A Comparison of the Syntax of C/C++ and Pascal

*** VERSION 1.6 ***

Russell C. Bjork

Professor of Computer Science

Gordon College, Wenham, MA

PRELIMINARY NOTE:

The C programming language was originally developed by Dennis Ritchie in 1972. Two major dialects of C have been available during the history of the language: "traditional C" (the original dialect which was distributed for many years with all Unix systems) and ANSI C (an improved dialect standardized by ANSI.) This document will describe features of both dialects.

An ANSI C compiler will accept programs written in traditional C (but not vice versa). There still are a few C compilers that accept only traditional C, and older books often use this dialect for example programs, so it is important to be aware of it. However, since the compiler can catch certain common programming errors in an ANSI C program that would not be caught in a tradtional C program, a person newly learning C is well-advised to learn the ANSI dialect.

C++ is a newer language built on the base of C by Bjarne Stroustrup in the early 1980's. It includes several improvements to C, plus many new features designed to support object-oriented programming, while still retaining the basic features of ANSI C. As a result, a properly-written ANSI C program will be accepted by a C++ compiler. (However, many traditional C programs will not be accepted. Given the growing popularity of C++, this is another good reason for learning ANSI C rather than traditional C.)

C++ and its supporting libraries are currently undergoing standardization by ANSI/ISO, with publication of a final standard expected in late 1998. Currently available C++ compilers and libraries do not yet support
all of the features of this standard.

For the most part, this handout will only describe features that are common to both ANSI C and C++. A few C++ features that make programming easier
or make for better programming style are also included - these are labelled C++ only and must not be used in programs that are to be compiled by a C compiler. (In each case, there is an alternate C construct that serves the same
purpose.) Finally, a few C++ features are described as new ANSI standard C++.
These may or may not be available yet in a particular C++ implementation.

Pascal	**C/C++**
Lexical Conventions	
Not case-sensitive; upper and lower case letters are equivalent - so	Case-sensitive; upper and lower case letters are different - so
somename, SOMENAME, Somename, SomeName etc. are all the same.	somename, SOMENAME, Somename, SomeName are all different
Comments	
(* This is a comment *)	/* This is a comment */
	C++ only
	// This is a also comment - in C++ only
	// A comment specified this way extends
	// only to the end of the current line
Constants	

The rules for numeric constants (integer, real) are essentially the same, though C is a bit more flexible. (See a C reference manual for details.)

'c'

'This is a string'

is

same

'c'

"This is a string"

NOTES:

1. Non-printing characters may be embedded in either character or string constants by using various escapes - e.g.

   \0 - null character
   \n - newline
   \t - tab
   \b - backspace
   \f - formfeed
   \\ - \
   \' - '
   \" - "
   \ddd - Character whose OCTAL code ddd - e.g. '\101' is the same as 'A'.

2. String constants are terminated by a null character \0. Thus, the total space allocated for a string is one more than the number of characters in it.

3. String constants may be continued over more than one line by having a \ be the very last character on the line to be continued - e.g.

   "This is a string constant that tends over more than one line."

ex\

C++ and some ANSI C compilers only

4. A string constant may be continued over more than one line by simply closing the quotes on one line and re-opening them on the next - e.g.

```
   "This is a string constant that
ex"
   "tends over more than one line."
```

```
const
    size = 100;
    pi = 3.14159;
    first = 'A';
    filename = 'XYZ.DAT';
```

```
#define size 100
#define pi 3.14159
#define first 'A'
#define filename "XYZ.DAT"
```

ANSI C and C++ only

```
const int size = 100;
const float pi = 3.14159;
const char first = 'A';
const char filename[] = "XYZ.DAT";
```

NOTE: In C (but not C++), an integer

constant declared this way may not be used to specify the size of an array in most cases.

Declarations of variables

```
var
    i: integer;
    r: real;
    b: boolean;
    c: char;
    j, k, l: integer;
```

```
int i;
float r;
int b;
char c;
int j, k, l;
```

New ANSI Standard C++

```
bool b;
```

NOTES:

1. C does not use a word like var to introduce variable declarations - it just declares them.

2. C does not have a separate type for boolean values. Int is used, with 0 = false and 1 = true. The new ANSI C++ standard calls for a built in type bool with values false and true, but not all C++ compilers implement it yet.

