

Borland C++ Compiler: Primitive Data Types, Variables and Constants

Introduction

A *primitive data type* is a data type provided as a basic building block by a programming language. It is predefined by the programming language and is named by a reserved keyword or keywords. In C++, primitive data types are used to define *variables* and *constants*. A variable's or constant's data type indicates what sort of value it represents, such as whether it is an integer, a floating-point number or a character, and determines the values it may contain and the operations that may be performed on it.

In C++ primitive data types can be used to represent data as *characters*, *integers*, *floating-point numbers* and *boolean values*, which are represented by data types as follows:

character

A *character* is a text character.

`char` The `char` data type can be used to represent a character.

`wchar_t` The `wchar_t` data type can be used to represent a wide character.

integer

An *integer* is a number without a fractional component.

In C++, declaration of most integer types can be prefixed with modifier `signed` or `unsigned` to indicate if the integer is signed, i.e. if its value contains an indication of whether it is positive or negative and/or modifier `short` or `long` to alter the range of values the integer can contain.

`char` The `char` data type can be used to represent a small integer.

Declaration of `char` type can be prefixed with modifier `signed` or `unsigned`.

`int` The `int` data type represents an integer.

Declaration of `int` type can be prefixed with modifier `signed` or `unsigned` and/or modifier `short` or `long`. If prefixed with any of these modifiers, the `int` keyword can be omitted.

`wchar_t` The `wchar_t` data type can be used to represent a short integer.

Declaration of `wchar_t` type cannot be prefixed with any modifier. Whether a `wchar_t` value is signed or unsigned varies between C++ compilers. In the Borland C++ Compiler a `wchar_t` value is unsigned.

floating-point number A *floating-point number* is a real number, or a number that may contain a fractional component. Floating-point types often contain an exponent that represents the number multiplied by 10^x .

`float` The `float` data type represents a single-precision floating-point number.

`double` The `double` data type represents a double-precision floating-point number.

Declaration of `double` type can be prefixed with modifier `long` to provide an extended-precision floating-point number.

boolean value A *boolean value* is a binary value, which can be in one of two states, often *true* or *false*.

`bool` The `bool` data type represents a boolean value.

This article demonstrates declaration and use of each primitive data type provided by the C++ programming language.

The primitive data types available in C++ are as follows:

Type	Description	Bytes *	Range *
char	character or small integer	1	signed: -128 to 127 unsigned: 0 to 255
int	integer	short: 2 normal: 4 long: 4	signed short: -32,768 to 32,767 unsigned short: 0 to 65,535 signed: -2,147,483,648 to 2,147,483,647 unsigned: 0 to 4,294,967,295 signed long: -2,147,483,648 to 2,147,483,647 unsigned long: 0 to 4,294,967,295
bool	boolean value	1	true or false
float	floating-point number	4	1.17549×10^{-38} to 3.40282×10^{38}
double	double-precision floating-point number	8	2.22507×10^{-308} to 1.79769×10^{308}
long double	extended-precision floating-point number	10	$3.36210 \times 10^{-4932}$ to 1.18973×10^{4932}
wchar_t	wide character or short integer	2	1 wide character

* The values in columns *Bytes* and *Range* depend on the system the program is compiled for and the compiler used. The values shown above are for the Borland C++ compiler and generally represent values found on most 32-bit systems. For other systems, the general specification is that `int` has the natural size suggested by the system architecture (one word) and the four integer types `char`, `short int`, `int` and `long int` must each be at least as large as the one preceding it, with `char` always being 1 byte in size. The same applies to the floating-point types `float`, `double` and `long double`, where each one must provide at least as much precision as the preceding one.

The C++ programming language is *strongly-typed*, which means that all variables and constants must first be declared before they can be used.

This article demonstrates declaration and use of constants and variables of each primitive data type provided by the C++ programming language.

Concepts

value

A *value* is a sequence of bits that is interpreted according to some data type. It is possible for the same sequence of bits to have different values, depending on the type used to interpret its meaning.

In the *primitiv* program, values as literal constants are assigned to variables and symbolic constants.

data

Data is a measurement which can be organised to become information.

In English, the word *datum* refers to “something given”. The word *data* is plural in English, but it is commonly treated as a mass noun and used in the singular. In everyday language, data is a synonym for information. However, in exact science there is a clear distinction between data and information, where data is a measurement that may be disorganised and when the data becomes organised it becomes information.

The values in the *primitiv* program are data organised according to their *data type*.

bit

The word *bit* is short for *binary digit*, which is the smallest unit of information on a computer. A single bit can hold only one of two values, which are usually represented by numbers 0 and 1.

More meaningful information is obtained by combining consecutive bits into larger units. For example, a *byte* is a unit of information composed of eight consecutive bits.

All values, including those used in the *primitiv* program, are represented by a sequence of bits.

byte

A *byte* is a unit of measurement of information storage, commonly composed of eight bits. If composed of eight bits, a single byte can hold one of 256 (2^8) values.

Data can be represented in a single byte or a combination of bytes.

All values used in the *primitiv* program are represented by a single byte or a combination of bytes.

character

A *character* is a text character.

The *primitiv* program declares character variables of type `char` and `wchar_t`.

integer

An *integer* is a number without a fractional component.

