B. Vasya and Types

time limit per test: 2 seconds
memory limit per test: 256 megabytes
input: standard input
output: standard output

Programmer Vasya is studying a new programming language &K*. The &K* language resembles the languages of the C family in its syntax. However, it is more powerful, which is why the rules of the actual C-like languages are unapplicable to it. To fully understand the statement, please read the language's description below carefully and follow it and not the similar rules in real programming languages.

There is a very powerful system of pointers on &K* — you can add an asterisk to the right of the existing type X — that will result in new type $X*$. That is called pointer-definition operation. Also, there is the operation that does the opposite — to any type of X, which is a pointer, you can add an ampersand — that will result in a type $\&X$, to which refers X. That is called a dereference operation.

The &K* language has only two basic data types — `void` and `errtype`. Also, the language has operators `typedef` and `typeof`.

- The operator "`typedef` A B" defines a new data type B, which is equivalent to A. A can have asterisks and ampersands, and B cannot have them. For example, the operator `typedef void** ptptvoid` will create a new type `ptptvoid`, that can be used as `void**`.
- The operator "`typeof` A" returns type of A, brought to `void`, that is, returns the type `void**...*`, equivalent to it with the necessary number of asterisks (the number can possibly be zero). That is, having defined the `ptptvoid` type, as shown above, the `typeof ptptvoid` operator will return `void**`.

An attempt of dereferencing of the `void` type will lead to an error: to a special data type `errtype`. For `errtype` the following equation holds true: `errtype*` = `&errtype` = `errtype`. An attempt to use the data type that hasn't been defined before that will also lead to the `errtype`.

Using `typedef`, we can define one type several times. Of all the definitions only the last one is valid. However, all the types that have been defined earlier using this type do not change.

Let us also note that the dereference operation has the lower priority that the pointer operation, in other words $\&T*$ is always equal to T.

Note, that the operators are executed consecutively one by one. If we have two operators "`typedef &void a`" and "`typedef a* b`", then at first `a` becomes `errtype`, and after that `b` becomes `errtype* = errtype`, but **not** `&void* = void` (see sample 2).

Vasya does not yet fully understand this powerful technology, that's why he asked you to help him. Write a program that analyzes these operators.

Input

The first line contains an integer n $(1 \le n \le 100)$ — the number of operators. Then follow n lines with operators. Each operator is of one of two types: either "`typedef` A B", or "`typeof` A". In the first case the B type differs from `void` and `errtype` types, and besides, doesn't have any asterisks and ampersands.

All the data type names are non-empty lines of no more than 20 lowercase Latin letters. The number of asterisks and ampersands separately in one type in any operator does not exceed 10, however if we bring some types to `void` with several asterisks, their number may exceed 10.

Output

For every `typeof` operator print on the single line the answer to that operator — the type that the given operator returned.

Examples

input

```
5
typedef void* ptv
typeof ptv
typedef &&ptv node
typeof node
typeof &ptv
```

output

```
void*
errtype
void
```

input

```
17
typedef void* b
typedef b* c
typeof b
typeof c
typedef &b b
typeof b
typeof c
typedef &&b* c
typeof c
typedef &b* c
typeof c
typedef &void b
typeof b
typedef b******* c
typeof c
typedef &&b* c
typeof c
```

output

```
void*
void**
void
void**
errtype
void
errtype
errtype
errtype
```

Note

Let's look at the second sample.

After the first two queries `typedef` the `b` type is equivalent to `void*`, and `c` — to `void**`.

The next query `typedef` redefines `b` — it is now equal to `&b = &void* = void`. At that, the `c` type doesn't change.

After that the `c` type is defined as `&&b* = &&void* = &void = errtype`. It doesn't influence the `b` type, that's why the next `typedef` defines `c` as `&void* = void`.

Then the `b` type is again redefined as `&void = errtype`.

Please note that the `c` type in the next query is defined exactly as `errtype******* = errtype`, and not `&void******* = void******`. The same happens in the last `typedef`.