3. In addition to the scalar types shown above, C has:

   short int (may be abbreviated short)

   long int (may be abbreviated long)

   double (double-precision real)

4. Any integer type (including char) may be prefixed by unsigned - e.g.

   unsigned int i;

   unsigned char c;

```
var                                     int a[10];

    a: array[0..9] of integer;          float b[5][10];

    b: array[0..4, 0..9] of real;

                                        NOTE: Arrays in C always have
subscripts

                                        ranging from 0 to declared size - 1.


    c: array[1..10] of integer;         NO DIRECT EQUIVALENT IN C.  WORK
AROUND

                                        THIS BY DECLARING:


                                        int c[10];


                                        AND LETTING SUBSCRIPTS RANGE FROM
0..9.

                                        NOTE: C does not distinguish between

                                        packed and unpacked arrays
```

```
var                                     struct

    student: record                       { int id;

        id: integer;                        char name[11];

        name: packed array[1..10]           float gpa;

                of char;                  } student;

        gpa: real
```

```
    end;
```

New ANSI Standard C++

In this and subsequent similar examples, the declaration of char name[11] can be replaced by:

```
    string name;
```

The standard header <string> must be #included to allow this.

For this example, suppose the types employee and student have been previously declared:

```
var
    borrower: record
        case boolean of
          false: (EBorr: employee);
          true:  (SBorr: student)
    end;
```

```
union
  { employee EBorr;
    student SBorr;
  } borrower;
```

NOTE:

There is no provision for an explicit tag field in C unions, analogous to the tag field of a Pascal variant record.

```
var
    borrower: record
        case IsStudent: boolean of
          false: (EBorr: employee);
          true:  (SBorr: student)
    end;
```

NO DIRECT EQUIVALENT IN C. THIS CAN BE HANDLED BY CREATING A STRUCT WHICH CONTAINS THE TAG AND A UNION AS ITS FIELDS. (SEE BELOW)

```
var
    borrower: record
        id: integer;
        name: packed array[1..10]
                of char;
        case boolean of
          false: (EBorr: employee);
          true:  (SBorr: student)
    end;
```

```
struct
  {
    int id;
    char name[11];
    union
      { employee EBorr;
        student SBorr;
      } Borr;
  } borrower;
```

We can access the variants as

We can access the variants as

borrower.EBorr, borrower.SBorr borrower.Borr.SBorr	borrower.Borr.EBorr,
	C++ only:
	The field name Borr for the union can be
	omitted; the variants can then be
	referenced as in Pascal.

---

var flavor: (chocolate, vanilla);	enum {chocolate, vanilla} flavor;
	NOTE: As in Pascal, it is usually
	better style with record and enumeration
	types (and often with array types) to
	first declare a named type and then a
	variable of that type. See below.

---

var	
squares: set of 1..9;	NO DIRECT EQUIVALENT IN C. SOMEWHAT OF
	THE SAME EFFECT CAN SOMETIMES BE
	BE OBTAINED BY

vector

to

used

operations,

```
    squares := [];
    ...
    squares := squares + [2]
```

efficiently,

```
const
    vowels = ['A','a','E','e',
              'I','i','O','o','U','u'];
...

    if c in vowels then
```

1. Treating a longword as a bit representing of basetype size up to 32. Bitwise operations can be used to get the effect of set operations, e.g.

```
int squares;
...
squares = 0;
...
squares |= 1 << 2;
```

2. Some C library routines allow a character string to be used like a set of char - but not as efficiently, e.g.

```
#define VOWELS "AaEeIiOoUu"
...

if (strchr(VOWELS, c))
```

...

...

3. Some C/C++ implementations offer library routines to implement bitsets that somewhat resemble Pascal sets.

NO PASCAL EQUIVALENT

The type specifier for a variable may optionally be preceeded by one of the following storage class specifiers:

auto, extern, register, static

e.g.

```
auto int i;
extern int r;
register int i;
static int i;
```

The type specifier for a function may be optionally preceeded by one of the following:

static, extern, **(C++ only)** inline

(Extern is the default for functions
and top-level variables; auto is the
default for local variables
declared in functions.)