The *primitiv* program declares integer variables of type `char`, `int` and `wchar_t`.

floating-point number

A *floating-point number* is a real number, or a number that may contain a fractional component.

The *primitiv* program declares floating-point variables of type `float` and `double`.

boolean value

A *boolean value* is a binary value, which can be in one of two states, often *true* or *false*.

The *primitiv* program declares boolean variables of type `bool`.

data type

A *data type*, or *type*, is a classification of a particular kind of information. It can be defined by its permissible values and operations.

Data types used in the *primitiv* program are C++ primitive data types.

primitive data type

A *primitive data type* is a data type provided as a basic building block by a programming language. It is predefined by the programming language and is named by a reserved keyword or keywords.

This article demonstrates declaration and use of each primitive data type provided by the C++ programming language.

type specifier

The data type of an entity is specified using a *data type specifier*, sometimes called a *type specifier*.

Data types specified in the *primitiv* program are the C++ primitive data types.

identifier

An *identifier* is a token that names an entity.

The concept of an identifier is analogous to that of a name. Naming entities makes it possible to refer to them, which is essential for any kind of symbolic processing.

Variables and constants in the *primitiv* program are represented by identifiers.

keyword

A *keyword* is a word or identifier that has a particular meaning to its programming language.

Keywords are used in the *primitiv* program to specify the types of variables and constants and to modify them.

modifier

A *modifier keyword*, or *modifier*, is a keyword that modifies an entity.

Modifiers `unsigned`, `short`, `long` and `const` are used in the *primitiv* program.

declaration statement

A *declaration statement*, often referred to simply as a *declaration*, specifies aspects of an entity, such as its dimensions, identifier and type and is used to announce its existence. This is important in C++ which requires variables and constants to be declared before use.

Variables and constants are declared in the *primitiv* program.

#define

The `#define` preprocessor directive declares a constant.

Constant `cid` in the *primitiv* program is declared using a `#define` directive.

Primitive data types are used in variables and constants. C++ has two kinds of constants: *symbolic constants* and *literal constants*.

variable A *variable* is a symbolic representation denoting a value or expression that can change. It is a symbol or identifier that stands for a value. For example, in expression
 `b + c`
`b` and `c` may be variables.

The *primitiv* program uses variables representing each of the primitive data types.

symbolic constant A *symbolic constant*, *constant variable*, or *constant* for short, is a variable whose value cannot be changed once it is initially bound to a value. A constant variable cannot be assigned to. It is specified only once, but can be referenced multiple times.

A constant can be declared by using either the `#define` preprocessor directive or the `const` keyword. A constant declared using the `const` keyword must be assigned a value when declared, and then that value can not be changed.

A constant is declared in the *primitiv* program using both a `#define` directive and a statement containing the `const` keyword.

literal constant A *literal constant*, or *literal*, is a literal value inserted into source code. It is a constant because its value cannot be changed.

The *primitiv* program assigns literal values to variables representing each of the primitive data types.

A *literal constant* literally declares its value, which is directly inserted into the source code of a program. The following kinds of data can be represented by a literal:

character A single character can be inserted into code by enclosing it in single quotes ' '.

For example, ' a ' represents character a.

A character literal can have an L prefix that specifies a double-byte literal.

string A string of characters can be inserted into code by enclosing the characters in double quotes " ".

For example, "Hello" represents word "Hello".

integer An integer can be inserted into code as a number without a decimal point or exponent.

For example, 1234 represents integral number 1234.

An integer literal can have a u or U suffix that means it is unsigned and/or an l or L suffix that means it is a long integer. However, these suffixes are often optional, as the compiler can frequently tell from context what kind of literal is required.

floating-point A floating-point number can be inserted into code as a number with a decimal point and/or exponent.

For example, 1.234 represents floating-point number 1.234, 3e2 represents floating-point number 300 and 1.234e2 represents floating-point number 123.4.

By default, a floating-point literal has a type of double. To specify a float value, the f or F suffix can be used. A floating-point literal can also have an l or L suffix to make it long double.

Source Code

The source code listing is as follows:

```

/*
  primitiv.cpp

  Primitive data types.

  environment: language C++
               platform Windows console
*/

#include <stdio.h>

#define cid 234

int main()
{
  // Declare variables.

  char      c                ; // character
  wchar_t  w                ; // wide character
  int       in , iz , ip , id , ic ; // integer
  float     fn , fz , fp , ff    ; // floating-point
  bool      bf , bt          ; // boolean

  // floating-point

  float fm6 , fm5 , fm4 , fm3 , fm2 , fm1 ,
        f0 , f1 , f2 , f3 , f4 , f5 , f6 , f7 ;

  // range: integer

          char      scn , scp ;
unsigned   char      uc      ;
          short     int      ssn , ssp ;
unsigned short     int      us      ;
          int       int      sin , sip ;
unsigned          int      ui      ;
          long      int      sln , slp ;
unsigned long     int      ul      ;
          wchar_t  uw      ;

  // range: floating-point

          float     fns , fnl , fps , fpl ;
          double    dns , dnl , dps , dpl ;
long double ldns , ldnl , ldps , ldpl ;

  // Populate variables.

  // character

          c = 'c' ;
char p = 'p' ;
          w = L'w' ;

