여 저장하였다.

```
enum day {SUN, MON, TUE} week;      cost SUN=0,
                               ⇨    const MON=1;
                                    const TUE=2;
                                    typedef int day;
                                    day week;
```

⑥ *inline*은 단지 하나의 독립된 함수로 변환하는데, 이는 이미 매크로 확장이 끝난 상태의 입력 화일을 사용하기 때문에 확장이 불가능하여 속도가 감소되는 것을 감수하고 하나의 독립된 함수로 변환하였다.

```
class x {                           struct x {
  int val;                            int _x_val;
  int fcn();                  ⇨     };
};                                    _x_fcn (this)
inine x :: fcn() {return val;}        struct x *this;
                                    {return this->_x_val;}
```

⑦ *overload* 선언은 반환형과 매개변수 자료형 모두를 함수의 이름에 첨가하는데, 이는 2.2.1의 생성자 변환과 동일하게 하나의 이름으로 여러가지 기능을 갖는 함수의 정의가 가능한 C++ 언어의 특성을 C 언어로 변환하기 위한 방법이다. 함수 호출시에는 호출하는 실매개변수의 형을 조사하여 실매개변수의 형에 맞는 함수를 호출하도록 하였다.

```
overload print;                 void _v_i_print ();
  void print (int);       ⇨     int _i_c_print ();
  int print (char);             int _i_cp_l_print ();
  print (char*, long);
```

⑧ *operator* 선언은 ⑦의 *overload*의 변환과 마찬가지로 함수의 이름에 반환형과 매개변수 자료형을 추가하는데, *operator*는 함수 형태가 아니기 때문에 임의

로 *-op-*와 *operator*의 발생 순서를 함수 이름에 첨가하고, *operator*는 operator표에 저장한다. 해당 *operator*를 발견하면 operator 표에서 *operator*의 발생 순서를 찾고 매개변수와 반환형을 포함한 함수 이름으로 해당 *operator*의 동작을 수행하는 함수를 호출한다.

⑨ default 인수가 있는 함수는 매개변수의 자료형을 매개변수의 갯수와 포인터로 변환하여 처리하였다.

```
void def(int i=0, int j=1, char*s="")      void def(_argc_, _argv_)
{                                          int _argc_;
  ...                                      Parmeter *_argv_;{
}                                          int i=0;
                                       ⇨   int j=1;
                                           char *s="";
                                           Parameter *_frmp_;
                                           int _index_;
                                          for(_index_=0;_index_<
                                               _argc_; _index_i++){
                                           _frmp_->name=_argv_->init;
                                           _frmp_=_frmp_->next;
                                           _argv_=_argv_->next;
                                           }
                                             ...
                                           }
```

2.2.3 **기타 변환 규칙**

① 함수의 반환형을 미리 지정하는 경우는 매개변수를 모두 제거하였다.

② 함수의 매개변수의 표현은 아래와 같이 변환시킨다.

```
void func(int i, char *str)         void func(i, str)
{                               ⇨   int i;
  ...                               char *str;
}                                   {
                                      ...
                                    }
```

③ 정의되지 않은 함수의 반환형은 int형으로 재정의한다.

```
func(int, char*);          ⇨      int func();
```

④ 모든 변수에는 영역(scope)에 따라 번호를 붙여 지역 변수들의 영역을 구별한다. C++ 언어에서는 함수 중간에 변수의 선언이 가능한데, 이의 처리를 위해 변수에 영역에 해당하는 번호를 붙여 식별하였다.

```
main()                           main()
{                                {
int j;                       ⇨   int _1_j;
  ...                            int _2_i;
for(int i=0; i<10;i++){            ...
  ...                            for(_2_i=0;_2_<10;_2_i++){
}                                  ...
                                 }
                                 }
```

결과 및 고찰

연결 리스트(linked list)를 구성하는 프로그램을 C++ 언어로 작성한 Fig. 2를 본 연구에서 구현한 변환 규칙을 적용하여 Fig. 3과 같이 생성되었다. 여기서 Fig. 2를 MIPS의 C++ 컴파일러로 번역하여 실행한 결과와 Fig. 3의 변환 결과를 MIPS-C 컴파일러와 Turbo-C 컴파일러로 번역하여 실행시킨 결과가 모두 동일하게 출력되었다. 또한, 기존의 컴파일러에 의존하는 실행 시간 라이브러리와 관계없이 실행된다는 것도 검증되었다.

```
class dlink {
  char* name;
  dlink *suc;
public :
  dlink();
  ~dlink();
  void init(dlint*);
  void append(dlink*, char*);
};
```

```
void dink : : init (dlink* p)
{
  p->suc=0;
}
void dlink : : append (dlink* p, char* str)
{
  p->suc=suc;
  strcpy (p->name, str) ;
  suc=p;
}
main ( )
{
  char *ptr1;
  dlink *list;
  list->init (list) ;
  while (1) {
    cout<<"Append Name : ";
    cint>>ptr1;
    dlink *aa=new dlink;
    a->append (aa, ptr1) ;
  }
}
```