NO STANDARD PASCAL EQUIVALENT (BUT MANY DIALECTS ALLOW THIS)

A C variable may be initialized when it is declared - e.g.

```
int i = 3;
char c = 'A';

float a[3] = { 1.0, 2.0, 3.0 };

struct
  {
    char name[11];
    float gpa;
  } student = { "AARDVARK", 4.0 };
```

NOTE:

In C++, initialization of a struct

this way is not allowed if any
field is of class type, requiring
a constructor. Thus, in particular,
this example would not work correctly
if name were declared as string,
since string is a library class.

Declarations of new types

----------- -- --- -----

```
type                                    typedef float realarray[10];
    realarray = array[0..9] of real;
                                        realarray a;
var
    a: realarray;
```

```
type                                    typedef struct
    student = record                      { int id;
        id: integer;                        char name[11];
        name: packed array[1..10]           float gpa;
                of char;                  } student;
```

```
        gpa: real
    end;

var
    someone: student;
```

```
student someone;
```

OR

```
struct student
  { int id;
    char name[11];
    float gpa;
  };

struct student someone;
```

OR

```
struct student
  { int id;
    char name[11];
    float gpa;
  } someone;
```

NOTES:

1. In C, the difference between these examples is that the first creates a

new type named simply 'student', while the second two create a new type whose name is 'struct student'.

The third example combines the type and variable declarations into one declaration. (Compare this with the similar example under "Declarations of Variables" above, where a named type was not created.)

2. In C++, all three examples create a new type that can be called either student or struct student.

For this example, suppose the types employee and student have been previously declared:

```
type                                    typedef union
    borrower = record                     { employee EBorr;
        case boolean of                     student SBorr;
          false: (EBorr: employee);       } borrower;
          true:  (SBorr: student)
```

```
    end;

var

    someone: borrower;
```

```
borrower someone;
```

OR

```
union borrower
  { employee EBorr;
    student SBorr;
  };

union borrower someone;
```

OR

```
union borrower
  { employee EBorr;
    student SBorr;
  } someone;
```

NOTE: See discussion of struct declarations above.

```
type

    flavortype = (chocolate, vanilla);
```

```
typedef

    enum {chocolate, vanilla}
flavortype;
```

```
var
    flavor: flavortype;
```

```
flavortype flavor;
```

OR

```
enum flavortype {chocolate, vanilla};

enum flavortype flavor;
```

OR

```
enum flavortype {chocolate, vanilla}
    flavor;
```

NOTE: See discussion of struct declarations above.

Pointers

```
var
   p: ^integer;
   first: ^student;
```

```
int *p;
student *first;
```

OR

```
                                        struct student *first;

                                        (In C, which form is used depends on
                                         whether the type student was
declared
                                         using typedef or not)
```

```
type                                    typedef
    stuptr = ^node;                         student *stuptr;
var
    first: stuptr;                      stuptr first;
```

OR

```
                                        typedef
                                            struct student *stuptr;

                                        stuptr first;

                                        (Depending, in C, on how student was
                                         declared)
```

```
type
    nodeptr = ^node;
    node = record
        info: integer;
        link: nodeptr
    end;
var
    first: nodeptr;
```

```
typedef
    struct node
      {
        int info;
        struct node *link;
      } node, *nodeptr;

nodeptr first;
```

OR

```
typedef
    struct node * nodeptr;
typedef
    struct node
      {
        int info;
        nodeptr link;
      } node;

nodeptr first;
```

(Given the above declarations)

p^ := p^ + 1;	*p = *p + 1;
new(first);	first = (nodeptr) malloc(sizeof(node));
...	...
first := first^.link;	first = first -> link;
...	...
dispose(first);	free(first);
	C++ only:
	first = new node;
	...
	delete first;

NO PASCAL EQUIVALENT	In C, the type array is equivalent to a pointer type to the element type of the array. Thus, the following pairs of operations are equivalent; either syntax can be used regardless of whether a is declared as an array or a pointer:

a[0] = 1; IS EQUIVALENT TO *a = 1;

a[3] = 1; IS EQUIVALENT TO *(a+3)= 1;

Note that arithmetic on pointers is done in units of the size of the basetype. Thus, if ints occupy 4 bytes, then *(a+3) actually adds 12 to to calculate the desired address.