```

```
// integer

const int cic = 345 ;

    in  = -123 ;
    iz  =  0  ;
    ip  = 123 ;
    id  = cid ;
    ic  = cic ;

// floating-point

fn  = -123      ;
fz  =  0        ;
fp  = 123       ;
ff  = 12.3      ;

fm6 = 1.23e-6  ;
fm5 = 1.23e-5  ;
fm4 = 1.23e-4  ;
fm3 = 1.23e-3  ;
fm2 = 1.23e-2  ;
fm1 = 1.23e-1  ;
f0  = 1.23e0   ;
f1  = 1.23e1   ;
f2  = 1.23e2   ;
f3  = 1.23e3   ;
f4  = 1.23e4   ;
f5  = 1.23e5   ;
f6  = 1.23e6   ;
f7  = 1.23e7   ;

// boolean

bf = false ;
bt = true  ;

// range: integer

scn = -128 ;
scp = 127  ;
uc  = 255  ;

ssn = -32768 ;
ssp = 32767 ;
us  = 65535 ;

sin = -2147483648 ;
sip = 2147483647  ;
ui  = 4294967295  ;

sln = -2147483648 ;
slp = 2147483647  ;
ul  = 4294967295  ;

uw  = 65535 ;

// range: floating-point
```

```

fns = -1.17549e-38 ;
fnl = -3.40282e+38 ;
fps = 1.17549e-38 ;
fpl = 3.40282e+38 ;

dns = -2.22507e-308 ;
dnl = -1.79769e+308 ;
dps = 2.22507e-308 ;
dpl = 1.79769e+308 ;

ldns = -3.36210e-4932 ;
ldnl = -1.18973e+4932 ;
ldps = 3.36210e-4932 ;
ldpl = 1.18973e+4932 ;

// Display variables.

printf( "char      " ) ;
printf( "%c" , c ) ;
printf( " " ) ;
printf( "%c" , p ) ;
printf( "\n" ) ;

printf( "wchar_t   %c\n" , w ) ;
printf( "int        %d %d %d\n" , in , iz , ip ) ;
printf( "float       %g %g %g %g\n" , fn , fz , fp , ff ) ;
printf( "           %g %g %g %g %g %g\n" , fm6 , fm5 , fm4 , fm3 , fm2 , fm1
) ;
printf( "           %g %g %g %g %g %g %g %g\n" ,
f0 , f1 , f2 , f3 , f4 , f5 , f6 , f7
) ;
printf( "bool        %d %d\n" , bf , bt ) ;
printf( "constants %d %d\n" , id , ic ) ;

// range

printf( "\nrange:\n\n" ) ;

printf( "char        %d %d %u\n" , scn , scp , uc ) ;
printf( "short       %hd %hd %hu\n" , ssn , ssp , us ) ;
printf( "int         %d %d %u\n" , sin , sip , ui ) ;
printf( "long        %ld %ld %lu\n" , sln , slp , ul ) ;
printf( "wchar_t     %u\n" , uw ) ;

printf( "float       %g %g %g %g\n" , fns , fnl , fps , fpl ) ;
printf( "double      %g %g %g %g\n" , dns , dnl , dps , dpl ) ;
printf( "long double %Lg %Lg %Lg %Lg\n" , ldns , ldnl , ldps , ldpl ) ;
}

```

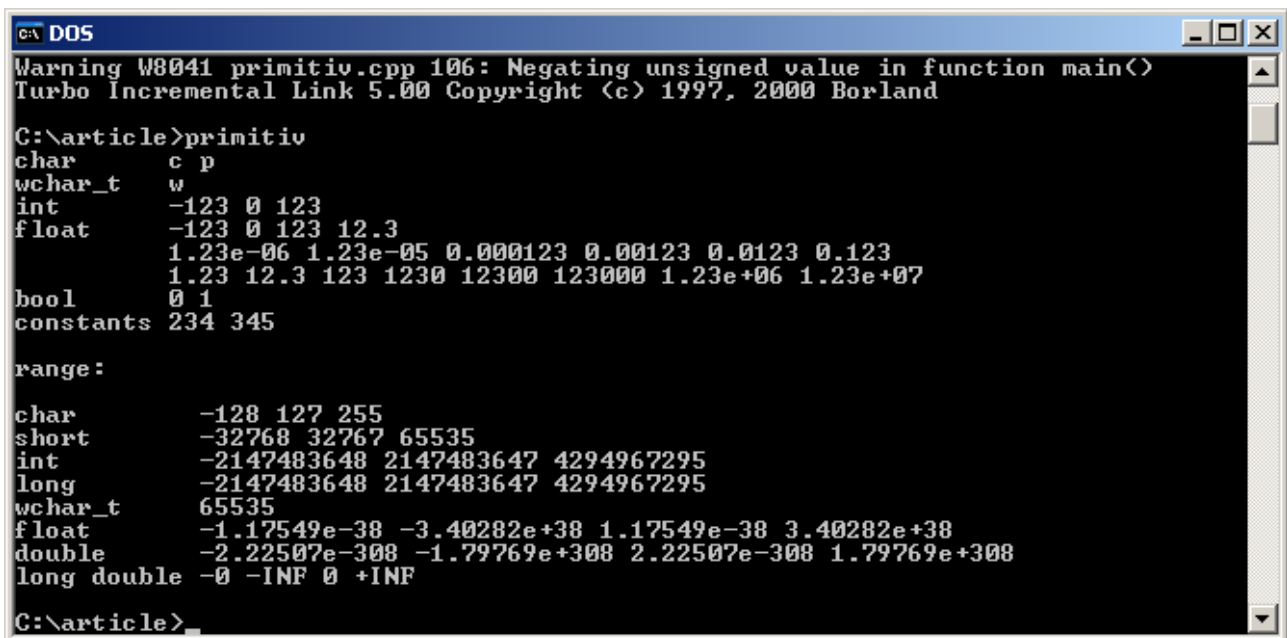
Compiling and Running

1. Save the source code listing into a file named `primitiv.cpp`.
2. Launch a Windows command prompt.
3. Navigate to the directory `primitiv.cpp` was saved in.
4. To compile the program, type:

```
> BCC32 primitiv
```

