1. **Definition**: An integer is a number without a fractional part. The set of integers is the union of the set of whole numbers and the set of negative counting numbers.
   1.1. Integers and whole numbers. C++ implements both integers (**signed**) and whole numbers (**unsigned**). Integers are the default thereby making the *type-specifier* **signed** optional.
   1.2. Ranges: The range of values a particular integer variable can take on is limited by the number of bits allocated to that variable. The *type-specifiers* **char, short**, **int**, and **long** define the relative range of values that a variable of that type can take on. **char ≤ short ≤ int ≤ long**
   1.3. Implementation: Integers are implemented as two's complement binary integers. Whole numbers are implemented as unsigned binary integers. Several field widths (*w*) are common including 8, 16, and 32 bits.
      1.3.1. **Integers: Let I be the set of integers.**

$$\boldsymbol{I = \{i \in I | -2^{s-1} \le i \le 2^{s-1}, s \in \{8, 16, 32, 64\}\}}$$

      1.3.1.1. An 8-bit integer c ranges from $-2^7 \le c \le 2^7 - 1$ or $-128 \le c \le 127$
      1.3.1.2. A 16-bit integer s ranges from $-2^{15} \le s \le 2^{15} - 1$ or $-32{,}768 \le s \le 32{,}767$
      1.3.1.3. A 32-bit integer i ranges from $-2^{31} \le i \le 2^{31} - 1$ or $-2{,}147{,}483{,}648 \le i \le 2{,}147{,}483{,}647$
      1.3.1.4. A 64-bit integer s ranges from $-2^{63} \le s \le 2^{63}$

      1.3.2. Whole numbers: Let U be the set of whole numbers.

$$U = \{u \in U \mid 0 \le u \le 2^s - 1, s \in \{8, 16, 32\}\}$$

      1.3.2.1. An 8-bit whole number c ranges from $0 \le c \le 2^8 - 1$ or $0 \le c \le 255$
      1.3.2.2. A 16-bit integer s ranges from $0 \le s \le 2^{16} - 1$ or $0 \le s \le 65{,}535$
      1.3.2.3. A 32-bit integer i ranges from $0 \le i \le 2^{32} - 1$ or $0 \le i \le 4{,}294{,}967{,}296$
   1.4. Integers and whole numbers. The relationship between integers and whole numbers for a given size is shown in Figure 1.

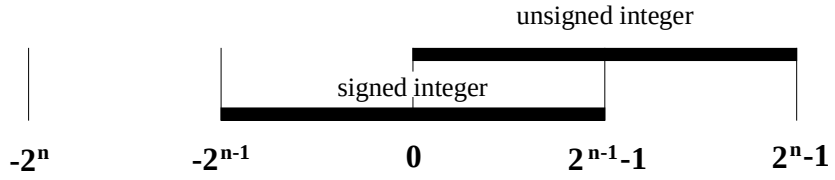
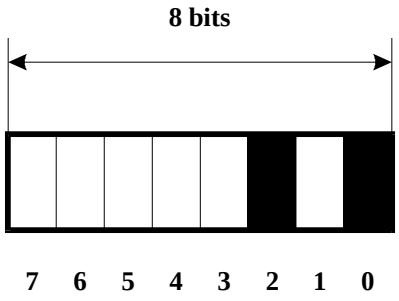
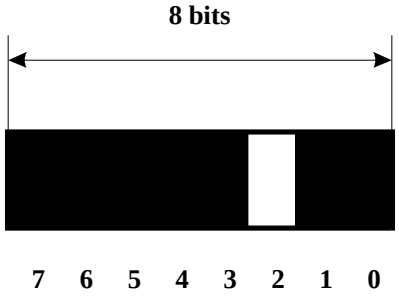


Figure 1. Signed and unsigned integer values

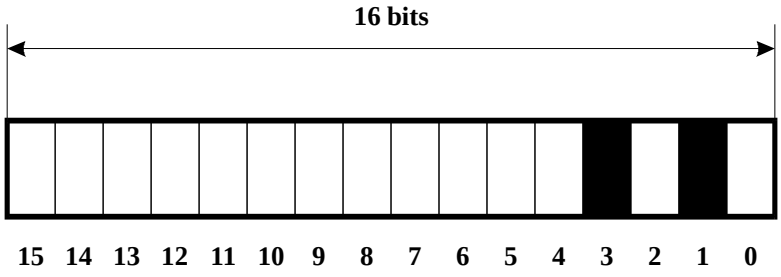
2. **Representation**: Implementation: Integers are implemented as two's complement binary integers. Whole numbers are implemented as unsigned binary integers. Several field widths (*w*) are common including 8, 16, and 32 bits.