Also, for formal parameters only,

int a[]; IS EQUIVALENT TO int *a;

As a consequence of this, the following code segments are NOT equivalent

```
var
    a, b: array[0..9] of real;          float a[10], b[10];
    ...                                 ...
    b := a;  <- NOT AT ALL THE SAME ->  b = a;
```

(In fact, this would not even be syntactically legal unless b were a formal parameter of a procedure.)

C allows assignment of entire structures, but NOT arrays. The equivalent C code for the Pascal array assignment is

```
for (i=0; i < 10; i++)
    b[i] = a[i];
```

Functions and Procedures

```
function f(x, y: integer;
           z: real): real;

    var
        q: integer;

    begin
        q := sqr(x) + y;
        f := q - z
    end;
```

TRADITIONAL C:

```
float f(x, y, z)
    int x, y;
    float z;
  {
    int q;

    q = x*x + y;
    return q - z;
  }
```

ANSI C/C++:

```
float f(int x, int y, float z)
  {
    int q;

    q = x*x + y;
    return q - z;
  }
```

NOTE WELL THE DIFFERENCES IN USAGE OF SEMICOLONS BETWEEN PASCAL AND C

```
procedure p(x: integer);
    var
        temp: integer;
    begin
        ...
    end;
```

TRADITIONAL C:

```
void p(x)
    int x;
  {
    int temp;
    ...
    return;
  }
```

ANSI C/C++:

```
void p(int x)
  {
    int temp;
    ...
    return;
  }
```

NOTE: In either dialect, the return statement is optional at the end of the code. A return statement or statements may also appear in the middle of the code.

Given:

```
function f: integer;
    ...
procedure p;
    ...
```

These routines are called by:

Given:

```
int f()
    ...
void p()
    ...
```

These routines are called by:

```
    x := f;
    p;
```

```
    x = f();
    p();
```

```
procedure p(var x: integer);
    begin
        x := 17
    end;
```

TRADITIONAL C:

```
void p(x)
    int *x;
  {
    *x = 17;
  }
```

ANSI C/C++:

```
void p(int *x)
  {
    *x = 17;
  }
```

This routine is called by:

```
    p(i);
```

In either dialect, this routine is called by:

```
    p(&i);
```

(where i is an integer variable)

(where i is an int variable)

C++ only:

```
void p(int &x)
  {
    x = 17;
  }
```

Declared this way, this routine is called by:

```
    p(i);
```

---

type

```
    realarray = array[0..9] of real;
```

TRADITIONAL C

```
procedure p(var a: realarray);
    begin
        a[1] := a[2] + a[3]
    end;
```

```
void p(a)
    float a[];
  {
    a[1] = a[2] + a[3];
  }
```

OR, BECAUSE OF THE EQUIVALENCE OF ARRAYS AND POINTERS:

```
void p(a)
    float *a;
  {
    a[1] = a[2] + a[3];
  }
```

ANSI C:

```
void p(float a[])
  {
    a[1] = a[2] + a[3];
  }
```

OR

```
void p(float *a)
  {
    a[1] = a[2] + a[3];
  }
```

```
function f(c: char): integer; forward;
```

TRADITIONAL C:

```
int f();
```

ANSI C/C++:

```
int f(char c);
```

OR

```
int f(char);
```

NOTE: Strictly speaking, no C dialect absolutely requires you to declare a function before it is used. If a call to a previously undeclared function is seen, the compiler assumes it is a function returning int and, in the case of ANSI C/C++, makes assumptions about the types of its formal parameters based on the types of the actual parameters appearing in the call. This serves as an implicit function declaration.

However, it is always good practice to declare a function before it is used - and this is mandatory if the return type is other than int. Further, the C++ compiler (and some ANSI C compilers) issue a warning about 'implicit declaration of function' if you call a previously undeclared function.