5. To run the program, type

```
> primitiv
```



```
C:\DOS
Warning W8041 primitiv.cpp 106: Negating unsigned value in function main()
Turbo Incremental Link 5.00 Copyright (c) 1997, 2000 Borland

C:\article>primitiv
char      c p
wchar_t   w
int       -123 0 123
float     -123 0 123 12.3
          1.23e-06 1.23e-05 0.000123 0.00123 0.0123 0.123
          1.23 12.3 123 1230 12300 123000 1.23e+06 1.23e+07
bool      0 1
constants 234 345

range:
char      -128 127 255
short     -32768 32767 65535
int       -2147483648 2147483647 4294967295
long      -2147483648 2147483647 4294967295
wchar_t   65535
float     -1.17549e-38 -3.40282e+38 1.17549e-38 3.40282e+38
double    -2.22507e-308 -1.79769e+308 2.22507e-308 1.79769e+308
long double -0 -INF 0 +INF

C:\article>
```

Code Explanation

```
#define cid 234
```

The `#define` preprocessor directive declares a constant.

This directive declares a constant identified by `cid` containing value of 234.

```
char c ; // character
```

A variable is a symbolic representation denoting a value or expression. The format of a simple variable declaration is as follows:

```
type identifier ;
```

where

`type` is the type specifier of the variable.

The type specifier in the declaration statement above is `char`, which is used to specify a character.

`identifier` is the identifier by which it will be possible to refer to the variable.

The identifier in the declaration statement above is `c`.

`;` closes the declaration and delimits it from the next statement.

All statements in C++, including the declaration statement above, are closed with the semicolon character `;`.

The statement above declares a character variable identified by `c`.

```
wchar_t w ; // wide character
```

This statement declares a wide character variable.

```
int in , iz , ip , id , ic ; // integer
```

Multiple variables of the same type can be declared in a single declaration statement. The format of a multiple variable declaration is as follows:

```
type identifier[ , identifier[ , identifier[ , ... ]]] ;
```

where

`type` is the type specifier of the variables.

The type specifier in the declaration statement above is `int`, which specifies integers.

`identifier` is the identifier by which it will be possible to refer to each variable.

The identifiers in the declaration statement above are `in`, `iz`, `ip`, `id` and `ic`.

`,` delimits the variables being identified.

All multiple declarations in C++, including the declaration above, delimit the variables with commas `,`.

`;` closes the declaration and delimits it from the next statement.

All statements in C++, including the declaration statement above, are closed with the semicolon character `;`.

The statement above declares multiple integer variables identified by `in`, `iz`, `ip`, `id` and `ic`.

```
float fn , fz , fp , ff ; // floating-point  
bool bf , bt ; // boolean
```

These statements declare floating-point and boolean variables.

```
float fm6 , fm5 , fm4 , fm3 , fm2 , fm1 ,  
      f0 , f1 , f2 , f3 , f4 , f5 , f6 , f7 ;
```

The floating-point variables declared in this statement are used to demonstrate a range of floating-point values.

```
char scn , scp ;
```

This statement declares variables for use as signed small integers.

The following declarations are equivalent:

```
    char scn , scp ;  
signed char scn , scp ;
```

```
unsigned char uc ;
```

This statement declares a variable for use as an unsigned small integer.

```
short int ssn , ssp ;
```

This statement declares signed short integer variables.

The following declarations are equivalent:

```
    short int ssn , ssp ;  
signed short int ssn , ssp ;  
    short      ssn , ssp ;  
signed short      ssn , ssp ;
```

```
unsigned short int us ;
```

This statement declares an unsigned short integer variable.

The following declarations are equivalent:

```
unsigned short int      us ;  
unsigned short          us ;  
                        wchar_t us ;
```

```
int sin , sip ;
```

This statement declares signed integer variables.

The following declarations are equivalent:

```
    int sin , sip ;  
signed int sin , sip ;  
signed      sin , sip ;
```

```
unsigned int ui ;
```

This statement declares an unsigned integer variable.

The following declarations are equivalent:

```
unsigned int ui ;  
unsigned      ui ;
```

```
long int sln , slp ;
```

This statement declares signed long integer variables.

The following declarations are equivalent:

```
        long int sln , slp ;  
signed long int sln , slp ;  
        long      sln , slp ;  
signed long      sln , slp ;
```

```
unsigned long int ul ;
```

This statement declares an unsigned long integer variable.

The following declarations are equivalent:

```
unsigned long int ul ;  
unsigned long      ul ;
```

```
wchar_t uw ;
```

This statement declares an unsigned short integer variable.