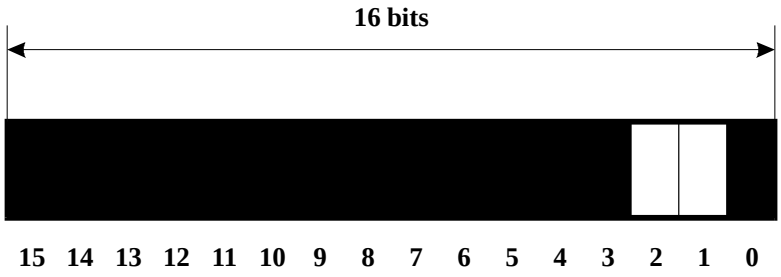
unsigned char *uc*=**5;**
Figure 2. 8-bit whole number representation.



char *sc*=**-5;**
Figure 3. 8-bit integer representation.



unsigned short *us*=**10;**
Figure 4. 16-bit whole number representation.



short *ss*=**-10;**
Figure 5. 16-bit integer representation.

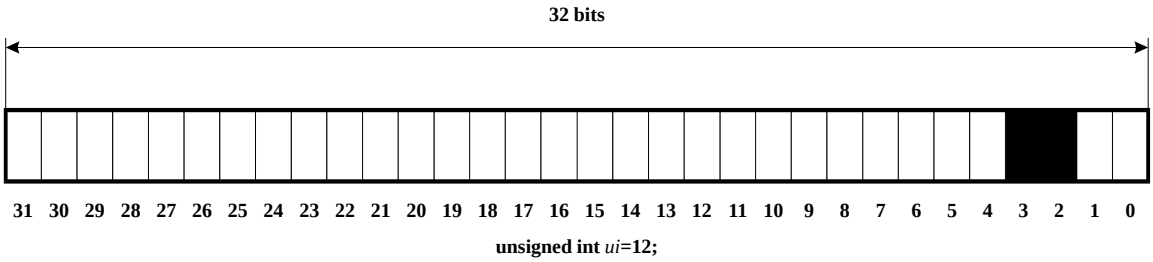


Figure 6. 32-bit whole number representation.

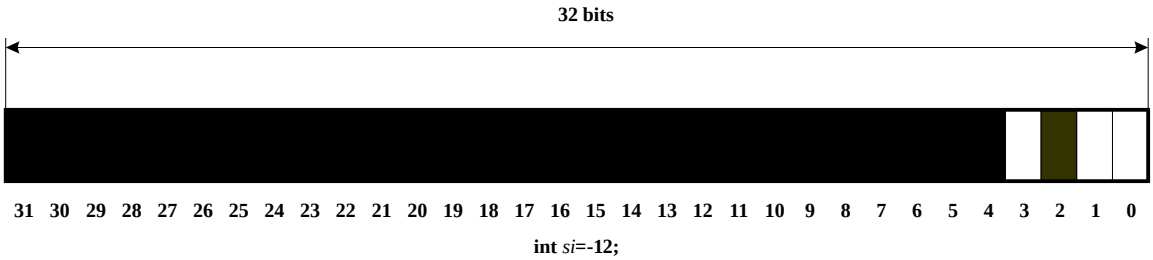


Figure 7. 32-bit integer representation.