Fig. 2. A Example of C++ Program.

```
struct dlink {
  char *_dlink_name;
  struct dlink *_dlink_suc;
}
struct dlink *_i_dlink ( ) ;
int _dtor_dlink ( ) ;
void _dlink_init ( ) ;
void _dlink_append ( ) ;

void _dlink_init (this, p)
struct dlink *this;
struct dlink *p;
{
p->suc=0;
}

void _dlink_append (this, p, str)
struct dlink *this;
struct dlink *p;
char *str;
{
p->suc=this->_dlink_suc;
strcpy (p->name, str) ;
this->_dink_suc=p;
}

int main ( )
{
char* _1_ptr1;
struct dlink *_1_list;
struct dlink *_2_aa;

_i_dlink (_1_list) ;
list->_dlink_init (_1_list, _1_list) ;
while (1) {
printf ("Append name : ") ;
scanf ("%s", _1_ptr1) ;
_i_dlink (_2_aa) ;
_2_aa= (struct dlink*) calloc
        (1, sizeof (struct dlink) ) ;
aa->_dlink_append (_2_aa, _1_ptr1) ;
}
}
```

Fig. 3. The Translated Programming Result of Example Fig. 2.

적　　요

C++는 1983년 AT&T Bell 연구소의 Stroustrup (1991)이 기존의 C 언어를 근간으로 하고, Smalltallk, Ada 등의 객체지향 언어들을 참조하여 만들어졌다. C++ 언어는 기존의 C 언어와 유사한 구문 구조를 가지고 있으며, 객체지향 개념을 쉽게 접할 수 있기 때문에 C 언어를 즐겨 사용하던 사용자들에 의해 그 사용이 점차 확산되는 추세에 있다. C++ 언어의 사용 증가에 따라 C++ 번역기의 요구도 따라서 증가되었다.

기존에 구현된 C++ 번역기에는 여러 가지 형태가

존재하지만 특징적으로 크게 2가지로 구분되어진다. 첫째로 직접 목적 코드를 생성하는 Borland C++ 컴파일러가 있고, 둘째로는 전처리 과정을 거치는 프리프로세서 방식인 GNU C++ 프리프로세서와 GuideLine C++ 프리프로세서 등이 있다. 전자인 Borland C++ 번역기는 전처리 과정을 거치지 않고 직접 목적 코드를 생성하기 때문에 생성된 목적코드가 C 언어 프로그램 화일이 아니므로 사용자가 판독하는 것은 거의 불가능하며, 후자인 GNU C++ 프리프로세서와 GuideLine C++ 프리프로세서는 본 연구와 마찬가지로 전처리 변환 과정을 거친 후, 목적 코드를 생성하지만 전처리 변환을 거친 후에는 별도의 실행시간 라이브러리를 이용하여 자체 함수로 사상(mapping)시키기 때문에 이러한 방법도 사용자가 판독하기에는 많은 어려움을 갖고 있다.

따라서, 본 연구에서는 실행시간 라이브러리와 이를 이용하는 C 컴파일러에 독립적인 C++ 프리프로세서를 구현하였다. 이 C++ 프리프로세서는 C++ 언어의 모든 기능을 C 프로그램 형식으로 번역되어지므로 그 번역된 결과는 기존의 C 언어를 사용하던 사용자가 판독 가능하기 때문에 C++ 언어를 학습하는 사용자에게 C++ 언어와 C 언어를 비교하는데 커다란 도움을 줄 것이다.

참 고 문 헌

Aho, V. and Sethi, Ravi and D., Ullman, 1986. Compilers Princeples, Techniques, and Tools, Addison Wesley.

AT&T, 1989. UNIX System V AT&T C++ Language System Release 2.0-Release Notes, AT&T.

AT&T, 1989. UNIX System V AT&T C++ Language System Release 2.0-C++ Reference Manual, AT&T.

AT&T, 1989. UNIX System V AT&T C++ Language System Release 2.0-Library Manual, AT&T.

AT&T, 1989. UNIX System V AT&T C++ Language System Release 2.0-Selected Readings, AT&T.

Booch, G., 1986. Object-Oriented Development, IEEE Trans. on S/E, SE-12(2) : 211-221, Feb.

Booch, G., 1991. Object-Oriented Design with Applications, Benjamin-Cummings.

Ege, R.K., 1989. An Object-Oriented Design Environment, TOOLS '89 Proc., 49-58.

Faison, T., 1992. Borland C++ : Object-Oriented Programming, SAMS.

Goldberg, A. and D., Robinson 1983. Smaltalk-80 : The Language and its Implementation, Addison-Wesley.

ISO, 1993. C++ Language ISO Standardization.

Lai, M., 1989. Hyper Hood++ : An Object-Oriented Design Tool for Developments in Object-Oriented Programming Language, TOOLS '89 Proc., 295-308.

Pressman, R.S., 1987. Software Engineering : A Practitioner's Approach, 2nd ed., McGraw-Hill.

Stroustrup, B., 1991. The C++ Programming Language, 2nd ed., Addison-Wesley.

Wiener, R.S. and L.J., Pinson 1988. An Introduction to Object-Oriented Programming and C++, Addison-Wesley.

원유헌, 1991. 프로그래밍 언어론, 정익사.