

A bit-vector implementation of sets in C++:

```
class BitSet
{
  short* bits;
  // ...
public:
  BitSet( int );
  // ...
  int contains( int );
};
const int BSS = 8 * sizeof(short); // bitstring size
int BitSet::contains( int n ){
  return (bits[n/BSS]>>(BSS-n%BSS-1)) & 1;
}
// ...

BitSet s(26);
cout << s.contains( 19 );
```

```
BSS = 8
n = 19
```

```
0 [0] 7 8 [1] 15 16 [2] 23 24 [3] 31
00101011|00110110|110?0010|00110110
```

```
bits[ n/8 ] 110?0010
8 - n%8 - 1 = 4 >>4
0000110?
&
1 00000001
0000000?
```

10 March 2005

Coercion (= automatic conversion)

Pascal: no coercion

Haskell: coercion of integer numerals only

C/C++: profusion of numeric types
⇒ coercion is a practical necessity

- integral *promotions* (“value preserving”)
 - [signed | unsigned] char → [unsigned] int
 - [unsigned] short → [unsigned] int
- “usual” conversions for binary operators
 1. long double, any → long double
double, any → double
float, any → float
other: integral promotions on both
 2. unsigned long, any → unsigned long
long, unsigned → [unsigned] long
long, any → long
unsigned, any → unsigned

Java: Coerces primitive values, when no magnitude information would be lost

```
char → int
byte → short → int → long
float → double
{byte, short, int, long} → floating-point
```

A value converted from int (32 bits) or long (64 bits) to float (32 bits) may lose precision.

10 March 2005

Casting

C/C++

The term cast refers to four different kinds of type conversion:

- `static_cast`: ordinary conversion, producing a new value “equivalent” to the original. Example:
`double x = static_cast<double>(3);`
- `dynamic_cast`: run-time type-checked conversion of pointers. Example:
`dynamic_cast<T*>(p)`
Value is 0 (i.e., **false**) if the conversion is invalid (hence is used in tests)
- `reinterpret_cast`: produces no new value; merely changes the set of functions that can be applied to the original value
`reinterpret_cast<IO_device*>(0XFFFF0)`
(completely unchecked)
- `const_cast`: removes const protection from a parameter

Java

Cast = explicit conversion $t_0 \rightarrow t_1$ `t0 y = ...;``t1 x = (t1)y;``double → float`: loss of precision, 0, ∞`floating-point → integer`: fractional part lost, 0, ∞`integer → smaller integer`: high-order bits lost`integer → char`: only low-order 16 bits are used`boolean → int`: *illegal*

10 March 2005

Literals (C constants)

Pascal:

boolean: **true**, **false** are reserved words

char: visible characters are enclosed in single quotes ('')

integer: $[+|-] \{ \textit{Digit} \}^+$

real (approximate syntax):

 $[+|-] \{ \textit{Digit} \} . \{ \textit{Digit} \} [(E|e) [+|-] \{ \textit{Digit} \}^+]$

C/C++/Java:

char:

visible characters are enclosed in single quotes (e.g., 'r');

invisible characters can be denoted by the following convention:

```
\n new-line      \r return      \' '
\f form-feed    \t tab
\b backspace    \\ backslash
\ddd up to three (octal!) digits :
                        ascii character code
```

integer:

decimal: $[+|-] (1 | \dots | 9) \{ 0 | \dots | 9 \}$ octal: $[+|-] 0 \{ 0 | \dots | 7 \}$ hexadecimal: $[+|-] 0X \{ 0 | \dots | 9 | A | \dots | F \}^+$ unsigned: $\langle \textit{integer literal} \rangle U$ long: $\langle \textit{integer literal} \rangle L$ float, double: similar to Pascal **real**
(with optional suffix f or d)

10 March 2005

Haskell literals

Char:

as in C (but ascii codes are decimal)

Int:

Integer literals are defined like C decimal numerals, but are *overloaded* (see Slide 70).

Float:

Floating-point literals are defined as in C, but are also overloaded.

Prelude> :t 3.0

3.0 :: Fractional a => a

Type-class Fractional includes Float, Double, Rational,...

Built-in operators

C's comparison operators

<u>op</u>	<u>example</u>	<u>value</u>	<u>comment</u>
==	x == y	$\begin{cases} 0 & \text{if } x \neq y \\ 1 & \text{if } x = y \end{cases}$	equality
!=	x != y	$\begin{cases} 0 & \text{if } x = y \\ 1 & \text{if } x \neq y \end{cases}$	inequality
<=	x <= y	$\begin{cases} 1 & \text{if } x \leq y \\ 0 & \text{if } x > y \end{cases}$	at most

etc.

C's logical operators

<u>op</u>	<u>example</u>	<u>value</u>	<u>comment</u>
&&	a && b	if a=0 then 0 else if b=0 then 0 else 1	<i>short-ckt</i> \wedge
	a b	if a=1 then 1 else if b=1 then 1 else 0	<i>short-ckt</i> \vee
!	!a	if a≠0 then 0 else 1	negation
?:	a?b:c	if a≠0 then b else c	conditional expression
&	a&b	$a \wedge b$	<i>bitwise</i> conjunction
	a b	$a \vee b$	<i>bitwise</i> disjunction
^	a^b	$a \neq b$	<i>bitwise</i> exclusive-or
~	~a	$\neg a$	<i>bitwise</i> negation

Example:

```
6 & 1  ~> 0
6 && 1 ~> 1
```

C's shift operators

<u>op</u>	<u>example</u>	<u>value</u>	<u>comment</u>
<<	1<<3 a<<n = a * 2 ⁿ , provided 0≤a and	8 0≤n<sizeof(a)*bitsPerChar	left shift (0 fill)
>>	15>>3 a>>n = a / 2 ⁿ , provided 0≤a and	1 0≤n<sizeof(a)*bitsPerChar	right shift (*)

* if a<0, 0- or 1-fill, depending on the implementation

C's side-effect operators

<u>op</u>	<u>example</u>	<u>value</u>	<u>sideeffect</u>	<u>comment</u>
=	x=e	x'(l-value)	x := e	assignment
⊗=	x+=e where ⊗∈{+, -, *, /, %, <<, >>, &, ^, }	x'(l-value)	x := x ⊗ e	compound assignment
++	++i	i ₀ +1	i := i+1	increment
++	i++	i ₀	i := i+1	increment
--	--i	i ₀ -1	i := i-1	decrement
--	i--	i ₀	i := i-1	decrement

The meaning of C's (++) and (--) operators

<u>C</u>	<u>Pascal</u>
++n	<i>preInc</i> (n) function <i>preInc</i> (var a: integer) : integer ; begin a := a + 1; <i>preInc</i> := a (* returns a's new value *) end ;
n++	<i>postInc</i> (n) function <i>postInc</i> (var b: integer) : integer ; var t: integer ; begin t := b; (* saves b's original value *) b := b + 1; (* ...but see note *) <i>postInc</i> := t (* returns b's original value *) end ;

Note: The update of n may be *delayed*.

An excerpt from the comp.lang.c FAQ:

Although the postincrement and postdecrement operators ++ and -- perform the operations after yielding the former value, the implication of "after" is often misunderstood. It is not guaranteed that the operation is performed immediately after giving up the previous value and before any other part of the expression is evaluated. It is merely guaranteed that the update will be performed sometime before the expression is considered "finished" (before the next "sequence point," in ANSI C's terminology).