3. **Declaration syntax**:

declarations:
integer-declaration-list **;**

integer-declaration-list:
integer-declaration
integer-declaration-list **,** *integer-declaration*

integer-declaration:
integer-declaration-specifier-sequence integer-variable-name integer-initialization$_{opt}$

integer-declaration-specifier-sequence:
integer-declaration-specifier
integer-declaration-specifier-sequence integer-declaration-specifier

integer-declaration-specifier:
storage-class-specifier
integer-type-specifier

storage-class-specifier:
auto
register
static
extern

integer type-specifier:
char
wchar_t
short
int
long
signed

unsigned

integer-variable-name:
identifier

integer-initialization:
= *assignment-expression*
(*assignment-expression* **)**

3.1. Examples:

```
unsigned char uc;
unsigned char uc1, uc2;
unsigned char uc=12;
unsigned char uc(12);
unsigned char uc1=12,uc2=14;
unsigned char uc1(12),uc2(14);
char uc;
char uc1, uc2;
char uc=12;
char uc(12);
char uc1=12,uc2=14;
char uc1(12),uc2(14);

unsigned short us;
unsigned short us1, us2;
unsigned short us=12;
unsigned short us(12);
unsigned short us1=12,us2=14;
unsigned short us1(12),us2(14);
short ss;
short ss1, ss2;
short ss=12;
short ss(12);
short ss1=12,ss2=14;
short ss1(12),ss2(14);

unsigned int ui;
unsigned int ui1, ui2;
unsigned int ui=12;
unsigned int ui(12);
unsigned int ui1=12,ui2=14;
unsigned int ui1(12),ui2(14);
int si;
int si1, si2;
int si=12;
int si(12);
int si1=12,si2=14;
int si1(12),si2(14);

unsigned long ul;
unsigned long ul1, ul2;
unsigned long ul=12;
unsigned long ul(12);
unsigned long ul1=12,ul2=14;
```

unsigned long *ul1***(12),***ul2***(14);**
long *sl***;**
long *sl1***,** *sl2***;**
long *sl***=12;**
long *sl***(12);**
long *sl1***=12,***sl2***=14;**
long *sl1***(12),***sl2***(14);**

4. **Integer constant syntax**:

integer-constants:
decimal-constant integer-suffix$_{opt}$
octal-constant integer-suffix$_{opt}$
hexidecimal-constant integer-suffix$_{opt}$

decimal-constant:
nonzero-digit
decimal-constant digit

octal-constal:
0
octal-constant octal-digit

hexadecimal-constant
0x *hex-digit*
hexadecimal-constant hex-digit

digit: one of
0 1 2 3 4 5 6 7 8 9

octal-digit: one of
0 1 2 3 4 5 6 7

hexadecimal-digit: one of
0 1 2 3 4 5 6 7 8 9
A B C D E F a b c d e f

integer-suffix:
long-suffix unsigned-suffix$_{opt}$
unsigned-suffix long--suffix$_{opt}$

long-suffix: one of
l L

unsigned-suffix: one of
u U

Constant	C++ type
dd…d	**int**
0*dd…d*	**int**
0x*dd…d*	**unsigned int**
*dd…d***U**	**unsigned int**
0*dd…d***U**	**unsigned int**
0x*dd…d***U**	**unsigned int**
*dd…d***L**	**long int**
0*dd…d***L**	**unsigned long int**
0x*dd…d***L**	**unsigned long int**
*dd…d***UL**	**unsigned long int**
0*dd…d***UL**	**unsigned long int**
0x*dd…d***UL**	**unsigned long int**

Examples:

C Constant	C++ Type	Decimal Value
0	**int**	0
0L	**long int**	0
0UL	**unsigned long int**	0
0xA	**unsigned int**	10
0xFFL	**unsigned long int**	255
017	**unsigned int**	15

5. **Operations:** The C++ programming language provides more integer types and operators that do most programming languages. The variety reflects the different word lengths and kinds of arithmetic operators found on most computers, thus allowing a close correspondence between C++ programs and the underlying hardware. Integer types in C and C++ are used to represent:
   5.1. signed or unsigned integer values, for which the usual arithmetic and relational operations are provided
   5.2. bit vectors, with the operations NOT, AND, OR, XOR, and left and right shifts
   5.3. *Boolean* values, for which zero is considered "false", and all nonzero values are considered "true," with the integer 1 being the canonical "true" value
   5.4. characters, which are represented by their integer encoding on the computer

Table 1 records operations and corresponding operators valid for signed and unsigned integers of all sizes. The usual unary conversions coerce **char**s and **short**s to **int**s.