---

function f(c: char): integer; external;

TRADITIONAL C:

```
int f();
```

ANSI C:

```
int f(char c);
```

OR

```
int f(char);
```

```
program ....

    ...

begin (* main program *)

    ...

end.
```

```
main()
  {
    ...
  }
```

OR (TRADITIONAL C)

```
main(argc, argv)
    int argc;
    char *argv[];
  {
    ...
  }
```

OR (ANSI C)

```
main(int argc, char * argv[])
  {
    ...
  }
```

Operators and Expressions

--- --- ---

C operators are grouped in decreasing order of precedence. All operators in the same group have the same precedence. L and R indicate left and right associativity, respectively. The names used in examples stand for variables or expressions of a certain type, as follows:

x, y - no particular type

i, j - integers

b, c - boolean

p - pointer

a - array or pointer

s - struct

f - function

t - type name

s.name	s.name
p^.name	p -> name
a[i]	a[i]
f or f(arguments ...)	f() or f(arguments ...)
NO PASCAL EQUIVALENT (post-increment)	x++
NO PASCAL EQUIVALENT (post-decrement)	x--
NO PASCAL EQUIVALENT (pre-increment)	++x
NO PASCAL EQUIVALENT (pre-decrement)	--x

p^	*p	
NO PASCAL EQUIVALENT (address of)	&x	
+x	(ANSI C/C++ only) +x	
-x	-x	
not x	!x	
NO PASCAL EQUIVALENT (bitwise not)	~i	
NO PASCAL EQUIVALENT (type cast)	(t) x	
NO PASCAL EQUIVALENT (size in bytes)	sizeof x	
NO PASCAL EQUIVALENT (size in bytes)	sizeof (t)	
x * y	x * y	L
x / y	x / y	L
i div j	i / j	L
i mod j	i % j	L
x + y	x + y	L
x - y	x - y	L
NO PASCAL EQUIVALENT (left shift)	i << j	L
NO PASCAL EQUIVALENT (right shift)	i >> j	L
x < y	x < y	L
x > y	x > y	L
x <= y	x <= y	L

x >= y	x >= y	L
x = y	x == y	L
x <> y	x != y	L
NO PASCAL EQUIVALENT (bitwise and)	i & j	L
NO PASCAL EQUIVALENT (bitwise xor)	i ^ j	L
NO PASCAL EQUIVALENT (bitwise or)	i | j	L
b and c	b && c	L
b or c	b || c	L
NO PASCAL EQUIVALENT (conditional expr)	b ? x : y	R
x := y	x = y	R
x := x + y	x += y	R
x := x - y	x -= y	R
x := x * y	x *= y	R
i := i div j	i /= j	R

x := x / y	x /= y	R
i := i mod j	i %= j	R
NO PASCAL EQUIVALENT (see above)	i <<= j	R
NO PASCAL EQUIVALENT " "	i >>= j	R
NO PASCAL EQUIVALENT " "	i &= j	R
NO PASCAL EQUIVALENT " "	i ^= j	R
NO PASCAL EQUIVALENT " "	i |= j	R
NO PASCAL EQUIVALENT (sequential eval)	x, y	L

The following operators are found only in C++

NO GENERAL PASCAL EQUIVALENT (type conversion)	t(x)	
new(p)	p = new t	
dispose(p)	delete p	

Executable Statements

```
x := y + z
```

than

```
x = y + z;
```

NOTE: This is an instance of the C expression statement, which is actually much more flexible its Pascal equivalent. The following is also legal (but contorted!)

```
if (w += x = y++ * z)
    u = ++w;
```

This is equivalent to:

```
x = y * z;
y = y + 1;
w = w + x;
if (w != 0)
  {
    w = w + 1;
    u = w;
  }
```

```
begin
    x := y + z;
    w := x
end
```

```
{
  x = y + z;
  w = x;
}
```

NOTES:

1. There is never a semicolon after the terminating } , but the last statement inside the compound statement does end with a semicolon.

2. The C compound statement may begin with declarations for local variables - e.g.

NO PASCAL EQUIVALENT

```
{
  int temp;

  temp = y;
  y = z;
  z = temp;
}
```

```
if x < 0 then
    x := -x
```

```
if (x < 0)
    x = -x;
```

```
if x > y then
    max := x
else
    max := y
```

```
if (x > y)
    max = x;
else
    max = y;
```

NOTE: In C, a semicolon is a statement terminator, not a statement separator - so it MUST be used before else in a case like this.

```
while x < y do
    x := 2 * x
```

```
while (x < y)
    x = 2 * x;
```

```
repeat
    x := 2 * x;
    y := y - 1
until x >= y
```

```
do
  {
    x = 2 * x;
    y--;
  }
while (x < y);
```

NOTE: The sense of the condition is the opposite from Pascal. C is consistent about always requiring compound statements to be surrounded by braces.

```
for i := 1 to n do
    x[i] := 0
```

```
for (i = 1 ; i <= n ; i++)
    x[i] = 0;
```

NOTE: The C for is considerably more flexible than the Pascal for.