The following declarations are equivalent:

```
        wchar_t uw ;  
unsigned short int  uw ;  
unsigned short      uw ;
```

```
        float fns , fnl , fps , fpl ;  
        double dns , dnl , dps , dpl ;  
long double ldns , ldnl , ldps , ldpl ;
```

These statements declare variables for each of the primitive floating-point types provided by the C++ language.


```
c = 'c' ;
```

A character variable stores a character as a single byte, which can contain one of 256 discrete values. A character variable can therefore contain one of 256 distinct characters.

To assign is to set or re-set a value denoted by an identifier. An assignment statement uses the assignment operator = to assign the value of an expression to an entity. It is formatted as follows:

```
assignee = expression ;
```

where

`assignee` is the entity (variable or constant) to which the value is assigned.

The assignee in the statement above is variable `c`.

`=` is the assignment operator.

The assignment operator in the statement above, as in all assignment statements is denoted by the equal character `=`.

`expression` is the expression that is evaluated.

The expression in the statement above is literal constant `'c'`.

`;` closes the statement and delimits it from the next statement.

All statements in C++, including the statement above, are closed with the semicolon character `;`.

The statement above assigns literal value “c” to character variable `c`.

```
char p = 'p' ;
```

A declaration and assignment can be combined into a single statement. A combined declaration and assignment statement contains the elements of both a declaration statement and an assignment statement, and is formatted as follows:

```
type identifier = expression ;
```

where

`type` is the type specifier of the entity being declared.

The type specifier in the statement above is `char`, which is used to specify a character.

`identifier` is the identifier by which it will be possible to refer to the entity.

The identifier in the statement above is `p`.

`=` is the assignment operator.

The assignment operator in the statement above, as in all assignment statements is denoted by the equal character `=`.

`expression` is the expression that is evaluated.

The expression in the statement above is literal constant `'p'`.

`;` closes the statement and delimits it from the next statement.

All statements in C++, including the statement above, are closed with the semicolon character `;`.

The statement above declares a character variable identified by `p` and assigns literal value “p” to it.

```
w = L'w' ;
```

A wide character variable stores a character in two bytes, which can contain one of 65,536 discrete values. A double-byte literal is written by prefixing the literal with `L`.

This statement assigns literal value “w” to wide character variable `w`.

```
const int cic = 345 ;
```

A constant can be declared in the same way as a variable, preceded by the `const` modifier.

This statement declares an integer constant identified by `cic` and assigns literal value 345 to it.

```
in = -123 ;  
iz = 0 ;  
ip = 123 ;
```

A signed integer variable can contain positive and negative integral values.

These statements assign literal values -123, 0 and 123 to integer variables `in`, `iz` and `ip` respectively.

```
id = cid ;  
ic = cic ;
```

Valid expressions in an assignment statement include literal and symbolic constants and variables.

These statements assign the value of symbolic constant `cid` to integer variable `id` and integer symbolic constant `cic` to integer variable `ic`.

```
fn = -123 ;  
fz = 0 ;  
fp = 123 ;  
ff = 12.3 ;
```

A floating-point variable can contain positive and negative values and may also contain a fractional component.

These statements assign literal values -123, 0, 123 and 12.3 to floating-point variables `fn`, `fz`, `fp` and `ff` respectively.

```

fm6 = 1.23e-6 ;
fm5 = 1.23e-5 ;
fm4 = 1.23e-4 ;
fm3 = 1.23e-3 ;
fm2 = 1.23e-2 ;
fm1 = 1.23e-1 ;
f0  = 1.23e0  ;
f1  = 1.23e1  ;
f2  = 1.23e2  ;
f3  = 1.23e3  ;
f4  = 1.23e4  ;
f5  = 1.23e5  ;
f6  = 1.23e6  ;
f7  = 1.23e7  ;

```

Floating-point types can contain an exponent that represents the number multiplied by $10^{<x>}$. This is represented by appending $e<x>$ to the number where $<x>$ represents a power of 10.

These statements assign literal values to floating-point variables as follows:

```

1.23 * 10-6    fm6
1.23 * 10-5    fm5
1.23 * 10-4    fm4
1.23 * 10-3    fm3
1.23 * 10-2    fm2
1.23 * 10-1    fm1
1.23 * 100     f0
1.23 * 101     f1
1.23 * 102     f2
1.23 * 103     f3
1.23 * 104     f4
1.23 * 105     f5
1.23 * 106     f6
1.23 * 107     f7

```

When the *primitiv* program runs these variables are displayed as 1.23e-006, 1.23e-005, 0.000123, 0.00123, 0.0123, 0.123, 1.23, 12.3, 123, 1230, 12300, 123000, 1.23e+006 and 1.23e+007 respectively. Each number is displayed with or without the decimal point or exponent as appropriate to make the number readable.

```

bf = false ;
bt = true  ;

```

A boolean value is a binary value, or one that can be in one of two states, often true or false.

