



This program shows how to display an `int` value using different bases:

```
int main()
{
    int x;
    cout << "input x: ";
    cin >> x;
    cout << "base 10: " << dec << x;
    cout << "      base 16: " << hex << x;
    cout << "      base  2 " << ((x&8)!=0) << ((x&4)!=0) <<
        ((x&2)!=0) << ((x&1)!=0) << endl;
    return 0;
}
```

Variables and constants can be combined in expressions using operators. Available operators are:

- '=' – The “assignment” operator assigns the value of the expression on right to the variable on left. Automatic type conversions will take place (e.g. converting `float` result on right hand side to `int` variable on left hand side.)
- ‘+’, ‘-’, ‘\*’, ‘/’, ‘%’ – These are the same as for a calculator but note that integer division truncates the result to the nearest integer (and ‘%’ gives the remainder). Note that operator precedence is important: \* and / will be evaluated before + and -.
- ‘++’, ‘--’ – These unary operators will increment or decrement the variable they operate on. For example, “x++” is the same as “x = x + 1.”
- ‘&’, ‘|’, ‘^’, ‘~’ – These “bitwise” operators will set each bit of the binary result according to the corresponding bits of the two operands according to the following table. For example, “101 & 011” evaluates to “001”, “101 | 011” evaluates to “111”, “101 ^ 011” evaluates to “110”, and “~101” evaluates to “010”.

**Table 2. Results of bitwise operators**

x	y	x & y (AND)	x   y (OR)	x ^ y (exclusive OR)	~x (comple- ment)
0	0				
0	1				
1	0				
1	1				

- '==', '!=', '>', '>=', '<', '<=' – These are “relational” operators. They evaluate to 1 if the relationship is true, 0 if it is false. Do NOT confuse '=' (assignment) with '==' (comparison)!!
- '&&', '||', '!' – These are “logical” operators. If  $k$  is a value not equal to zero, they evaluate as shown in Table 3.

**Table 3. Results of logical operators**

<b>x</b>	<b>y</b>	<b>x &amp;&amp; y (AND)</b>	<b>x    y (OR)</b>	<b>!x (NOT)</b>
0	0			
0	$k$			
$k$	0			
$k$	$k$			

The American National Standards Institute (ANSI)/International Standards Organization (ISO) and most graphics and CAD libraries, define a “Boolean” type, which takes on values as follows:

“False” = 0

“True” = 1

The Boolean type is the same as the integer type but may vary in terms of the number of digits maintained, depending on who defined it. The ANSI/ISO syntax is “bool”, “true”, and “false”. For example, “bool x = true;”, is a valid ANSI/ISO statement. The capitalization and abbreviations often vary between libraries, but the ANSI/ISO way should be used if possible.

## 2. Enumerated types

Enumerated types are also based on integer types, but they allow the user to specify names for each value:

E.g., Season: Summer = 0; Fall = 1; Winter = 2; Spring = 3

Usage:

```
enum Season { Summer, Fall, Winter, Spring };
Season baseball_season = Summer;
```

## 3. Text characters

Text characters are also based on integers. In the American Standard Code for Information Interchange (ASCII) standard they only have 8 bits. In the new “Unicode” standard, characters have 16 bits (I think) and can represent many more characters, including greek letters. Usage is as shown by example:

```
char c = 'a';
```

In ASCII, 'a' = 97, 'b' = 98, ... Some special characters can be typed with a backslash, as shown in Table 4.

**Table 4. Backslash (escape) characters.**

Character	What it means
\n	New line
\t	Tab
\b	Backspace
\"	Quote
\'	Apostrophe
\?	Question mark
\\	Backslash

E.g. `char tab_char = '\t';`

## 4. Pointer

Pointers are integer values that index a memory location. The pointer typically points to a single variable or array of variables, but can also point to a function. Usage:

```
float *pX;
```

In this example the asterisk makes the variable `pX` to be a pointer, which will point to some memory location holding a floating point value.

## 5. Real numbers

Real numbers are stored as fractions. Usage:

```
float x = 3.5;
```

## 6. Arrays

Arrays allow storing a series of values. E.g.:

```
int indivisible[6] = { 1, 2, 3, 5, 7, 11 };
char name[7] = "Bernie"; // For strings, the array must have
                        // room for a terminating 0 at the end.
```

The indices for arrays start at 0. For example, for `name` above,

```
name[1] == 'e'
```

## 7. Memory usage and range of each value type

Each variable type has limits to how large a number it can hold. Similarly, each variable type requires a certain amount of memory. Certain keywords can be used to modify the range and memory usage when the variable is defined, as shown in Table 5.

**Table 5. Fundamental data type value ranges**

Type	Size	Values
short int	2 bytes	-32,768 to 32,767
unsigned short int	2 bytes	0 to 65,535
long int (int default)	4 bytes	-2,147,483,648 to 2,147,483,647
unsigned long int	4 bytes	0 to 4,294,967,295
char	1 byte	0 to 255
pointer	4 byte	0x0 to 0xFFFFFFFF (hex)
float	4 bytes	1.2E-38 to 3.4E+38
double	8 bytes	2.2E-308 to 1.8E308

Note that 1 byte = 8 bits.

## 8. Constants

If the value of a variable should not change for the duration of the program execution, define it to be constant.

E.g. `const double pi = 3.1415926;`

## 9. Type Definitions

Use "typedef" to define your own variable type based on existing types.

E.g.: `typedef unsigned long int my_int;`

This creates a new type "my\_int" which can be used throughout the code. If you later want to change "my\_int" to be a signed long int, then only the typedef statement needs to be changed and the rest of the code can stay the same.

## 10. Casting

If one type of variable must be assigned to be another type, use a cast operator.

```
E.g. cout << 3/2 << endl;
      // prints 1 (truncated integer)
cout << (float)3/2 << endl;
      // prints 1.5 (result of expression is a float)
cout << static_cast<float>3/2 << endl;
      // newer alternative syntax for casting
```

## 11. Math Functions

The file math.h includes the prototypes for some important math functions:

`abs()` – absolute value of an integer

`fabs()` – absolute value of a real

`sqrt()` – square root

`pow()` – power  $x^y$

`exp()` – exponent

To find out more about these functions, use the “man” UNIX command. E.g. to find out about the `abs()` function, at the UNIX prompt, type:

```
>man abs
```

When using these functions, be sure to include the `math.h` header file at the top of your source file. I.e.:

```
#include <math.h>
```