Operation	Operator
Predecrement	--
Preincrement	++
Postdecrement	--
Postincrement	--
Multiplication	*
Division	/
Modulo (Remainder)	%
Addition	+
Subtraction	-
Left logical shift	<<
Right logical shift	>>
Less than	<
Less than or equal	<=
Greater than	>
Greater than or equal	>=
Equality	==
Inequality	!=
bitwise-and	&
bitwise-exclusive or	^
bitwise-or	|
logical-and	&&
logical-or	||

Table 1. Integer operations

Table 2 records the precedence and associativity of integer operators.

Operators	Precedence	Associativity
-- ++ (*postfix*)	17	left
-- ++ (*prefix*)	15	right
* / %	13	left
+ -	12	left
<< >>	11	left
< > <= >=	10	left
== !=	9	left
&	8	left
^	7	left
|	6	left
&&	5	left
||	4	left

Table 2. Precedence and associativity of binary integer operators

6. Example programs:

6.1. Program **p01** illustrates the predecrement operation.

```
#include <iostream>
#include <iomanip>
using namespace std;
int main()
{    int a=0;
     cout << "a  =" << setw(2) <<   a << endl;
     cout << "--a=" << setw(2) << --a << endl;
     cout << "a  =" << setw(2) <<   a << endl;
     return 0;
}
```

Figure 6.1. Program **p01**.

6.1.1. Program **p01** output.

```
a  = 0
--a=-1
a  =-1
```

6.1.2. *cout* **<<** **--***a*; is equivalent to *a*=*a*-**1;** *cout* **<<** *a*;

6.2. Program **p02** illustrates the preincrement operation.

```
#include <iostream>
#include <iomanip>
using namespace std;
int main()
{    int a=0;
     cout << "a  =" << setw(2) <<   a << endl;
     cout << "--a=" << setw(2) << ++a << endl;
     cout << "a  =" << setw(2) <<   a << endl;
     return 0;
}
```

Figure 6.2. Program **p02**.

6.2.1. Program **p02** output.

```
a  = 0
++a= 1
a  = 1
```

6.2.2. *cout* **<<** **++***a***;** is equivalent to *a***=***a***+1;** *cout* **<<** *a***;**

6.3. Program **p03** illustrates the postdecrement operation.

```
#include <iostream>
#include <iomanip>
using namespace std;
int main()
{    int a=0;
     cout << "a  =" << setw(2) <<   a << endl;
     cout << "a--=" << setw(2) << a-- << endl;
     cout << "a  =" << setw(2) <<   a << endl;
     return 0;
}
```

Figure 6.3. Program **p03**.

6.3.1. Program **p03** output.

```
a  = 0
a--= 0
a  =-1
```

6.3.2. *cout* **<<** *a***--;** is equivalent to *cout* **<<** *a***;** *a***=***a***-1;**

6.4. Program **p04** illustrates the postincrement operation.

```
#include <iostream>
#include <iomanip>
using namespace std;
int main()
{    int a=0;
     cout << "a  =" << setw(2) <<   a << endl;
     cout << "a++=" << setw(2) << a++ << endl;
     cout << "a  =" << setw(2) <<   a << endl;
     return 0;
}
```

Figure 6.4. Program **p04**.

6.4.1. Program **p04** output.

```
a  = 0
a++= 0
a  = 1
```

6.4.2. *cout* **<<** *a***++;** is equivalent to *cout* **<<** *a***;** *a***=***a***+1;**

6.5. Program **p05** illustrates addition, subtraction, multiplication, and division.

```
#include <iostream>
#include <iomanip>
using namespace std;
int main()
{    int a=10,b=5,c=2,d=1,e=13,f=5;
     cout << a << "+"
          << b << "*"
          << c << "-"
          << d << "-"
          << e << "/"
          << f << "="
          << a+b*c-d-e/f;
     cout << endl;
     return 0;
}
```

Figure 6.5. Program **p05**.

6.5.1. Program **p05** output.
10+5*2-1-13/5=17

6.5.2. Multiplication and division have higher precedence than addition and subtraction.

6.5.3. Operations are evaluated from left to right.

6.5.4. Since both operands (**13**,**5**) of the division operator (**/**) are integers, the quotient is also an integer. The quotient of **13/5** is **2**.