These statements assign literal values false and true to boolean variables *bf* and *bt* respectively.

```
scn =      -128 ;
scp =       127 ;
uc  =       255 ;

ssn =     -32768 ;
ssp =      32767 ;
us  =     65535 ;

sin = -2147483648 ;
sip =  2147483647 ;
ui  =  4294967295 ;

sln = -2147483648 ;
slp =  2147483647 ;
ul  =  4294967295 ;

uw  =       65535 ;
```

The range of values that can be contained in each integer type is as follows:

char	signed:	-128 to 127
	unsigned:	0 to 255
int	signed short:	-32768 to 32767
	unsigned short:	0 to 65535
	signed:	-2147483648 to 2147483647
	unsigned:	0 to 4294967295
	signed long:	-2147483648 to 2147483647
	unsigned long:	0 to 4294967295
wchar_t	unsigned:	0 to 65535

The statements above assign the minimum and maximum signed and maximum unsigned literal values to variables of each of the integer types.

The statements assigning literal -2147483648 to `sin` and `sln` cause the following warning to be issued:

```
    Negating unsigned value in function main()
```

This warning is issued because the absolute value of the literal is larger than the largest permissible positive value for a signed integer, and so an unsigned integer is used. If literal -2145483647 is used instead this warning is not issued. The warning, however, is a warning and not an error, and the program still compiles successfully.

```

fns  = -1.17549e-38   ;
fnl  = -3.40282e+38   ;
fps  =  1.17549e-38   ;
fpl  =  3.40282e+38   ;

dns  = -2.22507e-308  ;
dnl  = -1.79769e+308  ;
dps  =  2.22507e-308  ;
dpl  =  1.79769e+308  ;

ldns = -3.36210e-4932 ;
ldnl = -1.18973e+4932 ;
ldps =  3.36210e-4932 ;
ldpl =  1.18973e+4932 ;

```

The range of absolute values that can be contained in each floating-point type is as follows:

```

float          1.17549*10-38 to 3.40282*1038

double        2.22507*10-308 to 1.79769*10308

long double   3.36210*10-4932 to 1.18973*104932

```

The statements above assign the minimum and maximum positive and negative literal values to variables of each of the floating-point types.

Extended-precision floating-point variables can contain values outwith the range that can easily be displayed using the `printf` function. In the *primitiv* program, values $-3.36210 \times 10^{-4932}$, $-1.18973 \times 10^{4932}$, $3.36210 \times 10^{-4932}$ and 1.18973×10^{4932} are displayed as 0, `-INF` (negative infinity), 0 and `+INF` (positive infinity) respectively.

```
printf( "%c" , c ) ;
```

Variables and literal and symbolic constants can be printed to the standard output stream using the `printf` function.

To print a variable or symbolic constant using `printf`, a *tag* is included in the *format* argument (the first argument passed to the `printf` function). If a tag is included in the *format* argument, an additional argument must be passed to the `printf` function containing the value to be printed. When the *format* argument is printed to the standard output stream, each tag is replaced by the value of its corresponding argument. The first character in a tag is the `%` character, which identifies that a tag is being included. The last character in a tag is the *specifier* which defines the type of the tag. In this statement, the tag is `%c` and the specifier is therefore `c`, which specifies that the tag is to be replaced by a character.

This statement prints the value of `char` variable `c` as a character to the standard output stream.

```
printf( "\n" ) ;
```

A literal constant can contain characters that are outwith the normal alphanumeric range. To include such a character, an *escape sequence* can be used to represent it. The `\n` escape sequence represents a new-line character. A new-line character indicates the end of the current line and causes the next character to be displayed on the following line.

This statement prints a new-line character to the standard output stream.

```
printf( "wchar_t   %c\n" , w ) ;
```

The `format` argument passed to the `printf` function in this statement consists of text containing a character tag, which is replaced by the value of wide character variable `w`.

Since the value of `w` is “w”, the text printed to the standard output stream by this statement is “wchar_t w” followed by a new-line character.

```
printf( "int       %d %d %d\n" , in , iz , ip ) ;
```

If multiple values are to be printed to the standard output stream, instead of using repeated statements, a `format` argument containing multiple tags can be passed to the `printf` function. For each tag included in the `format` argument, a corresponding additional argument containing the value to be printed must also be passed.

The tags in this statement use the `d` specifier, which specifies that each tag is to be replaced by a signed decimal integer.

This statement prints text including the values of `int` variables `in`, `iz` and `ip` to the standard output stream.

```
printf( "float     %g %g %g %g\n" , fn , fz , fp , ff ) ;
```

The tags in this statement use the `g` specifier, which specifies that each tag is to be replaced by a decimal floating-point number, each number being displayed with or without the decimal point or exponent as appropriate to make the number readable.

This statement prints text including the values of `float` variables `fn`, `fz`, `fp` and `ff` to the standard output stream.

```
printf( "bool      %d %d\n" , bf , bt ) ;
```

This statement prints text including the values of `bool` variables `bf` and `bt` to the standard output stream. The `d` specifier in the tags causes the values to be converted to integers, where true is converted to 1 and false is converted to 0.

```
printf( "char          %d %d %u\n" , scn , scp , uc ) ;
```

The `u` specifier in a tag specifies that the tag is to be replaced by an unsigned decimal integer.

This statement prints text including the values of `char` variables `scn` and `scp` as signed integers and unsigned `char` variable `uc` as an unsigned integer.

```
printf( "short          %hd %hd %hu\n" , ssn , ssp , us ) ;
```

An integer specifier in a tag can be prefixed with length definition `h`, which interprets the length of the corresponding argument as short.