NO PASCAL EQUIVALENT USING FOR

```
for (i = 1; i <= n; i += 10)
    x[i] = 0;
```

```
                                        (Zeroes x[1], x[11], x[21] .. )

NO PASCAL EQUIVALENT USING FOR          for (i = 1; i <= n && x[i] !=

0; i++);


                                        (Finds first element of x[] that is
0;
                                         stops if all n elements are examined
                                         without finding one.)

-------------------------------------------------------------------------
---


case i of                               switch (i)
                                          {
    1: write('one');                        case 1:  printf("one");
    2: write('two');                                 break;
    3: write('three');                      case 2:  printf("two");
    4: begin                                         break;
         write('four');                     case 3:  printf("three");
         i := 3                                      break;
       end                                  case 4:  printf("four");
                                                     i = 3;
otherwise                                            break;
    write('Bad value')                      default: printf("Bad value");
                                          }
end;
```

```
                                        NOTE: If the breaks are omitted,
control
                                              flows through case to case.
Thus,
                                              the following are equivalent:


case i of                               switch(i)
                                          {
    0: write('ZeroOne');                    case 0: printf("Zero");
    1: write('One')                         case 1: printf("One");
                                          }
end;
```

```
if, while, repeat, for, or case         if, while, do, for, or switch
  begin                                   {
    ...                                     ...
    ...                                     ...
    goto 1                                  break;
    ...                                     ...
  end;                                    }
1:


while, repeat, or for                   while, do, or for
  begin                                   {
```

```
    ...
    ...
    goto 1;
    ...
1:
  end
```

```
    ...
    ...
    continue;
    ...
  }
```

```
procedure p ...

    ...
    goto 1;
    ...
1:
   end
```

```
void p( ...

    ...
    return;
    ...
```

```
function f ...

    ...
    f := somevalue;
    goto 1;
    ...
1:
```

```
int f( ...

    ...
    return (somevalue);
    ...
```

```
   end
```

```
goto 1;                                 goto fini;

...                                     ...

1: ...                                  fini: ...
```

```
;  -- the null statement                ;   -- the null statement
```

Input - Output

----- - ------

```
                                        #include <stdio.h>

var

    c: char;                            char c;

    i: integer;                         int i;

    r: real;                            float r;

    s: packed array[1..10] of char;     char s[10];

...                                     ...
```

read(c);	scanf("%c", & c);
read(i);	scanf("%d", & i);
read(r);	scanf("%f", & r);
NO STANDARD PASCAL EQUIVALENT	scanf("%s", s);
readln;	while (getchar() != '\n') ;
readln(c, i, r);	scanf("%c%d%f", & c, & i, &
r);	
	while (getchar() != '\n') ;
write(c);	printf("%c", c);
write(i);	printf("%d", i);
write(r);	printf("%f", r);
write(s);	printf("%s", s);
writeln;	printf("\n");
writeln('c=',c,' i=',i,' r=', r, s);	printf("c=%c i=%d r=%f%s\n",c, i, r, s);
	C++ only:
	#include <iostream.h>
	cin >> c;
	cin >> i;

```
                                        cin >> r;

                                        cin >> s;

                                        while (cin.get() != '\n') ;


                                        cin >> c, i, r;

                                        while (cin.get() != '\n') ;


                                        cout << c;

                                        cout << i;

                                        cout << r;

                                        cout << s;

                                        cout << endl;


                                        cout << "c=" << c << " i=" << i

                                                  << " r=" << r << s << endl;
```

```
(* Other variables declared as above *) /* Other variables declared as above
*/

fi, fo: text;                           FILE * fi, * fo;

(* fi will be opened for input,         /* fi will be opened for input,

fo will be opened for output *)         fo will be opened for output */

...                                     ...


readln(fi, c, i, r);                    fscanf(fi, "%c%d%f", & c, & i, & r);
```

```
                                        while (fgetc(fi) != '\n') ;

writeln(fo,'c=',c,' i=',i,' r=',r, s);  fprintf(fo,"c=%c i=%d r=%f%s\n",
                                                c,i,r,s);
```