6.6. Program **p06** illustrates integer division and the C++ operator that produces the remainder when one integer is divided by another.

```
#include <iostream>
#include <iomanip>
using namespace std;
int main()
{    for (int dividend=-10;dividend<10;dividend++) {
          int divisor=5;
          cout << endl;
          cout << setw(3) << dividend              << "/"
               << setw(3) << divisor               << "="
               << setw(3) << dividend/divisor      << " "
               << setw(3) << dividend              << "%"
               << setw(3) << divisor               << "="
               << setw(3) << dividend%divisor      ;
     }
     cout << endl;
     return 0;
}
```

Figure 6.6. Program **p06**.

6.6.1. Program **p06** output.

```
-10/  5= -2 -10%  5=  0
 -9/  5= -1  -9%  5= -4
 -8/  5= -1  -8%  5= -3
 -7/  5= -1  -7%  5= -2
 -6/  5= -1  -6%  5= -1
 -5/  5= -1  -5%  5=  0
 -4/  5=  0  -4%  5= -4
 -3/  5=  0  -3%  5= -3
 -2/  5=  0  -2%  5= -2
 -1/  5=  0  -1%  5= -1
  0/  5=  0   0%  5=  0
  1/  5=  0   1%  5=  1
  2/  5=  0   2%  5=  2
  3/  5=  0   3%  5=  3
  4/  5=  0   4%  5=  4
  5/  5=  1   5%  5=  0
  6/  5=  1   6%  5=  1
  7/  5=  1   7%  5=  2
  8/  5=  1   8%  5=  3
  9/  5=  1   9%  5=  4
```

6.6.2. The "**/**" operator performs integer division. For example **–9/5=-1**. Performing ordinary division and truncating the fractional portion of the result is a method for finding the quotient of integer division.

6.6.3. The "**%**" operator finds the remainder.

6.7. Program **p06** determines the value of *n* for the computer on which the program executes.

```
#include <iostream>
using namespace std;
int main()
{    signed char a;
     signed short b;
     signed int c;
     signed long d;
     cout << "size of char =" << (8*sizeof a) << " bits" << endl;
     cout << "size of short =" << (8*sizeof b) << " bits" << endl;
     cout << "size of int =" << (8*sizeof c) << " bits" << endl;
     cout << "size of long =" << (8*sizeof d) << " bits" << endl;
     return 0;
}
```

Figure 12. File **p06.cpp**

Program **p06** prints

```
size of char =8 bits
size of short =16 bits
size of int =32 bits
size of long =32 bits
```

6.8. Program **p07** prints minimum and maximum values for signed integers of various sizes.

```
#include <iostream>
#include <math.h>
using namespace std;
int main()
{    signed char lsc=-pow(2,7), hsc=pow(2,7)-1;
     signed short lss=-pow(2,15), hss=pow(2,15)-1;
     signed int lsi=-pow(2,31), hsi=pow(2,31)-1;
     signed long lsl=-pow(2,31), hsl=pow(2,31)-1;
     cout << "A signed char ranges from " << (int)lsc << " and " << (int)hsc << endl;
     cout << "A signed short ranges from " << lss << " and " << hss << endl;
     cout << "A signed int ranges from " << lsi << " and " << hsi << endl;
     cout << "A signed long ranges from " << lsl << " and " << hsl << endl;
     return 0;
}
```

Figure 13. File **p07.cpp**

Program **p07** prints
A signed char ranges from -128 and 127
A signed short ranges from -32768 and 32767
A signed int ranges from -2147483648 and 2147483647
A signed long ranges from -2147483648 and 2147483647

Notes:
1. The range for a **signed long** should be -2147483648 (-2^{31}) and 2147483647 ($2^{31}-1$).
2. Function pow returns b^e given pow(b,e)

6.9. Program **p08** prints minimum and maximum values for unsigned integers of various sizes.

```
#include <iostream>
#include <math.h>
using namespace std;
int main()
{    unsigned char lsc=0, hsc=0xFF;
     unsigned short lss=0, hss=0xFFFF;
     unsigned int lsi=0, hsi=0xFFFFFFFF;
     unsigned long lsl=0, hsl=0xFFFFFFFF;
     cout << "An unsigned char ranges between " << (int)lsc << " and " ;
     cout << (int)hsc << "." << endl;
     cout << "An unsigned short ranges between " << lss << " and " << hss << "." << endl;
     cout << "An unsigned int ranges between " << lsi << " and " << hsi << "." << endl;
     cout << "An unsigned long ranges between " << lsl << " and " << hsl << "." << endl;
     return 0;
}
```

Figure 14. File **p08.cpp**

Program **p08** prints
An unsigned char ranges between 0 and 255.
An unsigned short ranges between 0 and 65535.
An unsigned int ranges between 0 and 4294967295.
An unsigned long ranges between 0 and 4294967295.