This statement prints text including the values of `short int` variables `ssn` and `ssp` as signed short integers and unsigned `short int` variable `us` as an unsigned short integer.

```
printf( "long           %ld %ld %lu\n" , sln , slp , ul ) ;
```

An integer specifier in a tag can be prefixed with length definition `l`, which interprets the length of the corresponding argument as long.

This statement prints text including the values of `long int` variables `sln` and `slp` as signed long integers and unsigned `long int` variable `ul` as an unsigned long integer.

```
printf( "long double %Lg %Lg %Lg %Lg\n" , ldns , ldnl , ldps , ldpl ) ;
```

A floating-point specifier in a tag can be prefixed with length definition `L`, which interprets the corresponding argument as long double.

This statement prints text including the values of `long double` variables `ldns`, `ldnl`, `ldps` and `ldpl` as extended-precision floating-point numbers.

Extended-precision floating-point variables can contain values outwith the range that can easily be displayed using the `printf` function. This statement prints `ldns`, `ldnl`, `ldps` and `ldpl` values $-3.36210 \cdot 10^{-4932}$, $-1.18973 \cdot 10^{4932}$, $3.36210 \cdot 10^{-4932}$ and $1.18973 \cdot 10^{4932}$ as `0`, `-INF`, `0` and `+INF` respectively.

Terms

- assign** To *assign* is to set or re-set a value denoted by a variable name.
- Values are assigned to each of the variables and constants declared in the *primitiv* program.
- assignment statement** An *assignment statement* assigns a value to a variable. The assignment statement often allows a variable to contain different values at different times during program execution.
- Values in the *primitiv* program are assigned using assignment statements.
- assignment operator** An assignment statement uses the *assignment operator* = to assign the value of an expression to an entity.
- Assignment statements in the *primitiv* program use the assignment operator.

Further Comments

Declarations

The declaration formats provided earlier in this article were simplified to explain their use in context. A more complete outline of the format of a variable or constant declaration is as follows:

```
[const ][modifiers ]type
identifier[ = expression][ , identifier[ = expression][ , ...]]
;
```

where

<code>const</code>	is the <code>const</code> keyword and indicates that a constant is being declared. If the <code>const</code> modifier is omitted, a variable is declared.
<code>modifiers</code>	modify the range and precision of the variable or constant and whether it can contain negative numbers.
<code>type</code>	is the type specifier of the variable or constant.
<code>identifier</code>	is the identifier by which it will be possible to refer to the variable or constant.
<code>=</code>	is the assignment operator <code>=</code> and indicates that the value of an expression is to be assigned.
<code>expression</code>	is the expression that is evaluated.
<code>,</code>	delimits the variables or constants being identified.
<code>;</code>	closes the statement and delimits it from the next statement.

In C++ there are two ways to initialise variable values during declaration: one, known as *c-like initialisation*, is to use the assignment operator `=` to assign the initial value as described above; the other, known as *constructor initialisation*, is to enclose the initial value between parentheses `()`. For example, the following are equivalent:

```
int a = 123 ;
int a( 123 ) ;
```

Characters

Characters are represented by various encoding schemes and character sets:

SBCS In the *SBCS* (single-byte character set) encoding scheme, all characters are exactly one byte long. ASCII is an example of an SBCS.

Single-byte characters are represented by the `char` type.

MBCS An *MBCS* (multi-byte character set) encoding contains some characters that are one byte long and others that are more than one byte long. The MBCS schemes used in Windows contain two character types: single-byte characters and double-byte characters. Since the largest multi-byte character used in Windows is two bytes long, the term double-byte character set, or DBCS, is commonly used in place of MBCS.

Multi-byte characters are represented by the `char` type.

Unicode *Unicode* is an encoding standard in which all characters are two bytes long. Unicode characters are sometimes called wide characters because they are wider (use more storage) than single-byte characters.

Unicode characters are represented by the `wchar_t` type.

The `wchar_t` data type in ANSI/ISO C is intended to represent wide characters. Wide character is a vague term used to represent a data type that is richer than the traditional 8-bit characters. It is not the same thing as Unicode.

Floating-Point Storage

Floating-point representations vary from machine to machine. By far the most common is the IEEE-754 standard.

An IEEE-754 `float` (4 bytes) or `double` (8 bytes) has three components (there is also an analogous 96-bit extended-precision format under IEEE-854): a sign bit telling whether the number is positive or negative, an exponent giving its order of magnitude, and a mantissa specifying the actual digits of the number. The bit layouts used by the Borland C++ Compiler to represent floating-point numbers are as follows:

```
float      s eeeeeee mmmmmmm,mmmmmmmm,mmmmmmmm

double    s ee,eeeeeee mmm,mmmmmmmm,mmmmmmmm,mmmmmmmm,mmmmmmmm,mmmmmmmm,mmmmmmmm

long      s eeeeeee,eeeeeee
double    mmmmmmm,mmmmmmmm,mmmmmmmm,mmmmmmmm,mmmmmmmm,mmmmmmmm,mmmmmmmm,mmmmmmmm
```

These bit layouts illustrate the bytes used in sequence with spaces and commas added for clarity. The letters used are as follows:

s = sign; e = exponent; m = mantissa

The absolute value is mantissa * 2^{exponent} . Floating-point values are stored as binary fractions, so that 0.1 equates to $\frac{1}{2}$. The place values to the right of the binary point are 2^{-1} , 2^{-2} , etc., just as the place values to the right of the decimal point are 10^{-1} , 10^{-2} , etc. in decimal.