C++ only:

```
                                        #include <fstream.h>

                                        ifstream fi;
                                        ofstream fo;

                                        fi >> c >> i >> r;
                                        while (fi.get() != '\n') ;

                                        fo << "c=" << c << " i=" << i
                                           << " r=" << r << s << endl;
```

NOTE: One minor difference in the above scanf() is that Pascal read and C do NOT skip leading whitespace when reading into a char, but C++ >> does. (However, C++ get() does not skip whitespace.)

Order of Program Parts

----- -- ------- -----

In standard Pascal, a program consists of a program header, followed by a block, followed by a period - all residing in a single source file. A block, in turn consists of the following parts, in the order listed:

(optional) label declaration(s)

(optional) const declaration(s)

(optional) type declaration(s)

(optional) var declaration(s)

(optional) function/procedure declaration(s)

(required) compound statement

A function/procedure declaration consists of a header, followed by a block, followed by a semicolon -

A C/C++ program consists of a series of declarations/definitions, which can appear in any order (so long as items are declared before they are used when necessary.) A C/C++ program of any significant size is almost always spread out over multiple source files.

A function definition consists of a header followed by a compound statement.

Declarations of variables (including const variables) can appear at the start of <u>any</u> compound statment.

Example:

```
if (x > y)
```

thus blocks can be nested within blocks.

Many Pascal implementations relax these requirements to allow, for example:
may

1. Declarations to appear in any order or mixture.

2. A program to be spread out over multiple source files.

-

control

```
{ int z;
  scanf("%d", &z);
  x = z + 32;
}
```

However, declarations of functions not appear inside the definition of another functions - thus function definitions cannot be nested as in Pascal.

C++ only:

Variable declarations can appear anywhere within a compound statement not just at the start, and can also appear in the first part of the portion of a for statement.

Example:

```
main()
  {
    cout << "How many times? ";
```

```
                                        int times;
                                        cin >> times;
                                        for (int i = 1; i <= times; i++)
                                            cout << "Hello, world!" <<
endl;
                                    }
```

The C/C++ Pre-processor

C and C++ compilers include a pre-processor that performs certain modifications on the program text BEFORE the compiler proper sees it. The following are examples of pre-processor directives found in typical C/C++ implementations, though some do not have all of these and some may have more. Note the similarity between the macro facility typically found in assemblers and the C/C++ pre-processor's #define and #if/#ifdef directives.

```
const
    size = 10                       #define size            10



NO PASCAL EQUIVALENT                #define iszero(e)       (e == 0)
NO PASCAL EQUIVALENT                #define equal(x, y)     (x == y)
NO PASCAL EQUIVALENT                #define error(f, m)     if (f) \
                                                               printf(m)
NO PASCAL EQUIVALENT                #define VAX
```

NO PASCAL EQUIVALENT	#undef VAX
NO PASCAL EQUIVALENT	#if wordlength >= 32 typedef int myint; #else typedef long myint; #endif
NO PASCAL EQUIVALENT	#ifdef VAX ... VAX-specific code #endif
NO PASCAL EQUIVALENT	#ifndef size #define size 100 /* default */ #endif

---

C and C++ make extensive use of header files incorporating declarations for constants, data, and code to support splitting a program across multiple source files. The #include directive is used to incorporate the contents of a header file into a source program.

NO PASCAL EQUIVALENT	#include <stdio.h>
NO PASCAL EQUIVALENT	#include "myinclude.h"

NOTE: The <filename> form is used for standard header files defining C/C++ library routines, which reside in a system directory; the "filename" form is used for programmer-written header files normally residing in the same directory as the program being compiled.

A common practice in C/C++ is to implement a "module" as two files - a header (.h) file containing declarations that clients of the module need to use, and an implementation (.c/cc) file that defines the entities declared in the header. Clients of the module #include the header, and the compiled version of the implementation is included into the executable when the program is linked.