Notes:
1. Integer variables can be initialized using hexadecimal values.

6.10. Program **p09** illustrates the relationship between integers and characters.

```
#include <iostream>
using namespace std;
int main()
{   char a='a',z='z';
    cout << "The integer code for the letter" << a << " is " << (int)a << ".";
    cout << endl;
    cout << "The integer code for the letter" << z << " is " << (int)z << ".";
    cout << endl;
    return 0;
}
```

Figure 15. File **p09.cpp**

Program **p09** prints:
The integer code for the letter a is 97.
The integer code for the letter z is 122.

6.11. Program **p10** illustrates the relationship between integers and Boolean values. Zero is "false" and any nonzero value is "true" with one (1) being the canonical value for "true."

```
#include <iostream>
using namespace std;
int islower(char c)
{     return 'a' <= c && c <= 'z';
}
int main()
{     int a='a';
      cout << "Variable a does" ;
      if (!islower(a)) cout << " not";
      cout << " contain a lower case letter." << endl;
      return 0;
}
```

Figure 16. File **p10.cpp**

Program **p10** prints:
Variable a does contain a lower case letter.

Notes:
1. Function *islower* returns either a 1 or a 0 depending on whether the expression 'a' <= c && c <= 'z' is "true" or "false."
2. The logical-and operator **&&** has lower precedence than relational operators.

Left logical shift.
The left logical shift operator **<<** inserts a 0 in the least significant bit position and moves every bit one position to the left to bit having the next higher significance. The most significant bit is discarded. The effect of shifting an integer variable one position to the left is equivalent to multiplying it by 2.

Declaration	Expression	Binary Representation	Binary Result	Integer Result
char *a*=1;	***a*<<1**	**0000 0001<<1**	**0000 0010**	**2**
char *a*=1;	***a*<<2**	**0000 0001<<2**	**0000 0100**	**4**
char *a*=1;	***a*<<3**	**0000 0001<<3**	**0000 1000**	**8**
char *a*=1;	***a*<<4**	**0000 0001<<4**	**0001 0000**	**16**
char *a*=3;	***a*<<1**	**0000 0011<<1**	**0000 0110**	**6**

Table 3. Left logical shift examples

Right logical shift.
The right logical shift operator **>>** inserts a 0 in the most significant bit position and moves every bit one position to the right to bit having the next lower significance. The least significant bit is discarded. The effect of shifting an integer variable one position to the left is equivalent to dividing it by 2.

Declaration	Expression	Binary Representation	Binary Result	Integer Result
char *a*=2;	***a*>>1**	**0000 0010>>1**	**0000 0001**	**1**
char *a*=4;	***a*>>2**	**0000 0100>>2**	**0000 0001**	**1**
char *a*=8;	***a*>>3**	**0000 1000>>3**	**0000 0001**	**1**
char *a*=16;	***a*>>4**	**0001 0000>>4**	**0000 0001**	**1**
char *a*=7;	***a*>>1**	**0000 0111>>1**	**0000 0011**	**3**

Table 4. Left logical shift examples

Bitwise-and (**&**)

The bitwise-and (**&**) operator compares corresponding bits in the two operands. If both bits are equal to one (1) the result is one (1). If either bit is a zero (0), the result is zero (0). The diagram in Figure 7 specifies the bitwise-and operation. All values that can be assigned to one-bit operands are listed. Two one-bit operands are shown. Results are tabulated in the interior of the diagram.