There is a potential problem when storing both a mantissa and an exponent: $2 \cdot 10^{-1} = 0.2 \cdot 10^0 = 0.02 \cdot 10^1$ and so on. This could correspond to different bit patterns representing the same quantity, which would be wasteful. This problem is circumvented by interpreting the whole mantissa as being to the right of the binary point, with an implied 1 always present to the left of the binary point. Unless the number is zero, it will contain at least one 1. The number is shifted so that the most significant 1 is the only digit to the left of the binary point, the digits after the binary point are stored in the mantissa and the exponent is adjusted appropriately. For example, decimal $10 = \text{binary } 1010 = 1.01 \cdot 2^3$. $010000\dots$ is stored in the mantissa and the exponent is adjusted to represent 3. If the number is zero every bit in the variable is set to zero. If the exponent contained the exact power of two to which the number was raised it would be impossible to store the number 1 because $1 = \text{binary } 1.0 \cdot 2^0$ and storing the number would set every bit to zero, which would be interpreted as 0. The solution to this is to add a number to the exponent (normally half its range) when storing it. For example, the exponent of a float is *shift-127* encoded, meaning that the actual exponent is eeeeeee minus 127. When storing the number 1, the value of the exponent is therefore 127.

There are a number of bit patterns in a floating-point number that constitute special cases. These are as follows:

- zero Every bit in the sign, exponent and mantissa is reset (set to 0).
- infinity Every bit in the exponent is set (set to 1) and every bit in the mantissa is reset. The sign can be used to indicate positive or negative infinity.
- not-a-number (NaN) Every bit in the exponent is set and any bit in the mantissa is set. The sign can be used to indicate positive or negative NaN.
 A NaN value is generated as the result of an operation that does not make sense, for example, a non-real number or the result of an operation like infinity times zero.

In general: value = (<sign> ? -1:1) * 2^{<exponent>} * 1.<mantissa bits> where 1.<mantissa bits> is in binary

To clarify, some examples of float values follow:

Value	Sign	Exponent	Mantissa
0	0	00000000	0000000,00000000,00000000
1	0	01111111	0000000,00000000,00000000
-1	1	01111111	0000000,00000000,00000000
3	0	10000000	1000000,00000000,00000000
0.5	0	01111110	0000000,00000000,00000000
1.175494350822290000000e-038	0	00000001	0000000,00000000,00000000
1.401298464324820000000e-045	0	00000000	0000000,00000000,00000001
3.402823466000000000000e+038	0	11111110	1111111,11111111,11111111
+INF	0	11111111	0000000,00000000,00000000
-INF	1	11111111	0000000,00000000,00000000
+NaN	0	11111111	1000000,00000000,00000000

The table above illustrates another special case. The smallest float value available when the mantissa is represented by 1.<mantissa bits> is 1.17549435082229e-38. The smallest exponent allowed is -126, represented in exponent bits by 00000001. For smaller values the exponent bits are all reset and numbers other than zero are represented by setting mantissa bits. As long as there is an implied leading 1, the smallest number available is 2⁻¹²⁶; to get smaller values an exception is made. The 1.<mantissa bits> interpretation is no longer used, and the number's magnitude is determined purely by bit positions. For example, 1.40129846432482e-45 = 2⁻¹²⁶⁻²³, in other words the smallest exponent minus the number of mantissa bits.

Clearly, when using these extra-small numbers precision is sacrificed. When there is no implied 1, all bits to the left of the first set bit are leading zeros, which add no information to the number. Therefore the absolutely smallest representable number (1.40129846432482e-45, with only the lowest bit of the mantissa set) has only a single bit of precision.

The precision of floating-point numbers is dictated by the number of bytes in the mantissa, and varies between floating-point types. The exact precision for a particular type is most accurately quoted as a number of binary digits. Conversion to precision quoted in decimal digits requires an approximation.

The range quoted for each floating-point type earlier in this article was based on the smallest value maintaining full precision. However, by accepting reduced precision, smaller numbers can be obtained. Full precision in decimal digits and the range of positive values for each of the floating-point types in the Borland C++ Compiler is as follows:

Type	Full Precision (digits)	Extreme	Value
float	7	maximum	$3.40282346600000000000000000000000 * 10^{38}$
		minimum: full precision	$1.1754943508222900000000 * 10^{-38}$
		minimum: reduced precision	$1.4012984643248200000000 * 10^{-45}$
double	15	maximum	$1.797693134862315700000 * 10^{308}$
		minimum: full precision	$2.225073858507202000000 * 10^{-308}$
		minimum: reduced precision	$4.940656458412465400000 * 10^{-324}$
long double	19	maximum	$1.18973149535723176495 * 10^{4932}$
		minimum: full precision	$3.36210314311209350700 * 10^{-4932}$
		minimum: reduced precision	$3.645199531882474600000 * 10^{-4951}$

Loss of significance can raise unexpected problems when using floating-point data. Loss of significance refers to situations where precision can inadvertently be lost through information being discarded, potentially returning surprisingly poor results. When using floating-point types it is important to bear in mind that floating-point variables have a tendency to become corrupted as operations are performed on them. Often integers can be used instead of floating-point numbers, for example by storing integral numerators and denominators, providing more accurate results.