【C++】Day 13

Class	C++
Date	@December 1, 2021
Material	
Series Number	
Summary	

【Ch4】The sizeof Operator

4.9 sizeof Opertor

The `sizeof` operator returns the size, in bytes of an expression or a type name. The sizeof operator is unusual in that it does not evaluate its operand:

```
Sales_data data, *p;
sizeof(Sales_data); //size required to hold an object of type Sales_data
sizeof data; //size of data's type
sizeof p; //size of a pointer
sizeof *p; //size of the type to which p points
size of data.revenue; //size of the type of Sales_data's revenue memeber
sizeof Sales_data::revenue; //alternative way to get the size of revenue
```

Because `sizeof` does not evaluate its operand, it doesn't matter that p is an invalid(uninitialized) pointer. Dereferencing an invalid pointer as the operand to sizeof is safe because the pointer is not actually used. `sizeof` doesn't need dereference the pointer to know what type it will return.

- sizeof a reference type: Returns the size of an object of the referenced type
- sizeof a dereferenced pointer: Returns the size of an object of the type to which the pointer points; the pointer need not be valid
- sizeof an array: Is the size of the entire array. It is equivalent to taking the sizeof the element type times the number of elements in the array.

Note: sizeof does not convert the array to a pointer

- sizeof a string or a vector: Returns only the size of the fixed part of these types; it does not return the size used by the object's element

4.11 Type Conversions

In C++, two types are related if there is a conversion between them.

Consider the following expression:

```
int ival = 3.54 + 3; //the compiler might warn about loss of precision
```

The operands of the addition are values of two different types: 3,54 that has type double, and 3 is an int. Rather than attempt to add values of the two different types, C++ defines a set of conversions to transform the operands to a common type.

These conversions are carried out automatically without programmer intervention—and sometimes without programmer knowledge. For that reason, they are referred to as implicit conversions.

The implicit conversions among the arithmetic types are defined to preserve precision, if possible. Most often, if an expression has both integral and floatingpoint operands, the integer is converted to floating-point. In this case, 3 is converted to double, floating-point addition is done, and the result is a double.

The initialization happens next. In an initialization, the type of the object we are initializing dominates. The initializer is converted to the object's type. `int ival = 6.0;` In this case, the double result of the addition is converted to int and used to initialize ival. Converting a double to an int truncates the double's value, discarding the decimal portion.

When Implicit Conversions Occur

The compiler automatically converts operands in the following circumstances:

- In most expressions, values of integral types smaller than int are first promoted to an appropriate larger integral type.
- In conditions, non-bool expression are converted to bool.
- In initializations, the initializer is converted to the type of the variable; in assignments, the right-hand operand is converted to the type of the left-hand
- In arithmetic and relational expression with operands of mixed types, the types are converted to a common type
- Conversions also happen during function calls

4.11.1 The Arithmetic Conversions

The arithmetic conversions convert one arithmetic type to another. The rules define a hierarchy of the type conversions in which operands to an operator are converted to the widest type.

For example, if one operand is of type long double, then the other operand is converted to type long double regardless of what the second type is.

More generally, in expressions that mix floating-point and integral values, the integral value is converted to an appropriate floating-point type.

Integral Promotions

The integral promotions convert the small integral types to a larger integral type.

The types bool, char, signed char, unsigned char, short, and unsigned short are promoted to int if all possible values of that type fit in an int. Otherwise, the value is promoted to unsigned int.

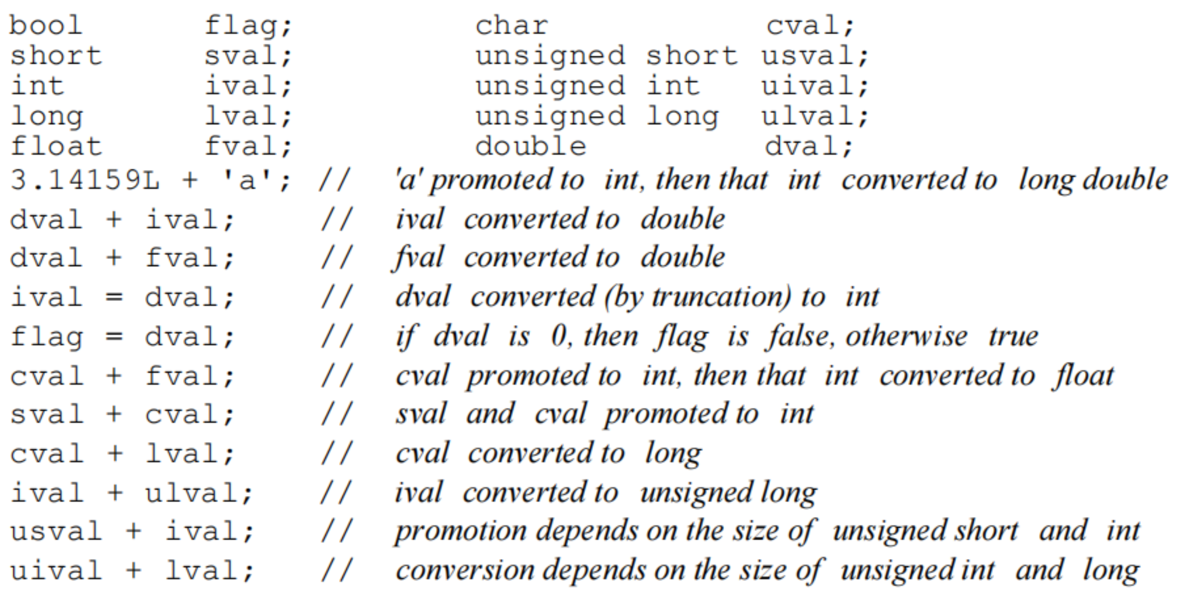
Operands of Unsigned Type

1. If the operands of an operator have differing types, those operands are ordinarily converted to a common type. If any operand is an unsigned type, the type to which the operands are converted depends on the relative size of the integral types on the machine.

2. As usualy, integral promotions happen first. If the resulting types match, no further conversion is needed. If both operands have the same signedness, then the operand with the smaller type is converted to the larger one.

3. When the signedness differs and the type of the unsigned operand is the same as or larger than that of the signed operand, the signed operand is converted to unsigned.

4. When the signed operand has a larger type than the unsigned operand, the result is machine dependent. If all values in the unsigned type fit in the larger type, then the unsigned operand is converted to the signed type. If the values don't fit, then the signed operand is converted to the unsigned type.



```
bool      flag;          char           cval;
short     sval;          unsigned short usval;
int       ival;          unsigned int   uival;
long      lval;          unsigned long  ulval;
float     fval;          double         dval;
3.14159L + 'a'; //  'a' promoted to int, then that int converted to long double
dval + ival;    //  ival converted to double
dval + fval;    //  fval converted to double
ival = dval;    //  dval converted (by truncation) to int
flag = dval;    //  if dval is 0, then flag is false, otherwise true
cval + fval;    //  cval promoted to int, then that int converted to float
sval + cval;    //  sval and cval promoted to int
cval + lval;    //  cval converted to long
ival + ulval;   //  ival converted to unsigned long
usval + ival;   //  promotion depends on the size of unsigned short and int
uival + lval;   //  conversion depends on the size of unsigned int and long
```