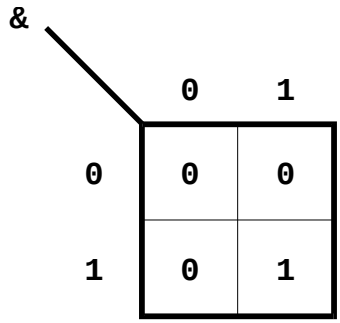


Figure 7. Bitwise-and (**&**) operation

Examples of the bitwise-and (**&**) operation are shown in Table 5.

Declaration	Expression	Binary Representation	Binary Result	Integer Result
char *a*=**0x55;** **char** *b*=**0x0f;**	*a*&*b*	*a*=**0101 0101** *b*=**0000 1111**	**0000 0101**	**5**

Table 5. Bitwise-and (**&**) examples

Bitwise-exclusive-or (**^**)

The bitwise-exclusive-or (**^**) operator compares corresponding bits in the two operands. If the bits in both operands are different, the result is one (1). If both bits are equal, the result is zero (0). The diagram in Figure 8 specifies the bitwise-exclusive-or operation. All values that can be assigned to one-bit operands are listed. Two one-bit operands are shown. Results are tabulated in the interior of the diagram.

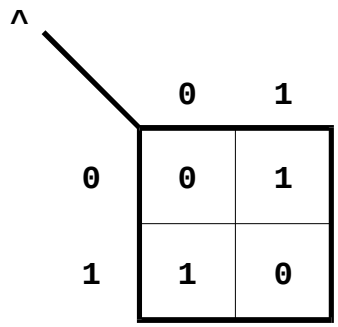


Figure 8. Bitwise-exclusive-or (**^**) operation

Examples of the bitwise-exclusive-or (**^**) operation are shown in Table 6.

Declaration	Expression	Binary Representation	Binary Result	Integer Result
char *a*=**0x55;** **char** *b*=**0x0f;**	*a*^*b*	*a*=**0101 0101** *b*=**0000 1111**	**0101 1010**	**90**

Table 6. Bitwise-exclusive-or (**^**) examples

Bitwise-inclusive-or (**|**)

The bitwise-inclusive-or (**|**) operator compares corresponding bits in the two operands. If either operand is a one (1), the result is one (1). If both operands are zero (0), the result is zero (0). The diagram in Figure 9 specifies the bitwise-inclusive-or operation. All values that can be assigned to one-bit operands are listed. Two one-bit operands are shown. Results are tabulated in the interior of the diagram.

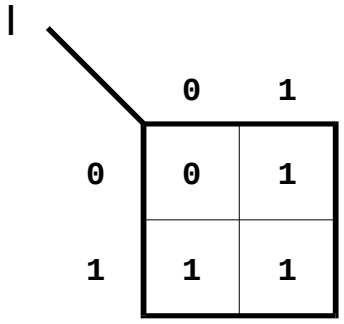


Figure 9. Bitwise-inclusive-or (**|**) operation

Examples of the bitwise-inclusive-or (**^**) operation are shown in Table 7.

Declaration	Expression	Binary Representation	Binary Result	Integer Result
char *a*=**0x55;** **char** *b*=**0x0f;**	*a*|*b*	*a*=**0101 0101** *b*=**0000 1111**	**0101 1111**	**95**

Table 7. Bitwise-inclusive-or (**|**) examples

References:

1. Stroustrup: p 73-74,78-85
2. Harbison and Steele *A Reference Manual* Prentice Hall 1995 ISBN 0-13-326224-3; p24-27, 110-114, 210-216
3. Horstman and Budd; Big C++; p 35, 39-40, 43, 48, 56-57, 59-62

Exercises:

1. Horstman and Budd; *Big C++*; p 73 Exercise P2.1
2. Horstman and Budd; *Big C++*; p 73 Exercise P2.2
3. Horstman and Budd; *Big C++*; p 73 Exercise P2.3
4. Horstman and Budd; *Big C++*; p 73 Exercise P2.4
5. Horstman and Budd; *Big C++*; p 73 Exercise P2.5
6. Horstman and Budd; *Big C++*; p 73 Exercise P2.6
7. Write a program that will initialize an integer variable to 0, insert a single bit in the least significant position, shift the bit through all the bit positions in the variable, and print the integer value at each position.
8. Write a program that will insert and 3-bit value in bit positions 3, 4, and 5 of another integer variable.